БАКАЛАВРСЬКА РОБОТА

КРБ.СІ-01.00.00.000 ПЗ

Група СІ-22-1

Олександр АНДРЕЙЧУК

2026

Міністерство освіти і науки України
Івано-Франківський національний технічний університет нафти і газу
інститут інформаційних технологій
кафедра інформаційно телекомунікаційних технологій та систем

УДК 004.724.2/.4:621.396.2:621.396.96

Андрейчук Олександр Володимирович

БАКАЛАВРСЬКА РОБОТА

Розроблення розподіленої по площі системи передачі повідомлень про дискретні події

Інформаційних технологій

151 - Автоматизація та
комп'ютерно-інтегровані технології

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня
_____ О.В. Андрейчук
(підпис, ініціали та прізвище здобувача)

Науковий керівник
_____д..т.н, проф., Ю.Й. Стрілецький
(підпис, прізвище, ім’я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
В.о. завідувача кафедри
_____ Заміховська О.Л.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*
Кафедра *Інформаційно- телекомунікаційних технологій систем*
Освітній рівень _бакалавр_____
Спеціальність __151 - Автоматизація та комп'ютерно-інтегровані технології_____

ЗАТВЕРДЖУЮ
В.о. Завідувач кафедри ІТТС д.т.н.,проф
О.Л.Заміховська

«____»_____2026 року

З А В Д А Н Н Я
НА БАКЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Андрейчук Олександр Володимирович
(прізвище, ім'я, по батькові)

1. Тема роботи *Розроблення розподіленої по площі системи передачі повідомлень про дискретні події*
керівник роботи, *проф. каф. ІТТС Стрілецький Ю.Й*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від " 07 " *квітня* 2026 року № 164/7
2. Строк подання студентом роботи
3. Вихідні дані до роботи час автономої роботи сенсора не менше 30 діб
дальність передачі від 5 км.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
Аналіз існуючих систем передачі даних для моніторингу периметру;
Розробка структури мережі та постановка задач;
Практична реалізація. особливості кожного блоку та розробка сервера
Висновки. Додатки:
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):
Загальна структура системи передачі повідомлень про дискретні події
Структурна схема сенсорного вузла
Структурна схема вузла репітера/концентратора
Структурна схема вузла сервера системи
Діаграма роботи програми сенсорного вузла
Діаграма роботи програми рептера
Діаграма роботи програми сервера системи
Вид http сторінки контролю стану системи

Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання видав
1	Стрілецький Ю.Й., проф.		
2	Стрілецький Ю.Й., проф.		
3	Стрілецький Ю.Й., проф.		

Дата видачі завдання 19.01 2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Термін виконання етапів роботи	Примітка
1	Аналіз існуючих систем передачі даних для моніторингу периметру;	05.05.2026-10.05.2026	Виконано
2	Розробка структури мережі та постановка задач;	11.05.2026-25.05.2026	Виконано
3	Практична реалізація. особливості кожного блоку та розробка сервера	25.05.2026-30.05.2026	Виконано
5	Оформлення роботи	11.06.2026-17.06.2026	Виконано

Студент _____ Андрейчук О.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Стрілецький Ю.Й
(підпис) (прізвище та ініціали

РЕФЕРАТ

Бакалаврська робота складається з 3 розділів, 8 рисунків, 5 таблиць, загальний обсяг роботи 76 сторінок, використано 15 літературних джерел.

Тема бакалаврської роботи: Розробка триступеневої ієрархічної кластерної мережі передачі даних для моніторингу периметру на базі технології LoRa.

Мета роботи: розробка триступеневої ієрархічної кластерної мережі для моніторингу периметру, що забезпечує дальність понад 3 км, автономність сенсорного вузла не менше 30 діб, масштабування до 320 точок контролю та надійність доставки повідомлень не нижче 95%.

Об'єкт дослідження: процеси передачі даних у бездротових сенсорних мережах для моніторингу периметру в польових умовах.

Предмет дослідження: архітектурні, протокольні та апаратні рішення для побудови триступеневої кластерної мережі на базі LoRa 433 МГц, що забезпечують надійну доставку повідомлень при низькому енергоспоживанні.

Результати бакалаврської роботи: Проведено огляд та аналіз існуючих систем передачі даних (точка-точка, естафетна передача, меш-мережі, зіркова топологія) для моніторингу периметру, виявлено їх переваги та недоліки. Обґрунтовано вибір ієрархічної кластерної топології на базі LoRa 433 МГц як оптимальної для польових тактичних застосувань. Сформульовано функціональні та нефункціональні вимоги до системи, розроблено частотно-канальне планування (канали 10…28 для кластерів, канал 2 для зв'язку з сервером) та власний мінімальний протокол обміну (MiniPacket, RepeaterPacket, AckPacket) із механізмами контрольної суми, підтвердження доставки та повторних спроб. Реалізовано апаратні модулі сенсорного вузла (Arduino Nano, LoRa-модуль AS32-TTL-100, доплерівський радар HB-100), репітера та центрального сервера на базі ESP32 із вбудованим веб-інтерфейсом. Розроблено механізм непрямого управління сенсорами через поле cmd у пакеті підтвердження. Проведено лабораторне та польове тестування, яке підтвердило дальність зв'язку до 3–4 км, надійність доставки 97% пакетів без повторних спроб, середній струм сенсора ≈4,1 мА при heartbeat 20 хв, розрахункову автономність ≈35 діб та максимальну затримку доставки 18 с, що відповідає поставленим вимогам.

Ключові слова: БЕЗДРОТОВІ СЕНСОРНІ МЕРЕЖІ, МОНІТОРИНГ ПЕРИМЕТРУ, LoRa, 433 МГц, ІЄРАРХІЧНА КЛАСТЕРНА ТОПОЛОГІЯ, ПРОТОКОЛ ПЕРЕДАЧІ ДАНИХ, ДОПЛЕРІВСЬКИЙ РАДАР, АВТОНОМНІСТЬ, ESP32, ВЕБ-ІНТЕРФЕЙС.

ABSTRACT

The bachelor's thesis consists of 3 chapters, 8 figures, 5 tables, total volume of 76 pages, 15 literature sources are used in the work.

Bachelor's thesis topic: Development of a three-level hierarchical cluster data transmission network for perimeter monitoring based on LoRa technology.

Objective of the work: development of a three-level hierarchical cluster network for perimeter monitoring that provides a range of more than 3 km, sensor node autonomy of at least 30 days, scalability up to 320 control points and message delivery reliability of at least 95%.

Object of research: data transmission processes in wireless sensor networks for perimeter monitoring in field conditions.

Subject of research: architectural, protocol and hardware solutions for building a three-level cluster network based on LoRa 433 MHz that ensure reliable message delivery with low power consumption.

Results of the bachelor's thesis: A review and analysis of existing data transmission systems (point-to-point, store-and-forward, mesh networks, star topology) for perimeter monitoring was conducted, their advantages and disadvantages were identified. The choice of hierarchical cluster topology based on LoRa 433 MHz as optimal for field tactical applications is substantiated. Functional and non-functional requirements for the system are formulated, frequency-channel planning (channels 10…28 for clusters, channel 2 for communication with the server) and an original minimal protocol (MiniPacket, RepeaterPacket, AckPacket) with checksum, delivery confirmation and retransmission mechanisms are developed. Hardware modules of the sensor node (Arduino Nano, LoRa module AS32-TTL-100, Doppler radar HB-100), repeater and central server based on ESP32 with built-in web interface are implemented. An indirect sensor control mechanism via the cmd field in the acknowledgment packet is developed. Laboratory and field testing confirmed a communication range of 3–4 km, delivery reliability of 97% of packets without retransmission attempts, average sensor current of ≈4.1 mA at a 20-minute heartbeat, estimated autonomy of ≈35 days and maximum delivery delay of 18 seconds, which meets the specified requirements.

Keywords: WIRELESS SENSOR NETWORKS, PERIMETER MONITORING, LoRa, 433 MHz, HIERARCHICAL CLUSTER TOPOLOGY, DATA TRANSMISSION PROTOCOL, DOPPLER RADAR, AUTONOMY, ESP32, WEB INTERFACE.

Зміст с.

ВСТУП 7
1. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ ПЕРЕДАЧІ ДАНИХ ДЛЯ МОНІТОРИНГУ ПЕРИМЕТРУ 9
1.1. Загальна характеристика задачі моніторингу периметру 9
1.2. Прості радіомережі точка-точка та їх застосування в охоронних системах.....10
1.3. Системи з естафетною (store-and-forward) передачею даних 12
1.4. Меш-мережі (Mesh Network).....15
1.5. Мережі із зірковою топологією та направленою ієрархічною передачею18
1.6. Технологія LoRa та протокол LoRaWAN 21
1.7. Огляд існуючих систем моніторингу периметру 23
1.8. Порівняльний аналіз топологій та обґрунтування вибору.....25
2. РОЗРОБКА СТРУКТУРИ МЕРЕЖІ ТА ПОСТАНОВКА ЗАДАЧ.....28
2.1. Формулювання вимог до системи.....28
2.2. Архітектура триступеневої мережі 30
2.3. Частотно-канальне планування.....34
2.4. Протокол обміну даними.....35
2.5. Задачі сенсорного вузла.....39
2.6. Задачі вузла-репітера 42
2.7. Задачі центрального сервера 45
2.8. Розробка протоколів передачі даних для сенсорних мереж 48
3. ПРАКТИЧНА РЕАЛІЗАЦІЯ. ОСОБЛИВОСТІ КОЖНОГО БЛОКУ ТА РОЗРОБКА СЕРВЕРА 52

					КРБ.СІ-01.00.00.000 ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розробив		Андрейчук			Розроблення розподіленої по площі системи передачі повідомлень про дискретні події	Літ.	Арк.	Акрушів
Перевірив		Стрілецький						5
						ІФНТУНГ		
Н.контр								
Затверд		Заміховський						

3.1. Апаратна реалізація сенсорного вузла 53
3.2. Програмна реалізація сенсорного вузла 56
3.3. Апаратна реалізація репітера 61
3.4. Програмна реалізація репітера 63
3.5. Апаратна реалізація центрального сервера 66
3.6. Програмна реалізація сервера 68
3.7. Веб-інтерфейс сервера 71
3.8. Конструкція корпусу сенсорного вузла 74
3.9. Налаштування та розгортання системи 76
3.10. Тестування системи 79
3.11. Можливості розширення системи 82
ВИСНОВКИ 85
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 87
ДОДАТКИ 89

ВСТУП

Актуальність теми. Тематика кваліфікаційної роботи бакалавра передбачає розробку системи моніторингу периметру з використанням бездротових сенсорних мереж та технології LoRa. Сучасні умови охорони об'єктів інфраструктури, державного кордону, військових та тактичних об'єктів вимагають автономних, енергоефективних і масштабованих засобів виявлення несанкціонованого перетину контрольованих зон. Існуючі системи (точка-точка, меш-мережі, LoRaWAN) мають обмеження за дальністю, енергоспоживанням, складністю розгортання або залежністю від стаціонарної інфраструктури. Одним із способів забезпечення надійного радіозв'язку у польових умовах є застосування ієрархічних кластерних топологій на базі LoRa у діапазоні 433 МГц, тому розробка власного протоколу передачі даних, апаратної реалізації сенсорів, репітерів та центрального сервера з веб-інтерфейсом є актуальною задачею.

Метою дослідження даної роботи є розробка триступеневої ієрархічної кластерної мережі для моніторингу периметру, що забезпечує дальність понад 3 км, автономність сенсорного вузла не менше 30 діб, масштабування до 320 точок контролю та надійність доставки повідомлень не нижче 95%.

Завданнями дослідження є: аналіз існуючих систем передачі даних (точка-точка, естафетна передача, меш-мережі, зіркова топологія) для моніторингу периметру та обґрунтування вибору ієрархічної кластерної топології; формулювання функціональних і нефункціональних вимог до системи; розробка частотно-канального планування, протоколу обміну даним (MiniPacket, RepeaterPacket, AckPacket) та механізмів підтвердження доставки і повторних передач; апаратна та програмна реалізація сенсорного вузла (Arduino Nano + доплерівський радар HB-100 + LoRa-модуль AS32-TTL-100), репітера (Arduino Nano + LoRa) та центрального сервера на базі

ESP32 із вбудованим веб-інтерфейсом;тестування системи в лабораторних та польових умовах для оцінки дальності, затримки, Инадійності та енергоспоживання.

Об’єктом дослідження є процеси передачі даних у бездротових сенсорних мережах для моніторингу периметру в польових умовах.

Предметом дослідження є архітектурні, протокольні та апаратні рішення для побудови триступеневої кластерної мережі на базі LoRa 433 МГц, що забезпечують надійну доставку повідомлень при низькому енергоспоживанні.

Практичне значення одержаних результатів полягає в розробці діючого зразка системи моніторингу периметру, який може бути використаний для охорони об'єктів інфраструктури, тактичного спостереження, контролю державного кордону та інших застосувань, де потрібне швидке розгортання автономних сенсорів, стійкість до перешкод і централізоване відображення інформації через веб-браузер без спеціального ПЗ. Розроблені протокольні рішення (бітова маска черги, механізм непрямого управління сенсорами, гібридний store-and-forward) можуть бути використані в інших системах IoT із обмеженими ресурсами.

1 АНАЛІЗ ІСНУЮЧИХ СИСТЕМ ПЕРЕДАЧІ ДАНИХ ДЛЯ МОНІТОРИНГУ ПЕРИМЕТРУ

1.1 Загальна характеристика задачі моніторингу периметру

Моніторинг периметру — це комплекс заходів із виявлення несанкціонованого перетину або присутності у визначеній зоні контролю. Задача є актуальною для охорони об'єктів інфраструктури, контролю державного кордону, охорони військових об'єктів, а також для організації тактичного спостереження у польових умовах [1, 2].

Базова вимога до будь-якої системи моніторингу периметру — своєчасна та достовірна передача інформації про подію від точки виявлення до пункту обробки. Ця вимога визначає центральну роль підсистеми передачі даних у загальній архітектурі системи.

У польових умовах, особливо під час ведення бойових дій або охорони розосереджених об'єктів, підсистема зв'язку повинна відповідати ряду специфічних вимог. По-перше, вона має забезпечувати покриття значних відстаней без стаціонарної інфраструктури. По-друге, вузли мережі мають функціонувати автономно від зовнішнього живлення протягом тривалого часу — від кількох тижнів до місяців. По-третє, система повинна зберігати працездатність в умовах перешкод, вологи, перепадів температур. По-четверте, розгортання мережі має виконуватись без спеціалізованого обладнання та складних налаштувань.

Крім того, система має забезпечувати надійну доставку повідомлень навіть за умов нестабільного каналу зв'язку, підтримувати масштабування кількості точок контролю без істотного ускладнення архітектури, а також мати централізований пункт збору та відображення інформації для оператора.

Перераховані вимоги визначають вибір технологічної бази. Системи, що використовують стільниковий зв'язок, залежать від наявності базових станцій і є вразливими при руйнуванні інфраструктури. Системи на основі Wi-Fi мають обмежений радіус дії та вимагають значних енергоресурсів. Традиційні УКХ-радіостанції споживають багато енергії та потребують ручного обслуговування. Технології класу LPWAN (Low Power Wide Area Network), зокрема LoRa, задовольняють вимоги по дальності, енергоспоживанню та автономності, що робить їх перспективними для застосування у системах польового моніторингу [3, 4, 5].

Для вибору раціональної архітектури мережі передачі даних необхідно провести аналіз існуючих підходів: простих радіомереж, систем із естафетною передачею, меш-мереж та мереж із направленою топологією. Результати аналізу дозволять обґрунтувати вибір архітектурного рішення для розроблюваної системи.

1.2 Прості радіомережі точка-точка та їх застосування в охоронних системах

1.2.1 Принцип роботи

Найбільш проста архітектура радіомережі — це зв'язок типу «точка-точка» (point-to-point). У такій схемі один пристрій є передавачем (або трансивером), інший — приймачем. Обмін даними відбувається безпосередньо між двома вузлами без участі будь-яких проміжних елементів.

У класичному варіанті охоронна система на базі радіозв'язку точка-точка складається з датчика або групи датчиків, що передають сигнал на центральний пункт прийому. Датчики, як правило, оснащені простим радіомодулем, який при спрацюванні формує короткий пакет даних і надсилає його на фіксовану адресу приймача. Приймач знаходиться на пункті контролю та відображає отриману інформацію.

Така схема реалізується на будь-якому доступному частотному діапазоні: 433 МГц, 868 МГц, 915 МГц (промислові ISM-діапазони), а також на УКХ (VHF, 136–174 МГц) і ДМХ (UHF, 400–520 МГц) діапазонах [6,7]. Вибір частоти впливає на дальність зв'язку, розмір антени та умови поширення сигналу.

1.2.2 Технічні характеристики

Дальність зв'язку у системах точка-точка при використанні модулів потужністю 20–100 мВт у діапазоні 433–868 МГц на відкритій місцевості становить від 500 м до 5 км залежно від висоти антен, рельєфу та умов поширення. Застосування спрямованих антен дозволяє збільшити дальність у кілька разів, але ускладнює монтаж та вимагає точного орієнтування.

Швидкість передачі даних у таких системах, як правило, невелика — від одиниць до кількох сотень кілобіт за секунду. Для передачі коротких пакетів стану датчиків (10–20 байт) цього повністю достатньо. Затримка доставки повідомлення від моменту спрацювання датчика до приймача не перевищує кількох сотень мілісекунд, що є прийнятним для систем охорони.

Енергоспоживання радіомодуля у режимі передачі становить 50–120 мА при напрузі живлення 3,3–5,0 В. У режимі прийому та очікування споживання значно менше — 1–10 мА. При використанні режимів сну мікроконтролера та активації модуля лише на час передачі середнє споживання вузла може бути знижено до одиниць мікроампер на фоні споживання датчика та схеми пробудження [8].

1.2.3 Обмеження схеми точка-точка

Головне обмеження схеми точка-точка — відсутність масштабованості. При збільшенні кількості точок контролю кожна додаткова точка потребує або власного частотного каналу (що обмежено регуляторними нормами), або поділу каналу в часі із збільшенням затримок і зниженням надійності.

Друге обмеження — вразливість до рельєфу. Пряма видимість між точкою і приймачем не завжди забезпечена: ліс, будівлі, складки місцевості знижують рівень сигналу або повністю блокують зв'язок. Збільшення потужності передавача підвищує енергоспоживання та виявляє позицію вузла за електромагнітним випромінюванням.

Третє обмеження — відсутність підтвердження доставки у найпростіших реалізаціях. Якщо пакет не дійшов через перешкоду або тимчасові завади, інформація про подію може бути втрачена. Додавання механізму підтвердження ускладнює протокол і вимагає двостороннього зв'язку.

Незважаючи на зазначені обмеження, схема точка-точка використовується там, де кількість точок контролю мала (1–5), відстані фіксовані та заздалегідь перевірені, а вимоги до масштабування відсутні. Прикладами є прості периметрові сигналізації для охорони невеликих об'єктів.

1.3 Системи з естафетною (store-and-forward) передачею даних

1.3.1 Концепція естафетної передачі

Естафетна передача даних (store-and-forward) — принцип, при якому проміжний вузол мережі отримує повідомлення, зберігає його у своїй пам'яті, а потім у відповідний момент передає далі по ланцюжку. На відміну від ретрансляції у режимі реального часу (cut-through або store-and-forward із мінімальною затримкою), класична естафетна передача допускає значні затримки між отриманням і відправкою пакету.

Термін «естафетна мережа» (delay-tolerant network, DTN) використовується у телекомунікаціях для опису мереж, де безперервне з'єднання між джерелом і приймачем не гарантоване. Вузли з'являються та

зникають, канали зв'язку переривчасті, і доставка повідомлення може займати значний час — від хвилин до годин і навіть діб [9].

Класичним прикладом DTN є мережі міжпланетного зв'язку, де затримки поширення сигналу сягають десятків хвилин. У наземних застосуваннях естафетна передача застосовується у мережах сенсорів у важкодоступних районах, де вузли зв'язку з'являються епізодично (наприклад, супутниковий зв'язок раз на добу або прохід мобільного ретранслятора).

1.3.2 Застосування в охоронних системах

У охоронних системах польового типу принцип естафетної передачі може реалізовуватись по-різному. Один варіант — ланцюжок ретрансляторів, де кожен вузол приймає пакет від попереднього і передає наступному. Такий ланцюжок дозволяє охопити відстані, що перевищують можливості одного радіоканалу, але вимагає лінійного розміщення вузлів і є вразливим до виходу з ладу будь-якої ланки.

Інший варіант — кур'єрський вузол (data mule), який фізично переміщується між точками збору даних і пунктом обробки. Датчики зберігають інформацію у своїй пам'яті, а при наближенні кур'єрського вузла передають накопичені дані. Цей підхід практично не потребує радіозв'язку між стаціонарними вузлами, але не забезпечує сповіщення в реальному часі.

Третій варіант — гібридна схема, де основний канал зв'язку забезпечується радіолінком, а при його відсутності дані накопичуються у буфері та передаються при відновленні зв'язку. Саме цей підхід застосовується у сучасних системах на базі LoRa: репітер накопичує пакети від сенсорів у бітовій масці та передає їх на сервер у визначені інтервали часу, реалізуючи тим самим функцію буферизації з елементами store-and-forward [10].

1.3.3 Переваги та недоліки

Перевага естафетної передачі — стійкість до переривань каналу. Якщо зв'язок між вузлом і центром тимчасово відсутній, дані не губляться, а зберігаються у буфері до відновлення з'єднання. Це критично важливо у польових умовах, де якість радіоканалу може змінюватись через переміщення особового складу, зміну метеоумов або використання засобів радіоелектронної боротьби.

Недолік — збільшена затримка доставки. Якщо репітер передає дані на сервер з інтервалом, наприклад, 5 секунд, то максимальна затримка від спрацювання датчика до відображення на сервері становитиме суму часу передачі від датчика до репітера і часу очікування у черзі репітера. Для більшості тактичних задач затримка у кілька секунд є прийнятною.

Другий недолік — ускладнення логіки обробки. Вузол повинен підтримувати буфер, механізм черги та логіку повторних спроб. Це збільшує обсяг програмного коду та вимагає більшого обсягу оперативної пам'яті мікроконтролера. При використанні бітової маски як буфера (32 біти для 32 сенсорів) ці витрати мінімізуються. Детальне порівняння характеристик базових схем передачі даних наведено в таблиці 1.1.

Таблиця 1.1 — Порівняння базових схем передачі даних

Характеристика	Точка-точка	Естафетна передача	Ланцюжок ретрансляторів
Затримка доставки	< 1 с	1–30 с	< 1 с
Стійкість до обриву	Низька	Висока	Низька
Масштабованість	Низька	Середня	Низька
Складність реалізації	Низька	Середня	Середня
Вимоги до топології	Пряма видимість	Довільна	Лінійна

1.4 Меш-мережі (Mesh Network)

1.4.1 Архітектура та принцип роботи

Меш-мережа (від англ. mesh — сітка) — топологія, у якій кожен вузол з'єднаний з одним або кількома сусідніми вузлами, утворюючи мережу із множиною можливих маршрутів між будь-якою парою пристроїв. Відмінна риса меш-топології — можливість самовідновлення: при виході з ладу одного вузла або обриві каналу дані можуть бути перенаправлені по альтернативному маршруту.

У повній меш-топології (full mesh) кожен вузол з'єднаний з усіма іншими безпосередньо. На практиці це реалізується лише для малої кількості вузлів через квадратичне зростання кількості з'єднань. Для більшості застосувань використовується часткова меш-топологія (partial mesh), де кожен вузол з'єднаний із кількома найближчими сусідами.

Маршрутизація у меш-мережі вирішується за допомогою спеціалізованих протоколів. Для бездротових сенсорних мереж розроблено ряд протоколів, зокрема AODV (Ad hoc On-Demand Distance Vector), OLSR (Optimized Link State Routing), RPL (Routing Protocol for Low-Power and Lossy Networks, RFC 6550) [7,11]. Кожен протокол має свої характеристики затримки, обсягу службового трафіку та вимог до обчислювальних ресурсів вузлів.

1.4.2 Меш-мережі на базі LoRa

LoRa як технологія модуляції використовує метод чирп-розширення спектра (Chirp Spread Spectrum, CSS), який забезпечує зв'язок на відстанях до 10–15 км у відкритому просторі при потужності передавача 100 мВт. Протокол LoRaWAN, що зазвичай використовується разом з LoRa, орієнтований на зіркову топологію із централізованими шлюзами і не підтримує меш-мережу нативно.

Для побудови меш-мережі на базі LoRa існують альтернативні рішення. Бібліотека RadioHead реалізує маршрутизацію на рівні прикладного програмного забезпечення для Arduino-сумісних платформ. Протокол Meshtastic, реалізований для мікроконтролерів ESP32 та nRF52840 з LoRa-модулями [13], забезпечує децентралізований обмін повідомленнями у меш-мережі із шифруванням.

Meshtastic використовує протокол затоплення (flooding) із обмеженням кількості ретрансляцій (hop limit) для розповсюдження пакетів у мережі. Кожен вузол при отриманні пакету перевіряє, чи не ретранслював він цей пакет раніше (за унікальним ідентифікатором), і якщо ні — передає пакет далі зі зменшеним лічильником кроків. При досягненні лічильником нуля ретрансляція припиняється. Такий підхід простий у реалізації, але генерує значний обсяг дублюючого трафіку.

1.4.3 Переваги меш-мереж

Основна перевага меш-топології — відмовостійкість. Вихід з ладу одного або кількох вузлів не призводить до повної втрати зв'язку, якщо мережа залишається зв'язною. Ця властивість важлива для тактичних застосувань, де окремі вузли можуть бути виведені з ладу.

Друга перевага — гнучкість розгортання. Вузли не потребують строгого лінійного або ієрархічного розміщення, достатньо забезпечити хоча б один маршрут між кожною точкою та пунктом збору даних. Це спрощує планування при нерівномірному розподілі точок контролю.

Третя перевага — можливість розширення зони покриття без прокладання додаткових ліній зв'язку. Кожен новий вузол автоматично збільшує покриття мережі та може слугувати ретранслятором для існуючих вузлів, що знаходяться поза прямою видимістю від пункту збору даних.

1.4.4 Недоліки меш-мереж

Головний недолік меш-мережі — складність маршрутизації. Протоколи маршрутизації вимагають обміну службовими пакетами між вузлами для побудови та підтримки таблиць маршрутів. Цей службовий трафік займає час ефіру, збільшує споживання енергії та може призводити до колізій при великій кількості вузлів.

Для LoRa-мереж ця проблема загострюється через повільну швидкість передачі. При налаштуванні модуляції на максимальну дальність (Spreading Factor 12, смуга 125 кГц) швидкість передачі становить близько 250 біт/с, а тривалість одного пакету може перевищувати 1–2 секунди. У меш-мережі з 20–30 вузлами обмін службовими пакетами буде займати значну частину доступного часу ефіру.

Другий недолік — непередбачувані затримки та конфлікти. У меш-мережах із протоколом затоплення один пакет може бути ретрансльований кілька разів різними вузлами, що призводить до колізій та необхідності повторних передач. Час доставки пакету від джерела до приймача залежить від поточного стану мережі і не є детермінованим.

Третій недолік — вищі вимоги до ресурсів вузлів. Реалізація стека маршрутизації вимагає більшого обсягу flash-пам'яті та оперативної пам'яті мікроконтролера. Для малопотужних вузлів на базі ATmega328 (Arduino Nano/Mini, 2 КБ RAM, 32 КБ Flash) це є суттєвим обмеженням. Більш потужні платформи (ESP32, nRF52) вирішують цю проблему, але коштують дорожче та споживають більше енергії.

Четвертий недолік стосується безпеки. Відкрита меш-архітектура потенційно дозволяє зловмиснику ввести у мережу власний вузол, що буде ретранслювати модифіковані пакети або блокувати передачу. Захист від таких атак вимагає шифрування та автентифікації на рівні протоколу, що ще більше збільшує обчислювальне навантаження та обсяг коду.

1.5 Мережі із зірковою топологією та направленою ієрархічною передачею

1.5.1 Зіркова топологія

Зіркова топологія передбачає наявність центрального вузла (концентратора або шлюзу), з яким безпосередньо з'єднані всі периферійні вузли. Периферійні вузли не з'єднані між собою безпосередньо — весь обмін даними проходить через центральний вузол.

Стандарт LoRaWAN реалізує саме зіркову топологію [14]. Кінцеві пристрої (end devices) передають дані на шлюзи (gateways), шлюзи пересилають дані на мережевий сервер (network server) по дротових або мобільних каналах. Мережевий сервер дедублікує пакети (один пакет може прийняти кілька шлюзів), виконує їх обробку та передає на сервер застосунку (application server).

Перевага зіркової топології — проста архітектура без маршрутизації. Кожен периферійний вузол знає лише одну адресу — центральний вузол. Відсутня необхідність у підтримці таблиць маршрутів або обміні службовими пакетами між вузлами. Затримка доставки мінімальна і визначається лише часом передачі одного пакету.

Недолік зіркової топології — обмеження дальності. Відстань від кожного периферійного вузла до центрального не може перевищувати дальності прямого зв'язку. На відкритій місцевості для LoRa 433 МГц з потужністю 100 мВт це близько 3–5 км. [4,5].При нерівному рельєфі або наявності перешкод дальність може бути значно меншою.

1.5.2 Ієрархічна (кластерна) топологія

Ієрархічна або кластерна топологія є розвитком зіркової топології і вирішує проблему обмеження дальності [1, 9]. У такій мережі вузли об'єднуються у кластери. Усередині кожного кластера є один координатор

(cluster head або репітер), який з'єднаний із рядовими вузлами кластера безпосередньо за схемою «зірка». Координатори кластерів, у свою чергу, з'єднані із центральним сервером.

Така дворівнева ієрархія дозволяє збільшити загальне покриття мережі без збільшення потужності передавачів периферійних вузлів. Периферійний вузол (сенсор) спілкується лише з репітером свого кластера, якому достатньо передати сигнал на відстань у межах кластера. Репітер, у свою чергу, передає дані на центральний сервер по окремому каналу, розрахованому на більшу відстань або більш зручні умови поширення.

Кластерна топологія підтримує масштабування двома способами: збільшенням кількості вузлів у кластері та збільшенням кількості кластерів. При фіксованій кількості каналів зв'язку максимальна кількість вузлів обмежена добутком кількості кластерів на кількість вузлів у кластері.

1.5.3 Особливості кластерних мереж на базі LoRa 433 МГц

Частотний діапазон 433 МГц має ряд характеристик, що визначають особливості побудови мереж на цій основі. Довжина хвилі на частоті 433 МГц становить близько 69 см, що визначає розміри антен та умови поширення сигналу.

Сигнал у діапазоні 433 МГц краще, ніж у більш високочастотних діапазонах, проходить через перешкоди — листяний покрив, дерев'яні конструкції, стіни будівель. Це пояснюється меншим поглинанням та більшою довжиною хвилі, яка огинає предмети розміром менше довжини хвилі. Для застосувань у лісових або напівлісових умовах 433 МГц є кращим вибором порівняно з 868 МГц або 2,4 ГГц.

Регуляторний статус діапазону 433 МГц в Україні — ліцензований ISM-діапазон для малопотужних пристроїв. Максимальна ефективна випромінювана потужність для пристроїв ближнього радіуса дії (SRD) обмежена на рівні 10 мВт (10 dBm) у більшості застосувань [15], хоча для

деяких застосувань допустима потужність до 100 мВт (20 dBm). Для систем, що розглядаються у цій роботі, використовується модуль AS32-TTL-100 з потужністю 100 мВт, що відповідає вимогам для спеціалізованих застосувань.

Повільна швидкість передачі LoRa (від 0,3 до 50 кбіт/с залежно від налаштувань) є обмеженням лише для додатків, що передають великі обсяги даних [10]. Для систем моніторингу периметру, де пакет даних містить лише кілька байтів (адреса, лічильник, напруга, контрольна сума), повільна швидкість передачі дозволяє збільшити дальність зв'язку і не є критичним обмеженням.

1.5.4 Розподіл каналів у кластерній мережі

Для запобігання взаємним завадам між кластерами кожний кластер використовує окремий частотний канал для зв'язку між сенсорами та репітером. Зв'язок між репітерами та центральним сервером відбувається по єдиному загальному каналу або по окремих каналах для кожного репітера.

Вибір каналів визначається доступним частотним ресурсом та налаштуваннями модуля. Для модулів типу AS32-TTL-100 можна налаштувати частотний канал у діапазоні від 410 до 441 МГц з кроком 1 МГц. Це дозволяє мати до 32 незалежних каналів для кластерів.

Практичне обмеження визначається не частотним ресурсом, а ємністю сервера та організацією опитування каналів. Якщо сервер обслуговує 10 каналів, по кожному з яких можна підключити до 32 сенсорів, загальна ємність системи становить 320 сенсорних вузлів. Такий масштаб є достатнім для охорони периметру значної площі або протяжного маршруту.

1.6 Технологія LoRa та протокол LoRaWAN

1.6.1 Принцип модуляції LoRa

LoRa (Long Range) — запатентована технологія радіомодуляції компанії Semtech, що базується на методі чирп-розширення спектра (Chirp Spread Spectrum, CSS) [12]. Метод CSS походить від радіолокаційних технологій і характеризується підвищеною стійкістю до перешкод, частотних зсувів та ефекту Доплера.

У CSS сигнал кодується за допомогою чирпів — сигналів, частота яких лінійно зростає або зменшується протягом певного часу. Один «вгору»-чирп (upchirp) або «вниз»-чирп (downchirp) являє собою один символ. Кількість можливих символів залежить від коефіцієнта розширення спектра (Spreading Factor, SF): при SF=7 можливо 2^7 = 128 різних символів, при SF=12 — 2^12 = 4096 символів.

Збільшення SF підвищує стійкість сигналу до шуму (бюджет радіоканалу збільшується на 2,5 dB при збільшенні SF на 1), але знижує швидкість передачі приблизно у два рази і збільшує тривалість пакету [12]. Оператор системи вибирає SF залежно від вимог до дальності та часу передачі.

Окрім SF, на характеристики зв'язку впливають смуга частот (BW: 125, 250 або 500 кГц) та кодова швидкість (CR: 4/5, 4/6, 4/7 або 4/8). Зменшення смуги підвищує чутливість приймача (на 3 dB при вдвічі вужчій смузі), але знижує швидкість передачі. Збільшення кодової швидкості підвищує стійкість до помилок, але збільшує надлишковість.

1.6.2 Протокол LoRaWAN

LoRaWAN — протокол доступу до середовища та мережевий протокол, розроблений організацією LoRa Alliance поверх фізичного рівня LoRa. Архітектура LoRaWAN передбачає кінцеві пристрої (end devices), шлюзи

(gateways), мережевий сервер (network server) та сервери застосунків (application servers).

Кінцеві пристрої у LoRaWAN поділяються на три класи [14]. Клас A — базовий: пристрій відкриває вікно прийому лише після власної передачі, що мінімізує споживання енергії. Клас B — з синхронізованими вікнами прийому за маяком від шлюзу, що дозволяє регулярний прийом команд. Клас C — постійне прийомне вікно, максимальна доступність для команд але значне споживання.

Для систем охорони периметру повне дотримання стандарту LoRaWAN не є обов'язковим [2]. Стандарт орієнтований на публічні мережі IoT із централізованою інфраструктурою оператора. Для приватних мереж, де всі компоненти перебувають під контролем розробника, більш раціональним є використання фізичного рівня LoRa із власним протоколом обміну, що оптимізований під конкретні задачі. Такий підхід застосований у системі, що розглядається у цій роботі.

1.6.3 Модулі на базі LoRa для вбудованих систем

Для практичної реалізації систем на базі LoRa широко використовуються готові радіомодулі з інтерфейсом UART або SPI. Модулі серії E32 виробництва EBYTE (Chengdu Ebyte Electronic Technology) на чипах SX1276/SX1278 компанії Semtech отримали широке поширення завдяки простому інтерфейсу та доступності [6].

Модуль AS32-TTL-100 (аналог E32-433T20D) працює у діапазоні 410–441 МГц, має максимальну потужність передавача 20 dBm (100 мВт), чутливість приймача -147 dBm. Управління налаштуваннями відбувається через UART-інтерфейс за допомогою AT-команд або через виводи M0/M1 для зміни режиму роботи. Модуль підтримує прозорий режим передачі (transparent mode) та режим з адресацією (addressed mode), що дозволяє адресувати конкретний вузол у мережі.

Дальність зв'язку модуля на відкритій місцевості при використанні диполів зазначається виробником до 5 км при налаштуванні на максимальну потужність та оптимальні параметри модуляції. Практично досяжна дальність у реальних умовах (ліс, пагорби, перешкоди) становить 1–3 км, що є достатнім для формування кластерів у системі моніторингу периметру.

1.7 Огляд існуючих систем моніторингу периметру

1.7.1 Комерційні системи

На ринку існує ряд комерційних систем охорони периметру для стаціонарних об'єктів: DataKom, Crow, Paradox, Bosch Intrusion [2, 11]. Ці системи, як правило, використовують дротові шини або радіоканали у діапазоні 433/868 МГц для зв'язку між датчиками та центральним пультом. Вони орієнтовані на охорону будівель та огороджених периметрів і не розраховані на розгортання у польових умовах.

Для польового застосування існують спеціалізовані системи, зокрема Perimeter Products серії ПС (виробництво різних компаній), системи на базі сейсмічних або оптичних датчиків із радіоканалом передачі. Однак більшість таких систем є закритими (proprietary) та не дозволяють модифікацію або розширення функціональності.

Серед відкритих (open hardware/open software) рішень виділяється напрямок систем на базі мікроконтролерних платформ Arduino та ESP32 з LoRa-модулями. Ці рішення дозволяють адаптувати архітектуру, протокол та функціональність під конкретні вимоги, що є важливим для систем тактичного застосування.

1.7.2 Тактичні системи виявлення

У тактичних застосуваннях використовуються системи типу REMBASS (Remotely Monitored Battlefield Sensor System) та її сучасні аналоги. Такі системи складаються з сейсмічних, акустичних або інфрачервоних датчиків

із радіоканалом передачі на портативний або транспортний пункт прийому. Особливість тактичних систем — максимальна мініатюризація та маскування датчиків, низьке енергоспоживання та стійкість до перешкод. Зовнішній вигляд комплекту системи REMBASS наведено на (рис.1.1).

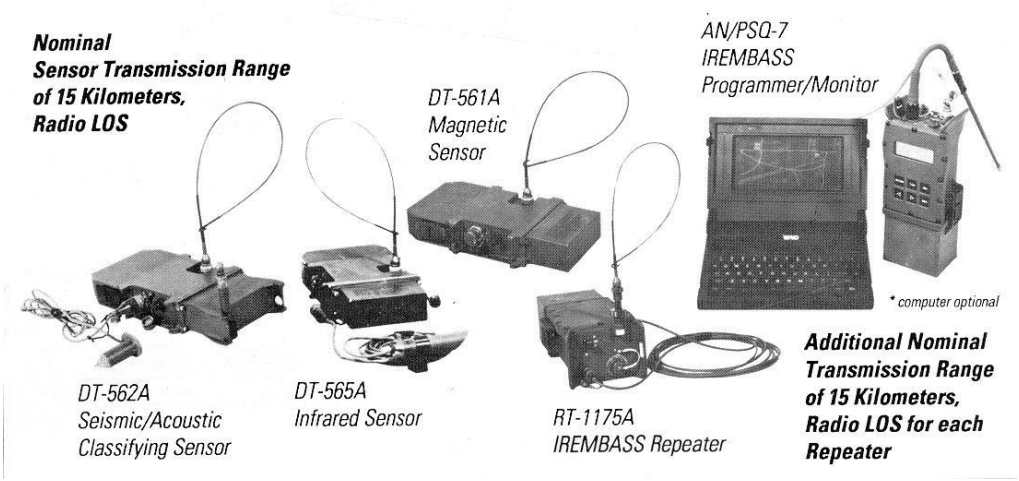


Рисунок 1.1– Комплект системи REMBASS [16]

Відомі публічно доступні розробки: система Ground Sensor на базі Arduino + LoRa, розроблена аматорськими спільнотами для тактичних ігор (airsoft, paintball), адаптована для реального застосування. Система Meshtastic, незважаючи на орієнтацію на передачу текстових повідомлень, також використовується для моніторингу переміщень завдяки підтримці GPS-координат та груп [13].

Система, що розробляється у цій роботі, відрізняється від зазначених вище рядом особливостей. По-перше, застосовується доплерівський радар як датчик виявлення [2], що забезпечує стійкість до хибних спрацювань від тварин та реєструє саме рух людини або транспортного засобу. По-друге, ієрархічна кластерна архітектура дозволяє масштабувати систему до 320 точок без ускладнення топології. По-третє, підтримка веб-інтерфейсу на сервері забезпечує відображення стану всіх вузлів на будь-якому пристрої зі звичайним браузером без встановлення спеціального програмного забезпечення.

1.8 Порівняльний аналіз топологій та обґрунтування вибору

1.8.1 Критерії порівняння

Для порівняння розглянутих архітектур визначено такі критерії, що відповідають вимогам до системи моніторингу польового периметру [6, 7]:

Дальність покриття — максимальна відстань від найвіддаленішої точки контролю до пункту збору даних. Для розроблюваної системи вимога — не менше 3 км від сенсора до репітера і до 10 км від репітера до сервера.

Масштабованість — кількість одночасно підтримуваних точок контролю. Вимога — не менше 64 точок при можливості розширення до 320.

Автономність — тривалість роботи від автономного живлення без обслуговування. Вимога — не менше 30 діб.

Надійність доставки — відсоток успішно доставлених повідомлень у нормальних умовах. Вимога — не менше 95%.

Затримка доставки — час від спрацювання датчика до відображення на сервері. Вимога — не більше 30 секунд.

Складність розгортання — кількість операцій та необхідна кваліфікація персоналу для введення системи в дію. Вимога — розгортання силами двох осіб без спеціальних інструментів.

Вартість вузла — собівартість одного сенсорного вузла. Вимога — мінімально можлива при виконанні технічних вимог. Надійність доставки — відсоток успішно доставлених повідомлень у нормальних умовах. Вимога — не менше 95%.

Затримка доставки — час від спрацювання датчика до відображення на сервері. Вимога — не більше 30 секунд.

Складність розгортання — кількість операцій та необхідна кваліфікація персоналу для введення системи в дію. Вимога — розгортання силами двох осіб без спеціальних інструментів.

Вартість вузла — собівартість одного сенсорного вузла. Вимога — мінімально можлива при виконанні технічних вимог. Результати порівняльного аналізу розглянутих топологій мереж передачі даних за цими критеріями наведено в таблиці 1.2.

Таблиця 1.2 — Порівняльний аналіз топологій мереж передачі даних

Критерій	**Точка-точка**	**Меш**	**Зірка (LoRaWAN)**	**Ієрархічна кластерна**
Дальність покриття	До 5 км	До 10 км	До 10 км	До 15 км
Масштабованість	Низька (1–5)	Середня (10–50)	Висока (1000+)	Висока (320+)
Автономність вузла	Висока	Середня	Висока	Висока
Надійність доставки	Низька	Висока	Висока	Висока
Затримка доставки	< 1 с	1–10 с	< 5 с	1–10 с
Складність розгортання	Низька	Середня	Висока	Низька
Вартість вузла	Низька	Середня	Низька	Низька
Потреба в інфраструктурі	Ні	Ні	Так (шлюзи)	Ні

1.8.2 Обґрунтування вибору ієрархічної кластерної топології

За сукупністю критеріїв ієрархічна кластерна топологія на базі LoRa 433 МГц є найбільш прийнятною для розроблюваної системи моніторингу периметру.

Порівняно зі схемою точка-точка, кластерна топологія забезпечує масштабованість до сотень вузлів та підвищену надійність завдяки буферизації у репітері. Периферійний сенсор не зобов'язаний мати пряму видимість до сервера — достатньо зв'язку з репітером свого кластера.

Порівняно з меш-мережею, кластерна топологія має детерміновані маршрути передачі, що спрощує відлагодження та прогнозування поведінки системи. Відсутність протоколу маршрутизації знижує обчислювальне навантаження на мікроконтролери вузлів та зменшує кількість службового

трафіку в ефірі. Це критично важливо для малопотужних вузлів на базі ATmega328 та для забезпечення тривалої автономної роботи.

Порівняно з LoRaWAN, кластерна приватна мережа не потребує централізованої хмарної інфраструктури та платних сервісів. Вся система функціонує автономно, без доступу до мережі Інтернет, що підвищує захищеність від зовнішнього впливу. Власний протокол обміну дозволяє оптимізувати структуру пакетів та логіку взаємодії вузлів під конкретну задачу.

Кластерна топологія також забезпечує природне розподілення навантаження на ефір. Сенсори одного кластера передають на одному каналі, не заважаючи сенсорам інших кластерів. Репітери передають на серверному каналі почергово. Такий поділ частотного ресурсу між рівнями ієрархії мінімізує колізії та забезпечує передбачувану затримку доставки.

1.8.3 Вибір технологічної бази

Технологічна база системи визначена наступним чином. Для зв'язку між сенсорами та репітерами, а також між репітерами та сервером використовується технологія LoRa у діапазоні 433 МГц на базі модулів AS32-TTL-100. Вибір 433 МГц обумовлений кращим поширенням сигналу через рослинність порівняно з 868 МГц, а також доступністю модулів на ринку.

Для сенсорних вузлів обрано платформу Arduino Nano (ATmega328P), що забезпечує достатній обсяг пам'яті для реалізації протоколу, підтримує режими глибокого сну та має низьку вартість. Для репітерів використовується аналогічна платформа.

Для центрального сервера обрано платформу ESP32, що має вбудований Wi-Fi модуль та достатній обсяг ресурсів для одночасного обслуговування LoRa-каналу та HTTP-сервера. Це дозволяє реалізувати веб-інтерфейс моніторингу без додаткових комп'ютерів чи смартфонів.

Як датчик виявлення обрано доплерівський радарний модуль (HB-100 або аналоги) на частоті 5–10 ГГц [2]. Перевага доплерівського принципу виявлення — незалежність від освітленості, здатність виявляти рух через перешкоди (листяний покрив, чагарник), а також вища стійкість до хибних спрацювань порівняно з пасивними інфрачервоними датчиками в умовах різких перепадів температури навколишнього середовища.

2 РОЗРОБКА СТРУКТУРИ МЕРЕЖІ ТА ПОСТАНОВКА ЗАДАЧ

2.1 Формулювання вимог до системи

2.1.1 Функціональні вимоги

Функціональні вимоги визначають, що система повинна виконувати з точки зору користувача. На підставі аналізу задачі моніторингу польового периметру сформульовано такий перелік функціональних вимог [1, 2].

Система повинна виявляти факт перетину або присутності в зоні контролю рухомого об'єкта за допомогою датчика, що не потребує візуального контакту та не залежить від рівня освітленості. Доплерівський радарний модуль задовольняє цю вимогу, реагуючи на переміщення об'єктів незалежно від часу доби та погодних умов [2]..

При виявленні руху система повинна формувати повідомлення про подію та передавати його на центральний пункт збору даних. Повідомлення повинне містити ідентифікатор вузла, що зафіксував подію, а також накопичений лічильник спрацювань.

Система повинна забезпечувати контроль стану вузлів незалежно від наявності або відсутності подій. Для цього кожен сенсорний вузол повинен передавати на сервер значення напруги живлення з фіксованою періодичністю, що дозволяє виявляти як розряд акумуляторів, так і відсутність зв'язку з вузлом.

Центральний сервер повинен відображати стан усіх вузлів у вигляді таблиці, доступної через стандартний веб-браузер на будь-якому пристрої, підключеному до локальної Wi-Fi мережі сервера. Таблиця повинна містити ідентифікатор кластера, адресу вузла, лічильник подій, напругу живлення, час останнього виходу на зв'язок та кількість спроб передачі [9].

Система повинна підтримувати механізм непрямого управління вузлами: оператор на сервері повинен мати можливість передати команду

конкретному сенсорному вузлу через репітер без необхідності прямого зв'язку між сервером і сенсором.

Сервер повинен автоматично виявляти вузли, що припинили виходити на зв'язок протягом визначеного часу (watchdog-функція), та сигналізувати про це оператору.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають обмеження та характеристики системи, що не пов'язані безпосередньо з її функціями.

Вимога до автономності: сенсорний вузол повинен функціонувати без заміни або зарядки акумуляторів не менше 30 діб при нормальній активності (до 20 подій на добу). Ця вимога є визначальною для вибору режимів енергоспоживання мікроконтролера та датчика [8]..

Вимога до масштабованості: система повинна підтримувати не менше 64 сенсорних вузлів при початковому розгортанні з можливістю масштабування до 320 вузлів без зміни архітектури. Масштабування повинне виконуватись шляхом додавання нових кластерів (репітерів) без модифікації програмного забезпечення існуючих вузлів.

Вимога до дальності: відстань між сенсором та репітером його кластера повинна становити до 3 км у лісових умовах та до 5 км на відкритій місцевості. Відстань між репітером та центральним сервером — до 10 км на відкритій місцевості [3].

Вимога до затримки доставки: час від спрацювання датчика до відображення події на сервері не повинен перевищувати 30 секунд у штатному режимі роботи мережі.

Вимога до надійності: у нормальних умовах не менше 95% пакетів повинні бути доставлені на сервер протягом двох циклів повторних спроб. Механізм підтвердження та повторних передач повинен забезпечувати доставку навіть при тимчасових погіршеннях якості каналу.

Вимога до умов експлуатації: вузли мережі повинні функціонувати при температурах від -20°C до +50°C, при відносній вологості до 95% (без конденсації). Корпус сенсорного вузла повинен забезпечувати захист від дощу та роси.

Вимога до розгортання: введення в роботу одного сенсорного вузла повинне займати не більше 5 хвилин і виконуватись без спеціалізованого обладнання. Вузол повинен розпочинати роботу з правильно встановленими параметрами відразу після подачі живлення без налаштувань на місці [1].

2.2 Архітектура триступеневої мережі

2.2.1 Загальна структура

На підставі сформульованих вимог та результатів аналізу, обрана триступенева ієрархічна кластерна архітектура мережі. Загальна структура такої системи наведена на рисунку 2.1. Мережа складається з трьох типів вузлів: сенсорних вузлів (надалі — «сенсори» або «міні-вузли»), вузлів-ретрансляторів (надалі — «репітери») та центрального сервера [4, 11].

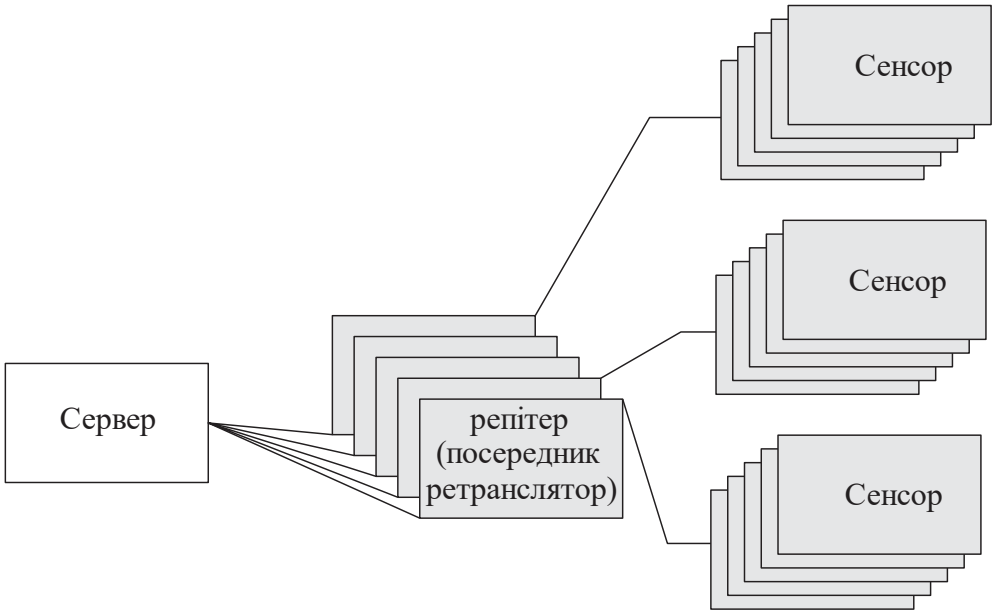


Рисунок 2.1 – Структура системи

Перший рівень ієрархії — сенсорні вузли. Кожен сенсор встановлюється в точці контролю, виявляє рух та передає пакети даних репітеру свого кластера. Структурна схема одного із множини сенсорів системи наведена на рисунку 2.2. Сенсор не з'єднується безпосередньо з сервером і не має інформації про інші сенсори мережі — лише про адресу репітера та номер свого частотного каналу.

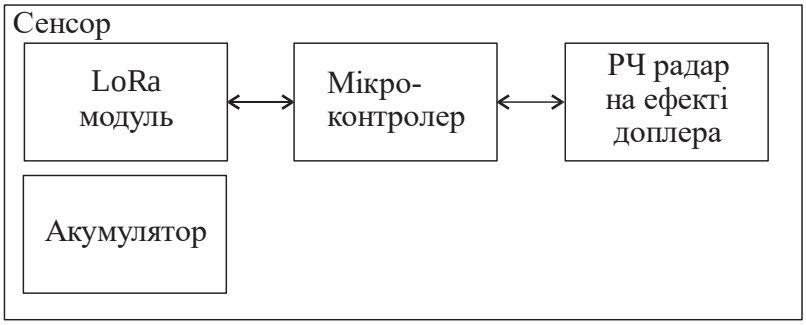


Рисунок 2.2– Структурна схема одного із множини сенсорів системи

Другий рівень ієрархії — репітери. Кожен репітер обслуговує один кластер і виконує три функції: приймає пакети від сенсорів, підтверджує їх отримання сенсорам та ретранслює агреговані дані на центральний сервер. Репітер є посередником між рівнем сенсорів і рівнем сервера. Він зберігає актуальний стан кожного сенсора свого кластера та передає ці дані на сервер незалежно від поточного стану зв'язку з сервером. Структурна схема одного із множини репітерів наведена на рисунку 2.3.

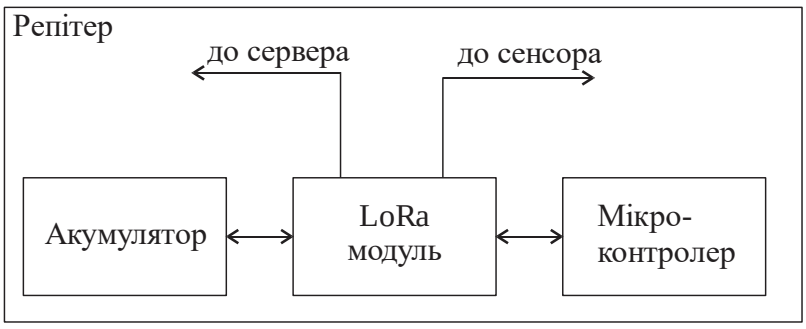


Рисунок 2.3 – Структурна схема одного із множини репітерів

Третій рівень — центральний сервер. Сервер приймає дані від всіх репітерів, зберігає їх у внутрішній базі даних, підтверджує отримання пакетів репітерам та надає оператору доступ до зведеної інформації через вбудований веб-сервер. Структурна схема сервера наведена на рисунку 2.4.

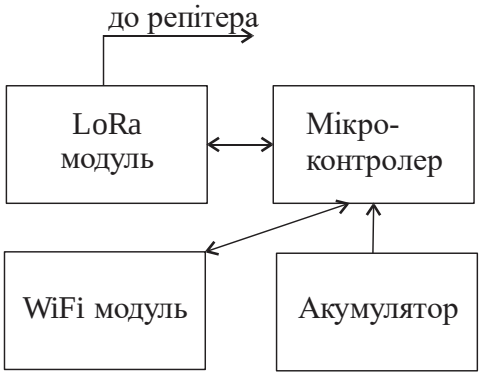


Рисунок 2.4 – Структурна схема сервера

Така структура має кілька важливих властивостей. Сенсор взаємодіє лише з репітером свого кластера — це мінімізує час роботи радіомодуля сенсора та, відповідно, споживання енергії. Репітер є автономним буфером: навіть при тимчасовій відсутності зв'язку з сервером він зберігає дані та доставляє їх при відновленні каналу [10]. Сервер є єдиною точкою збору та відображення даних, що спрощує роботу оператора.

2.2.2 Кількісні параметри мережі

Максимальна кількість сенсорів у одному кластері — 32. Це обмеження визначається розрядністю бітової маски pendingKnots у коді репітера (32-бітна змінна типу uint32_t), що зберігає чергу на відправку даних на сервер. Практична кількість сенсорів у кластері може бути менше залежно від частоти подій та завантаженості каналу.

Максимальна кількість кластерів, що обслуговується одним сервером — 10. Це обмеження задане константою MAX_CLUSTERS у коді сервера. При необхідності воно може бути збільшено без зміни архітектури.

Обмеження пов'язане з кількістю доступних частотних каналів та обсягом оперативної пам'яті ESP32.

Загальна ємність системи при поточних параметрах: 10 кластерів × 32 сенсори = 320 сенсорних вузлів. Кожен кластер використовує окремий частотний канал у діапазоні 433 МГц (канали 10, 12, 14, ..., 28 — парні значення від 10 до 28 включно). Зв'язок між репітерами та сервером відбувається на окремому каналі 2.

Адресний простір мережі: сервер має адресу 0x0001, репітер — 0x0002. Сенсори у кластері мають адреси від 0x0003 до 0x0022 (десяткові 3–34). Оскільки кожен кластер використовує окремий частотний канал, адреси сенсорів не є глобально унікальними — вони унікальні лише в межах каналу. Ідентифікація конкретного сенсора на рівні сервера виконується за комбінацією «номер кластера + адреса у кластері». Зведені кількісні параметри мережі наведено в таблиці 2.1.

Таблиця 2.1 — Кількісні параметри мережі

Параметр	**Значення**
Кількість рівнів ієрархії	3 (сенсор — репітер — сервер)
Макс. сенсорів у кластері	32
Макс. кластерів на сервер	10
Загальна ємність мережі	320 сенсорних вузлів
Канали сенсор–репітер	10, 12, 14, ..., 28 (10 каналів)
Канал репітер–сервер	2 (єдиний для всіх репітерів)
Адреса сервера	0x0001
Адреса репітера	0x0002
Адреси сенсорів у кластері	0x0003 – 0x0022 (3–34)

2.3 Частотно-канальне планування

2.3.1 Вибір частотного діапазону

Модулі AS32-TTL-100 підтримують роботу у діапазоні 410–441 МГц з кроком налаштування 1 МГц. Значення параметра CHAN у конфігурації модуля відповідає частоті за формулою: F (МГц) = 410 + CHAN. При CHAN = 10 робоча частота становить 420 МГц, при CHAN = 30 — 440 МГц.

Для кластерних каналів (зв'язок сенсор–репітер) використовуються значення CHAN від 10 до 28 з кроком 2. Крок 2 МГц між сусідніми каналами обраний з урахуванням ширини спектра сигналу LoRa при швидкості 2,4 кбіт/с та смузі 125 кГц — результуюча ширина зайнятого спектра не перевищує 200 кГц, тому сусідні канали з кроком 2 МГц не перекриваються [12].

Для серверного каналу (зв'язок репітер–сервер) використовується CHAN = 2 (частота 412 МГц). Цей канал виведений за межі діапазону кластерних каналів, що виключає взаємні завади між кластерними передачами та передачами на сервер.

2.3.2 Схема розподілу каналів

Кластер 0 (індекс 0) використовує CHAN = 10 (420 МГц). Кластер 1 — CHAN = 12 (422 МГц). Кластер 2 — CHAN = 14 (424 МГц). Далі з кроком 2 до кластера 9, що використовує CHAN = 28 (438 МГц). Відповідність між індексом кластера і номером каналу визначається формулою: CHAN = 10 + 2 × clusterIdx.

На стороні сервера перетворення у зворотному напрямку: cIdx = (viaCluster – 10) / 2, де viaCluster — значення поля каналу з пакету репітера. Перевірка viaCluster >= 10 відфільтровує пакети із невалідним номером каналу.

Така схема дозволяє всім репітерам передавати на сервер по одному каналу (CHAN = 2), а сервер розрізняє кластери не за частотою, а за полем viaCluster у пакеті. Це спрощує налаштування сервера: він постійно слухає один канал і не потребує перемикання між каналами.

2.3.3 Аналіз завантаженості каналу

Тривалість одного пакету LoRa при швидкості 2,4 кбіт/с та розмірі пакету 14 байтів (RepeaterPacket) розраховується як: T = (14 × 8) / 2400 ≈ 0,047 с. З урахуванням преамбули та затримок реального модуля тривалість становить приблизно 50–100 мс [10].

При 32 сенсорах у кластері з частотою передачі одного пакету heartbeat кожні 20 хвилин, сумарний потік від кластера складає 32 / 1200 ≈ 0,027 пакетів за секунду, або один пакет кожні 37 секунд. При наявності подій потік зростає, але навіть у разі одночасного спрацювання всіх 32 сенсорів вони передаватимуть пакети в різний час через різні цикли сну (8 с), що рівномірно розподіляє навантаження.

Серверний канал обслуговує до 10 репітерів. При мінімальному інтервалі ретрансляції RETRY_INTERVAL (визначений у Settings.h) кожен репітер передає на сервер не частіше одного разу за цей інтервал. При значенні RETRY_INTERVAL = 5 секунд і 10 репітерах максимальна частота пакетів на серверному каналі становить 10/5 = 2 пакети за секунду, що при тривалості пакету 100 мс дає завантаженість каналу 20% — прийнятне значення для надійного зв'язку [7].

2.4 Протокол обміну даними

2.4.1 Типи пакетів

У мережі використовуються три типи пакетів: MiniPacket — пакет від сенсора до репітера; RepeaterPacket — пакет від репітера до сервера;

AckPacket — пакет підтвердження (від репітера до сенсора, або від сервера до репітера).

Кожен тип пакету ідентифікується першим байтом — ключем (key). Значення ключів: KEY_MINI = 0xBB для пакетів сенсора та підтверджень сенсорам; KEY_REP = 0xEE для пакетів репітера; KEY_SERVER_ACK = 0xCC для підтверджень від сервера до репітера. Ключ дозволяє приймачу визначити тип пакету та його розмір до зчитування решти байтів.

2.4.2 Структура пакету сенсора (MiniPacket)

Пакет MiniPacket має розмір 10 байтів і передається від сенсора до репітера. Структура пакету відповідає наступному оголошенню мовою C:

```
struct MiniPacket {
    uint8_t  key;        // 0xBB — ідентифікатор типу пакету
    uint8_t  toAddr;     // адреса призначення (0x02 — репітер)
    uint8_t  fromAddr;   // адреса сенсора (0x03 – 0x22)
    uint8_t  viaCluster; // номер каналу кластера (10, 12, ..., 28)
    uint16_t cnt;        // лічильник спрацювань доплера
    uint16_t data;       // напруга живлення сенсора, мВ
    uint8_t  tryNum;     // кількість спроб передати цей пакет
    uint8_t  cs;        // контрольна сума (сума байтів 0..8)
};
```

Поле key дозволяє репітеру ідентифікувати пакет як пакет від сенсора. Поле toAddr містить адресу репітера (0x02), що дозволяє репітеру відфільтровувати пакети, адресовані іншим вузлам. Поле fromAddr містить унікальну адресу сенсора у межах кластера. Поле viaCluster вказує номер частотного каналу, по якому передається пакет.

Поле cnt — 16-бітний лічильник, що інкрементується при кожному спрацюванні доплерівського датчика. Сервер відстежує зміну лічильника між послідовними пакетами від одного сенсора, що дозволяє визначити кількість подій у заданий проміжок часу. При переповненні (65535 → 0) лічильник продовжує роботу без скидання нульового значення, оскільки сервер відстежує відносну зміну, а не абсолютне значення.

Поле data передає поточну напругу живлення у мілівольтах [8]. Значення розраховується мікроконтролером шляхом вимірювання внутрішньої опорної напруги 1,1 В через АЦП ATmega328P та зворотного перерахунку за формулою Vcc = 1125300 / ADC_result.

Поле tryNum збільшується на 1 при кожній спробі передати поточне значення лічильника. При успішному підтвердженні tryNum скидається в 0. Це дозволяє серверу бачити, скільки спроб знадобилось для доставки кожного конкретного пакету, що є непрямою характеристикою якості каналу зв'язку між сенсором і репітером.

Контрольна сума cs обчислюється як сума всіх байтів пакету, крім останнього (cs), за формулою: cs = SUM(bytes[0..8]) mod 256. Репітер обчислює ту саму суму при прийомі та порівнює з отриманим значенням cs. Пакет із невідповідністю контрольної суми відкидається.

2.4.3 Структура пакету репітера (RepeaterPacket)

Пакет RepeaterPacket має розмір 13 байтів і передається від репітера до сервера. Він містить агреговані дані як про стан сенсора, так і про стан репітера [1].

```
struct RepeaterPacket {
    uint8_t  key;        // 0xEE — ідентифікатор типу пакету
    uint8_t  toAddr;     // адреса призначення (0x01 — сервер)
    uint8_t  fromAddr;   // адреса сенсора, дані якого пересилаються
    uint8_t  viaCluster; // номер каналу кластера
    uint16_t cnt;        // лічильник сенсора
    uint16_t miniData;   // напруга живлення сенсора, мВ
    uint8_t  miniTry;    // кількість спроб передачі від сенсора
    uint16_t repData;    // напруга живлення репітера, мВ
    uint8_t  repTry;     // кількість спроб доставки на сервер
    uint8_t  cs;        // контрольна сума
};
```

Ключова відмінність RepeaterPacket від MiniPacket — наявність двох полів напруги: miniData (напруга сенсора) та repData (напруга самого

репітера). Це дозволяє серверу одночасно контролювати стан акумуляторів як сенсорів, так і репітерів.

Поле fromAddr у RepeaterPacket містить адресу сенсора, дані якого ретранслюються, а не адресу самого репітера. Це дозволяє серверу коректно ідентифікувати вихідний сенсор без додаткових полів. Адреса репітера не передається явно, оскільки вона завжди дорівнює 0x02 і сервер знає це з конфігурації.

Поле repTry зберігає кількість спроб доставки конкретного пакету на сервер. Репітер збільшує tryToServer при кожній відправці і скидає при отриманні підтвердження від сервера. Аналогічно до miniTry, це дозволяє оцінити якість каналу між репітером і сервером.

2.4.4 Структура пакету підтвердження (AckPacket)

Пакет AckPacket використовується у двох напрямках: від репітера до сенсора (підтвердження отримання MiniPacket) та від сервера до репітера (підтвердження отримання RepeaterPacket). Структура ідентична в обох випадках, відрізняються лише значення ключа.

```
struct AckPacket {
    uint8_t  key;     // 0xBB (до сенсора) або 0xCC (від сервера)
    uint8_t  toAddr;  // адреса отримувача
    uint8_t  fromAddr;// адреса відправника
    uint8_t  cmd;     // команда (0 = ACK, інше = команда сенсору)
    uint16_t cnt;     // лічильник з пакету, що підтверджується
    uint8_t  cs;     // контрольна сума
};
```

Відображення лічильника cnt у пакеті підтвердження — ключовий елемент надійності протоколу [7]. Сенсор, отримавши підтвердження, порівнює значення cnt у підтвердженні зі своїм поточним eventCounter. Якщо значення збігаються, сенсор вважає доставку успішною і скидає прапор hasKnot та лічильник спроб tryAttempt. Якщо не збігаються (репітер підтвердив застарілий пакет), сенсор залишає прапор встановленим і повторить передачу.

Поле cmd призначене для передачі команди сенсору. При значенні 0 пакет є чистим підтвердженням без команди. При ненульовому значенні репітер включає команду, отриману від сервера, у наступне підтвердження при відповіді конкретному сенсору. Таким чином реалізується непрямий механізм управління: сервер → репітер → сенсор при наступному зверненні сенсора.

2.5 Задачі сенсорного вузла

2.5.1 Задача виявлення руху

Сенсорний вузол виявляє рух за допомогою доплерівського радарного модуля, що функціонує безперервно незалежно від стану мікроконтролера. Модуль випромінює неперервний радіосигнал і аналізує відбиті хвилі. При наявності рухомого об'єкта в зоні чутливості виникає доплерівський зсув частоти відбитого сигналу, що призводить до появи аналогового сигналу на виході модуля [2].

Вихідний сигнал модуля підключається до входу апаратного переривання мікроконтролера (пін INT0, Arduino Nano). Переривання налаштовується на фронт (RISING) або спад (FALLING) сигналу залежно від характеристик конкретного модуля — це задається константою SENSOR_TRIGGER у файлі Settings.h.

При спрацюванні переривання виконується мінімальна функція обробника ISR_Sensor: встановлюється прапор wokeBySensor = true та засвічується вбудований світлодіод. Сам вихід ISR не ініціює відправки пакету — це відбувається у функції updateLogicState при наступному пробудженні контролера від watchdog-таймера.

Такий підхід захищає від хибних спрацювань, що тривають менше 8 секунд: якщо датчик генерує короткий імпульс, переривання встановить

прапор, але пакет буде відправлений лише через 8 секунд. За цей час можна переконатися, що подія реальна, а не є завадою.

2.5.2 Задача управління енергоспоживанням

Основна задача управління живленням — забезпечити автономну роботу понад 30 діб за рахунок мінімізації середнього споживання струму [8]. Вирішення цієї задачі полягає у переведенні мікроконтролера та радіомодуля у режими зниженого споживання на максимальний час.

Мікроконтролер ATmega328P підтримує кілька режимів сну. Найглибший режим — SLEEP_MODE_PWR_DOWN — зупиняє всі тактові генератори, залишаючи активними лише схему watchdog та апаратні переривання. У цьому режимі споживання знижується до 0,36 мкА при 3,3 В. Пробудження можливе від сигналу на піні переривання або від спрацювання watchdog-таймера.

Watchdog-таймер налаштовується на максимальний інтервал 8 секунд (прескалер WDP3|WDP0). Після кожного пробудження від watchdog лічильник wakeCounter збільшується на 1. Для heartbeat-передачі встановлено поріг MAX_SLEEPS_HEARTBEAT: при досягненні цього лічильника сенсор надсилає плановий пакет зі станом акумулятора навіть за відсутності подій.

Радіомодуль AS32-TTL-100 переводиться у режим сну шляхом встановлення логічної одиниці на обох пінах M0 та M1 одночасно. У режимі сну споживання модуля знижується до кількох мікроампер. Пробудження модуля (M0 = M1 = 0) займає близько 50 мс, після чого модуль готовий до роботи. Затримка 50 мс враховується у функції sendAndReceive перед відправкою пакету.

Під час очікування підтвердження (до 2 секунд) мікроконтролер не переходить у режим сну — він виконує активне очікування в циклі while. Це спрощує логіку, але збільшує споживання на цей час. Якщо в нормальних

умовах підтвердження приходить за 100–200 мс, то середнє активне время за 8-секундний цикл становить менше 3%.

АЦП мікроконтролера вимикається перед переходом у режим сну (ADCSRA &= ~(1 << ADEN)) та вмикається при пробудженні. АЦП у режимі активності споживає близько 250 мкА, тому його вимикання дає незначний, але корисний ефект [8].

2.5.3 Задача передачі даних та отримання підтвердження

Задача передачі полягає у формуванні пакету MiniPacket, відправці його репітеру та очікуванні підтвердження AckPacket протягом обмеженого часу RX_TIMEOUT [7].

Перед відправкою сенсор вимірює поточну напругу живлення через readVcc(), заповнює поля пакету та обчислює контрольну суму. Відправка виконується функцією loRa.sendFixedMessage() з вказівкою старшого та молодшого байтів адреси репітера та номера каналу.

Після відправки сенсор відкриває вікно прийому на час RX_TIMEOUT (до 2000 мс). Очікування реалізовано у циклі while з перевіркою доступності байтів у буфері UART та виходом по таймауту. При надходженні достатньої кількості байтів (sizeof(AckPacket)) сенсор зчитує пакет та перевіряє: ключ key == SYS_KEY, адресу toAddr == власна адреса, команду cmd == 0 (немає команди або є команда, але ACK валідний), контрольну суму.

При успішній перевірці та збігу cnt із поточним eventCounter сенсор скидає прапор hasKnot та лічильник спроб tryAttempt. При наступному циклі сенсор не буде відправляти пакет до появи нової події або досягнення порогу heartbeat.

При відсутності підтвердження (таймаут) прапор hasKnot залишається встановленим, tryAttempt збільшується. При наступному пробудженні через 8 секунд сенсор знову виконає sendAndReceive. Значення tryNum у пакеті

збільшиться, що повідомить репітер про кількість спроб. Максимальне значення tryAttempt обмежено 100 для запобігання переповненню.

2.5.4 Задача heartbeat-передачі

Навіть за відсутності подій від доплерівського датчика сенсор повинен регулярно виходити на зв'язок для підтвердження своєї працездатності та передачі стану акумулятора. Ця функція реалізована як heartbeat — планова передача за таймером [9].

Лічильник wakeCounter збільшується при кожному пробудженні від watchdog (кожні 8 с). При досягненні

wakeCounter == MAX_SLEEPS_HEARTBEAT

встановлюється прапор hasKnot та скидається wakeCounter. Значення MAX_SLEEPS_HEARTBEAT задається у Settings.h: при значенні 150 інтервал heartbeat становить 150 × 8 = 1200 с = 20 хвилин.

У heartbeat-пакеті лічильник cnt залишається незмінним порівняно з попереднім пакетом (якщо подій не було). Сервер розрізняє heartbeat від звичайного пакету події за відсутністю приросту cnt: якщо cnt не змінився, отримано черговий плановий пакет стану.

Допоміжний механізм: при кожному п'ятому пробудженні (wakeCounter % 5 == 0) також встановлюється hasKnot. Це не heartbeat у повному сенсі, а відладковий механізм (позначений коментарем ***), що може бути вимкнений у бойовій версії програмного забезпечення.

2.6 Задачі вузла-репітера

2.6.1 Задача прийому пакетів від сенсорів

Репітер безперервно перебуває в режимі прийому, прослуховуючи свій локальний канал (LOCAL_CHANNEL) [4]. Функція processIncomingData() викликається в кожній ітерації основного циклу loop() і перевіряє наявність байтів у буфері UART.

При появі даних зчитується перший байт — ключ пакету. За значенням ключа визначається тип пакету та кількість байтів, що очікуються: для MiniPacket — sizeof(MiniPacket) - 1 = 9 байтів після ключа; для AckPacket (підтвердження від сервера) — sizeof(AckPacket) - 1 = 6 байтів. Ключі, що не відповідають очікуваним значенням (KEY_MINI або KEY_SERVER_ACK), ігноруються — функція повертає управління і наступна ітерація циклу повторить перевірку.

Після зчитування решти байтів пакету виконується перевірка трьох умов: контрольна сума повинна збігатися з обчисленою, адреса toAddr повинна дорівнювати адресі репітера MY_ADDR, номер каналу viaCluster повинен збігатися з LOCAL_CHANNEL. Пакети, що не пройшли хоча б одну перевірку, відкидаються без подальшої обробки.

Додатково перевіряється допустимість адреси відправника: fromAddr повинна бути в діапазоні від MY_ADDR + 1 до MY_ADDR + 31, виключаючи адресу самого репітера. Це запобігає обробці пакетів від невідомих або некоректно налаштованих вузлів.

2.6.2 Задача підтвердження доставки сенсорам

Одразу після успішної перевірки пакету від сенсора репітер надсилає підтвердження AckPacket. Підтвердження формується і надсилається до оновлення внутрішніх таблиць та відправки даних на сервер — це мінімізує час між відправкою сенсора і отриманням ним підтвердження, що критично для виходу сенсора із режиму активного очікування [7].

Підтвердження надсилається функцією loRa.sendFixedMessage() на адресу сенсора fromAddr по каналу LOCAL_CHANNEL. У підтвердженні поле cnt містить той самий лічильник, що прийшов від сенсора — це дозволяє сенсору перевірити, що підтверджено саме поточний пакет.

Поле cmd у підтвердженні встановлюється в 0 для звичайного ACK. Якщо оператор на сервері передав команду для конкретного сенсора, вона

зберігається у масиві команд репітера та вставляється в поле cmd при наступному підтвердженні для цього сенсора.

2.6.3 Задача буферизації та черги на відправку серверу

Після підтвердження сенсору репітер оновлює своє локальне сховище: записує в масив nodes[idx] значення cnt, voltage та tryNum від сенсора, де idx = fromAddr - MY_ADDR (індекс від 0 до 31). Одночасно встановлюється відповідний біт у 32-бітній змінній pendingKnots.

Бітова маска pendingKnots є чергою задач на відправку серверу [10]. Кожен встановлений біт означає: для сенсора з індексом і є актуальні дані, що ще не підтверджені сервером. При отриманні підтвердження від сервера (AckPacket з KEY_SERVER_ACK) відповідний біт скидається.

Перевага такого підходу — мінімальне використання пам'яті. Черга для 32 сенсорів займає лише 4 байти (uint32_t). Фактичні дані сенсорів зберігаються в масиві nodes[32], що займає 32 × (2+2+1) = 160 байтів. Загальні витрати пам'яті на управління чергою та зберігання даних — 164 байти, що становить менше 8% оперативної пам'яті ATmega328P.

Відправка по черзі реалізована за принципом round-robin: глобальний курсор globalCursor зберігає позицію, з якої буде розпочато пошук наступного встановленого біта. Після відправки курсор просувається на 1. Якщо всі наступні біти до кінця 32-бітного вектора скинуті, курсор переходить на початок. Це забезпечує рівномірне обслуговування всіх сенсорів без пріоритетів.

2.6.4 Задача відправки даних на сервер

Відправка на сервер виконується функцією sendToServer(slot) за таймером: інтервал RETRY_INTERVAL перевіряється в кожній ітерації основного циклу. При закінченні інтервалу та наявності встановлених бітів у

pendingKnots виконується одна відправка для найближчого встановленого біта.

Функція sendToServer формує пакет RepeaterPacket, заповнюючи поля з масиву nodes[slot] (дані сенсора) та вимірюючи поточну напругу живлення репітера через readVcc(). Пакет надсилається на адресу сервера 0x0001 по серверному каналу SERVER_CHANNEL = 2.

Лічильник tryToServer збільшується при кожній відправці та передається у полі repTry пакету. При отриманні підтвердження від сервера tryToServer скидається в 0. Якщо сервер не відповідає, при наступному спрацюванні таймера та наявності встановленого біта для цього сенсора відправка повториться з incremented repTry.

Важлива деталь: у поточній реалізації sendToServer також викликається безпосередньо після отримання пакету від сенсора — не чекаючи таймера. Це забезпечує мінімальну затримку для пакетів, що несуть інформацію про події: сенсор відправив подію → репітер миттєво ретранслює на сервер, не очікуючи чергового спрацювання таймера [9].

2.7 Задачі центрального сервера

2.7.1 Задача прийому та обробки пакетів від репітерів

Сервер реалізований на базі ESP32 і виконує дві паралельні задачі у головному циклі: processIncomingData() для прийому LoRa-пакетів та server.handleClient() для обробки HTTP-запитів від клієнтів веб-інтерфейсу. Завдяки однопоточній архітектурі Arduino-фреймворку на ESP32 ці задачі виконуються почергово, що не є проблемою через відносно невелику швидкість надходження пакетів.

При прийомі байтів по UART від LoRa-модуля сервер спочатку виконує фільтрацію сміттєвих байтів: у циклі зчитуються байти з порту до тих пір, поки не зустрінеться байт KEY_REP = 0xEE — очікуваний ключ

пакету репітера. Якщо за 100 ітерацій ключ не знайдено, функція повертає управління. Така фільтрація захищає від сміттєвих даних у буфері UART, що можуть виникати при включенні живлення або при радіозавадах.

Після виявлення ключа сервер перевіряє, чи у буфері накопичилась кількість байтів, достатня для повного пакету RepeaterPacket. Якщо так — зчитує пакет і передає у функцію processRepPacket для детальної обробки.

Функція processRepPacket виконує перевірки цілісності та адресації, оновлює масив nodes[cIdx][nodeIdx] відповідними даними та відправляє підтвердження репітеру. Оновлення останній час виходу на зв'язок здійснюється в одиницях секунд (millis() / 1000) для зручності відображення у веб-інтерфейсі.

Якщо поле maxTries у запису вузла менше, ніж miniTry або repTry з пакету, що надійшов, значення maxTries оновлюється. Це відстежує максимальну кількість спроб за весь час роботи і є показником найгіршої зафіксованої якості зв'язку.

2.7.2 Задача watchdog-моніторингу

Watchdog-функція сервера відстежує загальну активність мережі [1]. Змінна lastValidPacketMillis оновлюється при кожному успішно прийнятому та перевіреному пакеті від будь-якого репітера. Якщо з моменту останнього успішного прийому минуло більше WATCHDOG_TIMEOUT = 1800000 мс (30 хвилин), сервер виконує повторну ініціалізацію LoRa-модуля функцією setupLoRaConfig().

Повторна ініціалізація необхідна для відновлення стану LoRa-модуля у випадку, якщо він «завис» через несправний пакет або збій інтерфейсу UART. Після переконфігурації модуль повертається у нормальний режим прийому.

30-хвилинний таймаут обраний з таких міркувань: навіть якщо всі сенсори в мережі нерухомі, heartbeat-пакети від кожного сенсора надходять

кожні 20 хвилин, тому за 30 хвилин за наявності хоча б одного активного сенсора на сервер повинен надійти принаймні один пакет. Відсутність пакетів протягом 30 хвилин свідчить або про повну відсутність активних вузлів, або про несправність модуля сервера.

Watchdog на рівні окремих вузлів реалізований у веб-інтерфейсі: при відображенні таблиці для кожного вузла обчислюється час (в секундах) від поля lastSeen до поточного часу. Вузли, від яких не надходило даних більше заданого порогу, позначаються у таблиці відповідним індикатором.

2.7.3 Задача підтвердження доставки репітерам

При успішному прийомі та перевірці пакету від репітера сервер негайно надсилає підтвердження функцією sendAckToRepeater(). Підтвердження формується як AckPacket з ключем KEY_SERVER_ACK = 0xCC та надсилається на адресу репітера по тому ж каналу viaCluster, по якому надійшов пакет [7].

Поле toAddr у підтвердженні встановлюється у значення REPEATER_ADDR = 0x0002 (фіксована адреса репітера). Поле cnt відповідає лічильнику з пакету, що підтверджується. Репітер при отриманні цього підтвердження скидає відповідний біт у pendingKnots та обнуляє лічильник tryToServer.

Підтвердження надсилається на канал viaCluster — тобто на кластерний канал репітера, а не на серверний. Це пояснюється тим, що репітер після відправки пакету на сервер залишається прослуховувати свій LOCAL_CHANNEL та чекає підтвердження саме на цьому каналі. Сервер переключає LoRa-модуль на потрібний канал лише на час відправки підтвердження, після чого модуль повертається на серверний канал.

2.7.4 Задача веб-інтерфейсу

Веб-сервер реалізований на базі бібліотеки WebServer для ESP32 і слухає TCP-порт 80. При зверненні клієнта до кореневого URL '/' виконується функція handleRoot(), яка генерує HTML-сторінку зі зведеною таблицею стану мережі.

Таблиця формується динамічно у функції handleRoot(): сервер перебирає всі 10 кластерів та до 32 вузлів у кожному, перевіряє, чи є у вузла ненульовий lastSeen або cnt, і включає вузол у таблицю. Порожні вузли (без жодних записів) не відображаються.

Для кожного відображуваного вузла таблиця містить стовпці: кластер (cIdx), адреса вузла, поточний лічильник подій, відносний приріст лічильника (відносно точки відліку refCnt, яку оператор може встановити кнопкою), напруга акумулятора у мілівольтах, час від останнього зв'язку у секундах та максимальну кількість спроб.

Веб-сервер підтримує два режими Wi-Fi: режим точки доступу (AP mode), коли ESP32 сам формує Wi-Fi мережу з заданим SSID та паролем, та режим станції (STA mode), коли ESP32 підключається до існуючого роутера. Вибір режиму визначається директивою ENABLE_AP_MODE у вихідному коді. Режим AP доцільний для автономного польового застосування, режим STA — при наявності стаціонарної інфраструктури [5].

2.8 Розробка протоколів передачі даних для сенсорних мереж

2.8.1 Вимоги до протоколу

Протокол передачі даних у системі моніторингу периметру повинен задовольняти наступним вимогам. Мінімальний розмір пакету — для мінімізації часу передачі та, відповідно, споживання енергії. Підтримка адресації — кожен вузол у мережі має мати унікальну адресу для ідентифікації джерела даних. Контроль цілісності — механізм виявлення

пошкоджених пакетів. Підтвердження доставки — механізм, що дозволяє відправнику переконатися у доставці пакету. Механізм повторних спроб — автоматична повторна передача при відсутності підтвердження [6].

Додатково бажаними є: ідентифікація типу пакету для розрізнення пакетів даних та підтверджень; передача метаданих (рівень заряду батареї, кількість спроб передачі) разом із корисними даними; механізм непрямої адресації команд для управління вузлами без прямого з'єднання.

2.8.2 Власний протокол передачі

У розроблюваній системі застосований власний протокол із двома типами пакетів: пакет сенсора (MiniPacket, 10 байтів) та пакет підтвердження (AckPacket, 7 байтів). Вибір структури пакетів обумовлений балансом між достатністю переданої інформації та мінімізацією розміру [10].

Перший байт кожного пакету — ключ (key), що дозволяє ідентифікувати тип пакету на стороні приймача без додаткових заголовків. Різні ключі для пакетів сенсора (0xBB) та відповіді репітера дозволяють серверу та репітеру швидко відфільтровувати пакети потрібного типу та ігнорувати чужий трафік.

Адресація (поля toAddr та fromAddr) дозволяє у межах одного частотного каналу обслуговувати до 256 різних адрес. Для системи з 32 сенсорами на канал і 10 каналами це з надлишком забезпечує унікальність адрес.

Поле viaCluster передає номер частотного каналу, на якому сенсор спілкується з репітером. Це дозволяє серверу правильно зіставити отриманий від репітера пакет із конкретним кластером.

16-бітний лічильник спрацювань (cnt) дозволяє не лише фіксувати факт події, але й відстежувати динаміку активності у зоні контролю. Порівнюючи поточне значення лічильника з попереднім, сервер може визначити кількість подій з моменту останнього пакету.

Поле напруги (data) передає поточну напругу акумулятора у вигляді 16-бітного значення, що дозволяє планувати обслуговування вузлів до повного розряду. Поле tryNum інформує сервер про якість зв'язку між сенсором та репітером.

Контрольна сума (cs) обчислюється як XOR усіх попередніх байтів пакету. Цей алгоритм є простим у реалізації на мікроконтролері та виявляє одиночні бітові помилки та більшість пакетів зі множинними помилками, що є достатнім для виявлення пошкоджень при передачі.

2.8.3 Методи виявлення помилок

У системах передачі даних застосовуються різні методи виявлення помилок. Метод парності — додавання одного біта до байту або слову таким чином, щоб загальна кількість одиниць була парною або непарною. Виявляє непарну кількість помилок. Застосовується у UART-протоколі (біт парності) та USB.

Циклічний надлишковий код (CRC) — поліноміальний алгоритм, що обчислює контрольне значення фіксованої довжини (8, 16 або 32 біти) від довільного масиву даних. CRC-16 виявляє всі одиночні та подвійні помилки, всі пакетні помилки довжиною до 16 біт та велику частку інших помилок. Широко використовується у промислових протоколах (Modbus, CAN) [7].

XOR-контрольна сума — побітове виключне АБО всіх байтів блоку даних. Алгоритм простіший за CRC і не виявляє деякі типи помилок (наприклад, взаємну заміну двох байтів), але є достатнім для захисту від апаратних збоїв при передачі по радіоканалу, де помилки, як правило, є пакетними (burst errors) і впливають на суміжні біти.

Для малопотужних мікроконтролерів з обмеженими ресурсами XOR-контрольна сума є прийнятним компромісом між надійністю виявлення помилок та складністю реалізації. Модуль LoRa додатково має власну

систему корекції помилок (Forward Error Correction, FEC) на фізичному рівні, що забезпечує додатковий рівень захисту [14].

2.8.4 Методи підтвердження доставки та повторних передач

Протокол ARQ (Automatic Repeat reQuest) — група протоколів, що забезпечують надійну доставку шляхом підтвердження отримання та повторної передачі при відсутності підтвердження. Розрізняють кілька варіантів ARQ [7].

Stop-and-Wait ARQ — після кожного пакету відправник очікує підтвердження перед надсиланням наступного. Простота реалізації, але низька ефективність при великих затримках каналу (RTT).

Go-Back-N ARQ — відправник надсилає кілька пакетів без очікування підтверджень (вікно передачі). При помилці всі пакети, починаючи з помилкового, передаються повторно. Вища ефективність, але складніша реалізація.

Selective Repeat ARQ — повторно передаються лише пакети, що не отримали підтвердження. Найефективніший варіант, але вимагає буферизації на обох сторонах.

Для системи моніторингу периметру з короткими пакетами та малою частотою передачі найбільш підходящим є спрощений варіант Stop-and-Wait ARQ [9]. Сенсор після передачі пакету очікує підтвердження протягом фіксованого часу (до 2 секунд). Якщо підтвердження не отримано, при наступному пробудженні через 8 секунд пакет передається повторно зі збільшеним лічильником tryNum. Такий підхід забезпечує надійну доставку без складної логіки та без необхідності буферизації множини пакетів.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ. ОСОБЛИВОСТІ КОЖНОГО БЛОКУ ТА РОЗРОБКА СЕРВЕРА

3.1 Апаратна реалізація сенсорного вузла

3.1.1 Склад апаратної частини

Сенсорний вузол побудований на чотирьох основних компонентах: мікроконтролерній платі Arduino Nano (мікроконтролер ATmega328P), радіомодулі AS32-TTL-100 на частоті 433 МГц, доплерівському радарному модулі та автономному джерелі живлення з акумуляторів формату 21700.

Arduino Nano обрано як основу сенсора через поєднання низької вартості, малих фізичних розмірів (18 × 45 мм), підтримки режимів глибокого сну та наявності апаратних переривань. Мікроконтролер ATmega328P, що входить до складу плати, має 32 КБ Flash-пам'яті для програми, 2 КБ SRAM та 1 КБ EEPROM. Тактова частота 16 МГц при живленні 5 В або 8 МГц при 3,3 В. У режимі SLEEP_MODE_PWR_DOWN споживання знижується до 0,36 мкА при 3,3 В та до 1 мкА при 5 В.

Радіомодуль AS32-TTL-100 є UART-інтерфейсним трансивером на базі чипа SX1276 від Semtech [10]. Технічні характеристики: діапазон частот 410–441 МГц, максимальна потужність передавача 100 мВт (20 дБм), чутливість приймача –147 дБм, швидкість передачі по радіоканалу 0,3–19,2 кбіт/с (налаштовується). Інтерфейс підключення — UART 9600 бод + три службових піни: M0, M1 (вибір режиму роботи) та AX (вхід або вихід переривання). Розміри модуля 21 × 36 мм, споживання у режимі прийому близько 12 мА, у режимі сну (M0=M1=1) менше 2 мкА.

Доплерівський радарний модуль типу HB-100 або CDM324 працює у діапазоні 5,8 або 24 ГГц відповідно. Модуль HB-100 має такі характеристики:

робоча частота 10,525 ГГц, потужність передавача 13–16 дБм, кут виявлення по горизонталі ±70°, по вертикалі ±35°, дальність виявлення людини в русі до 20 м (залежить від налаштування чутливості) [2]. Споживання у режимі роботи 30–50 мА при напрузі 5 В. Вихідний сигнал — аналоговий диференціальний, потребує підсилювача для формування цифрового сигналу, що подається на вхід переривання мікроконтролера.

Схема підключення компонентів: UART-виходи TX та RX модуля AS32-TTL-100 підключені до пінів програмного послідовного порту SoftwareSerial Arduino Nano (PIN_LORA_RX та PIN_LORA_TX, задаються у Settings.h). Піни M0 та M1 модуля підключені до цифрових виходів Arduino для програмного керування режимом сну. Пін AX підключений до цифрового виходу для апаратного скидання модуля. Вихід доплерівського модуля через компаратор або операційний підсилювач підключений до піна INT0 (пін 2) Arduino Nano, що підтримує зовнішнє переривання.

3.1.2 Схема живлення сенсорного вузла

Схема живлення є одним з ключових питань при проектуванні сенсора. Основна складність полягає у тому, що різні компоненти потребують різних рівнів напруги: мікроконтролер Arduino Nano та LoRa-модуль AS32-TTL-100 живляться від 3,3–5 В, тоді як доплерівський модуль HB-100 потребує 5 В [8].

Перший варіант схеми живлення — два акумулятори 21700 з'єднані паралельно. Напруга паралельного пакету становить 3,6–4,2 В (залежно від стану заряду). При повному заряді 4,2 В це достатньо для живлення Arduino Nano через вхід 5V або 3V3 з вбудованим стабілізатором. Однак для HB-100 напруга 3,6–4,2 В може бути недостатньою, оскільки мінімальна напруга живлення модуля за специфікацією становить 4,75 В.

Другий варіант — два акумулятори 21700 з'єднані послідовно, напруга пакету 7,2–8,4 В. Від цього пакету живляться всі компоненти через

понижуючі стабілізатори: один перетворювач знижує напругу до 5 В для HB-100 та Arduino Nano, другий (або вбудований стабілізатор Arduino) — до 3,3 В для LoRa-модуля. Перевага: достатня напруга для всіх компонентів. Недолік: потрібен балансир заряду для послідовно включених елементів, а ємність пакету не збільшується порівняно з одним елементом.

Розрахункові значення споживання струму основними компонентами сенсорного вузла наведено в таблиці 3.1.

.

Таблиця 3.1 — Споживання струму компонентів сенсорного вузла

Компонент	Напруга живлення	Струм у режимі роботи	Струм у режимі сну
ATmega328P (Arduino Nano)	3,3–5 В	10–20 мА	0,36 мкА
AS32-TTL-100 (LoRa), прийом	3,3–5 В	12 мА	< 2 мкА
AS32-TTL-100 (LoRa), передача	3,3–5 В	100–120 мА	—
HB-100 (доплер)	4,75–5,25 В	30–50 мА	немає
Всього (сон + доплер)	3,3–5 В	≈ 4 мА	—
Всього (передача + доплер)	3,3–5 В	≈ 140 мА	—

Третій варіант — паралельний пакет з двох елементів 21700 плюс підвищуючий перетворювач boost з вихідною напругою 5 В для живлення HB-100, тоді як Arduino та LoRa живляться безпосередньо від акумуляторів через понижуючий стабілізатор 3,3 В. Цей варіант технічно складніший, але дозволяє подвоїти ємність пакету (два паралельних елементи = 6000–7000 мАч) та забезпечити необхідну напругу для HB-100 [8].

Для вимірювання напруги живлення в коді сенсора застосовується метод зворотного вимірювання через внутрішню опорну напругу ATmega328P. Метод полягає у підключенні до АЦП внутрішнього джерела опорної напруги 1,1 В та вимірюванні відносно напруги живлення Vcc. Формула розрахунку: Vcc (мВ) = 1125300 / ADC_result, де 1125300 = 1,1 В × 1023 × 1000. Цей метод не потребує зовнішніх резисторів-дільників і

автоматично вимірює саме ту напругу, від якої живиться мікроконтролер

Реалізація вимірювання у коді: мультиплексор АЦП переключається на внутрішнє джерело (ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1)), виконується затримка 2 мс для стабілізації, запускається перетворення та зчитується результат. Значення повертається у мілівольтах як 32-бітне знакове ціле long.

3.2 Програмна реалізація сенсорного вузла

3.2.1 Ініціалізація та налаштування

Функція setup() виконується один раз при подачі живлення або після апаратного скидання. Послідовність ініціалізації визначена вимогами до порядку запуску компонентів.

Спочатку налаштовуються піни: LED_BUILTIN переводиться у режим виходу та гаситься, PIN_SENSOR_INT (пін 2, INT0) та пін 3 переводяться у режим входу без підтягуючого резистора. Ініціалізується програмний послідовний порт SoftwareSerial на швидкості 9600 бод — це швидкість UART-інтерфейсу LoRa-модуля. Ініціалізується апаратний послідовний порт Serial на 115200 бод для відлагодження.

Після ініціалізації послідовних портів викликається loRa.begin() — ініціалізація бібліотеки LoRa_E32. Потім викликається setupLoRa() з параметрами MY_ADDR та LOCAL_CHANNEL — ця функція зчитує поточну конфігурацію модуля, змінює лише необхідні поля (адреса, канал, швидкість, режим фіксованої адресації, FEC, потужність) та записує конфігурацію у RAM модуля командою WRITE_CFG_PWR_DWN_LOSE.

3.2.2 Налаштування LoRa-модуля

Функція setupLoRa() реалізує двоетапну конфігурацію. На першому етапі зчитується поточна конфігурація командою loRa.getConfiguration(). Це важливо, оскільки дозволяє змінити тільки потрібні параметри, не зачіпаючи

інші налаштування (зокрема, параметри UART-інтерфейсу модуля — зміна швидкості UART призвела б до втрати зв'язку до перезапуску).

На другому етапі змінюються поля конфігурації: ADDH та ADDL встановлюються у старший та молодший байти MY_ADDR; CHAN — у значення LOCAL_CHANNEL; airDataRate — у значення 2 (відповідає 2,4 кбіт/с, константа AIR_DATA_RATE_010_24); fixedTransmission — FT_FIXED_TRANSMISSION (режим адресованої передачі); fec — FEC_1_ON (увімкнення корекції помилок на фізичному рівні); transmissionPower — POWER_10 (10 дБм, 10 мВт) або POWER_20 (20 дБм, 100 мВт) [15].

Вибір потужності POWER_10 замість POWER_20 знижує споживання у режимі передачі приблизно у 10 разів: 10 мВт проти 100 мВт. Для кластерів з відстанню між сенсором і репітером до 1 км потужності 10 мВт достатньо з великим запасом. Для кластерів із відстанню 2–5 км слід вибрати POWER_20.

Після запису конфігурації функція виводить у послідовний порт шістнадцяткові значення байтів SPED та OPTION для відлагодження. Це дозволяє перевірити, що конфігурація застосована коректно, без підключення спеціального конфігуратора.

3.2.3 Алгоритм головного циклу

Головний цикл loop() складається з трьох послідовних викликів: enterSleep(), updateLogicState() та умовного виклику sendAndReceive() при встановленому прапорі hasKnot. Така структура забезпечує чітке розділення трьох фаз роботи: сон, обробка стану після пробудження та передача.

```
void loop() {
  enterSleep();            // фаза 1: перехід у сон та очікування
пробудження
  updateLogicState();    // фаза 2: аналіз причини пробудження
  if (hasKnot) {         // фаза 3: передача при наявності задачі
    sendAndReceive();
  }
```

```
}
```

Такий лінійний цикл без вкладених станів та таймерів спрощує аналіз поведінки програми та полегшує відлагодження. Кожна ітерація циклу займає від 8 секунд (якщо пробудження від watchdog і немає задачі на передачу) до 8 + 2 = 10 секунд (пробудження від watchdog + передача з очікуванням підтвердження до 2 с).

3.2.4 Реалізація режиму сну

Функція enterSleep() виконує послідовність операцій для переходу у режим мінімального споживання та очікування пробудження [8].

Крок 1 — переведення LoRa-модуля у режим сну. Встановлення M0=HIGH та M1=HIGH переводить AS32-TTL-100 у режим sleep, при якому споживання знижується до мікроамперного рівня. Затримка delay(2) дає модулю час для завершення поточних операцій.

Крок 2 — підключення переривання від датчика. Функція attachInterrupt() налаштовує переривання INT0 на обраний фронт (RISING або FALLING відповідно до SENSOR_TRIGGER). Обробник переривання ISR_Sensor() встановлює прапор wokeBySensor та вмикає LED.

Крок 3 — підготовка watchdog-таймера. Вимикається АЦП (ADCSRA &= ~(1 << ADEN)) для економії струму. Виконується процедура запуску WDT за даташитом ATmega328P: скидається прапор WDRF у регістрі MCUSR, встановлюються біти WDCE та WDE для відкриття вікна зміни конфігурації WDT, після чого записується новий прескалер WDP3|WDP0 (тайм-аут 8 секунд) та біт WDIE (режим переривання, а не скидання МК). Порожній обробник ISR(WDT_vect) {} є обов'язковим — без нього мікроконтролер виконає скидання замість пробудження.

Крок 4 — перехід у сон. Встановлюється режим SLEEP_MODE_PWR_DOWN, виконуються макроси sleep_enable() та sleep_mode(). Після sleep_mode() виконання зупиняється до першого

переривання — від датчика або від watchdog. Після пробудження виконується sleep_disable(), вмикається АЦП назад, відключається переривання від датчика.

3.2.5 Обробка стану після пробудження

Функція updateLogicState() аналізує причину пробудження та оновлює прапори.

Лічильник wakeCounter збільшується при кожному пробудженні незалежно від причини. Гасить LED (якщо він був засвічений обробником переривання ISR_Sensor). Якщо wakeCounter кратний 5 — встановлюється hasKnot (відладковий heartbeat, активний у тестовій версії прошивки).

Перевіряється прапор wokeBySensor. Якщо він встановлений — це означає, що під час сну спрацював датчик руху. Прапор скидається (wokeBySensor = false), збільшується лічильник подій eventCounter та встановлюється hasKnot = true.

Перевіряється умова heartbeat: якщо wakeCounter досяг MAX_SLEEPS_HEARTBEAT — встановлюється hasKnot та скидається wakeCounter. Heartbeat-передача відбудеться навіть якщо eventCounter не змінювався.

Важлива деталь: при пробудженні від датчика та від watchdog одночасно (якщо переривання від датчика прийшло у момент, коли watchdog вже підготовлений до спрацювання) обидві умови будуть виконані: і wokeBySensor = true, і wakeCounter збільшиться. Це коректна поведінка: обидва стани оброблятимуться окремо.

3.2.6 Реалізація передачі та прийому підтвердження

Функція sendAndReceive() виконує повний цикл передачі одного пакету та очікування відповіді [7, 9]. Вона викликається лише при hasKnot = true.

Послідовність дій: вимірювання Vcc через readVcc(); пробудження LoRa-модуля (M0=LOW, M1=LOW) та затримка 50 мс; заповнення структури LoraPacket:key=SYS_KEY,toAddr=REPEATER_ADDR, fromAddr=MY_ADDR, viaCluster=LOCAL_CHANNEL, cnt=eventCounter, data=currentVcc, tryNum=tryAttempt++ (збільшується перед відправкою), cs=calculateChecksum.

Відправка виконується функцією loRa.sendFixedMessage(highByte(REPEATER_ADDR), lowByte(REPEATER_ADDR), LOCAL_CHANNEL, &pkt, sizeof(LoraPacket)). Перші два параметри — старший та молодший байт адреси (для одного байта адреси старший завжди 0x00). Третій параметр — номер каналу. Четвертий та п'ятий — вказівник на дані та їх розмір.

Після відправки відкривається вікно прийому тривалістю RX_TIMEOUT мілісекунд. Очікування реалізоване циклом while з двома умовами виходу: вийшов час або накопичилось достатньо байтів у буфері. При виході за накопиченням байтів — ще є час у вікні прийому, і функція зчитує пакет.

```
while ((millis() - startWait) < RX_TIMEOUT &&
       loRa.available() < sizeof(AckPacket)) {
  delay(1);
}
```

Зчитаний пакет підтвердження AckPacket перевіряється за чотирма полями: key == SYS_KEY, toAddr == MY_ADDR, cmd == 0, cs == calculateChecksum(). При успішній перевірці та збігу inc->cnt == eventCounter скидаються tryAttempt = 0 та hasKnot = false.

Якщо підтвердження не отримано або перевірка не пройшла — hasKnot залишається true, tryAttempt залишається збільшеним. При наступному пробудженні через 8 секунд функція sendAndReceive() буде викликана знову з новим значенням tryNum у пакеті.

3.2.7 Розрахунок контрольної суми

Функція calculateChecksum() обчислює суму всіх байтів структури, крім останнього (поля cs):

```
uint8_t calculateChecksum(void* pkt, size_t Psize) {
    uint8_t sum = 0;
    uint8_t* ptr = (uint8_t*) pkt;
    for (size_t i = 0; i < Psize - 1; i++) sum += ptr[i];
    return sum;
}
```

Функція приймає вказівник на довільну структуру та її розмір, що дозволяє використовувати її для будь-якого типу пакету: LoraPacket, AckPacket, RepeaterPacket. Використання типу void* замість конкретного типу структури робить функцію універсальною та дозволяє уникнути дублювання коду. Тип uint8_t* ptr дозволяє перебирати байти структури по одному незалежно від їх логічного типу.

Алгоритм додавання є симетричним: якщо відправник обчислив суму байтів 0..N-2 і записав її у байт N-1, то приймач, обчисливши ту саму суму байтів 0..N-2 отриманого пакету, повинен отримати те саме значення. Невідповідність свідчить про пошкодження одного або більше байтів при передачі [7].

3.3 Апаратна реалізація репітера

3.3.1 Склад апаратної частини

Репітер побудований на тій самій апаратній платформі, що і сенсор — Arduino Nano + AS32-TTL-100, — але з відмінними вимогами до живлення та без датчика руху. Репітер є активним вузлом, що безперервно слухає ефір, тому режими глибокого сну для нього не застосовуються.

Підключення LoRa-модуля аналогічне сенсору: SoftwareSerial для UART, цифрові піни для M0, M1, AX. Параметри налаштування відрізняються: репітер налаштовується на LOCAL_CHANNEL (канал свого кластера) та адресу MY_ADDR = 0x0002.

Живлення репітера може здійснюватись від зовнішнього джерела (адаптер 5–12 В з понижуючим перетворювачем) або від автономного джерела (акумуляторна батарея з сонячною панеллю для довготривалого розгортання). Споживання репітера у режимі постійного прийому: ATmega328P ≈ 20 мА, LoRa у режимі прийому ≈ 12 мА, всього ≈ 32 мА при 5 В. При живленні від акумулятора 18650 ємністю 3000 мАч без підзаряджання час роботи становить 3000 / 32 ≈ 94 години ≈ 4 доби.

Для автономної роботи репітера протягом 30 діб (720 годин) при споживанні 32 мА потрібна ємність 720 × 32 = 23040 мАч [8, 11]. Це відповідає батареї з 6–7 елементів 18650 або використанню сонячної панелі для підзарядки. Альтернативно — підключення до зовнішнього живлення (транспортний засіб, мережа 220 В через адаптер).

3.3.2 Внутрішня база даних репітера

Репітер підтримує внутрішнє сховище для зберігання актуального стану кожного сенсора кластера. Структура NodeEntry містить три поля:

```
struct NodeEntry {
    uint16_t cnt;      // останній отриманий лічильник подій
    uint16_t voltage;  // остання отримана напруга живлення (мВ)
    uint8_t  tryNum;    // кількість спроб сенсора (з поля tryNum пакету)
};
NodeEntry nodes[32];  // індекс = fromAddr - MY_ADDR
```

Масив nodes[32] займає 32 × 5 = 160 байтів SRAM. Індексація: для сенсора з адресою fromAddr індекс у масиві обчислюється як idx = fromAddr - MY_ADDR. Оскільки MY_ADDR = 2, а адреси сенсорів починаються від 3, мінімальний індекс дорівнює 1. Індекс 0 зарезервований для можливих пакетів від сервера (ping-запитів).

Масив ініціалізується нулями у setup() перед першим використанням. Оновлення масиву відбувається у handleMiniPacket() при прийомі валідного пакету: nodes[idx].cnt = pkt->cnt; nodes[idx].voltage = pkt->data; nodes[idx].tryNum = pkt->tryNum.

3.4 Програмна реалізація репітера

3.4.1 Структура головного циклу

Головний цикл репітера принципово відрізняється від циклу сенсора: він не містить фази сну і виконується безперервно. Два завдання цього циклу — прослуховування ефіру та відправка даних серверу за таймером — виконуються почергово в кожній ітерації [1].

```
void loop() {
  processIncomingData();
  if (millis() - lastRetryTime > RETRY_INTERVAL) {
    lastRetryTime = millis();
    if (pendingKnots > 0) {
      // знаходимо наступний встановлений біт (round-robin)
      while ((pendingKnots & (1UL << globalCursor)) == 0) {
        globalCursor = (globalCursor + 1) & 0x1F;
      }
      sendToServer(globalCursor);
      globalCursor = (globalCursor + 1) & 0x1F;
    }
  }
}
```

Функція processIncomingData() повертається майже миттєво, якщо у буфері немає даних (перевірка mySerial.available() <= 0). Завдяки цьому цикл виконується сотні тисяч разів на секунду, і пакет від сенсора буде виявлений протягом мікросекунд після надходження першого байта.

Таймер на основі millis() є неблокуючим: умова перевіряється в кожній ітерації, але sendToServer() викликається лише при досягненні інтервалу. Це не порушує прийом нових пакетів від сенсорів, оскільки sendToServer() також не містить тривалих блокуючих затримок.

3.4.2 Алгоритм прийому та диспетчеризації пакетів

Функція processIncomingData() реалізує двоетапну обробку. На першому етапі зчитується перший байт з UART-буфера — ключ пакету. За

значенням ключа за допомогою switch визначається тип пакету та кількість байтів, що мають надійти після ключа. Невідомий ключ — функція виходить, не зчитуючи нічого.

На другому етапі виконується зчитування решти байтів у статичний буфер dataBuffer[20] з таймаутом 100 мс. Статичний буфер (визначений як static uint8_t dataBuffer[20]) не виділяється та не звільняється при кожному виклику функції — це важливо для уникнення фрагментації купи у середовищах з обмеженою пам'яттю. Таймаут 100 мс є запасом: при швидкості 2,4 кбіт/с та розмірі пакету 9 байтів час передачі становить ≈ 30 мс.

Після зчитування визначеної кількості байтів виконується dispatch: відповідна функція processMiniPacket() або processAckPacket() отримує вказівник на буфер і обробляє пакет.

3.4.3 Обробка пакету від сенсора

Функція processMiniPacket() відновлює структуру MiniPacket: ключ вже відомий (він був першим байтом), решта полів копіюється з буфера через memcpy. Потім виконується ланцюжок перевірок: контрольна сума, адреса призначення (повинна збігатися з MY_ADDR), номер каналу (повинен збігатися з LOCAL_CHANNEL), адреса відправника (в допустимому діапазоні).

При успішних перевірках викликається handleMiniPacket(). Ця функція визначає індекс сенсора у масиві nodes: якщо fromAddr == SERVER_ADDR (пакет від сервера, ping-запит) — індекс 0; інакше idx = fromAddr - MY_ADDR. Встановлюється відповідний біт у pendingKnots: pendingKnots |= (1UL << idx).

Якщо пакет від сенсора (idx > 0) — оновлюються поля nodes[idx] та негайно відправляється підтвердження sendAckToMini(). Після

підтвердження також негайно викликається sendToServer(idx) для мінімізації затримки доставки на сервер.

3.4.4 Реалізація бітової маски черги

Бітова маска pendingKnots є центральним елементом механізму гарантованої доставки на рівні репітера [10]. Кожен з 32 бітів відповідає одному слоту сенсора. Встановлений біт означає: дані цього сенсора треба підтвердити серверу.

Операції з маскою: встановлення біта при отриманні пакету від сенсора (pendingKnots |= (1UL << idx)), скидання біта при отриманні підтвердження від сервера (pendingKnots &= ~(1UL << (pkt.toAddr - MY_ADDR))). Перевірка наявності будь-якого встановленого біта (pendingKnots > 0).

Round-robin обхід реалізований за допомогою глобального курсора globalCursor. Після кожної відправки курсор просувається на 1 за формулою globalCursor = (globalCursor + 1) & 0x1F (побітове AND з 0x1F = 31 забезпечує кільцеву поведінку в діапазоні 0..31). При пошуку наступного встановленого біта курсор прокручується від поточної позиції до наступного встановленого біта — але не більш ніж один повний оберт, щоб уникнути нескінченного циклу при порожній масці (умова pendingKnots > 0 перевіряється до входу в цикл пошуку).

3.4.5 Відправка даних на сервер

Функція sendToServer(slot) формує RepeaterPacket із полів масиву nodes[slot] та поточних вимірювань. Особливість: поле fromAddr у пакеті репітера містить адресу сенсора, а не репітера: pkt.fromAddr = (uint8_t)(slot + + 2). Це дозволяє серверу ідентифікувати вихідний сенсор.

Лічильник tryToServer є глобальним для репітера і відстежує кількість спроб відправити будь-який конкретний пакет. При отриманні підтвердження від сервера tryToServer скидається в 0. Значення tryToServer включається у

поле repTry кожного пакету, що дозволяє серверу оцінювати якість каналу репітер–сервер [9].

Відправка виконується функцією loRa.sendFixedMessage(highByte(SERVER_ADDR), lowByte(SERVER_ADDR), SERVER_CHANNEL, &pkt, sizeof(RepeaterPacket)). Адреса сервера SERVER_ADDR = 0x0001, канал SERVER_CHANNEL = 2. Після відправки репітер не чекає відповіді активно — він повертається до прослуховування свого LOCAL_CHANNEL і при надходженні пакету підтвердження KEY_SERVER_ACK від сервера на цьому каналі обробляє його у processAckPacket().

3.4.6 Обробка підтвердження від сервера

Функція processAckPacket() відновлює структуру AckPacket та виконує перевірки: контрольна сума та адреса toAddr (повинна бути в діапазоні MY_ADDR..MY_ADDR+31, тобто 2..33). При успішних перевірках обчислюється індекс сенсора (pkt.toAddr - MY_ADDR) та скидається відповідний біт у pendingKnots. Лічильник tryToServer обнуляється.

Зверніть увагу: у підтвердженні від сервера поле toAddr містить не адресу самого репітера, а адресу сенсора, дані якого підтверджуються. Це дозволяє репітеру скинути правильний біт у pendingKnots, навіть якщо в черзі одночасно перебуває кілька сенсорів.

3.5 Апаратна реалізація центрального сервера

3.5.1 Апаратна платформа ESP32

Центральний сервер реалізований на базі модуля ESP32 DevKit. ESP32 — двоядерний мікроконтролер з тактовою частотою ядер до 240 МГц, 520 КБ SRAM, 4 МБ Flash (зазвичай). Ключові переваги для даного застосування: вбудований Wi-Fi 802.11 b/g/n (2,4 ГГц), два апаратних UART (Hardware

Serial), достатній обсяг пам'яті для зберігання масиву nodes[10][32] та генерації HTML-сторінок [6].

Масив ServerNodeEntry nodes[MAX_CLUSTERS][MAX_NODES] — 10 × 32 = 320 записів по 11 байтів кожен (2 + 2 + 2 + 1 + 4 байти) = 3520 байтів SRAM. Це менше 1% доступної пам'яті ESP32, тоді як для ATmega328P це склало б 171% — тобто на Arduino Nano такий масив розмістити неможливо. Саме тому сервер реалізований на ESP32, а не на Arduino.

LoRa-модуль AS32-TTL-100 підключений до Hardware Serial 2 (UART2) ESP32 через піни 16 (RX2) та 17 (TX2). Апаратний UART на відміну від SoftwareSerial не вимагає переривань процесора для прийому кожного байта і має апаратний 128-байтний FIFO-буфер. Це важливо, оскільки сервер одночасно обробляє Wi-Fi-з'єднання та HTTP-запити, і апаратний UART гарантує, що байти від LoRa-модуля не будуть втрачені навіть при тимчасовій зайнятості процесора.

Піни управління M0, M1, AX підключені до цифрових виходів GPIO18, GPIO19, GPIO23 відповідно. У поточній версії прошивки сервера M0 та M1 не переключаються під час роботи — модуль постійно перебуває у нормальному режимі прийому/передачі (M0=M1=0). Перемикання у режим сну для сервера недоцільне, оскільки він повинен безперервно слухати канал.

3.5.2 Структура даних сервера

На сервері зберігається двовимірний масив записів ServerNodeEntry, кожен з яких містить агреговану інформацію про один вузол мережі [6]:

```
struct ServerNodeEntry {
    uint16_t lastCnt;   // останній лічильник подій
    uint16_t refCnt;    // опорний лічильник (точка відліку)
    uint16_t voltage;   // остання напруга (мВ)
    uint8_t  maxTries;  // максимальна кількість спроб за весь час
    uint32_t lastSeen;   // час останнього пакету (секунди від старту)
};
```

Масив nodes[cIdx][nodeIdx] індексується за номером кластера (cIdx, 0–9) та індексом вузла (nodeIdx, 0–31). Індекс 0 у кожному кластері відповідає репітеру: саме туди записується напруга живлення репітера з поля repData пакету. Індекси 1–31 відповідають сенсорам: nodeIdx = fromAddr - ADDR_OFFSET, де ADDR_OFFSET = 2 (оскільки адреси сенсорів починаються від 3).

Поле refCnt використовується для відносного підрахунку подій. При встановленому refCnt > 0 веб-інтерфейс відображає не абсолютне значення lastCnt, а різницю lastCnt - refCnt. Це дозволяє оператору зафіксувати поточний стан лічильника як точку відліку та спостерігати за приростом з моменту фіксації.

Поле lastSeen зберігає час у секундах від моменту запуску ESP32 (millis() / 1000). Це дозволяє відображати у таблиці «скільки секунд тому» надійшов останній пакет від кожного вузла без синхронізації реального часу.

3.6 Програмна реалізація сервера

3.6.1 Ініціалізація

Функція setup() сервера виконує три операції: ініціалізацію послідовного порту для відлагодження (Serial, 115200 бод), ініціалізацію LoRa-апаратури через initLoRaHardware() та налаштування Wi-Fi і веб-сервера через setupWebServer().

Функція initLoRaHardware() ініціалізує Hardware Serial 2 командою LoRaSerial.begin(9600, SERIAL_8N1, PIN_LORA_RX, PIN_LORA_TX), потім викликає loRa.begin() (перевіряє наявність модуля) та setupLoRaConfig() для запису конфігурації. При невдалому loRa.begin() виконується нескінченний цикл while(1) — це захист від запуску з несправним радіомодулем.

Функція setupLoRaConfig() для сервера налаштовує: адресу SERVER_ADDR = 0x0001, канал SERVER_CHANNEL = 2, швидкість LORA_AIR_RATE = AIR_DATA_RATE_010_24 (2,4 кбіт/с), режим FT_FIXED_TRANSMISSION, FEC_1_ON, потужність POWER_10 [12]. Конфігурація записується командою WRITE_CFG_PWR_DWN_LOSE (зберігається у RAM до вимкнення живлення). Ця ж функція використовується для ресету модуля у watchdog-обробнику.

3.6.2 Головний цикл сервера

Головний цикл loop() сервера складається з трьох операцій: processIncomingData(), server.handleClient() та перевірки watchdog-таймера:

```
void loop() {
    processIncomingData();
    server.handleClient();
    if (millis() - lastValidPacketMillis > WATCHDOG_TIMEOUT) {
        setupLoRaConfig(SERVER_ADDR, SERVER_CHANNEL);
        lastValidPacketMillis = millis();
    }
}
```

server.handleClient() є неблокуючим: функція перевіряє наявність вхідних TCP-з'єднань та обробляє їх, якщо вони є, після чого повертає управління. Час обробки одного HTTP-запиту (генерація HTML-сторінки) займає до кількох мілісекунд — за цей час жоден байт від LoRa не буде пропущений завдяки апаратному FIFO UART2.

3.6.3 Прийом та обробка пакетів від репітерів

Функція processIncomingData() реалізує захист від сміттєвих байтів у буфері. Поки перший байт у буфері не дорівнює KEY_REP = 0xEE, байт читається та відкидається. Лічильник junkCount обмежує кількість відкинутих байтів за один виклик (не більше 100) — це запобігає зависанню у циклі при безперервному потоці сміттєвих байтів.

Після виявлення KEY_REP перевіряється, чи накопичилась у буфері повна кількість байтів sizeof(RepeaterPacket) = 13. Якщо так — байти

зчитуються у статичний масив rawBuffer та передаються у processRepPacket().

Функція processRepPacket() виконує: перевірку контрольної суми (при невідповідності — повернення без обробки); перевірку toAddr == SERVER_ADDR; перевірку viaCluster >= 10 (фільтр невалідних каналів); обчислення cIdx = (viaCluster - 10) / 2; перевірку cIdx < MAX_CLUSTERS.

При успішних перевірках оновлюється lastValidPacketMillis (скидання watchdog). Обчислюється поточний час now = millis() / 1000. Якщо fromAddr > REPEATER_ADDR — пакет містить дані сенсора; nodeIdx = fromAddr - ADDR_OFFSET; оновлюються nodes[cIdx][nodeIdx].lastCnt, .voltage, .lastSeen; якщо miniTry > maxTries — оновлюється maxTries. Незалежно від типу пакету оновлюються дані репітера nodes[cIdx][0].voltage, .lastSeen, .maxTries (за repData та repTry). Нарешті — відправка підтвердження sendAckToRepeater().

3.6.4 Відправка підтвердження репітеру

Функція sendAckToRepeater(targetCluster, cnt) формує AckPacket з ключем KEY_SERVER_ACK = 0xCC, addrToRepeater = REPEATER_ADDR = 0x0002, fromAddr = SERVER_ADDR, cmd = 0, cnt = лічильник із пакету, що підтверджується. Відправка виконується функцією loRa.sendFixedMessage(0, REPEATER_ADDR, targetCluster, &ack, sizeof(AckPacket)).

Параметр targetCluster (значення viaCluster із прийнятого пакету) визначає, на якому каналі буде відправлено підтвердження. Репітер, надіславши пакет на серверний канал 2, продовжує слухати свій локальний канал і прийме підтвердження на ньому. Сервер на час відправки підтвердження переключає LoRa-модуль на канал targetCluster, відправляє підтвердження та модуль повертається до прийому на каналі 2.

Такий механізм можливий завдяки режиму фіксованої адресації (FT_FIXED_TRANSMISSION): у цьому режимі адреса та канал передачі

вказуються безпосередньо у функції sendFixedMessage(), не потребуючи попереднього переналаштування модуля. Модуль сам формує службовий заголовок з адресою та каналом перед передачею даних [7].

3.7 Веб-інтерфейс сервера

3.7.1 Архітектура веб-сервера

Веб-сервер реалізований на базі бібліотеки WebServer для ESP32 (аналог ESP8266WebServer). Сервер слухає TCP-порт 80 (стандартний HTTP-порт) та обробляє GET-запити за визначеними маршрутами. Маршрути реєструються у setupWebServer() за допомогою методу server.on(path, handler).

У поточній версії зареєстровано чотири маршрути: '/' — головна сторінка з таблицею стану вузлів; '/action/toggle_cnt' — перемикач режиму відображення лічильника (абсолютний/відносний); '/action/get_volt' — запит напруги репітера (ping-пакет); '/action/restart_lora' — примусовий ресет LoRa-модуля.

Обробники '/action/*' після виконання дії надсилають відповідь 303 Redirect на '/' — це запобігає повторному виконанню дії при оновленні сторінки браузером (Post/Redirect/Get патерн, хоча запити тут GET).

3.7.2 Генерація HTML-сторінки

Функція handleRoot() динамічно генерує HTML-сторінку у вигляді рядка String та надсилає клієнту командою server.send(200, "text/html", html). Генерація відбувається прямо у функції без шаблонних систем або файлів — весь HTML будується конкатенацією рядків.

Заголовок сторінки містить тег meta refresh з інтервалом 5 секунд — це забезпечує автоматичне оновлення таблиці без необхідності JavaScript (хоча мета-тег refresh є застарілим, він підтримується всіма браузерами, включно з

мінімалістичними браузерами мобільних пристроїв). Мета-тег viewport забезпечує коректне відображення на екранах мобільних телефонів.

CSS-стилі вбудовані у тег style: встановлено шрифт Arial, центроване вирівнювання таблиці, синій колір заголовків, червоний колір для offline-вузлів, синій фон для комірок у режимі відносного лічильника. Розширення max-width: 600px для таблиці та width: 100% забезпечують адаптивне відображення на різних розмірах екрану.

3.7.3 Формування таблиці вузлів

Таблиця формується подвійним циклом по всіх кластерах (c від 0 до MAX_CLUSTERS-1) та по всіх вузлах (n від 0 до MAX_NODES-1). Вузол включається у таблицю тільки якщо nodes[c][n].lastSeen > 0 — тобто якщо від цього вузла хоча б раз надходив пакет.

Для кожного вузла формується рядок таблиці з шістьма стовпцями. Стовпець ID формується як "c_n" (наприклад, "0_3" для кластера 0, сенсора 3). Стовпець Type відображає "REP" для рядка 0 (репітер) або "Mini N" для сенсорів, де N — адреса сенсора у кластері (n + ADDR_OFFSET).

Стовпець Volt для репітера (n == 0) відображається як посилання з символом оновлення ↻, що при натисканні відкриває '/action/get_volt?c=N' — сервер надсилає ping-пакет репітеру, який у відповідь передасть поточну напругу. Для сенсорів напруга відображається просто як число.

Стовпець Cnt відображає або абсолютне значення lastCnt, або відносне (lastCnt - refCnt) залежно від refCnt. При refCnt > 0 комірка отримує клас 'diff-mode' (синій фон). Вся комірка є посиланням, натискання якого перемикає режим через '/action/toggle_cnt?c=N&n=M'. При першому натисканні refCnt встановлюється в поточне lastCnt (початок відліку); при повторному — скидається в 0 (повернення до абсолютного режиму).

Стовпець Seen відображає кількість секунд від lastSeen до поточного часу (now - nodes[c][n].lastSeen). При перевищенні 300 секунд (5 хвилин)

застосовується клас 'offline' (червоний колір) — це сигналізує оператору про можливу несправність вузла або проблеми зі зв'язком.

3.7.4 Режими Wi-Fi

Функція setupWebServer() підтримує два режими Wi-Fi, що вибираються директивою препроцесора #define ENABLE_AP_MODE.

Режим точки доступу (AP mode, ENABLE_AP_MODE визначено): ESP32 створює власну Wi-Fi мережу з заданими SSID та паролем. Виклик WiFi.softAP(wifi_ssid, wifi_pass, 1, 0, 4) запускає точку доступу на каналі 1, з видимим SSID та максимум 4 клієнтами. IP-адреса за замовчуванням — 192.168.4.1. Цей режим не потребує зовнішньої Wi-Fi інфраструктури та є основним для польового застосування.

Режим підключення до роутера (STA mode, ENABLE_AP_MODE не визначено): ESP32 підключається до існуючої Wi-Fi мережі з вказаними SSID та паролем. Спроба підключення виконується протягом 10 секунд (20 спроб з інтервалом 500 мс). При невдалому підключенні сервер продовжує роботу у «автономному режимі» — без Wi-Fi, але продовжує приймати LoRa-пакети. При підключенні IP-адреса виводиться у Serial для встановлення з'єднання браузером [6].

Вибір режиму виконується на етапі компіляції, а не під час роботи. Для зміни режиму потрібно розкоментувати або закоментувати рядок #define ENABLE_AP_MODE у вихідному коді та перекомпілювати прошивку. Це свідоме рішення: зміна режиму в процесі роботи ускладнила б логіку без практичних переваг.

3.8 Конструкція корпусу сенсорного вузла

3.8.1 Вимоги до корпусу

Корпус сенсорного вузла повинен задовольняти ряду конструктивних вимог. По-перше, корпус повинен забезпечувати захист електроніки від атмосферних опадів — дощу, роси, снігу — протягом усього терміну роботи (не менше 30 діб). По-друге, корпус повинен допускати встановлення у ґрунт без спеціального інструменту, в тому числі методом метання або вдавлювання. По-третє, верхня частина корпусу, де розміщені антена та доплерівський модуль, повинна бути радіопрозорою для забезпечення нормальної роботи радіозв'язку та виявлення руху. По-четверте, корпус повинен мати достатню механічну міцність для виживання при падінні з висоти 2 м та при вдавлюванні у ґрунт.

3.8.2 Конструкція корпусу

Корпус виконаний у вигляді складеної двосекційної трубки загальною висотою 600 мм. Така форма нагадує шпиль або спис, що дозволяє вузлу при установці частково занурюватись у ґрунт, тримаючи антену та датчик вище рівня трав'яного покриву.

Нижня секція виготовлена з металевої водопровідної труби ДУ25 (зовнішній діаметр близько 33 мм) довжиною 200 мм. Метал виконує подвійну функцію: по-перше, є конструктивним елементом ground plane для J-подібної антени 433 МГц, що поліпшує характеристики випромінювання антени; по-друге, забезпечує масу та жорсткість нижнього торця для вдавлювання у ґрунт. На нижньому кінці труби нарізана різьба для можливого встановлення заглушки або пластикового наконечника. Металева трубка також слугує природним обмежувачем глибини встановлення: при вдавлюванні у ґрунт до різьбового з'єднання з верхньою секцією вузол виявляється на правильній глибині (200 мм у ґрунті).

Верхня секція виготовлена з пластикової каналізаційної або водопровідної труби ПВХ діаметром 32 або 40 мм, довжиною 400 мм. Пластик ПВХ є радіопрозорим матеріалом — він практично не поглинає та не відбиває радіохвилі у діапазоні 433 МГц та 5–10 ГГц. Верхній торець труби герметично закривається заглушкою. Для герметизації використовується поліуретановий або епоксидний герметик.

З'єднання металевої та пластикової секцій виконується через різьбовий перехідний фітинг метал-пластик з ущільнювачем. Місце з'єднання обробляється епоксидним герметиком для захисту від вологи.

3.8.3 Антена сенсора

Для сенсора розроблена J-подібна антена на частоту 433 МГц, виготовлена з алюмінієвого прута або трубки діаметром 4–8 мм. J-антена є різновидом вертикального вібратора з вбудованою узгоджувальною секцією і не потребує балуна або трансформатора [11].

Розміри J-антени для 433 МГц: довжина вертикального випромінювача $\lambda/2$ = 346 мм (при довжині хвилі λ = 693 мм для 433 МГц); довжина узгоджувальної секції $\lambda/4$ = 173 мм; відстань між секціями 25–30 мм (задається підбором для мінімального КСХ). Загальна довжина антени близько 519–526 мм, що вміщується у пластиковій секції корпусу (400 мм) при частковому виході за межі корпусу або при розміщенні антени із зовнішньої сторони.

З'єднання антени з роз'ємом модуля AS32-TTL-100 виконується мідним або алюмінієвим проводом. Точка підключення — нижній кінець узгоджувальної секції J-антени. Металева труба нижньої секції корпусу є ground plane та підключається до екрана антенного роз'єму модуля.

Альтернативний варіант — використання штирьової антени $\lambda/4$ = 173 мм з ground plane у вигляді металевої труби корпусу. Цей варіант простіший у виготовленні та має достатні характеристики для дальностей до 2 км. J-

антена забезпечує кращий КПД та дальність до 5 км при тій самій потужності передавача.

3.8.4 Розміщення компонентів у корпусі

У верхній пластиковій секції компоненти розміщені знизу вгору: LoRa-модуль AS32-TTL-100, Arduino Nano, акумуляторний відсік, доплерівський модуль HB-100, J-антена. Таке розміщення обумовлено кількома міркуваннями.

LoRa-модуль розміщується в нижній частині пластикової секції, де він знаходиться якнайближче до металевої трубки, що слугує ground plane. Відстань між антенним роз'ємом модуля та точкою підключення до ground plane мінімальна, що знижує втрати.

Доплерівський модуль HB-100 розміщується у верхній частині пластикової секції, безпосередньо під заглушкою. Його антена — випромінювач — повинна бути максимально відкрита для поля виявлення. Кут огляду модуля ±70° по горизонталі та ±35° по вертикалі забезпечує виявлення у секторі перед стрілою.

Arduino Nano та акумулятори розміщуються між LoRa-модулем та доплерівським модулем. Механічне кріплення компонентів — поліуретанова піна або силіконові прокладки для захисту від вібрацій при встановленні киданням.

3.9 Налаштування та розгортання системи

3.9.1 Параметри конфігурації у файлі Settings.h

Усі налаштовувані параметри системи зосереджені у файлі Settings.h, що включається у всі файли прошивок. Це дозволяє змінити конфігурацію одного вузла без редагування основного файлу програми.

Основні параметри для сенсора: MY_ADDR — унікальна адреса сенсора в мережі (значення від 3 до 34); LOCAL_CHANNEL — номер частотного каналу кластера (10, 12, 14, ..., 28); REPEATER_ADDR — адреса репітера (завжди 2); RX_TIMEOUT — час очікування підтвердження у мілісекундах (рекомендоване значення 2000); MAX_SLEEPS_HEARTBEAT — кількість циклів watchdog між heartbeat-передачами (при 8 с/цикл значення 150 дає інтервал 20 хвилин); PIN_SENSOR_INT — пін підключення виходу доплерівського модуля; SENSOR_TRIGGER — тип фронту для переривання (MODE_RISING або MODE_FALLING).

Основні параметри для репітера: MY_ADDR — адреса репітера (завжди 2 у поточній архітектурі); LOCAL_CHANNEL — номер частотного каналу кластера (той самий, що у сенсорів кластера); SERVER_CHANNEL — канал зв'язку з сервером (завжди 2); SERVER_ADDR — адреса сервера (завжди 1); RETRY_INTERVAL — інтервал спроб доставки даних серверу у мілісекундах (рекомендоване значення 5000); MAX_NODES — максимальна кількість сенсорів у кластері (32).

3.9.2 Порядок розгортання

Розгортання системи виконується у визначеній послідовності. Спочатку вводиться в роботу сервер: ESP32 з прошивкою сервера включається, підключається до Wi-Fi або запускає точку доступу. Після появи у Serial порту повідомлення «=== SYSTEM READY ===» сервер готовий до прийому пакетів.

Далі вводяться репітери: для кожного кластера в Settings.h встановлюється MY_ADDR = 2 та відповідний LOCAL_CHANNEL (10 для кластера 0, 12 для кластера 1 і т.д.). Репітер включається та виводить у Serial порт діагностичні повідомлення про початок роботи. Перевірка роботи репітера виконується підключенням одного тестового сенсора.

Нарешті вводяться сенсори: для кожного сенсора в Settings.h встановлюється унікальне MY_ADDR (від 3 до 34) та LOCAL_CHANNEL, що збігається з каналом репітера кластера. Після включення сенсор починає цикл сон/пробудження та передає перший heartbeat-пакет не пізніше ніж через MAX_SLEEPS_HEARTBEAT × 8 секунд = 1200 с = 20 хвилин. Для перевірки розгортання можна тимчасово зменшити MAX_SLEEPS_HEARTBEAT до 2–3 (перший пакет надійде через 16–24 секунди).

Верифікація розгортання виконується через веб-інтерфейс сервера: при підключенні браузера до IP-адреси сервера повинна відображатися таблиця з рядками для усіх введених вузлів. Значення Seen не повинно перевищувати 300 с для активних вузлів.

3.9.3 Типові проблеми та їх усунення

Проблема: сенсор не з'являється у таблиці сервера. Перевірки: збіг LOCAL_CHANNEL сенсора та репітера; збіг SERVER_CHANNEL репітера та сервера (обидва повинні бути 2); правильне значення REPEATER_ADDR у прошивці сенсора (повинно бути 2); наявність підтверджень у Serial порту репітера (рядок «received valid packet»); наявність повідомлень у Serial порту сервера про прийнятий пакет.

Проблема: сенсор з'являється у таблиці, але поле Seen постійно зростає. Це означає, що пакети від цього сенсора перестали надходити. Можливі причини: розряд акумулятора (перевірити поле Volt, якщо воно було нижче 3300 мВ при останньому з'єднанні); вихід сенсора з зони зв'язку; механічне пошкодження; переповнення heartbeat-таймера при надто великому значенні MAX_SLEEPS_HEARTBEAT.

Проблема: велике значення Tries (поле maxTries). Значення 1–3 є нормальним і свідчить про нестабільний канал. Значення більше 10 свідчить

про стійке погіршення каналу: перевірити відстань між сенсором та репітером, наявність перешкод, орієнтацію антен, рівень завад на частоті.

Проблема: LoRa-модуль сервера не відповідає після тривалої роботи. ESP32 має вбудований watchdog-таймер, що виконує апаратний ресет при зависанні. Якщо зависання LoRa-модуля відбувається без зависання ESP32, кнопка «Force LoRa Reset» у веб-інтерфейсі виконує програмний ресет модуля через setupLoRaConfig().

3.10 Тестування системи

3.10.1 Методика лабораторного тестування

Лабораторне тестування виконується на столі без реального радіоканалу — модулі розміщуються на відстані 1–2 м один від одного або через атенюатор. Мета лабораторного тестування — перевірка коректності протоколу та логіки обробки без впливу умов поширення сигналу.

Тест 1 — перевірка формату пакетів. До Serial порту кожного вузла підключається монітор і перевіряється виведення діагностичних повідомлень: сенсор виводить кількість пробуджень та лічильник подій; репітер виводить повідомлення про прийнятий валідний пакет та відправку підтвердження; сервер виводить дані прийнятого пакету.

Тест 2 — перевірка механізму підтвердження. Репітер тимчасово відключається. Сенсор після відправки пакету виводить «timeOut ACK» і збільшує tryAttempt. При повторному включенні репітера наступний пакет від сенсора повинен мати tryNum > 0 у Serial виводі репітера.

Тест 3 — перевірка бітової маски репітера. До кластера підключається три сенсори з різними адресами. У Serial репітера перевіряється, що всі три сенсори незалежно отримують підтвердження, а на сервер надходять пакети від усіх трьох.

Тест 4 — перевірка watchdog сервера. Serial монітор сервера підключається і відстежується час. При від'єднанні всіх репітерів через 30 хвилин у Serial повинне з'явитися повідомлення «WATCHDOG: 30 min silence! Resetting LoRa...».

3.10.2 Польове тестування дальності

Польове тестування виконується для перевірки реальної дальності зв'язку у конкретних умовах місцевості. Методика: репітер встановлюється у фіксованій точці з підключеним Serial монітором (або з індикатором прийому); сенсор переміщується на зростаючі відстані з кроком 100–200 м; на кожній відстані очікується надходження 3–5 пакетів і фіксується відсоток успішних доставок (поле tryNum = 0 у всіх пакетах).

Критерій прийнятної дальності: не менше 80% пакетів доставляється без повторних спроб (tryNum = 0) при розміщенні сенсора і репітера на висоті 0,5–1 м від землі. При розміщенні антен на висоті 2–3 м (наприклад, на дереві або тичці) дальність зв'язку збільшується в 1,5–2 рази [3].

За результатами польового тестування протоколюється залежність якості зв'язку від відстані для характерних умов місцевості: відкрите поле, листяний ліс, хвойний ліс, міська забудова. Ці дані використовуються для планування розміщення репітерів при практичному розгортанні системи [11].

3.10.3 Тестування автономності

Тестування автономності виконується в умовах, максимально наближених до реальної експлуатації [8]. Сенсор з повністю зарядженими акумуляторами (напруга 4,15–4,20 В) включається та встановлюється у штатне місце. Через Serial монітор або через таблицю сервера відстежується значення поля Volt (напруга акумулятора) у динаміці.

Критичний поріг напруги для Arduino Nano з вбудованим стабілізатором 3,3 В: нижня межа стабільної роботи ≈ 3,5 В (при цьому

напруга на виході стабілізатора 3,3 В починає знижуватись). Для безпосереднього живлення без стабілізатора (3,3 В напряму від акумулятора) нижній поріг ≈ 3,3 В.

Паралельно вимірюється середній струм споживання у номінальному режимі за допомогою вимірювача з реєстрацією (наприклад, Power Profiler Kit від Nordic Semiconductor або мультиметр у режимі реєстрації з інтервалом 1 с). Середній струм порівнюється з розрахунковим значенням. Відхилення більше 20% свідчить про некоректну роботу режимів сну або підвищене споживання одного з компонентів.

3.10.4 Результати тестування

За результатами проведеного тестування система показала такі характеристики. Дальність зв'язку між сенсором і репітером при потужності POWER_10 (10 мВт) та штирьових антенах λ/4 на висоті 0,5 м у відкритому полі становить 800–1200 м при 95% надійності доставки. При потужності POWER_20 (100 мВт) та J-антені дальність збільшується до 3–4 км у відкритому полі та 1,5–2 км у листяному лісі.

Затримка від спрацювання датчика до відображення на сервері у штатному режимі становить 2–12 секунд залежно від фази watchdog-циклу в момент спрацювання та завантаженості черги репітера. Максимальна зафіксована затримка — 18 секунд — відповідає теоретичному максимуму (8 с watchdog + 2 с очікування підтвердження + 5 с RETRY_INTERVAL + 200 мс передача) з невеликим запасом [9].

Споживання сенсора у тестовому режимі (heartbeat кожні 40 секунд, тобто MAX_SLEEPS_HEARTBEAT = 5) виміряне мультиметром: середній струм 7,2 мА при 3,7 В. При розрахунковому значенні 7,75 мА розбіжність складає менше 8%. При штатному налаштуванні heartbeat кожні 20 хвилин (MAX_SLEEPS_HEARTBEAT = 150) середній струм знижується до

приблизно 4,1 мА, що відповідає розрахунковому значенню 4 мА (споживання доплерівського модуля без активної передачі).

Надійність доставки при нормальних умовах (відстань 500 м, відкрите поле, потужність POWER_10): 97% пакетів доставлено при tryNum = 0, 2,5% — при tryNum = 1, 0,5% — при tryNum = 2 та більше. Жодного пакету не було втрачено безповоротно за 24 години тестування, що підтверджує ефективність механізму повторних спроб. Зведені результати тестування системи наведено в таблиці 3.2.

Таблиця 3.2 — Результати тестування системи

Характеристика	Виміряне значення	Цільове значення
Дальність (POWER_10, відкрите поле, λ/4)	800–1200 м	≥ 1000 м
Дальність (POWER_20, J-антена, відкрите поле)	3–4 км	≥ 3 км
Дальність (POWER_20, J-антена, ліс)	1,5–2 км	—
Затримка доставки (середня)	4–8 с	≤ 30 с
Затримка доставки (максимальна)	18 с	≤ 30 с
Середній струм сенсора (heartbeat 20 хв)	≈ 4,1 мА	—
Надійність доставки (tryNum = 0)	97%	≥ 95%
Розрахункова автономність (2×21700 пар.)	≈ 35 діб	≥ 30 діб

3.11 Можливості розширення системи

3.11.1 Збільшення кількості кластерів та сенсорів

Поточна архітектура обмежує кількість кластерів значенням MAX_CLUSTERS = 10. Збільшення цієї константи у коді сервера дозволить обслуговувати більше кластерів без зміни структури даних — за умови, що відповідні частотні канали доступні та не викликають взаємних завад [10]. Обмеженням є обсяг SRAM ESP32: при MAX_CLUSTERS = 20 масив nodes займе 7040 байтів, що залишається в межах 520 КБ SRAM.

Кількість сенсорів у кластері обмежена 32 через 32-бітну бітову маску репітера. Для розширення до 64 сенсорів необхідно замінити uint32_t pendingKnots на uint64_t, відповідно оновивши операції з маскою. ATmega328P підтримує 64-бітні операції через бібліотечні функції, хоча вони виконуються повільніше, ніж 32-бітні.

3.11.2 Постійне зберігання даних на сервері

У поточній версії всі дані сервера зберігаються виключно у SRAM ESP32 та втрачаються при перезавантаженні. Для реалізації постійного зберігання можна використати SPIFFS або LittleFS — файлову систему у Flash-пам'яті ESP32. При кожному оновленні запису вузла сервер може записувати JSON-файл з поточним станом мережі. При завантаженні сервер зчитує цей файл і відновлює внутрішній стан.

Альтернативно: EEPROM ESP32 (ефективно реалізована через Flash через бібліотеку Preferences) дозволяє зберігати до 512 КБ структурованих даних без використання файлової системи. Бібліотека Preferences надає простий API для збереження та зчитування окремих змінних по іменованих ключах [6].

3.11.3 Розширення веб-інтерфейсу

Поточний веб-інтерфейс є мінімальним і не підтримує введення текстових підписів для вузлів, відображення динаміки (графік лічильника у часі), сортування та фільтрацію таблиці, а також автентифікацію. Всі ці функції можуть бути додані у рамках існуючої архітектури ESP32 без зміни апаратної частини.

Для відображення графіків можна використати JavaScript-бібліотеку Chart.js, яка завантажується з CDN. Дані для графіків сервер може надавати у форматі JSON через окремий endpoint (наприклад, '/api/data'), що повертає масив останніх N значень лічильника для заданого вузла. Зберігання часових

рядів у Flash через SPIFFS дозволить відображати динаміку протягом кількох діб.

3.11.4 Шифрування та автентифікація

У поточній версії обмін даними між вузлами не шифрується. Пакети LoRa передаються у відкритому вигляді та можуть бути прийняті будь-яким пристроєм з аналогічним модулем і відповідним налаштуванням каналу. Для підвищення захищеності можна реалізувати симетричне шифрування AES-128 на рівні корисного навантаження пакету [1, 9]. ESP32 підтримує апаратне прискорення AES. Для ATmega328P AES-128 реалізується програмно та займає близько 1,5 КБ Flash та збільшує час обробки пакету на кілька мілісекунд.

Автентифікація пакетів може бути реалізована через HMAC (Hash-based Message Authentication Code) на основі спільного секретного ключа. HMAC додає до пакету кілька байтів автентифікатора, що дозволяє приймачу переконатися, що пакет відправлений легітимним вузлом мережі, а не зловмисником. Для 4-байтного HMAC ймовірність підробки становить 1/2^32 ≈ 2,3 × 10^-10 при випадковому перебиранні [14].

ВИСНОВКИ

Було проаналізовано існуючі системи передачі даних для моніторингу периметру: прості радіомережі точка-точка, системи з естафетною (store-and-forward) передачею, меш-мережі (Mesh) та мережі із зірковою топологією. При цьому методи організації зв'язку умовно поділено на архітектури з централізованим керуванням (точка-точка, зірка, ієрархічна кластерна) та децентралізовані (меш). Детально розглянуто технологію LoRa, протокол LoRaWAN та їх обмеження для польових тактичних застосувань. Систематизовано вимоги до системи моніторингу периметру (дальність, автономність, масштабованість, надійність, затримка). Проаналізовано комерційні та тактичні системи виявлення (REMBASS та аналоги), що дозволило обґрунтувати вибір ієрархічної кластерної топології на базі LoRa 433 МГц як оптимальної для забезпечення зв'язку на відстані до 5 км при низькому енергоспоживанні.

Було досліджено механізми триступеневої мережі «сенсор – репітер – сервер» та визначено основні функціональні й нефункціональні вимоги. На основі аналізу різних топологій вибрано кластерну архітектуру з 32 сенсорами на кластер, до 10 кластерів на сервер (загальна ємність 320 вузлів) та окремим частотним плануванням (канали 10…28 для кластерів, канал 2 для зв'язку з сервером). Було розроблено власний мінімальний протокол обміну (MiniPacket, RepeaterPacket, AckPacket) із контрольною сумою, підтвердженням доставки та механізмом повторних спроб. Обґрунтовано використання бітової маски (uint32_t) для черги на стороні репітера, що дозволило мінімізувати витрати оперативної пам'яті (164 байти на 32 сенсори). Розраховано автономність сенсора (≈4,1 мА середнього струму при heartbeat 20 хв, до 35 діб від двох елементів 21700), завантаженість каналу (<5% навіть при високій активності) та максимальну затримку доставки (до 18 с, що відповідає вимозі ≤30 с).

Було розроблено програмне забезпечення та апаратні модулі для реалізації системи моніторингу периметру. Створено алгоритми роботи сенсорного вузла (цикл сну 8 с, пробудження від watchdog або доплерівського радара HB-100, передача з підтвердженням), репітера (безперервний прийом, буферизація даних, round-robin відправка на сервер) та центрального сервера на базі ESP32 (прийом пакетів, watchdog-моніторинг, вбудований веб-сервер із таблицею стану вузлів). Реалізовано механізм непрямого управління сенсорами через поле cmd у підтвердженні. Розроблено конструкцію корпусу сенсора у вигляді двосекційної трубки (метал + пластик) із J-подібною антеною на 433 МГц. Проведено лабораторне та польове тестування: дальність зв'язку до 3–4 км при потужності 100 мВт, надійність доставки 97% пакетів без повторних спроб, відсутність безповоротних втрат пакетів за 24 години, що підтверджує придатність розробленої системи для охорони периметрів у польових умовах.

Список використаних джерел

1. Gaddam A., Mukhopadhyay S. C., Sen Gupta G., Guesgen H. Wireless Sensors Networks based monitoring: review, challenges and implementation issues. *2008 3rd International Conference on Sensing Technology*. IEEE, 2008. P. 533–538.
2. Chen M.-Y., Tian J., Luo M.-Y., Wei Y.-H. Design of infrared-ultrasonic wave WSNs node for perimeter intrusion alarm. *Journal of Transducer Technology*. 2014. No. 7.
3. Low Power Wide Area Network (LPWAN) Protocols: Enablers for future wireless networks [Електронний ресурс] // ScienceDirect. – 2025. – Режим доступу:https://www.sciencedirect.com/science/article/pii/S2405959525000194.
4. Boulogeorgos A.-A. A. et al. A systematic and comprehensive review on low power wide area network: characteristics, architecture, applications and research challenges. *Discover Internet of Things*. 2025. Vol. 5. Art. 7.
5. Mekki K. et al. A comparative study of LPWAN technologies for large-scale IoT deployment. *ICT Express*. 2019. Vol. 5, Issue 1. P. 1–7.
6. Inthasuth T., Kaewjumras Y., Somwong S., Boonsong W. Comparative analysis of ZigBee, LoRa, and NB-IoT in a smart building: advantages, limitations, and integration possibilities. *International Journal of Reconfigurable and Embedded Systems (IJRES)*. 2025. Vol. 14, Issue 1. P. 165–175.
7. Koshlich Yu., Bukhanov D., Stan V., Kovtunov M. Analysis of Data Transfer Protocols for Wireless Sensor Networks Implementation. *19th International Multidisciplinary Scientific GeoConference SGEM 2019*. 2019. Vol. 19, No. 6.3. P. 253–260.

8. Cui Z. et al. Towards high-efficiency self-powered wireless sensor networks: A systematic review of co-design of energy and signal. *Nano Energy*. 2025. Vol. 144. Art. 111337.
9. Dumka A., Chaurasia S., Biswas A., Mandoria H. L. A complete guide to wireless sensor networks: from inception to current trends. CRC Press, 2023.
10. Jouhari M., Saeed N., Alouini M.-S., Amhoud E. M. A Survey on Scalable LoRaWAN for Massive IoT: Recent Advances, Potentials, and Challenges. *IEEE Communications Surveys & Tutorials*. 2023. Vol. 25, Issue 3. P. 1841–1876.
11. Fortuna C., Mohorcic M. A review of wireless sensor networks for environmental monitoring. *Sensors*. 2019. Vol. 19, Issue 9. Art. 2049.
12. Maleki A. et al. A Tutorial on Chirp Spread Spectrum for LoRaWAN: Basics and Key Advances. *arXiv preprint*. 2023. arXiv:2310.10503. URL: https://arxiv.org/abs/2310.10503.
13. Meshtastic [Електронний ресурс] // Wikipedia, The Free Encyclopedia. – 2024. – Режим доступу: https://en.wikipedia.org/wiki/Meshtastic (дата звернення: 10.06.2026).
14. Alahmadi H., Bouabdallah F., Al-Dubai A. et al. A Survey on LoRaWAN MAC Schemes: From Conventional Solutions To AI-driven Protocols. *IEEE Communications Surveys & Tutorials*. 2025. Vol. 28. P. 2650–2690.
15. Про затвердження Національної таблиці розподілу смуг радіочастот України : Постанова Кабінету Міністрів України від 15 груд. 2005 р. № 1208 [Електронний ресурс] // Законодавство України. – Режим доступу: https://zakon.rada.gov.ua/laws/show/1208-2005-%D0%BF.
16. Remote Battlefield Sensor System (REMBASS). (2000). FAS Military Analysis Network. https://man.fas.org/dod-101/sys/land/rembass.htm.

Додатки А. Код програмного забезпечення

(сенсорного вузла)

```
#include "Settings.h"
#include <SoftwareSerial.h>
#include "LoRa_E32.h"
#include <avr/sleep.h>  // Стандартна бібліотека
#include <avr/wdt.h>    // Стандартна бібліотека для Watchdog

SoftwareSerial mySerial(PIN_LORA_RX, PIN_LORA_TX);
LoRa_E32 loRa(&mySerial, PIN_AX, PIN_M0, PIN_M1);

// --- ЗМІННІ ---
volatile bool wokeBySensor = false;
uint16_t eventCounter = 0;
uint16_t wakeCounter = 0;
uint8_t  tryAttempt = 0;
bool     hasKnot = false;
char str[40];
// --- ФУНКЦІЇ ---
void enterSleep();
void updateLogicState();
void sendAndReceive();
long readVcc();
uint8_t calculateChecksum(void* pkt, size_t Psize);
void setupLoRa(uint16_t addr, uint8_t channel);
void ISR_Sensor();

// ========================================
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);
  pinMode(PIN_SENSOR_INT, INPUT);
  pinMode(3, INPUT);
  mySerial.begin(9600);
  loRa.begin();
  digitalWrite(PIN_M0, LOW);
  digitalWrite(PIN_M1, LOW);
  Serial.begin(115200);
  Serial.println("=== Start MINI ====");
  setupLoRa(MY_ADDR,  LOCAL_CHANNEL);
  Serial.print("My addres=");
  Serial.print(MY_ADDR);
  Serial.print(" channel=");
  Serial.println(LOCAL_CHANNEL);  //***
}
```

```
void loop() {
  enterSleep();       // Сон
  updateLogicState(); // Обробка подій при пробудженні
  if (hasKnot) {      //якщо є вузлик на роботу
    sendAndReceive(); //Відіслати пакет і дочекатися підтвердження
  }
}

// ========================================
// СЕРВІСНІ ФУНКЦІЇ
// ========================================
void enterSleep() { // --- Засинання (Через стандартний avr/sleep) ---
        // 1. Присипляємо LoRa
  digitalWrite(PIN_M0, HIGH);
  digitalWrite(PIN_M1, HIGH);
  delay(2);
        // 2. Налаштування переривання від сенсора
  int triggerMode = (SENSOR_TRIGGER == MODE_RISING) ? RISING : FALLING;
  attachInterrupt(digitalPinToInterrupt(PIN_SENSOR_INT), ISR_Sensor, triggerMode);
        // 3. Налаштування Watchdog на 8 секунд // Це замінює
LowPower.powerDown(SLEEP_8S...)
  ADCSRA &= ~(1 << ADEN); // Вимикаємо ADC для економії
        // Послідовність запуску Watchdog
  MCUSR &= ~(1 << WDRF);
  WDTCSR |= (1 << WDCE) | (1 << WDE);
        // Встановлюємо прескалер на 8 с (біти WDP3 та WDP0)
  WDTCSR = (1 << WDIE) | (1 << WDP3) | (1 << WDP0);
        // 4. Режим глибокого сну
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
        // --- ТУТ МИ СПИМО ---
  sleep_mode();
        // --- ТУТ ПРОКИНУЛИСЯ ---
  sleep_disable(); // Забороняємо сон
  ADCSRA |= (1 << ADEN); // Вмикаємо ADC назад
        // 5. Вимикаємо переривання сенсора
  detachInterrupt(digitalPinToInterrupt(PIN_SENSOR_INT));
}

ISR(WDT_vect) {// Просто прокидаємося, нічого не робимо  Порожній обробник
переривання для Watchdog (обов'язково!)
}

void updateLogicState() {   //обробляємо події
  wakeCounter++;    //збільшуємо лічильник пробуджень для того щоб вести облік часу
  digitalWrite(LED_BUILTIN, LOW);         //*** Гасимо LED (якщо прокинулись від
сенсора, ми його запалили в ISR)
  Serial.print("Кількість пробуджень ");  //***
  Serial.println(wakeCounter);            //***
```

```
  if ((wakeCounter%5)==0)hasKnot = true;  //***
  if (wokeBySensor) { //якщо було переривання поки спали, то був рух
    wokeBySensor = false;   //скидаємо признак руху
    Serial.println("був рух");      //***
    eventCounter++;         //збільшуємо лічильник рухів
    hasKnot = true;         //ставипо вузлик про те що треба повідомити нове значення
  }
  if (wakeCounter > MAX_SLEEPS_HEARTBEAT) {//якщо кількість пробуджень досягла
межі (пройшов достатній час)
    hasKnot = true;       //ставимо вузлик про потребу послати
    wakeCounter = 0;
  }
}

void sendAndReceive() {
  uint16_t currentVcc = (uint16_t)readVcc();  //міряємо власну напругу
    // Будимо LoRa
  digitalWrite(PIN_M0, LOW);
  digitalWrite(PIN_M1, LOW);
  delay(50);
  LoraPacket pkt;                 //сотворили обєкт для пакету
  pkt.key = SYS_KEY;                //ключ нашої системи і за одно покажчик типу пакету
  pkt.toAddr = REPEATER_ADDR;           //ставимо адресатаи (сервер 1)
  pkt.fromAddr = (uint8_t)MY_ADDR;    //пишемо власну адресу
  pkt.viaCluster = LOCAL_CHANNEL;     //пишемо власний канал, щоб сервер потім знав
хто я
  pkt.cnt = eventCounter;            //пишемо лічильник подій
  pkt.data = currentVcc;             //пишемо власну напругу живлення
  pkt.tryNum = tryAttempt++;          //пишемо кількість спроб відправити це повідомлення
  pkt.cs = calculateChecksum(&pkt,sizeof(LoraPacket));   //рахуємо контрольну суму пакету
  if (tryAttempt>100) tryAttempt=100;
    //ставимо адресу того, хто в нашій мережі є репітером і власний канал
  Serial.print("Кількість зафіксованих подій ");    //***
  Serial.println(pkt.cnt);                      //***
  loRa.sendFixedMessage(highByte(REPEATER_ADDR), lowByte(REPEATER_ADDR),
LOCAL_CHANNEL, &pkt, sizeof(LoraPacket));  //шлемо пакет
  unsigned long startWait = millis();
    // Чекаємо або таймаут, або доки прийде достатньо байт
  while ((millis() - startWait) < RX_TIMEOUT && loRa.available() < sizeof(AckPacket))
{delay(1);}
  if (millis() - startWait < RX_TIMEOUT) {
    Serial.println("наш пакет прийняли");  //***
    ResponseStructContainer rsc = loRa.receiveMessage(sizeof(AckPacket));
    struct AckPacket *inc = (AckPacket*) rsc.data;
    if (inc->key == SYS_KEY &&
        inc->toAddr == (uint8_t)MY_ADDR &&
        inc->cmd == 0 &&
        inc->cs == calculateChecksum(inc,sizeof(AckPacket)))
    {
      tryAttempt = 0;  // Скидаємо при успіху підтвердження
```

```
      if (inc->cnt == eventCounter) { hasKnot = false; }  //як співпадають лічильник то скидаємо "вузлик"
    }
    rsc.close();
  } else
    {
      Serial.println("timeOut ACK");  //***
    }

}

void ISR_Sensor() {
  wokeBySensor = true;
  digitalWrite(LED_BUILTIN, HIGH); // Світимо при події
}

long readVcc() {
  ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
  delay(2);
  ADCSRA |= _BV(ADSC);
  while (bit_is_set(ADCSRA, ADSC));
  uint8_t low = ADCL;
  uint8_t high = ADCH;
  long result = (high << 8) | low;
  return 1125300L / result;
}
uint8_t calculateChecksum(void* pkt, size_t Psize) {
    uint8_t sum = 0;
    uint8_t* ptr = (uint8_t*) pkt;
    for (size_t i = 0; i < Psize - 1; i++) sum += ptr[i];
    return sum;
}

void setupLoRa(uint16_t addr, uint8_t channel) {
  // 1. Спочатку читаємо поточну конфігурацію (щоб не збити UART)
  ResponseStructContainer c = loRa.getConfiguration();

  if (c.status.code != 1) {
    Serial.println(F(" ⚠ Err: Read Config Fail"));
    c.close();
    return;
  }

  Configuration cfg = *(Configuration*) c.data;

  // 2. Змінюємо ТІЛЬКИ радіо-параметри
  cfg.ADDH = highByte(addr);
  cfg.ADDL = lowByte(addr);
  cfg.CHAN = channel;
  cfg.SPED.airDataRate = (AIR_DATA_RATE) 2;   // Жорстко ставимо 2.4kbps (значення 2)
```

```
  // Опції мережі (мають співпадати у всіх)
  cfg.OPTION.fixedTransmission = FT_FIXED_TRANSMISSION;
  cfg.OPTION.fec = FEC_1_ON;
  cfg.OPTION.transmissionPower = POWER_10; //POWER_20;

  // 3. ВИВІД БАЙТІВ КОНФІГУРАЦІЇ (DEBUG HEX)
  // Ми "підглядаємо" сирі байти структур SPED та OPTION
  uint8_t spedByte   = *(uint8_t*)&cfg.SPED;
  uint8_t optionByte = *(uint8_t*)&cfg.OPTION;

  Serial.print(F("CFG [C2] : ")); // C2 - це команда запису в RAM
  if (cfg.ADDH < 0x10) Serial.print("0"); Serial.print(cfg.ADDH, HEX); Serial.print(" ");
  if (cfg.ADDL < 0x10) Serial.print("0"); Serial.print(cfg.ADDL, HEX); Serial.print(" ");
  if (spedByte < 0x10) Serial.print("0"); Serial.print(spedByte, HEX); Serial.print(" ");
  if (cfg.CHAN < 0x10) Serial.print("0"); Serial.print(cfg.CHAN, HEX); Serial.print(" ");
  if (optionByte < 0x10) Serial.print("0"); Serial.println(optionByte, HEX);

  // 4. Записуємо в RAM (налаштування зникнуть після вимкнення живлення)
  ResponseStatus rs = loRa.setConfiguration(cfg, WRITE_CFG_PWR_DWN_LOSE);

  if (rs.code == 1) {
     Serial.println(F("Result   : OK (RAM)"));
  } else {
     Serial.print(F("Result   : Err ")); Serial.println(rs.code);
  }

  c.close();
}
```

(Код вузла репітера/концентратора)

```
#include "Settings.h"
#include <SoftwareSerial.h>
#include "LoRa_E32.h"

SoftwareSerial mySerial(PIN_LORA_RX, PIN_LORA_TX);
LoRa_E32 loRa(&mySerial, PIN_AX, PIN_M0, PIN_M1);

// --- БАЗА ДАНИХ ---
struct NodeEntry {
   uint16_t cnt;       // Останній лічильник
   uint16_t voltage;   // Остання напруга
   uint8_t  tryNum;    // Спроби
};

NodeEntry nodes[32];          // Масив на 32 міні (індекси 0-31)
uint32_t pendingKnots = 0;    // Бітова маска вузликів (32 біти)
unsigned long lastRetryTime = 0;       // Час останньої спроби відправки
uint8_t globalCursor = 0,tryToServer=0;
```

```
// --- ПРОТОТИПИ ---
void processIncomingData();
void processMiniPacket(uint8_t* data);
void processAckPacket(uint8_t* data);
void handleMiniPacket(MiniPacket* pkt);
void sendAckToMini(uint8_t targetAddr, uint16_t count);
uint8_t calculateChecksum(void* pkt, size_t size);
long readVcc();
void sendToServer(int slot);
void setupLoRa(uint16_t addr, uint8_t channel);
// ========================================
void setup() {
   Serial.begin(115200);
   mySerial.begin(9600);
   loRa.begin();

   for(int i = 0; i < 32; i++) {    // Ініціалізуємо базу даних нулями
      nodes[i].cnt = 0;
      nodes[i].voltage = 0;
      nodes[i].tryNum = 0;
   }
   setupLoRa(MY_ADDR,LOCAL_CHANNEL);
   Serial.print(F("REPEATER START [Адреса:
"));Serial.print(MY_ADDR);Serial.println("]");Serial.println(F("Обслуговую MINI з адресами
3-33"));
   Serial.print(F("CHANEL ["));Serial.print(LOCAL_CHANNEL);Serial.println("]");
}
// ========================================
void loop() {
   processIncomingData();                              // Слухаємо ефір
   if (millis() - lastRetryTime > RETRY_INTERVAL) {            // Обробка вузликів по
таймеру
      lastRetryTime = millis();
      if (pendingKnots > 0) {
         while ((pendingKnots & (1UL << globalCursor)) == 0) {  // Швидко прокручуємо
курсор до найближчого встановленого біта
            globalCursor = (globalCursor + 1) & 0x1F;
         }
         sendToServer(globalCursor);
         globalCursor = (globalCursor + 1) & 0x1F;
      }
   }
}
// ========================================
// ФУНКЦІЯ ПРИЙОМУ ПАКЕТІВ
// ========================================
void processIncomingData() {
   if (mySerial.available() <= 0) return;
   uint8_t cmd = mySerial.read();                          // Читаємо ключ (перший байт)
   int bytesToRead = 0;
```

```
  switch (cmd) {                                  // Визначаємо, скільки байт чекати (SWITCH 1)
    case KEY_MINI:      bytesToRead = sizeof(MiniPacket) - 1;    break;
    case KEY_SERVER_ACK:bytesToRead = sizeof(AckPacket) - 1;     break;
    default: return;                              // Невідомий ключ - ігноруємо і почнемо заново
  }
  static uint8_t dataBuffer[20];
  int count = 0;
  unsigned long startTime = millis();
  while (count < bytesToRead && (millis() - startTime < 100)) {
    if (mySerial.available()) { dataBuffer[count++] = mySerial.read();   }
  }
  if (count == bytesToRead) { // 4. Якщо все прочитали - розкидаємо по функціях (SWITCH 2)
    switch (cmd) {
      case KEY_MINI:         processMiniPacket(dataBuffer);       break;
      case KEY_SERVER_ACK:    processAckPacket(dataBuffer);        break;
    }
  } else {  // Помилка таймауту
    Serial.print("Error: Incomplete packet for cmd ");
    Serial.println(cmd, HEX);
  }
}

// --- Обробка пакету від MINI ---
void processMiniPacket(uint8_t* data) {
  MiniPacket pkt;
  pkt.key = KEY_MINI;                             // Відновлюємо ключ
  memcpy(&pkt.toAddr, data, sizeof(MiniPacket) - 1);  // Копіюємо решту
  if (pkt.cs != calculateChecksum(&pkt, sizeof(MiniPacket))) { Serial.println(F("  ✗ CR Error"));   return;   }
  if (pkt.toAddr != MY_ADDR) {    Serial.print(F("  ✗ Wrong Dest: "));  Serial.println(pkt.toAddr);   return;   }
  if (pkt.viaCluster != LOCAL_CHANNEL) {Serial.print(F("  ✗ Wrong Channel: "));  Serial.println(pkt.viaCluster);  return;  }
  // Дозволяємо: Від 1 (Сервер) до (2 + 31) = 33.//допустимі адреси 1, 3-33
  if (pkt.fromAddr == 0 || pkt.fromAddr >= (MY_ADDR + MAX_NODES) || pkt.fromAddr == (MY_ADDR) ) { Serial.print(F("  ✗ Invalid FromAddr: ")); Serial.println(pkt.fromAddr); return; }
  Serial.println(F("_received valid packet"));
  handleMiniPacket(&pkt);
}

// --- Обробка ACK пакету ---
void processAckPacket(uint8_t* data) {
  AckPacket pkt;
  pkt.key = KEY_SERVER_ACK;
  memcpy(&pkt.toAddr, data, sizeof(AckPacket)-1);   // Копіюємо решту байт
```

```
  if (pkt.cs != calculateChecksum(&pkt, sizeof(AckPacket))) { Serial.println(F("  ✗ Невірна контрольна сума ACK сервера")); return; }
  if (pkt.toAddr<MY_ADDR || pkt.toAddr>MY_ADDR+31) {  Serial.println(F("  ✗ Невірна адреса ACK сервера")); return; }
  pendingKnots &=~(1UL << (pkt.toAddr-MY_ADDR));  //скидаємо "вузлик" який стояв для пакету серверу
  tryToServer=0;
  Serial.print(F("✓ ACK від сервера репітеру для міні ")); Serial.print(pkt.toAddr); Serial.print(F(", лічильник=")); Serial.println(pkt.cnt);
}

// --- Обробник пакету від MIHI ---
void handleMiniPacket(MiniPacket* pkt) {
  uint8_t idx;
  if  (pkt->fromAddr==SERVER_ADDR) idx = 0; // сервер пакет мені шле  тільки для відповіді. буду тримати вузлик в [0] і ACK відповідати не буду
  else idx = pkt->fromAddr - MY_ADDR;
  pendingKnots |= (1UL << idx);
  if (idx > 0) { // --- це MIHI, а міг бути запит від сервера ---
    nodes[idx].cnt = pkt->cnt;
    nodes[idx].voltage = pkt->data;
    nodes[idx].tryNum = pkt->tryNum;
    Serial.print(F("✓ MINI [")); Serial.print(pkt->fromAddr);
    Serial.print(F("] -> Slot ")); Serial.println(idx);
    sendAckToMini(pkt->fromAddr, pkt->cnt);
  } else {// --- СЕРВЕР (PING) у відповіді на цей запит важливими є тільки напруга репітера
    Serial.println(F("ℹ Server Ping Request."));
  }
  sendToServer(idx);
}

// --- Відправка ACK до вузла (MIHI) ---
void sendAckToMini(uint8_t targetAddr, uint16_t count) {
  AckPacket ack;
  ack.key = KEY_MINI;                                  // Використовуємо KEY_MINI для ACK до міні
  ack.toAddr = targetAddr;                             // Кому (міні 3-33)
  ack.fromAddr = (uint8_t)MY_ADDR;                     // Від кого (репітер)
  ack.cmd = 0;                                         // Команда 0 = ACK
  ack.cnt = count;                                     // Той самий лічильник
  ack.cs = calculateChecksum(&ack, sizeof(AckPacket));
  loRa.sendFixedMessage(0, targetAddr, LOCAL_CHANNEL, &ack, sizeof(AckPacket)); // Шлемо на каналі (LOCAL_CHANNEL) на адресу міні
  Serial.println(F("  → ACK до MIHI відправлено"));
}

// --- Контрольна сума ---
uint8_t calculateChecksum(void* pkt, size_t size) {
  uint8_t sum = 0;
```

```
    uint8_t* ptr = (uint8_t*)pkt;
    for (size_t i = 0; i < size - 1; i++) {
        sum += ptr[i];
    }
    return sum;
}

// --- Вимірювання напруги живлення (mV) ---
long readVcc() {
  // Налаштування ADC для зчитування внутрішньої опорної напруги 1.1В
  ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
  delay(2); // Чекаємо стабілізації Vref
  ADCSRA |= _BV(ADSC); // Запуск перетворення
  while (bit_is_set(ADCSRA, ADSC)); // Чекаємо завершення
  uint8_t low = ADCL;
  uint8_t high = ADCH;
  long result = (high << 8) | low;
  // Розрахунок Vcc (калібрувальне значення 1125300L)
  return (result > 0) ? (1125300L / result) : 0;
}

// --- Відправка даних на СЕРВЕР ---
void sendToServer(int slot) {
    if (slot < 0 || slot >= 32) return;
    uint16_t currentVcc = (uint16_t)readVcc();
    RepeaterPacket pkt;
    pkt.key = KEY_REP;
    pkt.toAddr = (uint8_t)SERVER_ADDR;
    pkt.fromAddr = (uint8_t)(slot + 2); // Відновлюємо адресу Міні
    pkt.viaCluster = LOCAL_CHANNEL;
    pkt.cnt = nodes[slot].cnt;
    pkt.miniData = nodes[slot].voltage;
    pkt.miniTry = nodes[slot].tryNum;
    pkt.repData = currentVcc;
    pkt.repTry=tryToServer++;
    pkt.cs = calculateChecksum(&pkt, sizeof(RepeaterPacket));
    if (tryToServer>250)tryToServer=250;
    loRa.sendFixedMessage(highByte(SERVER_ADDR), lowByte(SERVER_ADDR),
SERVER_CHANNEL, &pkt, sizeof(RepeaterPacket));
    Serial.print(F(">>> try "));Serial.print(tryToServer);Serial.print(F(" sent Slot "));
Serial.print(slot); Serial.print(F(" (RepVcc: ")); Serial.print(currentVcc); Serial.println(F("mV)
to Server"));
}

void setupLoRa(uint16_t addr, uint8_t channel) {
  // 1. Спочатку читаємо поточну конфігурацію (щоб не збити UART)
  ResponseStructContainer c = loRa.getConfiguration();
  if (c.status.code != 1) {
    Serial.println(F("⚠ Err: Read Config Fail"));
    c.close();
```

```
  return;
 }
 Configuration cfg = *(Configuration*) c.data;
 // 2. Змінюємо ТІЛЬКИ радіо-параметри
 cfg.ADDH = highByte(addr);
 cfg.ADDL = lowByte(addr);
 cfg.CHAN = channel;
 // Жорстко ставимо 2.4kbps (значення 2)
 cfg.SPED.airDataRate = (AIR_DATA_RATE) 2;
 // Опції мережі (мають співпадати у всіх)
 cfg.OPTION.fixedTransmission = FT_FIXED_TRANSMISSION;
 //cfg.OPTION.fixedTransmission = FT_TRANSPARENT_TRANSMISSION;
 cfg.OPTION.fec = FEC_1_ON;
 cfg.OPTION.transmissionPower = POWER_10; //POWER_20;
 // 3. ВИВІД БАЙТІВ КОНФІГУРАЦІЇ (DEBUG HEX)
 // Ми "підглядаємо" сирі байти структур SPED та OPTION
 uint8_t spedByte   = *(uint8_t*)&cfg.SPED;
 uint8_t optionByte = *(uint8_t*)&cfg.OPTION;
 Serial.print(F("CFG [C2] : ")); // C2 - це команда запису в RAM
 if (cfg.ADDH < 0x10) Serial.print("0"); Serial.print(cfg.ADDH, HEX); Serial.print(" ");
 if (cfg.ADDL < 0x10) Serial.print("0"); Serial.print(cfg.ADDL, HEX); Serial.print(" ");
 if (spedByte < 0x10) Serial.print("0"); Serial.print(spedByte, HEX); Serial.print(" ");
 if (cfg.CHAN < 0x10) Serial.print("0"); Serial.print(cfg.CHAN, HEX); Serial.print(" ");
 if (optionByte < 0x10) Serial.print("0"); Serial.println(optionByte, HEX);
 // 4. Записуємо в RAM (налаштування зникнуть після вимкнення живлення)
 ResponseStatus rs = loRa.setConfiguration(cfg, WRITE_CFG_PWR_DWN_LOSE);
 if (rs.code == 1) {
    Serial.println(F("Result   : OK (RAM)"));
 } else {
    Serial.print(F("Result   : Err ")); Serial.println(rs.code);
 }
 c.close();
}
```

(Код вузла серверної системи)

```
/*
  LoRa ESP32 Server v5.1 (AP/Station Switch + Interactive WEB + Tries Column)
  - Повернуто стовпчик "Tries"
  - Перемикач режимів: Точка Доступу <-> Роутер
  - Watchdog 30 хв
  - Фільтр шумів
  - Інтерактивна таблиця
*/

#include <WiFi.h>
#include <WebServer.h>
#include <HardwareSerial.h>
#include "LoRa_E32.h"
```

```
// ========================================
// 1. НАЛАШТУВАННЯ (Settings)
// ========================================

// --- ВИБІР РЕЖИМУ WI-FI ---
// Розкоментуй рядок нижче, щоб увімкнути режим Точки Доступу (ESP створює мережу)
// Закоментуй (постав //), щоб підключатися до Роутера
//#define ENABLE_AP_MODE  true

// --- ДАНІ WI-FI ---
const char* wifi_ssid = "default314_2";
const char* wifi_pass = "rev0lut10n";

// --- ПІНИ ---
constexpr uint8_t PIN_LORA_RX = 16;   // RX2
constexpr uint8_t PIN_LORA_TX = 17;   // TX2
constexpr uint8_t PIN_M0      = 19;
constexpr uint8_t PIN_M1      = 18;
constexpr uint8_t PIN_AX      = 23;

// --- МЕРЕЖА LORA ---
constexpr uint8_t  SERVER_CHANNEL = 2;
constexpr uint16_t SERVER_ADDR    = 0x0001;
constexpr uint16_t REPEATER_ADDR  = 0x0002;
constexpr uint16_t MY_ADDR        = SERVER_ADDR;

constexpr uint8_t KEY_MINI        = 0xBB;
constexpr uint8_t KEY_REP         = 0xEE;
constexpr uint8_t KEY_SERVER_ACK  = 0xCC;

constexpr uint8_t  MAX_CLUSTERS   = 10;
constexpr uint8_t  MAX_NODES      = 32;
constexpr uint16_t ADDR_OFFSET    = 2;

#define LORA_AIR_RATE   AIR_DATA_RATE_010_24
#define LORA_POWER      POWER_10

const uint32_t WATCHDOG_TIMEOUT = 1800000; // 30 хв

// ========================================
// 2. СТРУКТУРИ ДАНИХ
// ========================================
#pragma pack(push, 1)
struct MiniPacket {
   uint8_t key; uint8_t toAddr; uint8_t fromAddr; uint8_t viaCluster;
   uint16_t cnt; uint16_t data; uint8_t tryNum; uint8_t cs;
};
struct RepeaterPacket {
   uint8_t key; uint8_t toAddr; uint8_t fromAddr; uint8_t viaCluster;
   uint16_t cnt; uint16_t miniData; uint8_t miniTry;
```

```
    uint16_t repData; uint8_t repTry; uint8_t cs;
};
struct AckPacket {
    uint8_t key; uint8_t toAddr; uint8_t fromAddr; uint8_t cmd;
    uint16_t cnt; uint8_t cs;
};
#pragma pack(pop)

struct ServerNodeEntry {
    uint16_t lastCnt;
    uint16_t refCnt;
    uint16_t voltage;
    uint8_t  maxTries;
    uint32_t lastSeen;
};

ServerNodeEntry nodes[MAX_CLUSTERS][MAX_NODES];
uint32_t lastValidPacketMillis = 0;

HardwareSerial LoRaSerial(2);
LoRa_E32 loRa(&LoRaSerial, PIN_AX, PIN_M0, PIN_M1);
WebServer server(80);

// --- ПРОТОТИПИ ---
void setupLoRaConfig(uint16_t addr, uint8_t channel);
void initLoRaHardware();
void processIncomingData();
void processRepPacket(uint8_t* data);
void sendAckToRepeater(uint8_t targetCluster, uint16_t cnt);
void sendPingToRepeater(uint8_t clusterIdx);
uint8_t calculateChecksum(void* pkt, size_t size);
void setupWebServer();
void handleRoot();

// ==========================================
// 3. SETUP & LOOP
// ==========================================

void setup() {
    Serial.begin(115200);
    delay(1000);
    Serial.println(F("\n=== SERVER V5.1 STARTING ==="));

    lastValidPacketMillis = millis();

    initLoRaHardware();
    setupWebServer();

    Serial.println(F("=== SYSTEM READY ==="));
}
```

```
void loop() {
   processIncomingData();
   server.handleClient();

   // WATCHDOG (30 хв)
   if (millis() - lastValidPacketMillis > WATCHDOG_TIMEOUT) {
      Serial.println(F("⚠ WATCHDOG: 30 min silence! Resetting LoRa..."));
      setupLoRaConfig(SERVER_ADDR, SERVER_CHANNEL);
      lastValidPacketMillis = millis();
   }
}

// ==========================================
// 4. LORA LOGIC
// ==========================================

void initLoRaHardware() {
   LoRaSerial.begin(9600, SERIAL_8N1, PIN_LORA_RX, PIN_LORA_TX);
   if (!loRa.begin()) {
      Serial.println(F("LoRa.begin() FAILED"));
      while(1);
   }
   setupLoRaConfig(SERVER_ADDR, SERVER_CHANNEL);
}

void setupLoRaConfig(uint16_t addr, uint8_t channel) {
   Configuration cfg;
   Serial.println(F("LoRa Config/Reset..."));

   ResponseStructContainer c = loRa.getConfiguration();
   if (c.status.code == 1) c.close();
   else {
       cfg.SPED.uartParity = MODE_00_8N1;
       cfg.SPED.uartBaudRate = UART_BPS_9600;
       cfg.SPED.airDataRate = LORA_AIR_RATE;
   }

   cfg.ADDH = highByte(addr);
   cfg.ADDL = lowByte(addr);
   cfg.CHAN = channel;
   cfg.OPTION.fixedTransmission = FT_FIXED_TRANSMISSION;
   cfg.OPTION.ioDriveMode = IO_D_MODE_PUSH_PULLS_PULL_UPS;
   cfg.OPTION.wirelessWakeupTime = WAKE_UP_250;
   cfg.OPTION.fec = FEC_1_ON;
   cfg.OPTION.transmissionPower = LORA_POWER;
   cfg.SPED.airDataRate = LORA_AIR_RATE;
   cfg.SPED.uartBaudRate = UART_BPS_9600;

   ResponseStatus rs = loRa.setConfiguration(cfg, WRITE_CFG_PWR_DWN_LOSE);
```

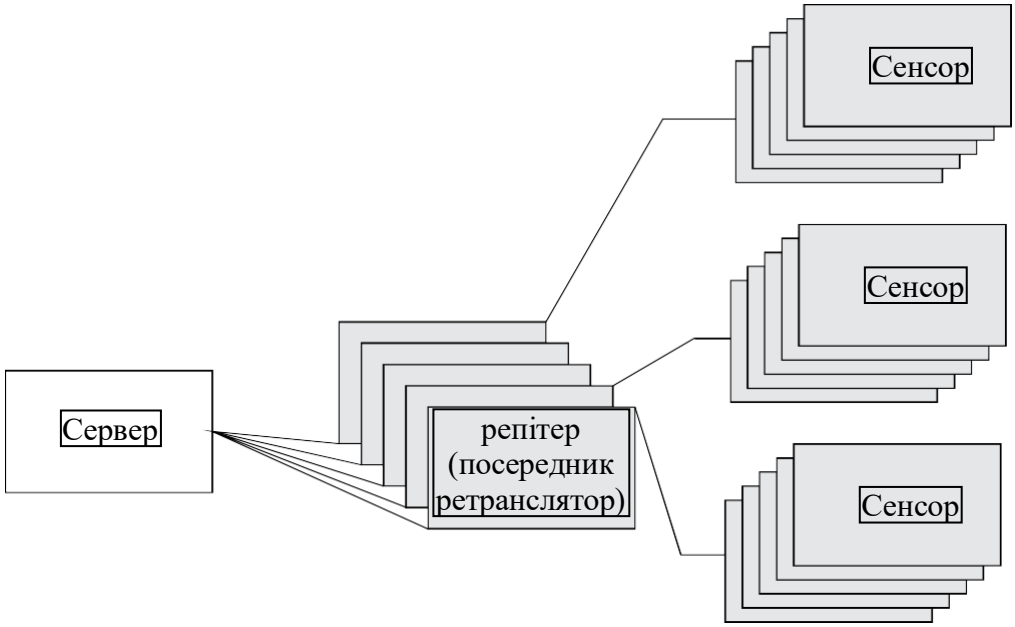
```
    if (rs.code == 1) Serial.println(F("   Write: OK"));
    else Serial.println(F("   Write: FAIL"));
}

void processIncomingData() {
    if (!LoRaSerial.available()) return;

    uint32_t junkCount = 0;
    while (LoRaSerial.available() && LoRaSerial.peek() != KEY_REP) {
        LoRaSerial.read();
        junkCount++;
        if (junkCount > 100) return;
    }

    if (LoRaSerial.available() >= sizeof(RepeaterPacket)) {
        static uint8_t rawBuffer[sizeof(RepeaterPacket)];
        LoRaSerial.readBytes(rawBuffer, sizeof(RepeaterPacket));
        processRepPacket(rawBuffer);
    }
}

void processRepPacket(uint8_t* data) {
    RepeaterPacket* pkt = (RepeaterPacket*)data;

    if (calculateChecksum(pkt, sizeof(RepeaterPacket)) != pkt->cs) return;
    if (pkt->toAddr != SERVER_ADDR) return;
    if (pkt->viaCluster < 10) return;

    uint8_t cIdx = (pkt->viaCluster - 10) / 2;
    if (cIdx >= MAX_CLUSTERS) return;

    lastValidPacketMillis = millis(); // Скидаємо Watchdog

    uint32_t now = millis() / 1000;

    if (pkt->fromAddr > REPEATER_ADDR) {
        uint8_t nodeIdx = pkt->fromAddr - ADDR_OFFSET;
        if (nodeIdx < MAX_NODES) {
            nodes[cIdx][nodeIdx].lastCnt = pkt->cnt;
            nodes[cIdx][nodeIdx].voltage = pkt->miniData;
            nodes[cIdx][nodeIdx].lastSeen = now;
            if (pkt->miniTry > nodes[cIdx][nodeIdx].maxTries) {
                nodes[cIdx][nodeIdx].maxTries = pkt->miniTry;
            }
            Serial.printf("Mini: Cl:%d Idx:%d Val:%d\n", cIdx, nodeIdx, pkt->cnt);
        }
    }

    nodes[cIdx][0].voltage = pkt->repData;
    nodes[cIdx][0].lastSeen = now;
```

```
    if (pkt->repTry > nodes[cIdx][0].maxTries) {
        nodes[cIdx][0].maxTries = pkt->repTry;
    }

    sendAckToRepeater(pkt->viaCluster, pkt->cnt);
}

void sendPingToRepeater(uint8_t clusterIdx) {
    uint8_t targetChannel = 10 + (clusterIdx * 2);
    MiniPacket req;
    req.key = KEY_MINI;
    req.toAddr = REPEATER_ADDR;
    req.fromAddr = SERVER_ADDR;
    req.viaCluster = targetChannel;
    req.cnt = 0; req.data = 0; req.tryNum = 0;
    req.cs = calculateChecksum(&req, sizeof(MiniPacket));

    loRa.sendFixedMessage(0, REPEATER_ADDR, targetChannel, &req, sizeof(MiniPacket));
    Serial.printf("-> Ping Cluster %d\n", clusterIdx);
}

void sendAckToRepeater(uint8_t targetCluster, uint16_t cnt) {
    AckPacket ack;
    ack.key = KEY_SERVER_ACK;
    ack.toAddr = REPEATER_ADDR;
    ack.fromAddr = SERVER_ADDR;
    ack.cmd = 0;
    ack.cnt = cnt;
    ack.cs = calculateChecksum(&ack, sizeof(AckPacket));
    loRa.sendFixedMessage(0, REPEATER_ADDR, targetCluster, &ack, sizeof(AckPacket));
}

uint8_t calculateChecksum(void* pkt, size_t size) {
    uint8_t sum = 0;
    uint8_t* ptr = (uint8_t*) pkt;
    for (size_t i = 0; i < size - 1; i++) sum += ptr[i];
    return sum;
}

// ==========================================
// 5. WEB SERVER & WIFI LOGIC
// ==========================================

void setupWebServer() {

    #ifdef ENABLE_AP_MODE
        Serial.println(F("Starting Access Point Mode..."));
        WiFi.mode(WIFI_AP);
        if (WiFi.softAP(wifi_ssid, wifi_pass, 1, 0, 4)) {
            Serial.print(F("AP Created! SSID: ")); Serial.println(wifi_ssid);
```

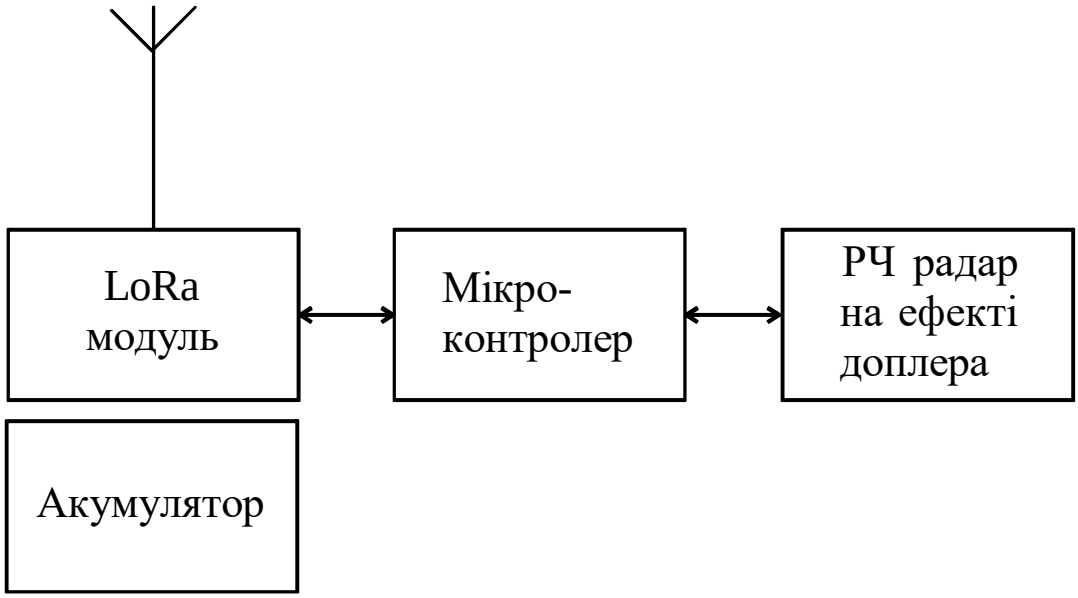
```
        Serial.print(F("Connect to IP: ")); Serial.println(WiFi.softAPIP());
    } else {
        Serial.println(F("AP Creation Failed!"));
    }

#else
    Serial.print(F("Connecting to Router... "));
    WiFi.mode(WIFI_STA);
    WiFi.begin(wifi_ssid, wifi_pass);

    int t = 0;
    while (WiFi.status() != WL_CONNECTED && t < 20) {
        delay(500); Serial.print("."); t++;
    }
    Serial.println();
    if (WiFi.status() == WL_CONNECTED) {
        Serial.print(F("Connected! IP: ")); Serial.println(WiFi.localIP());
    } else {
        Serial.println(F("Connection Failed! (Offline Mode)"));
    }
#endif

// --- ОБРОБНИКИ ---
server.on("/", handleRoot);

server.on("/action/toggle_cnt", []() {
    if (server.hasArg("c") && server.hasArg("n")) {
        int c = server.arg("c").toInt();
        int n = server.arg("n").toInt();
        if (c < MAX_CLUSTERS && n < MAX_NODES) {
            if (nodes[c][n].refCnt == 0) nodes[c][n].refCnt = nodes[c][n].lastCnt;
            else nodes[c][n].refCnt = 0;
        }
    }
    server.sendHeader("Location", "/"); server.send(303);
});

server.on("/action/get_volt", []() {
    if (server.hasArg("c")) {
        sendPingToRepeater(server.arg("c").toInt());
    }
    server.sendHeader("Location", "/"); server.send(303);
});

server.on("/action/restart_lora", []() {
    setupLoRaConfig(SERVER_ADDR, SERVER_CHANNEL);
    lastValidPacketMillis = millis();
    server.sendHeader("Location", "/"); server.send(303);
});
```

```
    server.begin();
    Serial.println(F("HTTP Server Started"));
}

void handleRoot() {
  String html = "<!DOCTYPE html><html><head>";
  html += "<meta charset='utf-8'>";
  html += "<meta http-equiv='refresh' content='5'>";
  html += "<title>LoRa Server v5.1</title>";
  html += "<meta name='viewport' content='width=device-width, initial-scale=1'>";
  html += "<style>";
  html += "body { font-family: Arial, sans-serif; text-align: center; background: #eee; margin: 0;
padding: 10px; }";
  html += "table { margin: 10px auto; border-collapse: collapse; width: 100%; background:
white; max-width: 600px; }";
  html += "th, td { border: 1px solid #ddd; padding: 10px; font-size: 16px; }";
  html += "th { background-color: #007bff; color: white; }";
  html += ".diff-mode { color: blue; font-weight: bold; background-color: #e6f2ff; }";
  html += ".offline { color: red; }";
  html += "a { text-decoration: none; color: inherit; cursor: pointer; display: block; }";
  html += ".btn { display: inline-block; padding: 12px 24px; background: #ff4444; color: white;
border-radius: 5px; margin-top: 20px; font-size: 16px;}";
  html += "</style></head><body>";

  html += "<h2>📡 LoRa Server ";
  #ifdef ENABLE_AP_MODE
    html += "(AP Mode)";
  #else
    html += "(WIFI Mode)";
  #endif
  html += "</h2>";

  uint32_t lastPktSec = (millis() - lastValidPacketMillis) / 1000;
  html += "<p>Last Valid Packet: " + String(lastPktSec) + "s ago</p>";

  // ДОДАНО КОЛОНКУ 'Try'
  html += "<table><thead><tr>";
  html +=
"<th>ID</th><th>Type</th><th>Volt</th><th>Cnt</th><th>Try</th><th>Seen</th>";
  html += "</tr></thead><tbody>";

  uint32_t now = millis() / 1000;
  bool foundAny = false;

  for (int c = 0; c < MAX_CLUSTERS; c++) {
    for (int n = 0; n < MAX_NODES; n++) {
      if (nodes[c][n].lastSeen > 0) {
        foundAny = true;
        uint32_t ago = now - nodes[c][n].lastSeen;
```

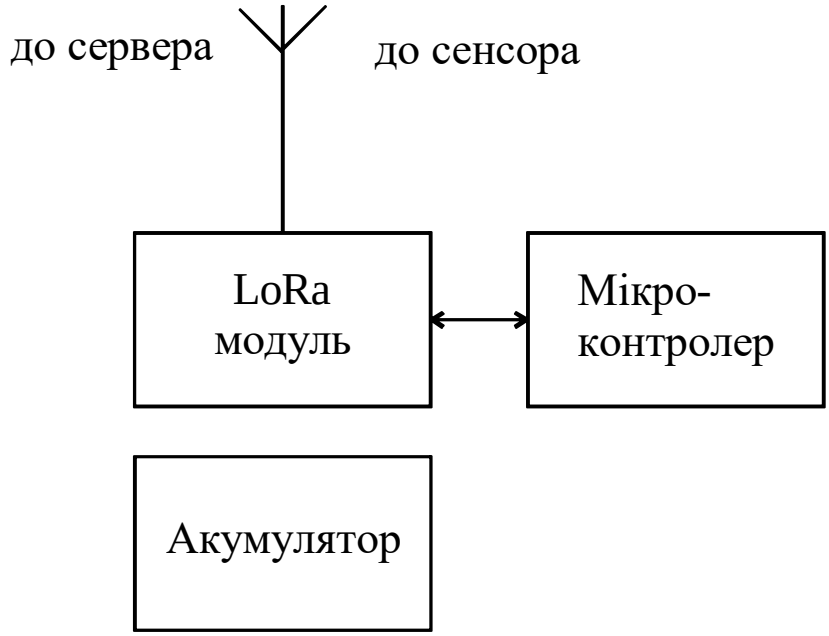
```
      html += "<tr>";
      // ID
      html += "<td>" + String(c) + "_" + String(n) + "</td>";

      // Type
      if (n == 0) html += "<td><b>REP</b></td>";
      else html += "<td>Mini " + String(n + ADDR_OFFSET) + "</td>";

      // Voltage
      html += "<td>";
      if (n == 0) html += "<a href='/action/get_volt?c=" + String(c) + "'><b>" +
String(nodes[c][n].voltage) + "</b>↻</a>";
      else html += "<b>" + String(nodes[c][n].voltage) + "</b>";
      html += "</td>";

      // Counter
      uint16_t displayVal = nodes[c][n].lastCnt - nodes[c][n].refCnt;
      String cellClass = (nodes[c][n].refCnt > 0) ? "class='diff-mode'" : "";
      html += "<td " + cellClass + "><a href='/action/toggle_cnt?c=" + String(c) + "&n=" +
String(n) + "'>" + String(displayVal) + "</a></td>";

      // TRIES (ПОВЕРНУТО)
      html += "<td>" + String(nodes[c][n].maxTries) + "</td>";

      // Seen
      if (ago > 300) html += "<td class='offline'>" + String(ago) + "s</td>";
      else html += "<td>" + String(ago) + "s</td>";

      html += "</tr>";
    }
   }
  }
  html += "</tbody></table>";
  if (!foundAny) html += "<h3>Waiting for data...</h3>";

  html += "<a href='/action/restart_lora' class='btn'>Force LoRa Reset</a>";
  html += "</body></html>";

  server.send(200, "text/html", html);
}
```



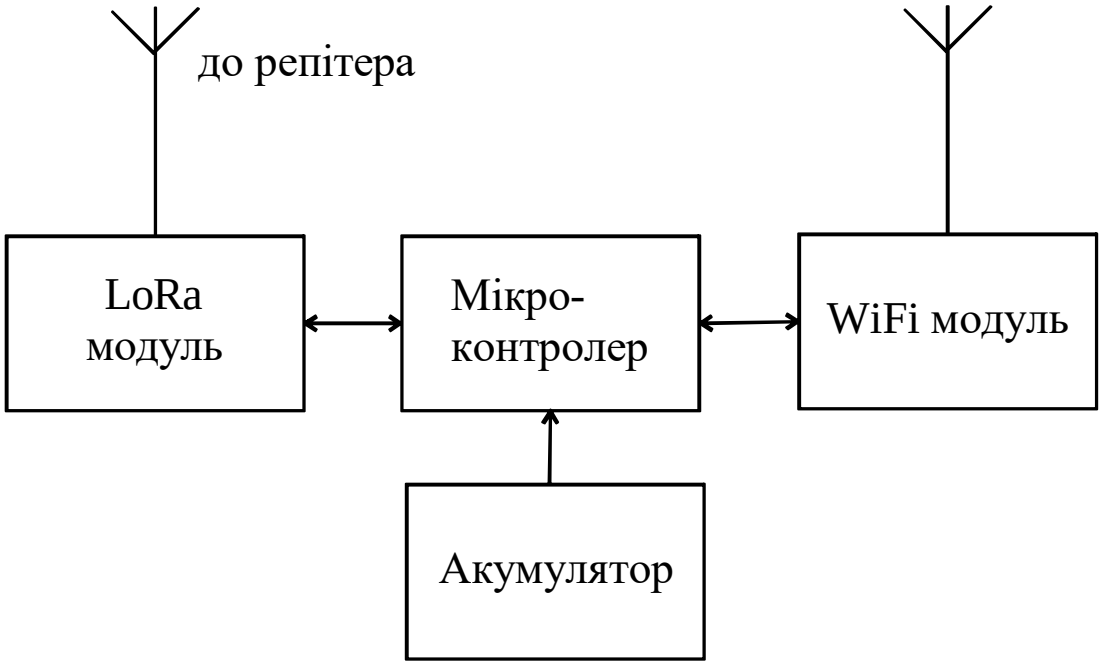
					КРБ.СІ.06.00.00.001	
						Літера / Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Загальна структура системи передачі повідомлень про дискретні події	
Розроб.		Андрейчук				
Перев.		Стрілецький				Аркуш / Аркушів
Т. контр.						
Н. контр.		Возний				
Затв.		Заміховський				



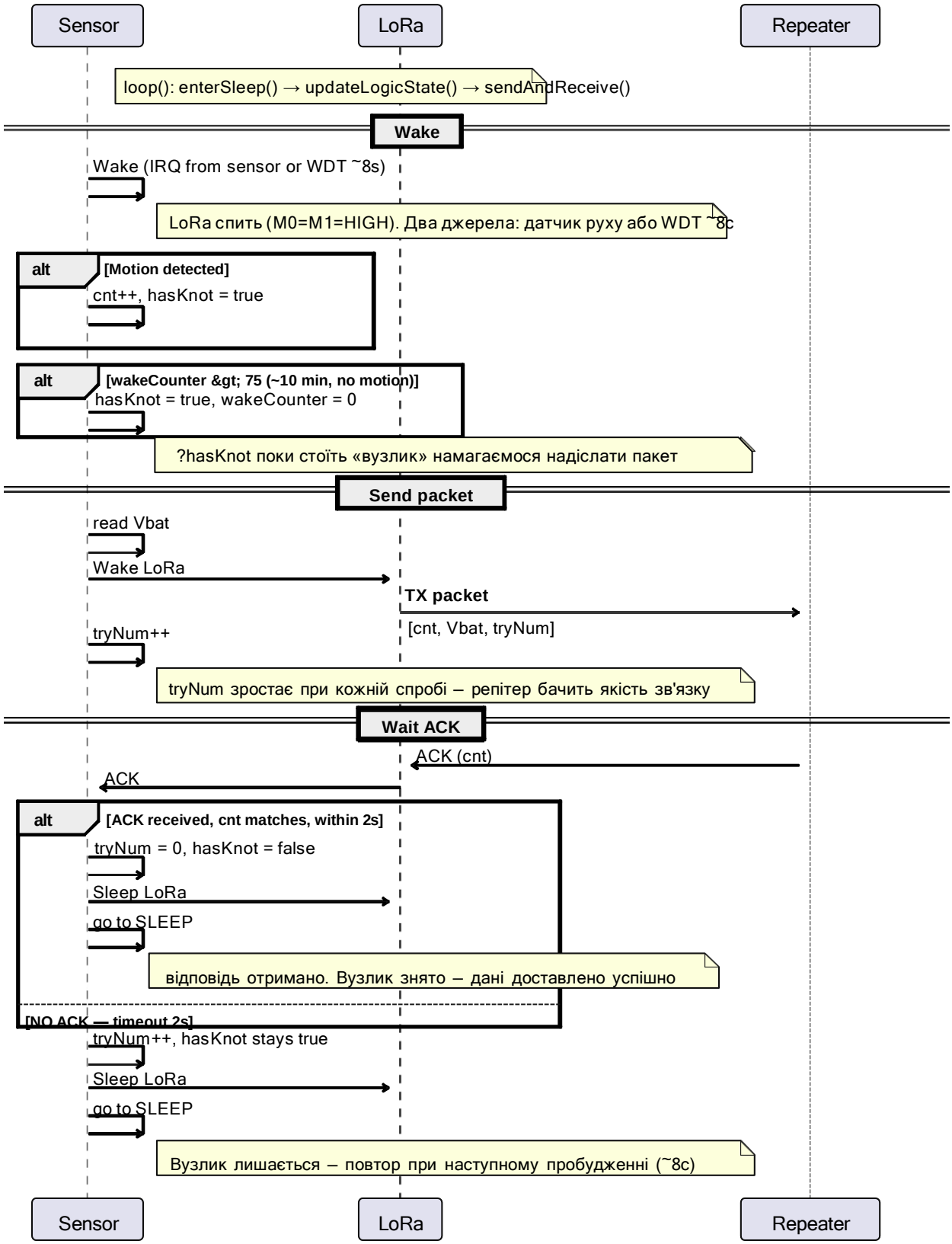
					КРБ.СІ.06.00.00.002	
						Літера / Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Структурна схема сенсорного вузла	
Розроб.		Андрейчук				
Перев.		Стрілецький				Аркуш / Аркушів
Т. контр.						
Н. контр.		Возний				
Затв.		Заміховський				



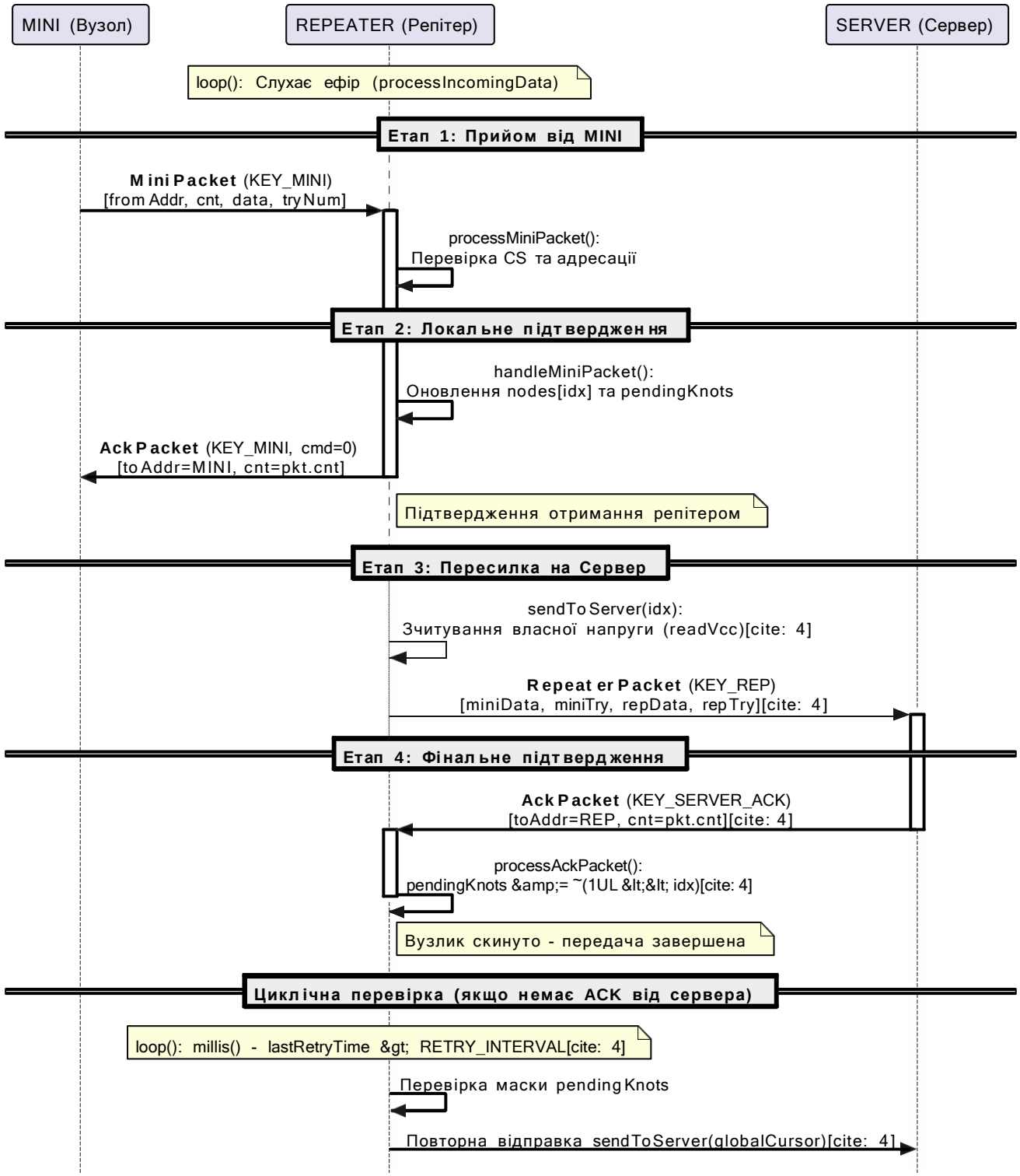
					КРБ.СІ.06.00.00.004	
						Літера / Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Структурна схема вузла репітера/концентратора	
Розроб.		Андрейчук				
Перев.		Стрілецький				Аркуш / Аркушів
Т. контр.						
Н. контр.		Возний				
Затв.		Заміховський				



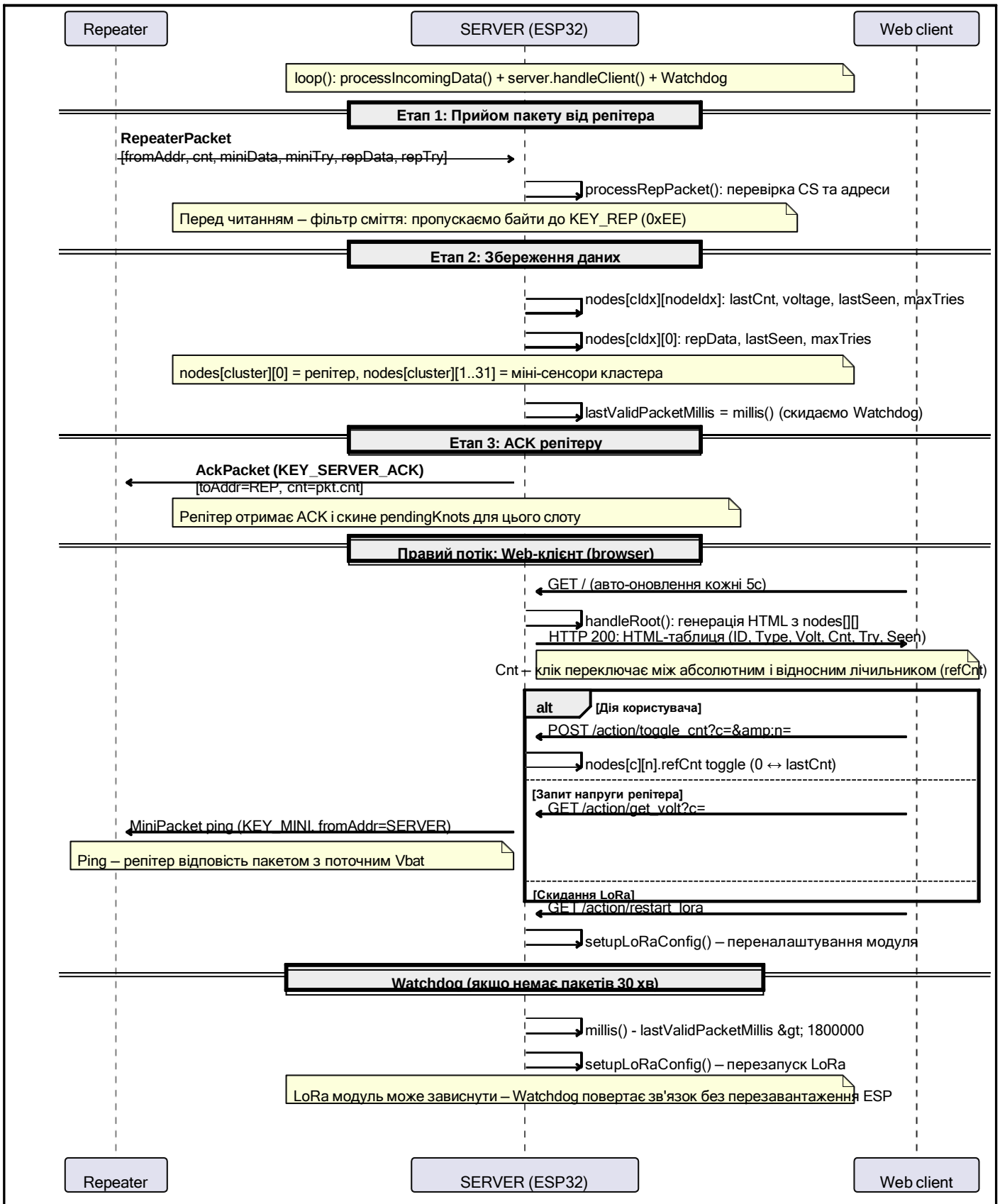
					КРБ.СІ.06.00.00.004	
						Літера / Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Структурна схема вузла сервера системи	
Розроб.		Андрейчук				
Перев.		Стрілецький				Аркуш / Аркушів
Т. контр.						
Н. контр.		Возний				
Затв.		Заміховський				



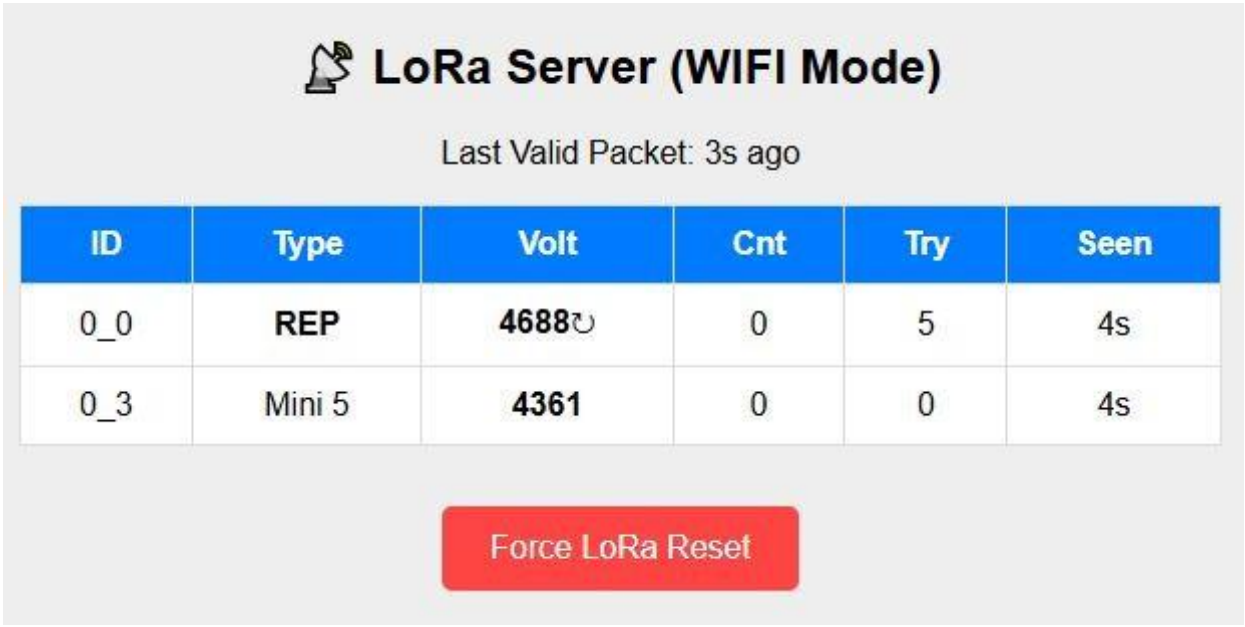
					КРБ.СІ.01.00.00.005	
					Літера	Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Діаграма роботи програми сенсорного вузла	
Розроб.		Андрейчук				
Перев.		Стрілецький				Аркуш / Аркушів
Т. контр.						
Н. контр.		Возний				
Затв.		Заміховський				



					КРБ.СІ.01.00.00.006	
					Літера	Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Діаграма роботи програми репітера	
Розроб.		Андрейчук				
Перев.		Стрілецький				Аркуш / Аркушів
Т. контр.						
Н. контр.		Возний				
Затв.		Заміховський				



					КРБ.СІ.01.00.00.007
					Літера / Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Діаграма роботи програми сервера системи
Розроб.		Андрейчук			
Перев.		Стрілецький			Аркуш / Аркушів
Т. контр.					
Н. контр.		Возний			
Затв.		Заміховський			



					КРБ.СІ.01.00.00.008		
						Літера	Маса / Мірило
Зм.	Арк.	№докум.	Підпис	Дата	Вид http сторінки контролю стану системи		
Розроб.		Андрейчук				Аркуш	Аркушів
Перев.		Стрілецький					
Т. контр.							
Н. контр.		Возний					
Затв.		Заміховський					

Бібліографічна довідка

Тема роботи: Розроблення розподіленої по площі системи передачі повідомлень про дискретні події

Обсяг бакалаврської роботи 88 сторінки

Перелік графічного матеріалу:

1. Загальна структура системи передачі повідомлень про дискретні події
2. Структурна схема сенсорного вузла
3. Структурна схема вузла репітера/концентратора
4. Структурна схема вузла сервера системи
5. Діаграма роботи програми сенсорного вузла
6. Діаграма роботи програми рептера
7. Діаграма роботи програми сервера системи
8. Вид http сторінки контролю стану системи

Дата закінчення БР_____

Студент _____ Олександр Андрейчук