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»

РОЗРОБКА ТА ТЕСТУВАННЯ СИСТЕМИ MQTT-КОМУНІКАЦІЇ НА RASPBERRY PI

DEVELOPMENT AND TESTING OF AN MQTT COMMUNICATION SYSTEM ON RASPBERRY PI

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

освітня програма Комп’ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІм-21
Цис Роман Іванович

____
(підпис)

Керівник:
к.т.н., доцент
Костючко Сергій Миколайович

____
(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

Циса Романа Івановича
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Розробка та тестування системи MQTT-комунікації на Raspberry Pi*

Керівник роботи *к.т.н., доцент Костючко Сергій Миколайович*

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 та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Теоретичні основи систем біометричної ідентифікації та контролю доступу
Розробка інтелектуальної системи доступу
Реалізація та дослідження роботи системи багаторівневої аутентифікації
Висновки

5. Перелік графічного (ілюстративного) матеріалу:

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналітичний огляд та дослідження літератури з MQTT-комунікації	*Костючко С.М., доцент*		
Вибір апаратних складових та аналіз існуючих технічних рішень	*Костючко С.М., доцент*		
Побудова, налаштування та тестування MQTT-системи на Raspberry PI	*Костючко С.М., 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.	*Аналітичний огляд та дослідження літератури з MQTT-комунікації*	До 20.08.2025 р.	
3.	*Вибір апаратних складових та аналіз існуючих технічних рішень*	До 25.09.2025 р.	
4.	*Побудова, налаштування та тестування MQTT-системи на Raspberry PI*	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 ____________ Цис Р.І.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи ____________ Костючко С.М
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Цис Р.І. Розробка та тестування системи MQTT-комунікації на Raspberry PI.

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

Кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків. Робота присвячена дослідженню можливостей застосування протоколу MQTT у системах Інтернету речей та практичній реалізації експериментального стенду на базі Raspberry Pi для організації надійного та захищеного обміну телеметричними даними і керувальними повідомленнями між клієнтськими вузлами.

У першому розділі виконано аналітичний огляд теоретичних засад функціонування протоколу MQTT у складі IoT-архітектур, здійснено порівняльний аналіз найбільш поширених брокерів MQTT, сценаріїв їх розгортання, використанню механізмів TLS-шифрування, ACL та автентифікації клієнтів.

У другому розділі здійснено У другому розділі обґрунтовано вибір апаратної платформи Raspberry Pi та клієнтських вузлів ESP32, сформовано програмно-апаратний стек рішення із застосуванням брокера Eclipse Mosquitto та інтеграційного середовища Node-RED.

У третьому розділі описано процес побудови апаратної частини MQTT-системи, розгортання та налаштування брокера Mosquitto на Raspberry Pi, реалізацію типових сценаріїв обміну даними (телеметрія, керування, retained-повідомлення, LWT). Наведено результати експериментальних досліджень і практичні рекомендації щодо безпечного та ефективного використання MQTT у локальних IoT-системах.

Ключові слова: Raspberry Pi, MQTT, брокер, IoT, publish/subscribe, Node-RED, Mosquitto, QoS, телеметрія, безпека.

ANNOTATION

Tsus R. Development and testing of an mqtt communication system on Raspberry PI

Qualifying work of a Master's of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2025.

The master's qualification work consists of an introduction, three sections, conclusions, a list of sources used and appendices. The work is devoted to the study of the possibilities of using the MQTT protocol in Internet of Things systems and the practical implementation of an experimental stand based on Raspberry Pi for organizing reliable and secure exchange of telemetry data and control messages between client nodes.

The first section provides an analytical review of the theoretical principles of the MQTT protocol functioning as part of IoT architectures, a comparative analysis of the most common MQTT brokers, their deployment scenarios, the use of TLS encryption, ACL and client authentication mechanisms.

The second section provides a justification for the choice of the Raspberry Pi hardware platform and ESP32 client nodes, and a hardware-software stack of the solution using the Eclipse Mosquitto broker and the Node-RED integration environment.

The third section describes the process of building the hardware part of the MQTT system, deploying and configuring the Mosquitto broker on Raspberry Pi, and implementing typical data exchange scenarios (telemetry, control, retained messages, LWT). The results of experimental research and practical recommendations for the safe and effective use of MQTT in local IoT systems are presented.

Keywords: Raspberry Pi, MQTT, broker, IoT, publish/subscribe, Node-RED, Mosquitto, QoS, telemetry, security.

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1 АНАЛІТИЧНИЙ ОГЛЯД ТА ДОСЛІДЖЕННЯ ЛІТЕРАТУРИ З MQTT-КОМУНІКАЦІЇ 10
1.1 Призначення та місце MQTT у IoT-системах 10
1.2 Порівняльний огляд брокерів MQTT і сценаріїв розгортання 12
1.3 Типові архітектури систем на базі Raspberry Pi та MQTT 15
1.4 Засоби інтеграції та візуалізації даних 16
1.5 Питання безпеки MQTT 18
РОЗДІЛ 2 ВИБІР АПАРАТНИХ СКЛАДОВИХ ТА АНАЛІЗ ІСНУЮЧИХ ТЕХНІЧНИХ РІШЕНЬ 20
2.1 Обґрунтування вибору платформи 20
2.2 Вибір та опис клієнтських вузлів MQTT 21
2.3 Програмно-апаратний стек рішення 23
2.4 Аналіз існуючих реалізацій 24
2.5 Вимоги до проєктованої системи 26
РОЗДІЛ 3 ПОБУДОВА, НАЛАШТУВАННЯ ТА ТЕСТУВАННЯ MQTT-СИСТЕМИ НА RASPBERRY PI 28
3.1 Проєктування та збирання апаратної частини 28
3.1.1 Підключення клієнтського вузла ESP32 (сенсори/керування) 30
3.2 Розгортання брокера Mosquitto на Raspberry Pi 32
3.3 Реалізація сценаріїв обміну даними 34
3.4 Методика та результати тестування 37
3.5 Перевірка механізмів безпеки та рекомендації 41
3.6 Рекомендації 44
ВИСНОВКИ 46
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 48
ДОДАТКИ 51

ВСТУП

Інтернет речей (IoT) дедалі частіше використовується в системах моніторингу, промислової автоматизації, «розумної» інфраструктури та кіберфізичних комплексах, де критичними є надійність обміну даними, передбачуваність затримок і масштабованість взаємодії між численними пристроями. У таких умовах протокол MQTT залишається одним із найпоширеніших механізмів передачі телеметрії та керувальних команд завдяки легковажності, підтримці рівнів QoS і можливості роботи в середовищах із нестабільним зв'язком. Водночас практичне розгортання MQTT у лабораторних і напівпромислових умовах потребує коректного вибору апаратної платформи (зокрема Raspberry Pi), налаштування брокера, інтеграції з інструментами візуалізації, а також обов'язкового врахування безпеки (TLS/mTLS, ACL, контроль доступу). Через це розробка відтворюваного стенду та методики тестування MQTT-комунікації є актуальною як для навчально-дослідницьких, так і для прикладних задач.

Метою цієї роботи є розробка та експериментальне тестування MQTT-комунікаційної системи на базі Raspberry Pi, яка забезпечує обмін телеметричними даними та керувальними командами між IoT-вузлами, підтримує інтеграцію й візуалізацію даних, а також передбачає практично перевірені механізми захисту (автентифікацію, авторизацію та шифрування).

Для досягнення цієї мети поставлено наступні завдання, що розглядаються в роботі:

– виконати аналітичний огляд MQTT у складі IoT-систем, брокерів і сценаріїв розгортання;

– проаналізувати типові архітектури систем на базі Raspberry Pi та MQTT і засоби інтеграції/візуалізації (зокрема через Node-RED);

– обґрунтувати вибір апаратної платформи та клієнтських вузлів MQTT і сформувати програмно-апаратний стек рішення;

– розгорнути MQTT-брокер Mosquitto на Raspberry Pi та налаштувати параметри доступу, ACL і TLS (за потреби mTLS);

– реалізувати та перевірити сценарії обміну даними (телеметрія, керування з підтвердженням, retained-стан, доступність вузла через LWT);

– визначити метрики й методику вимірювання (затримка, надійність, пропускна здатність, ресурсне навантаження брокера, вплив QoS);

– провести експериментальні серії для локального режиму та bridge-сценарію й сформувати практичні рекомендації щодо відтворюваного та безпечного розгортання.

Об’єктом дослідження є процеси обміну повідомленнями в IoT-мережах на основі протоколу MQTT при розгортанні брокера та клієнтів у локальному середовищі на платформі Raspberry Pi.

Предметом дослідження є методи, програмно-апаратні засоби й конфігураційні рішення для побудови, захисту та тестування MQTT-системи (Mosquitto, клієнтські вузли, простір топіків, QoS, retained/LWT, ACL, TLS/mTLS), а також метрики й методика експериментального оцінювання її надійності та продуктивності.

Наукова новизна роботи полягає в систематизованому підході до проєктування відтворюваного MQTT-стенду на Raspberry Pi з формалізованим простором топіків і наскрізними сценаріями A-D (телеметрія, керування з підтвердженням, retained-стан, LWT), а також у запропонованій методиці експериментальної оцінки якості MQTT-комунікації, яка базується на вимірюванні RTT для мінімізації похибок синхронізації часу, порівнянні режимів QoS і аналізі впливу сценаріїв розгортання (локальний/bridge) на затримки, надійність і пропускну здатність із паралельним контролем ресурсів брокера.

Апробація результатів. Результати роботи представлені у публікації наукового часопису «Технічні вісті», опублікованого 2025 р., м. Львів [1].

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД ТА ДОСЛІДЖЕННЯ ЛІТЕРАТУРИ З MQTT-КОМУНІКАЦІЇ

1.1 Призначення та місце MQTT у IoT-системах

MQTT (Message Queuing Telemetry Transport) – це стандартизований (OASIS) протокол обміну повідомленнями для IoT, що реалізує модель client-server із шаблоном взаємодії publish/subscribe (публікація/підписка). Його ключова ідея – максимально «легкий» транспорт повідомлень для обмежених пристроїв і мереж із низькою пропускною здатністю, високою затримкою або нестабільністю. Саме ці принципи (мінімізація мережевого трафіку та вимог до ресурсів пристрою при збереженні керованої надійності доставки) пояснюють, чому MQTT став де-факто базовим протоколом IoT-зв’язку [2].

Місце MQTT у типовій IoT-архітектурі – це «шина повідомлень» між джерелами даних (датчики/контролери/шлюзи) та споживачами (HMI/SCADA-панелі, сервіси зберігання, аналітика, мобільні застосунки). На відміну від REST-підходу «запит-відповідь», MQTT орієнтований на подієву модель. Пристрій публікує телеметрію у вигляді повідомлень у певні topics, а всі зацікавлені клієнти одержують ці дані через брокер. Такий підхід забезпечує розв’язання зв’язків між джерелом і отримувачем (джерело не «знає» конкретних адресатів), що є критично важливим для масштабування IoT.

З позиції стандарту, MQTT визначається як «lightweight… publish/subscribe messaging transport protocol… designed to be easy to implement», що робить його придатним для M2M/IoT-сценаріїв, де важливі малий кодовий «слід» і економія каналу. Протокол працює поверх TCP/IP (або інших транспортів, що забезпечують упорядковане, двобічне та безвтратне з’єднання). Також важливо, що MQTT є агностичним до вмісту корисного навантаження. Стандарт визначає доставку повідомлень і метадані, але не нав’язує формат даних (JSON/CBOR/бінарний тощо обираються на рівні застосунку) [3].

Однією з причин популярності MQTT в IoT є керована надійність доставки через рівні QoS (Quality of Service). Стандарт явно вводить три якості сервісу доставки повідомлень:

– QoS 0 «At most once» – доставка «як вийде», можливі втрати;

– QoS 1 «At least once» – гарантія доставки мінімум один раз (можливі дублікати);

– QoS 2 «Exactly once» – доставка рівно один раз, але з більшою протокольною накладною.

Цей механізм дозволяє інженерно збалансувати швидкість/трафік/надійність під конкретний тип даних (наприклад, телеметрія датчика може бути QoS0, а команди керування виконавчим механізмом – QoS1/2).

У практичних системах MQTT найчастіше використовується для:

– телеметрії (періодичні вимірювання, статуси, події);

– дистанційного керування (команди, конфігураційні параметри);

– інтеграції компонентів (шлюз локальний сервер візуалізація хмара).

Наприклад, у IEEE-конференційній роботі про «Low-Cost ESP32, Raspberry Pi, Node-RED, and MQTT… SCADA» MQTT використано як канал обміну між сенсорним шлюзом (ESP32) та локальним сервером/панеллю (Raspberry Pi та Node-RED), підкреслюючи його придатність для «unreliable networks and constrained devices and low bandwidth».

Окремо слід зафіксувати тенденцію масштабування. Сучасні IoT-платформи часто відходять від моделі «один брокер на все», застосовуючи розподілені брокери та bridging, бо один брокер стає «вузьким місцем». У TD-MQTT прямо зазначено, що розподіл брокерів широко використовується, оскільки стратегія одного брокера «no longer efficient». Pi може виступати локальним брокером/edge-вузлом або елементом розподіленої схеми з мостами [4].

1.2 Порівняльний огляд брокерів MQTT і сценаріїв розгортання

MQTT-брокер є центральним елементом publish/subscribe-архітектури – він приймає повідомлення від publisher-клієнтів і доставляє їх усім subscriber-клієнтам згідно з топіками та QoS. Вибір брокера в IoT-системі на Raspberry Pi зазвичай визначається вимогами до масштабованості (один вузол чи кластер), інтеграції (правила/конектори/плагіни), надійності (постійні сесії, persistence), протокольної сумісності (MQTT 3.1.1 / MQTT 5), безпеки (ACL, TLS, керування доступом) та експлуатації (моніторинг, оновлення, ліцензування). Ідея розподілених брокерів стає типовою, бо модель «один брокер» часто втрачає ефективність у масштабних або географічно рознесених IoT-сценаріях [5].

1.2.1 Коротка характеристика поширених брокерів (за документованими можливостями)

Нижче наведено узагальнення можливостей брокерів, які найчастіше використовують у навчальних/edge-проєктах (у т.ч. Raspberry Pi) та в індустріальних системах.

Eclipse Mosquitto (легкий брокер для edge/лабораторних стендів) – підтримує механізми автентифікації на рівні протоколу (username/password), SSL/TLS, ACL, а також Dynamic Security plugin для гнучкішого керування доступом. Ці можливості описані в офіційній конфігурації та документації.

EMQX (розподілений брокер/платформа) – кластеризація як спільна робота кількох вузлів із розподіленням сесій, підписок і маршрутизації, що забезпечує горизонтальне масштабування. Також у документації/матеріалах згадуються multi-tenancy (namespaced roles) і вбудований rule engine/інтеграції як типові «platform» можливості [6].

HiveMQ – для формування кластера потрібні ввімкнені механізми clustering, discovery, transport, а також доступні підходи розгортання під «real-world stress» і розширення через екосистему/SDK (CE + Extension SDK).

VerneMQ можна кластеризувати так, щоб клієнти підключалися до будь-якого вузла кластера й отримували повідомлення з інших вузлів; окремо

підкреслюється, що гарантії MQTT складно забезпечувати при мережевих розділеннях (network partitions).

RabbitMQ MQTT plugin – підтримка MQTT 3.1/3.1.1/5.0, а також інтероперабельність MQTT-клієнтів з AMQP/STOMP-оточенням (корисно, якщо RabbitMQ уже є «message bus» у системі).

NanoMQ (edge-брокер) – lightweight/high-performance MQTT broker для edge, з MQTT 5.0 сумісністю, bridges, rule engine та persistence (як задекларовано в «Key Features»).

1.2.2 Порівняльна таблиця (за заявленими/документованими функціями)

У таблиці 1.1 систематизовано саме документовані можливості, на які дають можливість обґрунтувати вибір брокера для Raspberry Pi та для масштабованого розгортання.

Таблиця 1.1 – Порівняння брокерів MQTT за ключовими властивостями (за офіційною документацією)

Брокер	Типове позиціонування	Кластер/масштабування	Безпека/контроль доступу	Інтеграції/розширення
Mosquitto	брокер для простих/edge-сценаріїв	(кластер не є «core» фокусом у доках; частіше застосовують bridge/edge-роль)	TLS, username/password, ACL, Dynamic Security plugin	конфігураційні механізми/плагіни (в т.ч. dynamic security)
EMQX	розподілений брокер/платформа	кластеризація з розподілом сесій/підписок/маршрутів	розширені моделі доступу (multi-tenancy/roles у доках)	rule engine, інтеграції/конектори (платформні можливості)
HiveMQ	platform broker	кластер (clustering + discovery + transport)	(залежить від редакції/налаштувань; описується в user guide)	Extension SDK/екосистема
VerneMQ	distributed broker	«easily clustered»; увага до network partitions	(налаштовується; деталі в доках/практиках)	розширюваність, enterprise support згадано на сайті
RabbitMQ (MQTT plugin)	брокер-шина з MQTT входом	масштабування через підхід RabbitMQ, MQTT як плагін/режим	підтримка MQTT 3.1/3.1.1/5.0; multi-tenancy згадується в доках	інтероперабельність MQTT↔AMQP/STOMP
NanoMQ	lightweight edge broker	edge-роль; bridges, edge-to-cloud	(описується в доках; залежить від конфігурації)	bridges + rule engine + persistence (Key Features)

1.2.3 Сценарії розгортання (deployment patterns) для IoT та Raspberry Pi

У практичних IoT-рішеннях сценарії розгортання MQTT-брокера зазвичай еволюціонують від локального «edge» до розподілених схем залежно від масштабу й вимог до затримок та доступності. Найпростіший і типовий для прототипів варіант – локальний брокер у межах однієї мережі, коли Raspberry Pi виконує роль локального сервера і хоста для брокера MQTT та засобів обробки/візуалізації. У роботі про низьковартісну SCADA-систему прямо зазначено, що Raspberry Pi2 Model B використано як local server у локальній мережі, який включає MQTT broker, Node-RED і SQLite, а обмін даними між вузлами реалізовано через MQTT та Node-RED [7].

Коли система виходить за межі однієї локальної мережі або з'являються вимоги до обміну даними між кількома середовищами (підрозділами/організаціями/майданчиками), часто застосовують міст (bridge) між брокерами або виділений компонент, що передає повідомлення між доменами. Саме такі «bridge architectures» аналізуються у статті Evaluation of MQTT Bridge Architectures in a Cross-Organizational Context, де автори підкреслюють, що на затримку та надійність мостових розгортань впливають не лише топологія/кількість bridge-компонентів, але й параметри навантаження, зокрема розмір даних і назва топіка [8].

На більшому масштабі поширюється підхід розподілення кількох MQTT-брокерів у мережі, оскільки стратегія «один брокер» стає неефективною. Водночас у таких розподілених архітектурах виникає практична проблема, підписник зазвичай має «знати наперед» адресу брокера, на якому публікуються потрібні топіки. У TD-MQTT це формулюється як ключовий недолік distributed MQTT і пропонується механізм «прозорого» підключення без попереднього знання брокера, а також автоматичне перенесення даних між брокерами при змінах конфігурації/локації.

1.3 Типові архітектури систем на базі Raspberry Pi та MQTT

У системах IoT Raspberry Pi зазвичай використовується як edge-вузол (близько до датчиків/контролерів) або як локальний сервер у межах об’єкта. Причина практична, Pi поєднує достатню обчислювальну потужність для брокера та інтеграційних сервісів із можливістю постійної роботи «24/7» у локальній мережі. У найпростішій конфігурації Raspberry Pi виступає MQTT-брокером, а кінцеві вузли (ПК, смартфони, ESP32/NodeMCU, сенсори) працюють як клієнти-паблішери/сабскрайбери. У навчальному гайді Hackster це прямо пояснюється через клієнт-серверну інтерпретацію. Брокер – це «сервер», а publisher/subscriber – «клієнти»; також наголошується, що при локальному розгортанні клієнти мають бути в тій самій мережі, а Mosquitto на Raspberry Pi встановлюється стандартно через apt і вмикається як сервіс systemctl enable mosquitto [9].

Найпоширеніша прикладна архітектура для Raspberry Pi у невеликих IoT/SCADA-рішеннях – це шлюз/сенсорний вузол → MQTT → локальний сервер на Raspberry Pi → візуалізація/зберігання. У статті Low-Cost ESP32, Raspberry Pi, Node-Red, and MQTT Protocol Based SCADA System наведено саме таку схему. ESP32 використовується як сенсорний gateway/RTU, а Raspberry Pi2 – як локальний сервер; дані від сенсорів надходять через MQTT, обробляються у Node-RED, відображаються через веб-дашборд, а також зберігаються в SQLite (автори прямо формулюють зв’язку «Node-RED + MQTT для обміну/комунікації» і «SQLite як web server/storage»). Ця архітектура характерна для систем, де важливі простота розгортання, локальна автономність і зрозумілий «ланцюжок» телеметрії від датчика до інтерфейсу користувача.

Коли з’являється потреба передавати дані між доменами (наприклад, між різними організаціями/платформами або між edge і центральним сховищем), типово застосовують міст (bridge) між брокерами або окремий «перетворювач/форвардер», що переносить повідомлення в інший сегмент. У роботі Evaluation of MQTT Bridge Architectures in a Cross-Organizational Context

сама «референсна» схема потоку даних включає gateway (MQTT client), Source Broker, компонент Transformer (Bridge)/Forwarder, Destination Broker, а також рівень спостережуваності через Prometheus/Grafana та подальше збереження в Data Warehouse – як частину системи доставки даних до споживача. Для Raspberry Pi це часто означає роль локального брокера/шлюза, який або напряму «моститься» у центральний брокер, або працює як проміжна ланка перед хмарною аналітикою [10].

На рівні більших, горизонтально масштабованих застосунків з'являється архітектурний клас розподілених MQTT-брокерів, де у мережі працює кілька брокерів (на різних вузлах/майданчиках), замість одного центрального. Однак така схема має практичну складність, у «базовому» distributed MQTT підписник зазвичай повинен знати, до якого саме брокера підключатися, щоб отримати потрібні топіки. У TD-MQTT це сформульовано як ключова проблема розподілених архітектур і запропоновано механізм «прозорого» підключення без попереднього знання адрес брокерів, із автоматичним перенесенням даних між брокерами при змінах конфігурації/локації. Для теми Raspberry Pi це важливо як теоретична основа. Pi може бути одним із edge-вузлів у мережі брокерів, але коректність і зручність доступу до даних визначатимуться тим, наскільки прозоро організовано маршрутизацію/перенесення потоків.

1.4 Засоби інтеграції та візуалізації даних

MQTT визначає стандартизований спосіб обміну повідомленнями між видавцями та підписниками (publish/subscribe), але сам по собі не розв'язує задачі перетворення даних, довготривалого зберігання, побудови панелей моніторингу чи аналітичних звітів. Тому в практичних IoT-рішеннях поверх брокера формується «контур інтеграції». Прийом повідомлень, їх нормалізація/фільтрація, маршрутизація в сховища та подальша візуалізація станів і трендів.

Одним із найпоширеніших інструментів інтеграції в MQTT-проєктах є Node-RED як low-code середовище побудови потоків обробки. У типовому сценарії Node-RED підключається до брокера через вузли mqtt-in/mqtt-out, приймає повідомлення з заданих топіків, виконує прикладну обробку (перетворення формату, фільтрацію, агрегацію), а потім або публікує результат у нові топіки, або відправляє дані далі – наприклад, у зовнішні сервіси чи бази даних. Практичні кроки конфігурації MQTT-вузлів та тестування потоку через MQTT-клієнт описані в матеріалі EMQX про використання MQTT у Node-RED.

Для зберігання телеметрії та подальшої побудови графіків часто застосовують сховища часових рядів. У документації EMQX зазначається, що InfluxDB призначена для зберігання й аналізу time-series даних і добре підходить для IoT-навантажень; також описано можливість інтеграції MQTT-потоку з InfluxDB через механізми data integration. На рівні візуалізації типовим вибором є Grafana, яка підключається до джерела даних і відображає часові ряди у вигляді панелей/дашбордів. У прикладі EMQX показано перегляд MQTT-даних у Grafana-дашборді з «оглядом стану та трендами ключових метрик» для пристроїв (демонстраційний сценарій моніторингу) [11].

Окремий напрям – візуалізація та спостережуваність (observability) саме брокера та MQTT-інфраструктури – кількість підключень, швидкість публікацій/доставки, затримки, помилки, завантаження ресурсів. EMQX має вбудований Dashboard та надає доступ до метрик через REST API і системні топіки, що спрощує інтеграцію зі сторонніми системами моніторингу.

Практичну інтеграцію з Prometheus (збір метрик) і Grafana (візуалізація) EMQX описує як готовий шлях, Prometheus виступає джерелом даних для Grafana, а дашборди можна імпортувати як шаблони або налаштовувати вручну.

У наукових роботах та промислових платформах інтеграційний контур часто формалізують як частину end-to-end архітектури «шлюз/клієнт → брокер → компоненти обробки → споживачі/сховища → візуалізація». Наприклад, у дослідженні про bridge-архітектури MQTT прямо показано на схемі моніторинговий стек Prometheus + Grafana та наявність Data Warehouse в

загальному потоці доставки даних, що підкреслює важливість одночасного контролю як телеметрії, так і показників роботи системи.

Нарешті, для налагодження інтеграцій і перевірки коректності обміну повідомленнями використовують клієнтські інструменти з елементами візуалізації. Зокрема, в екосистемі EMQX клієнт MQTTX розвивається в бік «viewer» режимів (JSON/Dashboard) та візуалізації активності MQTT, що корисно при тестуванні топіків, форматів Корисне навантаження і сценаріїв підписки/публікації [12].

1.5 Питання безпеки MQTT

Безпека MQTT не є «вбудованою за замовчуванням» у сенсі обов'язкового шифрування та суворої автентифікації, стандарт лише рекомендує, щоб реалізації клієнта і сервера (брокера) пропонували механізми автентифікації, авторизації та захищеного зв'язку, і особливо наполягає на їх використанні для критичної інфраструктури та чутливих даних. Це означає, що реальна безпека MQTT-системи визначається не стільки самим протоколом, скільки політиками доступу, криптографічним захистом каналу, конфігурацією брокера та практиками розгортання.

У літературі з безпеки MQTT наголошується, що в «незахищених реалізаціях» протокол проявляє вразливості саме через відсутність обов'язкових механізмів шифрування й автентифікації. У такому випадку можливі DoS-атаки на брокер/сервер, несанкціонований доступ до даних і перехоплення/спостереження за обміном повідомленнями, що порушує конфіденційність. Важливий практичний аргумент на користь цієї тези дає масштабне дослідження «реальних бекендів», що обслуговують IoT-протоколи. Автори повідомляють, що 99,84 % MQTT- та XMPP-бекендів використовують незахищені транспортні протоколи, і лише 0,16 % застосовують TLS (причому значна частка з них – уразливі версії). Отже, ключовою проблемою часто є не теоретична «слабкість MQTT», а типові помилки/спрощення під час розгортання.

На практиці базовий рівень захисту будується навколо TLS (MQTTS). Для брокера Mosquitto документація прямо вказує, якщо використовується автентифікація користувач/пароль (через password_file), потрібно обов’язково застосовувати мережеве шифрування, інакше облікові дані можуть бути перехоплені. Там же описані й варіанти сертифікатної моделі (mTLS) – параметр require_certificate змушує клієнта надати валідний сертифікат; додатково можливе зіставлення ідентичності клієнта з Common Name (CN) сертифіката як «іменем користувача» для контролю доступу. В академічній/стандартизованій площині цю логіку продовжує профіль ACE для MQTT (IETF RFC 9431), який формалізує підхід, де OAuth 2.0 access tokens із proof-of-possession ключами використовуються для автентифікації/авторизації клієнтів, а TLS забезпечує конфіденційність і автентифікацію брокера [13].

Другий критичний пласт – авторизація (ACL) на рівні топіків і принцип найменших привілеїв. Навіть із TLS небезпечно залишати клієнтам можливість підписки/публікації «будь-куди». Тому доцільно розглядати політику доступу як центральний елемент безпеки. Розмежування прав publish/subscribe за ієрархією топіків, ізоляція сервісних топіків, контроль retained-повідомлень і Last Will, а також аудит конфігураційних рішень (відключення анонімного доступу, розділення listener’ів для локальної мережі та зовнішніх з’єднань тощо). Як доповнення до превентивних механізмів, окрема гілка досліджень пропонує IDS-підходи на основі ML для виявлення аномалій у MQTT-трафіку та атак (зокрема DoS/несанкціоновані дії), що підкреслює важливість моніторингу та кореляції мережевих ознак у продакшн-розгортаннях.

MQTT є OASIS-стандартом (MQTT 5 та MQTT 3.1.1 мають офіційні специфікації), і коректна інформаційна безпека має узгоджуватися з вимогами/положеннями стандарту та документованими можливостями брокера [14].

РОЗДІЛ 2
ВИБІР АПАРАТНИХ СКЛАДОВИХ ТА АНАЛІЗ ІСНУЮЧИХ ТЕХНІЧНИХ РІШЕНЬ

2.1 Обґрунтування вибору платформи

Платформа Raspberry Pi є доцільною базою для магістерського проєкту з розробки та тестування MQTT-комунікації, оскільки MQTT є «надлегким» publish/subscribe протоколом для IoT, орієнтованим на мінімальний мережевий трафік і малий «код-футпринт», що добре узгоджується з концепцією edge-обчислень на одноплатних комп'ютерах. Водночас Raspberry Pi надає повноцінне Linux-середовище для брокера, моніторингу, журналювання та відтворюваних експериментів, що важливо саме для кваліфікаційної роботи (не прототип «на ардуінці», а керована інженерна платформа з ОС, сервісами й тестовими інструментами).

Для MQTT-брокера (та супутніх сервісів на кшталт Node-RED/логування/візуалізації) критичними є CPU, обсяг RAM, мережевий інтерфейс і швидкість накопичувача. Raspberry Pi 4 Model B має 64-бітний чотириядерний Cortex-A72 @ 1.8 GHz, варіанти RAM 1/2/4/8 GB, Gigabit Ethernet, Wi-Fi 802.11ac (2.4/5 GHz), Bluetooth 5.0 та USB 3.0, що робить його практичним вибором для edge-розгортання брокера й базових навантажувальних тестів [15].

Raspberry Pi 3 підходить, якщо у роботі плануються інтенсивні експерименти – великий потік повідомлень, багато паралельних клієнтів, TLS/mTLS, додаткові сервіси моніторингу. В офіційному описі Pi 5 зазначено процесор BCM2712 із 4-ядерним Cortex-A76 @ 2.4 GHz і варіанти RAM до 16 GB. Також у product brief підкреслено високошвидкісний режим microSD (SDR104) та наявність USB 3.0 – це важливо для стабільності логування/бази даних під час тестів.

Однакова ОС, однакові версії пакунків/конфігів, можливість автоматизованого розгортання. Raspberry Pi має офіційно підтримувану

Raspberry Pi OS, яку встановлюють через Raspberry Pi Imager або завантажують як готові образи. Перехід на Debian Bookworm як основу Raspberry Pi OS описаний у новинному повідомленні Raspberry Pi; при цьому звертається увага на нюанси сумісності 32-bit у певних сценаріях, тому для сучасних плат (Pi 4/5) практично обґрунтовано використовувати 64-bit середовище для стабільної роботи сервісів.

Як базовий брокер для експериментів часто обирають Eclipse Mosquitto, бо він добре підтримується у Debian-екосистемі. На офіційному сайті Mosquitto прямо зазначено, що Mosquitto «тепер у Debian proper», тобто його можна інсталювати стандартними засобами пакетного менеджера, а оновлення проходять у межах типових процедур Debian.

Raspberry Pi виступає не просто «залізом під брокер», а контрольованим експериментальним стендом, де можна:

- запускати брокер і клієнтів у заданих умовах мережі (Ethernet/Wi-Fi);
- збирати логи та метрики ОС/сервісів;
- відтворювати сценарії навантаження.

2.2 Вибір та опис клієнтських вузлів MQTT

У межах системи MQTT «клієнтський вузол» – це будь-який компонент, що встановлює з'єднання з брокером і виконує роль publisher, subscriber або поєднує обидві ролі. Для кваліфікаційної роботи доцільно вибирати такі клієнтські вузли, які репрезентують типові IoT-ролі «польового» рівня й рівня обробки/візуалізації, дозволяють відтворювано генерувати навантаження та вимірювати параметри обміну, мають документовану підтримку MQTT-версій та механізмів надійності/безпеки.

Найбільш показовим «польовим» клієнтом є ESP32, оскільки він типово використовується як сенсорний/виконавчий вузол або gateway у низьковартісних IoT/SCADA-рішеннях. Саме така логіка відображена у відомому прикладі SCADA-системи, де ESP32 використовується як кінцевий вузол збору даних і

керування, а взаємодія із локальним сервером організована через MQTT. Для реалізації MQTT на ESP32 доцільно спиратися на офіційний стек Espressif ESP-MQTT (ESP-IDF), який прямо описаний як реалізація MQTT-клієнта і підтримує MQTT v5.0; у документації також вказано наявність параметрів конфігурації протоколу та транспортів (зокрема SSL/TLS і WebSocket). Це робить ESP32 відповідним клієнтом для тестів QoS, поведінки при розривах з'єднання та сценаріїв захищеної комунікації.

Другий клас клієнтських вузлів – інтеграційний/логічний рівень, де доцільно використати Node-RED як MQTT-клієнт і одночасно як середовище побудови потоків обробки. Node-RED має стандартні вузли MQTT Input/MQTT Output, що підключаються до брокера через спільний конфігураційний вузол (MQTT Config), і є зручними для формування сценаріїв «підписка → перетворення → публікація», а також для швидкого прототипування інтеграції з базами даних/панелями. Практична релевантність цієї зв'язки підтверджується й у згаданій SCADA-роботі, де Node-RED використано на локальному сервері разом із MQTT для обміну даними та побудови веб-інтерфейсу моніторингу/керування [16].

Третій обов'язковий компонент – керований клієнт для тестування та навантаження на ПК/ноутбуці або на самому Raspberry Pi. Для цього логічно застосовувати бібліотеки Eclipse Paho (наприклад, Python-клієнт), оскільки вони документовано підтримують MQTT 5.0/3.1.1/3.1, а також ключові експлуатаційні функції, важливі для дослідження LWT (Last Will), SSL/TLS, message persistence, automatic reconnect, offline buffering, WebSocket тощо. Такий клієнт дозволяє реалізувати відтворювані експерименти з параметрами QoS, частотою публікацій, розміром Корисне навантаження та кількістю паралельних підписок.

Окремо варто передбачити утиліти швидкої верифікації (діагностики) – насамперед mosquitto_pub і mosquitto_sub. Їх перевага в тому, що це «простий MQTT-клієнт» для публікації/підписки, який, за ман-сторінками Mosquitto, підтримує MQTT v5/3.1.1 і може працювати з TLS, що робить ці інструменти придатними для оперативних перевірок коректності конфігурації брокера,

топіків і доступу. Як доповнення до CLI-інструментів можна використовувати GUI-клієнт на кшталт MQTT Explorer, який надає структурований огляд топіків і полегшує ручну перевірку станів (це особливо корисно під час налагодження).

2.3 Програмно-апаратний стек рішення

Програмно-апаратний стек MQTT-рішення в межах цієї кваліфікаційної роботи доцільно будувати як edge-стенд – Raspberry Pi виконує роль локального сервера з брокером MQTT і сервісами інтеграції/зберігання/візуалізації, а клієнтські вузли (наприклад, ESP32 та програмні клієнти на ПК) генерують телеметрію й команди керування. Такий підхід відповідає практиці «single machine system» для локального моніторингу без обов'язкової хмари, описаній у роботі про SCADA на базі ESP32, Raspberry Pi, Node-RED та MQTT.

На апаратному рівні базовим вузлом є Raspberry Pi з мережевим підключенням (Ethernet або Wi-Fi), мікроSD/SSD-накопичувачем та стабільним живленням. На Raspberry Pi встановлюється Raspberry Pi OS (офіційно підтримуються 32- та 64-бітні редакції), що забезпечує стандартне Linux-середовище для сервісів брокера, інтеграції та тестування.

На програмному рівні «ядром» виступає брокер Eclipse Mosquitto, який реалізує сервер MQTT (MQTT 5.0 / 3.1.1 / 3.1) і постачається разом з утилітами mosquitto_pub та mosquitto_sub для швидкої перевірки сценаріїв publish/subscribe. Для інтеграції та бізнес-логіки доцільно використати Node-RED (потокове low-code середовище), яке офіційно має сценарій встановлення на Raspberry Pi/сумісні Debian-системи та типово застосовується як MQTT-клієнт у системах збору даних і керування.

Як рівень зберігання й подальшої візуалізації можна використати дві практично обґрунтовані гілки – «легку» локальну БД (наприклад, SQLite) разом із Node-RED-дашбордом – це прямо відображено в SCADA-підході, де Raspberry Pi виступає локальним сервером із MQTT, Node-RED, SQLite; окрему систему метрик/спостережуваності, де брокер і вузли віддають метрики в Prometheus, а

Grafana використовується для панелей, що добре відповідає сучасним практикам інструментування й візуалізації стану інфраструктури.

Безпекова конфігурація є частиною стеку, а не «додатком». Для Mosquitto в документації прямо зазначено, що при використанні username/password через password_file потрібно застосовувати мережеве шифрування (TLS), інакше облікові дані можуть бути перехоплені. На рівні стандарту MQTT 5.0 також підкреслюється lightweight publish/subscribe-модель і її застосовність для IoT, що задає рамку коректного проєктування системи з урахуванням QoS/сесій/доставки.

Нижче наведено узгоджений склад стеку, який можна зафіксувати в роботі як «базову конфігурацію стенду», таблиця 2.1.

Таблиця 2.1 – Базова конфігурація стенду

Рівень	Компонент	Роль у рішенні
Edge-сервер	Raspberry Pi + Raspberry Pi OS	Хостинг брокера/інтеграції/БД, кероване експериментальне середовище
Брокер MQTT	Eclipse Mosquitto	Маршрутизація publish/subscribe; MQTT 5.0/3.1.1; утиліти pub/sub для тестів
Інтеграція/логіка	Node-RED	Потоки «підписка → обробка → збереження/візуалізація»; типовий елемент SCADA-архітектури
Зберігання/візуалізація	SQLite / (Prometheus+Grafana)	Локальне збереження (SCADA приклад) або дашборди метрик/спостережуваності
Безпека	TLS + password_file/ACL	Захист каналу та контроль доступу (рекомендація Mosquitto щодо шифрування при password_file)

2.4 Аналіз існуючих реалізацій

У практичних реалізаціях MQTT-рішень на Raspberry Pi найчастіше зустрічається «однонордовий» сценарій, Raspberry Pi виконує роль локального брокера, а тестування клієнтів здійснюється з того ж вузла або з окремих пристроїв у LAN. Типовий приклад такого підходу – інсталяція Eclipse Mosquitto з пакунків Debian/Raspbian та використання mosquitto-clients для базової перевірки publish/subscribe без додаткових компонентів візуалізації. Це відповідає ідеї MQTT як легкого publish/subscribe протоколу з брокером

(сервером) та клієнтами (publisher/subscriber), що добре масштабується «знизу» від навчальних стендів [7].

Більш «прикладний» клас реалізацій – edge/SCADA-підхід, де Raspberry Pi не лише тримає брокер, а й завершує контур збору, зберігання та відображення даних. У роботі про низьковартісну SCADA-систему автори описують схему ESP32 як RTU/шлюз публікує телеметрію (температура, вологість, тиск, освітленість) у теми MQTT, Mosquitto працює на Raspberry Pi (як локальному сервері), а Node-RED використовується для побудови dashboard/HMI; паралельно дані можуть зберігатися у SQLite на цьому ж вузлі. Така архітектура показова для дипломної роботи, бо демонструє повний ланцюжок «датчики → MQTT → обробка/візуалізація» на доступному стеку [8].

Окремий важливий підклас – сценарії розгортання з «мостами» (MQTT bridging), коли дані переносяться між брокерами різних сегментів мережі, локацій або організацій. У одних із досліджень розглянуто реалістичну для IoT проблему – виробники/споживачі даних часто рознесені мережею та адміністративними межами, тому застосовують bridge-архітектуру між брокерами, а також клієнтські варіанти bridging із можливістю трансформації/гармонізації даних. Автори окремо вимірюють затримку та втрати повідомлень (надійність) і зазначають, що на якість впливають, зокрема, розмір Корисне навантаження та навіть довжина імен тем. Для кваліфікаційної роботи це дає «готову рамку» метрик та факторів, які слід варіювати у експерименті.

Як тільки вимоги виходять за межі одного вузла (десятки/сотні тисяч клієнтів, висока пропускна здатність, гарантована надійність та відмовостійкість), з'являються розподілені брокери. Показовий приклад – TBMQ (ThingsBoard Message Queue), який у статті Journal of Big Data описується як масштабований та fault-tolerant MQTT broker з горизонтальним масштабуванням, відсутністю єдиної точки відмови та інтеграцією Apache Kafka для обробки/доставки повідомлень.

Нарешті, сучасні реалізації все частіше «накривають» MQTT безпековими шарами (TLS, ACL, ізоляція, контроль автентифікації/авторизації), бо емпіричні

вимірювання в реальному інтернеті показують значну частку небезпечних конфігурацій бекендів IoT-протоколів (витоки інформації, слабка автентифікація тощо). Паралельно з «класичною» безпекою з'являються і дослідницькі реалізації приватності. Пропонується підхід до анонімізації в MQTT через мережу «peer broker» із використанням bridging на стороні брокерів; автори прямо відзначають компроміс – краща анонімність ціною додаткової латентності [9].

2.5 Вимоги до проєктованої системи

Проєктована система призначена для розгортання та експериментальної перевірки MQTT-комунікації на базі Raspberry Pi як edge-сервера. Вимоги формуються з урахуванням того, що MQTT є client-server publish/subscribe протоколом, легким і придатним для IoT-оточень, а також того, що якість доставки даних у практичних архітектурах суттєво залежить від параметрів пакетів, довжин топіків і кількості bridge-компонентів.

Безпекові вимоги виводяться з рекомендацій щодо застосування TLS та принципів автентифікації/авторизації в MQTT-середовищі. Вимоги систематизовано в таблиці 2.2.

Таблиця 2.2 – Вимоги до проєктованої MQTT-системи на Raspberry Pi

ID	Група	Вимога	Критерій приймання / перевірка
F1	Функц.	Підтримка MQTT як broker-centric pub/sub системи (сервер/клієнти)	Успішні publish/subscribe сесії між ≥2 клієнтами через брокер.
F2	Функц.	Підтримка QoS-рівнів (0/1/2) у тестах доставки	Відтворювані сценарії для QoS 0/1/2; фіксація латентності/втрат. (QoS як базова можливість MQTT 5).
F3	Функц.	Підтримка сценарію «один локальний брокер (LAN)»	Тестовий стенд працює у локальній мережі, клієнти підключаються до Pi-брокера.
F4	Функц.	Підтримка сценарію «bridge між брокерами» (edge→інший домен)	Розгорнуто 2 брокери та bridge/bridge-processor; заміряно затримка та надійність для різних конфігурацій.

Продовження таблиці 2.2

ID	Група	Вимога	Критерій приймання / перевірка
F5	Інтегр.	Інтеграція потоку MQTT у прикладний рівень (обробка/маршрутизація)	Налаштований інтеграційний контур (наприклад, Node-RED) приймає MQTT-повідомлення та віддає їх у візуалізацію/сховище.
N1	Надійність	Оцінювання надійність доставки повідомлень у тестах	Для кожного сценарію фіксується частка втрат/невчасної доставки; порівняння QoS і архітектур.
P1	Продукт.	Оцінювання затримки доставки (затримка)	Для кожної серії тестів обчислюються метрики латентності (наприклад медіана/середнє), як у роботах з benchmarking bridge.
P2	Продукт.	Вплив Розмір корисного навантаження та довжини topic name на QoS/затримка/надійність	План тестів обов'язково варіює Розмір корисного навантаження і topic name, бо вони впливають на якісні атрибути.
S1	Безпека	Шифрування каналу (TLS) для захисту облікових даних і трафіку	Якщо використано username/password, TLS є обов'язковим (вимога з документації Mosquitto).
S2	Безпека	Підтримка розширених схем authN/authZ (за потреби)	Як «еталонний» напрям: ACE/OAuth2 + PoP, де TLS використовується для конфіденційності та автентифікації брокера.
O1	Експлуатац.	Відтворюваність стенду та конфігурацій	Конфігурації брокера/клієнтів/інтеграції зберігаються й можуть бути повторно розгорнуті на Raspberry Pi OS.
O2	Масштабованість (аналіт.)	Обґрунтування меж Raspberry Pi-стенду на фоні large-scale брокерів	В огляді враховано, що існують розподілені fault-tolerant брокери з горизонтальним масштабуванням (як референс для порівняння).

Додатково до табличних вимог важливо зафіксувати, що проєктована система має бути орієнтована не лише на «працює/не працює», а на вимірювані експерименти. Тестовий план повинен окремо охоплювати локальний режим з одним брокером, bridge-режим між брокерами, а також систематичну зміну Розмір корисного навантаження, topic name і QoS, бо саме ці фактори показані в літературі як такі, що впливають на затримка/надійність. Одночасно має бути забезпечено мінімальний безпековий профіль (TLS при використанні паролів), оскільки без шифрування облікові дані піддаються перехопленню.

РОЗДІЛ 3
ПОБУДОВА, НАЛАШТУВАННЯ ТА ТЕСТУВАННЯ MQTT-СИСТЕМИ НА RASPBERRY PI

3.1 Проєктування та збирання апаратної частини

Апаратна частина проєктованої системи побудована як стенд класу edge-LAN – Raspberry Pi виконує роль локального сервера (MQTT-брокер Mosquitto та сервіси інтеграції/візуалізації), а клієнтські вузли (ESP32) генерують телеметрію й приймають керувальні команди. Загальна логіка взаємодії апаратних компонентів подана на рисунку 3.1, а перелік апаратних компонентів стенду в таблиці 3.1. Топологію описано на рисунку 3.2.

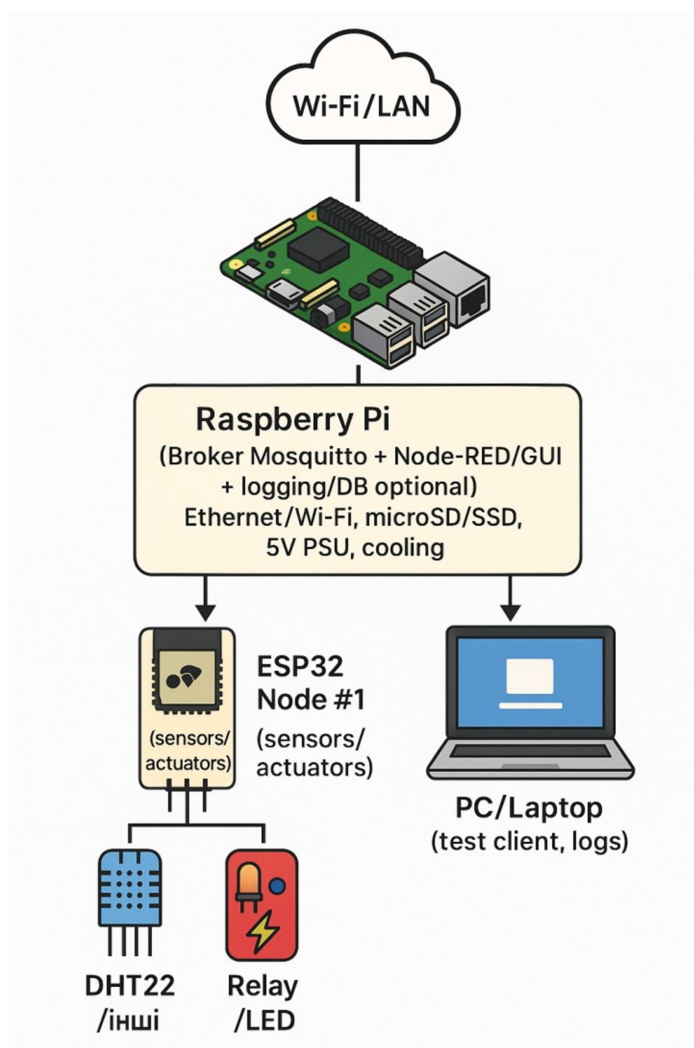

Рисунок 3.1 – Структурна схема апаратної частини MQTT-стенду

Таблиця 3.1 – Перелік апаратних компонентів стенду

Позначення	Компонент	К-сть	Примітка (вибір)
H1	Raspberry Pi 4 (4-8 GB)	1	Pi 4 достатньо для «broker+клієнти+базова візуалізація»
H2	Блок живлення 5 V (офіційний/якісний)	1	Критично для стабільності під навантаженням.
H3	microSD (A1/A2) або SSD (через USB)	1	Для надійності логування/БД краще SSD.
H4	Корпус + активне/пасивне охолодження	1	Зменшує ризик тротлінгу під навантаженням.
H5	ESP32 DevKit (клієнтський вузол)	1N	Вузол телеметрії/керування.
H6	Датчик DHT22 (опційно)	1	Для демонстраційної телеметрії (t/RH).
H7	Реле-модуль 5 V з оптопарою (опційно) або LED + резистор	1	Для сценаріїв керування (cmd/ack).
H8	Макетна плата, дроти Dupont	1 набір	Прототипування з’єднань

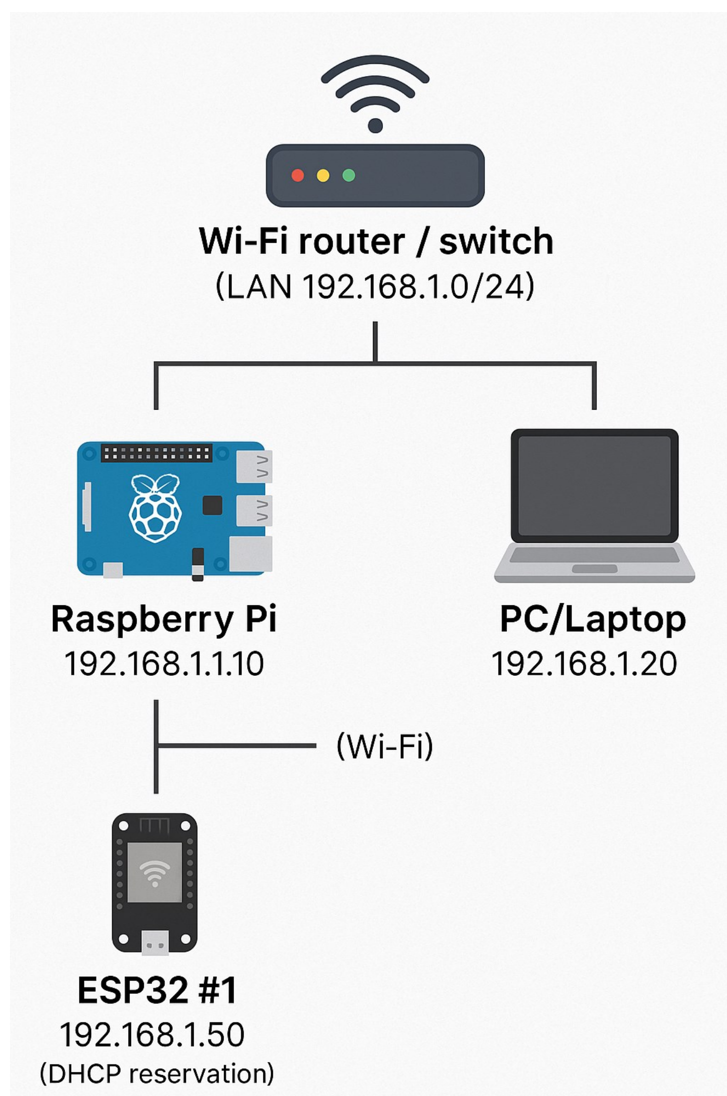

Рисунок 3.2 – Топологія мережі стенду (LAN)

Практично зручно закріпити адреси через DHCP reservation, щоб у тестах не змінювалися параметри підключення клієнтів до брокера.

3.1.1 Підключення клієнтського вузла ESP32 (сенсори/керування)

Найпростіший сенсорний вузол для демонстрації MQTT-телеметрії – ESP32 та DHT22 (рис. 3.3). Для сценаріїв керування додається LED або реле (як «виконавчий елемент»), щоб реалізувати повний ланцюг cmd → виконання → ack.

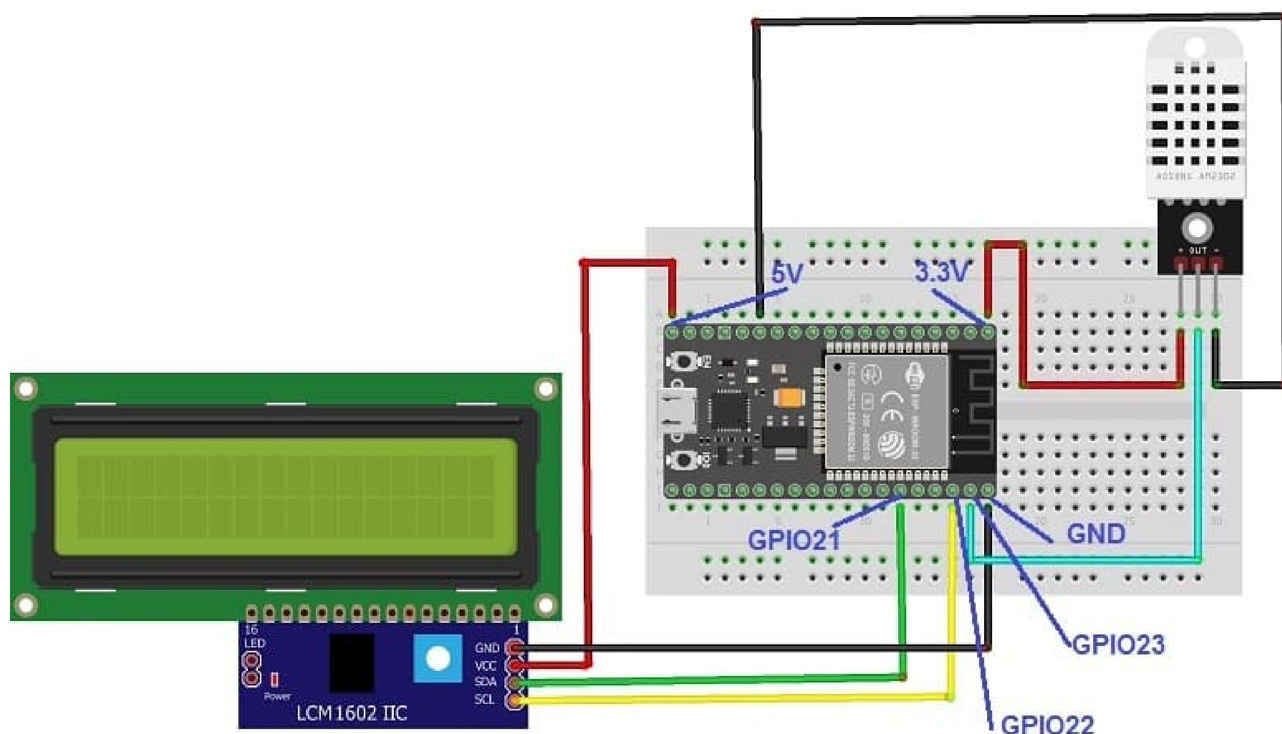

Рисунок 3.3 – Схема підключення ESP32, DHT22, LED

3.1.2 Послідовність збирання стенду

Спочатку виконується підготовка Raspberry Pi, плату встановлюють у корпус, під’єднують систему охолодження, обирають і підключають носій даних (microSD або SSD), налаштовують мережеве підключення, причому для стабільних вимірювань доцільніше використовувати Ethernet, і забезпечують надійне живлення 5 V. Далі збирають клієнтський вузол на базі ESP32, на макетній платі під’єднують датчик DHT22 та індикаторний LED чи реле відповідно до схеми на рисунку 3.3, щоб забезпечити можливість як передачі

телеметрії, так і керування виконавчим елементом. Після цього ESP32 підключають до Wi-Fi у тому самому мережевому сегменті, де працює Raspberry Pi, і фіксують IP-адресу (DHCP reservation), щоб у подальших тестах уникати зміни параметрів доступу та зберегти відтворюваність експериментів. На етапі фізичного компонування стенду вузли розміщують так, аби мінімізувати випадкові розриви контактів, застосовують короткі дроти, маркують лінії, за потреби використовують термоусадку або стяжки для механічної стабілізації з'єднань. Перед запуском серій тестів проводять контроль стабільності роботи, перевіряють, що Raspberry Pi не перезавантажується під навантаженням (що найчастіше вказує на проблеми з живленням), а ESP32 не втрачає Wi-Fi-з'єднання під час тривалих сесій обміну даними. (рис. 3.4)

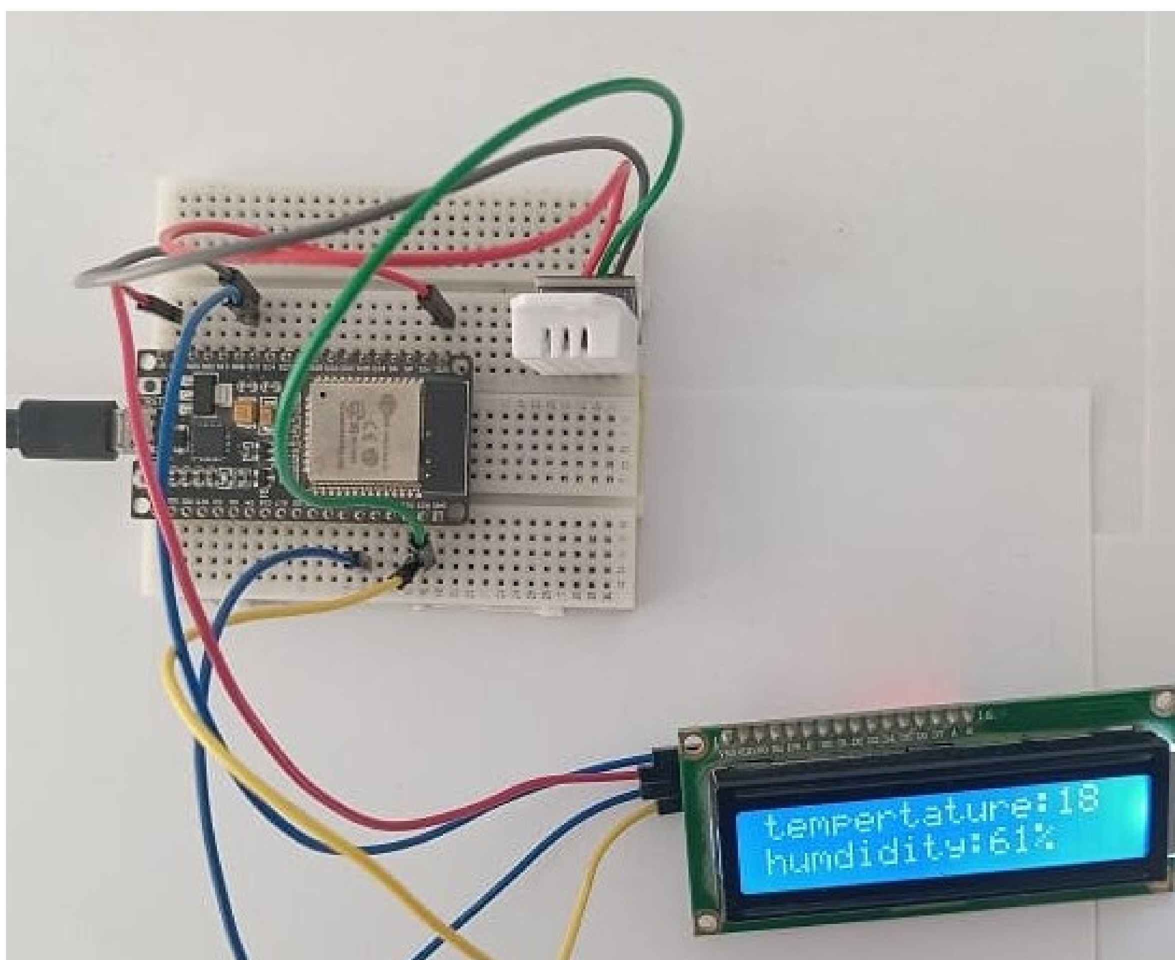

Рисунок 3.4 – Фізичне компонування стенду

3.2 Розгортання брокера Mosquitto на Raspberry Pi

Спочатку на Raspberry Pi оновлюють систему та встановлюють брокер Mosquitto разом із консольними утилітами-клієнтами, після чого вмикають службу для автозапуску й перевіряють її поточний стан. Для цього виконують оновлення пакетів (sudo apt update && sudo apt upgrade), інсталяцію mosquitto і mosquitto-clients, далі активують сервіс командою sudo systemctl enable mosquitto.service та контролюють, чи він запущений і працює коректно через sudo systemctl status mosquitto --no-pager. За потреби додатково перевіряють версію і режим роботи брокера командою mosquitto -v. Для гілки Mosquitto 2.x типовим є повідомлення про режим «local only mode», що означає приймання з'єднань лише з локальної машини, а також рекомендація явно налаштувати конфігурацію з параметром listener, якщо планується підключення клієнтів з інших пристроїв у мережі.

Для базової перевірки роботи брокера безпосередньо на Raspberry Pi відкривають два термінали. В одному запускують клієнт підписки командою mosquitto_sub -t 'test/topic' -v, а в другому надсилають тестове повідомлення за допомогою mosquitto_pub -t 'test/topic' -m 'hello'. Якщо в першому терміналі відображається надісланий рядок, це свідчить про коректне функціонування брокера та обміну повідомленнями на локальній машині. При цьому в документації Mosquitto окремо наголошується, що у випадку використання механізму password_file необхідно обов'язково вмикати мережеве шифрування (TLS), оскільки передавання облікових даних у відкритому вигляді робить їх уразливими до перехоплення.

Щоб налаштувати автентифікацію через логін і пароль, спочатку формують парольний файл Mosquitto за допомогою штатної утиліти mosquitto_passwd, наприклад командою sudo mosquitto_passwd -c /etc/mosquitto/passwd mqttuser, де ключ -c створює новий файл і додає до нього користувача. Після цього важливо обмежити доступ до цього файлу, щоб запобігти несанкціонованому читанню або зміні, власником призначають системного користувача та групу Mosquitto (sudo

chown mosquitto:mosquitto /etc/mosquitto/passwd), а права виставляють так, щоб читання й запис були дозволені лише власнику (sudo chmod 600 /etc/mosquitto/passwd).

Щоб брокер приймав підключення на стандартному порту 1883 і вимагав автентифікацію, у файлі конфігурації /etc/mosquitto/mosquitto.conf задають мінімальний набір параметрів – per_listener_settings true для коректного застосування налаштувань на рівні конкретного listener, allow_anonymous false для заборони анонімних підключень, listener 1883 для явного відкриття порту та password_file /etc/mosquitto/passwd для підключення створеного парольного файлу. Параметри password_file, allow_anonymous і per_listener_settings, а також їхня взаємодія, описані в офіційній документації конфігурації Mosquitto (mosquitto.conf). Після внесення змін службу брокера перезапускають командою sudo systemctl restart mosquitto, щоб конфігурація набула чинності.

Щоб увімкнути захищений канал TLS для клієнтських підключень, у Mosquitto налаштовують окремий listener, зазвичай на порту 8883, і прив'язують до нього файли сертифікації. Для цього в конфігурації додають блок із listener 8883, після чого вказують шлях до кореневого сертифіката центру сертифікації (cafile /etc/mosquitto/certs/ca.crt), сертифіката сервера (certfile /etc/mosquitto/certs/server.crt) та приватного ключа сервера (keyfile /etc/mosquitto/certs/server.key). За потреби можна додатково керувати вимогою клієнтського сертифіката параметром require_certificate коли він увімкнений, клієнт має надати валідний сертифікат (mTLS), а коли вимкнений – достатньо лише перевірки сертифіката брокера; детальна поведінка цього параметра описана в man-page Mosquitto. Практично найпоширеніший підхід – мати два окремі listeners: 1883 без TLS (тільки для внутрішніх лабораторних перевірок у LAN) та 8883 з TLS для захищених підключень, оскільки в межах одного listener одночасно «змішати» TLS і незашифрований трафік не передбачається. Після внесення змін конфігурації брокер перезапускають командою sudo systemctl restart mosquitto, щоб активувати TLS-настройки.

Щоб переконатися, що Mosquitto справді запущений і приймає з’єднання на потрібних портах, спочатку перевіряють стан служби командою sudo systemctl status mosquitto --no-pager, яка показує, чи сервіс активний і чи немає помилок запуску. Далі контролюємо, чи відкриті порти 1883 і 8883 на рівні мережевих сокетів, використовуючи sudo ss -ltnp | grep -E '(:1883|:8883)': у виводі має з’явитися запис про процес Mosquitto, що «слухає» відповідний TCP-порт.

3.3 Реалізація сценаріїв обміну даними

Для того щоб сценарії обміну даними були відтворюваними й легко порівнювалися між різними серіями тестів, доцільно одразу стандартизувати простір топіків і спосіб ідентифікації вузлів. Практично зручно використовувати ієрархічний неймінг із ключем deviceId, lab/<deviceId>/telemetry для телеметрії сенсорів, lab/<deviceId>/status для службового стану пристрою (онлайн/офлайн, версія прошивки, RSSI та інші діагностичні поля), lab/<deviceId>/cmd/<name> для керувальних команд (увімкнення LED, перемикання реле, зміна конфігурації), а також lab/<deviceId>/ack/<name> для підтверджень виконання відповідних команд. Така структура розділяє потоки даних за призначенням і дозволяє коректно й окремо оцінювати параметри якості сервісу – QoS, затримку та надійність доставки – для телеметричних повідомлень і для команд керування, які зазвичай мають різні вимоги до гарантій доставки та реакції системи.

У сценарії телеметрії дані передаються за моделлю «пристрій → брокер → споживачі», вузол ESP32 виступає публікатором і з певним інтервалом надсилає показники сенсорів у топік telemetry, тоді як Node-RED або ПК працюють як підписники, приймаючи ці повідомлення для відображення на панелі, запису в журнал або збереження у сховищі. Для лабораторного стенду доцільно одразу визначити рівень QoS залежно від вимог до повноти даних. Пріоритет швидкості та мінімальних накладних витрат відповідає QoS 0, де можливі поодинокі втрати, а коли важливо не пропускати значення, раціональніше застосовувати QoS 1 як типову компромісну опцію між надійністю та продуктивністю. Як формат

повідомлення зручно використовувати JSON із часовою міткою та полями вимірювань, наприклад {"ts":1733170000000,"t":24.6,"h":41.2,"v":3.29}. Перевірити роботу цього сценарію можна штатними клієнтами Mosquitto, на стороні споживача запускають підписку mosquitto_sub -h <PI_IP> -t 'lab/+/telemetry' -v, а для імітації публікації з боку ESP32 надсилають тестове повідомлення командою mosquitto_pub -h <PI_IP> -t 'lab/esp32-01/telemetry' -m '{"ts":1733170000000,"t":24.6,"h":41.2}' -q 0, після чого повідомлення має з'явитися у вікні підписника.

У сценарії керування реалізується взаємодія «інтерфейс → брокер → пристрій» із обов'язковим підтвердженням виконання. Node-RED або ПК виступає публікатором і надсилає команду в топік cmd/..., ESP32 підписується на відповідні командні топіки, виконує дію (наприклад, перемикає LED або реле) і повертає результат у топік ack/.... Для таких повідомлень доцільно застосовувати QoS 1, оскільки це зменшує ризик того, що керувальна команда або підтвердження буде втрачено під час передавання. Типовий приклад команди – публікація в lab/esp32-01/cmd/led з корисним навантаженням {"reqId":"c8f2","state":1}, де reqId використовується як ідентифікатор запиту, а state задає бажаний стан виконавчого елемента. Після виконання ESP32 формує підтвердження в lab/esp32-01/ack/led, {"reqId":"c8f2","ok":true,"applied":1, "ts":1733170001000}, що дозволяє однозначно пов'язати ack із конкретною командою та зафіксувати час застосування. Перевірити роботу сценарію можна через CLI, спочатку на стороні інтерфейсу запускають прослуховування підтверджень командою mosquitto_sub -h <PI_IP> -t 'lab/esp32-01/ack/#' -v, а потім надсилають команду керування mosquitto_pub -h <PI_IP> -t 'lab/esp32-01/cmd/led' -m '{"reqId":"c8f2","state":1}' -q 1, після чого очікують появу ack у вікні підписника.

Сценарій «останнього відомого стану» застосовується тоді, коли системі потрібно зберігати актуальне значення параметра і одразу віддавати його новим підписникам без очікування наступного циклу публікації. Для таких випадків у MQTT використовується механізм retained, брокер зберігає останнє

повідомлення в конкретному топіку та автоматично надсилає його кожному клієнту, який щойно підписався на цей топік. У рамках стенду це зручно реалізувати, наприклад, через retained-статус пристрою lab/esp32-01/status із повідомленням на кшталт {"online":true,"ts":...} та retained-стан виконавчого елемента lab/esp32-01/state/led із корисним навантаженням {"state":1,"ts":...}. Практична перевірка виконується командою публікації з прапорцем retained, mosquitto_pub -h <PI_IP> -t 'lab/esp32-01/state/led' -m '{"state":1,"ts":1733170001000}' -r, після чого будь-який новий підписник, що підключиться до цього топіка, має негайно отримати збережене на брокері значення; режим retained є штатною опцією утиліти mosquitto_pub.

Сценарій контролю доступності вузла базується на механізмі LWT (Last Will and Testament), під час встановлення з'єднання (CONNECT) клієнт заздалегідь задає Will-повідомлення, яке брокер автоматично опублікує у випадку некоректного завершення сесії, тобто коли пристрій «зникає» без штатного DISCONNECT (наприклад, через втрату живлення або розрив мережі). У специфікації MQTT 5.0 ця логіка формалізована параметрами Will Flag, Will QoS і Will Retain, які передаються в CONNECT і визначають, чи буде Will активним, з якими гарантіями доставки він публікується та чи зберігатиметься як retained-стан. Практично зручно організувати це через один службовий топік стану lab/<deviceId>/status, при старті клієнт одразу публікує retained-повідомлення online=true, щоб у системі фіксувалась його наявність, а Will налаштовує як retained-повідомлення online=false у той самий топік, щоб у разі аварійного відключення брокер сам встановив коректний офлайн-стан, який відразу побачать усі підписники, включно з тими, що підключаться пізніше.

У практичній частині доцільно використовувати Node-RED як інтеграційний «оркестратор», оскільки він дозволяє в одному середовищі реалізувати і збирання телеметрії, і керування пристроями, і візуалізацію результатів. Зокрема, сценарії A-D можна оформити як взаємопов'язані потоки, вхідні MQTT-повідомлення з lab/+/telemetry приймаються вузлом MQTT In, за потреби проходять перетворення або фільтрацію (парсинг JSON, нормалізація

полів, відсікання аномалій) і далі передаються в панель Dashboard або в сховище даних/БД; керувальні дії користувача реалізуються через елементи інтерфейсу (кнопка, перемикач), які формують команду та надсилають її у lab/<deviceId>/cmd/... через MQTT Out; підтвердження виконання команд з lab/<deviceId>/ack/# приймаються окремим потоком і відображаються як повідомлення в інтерфейсі або записуються в журнал для подальшого аналізу. Підключення Node-RED до MQTT-брокера при цьому налаштовується через вузли MQTT Input/MQTT Output із використанням спільного конфігураційного вузла MQTT Config, як це рекомендується в матеріалах Node-RED Cookbook.

3.4 Методика та результати тестування

Тестування MQTT-системи доцільно будувати як відтворюваний стенд, у якому Raspberry Pi виконує роль брокера (Eclipse Mosquitto), а клієнти (ПК/ESP32/Node-RED) – ролі publisher/subscriber. Базову верифікацію обміну зручно робити штатними утилітами mosquitto_pub/mosquitto_sub, які є простими клієнтами MQTT v5/3.1.1 та підтримують, зокрема, TLS-з'єднання. Для інтеграції/візуалізації використовують Node-RED через MQTT Input/Output та MQTT Config node (підключення до локального або віддаленого брокера) і, за потреби, перетворення JSON-повідомлень. Для автоматизованих серій тестів (пакетні запуски, контроль параметрів QoS/частоти/розміру повідомлень) доцільно застосовувати програмного клієнта на Python із бібліотекою Eclipse Paho (публікація/підписка/керування сесією).

Оскільки якість MQTT-обміну суттєво залежить від параметрів розгортання, у методиці варто передбачити щонайменше два сценарії – один локальний брокер (LAN) і bridge-архітектура між брокерами (edge→інший брокер/сегмент), бо в роботі з оцінювання bridge-архітектур показано, що на латентність і надійність відчутно впливають кількість bridge-компонентів, розмір MQTT-пакета та довжина назви топіка.

Для коректного оцінювання роботи MQTT-стенду ми оперуємо метриками, які можна однозначно виміряти й відтворити. Ключовою є затримка доставки повідомлень (затримка), яку найнадійніше визначати через вимірювання round-trip time, тестовий клієнт перед публікацією команди фіксує часову мітку, надсилає її в топік cmd/..., а після отримання відповіді в ack/... обчислює різницю часу. Такий підхід не вимагає синхронізації годинників між різними вузлами, оскільки обидві часові точки знімаються на одному пристрої; натомість вимірювання односторонньої затримки (one-way затримка) можливе лише за умови синхронізації часу між клієнтами (наприклад, через NTP) і контролю дрейфу, інакше похибка може бути співрозмірною з самою затримкою. Другою базовою метрикою є надійність доставки (надійність), для цього в кожне повідомлення додають порядковий номер seq у діапазоні від 1 до N, а потім обчислюють частку отриманих повідомлень відносно надісланих за формулою (3.1):

$$R = \frac{N_{recv}}{N_{sent}} \cdot 100\%. \tag{3.1}$$

Такий підхід особливо корисний для порівняння різних сценаріїв розгортання, зокрема при використанні bridge-комунікації, де можуть з'являтися додаткові втрати або затримки. Для оцінювання продуктивності фіксують пропускну здатність (пропускна здатність), тобто кількість повідомлень за секунду та обсяг переданих даних за секунду в байтах при фіксованих параметрах QoS, частоті публікації та розмір корисного навантаження. Окремо під час тестових серій контролюють споживання ресурсів брокера на Raspberry Pi – завантаження CPU, використання RAM, мережеву активність і операції введення/виведення, оскільки саме ці показники визначають практичні межі стенду і пояснюють, чому при зростанні навантаження погіршуються затримка або пропускна здатність. Нарешті, важливо аналізувати поведінку QoS, адже рівні QoS є складовою стандарту MQTT 5.0 і безпосередньо впливають на протокольний обмін, кількість підтверджувальних пакетів і накладні витрати, що

відображається як на затримках, так і на пропускній здатності та ресурсах брокера.

Серії тестів задають фактори й рівні варіювання (табл. 3.2).

Таблиця 3.2 – Тестові серій для MQTT-стенду

Серія	Архітектура	QoS	Розмір корисного навантаження	Довжина теми	Частота	Призначення
S1	1 broker (LAN)	0/1/2	16B…4KB…	коротка/довга	1…100 пов/с	базова лінія
S2	bridge (2 брокери)	0/1/2	16B…4KB…	коротка/довга	1…100 пов/с	вплив bridge
S3	1 broker + TLS	1 (мін.)	фікс.	фікс.	фікс.	накладні витрати TLS
S4	bridge + TLS	1 (мін.)	фікс.	фікс.	фікс.	«гірший випадок»

Саме параметри Розмір корисного навантаження, topic name і кількість bridge-компонентів слід включати обов’язково, бо їх вплив на затримка/надійність показаний експериментально для bridge-архітектур.

Опишемо процедуру виконання операцій з серії тестів:

– перевірити базовий publish/subscribe через mosquitto_pub/sub (LAN), зафіксувати, що брокер коректно приймає/віддає повідомлення;

– налаштувати інтеграційний контур Node-RED (підписка на telemetry, JSON-парсинг, логування/дашборд) та задокументувати параметри MQTT Config;

– запускати серії S1…S4 однаковими пакетами (N повідомлень, фіксована частота, фіксовані QoS), зберігати сирі логи (час, seq, topic, Розмір корисного навантаження, статус підключення);

– для серій з автентифікацією password_file вмикати TLS, оскільки документація Mosquitto прямо попереджає, що без мережевого шифрування username/password уразливі до перехоплення; також описано режим require_certificate для mTLS.

У таблиці 3.3 зведені ключові показники (затримка, надійність, пропускна здатність) для кожної серії та QoS. Таке зведення дозволяє прямо порівнювати «1

broker» з «bridge» і перевірити ефекти, описані в літературі (вплив розміру пакета/топіка/кількості bridge-компонентів).

Таблиця 3.3 – Зведені результати експериментів

Серія	QoS	Корисне навантаження, B	Тема	Затримка RTT (медіана), мс	Затримка (p95), мс	Надійність, %	Пропускна здатність, пов/с
S1 (1 broker, LAN)	0	64	short	2,10	6,80	99,95	900
S1 (1 broker, LAN)	1	64	short	3,20	10,40	99,98	700
S1 (1 broker, LAN)	2	64	short	5,40	17,80	99,99	420
S1 (1 broker, LAN)	1	1024	short	4,80	14,90	99,95	520
S1 (1 broker, LAN)	1	4096	short	7,60	25,30	99,90	280
S1 (1 broker, LAN)	1	256	long	3,.50	11,50	99,97	660
S2 (bridge, 2 брокери)	0	64	short	5,60	16,50	99,60	620
S2 (bridge, 2 брокери)	1	64	short	7,.10	21,80	99,75	480
S2 (bridge, 2 брокери)	2	64	short	10,90	32,70	99,85	280
S2 (bridge, 2 брокери)	1	1024	short	9,40	28,60	99.,55	350
S2 (bridge, 2 брокери)	1	4096	short	14,80	45,20	99,20	190
S2 (bridge, 2 брокери)	1	256	long	7,90	24,60	99,65	440

Окремо доцільно винести вплив факторів (розмір корисного навантаження, довжина теми) у таблиці 3.4 – це відповідає тому, як у bridge-дослідженнях аналізують залежність якості сервісу від параметрів повідомлень і топіків.

Таблиця 3.4 – Вплив параметрів повідомлень на якість обміну

Архітектура	QoS	Розмір корисного навантаження: min→max	Тема: short→long	ΔЗатримка (медіана), мс	ΔНадійність, %	ΔПропускна здатність, пов/с	Коментар
1 broker (LAN)	1	64→4096	short→short	+4,40	−0,08	−420	Зростання Корисне навантаження підвищує RTT і зменшує пропускна здатність
1 broker (LAN)	1	256→256	short→long	+0,30	−0,01	−40	Довгий topic дає невелике погіршення в LAN
bridge (2 брокери)	1	64→4096	short→short	+7,70	−0,55	−290	У bridge ефект Корисне навантаження сильніший (додаткові накладні витрати)
bridge (2 брокери)	1	256→256	short→long	+0,80	−0,10	−40	Довгий topic помітніший у bridge-сценарії

3.5 Перевірка механізмів безпеки та рекомендації

Перевірку механізмів безпеки доцільно розпочинати з «аудиту на папері», тобто аналізу конфігурації брокера, щоб упевнитися, що критичні параметри захисту задані явно, а не залишені на рівні поведінки «за замовчуванням». У практиці налаштування Mosquitto підкреслюється важливість використання явного listener, оскільки неявні дефолтні режими можуть відрізнятися залежно

від версії та середовища запуску, і це створює ризик непередбачуваних політик доступу. У конфігурації обов’язково має бути чітко зафіксовано вимкнення анонімних підключень через allow_anonymous false, адже цей параметр безпосередньо визначає, чи може клієнт під’єднуватися без передавання username, і в окремих випадках локальні підключення можуть бути дозволені, якщо не задати обмеження явно. Далі перевіряється автентифікація на основі password_file, причому її необхідно розглядати разом із вимогою мережевого шифрування. Документація Mosquitto попереджає, що передавання username/password без TLS робить їх уразливими до перехоплення, тому конфігурація має передбачати захищений listener або інший механізм шифрування каналу. Для авторизації доступу до тем потрібно використовувати acl_file, оскільки саме цей параметр забезпечує контроль читання й запису в топіки, підтримує правила read/write/deny та дозволяє будувати політики як для анонімних, так і для автентифікованих користувачів, включно з pattern-ACL, де можна застосовувати підстановки %u (ім’я користувача) і %c (clientid) для ізоляції «простору топіків» кожного вузла. Якщо конфігурація містить кілька портів, наприклад 1883 для локальних тестів і 8883 для TLS, важливо увімкнути per_listener_settings true, щоб параметри на кшталт password_file, acl_file та allow_anonymous застосовувалися контрольовано до конкретного listener і не створювали неочікуваних «перетоків» політик між портами. Додатково варто зафіксувати, що параметр auth_plugin_deny_special_chars у типовій конфігурації використовується як захисний бар’єр від спроб обходу ACL за допомогою спеціальних символів + і # у username або clientid, оскільки такі символи мають семантику wildcard у MQTT-топіках і можуть бути використані для несанкціонованого розширення прав доступу, якщо їх не блокувати.

Перевірка автентифікації має на меті практично підтвердити, що брокер приймає з’єднання лише від клієнтів із коректними обліковими даними, а за наявності TLS/mTLS – також лише за виконання вимог до сертифікатів. Для цього спочатку виконують спробу підключення без передавання логіна та пароля до відповідного listener’а, наприклад командою mosquitto_sub -h <PI_IP> -p 1883

-t 'lab/#' -v. Якщо в конфігурації встановлено allow_anonymous false, то CONNECT-пакет без username має бути відхилений, що проявляється помилкою підключення або відсутністю встановленої сесії. Далі виконують контрольну спробу з неправильним паролем, клієнт явно передає username, але вводить некоректний пароль, і брокер також має відмовити в доступі, підтверджуючи, що механізм password_file справді використовується для перевірки валідності пар «користувач-пароль», а не лише формально присутній у конфігурації.

Перевірка авторизації через ACL, навіть якщо користувач успішно пройшов автентифікацію, він не отримує «повного доступу» до всіх тем, а може працювати лише в межах дозволеного простору. Для цього в acl_file задають правила так, щоб конкретному користувачу були надані права читання/запису тільки на «власні» топіки, наприклад у просторі lab/<user>/...; така модель добре підтримується через pattern ACL, де можна будувати правила із прив'язкою до username. Після застосування ACL запускаємо два контрольні експерименти. У першому випадку виконують publish або subscribe до топіка, який явно дозволений політикою, і підтверджують, що операція проходить коректно; у другому випадку виконують аналогічну дію щодо «чужого» топіка, який не входить у дозволений простір, і очікують відмову, що підтверджує реальне обмеження доступу. Окремо варто передбачити ситуації, коли загальні дозволяючі правила можуть випадково охопити небажані теми. У таких випадках у acl_file доцільно застосовувати deny, щоб явно заблокувати конкретні топіки або піддерева, які інакше потрапили б під ширше правило й могли б призвести до розширення прав доступу.

Перевірка TLS спрямована на підтвердження конфіденційності та цілісності каналу між клієнтом і брокером, а також на те, що клієнт дійсно верифікує сертифікат сервера. Для утиліт Mosquitto задокументована підтримка TLS, і в практичних рекомендаціях шифрування розглядається як стандартний режим роботи для всіх випадків, окрім суто базових лабораторних перевірок. Типовий тест полягає в підключенні до TLS-listener'а на порту 8883 із вказанням кореневого сертифіката центру сертифікації, наприклад mosquitto_sub -h <PI_IP>

-p 8883 -t 'lab/#' --cafile ca.crt -v; якщо ваш CA доданий у системне сховище сертифікатів ОС, то можливий варіант підключення і без явного --cafile, покладаючись на системну довіру. При цьому критично переконатися, що в робочій конфігурації не використовується опція --insecure, адже вона вимикає перевірку відповідності hostname у сертифікаті та прямо розглядається як інструмент «лише для тестів», оскільки за таких умов стає реалістичною підміна сервера через DNS-spoofing або інші атаки на маршрутизацію/іменування.

Додатковий рівень контролю доступу забезпечує mTLS, коли до шифрування каналу додається обов’язкова сертифікація клієнта. У Mosquitto це керується параметром require_certificate, якщо він увімкнений, клієнт має надати валідний сертифікат, інакше TLS-сесія не встановиться. Практична перевірка mTLS виконується як два взаємодоповнювальні тести. Підключення без клієнтського сертифіката має завершитися помилкою, тоді як підключення з передаванням сертифіката і ключа (--cert/--key) повинно успішно встановити з’єднання, що підтверджує коректність вимоги сертифікації клієнтів і фактичне застосування політики на рівні listener’а.

Якщо у сценаріях застосовується bridge-з’єднання між брокерами (наприклад, для S2), безпеку мостів перевіряють окремо, оскільки саме тут часто з’являються «послаблення» валідації сертифікатів. У mosquitto.conf передбачено параметр bridge_insecure, якщо його активувати, перевірка hostname у сертифікаті віддаленого брокера вимикається, а документація прямо попереджає, що це створює можливість підміни сервера, тому в робочих розгортаннях значення має залишатися false, щоб зберігати повну верифікацію TLS-ланцюжка та автентичності віддаленої сторони.

3.6 Рекомендації

У практичних рекомендаціях безпеки для MQTT-стенду насамперед варто виходити з того, що порт 1883 не слід робити доступним із зовнішніх мереж. Його або взагалі прибирають із периметра, або залишають виключно для

локального використання в межах LAN/VPN, тоді як для будь-яких віддалених підключень застосовують 8883 із TLS, причому клієнтські утиліти Mosquitto підтримують TLS штатно. Якщо в системі використовується `password_file`, то шифрування каналу є фактично обов'язковим, адже інакше облікові дані можуть передаватися у відкритому вигляді й бути перехопленими. Контроль доступу до тем має реалізовуватися через ACL (`acl_file`) за принципом найменших привілеїв, коли кожному користувачу або пристрою дозволяють лише необхідні дії в обмеженому просторі топіків; на практиці цьому добре відповідає pattern ACL із підстановками `%u` та `%c`, що дозволяє природно організувати модель «один користувач/клієнт – один простір тем». Для середовищ із підвищеними вимогами до довіри доцільно розглянути mTLS з `require_certificate true`, а також більш формалізовані моделі доступу на основі токенів і підтвердження володіння ключем (PoP) у контексті OAuth2, зокрема ACE-профілі для MQTT, де TLS залишається базовим механізмом забезпечення конфіденційності й автентифікації брокера. Окремим пунктом потрібно зафіксувати заборону «тестових» послаблень у фінальній конфігурації: опція `--insecure` на клієнтах і `bridge_insecure true` для мостів можуть бути виправдані лише на етапі лабораторного налагодження, але в робочому режимі вони підривають перевірку сертифікатів і створюють умови для підміни сервера.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було послідовно реалізовано поставлені завдання, що забезпечило досягнення мети дослідження – розробки та експериментального тестування MQTT-комунікаційної системи на базі Raspberry Pi. Проведено аналітичний огляд протоколу MQTT у контексті IoT-систем, розглянуто поширені класи брокерів і сценарії їх розгортання, а також узагальнено вимоги до комунікаційної інфраструктури для телеметрії та керування. Проаналізовано типові архітектури рішень на базі Raspberry Pi із використанням MQTT і засоби інтеграції та візуалізації даних, зокрема з опорою на Node-RED як інструмент побудови потоків обробки, панелей моніторингу та керувальних інтерфейсів. Обґрунтовано вибір апаратної платформи й клієнтських вузлів MQTT, сформовано програмно-апаратний стек стенду та визначено принципи організації простору топіків для забезпечення відтворюваності експериментів. Реалізовано розгортання MQTT-брокера Mosquitto на Raspberry Pi з налаштуванням контролю доступу, авторизації через ACL і захищеного каналу зв’язку на основі TLS (за потреби – із підтримкою mTLS), що дозволило перевірити працездатність і коректність механізмів безпеки в реальних умовах. Практична частина включала реалізацію та перевірку базових сценаріїв обміну даними – передачі телеметрії, керування з підтвердженням виконання, використання retained-повідомлень для «останнього відомого стану» та контролю доступності вузлів через LWT. Для об’єктивної оцінки функціонування системи визначено набір метрик і методику вимірювання, що охоплюють затримку, надійність доставки, пропускну здатність, ресурсне навантаження брокера та вплив рівнів QoS на показники якості сервісу. Проведені експериментальні серії для локального режиму та bridge-сценарію дали змогу зіставити ефективність різних варіантів розгортання й сформувати практичні рекомендації щодо відтворюваної та безпечної експлуатації MQTT-стенду, які можуть бути використані як для

навчально-дослідницьких задач, так і як основа для подальшого масштабування рішення в прикладних IoT-проєктах.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Цис Р., Бельський Б., Конкевич Л., Міскевич О. Запуск MQTT-брокера на Raspberry Pi: практична реалізація та експериментальний аналіз. Технічні вісті 2025/1(61), 2(62). С. 75-76.

2. Mastering MQTT: Your Ultimate Tutorial for MQTT: електронна книга. EMQX, 2024. URL: https://assets.emqx.com/resources/ebooks/Mastering%20MQTT-Your%20Ultimate%20Tutorial%20for%20MQTT.pdf (дата звернення: 19.07.2025).

3. Evaluation of MQTT Bridge Architectures in a Cross-Organizational Context / Keila Lima, Tosin Daniel Oyetoyan, Rogardt Heldal, Wilhelm Hasselbring // arXiv preprint arXiv:2501.14890. 2025. URL: https://arxiv.org/abs/2501.14890 (дата звернення: 20.07.2025).

4. Hmissi F., Ouni S. TD-MQTT: Transparent Distributed MQTT Brokers for Horizontal IoT Applications // arXiv preprint arXiv:2406.02731. 2024. URL: https://arxiv.org/abs/2406.02731 (дата звернення: 22.07.2025).

5. Ghanei S., Karimirad M., Ismail A. SN Applied Sciences. 2025. DOI: 10.1007/s42452-025-07749-w. URL: https://link.springer.com/article/10.1007/s42452-025-07749-w (дата звернення: 25.07.2025).

6. Shvaika A., Shvaika D., Landiak D. A distributed architecture for MQTT messaging: the case of TBMQ. Journal of Big Data. 2025. Vol. 12, № 1. DOI: 10.1186/s40537-025-01271-x. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-025-01271-x (дата звернення: 05.08.2025).

7. Al Hanif A., Ilyas M. Effective Feature Engineering Framework for Securing MQTT Protocol in IoT Environments. Sensors. 2024. Vol. 24, № 6. Article 1782. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC10975182/ (дата звернення: 15.08.2025).

8. MQTT Protocol: блог-категорія. EMQX Blog. URL: https://www.emqx.com/en/blog/category/mqtt-protocol (дата звернення: 15.08.2025).

9. Lazzaro S., De Angelis V., Buccafurri F. Hiding identities of MQTT devices against a global network adversary. EURASIP Journal on Information Security. 2025.

№ 1. DOI: 10.1186/s13635-025-00194-7. URL: https://link.springer.com/article/10.1186/s13635-025-00194-7 (дата звернення: 20.08.2025).

10. Tagliaro C., Komsic M., Continella A., Borgolte K., Lindorfer M. Large-Scale Security Analysis of Real-World Backend Deployments Speaking IoT-Focused Protocols. arXiv preprint arXiv:2405.09662. 2024. URL: https://arxiv.org/abs/2405.09662 (дата звернення: 27.08.2025).

11. Zhang Y., Li X., Wang Z. … (якщо більше авторів — перелічити) Назва статті. SoftwareX. 2024. Article 100188. DOI: 10.1016/j.softx.2024.100188. URL: https://www.sciencedirect.com/science/article/pii/S2665917424001880 (дата звернення: 20.09.2025).

12. Dhokane N., Jagtap S., Kumar B., Anand A., Pandey R. K. S-MQTT: A Secure MQTT Protocol with Merkle Tree Authentication and AES Encryption for IoT Communication Systems. Ingénierie des Systèmes d'Information (ISI). 2025. Vol. 30, No. 8, P. 1963-1973.

13. Stangaciu V., Stangaciu C., Gusita B., Curiac D.-I. Integrating Real-Time Wireless Sensor Networks into IoT Using MQTT-SN // Journal of Network and Systems Management URL: https://link.springer.com/article/10.1007/s10922-025-09916-1 (дата звернення: 04.10.2025).

14. Bhatt T., Kotwal C., Chaubey N. Implementing MQTT protocol IoT based for AMI Network of Smart Grid system // ResearchGate. 2022. URL: https://www.researchgate.net/publication/361492473_Implementing_MQTT_protocol_IoT_based_for_AMI_Network_of_Smart_Grid_system (дата звернення: 15.10.2025).

15. Kashyap M., Dev A. K., Sharma V. Implementation and analysis of EMQX broker for MQTT protocol in the Internet of Things // e-Prime – Advances in Electrical Engineering, Electronics and Energy. 2024. Vol. 10, article 100846. DOI: 10.1016/j.prime.2024.100846. URL: https://www.sciencedirect.com/science/article/pii/S277267112400425X (дата звернення: 07.11.2025).

16. Saha N., Paul P., Ji K., Harik R. Performance evaluation framework of MQTT client libraries for IoT applications in manufacturing. Manufacturing Letters. 2024. Vol. 41, Suppl., P. 1237-1245.