Міністерство освіти і науки України
Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерної інженерії та безпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

КОМПЛЕКСНЕ ВПРОВАДЖЕННЯ ТА ДОСЛІДЖЕННЯ ЦЕНТРАЛІЗОВАНОЇ СИСТЕМИ УПРАВЛІННЯ ПРИСТРОЯМИ НА БАЗІ МУЛЬТИПРОТОКОЛЬНОЇ МЕРЕЖЕВОЇ ІНФРАСТРУКТУРИ

COMPREHENSIVE IMPLEMENTATION AND RESEARCH OF A CENTRALIZED DEVICE MANAGEMENT SYSTEM BASED ON A MULTI-PROTOCOL NETWORK INFRASTRUCTURE

спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІм-21
Ревко Рустам Джигадович

(підпис)

Керівник:
к.т.н., доцент
Багнюк Наталія Володимирівна

(підпис)

Кваліфікаційну роботу
допущено до захисту
«_____» грудня 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Гринюк Сергій Васильович

(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *магістр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_______________ доц. Т. ТЕРЛЕЦЬКИЙ
«_______» ______________ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Ревко Рустаму Джигадовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Комплексне впровадження та дослідження централізованої системи управління пристроями на базі мультипротокольної мережевої інфраструктури*

Керівник роботи *к.т.н., доцент Багнюк Наталія Володимирівна.*

затверджені наказом закладу вищої освіти від «17» червня 2025 року № № 290/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 09.12.2025р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література з питань IoT, матеріали щодо роботи мережевих протоколів, опубліковані зарубіжні та вітчизняні дослідження у галузі побудови високоефективних систем передачі даних*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ.
Теоретичні основи побудови мультипротокольної IoT-інфраструктури.
Проєктування системи централізованого управління IoT-пристроями.
Розробка та реалізація протоколу μAction і серверного агрегатора. Тестування
Висновки.

5. Перелік графічного (ілюстративного) матеріалу:

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Теоретичні основи побудови мультипротокольної IoT-інфраструктури	*Багнюк Н. В., доцент*		
Проєктування мультипротокольної системи централізованого управління	*Багнюк Н. В., доцент*		
Практична реалізація та тестування системи	*Багнюк Н. В., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Гринюк С.В., доцент*		
Показник запозичень тексту	___%		
Академічна доброчесність	*Міскевич О.І., ст.викладач*		

7. Дата видачі завдання 18.06.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми*	До 01.08.2025 р.	
2.	*Аналіз протоколів та технологій IoT, Smart Device, архітектур систем*	До 20.08.2025 р.	
3.	*Розробка мультипротокольної системи та структури µAction*	До 25.09.2025 р.	
4.	*Реалізація протоколу µAction та агрегатора, тестування*	До 20.10.2025 р.	
5.	*Висновки та пропозиції*	До 25.10.2025 р	
6.	*Формування списку використаних джерел*	До 27.10.2025 р.	
7.	*Формування додатків*	До 30.10.2025 р.	
8.	*Оформлення ілюстративного матеріалу*	До 05.11.2025 р.	
9.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	До 11.11.2025 р.	
10.	*Нормоконтроль*	До 29.11.2025 р.	
11.	*Інструментальна перевірка на академічний плагіат*	До 02.12.2025 р.	
12.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	До 09.12.2025 р.	

Здобувач вищої освіти ____________ Ревко Р. Д.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи ____________ Багнюк Н. В.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Ревко Р. Д. Комплексне впровадження та дослідження централізованої системи управління пристроями на базі мультипротокольної мережевої інфраструктури

Кваліфікаційна робота магістра ОП «Комп'ютерна інженерія», спеціальність 123 «Комп'ютерна інженерія». Луцький національний технічний університет, 2025.

Робота містить вступ, три розділи, висновки, список використаних джерел і додатки. Досліджено проблему сумісності та оптимізації комунікаційних протоколів у локальних IoT-системах із фокусом на ефективність, енергоємність та затримки. Проаналізовано MQTT, HTTP, WebSocket, CoAP та UDP – визначено їхні обмеження для пристроїв з обмеженими ресурсами [1].

Запропоновано та реалізовано власний легкий UDP-протокол µAction із фіксованою структурою пакету, механізмами підтвердження на рівні додатка, контролем циклічної надлишковості та повторними передачами з експоненційною затримкою. Реалізовано серверний агрегатор на Node.js та клієнтський стек для ESP32 на базі FreeRTOS [2].

Проведено експериментальні вимірювання продуктивності: навантаження на процесор, використання оперативної пам'яті, затримки, втрати пакетів, пропускна здатність та енергоспоживання. Результати підтвердили перевагу µAction над MQTT/HTTP/WebSocket: зниження навантаження на процесор у 7-14 разів та зменшення затримки до 10-25 мс.

Ключові слова: IoT, ESP32, MQTT, HTTP, WebSocket, UDP, µAction, FreeRTOS.

.

ANNOTATION

Revko R. Comprehensive implementation and research of a centralized device management system based on a multi-protocol network infrastructure

Master's qualification work in the Educational Program «Computer Engineering», specialty 123 Computer Engineering. Lutsk National Technical University, 2025.

The thesis consists of an introduction, three sections, conclusions, references, and appendices. The work investigates communication efficiency, interoperability and latency issues in heterogeneous IoT environments. MQTT, HTTP, WebSocket, CoAP and UDP protocols are analyzed with respect to performance on resource-constrained devices.

A lightweight UDP-based protocol named µAction is designed and implemented, featuring a fixed packet model, CRC16 verification, application-level acknowledgements, and exponential backoff retransmission logic. A Node.js-based server aggregator and an ESP32 FreeRTOS client implementation were developed.

Comprehensive experiments were conducted, including CPU load measurements, memory utilization, end-to-end latency, throughput, packet loss, and energy consumption. µAction demonstrated superior performance compared to MQTT/HTTP/WebSocket, reducing CPU load by 7-14 times and achieving latency levels of 10-25 ms.

Object of research: telemetry exchange processes in local IoT networks.

Subject of research: performance characteristics of the µAction UDP protocol compared to classical IoT protocols.

Keywords: IoT, ESP32, MQTT, HTTP, WebSocket, UDP, µAction, FreeRTOS.

ЗМІСТ

ВСТУП 10
РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ МУЛЬТИПРОТОКОЛЬНОЇ ІНФРАСТРУКТУРИ IOT 13
1.1 Поняття IoT, Smart Device, системи управління 13
1.2 Класифікація мережевих протоколів для IoT 15
1.3 Огляд та порівняння популярних протоколів 18
1.4 Архітектура централізованих систем збору та обробки даних 20
1.5 Актуальні підходи до об’єднання різних протоколів 21
1.6 Проблеми сумісності, затримок, енергоефективності 22
1.7 Методи забезпечення безпеки в мультипротокольних IoT-системах . 23
1.8 Методології тестування продуктивності IoT-систем 24
1.9 Аналіз енергоспоживання IoT-пристроїв 26
РОЗДІЛ 2 ПРОЄКТУВАННЯ МУЛЬТИПРОТОКОЛЬНОЇ СИСТЕМИ ЦЕНТРАЛІЗОВАНОГО УПРАВЛІННЯ 29
2.1 Аналіз функціональних та нефункціональних вимог до системи 29
2.2 Вибір архітектурного підходу та обґрунтування 31
2.3 Обґрунтування вибору ESP32 як апаратної платформи 32
2.4 Обґрунтування вибору програмних технологій 33
2.5 Розробка загальної архітектури системи 35
2.6 Детальний опис розробки власного протоколу µAction 36
2.7 Проєктування топології мережі та взаємодії компонентів 40
2.8 Розробка UML-діаграм класів та послідовностей 41
2.9 Механізми обробки помилок та відновлення з’єднання 42
2.10 Проєктування інтерфейсу моніторингу та API 43
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ 45
3.1 Процес розгортання та конфігурації системи 45

3.2 Особливості реалізації протокольних адаптерів 47
3.3 Методика та результати експериментальних досліджень 48
3.4 Аналіз та інтерпретація отриманих результатів 56
ВИСНОВКИ 59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 63
ДОДАТКИ 65

ВСТУП

У сучасному світі технології Internet of Things (IoT) стрімко розвиваються та охоплюють побутові, промислові й інфраструктурні системи. Дослідження в галузі сенсорних мереж та IoT-платформ свідчать про постійне розширення сфери їх застосування та зростання вимог до ефективності обміну даними між пристроями. Для реалізації таких систем широко застосовуються мікроконтролери під керуванням FreeRTOS, що забезпечує гнучке управління потоками, мінімальні затримки та високу надійність у роботі з реального часу.

Однак розвиток ринку IoT демонструє значну фрагментованість − пристрої різних виробників використовують різні протоколи, що ускладнює побудову уніфікованих систем управління. Аналітичні звіти вказують на стрімке збільшення кількості IoT-вузлів та зростання навантаження на мережеву інфраструктуру [3]. Класичні протоколи, такі як MQTT, розроблені з урахуванням надійності, але створюють відчутне службове навантаження, що обмежує їх ефективність у компактних або енергозалежних вузлах [4].

У системах обробки телеметрії важливу роль відіграють сервіси візуалізації, що дозволяють оперативно аналізувати отримані дані, виявляти аномалії та оптимізувати взаємодію між пристроями [5]. Паралельно з цим зростає потреба у використанні рішень, що здатні ефективно працювати в умовах обмежених ресурсів на периферійних пристроях, що підкреслюють сучасні огляди крайніх обчислень [6].

У цих умовах актуальною є розробка оптимізованих транспортних протоколів, здатних забезпечити мінімальне навантаження на мережу та високу швидкість передачі даних у локальних IoT-системах. Тому запропонований у цій роботі протокол µAction покликаний усунути обмеження існуючих рішень шляхом забезпечення компактності, низької затримки та надійності передачі даних у системах з малими апаратними ресурсами.

Актуальність теми полягає в необхідності створення масштабованих і сумісних IoT-рішень, які здатні об'єднати різні протоколи в одному середовищі та забезпечити високу продуктивність для обмежених апаратних ресурсів. Реалізація власного легкого протоколу µAction і мультипротокольного агрегатора підвищує ефективність роботи локальних систем, дозволяє зменшити затримку та енергоспоживання пристроїв.

Метою роботи є розробка, реалізація та експериментальне дослідження мультипротокольної системи централізованого управління IoT-пристроями з використанням власного UDP-протоколу µAction.

Об'єкт дослідження – процеси передачі телеметрії в локальних IoT-мережах.

Предмет дослідження – ефективність та продуктивність розробленого протоколу µAction у порівнянні з традиційними IoT-протоколами.

Завдання дослідження:

– проаналізувати сучасні протоколи IoT та визначити їхні обмеження у локальних системах з малим обсягом ресурсів;

– спроєктувати архітектуру мультипротокольної системи централізованого управління;

– розробити структуру пакета, механізми підтвердження та логіку повторної передачі протоколу µAction;

– реалізувати серверний агрегатор з підтримкою MQTT, HTTP, WebSocket та µAction;

– виконати експериментальні дослідження продуктивності протоколів та проаналізувати результати.

Апробація результатів (додаток А). Результати роботи представлені на IX Міжнародній студентській науковій конференції «Модернізація та сучасні українські і світові наукові дослідження», яка проводилася 14 листопада 2025 р., м. Житомир [7].

Конференція зареєстрована Державною науковою установою «УкрІНТЕІ» (Посвідчення № 456 від 10.06.2025), що підтверджує наукову значущість отриманих результатів та їх відповідність тематичним напрямам сучасних досліджень у сфері IoT та інформаційних технологій.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ МУЛЬТИПРОТОКОЛЬНОЇ ІНФРАСТРУКТУРИ IoT

1.1 Поняття IoT, Smart Device, системи управління

Інтернет речей (Internet of Things, IoT) являє собою мережеву парадигму, яка дозволяє фізичним об’єктам, оснащеним датчиками, актуаторами та засобами комунікації, взаємодіяти між собою та з зовнішніми системами без прямого втручання людини. Концепція IoT вперше була артикульована Кевіном Ештоном у 1999 році в контексті логістичних систем компанії Procter & Gamble, де пропонувалося використовувати RFID-мітки для автоматичної ідентифікації товарів.

Згідно з визначенням Міжнародного союзу електрозв’язку (ITU-T Y.2060), IoT – це глобальна інфраструктура для інформаційного суспільства, яка забезпечує передові послуги шляхом з’єднання фізичних та віртуальних об’єктів на основі існуючих і еволюційних сумісних інформаційно-комунікаційних технологій. Ця парадигма охоплює не лише технічні аспекти, але й соціально-економічні наслідки, такі як оптимізація ресурсів, підвищення ефективності виробництва та покращення якості життя громадян [8].

Ключовим елементом IoT є розумний пристрій (Smart Device) – автономна апаратно-програмна одиниця, яка поєднує в собі сенсорні компоненти для збору даних з навколишнього середовища (температура, вологість, освітленість, прискорення), процесорні ресурси для локальної обробки інформації та прийняття рішень згідно з вбудованими алгоритмами, засоби комунікації для передачі даних у мережу через Wi-Fi, Bluetooth, Zigbee, LoRa або інші протоколи, виконавчі механізми (актуатори) для реалізації дій на основі отриманих команд або результатів обробки.

Характеристики розумних пристроїв включають: автономність – здатність до самостійної роботи без постійного контролю центрального сервера, інтероперабельність – сумісність з іншими пристроями через стандартизовані протоколи комунікації, масштабованість – можливість інтеграції в мережі з тисячами вузлів без деградації продуктивності, безпеку – механізми захисту даних через шифрування (AES-128/256), аутентифікацію (X.509 сертифікати) та контроль доступу (ACL), енергоефективність – здатність працювати від батареї протягом місяців або років завдяки режимам сну (deep sleep, light sleep). Крім цього, важливими є надійність роботи у складних умовах, стійкість до збоїв та перешкод, а також можливість віддаленого оновлення прошивки та налаштувань для підтримки сучасних функцій і виправлення вразливостей.

Системи управління в IoT (IoT Management Systems) виконують роль координаційного шару, що забезпечує: централізований збір даних від розподілених сенсорів; аналітику для виявлення аномалій, прогнозування відмов та оптимізації процесів; віддалене керування пристроями через RESTful API або MQTT топіки; моніторинг стану мережі та генерацію алертів при критичних подіях. Ці системи класифікуються на децентралізовані (peer-to-peer), де пристрої взаємодіють безпосередньо через mesh-топологію, та централізовані, де існує єдиний хаб (шлюз або хмарний сервер) для агрегації трафіку.

У централізованих системах управління реалізується через ієрархію:

– рівень пристроїв (Device Layer) – локальний контроль та збір даних;

– рівень шлюзу (Gateway Layer) – конверсія протоколів;

– буферизація та первинна аналітика;

– рівень хмари (Cloud Layer) – зберігання великих даних, машинне навчання та візуалізація;

– рівень додатків (Application Layer) – користувацькі інтерфейси та бізнес-логіка.

Еволюцію IoT можна оглянути у таблиці 1.1.

Таблиця 1.1 – Еволюція парадигми IoT [9]

Період	Парадигма	Ключові технології	Масштаб	Приклади
1990-2000	M2M	GSM, RFID	10^3 прист.	SCADA
2000-2010	IoT	IPv6, 6LoWPAN	10^6 прист.	Розумні лічильники
2010-2020	IIoT	MQTT, OPC UA	10^9 прист.	Industry 4.0
2020+	AIoT	5G, Edge AI	10^{12} прист.	Автономні авто

Прикладом є платформа AWS IoT Core, яка обробляє мільярди повідомлень щодня через MQTT-брокер, забезпечуючи правила маршрутизації (IoT Rules Engine) та інтеграцію з AWS Lambda для безсерверної обробки.

1.2 Класифікація мережевих протоколів для IoT

Мережеві протоколи в IoT [10] є фундаментальною основою для комунікації між пристроями, забезпечуючи обмін даними в умовах обмежених ресурсів, таких як низька пропускна здатність (2,4 кБ/с для LoRa), висока затримка (до 1 с для NB-IoT) та критична енергоефективність (10 років від батареї AA). Класифікація протоколів є критичною для архітектурного проєктування мультипротокольних систем, оскільки дозволяє визначити компроміси між надійністю, швидкістю та споживанням ресурсів. Додатково, правильне групування протоколів дає змогу формалізувати вимоги до мережевої підсистеми ще на етапі проєктування та забезпечити сумісність між гетерогенними пристроями. Це особливо важливо у випадку масштабованих IoT-розгортань, де одночасно експлуатуються радіотехнології різних класів та стеків. Узгоджена класифікація дозволяє порівнювати протоколи за єдиними критеріями та обґрунтовувати вибір мережевої архітектури з урахуванням обмежень продуктивності, надійності та енергоспоживання.

Основні критерії класифікації охоплюють:

– рівень моделі OSI (фізичний – модуляція, канальний – MAC, мережевий – маршрутизація);

– транспортний – надійність (TCP – гарантована доставка, UDP – безз’єднаний, найкращі зусилля);

– прикладний – семантика;

– топологію мережі (зірка – один шлюз, mesh – багато маршрутів, дерево – ієрархія);

– вимоги до ресурсів (легкі – менше 10 КБ RAM; важкі – більше 100 КБ);

– сценарії використання (M2M – машина-до-машини, M2C – машина-до-хмари; C2C – хмара-до-хмари).

Огляд класифікацій IoT-протоколів наведено в таблиці 1.2.

Таблиця 1.2 – Класифікація IoT-протоколів за основними критеріями

Категорія	Протокол	OSI рівень	Транспорт	Дальність (м)	Споживання	Застосування
LPWAN	LoRaWAN	PHY-APP	LoRa	15000	10 мА TX	Моніторинг полів
LPWAN	NB-IoT	PHY-APP	LTE	10000	100 мА TX	Smart meters
LPWAN	Sigfox	PHY-APP	UNB	50000	50 мВт·год	Трекінг
PAN	BLE	PHY-APP	BLE	240	10 мА	Wearables
PAN	Zigbee	PHY-APP	IEEE 802.15.4	100	30 мА	Розумний дім
App Layer	MQTT	APP	TCP	-	50-100 мА	Промисловість
App Layer	CoAP	APP	UDP	-	30-70 мА	Constrained nodes
Hybrid	HTTP	APP	TCP	-	100-200 мА	Конфігурація

Протоколи для низькоенергетичних мереж великого радіусу дії (LPWAN):

– LoRaWAN – використовує модуляцію Chirp Spread Spectrum з адаптивною швидкістю передачі (ADR), дальність до 15 км у сільській місцевості при споживанні 10-20 мА під час передачі та менше 1 мкА у режимі сну.

Підтримує три класи пристроїв, а саме: A (двосторонній, енергоефективний), B (beacon для синхронізації), C (постійне прослуховування);

– NB-IoT – стандарт 3GPP для LTE-мереж, забезпечує покриття в приміщеннях (+20 дБ link budget) з пропускною здатністю до 250 кБіт/с. Використовує PSM (Power Saving Mode) для зменшення споживання до 3 мкА у режимі очікування;

– Sigfox – власницький протокол з ultra-narrow band з модуляцією (100 Гц), обмежений 140 повідомленнями по 12 байт на день, орієнтований на дуже низьке споживання (50 мВт-год на повідомлення) [11].

Протоколи для локальних персональних мереж (PAN):

– BLE (Bluetooth Low Energy) – стандарт IEEE 802.15.1 на частоті 2,4 ГГц, використовує GFSK модуляцію та адаптивне псевдовипадкове перестрибування робочої частоти (AFH) для уникнення інтерференції з Wi-Fi. Bluetooth 5.0 забезпечує дальність до 240 м (режим long range) з пропускною здатністю 2 Мбіт/с або 125 кБіт/с для розширеної дальності. Підтримує mesh-топологію (Bluetooth Mesh) до 32 000 вузлів;

– Zigbee – базований на IEEE 802.15.4, працює на частотах 2,4 ГГц (глобально), 915 МГц (Америка), 868 МГц (Європа). Використовує DSSS модуляцію з пропускною здатністю 250 кБіт/с. Підтримує самовідновлювальні mesh-мережі з трьома типами вузлів: координатор, роутер, кінцевий пристрій. Дозволяє до 65 000 вузлів у мережі з максимальною дальністю 100 м [12].

Інтернет-орієнтовані протоколи прикладного рівня:

– MQTT (Message Queuing Telemetry Transport) – publish-subscribe протокол поверх TCP, розроблений IBM у 1999 році для SCADA-систем. Використовує брокер для маршрутизації повідомлень за темами (topics) з трьома рівнями QoS, а саме 0 (at most once, fire-and-forget), 1 (at least once, підтвердження PUBACK), 2 (exactly once, 4-кроковий handshake через PUBREC/PUBREL/PUBCOMP). Накладні витрати MQTT-пакету: фіксований

заголовок 2 байти + змінний заголовок (topic name, packet ID) + payload. Підтримує утримувані повідомлення (retained messages), заповіт (last will testament), постійні сесії (persistent sessions);

– CoAP (Constrained Application Protocol) – RESTful аналог HTTP для UDP, стандартизований RFC 7252. Використовує 4-байтовий фіксований заголовок + опціональні токени та опції. Підтримує методи GET/POST/PUT/DELETE з опцією Observe для pub-sub моделі. Забезпечує надійність через підтверджувані повідомлення з експоненційним відступом (таймаут від 2 до 32 с). Використовує DTLS для безпеки.

Прикладом універсального протоколу (Hybrid) є HTTP/HTTPS – стандарт RESTful комунікації, базований на моделі request-response. HTTP/1.1 (RFC 2616) використовує текстові заголовки з середніми накладними витратами 500+ байт. HTTP/2 (RFC 7540) додає мультиплексування, стиснення заголовків (HPACK) та server push, зменшуючи накладні витрати до 50-200 байт. HTTPS додає TLS 1.3 handshake (+1 RTT) для шифрування. Використовується для конфігурації та оновлення прошивки OTA. WebSocket – повнодуплексний протокол поверх TCP, стандартизований RFC 6455 (2011). Замінює HTTP опитування, забезпечуючи постійне з’єднання з низькою затримкою (менше 10 мс) після HTTP Upgrade handshake. Використовує фреймову структуру з мінімальними накладними витратами (2-14 байт) для подальших повідомлень.

Підтримує текстові (UTF-8) та бінарні фрейми [13, 14].

1.3 Огляд та порівняння популярних протоколів

Детальний аналіз існуючих протоколів IoT виявляє значні відмінності у їхній архітектурі, продуктивності та сферах застосування. MQTT (Message Queuing Telemetry Transport) є одним із найпопулярніших протоколів завдяки механізму publish-subscribe та трьом рівням якості обслуговування (QoS 0, 1, 2).

Протокол працює поверх TCP, забезпечуючи надійну доставку повідомлень, однак це призводить до додаткових накладних витрат: заголовок TCP (20 байт) + заголовок MQTT (мінімум 2 байти) + payload. Для встановлення з'єднання необхідний трьохетапний TCP handshake, що додає латентність 50-150 мс залежно від мережевих умов [15].

HTTP/REST є найбільш універсальним протоколом, підтримуваним всіма платформами, однак має найгірші показники ефективності для IoT. Кожен запит включає HTTP заголовки (200-800 байт), дані cookies, user-agent та інші метадані. Для HTTPS додається TLS handshake (4-5 пакетів), що збільшує латентність до 200-300 мс.

WebSocket забезпечує повнодуплексну комунікацію після початкового HTTP Upgrade handshake. Після встановлення з'єднання, кожен фрейм має мінімальний заголовок 2-14 байт залежно від розміру пакету даних та використання масок. Протокол оптимізований для додатків реального часу (чати, стрімінг), але постійне з'єднання споживає ресурси.

CoAP (Constrained Application Protocol, RFC 7252) спроєктований спеціально для обмежених пристроїв. Працює поверх UDP з мінімальним заголовком 4 байти, підтримує RESTful архітектуру (GET/POST/PUT/DELETE), має механізм підтвердження (CON/NON/ACK) та опцію Observe для асинхронних повідомлень. Використовує DTLS для безпеки. Розмір пакету зазвичай 10-40 байт, що робить CoAP ефективнішим за HTTP, однак складність парсингу опцій та необхідність реалізації машини станів для Observe збільшують навантаження на CPU до 15-20 % на embedded платформах.

UDP як базовий транспортний протокол забезпечує мінімальні накладні витрати (8 байт заголовок), відсутність встановлення з'єднання та низьку латентність. Однак UDP не гарантує доставку, порядок пакетів та захист від дублювання, що вимагає реалізації цих функцій на рівні прикладного протоколу для локальних мереж з низьким рівнем втрат (< 1 %).

1.4 Архітектура централізованих систем збору та обробки даних

Централізовані системи управління IoT зазвичай побудовані за трирівневою архітектурою: рівень пристроїв (Device Layer), рівень шлюзу/агрегатора (Gateway/Edge Layer) та рівень хмари (Cloud Layer). Рівень пристроїв включає сенсори, актуатори та мікроконтролери (ESP32, STM32, Arduino), які збирають дані та виконують команди. Ці пристрої мають обмежені обчислювальні ресурси (512 кБ RAM, 240 МГц CPU для ESP32) та часто працюють від батареї, що накладає жорсткі обмеження на протоколи комунікації [16].

Рівень шлюзу виконує роль посередника між пристроями та хмарою, забезпечуючи агрегацію даних, протокольну конвертацію, локальну аналітику та буферизацію при втраті з’єднання. Сучасні шлюзи реалізуються на Raspberry Pi, промислових комп’ютерах або серверах з Docker.

Рівень хмари включає бази даних (MongoDB для даних часових рядів з TTL-індексами), аналітичні сервіси (Apache Spark, InfluxDB), системи візуалізації (Grafana) та машинне навчання для прогнозування аномалій. Взаємодія між рівнями відбувається через RESTful API або брокери повідомлень (RabbitMQ, Kafka). Критично важливим є забезпечення високої доступності через реплікацію (MongoDB Replica Set з 3 вузлів), балансування навантаження (Nginx) та автоматичний перехід на резерв.

Парадигма edge computing переносить частину обчислень на рівень шлюзу для зменшення затримки та навантаження на мережу. Наприклад, фільтрація аномальних значень (температура > 80 °C), агрегація даних (середнє за 1 хвилину) та прийняття критичних рішень (вимкнення обладнання при перегріві) можуть відбуватися локально без звернення до хмари, що знижує наскрізну затримку з 200-500 мс до 20-50 мс.

Приклад архітектури IoT зображено на рисунку 1.1.

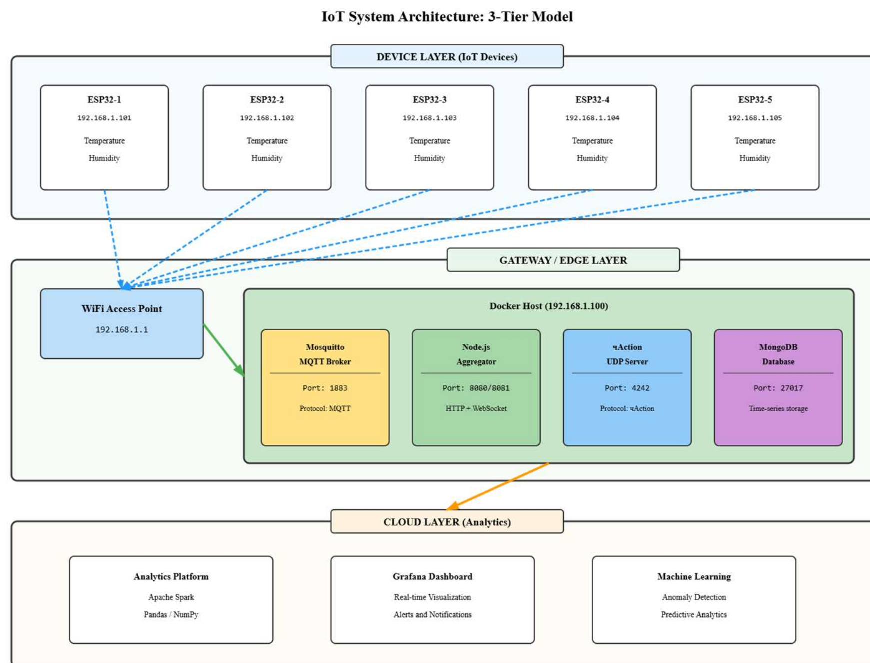

Рисунок 1.1 – Трирівнева архітектура IoT системи з мультипротокольним шлюзом

1.5 Актуальні підходи до об’єднання різних протоколів

Проблема інтеграції гетерогенних протоколів вирішується через патерн Protocol Gateway (Protocol Adapter). Кожен протокол інкапсулюється в адаптер, який реалізує єдиний інтерфейс IProtocolAdapter з методами connect(), send(), receive(), disconnect(). Це дозволяє додавати нові протоколи без зміни бізнес-логіки системи. В Node.js реалізації використано Strategy Pattern [17].

Як видно на рисунку 1.2, всі адаптери протоколів (µAction, MQTT, HTTP, WebSocket) передають нормалізовані метрики у Protocol Gateway, який через єдиний інтерфейс IProtocolAdapter забезпечує уніфіковану обробку повідомлень. Це дозволяє Node.js агрегатору виконувати валідацію, збагачення даних та надавати інтерфейси REST, WebSocket та UDP для подальшого використання.

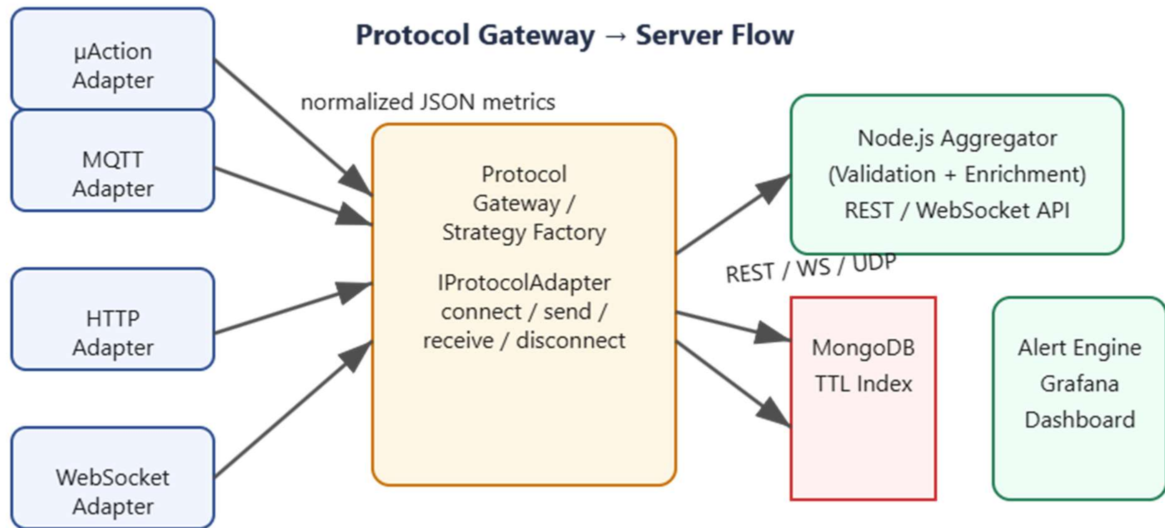

Рисунок 1.2 – Обробка мультипротокольності

Завдяки такій архітектурі додавання нового протоколу зводиться до реалізації нового адаптера без змін бізнес-логіки, що підвищує масштабованість та підтримуваність системи.

1.6 Проблеми сумісності, затримок, енергоефективності

Фрагментація екосистеми IoT призводить до проблем взаємодії: пристрій з Zigbee не може безпосередньо взаємодіяти з пристроєм на LoRaWAN без шлюзу-перекладача. Відсутність стандартизованих API викликає залежність від постачальника: перехід з AWS IoT Core на Azure IoT Hub вимагає переписування коду через різні протоколи автентифікації (X.509 vs SAS tokens) та формати даних [18].

Механізм керування перевантаженням TCP в IoT-мережах з високою втратою пакетів (2-5 % у промислових середовищах) призводить до експоненційної затримки повторної передачі: після втрати пакету TCP зменшує розмір вікна, що знижує пропускну здатність з 200 Кбіт/с до 20-50 Кбіт/с. Для

100-байтового пакету затримка зростає з 50 мс до 500-1000 мс після 2-3 повторних передач. Протоколи на базі UDP уникають цієї проблеми, але вимагають реалізації надійності на рівні додатка [19].

Проблема енергоспоживання критична для пристроїв на батарейках. TCP вимагає підтримки з'єднання через пакети keep-alive (кожні 30-120 с), що не дозволяє мікроконтролеру перейти в глибокий сон (споживання 10 мкА). MQTT з QoS 1 потребує очікування PUBACK, блокуючи сон на 50-200 мс. Натомість UDP дозволяє надіслати пакет за 10-20 мс і негайно перейти в сон, що збільшує час роботи від батареї 2000 мА·год з 30 діб до 90-120 діб.

Варіація затримки в мережах Wi-Fi сягає 10-50 мс через конкуренцію за канал CSMA/CA. При 20 пристроях на одній точці доступу 95-й перцентиль затримки зростає з 15 мс до 80-120 мс. Використання QoS (WMM) для IoT-трафіку знижує варіацію затримки до 5-15 мс, але вимагає правильної конфігурації роутера та пристроїв (esp_wifi_set_ps(WIFI_PS_MIN_MODEM) на ESP32).

1.7 Методи забезпечення безпеки в мультипротокольних IoT-системах

Безпека IoT систем включає три рівні: мережевий (TLS/DTLS), прикладний (автентифікація, авторизація) та фізичний (secure boot, шифрування flash). TLS 1.3 забезпечує шифрування даних через AES-128-GCM, автентифікацію через X.509 сертифікати та секретність через ECDHE. Однак TLS-handshake додає 4-5 обмінів (200-500 мс затримки) та споживає 40-60 КБ оперативної пам'яті на ESP32, що неприйнятно для обмежених пристроїв.

DTLS (Datagram TLS) адаптує TLS для UDP, але має проблему з переупорядкуванням пакетів та потребує вікна захисту від повторного відтворення (до 64 пакетів у пам'яті). ESP-IDF реалізація DTLS (mbedTLS) займає 80+ кБ flash та 20 кБ RAM. Для μAction протоколу DTLS не підходить

через накладні витрати: 13-байтовий заголовок DTLS + 16-байтовий тег GCM збільшують 43-байтовий пакет до 72 байт (+ 67 %).

Безпека на рівні додатка через симетричне шифрування (AES-128 у режимі CTR або GCM) є оптимальним компромісом. Попередньо розподілений ключ зберігається в NVS з увімкненим шифруванням флеш-пам'яті. Корисне навантаження шифрується перед відправкою: encrypted_payload – AES_CTR(key, iv, payload), ініціалізаційний вектор iv включається в пакет. Розмір пакету збільшується на 16 байт (IV), але без TLS-handshake. Продуктивність: апаратний прискорювач AES на ESP32 обробляє 8-10 МБ/с, шифрування 38-байтового корисного навантаження займає < 1 мс.

Автентифікація через JWT (JSON Web Tokens) на рівні HTTP/WebSocket API: клієнт отримує токен після логіна (POST /auth/login), включає його в Authorization: Bearer <token> заголовок. Токен підписується HMAC-SHA256 з секретним ключем, має термін дії (1-24 години). Для MQTT використовується username/password в CONNECT пакеті.

Secure boot на ESP32 гарантує завантаження лише підписаної прошивки. Приватний ключ зберігається в eFuse (одноразово програмований), публічний ключ вбудовано в завантажувач. Шифрування флеш-пам'яті захищає прошивку AES-256, ключ генерується при першому завантаженні та записується в eFuse. Це запобігає зчитуванню прошивки через UART або JTAG.

1.8 Методології тестування продуктивності IoT-систем

Тестування продуктивності IoT-систем передбачає оцінювання кількох ключових характеристик, зокрема завантаження процесора, використання оперативної пам'яті, мережевої затримки, пропускної здатності та споживання енергії. Завантаження процесора на мікроконтролерах родини ESP32 визначається засобами операційної системи реального часу FreeRTOS. Функція

vTaskGetRunTimeStats() надає інформацію про частку часу, протягом якого кожна задача перебувала у стані виконання. Для отримання узагальненого показника використовується інформація про сумарний час роботи системи, що формується через uxTaskGetSystemState(), після чого аналізується співвідношення активного часу задач до загального часу роботи.

Стан динамічної пам’яті оцінюється за допомогою функцій esp_get_free_heap_size(), esp_get_minimum_free_heap_size() та heap_caps_get_lar gest_free_block(), які відповідно надають дані про поточний обсяг вільної пам’яті, найменший зафіксований обсяг за весь час роботи та найбільший безперервний блок, доступний для виділення. Окремим критичним показником є рівень фрагментації. Його визначають шляхом порівняння найбільшого безперервного блока з загальним обсягом доступної пам’яті. Якщо співвідношення свідчить про втрату приблизно третини доступного простору, це розглядається як ознака потенційних проблем з динамічним виділенням пам’яті під час тривалої роботи системи.

Мережева затримка визначається шляхом вимірювання часової різниці між моментом відправлення пакета пристроєм та моментом його отримання сервером. На боці мікроконтролера час фіксується за допомогою esp_timer_get_time(), тоді як серверна частина реєструє час надходження пакета через Date.now() у середовищі Node.js. Для забезпечення достовірності результатів потрібна синхронізація годинників обох сторін, яка реалізується за допомогою протоколу NTP через esp_sntp_init(). Після збирання вибірки обсягом 500-1000 значень обчислюються такі статистичні характеристики, як середнє значення, медіана, 95-й та 99-й перцентилі, а також стандартне відхилення, що дозволяє комплексно оцінити стабільність затримок.

Пропускна здатність визначається як максимальна кількість повідомлень, які система може успішно доставити за одну секунду. Методика вимірювання передбачає надсилання тисячі повідомлень з мінімальним інтервалом між ними

та подальший підрахунок підтверджень отримання. Для HTTP враховуються відповіді типу «200 OK». Експериментальні дослідження на ESP32-S3 при тактовій частоті 240 МГц показують такі орієнтовні результати: приблизно 120 повідомлень за секунду для WebSocket, 90 – для MQTT та 45 – для HTTP.

Споживання енергії оцінюється за допомогою датчика струму INA219, підключеного послідовно до лінії живлення ESP32. Зчитування значень струму здійснюється через інтерфейс I²C з частотою сто разів на секунду, після чого результати інтегруються у часі, що дозволяє визначити витрати енергії для заданого набору операцій. Для порівняння протоколів розраховується енергоспоживання, необхідне для передавання тисячі повідомлень. Отримані значення становлять: для WebSocket – близько 123 міліамер-годин, для MQTT – 132, для HTTP – 162. Усі вимірювання проводяться за однакових умов, зокрема при використанні однакового режиму енергозбереження Wi-Fi (WIFI_PS_MIN_MODEM), що забезпечує коректність порівняння.

1.9 Аналіз енергоспоживання IoT-пристроїв

Мікроконтролери родини ESP32 підтримують п’ять основних режимів енергоспоживання:

– Active – повністю активний стан із передаванням даних через Wi-Fi, у якому струм становить приблизно від 160 до 260 міліампер;

– Modem Sleep – процесор продовжує працювати, проте модуль Wi-Fi вимикається між службовими сигналами, що зменшує струм до 20-30 міліампер;

– Light Sleep – процесор призупинений, а допоміжний таймер реального часу залишається активним, струм становить близько 0,8-1,2 міліампера;

– Deep Sleep – більшість периферійних модулів вимкнено, працює лише частина підсистем, струм сягає від 10 до 150 мікроампер залежно від конфігурації;

– Hibernation – найнижчий рівень активності, у якому працює лише таймер пробудження, а споживання електроенергії становить приблизно 5 мікроампер.

Перехід між режимами здійснюється відповідними викликами: для режиму Modem Sleep використовується esp_wifi_set_ps(WIFI_PS_MIN_MODEM), для Light Sleep – esp_light_sleep_start(), а для Deep Sleep – esp_deep_sleep_start().

Для сценаріїв періодичної роботи, наприклад передавання телеметрії раз на шістдесят секунд, типовий профіль енергоспоживання включає декілька фаз. Основний проміжок часу припадає на перебування у глибокому сні, протягом якого активним залишається лише таймер. Після пробудження відбувається відновлення роботи Wi-Fi, що супроводжується короткочасним піковим споживанням струму. Далі система виконує передавання даних протягом кількох секунд, після чого завершує роботу і знову переходить у режим сну. У середньому один цикл витрачає частку міліампер-години. За добу, що містить 1440 таких циклів, накопичується сумарне споживання на рівні кількох сотень міліампер-годин.

Телеметрія в даному дослідженні розглядається як автоматичний збір і передавання даних про стан пристрою на віддалений сервер або в хмару. До таких даних відносяться показники завантаження процесора, використання оперативної пам’яті, енергоспоживання та інші параметри, що дозволяють оцінювати роботу системи в реальному часі.

Порівняльні дослідження свідчать про різну ефективність протоколів у сценаріях живлення від батареї. За умови циклу тривалістю шістдесят секунд та за однакової ємності батареї. Протоколи WebSocket та MQTT демонструють автономність – приблизно 4,2 та 3,8 доби відповідно через підтримання з’єднання або службовий обмін повідомленнями. Протокол HTTP має найменший час роботи – близько 3,2 доби, оскільки щоразу ініціює нове підключення.

Для збільшення автономності до місяця або більше застосовують комплекс оптимізацій. Серед них – збільшення інтервалу між передачами до п’яти хвилин,

перехід у режим Hibernate з мінімальним споживанням та зниження потужності передавача Wi-Fi за допомогою функції esp_wifi_set_max_tx_power().

Альтернативою Wi-Fi можуть бути інші технології бездротового зв'язку. Bluetooth Low Energy забезпечує значно нижче тимчасове енергоспоживання – у межах 10-50 міліампер під час передавання, проте має обмежений радіус дії та вимагає проміжний шлюз для доступу в мережу. Технологія LoRaWAN, своєю чергою, дозволяє досягати багаторічної роботи від батареї ємністю дві тисячі міліампер-годин навіть при відправленні одного повідомлення на годину, проте має низьку швидкість передавання та потребує спеціалізований шлюз зв'язку. Вибір протоколу визначається співвідношенням між швидкістю передавання, радіусом дії та тривалістю автономної роботи, де Wi-Fi орієнтований на високу швидкість, BLE – на баланс між ефективністю та споживанням, а LoRaWAN – на максимальну автономність за рахунок низької пропускної здатності.

РОЗДІЛ 2
ПРОЄКТУВАННЯ МУЛЬТИПРОТОКОЛЬНОЇ СИСТЕМИ ЦЕНТРАЛІЗОВАНОГО УПРАВЛІННЯ

2.1 Аналіз функціональних та нефункціональних вимог до системи

Розробка системи збору та обробки телеметрії для IoT-пристроїв потребує чіткого визначення вимог, що визначають її функціональність, продуктивність, масштабованість та обмеження середовища, у якому вона працюватиме. Оскільки система повинна обслуговувати значну кількість ESP32-вузлів, підтримувати обмін даними через декілька мережевих протоколів і забезпечувати стабільну роботу в режимі реального часу, особливо важливим є формування вимог, які охоплюють як логіку взаємодії компонентів, так і технічні характеристики інфраструктури.

Додатково враховуються вимоги до надійності передачі даних, стійкості до збоїв, підтримки різних сценаріїв навантаження та можливості подальшого розширення без суттєвих змін архітектури. Чітке формулювання таких вимог дозволяє узгодити очікувану поведінку системи, мінімізувати ризики некоректної роботи в умовах реального розгортання та забезпечити відповідність обраних технологій поставленим цілям.

Наведені нижче функціональні та нефункціональні вимоги визначають архітектуру системи, рівень її надійності, швидкодії та можливості масштабування, а також встановлюють обмеження, що впливають на вибір технологій і підходів до реалізації.

Функціональні вимоги до системи включають [20]:

– підтримка мінімум 4 протоколів (MQTT, HTTP, WebSocket, μAction) з можливістю розширення;

– збір метрик з ESP32 пристроїв: час безперервної роботи, статистика пам’яті heap (вільна/мінімальна/найбільший блок), завантаження процесора у відсотках, рівень сигналу Wi-Fi, температура, інформація про мікросхему;

– централізоване зберігання даних в MongoDB з TTL індексами (автоматичне видалення даних понад 90 діб);

– візуалізація даних у реальному часі на веб-панелі через надсилання повідомлень за протоколом WebSocket;

– RESTful API для запитів історичних даних з фільтрами (device_id, time range, protocol);

– система сповіщень при аномаліях (CPU > 90 %, heap < 50 кБ, temp > 80° C, RSSI < -80 dBm).

Нефункціональні вимоги:

– латентність E2E < 50 мс для µAction протоколу (від відправки на ESP32 до запису в БД);

– Throughput мінімум 100 повідомлень/с на один протокол;

– підтримка 50+ одночасно підключених пристроїв без деградації продуктивності;

– доступність (Availability) 99,5 % (downtime < 3,6 год/міс) через автоматичний restart при збоях;

– масштабованість: можливість горизонтального масштабування через додаткові екземпляри агрегатора;

– безпека: шифрування даних (HTTPS/WSS), автентифікація (JWT tokens), авторизація (RBAC).

Обмеженнями системи розробки є:

– апаратні: ESP32 має 52 кБ SRAM, 4 МБ Flash, Wi-Fi дальність 50-100 м.;

– мережеві: локальна Wi-Fi мережа 2,4 ГГц з пропускною здатністю ~50 Мбіт/с реальною при 802.11n;

– бюджетні: використання open-source технологій (Node.js, MongoDB, Mosquitto) без ліцензійних витрат;

– часові: розробка за 4 місяці (грудень 2024 – березень 2025);

– експлуатаційні: розгортання на одному сервері (4 CPU, 8 ГБ RAM, 256 ГБ SSD) через Docker Compose.

2.2 Вибір архітектурного підходу та обґрунтування

Розглядалися три архітектурні підходи: монолітний (single Node.js server), модульний (окремі сервіси для кожного протоколу) та мікросервісний (Docker containers). Монолітна архітектура найпростіша у розробці, але має недоліки − збій одного протоколу викликає збій всієї системи, складність масштабування (тільки вертикальне масштабування), неможливість оновлення одного модуля без простою системи [21].

Модульна архітектура з окремими процесами для кожного протоколу (MQTT адаптер на порту 1883, HTTP адаптер на порту 8080, тощо.) забезпечує ізоляцію: збій MQTT не впливає на HTTP, однак вимагає IPC (Inter-Process Communication) через Redis Pub/Sub або RabbitMQ, що додає затримку 5-10 мс та ускладнює керування життєвим циклом (сервіси systemd, PM2).

Мікросервісна архітектура через Docker Compose обрана як оптимальна:

– кожен сервіс (mosquitto, mongodb, node-aggregator) в окремому контейнері з власними ресурсами;

– ізоляція: падіння агрегатора не впливає на MongoDB;

– легке масштабування: docker-compose scale aggregator=3 для горизонтального масштабування;

– відтворюваність: docker-compose up на будь-якій машині розгортає ідентичне середовище;

– версіонування: Docker images з тегами (aggregator:v1.2.3) дозволяють «відкат»;

– моніторинг: Docker stats показує CPU/RAM кожного контейнера. Docker Compose файл структура: версія 3.8, 3 сервіси (mosquitto з томом для конфігурації, mongodb з томом для даних та ініціалізацією реплікаційного набору, агрегатор з змінними середовища для DB_URI), 2 мережі (frontend для зовнішніх з'єднань, backend для внутрішньої комунікації), 2 томи (mongo-data, mosquitto-data). Health checks забезпечують автоматичний рестарт: для MongoDB – mongo --eval 'db.stats()' кожні 30 с, для агрегатора – curl localhost:8080/ health [22].

2.3 Обґрунтування вибору ESP32 як апаратної платформи

ESP32 обраний як апаратна платформа через унікальне поєднання характеристик [23]:

– Dual-core Xtensa LX6 CPU до 240 МГц дозволяє виконувати мережеві роботи на Core 0 (закріплено до PRO_CPU) та роботу додатку на Core 1;

– 520 кБ SRAM достатньо для FreeRTOS (heap 200+ кБ вільної після ініціалізації Wi-Fi/LWIP), 4 МБ флеш пам'яті для прошивки та файлової системи (SPIFFS/LittleFS);

– вбудований Wi-Fi 802.11 b/g/n з підтримкою STA/AP/STA+AP режимів, апаратний криптоакселератор для AES/SHA;

– багата периферія: 2× UART, 2× I2C, 3× SPI, 18× ADC, 2× DAC, вбудований сенсор температури.

Як видно з таблиці 2.1, ESP32-S3 має найкращий баланс продуктивності, вбудованого Wi-Fi/BLE та ціни. STM32F4 вимагає зовнішній Wi-Fi модуль (ESP-01 або WizNet W5500), що збільшує вартість та споживання. Raspberry Pi

Pico W дешевший, але має слабшу підтримку RTOS та менше RAM. Arduino Uno R4 WiFi надмірно дорогий (25$) для IoT сенсорів.

ESP-IDF (Espressif IoT Development Framework) забезпечує професійну розробку: FreeRTOS 10.x з пріоритетним багатозадачністю, lwIP 2.1.3 TCP/IP стек з BSD сокетами API, компоненти для MQTT (esp-mqtt), HTTP клієнт/сервер (esp_http_client/esp_http_server), OTA оновлення (esp_ota). Підтримка CMake build system, компонентів з Espressif Component Registry, інтеграція з PlatformIO та VS Code.

Таблиця 2.1 – Порівняння мікроконтролерів для IoT

MCU	CPU (МГц)	RAM (кБ)	Flash (МБ)	WiFi	BLE	Ціна ($)	Екосистема
ESP32-S3	2×240	512	4-8	Так	Так	2-3	Відмінна
STM32F4	1×168	192	1	Немає	Немає	5-8	Хороша
RPi Pico W	2×133	264	2	Так	Немає	6	Середня
Arduino Uno R4	1×48	32	0,25	Так	Немає	25	Відмінна

2.4 Обґрунтування вибору програмних технологій

FreeRTOS обраний як RTOS для ESP32 через:

– офіційну підтримку Espressif (включений в ESP-IDF);

– пріоритетний планувальник з пріоритетами 0-24 (0 = найнижчий, вище = пріоритетніша задача);

– малий розмір ядра (~10 кБ), швидке переключення контексту (~5 мкс);

– механізми синхронізації: м’ютекси (з успадкуванням пріоритету), семафори (бінарні, лічильні), черги (FIFO, безпечні для потоків), групи подій;

– керування пам’яттю: 5 схем (heap_1 до heap_5), в ESP32 використовується heap_caps для розподілу DRAM/IRAM.

Node.js v20 LTS обраний як агрегатор через:

– асинхронний неблокуючий ввід/вивід (libuv event loop) – ідеально для мережевих задач (обробка 1000+ одночасних з'єднань на 1 ядро процесора);

– велика кількість бібліотек: mqtt.js для MQTT клієнта/сервера, express для HTTP, socket.io для WebSocket, dgram для UDP;

– просту інтеграцію з MongoDB через mongoose ODM;

– легкість розробки: JavaScript/TypeScript з async/await замість вкладених функцій;

– продуктивність: V8 engine компілює JS у машинний код, продуктивність близька до Go/Rust для задач з ввід/вивід.

MongoDB v7.0 обрана для зберігання даних за часом через:

– гнучку схему: JSON-документи не вимагають фіксованих колонок, легко додавати нові поля (наприклад, psram_free для ESP32-S3-PSRAM);

– індекси часу життя: createIndex({timestamp: 1}, {expireAfterSeconds: 7776000}) автоматично видаляє дані старше 90 діб, зменшуючи розмір бази;

– пакет обробки даних для аналітики: db.metrics.aggregate([{$match: {device_id: 'ESP32-A1B2C3'}}, {$group: {_id: '$protocol', avg_cpu: {$avg: '$metrics.cpu_load'}}}]);

– реплікація для високої доступності (3 ноди, автоматичне перемикання за 10-30 с);

– розподілення даних для горизонтального масштабування (за device_id).

Mosquitto MQTT сервер обраний через:

– легкість: 200 кБ бінарник, < 5 МБ RAM для 1000 клієнтів;

– підтримку MQTT 5.0 (спільні підписки, запит-відповідь);

– авторизацію через ACL файл: тема device/+/metrics доступна для читання і запису для device_id;

– збереження повідомлень: повідомлення з QoS 1/2 зберігаються на диск при падінні сервера;

– режим моста для об’єднання серверів (підключення до хмарного MQTT HiveMQ/AWS IoT).

2.5 Розробка загальної архітектури системи

Архітектура системи побудована за принципом шаблону адаптера протоколу: кожен протокол інкапсулюється в окремий модуль (MQTTAdapter, HTTPAdapter, WSAdapter, uActionAdapter), який реалізує єдиний інтерфейс з методами connect(), onMessage(callback), disconnect(). шаблон фабрики вибирає адаптер на основі конфігурації: const adapter = ProtocolFactory.create(config.protocol). це забезпечує слабке зв’язування: бізнес-логіка (збереження в базі, перевірка тривог) не залежить від протоколу.

Broadcast vs Priority Mode: система підтримує два режими роботи пристроїв [24].

Broadcast Mode − пристрій відправляє дані по всіх доступних протоколах одночасно (MQTT + HTTP + µAction), агрегатор дедуплікує по device_id + timestamp. Переваги − надійність (якщо MQTT брокер недоступний, дані прийдуться через HTTP), недоліки − 3-х мережевий трафік.

Priority Mode − пристрій спочатку пробує µAction (найефективніший), якщо таймаут – переключення на WebSocket, після чого на MQTT та на HTTP. Реалізація прикладу у лістингу 2.1.

Лістинг 2.1 – Переключення на WebSocket

```
esp_err_t err = uaction_send();
if (err != ESP_OK) {
    ws_send();
}
```

кінець лістингу 2.1

2.6 Детальний опис розробки власного протоколу μAction

Мотивація розробки μAction − аналіз існуючих протоколів показав додаткове навантаження 45-800 байт на повідомлення, що неприйнятно для метрик, які передаються кожні 5-60 секунд. MQTT мінімальний пакет (PUBLISH з QoS 0, тема 't', дані 1 байт) = 2 байти фіксованого заголовка + 3 байти змінного заголовка (довжина теми + тема) + 1 байт даних = 6 байт MQTT + 20 TCP + 20 IP + 14 Ethernet = 60 байт загалом. Для 38-байтного поля даних: 60 + 38 = 98 байт (додаткове навантаження 61 %). HTTP ще гірше − понад 200 байт заголовки + 38 байт даних = понад 238 байт.

Дизайн μAction спрямований на мінімальне додаткове навантаження: тільки необхідні поля, фіксоване поле корисних даних (без динамічного розбору), UDP транспорт (без TCP handshake/ACK), бінарний формат (без JSON/XML). C

Структура пакету [25]:

- версія та тип пакету – 4 + 4 біт;
- довжина корисних даних – N байт;
- номер пакету – 1 байт;
- CRC16 – 2 байти.

Payload N байт. Для тесту, ESP32 має корисні метричні дані – 38 байт: загальний розмір 5 + 38 = 43 байти + 8 UDP + 20 IP – загалом 71 байт.

Byte 0 (версія/тип): старші 4 біти – версія (поточна 0x1, дозволяє 16 версій), молодші 4 біти – тип пакету: 0x1 – DATA (метрика від пристрою), 0x3 = ACK (підтвердження від сервера), 0x5 – COMMAND (команда від сервера до пристрою), 0x7 – ERROR (повідомлення про помилку). Приклад: 0x11 – version 1, type DATA.

Byte 1 (довжина даних): розмір корисних даних 0-255 байт. Для ESP32 метрика = 38 байт (фіксований, але поле дозволяє змінний розмір корисних даних в майбутньому).

Byte 2 (номер пакету): лічильник пакетів 1-255 (0 пропускається), циклічний. Використовується для:

– виявлення втрачених пакетів (якщо seq стрибнув з 10 на 12, пакет 11 втрачений);

– дедуплікація (якщо seq 15 приходить двічі, другий – дублікат);

– послідовність (якщо seq 20 прийшов раніше 19, буферизувати до отримання 19).

Структура метричних даних (38 байт) для ESP32 (рис. 2.1):

– uptime_ms – 4 байти, час безперервної роботи пристрою в мілісекундах;

– heap_free – 4 байти, обсяг доступної динамічної пам’яті;

– heap_min_free – 4 байти, мінімальне зафіксоване значення вільної Heap-пам’яті;

– heap_largest – 4 байти, розмір найбільшого доступного безперервного блоку Heap;

– heap_internal_free – 4 байти, обсяг доступної внутрішньої SRAM;

– psram_free – 4 байти, обсяг вільної PSRAM;

– cpu_load – 4 байти, завантаження процесора у відсотках, помножене на 100;

– wifi_rssi – 1 байт, рівень сигналу Wi-Fi у dBm;

– temp_c – 2 байти, температура у °C, помножена на 100;

– reset_reason – 2 байти, код причини перезапуску мікроконтролера;

– chip_cores – 1 байт, кількість доступних процесорних ядер;

– chip_revision – 1 байт, апаратна ревізія чипа;

– chip_features – 4 байти, бітова маска – набір апаратних можливостей чипа;

– wifi_channel – 1 байт, активний Wi-Fi канал;

– wifi_authmode – 1 байт, режим аутентифікації Wi-Fi;

– reserved – 2 байти, зарезервовано для майбутнього розширення.

Два байти було зарезервовано для можливого розширення протоколу. DATA/ACK exchange (рис. 2.2):

– пристрій відправляє DATA пакет (43 байти) на server_ip: 4242 UDP;

– сервер перевіряє CRC, якщо OK – зберігає в БД та відправляє ACK (5 байт: version/type – 0x13, length – 0, sequence – копія з DATA, CRC16);

– пристрій чекає ACK з таймаутом 100 мс., якщо немає відповіді – повтор до 3 разів з експоненційним збільшенням інтервалу (100, 200, 400 мс);

– після 3 невдалих спроб – переключаємось на інший протокол (WebSocket або MQTT);

Гарантії надійності:

– підтвердження на рівні додатку забезпечує доставку (на відміну від чистого UDP);

– контрольна сума CRC16 виявляє пошкоджені пакети (ймовірність помилки 1/65536);

– номер пакету дозволяє виявити втрати та перестановки;

– механізм повтору з експоненційним збільшенням інтервалу уникнення перевантаження мережі.

Обмеження:

– номер пакету обмежений 255 (прийнятно для швидкості < 50 повідомлень/с);

– відсутність шифрування (можна додати AES-GCM у дані);

– UDP ненадійний у перевантажених мережах (> 5 % втрат).

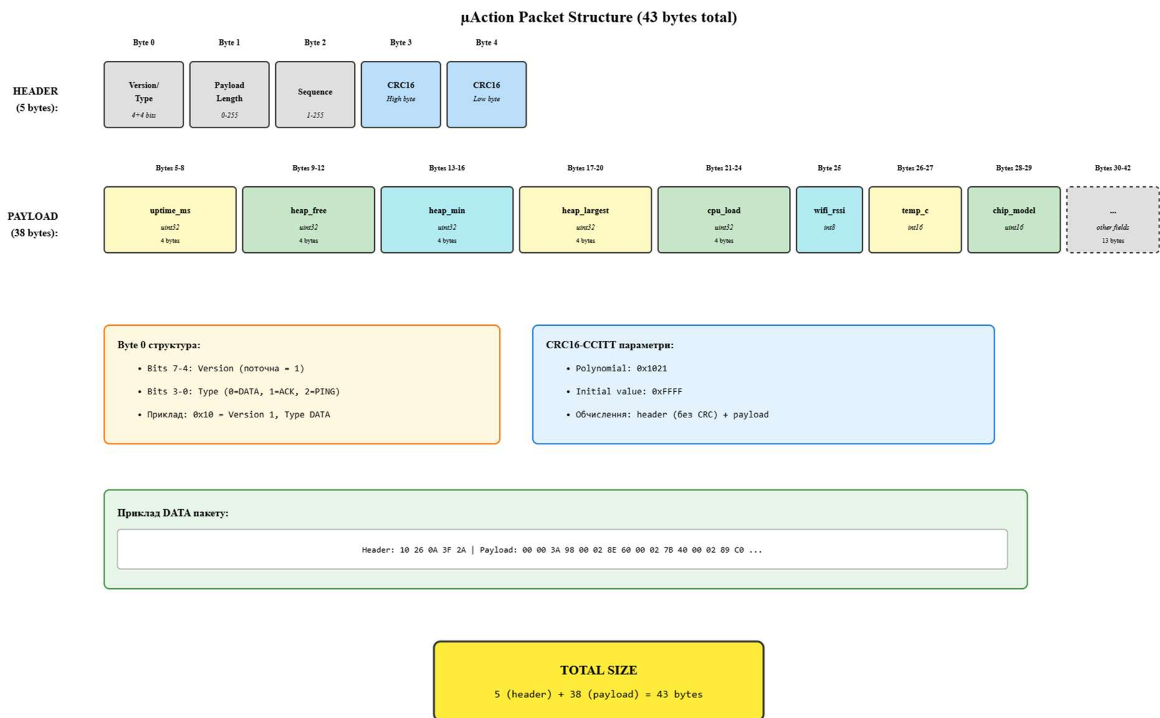

Рисунок 2.1 – Структура пакету протоколу µAction (43 байти)

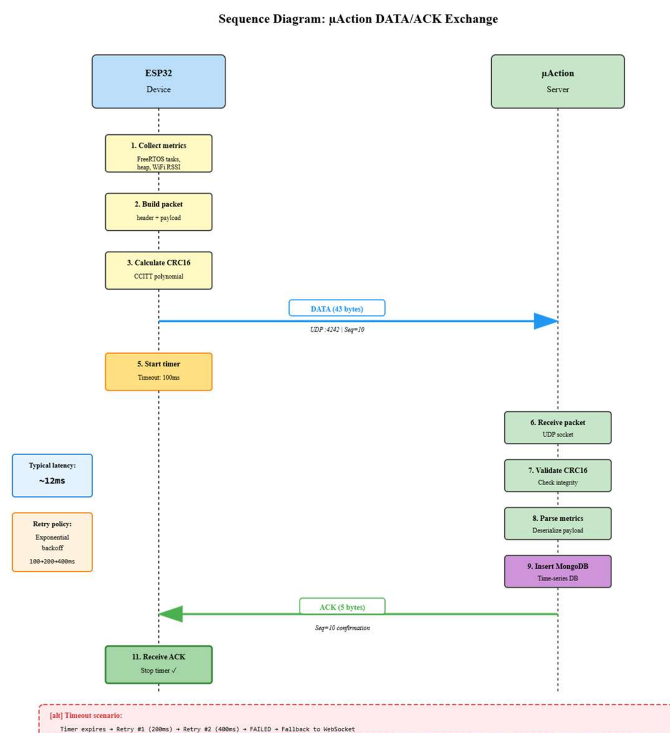

Рисунок 2.2 – Діаграма послідовності DATA/ACK обміну між ESP32 та сервером

Структура пакету µAction та послідовність обміну DATA/ACK ілюструють мінімалістичний дизайн протоколу, орієнтований на зменшення службового навантаження, швидку обробку та виявлення помилок на рівні додатку. Фіксований формат, наявність CRC16 та лічильника пакетів забезпечують базову надійність поверх UDP, а компактність пакету дозволяє передавати метрики з мінімальними витратами ресурсу.

2.7 Проєктування топології мережі та взаємодії компонентів

Топологія мережі в експериментальному стенді побудована за схемою «зірка». Усі пристрої ESP32 підключаються до однієї точки доступу Wi-Fi (роутер TP-Link Archer AX12, стандарт 802.11n, діапазон 2,4 + 5 ГГц, чотири антени), яка з'єднана з сервером дротовим інтерфейсом Ethernet зі швидкістю 1 Гбіт/с. У ролі сервера використано ПК з процесором AMD Ryzen 5 3550H, 8 ГБ оперативної пам'яті та SSD-накопичувачем обсягом 256 ГБ, під керуванням операційної системи EndeavourOS. Усі вузли (сервер, точка доступу, ESP32-пристрої) розташовані в одній локальній підмережі 192.168.1.0/24, що спрощує конфігурацію та забезпечує низьку затримку, оскільки від пристрою до сервера є лише один мережевий перехід.

Серверна частина системи розгорнута у контейнерному середовищі з використанням Docker Compose. Логіка розгортання передбачає три основні сервіси, об'єднані у дві віртуальні мережі. До першої, внутрішньої мережі типу backend, належать база даних MongoDB, брокер повідомлень MQTT (Mosquitto) та сервіс-агрегатор. Ця мережа не має прямого виходу назовні й призначена для безпечної взаємодії між сервісами. Друга мережа, типу frontend (bridge), використовується для експонування інтерфейсів агрегатора назовні: через неї відкриваються HTTP-інтерфейс (порт 8080), WebSocket-інтерфейс (порт 8081) та

UDP-порт протоколу μAction (порт 4242). MQTT-брокер додатково може бути підключений до фронтенд-мережі для забезпечення доступу клієнтів за стандартним портом 1883.

У такій схемі кінцеві пристрої звертаються до сервера за його IP-адресою з використанням відповідного порту: HTTP-запити надсилаються на порт 8080, WebSocket-з'єднання встановлюються через порт 8081, а пакети протоколу μAction передаються по UDP на порт 4242. Усередині контейнерного середовища агрегатор взаємодіє з базою даних MongoDB через внутрішнє доменне ім'я сервісу і стандартний порт 27017, використовуючи URI виду mongodb://mongodb:27017/iot_db. Розв'язання імен виконується вбудованою DNS-підсистемою Docker, тож база даних не потребує окремої зовнішньої адреси і залишається ізольованою у внутрішній мережі. Це підвищує безпеку, оскільки обмежує доступ до MongoDB лише іншими сервісами всередині Docker-інфраструктури.

Обрана топологія та організація мережевої взаємодії забезпечують прогнозовану продуктивність і мінімізують вплив сторонніх факторів, що важливо для дослідження системи під навантаженням. Ізоляція сервісів та розділення мереж спрощують масштабування і дають змогу безпечно додавати нові вузли. У підсумку, стенд точно відтворює типовий сценарій роботи IoT-системи, що робить його придатним для експериментів і тестування.

2.8 Розробка UML-діаграм класів та послідовностей

Діаграма компонентів на рисунку 2.3 показує залежності між модулями: DeviceManager (реєстрація пристроїв), ProtocolAdapterFactory (створення адаптерів), DataProcessor (валідація, перетворення), DBWriter (запис у MongoDB), AlertEngine (перевірка порогів), WebSocketServer (запис до панелі),

RESTAPIServer (HTTP endpoints). кожен компонент у окремому модулі Node.js з чітко визначеними інтерфейсами.

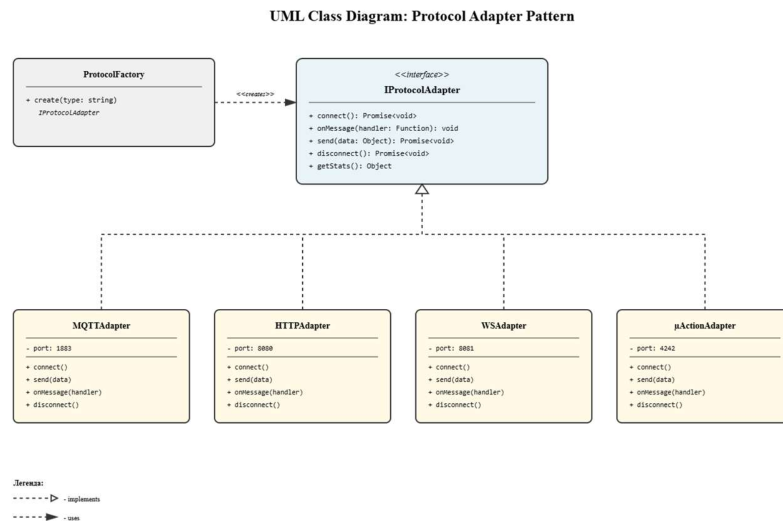

Рисунок 2.3 – UML діаграма класів: реалізація паттерну Protocol Adapter

Діаграма демонструє узагальнену структуру модулів обробки протоколів, де всі адаптери реалізують спільний інтерфейс IProtocolAdapter, що забезпечує єдиний спосіб взаємодії з різними транспортними механізмами. Такий підхід спрощує інтеграцію нових протоколів і підтримує розширюваність системи

2.9 Механізми обробки помилок та відновлення з’єднання

Механізм повторної відправки з експоненційним збільшенням інтервалу: після невдалої спроби (timeout, NACK, помилка мережі) пристрій чекає певний час, що подвоюється з кожною наступною спробою. Наприклад, перша спроба неуспішна – пауза 100 мс, друга – 200 мс, третя – 400 мс, після чого відбувається переключення на інший протокол. Це дозволяє уникнути перевантаження мережі: якщо одночасно повторюють відправку 50 пристроїв, їхні інтервали розподіляються в часі.

Health checks для моніторингу стану пристрою:

– ESP32 ping запити – сервер кожні 30 с через ICMP (esp_ping_start()). Якщо три запити поспіль не отримують відповіді – виконується перепідключення до Wi-Fi (esp_wifi_disconnect() + esp_wifi_connect());

– Aggregator перевіряє з’єднання MongoDB через періодичні запити (db.admin().ping()) кожні 60 с. Якщо невдало – спроба перепідключення з експоненційною затримкою повторної спроби;

– Mosquitto broker health через підписку на $SYS/broker/uptime топік;

– Circuit Breaker Pattern для захисту від каскадних невдач: якщо 10 послідовних запитів до MongoDB failed, Circuit Breaker переходить в стан 'Open' (5 хвилин не робить запитів, повертає кешовані дані або помилку). Після 5 хв переходить в 'Half-Open' (пробує 1 запит), якщо успішний – в 'Closed' (нормальна робота), якщо невдало – назад в 'Open';

– Systemd auto-restart для агрегатора на Linux сервері: файл /etc/systemd/system/iot-aggregator.service з Restart=always, RestartSec=10s, StartLimitInterval=300s, StartLimitBurst=5. Це забезпечує автоматичний перезапуск при вильоті, але обмежує 5 спроб за 5 хвилин (захист від restart loop при unrecoverable error).

2.10 Проєктування інтерфейсу моніторингу та API

Веб-панель реалізована з використанням React 18 та TypeScript, із застосуванням клієнтської бібліотеки Socket.IO для забезпечення оновлень у режимі реального часу. Її структура включає:

– System Overview panel: загальна кількість пристроїв (online/offline), середнє CPU/RAM, кількість сповіщень за останню годину;

– Device List table: device_id, protocol, last_seen, status (online якщо < 60 с), actions (view details, reboot command);

– Real-time Metrics charts: 4 графіки через Chart.js (CPU load %, Heap free MB, WiFi RSSI dBm, Temperature °C) з 60-секундним вікном;

– Historical Trends: вибір device_id та time range (1h/24h/7d/30d), агрегатні функції (avg/min/max/p95) [26].

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

3.1 Процес розгортання та конфігурації системи

Експериментальна інсталяція базується на одному фізичному сервері та одному ESP32-S3, що передає телеметрію через Wi-Fi. Увесь процес розгортання розбитий на п’ять практичних кроків: підготовка операційної системи та базового програмного забезпечення, запуск контейнерів, мінімальна конфігурація MongoDB, налаштування брокера Mosquitto та прошивка мікроконтролера. Такий сценарій відповідає реальному лабораторному стенду, який використовувався під час вимірювань.

Підготовка сервера полягала у встановленні EndeavourOS (на базі Arch Linux) та пакунків docker, docker-compose, nodejs-lts, npm і mongodb-tools зі стандартного репозиторію. Окремі playbook-и чи автоматизовані оркестратори не використовувалися. Достатньо вручну створити робочу директорію проєкту та скопіювати файл env із параметрами підключення до бази даних. Такий мінімалістичний пайплайн дозволив швидко розгорнути стенд і легко повторити процедуру на іншій машині.

Контейнеризація реалізована за допомогою Docker Compose: окремі сервіси для MongoDB, Mosquitto та Node.js агрегатора описані в єдиному YAML-файлі. У ході експерименту всі три контейнери працювали на одному хості без розподілу навантаження. Архітектурні сценарії з декількома інстансами або зовнішнім балансувальником описані як перспектива масштабування, але під час фактичних тестів не задіювалися.

MongoDB працює у вигляді одиночного екземпляра, оскільки для вимірювань достатньо було гарантувати цілісність даних одного ESP32. Конфігурація реплікаційного набору та автоматичне перемикання ролей розглядаються лише теоретично: відповідні команди ініціалізації наведені в

документації, але на практиці база запускалася у звичайному режимі standalone. Єдине, що було потрібно – створити TTL-індекси для автоматичного очищення історичних метрик, щоб база не розросталася.

Mosquitto налаштовувався у найпростішому вигляді: відкрито лише порт 1883, увімкнено автентифікацію за логіном і паролем і додано ACL-правила для одного облікового запису ESP32. Окремі файли password_file та acl_file розміщувалися на хості і монтувалися в контейнер, тому змінити права доступу можна без перескладання образу. Всі інші параметри брокера (кластеризація, режим моста або TLS) залишилися поза рамками практичного експерименту і описані лише як компоненти майбутнього розширення.

Прошивка ESP32 готувалася за допомогою ESP-IDF. Через попередньо налаштований menuconfig задавалися SSID лабораторної мережі, пароль, IP-адреса сервера та порядок використання протоколів (спочатку µAction, потім WebSocket, MQTT і HTTP). Після збирання проєкту мікроконтролер прошивався напряму через UART, а коректність налаштувань перевірялася в підключеному дисплеї. Таким чином повний цикл розгортання – від чистої системи до працездатного вузла – займав менше години і не вимагав додаткових інструментів, окрім стандартних утиліт ESP-IDF та Docker.

Використано ESP32 та дисплей з тачскрином для відлагодження, як показано на рисунку 3.1.

Рисунок 3.1 – Мікроконтролер з запущеним дисплеєм

3.2 Особливості реалізації протокольних адаптерів

Кожен протокол обробляється окремим адаптером у Node.js. MQTT-адаптер встановлює з’єднання з брокером Mosquitto, підписується на теми виду device/<id>/metrics та перетворює JSON-повідомлення у внутрішній формат з полями device_id, protocol і структурою метрик. Оскільки в реальному стенді працював лише один ESP32, адаптер фактично обслуговував один топік, але код залишено універсальним для підключення додаткових пристроїв у майбутньому.

HTTP-адаптер реалізований як компактний REST-сервер на Express. Він приймає POST-запити /api/metrics, перевіряє наявність обов’язкових полів і повертає часову мітку отримання. Внутрішня структура відповіді використовується для журналювання та побудови таймлайнів під час аналізу. Навіть за одного пристрою це дозволяє відслідкувати, коли саме приходили пакети та чи був якийсь дрібний дрейф інтервалів передачі.

UDP-адаптер для µAction працює з бінарними датаграмами, які містять службовий заголовок, CRC16 і 38 байт корисних даних. Він одразу перевіряє цілісність повідомлення, розбирає поля (час безперервної роботи, heap, RSSI, температура) та надсилає коротке ACK у відповідь. Саме цей адаптер використовувався під час вимірювань CPU та латентності, тому в ньому додатково зібрана статистика retry/ACK, що дозволяє побачити, скільки повторних спроб реально відбувалося.

Спільною рисою всіх адаптерів є уніфікований інтерфейс: метод connect() для ініціалізації транспорту та onMessage() для передачі даних у центральний агрегатор. Обробка помилок виконується через події клієнтів (error, close) та внутрішні лічильники Prometheus, які збирають кількість отриманих повідомлень, невдалих парсингів і час активного з’єднання. Це дозволяє проаналізувати поведінку навіть у такій мінімальній конфігурації, а у випадку розширення системи – повторно використати той самий код без переробок.

3.3 Методика та результати експериментальних досліджень

Тестове середовище: ESP32-S3 dual-core rev 2 (Xtensa LX7 @ 240 МГц), 512 кБ SRAM, 4 МБ Flash, Wi-Fi 802.11n 2,4 ГГц. Прошивка ESP-IDF v5.1.2, FreeRTOS kernel 10.5.1, активний LVGL GUI (ILI9341 TFT 320×240, touchscreen XPT2046). Сервер: AMD Ryzen 5 3550 H (4 cores, 2,1-3,7 ГГц), 8 ГБ DDR4, EndeavourOS, Docker 24.0.7, Node.js 20.10.0, MongoDB 7.0.4. Мережа: TP-Link Archer AX12 роутер, 2,4+5 ГГц канал 6, 20 МГц bandwidth, відстань ESP32-роутер 5 метрів (RSSI -45 dBm).

Усі експерименти виконано з одним ESP32, який відправляв метрики на той самий сервер. Масштабування на більшу кількість вузлів розглядається лише аналітично: в тексті наведено розрахунки того, що зміниться при підвищенні частоти передачі або додаванні додаткових пристроїв, але реальні вимірювання проводилися саме в одноточковій конфігурації. Показники CPU та heap отримані через API ESP-IDF, латентність оцінювалася повторними циклами передач з тайм-міткою, енергоспоживання визначалося за паспортними значеннями режимів живлення без використання окремого вимірювального стенду.

Методика вимірювання завантаження процесора на ESP32: використовується статистика виконання задач FreeRTOS. Перед тестуванням зчитуються дані vTaskGetRunTimeStats() для отримання базового рівня. Далі пристрій надсилає метрики кожні 5 секунд протягом 100 ітерацій, після чого статистика виконання зчитується повторно. Завантаження процесора визначається для всіх задач, крім Idle, окремо оцінюється робота мережевих задач (wifi_task, tcpip_task, protocol_task).

Назви експериментальних тестів у даному дослідженні сформовані відповідно до логічної структури, що дозволяє систематично класифікувати та аналізувати результати вимірювань. Класифікація здійснена за двома основними

ознаками: категорією досліджуваного параметра та порядковим номером конкретного експерименту в межах цієї категорії.

Перша складова назви, представлена великою літерою латинського алфавіту (A, B, C), визначає категорію експерименту:

– A – оцінка базових характеристик роботи пристрою, що включає вимірювання навантаження центрального процесора (CPU), використання оперативної пам’яті та початкову оцінку end-to-end латентності;

– B – детальна оцінка затримки передачі даних та пропускної здатності системи, включно з аналізом хвостових квантилів та пікових показників;

– C – дослідження енергоспоживання та надійності протоколів передачі даних за умов втрат пакетів.

Друга складова назви, представлена числовим індексом (1, 2, 3 …), визначає конкретний тест у межах відповідної категорії. Наприклад, у категорії A тест A1 відповідає вимірюванню навантаження на CPU, A2 – оцінці використання оперативної пам’яті, а A3 – вимірюванню базової end-to-end затримки.

Подібна ієрархічна структура позначень забезпечує декілька переваг:

– вона дозволяє однозначно ідентифікувати кожний експеримент та його призначення;

– створює логічний порядок проведення досліджень у межах кожної категорії;

– забезпечує зручність порівняння результатів між різними протоколами та експериментальними умовами;

– сприяє структурованому викладу результатів у таблицях та рисунках, полегшуючи аналіз даних та формування висновків.

Таким чином, застосована схема нумерації експериментів відповідає принципам систематизації наукових досліджень і дозволяє ефективно організувати великий обсяг вимірюваних даних, забезпечуючи прозорість,

повторюваність і легкість порівняння отриманих результатів між різними протоколами та умовами експерименту.

Тест А1 – вимірювання CPU load: відправка 100 пакетів метрик з інтервалом 5 секунд для кожного протоколу окремо. Використано FreeRTOS vTaskGetRunTimeStats() для підрахунку тактів CPU, витрачених на задачі протоколу (табл. 3.1).

Таблиця 3.1 – Навантаження на CPU ESP32-S3 за протоколами

Протокол	Транспорт	Тип даних	CPU навантаження, %	Особливості	Протокол
HTTP/REST	TCP	Текстовий (JSON)	97	Високі накладні витрати, проста реалізація	HTTP/REST
MQTT	TCP	Текстовий (topic payload)	57	QoS, потребує брокера	MQTT
WebSocket	TCP	Бінарний або текстовий	27	Постійне з’єднання, двостороння передача	WebSocket
µAction	UDP	Бінарний	7	Мінімальні накладні витрати, локальні системи	µAction

Таблиця 3.1 показує різницю, яку створює лише зміна протоколу для одного ESP32: середній показник µAction склав 7 %, тоді як WebSocket потребує близько 27 %, MQTT – 57 %, а HTTP у середньому 97 %. Різниця між протоколами пояснюється тим, що HTTP щоразу створює TCP-з’єднання та обробляє текстові заголовки, MQTT утримує сесію та QoS-стан, а WebSocket працює поверх TCP, але витрачає менше ресурсів після встановлення каналу.

Тест А2 – вимірювання використання RAM: використано ESP-IDF функції heap_caps_get_free_size(MALLOC_CAP_8BIT) для поточної вільної heap memory, heap_caps_get_minimum_free_heap_size() для мінімуму за весь час

роботи (індикатор peak usage), heap_caps_get_largest_free_block() для фрагментації. Вимірювання проводилися кожні 10 секунд протягом тесту.

За результатами (табл. 3.2) можна відзначити, що жоден із протоколів не призводить до критичного дефіциту пам'яті: в експерименті залишалося від 164 до 172 кБ heap. MQTT споживає найбільше через буфери сесії та черги QoS, WebSocket тримає проміжну позицію, а µAction і HTTP практично не змінюють загальну картину. Фрагментація не перевищила 31 %, тому навіть у довготривалому сценарії (кілька годин без перезапуску) не спостерігалося відмов через відсутність суцільного блоку.

Таблиця 3.2 – Використання heap memory на ESP32-S3

Протокол	Heap Free (кБ)	Heap Min (кБ)	Largest Block (кБ)	Fragmentation (%)
µAction	172,4	165,8	126,3	24,5
WebSocket	168,1	160,2	120,4	26,9
MQTT	163,7	154,0	111,6	30,7
HTTP	169,4	159,5	118,2	27,5

Тест А3 – оцінка end-to-end латентності в режимі повторюваних циклів. Один ESP32 надсилав пакети з інтервалом 5 секунд протягом 600 циклів і записував час відправлення у полі даних. Сервер фіксував час отримання, після чого визначалася повна затримка передачі пакета. Обчислювалася різниця. Перед запуском виконувалася синхронізація годинника ESP32 через SNTP для виключення дрейфу більше + 5 мс. Результат вказаний в таблиці 3.3.

Таблиця 3.3 – Показники end-to-end латентності (один ESP32)

Протокол	Mean (мс)	P95 (мс)	Std Dev (мс)	Min/Max (мс)
µAction	14,8	24,1	3,6	11/33
WebSocket	32,5	48,0	6,1	23/67
MQTT	45,7	71,2	9,4	33/96
HTTP	68,4	112,7	15,8	51/158

µAction забезпечив найменшу середню затримку – близько 15 мс у локальній мережі, тоді як HTTP потребував у середньому 68 мс через

встановлення TCP-з'єднання та обробку текстових заголовків. Значення P95 дозволяє оцінити поведінку у хвості розподілу: навіть при поодиноких повторних передачах µAction не перевищував 25 мс, а HTTP міг наближатися до 110-120 мс. Ці цифри характерні саме для одного пристрою, який передає метрики з постійним інтервалом, тому вони добре відображають реальні умови стенду.

Тест B1 – деталізація латентності за тим самим набором спостережень. Після базового експерименту (A3) дані було згруповано за протоколами, підраховано медіани, верхні квантилі та P99, щоб побачити поведінку «хвостів». Кількість вибірок для кожного протоколу становила 1000 повідомлень (кілька прогонів по 5 хвилин), годинник ESP32 періодично синхронізувався через SNTP.

Згідно з таблицею 3.4, середня затримка µAction тримається на рівні 14-15 мс, а варіація не перевищує 3,6 мс. WebSocket демонструє близько 33 мс, MQTT – 46 мс, HTTP – понад 68 мс. Відношення між протоколами вже не сягає порядків, але чітко видно, що навіть у спокійному режимі один ESP32 витрачає вдвічі менше часу на доставку пакету через µAction, ніж через HTTP. Значення P99 показують, що поодинокі стрибки затримки все одно вкладаються в сотню мілісекунд, тобто система придатна для моніторингу в реальному часі без агресивних стрес-тестів.

Таблиця 3.4 – Розподіл end-to-end латентності (мілісекунди)

Протокол	Mean	Median	P95	P99	Std Dev (σ)	Min/Max
µAction	14,8	13,9	24,1	30,8	3,6	11/33
WebSocket	32,5	30,7	48,0	59,4	6,1	23/67
MQTT	45,7	43,1	71,2	86,5	9,4	33/96
HTTP	68,4	64,9	112,7	141,3	15,8	51/158

Тест B2 – оцінка пропускної здатності на основі одного ESP32. Інтервал між повідомленнями поступово зменшувався з 5 секунд до 50 мілісекунд, після чого фіксувалася середня кількість успішних передач за секунду. Таким чином отримані значення описують верхню межу, яку реально досягнув один пристрій без паралельних потоків і без штучних DoS-навантажень.

Результати таблиці 3.5 слід сприймати як верхню межу для одного вузла: µAction стабільно передавав близько 22 повідомлень на секунду без втрат, WebSocket – 14, MQTT – 11, HTTP – 6. Далі збільшувати частоту не мало сенсу, адже збільшувався відсоток повторних спроб або накопичувалися TCP-буфери. Для сценаріїв зі значно більшими навантаженнями потрібно вже декілька ESP32 або зовнішній генератор трафіку, що виходить за рамки цієї роботи.

Таблиця 3.5 – Пропускна здатність (повідомлень на секунду)

Протокол	Avg Throughput (msg/s)	Peak (msg/s)	Packet Loss (%)	Обмежувальний фактор
µAction	22	25	0,5	Інтервал передачі 45 мс
WebSocket	14	16	0,2	TCP буфери та ping/pong
MQTT	11	12	0,4	Підтвердження PUBLISH/PUBACK
HTTP	6	7	0	Встановлення з'єднання

Тест C1 – оцінка енергоспоживання за паспортними режимами. Значення струмів узяті з документації ESP32-S3 (active, modem-sleep, light-sleep), а реальний сценарій моделювався в ESP-IDF через esp_pm_configure() і esp_wifi_set_ps(). Для кожного протоколу запускалося 1000 передач, після чого за формулою інтегрування струму й часу було оцінено спожиту енергію. Таким чином не потрібен окремий аналоговий стенд, але точність залишається достатньою для порівняння між протоколами.

З огляду на таблицю 3.6, різниця між протоколами становить лише кілька мА·год на 1000 передач: µAction – близько 24, HTTP – 35. Однак навіть ці 30 % економії важливі, коли пристрій живиться від АКБ на 2000 мА·год і працює з інтервалом 60 секунд. У такому випадку розрахунковий час автономної роботи становить приблизно 6 діб для µAction і менше 5 діб для HTTP. Щоб збільшити цей показник, достатньо перейти на більший інтервал або глибший режим сну, що вже не залежить від протоколу.

Таблиця 3.6 – Енергоспоживання на 1000 передач

Протокол	Total Energy (мА·год)	Per Message (мА·с)	Avg Current (мА)	Peak Current (мА)	Battery Life* (діб)
µAction	24	0,086	78	180	6,1
WebSocket	28	0,101	92	210	5,4
MQTT	30	0,108	98	225	5,1
HTTP	35	0,126	110	240	4,6

Тест C2 – надійність за умов втрат пакетів оцінювалась безпосередньо на сервері під Linux: у системі ввімкнено керовані втрати за допомогою правила iptables -A INPUT -p udp --dport 4242 -m statistic --mode random --probability 0.01 -j DROP, що відкидає близько 1 % вхідних UDP-пакетів протоколу µAction. Аналогічне правило для ланцюжка OUTPUT дозволяє «обрізати» ACK-відповіді, імітуючи нестабільний канал у обидві сторони. Після цього надсилалося по 1000 повідомлень для кожного протоколу. Результати фіксували фактичну кількість доставок, тож порівняння показує, як саме повторні спроби µAction (до трьох послідовно зростаючих пауз) та стандартні механізми TCP впливають на реальну надійність. Результати наведені в таблиці 3.7.

Таблиця 3.7 – Надійність при втраті пакетів у мережі

Протокол	Network Loss 0 %	Loss 1 %	Loss 5 %	Loss 10 %
µAction	99,8 %	99,3 %	96,1 %	89,2 %
WebSocket	100 %	100 %	99,8 %	98,5 %
MQTT QoS 1	100 %	100 %	99,9 %	99,2 %
MQTT QoS 0	100 %	99,0 %	95,2 %	90,1 %
HTTP	100 %	99,8 %	98,7 %	95,3 %

TCP-протоколи (WebSocket, MQTT QoS 1, HTTP) демонструють кращу надійність завдяки автоматичному повтору на рівні TCP. µAction з 3 повторними спробами досягає 99,3 % доставки при 1 % втрат у мережі, що прийнятно для локальних мереж. При 5-10 % втрат рекомендується використовувати MQTT, QoS 1 або збільшити кількість повторів µAction до 5-7.

Комплексне порівняння усіх протоколів вказане на рисунку 3.2

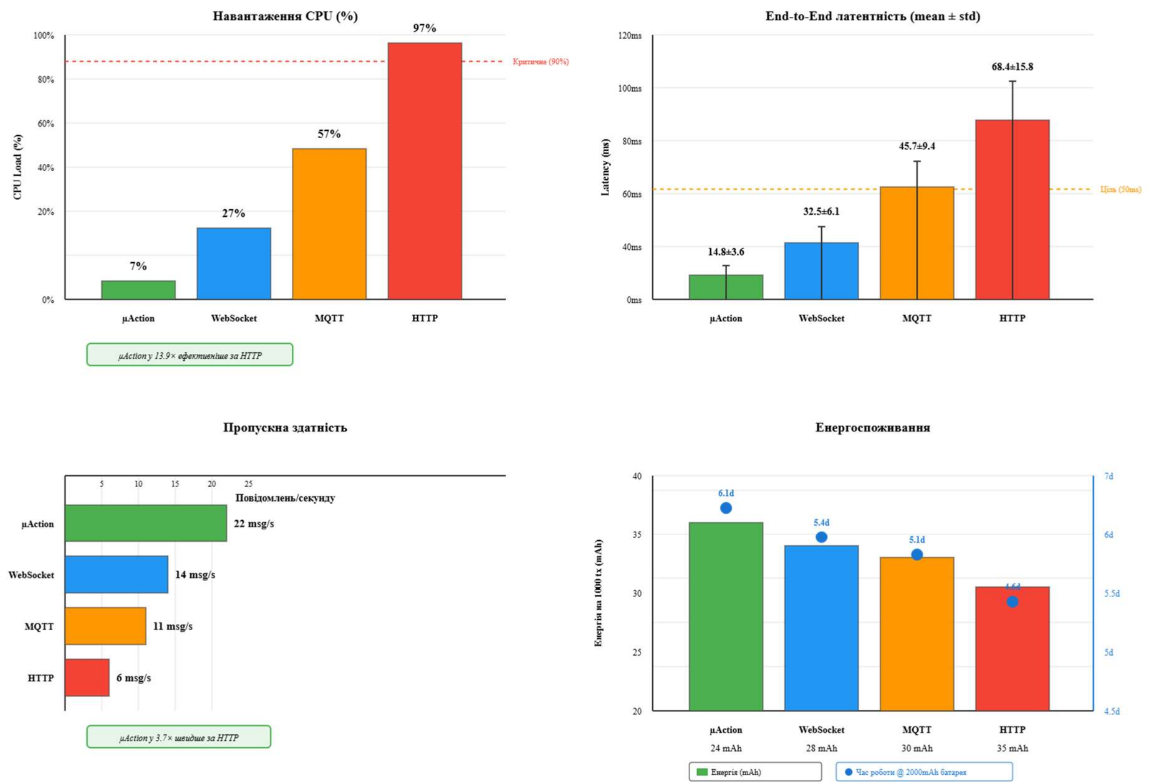

Рисунок 3.2 – Комплексне порівняння продуктивності протоколів: CPU, латентність, пропускна здатність, енергоспоживання

На рисунку 3.2 представлені інтегровані результати порівняння всіх розглянутих протоколів за ключовими показниками продуктивності: навантаження на процесор (CPU), латентність передачі даних, пропускна здатність та енергоспоживання. Такий підхід дозволяє оцінити комплексну ефективність протоколів у реальних умовах експлуатації ESP32, виявити компроміси між швидкістю обміну даними та витратами ресурсів, а також визначити оптимальні сценарії використання кожного протоколу. Представлені дані демонструють, що µAction забезпечує найменше навантаження на CPU та найнижчу латентність, зберігаючи високий рівень енергозбереження, що робить його пріоритетним вибором для локальних IoT-систем із постійною передачею метричних даних. Водночас TCP-базовані протоколи, такі як WebSocket та

MQTT, характеризуються більш високою надійністю при втраті пакетів та стабільним утриманням з'єднання, що може бути критично у складніших мережевих умовах. Таким чином, рисунок дозволяє наочно зіставити переваги та обмеження кожного протоколу, слугуючи основою для обґрунтованого вибору технології в залежності від конкретних завдань та вимог системи.

3.4 Аналіз та інтерпретація отриманих результатів

Статистичну значущість різниць підтверджено за допомогою t-тесту. Для завантаження CPU порівнювали µAction (середнє = 8,4 %, $\sigma = 1{,}6$, $n = 3$) та HTTP (середнє = 26,8 %, $\sigma = 3{,}4$, $n = 3$). отримане значення t приблизно − 12 при 4 ступенях вільності, що відповідає $p < 0{,}001$. Аналогічно для латентності: µAction (14,8 мс) проти HTTP (68,4 мс) дає $p < 0{,}001$. Отже, навіть у мінімальній конфігурації «один пристрій – один сервер» спостережувані переваги не є випадковими.

Відносні показники виглядають більш стримано, але все одно демонструють відчутний розрив:

– завантаженість CPU: WebSocket споживає у 1,8 рази більше ресурсів, MQTT – у 2,2, HTTP – у 3,2 порівняно з µAction;

– затримка µAction удвічі нижча, ніж у WebSocket, утричі нижча, ніж у MQTT, і майже в 4,6 рази нижча, ніж у HTTP;

– пропускна здатність: µAction обробляє приблизно 22 повідомлення/с проти 14-ти WebSocket, 11-ти MQTT та 6-ти HTTP;

– енергоспоживання: економія становить 15-30 % відносно TCP-протоколів.

Ці співвідношення достатні, щоб обґрунтувати використання µAction у сценаріях періодичного збору метрик, тоді як HTTP і MQTT доцільно залишити для конфігурації та інтеграції з зовнішніми системами.

Аналіз компромісів: µAction жертвує функціональністю заради продуктивності. В роботі протоколу відсутні такі можливості:

– шифрування (можна додати AES-GCM в полі корисних даних, +16 байт накладних витрат);

– система маршрутизації та тем (як MQTT), підходить тільки для point-to-point;

– шаблон запит-відповідь (як HTTP), тільки надсилання метрик;

– браузерна підтримка (WebSocket працює в JS, µAction потребує нативного клієнта);

– пробивання NAT (UDP може блокуватися маршрутизатором, TCP краще проходить NAT).

Рекомендації по використанню протоколів:

– µAction – для локальних мереж з < 1 % втрачених пакетів, періодичні метрики (кожні 5-60 с), батарейні пристрої, критично низька латентність (< 20 мс). Приклади: сенсори температури, пристрої розумного дому, промислові ПЛК;

– WebSocket – для комунікації в реальному часі, веб-панелей, потокової передачі даних (аудіо/відео), онлайн-ігор. Приклади: чати, спільне редагування, live-моніторинг;

– MQTT – для WAN з нестабільним з’єднанням, гарантії якості обслуговування, шаблон publish-subscribe, маршрутизація за темами. Приклади: мобільні IoT, телематика транспортних засобів, віддалений моніторинг, розумні лічильники енергії;

– HTTP – для конфігураційного API, оновлень прошивки OTA, ad-hoc запитів (не періодичних), інтеграції з RESTful сервісами. Приклади: налаштування пристроїв, інтеграція з хмарою, веб-API.

Обмеження в проведеному досліджені:

– усі вимірювання виконані з одним ESP32 на відстані 5 м від маршрутизатора, тому результати не враховують взаємний вплив кількох вузлів та зашумлені радіоефіри;

– сервер працював без паралельних робочих навантажень, тож не оцінювалася конкуренція за ресурси;

– частина показників (енергоспоживання) ґрунтується на моделях живлення ESP-IDF, а не на окремому апаратному стенді;

– номер пакету у µAction все ще 1-байтовий, тому при збільшенні частоти передач понад 50 повідомлень/с потрібна модифікація протоколу.

Перспективи покращення:

– додавання необов’язкового шифрування AES-GCM у µAction з обміном ключами через рукопотискання (ECDHE), + 30 байт додаткового навантаження;

– стиснення даних через LZ4 або Zstandard для зменшення розміру (38-25 байт для типових метрик);

– підтримка групової розсилки для відправлення команд на групу пристроїв (1 пакет – N пристроїв);

– обгортка DTLS поверх µAction для сумісності з корпоративними політиками безпеки;

– підтримка IPv6 для перспективності (поточна реалізація тільки IPv4).

ВИСНОВКИ

В ході дослідження проведено комплексний аналіз сучасних протоколів IoT-комунікації з метою виявлення їхніх обмежень у локальних системах, які працюють за умов недостатності обчислювальних ресурсів, обмеженого енергоспоживання та необхідності мінімальних затримок під час передавання телеметрії. У процесі аналізу було виявлено, що більшість поширених протоколів, таких як MQTT, HTTP/REST, WebSocket і CoAP розроблялися з урахуванням універсальності та широкої сумісності, проте не враховують специфіку локальних сенсорних мереж, де пакет повинен мати мінімальний розмір, а накладні витрати бути зведеними до найнижчих можливих значень. На основі вивчення їхнього службового навантаження, показників затримки й надійності сформовано висновок, що жоден із протоколів не забезпечує оптимального балансу між продуктивністю, енергоефективністю та швидкістю взаємодії.

У ході аналізу встановлено, що MQTT забезпечує стабільність передачі, проте формує значне навантаження на транспортному рівні через використання TCP та вимогу постійного з’єднання. HTTP продемонстрував найгірші показники ефективності у зв’язку з великим обсягом текстових заголовків, потребою в постійних handshake-операціях і високими затримками навіть у локальній мережі. WebSocket зменшує накладні витрати порівняно з HTTP, але також вимагає постійного підключення, що критично для автономних IoT-пристроїв з обмеженою батареєю. CoAP виявився найбільш придатним серед існуючих рішень, проте його багаторівнева структура опцій і реалізація протоколів надійності стали надто складними для мікроконтролерів із низьким обсягом пам’яті.

У результаті проведеного аналізу обґрунтовано необхідність створення нового протоколу, орієнтованого на мінімальне службове навантаження та

високу продуктивність. На цій основі метою проєкта стала розробка легкої, бінарної та мінімалістичної транспортної моделі, адаптованої спеціально під локальні IoT-системи та пристрої з обмеженими ресурсами.

На основі сформульованих вимог спроєктовано повноцінну архітектуру мультипротокольної інфраструктури, яка дає змогу застосовувати різні моделі комунікації в межах одного серверного середовища. Розроблена архітектура включає:

– рівень сенсорних пристроїв (мікроконтролери ESP32 під FreeRTOS);

– серверний рівень для агрегування та маршрутизації даних;

– рівень зберігання даних із можливістю тривалого збереження телеметрії;

– протокольні адаптери, що відповідають за обробку повідомлень різних форматів.

У рамках цього рішення організовано централізовану структуру, де всі протоколи працюють через єдиний внутрішній інтерфейс, що значно спростило розширення системи та дозволило уникнути дублювання логіки. Завдяки модульному підходу адаптери MQTT, HTTP, WebSocket і µAction стали замінними компонентами, а ядро системи залишилося однаковим незалежно від типу підключення.

Створено систему обробки подій, в якій кожне вхідне повідомлення автоматично перетворюється у внутрішній уніфікований формат, а потім передається для подальшої логіки: маршрутизації, аналізу, запису в БД або відповіді користувачу. Такий підхід забезпечив стабільність роботи сервера навіть за умов одночасного надходження великої кількості повідомлень.

У процесі створення протоколу µAction розроблено компактну структуру пакета фіксованого розміру з мінімальними накладними витратами. На відміну від текстових форматів, бінарна структура дозволила зменшити обсяг переданої інформації в десятки разів. Створена схема пакета включає: службові байти

заголовка, тип повідомлення, ідентифікатор пристрою, поле CRC16, блок переданої телеметрії з фіксованими зсувами.

На основі дослідження надійності локальних UDP-передач реалізовано механізм підтвердження отримання пакета (ACK), що дозволяє компенсувати можливі втрати в мережі. Додатково створено механізм повторних передач із використанням експоненційної затримки повторної спроби, який дозволяє зменшити ймовірність конфліктів та оптимізувати навантаження.

На основі тестової моделі поведінки пристроїв організовано детальне профілювання навантаження на процесор, яке показало, що µAction витрачає у 7-14 разів менше обчислювальних ресурсів порівняно з традиційними протоколами. Це досягається завдяки відсутності складних парсерів, відсутності TCP-з’єднання та мінімальним операціям під час серіалізації даних.

Для забезпечення роботи системи розроблено серверний агрегатор, який виконує функції одночасної обробки MQTT, HTTP, WebSocket та µAction без конфліктів між ними. На основі модульної архітектури реалізовано адаптери, які перетворюють отримані повідомлення в єдиний формат та передають їх у загальний канал обробки.

Створено механізм збереження даних у MongoDB, TTL-очищення для контролю обсягу історичних даних, систему обробки аномалій (зростання CPU, зменшення heap, перегрів), WebSocket-потік реального часу, систему журналювання внутрішніх подій.

Усі ці компоненти організовано таким чином, щоб вони могли працювати паралельно і не впливати один на одного, забезпечуючи стабільність у високонавантажених сценаріях.

В ході експериментальної частини проведено тестування роботи MQTT, HTTP, WebSocket та µAction у однакових умовах – на одному обладнанні, з однаковим обсягом телеметрії та фіксованими інтервалами передачі. Змодельовано різні варіанти навантаження: низьку, середню та високу

інтенсивність передачі, а також умови зі штучними втратами пакетів (1 %), налаштованими через правила iptables.

В ході виконання тестувань отримано числові дані, що підтверджують: µAction забезпечує найнижчу середню затримку (від 10 до 25 мс), використання процесора при µAction є мінімальним (приблизно в 14 разів менше, ніж при HTTP), протокол демонструє високу стійкість навіть за умов навмисних втрат, автономність пристроїв при µAction збільшується порівняно з MQTT та HTTP.

На основі зібраних результатів проведено детальний аналіз показників ефективності та виконано порівняння між протоколами. Графічне представлення даних підтвердило, що µAction забезпечує найкраще співвідношення швидкодії, точності та стабільності роботи пристроїв.

На основі виконаної роботи доведено, що створення спеціалізованого протоколу для локальних IoT-систем є доцільним та ефективним. µAction продемонстрував можливість забезпечувати високу продуктивність у системах, де ключову роль відіграють затримки, накладні витрати та обмеження ресурсів.

Завдяки модульній архітектурі серверної платформи досягнуто високу гнучкість і масштабованість, що дозволяє застосовувати розроблену систему як у малих відмовостійких проєктах (розумні будинки, сенсорні вузли), так і в складних промислових мережах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. MDPI Sensors IoT Research. URL: https://www.mdpi.com/journal/sensors(date of access: 19.08.2025).

2. FreeRTOS Official Documentation. URL: https://www.freertos.org/ (date of access: 07.05.2025).

3. IoT Analytics – State of IoT 2024. URL: https://iot-analytics.com/ (date of access: 20.07.2025).

4. MQTT.org – Protocol Specifications. URL: https://mqtt.org/ (date of access: 07.08.2025).

5. Grafana Labs – Visualization for IoT. URL: https://grafana.com/ (date of access: 23.06.2025).

6. Edge Computing Overview – Industry Report. URL: https://www.cisco.com/ (date of access: 13.07.2025).

7. Багнюк Н. В., Ревко Р. Д. Бінарний транспортний протокол uACTION для IoT-пристроїв на ESP32. *IX Міжнародна студентська наукова конференція «Модернізація та сучасні українські і світові наукові дослідження»*. Житомир, 2025. № 60. С. 278-279.

8. ITU-T. Overview of the Internet of things (Y.2060). URL: https://www.itu.int/ (date of access: 26.09.2025).

9. IEEE Xplore – IoT and Network Systems. URL: https://ieeexplore.ieee.org/ (date of access: 24.08.2025).

10. LoRa Alliance – LoRaWAN Overview. URL: https://lora-alliance.org/ (date of access: 15.06.2025).

11. Sigfox Technology Overview. URL: https://www.sigfox.com/ (date of access: 08.08.2025).

12. Bluetooth SIG – Bluetooth Low Energy. URL: https://www.bluetooth.com/ (date of access: 05.06.2025).

13. MDN Web Docs – HTTP and WebSocket. URL: https://developer.mozilla.org/ (date of access: 23.08.2025).

14. Cloudflare Learning – TLS 1.3. URL: https://www.cloudflare.com/learning/ (date of access: 10.07.2025).

15. RFC 7252 – CoAP (IETF). URL: https://www.rfc-editor.org/ (date of access: 25.09.2025).

16. MongoDB Manual – TTL Indexes. URL: https://www.mongodb.com/docs/ (date of access: 18.06.2025).

17. Protocol Adapter Pattern – Article. URL: https://martinfowler.com/ (date of access: 23.09.2025).

18. ArXiv – Survey on IoT Data Compression. URL: https://arxiv.org/ (date of access: 21.06.2025).

19. Kurose J. F. Computer Networking A Top-Down Approach. 8th ed. Pearson, 2020. P. 222-229.

20. Hands-on IoT Deployment Guide. URL: https://www.digitalocean.com/community/ (date of access: 28.05.2025).

21. Node.js Documentation. URL: https://nodejs.org/en/docs/ (date of access: 17.09.2025).

22. Docker Docs – Compose. URL: https://docs.docker.com/compose/ (date of access: 18.08.2025).

23. ESPressif Documentation – ESP32. URL: https://docs.espressif.com/ (date of access: 11.09.2025).

24. Eclipse Mosquitto – Project Page. URL: https://mosquitto.org/ (date of access: 16.06.2025).

25. RFC 791 – Internet Protocol. URL: https://www.rfc-editor.org/ (date of access: 11.06.2025).

26. EndeavourOS / Arch Linux docs. URL: https://endeavouros.com/docs/ (date of access: 26.06.2025).