ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ
КАФЕДРА ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМ ТА МЕРЕЖ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «МЕТОДИ ОБРОБКИ ЧЕРГ В ІНФОКОМУНІКАЦІЙНИХ МЕРЕЖАХ»

на здобуття освітнього ступеня магістра
зі спеціальності 172 Електронні комунікації та радіотехніка
(код, найменування спеціальності)
освітньо-професійної програми Телекомунікаційні системи та мережі
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Михайло НЕЩЕРЕТ
(підпис) *(Ім'я ПРІЗВИЩЕ)*

Виконав: здобувач вищої освіти гр. ТСДМ-62
Михайло НЕЩЕРЕТ
Ім'я, ПРІЗВИЩЕ

Керівник: Ганна ГРИНКЕВИЧ
д.т.н., професор *Ім'я, ПРІЗВИЩЕ*

Рецензент: _____
Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут Телекомунікацій

Кафедра ___Телекомунікаційних систем та мереж_____
Ступінь вищої освіти __магістр_____
Спеціальність ___172 Електронні комунікації та радіотехніка _____
Освітньо-професійна програма _____Телекомунікаційні системи та мережі_____

ЗАТВЕРДЖУЮ
Завідувач кафедри __ТСМ_____
_____ Віктор ЗАЇКА
«____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Нещерету Михайлу Володимировичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Методи обробки черг в інфокомунікаційних мережах
керівник кваліфікаційної роботи Ганна ГРИНКЕВИЧ, д.т.н., професор
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» 10. 2025 р. № 467

2. Строк подання кваліфікаційної роботи «___» _____ 2025 р.

3. Вихідні дані до кваліфікаційної роботи:
1. Обробка черг.
2. Інфокомунікаційні мережі.
3. Топологія черг.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Основні поняття черг в мережах.
2. Проблематика черг.
3. Топологія черг та додаткові ресурси.
4. Управління хмарними чергами.

5. Перелік ілюстративного матеріалу: ___презентація на ____слайдах_____

6. Дата видачі завдання «___» _____ 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	15.10.2025	Викон.
2	Розробка першого розділу	23.10.2025	Викон.
3	Розробка другого розділу	03.11.2025	Викон.
4	Розробка третього розділу	13.11.2025	Викон.
5	Розробка четвертого розділу	23.11.2025	Викон.
6	Висновки, вступ, реферат	03.12.2025	Викон.
7	Розробка презентації	13.12.2025	Викон.

Здобувач вищої освіти _____ Михайло НЕЩЕРЕТ
(підпис) *(Ім'я ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Ганна ГРИНКЕВИЧ
(підпис) *(Ім'я ПРІЗВИЩЕ)*

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут телекомунікацій
ПОДАННЯ
ГОЛОВІ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня магістра

Направляється здобувач Нещерет М.В. до захисту кваліфікаційної роботи
(прізвище та ініціали)
за спеціальністю 172 Електронні комунікації та радіотехніка
(код, найменування спеціальності)
освітньо-професійної програми Телекомунікаційні системи та мережі
(назва)
на тему: «Методи обробки черг в інфокомунікаційних мережах»

Кваліфікаційна робота і рецензія додаються.

Директор ННІТ _____ Владислав КРАВЧЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Висновок керівника кваліфікаційної роботи

Здобувач Нещерет Михайло Володимирович показав гарну теоретичну та інженерну підготовку, уміння володіти новими комп'ютерними технологіями, користуватися навчальною, довідковою і науково-технічною літературою. Працюючи над завданнями, які були поставлені керівником, проявив ініціативність, сумлінність та хист до наукової роботи.

Все це дозволяє оцінити виконану кваліфікаційну роботу здобувача Нещерета Михайла Володимировича на оцінку «відмінно» та присвоїти йому кваліфікацію магістр з електронних комунікацій та радіотехніки.

Керівник кваліфікаційної роботи _____ Ганна ГРИНКЕВИЧ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

«____» _____ 20___ року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач Нещерет М.В. допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедри телекомунікаційних систем та мереж _____ Віктор ЗАЇКА
(назва) *(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

«____» _____ 20___ року

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну роботу
на здобуття освітнього ступеня магістра

здобувача вищої освіти Нещерета Михайла Володимировича
(прізвище, ім'я, по батькові)
на тему «Методи обробки черг в інфокомунікаційних мережах»

Актуальність.
Стрімке зростання обсягів трафіку, розвиток мікросервісних архітектур та хмарних інфраструктур ставлять високі вимоги до надійності й продуктивності інфокомунікаційних систем. Черги повідомлень відіграють ключову роль у забезпеченні асинхронності, балансування навантаження та стійкості до збоїв. Неправильне проєктування черг може спричинити перевантаження, дублювання даних, затримки та деградацію сервісів. Актуальність дослідження полягає в необхідності аналізу моделей чергування, виявлення їх обмежень та розроблення рекомендацій для підвищення ефективності і надійності розподілених систем.
Позитивні сторони.
1. У роботі досліджено особливості алгоритмів FIFO, LIFO, Best Effort, механізми TTL/TTR, пріоритизацію, підчерги, повтори та архітектурні обмеження.
2. У роботі розглянуто моделі взаємодії (Put/Take, Pub/Sub, Request/Response) та проаналізовано протоколи (AMQP, MQTT, STOMP).
Недоліки.
1. У роботі бракує конкретних числових метрик, графіків або таблиць, які б демонстрували вплив черги на throughput, latency, retry-rate та інші важливі характеристики.
2. Робота описує проблеми та властивості платформ, але не пропонує уніфікованих критеріїв чи моделі оцінки ефективності різних підходів.

Відзначені зауваження не впливають на загальну позитивну оцінку магістерської кваліфікаційної роботи

Висновок: *кваліфікаційна робота на здобуття ступеня магістра заслуговує оцінку "відмінно", а здобувач Нещерет Михайло Володимирович заслуговує присвоєння кваліфікації: магістр з електронних комунікацій та радіотехніки*.

Рецензент _____ _____
науковий ступінь, вчене звання (підпис) *(Ім'я ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 78 стор., 23 рис., 25 джерел.

Мета роботи – проаналізувати існуючі підходи, протоколи та архітектури черг повідомлень, виявити їхні обмеження й проблеми роботи в реальних інфокомунікаційних системах та сформувати рекомендації щодо підвищення ефективності, продуктивності й надійності механізмів чергування.

Об’єкт дослідження – процеси обміну повідомленнями та організація черг у сучасних інфокомунікаційних і розподілених системах.

Предмет дослідження – моделі, алгоритми та архітектури черг повідомлень, а також їх вплив на продуктивність, надійність, масштабованість і стійкість розподілених систем під різними типами навантажень.

Короткий зміст роботи: у роботі досліджено методи обробки черг в інфокомунікаційних мережах та показано, що сучасні черги є критичним елементом надійних і масштабованих розподілених систем. Розглянуто моделі впорядкування задач (FIFO, LIFO, Best Effort), механізми пріоритизації, дублювання, використання dead-letter черг, TTL і повторної доставки. Систематизовано чотири рівні проблем чергування: алгоритмічні обмеження, системні ефекти продуктивності, мережеву невизначеність і відмови інфраструктури. Проаналізовано моделі підвищення надійності — реплікацію, кворумні механізми, кластеризацію. Підкреслено роль моніторингу та експлуатаційних практик у запобіганні збоям.

КЛЮЧОВІ СЛОВА: ЧЕРГИ ПОВІДОМЛЕНЬ, ІНФОКОМУНІКАЦІЙНІ МЕРЕЖІ, FIFO, LIFO, BEST EFFORT, ПРІОРИТИЗАЦІЯ, ДУБЛЮВАННЯ ПОВІДОМЛЕНЬ, DEAD-LETTER ЧЕРГА, TTL, TTR, ЗАТРИМКА, НАДІЙНІСТЬ, РОЗПОДІЛЕНІ СИСТЕМИ

ABSTRACT

Text part of the master's qualification work: 78 pages, 23 figures, 25 sources.

The purpose of the work is to analyze existing approaches, protocols and architectures of message queues, identify their limitations and problems of operation in real infocommunication systems and formulate recommendations for increasing the efficiency, productivity and reliability of queuing mechanisms.

Object of the research is message exchange processes and queue organization in modern infocommunication and distributed systems.

Subject of the research is models, algorithms and architectures of message queues, as well as their impact on the performance, reliability, scalability and stability of distributed systems under different types of loads.

Summary of the work: the work investigates methods of processing queues in infocommunication networks and shows that modern queues are a critical element of reliable and scalable distributed systems. Task ordering models (FIFO, LIFO, Best Effort), prioritization mechanisms, duplication, use of dead-letter queues, TTL and redelivery are considered. Four levels of queuing problems are systematized: algorithmic limitations, system performance effects, network uncertainty and infrastructure failures. Reliability improvement models are analyzed - replication, quorum mechanisms, clustering. The role of monitoring and operational practices in preventing failures is emphasized.

KEYWORDS: MESSAGE QUEUES, INFORMATION AND COMMUNICATION NETWORKS, FIFO, LIFO, BEST EFFORT, PRIORITIZATION, MESSAGE DUPLICATION, DEAD-LETTER QUEUE, TTL, TTR, DELAY, RELIABILITY, DISTRIBUTED SYSTEMS

ЗМІСТ

ВСТУП 9
РОЗДІЛ 1 ОСНОВНІ ПОНЯТТЯ ЧЕРГ В МЕРЕЖАХ 11
1.1 Поняття черг та їх необхідність 11
1.2 Класифікація підходів 17
1.3 Використання протоколів та платформ 25
РОЗДІЛ 2 ПРОБЛЕМАТИКА ЧЕРГ 35
2.1 Перший рівень проблем 35
2.2 Другий рівень проблем 39
2.3 Третій рівень проблем 43
2.4 Четвертий рівень проблем 47
РОЗДІЛ 3 ТОПОЛОГІЯ ЧЕРГ ТА ДОДАТКОВІ РЕСУРСИ 52
3.1 Топологія черг 52
3.2 Додаткові ресурси 61
РОЗДІЛ 4 УПРАВЛІННЯ ХМАРНИМИ ЧЕРГАМИ 65
4.1 Постановка задачі 65
4.2 Підходи та рішення 68
ВИСНОВКИ 77
ПЕРЕЛІК ПОСИЛАНЬ 79
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 81

ВСТУП

У сучасних інфокомунікаційних мережах обмін повідомленнями та керування потоками даних відіграють ключову роль у забезпеченні стабільності, швидкодії та надійності цифрових сервісів. Зі зростанням обсягів трафіку, розвитком мікросервісних архітектур, хмарних інфраструктур та розподілених обчислень виникає потреба у більш ефективних механізмах обробки черг. Саме черги забезпечують упорядкованість комунікації, асинхронність взаємодії та розмежування навантаження між системними компонентами, що особливо важливо для масштабованих та високодоступних мережевих рішень.

Методи обробки черг постійно еволюціонують: від класичних локальних структур даних до складних розподілених систем із гарантованою доставкою, механізмами реплікації, пріоритизацією та різними моделями підтвердження. В інфокомунікаційних мережах ці механізми застосовуються для обслуговування подій, маршрутизації трафіку, синхронізації станів між сервісами, балансування навантаження та забезпечення безперервної роботи систем під високими навантаженнями. Важливо розуміти як базові алгоритми на кшталт FIFO, LIFO чи Best Effort, так і складніші архітектури, включаючи репліковані брокери, кворумні системи, стримінгові платформи та хмарні черги.

Особливу увагу в дослідженні приділено фундаменальним викликам, притаманним розподіленим системам: мережевій невизначеності, затримкам, можливим втратам повідомлень, дедуплікації, повторній доставці та впливу апаратних обмежень на продуктивність. Ці фактори формують багаторівневу структуру проблем, серед яких — труднощі з пріоритизацією, накопичення підчерг, обмеження пропускної здатності, деградація продуктивності через обраний алгоритм доступу до даних, а також ризики, пов’язані з відмовами серверів, дисків чи навіть цілих дата-центрів.

У межах роботи розглянуто сучасні протоколи обміну повідомленнями та їх практичне застосування у мережевих інфраструктурах, різні архітектури обробки черг, моделі масштабування, принципи реплікації та забезпечення гарантій

доставки. Окрему увагу приділено проблемам надійності та доступності, а також підходам до підвищення стійкості інфокомунікаційних систем за рахунок коректно побудованої логіки черги.

Також було досліджено практичні аспекти моніторингу, у тому числі аналіз часу обробки повідомлень, виявлення повторів та втрат, поведінку черги під піковим навантаженням та сценарії деградації продуктивності. Окрема частина роботи присвячена розгляду реальних підходів до оптимізації — від хмарних сервісів до механізмів міжекземплярного обміну повідомленнями, що дозволяють мінімізувати витрати та затримки.

Таким чином, тема обробки черг є комплексною, охоплює широкий спектр методів і технологій та має критичне значення для побудови сучасних інфокомунікаційних мереж. У цій роботі систематизовано ключові теоретичні підходи, розглянуто практичні рішення та сформульовано рекомендації щодо проєктування ефективних, масштабованих і надійних систем обслуговування черг.

1 ОСНОВНІ ПОНЯТТЯ ЧЕРГ В МЕРЕЖАХ

1.1 Поняття черг та їх необхідність

Для більшості застосунків стандартні рішення дають змогу обробляти приблизно 10 000 запитів на секунду — і будь-який популярний брокер повідомлень легко з цим впорається. Але коли виникає потреба масштабуватися серйозніше, коли швидкість зростає в рази або десятки разів, тоді доводиться глибше занурюватися в принципи роботи черг та вчитися будувати правильну архітектуру.

У даному розділі увагу приділено тому, що таке черги повідомлень, навіщо вони потрібні і як вони функціонують. На прикладі реальних сценаріїв буде показано, як організована взаємодія між компонентами через черги, які існують підходи до проєктування та які інструменти використовуються у сучасній розподіленій інфраструктурі.

Також буде розглянуто найпоширеніші проблеми, що виникають при роботі з брокерами повідомлень: від втрати даних і затримок до надмірного зростання черги та проблем із балансуванням навантаження. Ви дізнаєтеся, як уникнути цих помилок, які стратегії масштабування допомагають забезпечити стабільність і що необхідно враховувати під час налаштування системи.

Окрему увагу буде приділено порівнянню найпопулярніших брокерів: їхнім архітектурним особливостям, перевагам, недолікам та оптимальним варіантам використання. Після цього ви зможете краще зрозуміти, які інструменти підходять саме вашим цілям, як вибудувати ефективну систему обробки повідомлень і чому грамотне використання черг може в рази підвищити надійність та продуктивність вашого програмного забезпечення.

Для чого потрібні черги?

Багато розробників чули про черги повідомлень, але не всі до кінця розуміють, навіщо вони насправді потрібні та яку роль відіграють у сучасних програмних архітектурах. Тому найпростіше пояснити їхнє призначення через

практичні приклади — саме вони найкраще демонструють сильні та слабкі сторони цього інструмента.

Почнімо з найпростішої моделі. Уявімо, що у нас є три сервери:

- **перший — слабкий**, з обмеженою продуктивністю;
- **другий — потужний**, здатний обробляти набагато більше запитів;
- **третій — тимчасово непрацюючий**, тобто повністю недоступний.

На всі три сервери йде вхідний трафік: користувачі надсилають запити, система повинна реагувати на них у реальному часі.

Якщо використати класичний балансувальник навантаження, який просто рівномірно розподіляє всі запити між доступними серверами, то отримаємо ситуацію, коли:

- слабкий сервер буде перевантажений і почне гальмувати;
- потужний сервер працюватиме нижче своїх можливостей;
- недоступний сервер взагалі не зможе обробити жодного запиту, але балансувальник усе одно буде намагатися щось на нього надсилати, що спричинятиме додаткові помилки.

Така схема здається логічною лише на перший погляд, але вона не враховує реальних особливостей роботи системи: різну продуктивність серверів, їхні тимчасові збої, непередбачувані піки навантаження та можливість черги запитів.

Черга виступає «буфером» між вхідним потоком даних і системами, які їх обробляють. Вона дозволяє:

- уникати миттєвого перевантаження слабких або тимчасово недоступних серверів;
- розподіляти роботу за фактичною продуктивністю кожного з обробників;
- гарантувати, що жоден запит не буде втрачений навіть у момент збою;
- гнучко масштабувати систему, просто додаючи нові сервери-споживачі;
- вирівнювати піки навантаження: коли запитів багато — черга росте; коли менше — сервери «доганяють» накопичене.

Таким чином, замість того щоб жорстко й примусово розкидати кожен новий запит по серверах, ми створюємо центральний буфер, через який проходить уся

робота. Сервери не отримують навантаження понад свої можливості — кожен з них бере рівно стільки задач, скільки здатен обробити в конкретний момент.

Черги дають архітектурі еластичність, стабільність і передбачуваність. Саме тому вони стали невід'ємним інструментом у мікросервісах, високонавантажених системах, e-commerce-проєктах, телекомі, фінансовій галузі та скрізь, де швидкість і надійність критично важливі.

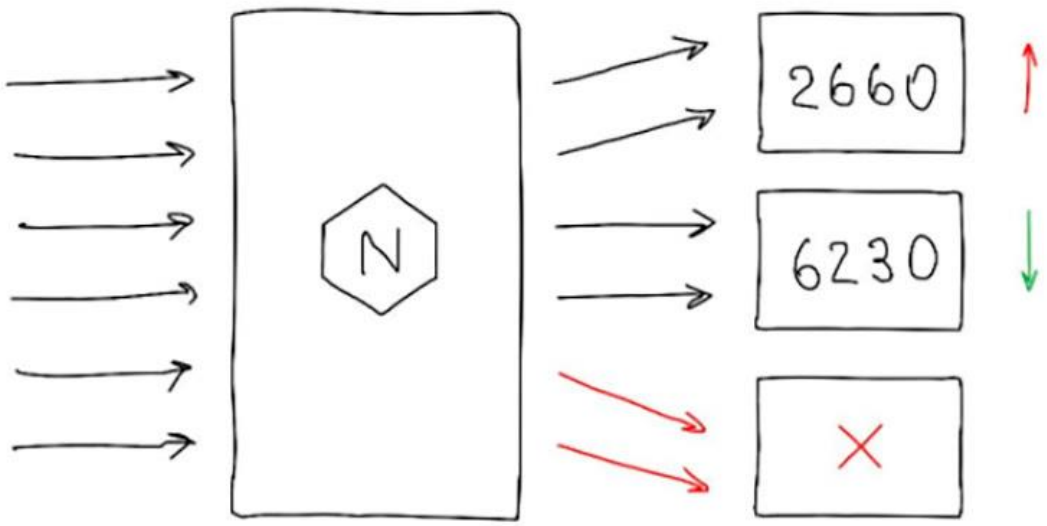


Рис. 1.1 Схема класичного балансувальника

У такій схемі слабкий сервер неминуче опинятиметься під надмірним навантаженням: він отримуватиме стільки ж запитів, скільки й потужніший вузол, хоча його ресурсів для цього просто не вистачає. У результаті він почне працювати повільно, утворювати затримки й накопичувати черги на своєму рівні. Потужний сервер, навпаки, залишатиметься недовантаженим — замість того щоб максимально реалізувати свій потенціал, він просто чекатиме на нові задачі. А всі запити, які будуть спрямовані на непрацюючий сервер, взагалі стануть помилковими, що створить додаткові збої й ускладнить діагностику проблем.

Однак ситуація кардинально змінюється, якщо замість прямого розподілу навантаження ми додаємо централізовану чергу повідомлень. У цьому випадку вхідні запити не надсилаються одразу на сервери, а спочатку потрапляють до черги — своєрідного буфера, який тимчасово зберігає всі задачі.

Після цього робочі сервери — незалежно від їхньої потужності — забирають задачі з черги у тому темпі, в якому здатні їх обробляти. Слабкий сервер бере менше запитів, потужний — більше, а непрацюючий узагалі не бере нічого, не створюючи жодних помилок. Завдяки цьому:

- **жоден запит не втрачається**, навіть якщо один із вузлів тимчасово «випав»;
- **навантаження розподіляється природним чином**, пропорційно можливостям кожного сервера;
- **система залишається стабільною**, оскільки задирки обробки перекладаються на буфер, а не на кінцеві вузли.

Таким чином, черга перетворює хаотичний потік запитів на впорядкований, контрольований процес, у якому кожен компонент працює рівно в тих межах, які для нього є оптимальними. Це дозволяє забезпечити надійність, гнучкість і високу продуктивність навіть у складних, розподілених системах.

Черги повідомлень часто застосовують не лише для балансування навантаження, а й для планування та оптимізації виконання задач у часі. Уявімо типову ситуацію: ваша система працює в умовах нерівномірного потоку запитів, де навантаження має форму класичної синусоїди — тобто чергуються пікові та мінімальні значення. Або ж ваш графік навантаження нагадує інший хвилеподібний патерн, що часто зустрічається в e-commerce, фінансових системах, системах бронювання, логістичних сервісах та багатьох інших галузях.

Коли навантаження росте й наближається до пікових значень, обладнання або інфраструктура повинні повністю перекривати цей пік, щоб система залишалася стабільною. Але це означає, що у години мінімальної активності ваші ресурси простоюватимуть, використовуючись неефективно. Ви платите за максимальну продуктивність, використовуючи її лише частину часу.

Саме тут у гру знову вступають черги. Якщо система вміє *відкладати* частину задач, які перевищують оптимальне навантаження, і тимчасово складати їх у чергу, ви отримуєте можливість:

- утримувати меншу кількість обладнання або інстансів;
- зменшувати витрати на інфраструктуру;

- вирівнювати навантаження в часі;
- спокійно обробляти накопичені задачі в моменти, коли робочі ресурси звільняються.

Іншими словами, черга дозволяє системі "дихати": під час піку вона бере на себе надлишкові задачі, а в періоди спаду — дає серверам можливість наздогнати накопичене. У результаті ви отримуєте стійку, передбачувану та економічно вигідну архітектуру, що не потребує постійного масштабування під непередбачувані піки.
Черги виконують цілу низку критично важливих функцій:

- **гнучкий розподіл ресурсів** між компонентами;
- **реплікацію та зберігання повідомлень**, що гарантує їх збереження;
- **високу відмовостійкість** навіть при часткових збоях системи;
- **надійну доставку задач** незалежно від стану споживачів;
- **організацію комунікації між мікросервісами**, які можуть працювати на різних вузлах і в різному темпі.

Саме тому черги трапляються практично всюди. Багато механізмів у низькорівневих системах базуються саме на їхніх принципах. Наприклад:

- у апаратній частині так працюють IRQ, NCQ та апаратні буфери,
- у ядрах ОС — механізми epoll/kqueue, різні форми обробки подій та сигналів,
- на рівні застосунків — міжпотокова взаємодія (cross-thread queues) та внутрішньопроцесна комунікація (IPC).

У розподілених системах, мережевих протоколах, інтеграційних шинах, API-шлюзах та мікросервісах черги стали стандартом де-факто. Вони дозволяють компонентам із різною швидкістю, продуктивністю та пріоритетами взаємодіяти без втрат і блокувань.

Фактично, якщо уважно подивитися на будь-яку складну систему, черги можна знайти майже в кожному її шарі — від апаратного рівня до бізнес-логіки. Вони стають тим «клеєм», який поєднує між собою різні частини архітектури, забезпечуючи стабільність, синхронізацію та плавну взаємодію всіх компонентів.

Найпростішим поясненням того, що таке черга на рівні структур даних, є уявлення про неї як про механізм передавання інформації за допомогою

повідомлень. Черга виступає проміжною ланкою, яка дозволяє одному компоненту системи відправити дані, а іншому — отримати їх у зручний для нього момент.

У найзагальнішому випадку це виглядає так: один процес, сервіс або модуль розміщує повідомлення в чергу, а інший — зчитує його з черги, обробляє та виконує відповідні дії. Цей принцип схожий на роботу поштової скриньки: відправник може покласти лист у будь-який час, а одержувач забере його тоді, коли зможе.

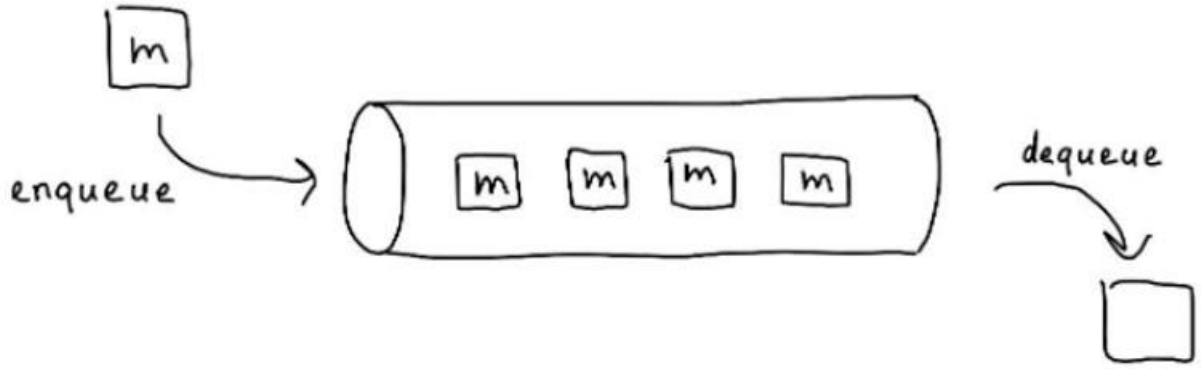


Рис. 1.2 Схема передавання повідомлення процесом, сервісом або модулем

Проте за цією простою моделлю ховається набагато ширший функціонал. Черги дозволяють:

- роз’єднати відправника та отримувача у часі, тобто їм не потрібно працювати одночасно;
- гарантувати, що повідомлення не зникне, навіть якщо отримувач тимчасово недоступний;
- масштабувати обробку, додаючи нових споживачів, які забиратимуть задачі паралельно;
- вирівнювати навантаження, перетворюючи хаотичний потік повідомлень у керовану послідовність;
- забезпечувати безпечну та асинхронну комунікацію між частинами системи, які можуть бути написані різними мовами, працювати на різних серверах і мати різну продуктивність.

Черга — це фундаментальний елемент архітектури, який дозволяє будувати системи, що не «ламаються» при пікових навантаженнях, залишаються

стабільними під час часткових збоїв і легко розширюються у міру збільшення вимог.

Якщо дивитися ширше, то черга — це не просто структура даних із операціями *enqueue* та *dequeue*. Це ціла концепція організації взаємодії між процесами, сервісами й навіть цілими кластерами. Вона забезпечує надійність комунікацій, контрольованість подій та узгодженість роботи незалежних частин системи.

І саме завдяки цій простій і водночас потужній ідеї сучасні високонавантажені системи можуть працювати швидко, стабільно й передбачувано.

1.2 Класифікація підходів

Підхід Put/Take: модель взаємодії типу 1 → 1

Однією з базових та найінтуїтивніших моделей роботи з чергами є підхід *Put/Take*, який передбачає прямолінійну логіку: кожне повідомлення, яке один раз покладено в чергу, згодом один раз забирається для обробки. Цей принцип є фундаментальним для багатьох систем обробки подій та завдань, адже забезпечує чітку та передбачувану роботу всієї інфраструктури.

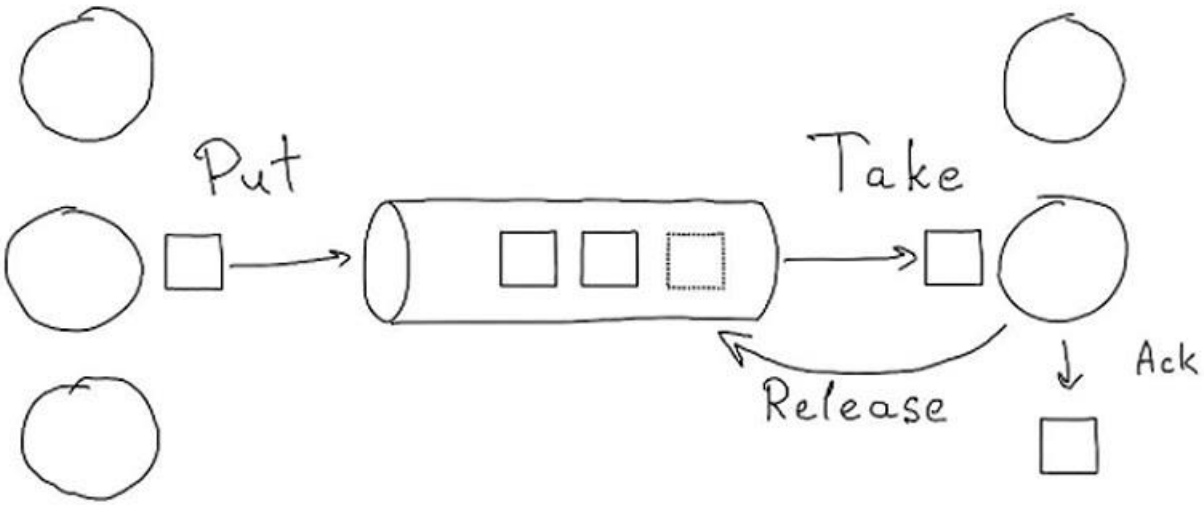


Рис. 1.3 Підхід Put/Take

Механізм складається з двох основних ролей:

1. **Продюсер (producer)** – компонент, що створює і публікує задачу у чергу. Він не турбується про те, хто саме буде обробляти це повідомлення або коли відбудеться обробка.
2. **Консюмер (consumer)** – компонент, який отримує повідомлення з черги і займається виконанням відповідної задачі. Кількість консюмерів може змінюватися залежно від потреб системи.

Після того як консюмер забирає повідомлення, є два можливих сценарії:

1. **Задача успішно оброблена.**

   У такому разі повідомлення вважається виконаним і безповоротно видаляється з черги. Це звична модель для обробки замовлень, транзакцій, сесійних подій, рендерингу зображень, формування звітів тощо.

2. **Під час обробки виникла помилка.**

   Консюмер не може завершити виконання задачі – наприклад, через:

   – тимчасову несправність зовнішнього сервісу;

   – нестачу ресурсів;

   – неправильні або неповні вхідні дані;

   – внутрішній збій алгоритму.

У цьому випадку повідомлення може бути повернуте назад у чергу. Завдяки цьому інший консюмер або той самий (після певної паузи) зможе повторно спробувати обробити задачу. Це забезпечує надійність системи та захищає від втрати інформації.

Ключові властивості та переваги моделі Put/Take

Одне повідомлення — один виконавець.

Система гарантує, що кожне повідомлення обробляється лише один раз і не дублюється між кількома споживачами. Це важливо для операцій, які не можна виконувати двічі, наприклад, для фінансових транзакцій або зміни стану об’єктів у базі даних.

Часове роз’єднання компонентів.

Продюсер і консюмер не зобов’язані працювати синхронно. Продюсер може генерувати задачі навіть у ті моменти, коли всі консюмери зайняті або тимчасово зупинені. Завдяки цьому черга згладжує різкі коливання навантаження і дозволяє системі ефективно переживати піки активності.

Стійкість до збоїв та можливість повторної обробки.

У випадку помилок система не втрачає повідомлення. Черга слугує своєрідним сховищем для задач, поки їх не оброблять успішно. Це значно підвищує відмовостійкість архітектури.

Просте та ефективне масштабування.

Щоб обробляти більше задач за одиницю часу, достатньо збільшити кількість консюмерів. Вони паралельно забиратимуть повідомлення з тієї самої черги без необхідності переписувати бізнес-логіку або проводити складне налаштування.

Розмежування відповідальності.

Продюсери відповідають лише за генерацію задач, а консюмери — за їх виконання. Такий поділ спрощує розробку та підтримку системи, знижує ризик виникнення помилок і дозволяє розвивати кожен компонент незалежно.

Значення підходу Put/Take в сучасних системах

Модель Put/Take використовується не лише у класичних брокерах повідомлень. Вона лежить в основі більшості черг у веб-сервісах, мікросервісних архітектурах, системах потокової обробки, а також у внутрішньо-процесних чергах між потоками. Саме завдяки цьому підходу стає можливим побудувати високопродуктивні та надійні системи, які справляються з мільйонами запитів, витримують несподівані піки навантаження та забезпечують передбачувану поведінку незалежно від стану окремих компонентів.

Таким чином, Put/Take — це не просто технічний термін. Це універсальний спосіб організації роботи, який забезпечує структурованість, безпечність і стабільність всієї архітектури, надаючи інженерам потужний інструмент для створення гнучких і масштабованих рішень.

Підхід Publish/Subscribe: модель взаємодії 1 → (Pub/Sub)

Інший фундаментальний спосіб організації обміну повідомленнями в розподілених системах — це модель Publish/Subscribe, яку зазвичай скорочують як Pub/Sub. На відміну від попередньої моделі Put/Take, де на одне повідомлення припадає лише один споживач, підхід Pub/Sub передбачає зовсім інший принцип взаємодії: один продюсер може передати повідомлення відразу багатьом консюмерам, кожному з яких це повідомлення може знадобитися для своїх унікальних задач.

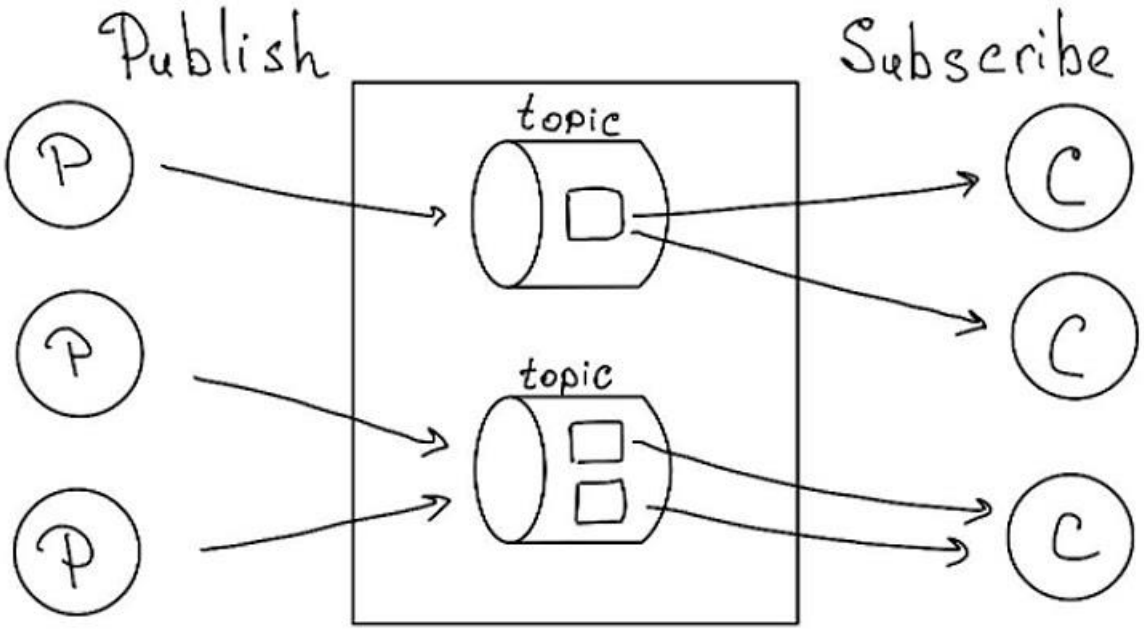


Рис. 1.4 Підхід Pub/Sub

Суть підходу полягає в тому, що продюсер (якого в Pub/Sub зазвичай називають *publisher*) публікує повідомлення в певний канал, топік або тему. Він не має жодного уявлення про те, хто саме слухає цей канал, скільки є підписників або як саме вони будуть використовувати повідомлення. Консюмери (тобто *subscribers*) заздалегідь підписуються на конкретні теми, які їх цікавлять, і отримують усі повідомлення, що надходять у ці теми.

Тут відбувається принципова зміна парадигми: продюсер не передає повідомлення конкретному одержувачу — він транслює його всім, хто бажає його отримати.

Основна різниця між Put/Take і Pub/Sub

У моделі Put/Take завдання має один кінцевий пункт призначення: його забере лише один споживач. У Pub/Sub повідомлення стає доступним відразу для багатьох консюмерів, причому кожен отримувач може використати його у власному контексті. Це дозволяє будувати одночасно кілька ланцюжків обробки, які працюють паралельно та не залежать один від одного.

Головна відмінність:

одне повідомлення → багато сценаріїв обробки.

Як працює модель Publish/Subscribe

Розглянемо типовий сценарій:

1. Publisher публікує подію — наприклад, «користувач створив замовлення».
2. Топік «order.created» транслює цю подію всім підписникам.
3. Кожен subscriber виконує власну частину обробки:
    - один сервіс надсилає email клієнту;
    - другий створює запис у системі аналітики;
    - третій інкрементує статистику продажів;
    - четвертий може ініціювати процес упаковки товару;
    - п’ятий запускає механізм antifraud-перевірки.

Усі ці операції виконуються паралельно та незалежно одна від одної — і всі вони базуються на одному-єдиному повідомленні.

Поведінка системи, якщо консюмерів немає

У Pub/Sub можливі різні варіанти реалізації, залежно від конкретного брокера та налаштувань:

- повідомлення може видалятися, якщо немає підписників;
- або зберігатися певний час, дозволяючи новим консюмерам переглянути історію;
- або зберігатися назавжди, формуючи історичний лог подій (як у Kafka).

Це робить підхід Pub/Sub значно гнучкішим, ніж класична черга Put/Take.

Переваги підходу Publish/Subscribe

1. Повна ізоляція компонентів

Publisher не знає нічого про консюмерів, а отже, система стає loosely coupled — слабозв’язаною. Це полегшує розробку, розширення та заміну компонентів без ризику «зламати» існуючу логіку.

2. Масштабування та паралелізм

Кількість subscribers може рости необмежено. Один і той самий потік подій може запускати десятки або сотні різних операцій у різних сервісах і застосунках.

3. Ідеальна модель для подійної архітектури

Pub/Sub є основою **event-driven architecture**, тобто архітектури, побудованої навколо подій. Саме так працюють:

- мікросервіси,
- системи моніторингу,
- стрімінгові платформи,
- IoT-мережі,
- брокери фінансових операцій,
- технології push-сповіщень і багато іншого.

4. Можливість створення нескінченної кількості сценаріїв

Одне повідомлення може запускати абсолютно різні бізнес-процеси — від логування до реагування на критичні інциденти.

Опубліковане повідомлення перетворюється на «сигнал» або «подію», яка розповсюджується по всій системі. Усі компоненти, які зацікавлені в цій події, отримують її незалежно від того, коли вона сталася чи в якому стані перебуває кожен сервіс під час її надходження.

Модель Pub/Sub дозволяє створювати системи, які:

- легко розширюються новими функціями;
- витримують високі навантаження;
- залишаються стійкими до збоїв окремих компонентів;
- забезпечують багатоканальну обробку інформації;
- працюють асинхронно й в реальному часі.

Підхід Request/Response: модель взаємодії 1 ⇆ 1

Цей підхід є менш поширеним порівняно з класичними моделями Put/Take та Publish/Subscribe, однак він відіграє важливу роль у системах, де необхідна двостороння, синхронізована взаємодія між компонентами. Модель Request/Response фактично повторює принцип роботи традиційного HTTP-запиту до сервера, але реалізується не через мережеві протоколи в реальному часі, а за допомогою черг повідомлень.

Суть взаємодії полягає в тому, що один компонент — умовний клієнт або ініціатор — створює спеціальне повідомлення типу Request і надсилає його у відповідну чергу. Це повідомлення містить опис задачі, параметри, контекст виконання та ідентифікатор, який дозволяє пов’язати запит з майбутньою відповіддю.

З іншого боку є консюмер, який обробляє запити — це може бути сервіс, мікросервіс, окремий процес, або навіть група споживачів, але відповідь формує завжди один із них. Після того як консюмер виконує обробку запиту, він формує повідомлення типу Response і повертає його назад у визначену відповідну чергу або на адресу, вказану у запиті. Ініціатор, що чекає на результат, забирає відповідь і продовжує роботу з отриманими даними.

Як відбувається взаємодія в моделі Request/Response

1. Клієнт створює запит: формує структуру Request, додає кореляційний ідентифікатор, параметри, дані.
2. Запит публікується у чергу запитів.
3. Консюмер отримує запит, обробляє його у своєму темпі та відповідно до своєї логіки.
4. Консюмер формує відповідь: результат виконання, коди помилок, повідомлення про статус тощо.
5. Відповідь повертається в чергу відповіді, яку слухає ініціатор запиту.
6. Клієнт отримує Response, зіставляє його із початковим запитом і завершує транзакцію.

Особливості та ключові властивості підходу

Чітка кореляція між запитом і відповіддю

Кожний запит має унікальний ідентифікатор, що дозволяє правильно співставити запит і відповідь навіть у системах з кількома паралельними процесами.

Асинхронність із поведінкою, схожою на синхронність

Попри те, що взаємодія відбувається асинхронно, зовні цей процес може поводитися як синхронний виклик — клієнт може "чекати" на відповідь або працювати з callback-механізмами.

Стійкість до збоїв у мережі

На відміну від класичного HTTP виклику, де будь-який обрив мережі означає повторення запиту або відмову, у системах, побудованих на чергах, повідомлення:

- зберігається до моменту обробки,
- не губиться при перезавантаженні сервісу,
- може оброблятися повторно у разі часткової невдачі.

Розподіленість і незалежність компонентів

Консюмери можуть бути розгорнуті на різних вузлах, у різних дата-центрах, в окремих мікросервісах або навіть на різних мовах програмування, але взаємодія залишатиметься стабільною.

Підтримка складної логіки обробки

Можна реалізувати:

- обробку довгострокових задач;
- проміжні відповіді;
- розподілення запиту між кількома етапами;
- відстеження процесів виконання.

Коли використовується модель Request/Response

Хоча цей підхід зустрічається рідше, він надзвичайно корисний у ситуаціях, коли:

- потрібно виконати запит, що вимагає **точної відповіді** — наприклад, обчислення, перевірка даних, авторизація;

- споживачі працюють у різних середовищах, але усі запити мають повертати результат;
- обробка займає значний час і не може бути виконана в межах HTTP timeouts;
- необхідно створити надійний RPC поверх черг (так званий message-based RPC);
- система має витримувати високий рівень помилок і нестабільність мережі.

Приклад застосування

Уявімо систему, яка здійснює складну фінансову перевірку транзакції. Клієнтський сервіс створює запит на перевірку, публікує його в чергу, а спеціальний обробник проводить розрахунки, звіряє дані, надсилає запити до сторонніх API й, отримавши результати, формує відповідь. Клієнт отримує Response та приймає рішення — схвалити транзакцію або відхилити. Усе це відбувається без прямого HTTP-з’єднання, але логіка взаємодії залишається знайомою та зручною.

1.3 Використання протоколів та платформ

Коли розробник починає вивчати систему черг повідомлень, він дуже швидко стикається з великою кількістю протоколів і підходів, які визначають, як саме повідомлення передаються, зберігаються та обробляються. Це може виглядати складно, адже різні технології використовують різні стандарти комунікації, мають власні моделі взаємодії та підходи до забезпечення надійності. Однак, якщо розглянути принципи роботи протоколів більш узагальнено, стає зрозуміло, що всі вони вирішують одну ключову задачу — організувати узгоджену, безпечну та керовану передачу даних між компонентами системи.

Протоколи як основа взаємодії

Протокол у контексті черг — це договір між учасниками про те, як повинні виглядати повідомлення, яким чином вони передаються та як мають поводитися процеси-учасники. Це свого роду «правила дорожнього руху» в системі обміну повідомленнями.

Серед численних протоколів, які використовуються для організації черг повідомлень і побудови систем асинхронної комунікації, можна виділити кілька основних категорій, кожна з яких має власну специфіку, переваги та сценарії застосування.

• AMQP (Advanced Message Queuing Protocol).

Це один із найпотужніших і найфункціональніших протоколів, який орієнтований на надійну доставку та гарантовану обробку повідомлень. Він підтримує складні механізми маршрутизації, підтвердження, повторної доставки, пріоритизації та організації черг у будь-яких топологіях. AMQP застосовується там, де важлива транзакційність, контроль стану та відмовостійкість — наприклад, у фінансових системах або критичних бізнес-процесах.

• MQTT (Message Queuing Telemetry Transport).

Це легковаговий протокол, спеціально створений для пристроїв з обмеженими ресурсами — IoT-сенсорів, мобільних девайсів, обладнання з нестабільним мережевим з’єднанням. MQTT мінімізує обсяг переданих даних і накладні витрати на комунікацію, забезпечує різні рівні якості обслуговування (QoS) та може ефективно працювати навіть у середовищах з великими затримками або частими розривами зв’язку.

• STOMP (Simple Text Oriented Messaging Protocol).

Протокол, побудований на простій текстовій структурі, легко читається людиною й просто інтегрується з веб-застосунками, браузерами та простими клієнтами. STOMP часто застосовується в системах, де потрібна легка інтеграція або швидке налаштування обміну повідомленнями без складних специфікацій і бінарних форматів.

• Протоколи сімейства Pub/Sub.

Ці протоколи реалізують модель “опублікуй–підпишись”, що дозволяє одному продюсеру розсилати події одразу великій кількості підписників. Pub/Sub підходить для широкомасштабного розповсюдження повідомлень, трансляції оновлень у реальному часі, організації потокових платформ і великих

мікросервісних екосистем, де важливо забезпечити багатоточковий доступ до подій.

• Легковагові високо швидкісні протоколи.

Це мінімалістичні рішення, наближені за концепцією до ZeroMQ та подібних систем, які орієнтовані на найнижчі можливі затримки. Вони не зберігають дані всередині себе та виступають легким шаром над сокетами, забезпечуючи надшвидку передачу повідомлень між вузлами. Такі протоколи застосовуються в реальному часі, у телеметрії, високопродуктивних розрахунках і системах, де головним пріоритетом є швидкість, а не гарантоване збереження.

Попри різницю в реалізаціях, усі протоколи вирішують низку спільних завдань:

- задають формат повідомлень;
- визначають правила їх доставки;
- дозволяють гарантувати або навпаки не гарантувати збереження;
- встановлюють моделі підтвердження обробки;
- керують чергами, топіками, групами споживачів.

Хмарні та локальні підходи

Сучасні інфраструктури часто використовують черги у хмарному середовищі. Це дозволяє автоматично масштабувати обробку повідомлень, не переймаючись фізичним розгортанням серверів чи підтримкою інфраструктури. Хмарні сервіси зазвичай реалізують моделі на кшталт SQS — Simple Queue Service, який став базовим стандартом для простої, стійкої та асинхронної доставки даних.

У локальних середовищах черги можуть бути побудовані навіть на основі звичайних баз даних. Це можуть бути списки, журнали подій або документи, що слугують тимчасовим сховищем для повідомлень. Такі реалізації працюють завдяки високій швидкості доступу до даних та можливості atomics-операцій, які гарантують цілісність черги.

Черги без збереження

Окремий клас рішень заснований на використанні високошвидкісних комунікацій рівня сокетів. Такі системи працюють без тривалого зберігання

повідомлень. Вони забезпечують надзвичайно швидкий обмін даними між вузлами. У подібних підходах:

- одна сторона може транслювати повідомлення безпосередньо іншим,
- повідомлення не зберігаються в довгостроковому вигляді,
- система не намагається бути історичним журналом подій,
- акцент робиться на мінімальній затримці та максимальній пропускній здатності.

Це робить подібні протоколи ідеальними для внутрішньосистемної комунікації, телеметрії, сигналів, потоків подій у реальному часі, де важлива саме швидкість, а не збереження кожного повідомлення.

Чому всі ці протоколи потрібні

Різноманіття протоколів пояснюється тим, що різні системи мають різні вимоги:

- декому потрібно зберігати повідомлення місяцями,
- іншим — обробляти їх з мінімальною затримкою,
- іншим — виконувати багатокрокові, транзакційні сценарії,
- а ще іншим — спілкуватися в умовах нестійкого мобільного інтернету.

Усі протоколи пропонують різні механізми для досягнення балансу між продуктивністю, надійністю, зручністю та складністю реалізації. Саме протокол визначає, якою буде поведінка черги:

- чи гарантується доставка,
- чи можуть повідомлення втрачатися,
- чи буде черга зберігати історію,
- чи можуть кілька споживачів працювати паралельно,
- як відбувається масштабування,
- чи можливі транзакції.

Протоколи черг повідомлень — це фундаментальні механізми, які формують спосіб взаємодії між компонентами розподілених систем. Вони задають правила, за якими повідомлення створюються, передаються, зберігаються та видаляються. Хоча реалізацій існує безліч, усі вони вирішують одну спільну задачу —

забезпечити надійну, керовану та передбачувану комунікацію між різними частинами програмного комплексу.

Розуміння цих протоколів дозволяє будувати ефективні архітектури, обирати оптимальні методи обробки подій і створювати системи, здатні працювати стабільно навіть під великим навантаженням та у середовищах із високою складністю.

Основні платформи для роботи з чергами та потоками повідомлень

Системи чергування та потокової обробки даних являють собою цілий спектр рішень, кожне з яких реалізує різну модель взаємодії з повідомленнями та має свої особливості в архітектурі, масштабуванні, зберіганні та продуктивності. Щоб розуміти специфіку цих інструментів, варто розглянути їх не як окремі продукти, а як клас платформи та принцип підходу, який вони представляють.

Потокові журнальні системи (лог-орієнтована модель)

Одним із найпотужніших підходів до організації обміну повідомленнями є модель реплікованого шардованого журналу, у якому всі повідомлення записуються у великий, незмінний лог. У таких системах передавання даних зазвичай організовано за принципом publish–subscribe, але ключовою особливістю є спосіб зберігання інформації.

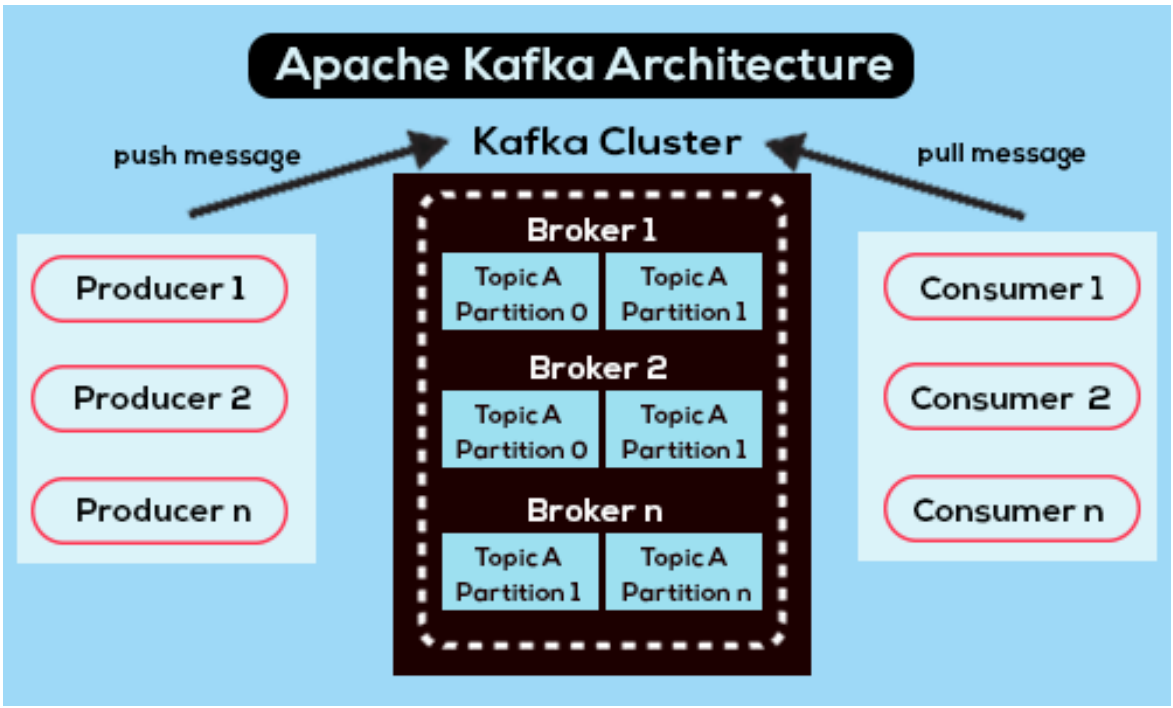


Рис. 1.5 Архітектура потокової платформи на основі шардованого журналу подій

Усі повідомлення записуються на диск у точному порядку приходу — у строго впорядковану послідовність. Оскільки лог є незмінним, порядок повідомлень залишається таким самим назавжди. Завдяки цьому платформи цього класу забезпечують властивість строгої впорядкованості FIFO, навіть у масштабованих кластерах.

На відміну від класичних черг, де повідомлення можуть видалятися після обробки, журнальні системи зберігають події скільки завгодно — годинами, днями, тижнями. Це відкриває можливість для повторного «програвання» логу. Якщо певна частина бізнес-логіки була виконана неправильно або стався збій, система може відтворити події повторно, повністю чи частково.

Така модель використовується тоді, коли потрібно працювати з великими потоками структурованих подій:

- обробка телеметрії, логів і метрик,
- побудова аналітики в реальному часі,
- аудит і контроль дій користувачів,
- реплікація даних між сервісами,
- побудова складних стрімінгових конвеєрів.

У таких систем прекрасно працює горизонтальне масштабування. Проте вони рідко застосовуються для класичного роутінгу або сценаріїв із динамічним керуванням чергами — їхня архітектура більше орієнтована на потоки, ніж на управління окремими задачами.

Традиційні брокери черг (класична модель черги)

Протилежністю потокових логів є класичні брокери черг. Їхня концепція базується на тому, що існує конкретна черга, у яку потрапляють повідомлення, та довільна кількість споживачів, які їх обробляють. На відміну від журнальних платформ, тут немає обмежень щодо кількості консюмерів, які можуть підключитися до однієї черги. Це означає, що масштабування може бути організоване дуже природно: чим більше споживачів, тим швидше очищається черга.

Такі системи зазвичай пропонують кілька моделей зберігання:

- лише в оперативній пам’яті — для надвисокої швидкості;
- на диску — для надійності;
- у вигляді реплікацій — для відмовостійкості;
- із застосуванням кворуму — щоб гарантувати, що повідомлення не буде втрачено.

Традиційні черги чудово підходять для сценаріїв pub/sub, fan-out, routing і point-to-point передачі. Часто вони використовуються як «шина даних», що з’єднує мікросервіси між собою.

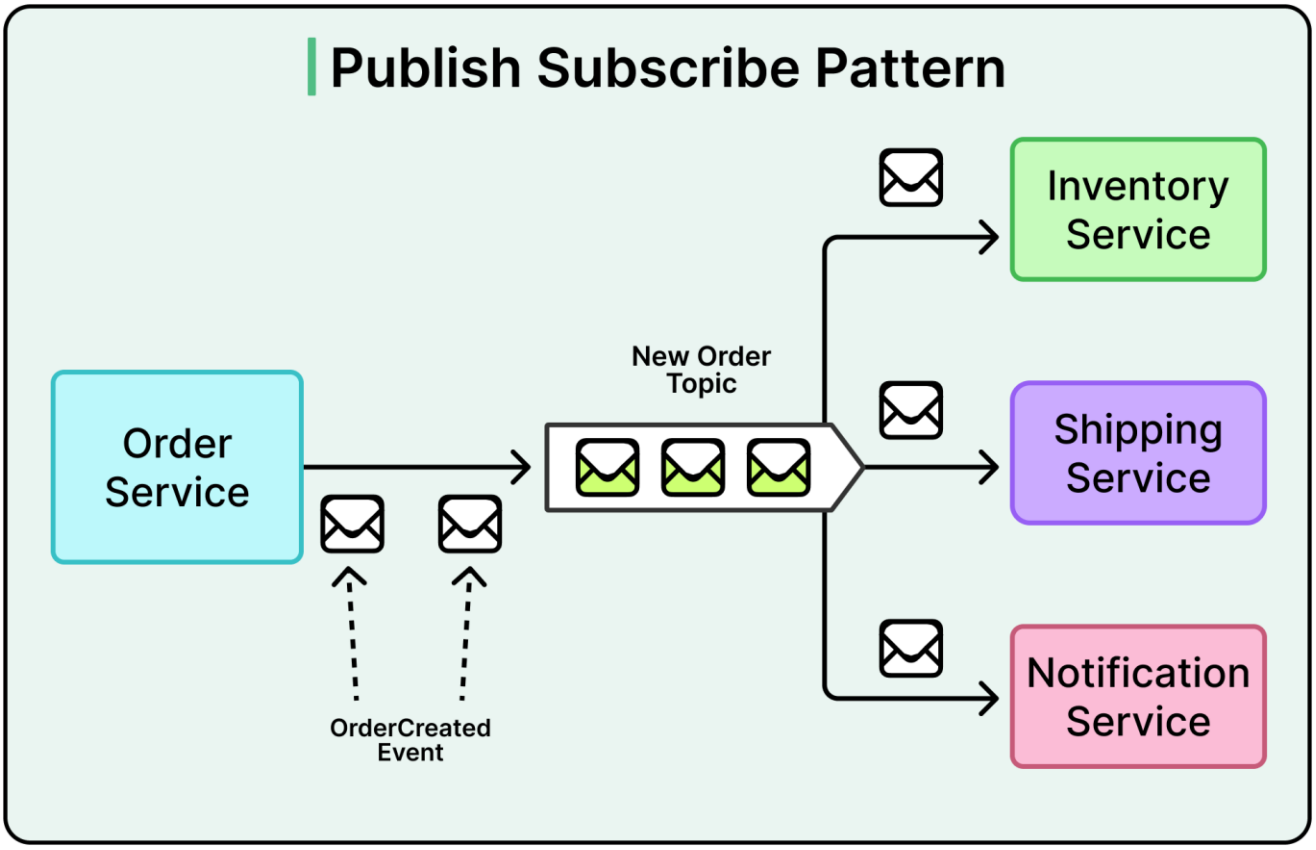


Рис. 1.6 Модель Publish/Subscribe для розподілу подій між мікросервісами

Сильні сторони такої моделі — простота, гнучкість, швидкість і велика різноманітність сценаріїв використання.

Слабкі сторони — складність при побудові дійсно відмовостійких конфігурацій. Забезпечити гарантовану доставку або створити кластер, який буде стабільним під великим навантаженням, можливо, але це потребує глибоких знань і ретельного налаштування.

Хмарний керований підхід

Ще один поширений варіант — хмарні керовані черги, які часто реалізуються за моделлю Simple Queue Service. Їхня головна перевага полягає в тому, що розробнику не потрібно розгортати сервери, стежити за їхнім станом чи займатися балансуванням навантаження. Весь технічний бік бере на себе провайдер хмари.

Такі черги підходять передусім для зв’язку між сервісами в межах хмарної інфраструктури. Вони добре інтегруються з іншими хмарними інструментами, підтримують гарантовану доставку й автоматичне повторне повернення повідомлень, а також дозволяють уникати прив’язки до конкретної технології — сьогодні система може працювати на одній платформі, завтра бути перенесена на іншу.

Хмарні черги — це простий, надійний і невибагливий варіант для невеликих та середніх проєктів, які цінують стабільність та мінімальну підтримку.

Високошвидкісні інструменти комунікації

Для проєктів, де важливими є мінімальні затримки, висока пропускна здатність і підтримка різних патернів взаємодії (pub/sub, queue, request/response), існує клас систем, які оптимізовані саме під швидкісний обмін повідомленнями між мікросервісами. Вони можуть працювати як у беззбережному (неперсистентному) режимі, так і з підтримкою потокової обробки й реплікації.

Особливість таких платформ — це можливість легко застосовувати різні моделі взаємодії, швидко розгортати інфраструктуру та масштабувати її при зростанні кількості сервісів. Такі системи ідеально підходять для:

- внутрішніх комунікацій між мікросервісами,
- подій реального часу,
- високошвидкісних конвеєрів,
- телеметрії, сигналів, повідомлень короткого життя.

Швидкість таких платформ часто досягається за рахунок зменшеної складності зберігання даних та ефективної архітектури.

Платформи для кастомних сценаріїв та транзакційної логіки

Існують системи, які дозволяють побудувати повністю індивідуальну логіку чергування, створювати черги зі складними правилами, залежностями, пріоритетами, власними алгоритмами маршрутизації. Вони використовують транзакційність, що дозволяє об’єднувати декілька операцій над чергами в одну атомарну дію.

Це відкриває можливість створювати сценарії, які недоступні у більшості класичних черг. Наприклад:

- переносити повідомлення між кількома чергами в межах однієї транзакції;
- будувати каскади операцій;
- реалізовувати складні пріоритетні моделі;
- створювати повноцінні інструменти оркестрації на основі черг.

Такі платформи можуть працювати як звичайні брокери, але також надають можливість зробити власну систему черг, адаптовану під унікальні бізнес-вимоги.

Протоколи взаємодії та їхні обмеження

При роботі з чергами існує два фундаментальних підходи до того, як повідомлення прив’язується до обробника:

1. Прив’язка задачі до постійного з’єднання

У цій моделі консюмер підтримує активне з’єднання з платформою. Переваги такого підходу:

- дуже низька затримка доставки;
- миттєве повернення задачі при помилці;
- точний контроль над життєвим циклом повідомлення.

Якщо з’єднання рветься, система може одразу повернути задачу в чергу. Проте масштабування тут ускладнюється: кожне з’єднання прямо взаємодіє з конкретним вузлом черги.

2. Взаємодія без постійного з’єднання (HTTP-подібна модель)

У цій моделі немає довгострокового підключення. Передавання повідомлень працює подібно до REST чи SQS. Масштабування значно простіше — достатньо додати нові вузли або балансувальники.

Але є критично важлива вимога: система повинна автоматично повертати повідомлення, якщо консюмер його не обробив.

Інакше повідомлення може «зависнути» назавжди. У більшості хмарних рішень таке автоматичне повернення реалізовано на рівні платформи.

Світ платформ повідомлень охоплює широкий спектр моделей — від потокових логів і класичних черг до швидкісних протоколів взаємодії та систем, що дозволяють створювати унікальну кастомну логіку. Кожен підхід має свої сильні сторони та сфери застосування.

- Журнальні потокові системи забезпечують аналіз, аудит і масштабовану роботу з подіями.
- Традиційні черги — універсальна основа для мікросервісних взаємодій.
- Хмарні керовані рішення — це простота, автономність і мінімум підтримки.
- Швидкісні комунікаційні платформи — вибір для систем реального часу.
- Кастомні платформи — оптимальний інструмент для складних бізнес-процесів.

Розуміння цих класів дозволяє приймати зважені рішення при побудові архітектури, обирати необхідні моделі та забезпечувати стабільність, продуктивність і масштабованість програмних систем будь-якого рівня складності.

2 ПРОБЛЕМАТИКА ЧЕРГ

2.1 Перший рівень проблем

Навіть тоді, коли система працює стабільно, не падає та не демонструє жодних очевидних збоїв, усередині самої логіки черги можуть бути закладені серйозні архітектурні обмеження. Вони не пов’язані з помилками програмування чи неполадками обладнання — це властивості алгоритмів, на яких побудована черга. Саме тому важливо розуміти, які існують моделі організації черг та які проблеми вони можуть породжувати в реальних системах.

Алгоритми впорядкування повідомлень

FIFO (First In, First Out) — це базовий і найбільш інтуїтивно зрозумілий механізм упорядкування повідомлень у черзі, який гарантує сувору послідовність їх обробки. Повідомлення, що потрапило до черги першим, буде і першим передане на обробку. Такий підхід забезпечує найвищу передбачуваність і прозорість роботи системи: порядок надходження завжди відповідає порядку виконання. Саме тому FIFO є незамінним у тих архітектурах, де хронологічна послідовність подій має вирішальне значення — наприклад, у фінансових транзакціях, обліку подій, потоковій аналітиці або в задачах, де навіть невелике порушення порядку може призвести до помилок у логіці роботи. Завдяки цьому FIFO формує рівний, послідовний і передбачуваний потік задач, що істотно спрощує подальшу обробку та діагностику системи.

LIFO (Last In, First Out) — це підхід, за яким останнє додане до черги повідомлення буде оброблене першим. По своїй суті LIFO є аналогом класичної структури даних *стек*, де обробка завжди відбувається «з вершини». У контексті розподілених систем та реальних брокерів повідомлень цей метод трапляється вкрай рідко, оскільки суперечить природній логіці більшості бізнес-процесів: нові задачі витісняють старі, що може спричинити затримки, збої в часі або навіть «зависання» повідомлень у глибині черги. Найчастіше LIFO застосовується у внутрішніх алгоритмах програм або при реалізації локальних механізмів, де

важливо обробляти саме найсвіжіші дані — наприклад, у реверс-операціях, внутрішніх буферах або при реалізації механізмів скасування дій. Проте в реальних системах обміну повідомленнями LIFO практично не використовується через непридатність для більшості сценаріїв та низьку сумісність з потоковою логікою.

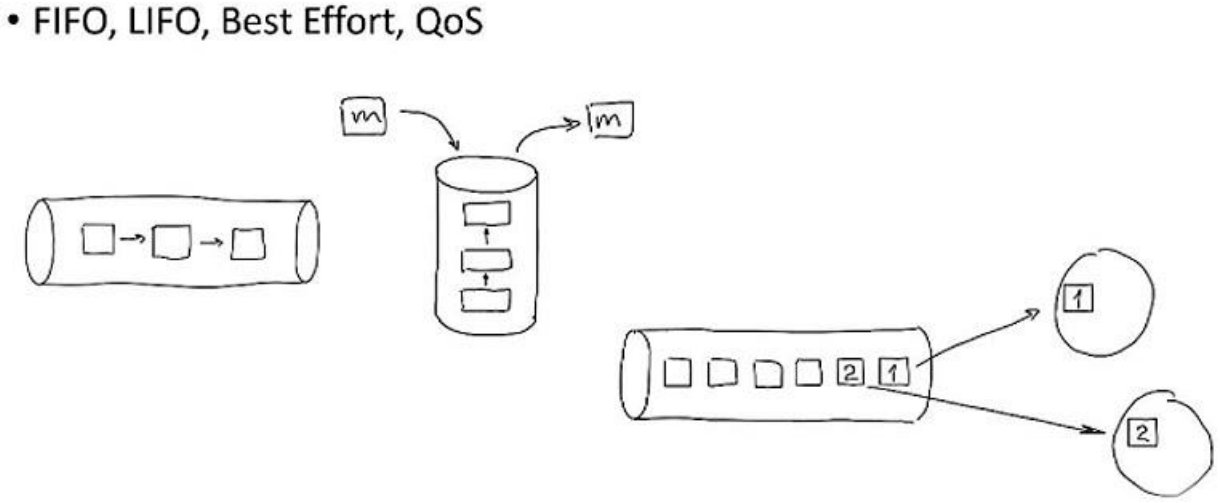


Рис. 2.1 Алгоритм LIFO

Best Effort Queue — найбільш уживаний варіант у практичних системах. За зовнішньою структурою така черга нагадує FIFO, але має одну важливу особливість: якщо споживач уже забрав повідомлення, але не зміг його обробити (завис, упав, отримав помилку), це повідомлення повертається назад. Після повернення брокер може змінити його позицію або вставити у місце, де буде найкоректніше з погляду загальної роботи системи. Цей підхід дозволяє не блокувати обробку всієї черги через одну невдалу задачу й забезпечує високий рівень живучості системи.

Пріоритизація повідомлень

Деякі задачі в системі потребують обробки раніше за інші — наприклад, VIP-запити, критичні транзакції, події з високим пріоритетом. Для цього брокери повинні підтримувати механізм пріоритизації.

Проте в певних архітектурах це неможливо. Наприклад, у потокових журнальних системах кожне нове повідомлення завжди додається в кінець журналу,

і порядок є незмінним. Переставляти повідомлення або штучно піднімати їх угору неможливо — лог завжди зберігає історичну послідовність.

Тому платформи з незмінним журналом подій чудово підходять для аналізу, але не підходять для сценаріїв з динамічними пріоритетами.

Проблеми підчерг

У масштабних розподілених системах досить часто виникає спокуса створити велику кількість дрібних, ізольованих черг — наприклад, формувати окрему чергу для кожного клієнта, підсистеми або навіть типу задачі. На перший погляд, такий підхід здається ефективним: кожен клієнт отримує власний «канал» обробки, немає конкуренції за ресурси, а задачі не блокують одна одну. Здається, що це забезпечує рівномірний розподіл навантаження та передбачувану обробку.

Однак у реальній практиці створення великої кількості підчерг породжує цілу низку нових архітектурних та експлуатаційних проблем, які часто перекривають усі передбачувані переваги.

По-перше, значне збільшення кількості черг неминуче створює додаткове навантаження на брокер. Кожна черга — це окрема структура з власними метаданими, індексами, механізмами блокування, статистикою, лічильниками та алгоритмами доставки. Усе це споживає пам’ять, процесорний час та ресурси зберігання.

По-друге, кожна окрема черга потребує власного керування, зокрема відстеження станів, TTL, затримок, dead-letter логіки, механізмів ретраю тощо. Чим більше таких «дрібних» черг, тим складнішою стає система з погляду адміністрування та підтримки.

По-третє, зростає час маршрутизації та внутрішнього розподілу повідомлень, адже брокеру доводиться постійно визначати, у яку саме чергу направити подію, обслуговувати правила маршрутизації та розподіляти навантаження між сотнями чи тисячами черг.

По-четверте, у системах із дуже великою кількістю підчерг виникає проблема неефективного балансування. Частина черг може бути перевантажена, тоді як інші

— простоювати. Це призводить до нерівномірного використання ресурсів і, відповідно, до загального зниження продуктивності.

У підсумку отримуємо парадоксальну ситуацію: архітектура, яка теоретично мала б оптимізувати обробку задач і підвищити швидкість роботи сервісів, на практиці може спричинити помітне уповільнення, збільшення затримок і зниження пропускної здатності системи. Тому використання великої кількості підчерг потребує дуже ретельного планування та зазвичай має сенс лише у специфічних, добре контрольованих сценаріях.

Повтори, відкладені задачі та неможливість повернення повідомлення

У системах, орієнтованих на потокове передавання, інколи виникають проблеми з повторним відтворенням або повторною обробкою задач. Якщо брокер не підтримує операцію «повернути повідомлення назад у чергу», доводиться вручну створювати повторне повідомлення або застосовувати окрему логіку повторів.

У деяких чергах повторне повернення просто заборонене. Тоді система потребує:

- dead-letter queue (DLQ) — окремої «черги померлих повідомлень»,
- механізму повторної постановки задач у роботу,
- логіки обробки помилок та фільтрації некоректних повідомлень.

Цей підхід працює, але вимагає додаткової логіки та підвищує складність архітектури.

Проблеми часу життя задачі (TTL, TTR, Putback)

Навіть у тих системах, де повернення задачі можливе, існують часові обмеження, які можуть спричинити проблеми.

• **TTL (Time To Live)** — це максимальний допустимий час, протягом якого повідомлення може зберігатися в черзі перед тим, як буде визнане «простроченим». Якщо жоден консюмер не зміг обробити повідомлення у межах цього періоду, брокер вважає його таким, що втратило актуальність, і виконує визначену політику: або повністю видаляє його, або автоматично переміщує до спеціалізованої черги невдалих або проблемних повідомлень — **Dead Letter Queue (DLQ)**. Такий

механізм запобігає нескінченному накопиченню застарілих задач і підтримує чистоту та актуальність основного потоку.

• **TTR (Time To Retry)** — це інтервал часу, який система виділяє консюмеру на виконання обробки повідомлення після того, як той забрав його з черги. Якщо протягом цього часу консюмер не підтвердив обробку — наприклад, завершився помилкою, завис, втратив зв’язок або просто не встиг виконати необхідні операції — повідомлення автоматично повертається назад до черги. Таким чином система гарантує, що жодна задача не пропаде через тимчасові збої або невдалі спроби обробки.

• **Putback** — це спеціальна операція або механізм примусового повернення повідомлення в чергу у разі невдалого виконання. На відміну від автоматичного повернення після закінчення TTR, putback може виконуватися за ініціативою самого консюмера чи логіки обробки. Наприклад, якщо під час обробки була виявлена тимчасова недоступність зовнішнього сервісу, інтеграції або бази даних, консюмер може свідомо повернути повідомлення, щоб повторити спробу пізніше, коли зовнішні умови стабілізуються. Це дозволяє будувати більш гнучкі та адаптивні механізми обробки складних задач.

Проблема полягає в тому, що часові межі не завжди відповідають реальній тривалості задач. Якщо консюмер занадто повільний або повідомлення вимагає значної обчислювальної роботи, воно може помилково вважатися «простроченим» і повернутися до черги передчасно, що викликає дублювання, нескінченні повтори або накопичення відкладених задач.

2.2 Другий рівень проблем

Проблеми другого рівня виникають не через випадкові збої чи помилки програмістів, а як природний наслідок тих архітектурних рішень і алгоритмів, які закладені в саму основу черги. Якщо перший рівень стосується особливостей упорядкування, пріоритетів, TTL чи моделей повернення задач, то другий рівень описує те, як вся система поводиться в реальному середовищі, коли вона стикається

з навантаженням, нерівномірними потоками даних, браком ресурсів або неправильним балансуванням.

Це вже проблеми не окремої задачі, а цілої архітектури, які можуть призвести до деградації, втрати пропускної здатності та невиконання критичних операцій.

Одним із перших проявів системної деградації є голодання низькопріоритетних задач. Воно виникає у платформах, які підтримують пріоритезацію. У таких системах існують два або більше потоки повідомлень:

- високопріоритетні — критичні для бізнесу або реального часу;
- низькопріоритетні — фонові операції, технічні задачі, менш важливі повідомлення.

Якщо потік високопріоритетних повідомлень стабільно заповнює всю пропускну здатність консюмерів, то низькопріоритетні задачі можуть робити нескінченне очікування, не маючи шансу бути обробленими.

Цей ефект особливо помітний у великих мікросервісних системах. Наприклад:

- події від критичних сервісів «забивають» чергу,
- а задачі обслуговування, як-от очищення кешу, оновлення індексів чи генерація статистики, взагалі не отримують процесорний час.

Таким чином пріоритізація, яка мала забезпечити оптимальне використання ресурсів, у певних випадках перетворюється на механізм, який блокує важливі, але не термінові операції, що згодом може вплинути на стабільність усієї системи.

Обмеження пропускної здатності та нерівномірність потоків

Будь-яка черга має верхню межу пропускної здатності (throughput). Це результат того, що певні компоненти системи:

- не можуть працювати паралельно,
- не масштабуються горизонтально,
- мають певні фізичні межі продуктивності.

Такі елементи стають вузькими місцями (bottlenecks).

Приклади:

- одиночний потік обробки, який не можна розділити;

- алгоритм, який вимагає суворої послідовності (як у деяких потокових логах);
- фізичний диск, який працює на межі швидкості запису;
- один сервер, на який припадають усі задачі певного типу.

Коли система впирається у свій ліміт, черга починає стрімко зростати, збільшується час обробки, а спроби «наздогнати» потік роблять ситуацію тільки гіршою.

Великі системи часто стикаються з тим, що навіть за достатньої кількості продюсерів та консюмерів, throughput визначається найслабшою ланкою — і саме вона диктує поведінку всієї інфраструктури.

Вплив алгоритмів зберігання та доступу на продуктивність

Продуктивність черги безпосередньо залежить від того, як саме вона читає та записує дані.

Наведемо кілька прикладів:

- Лінійний запис у лог — один із найшвидших можливих варіантів, він дозволяє додавати дані зі сталою швидкістю.
- Випадковий доступ до диску різко знижує продуктивність, особливо на HDD.
- Фрагментація записів та складні структури черг потребують додаткових операцій.
- Індексація, пріоритети, TTL, TTR, сортування — усе це збільшує накладні витрати.

Таким чином, навіть якщо API виглядає однаково, різні алгоритми всередині черг забезпечують абсолютно різний рівень продуктивності. Саме тому дві черги з однаковим функціоналом можуть відрізнятися у швидкості в десятки разів.

Обмеження масштабу: коли черга «не росте»

Масштабування — одна з ключових властивостей сучасних систем. Якщо система не масштабована горизонтально, вона неминуче стикається з проблемами.

Наприклад:

- черга, що працює на єдиному вузлі, навіть із реплікацією, забезпечує відмовостійкість, але не пропускну здатність;

- якщо всі повідомлення проходять через один лог, то масштабування напряму обмежене кількістю партицій;
- системи, побудовані без шардінгу, з часом не витримують навантаження.

Таким чином, система, що не враховує горизонтальну масштабованість, рано чи пізно досягає точки, коли обсяг запитів перевищує її можливості. Після цього відбувається накопичення задач, зростає latency, а в кінцевому підсумку — відмова сервісів.

Обмеженість ресурсів та поведінка під час переповнення

У реальних системах ресурси не є нескінченними:

- оперативна пам’ять закінчується,
- дисковий простір вичерпується,
- мережеві з’єднання сягнули максимуму,
- CPU не встигає обробляти задачі,
- файлові дескриптори вичерпуються.

Коли це стається, черга переходить у режим некерованості.

Типова поведінка при переповненні:

- повідомлення починають втрачатися;
- черга перестає приймати нові події;
- з’являються нескінченні ретраї;
- dead-letter черги заповнюються;
- брокер переходить у деградований режим або аварійно завершується;
- починається загальний лавиноподібний збій системи.

Найгірше те, що такі ситуації часто поширюються каскадно: одна черга «захлинається», інші не можуть отримати її події, мікросервіси перестають взаємодіяти між собою, і вся інфраструктура втрачає стабільність.

Другий рівень проблем — це рівень системних ефектів, які виникають не через помилку конкретного компонента, а через взаємодію всіх частин під навантаженням.

Голодання задач, низька пропускна здатність, вплив алгоритмів доступу на продуктивність, неможливість масштабуватися, обмежені ресурси — усе це

формується ще на архітектурному рівні та проявляється у великих, реальних системах.

Щоб побудувати стабільну, відмовостійку і масштабовану інфраструктуру, розробник має аналізувати не лише логіку окремої черги, а й її поведінку як частини складної, багаторівневої екосистеми.

Саме правильне розуміння і врахування цих обмежень дозволяє уникнути критичних провалів і забезпечити довготривалу ефективність системи.

2.3 Третій рівень проблем

Початкові концепції черг формувались у той час, коли обчислювальні системи працювали в умовах майже повної ізольованості. Черги створювалися на рівні заліза або програмної логіки всередині одного процесу, де існували стабільні, контрольовані умови. У таких локальних системах не було жодних мережевих втрат, а взаємодія між компонентами відбувалася через спільну пам’ять або прямий обмін даними.

У подібному середовищі:

- жоден байт даних не міг загубитися випадковим чином;
- алгоритми виконувалися послідовно й передбачувано;
- затримки були мінімальними та стабільними;
- операції не вимагали складних протоколів підтвердження;
- не було жодної невизначеності — усе було детермінованим.

Черги у таких системах працювали «чисто»: якщо повідомлення було додано до черги, воно гарантовано оброблялося. Але з розвитком обчислювальної техніки системи перестали бути ізольованими. Виникла потреба у взаємодії між серверами, між дата-центрами, між розподіленими мікросервісами — і саме тут ідеальна модель почала руйнуватися.

Мережева невизначеність та фундаментальні обмеження

Мережа за своєю природою є середовищем із невизначеною поведінкою (undefined behavior). На відміну від локальної пам’яті, у мережі неможливо бути впевненим, що надіслане повідомлення:

- обов’язково буде доставлене,
- не буде пошкоджене,
- не прийде повторно,
- не загубиться в дорозі,
- не буде отримане із критичною затримкою,
- не буде доставлене у неправильному порядку.

Ця невизначеність не є багом — це фундаментальна властивість мережевих протоколів. І саме тому у розподілених системах неможливо досягти абсолютної гарантії доставки чи синхронізації між вузлами.

Exactly once та строгий порядок у реальних системах

У реальних системах зазвичай реалізуються лише дві гарантії:

- At most once: повідомлення може бути втрачено, але не буде повторено.
- At least once: можливі дублювання, але втрата виключена.

Гарантія exactly once є фактично окремим випадком задачі двох генералів, а тому:

- або потребує складних протоколів збереження станів;
- або вимагає транзакцій між обома сторонами;
- або реалізується як marketing-функція, а насправді працює як «at least once + дедуплікація».

Більшість інженерів у своїй практиці рідко зустрічають системи, які дозволяють виконати дію рівно один раз без повторів або втрат.

І лише одного разу я побачив систему, де це було реалізовано — але рішення було далеко від традиційних алгоритмів.

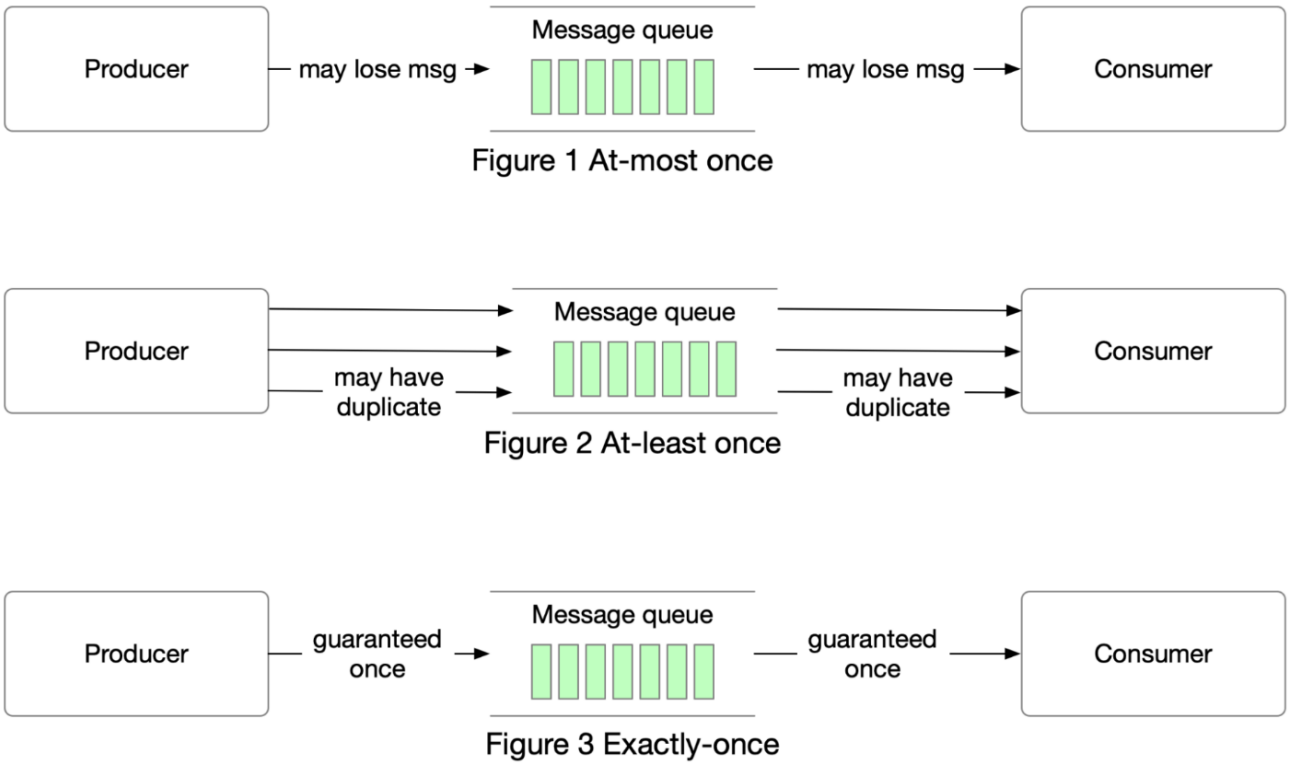


Рис. 2.2 Порівняння моделей доставки повідомлень: At-most once, At-least once та Exactly once

Exactly once через людський контроль

У цій системі при будь-якій невизначеності (наприклад, коли консюмер виконав дію, але підтвердження не дійшло до брокера) повідомлення автоматично переміщалось у dead-letter queue.

Далі ситуацію аналізував оператор, людина, яка:

- вручну перевіряла історію виконання;
- вирішувала, чи потрібно повторити повідомлення;
- чи було воно вже виконане;
- чи слід його відправити іншому консюмеру.

Тобто в архітектурі фактично існував «третій генерал», або «божественний спостерігач», який знаходився над усією системою та міг приймати остаточне рішення. Саме цей зовнішній фактор дозволяв обійти класичну нерозв’язність задачі.

Роль дисків та апаратних затримок у розподілених чергах

Окрім мережі, на черги впливають ще й характеристики апаратного забезпечення, зокрема дисків. Черги, побудовані на персистентних логах або файлових системах, дуже залежні від того, як швидко вони можуть писати й читати дані.

Типові проблеми дискової підсистеми:

- Пропускна здатність (Throughput)

  Диск може просто не встигати обробляти вхідний потік повідомлень.
- Затримки (Latency)

  Деякі сховища мають передбачувану затримку (NVMe SSD), інші — хаотичну (HDD, мережеві диски).
- Переповнення буферів

  Під час піків навантаження записи починають збиратися в черги на рівні самого диска.
- Час доступу

  Лінійний запис у лог надзвичайно швидкий; випадковий доступ ускладнює обробку.
- Зношування та деградація

  SSD у потокових системах зношуються швидше, що впливає на стабільність у довгостроковій перспективі.

Ці фактори накладаються на мережеву невизначеність і створюють багатошарову систему, у якій навіть найідеальніший алгоритм черги змушений працювати в умовах постійної непередбачуваності.

Черги були створені для роботи в стабільних, контрольованих середовищах, але сьогодні вони існують у світі, де панують невизначеність, затримки, втрати, переповнення, збої та асиметричні канали зв’язку.

Мережева поведінка, яка не гарантує доставки, фундаментальні логічні парадокси розподілених систем, обмеження дисків, час доступу до сховища, переповнення пам’яті — усе це створює реальність, у якій неможливо забезпечити ідеальну доставку та синхронізацію.

Саме тому на практиці черги мають працювати не в умовах ідеальної моделі, а в середовищі постійних ризиків. Розробник повинен враховувати:

- можливість втрат,
- необхідність повторів,
- механізми відновлення,
- існування dead-letter черг,
- вплив апаратних обмежень,
- людський фактор як крайній інструмент вирішення неоднозначностей.

2.4 Четвертий рівень проблем

У будь-якій розподіленій системі, рано чи пізно, настає момент, коли щось виходить з ладу. Це можуть бути як короткочасні, так і тривалі або навіть незворотні відмови. І саме в цей момент проявляється справжня якість черги та архітектури, на якій вона побудована.

Хоч у документації часто обіцяють «надійність», «високу доступність» чи «гарантії збереження», у реальності кожен шар інфраструктури має слабкі місця, а помилки можуть накопичуватись або призводити до критичних наслідків.

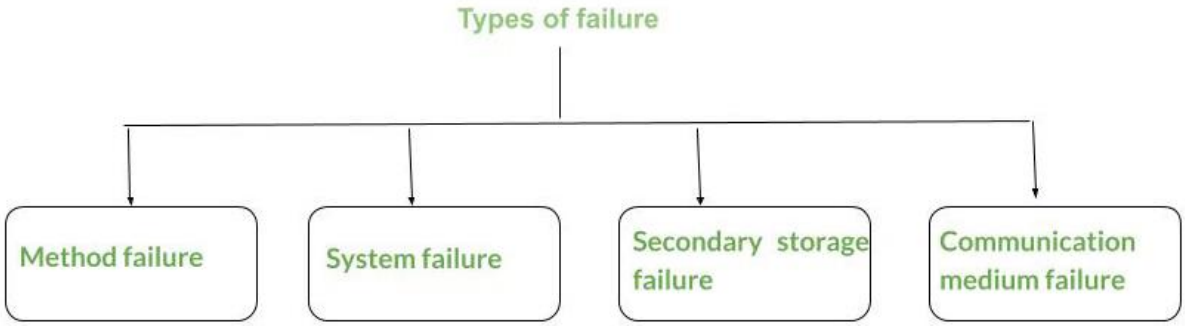


Рис. 2.3 Основні типи відмов у розподілених системах

Хибні уявлення про захист даних та міфи про безвідмовність

Серед інженерів існує поширена ілюзія, що достатньо використати RAID або іншу технологію відновлення дисків — і втрати даних не станеться. Але RAID захищає лише від дуже вузького сценарію: виходу з ладу одного окремого диска.

У реальних системах можливі значно складніші й руйнівніші сценарії:

- виходить з ладу не один диск, а цілий сервер;
- виходять з ладу кілька дисків одночасно;
- відбувається коротке замикання або стрибок напруги;
- ламається контролер живлення;
- ураження струмом приходить через мережеве обладнання;
- пошкоджується RAID-контролер, який містить власні кеші.

У моїй практиці був випадок, коли машина з 16 дисками мала RAID-масив, продуману схему резервування, але один потужний електричний розряд через мережевий інтерфейс спалив усе обладнання — уціліли лише два диски. RAID у цьому випадку виявився абсолютно безсилим.

Тобто потрібно усвідомлювати: відновлення дисків ≠ відмовостійкість системи, і навіть надійне сховище може виявитися лише однією з ланок, що може відмовити.

Тимчасові відмови та ефект “Split Brain”

Тимчасові проблеми — це найчастіше:

- зникнення живлення;
- розрив мережевого каналу;
- короткочасна недоступність одного з вузлів;
- зависання процесу або сервісу.

Коли система розподілена на кілька вузлів, тимчасова відмова може призвести до одного з найнебезпечніших станів — split brain.

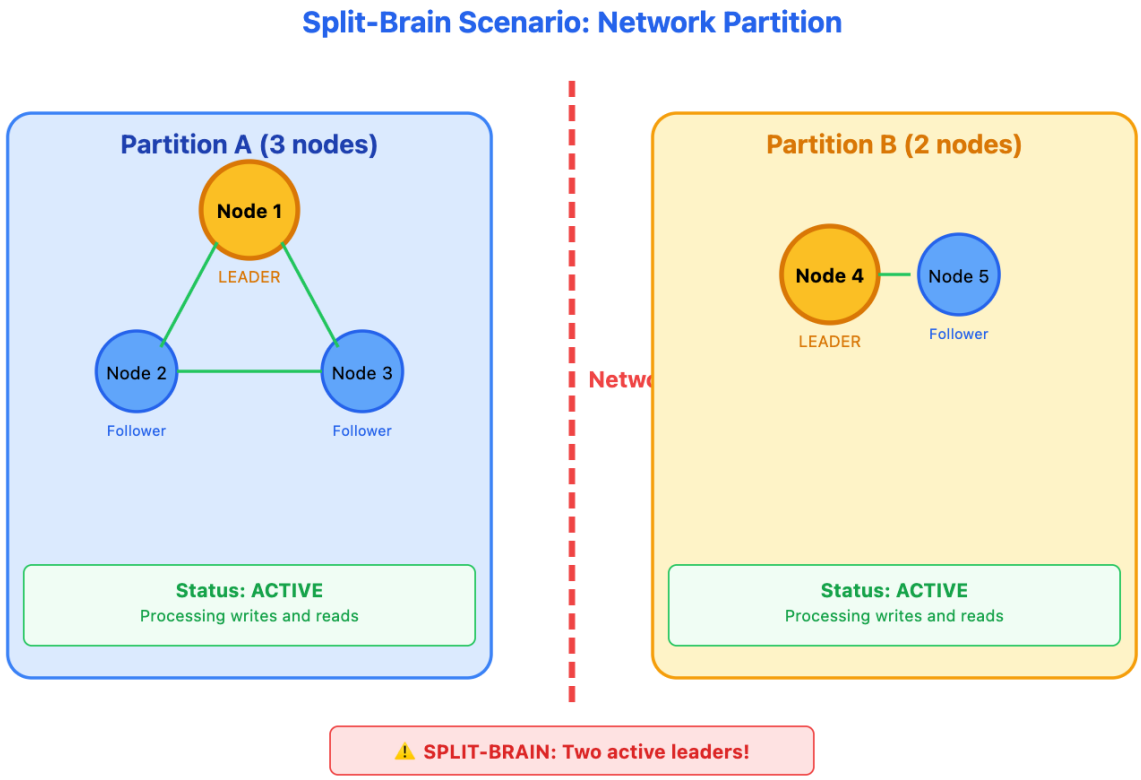


Рис. 2.4 Сценарій split-brain, спричинений мережевою сегментацією кластера

Що таке split brain?

Це стан, коли:

- система розпадається на дві (або більше) «підсистеми»,
- кожна з них вважає себе основною,
- обидві продовжують робити записи,
- але не можуть узгодити свій стан.

Обидві частини «живі», але не знають про це. У результаті виникають:

- розбіжності у даних;
- дублювання або втрата подій;
- одночасне виконання конфліктних операцій;
- порушення цілісності черг або логів.

Стан split brain майже завжди призводить до складних сценаріїв відновлення й часто вимагає ручного втручання, реконструкції даних або навіть повного відкату.

Постійні та незворотні відмови

Постійні відмови — це вже не короткочасна аномалія, а ситуація, у якій компонент повністю виходить з експлуатації. Це може бути:

- фізичне пошкодження диска;
- вихід з ладу сервера;
- неправильне оновлення прошивки;
- пожежа або затоплення обладнання;
- повний вихід з ладу цілого дата-центру.

Часто вважають, що дата-центр — це непорушна, майже міфічна фортеця, яка захищена від усього. Але історія знає багато випадків:

- займання систем охолодження;
- аварії на електростанціях, що живлять дата-центр;
- повне відключення живлення;
- помилки адміністраторів, що виводили з ладу тисячі серверів;
- стихійні лиха.

Тому справжня стратегія відмовостійкості повинна враховувати масштаб катастрофи аж до втрати дата-центру, а не лише дрібні апаратні збої.

Як відмови впливають на ключові метрики системи

Будь-який збій в інфраструктурі впливає на дві фундаментальні характеристики, які визначають поведінку черги:

1. Доступність (Availability)

Це здатність системи приймати нові повідомлення. Ця властивість критична для продюсерів: якщо черга недоступна, вони не можуть продовжувати роботу, і система частково чи повністю зупиняється.

Низька доступність призводить до:

- затримок у бізнес-процесах;
- накопичення запитів;
- необхідності застосовувати backpressure;
- втрат даних або повторних спроб доставки вручну.

2. Надійність (Durability)

Це гарантія, що повідомлення, яке вже записане у систему, не буде втрачено, незважаючи на збої.

Якщо durability недостатня, то навіть ті дані, які продюсер успішно відправив і отримав підтвердження, можуть бути втрачені через:

- поломку дисків;
- падіння сервера;
- некоректні алгоритми запису;
- відмову дата-центру.

Надійність — це те, що дозволяє вважати чергу «довіреним середовищем» для обміну подіями. Її втрата означає, що система перестає бути передбачуваною.

Відмови — як тимчасові, так і постійні — є невід'ємною частиною роботи будь-якої розподіленої системи. Черги, що здаються простими у теорії, піддаються цілому спектру ризиків у реальності: від несуттєвих перебоїв у мережі до катастрофічних збоїв на рівні обладнання чи дата-центрів.

Саме тому архітектура чергування повинна бути побудована з урахуванням:

- втрати вузлів,
- втрати дисків,
- втрати цілих кластерів,
- зникнення мережі,
- затримок у записі,
- розділення кластерів на частини (split brain),
- ризиків, які неможливо передбачити алгоритмічно.

Система, яка не враховує ці фактори, рано чи пізно зазнає критичного збою.

3 ТОПОЛОГІЯ ЧЕРГ ТА ДОДАТКОВІ РЕСУРСИ

3.1 Топологія черг

Розглянемо, яким чином у розподіленому середовищі забезпечуються надійність та доступність, коли в архітектурі присутній брокер черг. Найпростіша, але, на жаль, одна з найпоширеніших у реальних системах конфігурацій — це так званий single instance.

У такій схемі існує лише один екземпляр брокера, до якого підключена одна черга. З одного боку до неї надсилає повідомлення продюсер, з іншого — єдиний консюмер їх зчитує та обробляє. На перший погляд це виглядає привабливо: мінімум компонентів, проста логіка, легко налаштувати та супроводжувати. Однак така модель має очевидний структурний недолік — будь-який збій брокера або вузла, на якому він запущений, одразу зупиняє всю систему обміну повідомленнями. У цьому випадку черга стає єдиною точкою відмови, а всі гарантії надійності та доступності фактично зводяться до якості одного сервера та його диска.

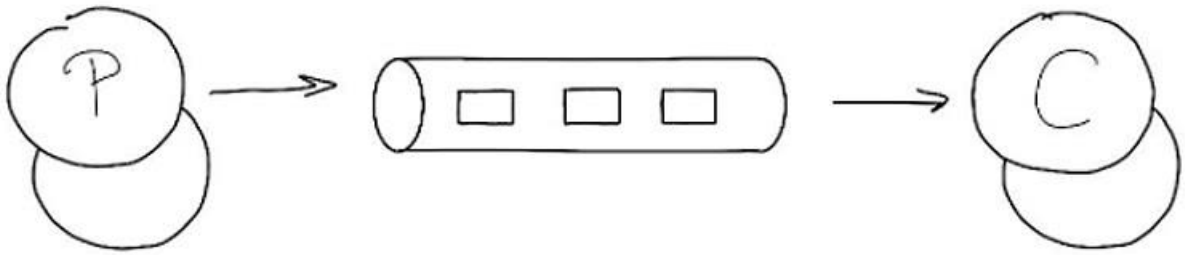


Рис. 3.1 Один брокер, одна черга

Недоліки такого підходу є очевидними та суттєвими, особливо коли мова йде про реальні розподілені системи, що повинні обробляти великі обсяги даних і працювати безперервно.

Перш за все, конфігурація з одним брокером не має жодної масштабованості. Уся система спирається на продуктивність одного-єдиного вузла. Коли цей брокер

досягає своїх апаратних чи програмних меж — процесорних, дискових або мережевих — вся інфраструктура перестає приймати нові повідомлення, навіть якщо продюсерів багато і вони готові продовжувати роботу. Це створює природне вузьке місце, яке неможливо обійти без зміни архітектури.

Другий критичний недолік — низький рівень доступності. Оскільки вся логіка черги розміщена на одному екземплярі, будь-яке переривання його роботи — коротке падіння, зависання, перезавантаження, оновлення, втрата мережевого з’єднання — одразу робить чергу недоступною. Продюсери не можуть надсилати події, консюмери не можуть завершити обробку, а вся система “застигає” в очікуванні відновлення єдиного вузла.

Третя проблема — високий ризик втрати даних і низька надійність. Якщо broker раптово завершує роботу або сервер перезавантажується внаслідок збою, велика ймовірність того, що частина повідомлень, які знаходилися в оперативній пам’яті або не встигли потрапити на диск, буде безповоротно втрачена. Навіть використання журналювання або дискової персистентності не гарантує абсолютної цілісності, якщо відсутні механізми реплікації та дублювання даних на інші вузли.

Таким чином, single instance робить систему крихкою, негнучкою та схильною до збоїв, а будь-яка відмова або пікове навантаження здатні миттєво вивести її з робочого стану. Це підкреслює необхідність переходу до більш стійких, кластерних архітектур, де відмова одного вузла не означає зупинку всієї інфраструктури.

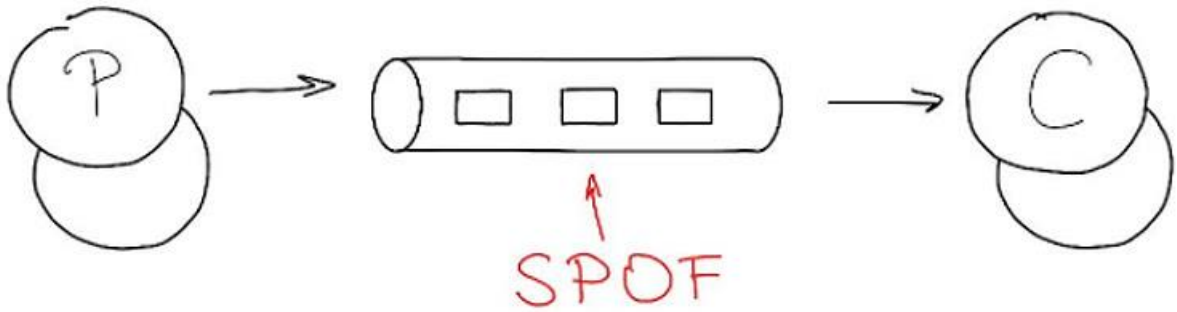


Рис. 3.2 Низький рівень варіативності відмовостійкості системи

Гарантії доставки у такій конфігурації залишаються на рівні $X \leq 1$ або $X \geq 1$, тобто повідомлення або можуть бути втрачені, або можуть бути доставлені з дублюванням. Проте саму систему можна поліпшити доволі просто — достатньо розгорнути декілька незалежних черг і направляти повідомлення в будь-яку чергу, яка доступна та працює в конкретний момент.

У варіанті multi-instance загальна поведінка системи помітно покращується. Така архітектура дозволяє масштабувати інфраструктуру горизонтально: що більше інстансів розгорнуто, то більше навантаження можна обробити. Крім того, зростає й доступність — якщо один брокер виходить з ладу або тимчасово недоступний, система все одно продовжує працювати, тому що інші черги залишаються активними.

Однак рівень надійності все ще залишається середнім. Кожен інстанс функціонує окремо, тому при його відмові існує ризик втрати частини повідомлень. У підсумку навіть у multi-instance-конфігурації гарантії залишаються слабкими та обмежуються все тими ж моделями $X \leq 1$ і $X \geq 1$, які не забезпечують повної безпеки даних.

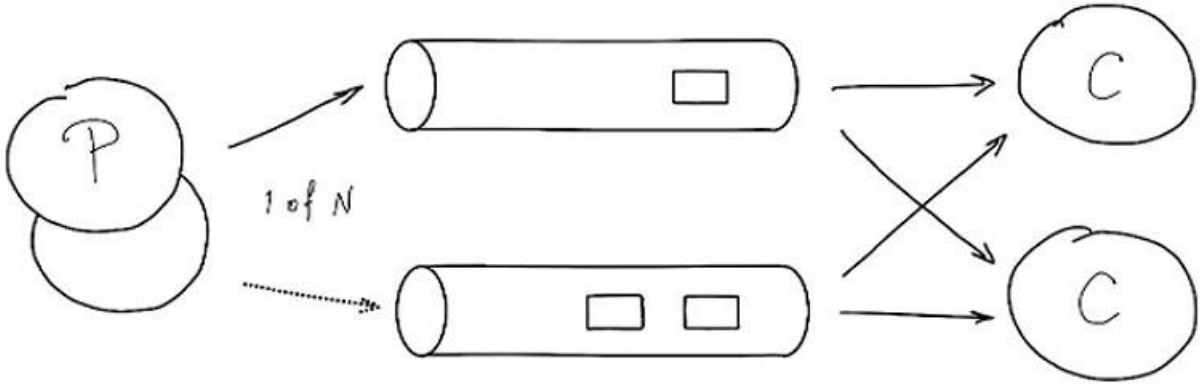


Рис. 3.3 Схема системи multi-instance

Суттєвим недоліком такої системи є те, що кожне повідомлення прив’язане лише до однієї конкретної черги. Це означає, що вся інформація зберігається на одному вузлі, і якщо з цією машиною щось трапляється — вона вимикається, виходить із ладу чи втрачає дані через збій — разом із нею зникають і всі повідомлення, які ще не були оброблені. Таким чином, навіть у масштабованій системі з кількома інстансами зберігається ризик втрати частини подій.

Вирішувати цю проблему можна різними способами. Один із найпростіших підходів — дублювання повідомлень у кілька черг одночасно, щоб одна й та сама подія зберігалася не на одному, а на кількох вузлах. Якщо один інстанс виходить з ладу, копії повідомлень усе ще залишаються в інших чергах, що підвищує загальну стійкість системи й зменшує ризик незворотних втрат.

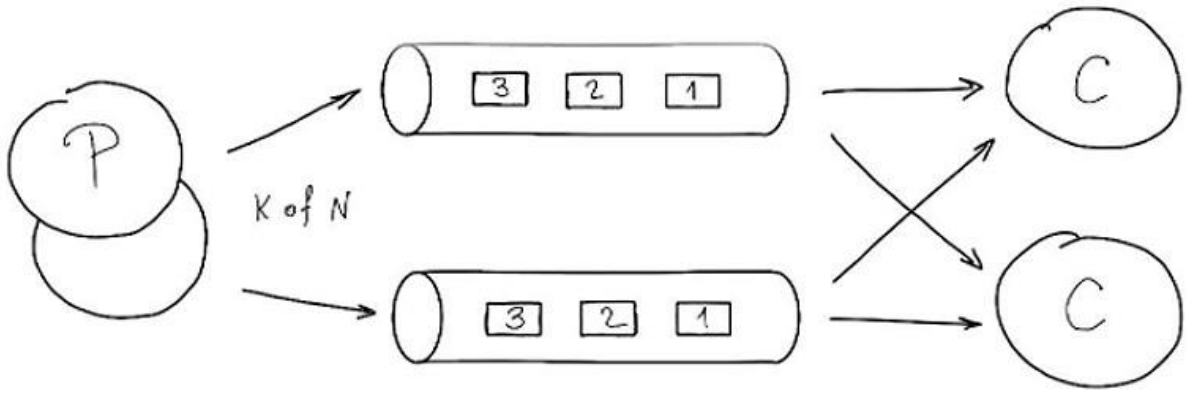


Рис. 3.4 Повідомлення належить лише одній черзі

Суттєвим недоліком такої архітектури є те, що кожне повідомлення фізично належить лише одній конкретній черзі. Якщо з машинною, на якій працює цей інстанс, трапляється будь-який збій — відмова диска, падіння сервера чи навіть звичайний перезапуск, — зберігання повідомлень опиняється під загрозою, і частина даних може бути безповоротно втрачена.

Щоб зменшити цей ризик, застосовують різні підходи. Один із найпростіших — дублювати повідомлення одразу в кілька черг, тобто створювати кілька копій одного й того ж запису. У такому разі втрата одного інстансу не призводить до втрати подій загалом, адже інші черги продовжують зберігати дублікати повідомлень і можуть віддати їх на обробку.

У такій конфігурації система отримує значно вищі рівні доступності та надійності. Навіть якщо одна з черг перестане працювати або стане недоступною, повідомлення все одно «доживе» до свого споживача через інші інстанси, у яких зберігаються його копії. Таким чином, ймовірність повної втрати даних суттєво зменшується.

Втім, разом із цим виникає інша складність. Під час багаторазової доставки того самого повідомлення потрібно стежити, щоб воно не оброблялося кілька разів. Для цього або роблять операції ідемпотентними (тобто такими, що можуть виконуватися повторно без змін результату), або впроваджують механізм перевірки: чи вже застосовувалося повідомлення раніше. Це додає додаткову логіку та накладні витрати.

Більш елегантним рішенням є використання підходу, запозиченого з реплікації баз даних. У ньому продюсер і консюмер працюють лише з лідером, тобто основним вузлом, який приймає всі записи. Інші вузли виконують роль реплік і просто синхронно або асинхронно копіюють дані від лідера. Якщо лідер виходить з ладу, будь-яка з реплік може автоматично стати новим лідером і продовжити роботу, зберігши цілісність і доступність системи без втрати повідомлень.

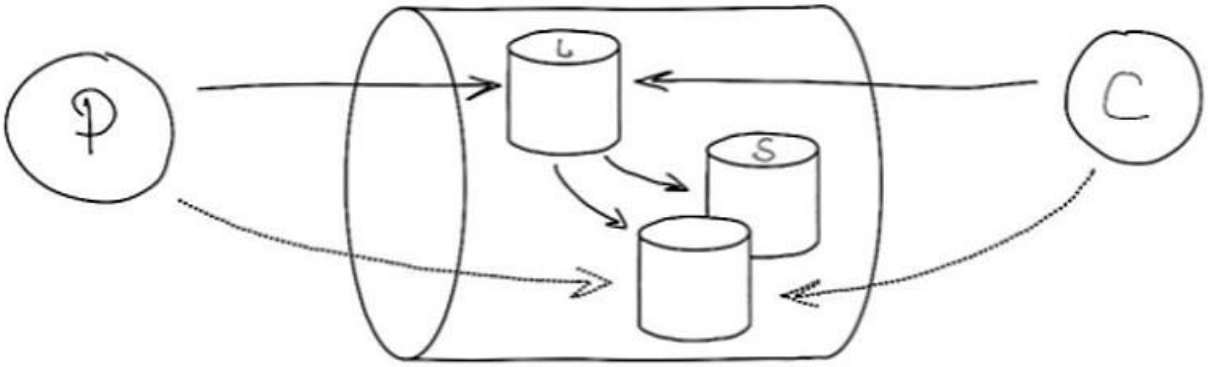


Рис. 3.5 Процес реплікації

Якщо замість окремих, ізольованих черг використовувати репліка-мережі, система одразу переходить на суттєво вищий рівень якості. У такій конфігурації кожна черга перестає бути самотнім вузлом і перетворюється на групу серверів, які зберігають одні й ті самі дані та дублюють роботу одне одного. Завдяки цьому вдається отримати значно кращі показники масштабованості, адже навантаження розподіляється між кількома репліками.

Крім того, така архітектура забезпечує набагато вищу доступність: якщо один вузол виходить з ладу, інші репліки миттєво підхоплюють роботу й продовжують обробку повідомлень без зупинки сервісу. На додачу, зростає і загальна надійність

системи — дані не зберігаються на одному сервері, а дублюються, що різко знижує ризик їх втрати.

У підсумку репліка-сети дозволяють отримати систему, яка поєднує у собі високу відмовостійкість, ефективність і стабільні гарантії доставки на рівні X ≈ 1 (або ж X ≥ 1), де ризики втрати повідомлень мінімізовані, а повтори контролюються.

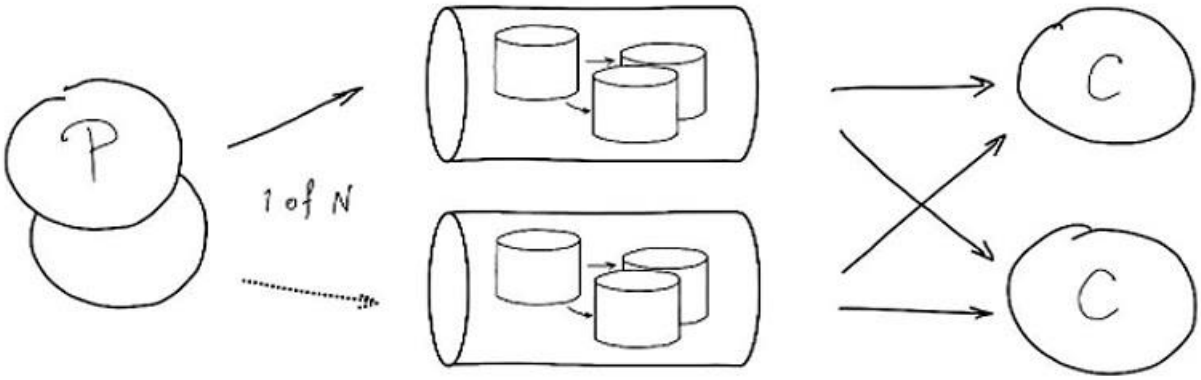


Рис. 3.6 Репліковані черги, 1 із N (підхід «як база даних»)

У подібній системі виконання кожної окремої задачі фактично наближається до значення 1, тобто кожне повідомлення обробляється рівно один раз і саме ця гарантія при будь-яких умовах не опускається нижче цього рівня. Іншими словами, навіть за наявності збоїв або переключення ролей між вузлами, система прагне забезпечити стабільну та передбачувану доставку.

Додатково замість класичної реплікації можна застосувати підхід, який використовується в кворумних базах даних. Принцип роботи залишається аналогічним: є лідер і кілька реплік, але механізм прийняття рішення ґрунтується не на одному головному вузлі, а на кворумі — мінімальній кількості вузлів, які мають підтвердити операцію.

Кворумний запис забезпечує значно вищу надійність. Якщо продюсер надіслав повідомлення і більшість вузлів підтвердили його приймання, то вважається, що дані збережені гарантовано і не будуть втрачені навіть у разі виходу з ладу окремих реплік. Саме тому кворумні системи дають дуже сильні гарантії виду X → 1 (X ≥ 1), коли виконання задачі прагне до ідеально надійної доставки.

Однак така схема має свою ціну: зниження пропускної здатності. Оскільки кожен запис повинен пройти підтвердження від більшості реплік, обробка

повідомлень уповільнюється. Це природний компроміс — чим вищі гарантії збереження та захисту від втрат, тим більше часу система витрачає на узгодження між вузлами.

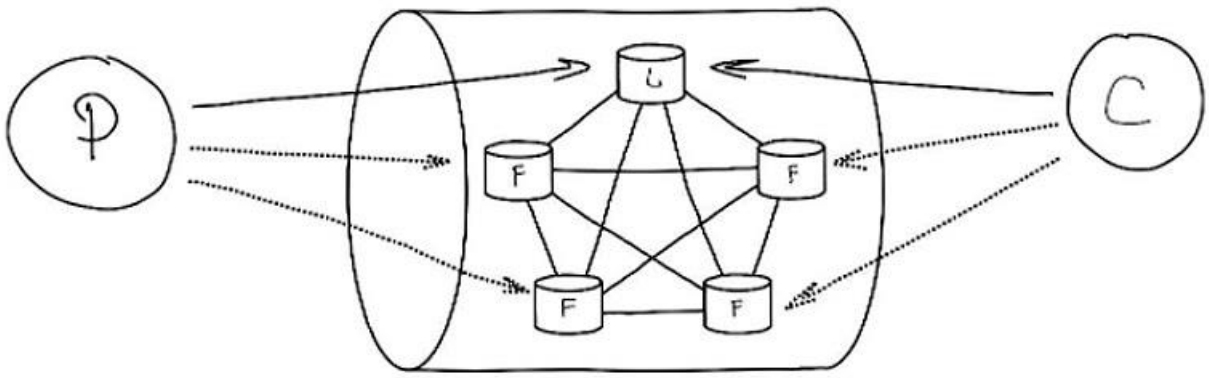


Рис. 3.7 Підхід баз даних: кворум

У певних випадках до кворумного брокера намагаються додати ще одного додаткового консюмера, сподіваючись, що в кластерній архітектурі це дозволить підвищити продуктивність або паралелізм обробки. Зовні це може здаватися логічним кроком: якщо є кілька вузлів, здається природним дозволити кільком консюмерам читати з реплік. Однак на практиці така схема не працює, щойно система потрапляє в стан мережевої сегментації.

Під час partitioning кластер ділиться на частини, і гарантії роботи зберігаються тільки в тій секції, де присутній кворум. Формально система має високу надійність та гарантії рівня $X \approx 1$ ($X \geq 1$) — тобто дані з великою ймовірністю не буде втрачено, адже репліки зберігають копії повідомлень і підтверджують їхній запис. Проте з точки зору продуктивності та масштабованості виникають суттєві обмеження.

По-перше, усі повідомлення обробляє лише один лідер, незалежно від того, скільки реплік у кластері та скільки консюмерів під’єднані. Репліки лише зберігають копії даних та підтверджують їхню валідність, але не виконують саму операцію обробки. Тому горизонтального масштабування тут немає: збільшення кількості вузлів не пришвидшує процесинг повідомлень.

По-друге, доступність системи обмежена кворумом. Якщо частина вузлів опиняється в окремій мережевій секції, де кворуму недостатньо, то ця частина кластера:

- не може приймати нові повідомлення,
- не може підтверджувати записи,
- не може гарантувати цілісність даних.

У результаті, під час мережевої сегментації, система втрачає доступність у тій частині, де кількість вузлів не досягає кворуму, і продюсери просто фізично не можуть надсилати туди події. Це робить роботу нерівномірною: одна частина кластера залишається працездатною, а інша фактично "завмирає".

Таким чином, хоч кворумний брокер і забезпечує високі гарантії збереження даних, його масштабованість залишається низькою, а доступність у ситуаціях partitioning істотно просідає. Система отримує сильну надійність, але ціною зниження гнучкості й можливості працювати при часткових збоях мережі.

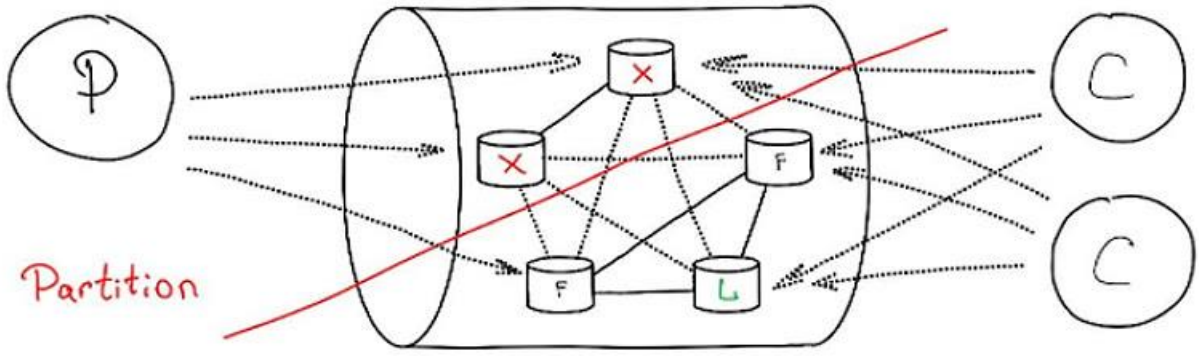


Рис. 3.8 Відмови у ситуаціях partitioning

Якщо замінити звичайну модель реплікації бази даних на кворумну реплікацію і застосувати цей підхід до черг, система отримує значно вищий рівень стійкості й передбачуваності. Завдяки узгодженій роботі кількох вузлів, що беруть участь у кворумі, ми можемо досягти одразу кількох важливих властивостей: горизонтального масштабування, високої доступності, стійкої надійності та гарантій доставки на рівні $X \approx 1$ ($X \geq 1$), тобто виконання й збереження повідомлень наближається до гарантованого, навіть у випадках відмов окремих компонентів.

У такій архітектурі операції запису та приймання повідомлень не покладаються на один сервер, а підтверджуються більшістю вузлів кворуму. Це означає, що система може продовжувати роботу навіть тоді, коли один або кілька вузлів виходять з ладу — інші члени кворуму забезпечують цілісність даних та приймають на себе обов’язки недоступних вузлів. Таким чином черга зберігає працездатність у ширшому діапазоні аварійних ситуацій, ніж у класичних конфігураціях із копіюванням або простими replica-set.

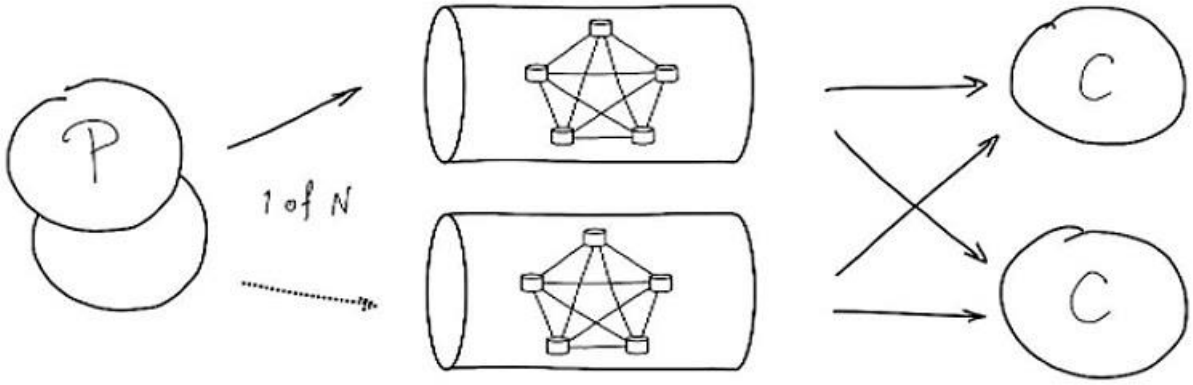


Рис. 3.9 Кворумні черги, 1 з N

Якщо у системі використовується більше ніж один кворум, виникає необхідність правильно та продумано розподілити їх між доступними ресурсами. Важливо забезпечити таке розміщення вузлів, щоб кворум одного кластеру був повністю зосереджений в одній зоні доступності, а кворум іншого — у іншій зоні.

Такий підхід дає можливість мінімізувати ризики одночасного виходу з ладу кількох вузлів через локальний збій інфраструктури. Завдяки цьому навіть у випадку серйозної аварії або відмови цілої availability-зони інша частина системи зможе продовжувати працювати, зберігаючи як доступність, так і цілісність даних.

Правильний вибір розташування кворумів дозволяє побудувати більш стійку до збоїв інфраструктуру, де кожен кластер залишається незалежним, а вся система — збалансованою, масштабованою й захищеною від локальних відмов.

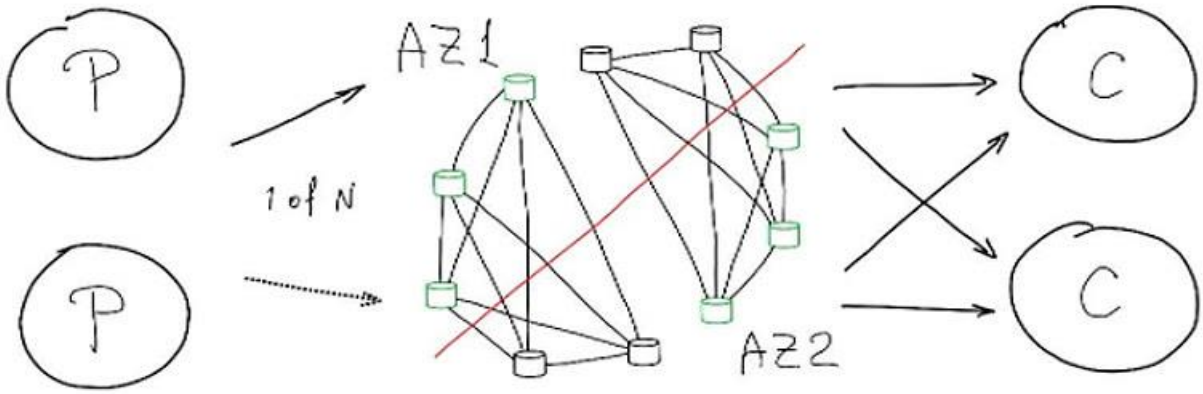


Рис. 3.10 Кворумні черги, 1 з N

3.2 Додаткові ресурси

Під час введення нової системи чергування у промислову експлуатацію дуже важливо одразу розуміти, як саме її слід моніторити та контролювати. Черга, незалежно від налаштувань і обраної технології, завжди має природні межі за обсягом і пропускною здатністю. Її нормальний, оптимальний робочий стан — це коли вона максимально порожня, тобто повідомлення надходять і одразу обробляються без накопичення. Якщо ж розмір черги починає зростати, це майже завжди означає, що у системі виникає певний збій або просідання в продуктивності. Тому першим і найважливішим індикатором стану системи є саме обсяг черги, який необхідно постійно моніторити.

Коли ж черга все-таки починає збільшуватися, корисним стає другий ключовий показник — часові метрики обробки. Їх можна вимірювати у двох площинах: повний життєвий цикл повідомлення (QoS) — від моменту його надсилання до завершення обробки, та окремо — час роботи консюмера над уже взятою задачею. Ці заміри дозволяють визначити, на якому етапі виникає затримка: продюсер не встигає, консюмер перевантажений або сама черга погано справляється з навантаженням.

Проте іноді розміру черги та часів обробки недостатньо для оперативної діагностики. За високої інтенсивності — наприклад, коли система обробляє понад 10 тисяч повідомлень за секунду — ці метрики практично «миттєві» і не

відображають справжньої картини. У таких випадках важливо контролювати кількість повторів, відмов та втрат повідомлень. Повтори сигналізують, що консюмери не справляються з навантаженням або виникають помилки при обробці; втрати можуть свідчити про проблеми з брокером, диском або мережевою інфраструктурою. Моніторинг самого потоку подій — тобто того, які повідомлення надходять, скільки їх повертається та що саме відкидається системою — дає змогу зрозуміти загальний стан системи значно глибше.

Якщо система дозволяє логувати усі повідомлення в момент їх додавання в чергу, це варто робити. Так, логів буде багато, але у випадку аварії вони стануть єдиним надійним джерелом даних, що дозволяє відновити втрачені події. Набагато краще мати надлишок журналів, ніж одного дня виявити, що критичні повідомлення безслідно зникли.

Під час експлуатації будь-якої черги необхідно правильно налаштовувати політики поведінки при збоях. У деяких випадках правильним рішенням буде тимчасово припинити прийом нових повідомлень, аби не погіршувати ситуацію. Якщо такої можливості немає, можна застосовувати інший підхід — видаляти неактуальні, застарілі повідомлення, які вже втратили цінність, і водночас продовжувати приймати нові. Це дозволяє зменшити накопичення непотрібного навантаження.

У практиці часто трапляються ситуації, коли частина повідомлень у черзі ще актуальна, а частина — давно прострочена. У такому разі доцільно спершу обробити «живі» події, а вже потім повертатися до тих, термін дії яких уже минув. Це дозволяє оптимізувати використання ресурсів і зменшити затримки у критичних потоках.

Не менш важливо планувати відмови наперед. Будь-яка система, якою б надійною вона не здавалася, одного дня неодмінно впаде. І саме від того, як добре продумана стратегія відновлення, залежить, наскільки швидко й безболісно її вдасться повернути в робочий стан. Розробник або команда експлуатації повинні чітко знати, як саме система падає, які процеси при цьому відбуваються і як

виглядає її правильний запуск після збою. Якщо цього розуміння немає, відновлення може зайняти в рази більше часу, ніж допускає ваш SLA.

Після аналізу особливостей роботи різних систем чергування варто підсумувати, який інструмент доцільно обирати залежно від конкретних потреб.

Якщо ваш сервіс толерує незначні втрати — наприклад, допустимо загубити 1–2 повідомлення на мільйон — і водночас потрібні висока пропускна здатність та масштабованість, тоді хорошим вибором буде NATS. Це просте, сучасне рішення, яке активно розвивається і має можливість переходу до більш стійких моделей зберігання. Альтернативою може бути NSQ, хоч його характеристики дещо слабші, або ж ZeroMQ, який повністю позбавлений персистентності та підійде для легких, швидкоплинних задач.

Якщо ваша система працює в хмарі й вам необхідно швидко підключити чергу без складного налаштування, можна використовувати хмарні черги класу SQS від провайдерів на кшталт Amazon Web Services, Google Cloud чи Azure. Для знайомства з концепціями черг і побудови нескладних пайплайнів підійдуть базові брокери на кшталт RabbitMQ або знову ж таки NATS, але за їхньою конфігурацією та відмовостійкістю слід стежити особливо уважно.

Коли ж вам потрібен стримінг повідомлень, робота з великими потоками даних, реплікація у системи Big Data, суворий порядок або висока збереженість повідомлень, тоді найкращим вибором буде Apache Kafka. Вона спеціально створена для потокових сценаріїв і має усі необхідні механізми для повторного відтворення подій, зберігання логів та обробки великих обсягів даних. Дещо спрощеною, але також потужною альтернативою є JetStream у складі NATS або корпоративні рішення на кшталт Tarantool Enterprise, які теж мають можливості потокової обробки.

Для складних сценаріїв оркестрації чергування — коли потрібні відкладені задачі, розгалужені пайплайни, об’єднання та поділ потоків, складні графи обробки — підійдуть два варіанти. Для простіших конфігурацій вистачить RabbitMQ, але для складних конвеєрів, де потрібні нетипові моделі чергування, гнучкі механізми

маршрутизації та настроювані схеми пріоритетів, оптимальним рішенням стане Tarantool Queue, що дозволяє побудувати практично будь-яку логіку черги.

У результаті правильний вибір черги залежить від конкретної мети: чи це стримінг, чи комунікація між сервісами, чи складний пайплайн обробки, чи високонавантажена система реального часу. Важливо не просто встановити брокер, а розуміти, як він поводиться, де його межі, як він падає і як відновлюється. Саме це визначає, чи буде система стабільною в реальних умовах.

4 УПРАВЛІННЯ ХМАРНИМИ ЧЕРГАМИ

4.1 Постановка задачі

У розподілених системах одним із найпоширеніших способів організації взаємодії між окремими екземплярами застосунку є використання хмарних черг, зокрема Azure Queue Storage. Цей механізм часто виступає основним каналом комунікації між компонентами. Проте покладатися лише на нього — не завжди ефективно. У такій конфігурації виникає низка обмежень, які заважають повністю використати доступні обчислювальні ресурси й забезпечити мінімальний час реакції системи.

У цій інтерпретації розглянемо, як можна:

- зменшити затримку між моментом надсилання задачі та її фактичним початком опрацювання екземпляром сервісу;
- мінімізувати кількість транзакцій до Azure Queue, що є важливою перевагою з погляду вартості й навантаження;
- підвищити загальну ефективність роботи системи, зменшивши кількість непотрібних операцій і забезпечивши більш плавне реагування на зміни потоку подій.

Проблематика стандартного підходу Azure SDK

Типова архітектура, передбачена інструментами Azure, передбачає, що взаємодія між екземплярами побудована виключно через чергу. Кожен робочий вузол періодично опитує Azure Queue, щоб дізнатися, чи не з’явилися в ній нові повідомлення. Такий підхід здається простим і універсальним, але всередині нього приховані суттєві недоліки.

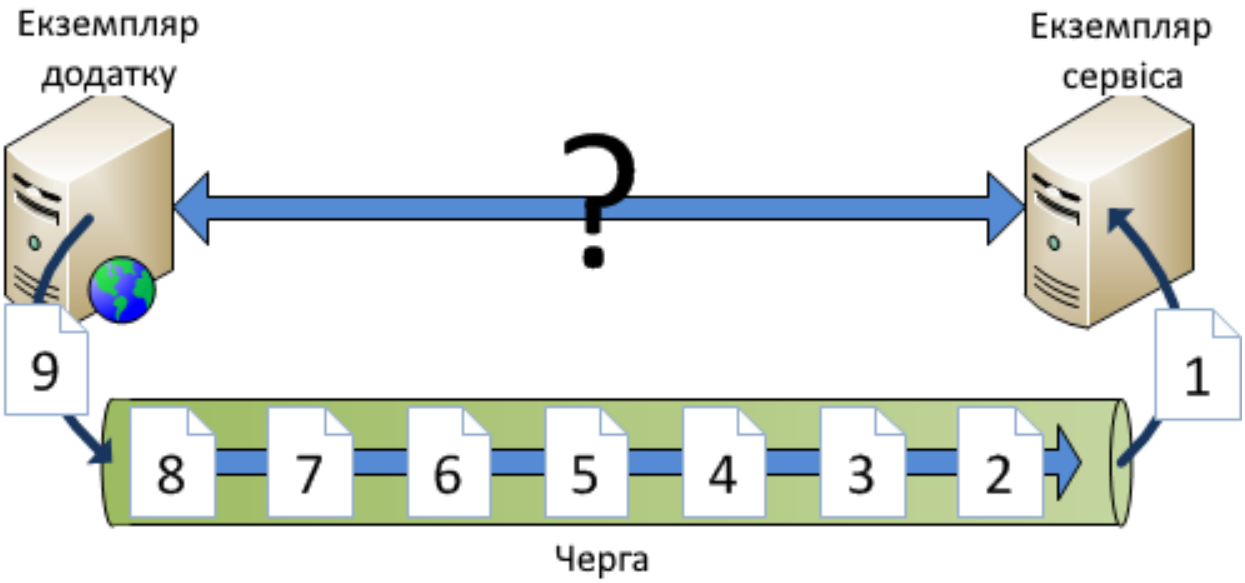


Рис. 4.1 Стандартні засоби SDK для Azure передбачають сценарій взаємодії виключно через чергу

Екземпляр сервісу не має уявлення, коли саме інший компонент покладе в чергу нове завдання. Це створює невизначеність: він змушений регулярно звертатися до черги, навіть якщо жодних нових задач немає. Таким чином система працює «всліпу», що і породжує ряд проблем.

Затримки в обробці завдань

Перша суттєва проблема — неминучі затримки між моментом надсилання завдання та його стартом обробки. Вони з’являються через те, що сервіс може побачити нову задачу лише під час чергової перевірки черги. Якщо ж повідомлення потрапляє до черги одразу після того, як екземпляр здійснив перевірку, воно просто чекатиме наступного інтервалу опитування.

У такому разі:

- якщо опитування відбувається часто, система швидко реагує, але це породжує величезну кількість транзакцій;
- якщо опитування відбувається рідко, витрати зменшуються, але затримка між фактом появи задачі та початком її виконання може бути значною.

У реальних сценаріях це може давати затримку у декілька секунд чи навіть десятки секунд, якщо інтервал опитування великий. Причому такі затримки

накопичуються й суттєво впливають на пропускну здатність, особливо коли задачі короткі й мають виконуватися майже миттєво.

Проблема «холостих» транзакцій

Другою ключовою проблемою стають порожні звернення до черги—так звані “idle transactions”. Навіть якщо жодних завдань немає, сервіс змушений робити запити, щоб переконатися в цьому. У великих системах із сотнями екземплярів такі транзакції можуть становити більшість навантаження на чергу.

Це призводить до:

- зайвих витрат на транзакції;
- збільшення загального навантаження на інфраструктуру;
- зменшення ефективності роботи сервісів;
- зростання часу відгуку при пікових навантаженнях.

Фактично частина обчислювальних ресурсів витрачається на операції, що не несуть жодної користі.

Дилема стандартного підходу

У результаті система стикається з класичною дилемою, заданою самим підходом Azure SDK:

- або високі витрати на постійні транзакції при швидкому опитуванні,
- або високі затримки при рідкісному опитуванні.

У деяких випадках швидкість реакції не є критичною, і система може працювати навіть із помітною затримкою. Якщо завдання надходять рівномірно або постійним потоком, «класичний» підхід справді може бути достатньо ефективним.

Проте в більшості реальних сценаріїв завдання надходять нерівномірно — з періодами пікових навантажень і тривалими «тихими» проміжками. Саме в таких умовах стандартний механізм починає працювати значно гірше: черга росте, екземпляри сервісів пропускають момент появи нових задач, а витрати на інфраструктуру збільшуються без реальної причини.

4.2 Підходи та рішення

Вирішити описані проблеми можна, застосувавши механізм обміну повідомленнями між екземплярами та ролями сервісу. Зупинимося детальніше на тому, як він працює та чому саме він здатен усунути недоліки традиційної взаємодії через Azure Queue.

Цей механізм забезпечує можливість надсилання службових повідомлень від одного екземпляра до іншого або ж одночасно до всіх екземплярів певної ролі. Фактично це внутрішній канал комунікації платформи, який дає змогу миттєво інформувати інші вузли про важливі події, зміну станів чи необхідність запуску певних процесів. За його допомогою можна активувати спеціальні обробники на будь-якому екземплярі, синхронізувати стан сервісів або координувати виконання довготривалих чи критичних задач.

У контексті нашої задачі цей підхід має вирішальну перевагу: він дозволяє розпочати обробку черги одразу після того, як у неї було додано нове повідомлення. Іншими словами, екземпляр, який додає завдання до черги, одразу надсилає сигнал іншим екземплярам сервісу. Цей сигнал повідомляє їх про появу нового завдання — і обробка запускається негайно, без необхідності чекати наступної циклічної перевірки.

Саме так вдається уникнути постійного опитування черги та, відповідно, прибрати «холості» транзакції, які раніше тільки збільшували навантаження на Azure Queue та спричиняли додаткові витрати. Екземпляри сервісу більше не зобов’язані періодично перевіряти, чи з’явилося нове повідомлення — їх активує сам факт надходження повідомлення в чергу.

У результаті досягається одразу кілька важливих ефектів:

- мінімізується затримка між відправленням завдання та початком його виконання,
- зникає потреба у великій кількості транзакцій до черги,
- суттєво підвищується ефективність використання обчислювальних ресурсів,
- система стає значно реактивнішою й чутливішою до подій.

Таким чином, впровадження механізму обміну повідомленнями між екземплярами дозволяє побудувати більш оптимальну, економну та швидкодіючу модель обробки завдань, яка не залежить від частоти опитування черги та працює у подійно-орієнтованому режимі.

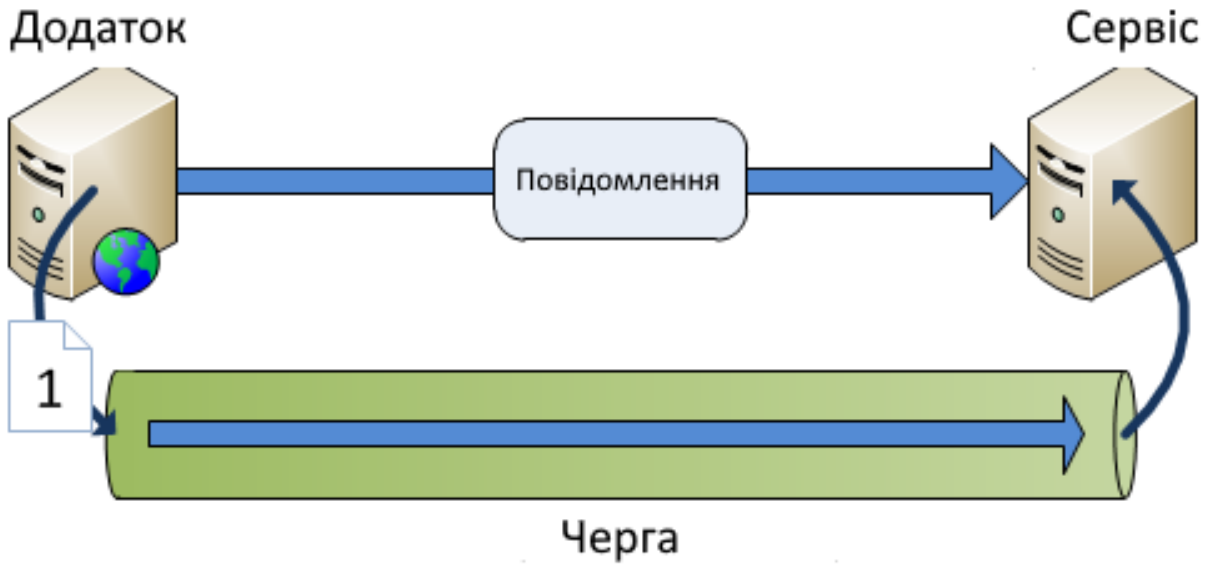


Рис. 4.2 Обробка черги відразу ж після додавання завдання до черги

Якщо розглянути цей механізм докладніше, ми побачимо таке:

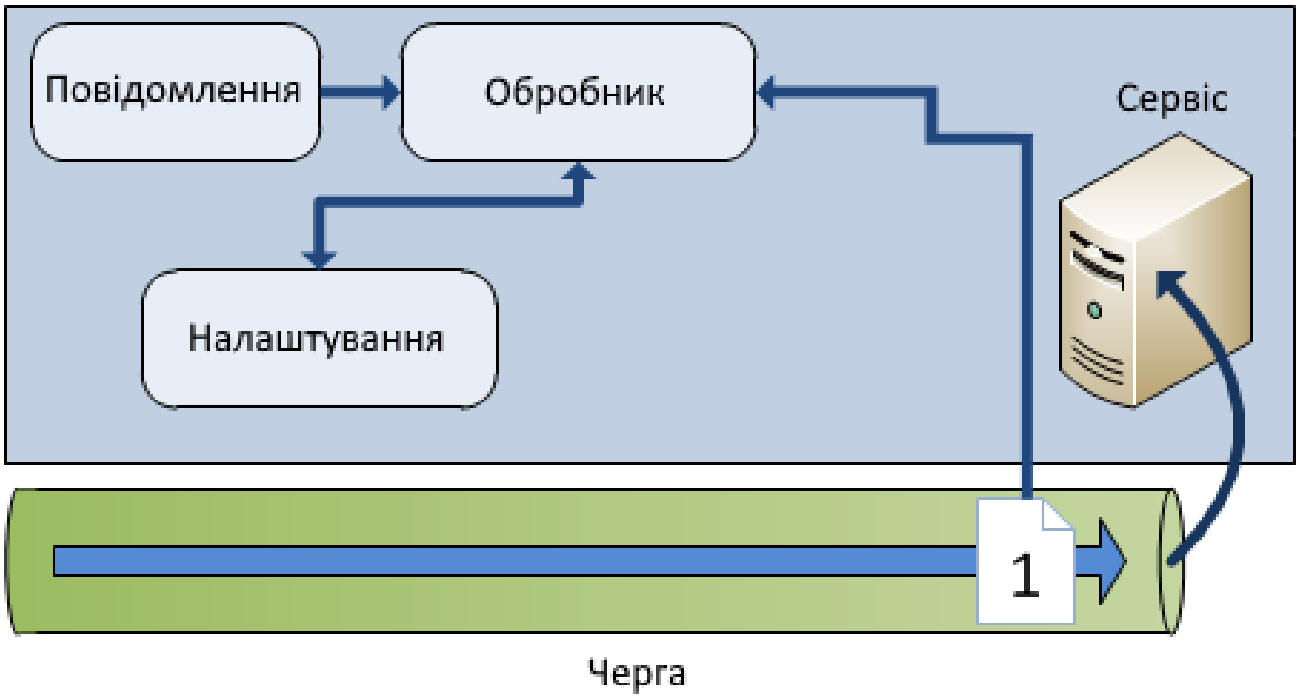


Рис. 4.3 Детальний процес виконання завдання

Екземпляр застосунку в середовищі Azure формує нове завдання, додає його до черги Azure Queue і паралельно надсилає службове повідомлення на відповідний

екземпляр сервісу. Це повідомлення виступає своєрідним тригером, який активує спеціальний обробник. Подальша логіка роботи залежить від того, які параметри та стан збережені в конфігурації обробника.

Якщо в налаштуваннях вже зафіксовано, що процедура обробки черги була запущена іншим екземпляром або раніше активованим процесом, тоді повторний старт не потрібен — система просто проігнорує запит і продовжить роботу у звичайному режимі. У протилежному випадку обробник ініціює процес обробки черги, одночасно фіксуючи у конфігурації, що обробка вже запущена, аби уникнути дублювання.

Після активації обробник починає послідовно зчитувати завдання з Azure Queue, виконувати їх і переходити до наступних, поки черга не буде повністю очищена. Такий підхід забезпечує безперервну роботу з реальними задачами та максимально ефективне використання наявних екземплярів сервісу.

Стан і параметри обробника можуть зберігатися у різних сховищах. Найпростіший, але найменш надійний варіант — це зберігання інформації в оперативній пам’яті екземпляра. Однак при рестарті або падінні такий стан буде втрачений. Значно надійнішим є використання стійкого сховища екземпляра або зовнішнього файлового сервісу, наприклад, Azure Blob Storage. Зберігання конфігурації в BLOB-контейнерах дозволяє забезпечити доступ до актуального стану незалежно від того, який саме екземпляр запускає обробник, а також гарантує відновлюваність інформації навіть після аварійного завершення роботи вузла.

Таким чином, поєднання Azure Queue з механізмом міжекземплярного сповіщення дозволяє будувати ефективні системи, де обробка запускатиметься автоматично та лише тоді, коли це дійсно необхідно, без зайвих запитів, дублювання та холостого навантаження на інфраструктуру.

У бібліотеці для роботи з чергами передбачено спеціальний клас Queue, який дає змогу створювати власні обробники черг та керувати їхньою логікою. Для підтримки цього механізму бібліотека також містить інтерфейс IWorkerQueueHandler, а разом із ним — набір розширень, що значно спрощують

взаємодію з чергами та роблять процес обробки повідомлень більш зручним і гнучким.

```
        public interface IWorkerQueueHandler
{
   bool HandleQueue(string Message);
}

public static class WorkerQueueHandlerExtensions
{
   private static string CleanUpQueueName(string DirtyQueueName)
   {
      return DirtyQueueName.Substring(0,
DirtyQueueName.IndexOf(",")).ToLowerInvariant().Replace(".", "-");
   }

   public static string GetQueueName(this IWorkerQueueHandler Handler)
   {
      return CleanUpQueueName(Handler.GetType().AssemblyQualifiedName);
   }

   public static string GetQueueName(Type HandlerType)
   {
      return CleanUpQueueName(HandlerType.AssemblyQualifiedName);
```

Інтерфейс є максимально простим і зручним — він містить лише **один метод**, у межах якого розробник реалізує всю необхідну логіку обробки повідомлень, що надходять до черги. Така структура значно спрощує створення власних обробників і дозволяє зосередитися виключно на бізнес-логіці, не переймаючись деталями інфраструктури.

Розширення, що супроводжує цей інтерфейс, виконує важливу допоміжну функцію — воно автоматично формує коректну назву черги на основі типу

обробника. Завдяки цьому програмісту не потрібно вручну контролювати унікальність імен для різних черг, турбуватися про конфлікти назв чи дотримання єдиних правил неймінгу. Це знижує ризик помилок і робить систему більш узгодженою.

У самому класі Queue реалізовано повний цикл роботи з чергами. Зокрема, він відповідає за:

- вибір та ідентифікацію потрібної черги для певного обробника;
- створення нової черги у хмарному сховищі Azure, якщо вона ще не існує;
- додавання нового завдання до черги;
- автоматичне надсилання повідомлення на відповідний екземпляр сервісу Azure з метою ініціації обробки.

Як і зазначалося раніше, тут використовується механізм повідомлень між екземплярами та ролями, що дає можливість одразу активувати процес обробки після появи нового завдання. Завдяки цьому система працює більш ефективно: немає потреби у періодичному опитуванні черги або виконанні "порожніх" перевірок, що зменшує накладні витрати й пришвидшує реакцію на нові повідомлення.

Такий підхід робить архітектуру значно більш передбачуваною та керованою, а роботу з чергами — інтуїтивною, гнучкою та надійною.

```
        public static CloudQueue GetQueue(string QueueName)
{
   CreateOnceQueue(QueueName);
   return queueStorage.GetQueueReference(QueueName);
}
public static bool AddToQueue<QueueHandlerType>(string Task) where
QueueHandlerType : IWorkerQueueHandler
{
   var Queue =
GetQueue(WorkerQueueHandlerExtensions.GetQueueName(typeof(QueueHandlerType
)));
```

```
    return AddToQueue<QueueHandlerType>(Queue, Task);
}
public static bool AddToQueue<QueueHandlerType>(CloudQueue Queue, string
Task)
{
    try
    {
        var Message = new CloudQueueMessage(Task);
        Queue.AddMessage(Message);
        Internal.RoleCommunicatior.WorkerRoleCommand(typeof(QueueHandlerType));
        return true;
    }
    catch { return false; }
```

В якості основного каналу взаємодії між різними екземплярами сервісів у хмарному середовищі зазвичай рекомендується використовувати Azure Queue — вбудовану чергу повідомлень платформи Azure. Це найпростіший і найпоширеніший інструмент комунікації між компонентами. Однак використання *лише* цього каналу далеко не завжди дозволяє повністю розкрити потенціал сервісів і ефективно розподілити навантаження між екземплярами.

Проблема полягає в тому, що стандартна схема SDK передбачає взаємодію виключно через чергу. Екземпляр сервісу не знає, коли саме застосунок поставить нове завдання в Azure Queue, тому він змушений періодично опитувати чергу, перевіряючи наявність нових записів. Такий механізм породжує два ключові недоліки:

1. Затримка між надсиланням завдання та початком його обробки

Екземпляр сервісу перевіряє чергу через певні інтервали часу. Якщо нове повідомлення потрапляє в чергу одразу *після* перевірки, то обробка розпочнеться тільки під час наступного циклу опитування. В результаті виникає затримка, яка може досягати декількох секунд або й більше — залежно від частоти перевірки.

Чим більший інтервал між перевірками — тим більша затримка. Чим менший інтервал — тим більше непотрібних звернень до Azure Queue.

2. «Порожні» транзакції та зайві витрати

Якщо перевіряти чергу надто часто, екземпляр надсилатиме величезну кількість запитів *навіть у ті моменти, коли черга порожня*. Це створює:

- значні накладні витрати,
- зайве навантаження на інфраструктуру,
- непотрібне споживання хмарних ресурсів,
- додаткові грошові витрати.

Таким чином, у стандартному підході доводиться балансувати між оперативністю обробки та вартістю транзакцій.

Вирішення: прямий обмін повідомленнями між екземплярами та ролями

Уникнути обох проблем можна завдяки механізму внутрішнього обміну повідомленнями між екземплярами та ролями Azure. Це окремий канал комунікації, який дозволяє одному екземпляру надсилати повідомлення іншому, або транслювати їх на всі екземпляри ролі.

Цей механізм має низку корисних властивостей:

- дозволяє миттєво запускати обробники на конкретних екземплярах,
- дає можливість підтримувати синхронний стан між вузлами,
- забезпечує події для негайної реакції,
- знімає необхідність постійного опитування черги.

У контексті роботи з Azure Queue це означає, що обробка може запускатися відразу після надсилання завдання в чергу, без будь-яких затримок і без жодного «порожнього» запиту.

Як працює механізм у розширеному варіанті

Алгоритм виглядає так:

1. Екземпляр застосунку додає в Azure Queue нове завдання.
2. Одразу після цього він надсилає спеціальне службове повідомлення іншому екземпляру сервісу.
3. Екземпляр-сервіс отримує це повідомлення і викликає *обробник черги*.

4. Обробник перевіряє, чи вже йде обробка черги.
    - Якщо обробка запущена — другий запуск не ініціюється.
    - Якщо ні — він розпочинає процес, зберігає стан «обробка активна» і по черзі вибирає всі завдання, доки черга не спустіє.
5. Стани обробки зберігаються або в пам’яті екземпляра, або у зовнішньому сховищі — наприклад Azure Blob Storage, що є надійнішим варіантом.

Такий підхід забезпечує:

- миттєву реакцію на появу нових повідомлень;
- відсутність постійних перевірок черги;
- повну ліквідацію зайвих транзакцій;
- ефективніший розподіл навантаження між екземплярами.

Оцінка економії та ефективності

Переваги такого підходу легко оцінити на практиці.

Припустімо, що більшість завдань надходить лише у вечірні години — наприклад, з 20:00 до 23:00.
Усі інші 21 година доби черга порожня, але стандартна система продовжує перевіряти її щосекунди.

Це означає:

- 21 × 3600 = 75 600 порожніх транзакцій на добу,
- що становить приблизно $0.075 на день або $2.25 на місяць,
- а при 100 чергах це вже $225 додаткових витрат щомісяця — і все це на порожніх запитах.

Одночасно інтервальний підхід створює проблему затримок у критичних задачах.

Якщо завдання треба виконувати негайно — наприклад, фінансові транзакції або розподіл прибутку — затримка навіть у 1 секунду може бути неприйнятною. При великому потоці вона штучно обмежує можливості системи до 60 оброблених подій на хвилину, що швидко призведе до роздування черги.

У підсумку розробнику доводиться балансувати між:

- високою вартістю порожніх транзакцій,

- повільною реакцією системи на пікові навантаження.

Запропонований підхід повністю усуває цю дилему:

- більше не потрібно опитувати чергу,
- немає жодних холостих транзакцій,
- обробка запускається негайно,
- продуктивність і швидкість реакції зростають у рази.

ВИСНОВКИ

У ході дослідження методів обробки черг в інфокомунікаційних мережах було встановлено, що сучасні черги повідомлень відіграють ключову роль у побудові надійних, масштабованих та ефективних розподілених систем. Незважаючи на те, що концепція черги здається простою — повідомлення додається продюсером і згодом обробляється консюмером — реальна робота таких систем значно складніша та включає цілий спектр архітектурних, алгоритмічних і мережевих обмежень.

У роботі розглянуто основні моделі впорядкування задач, такі як FIFO, LIFO та Best Effort, а також показано, що навіть базові відмінності між цими підходами визначають характеристики затримки, пропускної здатності та надійності системи. Окрему увагу приділено проблемам, що виникають у складних інфраструктурах: пріоритизації, дублюванню, потребі у dead-letter чергах, роботі з TTL та тригерами повернення повідомлень.

Одним із ключових результатів роботи є систематизація чотирирівневої структури проблем, властивих системам чергування. Перший рівень пов’язаний із внутрішніми алгоритмами упорядкування й повернення задач; другий – зі системними ефектами, такими як голодування задач, обмеження пропускної здатності та ресурсів; третій – із фундаментальною невизначеністю мережі та неможливістю реалізувати повний «exactly once» без додаткових механізмів; четвертий – з відмовами апаратного та інфраструктурного характеру, включаючи split brain і втрату цілих вузлів або дата-центрів. Виявлено, що реальна ефективність черг багато в чому залежить не лише від вибраного брокера, а від здатності інженерів коректно проєктувати архітектуру з урахуванням цих ризиків.

У роботі також проаналізовано моделі підвищення надійності, включаючи системи з одним брокером, multi-instance підходи, реплікацію, використання quorum-механізмів та побудову кластерів із розподілом вузлів по різних зонах доступності. Показано, що підвищення доступності неминуче потребує жертв у пропускній здатності, а гарантії доставки $X \leq 1$, $X \geq 1$ або $X \approx 1$ залежать від

обраного підходу та його архітектури. Ефективне поєднання реплікації, кворумного запису та горизонтального масштабування забезпечує найвищий рівень надійності без втрати продуктивності.

Окремо підкреслено важливість моніторингу систем чергування. Розмір черги, час обробки, кількість повторів і втрат були визначені як критично важливі метрики, що дозволяють попереджувати збої задовго до того, як вони вплинуть на продуктивність. Наголошено на необхідності логування, правильного налаштування політик відмови, а також готовності системи до падіння та швидкого відновлення. У роботі доведено, що черги не можуть бути абсолютно стійкими, тому грамотна експлуатація є такою ж важливою, як і правильний вибір брокера чи алгоритму.

У підсумку можна стверджувати, що саме глибоке розуміння природи черг, їхніх обмежень, моделей масштабування, поведінки при збоях та принципів моніторингу дозволяє створювати інфокомунікаційні системи, здатні працювати стабільно, ефективно та передбачувано навіть у складних та динамічних умовах сучасних мережевих інфраструктур.

ПЕРЕЛІК ПОСИЛАНЬ

1. Giambene G. Queuing Theory and Telecommunications: Networks and Applications. 2nd ed. – Springer, 2014.
2. Giambene G. Queuing Theory and Telecommunications: Networks and Applications. – Springer, 2021 (revised edition).
3. Bolch G., Greiner S., de Meer H., Trivedi K. S. Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. 2nd ed. – Wiley, 2006.
4. Trivedi K. S. Probability and Statistics with Reliability, Queueing, and Computer Science Applications. 2nd ed. – Wiley, 2001.
5. Sztrik J. Basic Queueing Theory. – University of Debrecen, 2010.
6. Arunachalam V. Queueing Modelling Fundamentals: With Applications in Communication Networks. 2nd ed. – Wiley, 2008.
7. Seo J. B. Introduction to Queueing Theory. Lecture notes, IIT Delhi, 2017.
8. Tanenbaum A. S., van Steen M. Distributed Systems: Principles and Paradigms. 2nd ed. – Prentice Hall, 2007.
9. Coulouris G., Dollimore J., Kindberg T., Blair G. Distributed Systems: Concepts and Design. 5th ed. – Addison-Wesley, 2011.
10. Amazon Web Services. Queue-based System Architecture: Build Scalable, Fault-tolerant Distributed Systems. – Whitepaper / e-book.
11. Birman K. Reliable Distributed Systems: Technologies, Web Services, and Applications. – Springer, 2005.
12. IBM. What is a Message Queue? – IBM Think documentation.
13. More S. Message Queues in Distributed Systems. – Medium, 29.09.2025.
14. The Simple Guide to Message Queues. – Dev.to, 2021.
15. Message Queues. – Abilian Labs: Tech / Architecture & Software Design.
16. AppMaster.io. Microservices Middleware. – Глосарій, 2023.
17. Antoniak Ł. Building Modern Distributed Systems with Java: Learn how to design large-scale modern distributed systems with NoSQL databases, messaging queues and

cluster coordination using Java. – Packt, 2022. Packt

18. Message Queuing for Developers. – Microsoft Press / ACM, 1998. ACM Digital Library
19. Bouchenak S. Message Queuing Systems. – Encyclopedia of Database Systems, Springer, 2009.
20. Introduction to Messaging Queues in Spring Boot Microservices. – GeeksforGeeks, 2025.
21. Overview on Messaging System in Microservice Architecture. – Innovinds Blog, 2023.
22. Кривуца В. Г. Основи інфокомунікацій. – Навчальний посібник. – Київ, 2011. Цифрова бібліотека НАПН України
23. Воробієнко П. П. Телекомунікаційні та інформаційні мережі. – Навчальний посібник. – Київ: Саміт-Книга, 2010.
24. Основи інфокомунікаційних технологій. – Навчальний посібник / Сумський державний університет. – Суми, 2013.
25. Тарбаєв О. В. Проєктування інфокомунікаційних мереж. – Навчальний посібник. – Суми, 2019.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ

КВАЛІФІКАЦІЙНА РОБОТА

МЕТОДИ ОБРОБКИ ЧЕРГ В ІНФОКОМУНІКАЦІЙНИХ МЕРЕЖАХ

Виконав:
студент групи ТСДМ-62
Михайло НЕЩЕРЕТ

МЕТА РОБОТИ ТА АКТУАЛЬНІСТЬ

- Мета роботи – проаналізувати існуючі підходи, протоколи та архітектури черг повідомлень, виявити їхні обмеження й проблеми роботи в реальних інфокомунікаційних системах та сформувати рекомендації щодо підвищення ефективності, продуктивності й надійності механізмів чергування.
- Об’єкт дослідження – процеси обміну повідомленнями та організація черг у сучасних інфокомунікаційних і розподілених системах.
- Предмет дослідження – моделі, алгоритми та архітектури черг повідомлень, а також їх вплив на продуктивність, надійність, масштабованість і стійкість розподілених систем під різними типами навантажень.

Актуальність.

- Стрімке зростання обсягів трафіку, розвиток мікросервісних архітектур та хмарних інфраструктур ставлять високі вимоги до надійності й продуктивності інфокомунікаційних систем. Черги повідомлень відіграють ключову роль у забезпеченні асинхронності, балансування навантаження та стійкості до збоїв. Неправильне проєктування черг може спричинити перевантаження, дублювання даних, затримки та деградацію сервісів. Актуальність дослідження полягає в необхідності аналізу моделей чергування, виявлення їх обмежень та розроблення рекомендацій для підвищення ефективності і надійності розподілених систем.

ПОНЯТТЯ ЧЕРГ ТА ЇХ НЕОБХІДНІСТЬ

- Черги повідомлень потрібні для стабільної обробки запитів у розподілених системах. Вони працюють як буфер між вхідним потоком і серверами: слабкі вузли беруть менше задач, потужні — більше, недоступні — нічого. Це дозволяє уникати перевантажень, втрат даних і збоїв. Черги згладжують піки навантаження, забезпечують асинхронність, відмовостійкість та масштабованість. Саме тому їх застосовують у мікросервісах, телекомі, фінансових системах, потоковій обробці й IPC. Вони формують основу надійної та передбачуваної архітектури.

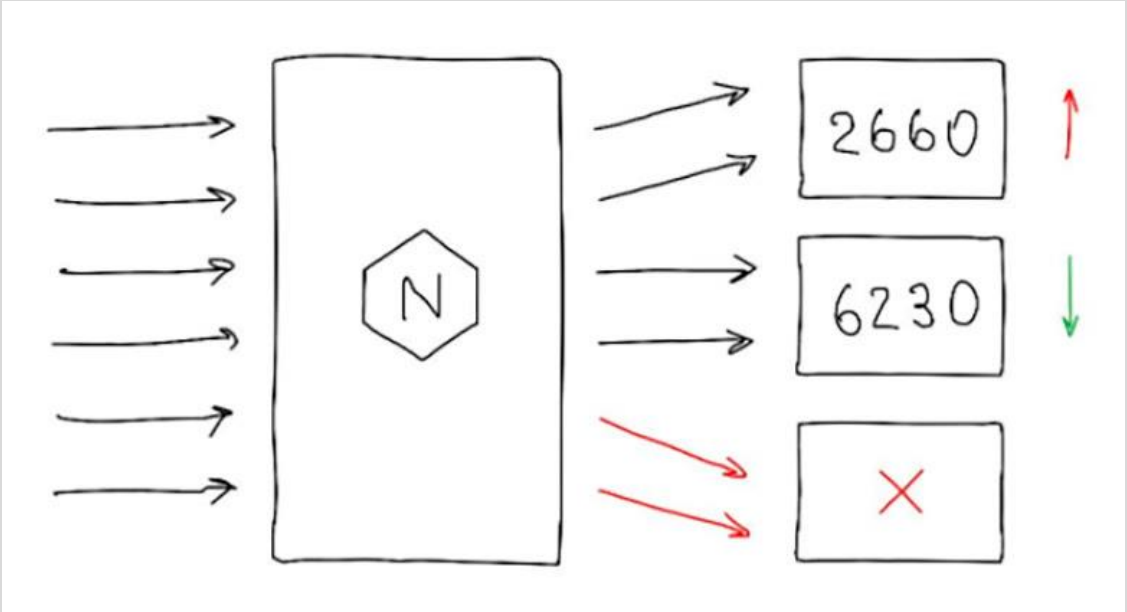


Схема класичного балансувальника

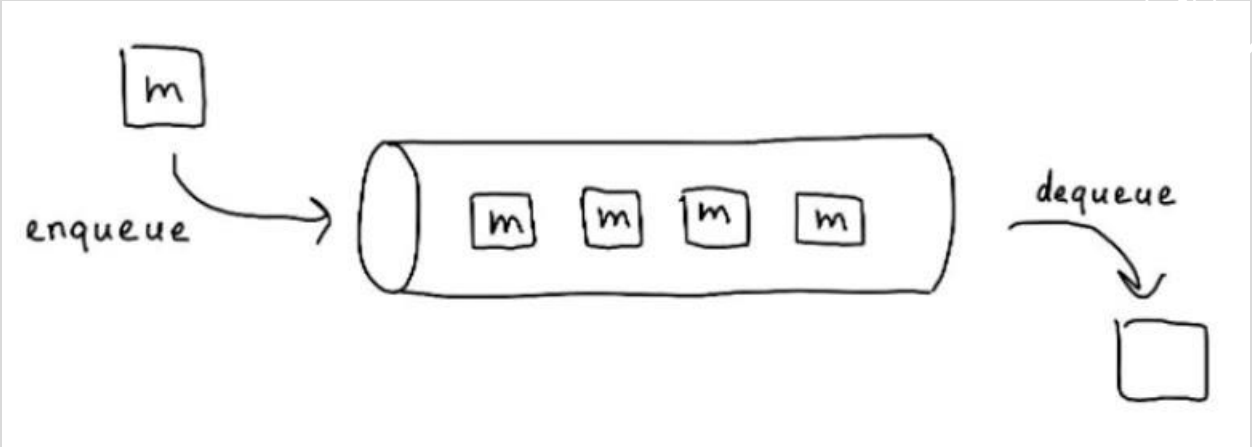


Схема передавання повідомлення процесом, сервісом або модулем

КЛАСИФІКАЦІЯ ПІДХОДІВ

- **Put/Take (1→1)**

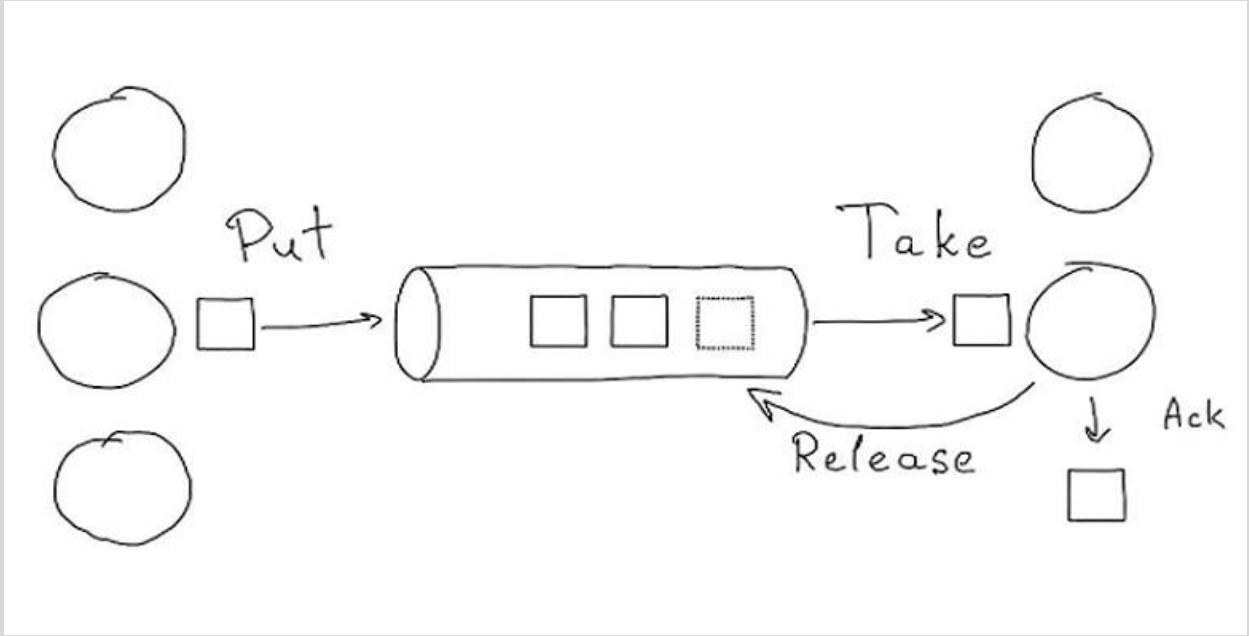
Продюсер кладе повідомлення в чергу, консюмер забирає й обробляє. Успіх — повідомлення видаляється, помилка — повертається назад. Модель проста, надійна, масштабується додаванням консюмерів.

- **Publish/Subscribe (1→багато)**

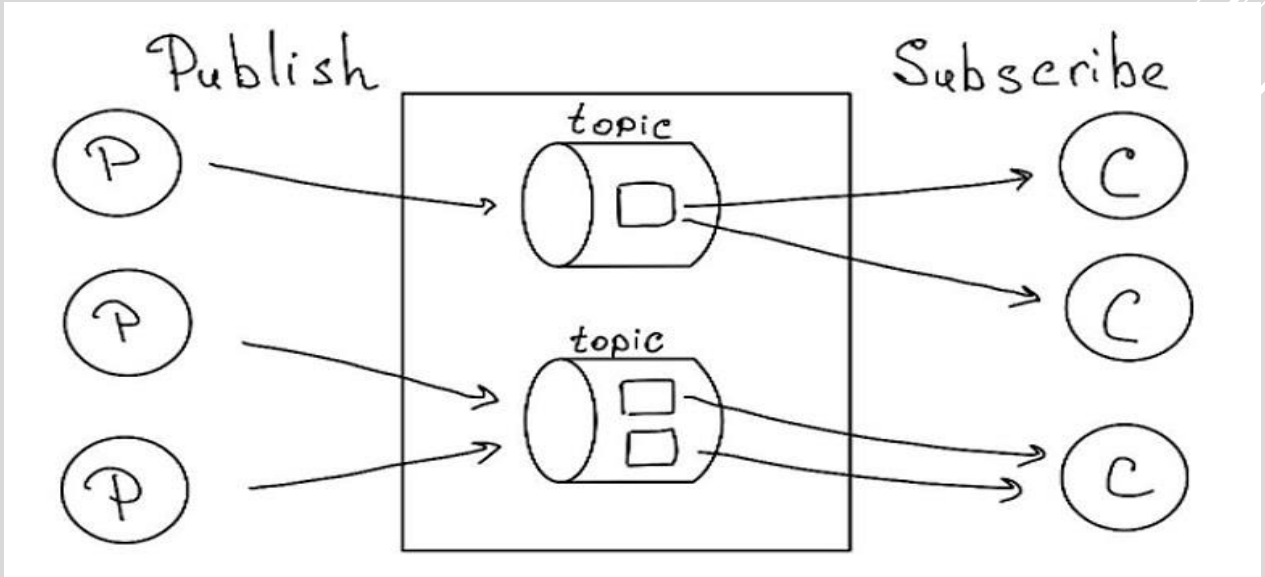
Publisher надсилає подію в топік, і всі підписники отримують її паралельно. Одне повідомлення запускає кілька незалежних процесів. Основний механізм подійної архітектури.

- **Request/Response (1⇆1)**

Клієнт надсилає запит у чергу, консюмер обробляє й повертає відповідь. Підтримує кореляцію, асинхронність і стійкість до збоїв. Використовується для надійного RPC і довгих операцій.



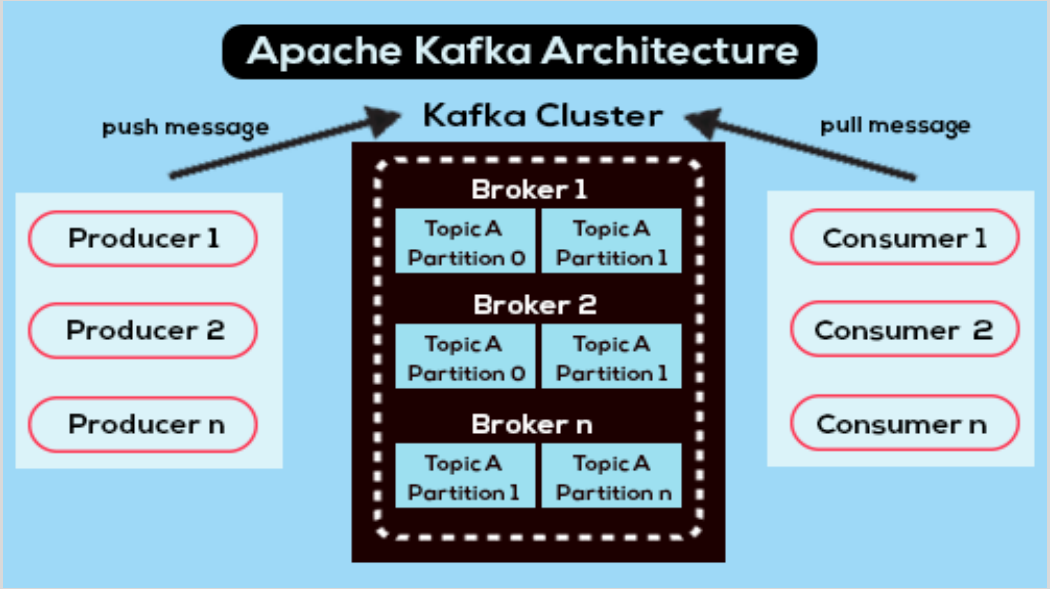
Підхід Put/Take



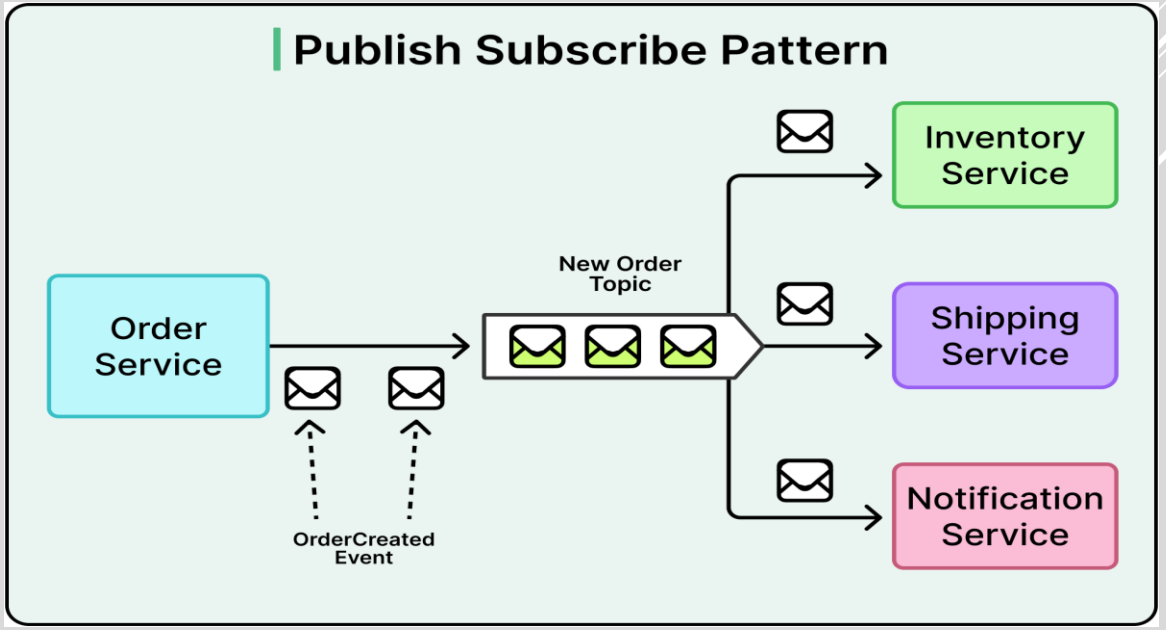
Підхід Pub/Sub

ВИКОРИСТАННЯ ПРОТОКОЛІВ ТА ПЛАТФОРМ

- Черги повідомлень працюють на різних протоколах, але всі вони вирішують одну задачу — забезпечити узгоджену й надійну передачу даних між компонентами системи. AMQP дає складну маршрутизацію та гарантії доставки, MQTT підходить для IoT, STOMP — для простих клієнтів, Pub/Sub — для широкої розсилки подій.
- Черги можуть бути хмарними (SQS), локальними (на базах даних) або високошвидкісними без збереження. Існують потокові журнальні системи, класичні брокери, швидкісні комунікаційні платформи та кастомні рішення з транзакційністю.
- Протокол визначає порядок, гарантії доставки, масштабування та затримки.



Архітектура потокової платформи на основі шардованого журналу подій



Модель Publish/Subscribe для розподілу подій між мікросервісами

ПЕРШИЙ ТА ДРУГИЙ РІВЕНЬ ПРОБЛЕМ

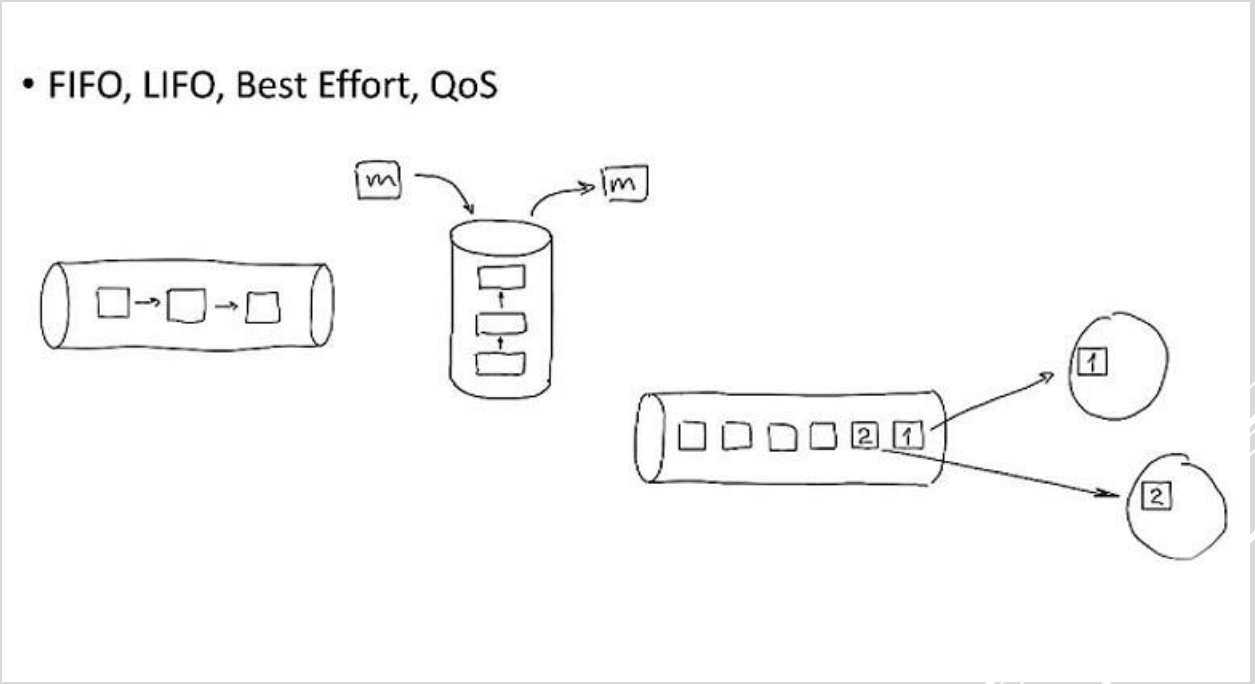
- Перший рівень проблем

Черги мають вбудовані архітектурні обмеження, навіть коли система працює стабільно. FIFO забезпечує строгий порядок, але підходить не для всіх сценаріїв; LIFO у розподілених системах майже не використовується. Найпоширеніший варіант — Best Effort, де повідомлення повертаються після помилки. Багато підчерг ускладнюють роботу брокера й адміністрування. TTL, TTR і putback можуть спричиняти дублювання та нескінченні повтори.

- Другий рівень проблем

Під навантаженням проявляються системні ефекти: голодування низьких пріоритетів, обмеження пропускної здатності, повільний доступ до дисків, неможливість горизонтального масштабування. Ресурси вичерпуються — пам’ять, диск, CPU — що веде до затримок, переповнення, втрат повідомлень і каскадних збоїв.

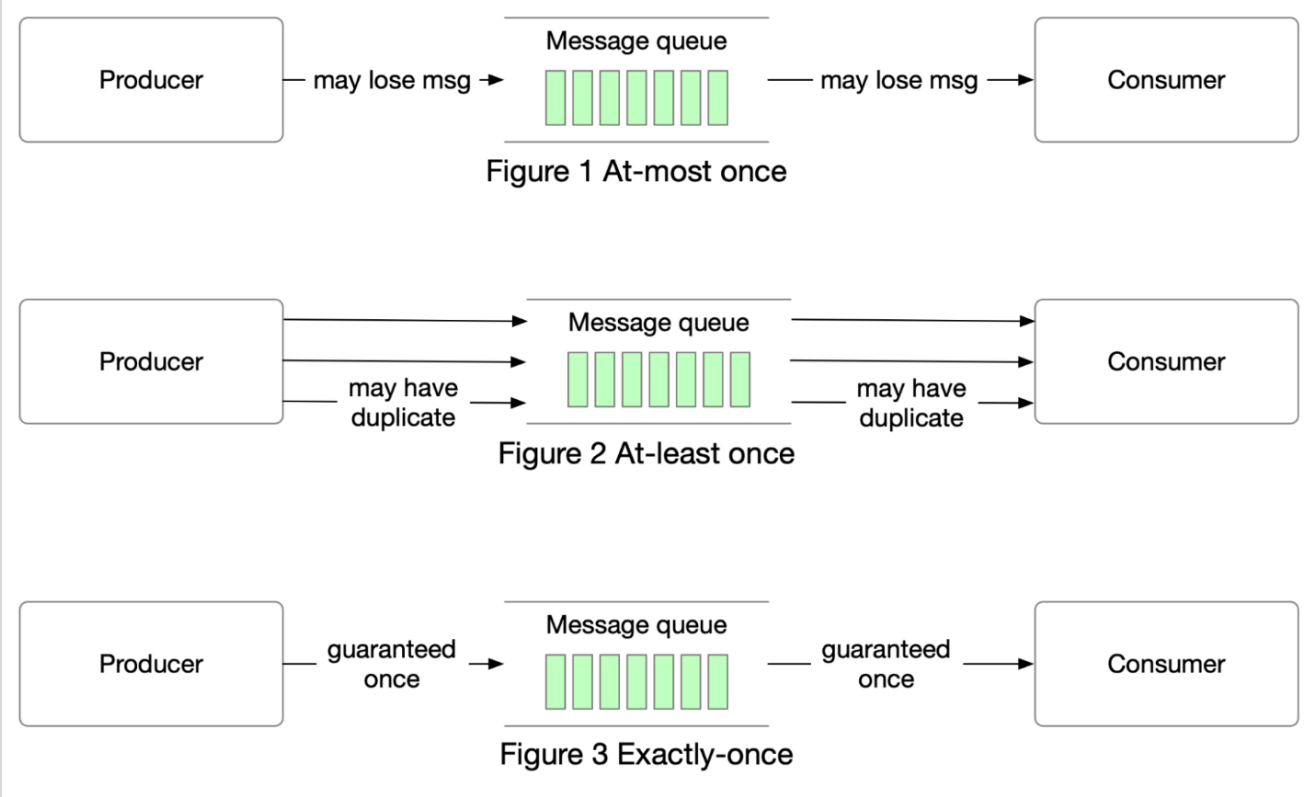
Другий рівень — це вже поведінка всієї архітектури, а не окремих задач.



Алгоритм LIFO

ТРЕТІЙ РІВЕНЬ ПРОБЛЕМ

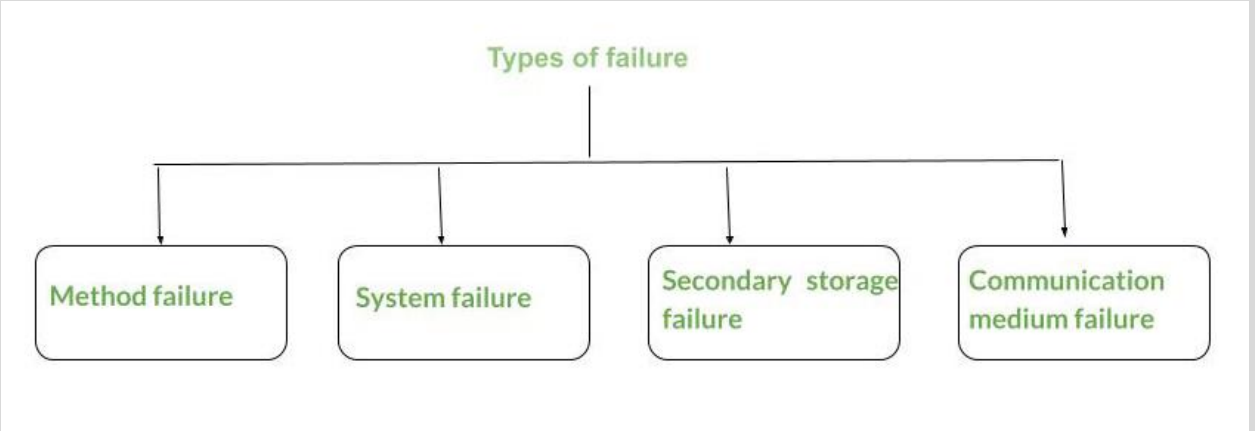
- Перші черги працювали в ізольованих системах, де не було втрат, затримок чи невизначеності — усе виконувалося детерміновано. У розподілених середовищах це руйнується: мережа не гарантує доставки, порядок може порушуватися, повідомлення дублюються або губляться. Тому реально можливі лише моделі *at most once* або *at least once*, тоді як *exactly once* зазвичай симулюється через дедуплікацію або ручний контроль (dead-letter та оператор).
- Додаткову невизначеність вносять диски: затримки, пропускна здатність, зношування та переповнення буферів. Усе це робить черги вразливими до втрат, дублювання й затримок.
- Тож сучасні черги повинні враховувати ризики: втрати повідомлень, повтори, dead-letter, механізми відновлення та апаратні обмеження.



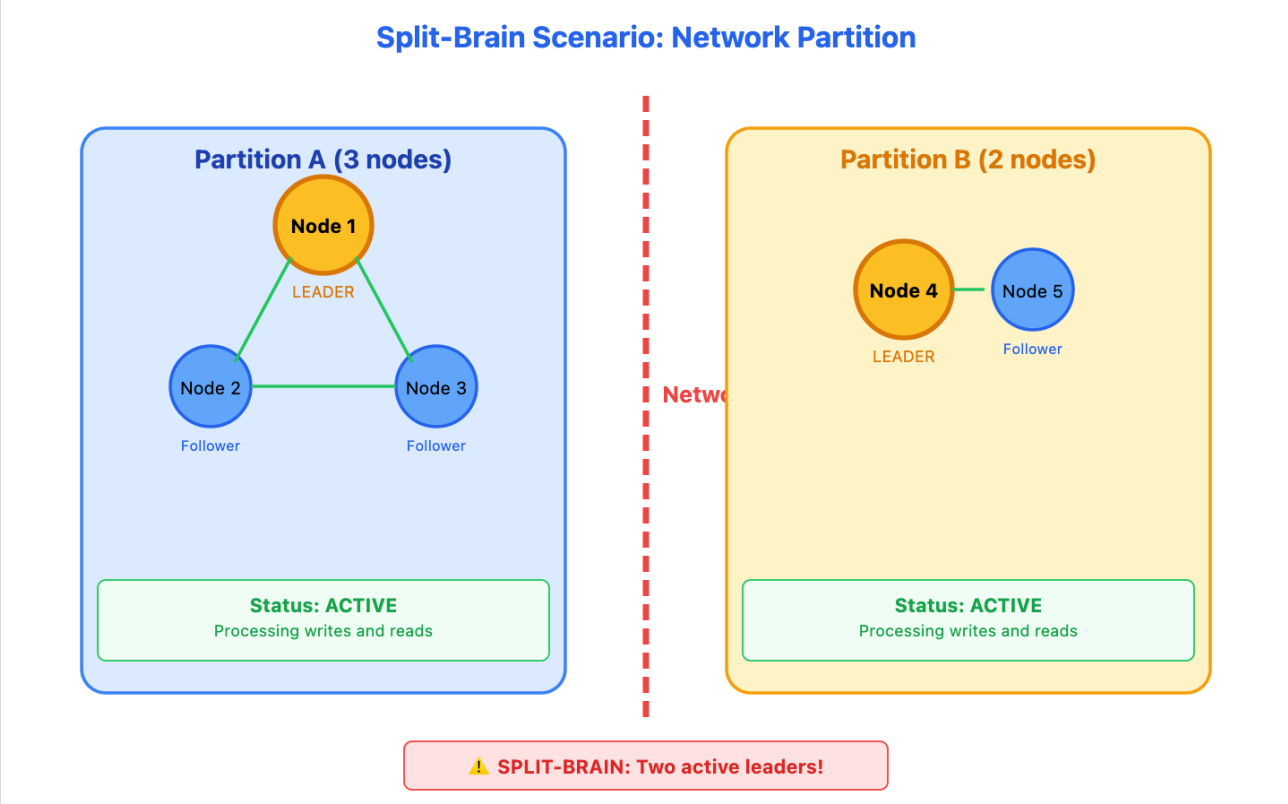
Порівняння моделей доставки повідомлень: At-most once, At-least once та Exactly once

ЧЕТВЕРТИЙ РІВЕНЬ ПРОБЛЕМ

- У розподілених системах відмови неминучі, і саме вони визначають реальну надійність черги. RAID та подібні технології не захищають від збоїв серверів, мережі чи дата-центрів. Тимчасові проблеми можуть призводити до split brain, коли частини кластера працюють незалежно й створюють конфліктні дані. Постійні відмови — це вихід з ладу вузлів, обладнання або цілих майданчиків. Вони впливають на дві ключові властивості: доступність і надійність повідомлень. Тому архітектура черг повинна враховувати втрату вузлів, мережеві розриви та ризик втрати даних.



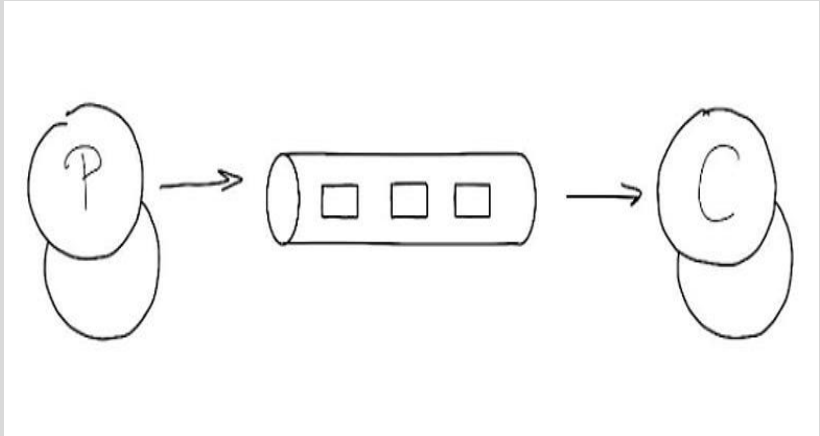
Основні типи відмов у розподілених системах



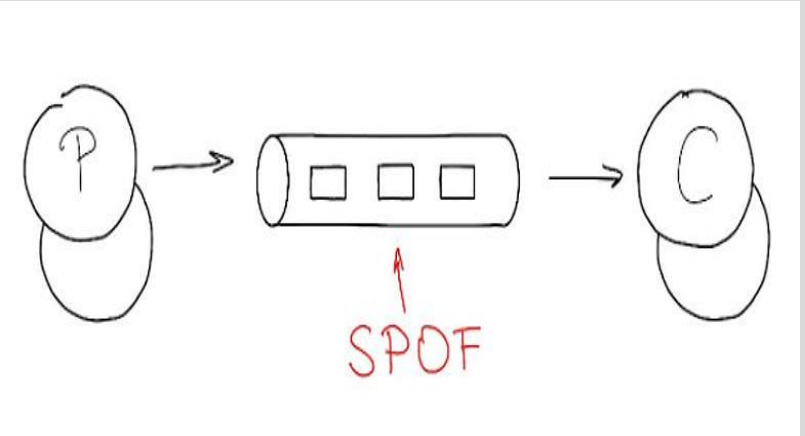
Сценарій split-brain, спричинений мережевою сегментацією кластера

ТОПОЛОГІЯ ЧЕРГ

- У конфігурації single instance один брокер і одна черга забезпечують простоту, але створюють єдину точку відмови. Збій сервера одразу зупиняє систему, спричиняє затримки або втрату повідомлень, а також обмежує масштабованість — уся робота залежить від можливостей одного вузла.
- У варіанті multi-instance кілька брокерів працюють паралельно, що підвищує продуктивність і доступність. Однак надійність залишається обмеженою, оскільки кожне повідомлення належить одній конкретній черзі й може бути втрачено при відмові інстансу.
- Частково це вирішує дублювання повідомлень у кілька черг, що зменшує ризик втрат, але вимагає контролю, щоб уникнути повторної обробки, або застосування ідемпотентних операцій.



Один брокер, одна черга



Низький рівень варіативності відмовостійкості системи

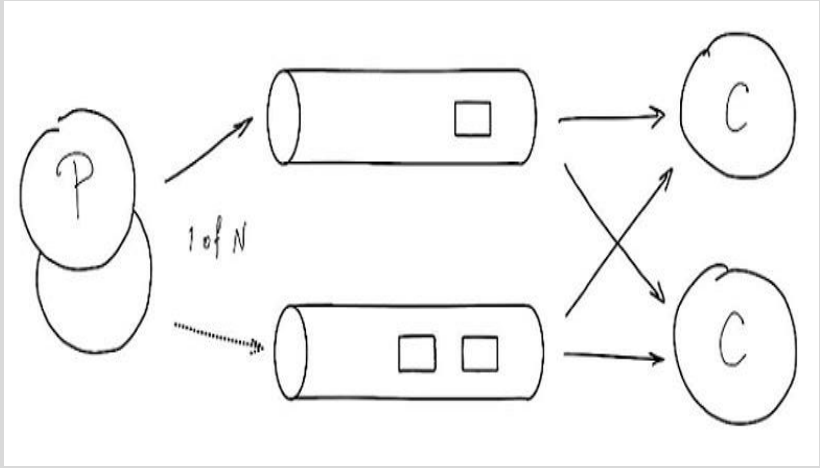
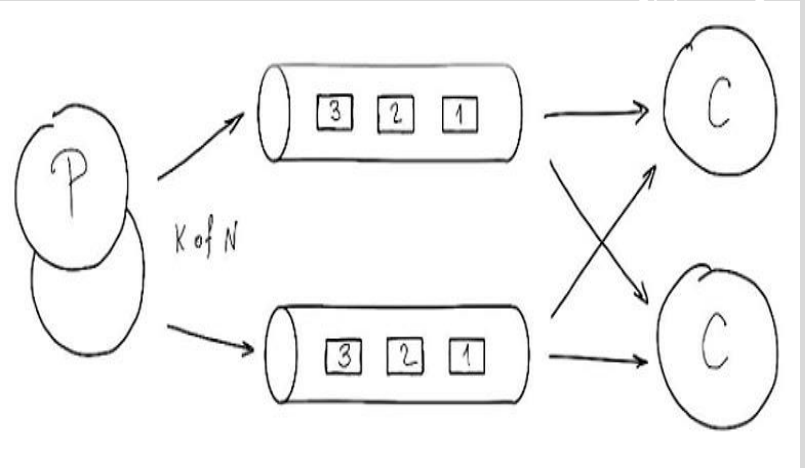
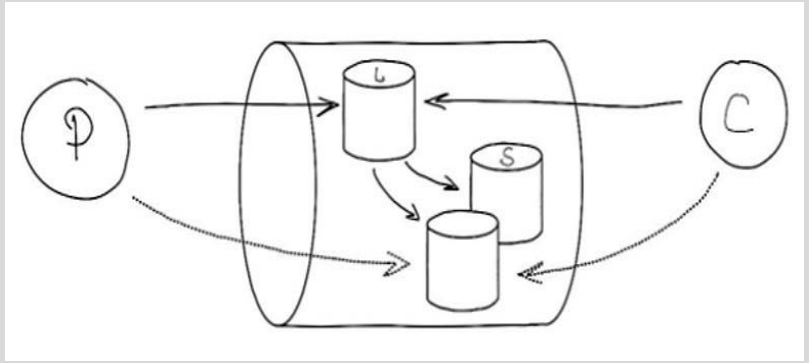


Схема системи multi-instance

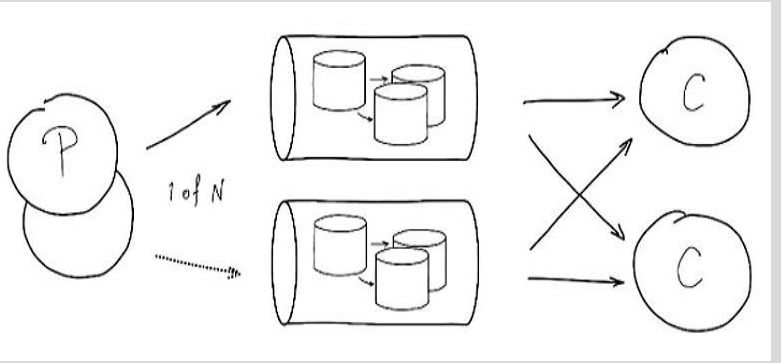


Повідомлення належить лише одній черзі

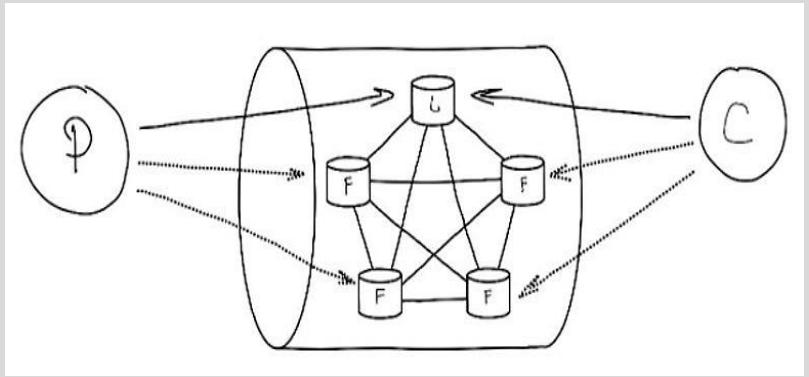
- Ефективнішим за окремі брокери є підхід із реплікацією, де продюсер і консюмер працюють із лідером, а інші вузли синхронно чи асинхронно копіюють дані. У разі збою лідера репліка автоматично перебирає його роль, що забезпечує доступність і захист від втрати повідомлень.
- Реплікована архітектура підвищує масштабованість і надійність, оскільки дані зберігаються на кількох вузлах. Кворумна реплікація дає ще сильніші гарантії ($X \approx 1$), адже запис вважається успішним лише після підтвердження більшістю реплік, хоча це знижує швидкість обробки.
- Під час мережевої сегментації працює лише частина кластера з кворумом, тому важливо правильно розподіляти вузли між зонами доступності.



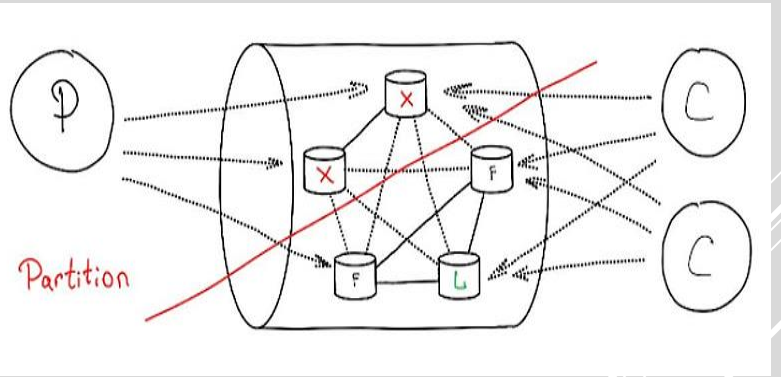
Процес реплікації



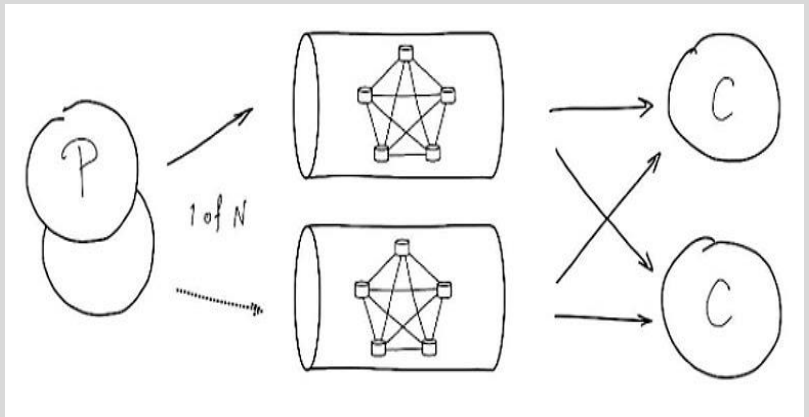
Репліковані черги, 1 із N (підхід «як база даних»)



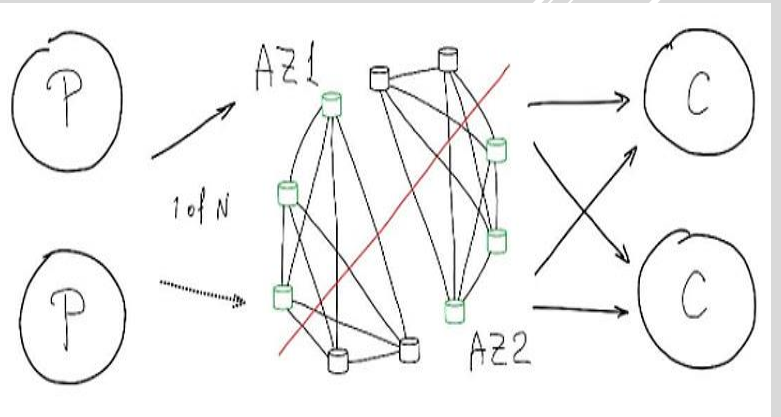
Підхід баз даних: кворум



Відмови у ситуаціях partitioning



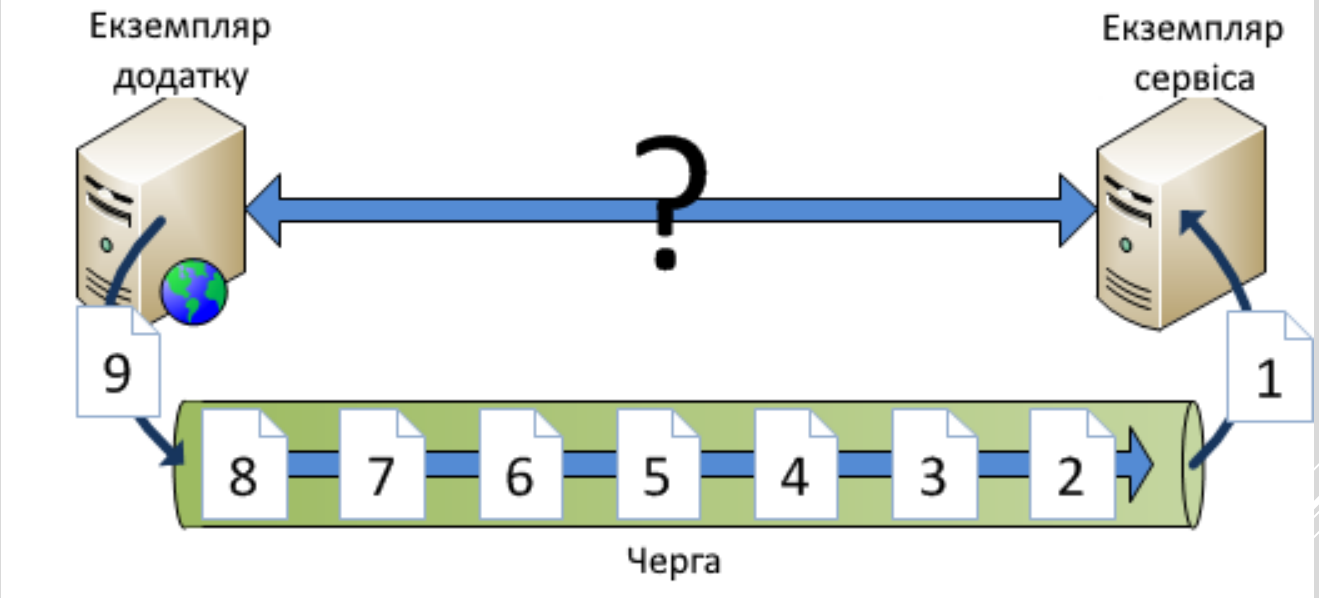
Кворумні черги, 1 з N



Кворумні черги, 1 з N

УПРАВЛІННЯ ХМАРНИМИ ЧЕРГАМИ

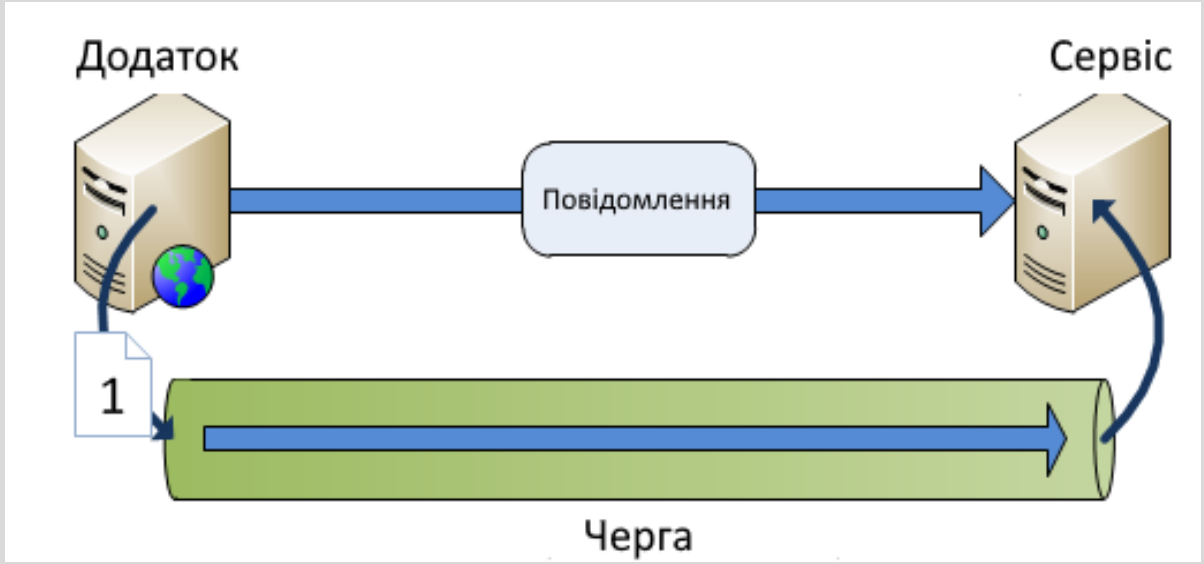
- У розподілених системах одним із найпоширеніших способів організації взаємодії між окремими екземплярами застосунку є використання хмарних черг, зокрема Azure Queue Storage. Проте покладатися лише на періодичне опитування черги неефективно: воно створює затримки, збільшує кількість транзакцій і не дає системі реагувати миттєво. Екземпляр не знає, коли з’явиться нове повідомлення, тому змушений регулярно робити запити, навіть якщо черга порожня. Це призводить до двох проблем: затримки між надходженням задачі та початком її обробки, а також великої кількості «холостих» транзакцій. Система опиняється перед дилемою: або швидке реагування за високу ціну, або низькі витрати, але значні затримки. У сценаріях із нерівномірним потоком задач такий механізм працює особливо погано, спричиняючи пропущені події, зростання черги та необґрунтовані витрати ресурсів.



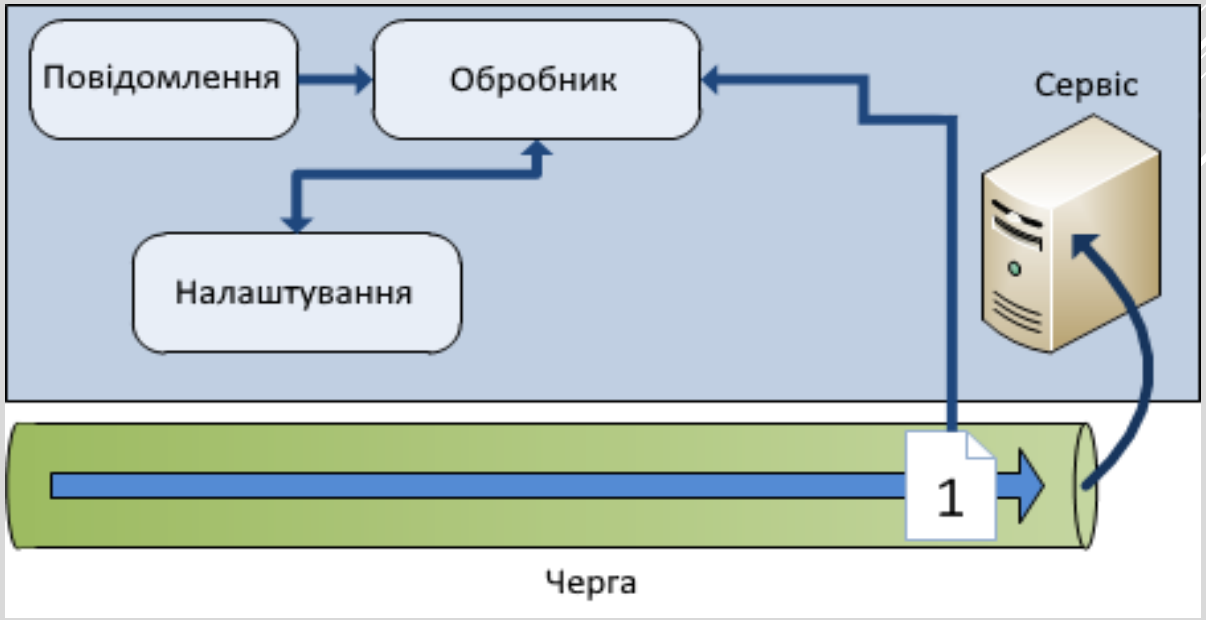
Стандартні засоби SDK для Azure передбачають сценарій взаємодії виключно через чергу

ПІДХОДИ ТА РІШЕННЯ

- Механізм обміну повідомленнями між екземплярами сервісу дозволяє усунути затримки та зайві транзакції, характерні для стандартного підходу з постійним опитуванням Azure Queue. Після додавання нового завдання екземпляр одразу надсилає сигнал іншим вузлам, і обробка стартує миттєво, без циклічних перевірок. Це робить систему подійно-орієнтованою, знижує навантаження на інфраструктуру, усуває порожні запити та забезпечує швидку реакцію на події. Збереження стану у стійкому сховищі гарантує надійність та відновлення після збоїв. У результаті модель стає ефективнішою, швидшою та економнішою.



Обробка черги відразу ж після додавання завдання до черги



Детальний процес виконання завдання

ВИСНОВКИ

- У роботі досліджено методи обробки черг в інфокомунікаційних мережах та показано, що сучасні черги є критичним елементом надійних і масштабованих розподілених систем.
- Розглянуто моделі впорядкування задач (FIFO, LIFO, Best Effort), механізми пріоритизації, дублювання, використання dead-letter черг, TTL і повторної доставки.
- Систематизовано чотири рівні проблем чергування: алгоритмічні обмеження, системні ефекти продуктивності, мережеву невизначеність і відмови інфраструктури.
- Проаналізовано моделі підвищення надійності — реплікацію, кворумні механізми, кластеризацію.
- Підкреслено роль моніторингу та експлуатаційних практик у запобіганні збоям.
- Зроблено висновок, що глибоке розуміння архітектурних і алгоритмічних особливостей черг є ключем до створення стабільних і ефективних систем.

ДЯКУЮ ЗА УВАГУ!

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Михайло НЕЩЕРЕТ

Назва: МЕТОДИ ОБРОБКИ ЧЕРГ В ІНФОКОМУНІКАЦІЙНИХ МЕРЕЖАХ

Координатор: Ганна ГРИНКЕВИЧ

Підрозділ: ННІТ

Коефіцієнт подібності 1:0

Коефіцієнт подібності 2:0

Тривога: 0

Після аналізу Звіту подібності констатую наступне:

☐ виявлені в роботі запозичення є сумлінними і не мають ознак плагіату. Тому робота визнається самостійною і допускається до захисту;

☐ виявлені в роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і самостійності її автора. Роботу направити на доопрацювання;

☐ виявлені в роботі запозичення є недобросовісними і мають ознаки плагіату або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. У зв'язку з чим, робота не допускається до захисту.

…… ……

Дата *Підпис Наукового керівника*