https://doi.org/10.31891/2219-9365-2026-87-4
УДК 004.7:681.5

ОСОЛІНСЬКИЙ Олександр
Західноукраїнський національний університет
http://orcid.org/0000-0002-0136-395X
e-mail: oso@wunu.edu.ua
ЦЮПА Іван
Західноукраїнський національний університет
https://orcid.org/0009-0008-5064-0256
e-mail: ivan.tsupa111@gmail.com
ЗАГОРОДНЯ Діана
Західноукраїнський національний університет
https://orcid.org/0000-0002-9764-3672
e-mail: dza@wunu.edu.ua
ПОЙДИЧ Владислав
Західноукраїнський національний університет
https://orcid.org/0009-0007-5290-051X
e-mail: poydich.vladislav@gmail.com

АРХІТЕКТУРА IoT-ШЛЮЗУ ДЛЯ ІНТЕГРАЦІЇ З ПЛАТФОРМАМИ ЦИФРОВИХ ДВІЙНИКІВ

У статті досліджено особливості забезпечення якості обслуговування (QoS) в IoT edge-gateway, реалізованому на ресурсно-обмеженій платформі Raspberry Pi, у контексті застосування в системах цифрових двійників. Обґрунтовано доцільність переходу від cloud-only підходу до edge-архітектур для зменшення затримок, мережевого навантаження та підвищення стійкості до перебоїв зв'язку. Розроблено та реалізовано прототип IoT-шлюзу, який забезпечує приймання телеметрії від сенсорних вузлів (ESP32/Arduino), буферизацію, збереження та доступ до даних через API.

Запропоновано методику оцінювання впливу режимів збереження телеметричних даних на показники QoS шлюзу, зокрема на затримку збереження, пропускну здатність та рівень втрат. Для експериментального дослідження сформовано контрольований профіль навантаження в діапазоні частот 1–10 Гц із використанням стабільного генератора телеметрії. Оцінювання виконано на основі статистичних характеристик затримок (середнє значення, медіана, p95, p99).

Проведено порівняльний аналіз двох стратегій збереження даних: синхронного (sync) та пакетного (batch). Встановлено, що режим sync забезпечує низьку та стабільну затримку збереження на рівні близько 1.6 мс у всьому дослідженому діапазоні частот, тоді як режим batch зменшує інтенсивність дискових операцій, але призводить до зростання затримки до сотень мілісекунд і формування верхніх перцентилів на рівні близько 1 секунди.

Отримані результати показують, що режим збереження є визначальним фактором QoS IoT edge-gateway і повинен обиратися залежно від вимог прикладної системи: мінімальна затримка оновлення стану або оптимізація використання ресурсів. Запропонована методика та реалізований стенд можуть бути використані для подальших досліджень і практичного налаштування edge-вузлів у платформах цифрових двійників.

Ключові слова: IoT, edge computing, IoT gateway, Raspberry Pi, QoS, телеметрія, цифрові двійники, затримка, буферизація.

OSOLINSKIY Oleksandr, TSIUPA Ivan, ZAHORODNIA Diana, POIDYCH Vladyslav
West Ukrainian National University

IoT GATEWAY ARCHITECTURE FOR INTEGRATION WITH DIGITAL TWIN PLATFORMS

This article investigates the quality of service (QoS) characteristics of an IoT edge gateway implemented on a resource-constrained Raspberry Pi platform in the context of digital twin systems. The study justifies the transition from a cloud-only approach to edge architectures to reduce latency, network load, and dependency on communication reliability. A prototype IoT gateway is designed and implemented to collect telemetry from sensor nodes (ESP32/Arduino), perform buffering and storage, and provide data access via an API.

A method for evaluating the impact of telemetry storage modes on QoS metrics is proposed, including storage latency, throughput, and packet loss. A controlled workload profile in the range of 1–10 Hz is generated using a stable telemetry source to ensure experimental consistency. The evaluation is based on statistical latency characteristics, including mean values, median, and upper percentiles (p95, p99).

A comparative analysis of two storage strategies—synchronous (sync) and batch—has been conducted. The results show that the sync mode ensures low and stable storage latency of approximately 1.6 ms across all tested frequencies. In contrast, the batch mode reduces the frequency of disk operations but increases latency to hundreds of milliseconds, with upper percentiles reaching approximately 1 second.

The findings demonstrate that the storage mode is a critical factor affecting the QoS of an IoT edge gateway and should be selected based on application requirements, such as minimizing latency or optimizing resource usage. The proposed methodology and experimental testbed can be used for further research and practical configuration of edge nodes in digital twin platforms.

Keywords: IoT, edge computing, IoT gateway, Raspberry Pi, QoS, telemetry, digital twins, latency, buffering.

Стаття надійшла до редакції / Received 18.04.2026
Прийнята до друку / Accepted 25.06.2026
Опубліковано / Published 10.09.2026

© ОСОЛІНСЬКИЙ Олександр, ЦЮПА Іван, ЗАГОРОДНЯ Діана, ПОЙДИЧ Владислав

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Кількість IoT-пристроїв та обсягів телеметрії поступово робить підхід (cloud-only) недостатнім для багатьох практичних сценаріїв. Коли всі дані без попередньої обробки відправляються в хмару, система починає залежати від якості інтернет-каналу, а затримки та втрати зв'язку прямо впливають на спостереження та керування. Для задач моніторингу середовища критичними стають середні значення та своєчасність реакції, тобто якщо вузол не може швидко передати дані або отримати команду, зникає сенс оперативної автоматизації. Огляди сучасних IoT-архітектур показує, що перенесення частини обчислень ближче до джерела даних (edge) дозволяє зменшувати навантаження на мережу, скорочувати затримку, а також краще контролювати приватність та локальну доступність сервісів [1, 2].

В такій архітектурі ключову роль відіграє edge-gateway. Перш ніж надсилати сирі дані в хмару або локальний сервіс шлюз виконує наступні функції: а) працює як точка збору для багатьох сенсорів і контролерів; б) проводить агрегацію потоків телеметрії та попередньої обробки даних, куди входять очищення, згладжування, фільтрація даних та виявлення аномалій за допомогою простої логіки.

ПОСТАНОВКА ПРОБЛЕМИ ТА АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

Перехід від хмарної моделі до граничних архітектур викликаний потребою обробляти дані ближче до джерела, зменшувати затримки та мережеве навантаження, а також підвищувати стійкість системи до перебоїв зв'язку.

Для задач цифрових двійників це критично, оскільки стан двійника має оновлюватись подіями з реального середовища, а інтеграційний шар повинен забезпечити узгоджені одиниці вимірювання, часові мітки та інтероперабельний формат даних. Якщо якість обслуговування на рівні шлюзу погіршується, то це негативно впливає на своєчасність оновлення станів і на роботу подальших модулів моніторингу, моделювання та аналізу даних.

В роботі пропонується модульний шлюз для Інтернету речей для платформи цифрових двійників, який реалізує приймання даних, хешування, збереження та надає доступ до даних.

Такий підхід є актуальним , тому що значна частина вимірювань є надлишковою і тому замість того щоб передавати кожну зміну, шлюз може передавати події, тобто тільки важливі зміни, усереднені значення за інтервал. Такий підхід зменшує трафік і робить систему більш стабільною при обмеженій пропускній здатності та нестабільному зв'язку. Паралельно шлюз може забезпечувати буферизацію даних під час відсутності інтернету, щоб після відновлення каналу відправити накопичені вимірювання без втрат [1, 2]

Ще одною проблемою стоїть гетерогенність протоколів і форматів даних. Якщо взяти просту реалізацію де є збірка, сенсор-контролер-шлюз одночасно можуть бути різні канали зв'язку та інтерфейси, наприклад, Wi-Fi, BLE, а на прикладному рівні протоколи MQTT/HTTP/CoAP та ін. Через це шлюз стає елементом узгодження між сенсорами та сервісами, виконуючи протокольну трансляцію, нормалізацію даних і керування підключеннями. На даний час IoT-шлюзи еволюціонували від базової станції з функцією проксі, який просто пересилає пакети, до розумного шлюзу, де з'являються механізми керування QoS, локальна обробка, автономність та функції безпеки [3].

Сучасні мікрокомпютери, такі як Raspberry Pi є доступним і поширеним edge-вузлом, який часто використовують в прикладних IoT-проєктах. В поєднанні з Arduino або ESP32 в якості сенсорних вузлів для збору параметрів навколишнього середовища формується типовий реальний сценарій, де потрібно одночасно вирішити задачі збору даних, стабільної доставки, локальної стійкості та прийнятної затримки. Але такі шлюзи на основі мікрокомп'ютерів мають обмежені системні ресурси. На QoS шлюзу впливають поведінка ОС, планування, мережеві буфери, пріоритети процесів, робота з диском. Тому дослідження edge-gateway на базі доступних моделей мікрокомпютерів в якості точки відповідальності за затримку, надійність і керованість IoT-системи є актуальною задачею [4].

Крім того в контексті проектів цифрових двійників edge-gateway стає частиною інтеграційного шару, який зв'язує фізичні сенсори з моделлю об'єкта та аналітичними сервісами. Для цього потрібні: протокольні адаптери, нормалізація одиниць і часу, буферизація та гарантована своєчасна доставка подій у контекстний шар і ядро стану двійника [5].

АНАЛІЗ ОСТАННІХ ДОСЛІДЖЕНЬ І ПУБЛІКАЦІЙ

В класичній cloud-only моделі сенсори або контролери передають телеметрію напряму в хмару, а всі обчислення, зберігання і правила обробки виконуються централізовано. Такий підхід зручний тим, що логіка системи зосереджена в одному місці. Легше робити єдині оновлення, вести централізовані журнали, будувати

звіти та аналітику. Хмара також добре підходить, коли дані треба довго зберігати і обробляти у великому обсязі.

У великій системі, де зростає кількість пристроїв, значна частина даних постійно "тече" в хмару. Це збільшує навантаження на хмарні ресурси та може призводити до неприйнятної затримки в мережі, що фактично зменшує практичну користь cloud-only підходу для частини сценаріїв [2].

Також у cloud-only моделі сильніше проявляється залежність від якості зв'язку. Якщо канал нестабільний або переривається, сенсорні дані можуть втрачатися, а керуючі рішення можуть затримуватися. Окремо стоїть питання приватності та регуляторних вимог: передача "сирих" даних назовні не завжди прийнятна. У роботах про перехід до edge підкреслюється, що одна з причин цього переходу — зменшення передачі даних через зовнішні мережі, що допомагає з приватністю і безпекою [1].

До типових функції, які переносять на край, виглядають так:

- агрегація (збір телеметрії з багатьох вузлів у єдиний потік);
- фільтрація (відсікання "зайвих" або надто частих повідомлень);
- попередня обробка (нормалізація, базові правила, обчислення простих похідних показників);
- кешування/буферизація (щоб пережити розрив зв'язку);
- локальні правила (наприклад, "якщо температура вище порогу — ввімкнути дію");
- інколи — локальна аналітика/інференс, якщо дозволяють ресурси.

Fog-підхід часто описують як "проміжний шар" між edge і cloud. Він робить архітектуру більш розподіленою, ділить хмарні можливості на менші підмережі та наближає обчислення до джерела даних. [1]. При цьому cloud не зникає. Він лишається корисним для довгого зберігання, важкої аналітики та централізованого управління. Але частина "швидких" функцій переноситься ближче до сенсорів.

У практичних IoT-системах edge-gateway часто стає центральним вузлом. Причина проста: не всі пристрої можуть напряму працювати з хмарою. Частина сенсорів може бути "non-IP", або використовувати локальні [6]. У таких випадках шлюз виступає посередником: він збирає дані через спеціалізовані або пропрієтарні протоколи, а далі передає їх у хмару вже через IP-мережі. Також шлюз робить перетворення протоколів, щоб "зшити" різні технології між собою.

Окремо важливо, що шлюз працює з гетерогенними пристроями. В літературі [3] підкреслюється, що сенсори та інші "розумні речі" збирають дані і передають їх до шлюзу, а шлюз агрегує потоки від різних пристроїв і різних протоколів, після чого відправляє їх до fog або cloud для подальшої обробки. Тобто шлюз — це не просто "перехідник". Це точка, де зручно робити інтеграцію, узгодження форматів, первинну обробку, і вже потім — передачу вище. Шлюз може виступати конвертером протоколів і є ключовою частиною IoT-архітектури, бо багато IoT-пристроїв не підключаються до хмари напряму. При цьому на шлюзі з'являються задачі "як у маленького сервера": планування процесів, робота з диском, мережеві черги, ізоляція компонентів, контроль навантаження.

Коли шлюз стає центром збору телеметрії, з'являється практичне питання: чи витримає він навантаження і чи буде система стабільною. Для сенсорних систем QoS зазвичай зводиться до кількох простих вимірюваних речей:

- затримка (latency): скільки часу проходить від вимірювання до доступності даних у сервісі/сховищі;
- втрати (loss): частка повідомлень, які не дійшли або були відкинуті;
- пропускна здатність (throughput): скільки повідомлень/зразків на секунду система реально обробляє;
- доступність/стабільність: чи не "просідає" система під фоновим навантаженням.

У визначеннях QoS в IoT-комунікаціях [7] часто прямо фігурують такі показники як доступність, рівень втрат пакетів та пропускна здатність. Також підкреслюється, що метрики якості треба визначати явно, і під них вже підбирати механізми/налаштування.

Для IoT-шлюзу це означає, що навіть якщо сама архітектура "edge" правильна, важливо ще й те, як саме реалізовані процеси, черги, запис на диск і мережеві обміни.

МЕТА ТА ЗАВДАННЯ ДОСЛІДЖЕННЯ

Метою роботи є реалізація прототипу IoT edge-gateway на Raspberry Pi 3, який збирає телеметрію від сенсорних вузлів (Arduino/ESP32), виконує буферизацію та збереження, а також забезпечує стабільну якість обслуговування (QoS) при змінному навантаженні як базовий компонент інтеграційного шару для цифрового двійника.

Для досягнення мети вирішуються такі завдання: (1) спроєктувати конвеєр прийому–буферизації–збереження–видачі даних; (2) визначити формат телеметрії та профіль навантаження; (3) реалізувати механізми "балансування" через пріоритезацію критичних етапів і обмеження ресурсів фонових задач; (4) розробити план експериментів і набір метрик QoS/ресурсів; (5) провести вимірювання та сформувати практичні рекомендації для налаштування Linux-параметрів на edge-вузлі.

Об’єктом дослідження є процес збору та обробки телеметричних даних у IoT-системі на рівні edge. Предметом дослідження є програмно-апаратна реалізація IoT-шлюзу на Raspberry Pi та вплив механізмів ОС Linux на показники QoS конвеєра, що визначають швидкість і надійність оновлення стану цифрового двійника.

Практична цінність роботи полягає у створенні відтворюваного лабораторного стенду та працездатного прототипу IoT edge-gateway на Raspberry Pi, який може використовуватися як навчальний або прикладний інтеграційний компонент для систем цифрових двійників. Реалізований шлюз забезпечує приймання телеметрії від ESP32, буферизацію потоку повідомлень, локальне збереження даних та подальший доступ до них.

Практичним результатом є також експериментально підтверджені характеристики двох режимів збереження даних. Встановлено, що синхронний режим sync забезпечує низьку та стабільну затримку збереження на рівні одиниць мілісекунд у діапазоні частот 1–10 Гц, тоді як пакетний режим batch зменшує інтенсивність дрібних операцій запису. Це дає можливість обґрунтовано вибирати режим роботи шлюзу залежно від вимог прикладної системи: мінімальна затримка оновлення стану або зменшення навантаження на підсистему збереження.

Отримані результати можуть бути використані під час розроблення IoT-прототипів та інтеграційних вузлів цифрових двійників, де важливо поєднати доступність апаратної платформи, відтворюваність експериментів і кількісно оцінений вплив режимів роботи на QoS.

Наукова новизна роботи полягає у запропонованій та експериментально апробованій методиці оцінювання впливу режиму збереження телеметричних даних на якість обслуговування IoT edge-gateway на ресурсно-обмеженій платформі Raspberry Pi. На відміну від загальних описів edge-підходу, в роботі виконано кількісне порівняння двох практично важливих стратегій збереження — синхронного та пакетного запису — за єдиним контрольованим профілем навантаження, сформованим сенсорним вузлом.

Для оцінювання роботи шлюзу використано єдину систему показників, яка включає затримку збереження lat_store, медіану та верхні процентилі розподілу затримок (p50, p95, p99) у діапазоні частот 1, 2, 3, 5 і 10 Гц. За результатами вимірювань показано, що режим sync забезпечує стабільну затримку на рівні близько 1.6 мс, тоді як режим batch формує характерний часовий профіль із середньою затримкою близько 0.5 с і верхніми процентилями на рівні близько 1 с. Це дозволяє розглядати режим збереження як один із визначальних факторів QoS edge-gateway для платформ цифрових двійників.

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ

Типові підходи реалізації шлюзу

Одним із поширених варіантів для edge-шлюзу є single-board computer (SBC), зокрема Raspberry Pi. В літературі [8] Raspberry Pi часто розглядають як універсальну платформу для IoT-gateway, тому що вона має достатню обчислювальну потужність для базової обробки даних, може запускати повноцінну ОС і підтримує різні середовища розробки.

Також в сучасних підходах підкреслюють, що gateway на SBC можна будувати з модульних компонентів і запускати їх у контейнерах. Це спрощує розгортання, оновлення і керування компонентами (наприклад, окремо сервіс збору даних, окремо сервіс збереження). В роботі [9]наведено приклад архітектури, де gateway реалізовано як контейнеризований набір застосунків для SBC, таких як Raspberry Pi.

Типова логіка роботи IoT-шлюзу виглядає як “конвеєр”. Сенсори і контролери передають вимірювання на gateway, а gateway далі узгоджує дані і передає їх у fog/cloud або в локальні сервіси. У систематичному огляді шлюзів [3] описано, що “розумні речі” (сенсори/актуатори) надсилають дані на gateway, а gateway агрегує дані від гетерогенних пристроїв і передає їх далі для більш “важкої” обробки.

Якщо перенести це на практичну реалізацію на Raspberry Pi, то зазвичай виділяють такі логічні частини:

- прийом телеметрії (наприклад, по MQTT/HTTP або через локальний інтерфейс від ESP32/Arduino);
- черга або буфер (щоб короткі піки не “ламали” систему і щоб пережити тимчасові затримки запису або мережі);
- збереження (файл, SQLite або інша база залежно від задачі);
- API/візуалізація (простий веб-інтерфейс або REST-API для доступу до даних).

Raspberry Pi є реалістичною платформою для edge-шлюзу, однак її ресурси обмежені, тому архітектура має враховувати CPU, пам’ять і дисковий I/O. В [10] підкреслено, що IoT-пристрої на краю часто є ресурсно обмеженими. Це означає, що при рості частоти телеметрії або при додаванні фонових задач затримка може збільшуватися, а частина повідомлень може втрачатися.

Окремо підкреслюється [8], що Raspberry Pi дає більшу гнучкість, але є і недолік. В порівнянні SBC і microcontroller-gateway, автори відзначають, що Raspberry Pi підходить для складніших задач (обробка даних, edge-аналітика), але при ресурсомістких сценаріях зростають обчислювальні витрати та енергоспоживання, тому потрібен баланс між продуктивністю і ефективністю. Також як практичний фактор згадується складність

налаштування, часті записи на SD-носій та мережа Wi-Fi. Це особливо помітно в шлюзі, який одночасно приймає повідомлення, записує їх і обслуговує API. В таких режимах важливими стають правильні черги, режим запису (одразу чи пакетами) і розділення компонентів.

Протоколи передачі телеметрії (MQTT/HTTP/CoAP) для сенсорних даних

Для сенсорних систем часто обирають MQTT [11], бо він підтримує модель publish/subscribe.

Така схема зручна для масштабування і для розділення ролей "сенсор → споживач даних". MQTT також має механізм надійності через три рівні QoS: QoS 0 ("at most once"), QoS 1 ("at least once"), QoS 2 ("exactly once"). Це дозволяє вибирати компроміс між швидкістю і гарантіями доставки залежно від типу даних. У стандарті MQTT v5.0 закладені важливі для IoT можливості, зокрема параметри Will-повідомлення (Will QoS) та ознака Will Retain, яка визначає, чи буде Will повідомлення збереженим як retained. Обмеження MQTT теж важливо враховувати. В огляді протоколів зазначено, що MQTT працює поверх TCP/IP, потребує брокера, а відмова брокера може зупинити обмін повідомленнями. Для шлюзу це означає, що при виборі MQTT треба або забезпеч продумувати буферизацію/повторну відправку на рівні шлюзу.

HTTP — найпоширеніший протокол веб-взаємодії. Він добре підходить для API шлюзу. В HTTP [12] кожне повідомлення є запитом або відповіддю, а клієнт і сервер обмінюються ними для виконання дій над ресурсами. Також HTTP визначає семантику відповідей через статус-коди (класи 1xx–5xx), що спрощує контроль помилок і обробку результатів. У той же час у контексті IoT HTTP має недолік: він "важчий" за спеціалізовані IoT-протоколи. В [13] показано, що за метрикою HTTP має найбільший overhead через заголовки та текстове кодування. Тому краще використовувати HTTP на рівні шлюзу та сервісів, а не на ресурсно обмежених сенсорних вузлах.

CoAP також використовує модель request/response, схожу на HTTP, але працює поверх UDP, щоб зменшити накладні витрати. Такий підхід може застосовуватись в локальних сенсорних мережах або для низькопотужних вузлів, але якщо потрібна асинхронна модель pub/sub по типу MQTT", тоді CoAP не завжди зручний, бо це інша логіка обміну.

IoT-система використовує в більшості випадків декілька протоколів під різні задачі, [14] тому що вони відрізняються моделлю обміну (pub/sub або request/response), транспортом (TCP/UDP), рівнем надійності, QoS та накладними витратами. В основному для застосунків поєднують MQTT, HTTP - для сервісів та інтеграцій та CoAP для обмежених вузлів.

В такій моделі роль шлюзу полягає в протокольному узгодженні та підтримці взаємодії через схеми на по типу "CoAP ↔ MQTT" або кешування MQTT-повідомлень для REST-доступу.

Організація зберігання телеметрії на edge

Найпростіший спосіб зберігання телеметрії на edge — це CSV формат для сумісності та JSON, як легкий формат обміну, також у практичних реалізаціях зустрічається SQLite для ефективних вибірок [15].

Але файлова схема швидко упирається в практичні обмеження. Якщо дані пишуться часто, то файл росте, і стає складно робити швидкі вибірки "за часом" або "за сенсором". Також виникає питання конкурентного доступу. Тому на шлюзі часто переходять до структурованого зберігання, де є контроль доступу і цілісності.

Компромісним рішенням для edge часто стає SQLite. Для практичного стенду на Raspberry Pi SQLite використовують як "простий" варіант [16]. Це показує, що SQLite добре вписується у типову схему шлюзу.

Разом із тим, SQLite чутливий до I/O. Для Raspberry Pi, де часто використовується SD-карта, часті дрібні записи можуть створювати додаткове навантаження на носій і збільшувати затримки інших операцій. Саме тому в edge-системах важливо продумувати, як часто виконувати записи і як обробляти піки навантаження.

Так як телеметрія є часовими рядами. Для таких даних існують бази даних, такіяк InfluxDB або TimescaleDB [17] Їх перевага -це агрегації по часу, ефективні вставки та оптимізація під запити по інтервалу. Це потрібно коли багато історії, часті запити та аналітика. Але в той же час такі СУБД споживають значні системні ресурси. Тому на edge потрібно приймати компроміси щодо складності обробки та зберігання.

SQLite підходить для базового логування або дуже малих потоків. TSDB доцільні тоді, коли зростає обсяг телеметрії, потрібні часті агрегації та складніші запити, і коли платформа має достатній ресурс для стабільної роботи.

Роль механізмів ОС у забезпеченні якості

Edge-gateway одночасно виконує багато задач. Це означає конкурентне виконання декількох процесів, де одна повільна операція може впливати на інші. Критичним є блокування на дисковому I/O, якщо запис в сховище затримується, то можуть зростати черги повідомлень і затримка кінцевого обслуговування. Тому є необхідність контролю різкого збільшення навантаження та уникнення ситуацій, коли прийом/керування перевантажується від фонових операцій [8].

Коли на шлюзі йдуть операції прийому -кешування-збереження та надання API, між компонентами потрібна міжпроцесна взаємодія. До цього можна віднести черга в пам'яті, локальні сокети або пайпи. Це дозволяє ізолювати помилки і дає можливість незалежно масштабувати або перезапускати окремі

компоненти. В роботі [18] показано, що компоненти системи потрібно розділяти і керувати ними як сервісами для спрщення контролю стану та відмовостійкості.

Якщо розглядати сенсорні системи основною вимогою є те щоб шлюз стабільно приймав повідомлення в ситуації, коли одночасно виконується фоновий аналіз або запис у БД. В огляді вимог до IoT-gateway [8] показано, що планування ресурсів пов'язане з призначенням пріоритетів для задач, орієнтованих на затримку або критичність.

Тому на рівні ОС потрібно надавати вищий пріоритет процесам прийому або кешування, а фонову обробку аналітики, формування звітів запускати з меншим приорітетом [8].

Так як edge-пристрої мають обмежені ресурси у порівнянні з хмарними серверами, неконтрольована фонова задача може повністтю окупувати CPU або RAM і погіршити QoS. Це є властивістю edge-підходу [8]. Тобто для стабільної роботи бажано встановлювати бюджети ресурсів для окремих компонентів, тобто обмежувати фонову обробку.

Для шлюзів з SD-носієм запис телеметрії стає одним із головних джерел затримок. Якщо кожне повідомлення записувати синхронно, зростає кількість дрібних операцій і потенційно збільшується iowait, тому застосовують пакетний запис і локальну агрегацію, щоб нормалізувати навантаження. В роботах [19], [8] показано, що система може застосовувати окрему чергу повідомлень для кожного пристрою та пакетної обробки. Крім того показано, що великі обсяги операцій запису викликають збільшення зносу flash-пам'яті, тому потрібні більш ефективні стратегії, такі як буферизований запис і компресія.

Так як Gateway має працювати стабільно, потрібне застосування механізмів автоматичного перезапуску компонентів при збої та контроль їхнього стану [18]. тобто система має підтримувати працездатність, перезапускаючи компоненти.

Опис апаратної частини стенду

Апаратний стенд побудований як трирівнева система: сенсори далі вузол зв'язку і в кінці edge-шлюз. Такий поділ спрощує тестування, тому що кожен рівень має свою роль і свої обмеження.

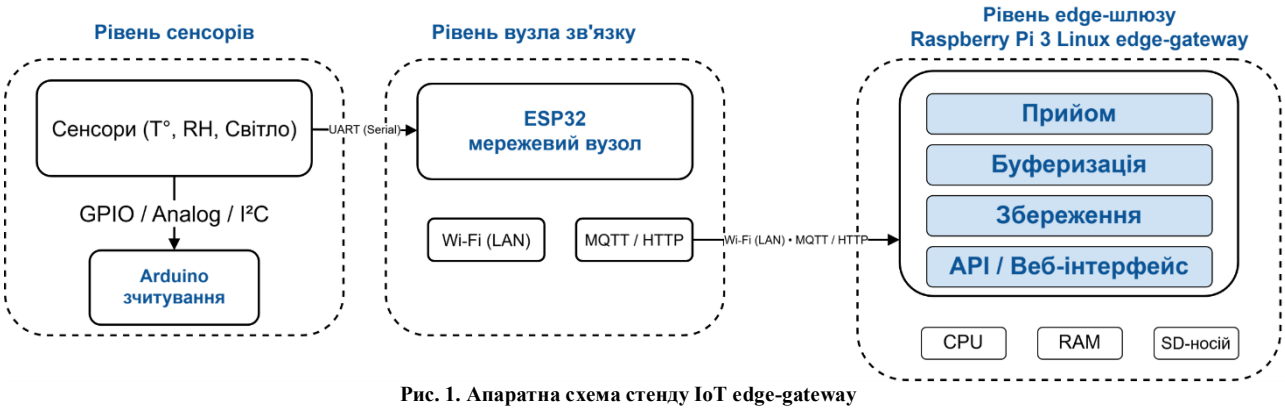

Рис. 1. Апаратна схема стенду IoT edge-gateway

Arduino використовується як вузол зчитування вимірювань. До нього підключені датчики температури, вологості та освітленості. Arduino формує "сирі" значення та готує їх до передачі наступному рівню.

ESP32 виконує роль мережевого вузла. Він отримує вимірювання від Arduino по локальному інтерфейсу і відповідає за передачу телеметрії в локальну мережу через Wi-Fi. На цьому рівні додаються службові поля, лічильник повідомлень і параметри якості зв'язку, щоб потім оцінювати втрати та поведінку системи при мережевих коливаннях.

Raspberry Pi працює під Linux і використовується як edge-gateway. Він приймає телеметрію по мережі, виконує буферизацію, запис у сховище та надає доступ до даних через API або веб-інтерфейс. На цьому рівні реалізується балансування навантаження між компонентами шлюзу, щоб при піках запису або обробки не погіршувалася якість прийому телеметрії.

Ключова ідея стенду полягає в тому, що сенсорні вузли генерують потік телеметрії, а Raspberry Pi обробляє його як сервіс в реальних умовах. Це дозволяє оцінювати QoS з врахуванням практичних факторів таких як Wi-Fi, SD-запис, конкуренція процесів.

Загальна архітектура системи

Загальна архітектура побудована як послідовний конвеєр обробки, де кожен етап має просту роль і чітку відповідальність (рис.2). Такий підхід зручний для оцінювання QoS, оскільки можна окремо вимірювати затримки прийому, буферизації та запису.

Потік даних організовано так:

1. Arduino опитує датчики температури, вологості та освітленості й формує базові значення вимірювань.

2. ESP32 приймає “сирі” значення та формує повідомлення телеметрії у визначеному форматі. На цьому етапі додаються службові поля, які потрібні для контролю втрат.

3. Телеметрія передається через Wi-Fi у локальній мережі до Raspberry Pi. На транспортному рівні може використовуватись MQTT або HTTP, повідомлення надходять як потік з певною частотою.

4. Collector приймає повідомлення з мережі, перевіряє їх базову коректність і передає далі в буфер. Collector не виконує складні операцій, щоб мінімізувати затримку прийому.

5. Буфер виконує роль амортизатора між нестабільним потоком вхідних повідомлень і більш повільними операціями збереження або обробки. Якщо виникає короткий пік навантаження або затримка запису на диск, буфер дозволяє не втратити дані одразу.

6. Storage витягує повідомлення з буфера і записує їх у вибране сховище. Режим запису може бути синхронним або пакетним, щоб зменшити дрібні I/O операції на SD-носії.

7. Processor виконує фонову обробку: агрегації за часом, прості правила, підготовку даних для візуалізації. Цей компонент не повинен блокувати прийом і запис, тому його робота розглядається як не критична до затримки.

8. Кінцевий користувач отримує дані через веб-сторінку або REST API.

Даний конвеєр відповідає нижньому рівню інтеграції, тобто після первинної валідації та нормалізації повідомлення можуть публікуватися як події у контекстний шар та синхронізувати стан двійника в ядро керування станом.

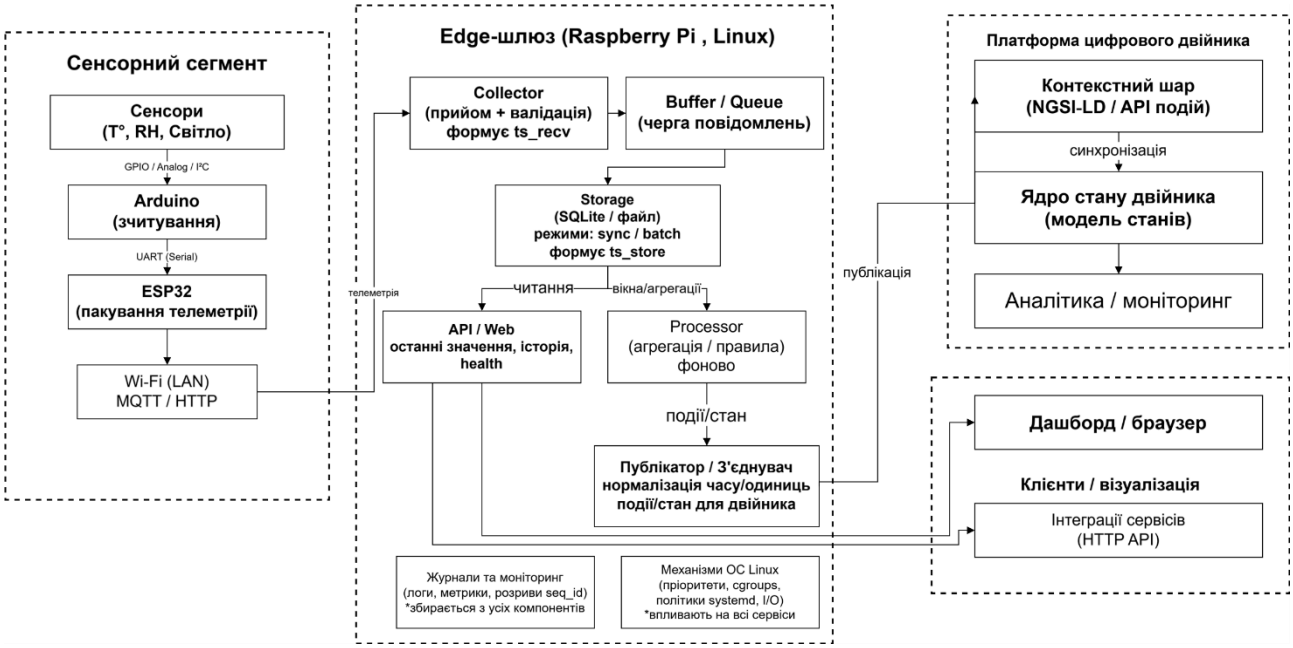

Рис. 2. Загальна архітектура системи та потік даних

Балансування навантаження в архітектурі

В даній реалізації балансування навантаження це набір рішень, які забезпечують, щоб критичний етап прийому телеметрії не деградував через фонові задачі, такі як запис у БД, агрегація, веб-віддача.

Буферизація розділяє прийом і запис, зменшуючи вплив повільного диску на прийом. Пріоритети процесів задають перевагу Collector/Storage над Processor, щоб у піках першим зберігався потік даних. Обмеження ресурсів CPU та RAM і не дозволяють фоновим компонентам забирати весь ресурс системи. Така схема дозволяє надалі експериментально перевірити, як зміни параметрів ОС і режимів запису впливають на QoS.

Формат даних телеметрії та частота надходження

В даному підході використовується формат телеметрії, який легко передавати мережею, проводити логування на шлюзі та аналізувати під час експериментів. Повідомлення містить мінімальний набір полів, достатній для контролю цілісності потоку, оцінювання затримок і збереження значень вимірювань. Кожне повідомлення описує одне вимірювання або набір вимірювань, знятих в один момент. До складу полів входить: а)ідентифікатор джерела даних; б)порядковий номер повідомлення, який збільшується на 1 при кожній відправці; в) час формування повідомлення. Для спрощення можна використовувати:

час на ESP32, або локальний час шлюзу, коли повідомлення прийнято.
temperature, humidity, light — значення вимірювань.
rssi — рівень сигналу Wi-Fi для оцінювання якості каналу.

Також допускається подання даних як JSON або як компактний текстовий формат.

Приклад JSON-повідомлення:

```
{
 "device_id": "esp32-01",
 "seq_id": 15420,
 "timestamp": 1739981234567,
 "temperature": 23.6,
 "humidity": 41.2,
 "light": 178
}
```

Поле seq_id потрібне, щоб оцінювати, які дані були сформовані та надіслані, але не були доставлені або збережені без складних мережевих інструментів. Якщо на шлюзі збережені повідомлення мають розриви в послідовності, це означає, що частина повідомлень була втрачена або не дійшла до етапу збереження. Також seq_id дозволяє бачити дублювання, якщо одне й те саме значення приходить повторно.

Під час експериментів розглядається дві практичні ситуації. В першому випадку коли Timestamp формується на стороні ESP32, якщо час на ESP32 синхронізований, можна оцінювати майже повну end-to-end затримку. Інший момент, коли Timestamp формується на стороні Raspberry Pi, якщо синхронізація часу на ESP32 не гарантується, тоді основні затримки оцінюються всередині шлюзу: а)ts_recv — час прийому повідомлення (Collector); б)ts_store — час підтвердження запису (Storage)

Тоді внутрішня затримка шлюзу визначається як різниця між ts_store та ts_recv:

$$lat_{store,i} = ts_{store,i} - ts_{recv,i} \quad (1)$$

де $lat_{store,I}$ — внутрішня затримка збереження i -го повідомлення, $ts_{recv,i}$ — час приймання повідомлення модулем Collector, $ts_{store,i}$ — час підтвердження його запису модулем Storage.

Цей підхід не потребує точного годинника на ESP32, але дозволяє порівнювати режими ОС за однаковою методикою.

Частота надходження визначається періодом опитування сенсорів та відправкою повідомлень ESP32. Для тестування QoS потрібно мати декілька режимів навантаження, наприклад, 1 Гц буде відповідати базовому режиму з низьким навантаженням, 5 Гц відповідає робочому режиму з середнім навантаженням, а 10–20 Гц — режим навантаження, тобто на цій частоті будуть перевірятись межі стабільності. При збільшенні частоти потік повідомлень стає щільнішим, і на Raspberry Pi сильніше проявляються обмеження CPU та дискового I/O. Тому саме зміна частоти є способом перевірки, як система поводиться в різних режимах та де починають зростати p95/p99 затримки або втрати. Крім того всі експериментальні сценарії використовують однаковий формат повідомлення куди як мінімум входить device_id, seq_id, значення сенсорів, а часові мітки формуються на шлюзі через ts_recv і ts_store. Це дозволяє порівнювати результати між сценаріями без впливу різних форматів або різної логіки формування даних.

Частота надходження телеметрії задається як параметр генерації навантаження:

S0 фіксується помірна частота від 1 Гц до 5 Гц, щоб отримати еталонні значення затримки та споживання ресурсів.

S1: вплив планування та пріоритетів під фоновим навантаженням. Частота надходження телеметрії залишається такою ж, як у S0, щоб зміни показників QoS пояснювалися саме налаштуваннями операційної системи та фоновим навантаженням, а не іншим профілем трафіку.

S2: вплив обмеження ресурсів через механізми cgroups. Частота надходження телеметрії також зберігається на рівні S0, щоб оцінити, як ліміти CPU та RAM для фонових компонентів впливають на стабільність прийому і запису.

S3: порівняння монолітної реалізації та конвеєрної архітектури з міжпроцесною взаємодією. Частота фіксується на одному рівні, як у S0, щоб порівняння архітектур було коректним.

S4: вплив режиму збереження даних у сховищі. Частота залишається такою ж, як у S0, щоб ізолювати вплив синхронного та пакетного запису на показники I/O та QoS.

Додатковий прогін для перевірки межі стабільної роботи. Окремо виконується тест із підвищеною частотою 10–20 Гц як навантажувальний сценарій, який показує, за яких умов на Raspberry Pi зростають затримки p95/p99 або з'являються втрати повідомлень за послідовністю `seq_id`.

Програмні компоненти на Raspberry Pi

На Raspberry Pi програмна частина шлюзу реалізована як конвеєр модулів, який забезпечує прийом телеметрії, її буферизацію, збереження та надання доступу до даних. Окремо передбачено адаптер-публікатор для інтеграції з платформою цифрових двійників. Він нормалізує одиниці вимірювання та часові мітки і передає події або стани у контекстний шар чи ядро стану двійника.

Модуль Collector виконує роль протокольного адаптера і приймає повідомлення від ESP32 через обраний вхідний протокол, перевіряє коректність полів `device_id`, `seq_id` і значень, формує мітку `ts_recv` та передає дані в буфер. Запис на диск на цьому етапі не виконується, щоб мінімізувати затримку прийому.

Модуль Buffer виконує функцію буфера та розділяє етапи прийому, запису, крім того працює як амортизатор для потоку повідомлень. Буфер має обмеження за розміром і дозволяє переживати короткі піки навантаження або тимчасове уповільнення запису на SD-носій.

Модуль Storage відповідає за збереження і записує телеметрію у локальне сховище. Передбачено синхронний запис, коли кожне повідомлення фіксується одразу, і пакетний запис, коли дані накопичуються та зберігаються порціями. Це дає змогу порівняти вплив дискових операцій на показники QoS. Під час запису формується мітка `ts_store`, що дозволяє обчислювати внутрішню затримку `lat_store = ts_store − ts_recv`.

Модуль Processor виконує фонову обробку та формує агрегації за часовими інтервалами, застосовує прості правила або готує дані для візуалізації. Цей модуль не є критичним до затримки, тому для нього можна встановлювати обмеження за CPU та пам'яттю, щоб не впливати на прийом і запис.

Модуль Publisher забезпечує інтеграцію з цифровим двійником і перетворює телеметрію на нормалізовані події або стани, після чого публікує їх у компоненти контекстного шару та ядра стану. Це забезпечує узгоджені одиниці вимірювання, коректні часові мітки та відтворювану синхронізацію станів.

Модуль API забезпечує доступ до даних та надає інтерфейси для читання останніх значень, отримання історії за часовий інтервал і виконання базових агрегацій.

Модуль Monitoring відповідає за журнали і фіксує події прийому, запису та помилки, включно з розривами послідовності `seq_id`, а також службові події, такі як перезапуски сервісів або переповнення черги. Обсяг журналів обмежується, щоб уникнути швидкого заповнення SD-носія.

Використані механізми ОС Linux в реалізації

Модулі Collector, Storage, API та Publisher запускаються як окремі служби, що спрощує ізоляцію збоїв і керування ресурсами. Фонова обробка в модулі Processor виконується окремо та не блокує критичні етапи. Взаємодія між прийомом і збереженням організована через локальний доменний сокет Unix, що дає змогу передавати повідомлення без мережевих накладних витрат і підтримувати контроль черги.

Модулі Collector і Storage мають підвищений пріоритет виконання за допомогою параметра nice, тоді як модуль Processor працює з нижчим пріоритетом. Це забезпечує розподіл навантаження, оскільки прийом і запис мають перевагу над фоновою обробкою. Для фонового компонента застосовуються обмеження процесорного часу та оперативної пам'яті, щоб уникнути погіршення якості обслуговування при піках навантаження. Налаштування виконуються через параметри CPUQuota та MemoryMax у файлах опису служб systemd.

Щоб зменшити очікування введення-виведення і знос SD-карти, використовується пакетний запис, контроль частоти примусового скидання даних на диск та ротація журналів. Синхронний режим запису залишається базовим варіантом для порівняння найгірших випадків.

Компоненти запускаються як служби systemd з політиками автоматичного перезапуску при збоях. Для контролю стану використовується перевірка працездатності через спеціальний URL-запит, а параметри перезапуску і тайм-аути обмежують циклічні перезапуски. Служби запускаються від окремого непривілейованого користувача з обмеженим доступом до каталогів даних і журналів та з мінімальною кількістю відкритих мережевих портів.

Методика оцінювання впливу механізмів ОС на QoS

Мета оцінювання полягає в тому, щоб кількісно показати, як налаштування та механізми операційної системи Linux на Raspberry Pi впливають на якість обслуговування потоку телеметрії. Для цього застосовується керований профіль навантаження, визначені сценарії роботи та однакові правила збору й порівняння показників.

В межах експериментів змінюються ті параметри, які безпосередньо впливають на конкуренцію за ресурси та на затримки в конвеєрі. До них належать пріоритети виконання процесів, обмеження процесорного часу та оперативної пам'яті для фонової обробки, спосіб організації конвеєра обміну між компонентами, а також стратегія збереження телеметрії. Окремо створюється фонове навантаження, щоб перевірити, як шлюз поводиться в умовах перевантаження і чи зберігає стабільність роботи.

Якість обслуговування оцінюється за кількома вимірюваними показниками. Основним є затримка до збереження, яка визначається як різниця між часом прийому повідомлення на шлюзі та часом підтвердження його запису у сховище.

$$lat_{store} = \frac{1}{N}\sum_{i=1}^{N} lat_{store,i} \quad (2)$$

де N— кількість збережених повідомлень у межах одного прогону, а lat_{store} — середня затримка збереження.

Для аналізу стабільності розраховуються медіанне значення затримки та значення для гірших випадків, щоб бачити, чи з'являються довгі затримки при навантаженні. Додатково вимірюється фактична

пропускна здатність, тобто скільки повідомлень за секунду реально потрапляє у сховище, а також втрати, які визначаються за розривами в послідовності лічильника повідомлень або за різницею між кількістю прийнятих і збережених записів.

$$Throughput = \frac{N_{saved}}{T_{run}} \quad (3)$$

де N_{saved} — кількість повідомлень, збережених у межах одного прогону, Trun — тривалість прогону, Throughput — фактична пропускна здатність шлюзу.

$$Loss = \frac{N_{sent} - N_{saved}}{N_{sent}} \cdot 100\% \quad (4)$$

де N_{sent} — кількість сформованих або надісланих повідомлень, N_{saved} — кількість повідомлень, збережених у сховищі, Loss — відносний рівень втрат повідомлень.

Щоб пояснювати причини зміни якості обслуговування, додатково фіксуються системні показники роботи шлюзу. Збирається завантаження процесора та середнє навантаження системи, використання оперативної пам'яті, а також частка часу очікування введення-виведення як ознака затримок дискової підсистеми. Окремо оцінюється інтенсивність запису на носій.

Для кожного повідомлення фіксується ідентифікатор пристрою, порядковий номер повідомлення та час прийому на шлюзі. Під час збереження додатково фіксується час підтвердження запису і обрана стратегія збереження, що дозволяє розділяти синхронний та пакетний режими. За потреби ведеться журнал також час відповіді на запити інтерфейсу доступу до даних. Системні показники збираються паралельно з фіксованим інтервалом, щоб надалі співставляти їх із ростом затримок і появою втрат.

Для забезпечення повторюваності кожен сценарій виконується три рази. Для кожного прогону обчислюються статистичні характеристики затримок, фактична пропускна здатність і рівень втрат повідомлень. Підсумковий результат визначається як середнє значення або медіана по прогонах.

Результати попереднього експериментального оцінювання профілю телеметричного навантаження

На першому етапі експериментальних досліджень було попередню перевірку стабільності формування телеметричного потоку на стороні ESP32 до розгортання повного шлюзу на Raspberry Pi. Такий підхід дозволив ізолювати джерело навантаження та окремо оцінити, наскільки стабільно сенсорний вузол формує повідомлення з заданою частотою, а також чи виникають втрати або часові відхилення вже на етапі передачі через послідовний інтерфейс. Приймання та логування виконувалися на ПК, що дало змогу перевірити коректність профілю навантаження перед подальшими експериментами на Raspberry Pi.

У всіх режимах тривалість одного прогону становила 180 с. Було досліджено режими генерації 1, 2, 3, 5 і 10 Гц. Для кожного пакета фіксувалися порядковий номер, часові мітки на стороні пристрою та на стороні ПК, а також розраховувалися середні інтервали між сусідніми повідомленнями та стандартне відхилення інтервалів приймання. Підсумкові результати наведено в таблиці 1.

Таблиця 1.

Результати попередньої перевірки стабільності синтетичного телеметричного потоку

Частота, Гц	Очікуваний інтервал, мс	Кількість прийнятих пакетів	Втрати пакетів	Середній інтервал на пристрої, мс	Середній інтервал на ПК, мс	Стандартне відхилення інтервалу на ПК, мс
1	1000.00	181	0	1000.00	999.94	0.63
2	500.00	361	0	500.00	499.97	0.45
3	333.33	542	0	333.00	332.98	0.53
5	200.00	901	0	200.00	199.99	0.38
10	100.00	1801	0	100.00	99.99	0.39

Отримані результати показують, що в усіх протестованих режимах потік формувався стабільно, а втрати пакетів були відсутні. Послідовність `packet_id` не містила розривів, що свідчить про безперервне приймання повідомлень за весь експериментальний період. Для режимів 1, 2, 5 і 10 Гц середній інтервал на стороні пристрою повністю збігався із заданими значеннями, а на стороні ПК відхилення від номіналу становило лише соті частки мілісекунди.

Для проміжного режиму 3 Гц середній інтервал на стороні пристрою склав 333.00 мс, а на стороні ПК – 332.98 мс. Це значення практично відповідає заданому режиму 333.33 мс і підтверджує, що навіть для нецілого інтервалу система підтримує стабільний темп генерації телеметрії без втрат і без накопичення помітного дрейфу в межах одного прогону.

Окремо слід відзначити малий розкид інтервалів приймання на стороні ПК. Значення стандартного відхилення перебували в межах від 0.38 до 0.63 мс. Це показує низький часовий джитер потоку і дозволяє вважати сформований профіль навантаження достатнім для подальших досліджень QoS шлюзу на Raspberry

Pi. Тобто в подальших експериментах зміни затримок, пропускної здатності або втрат можна буде пов'язувати вже переважно з роботою edge-gateway та механізмів ОС Linux, а не з нестабільністю джерела телеметрії.

Отже попередній експериментальний етап підтвердив коректність використаного генератора синтетичних даних на ESP32 та його придатність для подальших випробувань в складі повного стенду. Отриманий телеметричний потік в діапазоні 1–10 Гц характеризується відсутністю втрат, правильним дотриманням заданого темпу передавання та мінімальним часовим розкидом. Це дає можливість використовувати цей профіль як базовий вхідний вплив в подальших сценаріях S0–S4 під час оцінювання впливу механізмів ОС Linux на QoS edge-gateway.

Результати експериментального оцінювання режимів sync і batch на Raspberry Pi

Після завершення попереднього етапу валідації генератора телеметричного потоку було виконано експериментальне оцінювання повного стенду на Raspberry Pi з використанням конвеєра Collector -> Buffer -> Storage. При тестуванні порівнювалися два режими збереження телеметрії: синхронний запис sync (таблиці 2) та пакетний запис batch (таблиці 3). Для кожного режиму виконано по три прогони на частотах 1, 2, 3, 5 і 10 Гц.

Під час кожного прогону фіксувалися кількість збережених повідомлень та статистичні характеристики внутрішньої затримки збереження lat_store_ms: середнє значення, медіана p50, верхні перцентилі p95 і p99, а також максимальне зафіксоване значення. Так як деякі прогони завершувалися вручну, тривалість серій була не однаковою. Тому кількість записів наведено як допоміжний показник, а основну увагу приділено статистикам затримки.

Таблиця 2.

Узагальнені результати для режиму sync

Частота, Гц	Кількість записів у 3 прогонах	Сер. avg, мс	Сер. p50, мс	Сер. p95, мс	Сер. p99, мс	Найбільше max, мс
1	1247 / 386 / 165	1.608	1.406	2.428	2.504	3.471
2	287 / 356 / 635	1.564	1.340	2.421	2.490	2.820
3	1672 / 723 / 373	1.592	1.384	2.424	2.496	3.246
5	1015 / 1311 / 1398	1.581	1.425	2.405	2.472	62.238
10	1653 / 1798 / 1070	1.582	1.428	2.362	2.456	98.775

Як показують результати таблиці 2, в режимі sync система зберігала стабільні часові характеристики на всьому дослідженому діапазоні частот. Середня затримка збереження для всіх частот перебувала в межах приблизно 1.56–1.61 мс, а медіана p50 залишалася близькою до 1.34–1.43 мс. Верхні перцентилі p95 і p99 також змінювалися незначно, що вказує на стабільну роботу конвеєра під час прийому та запису телеметрії. Крім того при частотах 5 і 10 Гц було зафіксовано викиди максимального часу збереження. Для 5 Гц найбільше значення max досягло 62.238 мс, а для 10 Гц - 98.775 мс, але при цьому значення p95 і p99 залишилися на низькому рівні.

Таблиця 3.

Узагальнені результати для режиму batch

Частота, Гц	Кількість записів у 3 прогонах	Сер. avg, мс	Сер. p50, мс	Сер. p95, мс	Сер. p99, мс	Найбільше max, мс
1	862 / 514 / 665	500.580	469.359	1002.890	1172.912	1201.430
2	948 / 589 / 553	498.395	500.743	1000.487	1004.893	1503.169
3	682 / 660 / 634	498.505	533.566	1001.468	1005.147	1013.134
5	949 / 599 / 922	519.629	576.840	1002.255	1198.477	1201.565
10	818 / 714 / 730	480.799	500.520	993.539	1002.680	1195.619

В режимі batch середня затримка збереження зросла приблизно до 481–520 мс, а медіана p50 перебувала в межах близько 469–577 мс. Значення p95 для всіх частот виявилися близькими до 1 секунди, а p99 в частині режимів перевищував 1.17–1.20 с, тому що в пакетному режимі запис у сховище виконується після накопичення пакета або після спрацювання таймауту скидання.

Найбільший зафіксований викид у режимі batch становив 1503.169 мс у прогоні для частоти 2 Гц. В інших серіях максимальні значення також переважно перебували на рівні близько 1.0–1.2 с. Отже, пакетний режим зменшує частоту дискових операцій, але за рахунок зростання латентності окремого повідомлення.

Порівняння таблиць 2 і 3 показує, що режим sync є значно кращим з погляду мінімізації затримки.

Як видно з рисунку 3, режим sync зберігає середню затримку на рівні близько 1.6 мс у всьому діапазоні частот, тоді як для batch середня затримка перебуває на рівні близько 0.5 с. Для нього типовими є значення на рівні одиниць мілісекунд, тоді як для batch затримка переходить у діапазон сотень мілісекунд, а верхні перцентилі наближаються до однієї секунди. Отже, якщо пріоритетом є своєчасне збереження кожного повідомлення, доцільно використовувати sync. Якщо ж важливішим є зменшення інтенсивності дрібних операцій запису на носій, можна застосовувати batch, але з врахуванням суттєвого погіршення часових

характеристик.

Проведені експерименти підтвердили, що Raspberry Pi може забезпечувати стабільну якість обслуговування телеметричного потоку в режимі sync на всьому діапазоні 1–10 Гц. Але вибір режиму збереження істотно впливає на QoS шлюзу, а тому повинен розглядатися як один із ключових параметрів налаштування інтеграційного шару цифрового двійника.

Як видно з рисунку 4, для режиму sync значення p95 залишаються низькими і майже незмінними на всьому діапазоні частот, тоді як для режиму batch вони залишаються в районі 1 с.

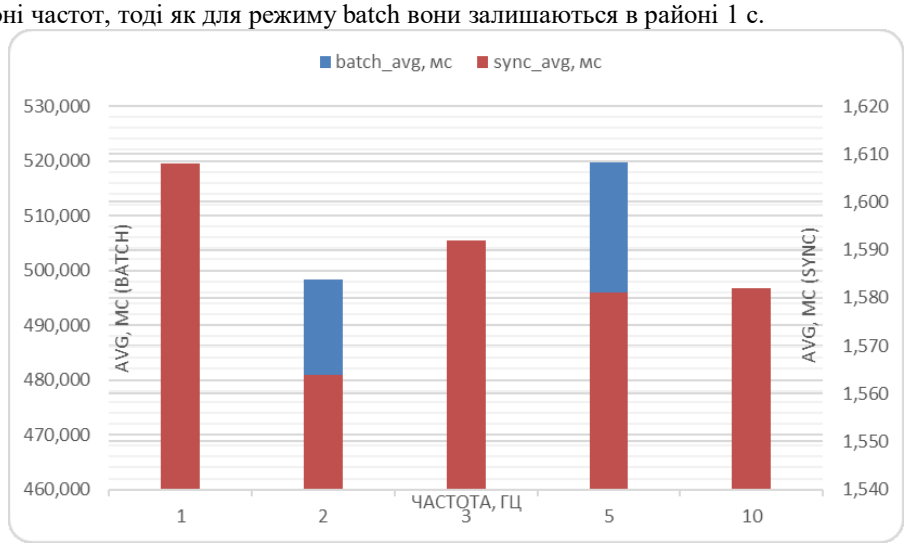

Рис. 3. Порівняння середньої затримки збереження avg для режимів sync і batch при різних частотах телеметрії

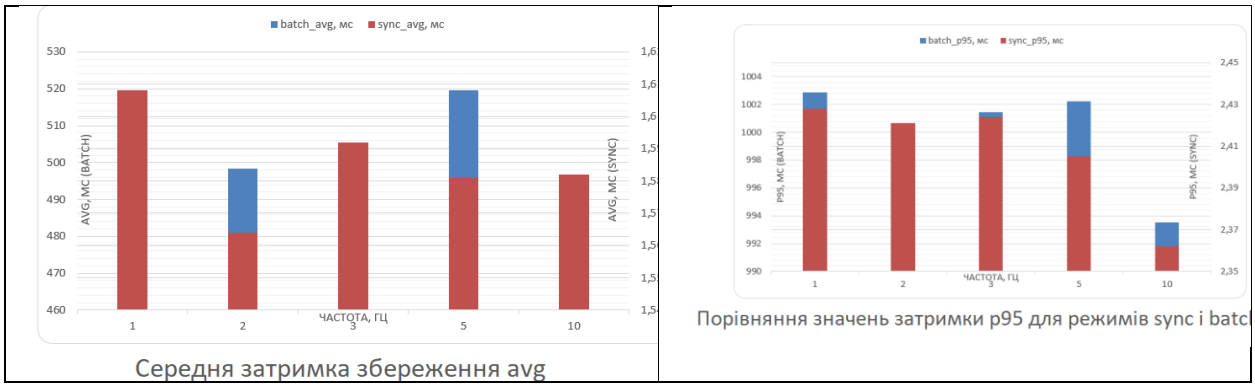

Рис. 4. Порівняння верхнього процентиля затримки p95 для режимів sync і batch при різних частотах телеметрії

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

В роботі реалізовано прототип IoT edge-gateway на Raspberry Pi для платформи цифрових двійників, який забезпечує приймання телеметрії від сенсорного вузла на ESP32, буферизацію, збереження та подальший доступ до даних. Побудований стенд дав змогу дослідити вплив організації програмного конвеєра і режиму запису на показники QoS в умовах ресурсно обмеженої edge-платформи.

Підтверджено, що генератор синтетичної телеметрії на ESP32 формує стабільний потік повідомлень в діапазоні 1–10 Гц без втрат пакетів та з дуже малим часовим розкидом на стороні ПК. Це дозволило використати сформований профіль навантаження як коректний вхідний вплив для подальших досліджень шлюзу на Raspberry Pi та ізолювати вплив механізмів операційної системи і підсистеми збереження від нестабільності джерела даних.

Основний етап експериментів показав, що в режимі синхронного запису `sync` шлюз забезпечує низьку і стабільну затримку збереження на всьому дослідженому діапазоні частот. Для частот 1, 2, 3, 5 і 10 Гц середні значення `avg` перебували приблизно в межах 1.56–1.61 мс, медіана `p50` — у межах 1.34–1.43 мс, а верхні порогові значення розподілу `p95` і `p99` залишалися близькими до 2.36–2.50 мс. Це показує, що обраний конвеєр `Collector → Buffer → Storage` в синхронному режимі забезпечує практично однакову якість обслуговування як при низькому, так і при підвищеному навантаженні.

На частотах 5 і 10 Гц були зафіксовані викиди максимального часу збереження, які досягали десятків мілісекунд, а в окремих прогонах — понад 60–90 мс. Такі пікові значення не змінили розподіл `p95/p99`, але показали, що під час інтенсивнішого потоку можливі короткочасні затримки, пов’язані з підсистемою запису або внутрішнім плануванням обробки.

Для режиму пакетного запису `batch` отримано інший часовий профіль. Середня затримка збереження для всіх частот становила приблизно 480–520 мс, медіана `p50` була в межах близько 470–580 мс, а значення `p95` були близькими до 1 секунди. Для частини прогонів `p99` перевищував 1.1–1.2 секунди, а максимальні значення в окремих випадках досягали близько 1.5 секунди. Такий результат є логічним наслідком самої ідеї пакетного запису: повідомлення не фіксуються одразу після надходження, а накопичуються та зберігаються групами. Отже, режим `batch` зменшує інтенсивність дрібних операцій запису, але збільшує латентність.

Порівняння двох режимів показує, якщо для системи критичною є мінімальна затримка оновлення стану цифрового двійника або швидка фіксація кожного окремого телеметричного пакета, то доцільним є використання режиму `sync`. Якщо ж для конкретного сценарію допускається відкладене збереження і важливішим є зменшення кількості операцій введення-виведення на SD-носій, то може застосовуватися `batch`, але з прийняттям затримки на рівні сотень мілісекунд і верхніх перцентилів близько 1 секунди.

Практична цінність одержаних результатів полягає в тому, що вони дають чіткі налаштування для edge-gateway на Raspberry Pi в складі платформи цифрових двійників. Для режимів, орієнтованих на оперативне оновлення стану, рекомендовано синхронний запис і пріоритезацію критичних компонентів конвеєра. Для режимів, орієнтованих на довше накопичення та економніший запис, можливе застосування пакетного збереження. Запропонований стенд і методика можуть бути використані як відтворювана основа для подальших досліджень впливу пріоритетів процесів, міжпроцесної взаємодії, обмеження ресурсів і фонового навантаження на показники QoS edge-вузлів.

Перспективи подальших досліджень полягають у розширенні експериментів на сценарії S1–S3, тобто в оцінюванні впливу пріоритетів планування, cgroups-обмежень та різних варіантів організації конвеєра між модулями.

References

1. Dauda A., Flauzac O., Nolot F. A survey on IoT application architectures // *Sensors*. 2024. Vol. 24, No. 16. Art. 5320. DOI: https://doi.org/10.3390/s24165320
2. Kong L. et al. Edge-computing-driven Internet of Things: a survey // *ACM Computing Surveys*. 2022. Vol. 55, No. 8. P. 1–41.
3. Beniwal G., Singhrova A. A systematic literature review on IoT gateways // *Journal of King Saud University – Computer and Information Sciences*. 2022. Vol. 34, No. 10. P. 9541–9563.
4. Kong L. et al. Edge-computing-driven Internet of Things: a survey // *ACM Computing Surveys*. 2023. Vol. 55, No. 8. Art. 174. 41 p. DOI: https://doi.org/10.1145/3555308
5. VanDerHorn E., Mahadevan S. Digital twins standards and platforms: OGC, NGSI-LD, Eclipse Ditto. 2021–2025.
6. Network Interface Protocol for IoT. IETF Draft. URL: https://www.ietf.org/archive/id/draft-ietf-asdf-nipc-14.html
7. Hmissi F., Ouni S. A survey on application layer protocols for IoT networks // *arXiv preprint*. 2024. URL: https://arxiv.org/abs/2405.15901
8. Serepas F. et al. Lightweight embedded IoT gateway for smart homes based on an ESP32 microcontroller // *Computers*. 2025. Vol. 14, No. 9. Art. 391.
9. Vlachos M., Lopardo R., Amditis A. Real-time vehicle status monitoring under moving conditions using a low power IoT system // *Journal on Internet of Things*. 2022. Vol. 4, No. 4. P. 235–261. DOI: https://doi.org/10.32604/jiot.2022.040820
10. Bernardos R., Di Benedetto M., Mourad A. Internet of Things (IoT) Edge Challenges and Functions. RFC 9556. IETF, 2024. URL: https://www.rfc-editor.org/rfc/rfc9556
11. MQTT Version 5.0. OASIS Standard. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/csprd01/mqtt-v5.0-csprd01.html
12. Fielding R. et al. HTTP Semantics. RFC 9110. IETF, 2022. URL: https://datatracker.ietf.org/doc/html/rfc9110
13. Žatuchin D., Azarskov M. An adaptive protocol selection framework for energy-efficient IoT communication: dynamic optimization through context-aware decision making // *Informatics*. 2025. Vol. 12, No. 4. Art. 125. DOI: 10.3390/informatics12040125. URL: https://www.mdpi.com/2227-9709/12/4/125
14. Petrescu I. et al. Transport and application layer protocols for IoT: comprehensive review // *Technologies*. 2025. Vol. 13. Art. 583. DOI: https://doi.org/10.3390/technologies13120583
15. Parween G. et al. Digital twin prospects in IoT-based human movement monitoring model // *Sensors*. 2025. Vol. 25. Art. 6674. DOI: https://doi.org/10.3390/s25216674
16. [Електронний ресурс]. DOI: https://doi.org/10.52058/2786-6025-2025-11(52)-2564-2575
17. Khelifati A. et al. TSM-bench: benchmarking time series database systems for monitoring applications // *Proceedings of the VLDB Endowment*. 2023. Vol. 16, No. 11. P. 3363–3376.
18. Lynn T. The cloud-to-thing continuum: opportunities and challenges in cloud, fog and edge computing. 2020.
19. Mahmoudi C., Mourlin F., Battou A. Formal definition of edge computing: an emphasis on mobile cloud and IoT composition // *Proceedings of the 3rd International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 2018. P. 34–42.