МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

Раківський Роман Зиновійович

Програмно-апаратний шлюз IoT з балансуванням навантаження для протоколу MQTT / Software–Hardware IoT Gateway with Load Balancing for the MQTT Protocol

Спеціальність 122 – Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-43
Р. З. Раківський

Науковий керівник:
к.т.н., доцент, О.Р.Осолінський

Кваліфікаційну роботу допущено до захисту
«___»_____ 2026 р.

Завідувач кафедри
_____ Н.В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
РАКІВСЬКОМУ Роману Зиновійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Програмно-апаратний шлюз IoT з балансуванням навантаження для протоколу MQTT / Software–Hardware IoT Gateway with Load Balancing for the MQTT Protocol
керівник роботи О.Р. Осолінський, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджений наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- проаналізувати особливості IoT-систем та роль MQTT-шлюзу;
- провести огляд і аналіз існуючих рішень побудови MQTT-інфраструктури;
- сформувати постановку задачі;
- розробити архітектуру IoT-шлюзу з балансуванням MQTT;
- реалізувати алгоритмічне забезпечення IoT-шлюзу;
- здійснити вибір та аналіз програмно-апаратних засобів реалізації IoT-шлюзу;
- реалізувати програмно технологічне IoT-шлюзу;
- провести експериментальне тестування IoT-шлюзу.
-

5. Перелік графічного матеріалу в роботі:
- структурна схема програмно-апаратного IoT-шлюзу з балансуванням MQTT;
- схема деталізованого алгоритму роботи IoT-шлюзу.

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент_____ Р. З. Раківський
(підпис) (прізвище та ініціали)

Керівник кваліфікаційної роботи_____ к.т.н., доцент О.Р.Осолінський
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи: 62 с., 35 рис., 4 додатки, 38 джерел.

Метою кваліфікаційної роботи є розробка програмно-апаратного шлюзу IoT з балансуванням навантаження для протоколу MQTT, який реалізованого на платформі Raspberry Pi з підтримкою клієнтів на ESP32.

Об’єкт дослідження: процес організації обміну MQTT-повідомленнями в IoT-системі з великою кількістю клієнтів.

Методи дослідження: аналіз особливостей MQTT-обміну в IoT-системах, методи балансування навантаження між MQTT-брокерами, методи перевірки доступності вузлів на основі health-check, методи маршрутизації MQTT-сесій і команд керування, методи побудови веб-інтерфейсу моніторингу засобами FastAPI.

Результатом роботи є розроблений програмно-апаратний IoT-шлюз, який забезпечує прийом MQTT-підключень від клієнтів ESP32, розподіл сесій між кількома MQTT-брокерами, автоматичне виключення недоступних вузлів із пулу, збір телеметрії, журналювання подій, накопичення системних метрик і візуальний моніторинг стану інфраструктури.

Розроблена система може використовуватись як основа для локальних IoT-інфраструктур збору сенсорних даних.

Ключові слова: IOT-ШЛЮЗ, MQTT, RASPBERRY PI, ESP32, BALANCING, HAPROXY, ECLIPSE MOSQUITTO, SQLITE, DOCKER, FASTAPI, HEALTH-CHECK, WEB-МОНІТОРИНГ.

ANNOTATION

Explanatory note to the qualification thesis: 62 pages, 35 figures, 4 annexes, 38 references.

The aim of the qualification thesis is to develop a software–hardware iot gateway with load balancing for the MQTT protocol, implemented on the Raspberry Pi platform with support for ESP32 clients.

The object of research is the process of organizing MQTT message exchange in an IoT system with a large number of clients.

Research methods: analysis of MQTT exchange features in IoT systems, methods of load balancing between MQTT brokers, methods of node availability checking based on health checks, methods of routing MQTT sessions and control commands, and methods of building a web monitoring interface using FastAPI.

The result of the work is a developed hardware-software IoT gateway that provides acceptance of MQTT connections from ESP32 clients, distribution of sessions among several MQTT brokers, automatic exclusion of unavailable nodes from the pool, telemetry collection, event logging, accumulation of system metrics, and visual monitoring of the infrastructure state.

The developed system can be used as a basis for local IoT infrastructures for sensor data collection.

Keywords: IOT GATEWAY, MQTT, RASPBERRY PI, ESP32, LOAD BALANCING, HAPROXY, ECLIPSE MOSQUITTO, SQLITE, DOCKER, FASTAPI, HEALTH CHECK, WEB MONITORING.

ЗМІСТ

Вступ 7
1 Аналіз предметної області та існуючих рішень для IoT MQTT-шлюзу 9
1.1 Особливості IoT-систем та роль MQTT-шлюзу 9
1.2 Огляд і аналіз існуючих рішень побудови MQTT-інфраструктури 12
1.3 Постановка задачі 17
2 Архітектура, алгоритмічне та інформаційне забезпечення системи 20
2.1 Розробка архітектури IoT-шлюзу з балансуванням MQTT 20
2.2 Алгоритмічне забезпечення IoT-шлюзу 23
2.3 Інформаційне забезпечення IoT-шлюзу 26
3 Програмно-технологічна реалізація та тестування системи 29
3.1 Вибір та аналіз програмно-апаратних засобів реалізації IoT-шлюзу 29
3.2 Програмно-технологічна реалізація IoT-шлюзу 32
3.3 Експериментальне тестування IoT-шлюзу 39
Висновки 44
Список використаних джерел 45
Додаток А Структурна схема програмно-апаратного IoT-шлюзу з балансуванням MQTT 50
Додаток Б Схема деталізованого алгоритму роботи IoT-шлюзу 51
Додаток В Псевдокоди основних програмних модулів системи 52
Додаток Г Апробація отриманих результатів 58

ВСТУП

На сьогоднішній день IoT-пристрої та локальні edge-вузли стають звичайною частиною систем моніторингу, розумного дому, промислових датчиків і невеликих сервісів збору телеметрії. В таких системах часто використовуються мікроконтролери наприклад ESP32, які передають дані на центральний вузол або шлюз. Для обміну повідомленнями широко застосовується MQTT, тому що він простий, економний за трафіком і підходить для великої кількості клієнтів. Проте із збільшенням числа підключених пристроїв і частоти повідомлень зростають вимоги до стійкості та продуктивності брокера. В реальній експлуатації можливі пікові навантаження, перепідключення через нестабільний Wi-Fi, а також відмови окремих сервісів.

Використання шлюзу дозволяє приховати внутрішню структуру брокерів від клієнтів і надати єдину точку входу для підключення. Балансувальник розподіляє сесії між кількома брокерами, що підвищує пропускну здатність системи і зменшує ризик перевантаження одного вузла. Додатково важливо мати механізм моніторингу стану брокерів якщо один із них стає недоступним, шлюз повинен виключити його з пулу та перенаправити трафік на працюючі вузли.

Метою кваліфікаційної роботи є розробка програмно-апаратного IoT-шлюзу з балансуванням навантаження для протоколу MQTT, реалізованого на платформі Raspberry Pi з підтримкою клієнтів на ESP32.

Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати особливості організації MQTT-обміну в IoT-системах та визначити причини виникнення перевантаження, затримок і відмов у системах з великою кількістю клієнтів;

– розробити загальну архітектуру програмно-апаратного IoT-шлюзу з єдиною точкою входу для MQTT-клієнтів;

– вибрати програмні та апаратні засоби реалізації шлюзу з врахуванням обмежених ресурсів Raspberry Pi та особливостей роботи ESP32-клієнтів;

– реалізувати пул MQTT-брокерів і механізм балансування підключень між ними;

– розробити та реалізувати механізм перевірки доступності брокерів для автоматичного визначення їх стану, виключення недоступних вузлів із пулу та повторного підключення після відновлення;

– розробити алгоритми маршрутизації MQTT-сесій і команд керування з урахуванням активного брокера, до якого підключений клієнт;

– розробити інформаційне забезпечення шлюзу, зокрема структуру даних для збереження відомостей про брокери, сесії, події, метрики та пристрої;

– реалізувати локальне сховище даних і модулі збору телеметрії, подій та статистики роботи системи;

– реалізувати веб-інтерфейс моніторингу для відображення стану брокерів, активних сесій, журналу подій і часових рядів метрик;

– провести тестування розробленого шлюзу в умовах звичайного обміну повідомленнями, пікового навантаження, масових повторних підключень, відмови брокера та його подальшого відновлення.

Об’єктом дослідження є процес організації обміну MQTT-повідомленнями в IoT-системі з великою кількістю клієнтів.

Предметом дослідження є методи й програмно-апаратні засоби балансування навантаження та маршрутизації MQTT-сесій у локальній IoT-інфраструктурі.

Результати кваліфікаційної роботи апробовані та опубліковані в матеріалах IX Всеукраїнської студентської конференції «Науковий простір: аналіз, сучасний стан, тренди та перспективи», м. Київ, Україна.

Практична цінність роботи полягає в тому, що розроблений прототип можна використати як основу для локальних систем збору сенсорних даних. Шлюз розподіляє MQTT-сесії між брокерами, знижує ризик простою, веде журнали подій і збирає метрики, а стан системи можна переглядати через веб-інтерфейс.

Структура роботи включає вступ, три розділи, висновки, список використаних джерел і додатки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ ДЛЯ IOT MQTT-ШЛЮЗУ

1.1 Особливості IoT-систем та роль MQTT-шлюзу

Більшість IoT-систем на початковому етапі мають досить просту структуру. В більшості це один мікроконтролер, кілька датчиків і програма, яка передає виміряні значення на сервер. Однак із збільшенням кількості пристроїв система починає працювати менш передбачувано. Зростає кількість підключень в мережі, збільшується кількість повідомлень а також з’являються типові проблеми які не видно на малих макетах. Наприклад, Wi-Fi може періодично пропадати, частина вузлів може перезавантажуватися а телеметрія може приходити хвилями в той момент коли всі датчики відправляють дані майже в один момент. В результаті виникає потреба не просто передати дані, а організувати обмін так, щоб він був стабільний, керований і зрозумілий для адміністрування. Для таких рішень часто використовують ESP32, Arduino або інші мікроконтролери з вбудованим Wi-Fi, ресурсів яких достатньо для базових задач збору даних. У типовому режимі такий вузол зчитує дані з сенсора, формує повідомлення, надсилає його на сервер, а далі повторює цикл через заданий інтервал або очікує на команду. Але в реальності вузол часто працює в нестабільному середовищі. Тому до таких систем часто додають локальний сервер або edge-шлюз, наприклад на базі Raspberry Pi. Через це вузол має вміти перепідключатись і відновлювати роботу без участі людини.

Для зв’язку між вузлами та шлюзом в таких системах можна використовувати MQTT. Це протокол обміну повідомленнями який працює за моделлю publish/subscribe. Основна ідея полягає в тому, що відправник не передає дані конкретному отримувачу, а публікує повідомлення в певну тему, на яку підписуються зацікавлені клієнти. Центром цієї взаємодії є брокер (broker). Він приймає повідомлення від клієнтів і доставляє їх клієнтам-підписникам відповідних тем [1]. Topic у MQTT — це логічний канал повідомлень, який зазвичай будують ієрархічно для зручного групування даних. Така структура потрібна для керованості в ті моменти коли тем багато, і зрозуміла схема

іменування сильно полегшує підтримку. В протоколі MQTT також існують wildcard-підписки, які дозволяють підписатися відразу на групу тем. Це зручно для моніторингу коли треба бачити дані з багатьох вузлів без окремих підписок на кожну тему. Модель publish/subscribe зменшує залежність між компонентами: вузол публікує телеметрію у свою тему й не залежить від того, які сервіси її обробляють. Тому можна додавати нові сервіси без змін у прошивках IoT пристроїв. Це корисно, коли система розвивається і доповнюється [2]. Крім того MQTT-шлюз виконує периферійні функції під час експлуатації системи. Він допомагає переводити дані до одного вигляду, накладає правила доступу, тимчасово зберігає дані при проблемах мережі, показує стан системи і виконує просту локальну обробку. Такий підхід не замінює брокер, а доповнює його, тому що шлюз бере на себе задачі, пов’язані з різними типами пристроїв і нестабільними умовами мережі [6].

Також під час роботи мереж IoT виникає питання, що робити, якщо повідомлення загубиться під час його відправки або отримання. З врахуванням таких проблем було розроблено механізм QoS який задає рівень гарантії доставки. MQTT визначає три рівні QoS: 0, 1 і 2.

QoS 0 має найменші накладні витрати, але не гарантує повторної доставки повідомлення.

QoS 1 є компромісом, коли повідомлення має бути доставлене хоча б один раз.

QoS 2 забезпечує найвищий рівень надійності, однак потребує найбільшої кількості службових обмінів.

В практичній системі їх обирають залежно від типу даних. Телеметрія, яка оновлюється часто наприклад кожні 5–10 секунд, може працювати на QoS 0 або QoS 1. Якщо одне значення загубиться, це не критично, бо скоро прийде наступне. Команда керування потребує більшої надійності, тому її можуть передавати за допомогою QoS 2. При цьому вона створює більше навантаження на брокер і клієнтів, тому його використовують лише там де дійсно потрібна максимальна надійність. Важливо також враховувати, що фактичний рівень QoS для отримувача

залежить не лише від параметрів публікації, а й від налаштувань підписки. Тобто це не завжди один фіксований рівень на всю систему. Тому під час проектування потрібно планувати QoS окремо для різних класів даних і тем [3].

Часто в IoT потрібен не лише потік даних, а й їх поточний стан. Для цього в MQTT існує retained повідомлення. Якщо повідомлення позначене прапором retain, брокер зберігає його як останнє актуальне значення для відповідної теми. Коли новий клієнт підписується на тему або на шаблон тем, що її включає, брокер може одразу віддати це retained значення. Його зручно застосовувати для тем типу стан пристрою, поточний режим або останні вимірювання. Такі повідомлення також доцільно використовувати для конфігураційних параметрів або статусів, які мають бути доступні відразу після підключення клієнта [4].

Незважаючи на надійність системи потрібно враховувати аварійне зникнення вузла. При нормальній роботі він може надіслати повідомлення про закінчення роботи. Але під час поломки або збою він цього зробити не може. Через це у MQTT є механізм Will або як називають частіше LWT. Суть цього механізму полягає в тому, що у разі аварійного відключення клієнта брокер сам публікує заздалегідь задане повідомлення, і система моніторингу одразу бачить, що пристрій став недоступним. Це не вирішує всі проблеми доступності, але дає зрозумілий базовий сигнал для системи [1].

Ще одним критично важливим завданням MQTT-шлюзу є контроль навантаження та балансування трафіку. Так як MQTT працює поверх TCP, кінцеві пристрої зазвичай підтримують постійні з’єднання та періодично обмінюються службовими або телеметричними повідомленнями. Відповідно, наявність у системі сотні вузлів означає підтримку сотні паралельних сесій. Кожне таке підключення вимагає від сервера виділення пам’яті, постійної перевірки таблиць підписок а також створення окремих черг та обробки підтверджень для повідомлень із рівнем якості обслуговування (QoS) 1 або 2. Через це, чим вищий QoS тим більше службового обміну і тим більше роботи для брокера [1].

Незважаючи на те як система справляється в нормальному режимі вона може просідати в пікові моменти. Найтиповішою є ситуація масового перепідключення

після аварійних збоїв. Коли всі клієнти одночасно намагаються відновити з'єднання та передати накопичені дані, виникає різкий дефіцит ресурсів, що призводить до затримок або відхилення нових підключень. Ще однією проблемою є синхронна телеметрія, коли багато вузлів надсилають дані з однаковим інтервалом і майже одночасно створюють навантаження на брокер. Це генерує регулярні мікропіки трафіку, що створюють нестабільність в системі. Також використання єдиного брокера створює вразливість його відмови і паралізацію всієї мережі, що запускає хвилю перепідключень. Тому архітектура надійного шлюзу вимагає впровадження механізмів балансування, які охоплюють розподіл клієнтських підключень та паралельну обробку вхідних даних. Один з найпростіших прикладів балансування на рівні обробників даних є shared subscriptions. Це режим, коли кілька підписників входять у групу, а повідомлення з потоку розподіляються між ними так, щоб кожне повідомлення обробляв один учасник групи, а не всі одразу. В документації EMQX shared subscription описується саме як механізм load balancing серед кількох підписників, які спільно обробляють дані. На практиці це корисно тоді, коли працює кілька однакових обробників, наприклад процеси запису в базу даних, агрегації або виконання правил [6].

1.2 Огляд і аналіз існуючих рішень побудови MQTT-інфраструктури

Збільшення масштабів IoT-мережі до сотень активних пристроїв призводить до того, що класична модель з єдиним MQTT-брокером перетворюється на вузьке місце, що загрожує стійкості всієї системи. Тому для забезпечення стабільності архітектурний підхід зміщується від базового запуску сервісу до розробки комплексного IoT-шлюзу. Така платформа має об'єднувати ядро обміну повідомленнями, механізми балансування навантаження та підсистему безперервного моніторингу. Створювати всі ці механізми з нуля складно та недоцільно, тому в інженерній практиці зазвичай поєднують готові перевірені компоненти. В сучасній інженерній практиці надійні системи будуються шляхом інтеграції перевірених готових рішень. Тому доцільно розглянути програмні

компоненти, які найчастіше використовують для побудови масштабованої MQTT-інфраструктури.

Найпоширенішим вибором для базових інсталяцій є Eclipse Mosquitto. Цей брокер ідеально підходить для проєктів, де головним пріоритетом є простота, низьке споживання ресурсів та прозора конфігурація. Завдяки легкій архітектурі його найчастіше розгортають на одноплатних комп'ютерах. Mosquitto працює як єдиний системний сервіс, не вимагає складної інфраструктури, а управління здійснюється через класичний конфігураційний файл. Базових можливостей Mosquitto достатньо для більшості простих IoT-шлюзів (рисунок 1.1).

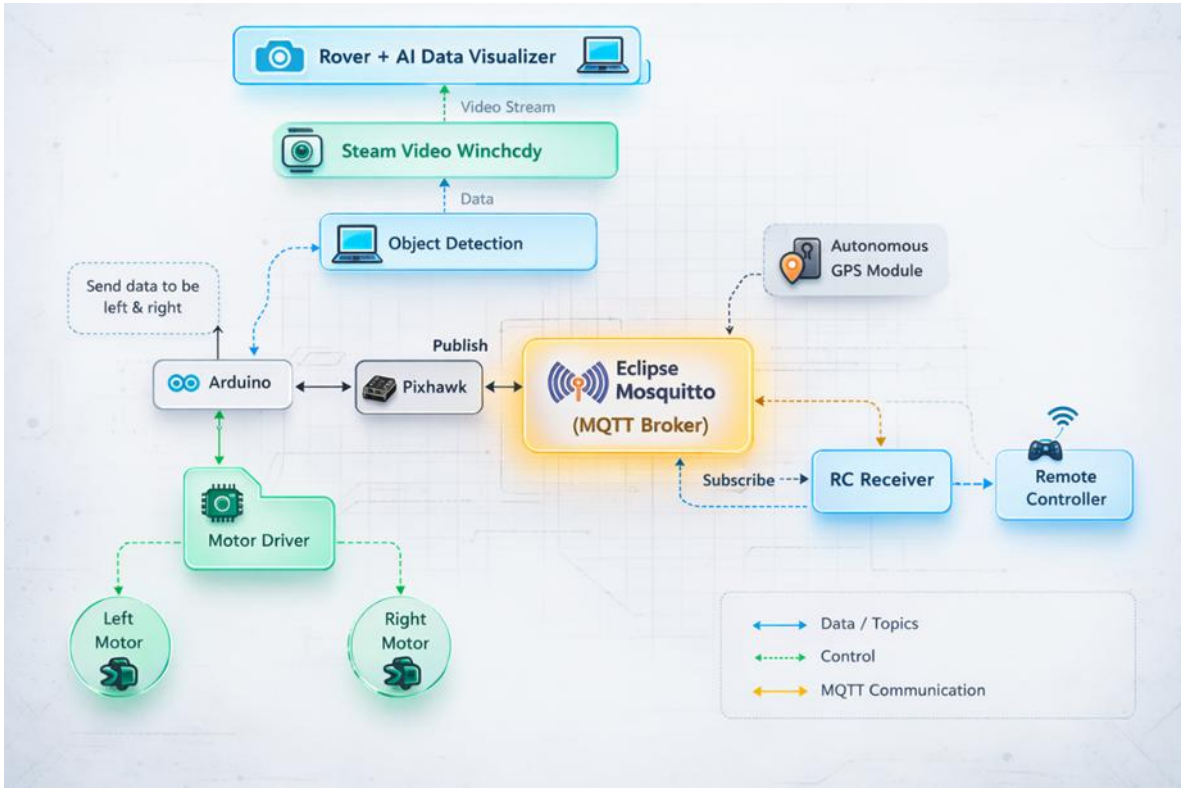


Рисунок 1.1 – Взаємодія IoT-пристроїв через MQTT-брокер Mosquitto

Водночас зі збільшенням масштабу мережі стають помітними архітектурні обмеження Mosquitto. В Mosquitto відсутні вбудовані інструменти для розширеної маршрутизації, збору детальної статистики чи зручного керування групами споживачів. Щоб компенсувати ці недоліки, базову архітектуру доповнюють зовнішніми компонентами.

Одним із практичних способів масштабування є розміщення перед брокерами проксі-сервера HAProxy. Його розміщення перед екземплярами брокера дозволяє ефективно розподіляти вхідні клієнтські підключення та організувати просте резервування на випадок збою. Практичність та поширеність такої зв'язки підтверджується тим, що сценарії інтеграції з HAProxy детально описані навіть в офіційній документації самого Mosquitto [7].

Якщо проєкт із самого початку потребує розширених можливостей, доцільно розглядати комплексні платформи, в яких брокер є частиною ширшої екосистеми. Прикладом такого підходу є EMQX. Цю систему доцільно обирати тоді, коли архітектура потребує готових інструментів масштабування, гнучкого керування сесіями та інтегрованої підсистеми моніторингу (рисунок 1.2).

Рисунок 1.2 – Взаємодія IoT-пристроїв та додатків через кластер EMQX

Для побудови шлюзу з балансуванням навантаження EMQX має дві важливі можливості. А саме мультивузлове масштабування та розподіл завдань між споживачами даних. Зокрема, платформа має повноцінну підтримку механізму спільних підписок (shared subscriptions). Цей режим функціонує як вбудований балансувальник навантаження на рівні клієнтів-підписників. Завдяки йому повідомлення з певного topic доставляється не всім активним клієнтам, а лише одному учаснику логічної групи. Завдяки цьому повідомлення не дублюються між

усіма обробниками, а навантаження рівномірніше розподіляється між кількома процесами-виконавцями.

Але при виборі EMQX потрібно враховувати обмеження, пов’язане з моделлю ліцензування. Хоча платформа має потужну безкоштовну версію, частина механізмів для побудови відмовостійких кластерів і забезпечення високої доступності доступна переважно в комерційній редакції [8].

Ще одним брокером, який розглядають під час проектування MQTT-інфраструктури, є VerneMQ. Він суттєво відрізняється своїм операторським підходом, оскільки від самого початку спроектований для розподілених інсталяцій та обслуговування масивних пулів підключень. Архітектура VerneMQ базується на master-less кластеризації де немає головних і підпорядкованих вузлів, тому кожен сервер може самостійно обробляти клієнтський трафік, забезпечуючи високу доступність і горизонтальне масштабування (рисунок 1.3) [9].

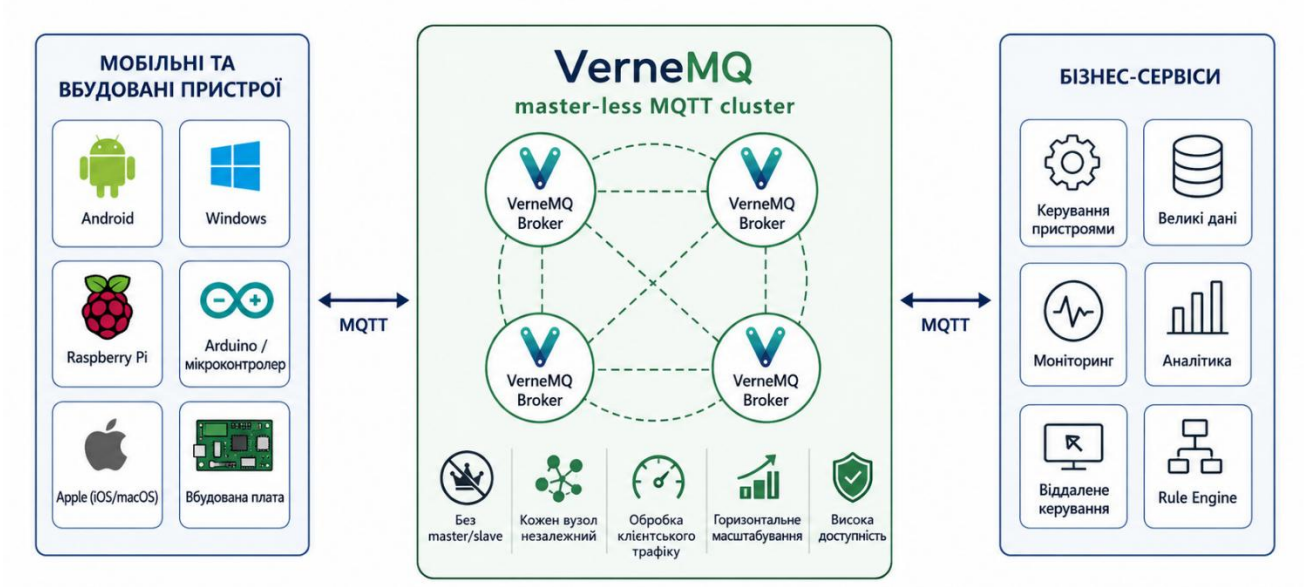


Рисунок 1.3 – Топологія розподіленого кластера VerneMQ

VerneMQ постійно синхронізує метадані та інформацію про сесії. Якщо зв’язок між вузлами зникає, система трактує це як розрив мережі, і після відновлення з’єднання намагається автоматично синхронізувати дані. Якщо ж вузол виведено з ладу безповоротно, його необхідно примусово видалити з кластера. Під час примусового видалення непрацюючого вузла потрібно

враховувати ризик втрати офлайн-повідомлень із QoS 1 або QoS 2, оскільки за замовчуванням вони не реплікуються між внутрішніми сховищами. Водночас, для мінімізації подібних збоїв та стабільної роботи кластера критично важливо правильно налаштувати міжвузлову комунікацію. Оскільки VerneMQ використовує механізми Erlang distribution, вузли знаходять один одного через службу epmd. Проте порти для безпосереднього обміну даними генеруються динамічно, тому при розгортанні шлюзу за брандмауером необхідно жорстко обмежити діапазон цих портів [10].

Також є брокер HiveMQ. На відміну від попередніх рішень, HiveMQ більше орієнтований не на мінімальне споживання ресурсів, а на надійність, масштабування та гнучке керування трафіком. Цей брокер належить до іншого класу рішень і орієнтований на високу надійність та детальне керування MQTT-трафіком. Він розрахований на високонавантажені системи з жорсткими вимогами до високої доступності та масштабування. Його кластер функціонує як розподілена система, що виглядає для клієнтів як єдиний логічний брокер. Завдяки masterless-архітектурі система не має єдиної точки відмови: якщо один вузол виходить з ладу, інші продовжують обробляти трафік. Кластер підтримує горизонтальне масштабування і здатний змінювати розмір без зупинки роботи клієнтів (рисунок 1.4) [11].

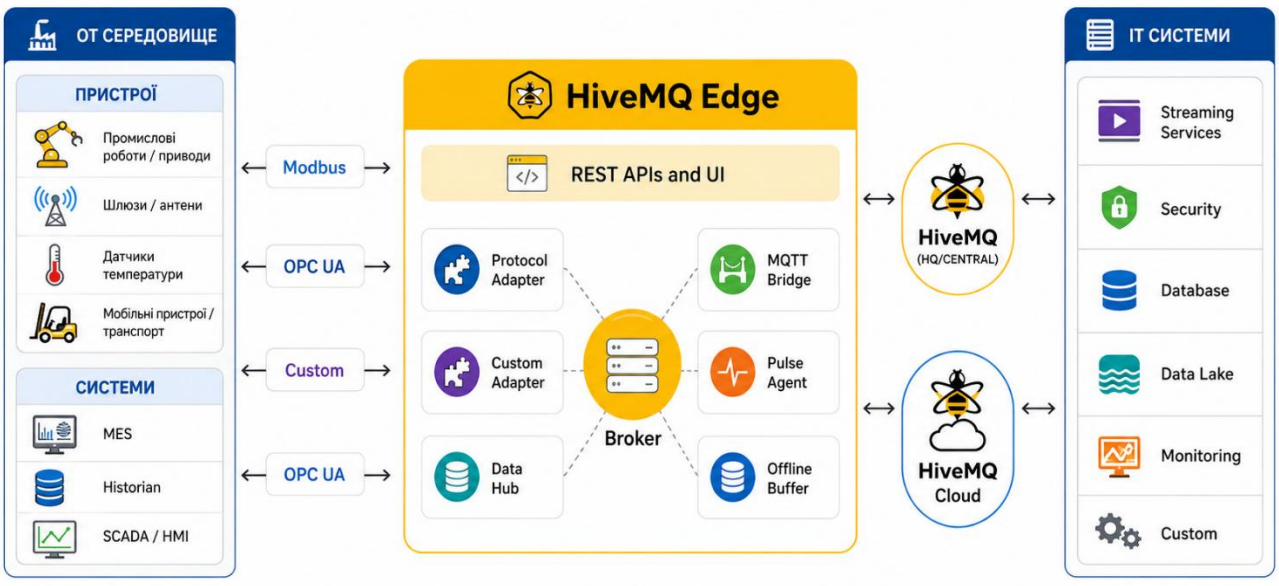


Рисунок 1.4 – Архітектура корпоративної платформи HiveMQ

Перевагою HiveMQ є гнучка система розширень (extension system), яка дозволяє централізовано керувати правилами безпеки. Брокер підтримує як базові дозволи так і динамічні перевірки. Вони викликаються для кожного окремого пакета PUBLISH або SUBSCRIBE і приймають рішення про допуск на основі його вмісту [12]. Це дає змогу задавати складні правила доступу на стороні шлюзу без оновлення прошивок мікроконтролерів.

Крім того, брокер надає механізм інтерцепторів для перехоплення та модифікації MQTT-пакетів безпосередньо під час їхньої маршрутизації [13]. Завдяки цьому шлюз може виконувати роль контрольної точки: блокувати публікації в заборонені topic, обмежувати надто активні пристрої або змінювати дані під час маршрутизації. Проте впровадження таких функцій вимагає написання додаткового коду, що підвищує складність підтримки системи. Оскільки HiveMQ працює у середовищі Java, він потребує більше оперативної пам’яті та ресурсів процесора, ніж легші рішення на C чи Erlang, тому для мікрокомп’ютерів його використання часто є надмірним.

1.3 Постановка задачі

Аналіз показав, що зі збільшенням кількості MQTT-клієнтів в IoT-системах зростає ризик перевантаження брокера, частих перепідключень і зниження загальної відмовостійкості. Особливо це відчувається на локальних edge-інсталяціях на Raspberry Pi, де обчислювальні ресурси є обмеженими, а мережеві умови можуть бути нестабільними. Використання одного MQTT-брокера спрощує побудову системи, однак створює єдину точку відмови та ускладнює підтримку стабільної роботи при зростанні навантаження

З огляду на це доцільно розробити програмно-апаратний IoT-шлюз для MQTT, який надаватиме клієнтам єдину точку входу, розподілятиме підключення між брокерами, виявлятиме недоступні вузли та збиратиме дані для моніторингу стану системи. Такий підхід має забезпечити стабільну роботу MQTT-

інфраструктури навіть за умов пікових навантажень, шторму перепідключень та часткової відмови брокерів

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Проаналізувати особливості організації MQTT-обміну в IoT-системах та визначити причини виникнення перевантаження, затримок і відмов у системах з великою кількістю клієнтів.

2. Розробити загальну архітектуру програмно-апаратного IoT-шлюзу з єдиною точкою входу для MQTT-клієнтів.

3. Обрати програмні та апаратні засоби реалізації шлюзу з урахуванням обмежених ресурсів Raspberry Pi та особливостей роботи ESP32-клієнтів.

4. Реалізувати пул MQTT-брокерів та механізм балансування підключень між ними на основі HAProxy.

5. Розробити та реалізувати механізм health-check для автоматичного визначення стану брокерів, виключення недоступних вузлів із пулу та їх повторного підключення після відновлення.

6. Розробити алгоритми маршрутизації MQTT-сесій і команд керування з урахуванням активного брокера, до якого підключений клієнт.

7. Розробити інформаційне забезпечення шлюзу, зокрема структуру даних для збереження відомостей про брокери, сесії, події, метрики та пристрої.

8. Реалізувати локальне сховище даних на базі SQLite та модулі збору телеметрії, подій і статистики роботи системи.

9. Реалізувати веб-інтерфейс моніторингу для відображення стану брокерів, активних сесій, журналу подій і часових рядів метрик.

10. Провести тестування розробленого шлюзу в основних сценаріях роботи: при звичайному обміні повідомленнями, піковому навантаженні, штормі перепідключень, відмові брокера та його подальшому відновленні.

В результаті виконання поставлених завдань має бути розроблено та експериментально перевірено програмно-апаратний IoT-шлюз, який підвищує відмовостійкість, масштабованість і керованість локальної MQTT-інфраструктури

та забезпечує зручний контроль її стану засобами журналювання, збору метрик і веб-моніторингу.

2 АРХІТЕКТУРА, АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Розробка архітектури IoT-шлюзу з балансуванням MQTT

В простих IoT-проєктах мікроконтролери зазвичай надсилають зібрані дані безпосередньо на один MQTT-брокер. Проте, коли кількість пристроїв зростає, такий підхід стає вразливим. Для вирішення цієї проблеми в роботі запропоновано програмно-апаратний IoT-шлюз, який розподіляє MQTT-підключення між кількома брокерами. Тоді центральною частиною в системі стане мікрокомп’ютер Raspberry Pi. Перевага такого підходу полягає у тому, що для кінцевих пристроїв ESP32 конфігурація мережі залишається максимально простою. Вони продовжують використовувати лише одну статичну IP-адресу та працюють за стандартним протоколом MQTT. Водночас логіка балансування, перевірки доступності брокерів, журналювання подій і збору метрик залишається прихованою від сенсорів та виконується на рівні шлюзу.

Вибір MQTT для такої взаємодії є виправданим, тому що це легкий стандарт publish/subscribe для обмежених IoT-середовищ. На практиці ESP32 легко утримує стабільну сесію, передає телеметрію малими пакетами та приймає команди керування. Завдяки механізмам MQTT, таким як рівні QoS, keep-alive та повідомлення Last Will, система коректно реагує на перепідключення вузлів без критичних збоїв. До того ж використання Raspberry Pi в ролі центрального локального хабу дозволяє мінімізувати залежність від зовнішніх хмарних сервісів і водночас зберегти повну керованість мережею. Разом з цим потрібно враховувати, що Raspberry Pi є ARM-вузлом з обмеженими апаратними ресурсами. Тому програмна частина шлюзу має бути максимально легкою, передбачуваною і не створювати зайвого навантаження у вигляді неконтрольованих записів на карту пам’яті чи важких фонових процесів. Загальна структура зображена на рисунку 2.1.

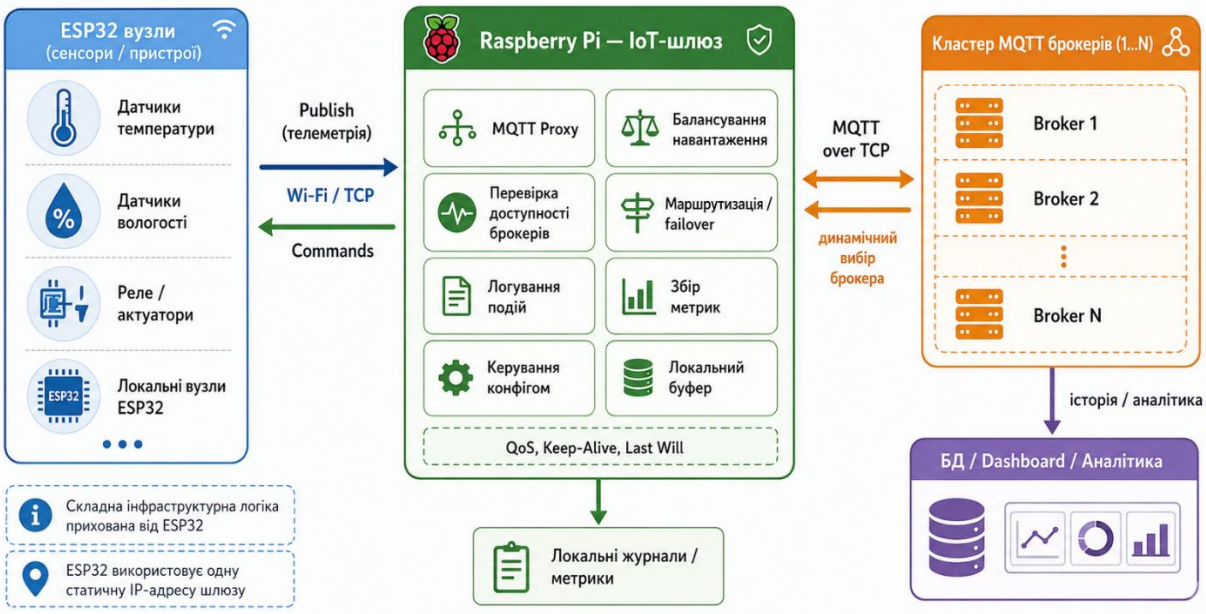


Рисунок 2.1 – Загальна структура IoT-шлюзу з балансуванням MQTT

Мікроконтролер ESP32 у цій архітектурі виконує подвійну роль - джерела та споживача повідомлень. Він зчитує значення сенсорів, формує телеметрію і публікує її в MQTT-топіки, а також підписується на топіки команд для керування пристроями. Така модель publish/subscribe робить систему гнучкою, оскільки кілька споживачів, наприклад дашборд і база даних, можуть одночасно отримувати одну й ту саму телеметрію. При цьому ESP32 не потребує інформації про те, хто саме обробляє його дані. Зворотний зв’язок працює так само прозоро, команда згенерована через веб-інтерфейс або автоматичне правило, доставляється до мікроконтролера через брокер. Такий підхід добре відповідає принципам роботи MQTT.

Мережевий сегмент між ESP32 і Raspberry Pi реалізується через Wi-Fi. Це найдоступніший варіант для прототипу та домашньої експлуатації. Водночас Wi-Fi не є повністю стабільним каналом для IoT-систем, тому що можливі періодичні перепідключення пристроїв, зміни затримок і короткочасні втрати пакетів. Через це MQTT-сесії мають динамічний характер. Архітектура шлюзу спроєктована так, щоб бути стійкою до такої поведінки і завжди звертається до єдиної точки входу, а шлюз бере на себе всю логіку маршрутизації сесій.

Найважливішим компонентом системи є сам шлюз на базі Raspberry Pi. Шлюз є центральною точкою для балансування, контролю стану, моніторингу та конфігурації. На практиці шлюз виконує дві основні функції: приймає MQTT-підключення від клієнтів ESP32 і керує внутрішньою MQTT-інфраструктурою, розподіляючи навантаження та зменшуючи вплив одиничних відмов. Цей підхід є типовим для edge-систем, де центральний вузол виступає одночасно хабом і спостерігачем, який реагує на стан підсистем.

Всередині шлюзу ключову роль відіграє вхідний проксі-модуль. Його завдання прийняти клієнтське TCP-з'єднання з MQTT-трафіком і переадресувати його на один із брокерів у пулі. Так як MQTT передбачає тривалі з'єднання, балансування має зберігати прив'язку клієнта до одного брокера в межах поточної сесії. Цей метод відомий як балансування з прив'язкою сесії, що зменшує кількість нестабільних переходів. У шлюз закладена така логіка де вибір брокера відбувається при новому підключенні, після чого сесія працює стабільно. Повторний вибір виконується лише при перепідключенні клієнта або відмові брокера. Сам пул брокерів необхідний для масштабування системи. У найпростішому випадку це може бути один сервер, але при збільшенні кількості клієнтів шлюз починає виконувати свою головну функцію, розподіляти підключення так, щоб жоден брокер не став вузьким місцем мережі. Архітектура прототипу дозволяє легко перейти від одного брокера до кількох виключно через конфігурацію шлюзу, без будь-яких змін у програмному коді ESP32. Щоб балансувальник не спрямовував клієнтів на недоступні вузли, шлюз має постійно перевіряти стан брокерів. Для цього передбачено модуль health-check, який періодично перевіряє доступність порту брокера та час його відповіді. Головне завдання цього модуля заборонити шлюзу відправляти нові підключення на брокер, який завис або відповідає нестабільно. Якщо сервер зупиняється, шлюз тимчасово виключає його зі списку доступних, і нові клієнти перенаправляються на інші вузли. Після відновлення роботи брокер автоматично повертається в пул.

Важливою вимогою до системи є її спостережуваність. Для діагностики причин затримок або частих перепідключень в архітектурі реалізовано модулі

логування та збору метрик. Журнал подій фіксує підключення клієнтів, помилки проксіювання та реакції на відмову серверів. Метрики відображають кількісні показники: кількість активних сесій, розподіл навантаження та частоту помилок. Поєднання цих інструментів дозволяє контролювати тривалу роботу системи без постійного ручного втручання. Збереження даних в базу та їх відображення на дашборді не є обов'язковими для самого протоколу MQTT, але вони потрібні для повноцінної експлуатації системи. Локальне збереження історії телеметрії та візуалізація метрик продуктивності дозволяють аналізувати поведінку мережі під навантаженням.

2.2 Алгоритмічне забезпечення IoT-шлюзу

Робота шлюзу починається зі старту сервісів і завантаження конфігурації. На цьому етапі система читає параметри мережі, список брокерів у пулі, обрану політику балансування, інтервали перевірок health-check та ліміти сесій. Основне завдання початкового етапу — перевірити наявність заданих брокерів, відсутність конфліктів портів і коректність значень таймерів (рисунок 2.2).

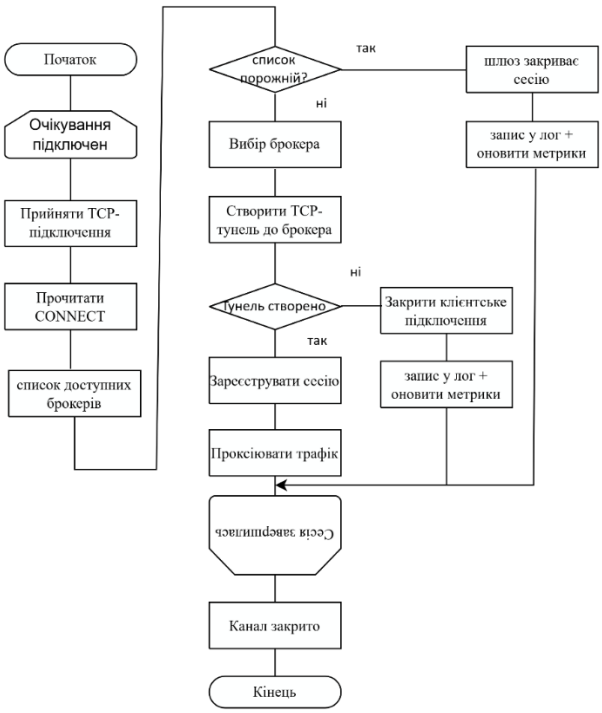


Рисунок 2.2 – Алгоритм прийому MQTT-сесії та вибору брокера

Якщо пул брокерів порожній, старт переривається помилкою. Після успішної перевірки ініціалізуються підсистеми журналу подій та метрик, щоб усі подальші дії фіксувалися одразу.

Паралельно з прийомом клієнтів працює цикл health-check. Він з фіксованим інтервалом проходить по списку брокерів і виконує просту перевірку доступності намагаючись встановити TCP-з'єднання до broker port у межах заданого таймауту.

Перевірка має бути швидкою і не створювати зайвого навантаження, але водночас повинна визначати, чи приймає брокер підключення на мережевому рівні. Щоб уникнути випадкових спрацьовувань через короткі затримки, використовується правило порогів, де брокер переводиться у стан DOWN не після одного збою, а після кількох невдалих перевірок підряд, і повертається в UP лише після кількох успішних перевірок підряд. Після завершення кожного циклу перевірки health-check формує оновлений список healthy broker, в якому усі брокери зі станом UP. Балансувальник використовує цей список як фільтр: брокери зі станом DOWN не розглядаються як кандидати для нових MQTT-сесій (рисунок 2.3).

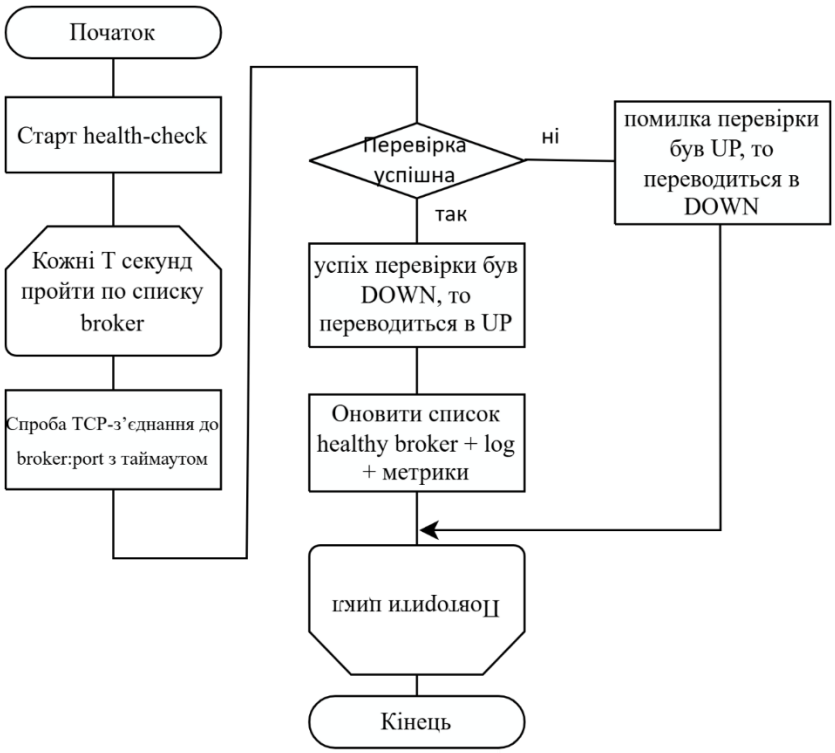


Рисунок 2.3 – Алгоритм health-check та оновлення стану брокерів

Окремої уваги потребує маршрутизація команд керування, яка безпосередньо залежить від балансу навантаження та структури пулу брокерів. Якщо брокерів кілька і вони не синхронізують підписки між собою, команда має потрапити саме у той брокер, через який зараз підключений потрібний ESP32. Саме тому таблиця сесій у шлюзі є важливою для least-connections, та для маршрутизації команд (рисунок 2.4).

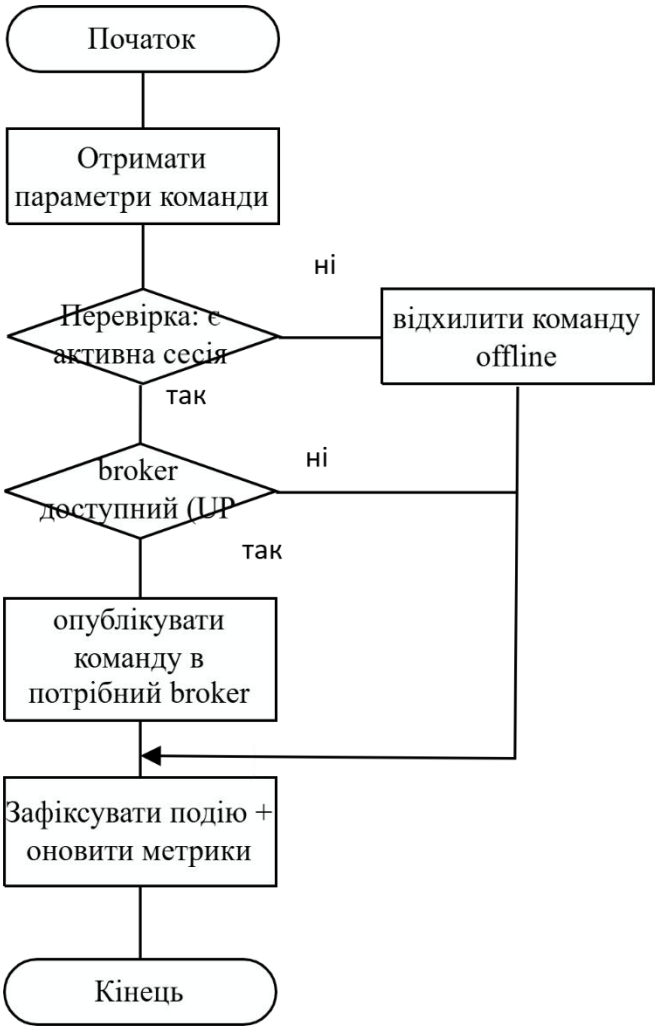


Рисунок 2.4 – Схема алгоритму маршрутизації команд

Алгоритм включає наступні кроки: команда надходить у систему з веб-інтерфейсу або сервісу керування, шлюз зчитує target_client_id і перевіряє в таблиці відповідність client_id – broker. Якщо для клієнта знайдено активну сесію і брокер доступний, команда публікується саме в цей брокер. Якщо клієнт перебуває офлайн

або його брокер недоступний, можна застосувати резервний варіант — надіслати команду на всі доступні брокери.

2.3 Інформаційне забезпечення IoT-шлюзу

Щоб забезпечити стабільну роботу програмно-апаратного IoT-шлюзу на базі Raspberry Pi визначено інформаційні потоки, якими оперує система. Взаємодія між кінцевими вузлами (ESP32), внутрішніми модулями шлюзу (MQTT-проксі, health-check) та пулом брокерів базується на постійному обміні даними телеметрії, командами керування та службовими статусами. Цей обмін дозволяє ефективно балансувати мережеве навантаження, забезпечувати стійкість до збоїв та контролювати актуальний стан усієї IoT-інфраструктури.

Вхідна інформація надходить у вигляді MQTT publish від ESP32. Мінімальний набір таких даних містить topic, payload, рівень QoS і час прийому повідомлення на шлюзі. На практиці payload доцільно передавати у форматі JSON із полями device_id, назвою метрики, значенням value і часом вимірювання ts. Окремо існують вхідні дані керування командами з веб-інтерфейсу або сервісу керування, де вказано target_client_id, topic команди і payload. Окремим типом вхідних даних для алгоритму балансування є внутрішня службова інформація, що генерується самим шлюзом. Ключовим елементом тут виступає динамічний список брокерів пулу із зазначенням їхнього поточного стану (UP або DOWN), який безперервно оновлюється за результатами роботи модуля health-check.

Вихідну інформацію, яку генерує шлюз, розділено на дві складові. Перша складова це результати фактичної маршрутизації трафіку коли успішно створено проксі-тунель до обраного брокера, цільової доставки команди керування або ж відхилення вхідного підключення. Друга складова це дані спостережуваності до яких належать журнал подій та системні метрики. Такий підхід дає змогу оцінювати роботу шлюзу не лише за фактом передавання даних, а й за журналами подій та числовими показниками стану сервісів.

До ключових метрик шлюзу належать затримка встановлення MQTT-сесії, кількість активних підключень, інтенсивність потоку повідомлень, кількість відхилених з'єднань і статистика переходів брокерів між станами UP та DOWN. Для забезпечення ідентифікації потоків даних вводиться ієрархічна структура MQTT-топіків. Найбільш зручним є формат, у якому локація та ідентифікатор пристрою закладені в базовий шлях, а тип даних винесений в окремий сегмент. Така семантика дозволяє шлюзу, базі даних та графічному дашборду легко фільтрувати повідомлення за конкретною локацією чи пристроєм, а головне надійно ізолювати потоки сирих даних від керівного трафіку (рисунок 2.5).

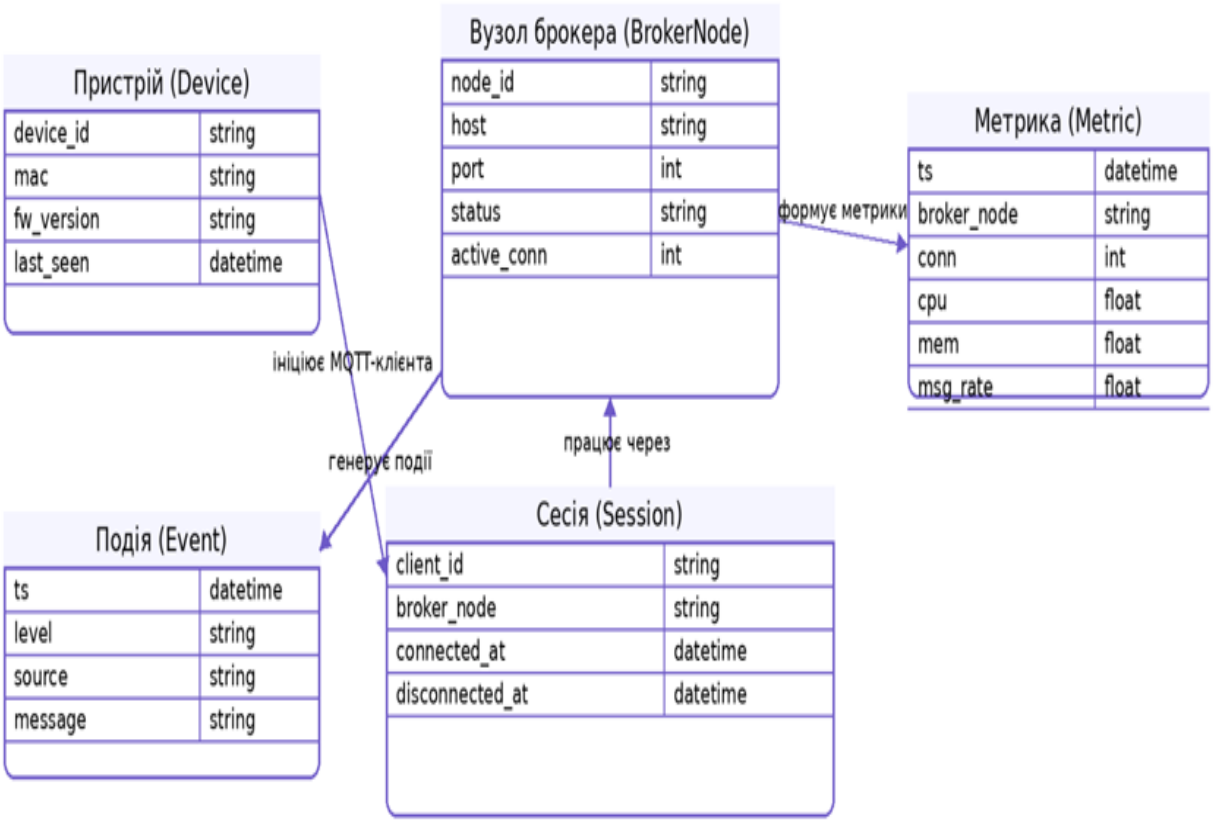


Рисунок 2.5 – Інфологічна модель внутрішніх даних IoT-шлюзу

Реляційну модель доцільно реалізувати в SQLite, тому що обсяг службових даних є невеликим, а локальне зберігання спрощує тестування та підготовку звітів. Таблиці відповідають сутностям вище: Device, BrokerNode, Session, Metric, Event. Схема зв'язків між сутностями представлена на рисунок 2.6.

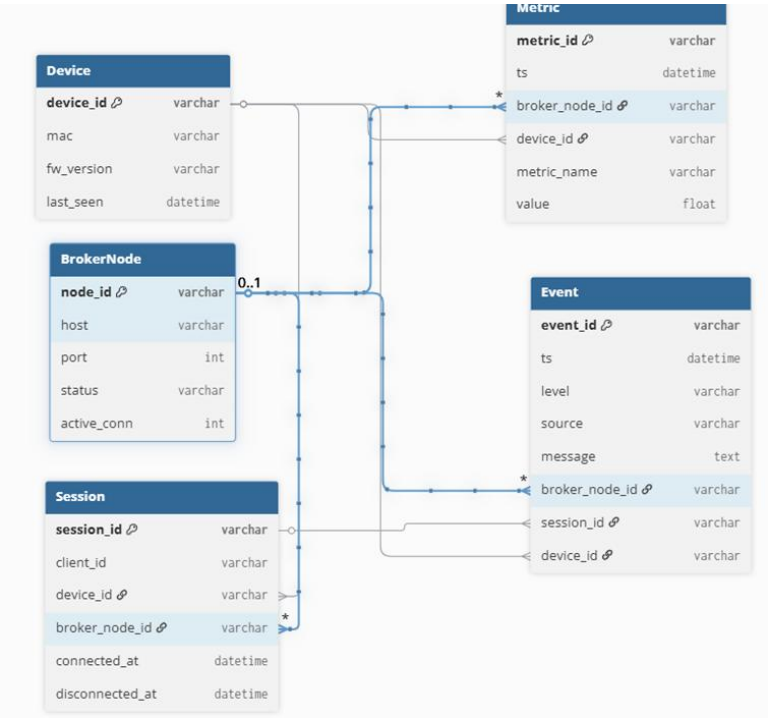


Рисунок 2.6 – ER-діаграма логічної структури бази даних

Основною таблицею для відстеження маршрутизації є Session, яка фіксує історію підключень клієнтів до конкретних брокерів. Стан самої інфраструктури та кінцевих пристроїв зберігається в таблицях BrokerNode та Device. Результати системного моніторингу зберігаються в таблицях Metric та Event, які пов’язані з пристроями й брокерами через зовнішні ключі. Така структура бази даних дозволяє легко аналізувати розподіл навантаження та контролювати стан усієї IoT-мережі.

3 ПРОГРАМНО-ТЕХНОЛОГІЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Вибір та аналіз програмно-апаратних засобів реалізації IoT-шлюзу

Для реалізації IoT-шлюзу було обрано апаратні та програмні засоби, які забезпечують стабільну роботу вузла, просте розгортання сервісів і можливість перевіряти сценарії перевантаження та відмови. Сам шлюз будується на Raspberry Pi (рисунок 3.1).

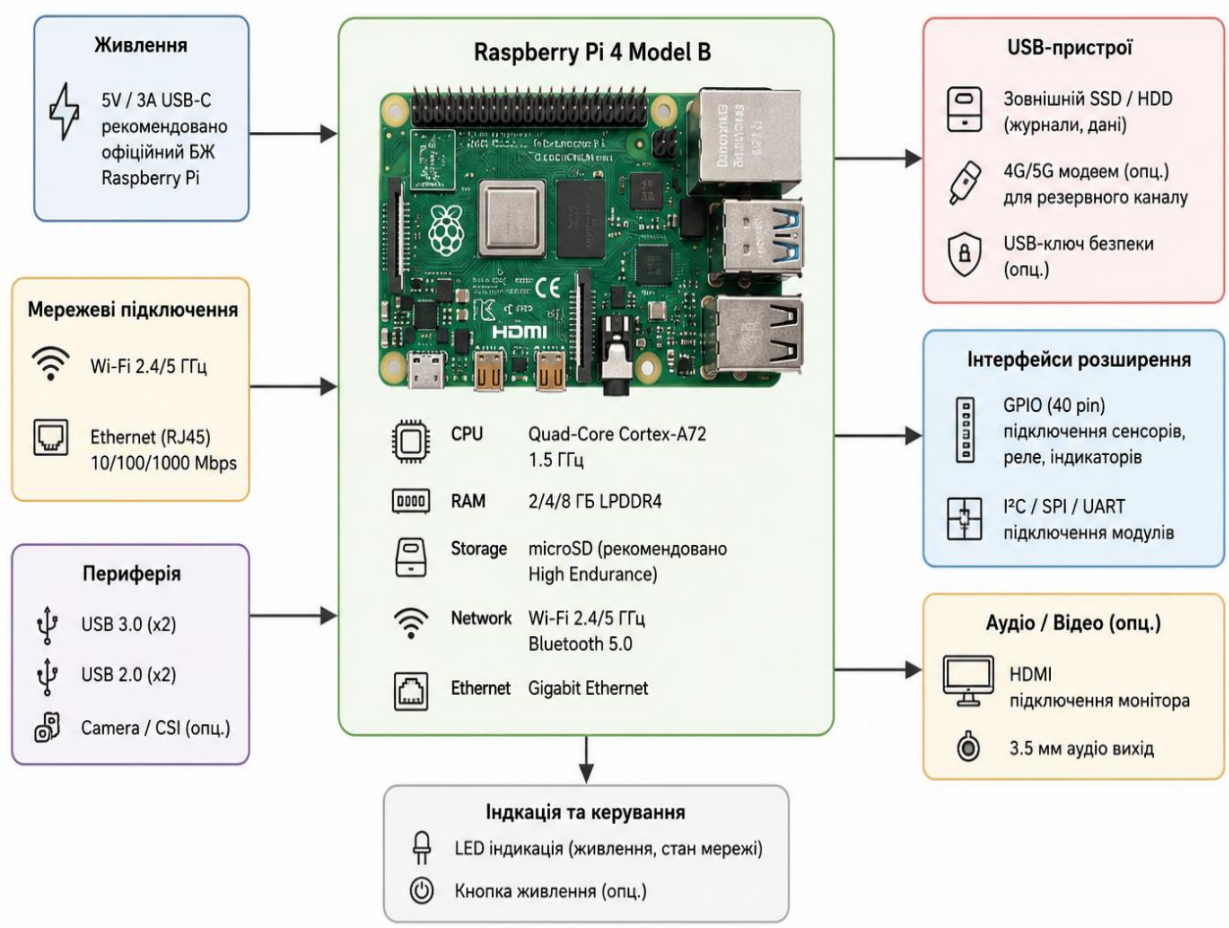


Рисунок 3.1 – Модуль розробки Raspberry Pi

Пристрої, що передають телеметрію, реалізовано на базі ESP32 (рисунок 3.2). Загальна логіка роботи полягає в тому, що ESP32 збирає дані й публікує їх через MQTT, а шлюз приймає підключення, виконує маршрутизацію та забезпечує роботу брокерів, балансувальника, журналювання і моніторингу.

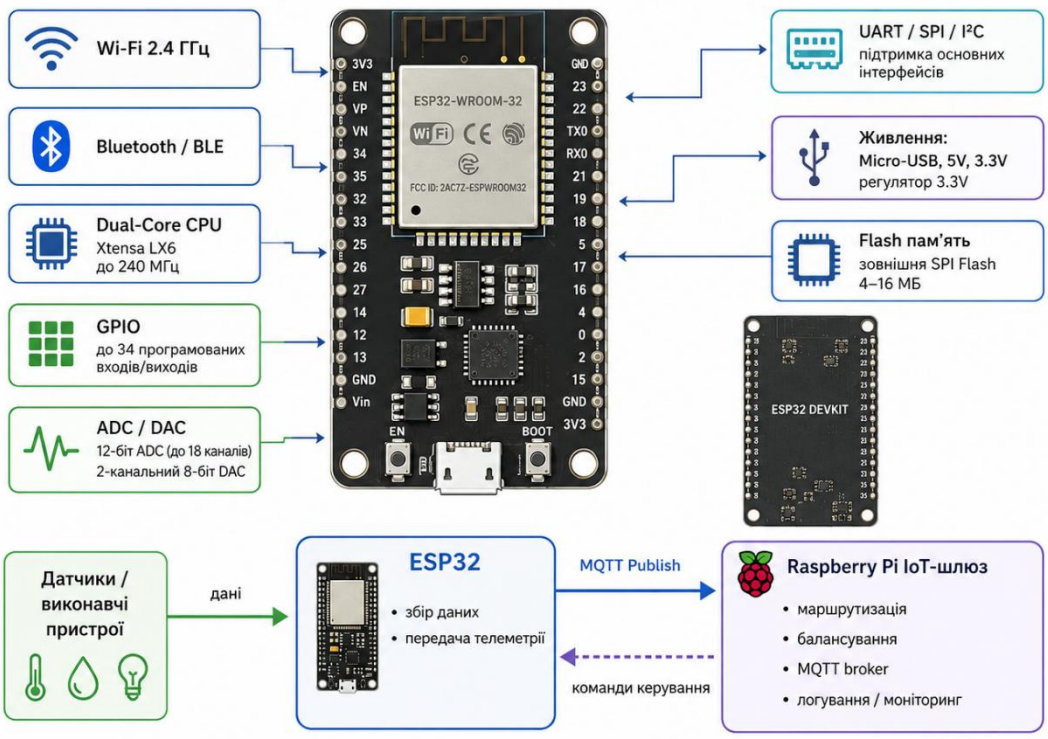


Рисунок 3.2 – Мікроконтролер ESP32

В програмній частині IoT-шлюзу для розгортання сервісів використовується Docker Engine, а для опису та запуску набору компонентів Docker Compose. Як базовий MQTT-брокер у пулі використовується рішення Eclipse Mosquitto (рисунок 3.3).

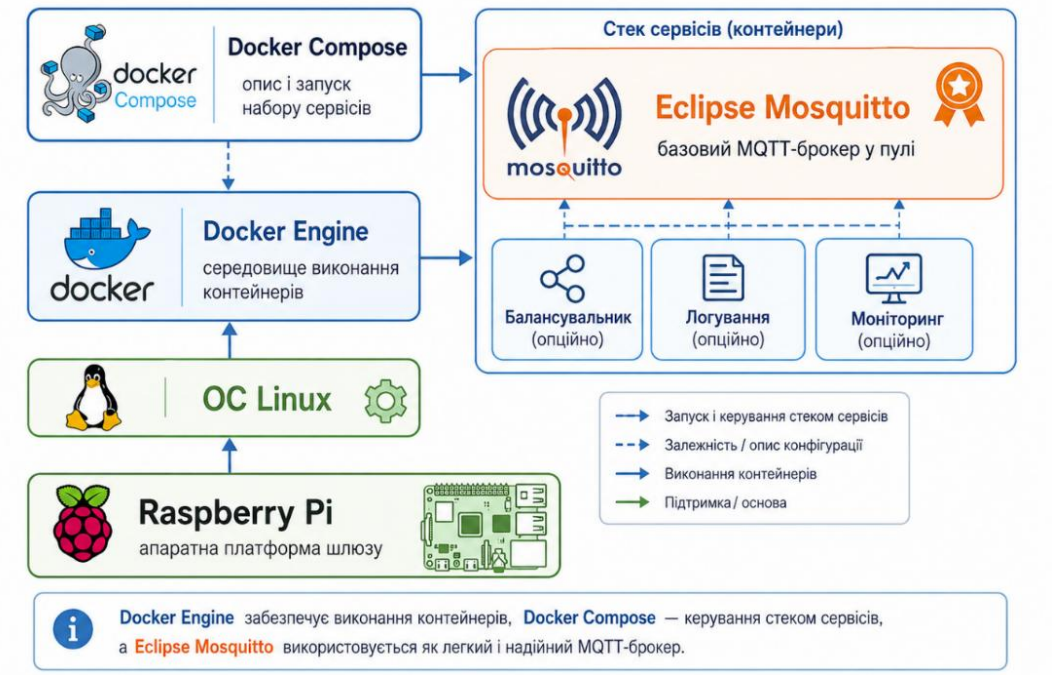


Рисунок 3.3 – Програмний стек для розгортання IoT-шлюзу

Контейнерний підхід забезпечує відтворюваність стенду, оскільки параметри Mosquitto, шлюзу, балансувальника, томів даних і мереж задаються у compose-файлах та зберігаються в репозиторії. Це максимально спрощує запуск, конфігурацію і масштабування пулу брокерів під час тестування алгоритмів балансування.

На етапі розробки IoT-шлюзу та його тестування було використано середовище віртуалізації Oracle VM VirtualBox з встановленою операційною системою Debian (рисунок 3.4). Такий формат зручний для імітації локальної мережі та моделювання навантаження, зокрема масових перепідключень пристроїв або відмов серверів, що дає змогу безпечно перевірити алгоритми балансування.

Додатковою перевагою є ізольованість середовища: стрес-тести не впливають на реальне фізичне обладнання й не створюють ризику його пошкодження.

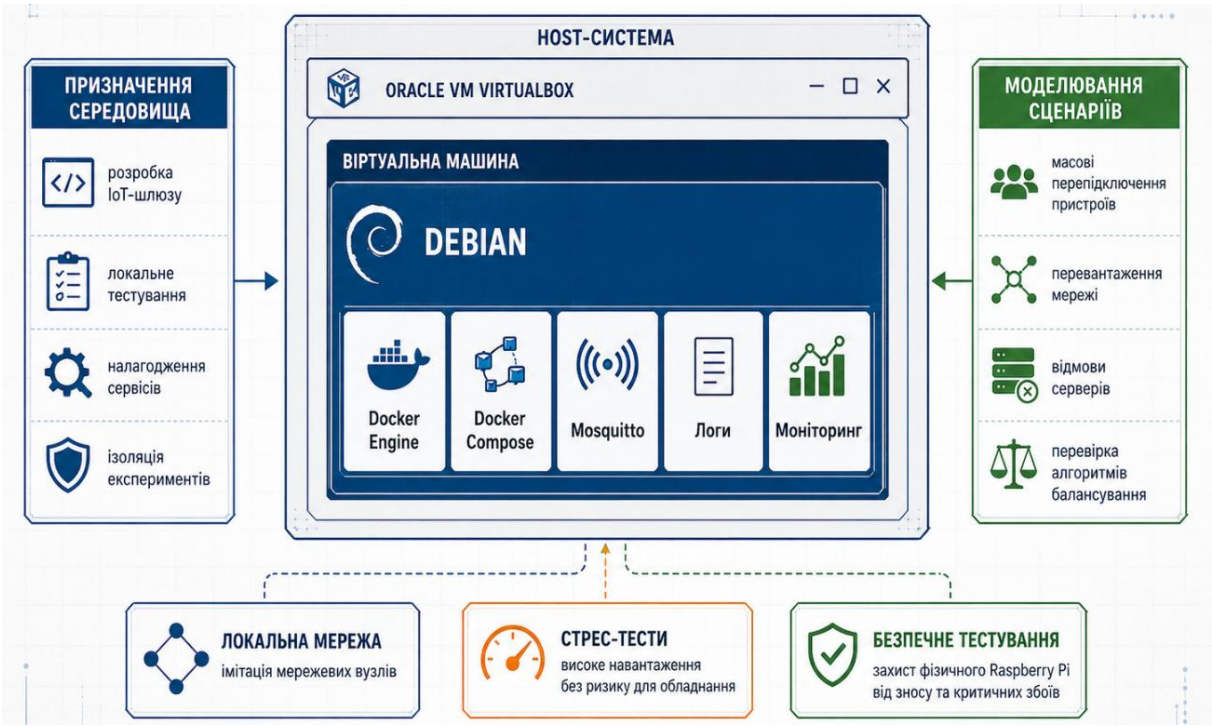


Рисунок 3.4 – Операційна система Debian у середовищі VirtualBox

Така архітектура дає змогу масштабувати шлюз, додавати нові політики маршрутизації та інтегрувати його в наявну IoT-інфраструктуру без суттєвих змін у клієнтських пристроях.

3.2 Програмно-технологічна реалізація IoT-шлюзу

Програмна реалізація шлюзу побудована модульно. Це дає змогу окремо тестувати компоненти маршрутизації, змінювати конфігурацію пулу брокерів під різну кількість IoT-пристроїв і вдосконалювати алгоритм вибору брокера без повної переробки системи.

На першому етапі практичної реалізації виконується встановлення та базова конфігурація середовища Docker Engine спільно з утилітою Docker Compose (рисунок 3.5).

```
       :~$ sudo apt install -y docker.io docker-compose
Installing:
  docker-compose  docker.io

Installing dependencies:
  containerd     iptables        libip6tc2                python3-protobuf
  criu           libcompel1      libmodule-find-perl      python3-pycriu
  docker-buildx  liberror-perl   libproc-processtable-perl runc
  docker-cli     libintl-perl    libsort-naturally-perl   tini
  git            libintl-xs-perl libterm-readkey-perl
  git-man        libip4tc2       needrestart

Suggested packages:
  containernetworking-plugins  xfsprogs          git-cvs
  docker-doc                   zfs-fuse          git-mediawiki
  aufs-tools                   | zfsutils-linux  git-svn
  btrfs-progs                  git-doc           firewalld
  cgroupfs-mount               git-email         needrestart-session
  debootstrap                  git-gui           | libnotify-bin
  rinse                        gitk
  rootlesskit                  gitweb
```

Рисунок 3.5 – Завантаження Docker Engine і Docker Compose

Після базового налаштування середовища наступним етапом є розробка конфігураційного файлу docker-compose.yml для автоматизованого розгортання MQTT-брокера Eclipse Mosquitto (рисунок 3.6).

```
services:
  broker1:
    image: eclipse-mosquitto:2
    container_name: broker1
    ports:
      - "1883:1883"
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf:ro
      - mosq1_data:/mosquitto/data
      - mosq1_log:/mosquitto/log
    restart: unless-stopped

  broker2:
    image: eclipse-mosquitto:2
    container_name: broker2
    ports:
      - "1884:1883"
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf:ro
      - mosq2_data:/mosquitto/data
      - mosq2_log:/mosquitto/log
    restart: unless-stopped

  gateway:
    image: haproxy:2.9
    container_name: mqtt-gateway
    ports:
      - "1885:1885"
```

Рисунок 3.6 – Створення конфігурації брокерів для Eclipse Mosquitto

Так як в Docker Compose передбачено монтування локальних налаштувань, наступним кроком є створення файлу mosquitto.conf. Для того щоб брокер приймав мережеві підключення від мікроконтролерів (ESP32) та проксі-модуля (рисунок 3.7).

```
:~/iot-gateway-lab/mosq1$ cat > mosquitto.conf <<'EOF'
listener 1883
allow_anonymous true

persistence true
persistence_location /mosquitto/data/

log_dest stdout
EOF
```

Рисунок 3.7 – Мінімальна конфігурація для запуску MQTT-брокера

Після розгортання пулу MQTT-брокерів налаштовується вузол маршрутизації. Як програмний балансувальник навантаження було обрано HAProxy, що є високопродуктивним TCP/HTTP проксі-сервером (рисунок 3.8).

```
:~/iot-gateway-lab/mosq1$ cat > haproxy.cfg <<'EOF'
global
  daemon
  maxconn 2048

defaults
  mode tcp
  timeout connect 5s
  timeout client  1m
  timeout server  1m

frontend mqtt_in
  bind *:1885
  default_backend mqtt_back

backend mqtt_back
  balance roundrobin
  option tcp-check
  default-server inter 2s fall 3 rise 2

  stick-table type ip size 200k expire 30m
  stick on src

  server b1 broker1:1883 check
  server b2 broker2:1883 check
EOF
```

Рисунок 3.8 – Налаштування правил розподілу MQTT-з’єднань

На наступному кроці здійснювався запуск програмних компонентів IoT-шлюзу у контейнерному середовищі Docker Compose (рисунок 3.9). В результаті розгортаються два екземпляри MQTT-брокера та контейнер шлюзу, який забезпечує балансування навантаження між брокерами. Перевірка списку активних контейнерів і аналіз журналу роботи підтверджують коректне функціонування сервісів та готовність системи до подальших експериментів.

```
~/iot-gateway-lab/mosq1$ sudo docker-compose up -d
[sudo] password for     :
[+] Running 7/7
 ✔ gateway Pulled                                                    5.8s
   ✔ 6e909acdb790 Pull complete                                      1.7s
   ✔ 14f01b12e34b Pull complete                                      1.0s
   ✔ dad818277bbc Pull complete                                      0.7s
   ✔ 9dd2eec60d9c Pull complete                                      2.4s
   ✔ a48d17b7c7ea Pull complete                                      1.8s
   ✔ 4f4fb700ef54 Pull complete                                      2.4s
[+] Running 3/3
 ✔ Container broker2       Running                                   0.0s
 ✔ Container broker1       Running                                   0.0s
 ✔ Container mqtt-gateway  Started                                   0.3s
:~/iot-gateway-lab/mosq1$ sudo docker-compose ps
NAME           IMAGE                  COMMAND                  SERVICE   CREATED
          STATUS              PORTS
broker1        eclipse-mosquitto:2   "/docker-entrypoint.…"   broker1   About an
 hour ago   Up About an hour   0.0.0.0:1883->1883/tcp, :::1883->1883/tcp
broker2        eclipse-mosquitto:2   "/docker-entrypoint.…"   broker2   18 minut
es ago      Up 18 minutes      0.0.0.0:1884->1883/tcp, :::1884->1883/tcp
mqtt-gateway   haproxy:2.9           "docker-entrypoint.s…"   gateway   16 secon
ds ago      Up 15 seconds      0.0.0.0:1885->1885/tcp, :::1885->1885/tcp
~/iot-gateway-lab/mosq1$ sudo docker-compose logs --tail=30 gateway
mqtt-gateway  | [NOTICE]   (1) : New worker (8) forked
mqtt-gateway  | [NOTICE]   (1) : Loading success.
~/iot-gateway-lab/mosq1$
```

Рисунок 3.9 – Запуск контейнерів MQTT-брокерів і шлюзу

Після цього створюється локальна база даних для збереження службових подій шлюзу. Це дозволяє розгорнути необхідні таблиці та розпочати безперервний запис системних логів, метрик навантаження та результатів роботи механізму перевірки стану брокерів health-check (рисунок 3.10).

```
:~/iot-gateway-lab/mosq1$ sqlite3 ~/iot-gateway-lab/db/gateway.db ".tables"
:~/iot-gateway-lab/mosq1$ sqlite3 ~/iot-gateway-lab/db/gateway.db <<'SQL'
CREATE TABLE IF NOT EXISTS BrokerNode (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  host TEXT NOT NULL,
  port INTEGER NOT NULL,
  status TEXT NOT NULL,
  last_check_ts TEXT
);

CREATE TABLE IF NOT EXISTS Session (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  client_id TEXT NOT NULL,
  broker_name TEXT NOT NULL,
  connected_ts TEXT NOT NULL,
  disconnected_ts TEXT
);

CREATE TABLE IF NOT EXISTS Event (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  ts TEXT NOT NULL,
  level TEXT NOT NULL,
  source TEXT NOT NULL,
  message TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS Device (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  device_id TEXT NOT NULL UNIQUE,
  type TEXT,
  last_seen_ts TEXT
);

CREATE TABLE IF NOT EXISTS Metric (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  device_id TEXT NOT NULL,
  metric TEXT NOT NULL,
  value REAL NOT NULL,
  ts TEXT NOT NULL
);
SQL
```

Рисунок 3.10 – SQL-скрипт ініціалізації БД шлюзу

Для збереження історії вимірювань створено окремий Python-скрипт (collector). Він підключається до балансувальника HAProxy, безперервно прослуховує MQTT-повідомлення від IoT-пристроїв і парсить їх на базові параметри ID вузла, назва метрики та її значення. Після обробки ці дані автоматично записуються у відповідні таблиці локальної бази SQLite (рисунок 3.11).

```
~/iot-gateway-lab/mosq1$ cat > ~/iot-gateway-lab/collector/collector.py <<'PY'
import os, json, sqlite3, time
import paho.mqtt.client as mqtt

MQTT_HOST = os.getenv("MQTT_HOST", "gateway")
MQTT_PORT = int(os.getenv("MQTT_PORT", "1885"))
TOPIC = os.getenv("MQTT_TOPIC", "tele/#")
DB_PATH = os.getenv("DB_PATH", "/data/gateway.db")

def db_exec(cur, sql, params=()):
    cur.execute(sql, params)

def on_connect(client, userdata, flags, rc, properties=None):
    client.subscribe(TOPIC)

def on_message(client, userdata, msg):
    topic = msg.topic
    payload = msg.payload.decode(errors="ignore").strip()

    # topic: tele/<device_id>/<metric>
    parts = topic.split("/")
    if len(parts) >= 3 and parts[0] == "tele":
        device_id = parts[1]
        metric = parts[2]
    else:
        return

    ts = time.strftime("%Y-%m-%d %H:%M:%S")
```

Рисунок 3.11 – Фрагмент коду скрипта для запису MQTT-повідомлень

Також додано сервіс collector у docker-compose.yml (рисунок 3.12).

```
  collector:
    build: ../collector
    container_name: mqtt-collector
    environment:
      MQTT_HOST: gateway
      MQTT_PORT: "1885"
      MQTT_TOPIC: "tele/#"
      DB_PATH: "/data/gateway.db"

    volumes:
      - ../db:/data
    depends_on:
      - gateway
    restart: unless-stopped

volumes:
```

Рисунок 3.12 – Фрагмент конфігурації для сервісу збору телеметрії

Після інтеграції колектора даних налаштовується безперервний моніторинг працездатності самих MQTT-брокерів. Для цього додано програмний модуль broker_stats.py (рисунок 3.13). До його логіки додано механізм активної перевірки доступності серверів на транспортному рівні TCP-check.

```
        ~/iot-gateway-lab/mosq1$ cat > ~/iot-gateway-lab/broker_stats/broker_stats.py <<'PY'
import time, sqlite3, os, socket

)B = os.getenv("DB_PATH", "/data/gateway.db")
INTERVAL = int(os.getenv("INTERVAL", "5"))
TIMEOUT = float(os.getenv("TCP_TIMEOUT", "1.0"))

SQL_ACTIVE_CONN = """
JPDATE BrokerNode
SET active_conn = (
  SELECT COUNT(*) FROM Session
  WHERE Session.broker_node_id = BrokerNode.node_id
    AND Session.disconnected_at IS NULL
),
last_check_ts = datetime('now');
"""

def tcp_check(host: str, port: int) -> bool:
    try:
        s = socket.create_connection((host, port), timeout=TIMEOUT)
        s.close()
        return True
    except Exception:
        return False

while True:
```

Рисунок 3.13 – Фрагмент коду модуля моніторингу стану брокерів та TCP-перевірки

Також додається сервіс broker_stats у docker-compose.yml (рисунок 3.14).

```yaml
broker-stats:
  build: ../broker_stats
  container_name: broker-stats
  environment:
    DB_PATH: "/data/gateway.db"
    INTERVAL: "5"
  volumes:
    - ../db:/data
  restart: unless-stopped
```

Рисунок 3.14 – Фрагмент конфігурації для сервісу збору статистики

Далі для відстеження активності клієнтів реалізовано програмний модуль session_watcher.py, який розбирає логи брокерів та зберігає історію їх з'єднань з IoT-вузлами у реляційних таблицях Session та Event (рисунок 3.15).

```
~/iot-gateway-lab/mosq1$ cat > ~/iot-gateway-lab/session_watcher/session_watcher.py <<'PY'

import re, sqlite3, time, sys, os

DB_PATH = os.path.expanduser("~/iot-gateway-lab/db/gateway.db")

RE_CONN = re.compile(r'^(broker\d)\s+\|\s+(\d+):.*New client connected.* as (\S+)')
RE_DISC = re.compile(r'^(broker\d)\s+\|\s+(\d+):.*Client (\S+).*\sdisconnected')

def ts_from_epoch(epoch: str) -> str:
    return time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(int(epoch)))

def main():
    con = sqlite3.connect(DB_PATH)
    cur = con.cursor()

    cur.execute("CREATE INDEX IF NOT EXISTS idx_sess_open ON Session(client_id, broker_node_id, dis
connected_at)")
    con.commit()

    for line in sys.stdin:
        line = line.strip()

        m = RE_CONN.match(line)
        if m:
            broker, epoch, client_id = m.group(1), m.group(2), m.group(3)
            ts = ts_from_epoch(epoch)
            session_id = f"{client_id}-{epoch}"
            device_id = client_id  # у прототипі: client_id == device_id

            cur.execute(
                "INSERT INTO Session(client_id, broker_node_id, connected_at, disconnected_at, sess
ion_id, device_id) "
                "VALUES (?,?,?,?,?,?)",
```

Рисунок 3.15 – Фрагмент коду модуля аналізу журналів MQTT-брокера

Для візуального контролю стану IoT-шлюзу розроблено веб-дашборд. На початковому етапі створено робочу директорію проекту та виконано перевірку структури локальної бази даних gateway.db. Після перевірки структури бази даних було створено ізольоване середовище Python і встановлено FastAPI разом із бібліотеками, потрібними для серверної частини веб-інтерфейсу (рисунок 3.16).

```
          ~/iot-gateway-lab/mosq1$ cd ~/iot-gateway-lab && mkdir -p dashboard/templates && cd dashbo
ard && pwd
/home,    a/iot-gateway-lab/dashboard
          ~/iot-gateway-lab/dashboard$ ls -lah ~/iot-gateway-lab/db/gateway.db && sqlite3 ~/iot-gate
way-lab/db/gateway.db ".tables"
-rw-r--r-- 1           240K Feb 26 12:35 /home/    /iot-gateway-lab/db/gateway.db
BrokerNode  Device      Event       Metric      Session
          ~/iot-gateway-lab/dashboard$ python3 -m venv .venv && source .venv/bin/activate && pip ins
tall fastapi uvicorn jinja2
Collecting fastapi
  Downloading fastapi-0.133.1-py3-none-any.whl.metadata (30 kB)
Collecting uvicorn
  Downloading uvicorn-0.41.0-py3-none-any.whl.metadata (6.7 kB)
Collecting jinja2
  Downloading jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting starlette>=0.40.0 (from fastapi)
  Downloading starlette-0.52.1-py3-none-any.whl.metadata (6.3 kB)
Collecting pydantic>=2.7.0 (from fastapi)
  Downloading pydantic-2.12.5-py3-none-any.whl.metadata (90 kB)
Collecting typing-extensions>=4.8.0 (from fastapi)
  Downloading typing_extensions-4.15.0-py3-none-any.whl.metadata (3.3 kB)
Collecting typing-inspection>=0.4.2 (from fastapi)
  Downloading typing_inspection-0.4.2-py3-none-any.whl.metadata (2.6 kB)
Collecting annotated-doc>=0.0.2 (from fastapi)
```

Рисунок 3.16 – Підготовка середовища для веб-дашборду

Після підготовки середовища створюється головний файл застосунку app.py. В ньому ініціалізується фреймворк FastAPI та налаштовується зв’язок із локальною базою даних gateway.db (рисунок 3.17).

```
(.venv)          ~/iot-gateway-lab/dashboard$ cat > ~/iot-gateway-lab/dashboard/app.py <<'PY'
import os, sqlite3
from typing import Any, Dict, List
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.templating import Jinja2Templates

DB_PATH = os.getenv("DB_PATH", os.path.expanduser("~/iot-gateway-lab/db/gateway.db"))

app = FastAPI(title="IoT Gateway Dashboard")
templates = Jinja2Templates(directory="templates")

def q(sql: str, params: tuple = ()) -> List[Dict[str, Any]]:
    con = sqlite3.connect(DB_PATH)
    con.row_factory = sqlite3.Row
    try:
        cur = con.execute(sql, params)
        return [dict(r) for r in cur.fetchall()]
    finally:
        con.close()
```

Рисунок 3.17 – Фрагмент коду ініціалізація веб-додатку FastAPI та підключення до бази даних

Після налаштування серверної логіки та доступу до даних виконується розробка користувацького інтерфейсу веб-дашборду (рисунок 3.18).

```
PY  return [](since,))ts_col} ASC;'conn','active_conn')er, value AS conn" if "broker" in c else Non
(.venv)         ~/iot-gateway-lab/dashboard$ cat > ~/iot-gateway-lab/dashboard/templates/index.htm
l <<'HTML'
<!doctype html>
<html lang="uk">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <title>MQTT Gateway Dashboard</title>

  <!-- Chart.js + Luxon adapter (для time scale) -->
  <script src="https://cdn.jsdelivr.net/npm/chart.js@4.4.3"></script>
  <script src="https://cdn.jsdelivr.net/npm/luxon@3"></script>
  <script src="https://cdn.jsdelivr.net/npm/chartjs-adapter-luxon@1"></script>

  <style>
    :root{
      --bg:#ffffff; --card:#fff; --border:#e6e6e6; --muted:#666;
      --shadow: 0 1px 10px rgba(0,0,0,.04);
      --radius: 14px;
    }
    body { font-family: system-ui, sans-serif; margin: 18px; background: var(--bg); }
    h1 { margin: 0 0 6px; font-size: 26px; }
    .muted { color:var(--muted); font-size: 12px; margin: 0 0 14px; }
    code { background:#f6f6f6; padding:2px 6px; border-radius:8px; }

    .topbar{ display:flex; gap:10px; align-items:center; flex-wrap:wrap; margin-bottom: 12px; }
    .pill{ border:1px solid var(--border); padding:6px 10px; border-radius:999px; font-size:12px; b
ackground:#fafafa; }
    .dot{ display:inline-block; width:8px; height:8px; border-radius:50%; margin-right:6px; vertica
l-align:middle; background:#999; }
    .dot.ok{ background:#22c55e; }
    .dot.bad{ background:#ef4444; }
```

Рисунок 3.18 – Розробка структури сторінки веб-моніторингу

Інтерфейс відображає основні показники роботи системи: стан сервісів, журнал подій, мережеву активність та інші параметри функціонування IoT-шлюзу.

3.3 Експериментальне тестування IoT-шлюзу

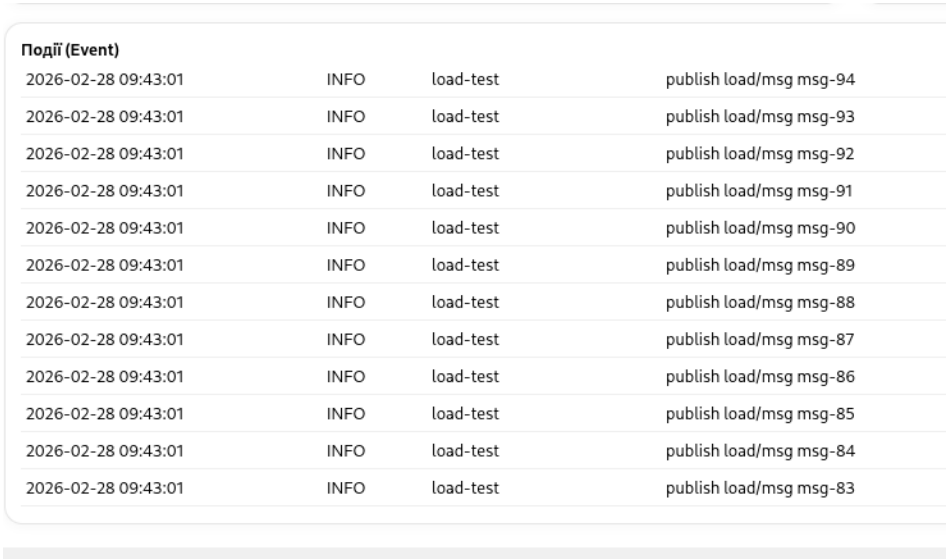
Після налаштування програмних і апаратних компонентів було проведено тестування роботи IoT-шлюзу. Для візуалізації процесів використовується розроблений веб-дашборд, який відкривається у локальному браузері та містить основні інформаційні блоки.

На першому етапі тестування перевіряється здатність шлюзу обробляти інтенсивний потік вхідних даних. За допомогою утиліти mosquitto_pub у циклі згенеровано 100 послідовних підключень і публікацій телеметрії, що імітують масове надсилання даних від IoT-сенсорів (рисунок 3.19).

```
    :~/iot-gateway-lab/mosq1$ for i in $(seq 1 100); do
  mosquitto_pub -h 127.0.0.1 -p 1885 -t load/msg -m "msg-$i" -q 0
  sqlite3 ~/iot-gateway-lab/db/gateway.db \
    "INSERT INTO Event(ts,level,source,message) VALUES(datetime('now'),'INFO','load-te
st','publish load/msg msg-$i');"
done
```

Рисунок 3.19 – Імітація пікового навантаження

Кожна дія додатково фіксується в системній таблиці Event локальної бази даних, що дозволяє оцінити швидкість реакції шлюзу на пікові сплески трафіку (рисунок 3.20).



Події (Event)

2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-94
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-93
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-92
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-91
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-90
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-89
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-88
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-87
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-86
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-85
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-84
2026-02-28 09:43:01	INFO	load-test	publish load/msg msg-83

Рисунок 3.20 – Фіксація подій навантаження

Далі було протестовано стійкість шлюзу до reconnect storm — багаторазових коротких підключень клієнтів до MQTT через gateway. Кожен клієнт виконує серію публікацій із паузами, через що в TCP/MQTT виникають часті події connect/disconnect (рисунок 3.21).

```
~/iot-gateway-lab/mosq1$ for i in $(seq 1 50); do
  (
    for r in $(seq 1 30); do
      mosquitto_pub -h 127.0.0.1 -p 1885 -i "reconn-$i" -t "load/reconn" -m "hi $i $r"
 -q 0
      sqlite3 ~/iot-gateway-lab/db/gateway.db "INSERT INTO Event(ts,level,source,messa
ge) VALUES(datetime('now'),'INFO','load-test','reconn reconn-$i hi $i $r');"
      sleep 0.2
    done
  ) &
done
```

Рисунок 3.21 – Скрипт імітації шторму перепідключень

Результати проведеного тестування та реакція системи на змодельоване навантаження в режимі реального часу відображаються в розробленому веб-інтерфейсі. Панель моніторингу показала, що балансувальник розподілив активні сесії між доступними брокерами, а події, пов’язані зі штормом перепідключень, були збережені в базі даних (рисунок 3.22).

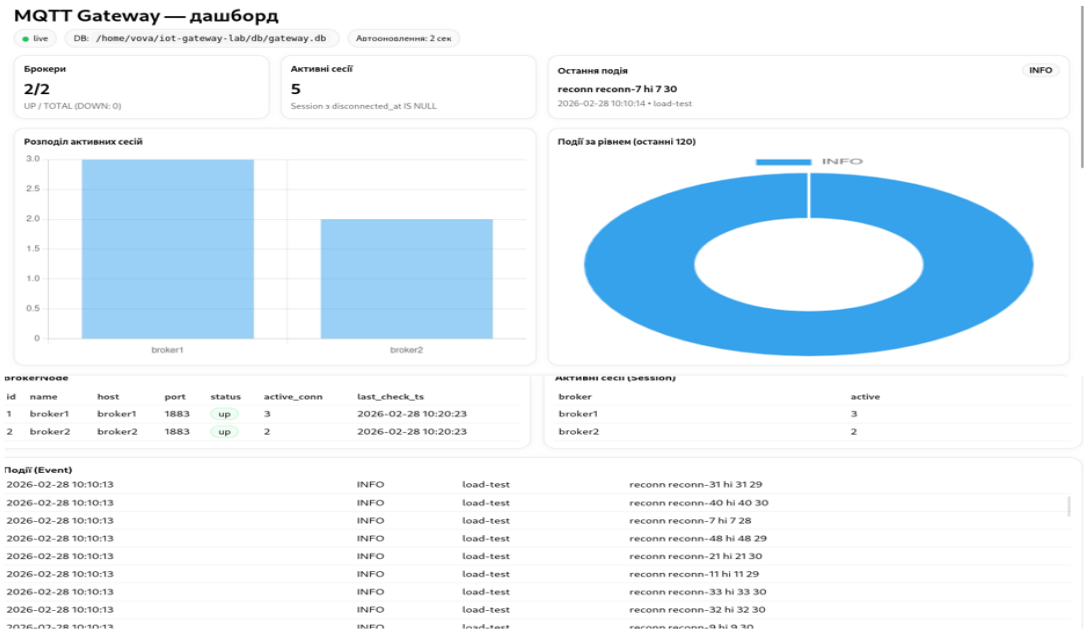


Рисунок 3.22 – Події після reconnect storm

Важливим етапом тестування стала перевірка відмовостійкості та автоматичного відновлення роботи шлюзу після збоїв. Для проведення цього експерименту було розроблено скрипт, який формує безперервний фоновий потік MQTT-публікацій, імітуючи активне функціонування сукупності підключених пристроїв (рисунок 3.23). Під час інтенсивної передачі даних було змодельовано аварійну ситуацію — примусово зупинено сервери.

```
:~/iot-gateway-lab/mosq1$ cd ~/iot-gateway-lab/mosq1

N=50
DUR_BEFORE=5
TOPIC="load/failover"

pids=()
for i in $(seq 1 $N); do
  (
    j=0
    while true; do
      j=$((j+1))
      mosquitto_pub -h 127.0.0.1 -p 1885 -i "load-$i" -t "$TOPIC" -m "i=$i j=$j" -q 0
>/dev/null 2>&1 || true
```

Рисунок 3.23 – Скрип відмовостійкості

Результати тестування та реакція системи на змодельовані аварійні ситуації відстежувалися у веб-інтерфейсі в режимі реального часу (рисунок 3.24).

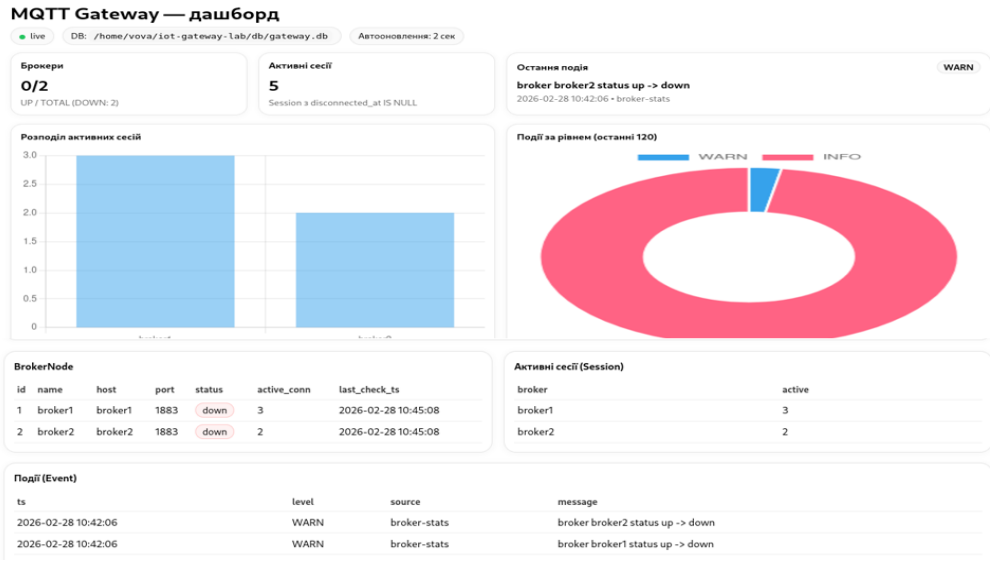


Рисунок 3.24 – Реєстрація подій в інтерфейсі

Після зупинки системи сервери було запущено повторно. Панель моніторингу показала, що модуль health-check розпізнав відновлення TCP-з’єднання, повернув брокери у стан up і зафіксував ці зміни в системному журналі (рисунок 3.25).

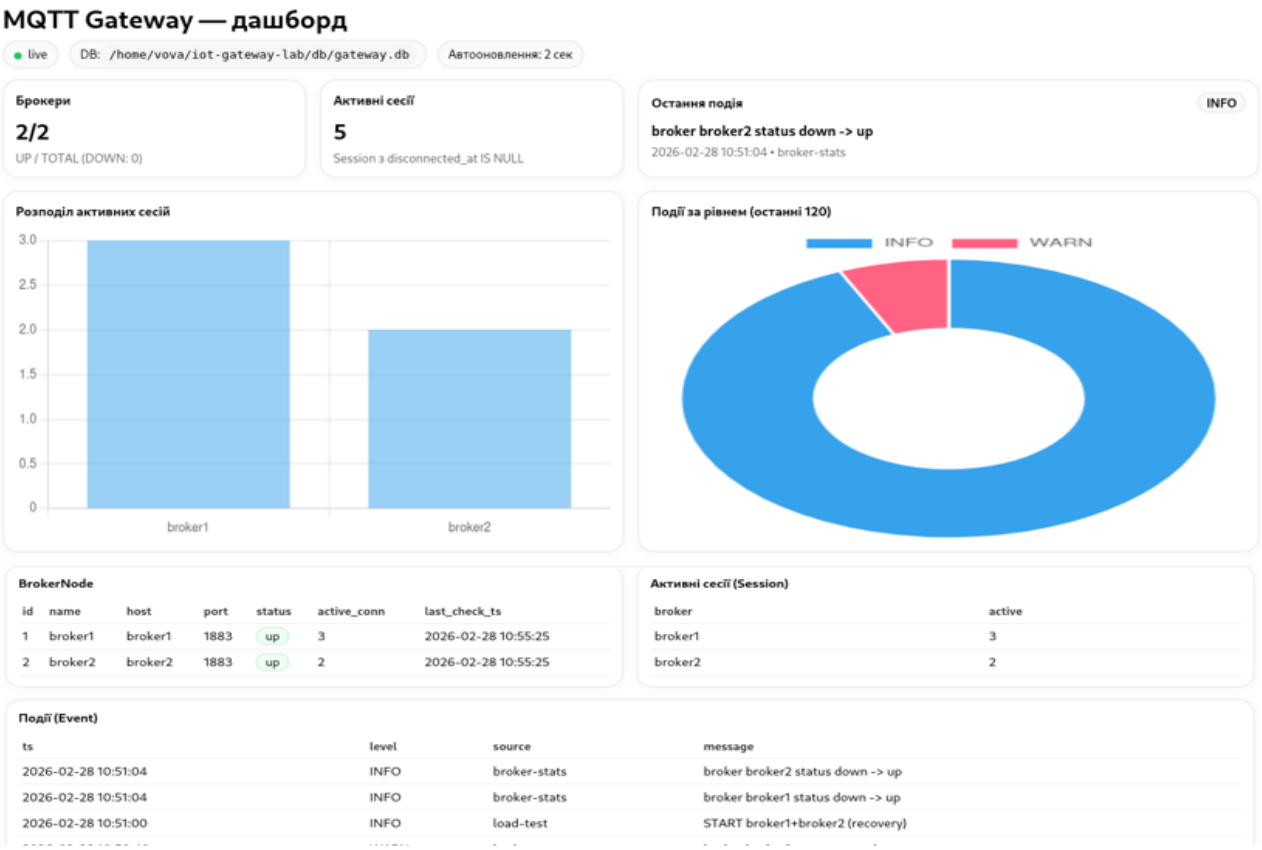


Рисунок 3.25 – Автоматичного самовідновлення шлюзу

Результати експериментів підтвердили, що IoT-шлюз на базі Raspberry Pi стабільно працює та коректно виконує основні функції. Отримані результати можна використати для подальшої оптимізації системи та вдосконалення її окремих компонентів. Веб-інтерфейс моніторингу підвищує практичну цінність розробки, оскільки дає змогу переглядати історію підключень, знаходити причини відмов і швидше реагувати на нештатні ситуації.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи отримано такі результати:

1. Проаналізовано предметну область IoT-систем, особливості протоколу MQTT, а також причини перевантаження, затримок і зниження відмовостійкості зі збільшенням кількості клієнтів.

2. Виконано огляд існуючих рішень для побудови MQTT-інфраструктури, зокрема Eclipse Mosquitto, EMQX, VerneMQ та HiveMQ. В результаті аналізу встановлено, що для реалізації прототипу локального IoT-шлюзу на мікрокомп'ютері доцільним є використання брокера Eclipse Mosquitto в поєднанні з HAProxy.

3. Розроблено загальну архітектуру програмно-апаратного IoT-шлюзу з балансуванням навантаження.

4. Розроблено алгоритмічне забезпечення системи, яке охоплює прийом MQTT-сесій, вибір брокера з пулу, періодичну перевірку доступності вузлів, оновлення їхнього стану, а також маршрутизацію команд керування до потрібного клієнта через відповідний брокер.

5. Розроблено інформаційне забезпечення IoT-шлюзу: визначено основні інформаційні потоки та побудовано логічну структуру локального сховища даних.

6. Обґрунтовано вибір програмно-апаратних засобів реалізації шлюзу.

7. Реалізовано програмно-технологічне забезпечення IoT-шлюзу, контейнеризоване середовище MQTT-брокерів і балансувальника, модулі збору телеметрії, моніторингу брокерів, аналізу журналів підключень і веб-дашборд для контролю стану системи.

8. Проведено експериментальне тестування системи в основних сценаріях: звичайний обмін повідомленнями, пікове навантаження, масові повторні підключення клієнтів, відмова брокера та його подальше відновлення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. MQTT Version 5.0. OASIS Standard. 2019. URL: https://www.oasis-open.org/standard/mqtt-v5-0-os/ (дата звернення: 15.05.2026).

2. ESP32-WROOM-32E & ESP32-WROOM-32UE Datasheet. Espressif Systems. Version 2.0. 2024. URL: https://documentation.espressif.com/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.html (дата звернення: 15.05.2026).

3. Raspberry Pi 4 Model B Datasheet. Raspberry Pi Ltd. 2024. URL: https://pip.raspberrypi.com/documents/RP-008341-DS-raspberry-pi-4-datasheet.pdf (дата звернення: 15.05.2026).

4. Docker Engine. Docker Docs. URL: https://docs.docker.com/engine/ (дата звернення: 15.05.2026).

5. Compose Specification. URL: https://www.compose-spec.io/ (дата звернення: 15.05.2026).

6. Eclipse Mosquitto Documentation. URL: https://mosquitto.org/documentation/ (дата звернення: 15.05.2026).

7. mosquitto.conf - configuration file for Mosquitto. URL: https://mosquitto.org/man/mosquitto-conf-5.html (дата звернення: 15.05.2026).

8. HAProxy version 2.9.15 - Configuration Manual. 2025. URL: https://docs.haproxy.org/2.9/configuration.html (дата звернення: 15.05.2026).

9. CREATE TABLE. SQLite Documentation. 2025. URL: https://www.sqlite.org/lang_createtable.html (дата звернення: 15.05.2026).

10. FastAPI Tutorial - User Guide. URL: https://fastapi.tiangolo.com/tutorial/ (дата звернення: 15.05.2026).

11. Kanellopoulos D., Sharma V. K. Dynamic Load Balancing Techniques in the IoT: A Review. Symmetry. 2022. Vol. 14, no. 12. Art. 2554. DOI: https://doi.org/10.3390/sym14122554.

12. Koziolek H., Grüner S., Rückert J. A Comparison of MQTT Brokers for Distributed IoT Edge Computing. Software Architecture : 14th European Conference,

ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings. Cham, 2020. P. 352–368. DOI: https://doi.org/10.1007/978-3-030-58923-3_23.

13. Morabito R., Petrolo R., Loscrì V., Mitton N. LEGIoT: A Lightweight Edge Gateway for the Internet of Things. Future Generation Computer Systems. 2018. Vol. 81. P. 1–15. DOI: https://doi.org/10.1016/j.future.2017.10.011.

14. Kenitar S. B., Salhaoui M., Arioua M., Younes A., Gonzalez A. G. Evaluation of the MQTT Protocol Latency over Different Gateways. Proceedings of the 3rd International Conference on Smart City Applications. 2018. Art. 87. DOI: https://doi.org/10.1145/3286606.3286864.

15. Fawwaz D. Z., Chung T.-M., Jaber M. M. Optimal Distributed MQTT Broker and Services Placement for SDN-Edge Based Smart City Architecture. Sensors. 2022. Vol. 22, no. 9. Art. 3431. DOI: https://doi.org/10.3390/s22093431.

16. Kurdi H., Thayananthan V. A Multi-Tier MQTT Architecture with Multiple Brokers Based on Fog Computing for Securing Industrial IoT. Applied Sciences. 2022. Vol. 12, no. 14. Art. 7173. DOI: https://doi.org/10.3390/app12147173.

17. Mishra B., Mishra B., Kertesz A. Stress-Testing MQTT Brokers: A Comparative Analysis of Performance Measurements. Energies. 2021. Vol. 14, no. 18. Art. 5817. DOI: https://doi.org/10.3390/en14185817.

18. El-Basioni B. M. M. A conceptual modeling approach of MQTT for IoT-based systems. Journal of Electrical Systems and Information Technology. 2024. Vol. 11. Art. 62. DOI: https://doi.org/10.1186/s43067-024-00181-x.

19. Shvaika A., Shvaika D., Landiak D., Artemchuk V. A distributed architecture for MQTT messaging: the case of TBMQ. Journal of Big Data. 2025. Vol. 12. Art. 224. DOI: https://doi.org/10.1186/s40537-025-01271-x.

20. Chai A. et al. DUA-MQTT: A Distributed High-Availability Message Communication Model for the Industrial Internet of Things. Sensors. 2025. Vol. 25, no. 16. Art. 5071. DOI: https://doi.org/10.3390/s25165071.

21. Ko K. I., Lee M. H. MQTT-Based Architecture for Real-Time Data Collection and Anomaly Detection in Smart Livestock Housing. Sensors. 2025. Vol. 25, no. 23. Art. 7186. DOI: https://doi.org/10.3390/s25237186.

22. Pham Le P. H., Do Q. N., Dinh T. Q., Pham H. T. N., Nguyen L. V. A comparative security analysis of MQTT brokers against DoS attacks. Journal on Information Security. 2026. Vol. 2026. Art. 5. DOI: https://doi.org/10.1186/s13635-026-00224-y.

23. Ford T. N., Gamess E., Ogden C. Performance Evaluation of Different Raspberry Pi Models as MQTT Servers and Clients. International Journal of Computer Networks & Communications. 2022. Vol. 14, no. 2. P. 1–18. DOI: https://doi.org/10.5121/ijcnc.2022.14201.

24. Seoane V., Garcia-Rubio C., Almenares F., Campo C. Performance evaluation of CoAP and MQTT with security support for IoT environments. Computer Networks. 2021. Vol. 197. Art. 108338. DOI: https://doi.org/10.1016/j.comnet.2021.108338.

25. Ai Y. et al. MIGS: A Modular Edge Gateway with Instance-Based Isolation for Heterogeneous Industrial IoT Interoperability. Sensors. 2026. Vol. 26, no. 1. Art. 314. DOI: https://doi.org/10.3390/s26010314.

26. Одарченко Р. С., Ткаліч О. П., Дика Н. В., Котелянець В. В. Studying the possibilities of communication protocols for the needs of IoT. Problems of Informatization and Control. 2017. № 3(59). P. 43–55. DOI: https://doi.org/10.18372/2073-4751.3.12810.

27. Павлюс В. П. Мережевий шлюз інтернету речей на основі одноплатного комп’ютера. Кібербезпека та комп’ютерно-інтегровані технології : зб. матеріалів наук. конф. Тернопіль, 2019. URL: http://cs.tneu.edu.ua/ZBIRNUK_2019.pdf (дата звернення: 15.05.2026).

28. Андрущак М. А. Мережевий шлюз Інтернету речей на базі Raspberry Pi : кваліфікаційна робота бакалавра : 123 Комп’ютерна інженерія / Хмельниц. нац. ун-т. Хмельницький, 2025. URL: https://elar.khmnu.edu.ua/items/b44b1497-9155-46ce-8909-699dcf1fb495 (дата звернення: 15.05.2026).

29. Гаврилюк О. В. Розробка шлюзу IoT на основі мікроконтролера Raspberry PI та інтерфейсу LTE/NB-IoT SIM7000E : дипломна робота ... бакалавра : 172 Телекомунікації та радіотехніка. Київ, 2025. 80 с. URL:

https://ela.kpi.ua/items/ffbf5d37-b85a-4407-a384-47c788b392cb (дата звернення: 15.05.2026).

30. Перетятько І. О. Система розподіленого управління для робототехнічних систем із застосуванням IoT-протоколів : дипломний проєкт ... бакалавра : 126 Інформаційні системи та технології. Київ, 2025. 88 с. URL: https://ela.kpi.ua/items/62316fcd-9475-4823-9d9a-9b72ebc93fc0 (дата звернення: 15.05.2026).

31. Мосійчук В. Архітектура IoT на основі контролю передачі даних та туманних обчислень : кваліфікаційна робота магістра : 123 Комп'ютерна інженерія / Хмельниц. нац. ун-т. Хмельницький, 2025. URL: https://elar.khmnu.edu.ua/items/2c76cfb8-f8a7-49c7-a1e1-d6a74ac5e028 (дата звернення: 15.05.2026).

32. Hovorushchenko T., Baranovskyi V., Ivanov O., Hnatchuk A. Subsystem for Monitoring Atmospheric Air Quality in the Cyber-Physical System "Smart City". Computer Systems and Information Technologies. 2024. № 1. P. 17–26. DOI: https://doi.org/10.31891/csit-2024-1-2.

33. Haidai A., Klymenko I. Method of Load Balancing in Distributed Three-Layer IoT Architecture. Information, Computing and Intelligent Systems. 2024. № 4. P. 60–68. DOI: https://doi.org/10.20535/2786-8729.4.2024.311595.

34. Кривіцький Б. С. Система постачання туманних послуг для керування даними Інтернету речей : кваліфікаційна робота магістра : 123 Комп'ютерна інженерія / Хмельниц. нац. ун-т. Хмельницький, 2025. URL: https://elar.khmnu.edu.ua/items/355858b6-1dbe-46cf-9289-02055bbd7671 (дата звернення: 15.05.2026).

35. Коломієць О. Принципи оцінки коефіцієнту продуктивності IoT-системи. ПОЛІТ. Сучасні проблеми науки. Сучасні інформаційні та комунікаційні технології в авіації : тези доповідей XXV Міжнар. наук.-практ. конф. здобувачів вищої освіти і молодих учених, Київ, 1–4 квіт. 2025 р. Київ : Держ. ун-т «Київський авіаційний інститут», 2025. С. 58–59. URL:

https://nau.edu.ua/site/variables/docs/docsmenu/studnauka/polit2025/Polit-2025_FKNT.pdf (дата звернення: 15.05.2026).

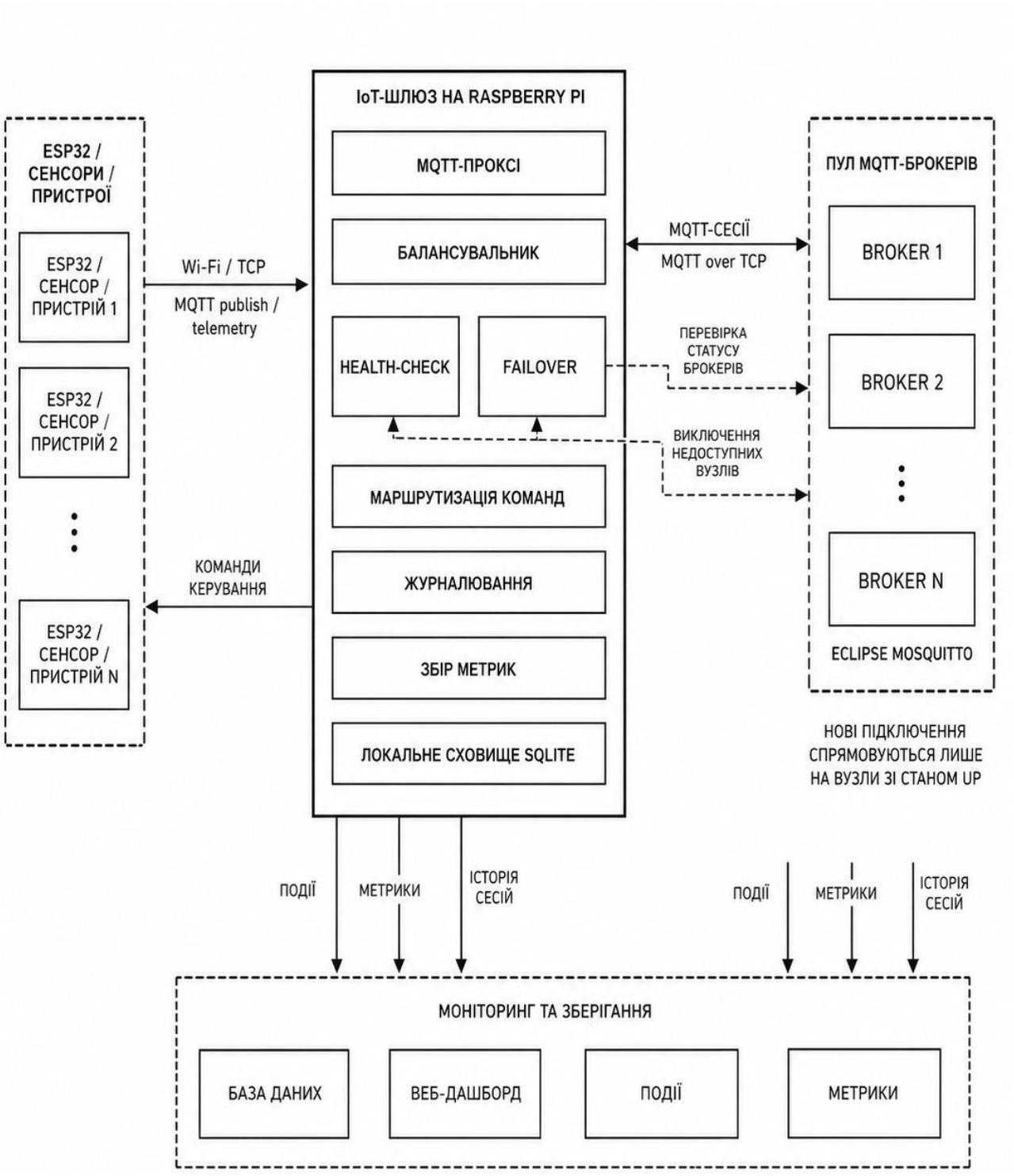
36. Строюк Б. Б. Система безпечного обміну даними в IoT : магістерська дисертація : 126 Інформаційні системи та технології. Київ, 2022. URL: https://ela.kpi.ua/items/e681dfaa-72f4-4595-9e95-9516e81d2dff (дата звернення: 15.05.2026).

37. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп'яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп'ютерні науки» спеціальності 122 «Комп'ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

38. Раківський Р.З. Програмно-апаратний шлюз IoT з балансуванням навантаження для протоколу MQTT // Науковий простір: аналіз, сучасний стан, тренди та перспективи : матеріали IX Всеукраїнської студентської наукової конференції, 19 червня 2026 р., Київ, Україна. Київ, 2026.

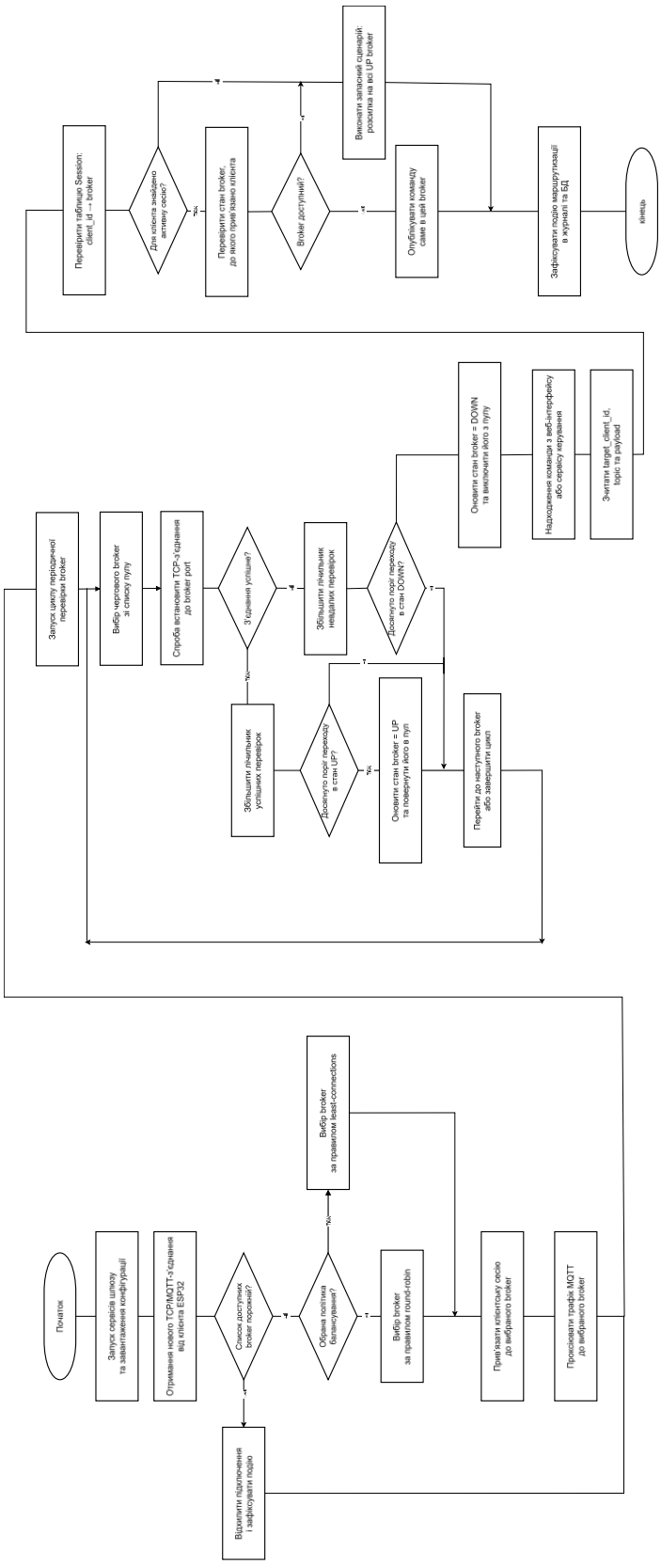
Додаток А

Структурна схема програмно-апаратного IoT-шлюзу з балансуванням MQTT



Додаток Б

Схема деталізованого алгоритму роботи IoT-шлюзу



Додаток В

Псевдокоди основних програмних модулів системи

В.1 Псевдокод розгортання контейнерного середовища IoT-шлюзу

```
АЛГОРИТМ DeployIoTGatewayEnvironment
ПОЧАТОК
    створити сервіс broker1 на базі eclipse-mosquitto
    створити сервіс broker2 на базі eclipse-mosquitto
    створити сервіс gateway на базі HAProxy
    створити сервіс collector для збору MQTT-телеметрії
    створити сервіс broker-stats для перевірки стану broker

    для broker1 і broker2:
        підключити конфігураційний файл mosquitto.conf
        підключити томи для даних і журналів
        увімкнути автоматичний перезапуск
    кінець для

    для gateway:
        підключити конфігураційний файл haproxy.cfg
        встановити залежність від broker1 і broker2
    кінець для

    для collector:
        передати параметри MQTT_HOST, MQTT_PORT, MQTT_TOPIC, DB_PATH
        встановити залежність від gateway
    кінець для

    для broker-stats:
        передати параметри DB_PATH, INTERVAL
        підключити том з базою даних
    кінець для

    створити томи mosq1_data, mosq1_log, mosq2_data, mosq2_log
КІНЕЦЬ
```

В.2 Псевдокод ініціалізації MQTT-брокера

```
АЛГОРИТМ InitMosquittoBroker
ПОЧАТОК
    встановити порт прослуховування 1883
    дозволити анонімні підключення
    увімкнути збереження стану broker
    задати каталог збереження даних
    спрямувати журнал у stdout
КІНЕЦЬ
```

В.3 Псевдокод балансування MQTT-з’єднань

```
АЛГОРИТМ ConfigureMQTTLoadBalancing
ПОЧАТОК
    ініціалізувати глобальні параметри HAProxy
```

```
    встановити максимальну кількість з'єднань

    налаштувати режим TCP
    задати таймаут підключення
    задати таймаут клієнта
    задати таймаут сервера

    створити frontend mqtt_in
    прив'язати його до порту 1885
    призначити backend mqtt_back

    створити backend mqtt_back
    встановити політику balance = roundrobin
    увімкнути option tcp-check
    встановити параметри перевірки inter, fall, rise

    створити stick-table для прив'язки клієнтів за IP-адресою
    увімкнути правило stick on src

    додати сервер b1 -> broker1:1883
    додати сервер b2 -> broker2:1883
КІНЕЦЬ
```

В.4 Псевдокод ініціалізації локальної бази даних

```
АЛГОРИТМ InitGatewayDatabase
ПОЧАТОК
    відкрити базу даних gateway.db

    якщо таблиця BrokerNode не існує
        створити таблицю BrokerNode
    кінець якщо

    якщо таблиця Session не існує
        створити таблицю Session
    кінець якщо

    якщо таблиця Event не існує
        створити таблицю Event
    кінець якщо

    якщо таблиця Device не існує
        створити таблицю Device
    кінець якщо

    якщо таблиця Metric не існує
        створити таблицю Metric
    кінець якщо

    зберегти зміни
    закрити базу даних
КІНЕЦЬ
```

В.5 Псевдокод модуля збору телеметрії

```
АЛГОРИТМ CollectorMain
ПОЧАТОК
    зчитати MQTT_HOST, MQTT_PORT, MQTT_TOPIC, DB_PATH
    підключитися до MQTT broker через gateway
    підписатися на topic tele/#

    ПОКИ клієнт активний ВИКОНУВАТИ
        чекати нове повідомлення MQTT
        отримати topic і payload

        розділити topic за символом "/"
        якщо topic має формат tele/device_id/metric
            зчитати device_id
            зчитати metric
        інакше
            пропустити повідомлення
        кінець якщо

        встановити поточний timestamp
        спробувати перетворити payload у JSON
            якщо успішно
                зчитати value
                якщо в JSON є ts
                    використати ts
                кінець якщо
            інакше
                спробувати перетворити payload у число
                якщо перетворення неуспішне
                    пропустити повідомлення
                кінець якщо
        кінець спроби

        відкрити базу даних
        додати пристрій у Device, якщо його ще немає
        оновити last_seen_ts для device_id
        вставити новий запис у Metric
        вставити службову подію у Event
        зберегти зміни
        закрити базу даних
    КІНЕЦЬ ПОКИ
КІНЕЦЬ
```

В.6 Псевдокод модуля моніторингу стану broker

```
АЛГОРИТМ BrokerStatsMain
ПОЧАТОК
    зчитати DB_PATH
    зчитати INTERVAL
    зчитати TCP_TIMEOUT

    ПОКИ істина ВИКОНУВАТИ
        відкрити базу даних
        оновити кількість активних сесій для кожного broker
        вибрати всі broker з таблиці BrokerNode

        для кожного broker
            виконати TCP-перевірку host:port
```

```
            якщо TCP-перевірка успішна
                new_status := up
            інакше
                new_status := down
            кінець якщо

            якщо old_status відрізняється від new_status
                оновити status у BrokerNode
                додати подію зміни статусу у Event
            кінець якщо

            додати в Metric значення active_conn
            додати в Metric значення status
        кінець для

        зберегти зміни
        закрити базу даних
        чекати INTERVAL секунд
    КІНЕЦЬ ПОКИ
КІНЕЦЬ
```

В.7 Псевдокод модуля відстеження сесій клієнтів

```
АЛГОРИТМ SessionWatcherMain
ПОЧАТОК
    відкрити базу даних
    створити індекс для швидкого пошуку відкритих сесій

    ПОКИ надходять рядки журналу ВИКОНУВАТИ
        зчитати поточний рядок

        якщо рядок відповідає шаблону нового підключення
            визначити broker
            визначити epoch
            визначити client_id
            перетворити epoch у timestamp
            сформувати session_id
            device_id := client_id

            вставити новий запис у Session
            вставити подію connect у Event
            зберегти зміни
        кінець якщо

        якщо рядок відповідає шаблону відключення
            визначити broker
            визначити epoch
            визначити client_id
            перетворити epoch у timestamp
            device_id := client_id

            знайти останню відкриту сесію цього клієнта
            оновити поле disconnected_at
            вставити подію disconnect у Event
            зберегти зміни
        кінець якщо
    КІНЕЦЬ ПОКИ
```

```
КІНЕЦЬ
```

В.8 Псевдокод серверної частини веб-дашборду

```
АЛГОРИТМ DashboardServerMain
ПОЧАТОК
    зчитати DB_PATH
    створити FastAPI-додаток
    ініціалізувати шаблони Jinja2

    визначити функцію q(sql, params)
        відкрити БД
        виконати запит
        повернути список рядків
        закрити БД
    кінець функції

    визначити функцію one(sql, params)
        відкрити БД
        виконати запит
        повернути один рядок або NULL
        закрити БД
    кінець функції

    визначити функцію cols(table)
        відкрити БД
        зчитати опис полів таблиці
        повернути множину назв полів
        закрити БД
    кінець функції

    маршрут "/"
        повернути HTML-шаблон index.html
    кінець маршруту

    маршрут "/api/brokers"
        повернути всі записи BrokerNode
    кінець маршруту

    маршрут "/api/sessions_active"
        повернути кількість активних Session по broker
    кінець маршруту

    маршрут "/api/events"
        повернути останні записи Event
    кінець маршруту

    маршрут "/api/summary"
        обчислити кількість broker у станах up/down
        обчислити кількість активних сесій
        вибрати останню подію
        вибрати час останньої перевірки
        повернути зведену структуру JSON
    кінець маршруту

    маршрут "/api/metrics/conn"
        визначити наявну схему таблиці Metric
        якщо є поле conn
```

```
            повернути часовий ряд conn
        інакше якщо є metric_name і value
            повернути часовий ряд active_conn
        інакше
            повернути порожній список
        кінець якщо
    кінець маршруту
КІНЕЦЬ
```

В.9 Псевдокод логіки оновлення веб-інтерфейсу

```
АЛГОРИТМ DashboardClientRefresh
ПОЧАТОК
    завантажити Chart.js
    ініціалізувати порожні графіки і таблиці

    визначити функцію refresh()
        отримати /api/summary
        оновити KPI broker, active_sessions, last_event

        отримати /api/brokers
        оновити таблицю broker і статуси up/down

        отримати /api/sessions_active
        оновити таблицю активних сесій
        оновити стовпчикову діаграму розподілу сесій

        отримати /api/events
        оновити таблицю подій
        підрахувати кількість подій за рівнями
        оновити кругову діаграму рівнів подій

        отримати /api/metrics/conn
        якщо дані існують
            згрупувати їх за broker
            оновити лінійний графік conn у часі
        інакше
            показати повідомлення про відсутність даних
        кінець якщо
    кінець функції

    викликати refresh() при відкритті сторінки
    кожні 2 секунди викликати refresh()
КІНЕЦЬ
```

Додаток Г

Апробація отриманих результатів

ДОВІДКА

ПРО ПРИЙНЯТТЯ ТЕЗ ДО ПУБЛІКАЦІЇ

10.06.2026

Шановний(і) авторе(и):
Раківський Роман Зиновійович,

Організаційний комітет з радістю повідомляє, що тези «ПРОГРАМНО-АПАРАТНИЙ ШЛЮЗ IoT З БАЛАНСУВАННЯМ НАВАНТАЖЕННЯ ДЛЯ ПРОТОКОЛУ MQTT» прийняті до публікації в збірнику за матеріалами IX Всеукраїнської студентської наукової конференції «Науковий простір: аналіз, сучасний стан, тренди та перспективи» (19.06.2026, м. Київ, Україна).

Опубліковані тези будуть доступні з 19.06.2026 за посиланням:
https://archive.liga.science/index.php/conference-proceedings/issue/view/ukr-19.06.2026

Електронні сертифікати учасників конференції та подяки науковим керівникам будуть доступні з 19 червня. Розсилка замовлених друкованих примірників, сертифікатів та подяк відбудеться до 3 липня.

Конференцію зареєстровано Державною науковою установою «УкрІНТЕІ» в базі даних науково-технічних заходів України та інформаційному бюлетені «План проведення наукових, науково-технічних заходів в Україні» (Посвідчення № 124 від 26.01.2026).

З повагою,

Директор Молодіжної наукової ліги
Голова оргкомітету конференції
ІГОР КОРЕНЮК

ПРОГРАМНО-АПАРАТНИЙ ШЛЮЗ IoT З БАЛАНСУВАННЯМ НАВАНТАЖЕННЯ ДЛЯ ПРОТОКОЛУ MQTT

Раківський Роман Зиновійович
Бакалавр спеціальності 122 – Комп'ютерні науки
Західноукраїнський національний університет, Україна
Науковий керівник: Осолінський Олександр Романович
к.т.н., доцент кафедри інформаційно-обчислювальних систем і управління
Західноукраїнський національний університет, Україна

Вступ

Інтернет речей сьогодні активно використовується в системах моніторингу, розумного дому, локальної телеметрії та промислового контролю. В таких рішеннях кінцеві вузли, такі як мікроконтролери ESP32, періодично передають вимірювання на центральний сервер або шлюз. Для обміну повідомленнями часто застосовують MQTT [1].

Разом з тим збільшення кількості клієнтів створює нові проблеми. Один MQTT-брокер може стати вузьким місцем, особливо під час пікових публікацій, масових перепідключень після збоїв Wi-Fi або тимчасової відмови сервісу. В таких умовах важливо приймати телеметрію, забезпечити розподіл підключень, контроль доступності брокерів, фіксацію подій та спостереження за станом системи. Тому актуальною є розробка програмно-апаратного IoT-шлюзу, який приховує внутрішню структуру брокерів від клієнтів і надає їм одну стабільну точку входу.

Архітектура системи

Основою запропонованого модуля є шлюз на базі Raspberry Pi, який виконує роль центрального вузла маршрутизації та моніторингу (рисунок 1). Кінцеві пристрої ESP32 передають телеметрію через Wi-Fi/TCP і працюють з однією IP-адресою шлюзу. Це спрощує налаштування мікроконтролерів, оскільки вони не повинні знати про кількість брокерів, їхні адреси або поточний стан.

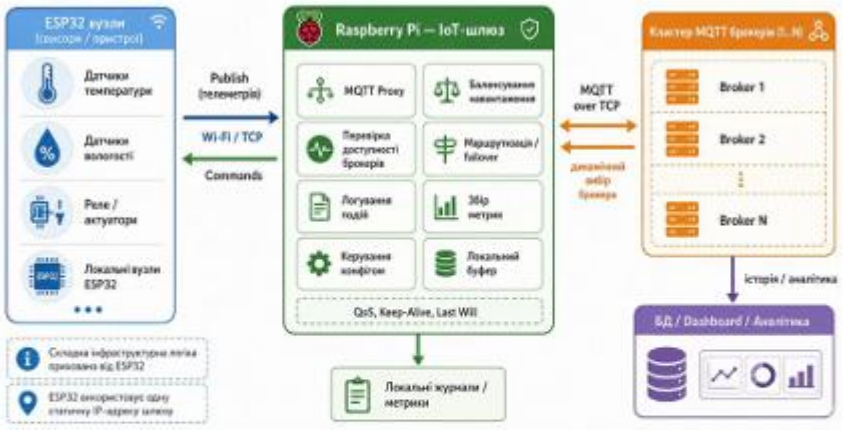


Рис. 1. **Загальна структура IoT-шлюзу з балансуванням MQTT**

В структурі шлюзу виділено кілька взаємопов'язаних компонентів. MQTT-проксі приймає TCP-з'єднання від клієнтів і передає їх до одного з брокерів у пулі. Балансувальник навантаження розподіляє нові сесії між доступними брокерами, а механізм прив'язки сесії зменшує кількість нестабільних переходів клієнта між вузлами. Модуль health-check періодично перевіряє працездатність брокерів і не допускає направлення нових підключень на вузол, який перестав відповідати.

Для збереження службової інформації використано локальне сховище SQLite. Така структура дає змогу бачити не лише факт передавання повідомлень, а і історію роботи інфраструктури, коли клієнт підключився, який брокер було обрано, коли виникли помилки та як змінювався стан вузлів. Для адміністратора це важливо, тому що причини затримок або втрату доступності не завжди видно з боку кінцевого пристрою.

Спочатку на шлюзі йде старт сервісів, перевірка конфігурації, вибір брокера для нової MQTT-сесії, оновлення стану вузлів за результатами health-check і маршрутизація команд керування. Якщо потрібний ESP32 має активну сесію, команда публікується саме у той брокер, через який він підключений. Якщо клієнт недоступний або відповідний брокер перебуває у стані DOWN, система

може відхилити команду або використати резервну публікацію через доступні вузли.

Програмна реалізація та тестування

Програмна частина реалізована в контейнерному середовищі. Для розгортання сервісів використано Docker Engine і Docker Compose, що дозволяє описати MQTT-брокери, балансувальник, колектори телеметрії, модуль статистики та веб-інтерфейс у єдиній конфігурації [2], [3]. Як брокер застосовано Eclipse Mosquitto [4]. Для розподілу TCP-з'єднань використано HAProxy [5].

Розроблена система містить кілька допоміжних програмних модулів. Collector підписується на телеметрійні MQTT-топіки, зчитує повідомлення від пристроїв і записує значення у базу даних. Модуль broker_stats виконує TCP-перевірку брокерів, оновлює їхні статуси UP/DOWN та формує метрики активних з'єднань. Session_watcher аналізує журнали Mosquitto і зберігає історію підключень та відключень клієнтів. Для відображення стану системи створено веб-дашборд (рисунок 2) на базі FastAPI, який показує кількість брокерів, активні сесії, журнал подій і графіки навантаження.

Рис. 2. **Веб-дашборд після автоматичного відновлення роботи брокерів**

Експериментальне тестування проводилося за наступними сценаріями: звичайний обмін повідомленнями, пікове навантаження, шторм перепідключень, примусова зупинка брокера та його подальше відновлення. Під час навантажувального тесту генерувалася серія MQTT-публікацій, що імітувала активну передачу даних від сенсорів. Після зупинки брокера шлюз припиняє направляти на нього нові сесії, а після відновлення автоматично повертає вузол у пул доступних серверів.

Отримані результати показали, що балансувальник розподіляв активні сесії між брокерами, модуль health-check коректно реєстрував зміну стану вузлів, а локальна база даних накопичувала події та метрики для подальшого аналізу.

Висновки. Було розроблено програмно-апаратний IoT-шлюз з балансуванням навантаження для протоколу MQTT. Запропонована архітектура поєднує ESP32-вузли, Raspberry Pi, пул брокерів Eclipse Mosquitto, HAProxy, локальну базу SQLite та веб-дашборд моніторингу. Рішення забезпечує єдину точку входу для MQTT-клієнтів, розподіл сесій між брокерами, автоматичне виключення недоступних вузлів і накопичення інформації про роботу системи.

Список використаних джерел:

1. MQTT Version 5.0. OASIS Standard. 2019. URL: https://www.oasis-open.org/standard/mqtt-v5-0-os/.
2. Docker Engine. Docker Docs. URL: https://docs.docker.com/engine/.
3. Compose Specification. URL: https://www.compose-spec.io/.
4. Eclipse Mosquitto Documentation. URL: https://mosquitto.org/documentation/.
5. HAProxy version 2.9.15 - Configuration Manual. 2025. URL: https://docs.haproxy.org/2.9/configuration.html.
6. KOZIOLEK, Heiko; GRÜNER, Sten; RÜCKERT, Julius. A comparison of MQTT brokers for distributed IoT edge computing. *In: European Conference on Software Architecture*. Cham: Springer International Publishing, 2020. p. 352-368.