

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.043

«До захисту допущено»

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___» _____ 2025 р.

Магістерська дисертація

на здобуття ступеня магістра

**за освітньо-професійною програмою «Інженерія програмного забезпечення
інформаційних систем»**

зі спеціальності 121 «Інженерія програмного забезпечення»

**на тему: «Архітектура високонавантажених реактивних систем
комунікації в реальному часі»**

Виконав:

студент II курсу, групи ІІ-42мп

Гнітій Єгор Олександрович _____

Керівник:

доцент кафедри ІІІ, к.т.н.

Крамар Юлія Михайлівна _____

Рецензент:

доцент кафедри ІСТ, к.т.н.

Амонс Олександр Анатолійович _____

Засвідчую, що у цій магістерській дисертації
немає запозичень з праць інших авторів без
відповідних посилань.

Студент (-ка) _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

« ____ » _____ 2025р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Гнітію Єгору Олександровичу

1. Тема дисертації «Архітектура високонавантажених реактивних систем комунікації в реальному часі», науковий керівник дисертації Крамар Юлія Михайлівна, к.т.н., доцент, затверджені наказом по університету від «06» листопада 2025 р. № 4841-с

2. Термін подання студентом дисертації «15» грудня 2025 р.

3. Об'єкт дослідження – архітектура високонавантажених систем комунікації в реальному часі.

4. Предмет дослідження – архітектури програмного забезпечення та методи підвищення продуктивності і масштабованості високонавантажених систем комунікації в реальному часі.

5. Перелік завдань, які потрібно розробити – аналіз існуючих рішень; формулювання вимог та критеріїв до архітектури системи комунікації в реальному часі з урахуванням реактивних принципів і сучасних викликів масштабованості; розробка відповідної архітектури і алгоритма балансування персистентних з'єднань на основі доменних метрик із підтримкою безпечного відновлення сесій без втрат; моделювання та експериментальна оцінка

ефективності запропонованого рішення порівняно з традиційними підходами та існуючими рішеннями.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 5 плакатів.

7. Орієнтовний перелік публікацій – одна публікація.

8. Консультанти розділів дисертації

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

9. Дата видачі завдання «1» вересня 2025 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Збір та аналіз інформації	01.09.2025	
2	Аналіз існуючих рішень	05.09.2025	
3	Визначення функціональних вимог до архітектури	12.09.2025	
4	Проектування архітектури і розробка методу балансування	03.10.2025	
5	Написання програмної реалізації	10.10.2025	
6	Моделювання та експериментальна оцінка рішення	20.10.2025	
7	Розроблення стартап-проекту	22.11.2025	
8	Оформлення пояснювальної записки	23.11.2025	
9	Подання дисертації на попередній захист	26.11.2025	
10	Подання дисертації на захист	15.12.2025	

Студент

Єгор ГНІТІЙ

Науковий керівник

Юлія КРАМАР

РЕФЕРАТ

Дипломний проект на тему «Архітектура високонавантажених реактивних систем комунікації в реальному часі» складається з 108 аркушів, містить 17 ілюстрацій, 10 таблиць, 5 додатків, 20 посилань на джерела.

Актуальність теми: У сучасному інформаційному просторі все більшу роль відіграють системи комунікації в реальному часі, які забезпечують миттєвий обмін даними між користувачами. Зростання обсягів інформації та кількості користувачів створює підвищені вимоги до продуктивності, масштабованості та надійності таких систем. Традиційні підходи, засновані на блокуючих рішеннях, не завжди здатні забезпечити необхідний рівень продуктивності та навантаження. Використання реактивного стеку технологій може суттєво покращити ці параметри, дозволяючи зменшити витрати на ресурси для обробки великої кількості паралельних з'єднань.

Мета дослідження: Підвищення продуктивності, масштабованості та відмовостійкості систем комунікації в реальному часі, що вимагають обробки і підтримки великої кількості одночасних з'єднань.

Об'єкт дослідження: Архітектура високонавантажених систем комунікації в реальному часі.

Предмет дослідження: Архітектури програмного забезпечення та методи підвищення продуктивності і масштабованості високонавантажених систем комунікації в реальному часі.

Для реалізації поставленої мети **сформульовані наступні завдання:**

- аналіз існуючих рішень;
- формулювання вимог та критеріїв до архітектури системи комунікації в реальному часі з урахуванням реактивних принципів і сучасних викликів масштабованості;
- розробка відповідної архітектури і алгоритма балансування персистентних з'єднань на основі доменних метрик із підтримкою безпечного відновлення сесій без втрат;

— моделювання та експериментальна оцінка ефективності запропонованого рішення порівняно з традиційними підходами та існуючими рішеннями.

Наукова новизна: Пропонується нова архітектура високонавантажених систем комунікації в реальному часі за рахунок використання реактивного стеку та покращених методів балансування навантаження для оптимізації обробки і підтримки великої кількості паралельних з'єднань.

Практичне значення: Запропонована в результаті дослідження архітектура може бути використана розробниками для створення ефективних та масштабованих систем комунікації в реальному часі. Це сприятиме покращенню якості сервісів, що вимагають обробки великої кількості одночасних з'єднань та підвищить відмовостійкість серверів при збільшенні навантаження і зменшенні витрат на серверні ресурси.

Зв'язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

Апробація. Наукові положення дисертації пройшли апробацію на IX Міжнародній науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології (SoftTech-2025)»

Публікації. Наукові положення дисертації опубліковані в:

– Гнітій Є.О., Крамар Ю.М. Архітектура високонавантажених реактивних систем комунікації в реальному часі. Матеріали IX Міжнародної науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології (SoftTech-2025)». Секція кафедри інформатики та програмної інженерії. 26-28 листопада 2025 р. Київ.

Ключові слова: АРХІТЕКТУРА, БАЛАНСУВАННЯ, GRACEFUL RECONNECT, KUBERNETES, WEBSOCKET, SCALING.

ABSTRACT

Explanatory note size – 108 pages, contains 17 illustrations, 10 tables, 5 applications.

Topicality. In the modern information space, an increasingly important role is played by real-time communication systems that provide instant data exchange between users. The growth in data volumes and the number of users leads to higher requirements for the performance, scalability, and reliability of such systems. Traditional approaches based on blocking solutions are not always able to ensure the required level of performance under high load. The use of a reactive technology stack can significantly improve these parameters, reducing resource consumption for processing a large number of parallel connections.

The aim of the study. Increasing the performance, scalability, and fault tolerance of real-time communication systems that must process and maintain a large number of concurrent connections.

Object of research: the architecture of high-load real-time communication systems.

Subject of research: software architectures and methods of increasing the performance and scalability of high-load real-time communication systems.

To achieve this goal, the **following tasks** were formulated:

- analysis of existing solutions;
- formulation of requirements and criteria for the architecture of a real-time communication system, taking into account reactive principles and modern scalability challenges;
- development of an appropriate architecture and an algorithm for balancing persistent connections based on domain-specific metrics with support for safe, lossless session recovery;
- modeling and experimental evaluation of the effectiveness of the proposed solution in comparison with traditional approaches and existing systems.

The scientific novelty. A new architecture of high-load real-time communication systems is proposed through the use of a reactive technology stack and improved load-balancing methods to optimize the processing and maintenance of a large number of parallel connections.

The practical value. The architecture proposed as a result of the study can be used by developers to build efficient and scalable real-time communication systems. This will help improve the quality of services that require processing a large number of simultaneous connections, increase server fault tolerance under growing load, and reduce server resource costs.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Informatics and Software Engineering of the National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute».

Approbation. The scientific provisions of the dissertation were tested at the First All-Ukrainian Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies" (SoftTech-2025) - Kyiv, Ukraine.

Publications. The scientific provisions of the dissertation published in:

– Hnitiy Ye.O., Kramar Yu.M. Architecture of high-load reactive real-time communication systems. Proceedings of the 9th International Scientific and Practical Conference of Young Scientists and Students “Software Engineering and Advanced Information Technologies (SoftTech-2025)”, Section of the Department of Computer Science and Software Engineering, 26–28 November 2025, Kyiv.

Keywords: ARCHITECTURE, LOAD BALANCING, GRACEFUL RECONNECT, KUBERNETES, WEBSOCKET, SCALING

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	11
ВСТУП	12
1 ЛІТЕРАТУРНИЙ ОГЛЯД ЗАСТОСУВАННЯ ВІДМОВОСТІЙКИХ СИСТЕМ КОМУНІКАЦІЙ В РЕАЛЬНОМУ ЧАСІ.....	14
1.1 Опис предметної області	14
1.2 Етапи побудови відмовостійкої real-time системи.....	16
1.3 Постановка задачі дослідження	18
1.4 Висновок до розділу.....	19
2 МЕТОДИ ТА ЗАСОБИ ВИРІШЕННЯ ЗАДАЧІ ПРОЕКТУВАННЯ ВІДМОВОСТІЙКОЇ СИСТЕМИ КОМУНІКАЦІЇ В РЕАЛЬНОМУ ЧАСІ.....	21
2.1 Огляд існуючих рішень.....	21
2.2 Вимоги до цільової архітектури.....	24
2.2.1 Функціональні вимоги до архітектури системи	24
2.2.2 Нефункціональні вимоги до архітектури системи	25
2.3 Цільова архітектура реактивної системи комунікації реального часу	27
2.4 Детермінована маршрутизація та балансування персистентних з'єднань	27
2.4.1 Аргументація обраного метода консистентного хешування	28
2.4.2 Формула розрахунку ваг вебсокет-вузлів	32
2.4.3 Формула визначення цільового вузла для підключення клієнта.....	34
2.4.4 Маршрутизація повідомлень.....	35
2.5 Автоматичне масштабування на доменних метриках	37
2.6 Висновки до розділу.....	40

3 РЕАЛІЗАЦІЯ СИСТЕМИ З ВИКОРИСТАННЯМ РОЗРОБЛЕНОЇ АРХІТЕКТУРИ	41
3.1 Складові системи.....	41
3.1.1 Kubernetes-кластер та вхідний балансувальник	41
3.1.2 Кластер WebSocket Load Balancer	42
3.1.3 Кластер WebSocket-серверів	43
3.1.4 Реєстр станів і буфер повідомлень	43
3.1.5 Брокер повідомлень та доменні сервіси.....	44
3.1.6 Спостережуваність та автоматичне масштабування.....	44
3.2 Реалізація Websocket Load-balancer	45
3.2.1 Загальна структура сервісу.....	45
3.2.2 Алгоритм перерахунку ваг вузлів та прийняття рішення про масштабування.....	46
3.2.3 Алгоритм висушування веб-сокет вузлів при згортанні	50
3.3 Реалізація Websocket сервера	53
3.4 Розгортання системи в кластері Kubernetes	56
3.4.1 Підготовка кластера та налаштування файлових дескрипторів	56
3.4.2 Розгортання інфраструктури та сервісів застосунку	58
3.4.3 Конфігурація nginx-шару як edge-шлюзу	59
3.4.4 Система моніторингу - Prometheus і Grafana	60
3.4.5 Завершення розгортання та навантажувальне тестування	61
3.5 Висновки до розділу.....	62
4 МОДЕЛЮВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ.....	64
4.1 Експериментальне оточення.....	64

	10
4.2 Перебір коефіцієнтів і аналіз внеску метрик	65
4.3 Порівняння запропонованого рішення з традиційними підходами	72
4.4 Тестування стабілізації розподілу при змінах топології	79
4.5 Висновки до розділу.....	81
5 РОЗРОБЛЕННЯ СТАРТАП ПРОЄКТУ	84
5.1 Ідея стартап проєкту.....	84
5.2 Порівняння з конкурентами	86
5.3 Технологічний аудит ідеї проєкту	87
5.4 Аналіз ринкових можливостей запуску стартап-проєкту.....	89
5.5 Фактори загроз та можливостей	91
5.6 Ступеневий аналіз конкуренції на ринку	93
5.7 Висновки до розділу.....	96
ВИСНОВКИ	98
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	101
ДОДАТОК А	104
ДОДАТОК Б.....	105
ДОДАТОК В	106
ДОДАТОК Г	107
ДОДАТОК Д	108

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

- WS – протокол двостороннього обміну даними поверх TCP.
- API – програмний інтерфейс прикладного програмування (Application Programming Interface).
- TCP – протокол керування передаванням (Transmission Control Protocol).
- TTL – час життя (пакету) даних (Time to Live).
- TLS – протокол захисту транспортного рівня (Transport Layer Security).
- CPU – центральний процесор (Central Processing Unit).
- RAM – оперативна пам'ять (Random Access Memory).
- I/O – операції введення/виведення (Input/Output).
- MPS – кількість повідомлень на секунду (Messages Per Second).
- RPS – кількість запитів на секунду (Requests Per Second).
- SLA – угода про рівень сервісу (Service Level Agreement).
- SLO – цільовий показник рівня сервісу (Service Level Objective).
- SLI – індикатор рівня сервісу (Service Level Indicator).
- Kubernetes (K8s) – система оркестрації контейнеризованих застосунків.
- HPA – горизонтальний автомасштабувач подів Kubernetes (Horizontal Pod Autoscaler).
- KEDA – Kubernetes Event-Driven Autoscaling, фреймворк подійно-орієнтованого автоскейлінгу.
- CPaaS – платформа комунікацій як сервіс (Communications Platform as a Service).
- SaaS – програмне забезпечення як сервіс (Software as a Service).

ВСТУП

Стрімке зростання кількості онлайн-сервісів з миттєвим обміном даними – від месенджерів, біржових платформ до телемедицини й IoT-шлюзів – висуває підвищені вимоги до архітектур реального часу. Такі системи мають обробляти мільйони довготривалих підключень, гарантувати низьку затримку (p95/p99), бути відмовостійкими та еластично масштабуватись за непередбачуваних сплесків трафіку. Традиційні підходи з блокуючими моделями I/O і лінійним балансуванням (на кшталт простого round-robin) часто не забезпечують потрібної пропускну здатності й стабільності, погано керуються зворотним тиском (backpressure) і ускладнюють безпечне масштабування під час піків. Реактивна парадигма пропонує системний спосіб розв’язання цих викликів. Принципи Reactive Systems – responsiveness, resilience, elasticity та message-driven взаємодія – разом з подієво-орієнтованими мікросервісами, акторною моделлю та неблокуючими драйверами формують технологічний фундамент для високонавантажених рішень реального часу. Водночас у прикладних сценаріях залишаються невирішеними питання точного балансування персистентних підключень, збереження та відновлення сесій під час масштабування, а також адресації повідомлень без централізованого реєстру станів, що створює додаткові місця відмови й додатковий міжвузловий трафік.

У цій роботі досліджується архітектура високонавантажених реактивних систем комунікації в реальному часі, яка поєднує консистентне хешування для детермінованої маршрутизації через брокер повідомлень, механізм відновлення сесій без втрат для контрольованого перебалансування сесій, доменно-орієнтоване масштабування за ключовими метриками (кількість активних підключень, кількість повідомлень на секунда, p95-затримка доставки повідомлень, затримку черг повідомлень) та буферизацію пакетів для клієнтів зі змінною пропускну здатністю каналу (at-least-once delivery на базі кешу). Такий підхід дозволяє уникнути централізованого сховища стану, зменшити

міжвузловий трафік, підвищити відмовостійкість і стабільно тримати низьку затримку під час різких змін навантаження.

Методи дослідження. Аналітичний огляд і порівняльний аналіз архітектур; проектування та моделювання компонентів; імітаційне та натурне навантажувальне тестування (сценарії з різною інтенсивністю та тривалістю сесій); статистична обробка результатів за метриками p95/p99-latency, throughput, рівень відмов і повторних доставок; експерименти з політиками автоскейлінгу (у т.ч. експоненційні реакції на доменні метрики) у контейнеризованому середовищі.

Межі та припущення. Основна увага приділяється повідомленням даних користувача поверх WebSocket у застосунках з комунікацією реального часу (не медіастрімінг). Передбачається наявність надійного брокера повідомлень і мережевої інфраструктури дата-центру; питання глобальної консистентності даних поза межами маршрутизації повідомлень не є предметом детального розгляду.

1 ЛІТЕРАТУРНИЙ ОГЛЯД ЗАСТОСУВАННЯ ВІДМОВОСТІЙКИХ СИСТЕМ КОМУНІКАЦІЙ В РЕАЛЬНОМУ ЧАСІ

1.1 Опис предметної області

Високонавантажені реактивні системи комунікації в реальному часі (real-time communication, RTC) - це клас розподілених програмних систем, призначених для обміну подіями та повідомленнями з мінімальною затримкою між великою кількістю одночасно підключених клієнтів. Такі системи лежать в основі месенджинг-платформ, сервісів нотифікацій і колаборації, біржових та фінтех-рішень, телеметрії IoT і промислових шлюзів. Їхня відмінність від традиційних веб-застосунків полягає у персистентності з'єднань (WebSocket/MQTT/SSE), подієвій моделі взаємодії та необхідності витримувати мільйони одночасних сесій за гарантованих SLO щодо затримки, доступності та стабільності пропускну здатності.

З погляду архітектури предметна область формується перетином кількох дисциплін: розподілених систем, мережевих протоколів, реактивного програмування та інженерії надійності (SRE). Методологічною основою є принципи Reactive Systems – responsiveness, resilience, elasticity, message-driven взаємодія [1]. Реактивний підхід передбачає неблокуючі I/O-механізми, керування зворотним тиском (backpressure), ізоляцію збоїв та горизонтальне масштабування, що робить його природним вибором для реального часу.

Межі системи окреслюються каналами зв'язку між клієнтами та серверною платформою (інтернет/корпоративні мережі), а також внутрішньокластерною шиною повідомлень. Користувачі – кінцеві клієнти (web/mobile/IoT-пристрої) і внутрішні сервіси, що споживають події. Вхід – події/повідомлення, запити на підписку, сигнали керування, а вихід – доставлені повідомлення, підтвердження, push-події, метрики спостережуваності.

Типовий склад компонентів (рис 1.1):

- edge-шар / API-шлюзи: термінація TLS, авторизація, ліміти запитів, первинне балансування;
- шар довготривалих з'єднань: вузли WebSocket, що підтримують персистентні сесії та керують життєвим циклом підключень;
- брокер повідомлень: публікація/підписка, партиціонування, зберігання та ретрансляція подій (log-based або in-memory) [2].
- обчислювальні мікросервіси: маршрутизація, агрегація, фільтрація, трансформації, інтеграція з доменними сервісами;
- сховище станів і кеші: профілі сесій, маркери підтверджень, буферизація пакетів для клієнтів зі змінною пропускнуою здатністю;
- спостережуваність (observability): метрики затримки (p95/p99), пропускну здатність, активні з'єднання, черги, логи, трейси;
- автомасштабування/оркестрація: політики масштабування, розміщення та відновлення у контейнерній інфраструктурі.

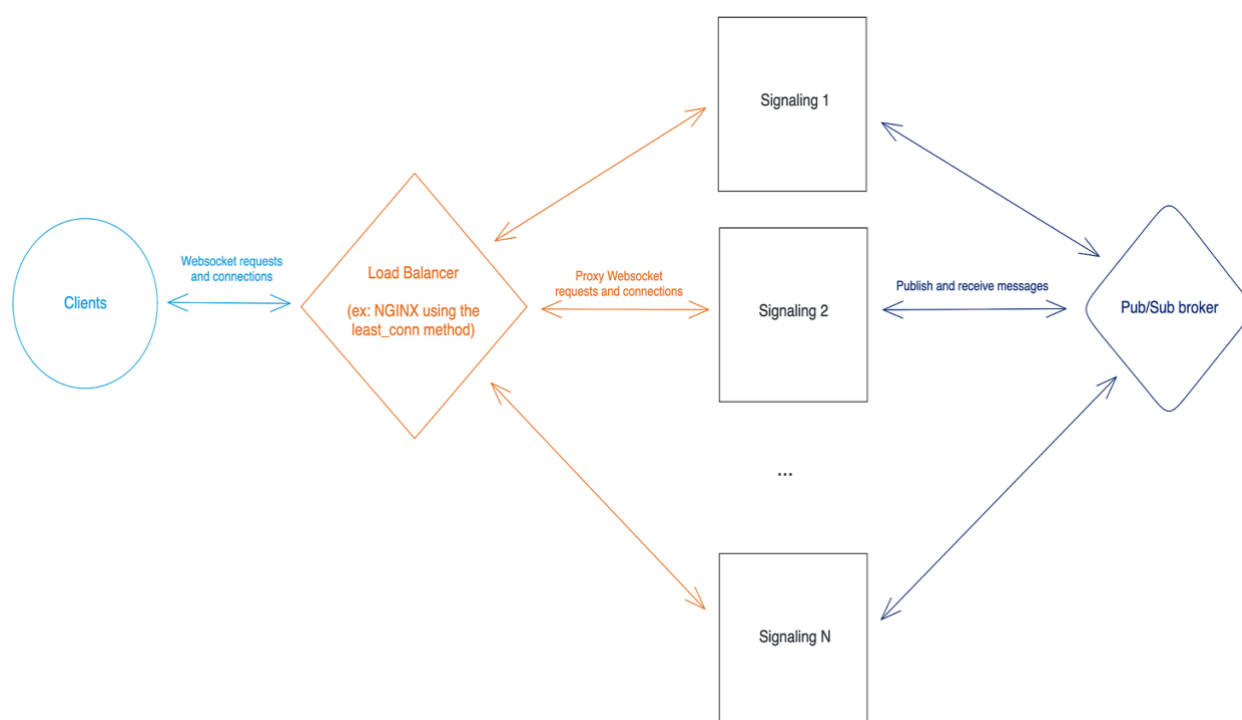


Рисунок 1.1 – Типова архітектура системи комунікації реального часу.

Ключові властивості та обмеження:

- низька та стабільна затримка. Важливі не лише середні значення, а й межі розподілу (p95/p99). На практиці це зумовлює відмову від блокуючих моделей і перехід до неблокуючих драйверів, реактивних потоків;
- масштабованість і еластичність. Система має коректно працювати як у «спокій» (тисячі підключень), так і під час піків (мільйони), адаптуючись до змін трафіку без деградації SLO;
- відмовостійкість. Ізоляція збоїв, самовідновлення, реплікація та узгоджені політики повторів;
- стан і адресація. Облік прив'язки клієнта до вузла, ідентифікація каналів доставки, гарантії (at-least-once/at-most-once/exactly-once) та механізми відновлення (graceful reconnect);
- ефективне балансування персистентних з'єднань. Статичні схеми (round-robin) часто ігнорують фактичний стан вузлів; потрібні детерміновані стратегії на кшталт консистентного хешування та балансування за доменними метриками;
- безпека. Автентифікація, авторизація, шифрування трафіку, контроль доступу до черг/каналів, аудит подій.

1.2 Етапи побудови відмовостійкої real-time системи

Аналогічно до процесу створення корпоративних IC, побудова реактивної real-time системи (рис. 1.1) включає впорядкований набір етапів:

- дослідження домену та моделювання навантаження: актори взаємодії, профілі трафіку, очікувані SLO/SLI (latency, availability, throughput);
- вибір архітектурних принципів і протоколів: Reactive Systems, подієва модель, WebSocket/MQTT/SSE, модель гарантій доставки;
- проєктування логіки маршрутизації та стану: консистентне хешування, політики прив'язки клієнтів, механізми graceful reconnect і повторних доставок;

- вибір інфраструктурних компонентів: брокер/стрімінг-шина, реактивні драйвери, кеші, сховища, засоби спостережуваності;
- розроблення політик автоскейлінгу: реакція на доменні метрики (кількість активних підключень, MPS, p95-latency) і запобігання «лавинним» чергам;
- безпека: аутентифікація/авторизація, шифрування, контроль доступів до каналів;
- тестування під навантаженням і стрес-тестування: валідація SLO, моделювання відмов, оцінка затримок;
- розгортання та експлуатація: оркестрація в Kubernetes, політики розміщення/відновлення, безперервний моніторинг і інцидент-менеджмент.

Патерни та технологічні підходи для побудови такої системи:

- подієво-орієнтовані мікросервіси: publish/subscribe, event sourcing, saga для узгодження довгих процесів без жорстких транзакцій [3];
- акторна модель: ізоляція стану та поштовхова (message-driven) взаємодія між «акторами» з чіткими кордонами відмов і природним backpressure;
- реактивні потоки: неблокуюче споживання/вироблення подій з контрактом зворотного тиску, що запобігає переповненню черг [4], [5];
- детермінована маршрутизація: консистентне хешування ідентифікаторів користувачів/каналів для прямої адресації через брокер, мінімізуючи міжндовий трафік і відмовляючись від централізованих реєстрів;
- graceful reconnect з джитером: контрольоване перебалансування сесій і безпечний scale-in/scale-out, що запобігає одночасним «штормам» перепідключень;
- буферизація та повтори: кеш-буфери для клієнтів зі «слабким» каналом і гарантована доставка (at-least-once) після відновлення зв'язку.

Причиною переходу до реактивної парадигми (рис. 1.2) є глобальне зростання кількості підключених пристроїв і сервісів, географічна розподіленість користувачів, мінливий характер трафіку (пікові «сплески», фан-

аут-розсилання), потреба у підтримці стабільних затримок та оптимізації витрат на інфраструктуру роблять традиційні блокуючі підходи малоприматними. Реактивні системи, спираючись на подієві інтерфейси, backpressure та горизонтальне масштабування, дозволяють ефективніше використовувати ресурси та зберігати керованість системи під навантаженням.

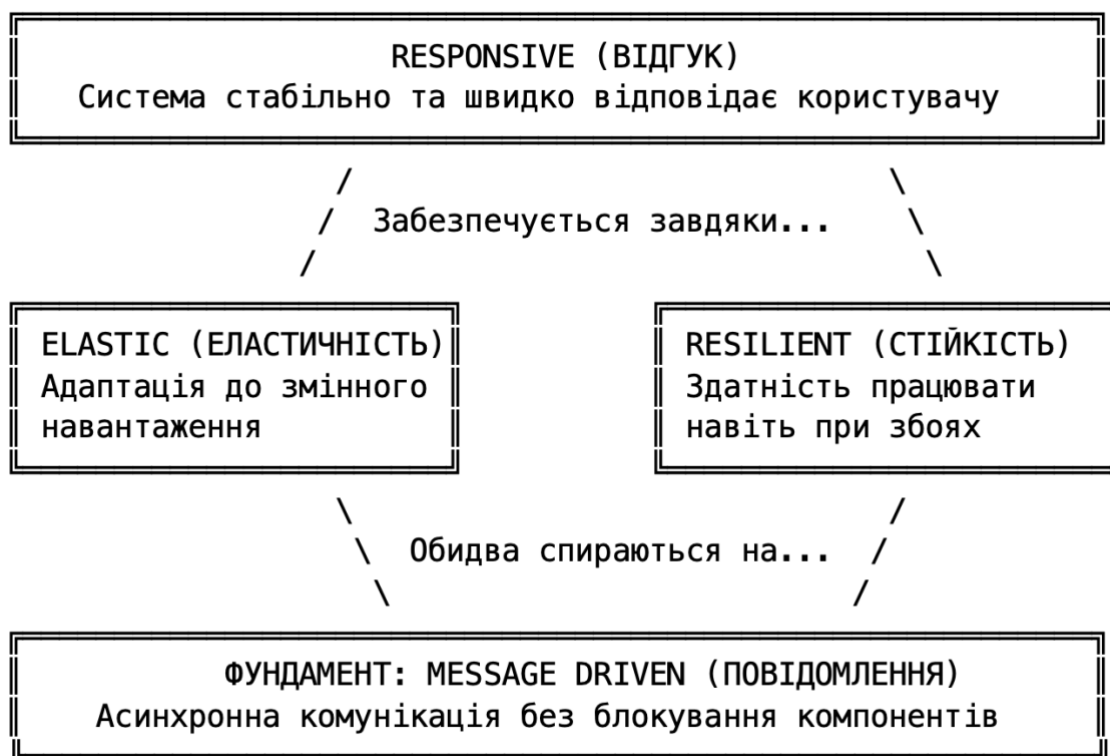


Рисунок 1.2 – Реактивні принципи.

1.3 Постановка задачі дослідження

Метою дослідження є підвищення продуктивності, масштабованості та відмовостійкості систем комунікації в реальному часі, що вимагають обробки і підтримки великої кількості одночасних з'єднань.

Для реалізації поставленої мети сформульовані наступні завдання:

- аналіз існуючих рішень;

- формулювання вимог та критеріїв до архітектури системи комунікації в реальному часі з урахуванням реактивних принципів і сучасних викликів масштабованості;
- розробка відповідної архітектури і алгоритма балансування персистентних з'єднань на основі доменних метрик із підтримкою безпечного відновлення сесій без втрат;
- моделювання та експериментальна оцінка ефективності запропонованого рішення порівняно з традиційними підходами та існуючими рішеннями.

1.4 Висновок до розділу

У розділі узагальнено предметну область високонавантажених real-time систем комунікацій і показано, що реактивний підхід (responsiveness, resilience, elasticity, message-driven) є природним базисом для систем із мільйонами персистентних з'єднань. Окреслено типову архітектуру (edge-шар, вузли довгих з'єднань, брокер подій, мікросервіси, кеші/сховища, спостережуваність, оркестрація) та ключові вимоги: стабільні затримки (p95/p99), еластичне масштабування, ізоляція збоїв, коректна робота з гарантіями доставки і безпекою. Проаналізовано обмеження класичних рішень (блокуючі I/O, «потік на з'єднання», алгоритм балансування round-robin) і обґрунтовано перехід до неблокуючих драйверів та подієвої взаємодії.

Сформовано набір архітектурних принципів, що адресують вузькі місця систем комунікації реального часу: детермінована маршрутизація через консистентне хешування (щоб уникнути централізованого реєстру станів і зменшити міжндовий трафік), безпечне відновлення сесій із джитером (для безпечного scale-in/scale-out без «штормів» перепідключень), доменно-орієнтоване масштабування (active connections, messages/sec, p95-latency) замість

суто CPU/RAM тригерів, кеш-буферизація для клієнтів зі змінною пропускнуою здатністю (підвищення гарантій доставки без непропорційного росту затримок).

Саме через відсутність цілісних рішень, які одночасно й ефективно вирішують stateful-аспекти з'єднань, точне балансування, еластичність і гарантовану доставку без централізованих «реєстрів» - подальша робота зосереджується на проектуванні та емпіричній перевірці реактивної архітектури з інтегрованим алгоритмом балансування персистентних з'єднань і політиками масштабування за доменними метриками. Критеріями оцінки стануть p95-затримка доставлень повідомлень, пропускну здатність (кількість повідомлення на секунду здатних бути обробленими сервером), стабільність під сплесками навантаження, обсяг міжвузлового трафіку й вартість ресурсів. Перехід до розділу 2 присвячено формалізації вимог і детальному проектуванню цільової архітектури.

2 МЕТОДИ ТА ЗАСОБИ ВИРІШЕННЯ ЗАДАЧІ ПРОЕКТУВАННЯ ВІДМОВСТІЙКОЇ СИСТЕМИ КОМУНІКАЦІЇ В РЕАЛЬНОМУ ЧАСІ

2.1 Огляд існуючих рішень

Протягом останнього десятиліття архітектура високонавантажених реактивних систем комунікації зазнала суттєвого розвитку: від статичних WebSocket-кластерів до гнучких мікросервісних платформ, що динамічно масштабуються і витримують мільйони одночасних з'єднань. Ключовими векторами еволюції стали суворе дотримання принципів Reactive Manifesto [1], що формалізує вимоги до responsiveness, resilience, elasticity та message-driven interaction, перехід на event-driven мікросервіси [2], а також інтеграція акторної моделі для ізоляції збоїв та backpressure-керування трафіком [6].

Утім, попри велику кількість публікацій, комплексні рішення, які б однаково ефективно забезпечували збереження стану з'єднань, динамічне масштабування і точне балансування трафіку для персистентних підключень, залишаються поодинокими.

Значна частина сучасних досліджень зосереджується на подієво-орієнтованих мікросервісах. Робота [2] описує багаторегіональний notification-сервіс, де деревоподібний шар Kafka і Redis-кеш служать інструментами масштабованого fan-out-розсилання повідомлень. В [3] роботі узагальнюється best practice-підходи до event-driven microservices, виділяючи publish/subscribe, event sourcing та saga як ключові патерни для реактивних транзакцій. Проте в обох роботах stateful-аспекти з'єднань перекладені на централізовані сховища, що створює точку потенційного відмовлення та збільшує міжндовий трафік.

Експериментальний перехід від класичного multi-threading до реактивних потоків засвідчує відчутне зниження затримки та підвищення пропускну здатності. В [4] продемонстровано, що використання реактивного стеку і методів спрощує горизонтальне масштабування, а акторна модель природно ізолює збої у високонавантажених системах. Аналогічні переваги акторного підходу

доведено в [6], де актори не лише розділяють обчислення й комунікацію, а й застосовують типобезпечні повідомлення, підвищуючи надійність IoT-шлюзів.

Проблема еластичності набуває особливої ваги у глобально розподілених кластерах. Робота [7] пропонує поєднання прогностичних моделей навантаження із реактивною стратегією *proactive-reactive global scaling*, що скорочує час реакції під час пікових підвищень навантаження. У середовищі *EdgeX Foundry autoscaling* реалізовано засобами *Kubernetes*, однак тригери базуються винятково на метриках CPU та RAM [8], а отже не враховують специфіку *real-time* трафіку, зокрема кількість відкритих з'єднань чи швидкість надходження повідомлень.

Ще один критичний аспект – *backpressure*, в публікації [5] показано, що у *Spark Streaming* стандартний механізм зворотного тиску утримує *throughput* лише для помірних навантажень, а за «burst»-трафіку виникають затримки й часті зупинки GC (*garbage collector*), [9] цю проблему вирішено, інтегруючи акторну модель з алгоритмами *backpressure* всередині компонентної шини реактивних потоків. Попри це, підхід не розглядає буферизацію повідомлень для клієнтів із мінливою пропускною здатністю каналу, для повторного надсилання повідомлень під час відновлення швидкості каналу.

Польові кейси провідних компаній підтверджують, що управління станом з'єднань і маршрутизація повідомлень в реальному часі без централізованого реєстру залишаються критичними викликами для масштабованих *real-time* систем. У статті Netflix “Pushy to the Limit” [10] описано перехід від Redis-кластерів до внутрішнього шардованого реєстру з міжрегіональною синхронізацією для зменшення крос-трафіку при широкомовних повідомленнях. Аналогічно, Dyte [11] наголошує на недоліках традиційного балансування типу *round-robin*, яке не враховує фактичну кількість активних підключень до кожного сервера, що призводить до перевантаження окремих нод.

Ці приклади підтверджують актуальність підходу, заснованого на консистентному хешуванні, якого дотримується ця робота: маршрутизація здійснюється одразу на рівні брокера повідомлень, спираючись на стабільне

перетворення ідентифікатора користувача для визначення з'єднання з сервером, без залучення централізованого реєстру. Це дозволяє забезпечити ефективну адресацію, уникнути дублювання трафіку між нодами та суттєво підвищити масштабованість системи при зростанні навантаження.

Запропонована архітектура вирішує зазначені проблеми комплексно. Graceful reconnect із псевдовипадковим інтервалом дозволяє поступово перебалансовувати сесії та безпечно згортати надлишкові сервери у фазі зниження навантаження, що особливо корисно для нічних «спадів» трафіку в глобальних сервісах та економії коштів на інфраструктурі.

Консистентне хешування ідентифікатора користувача для визначення з'єднання з сервером забезпечує одразу правильну адресацію повідомлень через message-broker. Таким чином усувається необхідність у централізованому сховищі стану, а мережевий трафік між нодами знижується.

Автоматичне масштабування побудоване на експоненційному алгоритмі, що реагує на доменно-специфічні метрики: activeConnections, messages per second, p95-latency. Такий підхід швидше адаптується до сплесків порівняно з лінійним НРА, зменшуючи ризик лавиноподібного накопичення черг на з'єднання. Для користувачів зі слабким каналом введено шар Redis-кешу: пакети буферизуються й доставляються після відновлення каналу, реалізуючи гарантію at-least-once delivery.

Для реалізації запропонованої архітектури буде використано реактивний програмний стек та застосовано принципи реактивних систем як зазначено в роботі [12], такий підхід дозволить ефективно масштабувати систему в умовах непередбачуваного навантаження. Крім того, результати досліджень підтверджують доцільність переходу до реактивних інтерфейсів взаємодії з інфраструктурними компонентами. Зокрема, у [13] доведено, що використання message broker-клієнтів із реактивним API дозволяє зменшити затримки у передачі повідомлень під час пікового навантаження, а робота [14] демонструє, що реактивні драйвери для баз даних істотно знижують кількість одночасних

потоків у системі, вивільняючи ресурси для обробки real-time подій. Це відповідає принципам створення високонавантажених систем, де ключовими є контроль над ресурсами та здатність до відновлення без втрати продуктивності.

2.2 Вимоги до цільової архітектури

Реактивна система комунікації в реальному часі – це розподілена мікросервісна система, орієнтована на події, що забезпечує двосторонній обмін повідомленнями з гарантованими межами затримок (SLO), високою доступністю (HA) та контрольованою деградацією під навантаженням (backpressure). Через це система має відповідати поставленим вимогам.

2.2.1 Функціональні вимоги до архітектури системи

Функціональні вимоги до цільової архітектури високонавантаженої системи комунікації реального часу:

- підтримка персистентних з'єднань. Система має підтримувати встановлення та обслуговування довготривалих WebSocket з'єднань для великої кількості одночасно підключених клієнтів (десятки-сотні тисяч);

- управління життєвим циклом сесій. Архітектура повинна забезпечувати створення, підтримку, коректне завершення та відновлення клієнтських сесій, включно з обробкою тайм-аутів і неочікуваних розривів;

- відновлення сесій без втрат із збереженням стану. Система має реалізовувати механізм керованого повторного підключення (graceful reconnect) з використанням ідентифікатора клієнта/сесії, що дозволяє відновлювати сесію без втрати логічного порядку повідомлень;

- детерміноване балансування персистентних з'єднань. Архітектура повинна забезпечувати маршрутизацію клієнтів та їхніх повідомлень на основі консистентного (weighted) хешування, використовуючи доменні метрики

(кількість активних з'єднань, MPS, latency, затримки брокера повідомлень, використання процесора і оперативної пам'яті) для встановлення ваг вузлів;

- маршрутизація повідомлень через брокер без централізованого реєстру станів. Система має виконувати адресацію та доставку повідомлень через брокер повідомлень без використання централізованого «реєстру сесій», покладаючись на консистентне хешування та локальний стан вузлів, але може використовувати реєстр станів як резервний варіант при неможливості визначити вузол по клієнтському ідентифікатору (наприклад в момент згортання якихось вузлів);

- буферизація та гарантована доставка для клієнтів зі змінною пропускнуою здатністю. Архітектура повинна забезпечувати кеш-буферизацію повідомлень для «повільних» клієнтів і реалізовувати як мінімум гарантію at-least-once delivery при відновленні з'єднання.

- збір та агрегація метрик для балансування і масштабування. Система має збирати технічні та доменні метрики (кількість активних з'єднань, messages/sec, p95/p99-latency, CPU, RAM, Kafka-lag тощо) та на їх основі коригувати конфігурацію балансування і політики масштабування;

- підтримка масштабування в контейнерному середовищі. Архітектура повинна інтегруватися з механізмами Kubernetes (HPA/KEDA або еквівалент) та підтримувати сценарії автоматичного масштабування (scale-out/scale-in) вузлів реального часу з попереднім переведенням їх у стан drain (деградації);

- спостережуваність та тестування під навантаженням. Система має експортувати метрики, логи та трейси для моніторингу, а також підтримувати запуск навантажувальних сценаріїв (імітація різних профілів клієнтів і сесій) для експериментальної оцінки ефективності архітектури.

2.2.2 Нефункціональні вимоги до архітектури системи

Нефункціональні вимоги до цільової архітектури високонавантаженої системи комунікації реального часу:

- продуктивність та затримка. Система повинна забезпечувати низьку та стабільну затримку доставки повідомлень, значення p95 latency мають відповідати встановленим SLO (500мс) для цільових сценаріїв навантаження;
- масштабованість та еластичність. Архітектура має підтримувати горизонтальне масштабування до мільйонів персистентних з'єднань і еластично адаптуватися до різких сплесків і спадів трафіку без суттєвої деградації сервісу;
- відмовостійкість і стійкість до збоїв. Відмова окремих вузлів не повинна призводити до загальної недоступності системи, має бути забезпечено автоматичне вимкнення проблемних нод, повторне підключення клієнтів та збереження повідомлень на період відновлення;
- обмеження міжнодового трафіку. Архітектура повинна мінімізувати міжнодовий трафік за рахунок локальної доставки там, де це можливо, та відмови від централізованих сховищ стану як «єдиної точки істини» для маршрутизації;
- ефективне використання ресурсів. Завдяки реактивному стеку та неблокуючим драйверам система має забезпечувати сублінійне зростання споживання ресурсів (CPU, пам'ять, кількість потоків) відносно кількості з'єднань;
- спостережуваність та керованість. Система має бути спроектована з урахуванням повної спостережуваності (metrics, logs, traces), забезпечувати наявність дашбордів для моніторингу та спрощувати аналіз інцидентів і налаштування політик масштабування;
- підтримуваність та розширюваність. Архітектура і структура коду повинні дозволяти подальше розширення (нові метрики, політики балансування, типи клієнтів, додаткові протоколи) без радикальної перебудови базових компонентів;
- межі застосовності та припущення. Система орієнтована на обробку даних застосунків поверх WebSocket (не медіастрімінг) і передбачає наявність

надійної інфраструктури (Kafka, Redis, Kubernetes-кластер), яка не є об'єктом оптимізації в даній роботі.

2.3 Цільова архітектура реактивної системи комунікації реального часу

Цільова архітектура (діаграма архітектури наведена в додатку А) має включати наступні компоненти:

- шар балансування: термінація TLS, лімітування запитів, маршрутизація до вузлів довгих з'єднань;
- вузли персистентних з'єднань: керування сесіями, локальні буфери відновлення, доставка і пересилання повідомлень на потрібні вузли;
- брокер подій: упорядкування, збереження, реплікація; маршрутизація повідомлень;
- сервіси обробки: доменна логіка;
- сховище/кеш: структури зі швидким доступом (Redis/Key-Value) для буферів повідомлень, резервного зберігання стану сесій;
- спостережуваність і керування: метрики, логи, трейси, автоскейлінг, політики.

Архітектурний принцип - стан сесій не централізується. Призначення вузла й шард подій відбуваються детерміновано за одними й тими самими ключами, тому міжндові перекидання повідомленнями мінімізуються. Для планових операцій масштабування використовується graceful reconnect з джитером і відновленням.

2.4 Детермінована маршрутизація та балансування персистентних з'єднань

Запропонована архітектурна політика забезпечує узгоджений для всіх компонентів кластеру вибір «власника» ключа ідентифікатора клієнта (clientId) і

мінімізує міжндовий трафік та кількість примусових перепідключень під час змін навантаження чи складу кластера. У центрі підходу - версійована політика розподілу (distVersion), яку публікує керуючий WebSocket Load Balancer (WS-LB). Політика містить перелік активних вузлів і їхні ваги, розраховані за доменними метриками (кількість активних з'єднання, швидкість доставки повідомлень, затримки, глибини і затримки черг повідомлень, використання процесора і оперативної пам'яті). Усі вузли застосовують одну й ту саму детерміновану вагову функцію (зважений rendezvous-хешинг [15] – рис. 2.1) з параметрами, заданими WS-LB. За потреби вибір обмежується кандидатною підмножиною нод (зона доступності), не втрачаючи детермінованості. Таким чином для будь-якого `clientId` однозначно визначається `policy owner` (первинний власник), але за необхідності власник може бути визначений через запит до реєстру станів - резервний метод.

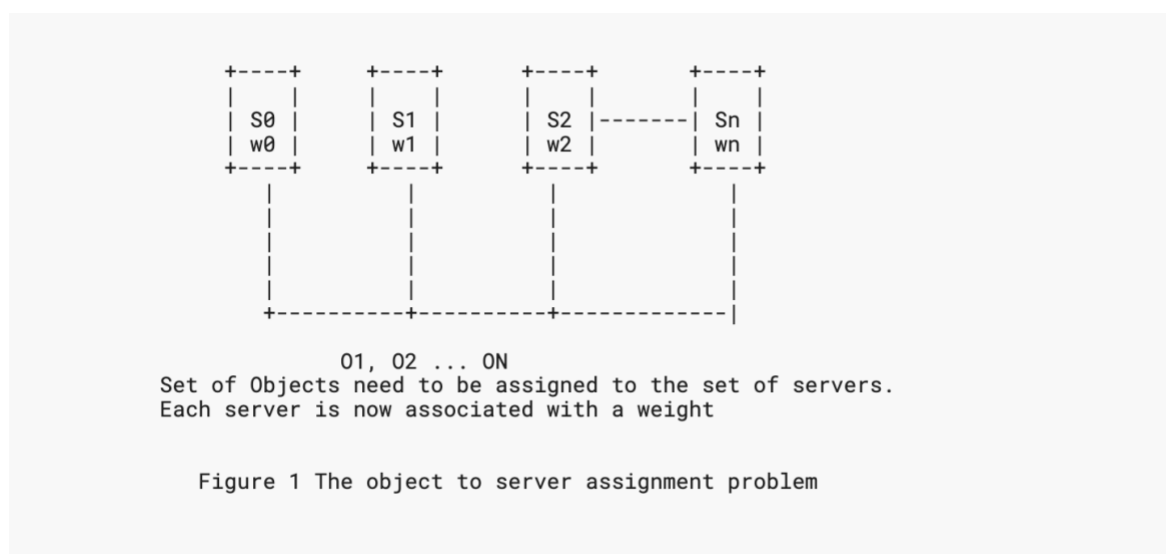


Рисунок 2.1 - Зважений rendezvous-хешинг.

2.4.1 Аргументація обраного метода консистентного хешування

Ключовою вимогою до архітектури є детерміноване й стабільне балансування персистентних WebSocket-з'єднань між вузлами кластера без централізованого «реєстру сесій» та з урахуванням поточного навантаження на

кожен вузол. В межах цієї роботи для розв'язання задачі обрано і модифіковано під прикладний сценарій зважений rendezvous-хешинг (HRW, Highest-Random-Weight) з ієрархічним «скелетоном» (skeleton-based hierarchical weighted rendezvous hashing) та складною системою ваг, побудованою на доменних метриках (CPU, RAM, active connections, messages/sec, p95-latency, Kafka lag).

На відміну від класичних балансувальників (round-robin, random, least-connections), які працюють із короткими HTTP-запитами й не враховують того, що WebSocket-з'єднання є довготривалими та станоорієнтованими, HRW дозволяє прив'язати клієнта до конкретного вузла через детерміновану функцію від `clientId` (клієнтського ідентифікатора) і набору вузлових ваг. Поки версія політики розподілу (`distVersion`) не змінюється, будь-який компонент кластера, маючи лише ідентифікатор клієнта, прийде до того самого рішення щодо «власника» з'єднання. Це важливо для наскрізної маршрутизації через брокер: і вузол персистентних з'єднань, і WS-load-balancer виконують один і той самий обчислювальний крок і не звертаються до спільного сховища стану лише заради визначення цільового сервера. Що у свою чергу зменшує використання трафіка, та додає відмовостійкості системі через можливість працювати без реєстру станів, вихід зі строю якого може зупинити доставку повідомлень.

У базовому HRW кожен кандидат-вузол отримує псевдовипадковий бал $score_i(x)$ для конкретного клієнта x , і обирається вузол з максимальним значенням. У роботі використано Gumbel-трюк [16]: бал складається з логарифма ваги вузла та Gumbel-шуму, згенерованого з хешу від (`clientId` і `nodeId`). Це дозволяє коректно враховувати різні ваги серверів (менш навантажені вузли отримують більшу частку ключів) без введення великої кількості віртуальних нод, як у класичному хеш-кільці. Ваги, у свою чергу, обчислюються на основі поточних метрик вузла: якщо сервер перевантажений (високий CPU/RAM, велика кількість з'єднань, зростання p95-latency чи Kafka-lag), його вага автоматично зменшується, і HRW призначає йому меншу частку нових клієнтів. Таке поєднання консистентного ключ-вузол відображення й динамічної

адаптації ваг забезпечує балансування саме за доменними SLI, а не лише за інфраструктурними показниками.

Через те, що реалізація має працювати з потенційно сотнями й тисячами вузлів, пряме застосування HRW з повним перебором усіх кандидатів для кожного клієнта ($O(N)$) було б надто дорогим. Тому запропоновано ієрархічну «скелетну» структуру: вузли групуються у сегменти з обмеженим розміром листя, над ними будується дерево з фіксованим коефіцієнтом розгалуження, а на кожному рівні HRW застосовується не до всіх вузлів, а лише до невеликої кількості сегментів. Це дає логарифмічну складність вибору вузла ($O(\log N)$), зберігаючи при цьому математичні властивості HRW та можливість враховувати сумарну вагу піддерев. Такий підхід добре узгоджується з вимогами до продуктивності та затримки у високонавантаженій системі реального часу.

Порівнюючи обраний метод із традиційними алгоритмами балансування персистентних з'єднань, можна виділити кілька принципових відмінностей. Класичний round-robin добре працює для коротких запитів, але в умовах довготривалих WebSocket-сесій він швидко призводить до дисбалансу: якщо частина клієнтів активніша, а частина – пасивна, кількість з'єднань перестає бути хорошим наближенням до реального навантаження, а повторні підключення після збоїв розподіляються випадково, не враховуючи попереднього розміщення стану. Найменша кількість з'єднань (least-connections) частково розв'язує цю проблему, проте вимагає централізованого чи розподіленого реєстру кількості підключень по всіх вузлах, що вводить додатковий оверхед синхронізації і створює потенційну точку відмови. IP-hash [17] фіксує клієнта на конкретному сервері, але не дає можливості контролювати частку ключів на кожному вузлі за доменними метриками, крім того, у сучасних мобільних мережах IP-адреса клієнта може змінюватися, що робить метод непридатним.

Серед власне методів консистентного хешування найчастіше застосовують хеш-кільце (ring-based consistent hashing [18]). Воно забезпечує стабільне відображення ключів на вузли та мінімізує частку ключів, які мігрують при

додаванні або видаленні вузла. Проте для підтримки різних потужностей серверів доводиться використовувати віртуальні ноди: вузол з більшою продуктивністю додається на кільце багаторазово. Це ускладнює конфігурацію, а головне – робить динамічну зміну «ваг» дорогою операцією: для корекції частки навантаження потрібно регенерувати множину точок на кільці, що призводить до помітної міграції ключів. У запропонованій реалізації вага вузла змінюється без модифікації «топології» хеш-простору: HRW використовує логарифм ваги безпосередньо у формулі бала, а ієрархічний скелет перебудовується атомарно на основі оновлених ваг. Це дозволяє частіше перебалансовувати ваги відповідно до поточного стану системи (кожні кілька хвилин або при виявленні аномалій), не спричиняючи лавиноподібних міграцій сесій.

Ще один відомий метод – `jump consistent hash` [19], який дає $O(1)$ -обчислення відображення ключ – `bucket` та мінімізує перерозподіл ключів під час зміни кількості «відер». Однак класичний `jump-hash` орієнтований на рівномірний розподіл, а підтримка різних ваг для вузлів реалізується або через складні модифікації алгоритму, або знову ж таки через непряму схему з множинними `bucket-ідентифікаторами`. У контексті цієї роботи важливо не лише рівномірно розподілити клієнтів, а й робити це з урахуванням композитного навантажувального бала, що враховує і доменні, і інфраструктурні метрики. Зважений HRW природно підтримує такі ваги, а `skeleton-структура` дозволяє зберегти прийнятну складність обчислення, тому перехід до `jump-hash` не дав би суттєвих переваг, але ускладнив би інтеграцію з існуючою системою моніторингу та перерахунку ваг.

Важливою перевагою обраного підходу є його інтеграція зі стратегіями `graceful reconnect` і процесом масштабування. Завдяки версійованій політиці розподілу вузли можуть позначатися як `draining`, зменшуючи їхні ваги до нуля і поступово виводячи їх із обслуговування без миттєвого розриву активних сесій. З точки зору HRW такі вузли просто перестають вигравати конкуренцію за нові

клієнтські ключі, тоді як існуючі з'єднання обслуговуються до закінчення relay-вікна. При аварійному виході вузла з ладу консистентний характер відображення дозволяє load-balancer-у однозначно визначити нового власника ключа й організувати двох ходову (2-hop) доставку залишкових повідомлень, не вдаючись до ширококомовних пошуків.

Отже, skeleton-based weighted rendezvous-хешинг, доповнений вагами на основі доменних метрик підходить для вирішення задачі. Він одночасно забезпечує: детерміновану адресацію без централізованого реєстру станів, підтримку персистентних з'єднань із мінімальною кількістю міграцій під час масштабування, можливість тонкого налаштування частки навантаження на вузол за рахунок адаптивних ваг та прийнятну обчислювальну складність завдяки ієрархічному «скелету». У порівнянні з класичними балансувальниками, кільцевим консистентним хешуванням та jump-hash запропонований метод краще узгоджується з вимогами до відмовостійкості, керованості й еластичності високонавантаженої системи комунікації реального часу.

2.4.2 Формула розрахунку ваг вебсокет-вузлів

Нехай маємо множину активних вузлів: $S = \{1, 2, \dots, N\}$, $N = |S|$. Для кожного вузла $i \in S$ відомі:

- c_i – завантаження CPU (нормалізовано 0...1);
- m_i – завантаження пам'яті (нормалізовано 0...1);
- l_i – p95 latency (затримка, мс);
- k_i – Kafka lag / read latency (затримка, мс);
- q_i – кількість активних з'єднань.

Максимальні значення метрик по кластеру (формула 2.2):

$$\begin{aligned} l_{\max} &= \max_j l_j, \\ k_{\max} &= \max_j k_j, \\ q_{\max} &= \max_j q_j. \end{aligned} \tag{2.2}$$

Нормовані значення, обрізані зверху до 1 (формула 2.3):

$$\begin{aligned}\hat{l}_i &= \min\left(\frac{l_i}{l_{\max}}, 1\right), \\ \hat{k}_i &= \min\left(\frac{k_i}{k_{\max}}, 1\right), \\ \hat{q}_i &= \min\left(\frac{q_i}{q_{\max}}, 1\right).\end{aligned}\quad (2.3)$$

Коефіцієнти важливості метрик виносимо у змінні (формула 2.4):

$$\begin{aligned}\alpha_{\text{cpu}}, \alpha_{\text{mem}}, \alpha_{\text{lat}}, \alpha_{\text{kafka}}, \alpha_{\text{conn}} &\geq 0 \\ \alpha_{\text{cpu}} + \alpha_{\text{mem}} + \alpha_{\text{lat}} + \alpha_{\text{kafka}} + \alpha_{\text{conn}} &= 1\end{aligned}\quad (2.4)$$

Композитний “навантажувальний” бал вузла i , чим більше – тим більше вузол завантажений (формула 2.5):

$$L_i = \alpha_{\text{cpu}} c_i + \alpha_{\text{mem}} m_i + \alpha_{\text{lat}} \hat{l}_i + \alpha_{\text{kafka}} \hat{k}_i + \alpha_{\text{conn}} \hat{q}_i. \quad (2.5)$$

Вводимо малий $\varepsilon > 0$, щоб уникнути ділення на нуль. Інверсний бал - чим менше навантаження, тим більша “цінність” вузла (формула 2.6):

$$S_i = \frac{1}{L_i + \varepsilon}. \quad (2.6)$$

Сума інверсних балів (формула 2.7):

$$S_{\text{tot}} = \sum_{j \in S} S_j. \quad (2.7)$$

Базова цільова вага на один вузол $W_{\text{node}} > 0$ (наприклад, $W_{\text{node}} = 100$).

Загальна сума ваг (формула 2.8):

$$W_{\text{tot}} = W_{\text{node}} \cdot N. \quad (2.8)$$

Попередня (ідеальна) вага вузла (формула 2.9):

$$w'_i = \frac{S_i}{S_{\text{tot}}} W_{\text{tot}}. \quad (2.9)$$

Мінімально допустима вага $w_{\min} > 0$.

Остаточні ваги для перших $N - 1$ вузлів (формула 2.10):

$$w_i = \max(w_{\min}, \text{round}(w'_i)), \quad i = 1, 2, \dots, N - 1. \quad (2.10)$$

де $\text{round}(\cdot)$ – округлення до найближчого цілого.

Вага останнього вузла добирається так, щоб сума точно дорівнювала W_{tot} (формула 2.11):

$$w_N = W_{\text{tot}} - \sum_{i=1}^{N-1} w_i. \quad (2.11)$$

У конкретній імплементації можуть бути використані такі значення коефіцієнтів (де α_i значення були виведені в ході подальших експериментів):

- $\alpha_{\text{cpu}} = 0.10$;
- $\alpha_{\text{mem}} = 0.20$;
- $\alpha_{\text{lat}} = 0.10$;
- $\alpha_{\text{conn}} = 0.50$;
- $\alpha_{\text{kafka}} = 0.10$;
- $\varepsilon = 0.1$;
- $W_{\text{node}} = 100$;
- $w_{\text{min}} = 10$.

Фінальну компактну формула ваги вузла без урахування округлення та мінімальної ваги, вагу вузла i можна записати у вигляді (формула 2.12):

$$w_i \approx W_{\text{tot}} \cdot \frac{\frac{1}{\alpha_{\text{cpu}}c_i + \alpha_{\text{mem}}m_i + \alpha_{\text{lat}}\hat{l}_i + \alpha_{\text{kafka}}\hat{k}_i + \alpha_{\text{conn}}\hat{q}_i + \varepsilon}}{\sum_{j \in S} \frac{1}{\alpha_{\text{cpu}}c_j + \alpha_{\text{mem}}m_j + \alpha_{\text{lat}}\hat{l}_j + \alpha_{\text{kafka}}\hat{k}_j + \alpha_{\text{conn}}\hat{q}_j + \varepsilon}} \quad (2.12)$$

Після цього до w_i застосовується:

- $w_i := \max(w_{\text{min}}, \text{round}(w_i))$,
- корекція останнього вузла так, щоб $\sum_i w_i = W_{\text{tot}}$.

2.4.3 Формула визначення цільового вузла для підключення клієнта

Нехай x – `clientId` (ідентифікатор клієнта), для кожного вузла $i \in S$:

- id_i – текстовий ідентифікатор вузла (`nodeId`);
- w_i – його вага (з попереднього блоку).

Використовується детермінована хеш-функція (формула 2.13):

$$h(x, \text{id}_i) \in \{0, 1, \dots, 2^{32} - 1\}. \quad (2.13)$$

де h – функція `Murmur3_32` (32-розрядна версія швидкої хеш-функції загального призначення `MurmurHash3` [20]);

x, id_i – конкатенована строка, що включає `clientId` та `nodeId`.

Нормалізуємо хеш у інтервал $[0, 1]$ (формула 2.14):

$$u_i = \frac{h(x, id_i) + 0.5}{2^{32}} \quad (2.14)$$

Будуємо Gumbel-шум (формула 2.15):

$$G_i = -\ln(-\ln(u_i)). \quad (2.15)$$

Ваговий HRW-бал вузла i для клієнта x (формула 2.16):

$$\text{score}_i(x) = \ln(w_i) + G_i \quad (2.16)$$

Тоді обраний вузол (target node) – це вузол із максимальним балом (формула 2.17):

$$k = \arg \max_{i \in S} \text{score}_i(x). \quad (2.17)$$

А відповідний ідентифікатор $\text{targetNodeId}(x) = id_k$.

Підставляючи G_i у функцію визначення вагового HRW-бала, маємо (формула 2.18):

$$\text{score}_i(x) = \ln(w_i) - \ln(-\ln(u_i)), \quad u_i = \frac{h(x, id_i) + 0.5}{2^{32}} \quad (2.18)$$

Фінальну компактну формулу визначення цільового вузла (id_k) для підключення клієнта (`clientId` – x) наведено в (2.19):

$$k = \arg \max_{i \in S} \left[\ln(w_i) - \ln\left(-\ln\left(\frac{h(x, id_i) + 0.5}{2^{32}}\right)\right) \right] \quad (2.19)$$

2.4.4 Маршрутизація повідомлень

Маршрутизація подій реалізована як дворівневий 2-hop routing без ширококомовних звернень. Відправник локально обчислює policy owner-а за поточною `distVersion` і робить рівно одну публікацію: або у відповідну чергу брокера (черга = ідентифікатору вебсокет сервера, модель «один споживач у групі»), або в буфер повідомлень, якщо власника не вдалось визначити. Весь подальший шлях контролює саме власник ключа, що гарантує збереження порядку для даного ключа: якщо активна сесія вже розміщена на цьому вузлі –

доставка локальна, якщо ж під час керованої міграції сесія тимчасово приєднана до попереднього вузла, власник виконує одну точкову внутрішньо-кластерну пересилку (relay) на фактичний хост. Для адресації такого винятку використовується реєстр станів (на кшталт Redis/JetStream KV) із записами «clientId - фактичний хост», задля підтримки масштабованості буде використано кластер presence-реєстру з автоматичном масштабування відповідно до використання CPU та RAM ресурсів, як зображено на рисунку 2.20. У підсумку повідомлення торкається не більше двох серверів (власник та за потреби - колишній хост), а всі інші сервери не включаються в обробку.

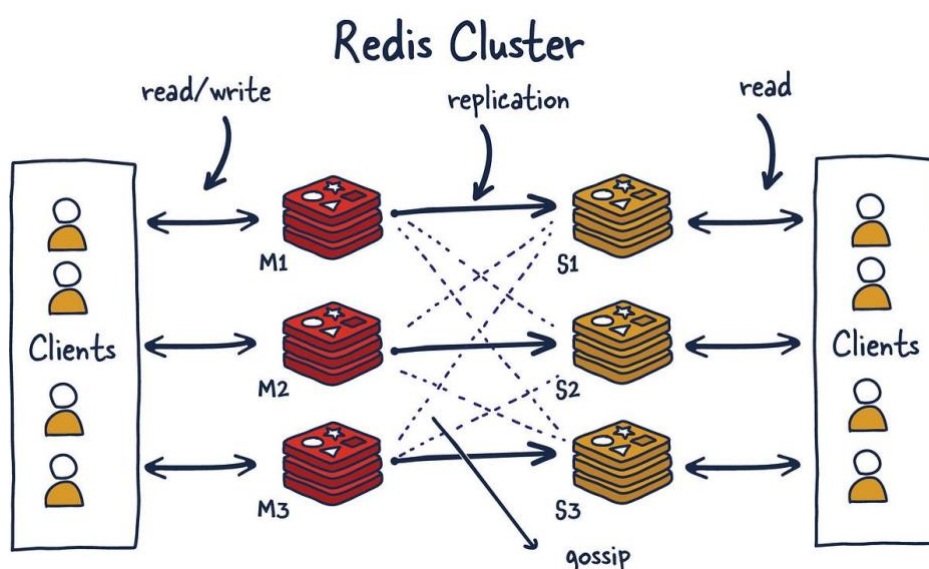


Рисунок 2.20 - Presence-реєстр (Redis) cluster.

Керування змінами складу/ваг відбувається двофазно і без масових перепідключень. Під час фази announce & drain LB позначає вузли, що виводяться, як draining, забороняючи нові підключення до них, потім черги осушуються протягом 5 хв, щоб не від'єднувати всіх одночасно, при цьому для вимкнення обирається сервер з найменшою кількістю підключень, а ваги інших вузлів коригуються так, щоб підготувати нове рівноважне розміщення. На фазі commit публікується нова distVersion і всі компоненти одночасно переходять на неї - від цього моменту будь-який вузол, маючи лише clientId, приходить до того

самого рішення щодо власника. Щоб мінімізувати примусові перепідключення діє наступний принцип активних сесій: існуючі з'єднання продовжують обслуговуватись там, де вони є, у межах relay-вікна з періодичним відключенням з'єднань невеликими порціями, тоді як нові чи перепідключені клієнти вже потрапляють на вузли згідно з новою політикою.

Семантика доставки – at-least-once з ідемпотентністю: кожне повідомлення має стабільний ідентифікатор, дублікати поглинаються бізнес-логікою без побічних ефектів. Після перепідключення клієнт відновлюється за токеном і отримує непрочитані події із буфера (в який були записані повідомлення які не були доставлені клієнту в зв'язку з його низькою пропускнуою здатністю або його перепідключенням). Операційні угоди включають цілі на доступність вузла з'єднань, затримки та частку переміщених шард на оновлення, час конвергенції до нової distVersion і частку relay-трафіку. Безпека політики гарантується підписаними оновленнями від WS-LB, короткоживучими ключами між сервісами та валідацією записів рестра станів. Сукупно ця політика зменшує міграції ключів і міжндовий трафік, забезпечує безвтратну доставку та плавну конвергенцію під час змін, не вимагаючи централізованого сховища сесій і не провокуючи масових перепідключень.

2.5 Автоматичне масштабування на доменних метриках

Автоматичне масштабування в запропонованій системі здійснюється не тільки за універсальними інфраструктурними ознаками (CPU/RAM), а за доменними SLI – індикаторами, що показують реальні числа пропускнуої здатності (рис. 2.21), що безпосередньо відображають стан реального часу: кількість активних з'єднань, швидкість надходження/доставки повідомлень, p95-latency на критичному шляху «Edge – Node – Broker – Node – Client», глибини черг повідомлень.



Рисунок 2.21 – SLA, SLO, SLI – рівні обслуговування.

Контур керування тісно інтегрований із політикою розподілу ключів: WS-LB публікує версійовані політики (distVersion) з вагами вузлів, а контур автоскейлінгу підтримує цільові рівні пропускної здатності та затримок, змінюючи кількість реплік вузлів персистентних з'єднань і споживачів брокера в межах заздалегідь визначених коридорів. Під час згортання (downscaling) застосовується двофазна процедура announce & drain – commit з readiness-гейтами, рейт-лімітами на нові hand-shake та джитером, для активних сесій застосовується «grandfathering» у межах relay-вікна, а presence-реєстр із коротким TTL дозволяє policy owner виконати лише один адресний relay для вже підключених клієнтів. Під час змін топології завжди йде перерахунок ваг серверів, щоб розділити навантаження рівномірно між новими, або існуючими серверами.

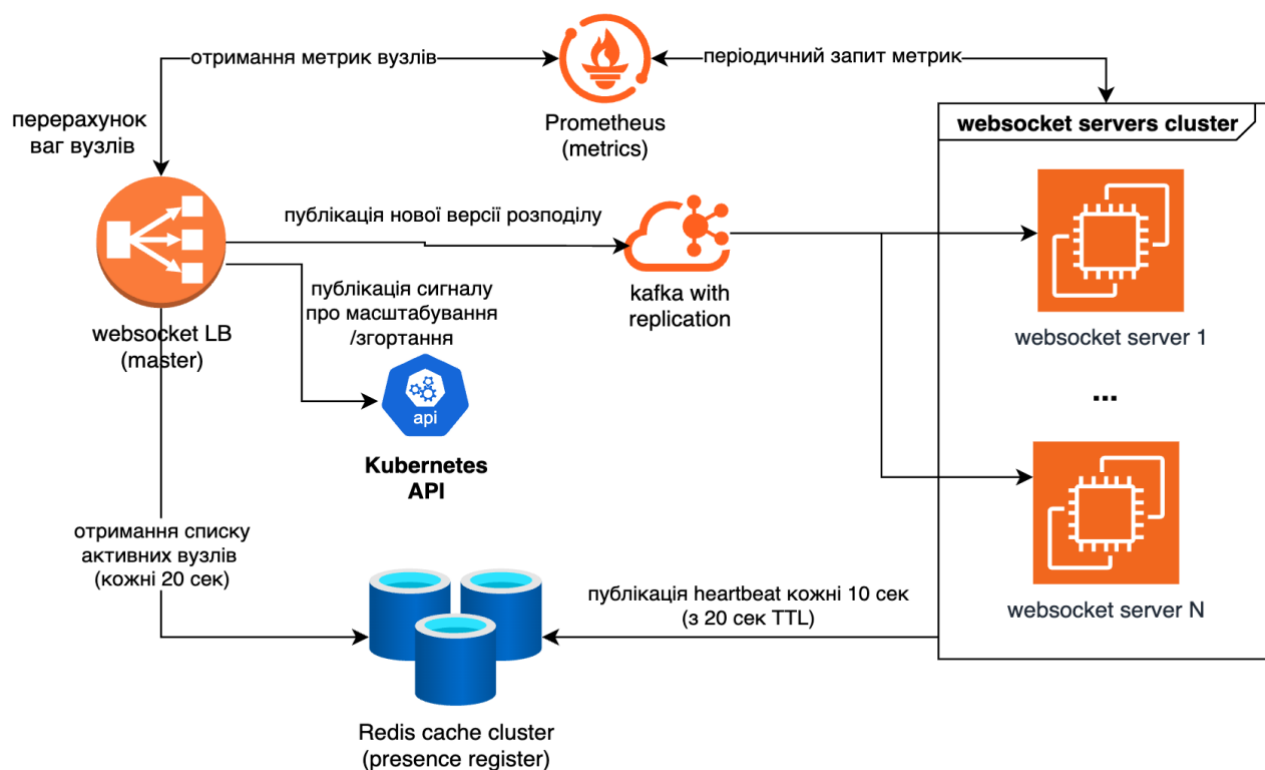


Рисунок 2.22 – Діаграма сповіщення серверів про оновлення стану (heartbeat) та нової версії функції балансування.

Кожен сервер, що тримає персистентні підключення записує свій стан в розподілений реєстр кожні 10 секунд (як показано на рисунку 2.22), потім WS-LB вичитує ці значення і порівнює з локально збереженими даними, що дає змогу швидко виявити коли і який сервер перестав відповідати, або що новий сервер піднявся і на нього потрібно направити трафік.

При зміні ваг деякі сервери мають від'єднати своїх клієнтів, щоб вирівняти навантаження, ця процедура проходить в межах заданого relay вікна (3хв), щоб уникнути масових перепідключень під час змін топології. При виході якогось сервера зі строю WS-LB ініціює підписку на його чергу вхідних повідомлень в межах заданого вікна, потім прочитані повідомлення адресуються до нового власника, це зроблено щоб уникнути ситуацій, коли після виходу серверу з ладу, його черга повідомлень була допрацьована. Спостережуваність реалізовано через єдину схему телеметрії (метрики, логи, трейси) з SLO/SLI на доступність, затримки. Під час сплескового навантаження можливе експоненціальне

масштабування шляхом порівняння зміни навантаження з часу останнього прийнятого рішення масштабуватись. Сукупність цих механізмів уніфікує поведінку системи під навантаженням, дає керовані, прогнозовані перебалансування й уникає масових перепідключень під час зміни складу кластера.

2.6 Висновки до розділу

У розділі запропоновано цілісну методику проєктування відмовостійкої системи комунікації в реальному часі, яка поєднує мікросервісну подієво-орієнтовану архітектуру з детермінованим, ваговим розподілом персистентних з'єднань без широкомовних звернень. Ключові рішення - версійована політика розподілу (`distVersion`) від `WS-LB`, короткоживучий `presence-реєстр` для адресного `relay` під час перехідного вікна і зберігання буферів повідомлень, а також доменно-орієнтоване масштабування - мінімізують міжндовий трафік, мінімізуються залежність від централізованого реєстру сесій, зменшують обсяг вимушених міграцій і забезпечують плавну конвергенцію під час змін складу та ваг нод. Обґрунтовано застосування `at-least-once` гарантії доставки із ідемпотентністю та механізмів відновлення сесій без втрат, що гарантує доставку й зберігає порядок повідомлень навіть під час планових і аварійних перебалансувань. Запропоновані `SLO/SLI`, процедури `announce & drain - commit` формують фундамент, за якого система зможе досягти вимог `responsiveness`, `resilience` та `elasticity` для мільйонів одночасних з'єднань у змінних умовах навантаження.

3 РЕАЛІЗАЦІЯ СИСТЕМИ З ВИКОРИСТАННЯМ РОЗРОБЛЕНОЇ АРХІТЕКТУРИ

3.1 Складові системи

Реалізація розробленої архітектури виконана у вигляді набору сервісів, розгорнутих у кластері Kubernetes (додаток А). Кожен компонент відповідає за окремий аспект роботи системи: приймання клієнтських підключень, балансування персистентних WebSocket-сесій, зберігання стану присутності, доставку повідомлень, доменну обробку, спостережуваність та автоматичне масштабування.

3.1.1 Kubernetes-кластер та вхідний балансувальник

Базою для всієї системи є кластер Kubernetes, у якому розгорнуто всі сервіси. Зовнішні клієнти під'єднуються до кластера через HTTP/WS-ендпоїнт, опублікований nginx-ingress-контролером.

Ingress виконує такі функції:

- термінація TLS та прийом вхідних HTTP/WS-з'єднань;
- початкове балансування трафіку між репліками WebSocket-load-balancer'а й допоміжних сервісів за схемою round-robin;
- проксування вебсокет з'єднань та їх адресація на основі змінних в шляху запитів;
- інтеграція з Horizontal Pod Autoscaler (HPA) для масштабування самого nginx-ingress-шару.

Таким чином, edge-рівень є входом до внутрішньої архітектури й не містить бізнес-логіки.

3.1.2 Кластер WebSocket Load Balancer

Наступний рівень – кластер WS load balancer'ів (websocket LB 1 ... websocket LB N). Це окремий сервіс, реалізований на Java з використанням реактивного стеку, який:

- приймає перші HTTP від клієнтів;
- обчислює цільовий WebSocket-вузол за допомогою алгоритму skeleton-based weighted rendezvous hashing;
- повертає клієнту URL конкретного WebSocket-сервера (ws-endpoint), до якого потрібно встановити персистентне з'єднання;
- зчитує з Redis-реєстру інформацію про тимчасові перенаправлення сесій (relay) у періоди масштабування;
- отримує від Prometheus агреговані метрики вузлів (CPU, RAM, active connections, MPS, p95-latency, Kafka-lag) і на їх основі перераховує ваги в хеш-функції та розсилає сигнали про масштабування або згортання;
- публікує оновлення політики розподілу (distVersion) через Kafka, щоб всі вузли кластера використовували однакову версію функції балансування.

Кожен вузол WS-LB не зберігає клієнтські сесії (stateless), тому його можна вільно масштабувати горизонтально. При цьому за допомогою Kubernetes Leases API один веб-сокет балансувальни обирається як лідер і тільки він може надсилати нові політики розподілу та сигнали масштабування, інші сервери можуть лише слухати його сигнали та обчислювати цільовий WebSocket-вузли для клієнтів та перенаправляти повідомлення на потрібні веб-сокет вузли. Для цього сервера налаштовані прості правила масштабування за допомогою Kubernetes HPA (вбудований контролер Kubernetes, який автоматично регулює кількість реплік (подів), щоб відповідати поточним потребам) вгору або вниз на основі CPU метрики.

3.1.3 Кластер WebSocket-серверів

WebSocket-servers cluster містить набір вузлів (websocket server 1 – websocket server N), які безпосередньо тримають персистентні WebSocket-з'єднання з клієнтами та виконують:

- установлення й підтримку сесій (handshake, ping/pong, heartbeats);
- локальну буферизацію повідомлень для клієнтів з нестабільним каналом;
- проксювання користувацьких повідомлень до Kafka та отримання подій з відповідних каналів;
- відправлення повідомлень назад клієнтам у правильному порядку.

Кожен вузол періодично надсилає heartbeat із власними метриками у Prometheus та WS-LB, що дозволяє Load Balancer'у підтримувати актуальний стан кластера й коригувати ваги вузлів.

3.1.4 Реєстр станів і буфер повідомлень

Redis cache cluster використовується як високошвидкісне розподілене key-value-сховище для:

- реєстру-станів (зв'язок clientId – фактичний вузол, який тимчасово відрізняється від того, що повертає консистентне хешування під час drain/relay-фази);
- зберігання короточасних буферів непрочитаних повідомлень, які потрібно доставити клієнту після відновлення з'єднання;
- фіксації heartbeats від WebSocket-вузлів з TTL, що дозволяє швидко виявляти вилучені сервери.

Redis розгорнуто у вигляді кластера з реплікацією, що усуває єдину точку відмови для presence-даних.

3.1.5 Брокер повідомлень та доменні сервіси

Kafka (з реплікацією) виконує роль надійного брокера повідомлень:

- приймає повідомлення від WebSocket-вузлів і розподіляє їх по партиціях за ключами (наприклад, `clientId` або `nodeId`);
- забезпечує впорядковану доставку повідомлень і можливість повторного читання в разі перебоїв зв'язку;
- слугує транзитним шаром між WebSocket-сервером і доменними мікросервісами.

На боці доменних сервісів розташована (application-level логіка), яка споживає події з Kafka, виконує бізнес-обробку (перевірки прав, оновлення стану, нотифікації тощо) і за потреби публікують нові повідомлення назад у брокер для подальшої доставки клієнтам.

3.1.6 Спостережуваність та автоматичне масштабування

Prometheus збирає технічні та доменні метрики з усіх компонентів: nginx-ingress, WS-LB, WebSocket-серверів, Redis- та Kafka-кластера, а також обчислювальних сервісів. На основі цих метрик будуються графіки в Grafana, а також приймаються рішення про масштабування.

Kubernetes API надає централізований інтерфейс керування станом кластера (кількість реплік, статус Pod'ів, їх readiness/liveness-проби). З ним інтегрується KEDA який:

- аналізує метрики (у тому числі доменні: `active connections`, `MPS`, `latency`);
- приймає рішення про scale-out/scale-in WebSocket-вузлів і, за потреби, WS-LB та доменних сервісів;
- ініціює drain-процедуру для вузлів, що вимикаються, щоб уникнути масових розривів з'єднань.

WS-LB у свою чергу враховує ці зміни, оновлюючи `distVersion` та ваги вузлів, що забезпечує узгоджене балансування в усьому кластері.

3.2 Реалізація WebSocket Load-balancer

WebSocket Load-balancer реалізовано як окремий мікросервіс на Java з використанням реактивного стеку (Project Reactor, Reactor Netty, Reactor Kafka, реактивний клієнт до Redis, Micrometer). Сервіс розгортається в окремому Deployment у Kubernetes і масштабується незалежно від вузлів WebSocket-серверів.

3.2.1 Загальна структура сервісу

На рисунку 3.1 зображено структуру проєкту модуля WebSocket Load-balancer-у.

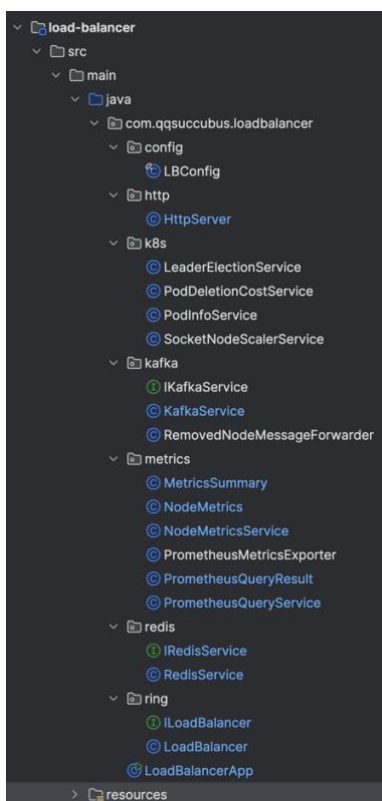


Рисунок 3.1 – Структура проєкту модуля WebSocket Load-balancer-у.

Точка входу – клас `LoadBalancerApp` (діаграма класів модуля зображена в додатку Б) відповідає за ініціалізацію всіх допоміжних сервісів і запуск HTTP-шару (`HttpServer`).

HTTP-інтерфейс використовується для:

- видачі клієнтам адреси цільового WebSocket-вузла (endpoint для початкового hand-shake, `resolve clientId - ws://.../serverId/...`);
- службових запитів (health-check, debug-ендпоїнти для огляду поточної конфігурації кільця, версії політики розподілу тощо).

Уся бізнес-логіка балансування зосереджена в класі `LoadBalancer`, який зберігає:

- реєстр активних вузлів (`NodeEntry: nodeId`, поточну вагу, останній heartbeat, прапор draining для вузла);
- поточну версію розподілу `DistributionVersion (distVersion)`;
- екземпляр алгоритму консистентного хешування `SkeletonWeightedRendezvousHash`, який реалізує skeleton-based weighted rendezvous hashing з логарифмічною складністю пошуку.

Функція `resolveNode(clientId)` викликається як з HTTP-шару, так і з модулів адресації, і повертає детерміновано обраний `NodeEntry` згідно з вагами та формулами, наведеними у підрозділі 2.4.2.

3.2.2 Алгоритм перерахунку ваг вузлів та прийняття рішення про масштабування

Алгоритм перерахунку ваг вузлів та прийняття рішення про масштабування (наведено в додатку В) базується на періодичній обробці heartbeat-повідомлень від WebSocket-серверів та їхніх метрик. На кожній ітерації `WebSocket Load-balancer` (вузол – лідер) отримує актуальний список активних вузлів разом із технічними та доменними показниками: завантаженням процесора і пам'яті, кількістю активних з'єднань, швидкістю обробки

повідомлень, р95-затримкою та затримками на стороні брокера повідомлень. На основі цих даних формується внутрішній реєстр NodeEntry (вузлів), де для кожного вузла зберігаються його ідентифікатор, останній heartbeat, поточна вага та ознака перебування у стані draining. Одночасно визначається, чи змінилася топологія кластера: чи з'явилися нові вузли, чи зникли існуючі, чи було оновлено список вузлів, що виводяться з експлуатації.

Далі Load-balancer вирішує, чи потрібно перераховувати ваги. Якщо склад кластера змінився (додано або вилучено вузли, змінено draining-стан), перерахунок виконується примусово. У протилежному випадку застосовується евристика, яка враховує час від попереднього перерахунку, ступінь дисбалансу за метриками (різницю між мінімальними та максимальними значеннями CPU, пам'яті та затримок), а також поточний стан «збіжності» системи. Якщо метрики свідчать про те, що навантаження вже рівномірно розподілене, а ваги вузлів узгоджені з фактичним станом, перерахунок пропускається, і оновлюються лише heartbeat-дані в локальному реєстрі. Також передбачено спеціальну перевірку перевантажених вузлів: якщо якийсь вузол демонструє високе завантаження й при цьому має вагу не нижчу за середню, алгоритм примусово запускає перебалансування, навіть якщо ще не сплив заданий мінімальний інтервал між перебалансуваннями, щоб швидко зменшити частку трафіку, яка спрямовується на проблемний сервер.

Якщо прийнято рішення про корекцію ваг, Load-balancer переходить до розрахунку. У випадку, коли дійсні метрики ще відсутні (усі значення близькі до нуля, система щойно стартувала або вузли ще не встигли зібрати телеметрію), усім вузлам призначаються рівні ваги. Загальна вага кластера задається як добуток базової ваги на один вузол та кількості вузлів, усім вузлам, крім останнього, встановлюється однакова вага, а вага останнього коригується так, щоб сума точно дорівнювала цільовому значенню. Це дає просту та передбачувану початкову конфігурацію для нових або щойно відновлених кластерів.

Коли ж метрики доступні, ваги обчислюються на основі композитного навантажувального бала, описаного у попередньому розділі. Для кожного вузла формуються нормалізовані значення затримок, кількості з'єднань та інших показників, після чого вони комбінуються із коефіцієнтами важливості, визначеними в розділі 2.4.2 (CPU, пам'ять, p95-latency, затримка брокера та кількість активних з'єднань). Результатом є числовий бал навантаження: чим він більший, тим більш завантаженим вважається вузол. Потім для кожного вузла розраховується інверсний бал і вузли з меншим навантаженням отримують більші інверсні значення, а отже й більшу частку цільової сумарної ваги. Вага кожного вузла обчислюється пропорційно до його інверсного балу, при цьому для запобігання «виключення» вузла зі схеми балансування встановлюється мінімально допустиме значення ваги. Для останнього вузла сума ваг коригується так, щоб зберегти загальну цільову вагу, визначену раніше. Паралельно сервіс збирання метрик фіксує окремі компоненти балу та кінцеві ваги для подальшого виводу графіків в Grafana.

Після завершення розрахунків Load-balancer оновлює локальний реєстр вузлів новими вагами, перезбирає структуру консистентного хешу `SkeletonWeightedRendezvousHash` і формує нову версію політики розподілу (`distVersion`). Ця версія містить перелік активних вузлів, їхні ваги та список вузлів у стані `drain` і є єдиною «точкою істини» для всієї системи. Оновлена політика публікується через брокер повідомлень і записується в Redis, завдяки чому всі екземпляри балансувальника і WebSocket-сервери у кластері одночасно переходять на нову конфігурацію. А сервери, що піднімаються після публікації цього повідомлення зможуть дістати актуальну версію розподілу з Redis-a. Детермінована функція вибору вузла, описана у розділі 2.4.3, при цьому залишається незмінною, змінюються лише параметри (ваги та склад вузлів), що забезпечує стабільне, але адаптивне балансування.

В цьому ж контурі (рис. 2.4) працює алгоритм прийняття рішення про масштабування. Після оновлення ваг і топології Load-balancer агрегує метрики

по всіх вузлах: середні та максимальні значення CPU, пам'яті, затримок, Kafka-lag, кількості активних з'єднань і швидкості обробки повідомлень. На основі цих показників формується «знімок навантаження», який порівнюється з попереднім станом. Далі застосовується набір правил, що визначає, чи варто додавати нові вузли, прибирати частину існуючих чи залишити конфігурацію без змін. Для збільшення кількості вузлів аналізуються ситуації перевантаження: високі середні або пікові значення CPU та пам'яті, зростання затримок і Kafka-lag, поєднання кількох «помірних» сигналів стресу. При цьому алгоритм враховує не лише поточний стан, а й динаміку: якщо кластер нещодавно уже масштабувався вгору і навантаження стабілізується або падає, повторний scale-out відкладається, щоб уникнути надлишкового розширення.

Кількість додаткових вузлів для scale-out обирається консервативно. За замовчуванням додається один вузол, а збільшення кроку можливе лише у випадку, коли навіть після останнього масштабування навантаження продовжує зростати й перевищує попередні значення із суттєвим запасом. Таким чином реалізується обережне експоненційне масштабування: система може реагувати більш агресивно на різкі піки, але в кожному рішенні враховується історія попередніх кроків, що зменшує ризик неконтрольованого росту кластера. Для зменшення кількості вузлів (scale-in) висуваються більш жорсткі умови: дуже низькі значення навантаження CPU і пам'яті, низькі затримки доставки та затримки брокера, а також перевірка, що після видалення одного вузла перерозподілене навантаження все ще залишатиметься в безпечних межах. До того ж scale-in можливий лише за умови, що у кластері працює більше двох вузлів (надлишковість задля забезпечення доступності), і відбувається не частіше, ніж через заданий певний інтервал часу.

Коли ухвалюється рішення про масштабування, Load-balancer формує пояснення (reason), що включає поточну кількість вузлів, запроповану цільову кількість, агреговані метрики та тип дії (scale-out або scale-in). Перед масштабуванням у бік зменшення окремий сервіс оновлює «вартість видалення»

pod'ів, щоб виводити з експлуатації вузли з найменшою кількістю активних підключень. Потім викликається сервіс, що відповідає за масштабування WebSocket-вузлів у Kubernetes. Паралельно «кандидатам на вимкнення» встановлюється стан drain, їхні ваги поступово зменшуються, а присутні в системі механізми graceful reconnect і relay забезпечують безпечне завершення або перенесення активних сесій у межах заданого вікна часу, не допускаючи масових одночасних перепідключень.

У підсумку описаний алгоритм формує замкнений керований цикл: обробка heartbeat-повідомлень та оновлення метрик, виявлення змін топології й дисбалансів, адаптивний перерахунок ваг на основі композитного навантажувального бала, оновлення консистентного хешу й розповсюдження нової політики розподілу, а також прийняття рішень про масштабування за доменними метриками з урахуванням історії навантаження. Така схема дозволяє підтримувати баланс між ефективним використанням ресурсів, стабільною затримкою й відмовостійкістю системи навіть за умов різких змін трафіку.

3.2.3 Алгоритм висушування веб-сокет вузлів при згортанні

Алгоритм висушування (draining) веб-сокет вузлів (наведено в додатку Г) спрацьовує у випадку, коли контур масштабування, описаний у підрозділі 3.2.2, ухвалює рішення про масштабування в бік зменшення. Після формування відповідного рішення Load-balancer обирає кандидатів на вимкнення серед активних pod'ів WebSocket-серверів. Для цього використовуються актуальні heartbeat-метрики: кількість активних з'єднань, затримки обробки й додаткова «вартість видалення» (deletion cost), розрахована окремим сервісом з урахуванням мінімального впливу на користувачів. У результаті вибирається вузол (або декілька), на якому зосереджена найменша частина навантаження й вимкнення якого є найменш болючим для всієї системи.

Після вибору кандидата Load-balancer переводить відповідний WebSocket-вузол у стан `draining`. На цьому етапі для нього вага повністю обнуляється у політиці розподілу, описаній у розділі 2.4. Це означає, що консистентна хеш-функція перестає призначати нові підключення на `draining`-вузол: усі нові клієнти відразу спрямовуються на інші, повністю працездатні вузли. Оновлена версія політики розподілу (`distVersion`), де відповідний вузол позначено як `draining`, публікується через брокер повідомлень і записується в Redis, завдяки чому всі учасники системи синхронно отримують оновлення складу кластера.

Далі протягом усього `drain`-вікна вузол продовжує обслуговувати вже встановлені сесії, але не приймає нові підключення. Для клієнтів, які за новою політикою розподілу логічно «належать» іншому вузлу, використовуються реєстр станів і механізм `relay`. Якщо поточний вузол ще тримає сесію, але консистентне хеш-відображення вже вказує на нового власника цього `clientId`, у реєстрі присутності досі фіксується фактичний хост, а повідомлення маршрутизуються через брокер так, щоб зберегти порядок доставки для конкретного клієнта. Таким чином забезпечується сумісність між старим розміщенням стану та новою політикою без широкомовних запитів і повного перерозподілу сесій.

Паралельно з цим `SessionManager` на `draining`-вузлі поступово зменшує кількість активних з'єднань. Протягом усього `drain`-вікна він періодично від'єднує невеликі порції клієнтів, додаючи випадковий джитер до моменту розриву з'єднань. Клієнти, які підтримують описаний у розділі 2 механізм `graceful reconnect`, автоматично перепідключаються за тим самим `clientId` і потрапляють вже на новий вузол згідно з оновленою консистентною хеш-функцією. Непрочитані події та повідомлення для цих клієнтів записуються в буфер повідомлень у Redis, звідки відновлюються після успішного перепідключення, зберігаючи `at-least-once` семантику та логічний порядок доставки.

Коли drain-вікно завершується або кількість активних з'єднань на вузлі практично досягає нуля, Load-balancer фіксує фінальний стан: переконується, що всі локальні черги повідомлень для користувачів або спорожнені, або безпечно перенесені в буфер, а нових сесій на вузол більше не надходить. Після цього вузол повністю вилучається з внутрішнього реєстру nodeRegistry та з політики розподілу: формується нова distVersion без згаданого вузла, яка, як і в попередніх кроках, публікується через брокер та зберігається в Redis. На основі цього оновлення сервіс масштабування SocketNodeScalerService ініціює зменшення реплік WebSocket-серверів у Kubernetes, і відповідний pod коректно видаляється з кластера.

Навіть після фізичного видалення pod'а система ще деякий час продовжує обробляти залишкові повідомлення, що могли потрапити в чергу видаленого вузла до моменту зміни конфігурації. Для цього на стороні WebSocket Load-balancer'а працює компонент RemovedNodeMessageForwarder, який тимчасово підписується на відповідні черги брокера, дочитує їхній залишок і переадресовує повідомлення новим власникам ключів згідно з актуальною політикою розподілу. Після очищення черг і завершення relay-періоду форвардинг вимикається, а draining-процедура для цього вузла вважається повністю завершеною.

Описаний алгоритм висушування забезпечує кероване, поетапне згортання кластера без масових одночасних перепідключень і втрати повідомлень. Завдяки поєднанню стану draining, механізмів graceful reconnect, presence-реєстру та адресного relay через брокер система зберігає стабільність затримок і цілісність сесій навіть під час агресивного scale-in, а зміна складу вузлів кластера відбувається для користувачів максимально прозоро.

3.3 Реалізація WebSocket сервера

WebSocket сервер (діаграма класів надана в додатку Д) реалізовано як окремий реактивний мікросервіс на Java, який безпосередньо тримає персистентні підключення клієнтів, доставляє їм події та взаємодіє з брокером повідомлень і Redis-кешем. Сервіс розгортається в окремому Deployment Kubernetes і масштабується горизонтально, а стан кожного підключення зберігається локально на вузлі й дублюється у розподіленому реєстрі присутності.

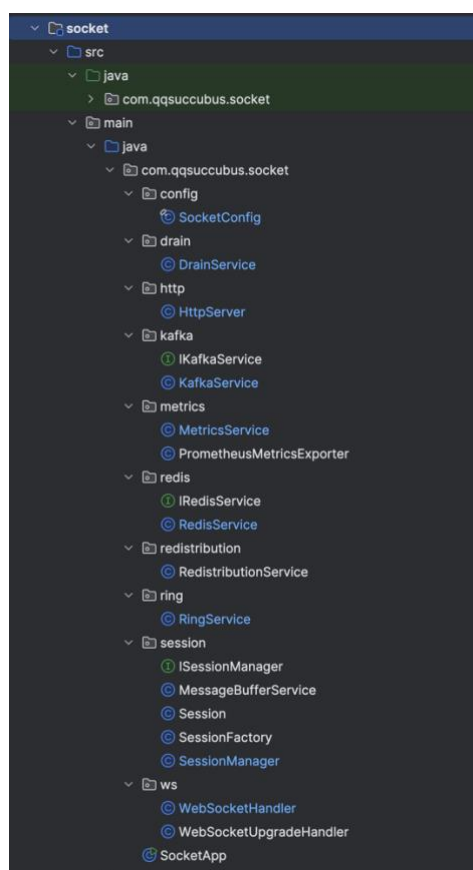


Рисунок 3.2 – Структура проекту модуля WebSocket сервер.

Структурно модуль складається з кількох підпакетів (рис. 3.2). Точка входу SocketApp ініціалізує конфігурацію (config.SocketConfig), створює екземпляри сервісів доступу до Redis і Kafka, сервіс буферизації повідомлень, менеджер сесій, сервіс розподілу, сервіс drain/redistribution, експорт телеметрії та

HTTP/WS-сервер. Після побудови графа залежностей SocketApp запускає HTTP-сервер і реєструє обробник коректного завершення: під час зупинки вузол переводиться в режим осушування, активні сесії поступово завершуються, а невідправлені повідомлення переміщуються в буфер.

Клас `WebSocketUpgradeHandler` приймає HTTP-запит на ендпоінт `WebSocket`-підключення, виконує `upgrade`, витягує з URL і параметрів `clientId`, токен відновлення й інші службові атрибути, після чого передає керування в `WebSocketHandler`. Останній працює поверх `Reactor Netty` й реалізує повністю неблокуючу обробку пакетів: для кожного з'єднання формує реактивний конвеєр читання/запису, розбирає вхідні повідомлення (користувацькі події, керуючі пакети), інкрементує метрики та делегує роботу з сесією в `SessionManager`. Вихідні події для клієнта формуються як реактивний стрім, який зчитує повідомлення з локального буфера/черги та надсилає їх через `WebSocket`, дотримуючись порядку доставки для конкретного клієнтського ключа.

За управління життєвим циклом сесій відповідає пакет `session`. Клас `SessionManager` утримує реєстр активних сесій (об'єкти `Session`), відповідає за їх створення, відновлення, закриття та оновлення стану у `Redis`-реєстрі. Під час нового підключення `SessionManager` визначає, чи це нова сесія, чи відновлення існуючої за токеном і останнім підтвердженням офсетом, у першому випадку створюється новий об'єкт `Session` через `SessionFactory`, у другому – сесія піднімається з метаданих, а непрочитані повідомлення дочитуються з буфера. Для кожної сесії ведеться облік останнього підтвердженого повідомлення й поточного стану (`active`, `draining`, `closed`).

Буферизація повідомлень і робота з відновленням реалізована в `MessageBufferService`. Цей сервіс зберігає для кожного `clientId` послідовність подій у `Redis` (короткоживучі ключі або стріми) та надає `SessionManager`'у операції «записати повідомлення в буфер» і «прочитати непрочитані повідомлення, починаючи з». Таким чином, під час тимчасового розриву з'єднання або переведення сесії на інший вузол, події не втрачаються: клієнт

отримує їх повторно після успішного перепідключення, а бізнес-рівень працює з ідемпотентними повідомленнями.

Пакет `redis` інкапсулює доступ до розподіленого кешу й реєстру присутності. Сервіс `Redis` відповідає за зберігання записів виду «`clientId` – фактичний хост» із коротким `TTL`, за `heartbeat`-записи «`nodeId` – метрики вузла», а також за допоміжні структури, які використовуються для буферизації повідомлень. Завдяки цьому `WebSocket`-сервер може швидко відновити власний стан після рестарту, а інші компоненти (`load-balancer`, інші вузли) – дізнатися, де фізично розміщена конкретна сесія в межах перехідного вікна.

Пакет `ring` реалізує локальну інтерпретацію політики розподілу, що надходить від `WebSocket Load-balancer`'а. Клас `RingService` зберігає поточну версію кільця з вагами вузлів, відновлює її з `Redis` під час старту й оновлює при отриманні нових повідомлень з брокера. На його основі `RedistributionService` періодично перевіряє, чи збігається фактичний власник сесії з тим, кого визначає консистентний хеш для даного `clientId`. Якщо внаслідок зміни ваг або складу кластера «правильним» власником стає інший вузол, `RedistributionService` позначає поточну сесію як таку, що потребує перенесення, і через `SessionManager` ініціює контрольований розрив з'єднання з джитером. Клієнт перепідключається вже до нового вузла, а непрочитані події дочитуються з буфера.

Окремо виділено пакет `drain`, в якому `DrainService` реагує на зміну стану поточного вузла (перехід у `draining`, завершення `drain`-вікна). У режимі `draining` сервер припиняє приймати нові підключення, а `SessionManager` і `RedistributionService` прискорюють поступове виведення сесій з вузла, зберігаючи при цьому гарантії доставки. Завершення `drain`-режиму супроводжується очищенням локальних структур, оновленням записів у `Redis` та коректним закриттям `WebSocket`-серверу.

Нарешті, пакет `metrics` відповідає за спостережуваність. `MetricsService` збирає технічні (`CPU`, пам'ять, кількість відкритих `WebSocket`-з'єднань, розмір

буферів) і доменні метрики (кількість активних клієнтів, швидкість надходження й доставки повідомлень, кількість перепідключень, кількість сесій у стані draining). Ці показники експортуються через PrometheusMetricsExporter у форматі Prometheus і паралельно використовуються для формування heartbeat-повідомлень у Redis. Таким чином WebSocket-сервер не лише обслуговує клієнтів, а й виступає джерелом телеметрії для контуру балансування та масштабування, описаного в попередніх підрозділах.

У сукупності наведені компоненти реалізують реактивний WebSocket-вузол, здатний підтримувати велику кількість персистентних з'єднань, гарантувати відновлення сесій без втрат, інтегруватися з консистентним хешуванням і механізмами drain/scale-in та видавати повний набір доменних метрик для подальшого керування всією системою.

3.4 Розгортання системи в кластері Kubernetes

Розгортання системи виконується в локальному кластері Kubernetes на базі Minikube й максимально наближене до сценарію хмарної експлуатації. Такий підхід дозволяє на одному фізичному вузлі відтворити типову інфраструктуру «Kubernetes - брокер повідомлень - кеш - система моніторингу» та відразу тестувати всі механізми — від балансування персистентних з'єднань до масштабування за доменними метриками.

3.4.1 Підготовка кластера та налаштування файлових дескрипторів

Перший крок — запуск сценарію ініціалізації Minikube, який готує середовище до роботи з великою кількістю WebSocket-з'єднань. На відміну від стандартного сценарію «minikube start», де кластер орієнтовано на декілька десятків pod'ів і типові web-навантаження, у цьому проєкті ціллю є експерименти із сотнями тисяч одночасних TCP/WS-підключень.

Скрипт `minikube-high-conn.sh` запускає Minikube з підвищеними ресурсами вузла (10 vCPU, близько 16 ГБ оперативної пам'яті) та з додатковими параметрами kubelet:

- `kubelet.allowed-unsafe-sysctls` увімкнено для `net.core.somaxconn` та `net.ipv4.tcp_tw_reuse`. Перше значення визначає максимально можливу довжину черги очікуючих TCP-з'єднань (backlog) для сокетів у стані LISTEN. Збільшення `somaxconn` дозволяє WebSocket-серверам одночасно приймати велику кількість нових hand-shake без ситуацій, коли ядро відкидає з'єднання ще до того, як їх опрацює застосунок. Параметр `tcp_tw_reuse` дозволяє повторно використовувати сокети у стані TIME_WAIT, що зменшує кількість «завислих» з'єднань і ризик вичерпання локального пулу портів при масових перепідключеннях;

- `kubelet.max-pods` збільшується до 250, завдяки чому в межах одного вузла можна розгорнути всі компоненти системи (Kafka, Redis, WebSocket-узли, load-balancer, nginx, Prometheus, Grafana та допоміжні сервіси) без штучних обмежень.

Окремо сценарій після старту Minikube перевіряє, що kubelet дійсно підхопив налаштування `allowedUnsafeSysctls`, читаючи його конфігурацію всередині віртуальної машини. Це дає гарантований зворотній зв'язок: якщо параметри не застосовані, розробник побачить попередження ще до спроби розгорнути застосунок.

Паралельно з налаштуванням kubelet передбачається підняття лімітів на файлові дескриптори та сокети всередині контейнерів – як на рівні образу (через `ulimit -n` у стартових скриптах), так і на рівні pod'ів (через `securityContext` і `sysctls`). Для високонавантаженої WebSocket-системи ключовими є три аспекти:

- `nofile` / кількість дескрипторів – кожне активне з'єднання споживає файловий дескриптор, тому ліміт має масштабуватись щонайменше лінійно від цільової кількості підключень;

- діапазон ефемерних портів (`ip_local_port_range`) – розширений до 1024–65535, що дозволяє ядру виділяти достатню кількість локальних портів для клієнтських і проксі-з'єднань;

– backlog на рівні застосунку – WebSocket-сервери конфігуруються так, щоб використовувати збільшене значення `somaxconn`, інакше робота з великими чергами на вході не дасть бажаного ефекту.

Таким чином, ще до розгортання мікросервісів кластер налаштовано не як «типовий dev-кластер», а як середовище, здатне витримувати реалістичні навантаження на мережевий стек і файл-дескриптори.

3.4.2 Розгортання інфраструктури та сервісів застосунку

Другий етап – виконання сценарію `deploy.sh`, який автоматизує послідовне виконання всіх YAML-маніфестів. Список YAML-маніфестів наведено на рисунку 3.3.

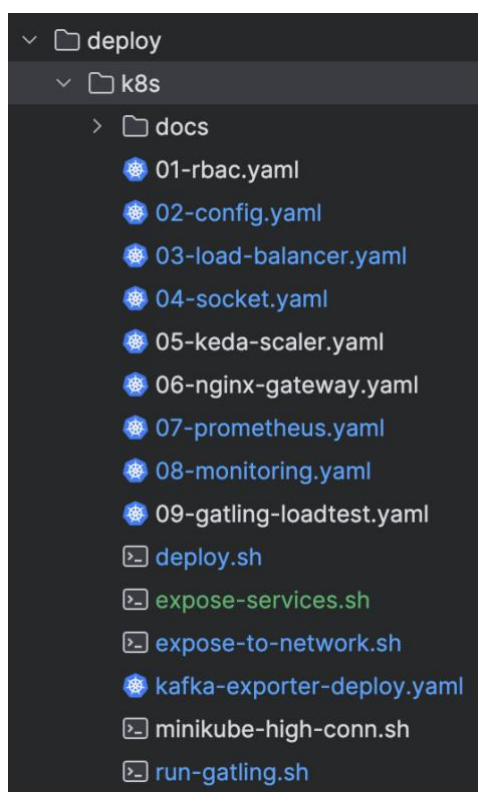


Рисунок 3.3 – Список YAML-маніфестів для розгортання k8s кластеру.

В окремому просторі імен `rtc` створюються:

– RBAC-об’єкти (`01-rbac.yaml`) – облікові записи сервісів і ролі, що дозволяють WebSocket Load-balancer-у зчитувати ConfigMap-и, звертатися до

Kubernetes API (наприклад, для перевірки стану pod'ів) та працювати з CRD-ресурсами KEDA. Це важливо для того, щоб керування масштабуванням та drain-процесами виконувалося з мінімально необхідними, але не завищеними правами;

- конфігураційні об'єкти (02-config.yaml) – ConfigMap-и та Secret-и, в яких зберігаються адреси Kafka, Redis, часові параметри drain-вікна, ім'я кластеру, цільові значення SLO/SLA, таймаути WebSocket та HTTP-проксі. Всі чутливі значення (паролі, ключі, сертифікати) винесено у Secret-и, що дозволяє при потребі замінювати їх без перезбірки образів;

- Kafka-кластер (03-load-balancer.yaml / окремі маніфести Strimzi) – брокер повідомлень розгортається в режимі KRaft з реплікацією журналів. Обрані налаштування орієнтовані на високий throughput і низьку затримку: збільшені розміри мережевих буферів, кількість ІО-потоків, налаштування партицій, що дозволяють рівномірно розподіляти трафік між брокерами;

- Redis-кластер – використовується як presense-реєстр та буфер повідомлень. Для нього задаються ліміти пам'яті, політики eviction (наприклад, allkeys-lru для буферів короткого життя) та параметри реплікації, щоб уникнути єдиної точки відмови.

Після підготовки інфраструктури сценарій перемикає Docker-клієнт на вбудований Minikube, збирає два основні образи (reactive-rtc-load-balancer та reactive-rtc-socket) і вбудовує їх безпосередньо в кластер.

3.4.3 Конфігурація nginx-шару як edge-шлюзу

Вхідною точкою в кластер виступає NGINX, розгорнутий як Gateway/Ingress (06-nginx-gateway.yaml). Обрано саме nginx з кількох причин: зріла підтримка WebSocket-проксі, стабільна інтеграція з Kubernetes, можливість детально налаштовувати таймаути, ліміти з'єднань та логування.

У налаштуваннях серверного блоку передбачено:

- підтримку протоколу WebSocket шляхом прокиду HTTP заголовків Upgrade і Connection, використання proxy_http_version 1.1 та збільшення proxy_read_timeout/proxy_send_timeout до значень, достатніх для довготривалих сесій;

- збільшений worker_connections та, відповідно, ліміт одночасних з'єднань на процес nginx, щоб edge-шар не ставав точкою відмови при масовому підключенні клієнтів;

- налаштування keep-alive: оптимальні значення keepalive_timeout і keepalive_requests дають змогу зменшити кількість повторних TCP-handshake між nginx та WebSocket Load-balancer-ом, тим самим знижуючи накладні витрати на встановлення з'єднання;

- обмеження на розмір тіла запиту (client_max_body_size) та максимальну кількість одночасних з'єднань з одного IP – як базовий захист від помилок клієнтів і простих DoS-сценаріїв.

З боку Kubernetes nginx представлено через Service типу LoadBalancer, а для локальної розробки сценарій deploy.sh автоматично налаштовує port-forward на localhost:8080 для спрощення тестування, щоб можна бул підключитися до системи та виконувати WebSocket-handshake за звичайною URL без додаткової мережевої конфігурації.

3.4.4 Система моніторингу - Prometheus і Grafana

Щоб підтвердити досягнення цільових SLO і коректну роботу алгоритмів балансування й масштабування, у кластері розгорнуто повноцінну систему спостережуваності.

Маніфести 07-prometheus.yaml і 08-monitoring.yaml створюють:

- Prometheus – як центральний агрегатор метрик. Для нього задається окремий ConfigMap із правилами scrape, що охоплюють усі ключові групи сервісів: nginx, WebSocket Load-balancer, WebSocket-сервери, Kafka, Redis та

компоненти KEDA. Інтервали scrape підібрані так, щоб з одного боку достатньо точно фіксувати зміни p95/p99-latency та навантаження, а з іншого – не створювати зайвого трафіку всередині кластера. Також задаються retention-параметри, які дозволяють зберігати історію метрик у межах, достатніх для аналізу навантажувальних експериментів, але не переповнюють локальний диск Minikube;

- recording і alerting-правила – розрахунок агрегаційних метрик (p95-latency по всіх WebSocket-вузлах, середній Kafka lag, кількість активних з’єднань на вузол) та базові алерти (перевищення порогів latency, недоступність pod’ів, аномально висока кількість помилок). Це дозволяє швидко виявляти деградації під час експериментів;

- Grafana – інтерфейс для візуалізації. У кластер завантажується готовий дашборд, який відображає основні SLI системи: кількість активних WebSocket-з’єднань, throughput повідомлень, p95/p99-latency для критичного шляху «клієнт – nginx – Load-balancer – WebSocket-вузол – Kafka – доменний сервіс – WebSocket-вузол – клієнт», завантаження CPU/RAM для ключових компонентів, а також поточні значення ваг вузлів та distVersion. Доступ до Grafana організовано через локальний port-forward на localhost:3000, що дозволяє аналізувати експерименти без додаткових проксі.

Prometheus одночасно виступає джерелом метрик для KEDA: спеціальні ScaledObject-и періодично виконують запити до Prometheus на основі цих значень ухвалюють рішення про scale-out або scale-in.

3.4.5 Завершення розгортання та навантажувальне тестування

Після застосування всіх маніфестів deploy.sh очікує, поки основні pod’и перейдуть у стан Ready, і виконує пробний HTTP-запит до API веб-сокет балансувальника (/api/v1/resolve?clientId=test-user) через nginx-шар. Це дозволяє

переконалися, що шлях «nginx – Load-balancer – WebSocket-кластер – Redis/Kafka» коректно працює ще до запуску повномасштабного тестування.

Для моделювання користувацького навантаження передбачено окремий маніфест 09-gatling-loadtest.yaml, який розгортає Gatling-агента в тому ж кластері. Такий підхід дозволяє:

- уникнути впливу нестабільного зовнішнього каналу між генератором навантаження та кластером;
- точно контролювати профіль сесій (тривалість, частоту повідомлень, сценарії перепідключень) і порівнювати результати між різними експериментами;
- автоматично збирати накопичувальні звіти, які потім зіставляються з метриками Prometheus і використовуються в аналітичній частині роботи.

У підсумку розгортання системи зводиться до двох кроків: запуску сценарію підготовки Minikube та виконання deploy.sh. Внаслідок цього на локальній машині розгортається повноцінний еквівалент хмарного середовища з налаштованими дескрипторами, edge-шлюзом nginx, брокером повідомлень, Redis-реєстром, WebSocket Load-balancer-ом, кластером WebSocket-серверів і системою моніторингу. Така конфігурація дозволяє не лише розробляти й відлагоджувати окремі компоненти, а й проводити комплексні навантажувальні експерименти, що наближені до реальних умов експлуатації.

3.5 Висновки до розділу

У цьому розділі було показано, як запропонована архітектура реалізована у вигляді конкретного набору мікросервісів, розгорнутих у кластері Kubernetes. Система розбита на логічні підсистеми: edge-шар на базі nginx, кластер WebSocket Load-balancer'ів, кластер WebSocket-серверів, Redis як реєстр станів і буфер повідомлень, Kafka як надійний брокер подій, а також контур спостережуваності й масштабування (Prometheus, Grafana, KEDA, Kubernetes

API). Така декомпозиція дозволяє незалежно масштабувати й оновлювати окремі компоненти, а також чітко розмежовує відповідальність між прийманням трафіку, балансуванням, зберіганням стану й доменною обробкою.

Окрему увагу приділено реалізації WebSocket Load-balancer'а. Він поєднує детермінований алгоритм консистентного хешування (skeleton-based weighted rendezvous hashing) із періодичним перерахунком ваг на основі технічних і доменних метрик. У розділі описано, як Load-balancer формує внутрішній реєстр вузлів, виявляє зміни топології кластера, приймає рішення про scale-out/scale-in та розповсюджує єдину «версію істини» щодо розподілу навантаження (distVersion) між усіма учасниками системи. Завдяки цьому балансування залишається стабільним і передбачуваним навіть за умов динамічних змін кількості WebSocket-вузлів.

Запропонований алгоритм висушування (draining) веб-сокет вузлів забезпечує контрольоване згортання кластера без масових одночасних перепідключень. При переході вузла в стан draining його вага обнуляється в політиці розподілу, нові підключення більше не спрямовуються на цей вузол, а наявні сесії поступово виводяться через механізми відновлення сесій без втрат, реєстр станів та адресний relay через брокер повідомлень. Додатковий компонент RemovedNodeMessageForwarder гарантує дочитування залишкових повідомлень із каналів видалених вузлів. Сукупно це дозволяє виконувати scale-in без втрати повідомлень і без суттєвих стрибків затримки.

4 МОДЕЛЮВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ

4.1 Експериментальне оточення

У цьому розділі описано, як була побудована експериментальна інфраструктура для моделювання роботи системи та як на її основі були підібрані фінальні вагові коефіцієнти (формула 2.4) для композитного бала навантаження вузлів. Також запропоноване рішення було порівняно з традиційними підходами (використанні брокера повідомлень для широкомовного пересилання повідомлень та використання централізованого реєстру станів для визначення вузла підключення користувача).

Для моделювання використовувався той самий кластер Kubernetes, що описаний у розділі 3: два вузли WebSocket-серверів (2GB RAM, 2 CPU), один WebSocket Load-balancer (1Gb RAM, 1 CPU) та декілька реплік nginx-проху. Автоматичне масштабування під час калібрування ваг вимикалося, щоб не ускладнювати інтерпретацію результатів: нас цікавило саме розподілення навантаження між socket-вузлами при фіксованій топології.

Навантаження генерувалося за допомогою окремої Gatling-job в кластері. Скрипт (run-gatling.sh) дозволяє параметризувати кількість одночасних клієнтів, час плавного «розігріву» (ramp-up), тривалість тесту та інтервал між повідомленнями кожного клієнта.

```
user@MacBook-Pro k8s % ./run-gatling.sh -n bob-a -c 15000 -r 10 -d 12 -i 20 &
./run-gatling.sh -n lem-b -c 15000 -r 10 -d 12 -i 20 &
./run-gatling.sh -n koa-c -c 15000 -r 10 -d 12 -i 20 &
wait

[1] 13105
[2] 13106
[3] 13107
```

Gatling Load Test - Kubernetes	
Job Name:	gatling-bob-a
Clients:	15000
Ramp-up:	10 minutes
Duration:	12 minutes
Interval:	20 seconds
Keep Alive:	false

Рисунку 4.1 – Профіль підсумкового експерименту.

Для підсумкового експерименту з каліброваними коефіцієнтами використовувався профіль наведений на рисунку 4.1. Тобто в системі одночасно працювало до 45 000 клієнтів з однаковим сценарієм: плавний розгін протягом 10 хвилин, тривалість 12 хвилин, кількість повідомлень на секунду – 300.

З боку кластеру всі компоненти експортували метрики в Prometheus, а окремі скрипти `export_metrics.sh` та `analyze_metrics.py` періодично вивантажували метрики у вигляді snapshot'ів та будували звіти, де кожен звіт містив:

- конфігурацію коефіцієнтів;
- статистику збалансованості (`stddev` та `spread CPU і пам'яті`);
- реальний внесок кожної компоненти в підсумковий `loadScore`;
- затримки обробки WebSocket-повідомлень та Kafka-lag;
- сумарний баланс-бал (0 – 1), який відображає наскільки «чесно» розподілене навантаження між двома вузлами.

4.2 Перебір коефіцієнтів і аналіз внеску метрик

Підбір фінальних коефіцієнтів відбувався поетапно й спирався не на разові спостереження, а на систематичний аналіз результатів декількох десятків навантажувальних запусків. Для кожного набору коефіцієнтів α_{cpu} , α_{mem} , α_{lat} , α_{kafka} , α_{conn} використовувався однаковий профіль тесту згаданий в розділі 4.1.

Кожен тестовий запуск давав повний зріз поведінки системи: час стабілізації ваг і кількості підключень, дисбаланс по ресурсах, розподіл затримок (p95/p99), Kafka-lag, частку локальних / relay-доставок, частоту перерахунку ваг.

На першому етапі тестування було проведено грубий перебір конфігурацій. Спочатку був проведений грубий пошук по сітці коефіцієнтів. Перевірялися типові «сімейства»:

- ресурс-центричні значення: CPU + Mem = 0.6–0.7, решта метрик – допоміжні;
- latency/lag-центричні: Latency + KafkaLag = 0.6–0.7, CPU/Mem/Conn – мінімальні;
- connection-центричні: Conn = 0.6–0.7, решта метрик – 0.1–0.15;
- рівномірні: усі коефіцієнти 0.2.

Для кожного варіанта виконувався повноцінний Gatling-запуск, після чого `analyze_metrics.py` формував звіт з агрегованими показниками для обох вузлів. На цьому етапі завданням було не знайти ідеальні числа, а відфільтрувати явно непридатні області параметрів – наприклад, такі, де:

- один вузол стабільно отримує на 20–30 % більше клієнтів, ніж інший;
- спостерігається періодичне перекачування клієнтів між вузлами без необхідності;
- Kafka-lag зростає до тисяч повідомлень, попри невисоке використання CPU/пам'яті.

Після запуску тестів стало видно, що:

- resource-центричні конфігурації добре тримають CPU/Mem, але гірше вирівнюють кількість з'єднань;
- latency/lag-центричні конфігурації у нормальних умовах майже не впливають на поведінку – latency та Kafka-lag далекі від SLO, їх нормалізовані значення малі, а внесок у loadScore виявився на рівні < 1–2 % навіть при великих коефіцієнтах;
- connection-центричні варіанти давали найкращий баланс за кількістю сесій, але за надто великої ваги Conn система ставала «сліпою» до CPU/пам'яті та могла підвантажувати вузол, у якого ресурси вже близькі до межі;
- рівномірно розділені коефіцієнти показали, що кількість з'єднань впливає на фінальний бал значно вище за інші коефіцієнти. На рисунку 4.2 зображені результати тестування рівних коефіцієнтів, та їхній реальний вплив на бал навантаження.

Coefficient Contributions			
How much each coefficient actually contributed to the load score:			
Component	Mean Score	Contribution %	Expected %
CPU	0.0485	8.2%	20%
Memory	0.0500	8.5%	20%
Latency	0.1454	24.7%	20%
Kafka Lag	0.1466	24.9%	20%
Connections	0.1981	33.6%	20%

Рисунку 4.2 – Результати тестування рівних коефіцієнтів.

Таким чином, подальший пошук зосередили навколо таких діапазонів:

- Conn знаходиться в діапазоні 0.4–0.6;
- Mem – у межах 0.15–0.25;
- CPU – 0.05–0.2;
- Latency та KafkaLag – по 0.05–0.15 кожна.

Далі йшов другий етап – формування показників якості для кожної конфігурації. На другому етапі для кожної протестованої конфігурації коефіцієнтів α_i автоматично обчислювався набір агрегованих показників якості:

- дисбаланс по з'єднаннях: середня кількість підключень на вузол, стандартне відхилення та відносний розрив $(\max - \min) / \text{avg}$. Це був основний критерій, оскільки вузол із більшою кількістю сесій споживає більше пам'яті й файл дескрипторів;

- дисбаланс по CPU та пам'яті: середнє та p95 значень CPU і RAM; spread між вузлами (наскільки один вузол навантаженіший за інший).

- стан latency та Kafka-lag: p95/p99 WebSocket-latency, середній Kafka-consumer-lag і сумарний об'єм черги (інтеграл $\text{lag}(t)$ за час тесту); p99 затримки по Kafka-topic (latency доставки від брокера до вузла);

- поведінка алгоритму балансування: як часто відбувався перерахунок ваг, чи були періоди «гойдалок», коли ваги вузлів змінюються й перерозподіляють

сесії безпідставно, як швидко система виходить на стабільний стан після старту (час до моменту, коли дисбаланс по з'єднаннях падає нижче фіксованого порогу);

– ефективність доставлення повідомлень: частка локальних доставок проти переадресації повідомлення на інший вузол, мережевий трафік (щоб уникнути варіантів, де баланс досягається ціною вибуху трафіку).

Для зручності порівняння ці показники нормалізувалися до інтервалу $[0;1]$ (по відношенню до найгірших / найкращих значень у всіх експериментах), після чого для кожної конфігурації будувався інтегральний «quality score», де:

- найбільшу вагу мав дисбаланс по з'єднаннях;
- середню – дисбаланс по пам'яті та CPU;
- додаткові штрафи накладалися за високий Kafka-lag, погіршення p95/p99-latency.

На третьому етапі відбулося звуження області пошуку та локальна оптимізація. Після обробки всіх звітів стало видно, що в «верхній частині» рейтингу (тобто з мінімальним quality score) опинилися конфігурації з такими властивостями:

- Conn – завжди не менше 0.4 і не більше 0.6;
- Mem – між 0.15 і 0.25;
- CPU – 0.05–0.15;
- Latency і KafkaLag – у діапазоні 0.05–0.15 кожна.

Конфігурації з надто малим Conn (≤ 0.3) мали гірший дисбаланс по з'єднаннях, із надто малим Mem – періодично допускали суттєвий перекид по пам'яті, із надто великим CPU – реагували на короткі піки й іноді створювали коливання ваг.

На цьому етапі було проведено локальну оптимізацію: для кількох перспективних «центральных» точок (Conn=0.4/0.5/0.6; Mem=0.15/0.2/0.25) проганявся повторний набір тестів, і порівнювався їх quality score та окремі

показники (особливо баланс з'єднань і Kafka-lag). Саме в цих повторних запусках найкраще себе показала конфігурація:

- $\alpha_{\text{cpu}} = 0.10$;
- $\alpha_{\text{mem}} = 0.20$;
- $\alpha_{\text{lat}} = 0.10$;
- $\alpha_{\text{conn}} = 0.50$;
- $\alpha_{\text{kafka}} = 0.10$.

З такими коефіцієнтами було досягнуто:

- мінімальний/близький до мінімального дисбаланс за кількістю з'єднань;
- стабільно низький Kafka-lag (як середній, так і сумарний);
- хорошу рівномірність по CPU/пам'яті;
- швидкий вихід на стабільний стан після старту.

Іншими словами, ця конфігурація не лише мала один із кращих інтегральних quality score, а й виглядала «здоровою» за всіма окремими графіками в Grafana: криві з'єднань сходилися до 50/50, ваги вузлів згладжувалися, latency та lag залишались далекі від порогів SLO, а кількість перерахунків ваг була помірною. В таблиці 4.3 представлено результати тестів з різними коефіцієнтами. Через аналіз результатів таблиці – вибір коефіцієнтів згаданих вище є виправданим.

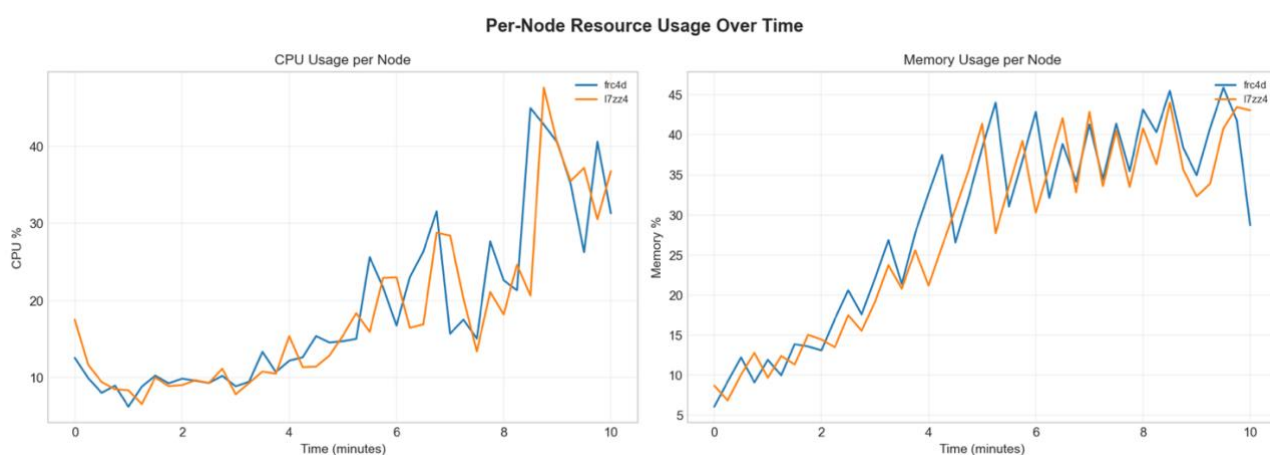
Таблиця 4.3 – Результати тестів з різними коефіцієнтами.

Метрики	T1	T2	T3	T4	T5	T6
Коефіцієнт (CPU/Mem /Lat/Kafka /Conn)	0.35/0.35/ 0.15/0.10/ 0.05	0.20/0.20 /0.20/0.2 0/0.20	0.05/0.05 /0.33/0.3 3/0.33	0.15/0.35/ 0.05/0.05/ 0.40	0.15/0.15/ 0.05/0.05/ 0.60	0.10/0.20 /0.10/0.1 0/0.50
Balance Score	0.93	0.92	0.93	0.94	0.93	0.94
CPU Spread	0.0237	0.0330	0.0212	0.0381	0.0541	0.0383

Продовження таблиці 4.3

Метрики	T1	T2	T3	T4	T5	T6
Mem Spread	0.3034	0.3401	0.3016	0.2327	0.2885	0.2380
Avg CPU	21.0%	27.1%	19.2%	18.1%	19.4%	18.4%
Avg Mem	31.5%	30.0%	30.9%	27.1%	28.2%	28.4%
WS p95 ms	1.59	4.31	1.41	1.49	1.94	1.36
Kafka p95 ms	4.99	8.56	4.08	4.13	6.15	3.82

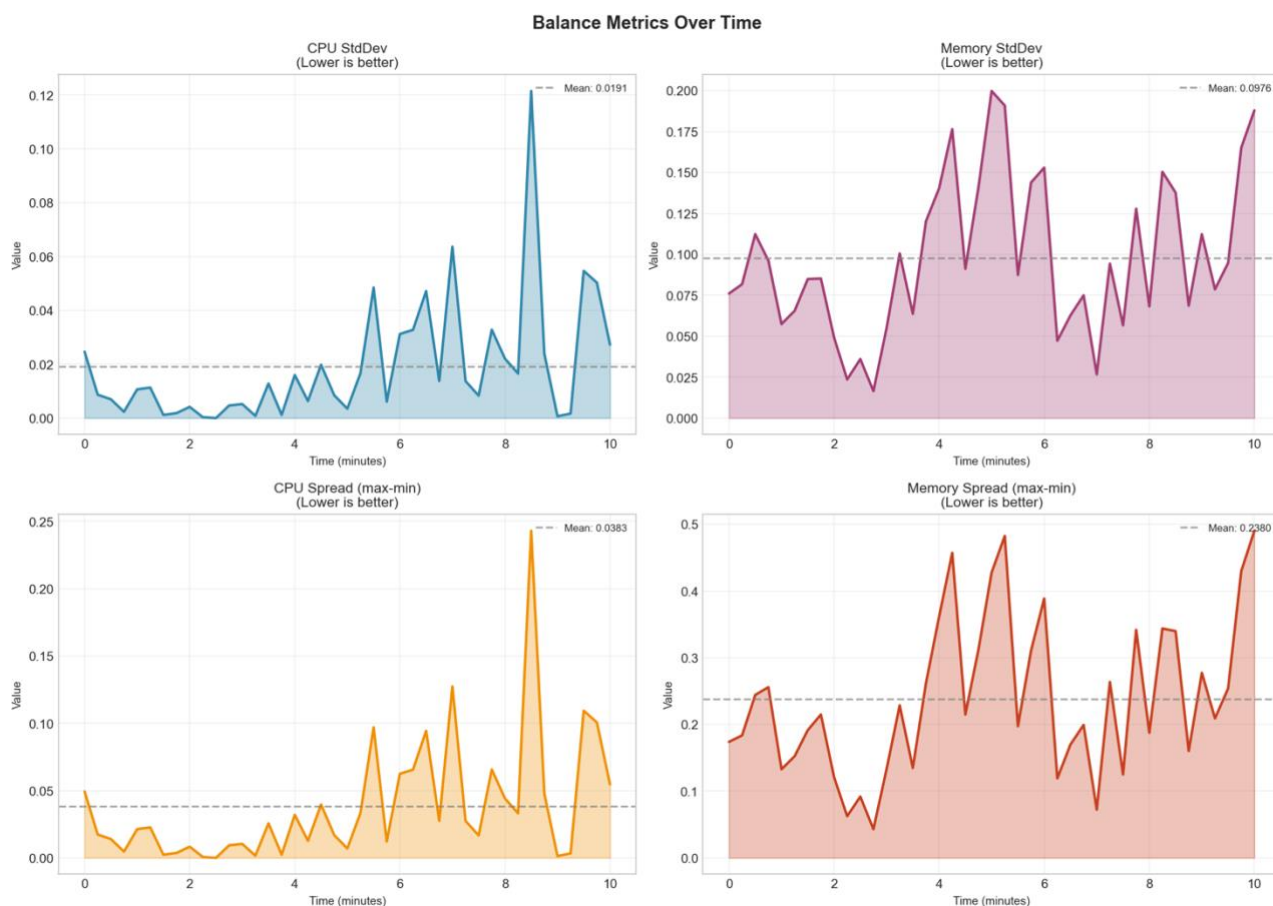
На рисунку 4.4 зображено порівняльні графіки використання ресурсів двома веб-сокет вузлами при обраних коефіцієнтах ваг, як ми бачимо – розбіжність в споживанні ресурсів помітно мала.



Рисунку 4.4 – Результати тестування рівних коефіцієнтів.

На рисунку 4.5 зображено графіки дисперсії та різниці між min/max використанням CPU/MEM двома веб-сокет вузлами при обраних коефіцієнтах ваг, як видно CPU здебільшого збалансований (середнє відхилення 1.9%,

min/max різниця – 3.8%) із не частими стрибками, а пам'ять має значно більші та частіші перекося (середнє відхилення 10%, min/max різниця – 23.8%), тому якщо потрібно вирівнювання, варто підсилити вагу пам'яті або зменшити навантаження, з локальних state/buffer сховищ на окремих вузлах.



Рисунку 4.5 – Графіки дисперсії та різниці між min/max використанням CPU/MEM веб-сокет вузлами.

Аргументація обраних коефіцієнтів:

– домінування структурного навантаження. 50 % ваги припадає на кількість з'єднань. Це гарантує, що навіть за ідеальних метрик CPU/latency алгоритм не дозволить одному вузлу накопичити істотно більше сесій, ніж іншому. У двовузловому сценарії це приводить до швидкої конвергенції до майже 50/50 розподілу клієнтів;

– підсилений контроль пам'яті. 20 % для пам'яті виявилися достатніми, щоб виправити недоліки конфігурацій, де memory-spread був «червоним». У попередніх варіантах із малими коефіцієнтами для Mem стандартне відхилення по пам'яті було високим, попри гарний баланс по CPU; збільшення ваги пам'яті дозволило зблизити її поведінку з кількістю з'єднань без втрати стабільності;

– резерв 10 % для CPU дозволяють алгоритму реагувати на вузли, де з'являються CPU-піки через бізнес-обробку, але не роблять CPU критерієм номер один. У практичних експериментах це проявлялося в тому, що іноді саме CPU-компонента «переконувала» алгоритм трохи зменшити вагу вузла, навіть коли кількість з'єднань була близькою до рівноважної. В загальному 10% має бути достатньо так як веб-сокет з'єднання в більшості часу знаходяться в неактивному користуванні, через що пам'ять стає більш важливою;

– Latency та Kafka-lag – кожна з цих метрик отримала по 10 %. У нормальному режимі, коли p95-latency й Kafka-lag займають лише декілька відсотків від SLO-бюджету, їх внесок у loadScore залишається невеликим. Але при зростанні лагів вони починають формувати суттєву частину бала, змушуючи алгоритм зменшувати вагу проблемного вузла навіть тоді, коли з'єднання та пам'ять ще виглядають здоровими. Це дає додаткову «подушку безпеки» проти деградацій, які спершу проявляються саме в затримках.

У сукупності ця конфігурація забезпечила найкращий компроміс між рівномірним розподілом сесій, стійкістю до піків та чутливістю до деградації latency, яку також показали наступні експерименти.

4.3 Порівняння запропонованого рішення з традиційними підходами

Щоб оцінити ефективність розробленого рішення з вагами обраними з розділу 4.2 – було проведено порівняльне тестування із двома традиційними архітектурами:

– широкомовні повідомлення через брокер (broadcast). Кожне повідомлення одразу відправляється до всіх WebSocket-вузлів, а вже на вузлі фільтрується за цільовим клієнтом;

– повна маршрутизація через реєстр станів (presence-registry). Перед відправкою кожного повідомлення виконується звернення до Redis-реєстру, щоб дізнатися, на якому вузлі зараз знаходиться потрібний clientId, і повідомлення публікується лише на цей вузол.

Для всіх трьох варіантів використовувався однаковий профіль навантаження та однакова конфігурація кластера (2 socket-node, 1 WS-LB, nginx-проху).

Результати запуску тесту з використанням розробленого рішення представлені в таблиці 4.6.

Таблиця 4.6 – Результати запуску тесту з використанням розробленого рішення.

Показник	Значення	Коментар
Коефіцієнти (CPU / Mem / Lat / Lag / Conn)	0.10 / 0.20 / 0.10 / 0.10 / 0.50	Підсумковий варіант, обраний за результатами експериментів
Використання пам'яті	350–450 МБ	Стабільне споживання пам'яті
Середнє завантаження CPU	≈ 15 %	Без різких стрибків
Загальна кількість повідомлень	~356 тис.	Сумарно за час тесту
Пропускна здатність WebSocket	~300 MPS	Повідомлення/с через WS

Продовження таблиці 4.6

Показник	Значення	Коментар
Пропускна здатність Kafka relay	~150 MPS	Повідомлення/с через relaykafka)
Частка локальної доставки	~66 %	Більшість повідомлень без relay (доставлені локально)
Частка доставки через relay	~34 %	Тільки для міжвузлових сценаріїв
Kafka consumer lag (повідомлень/с)	≈ 41	Середній лаг по швидкості накопичення
Kafka consumer lag (сумарний), повідомлень	≈ 450	Невелика черга
Середня затримка Kafka-topic	≈ 4 мс	Дуже низька середня затримка
p99 затримки Kafka-topic	≈ 420 мс	Пікові значення залишаються в прийнятних межах
Вхідний мережевий трафік	≈ 419 КБ/с	Враховуючи всі WS/Kafka взаємодії
Вихідний мережевий трафік	≈ 420 КБ/с	Симетричний профіль трафіку (відносно вхідного)

Результати запуску тесту з надсиланням широкомовних повідомлень через брокер повідомлень представлені в таблиці 4.7.

Таблиця 4.7 – Результати запуску тесту з надсиланням широкомовних повідомлень через брокер повідомлень.

Показник	Значення	Коментар
Тип маршрутизації	Broadcast через брокер	Кожне повідомлення надсилається на всі вузли
Використання пам'яті	400–450 МБ	Трохи вище ніж у розробленому методі
Середнє завантаження CPU	$\approx 17\%$	Вище, ніж у розробленому методі
Максимальне завантаження CPU	до $\sim 35\%$	Відчутні піки при навантаженні
Загальна кількість повідомлень	~ 353 тис.	Схожий обсяг трафіку
Пропускна здатність WebSocket	~ 300 MPS	Аналогічно до розробленого методу
Пропускна здатність Kafka relay	~ 150 MPS	Аналогічно до розробленого методу
Частка локальної доставки	$\sim 66.5\%$	Не відчутна різниця
Частка доставки через relay	$\sim 33.5\%$	
Kafka consumer lag (повідомлень/с)	≈ 105	Лаг суттєво більший
Kafka consumer lag (сумарний), повідомлень	≈ 1300	Утричі більший, ніж у розробленого методу

Продовження таблиці 4.7

Показник	Значення	Коментар
Середня затримка Kafka-topic	≈ 8.2 мс	Вище вдвічі ніж у розробленому методі
p99 затримки Kafka-topic	≈ 619 мс	Гірший розподіл ніж у розробленому методі
Вхідний мережевий трафік	≈ 558 КБ/с	Зростає лінійно з кількістю socket-вузлів
Вихідний мережевий трафік	≈ 427 КБ/с	Вищий, ніж при таргетованій доставці
Додаткова особливість		Не масштабується ефективно при $N > 2$, дублювання трафіку

Результати запуску тесту з використанням реєстру станів для маршрутизації повідомлень представлені в таблиці 4.8.

Таблиця 4.8 – Результати запуску тесту з використанням реєстру станів для маршрутизації повідомлень.

Показник	Значення	Коментар
Тип маршрутизації	Presence-registry	Визначення вузла за записом у Redis
Використання пам'яті	380–450 МБ	Між broadcast'ом і розробленим методом
Середнє завантаження CPU	≈ 16.5 %	Трохи вище, ніж у розробленого методу

Продовження таблиці 4.8

Показник	Значення	Коментар
Максимальне завантаження CPU	$\approx 31.7 \%$	Менше піків, ніж у broadcast-підходу
Загальна кількість повідомлень	~ 355 тис.	Порівнюваний обсяг
Пропускна здатність WebSocket	~ 250 MPS	Нижчий MPS через Redis-lookup
Пропускна здатність Kafka relay	~ 125 MPS	Також нижча ніж у розробленого методу
Частка локальної доставки	$\sim 66.6 \%$	Не відчутна різниця
Частка доставки через relay	$\sim 33.4 \%$	
Kafka consumer lag (повідомлень/с)	≈ 59	Краще, ніж у broadcast, але гірше, ніж у розробленого методу
Kafka consumer lag (сумарний), повідомлень	≈ 650	Середнє між другим і першим підходами
Середня затримка Kafka-topic	≈ 7.3 мс	Гірше, ніж у розробленого методу
p99 затримки Kafka-topic	≈ 461 мс	Трохи гірше, ніж у розробленого методу
Вхідний мережевий трафік	≈ 391 КБ/с	Менший за broadcast

Кінець таблиці 4.8

Показник	Значення	Коментар
Вихідний мережевий трафік	≈ 355 КБ/с	Менше ніж у розробленому методі, може свідчити про нищу пропускну здатність
Додаткове навантаження на Redis		Постійні запити до реєстру-станів. Додає затримку і створює ще одну точку відмови

Таким чином, запропонований метод:

- забезпечує найменший Kafka-lag (час від моменту надсилання повідомлення в брокер повідомлень, до моменту поки повідомлення буде доставлено цільовому клієнту) (рисунок 4.9), як за середнім значенням, так і за сумарним обсягом черги;
- демонструє кращі затримки доставки (середня затримка – 4 мс проти 8.2 мс і 7.3 мс у альтернатив);
- зберігає порівняно низьке завантаження процесора (CPU) і пам'яті (рисунок 4.10), оскільки не генерує зайвих копій повідомлень (як при широкомовному надсиланні повідомлень) і не звертається до Redis (реєстру-станів) при надсиланні кожного повідомлення;
- має помірний мережевий трафік, який не зростає лінійно з кількістю вузлів.

Отже, в обраній конфігурації саме поєднання консистентного хешування і композитного бала з домінуванням кількості активних підключень дає найкращий компроміс між продуктивністю, використанням ресурсів і якістю доставки.



Рисунку 4.9 – Графіки затримок і черг брокера повідомлень під час тесту.



Рисунку 4.9 – Графіки використання CPU/MEM веб-сокет вузлами.

4.4 Тестування стабілізації розподілу при змінах топології

Окремий набір експериментів був присвячений перевірці здатності алгоритму стабілізувати розподіл сесій при зміні топології. Для цього запускалися два WebSocket-сервери зі зсунутим у часі стартом:

– перший вузол починав приймати з’єднання ще до старту другого і до моменту його появи в кластері встигав накопичити істотно більше клієнтів.

– після запуску другого вузла Load-balancer включав його в розрахунок ваг, але на перших ітераціях ваги суттєво відрізнялись, а розподіл сесій був далеким від 50/50.

На графіках Grafana (рисунок 4.11) було видно, як на початку перший вузол мав більшу вагу й значно більше підключень, оскільки був у системі під час старту навантаження, після кількох циклів перерахунку алгоритм почав систематично піднімати вагу другого вузла (який демонстрував менше завантаження) і знижувати вагу першого.



Рисунок 4.11 – Графік розподілу з’єднань при асиметричному запуску вузлів.

Поступово нерівні кількості з’єднань для обох вузлів зближувалися й у підсумку практично збігалися, а ваги вузлів стали однаковими (рисунок 4.12) .



Рисунок 4.13 – Графік стабілізації ваги вузлів.

Важливо, що при цьому не спостерігалось коливань або «гойдалок»: зміна ваг була плавною, без різких стрибків, а кількість перепідключень клієнтів залишалася помірною. Це означає, що: з одного боку, обрана конфігурація коефіцієнтів досить чутлива, щоб реагувати на суттєвий дисбаланс, а з іншого боку, вона не переоцінює короточасні піки й не провокує нестабільну поведінку (наприклад, періодичне перекладання більшості сесій з одного вузла на інший).

Такий експеримент додатково підтвердив, що розроблений композитний бал з експериментально визначеними вагами працює не лише для стаціонарних сценаріїв, а й для динамічних ситуацій, коли до кластера додаються або вилучаються вузли.

4.5 Висновки до розділу

У цьому розділі було побудовано й досліджено повноцінне експериментальне оточення. На базі локального кластера Kubernetes із двома WebSocket-вузлами, одним WebSocket Load-balancer-ом та nginx-проксі було змодельовано сценарій з десятками тисяч одночасних клієнтів і стабільним потоком повідомлень. Завдяки інтеграції з Prometheus і автоматизованому експорту метрик вдалося отримати детальний зріз поведінки системи: від розподілу навантаження між вузлами до затримок доставки й стану черг брокера повідомлень.

На основі серії навантажувальних експериментів з різними наборами вагових коефіцієнтів для композитного бала навантаження (CPU, пам'ять, latency, Kafka-lag, кількість з'єднань) було показано, що підбір параметрів не можна звести до інтуїтивного налаштування лише однієї метрики. Початковий грубий перебір (resource-центричні, latency-центричні, connection-центричні та рівномірні конфігурації) продемонстрував:

- надмірна орієнтація на CPU/пам'ять погано контролює розподіл сесій;
- підвищені ваги latency/Kafka-lag мало впливають на поведінку в звичайному режимі, але можуть викликати зайву чутливість до коротких піків навантаження;
- занадто великий пріоритет кількості з'єднань робить систему сліпою до ресурсних обмежень вузлів.

Подальший аналіз показників якості (balance score, spread та stddev по CPU/пам'яті, Kafka-lag, latency, частка локальних доставок, частота перерахунку ваг) дозволив звужити область пошуку та виявити, що найкращі результати дають конфігурації, де кількість з'єднань домінує в композитному балі, але CPU та пам'ять залишаються суттєвими коригуючими факторами, а latency та Kafka-lag виконують роль страхувальних сигналів деградації. У підсумку оптимальною виявилася комбінація $\alpha_{cpu} = 0.10$, $\alpha_{mem} = 0.20$, $\alpha_{lat} = 0.10$, $\alpha_{conn} = 0.50$, $\alpha_{kafka} = 0.10$, яка забезпечила найкращий компроміс між:

- рівномірним розподілом сесій;
- збалансованим використанням CPU і пам'яті;
- низьким Kafka-lag та хорошими p95/p99-latency;
- стабільною поведінкою без «гойдалок» у перерахунку ваг.

Порівняння запропонованого рішення з двома базовими архітектурами – широкомовною розсилкою через брокер повідомлень і централізованим реєстром станів – показало, що поєднання консистентного хешування з композитним бальним алгоритмом дає відчутні переваги. У порівнянні з broadcast-підходом запропонований метод:

- зменшує Kafka-lag як за середнім значенням, так і за сумарною довжиною черги;
- забезпечує нижчу середню й p99-затримку доставки;
- не множить трафік пропорційно кількості вузлів і тим самим краще масштабується при збільшенні кластера.

У порівнянні з підходом, що повністю спирається на реєстр станів, розроблене рішення:

- усуває постійну залежність від Redis на кожне повідомлення, не створює додаткової точки відмови;
- забезпечує вищу пропускну здатність (MPS) при близьких ресурсних витратах;
- дає менший Kafka-lag та кращі показники затримки без надмірного мережевого чи обчислювального навантаження.

Окремі експерименти зі зміною топології (асиметричний запуск двох WebSocket-вузлів) підтвердили, що алгоритм з обраними коефіцієнтами не лише працює добре в стаціонарному режимі, а й коректно поводить себе в динамічних сценаріях. Вузол, який стартував пізніше, поступово отримував більшу вагу, поки кількість з'єднань не вирівнювалась, після чого ваги стабілізувались без різких стрибків і масових перепідключень клієнтів. Це свідчить про достатню чутливість алгоритму до істотного дисбалансу та водночас про його стійкість до короткочасних тригерів.

У підсумку, проведене моделювання та експериментальна оцінка підтверджують, що запропонований підхід до балансування персистентних з'єднань — із використанням skeleton-based weighted rendezvous hashing і композитного навантажувального бала забезпечує високу якість розподілу, низькі затримки та більш ефективне використання ресурсів у порівнянні з традиційними архітектурними рішеннями.

5 РОЗРОБЛЕННЯ СТАРТАП ПРОЄКТУ

5.1 Ідея стартап проєкту

Стартап-проєкт передбачає створення реактивної платформи для маршрутизації WebSocket-повідомлень з адаптивним балансуванням навантаження, консистентним хешуванням та Kubernetes-нативним масштабуванням. Платформа дозволяє обслуговувати мільйони персистентних з'єднань при стабільних p95/p99-затримках, автоматично перерозподіляти трафік між вузлами та виконувати перепідключення без втрат. В основі рішення є зважений rendezvous-хешинг – метод консистентного хешування, дворівневий 2-hop роутинг – що дозволяє знайти цільовий вузол отримувача, навіть при розсинхронізації даних між вузлами, реєстр-станів у Redis (як резервний метод), що використовується лише, коли повідомлення не вийшло доставити після 2-hop роутинга, брокер повідомлень (Kafka) для доставки повідомлень між вузлами та доменно-орієнтовані метрики для автоматичного масштабування (кількість активних підключень, кількість повідомлень, що оброблюються сервером на секунда, p95-затримка доставки повідомлень, затримку черг повідомлень).

На відміну від керованих хмарних сервісів на кшталт Amazon WebSocket API чи Ably, а також бібліотек, що дають змогу легко створити власне рішення для роботи з WebSocket з'єднаннями, на кшталт Socket.IO, запропонована платформа орієнтована на self-hosted-сценарії в Kubernetes-кластерах, де команді потрібен повний контроль над маршрутизацією, політиками масштабування та вартістю інфраструктури, команда розробників напряму може вирішити, як рішення буде масштабуватись та який рівень затримок повинен бути підтриманий. Система мінімізує міжвузловий трафік, не потребує централізованого реєстру сесій (лише як резервний варіант) і забезпечує передбачувану поведінку під високим навантаженням, що критично для фінтех-, IoT- та біржових сервісів.

Таблиця 5.1 – Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Реактивна платформа керованих WebSocket-з'єднань із детермінованим балансуванням та автоскейлінгом за доменними метриками, орієнтована на self-hosted-розгортання в Kubernetes.	- B2B-сервіси реального часу (біржові платформи, фінтех-додатки, системи моніторингу та алертингу). - Масштабовані сервіси нотифікацій, чат-та колабораційні платформи. - IoT-шлюзи та телеметрія з великою кількістю одночасних підключень. -SaaS продукти, які нині використовують Amazon WebSocket API, Ably	- Керована собівартість: можливість розгорнути платформу у власному або гібридному кластері без жорсткої прив'язки до конкретного хмарного провайдера (менше vendor lock-in порівняно з Amazon WebSocket API / Ably). - Вища прогнозованість під навантаженням: консистентне хешування й доменні метрики дозволяють уникати перевантаження окремих вузлів і зберігати стабільні p95/p99-затримки при піках трафіку, що складно досягти в типових сценаріях Socket.IO з простим round-robin. - Мінімальний міжвузловий трафік завдяки точній маршрутизації; це зменшує витрати на мережу й прискорює доставку повідомлень. - Висока відмовостійкість, що забезпечує безвтратну доставку та швидке відновлення після збоїв. - Простота експлуатації: використання стандартних механізмів Kubernetes, збір метрик у Prometheus/Grafana та прозорі політики масштабування.

5.2 Порівняння з конкурентами

Для оцінки ринкових перспектив стартап-проєкту доцільно порівняти його з найближчими аналогами: Amazon WebSocket API, Ably та Socket.IO. За підсумком порівняння для кожної характеристики визначено, чи є вона для нашого проєкту слабкою (W), нейтральною (N) або сильною (S) стороною.

Таблиця 5.2 – Оцінка характеристик проєкту

№	Технічні характеристики	Продукція конкурентів			W	N	S
		Мій проєкт	Amazon WebSocket API	Ably			
1	Кастомне балансування/маршрутизація (weighted rendezvous, affinity)	Є, повний контроль	Маршрути через інтеграції, без LB-логіки	Канали/presence, без кастом-LB			+
2	Масштабування за власними метриками (CPU, mem, conn, p95)	K8s HPA/KEDA, свої метрики	Кероване сервісом, обмежена кастомізація	Кероване провайдером			+
3	Гарантії доставки/історія/повторна відправка	Базово (ACK/Redis за потреби)	Best-effort, без історії	Історія, idempotency, QoS	+		

Продовження таблиці 5.2

№	Технічні характеристики	Продукція конкурентів			W	N	S
		Мій проєкт	Amazon WebSocket API	Aby			
4	Георозподіл/за тримки	Залежить від нашого розгортання	Регіональні ендпоінти	Глобальна мережа PoP	+		
5	Вартість на великому трафіку	лінійне масштабування	Оплата за повідомлення /хвилини	Тарифи за повідомлення/канали			+
6	Vendor lock-in / контроль даних	Мінімальний (self-host/K8s)	Високий	Високий			+
7	Спостережуваність/метрики	Prometheus/Alerting, повні трейси	CloudWatch, базово	Вбудована аналітика, менше кастому			+
8	Час виходу у прод / швидкий старт	Потрібна інфраструктура	Дуже швидко	Дуже швидко	+		

5.3 Технологічний аудит ідеї проєкту

Технологічний аудит показує, чи може ідея платформи для балансування та маршрутизації WebSocket-з'єднань бути реалізована з використанням наявних технологій, а також чи є ці технології доступними авторам проєкту.

Ідея передбачає побудову реактивної, Kubernetes-нативної платформи, що забезпечує масштабоване оброблення WebSocket-трафіку, консистентне хешування користувачів, використання брокера подій та кешу стану, а також автоматичне масштабування на основі доменних метрик.

Таблиця 5.3 – Технологічна здійсненність ідеї проєкту

№ п/п	Ідея проєкту	Технології реалізації	Наявність технологій	Доступність технологій
1	Платформа оброблення WebSocket-з'єднань	WebSocket поверх HTTP(S), TLS, Java 21 (Netty / Spring WebFlux), nginx / Ingress	Стандартизовані технології, є зрілі бібліотеки й приклади	Open-source, безкоштовні; частина вже використовувалась у навчальних проєктах
2	Реактивна обробка трафіку та балансування сесій	Java Reactive Streams, Project Reactor, алгоритми консистентного та зваженого rendezvous-хешування	Реактивні фреймворки й алгоритми добре описані, існують open-source реалізації	Доступні бібліотеки; є прототип реалізації, який може бути доопрацьований
3	Зберігання стану, heartbeat-ів та кешування	Redis (key–value, TTL, pub/sub), клієнти для Java	Redis є зрілою технологією, масово використовується для кешів і сесій	Open-source, готові Docker-образи; автори мають базовий досвід роботи з Redis

Продовження таблиці 5.3

№ п/п	Ідея проекту	Технології реалізації	Наявність технологій	Доступність технологій
4	Оркестрація, автоскейлінг та моніторинг платформи	Docker, Kubernetes, HPA/KEDA, Prometheus, Grafana	Інструменти де-факто стандарт у промислових системах, добре задокументовані	Можна розгорнути локально або в хмарі; автори вже працювали з Docker та базовим Kubernetes

5.4 Аналіз ринкових можливостей запуску стартап-проекту

Далі було проведено попередню характеристику ринку для нашого стартап-проекту системи кастомного балансування WebSocket-з'єднань. На основі аналізу відкритих даних та огляду рішень основних провайдерів хмарних сервісів у таблиці 5.4 узагальнено ключові параметри ринку, що дозволяє оцінити загальну привабливість ринку для запуску проекту.

Таблиця 5.4 – Оцінка характеристик проекту

№ п/п	Показник	Характеристика ринку для проекту
1	Кількість головних гравців	Олігополія: 5–7 глобальних лідерів (Twilio, AWS); нішеві та open-source рішення (Socket.IO тощо)
2	Орієнтовний обсяг ринку	CPaaS 14,2 млрд дол. США (2024); прогноз 62,7 млрд дол. США (2033)
3	Динаміка ринку	Високі темпи зростання; CAGR 18–28 %; драйвери: IoT, FinTech, telehealth
4	Основні бар'єри входу	Високі технічні бар'єри (low latency, high availability); значні інвестиції; вимоги SLA 99,99 %

Продовження таблиці 5.4

№ п/п	Показник	Характеристика ринку для проєкту
5	Норми / стандарти	Протоколи: RFC 6455, HTTP/2, QUIC; комплаєнс: GDPR, SOC 2, HIPAA; безпека: mTLS, end-to-end шифрування
6	Середня рентабельність галузі	Висока рентабельність SaaS; валова маржа 50–60 %; вища за середні депозитні ставки

Аналіз показує, що ринок є привабливим для входження завдяки високій динаміці зростання (~18-28% на рік) та значній нормі рентабельності (понад 50%). Однак, наявність сильних гравців-олігополістів та високі технологічні бар'єри вимагають від стартап-проєкту чіткої диференціації, як-от кастомне балансування навантаження, self-hosted модель та оптимізація витрат на інфраструктуру, що є слабким місцем хмарних гігантів.

Таблиця 5.5 – Характеристика потенційних клієнтів стартап-проєкту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці клієнтів	Вимоги споживачів
1	Миттєвий обмін даними (чат, нотифікації, стріми)	SaaS-сервіси, fintech, online-ігри, провайдери LLM	Чутливість до затримок, робота 24/7	Низька латентність, стабільні WebSocket-з'єднання
2	Масштабування real-time трафіку	Високонавантажені продукти, маркетплейси	Пікові навантаження, часті релізи	Горизонтальне масштабування, підтримка Kubernetes

Продовження таблиці 5.5

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці клієнтів	Вимоги споживачів
3	Контроль витрат і уникнення vendor lock-in	Компанії, що вже користуються AWS/Ably	Порівнюють TCO, прагнуть більшого контролю	Прозорі витрати, можливість self-hosted розгортання
4	Спостережуваність та керуваність сервісу	DevOps, SRE, backend-команди	Постійний моніторинг та швидка реакція	Інтеграція з Prometheus, детальні метрики та логування, можливість перегляду графіків в Grafana

5.5 Фактори загроз та можливостей

Після визначення цільових груп клієнтів було проаналізовано ринкове середовище з погляду можливих загроз і можливостей для стартап-проекту системи маршрутизації та балансування WebSocket-з'єднань.

З боку ринку існує кілька суттєвих загроз. Найперше – це висока конкуренція з боку великих глобальних провайдерів рішень реального часу (зокрема, Amazon WebSocket API, Ably тощо), які вже мають розвинену інфраструктуру, впізнаваний бренд та широку клієнтську базу. Це ускладнює залучення перших користувачів і змушує запропонувати помітно відмінну ціннісну пропозицію. Другою загрозою є швидка еволюція хмарних сервісів: великі гравці можуть відносно швидко додати схожий функціонал (кастомні метрики, розумний роутинг, інтеграції з Kubernetes) і нівелювати частину

конкурентних переваг. Ще один ризик пов'язаний із залежністю від інфраструктури хмарних провайдерів і Kubernetes-екосистеми: зміни в тарифах, політиках безпеки чи технічних обмеженнях можуть збільшити собівартість рішення або вимагати значних переробок. Для зниження впливу цих загроз можна фокусуватись на вузькій ніші високонавантажених real-time-систем, робити акцент на гнучкій архітектурі (multi-cloud / on-prem), прозорій вартості володіння та глибокій технічній підтримці для інженерних команд, а також максимально швидко валідувати гіпотези з реальними клієнтами.

Паралельно ринок створює низку можливостей для розвитку проєкту. Зростає попит на системи, що забезпечують масштабовану двосторонню взаємодію в реальному часі (ігрові платформи, трейдинг, колаборативні інструменти, онлайн-події, AI), при цьому компанії часто стикаються з обмеженнями «типових» managed-рішень і шукають більш гнучкі, кастомізовані варіанти. Це відкриває можливість зайняти нішу «інженерно-орієнтованого» продукту з прозорою метрикою продуктивності та можливістю тонкого налаштування. Додатковою можливістю є посилення вимог до контролю за даними та їх розміщенням: частина клієнтів віддає перевагу гібридним або on-prem-рішенням, де власна команда повністю контролює трафік і безпеку. Стартап може використати це, пропонуючи архітектуру, яка однаково добре працює в публічній хмарі, приватному кластері чи змішаному середовищі. Нарешті, зростає інтерес до спостережності (observability) та data-driven-масштабування: компанії прагнуть приймати інфраструктурні рішення на основі метрик p50/p95-затримок, завантаження CPU/пам'яті та кількості активних з'єднань. Запропонована система, що базується на збиранні та аналізі цих метрик для розумного балансування WebSocket-з'єднань, може перетворити цю тенденцію на ключову конкурентну перевагу.

Таким чином, ринкове середовище поєднує високий рівень ризику через силу глобальних гравців зі значними можливостями для спеціалізованого,

технічно глибокого продукту, який закриває потреби команд, що працюють з критично важливим real-time-трафіком.

5.6 Ступеневий аналіз конкуренції на ринку

На ринку рішень для обробки трафіку в реальному часі конкуренція має кілька вимірів.

Ринок належить до олігополістичного: основну частку займають кілька глобальних провайдерів (AWS, Ably, інші CPaaS-рішення), доповнені нішевими й open-source продуктами. Це означає високу роль бренду, екосистеми та «залипання» клієнтів у вибраного провайдера. Щоб бути конкурентоспроможною, компанія має чітко відрізнитись функціоналом (гнучкі метрики, розумний роутинг, multi-cloud/on-prem), сервісом і прозорою вартістю, а також демонструвати технічні переваги.

Конкуренція переважно міжнародна: клієнти можуть обрати будь-яке хмарне рішення, незалежно від країни. Це підвищує вимоги до якості продукту, надійності та відповідності міжнародним стандартам безпеки. Реакцією компанії має стати орієнтація на глобальний ринок: англомовний продукт і підтримка, інтеграції з популярними хмарними платформами, гнучка політика ціноутворення в різних регіонах.

Переважає міжгалузева конкуренція: за ті самі бюджети клієнт може обирати не лише інші WebSocket-платформи, а й альтернативні архітектури (наприклад, long-polling, server-sent events, готові SaaS-платформи для чату чи колаборації). Тому продукт повинен показувати очевидну вигоду саме від вибору WebSocket-архітектури з «розумним» балансуванням: кращу продуктивність, стабільність під навантаженням і нижчу повну вартість володіння.

Конкуренція є і товарно-родовою (між різними рішеннями для real-time-комунікацій), і товарно-видовою (між конкретними WebSocket-/RTS-платформами). Серед «між бажаннями» клієнт може взагалі обрати інший спосіб

вирішити бізнес-потребу (наприклад, асинхронні оновлення замість live-оновлень). Це спонукає компанію чітко формулювати ціннісну пропозицію: показувати, які саме бізнес-метрики покращує перехід на кероване WebSocket-рішення (менше відмов, швидша реакція, кращий досвід користувачів).

Основна конкуренція відбувається за неціновими параметрами: надійність, масштабованість, затримки, зручність інтеграції, аналітика й підтримка. Ціна важлива, але рідко є єдиним критерієм. Відповідно, компанії слід інвестувати в технічну якість, UX для розробників (SDK, документація, dashboards) та сервіс, використовуючи цінову політику скоріше як інструмент стимулювання входу (фріміум, пілоти).

Сильні бренди хмарних провайдерів часто обирають «за замовчуванням». Щоб зменшити цю перевагу великих гравців, стартап має будувати власний бренд як експерта з високонавантажених WebSocket-систем: публікувати технічні статті, кейси, результати перформанс-тестів, брати участь у профільних конференціях і спільнотах розробників. Це допоможе сформувати довіру й мотивувати клієнтів тестувати рішення навіть за наявності «зручного» стандартного варіанту від великої платформи.

Для детальнішої оцінки конкурентного середовища проєкт системи розумного балансування WebSocket-з'єднань проаналізовано за моделлю 5 сил М. Портера.

Таблиця 5.6 – Аналіз конкуренції в галузі за М. Портером

Складова аналізу	Основні гравці / фактори	Висновки
Прямі конкуренти в галузі	AWS API Gateway (WebSocket), Ably Realtime, PubNub та інші CPaaS-провайдери real-time-комунікацій.	Висока інтенсивність конкуренції з боку глобальних гравців; потрібна чітка нішева спеціалізація та помітні технічні переваги.

Продовження таблиці 5.6

Складова аналізу	Основні гравці / фактори	Висновки
Потенційні конкуренти	Інші хмарні провайдери й платформи, що можуть додати схожий функціонал; великі компанії, які створюють власні внутрішні рішення.	Бар'єри входу технологічні та репутаційні; загроза з боку великих хмар помірно висока, з боку нових стартапів – середня.
Постачальники	Хмарні провайдери (AWS, GCP, Azure тощо), Kubernetes-інфраструктура, провайдери баз даних і черг повідомлень.	Сила постачальників середня: є залежність від тарифів і політик хмар, але використання multi-cloud / on-prem архітектури дозволяє зменшити ризики.
Клієнти	B2B-клієнти з високим real-time-навантаженням: ігрові платформи, трейдинг, колаборативні сервіси.	Сила клієнтів середня–висока: технічно підковані замовники легко порівнюють альтернативи, очікують прозорих цін.
Товари-замінники	Альтернативні технології комунікації: long-polling, Server-Sent Events, періодичне опитування через REST, спеціалізовані SaaS-сервіси для чату чи колаборації.	Загроза замінників середня: для простих сценаріїв клієнти можуть обрати інші підходи, однак для low-latency high-load real-time-трафіку WebSocket-рішення залишаються більш придатними й ефективними.

На основі аналізу ринку та моделі 5 сил М. Портера можна зробити висновок, що вихід на ринок рішень для real-time-комунікацій можливий, але супроводжується високою інтенсивністю конкуренції з боку глобальних гравців.

Щоб бути конкурентоспроможним, стартап має зайняти чітку нішу високонавантажених WebSocket-систем і запропонувати відчутні технічні переваги, орієнтовані на інженерні команди.

5.7 Висновки до розділу

У цьому розділі було сформовано та проаналізовано стартап-ідею платформи для керованого балансування та маршрутизації WebSocket-з'єднань, побудованої на принципах консистентного хешування, реактивної обробки та Kubernetes-нативного автоскейлінгу. На відміну від керованих хмарних сервісів (Amazon WebSocket API, Ably) та фреймворків рівня Socket.IO, проєкт орієнтовано на self-hosted сценарії з повним контролем над інфраструктурою, політиками масштабування та витратами. Запропонована архітектура використовує напрацювання попередніх розділів (weighted rendezvous hashing, 2-hop-роутинг, presence-реєстр, drain/scale-in, метрики p95/p99 тощо) і демонструє, що ці технічні рішення можуть бути «упаковані» у продукт із чіткою ринковою цінністю для B2B-клієнтів.

Технологічний аудит показав, що всі ключові компоненти ідеї — реактивний WebSocket-шар, розподілене зберігання стану в Redis, брокер подій Kafka, Kubernetes-нативне розгортання та спостережуваність на базі Prometheus/Grafana — є зрілими й доступними технологіями. Це зменшує технологічні ризики й підтверджує здійсненність переходу від навчального проєкту до продукту стартап-рівня. Порівняння з конкурентами засвідчило, що сильні сторони запропонованого рішення лежать у площині кастомного балансування, контролю витрат (self-hosted / hybrid), мінімізації міжнодового трафіку та глибокої інтеграції зі стандартними DevOps-інструментами, тоді як слабшою стороною лишається відсутність глобальної інфраструктури та бренду на рівні великих CPaaS-провайдерів.

Аналіз ринку показав поєднання високих бар'єрів входу (олігополія глобальних гравців, вимоги до низьких затримок і високої доступності) із суттєвими можливостями для нішевого, технічно «глибокого» продукту, орієнтованого на інженерні команди, що працюють з критичним real-time-трафіком і хочуть уникнути vendor lock-in. Модель 5 сил М. Портера вказує на значну силу наявних постачальників і прямих конкурентів, але водночас підтверджує, що саме в сегменті high-load WebSocket-систем із прозорою TCO, гнучким роутингом і повною спостережуваністю є простір для спеціалізованого рішення. У підсумку можна зробити висновок, що запропонований стартап-проект є як технологічно здійсненним, так і ринково доцільним за умови фокусування на чітко визначеній ніші та подальшого розвитку прототипу до продуктового рівня (SDK, документація, сценарії розгортання, підтримка multi-cloud / on-prem).

ВИСНОВКИ

У цій роботі було запропоновано, реалізовано та експериментально оцінено архітектуру високонавантаженої системи оброблення WebSocket-трафіку, орієнтованої на роботу з сотнями тисяч і потенційно мільйонами персистентних з'єднань. Ключовою ідеєю стало поєднання зваженого rendezvous-хешування, композитного навантажувального бала вузлів і Kubernetes-нативного масштабування, що дозволило відмовитися від централізованого реєстру сесій у критичному шляху маршрутизації й мінімізувати міжвузловий трафік. Запропонована схема забезпечує детерміноване призначення клієнтів до вузлів, зберігає локальність обробки, а при зміні топології (scale-out/scale-in, падіння вузла) спирається на поєднання консистентного хешування, реєстру-станів та механізмів відновлення сесій без втрат і drain (осушення), що дає змогу уникати масових одночасних перепідключень і втрати повідомлень.

Важливою науково-практичною складовою роботи став розроблений композитний бал навантаження вузлів, у якому поєднано апаратні показники (CPU, пам'ять), мережеві та доменні метрики (p95-latency, лаг черги повідомлень, кількість активних з'єднань). На основі серії навантажувальних експериментів з різною конфігурацій коефіцієнтів, аналізу балансу по ресурсах, затримок доставки та стабільності ваг було обґрунтовано вибір фінального набору ваг. Обрана конфігурація продемонструвала найкращий компроміс між рівномірним розподілом сесій, низьким лагом черги повідомлень, помірним використанням CPU/пам'яті та відсутністю значних коливань при перерахунку ваг. Додаткові експерименти із асиметричним запуском вузлів показали, що алгоритм здатен плавно стабілізувати розподіл з'єднань і ваг навіть у динамічних сценаріях зміни топології.

Порівняння з традиційними підходами – широкомовним розсиланням повідомлень через брокер та повною маршрутизацією через централізований

реєстр станів – підтвердило переваги розробленого рішення. Запропонований метод забезпечив менший лаг черги повідомлень (як за середнім значенням, так і за сумарною чергою), кращі p95/p99-затримки доставки, відсутність дублювання трафіку на всі вузли та нижче навантаження на CPU й мережу. Широкомовний-підхід виявився простим у реалізації, але погано масштабується при зростанні кількості вузлів через лінійне збільшення трафіку й черг у брокері. Варіант із постійним зверненням до реєстру станів дав кращий контроль над маршрутизацією, але створив додаткову точку відмови й накладні витрати на мережу й CPU, що знижують пропускну здатність. На цьому фоні поєднання консистентного хешування, локального стану на вузлі, резервного реєстру-станів «на випадок відхилень» і композитного бала з домінуванням кількості з'єднань продемонструвало найбільш збалансовані результати.

Отримані результати мають не лише дослідницьке, а й прикладне значення. На базі реалізованого прототипу було сформульовано концепцію стартап-проєкту – реактивної self-hosted платформи для керованих WebSocket-з'єднань у Kubernetes, орієнтованої на B2B-сегмент високонавантажених real-time-систем. Технологічний аудит показав, що всі необхідні компоненти (Java/Project Reactor, Redis, Kafka, Kubernetes, Prometheus/Grafana) є зрілими й доступними, а ринковий аналіз – що існує нішевий попит на рішення, які поєднують гнучке кастомне балансування, прозору вартість володіння та відсутність жорсткого vendor lock-in. Це відкриває можливість еволюції прототипу у комерційний продукт, орієнтований на інженерні команди, що працюють із критичним WebSocket-трафіком.

У перспективі розвиток роботи може відбуватися в кількох напрямках: розширення моделі масштабування за рахунок прогнозних та RL-підходів, інтеграція георозподілених кластерів з урахуванням мережових затримок між регіонами, формальна перевірка властивостей системи під час агресивних scale-in/scale-out сценаріїв, а також подальша оптимізація буферизації та семантики доставки (exactly-once / transactional messaging) для найбільш вимогливих

доменів. У сукупності це дає підстави стверджувати, що запропонована архітектура та реалізовані механізми балансування й масштабування можуть стати основою для створення нового покоління комунікаційних платформ реального часу, здатних стабільно працювати з мільйонами одночасних з'єднань, підтримувати низьку затримку в пікові періоди й ефективно використовувати обчислювальні ресурси.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Reactive Manifesto (2014). The Reactive Manifesto [Електронний ресурс] // Режим доступу: <https://www.reactivemanifesto.org/>
2. Kamat, A. (2025). Scalable Architectures for Real-Time Notification Systems in Distributed Environments [Електронний ресурс] // Режим доступу: https://iaeme.com/MasterAdmin/Journal_uploads/IJITMIS/VOLUME_16_ISSUE_2/IJITMIS_16_02_005.pdf
3. Chavan, A. (2021). Exploring Event-Driven Architecture in Microservices – Patterns, Pitfalls and Best Practices [Електронний ресурс] // Режим доступу: <https://ijsra.net/sites/default/files/IJSRA-2021-0166.pdf>
4. Zbarcea, A., et al. (2024). Migrating from Developing Asynchronous Multi-Threading Programs to Reactive Programs in Java. Applied Sciences [Електронний ресурс] // Режим доступу: <https://www.mdpi.com/2076-3417/14/24/12062>
5. Matteussi, K. J., et al. (2022). Performance Evaluation Analysis of Spark Streaming Backpressure for Data-Intensive Pipelines. Sensors [Електронний ресурс] // Режим доступу: <https://www.mdpi.com/1424-8220/22/13/4756>
6. Eze, K., et al. (2022). SecIoTComm: An Actor-Based Model and Framework for Secure IoT Communication. Sensors [Електронний ресурс] // Режим доступу: <https://pubmed.ncbi.nlm.nih.gov/36236412>
7. Bacchiani, L., et al. (2025). Proactive Reactive Microservice Architecture Global Scaling. Journal of Systems and Software [Електронний ресурс] // Режим доступу: <https://www.saveriogiallorenzo.com/publications/jss2025/jss2025.pdf>
8. Lee, S., et al. (2022). EdgeX over Kubernetes: Enabling Container Orchestration in EdgeX. Applied Sciences [Електронний ресурс] // Режим доступу: <https://www.mdpi.com/2076-3417/12/1/140>
9. Gavrilova, I., et al. (2025). Design of a Distributed Component-Based Framework Using the Actor Model and Back-Pressure for Reactive Stream Processing

[Электронный ресурс] // Режим доступа:

<https://www.researchgate.net/publication/390691713>

10. Netflix Technology Blog (2023). Pushy to the Limit: Evolving Netflix's WebSocket Proxy for the Future [Электронный ресурс] // Режим доступа:

<https://netflixtechblog.com/pushy-to-the-limit-evolving-netflixs-websocket-proxy-for-the-future-b468bc0ff658>

11. Dyte Engineering (2022). Scaling WebSockets to Millions [Электронный ресурс] // Режим доступа: <https://dyte.io/blog/scaling-websockets-to-millions/>

12. Bergé, J. L. F. (2023). Tackling the “Awkward Squad” for Reactive Programming: The Actor–Reactor Model [Электронный ресурс] // Режим доступа:

<https://www.researchgate.net/publication/371758827>

13. Fuentes, M., et al. (2024). A Reliable Architecture Based on Reactive Micro-services for IoT Applications [Электронный ресурс] // Режим доступа:

<https://www.researchgate.net/publication/336422292>

14. Guerra, R., et al. (2024). Analysing the Performance Impact of Reactive Relational Database Connections on Micro-services [Электронный ресурс] // Режим доступа:

<https://lume.ufrgs.br/bitstream/handle/10183/285401/001241300.pdf>

15. IETF BESS Working Group (2023). Weighted HRW and its Applications [Электронный ресурс] // Режим доступа: <https://www.ietf.org/archive/id/draft-ietf-bess-weighted-hrw-00.html>

<https://www.ietf.org/archive/id/draft-ietf-bess-weighted-hrw-00.html>

16. Huijben, I. A. M., Kool, W., Paulus, M. B., & van Sloun, R. J. G. (2021). A Review of the Gumbel-Max Trick and Its Extensions for Discrete Stochasticity in Machine Learning [Электронный ресурс] // Режим доступа:

<https://arxiv.org/abs/2110.01515>

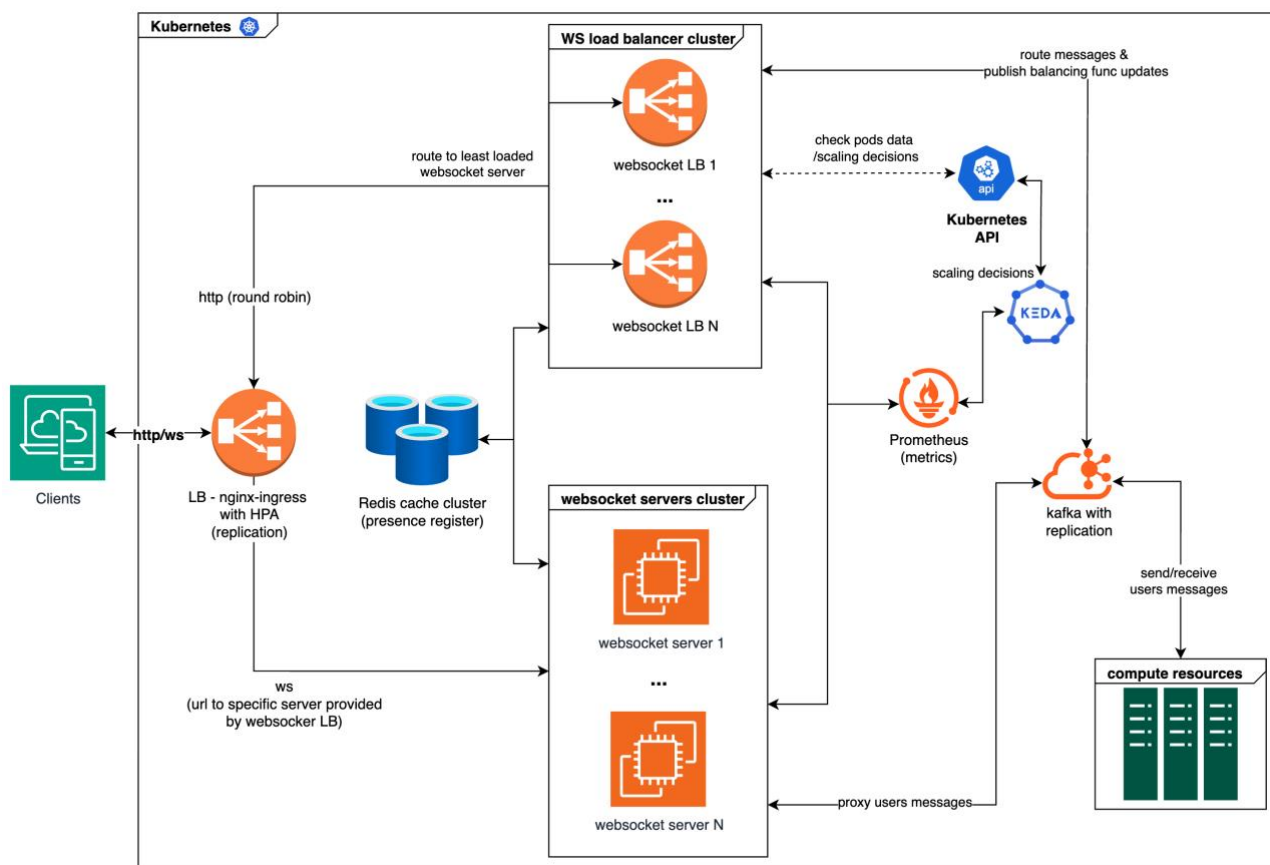
17. HAProxy Technologies (2020). Client IP Persistence or Source IP Hash Load Balancing [Электронный ресурс] // Режим доступа:

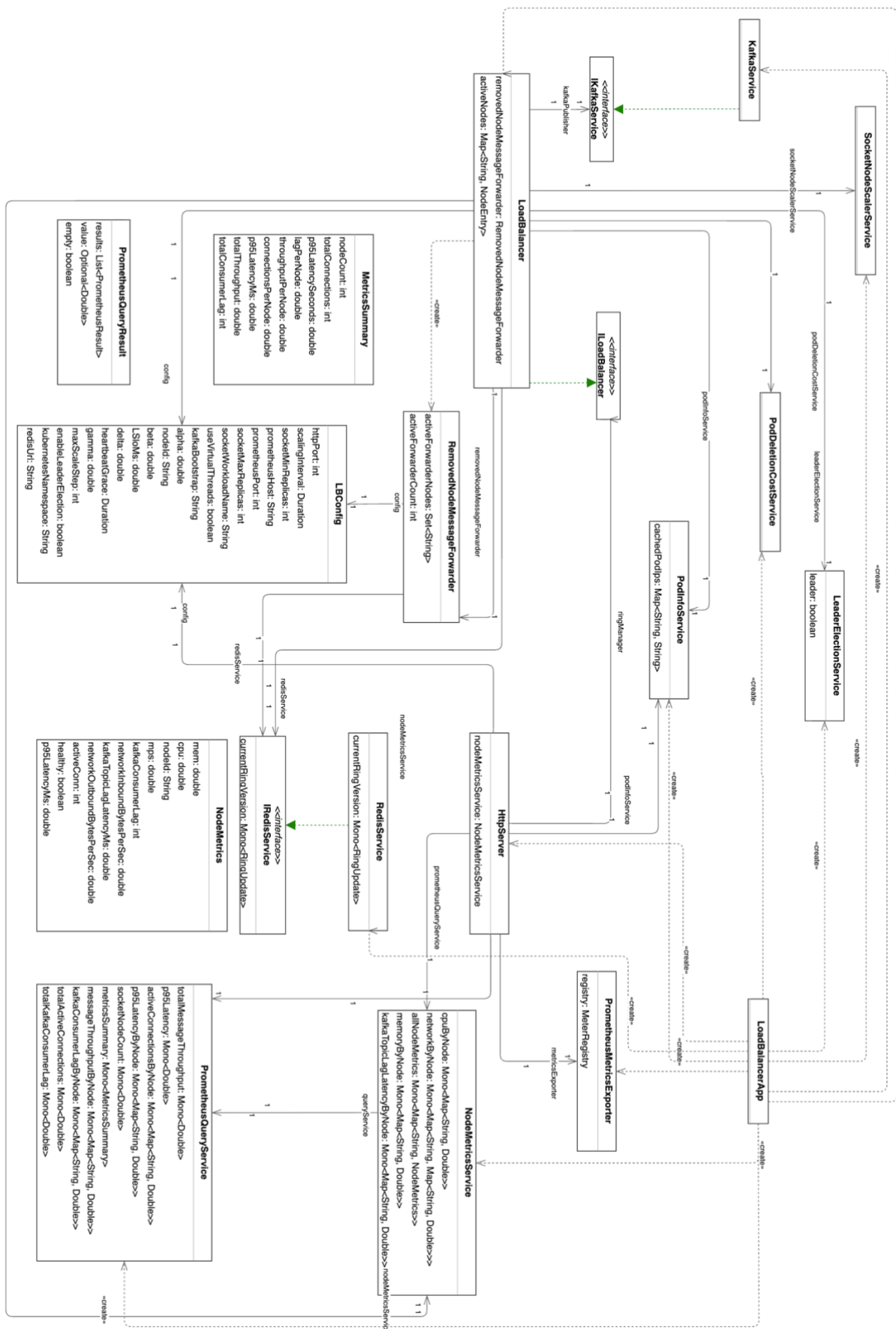
<https://www.haproxy.com/blog/client-ip-persistence-or-source-ip-hash-load-balancing>

18. High Scalability (2015). Consistent Hashing Algorithm [Электронный ресурс] // Режим доступа: <https://highscalability.com/consistent-hashing-algorithm/>
19. Lamping, J., & Veach, E. (2014). A Fast, Minimal Memory, Consistent Hash Algorithm [Электронный ресурс] // Режим доступа: <https://arxiv.org/pdf/1406.2294>
20. Dahlgaard, S., Knudsen, M. B. T., & Thorup, M. (2017). Practical Hash Functions for Similarity Estimation and Dimensionality Reduction [Электронный ресурс] // Режим доступа: <https://arxiv.org/pdf/1711.08797>

ДОДАТОК А

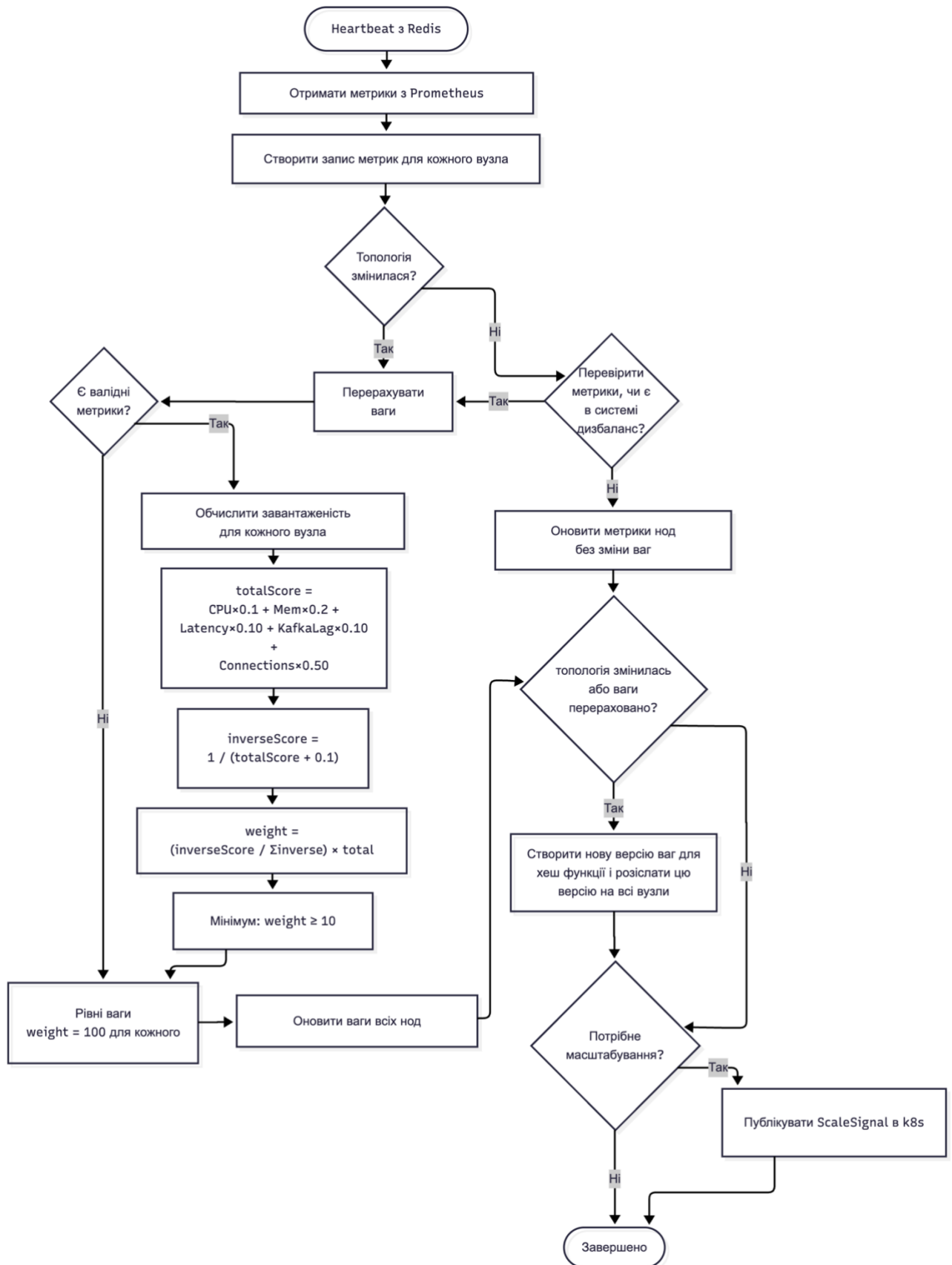
Архітектура високонавантаженої реактивної системи комунікації реального часу





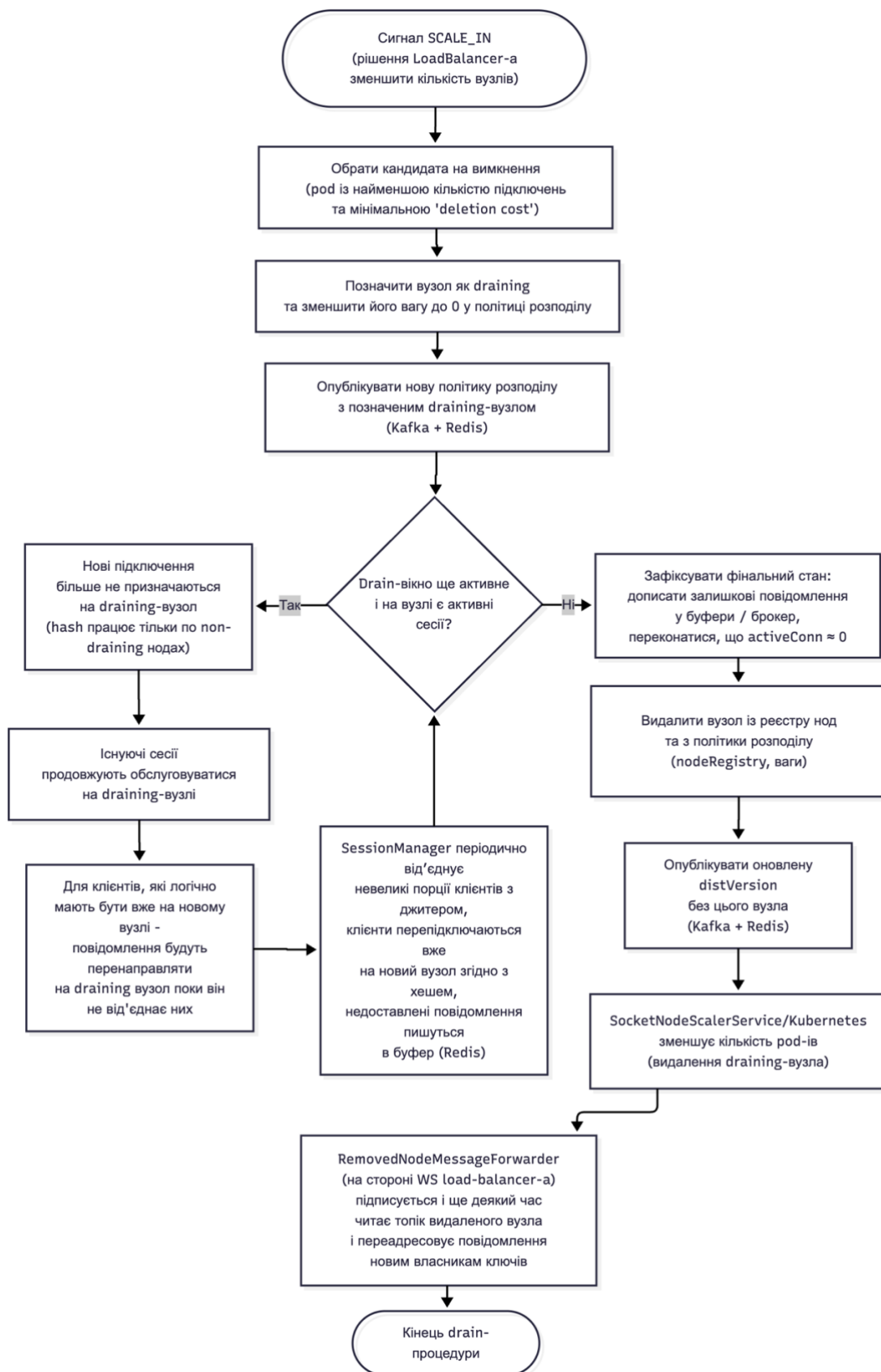
ДОДАТОК В

Алгоритм перерахунку ваг веб-сокет вузлів



ДОДАТОК Г

Алгоритм висушування (draining) веб-сокет вузлів при згортанні (scale-in)



ДОДАТОК Д

Діаграма класів модуля WebSocket сервер-а

