Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Планувальник задач для систем реального часу
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

КвРКІ 022054.22.02.49 ПЗ

Виконав здобувач IV курсу, група КІ2-22-2 — Підпис — Антон МАЗУР (Ініціали, прізвище)

Керівник доктор техн. наук, проф. (Науковий ступінь, учене звання) — Підпис — Антоніна КАШТАЛЬЯН (Ініціали, прізвище)

Нормоконтролер канд. фіз.-мат. наук, доц. (Науковий ступінь, учене звання) — Підпис — Тетяна КИСІЛЬ (Ініціали, прізвище)

До захисту допускаю:
завідувач кафедри КІІС — Підпис — Ольга ПАВЛОВА (Ініціали, прізвище)
«01» червня 2026 р.
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Рівень вищої освіти перший (бакалаврський)

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
Ольга ПАВЛОВА

"10" 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Мазуру Антону Івановичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Планувальник задач для систем реального часу

Керівник проекту (роботи) Каштальян А.С., доктор техн. наук, проф.

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Аналіз відомих рішень та вибір стратегії і засобів для реалізації завдання

Проектування планувальника задач для систем реального часу

Алгоритмічне та програмне забезпечення розподіленої універсальної системи

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Алгоритм планувальника завдань

Архітектура програмного забезпечення системи

Результати роботи планувальника

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання при[йняв]

7. Дата видачі завдання « 10 » 01 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Прим[ітка]
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	викона[но]
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2026	викона[но]
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2026	викона[но]
4	Робота над розділом 2 – вибір компонентів для проектування планувальника задач для систем реального часу	01.04.2026	викона[но]
5	Робота над розділом 3 – проєктування алгоритмічного та програмного забезпечення розподіленої універсальної системи	29.04.2026	викона[но]
6	Оформлення пояснювальної записки згідно вимог	25.05.2026	викона[но]
7	Попередній захист ВКР	26.05.2026	викона[но]
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач ____________ Антон МАЗУР
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи ____________ Антоніна КАШТАЛЬЯН
Підпис Ім'я, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 022054.22.02.47 ПЗ	Пояснювальна записка	57		
			Графічні матеріали			
2		КвРКІ 022054.22.02.49 Е8	Алгоритм планувальника задач	1		
3		КвРКІ 022054.22.02.49 Е8	Архітектура програмного забезпечення системи	1		
4		КвРКІ 022054.22.02.49 Е8	Результати роботи планувальника	1		

КвРКІ 022054.22.02.49 ВП

Зм	Арк	№ докум	Підпис	Дата
Розробив		Мазур		
Перевір.		Каштальян		
Н. контр.		Кисіль		
Затв.		Павлова		

Відомість проекту

Літера	Аркуш	Аркушів
У	1	1

ХНУ, КІ2-22-2

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Планувальник задач для систем реального часу».

Автор роботи: Антон МАЗУР

Керівник роботи: Антоніна КАШТАЛЬЯН.

Пояснювальна записка: 57 с., 9 рис., 3 дод., 40 джерел.

ІНТЕРНЕТ РЕЧЕЙ, РОЗПОДІЛЕНА СИСТЕМА, ПЛАНУВАННЯ ЗАДАЧ, ЧЕРГА ПОВІДОМЛЕНЬ.

Метою роботи є розробка розподіленої універсальної системи планування та виконання задач для IoT-пристроїв, яка забезпечує автоматичний прийом, пріоритезацію, розподіл та виконання задач на множині виконавчих вузлів без втручання адміністратора. У роботі розроблено програмну систему, що реалізує архітектуру розподіленого планувальника задач. Вхідні задачі автоматично надходять до системи через API або імітаторів пристроїв, після чого проходять етапи валідації, визначення пріоритетів, вибору алгоритму планування та призначення на виконавчий вузол.

У системі реалізовано механізми самоорганізації, зокрема автоматичний розподіл задач, витіснення менш пріоритетних задач, повторне призначення задач у разі відмови виконавчих вузлів та функціонування в режимі часткової деградації. Розроблена система може бути використана для моделювання та дослідження алгоритмів планування задач у розподілених системах реального часу та в інформаційно-керуючих системах Інтернету речей.

30.05.2026

Дата

Підпис здобувача

ЗМІСТ

Вступ 3
1 Аналіз рішень та вибір стратегії і засобів для реалізації завдання 7
1.1 Аналіз предметної області і виявлення наявних проблем і завдань 7
1.2 Порівняльний аналіз переваг та недоліків існуючих рішень 13
1.3 Підходи до вирішення задачі за темою дослідження 19
1.4 Постановка задачі 20
1.5 Висновки до першого розділу 20
2 Проектування планувальника задач для систем реального часу 22
2.1 Розробка загальної архітектури системи 22
2.2 Функціонал завдань розподіленої системи та механізми їх розпаралелювання 35
2.3 Реалізація механізмів самоорганізації та відмовостійкості в архітектурі системи 38
2.4 Висновок до другого розділу 40
3 Алгоритмічне та програмне забезпечення розподіленої системи планувальника 41
3.1 Алгоритми псевдокодом з поясненнями 41
3.2 Структура і склад пз 56
3.3 Веб-базований інтерфейс 59
3.4 Практичні приклади застосування та сценарії функціонування системи 62
3.5 висновок до третього розділу 64
Висновки 66
Перелік джерел посилань 67
Додаток А Алгоритм планувальника задач 71
Додаток Б Архітектура програмного забезпечення системи 72
Додаток В Результат роботи планувальника 73

Зм.	Арк.	№ докум.	Підпис	Дата
Виконав		Антон МАЗУР		
Перевір.		Антоніна КАШТАЛЬЯН		
Н.контр.		Тетяна КИСІЛЬ		
Затвер.		Ольга ПАВЛОВА		

КвРКІ.022054.22.02.49 ПЗ

Планувальник завдань для систем реального часу. Пояснювальна записка

Літера	Аркуш	Аркушів
У	2	74

ХНУ КІ2-22-2

ВСТУП

Сучасний етап розвитку глобальної промислової автоматизації, що проходить під егідою четвертої індустріальної революції (Industry 4.0) та концепції «розумних фабрик», характеризується стрімким впровадженням та масштабуванням систем індустріального Інтернету речей (IIoT). Сучасні високотехнологічні виробництва, роботизовані конвеєрні лінії, системи управління енергомережами та безпілотний транспорт генерують щосекунди колосальні масиви телеметричної інформації. Особливістю функціонування таких гетерогенних інфраструктур є наявність жорстких або м'яких часових обмежень (дедлайнів) на обробку сигналів, що надходять від тисяч периферійних датчиків, виконавчих механізмів та програмованих логічних контролерів.

Традиційна парадигма організації обчислень, що спирається на централізовані хмарні сервіси, демонструє повну неспроможність задовольнити суворі критерії детермінованості у системах реального часу. Географічна віддаленість дата-центрів, нестабільність пропускної здатності глобальних мереж та непередбачувані затримки маршрутизації роблять хмарну інфраструктуру непридатною для критично важливих контурів управління. Будь-яка затримка у доставці або обробці керівного сигналу в таких умовах може призвести до каскадного виходу з ладу промислового обладнання, браку продукції або виникнення техногенних аварій.

Як наслідок, архітектура сучасних АСУ ТП еволюціонує в бік периферійних та туманних обчислень, де первинна обробка даних та прийняття рішень переносяться максимально близько до фізичних джерел сигналів. Проте побудова розподілених периферійних систем стикається з серйозними технологічними викликами. Існуючі універсальні планувальники операційних систем загального призначення орієнтовані на максимізацію середньої пропускної здатності, а не на гарантування дедлайнів. Водночас класичні вбудовані операційні системи реального часу є локальними, монолітними та

жорстко прив'язаними до конкретної апаратної платформи мікроконтролера. Вони не мають вбудованих інструментів для динамічного просторового балансування навантаження в мережевих кластерах, асинхронного міжсервісного обміну повідомленнями та автоматичного відновлення після раптових апаратних збоїв окремих обчислювальних вузлів. Таким чином, виникає гостра науково-практична потреба в розробці розподілених, універсальних, сервіс-орієнтованих систем планування завдань реального часу, що обумовлює вибір теми цієї кваліфікаційної роботи.

Об'єктом дослідження цієї дипломної роботи виступають безпосередні процеси часового планування, просторового розподілу, диспетчеризації та преемптивної обробки потоків гетерогенних завдань реального часу у розподілених обчислювальних середовищах. При цьому предметом дослідження є архітектурні шаблони, асинхронні алгоритми динамічного витіснення завдань, методи інтегрального балансування апаратних ресурсів та мікросервісні протоколи забезпечення відмовостійкості розподіленого планувальника.

Головною метою роботи є проєктування, алгоритмічна оптимізація та програмна реалізація відмовостійкої розподіленої універсальної системи планування задач реального часу, яка здатна функціонувати в умовах висконавантажених індустріальних мереж Інтернету речей, гарантуючи мінімальний системний джитер, абсолютну пріоритезацію з витісненням та автоматичне відновлення після збоїв без залучення системного адміністратора.

Для успішного досягнення поставленої мети в межах дипломного проєктування було визначено та послідовно вирішено комплекс інженерних завдань. Першочергово було виконано системний аналітичний огляд теоретичних засад функціонування систем реального часу, класифіковано існуючі алгоритми планування та проаналізовано обмеження чинних програмних рішень. На основі отриманих висновків було розроблено сервіс-орієнтовану архітектуру розподіленого планувальника за паттерном «Master-Worker» з повною ізоляцією контурів вхідної валідації, центральної координації

та обчислювальних потужностей кластера. Далі у роботі було формалізовано у вигляді псевдокоду математичні алгоритми тикового планування, асинхронного пакетування, динамічного витіснення та балансування навантаження. На заключних етапах проєктування було обґрунтовано вибір сучасного технологічного стеку розробки, побудовано топологію черг асинхронного брокера повідомлень, реалізовано веб-базований інтерфейс оператора та детально проаналізовано поведінку системи у штатних та аварійних сценаріях експлуатації.

Методологічну основу проведеного дослідження складають положення теорії планування реального часу, зокрема системний аналіз алгоритмів RMS та EDF, принципи проектування розподілених та високонавантажених мікросервісних систем, теорія масового обслуговування, методи асинхронного подієвого програмування та сучасні технології контейнеризації обчислювальних ресурсів.

Наукова та практична цінність отриманих результатів полягає у створенні універсального архітектурного шаблону та комплексу програмного забезпечення, що може бути використаний як готовий технологічний фундамент при побудові сучасних систем промислової автоматизації, периферійних обчислювальних шлюзів та платформ збору даних з високими вимогами до часу реакції. Запропоновані алгоритми асинхронного витіснення та балансування дозволяють гнучко адаптувати систему під специфіку конкретного промислового підприємства без кардинальної перебудови вихідного коду.

Структурно пояснювальна записка дипломної роботи складається зі вступу, трьох логічно пов'язаних розділів, загальних висновків, списку використаних літературних джерел та додатків, що містять графічний матеріал, структурні схеми та вихідний код розробленого програмного забезпечення.

Для того, щоб спроєктована архітектура розподіленого планувальника повністю задовольняла суворі індустріальні критерії детермінованості, надійності та низької латентності, необхідно детально дослідити теоретичний

фундамент побудови систем реального часу. Потреба у виявленні сильних і слабких сторін існуючих ринкових рішень, а також у формалізації точних критеріїв оцінки планувальників, обумовлює необхідність проведення глибокого аналітичного огляду літературних джерел. Саме цим питанням присвячено перший розділ дипломної роботи, результати якого стануть основою для подальшого інженерного проєктування програмного комплексу.

1 АНАЛІЗ ВІДОМИХ РІШЕНЬ ТА ВИБІР СТРАТЕГІЇ І ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ ЗАВДАННЯ

1.1 Аналіз предметної області і виявлення наявних проблем і завдань

Історичний розвиток обчислювальних систем пройшов еволюцію від простих калькуляторів до складних розподілених мереж, де на першому етапі ключовим критерієм була загальна пропускна здатність процесора [3, 4]. Проте з розвитком мікропроцесорної техніки та інтеграцією комп’ютерів у системи керування фізичними об’єктами виникла потреба в детермінізмі. Сучасні обчислювальні системи перейшли від парадигми «найшвидшого виконання» до парадигми «виконання у чітко визначений термін», що стало фундаментом для створення спеціалізованих архітектур реального часу [1, 9].

Система реального часу (Real-Time System, RTS) - це логічна система, правильність функціонування якої залежить не тільки від отриманого результату, а й від моменту часу, коли цей результат був наданий [1]. Якщо часові обмеження не виконуються, це класифікується як відмова системи [2]. У науковій літературі виділяють два критичні класи:

Жорсткі СРЧ (Hard Real-Time): де порушення часового регламенту (дедлайну) призводить до фатальних наслідків для об’єкта керування або людини [2, 5].

М’які СРЧ (Soft Real-Time): де затримка обробки даних призводить лише до деградації якості послуг, але не робить систему непрацездатною [1, 3].

Завдяки здатності обробляти події з мінімальною латентністю, СРЧ знайшли широке застосування в критично важливих галузях [6]. Основними сферами використання є: промислова робототехніка та автоматика: керування приводами та сенсорами на виробничих лініях [9], аерокосмічна галузь та авіоніка: системи навігації та автоматичного пілотування, де затримка сигналу є неприпустимою [1,5], медична апаратура: пристрої підтримки життєдіяльності

та хірургічні роботи [2], телекомунікації: комутація пакетів у мережах 5G та обробка сигналів у реальному часі [3, 8].

У системах реального часу час є таким же дефіцитним ресурсом, як пам'ять або цикли процесора [1]. На відміну від систем загального призначення, де планувальник прагне забезпечити справедливий розподіл ресурсів (Fairness), планувальник СРЧ має гарантувати Worst-Case Execution Time (WCET) - найгірший час виконання завдання [2, 10].

Важливість часу обумовлена необхідністю синхронізації обчислювальних процесів із фізичними процесами, що відбуваються в реальному світі. Будь-яка непередбачувана затримка (jitter) може викликати резонанс у механічних вузлах або втрату стійкості контуру керування [5, 9]. Таким чином, планувальник завдань стає центральним компонентом системи, що відповідає за передбачуваність та надійність всієї структури [7, 10].

Планувальник завдань (Scheduler) - це програмний механізм ядра операційної системи, відповідальний за реалізацію стратегії впорядкування доступу конкуруючих потоків (threads) або процесів до ресурсів центрального процесора (CPU) [2, 11]. У контексті систем реального часу (СРЧ) планувальник перестає бути просто інструментом розподілу часу і стає гарантом часової детермінованості [1, 13]. Його ключова роль полягає у забезпеченні такої черговості виконання, при якій кожне критичне завдання гарантовано завершується до настання свого дедлайну, навіть у ситуаціях граничного навантаження на систему [12, 17].

На відміну від ОС загального призначення (GPOS), планувальник СРЧ базується на концепції жорстких пріоритетів та мінімізації системних накладних витрат [3,14]. До його ключового функціоналу належать:

1) диспетчеризація та перемикання контексту: Процес збереження стану поточного завдання (регістри, лічильник команд) та завантаження стану наступного. Для СРЧ критично важливим є параметр latency - час, що

витрачається на це перемикання, який має бути константним і мінімальним [4, 18];

2) стратегії витіснення (Preemption): Здатність планувальника негайно перервати виконання поточного процесу, якщо з'явилося завдання з вищим пріоритетом. Це забезпечує швидку реакцію на зовнішні події [10, 16];

3) керування часовими обмеженнями (Deadline Management): Робота часовими параметрами задач - періодом, часом виконання та дедлайном. Планувальник повинен відстежувати дотримання умови C <= D <= T для кожного потоку [1, 13];

4) синхронізація та взаємодія процесів (IPC): Забезпечення безпечного доступу до спільних ресурсів (м'ютекси, семафори). Тут планувальник має вирішувати специфічні проблеми, такі як блокування та взаємовиключення [6, 15].

Однією з найскладніших проблем, яку вирішує планувальник, є інверсія пріоритетів - ситуація, коли низькопріоритетне завдання утримує ресурс, необхідний високорпіоритетному, що призводить до непередбачуваної затримки останнього [10, 15]. Для подолання цього використовують протоколи успадкування пріоритетів (Priority Inheritance Protocol, PIP) та протоколи пріоритетної межі (Priority Ceiling Protocol, PCP) [15, 19].

Іншим важливим аспектом є аналіз вчасності (Schedulability Analysis). Планувальник повинен не просто перемикати завдання, а бути частиною системи, для якої можна математично довести, що за жодних умов дедлайни не будуть пропущені. Це досягається шляхом перевірки завантаженості процесора за формулами для статичних пріоритетів (RMS) або динамічних (EDF) [13, 20].

Планувальник тісно взаємодіє з підсистемою обробки переривань. У СРЧ важливо, щоб обробники переривань (ISR) були максимально короткими, передаючи основну роботу планувальнику, щоб не порушувати часові гарантії інших потоків [8, 11]. Сучасні архітектури також вимагають від планувальника

підтримки багатоядерності (SMP), що вносить додаткову складність у балансування навантаження при збереженні детермінованості [9, 14].

Вибір алгоритму планування визначає здатність системи реального часу (СРЧ) виконувати свої часові зобов'язання. Алгоритми поділяються на статичні (пріоритети фіксуються до запуску) та динамічні (пріоритети змінюються під час виконання) [13, 21].

Алгоритм циклічного планування (Round Robin, RR). Хоча Round Robin широко використовується в операційних системах загального призначення, у системах реального часу він застосовується переважно для завдань з однаковим пріоритетом або у «м'яких» СРЧ [3, 22]. Суть алгоритму полягає у виділенні кожному завданню фіксованого кванта часу. Основною проблемою RR у жорстких СРЧ є низька передбачуваність часу відгуку для критичних подій, оскільки виконання важливого завдання може бути відкладене через чергу менш пріоритетних процесів [4, 11].

Планування на основі пріоритетів (Fixed Priority Scheduling, FPS) Це найбільш розповсюджений підхід у комерційних ОСРЧ (наприклад, FreeRTOS, VxWorks). Кожному завданню присвоюється унікальний рівень пріоритету [8, 16].

Rate Monotonic Scheduling (RMS): Оптимальний алгоритм статичного планування для періодичних завдань. Згідно з RMS, чим менший період завдання (тобто чим частіше воно має виконуватися), тим вищий його пріоритет [13, 23]. Фундаментальне дослідження Лью та Лейланда довело, що для набору з n завдань успішне планування гарантоване, якщо завантаження процесора не перевищує $n(2^{\frac{1}{n}} - 1)$, що при великих n прямує до ≈ 69 [13, 24].

$$n\left(2^{\frac{1}{n}} - 1\right), \tag{1.1}$$

де n – кількість завдань;

Динамічне планування (Dynamic Priority Scheduling) На відміну від статичних методів, динамічні алгоритми перераховують пріоритети в реальному часі залежно від стану черги завдань.

Earliest Deadline First (EDF): Алгоритм, у якому найвищий пріоритет отримує завдання, дедлайн якого настане найшвидше [2, 25]. EDF є оптимальним у тому сенсі, що якщо набір завдань взагалі можна спланувати, то EDF це зробить. Його теоретична межа завантаження процесора становить 100%, що робить його ефективнішим за RMS, проте він складніший у реалізації та менш стійкий до перевантажень системи [13, 26].

Least Laxity First (LLF): Пріоритет базується на «запасі часу» (laxity), який розраховується як різниця між часом до дедлайну та залишком часу виконання завдання. Це дозволяє ще гнучкіше реагувати на критичні ситуації, але потребує частих перемикань контексту [12, 27].Порівняльний аналіз та вибір стратегії

Аналіз показує, що для систем із жорсткими часовими обмеженнями частіше обирають RMS через його простоту та передбачуваність при перевантаженнях (алгоритм просто відсікає найменш пріоритетні задачі). EDF частіше використовується в складних вбудованих системах, де важливо максимізувати використання обчислювальних ресурсів [5, 28]. При розробці планувальника необхідно враховувати не лише математичну модель, а й апаратні затримки на перемикання контексту та обробку переривань [18, 29].

Проблеми та критичні стани в плануванні завдань. Навіть при використанні перевірених алгоритмів, динамічна природа систем реального часу може призводити до виникнення ряду специфічних проблем, що ставлять під загрозу цілісність системи.

Інверсія пріоритетів. Це найбільш критична проблема в системах, що використовують спільні ресурси (м'ютекси, семафори). Вона виникає, коли високопріоритетне завдання змушене чекати на звільнення ресурсу, який утримується низькопріоритетним завданням, а те, у свою чергу, витісняється завданням із середнім пріоритетом [10, 15]. У результаті час очікування

високорпіоритетної задачі стає непередбачуваним, що порушує принцип детермінованості [2, 19].

Голодування задач. Ефект «голодування» виникає переважно в алгоритмах зі статичними пріоритетами (наприклад, FPS або RMS). Якщо потік високорпіоритетних завдань є безперервним або занадто інтенсивним, низькопріоритетні завдання можуть ніколи не отримати доступ до CPU [3, 12]. У системах загального призначення це вирішується механізмом «старіння», але в жорстких СРЧ такий підхід є неприпустимим, оскільки він змінює пріоритети всупереч логіці безпеки [1, 23].

Перевантаження системи (Transient Overload). Ситуація перевантаження виникає, коли сумарний запит на обчислювальні ресурси перевищує можливості процесора (U > 1.0). У системах з EDF перевантаження може спричинити «ефект доміно», де пропуск одного дедлайну тягне за собою затримку всіх наступних задач [2, 13]. Для стабільних систем важливо впроваджувати механізми контролю входу (admission control) або стратегії вибіркового скидання завдань (task shedding), щоб гарантувати виконання хоча б найбільш критичного набору функцій [12, 26].

Часова неозначеність та Jitter. Варіативність часу відгуку, або джиттер, виникає через затримки в обробці переривань та неоднаковий час перемикання контексту залежно від стану кеш-пам'яті або конвеєрів процесора [11, 18]. Хоча середня швидкість може бути високою, велике значення джиттера є критичним для систем цифрового керування, де точність вимірювань залежить від суворої періодичності опитування сенсорів [9, 14].

Вирішення цих проблем потребує не лише вибору алгоритму, а й правильного налаштування протоколів взаємодії між завданнями, що є ключовим етапом при проектуванні планувальника для конкретної вбудованої системи [5, 30].

1.2 Порівняльний аналіз переваг та недоліків існуючих рішень

Для вибору оптимальної архітектури планувальника необхідно провести порівняльний аналіз механізмів, що реалізують базові функції керування процесами. Основна відмінність між існуючими рішеннями (наприклад, комерційними ОСРЧ, такими як QNX, та вільнопоширеними, як FreeRTOS) полягає в способі досягнення балансу між продуктивністю та детермінованістю [1, 8].

Для розподілу процесорного часу та забезпечення дедлайнів існуючі рішення зазвичай базуються або на принципах квантування часу , або на подійно-орієнтованому плануванні (Event-driven). Зокрема, підхід із використанням квантування за алгоритмом Round Robin забезпечує справедливий розподіл обчислювальних ресурсів між потоками, проте в системах реального часу він часто призводить до порушення дедлайнів критичних задач через непередбачуване очікування своєї черги [3, 22].

З іншого боку, подійно-орієнтовані системи з підтримкою механізмів витіснення (preemptive) значно краще гарантують дотримання жорстких часових обмежень, але водночас потребують складного математичного аналізу на етапі проєктування з метою повного уникнення конфліктів доступу до спільних ресурсів [2, 11].

Для підтримки пріоритетів та мінімізації затримок більшість сучасних операційних систем реального часу орієнтуються на підтримку до 256 рівнів пріоритетів, що дозволяє гнучко налаштовувати обчислювальні потоки під специфіку конкретного виробництва. Проте безпосередній механізм обробки цих пріоритетів критично впливає на загальну латентність усієї системи [14, 16].

Зокрема, використання статичних пріоритетів у рамках алгоритму RMS супроводжується низькими накладними витратами на обчислення наступної задачі, що дозволяє мінімізувати час роботи самого планувальника та знизити системні затримки [13, 23]. На противагу цьому, підходи з динамічними

пріоритетами, такі як алгоритм EDF, хоча й дозволяють завантажити обчислювальний процесор майже на 100%, проте вносять значні затримки на кожному кроці планування, що зумовлено необхідністю постійного ресурсомісткого сортування черги дедлайнів під час надходження нових запитів [2, 13].

Для перемикання задач та системних витрат ефективність функціонування планувальника критично залежить від швидкості перемикання контексту обчислень. Сучасні рішення на базі мікроядер (наприклад, ОС QNX) забезпечують високу модульність та безпеку, проте кожне перемикання між завданнями в такій архітектурі може потребувати повної зміни адресного простору, що суттєво збільшує системні затримки [4, 9].

На противагу цьому, монолітні або бібліотечні операційні системи реального часу, такі як FreeRTOS, виконують перемикання контексту максимально швидко саме за рахунок використання спільного адресного простору, але водночас мають нижчий рівень захисту пам’яті від критичних помилок у коді окремих завдань [8, 11]. Через це формування технічних вимог до планувальників задач реального часу суттєво відрізняється від проєктування систем загального призначення. Якщо для десктопних операційних систем головним критерієм виступає середня швидкість реагування на запити користувача (пропускна здатність), то для ОСРЧ на перший план виходять параметри абсолютної стабільності та детермінованості у найгірших сценаріях пікового навантаження [1, 11].

Детермінованість (Determinism). Це фундаментальна вимога, яка визначає здатність планувальника надавати однаковий результат за однаковий час при ідентичних вхідних умовах [9]. Детермінованість означає, що час, необхідний для прийняття рішення про запуск наступного завдання, не повинен залежати від кількості активних процесів у черзі або стану зовнішніх переривань [12, 14]. Будь-яка нелінійність у роботі алгоритмів планування робить систему непридатною для жорсткого реального часу.

Передбачуваність (Predictability). Тісно пов'язана з детермінованістю, передбачуваність дозволяє розробнику на етапі проектування математично довести, що всі часові обмеження будуть виконані [2, 13]. Це досягається за допомогою аналізу вчасності (schedulability analysis). Планувальник повинен надавати механізми, які дозволяють чітко розрахувати найгірший час виконання завдання (WCET), враховуючи всі можливі системні затримки [5, 26].

Мінімізація затримок (Latency Requirements). Для планувальника критичними є три види затримок:

1) interrupt latency: час від виникнення апаратного переривання до початку виконання його обробника;

2) scheduling latency: час, необхідний планувальнику для вибору найбільш пріоритетної задачі;

3) context switch latency: час на фізичне перемикання контексту процесора [4, 18].

Вимогою до сучасних систем є зведення цих значень до константних величин, що вимірюються мікросекундами [8, 29].

Стабільність та стійкість (Stability). Система повинна залишатися керованою навіть у стані перевантаження (transient overload). У разі, якщо всі завдання не можуть бути виконані вчасно, планувальник зобов'язаний забезпечити виконання найбільш критичних потоків, жертвуючи менш важливими [10, 12]. Це виключає виникнення «ефекту доміно», коли одна затримка призводить до повного краху системи [13, 27].

Ефективність використання ресурсів (Efficiency). Хоча надійність є пріоритетом, планувальник не повинен витрачати надмірну кількість ресурсів CPU та пам'яті на власні потреби. Ефективність визначається як відношення часу корисної роботи прикладних завдань до загального часу роботи системи. В ідеальних ОСРЧ системні витрати (overhead) на роботу планувальника не повинні перевищувати 1-3% загальної потужності процесора [3, 21].

Типізація планувальників реального часу. Класифікація планувальників дозволяє визначити межі застосування конкретного архітектурного рішення залежно від вимог об’єкта керування. Систематизація проводиться за декількома ключовими ознаками [2, 31].

За алгоритмом призначення пріоритетів, що є найбільш фундаментальною ознакою, визначається безпосередня математична модель поведінки всієї обчислювальної системи. У межах цього підходу виділяють планування зі статичними пріоритетами (Fixed Priority Scheduling, FPS), де пріоритети завдань жорстко визначаються ще на етапі розробки та не змінюються під час виконання. Класичним прикладом тут виступає алгоритм Rate Monotonic Scheduling (RMS) [13]. Такі планувальники активно використовуються в критичних системах, де ключову роль відіграє простота сертифікації софту та повна передбачуваність поведінки заліза, як це реалізовано, наприклад, у ядрі операційної системи RTEMS [32].

З іншого боку, планування з динамічними пріоритетами (Dynamic Priority Scheduling) передбачає, що пріоритети розраховуються безпосередньо «на льоту» залежно від дедлайнів або залишку часу, необхідного на виконання кожної задачі. Найвідомішим представником цього напряму є алгоритм Earliest Deadline First (EDF) [25]. Динамічні планувальники дозволяють досягти значно вищої утилізації процесорного часу розподіленого кластера, що є критично важливим для складних мультимедійних або сучасних високонавантажених мережевих систем [1, 14].

За типом систем реального часу. Типізація за рівнем жорсткості часових обмежень визначає механізми обробки помилок у планувальнику:

1) жорсткі СРЧ (Hard Real-Time): Планувальник повинен гарантувати виконання дедлайну в 100% випадків. Прикладом системи з жорстким плануванням є QNX Neutrino, де кожен процес має строго визначений бюджет часу [9, 33];

2) м’які СРЧ (Soft Real-Time): Допускаються епізодичні пропуски дедлайнів, що не призводять до краху. Типовим представником є Linux з патчем PREEMPT_RT, який покращує чутливість системи, але не дає математичних гарантій жорсткого реального часу [3, 34].

За структурою та механізмом перемикання (Preemption). Ця категорія визначає, наскільки оперативно планувальник може реагувати на зовнішні події:

1) витісняючі планивальники (Preemptive Scheduling): Поточне завдання може бути призупинене в будь-який момент, якщо з'явилося завдання з вищим пріоритетом. Це стандарт для сучасних ОСРЧ, таких як FreeRTOS [8, 35];

2) невитісняючі (кооперативні) планувальники (Non-preemptive / Cooperative): Завдання саме передає управління планувальнику після завершення певного етапу роботи. Це спрощує проектування (немає проблем із спільним доступом до даних), але робить систему вразливою до «зависання» одного завдання, що блокує всю чергу [4, 11].

За кількістю процесорів

1) однопроцесорні (Uniprocessor): Класичні моделі, де планувальник працює з однією чергою завдань [1, 13];

2) багатопроцесорні / багатоядерні (Multiprocessor Scheduling): Планувальник повинен вирішувати завдання міграції потоків між ядрами (Global Scheduling) або закріплення їх за конкретними ядрами (Partitioned Scheduling), що значно ускладнює аналіз вчасності [21, 31].

Для розуміння сучасних стандартів у галузі планування завдань необхідно розглянути найбільш вживані операційні системи реального часу (ОСРЧ), які де-факто є індустріальними стандартами. Кожна з них реалізує власні підходи до архітектури планувальника, обробки переривань та керування ресурсами [8, 11].

FreeRTOS (Real Time Engineers Ltd. / Amazon Web Services). FreeRTOS на сьогодні є найпопулярнішою системою для мікроконтролерів завдяки своїй мінімалістичності та відкритому коду [35].

Переваги:

1) простота та компактність: Ядро системи займає всього кілька кілобайт, що дозволяє використовувати її на пристроях із суворими обмеженнями пам'яті;

2) висока швидкість: Мінімальний час перемикання контексту та латентність переривань роблять її ідеальною для завдань, що потребують швидкої реакції;

3) відкритий вихідний код: Ліцензія MIT дозволяє вільно модифікувати систему під конкретні потреби проекту [8];

4) популярність та екосистема: Величезна кількість документації та прикладів значно полегшує процес розробки;

Недоліки:

1) обмежений функціонал: Система надає лише базові механізми (завдання, черги, семафори), залишаючи реалізацію драйверів та складних мережевих стеків на розробника;

2) прості алгоритми планування: В основному використовується фіксована пріоритетність із витісненням та Round Robin для однакових пріоритетів. Складні механізми (наприклад, EDF) за замовчуванням відсутні [35];

Zephyr RTOS (Linux Foundation). Zephyr - це сучасна масштабована система, яка розробляється як універсальне рішення для Інтернету речей (IoT) [36].

Переваги:

1) сучасна архітектура: Система побудована за модульним принципом, що дозволяє легко додавати або видаляти компоненти;

2) багатий набір можливостей: Включає вбудовані стеки для Bluetooth, Wi-Fi, Ethernet, а також підтримку багатьох файлових систем;

3) безпека: Має вбудовані механізми захисту пам'яті та безпечного завантаження, що важливо для промислових пристроїв.

Недоліки:

1) висока складність: Процес конфігурації через DeviceTree та Kconfig є складнішим у порівнянні з FreeRTOS і потребує більше часу на навчання;

2) ресурсомісткість: Через багатий функціонал Zephyr потребує більше оперативної та флеш-пам'яті, що може бути критичним для найпростіших 8- або 16-бітних контролерів [36];

3) vxworks (Wind River Systems). VxWorks - це одна з найбільш стабільних та надійних комерційних ОСРЧ у світі, яка використовується в космічних апаратах (наприклад, марсоходах NASA) та авіоніці [12, 37].

1.3 Підходи до вирішення задачі за темою дослідження

Так як необхідно реалізувати підсистему планування задач у системі реального часу, доцільним є використання сучасних обчислювальних платформ, що забезпечують підтримку багатозадачності та ефективне керування ресурсами. У якості програмної основи може бути використана операційна система сімейства Linux або спеціалізовані системи реального часу. Для реалізації алгоритмів планування задач можуть застосовуватися мови програмування загального призначення, зокрема C або C++, які забезпечують достатню продуктивність та можливості низькорівневого керування ресурсами системи.

Для розробки програмного забезпечення можуть використовуватися стандартні бібліотеки роботи з потоками виконання та засоби синхронізації задач. Також доцільним є використання сучасних засобів розробки програмного забезпечення та систем контролю версій, що забезпечують зручність створення та супроводу програмної системи.

Обрані підходи дозволяють забезпечити ефективну реалізацію підсистеми планування задач та створюють можливість подальшого розширення функціональних можливостей системи.

1.4 Постановка задачі

Завданнями роботи є:

1) дослідити принципи функціонування планувальників задач у системах реального часу;

2) провести теоретичний аналіз предметної області систем реального часу;

3) охарактеризувати структуру предметної області та побудувати базову модель планувальника задач;

4) описати існуючі механізми реалізації планувальників задач, визначити їх переваги та недоліки, а також виділити основні проблеми та можливі шляхи їх вирішення;

5) на основі проведених досліджень визначити основні функції системи, сформувати функціональні та нефункціональні вимоги до планувальника задач та розробити модель його роботи;

6) обґрунтувати необхідність розробки планувальника задач для систем реального часу;

7) визначити об’єкт та мету подальших досліджень;

8) оцінити ступінь виконання поставлених завдань.

На основі проведених досліджень необхідно розробити працездатний планувальник задач для системи реального часу та зробити висновки щодо результатів виконаної роботи.

1.5 Висновки до першого розділу

У першому розділі було проведено детальний аналітичний огляд теоретичних засад функціонування систем реального часу (СРЧ) та визначено фундаментальну роль підсистеми планування завдань у забезпеченні їхнього часового детермінізму. Дослідження предметної області показало, що на відміну від операційних систем загального призначення, де головним критерієм є

максимальна середня пропускна здатність, у системах реального часу визначальними вимогами є передбачуваність поведінки, своєчасність обробки запитів та гарантоване дотримання жорстких або м'яких часових обмежень (дедлайнів) [1, 2]. Визначено та класифіковано ключові вимоги до архітектури сучасних планувальників задач реального часу. Встановлено, що для успішного функціонування в умовах високоінтенсивних потоків даних від периферійних пристроїв (IoT/ПЛК), планувальник повинен мати низький рівень системного джитера, підтримувати механізми абсолютної або динамічної пріоритезації завдань, а також володіти інструментами преемптивності (витіснення) для негайного надання обчислювальних ресурсів критично важливим аперіодичним процесам [5, 10]. Проведено порівняльний аналіз відомих класичних алгоритмів планування реального часу, таких як RMS (Rate Monotonic Scheduling) та EDF (Earliest Deadline First), а також досліджено чинні ринкові та відкриті реалізації на базі спеціалізованих операційних систем (RTOS, зокрема Zephyr, FreeRTOS, RTEMS) [19, 21]. Встановлено, що більшість існуючих рішень орієнтовані на локальне монолітне використання всередині мікроконтролерів або вбудованих систем і не мають вбудованих засобів для автоматичного масштабування, гнучкого просторового балансування навантаження та самоорганізації у великих розподілених мережах (Fog/Cloud середовищах) [7, 26]. На основі виявлених недоліків відомих реалізацій та з урахуванням специфіки сучасних розподілених індустріальних IoT-платформ було сформульовано та обґрунтовано постановку задачі на дипломне проєктування. Визначено необхідність розробки сервіс-орієнтованого архітектурного рішення, яке поєднувало б у собі класичні принципи преемптивного тикового планування реального часу із сучасними інструментами асинхронної маршрутизації повідомлень, динамічного балансування обчислювальних вузлів та автономного відновлення після апаратних збоїв без залучення системного адміністратора.

2 ПРОЕКТУВАННЯ ПЛАНУВАЛЬНИКА ЗАДАЧ ДЛЯ СИСТЕМ РЕАЛЬНОГО ЧАСУ

2.1 Розробка загальної архітектури системи

Для детального аналізу принципів функціонування проектованої системи необхідно розглянути її компонентну структуру та визначити системну роль кожного елемента. Відповідно до розробленої структурної схеми (Рисунок 2.1), процес обробки інформації та керування обчислювальними ресурсами розподілений між логічними модулями, що взаємодіють через стандартизовані інтерфейси [3, 4].

IoT Users (Джерела запитів та даних). Початковою ланкою системи є множина кінцевих користувачів або індустріальних IoT-пристроїв (периферійних датчиків, контролерів тощо). Головне призначення цього рівня полягає у безперервному зборі метрик і фізичних параметрів зовнішнього середовища та їх перетворенні у цифрові запити на обробку інформації [1, 7]. Оскільки трафік від периферійних пристроїв є високоінтенсивним і може мати спорадичний характер, безпосереднє передавання кожного поодинокого запиту до центрального вузла є неефективним через критичне зростання накладних витрат мережевого рівня [8, 9].

Batch (Модуль часового квантування та групування). Для мінімізації негативного впливу мережевих затримкових коливань (джитера) запити від IoT-пристроїв спрямовуються до спеціалізованого модуля накопичення. Його системна роль полягає у квантуванні вхідного потоку за часом та формуванні пакетів завдань (Batch Of Real-Time Tasks) [5]. Робота модуля регламентується строго визначеним часовим інтервалом t. Використання механізму групування дозволяє згладжувати пікові навантаження та гарантувати, що накопичений пакет буде відправлений до черги не пізніше встановленого критичного ліміту часу [10]. При цьому архітектурою передбачено контур негайного переривання накопичення при виникненні аперіодичних подій з високим рівнем критичності

(Elapsed Time < t), що дозволяє миттєво доставляти аварійні запити до обчислювального ядра [1, 2].

Task Validator (Модуль валідації та попереднього аналізу). Перш ніж сформований пакет завдань потрапить до системної черги, він проходить етап верифікації у модулі валідації. Системна роль Task Validator полягає у проведенні статичного аналізу метаданих кожної задачі, перевірці цілісності структури пакета та первинній оцінці часової виконуваності дедлайнів [2, 12]. Якщо завдання містить некоректні параметри або його жорсткий дедлайн гарантовано не може бути дотриманий через поточний стан затримок у мережі, валідатор відкидає таку задачу на вході. Це запобігає забиванню черги завідомо невиконуваними процесами та оптимізує витрати ресурсів центрального процесора [26, 29].

Task Queue (Очередь завдань). Валідовані завдання, готові до планування, надходять до буфера проміжного зберігання - Task Queue. Системна роль черги полягає у забезпеченні асинхронності між етапом генерації/валідації задач та етапом їх безпосереднього розподілу на обчислювачі [4, 12]. У контексті систем реального часу цей компонент є багатоуровневою структурою, яка не просто накопичує елементи за принципом FIFO, а динамічно ранжує та впорядковує завдання на основі їхніх часових обмежень, пріоритетів чи близькості критичних дедлайнів, готуючи їх до вилучення центральним керуючим вузлом [10, 27].

Master Node (Центральний керуючий вузол). Master Node виступає як головний оркестратор та єдиний центр прийняття рішень у розподіленому середовищі [3, 16]. Цей вузол не виконує прикладні обчислення, а акумулює в собі всю керуючу логіку системи. Фізично він ізольований від виконавчих серверів, що підвищує загальну відмовостійкість: збій на будь-якому виконавчому рівні не порушує цілісність ядра планування [12, 17].

Всередині Master Node взаємодіють три ключові підсистеми:Task Scheduler (Диспетчер-планувальник): Системна роль полягає у вилученні задач із черги та визначенні чіткого часового порядку їх виконання. Він працює на основі

математичних моделей планування реального часу, аналізує дедлайни та пріоритети, формуючи оптимальну послідовність запуску процесів [1, 22].

Load Balancer (Балансувальник навантаження). Відповідає за просторовий розподіл завдань. Якщо планувальник визначає коли задача має виконуватися, то балансувальник вирішує де саме вона буде запущена. Він забезпечує рівномірне завантаження виконавчих потужностей та координує паралельне виконання незалежних гілок обчислень [3, 28].

Resource Monitor (Модуль моніторингу обчислювальних ресурсів). Виконує роль контуру зворотного зв'язку [7]. Він безперервно збирає телеметрію та сигнали доступності від виконавчих вузлів, формуючи в оперативній пам'яті Мастера актуальну карту стану всієї системи. Дані монітора використовуються балансувальником для прийняття рішень щодо виділення ресурсів під нові задачі [16, 17].

Worker Nodes (Множина виконавчих вузлів). Worker Nodes представлені мережею ізольованих обчислювальних серверів або віртуальних машин. Їх функціональне призначення - виключно виконання безпосереднього коду прикладних завдань, надісланих від Master Node [12, 16]. Кожен воркер працює у автономному режимі, приймає команди через мережеві сокети та забезпечує багатопотокову або багатопроцесорну ізоляцію обчислень, щоб важкі фонові завдання не перешкоджали швидкій обробці критичних високопріоритетних сигналів [4, 13]. Також виконавчі вузли зобов'язані періодично відправляти повідомлення про свій поточний стан процесора та пам'яті до модуля моніторингу Мастера [17].

Results (Модуль збору та агрегації результатів)Після завершення обчислень на рівні Worker Nodes, сформовані результати виконання не повертаються назад у планувальник, щоб не створювати додаткового навантаження на ядро системи [3]. Вони транслюються до виділеного модуля агрегації результатів (Results), який веде облік успішно виконаних задач, формує фінальні пакети відповідей і відправляє їх кінцевим IoT Users [12]. Крім того, цей

модуль фіксує статистику дотримання дедлайнів, що необхідно для подальшого аналізу ефективності алгоритмів планування [25], [26].

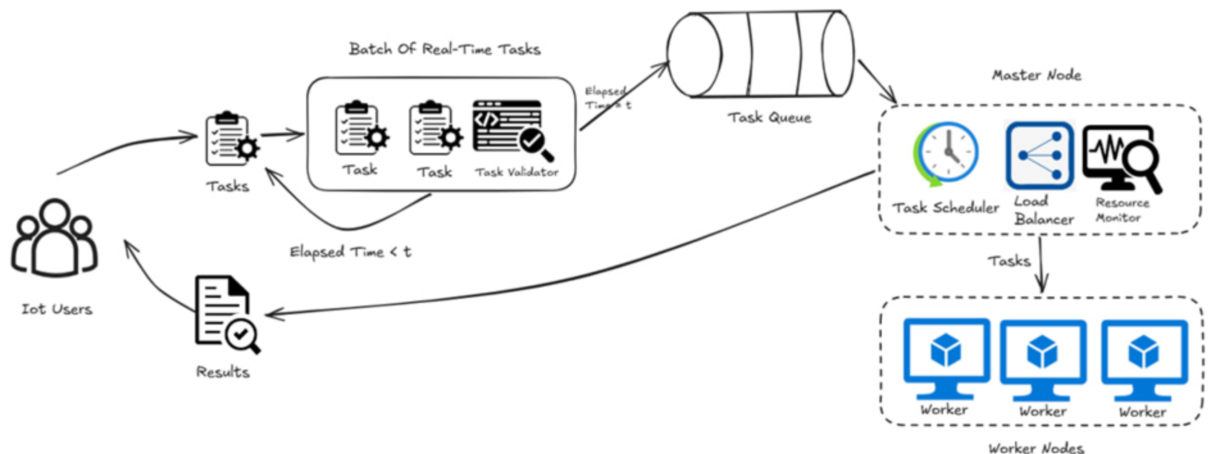


Рисунок 2.1 – Структурна схема планувальника задач

Після визначення концептуальних меж системи реального часу, необхідно перейти до її детального технічного проєктування. Усі програмні модулі розподіленої системи (Master Node, Batching Engine, Worker Nodes) реалізовані з використанням екосистеми мови програмування Python. Вибір Python обумовлений високою швидкістю розробки, наявністю зрілих бібліотек для асинхронного мережевого взаємодії (asyncio) та низьким системним оверхедом при інтеграції з брокерами повідомлень, що дозволяє мінімізувати джитер при диспетчеризації задач [13, 14].

Оскільки проектована система орієнтована на концепцію туманних обчислень (Fog Computing), як кінцеві пристрої (IoT Users) у роботі обрано комплекс промислових контролерів та сенсорів, що функціонують у складі автоматизованих систем керування (АСУ ТП) [7].

Для забезпечення повноти дослідження, джерела запитів розділені на три класи апаратних пристроїв:Промислові програмовані логічні контролери (ПЛК) серії Siemens S7-1500: Вони підключені до виконавчих механізмів (сервоприводів, клапанів) і генерують регулярні періодичні задачі розрахунку керуючого сигналу з жорсткими дедлайнами (D = 10 мс) [16].

Множина інтелектуальних датчиків вібрації та температури (на базі мікроконтролерів ESP32-WROOM-32E): Вони виконують безперервний моніторинг стану підшипників турбін і транслюють телеметричний потік завдань із середнім пріоритетом [17].Аварійні оптичні сенсори бар'єрів безпеки: Вони працюють у спорадичному режимі. При фізичному перетині лінії безпеки людиною або стороннім об'єктом, сенсор миттєво генерує аперіодичне завдання екстреної зупинки конвеєра з найвищим пріоритетом та критичним часовим обмеженням [2].

Компонент Tasks Batching функціонує як локальний мережевий шлюз (Edge Gateway), написаний на Python із використанням асинхронного таймера. Його завдання - агрегувати поодинокі виклики від датчиків у структуризовані пакети [8].

Алгоритм пакетування спирається на наступні часові обмеження:

1) При отриманні першого завдання запускається таймер.
2) Поточний час накопичення пакета визначається як:

$$Elapsed\ Time = T_{current} - T_{start} \quad (2.1)$$

де $T_{current}$ – поточний час;

T_{start} – час початку;

Якщо $Elapsed\ Time \geq \tau$, то пакет закривається та відправляється далі.

Механізм відкидання застарілих задач. Перед закриттям пакета модуль перевіряє часову валідність кожної задачі. Якщо для задачі і виконується умова:

$$T_{current} = D_i - C_i \quad (2.2)$$

де D_i – дедлайн завдання;

C_i – чистий час, який воркер витратить на процесорні обчислення цієї задачі;

це означає, що навіть за умови миттєвої передачі на воркер задача вже фізично не встигне виконатися до настання дедлайну D_i. Таке завдання маркується як застаріле, вилучається з пакета та утилізується без відправлення на Master Node, що суттєво знижує паразитне навантаження на мережу [5, 26].

Пробиття буфера: Якщо Batch Engine отримує задачу з найвищим пріоритетом (priority == 31), алгоритм негайно зупиняє асинхронний таймер, закриває пакет достроково $Elapsed\ Time < \tau$ і миттєво транслює його на етап валідації [1].

Task Validator реалізований як високопродуктивний Python-демон, що виконує первинну семантичну та часову перевірку. Оскільки система працює в парадигмі реального часу, валідатор повинен динамічно оцінювати мережевий стан [29].Для цього модуль взаємодіє з підсистемою моніторингу і володіє актуальним значенням середньої мережевої затримки до кластера воркерів ($T_{latency}$). Для кожного завдання і всередині пакета валідатор обчислює нерівність динамічної виконуваності:

$$D_i - T_{current} \geq C_i + T_{latency} + \Delta \qquad (2.3)$$

де D_i – дедлайн завдання;

$T_{current}$ – поточний час;

C_i – чистий час, який воркер витратить на процесорні обчислення цієї задачі;

$T_{latency}$ – час затримки;

Δ - фіксований часовий запас (квант безпеки) на переключення контексту планувальником [22];

Якщо нерівність не виконується, задача відкидається на етапі входу з логуванням метрики пропущеного дедлайну. Завдання, що успішно пройшли фільтрацію, переводяться в стан Inited і передаються в брокер повідомлень [12].

Для організації розподіленої взаємодії між компонентами системи, забезпечення асинхронності, ізоляції потоків даних та гарантування детермінованості часових затримок у роботі планувальника реального часу було обрано брокер повідомлень RabbitMQ [3, 4].

У розподілених системах реального часу вибір проміжного програмного забезпечення (Middleware) для обміну повідомленнями є критичним. Під час проєктування системи розглядалися три альтернативні варіанти: Apache Kafka, Redis (Pub/Sub) та RabbitMQ. Вибір було зроблено на користь RabbitMQ з огляду на такі архітектурні переваги:

Апаратна підтримка пріоритетності черг (Priority Queues): На відміну від Apache Kafka, яка оптимізована під лінійні лог-потоки (append-only logs) і вимагає створення окремих топіків для різних пріоритетів, RabbitMQ має вбудовану підтримку прапора x-max-priority [14]. Це дозволяє динамічно сортувати повідомлення всередині однієї черги на рівні операційної пам'яті брокера за константний час.

Гнучка топологія маршрутизації (AMQP Exchanges): Протокол AMQP 0-9-1, який лежить в основі RabbitMQ, надає механізми обмінників (Direct, Topic, Fanout). Це дозволяє розділити системний трафік (телеметрію), керуючі команди планувальника та прикладні завдання за допомогою точних маршрутних ключів (routing_key) [12].

Ізоляція та низька латентність доставки: Redis Pub/Sub не забезпечує гарантовану доставку при відключенні споживача (механізм «fire-and-forget»), що є неприпустимим для жорстких дедлайнів реального часу. RabbitMQ гарантує доставку повідомлень за рахунок механізму підтверджень (AMQP ACK) та транзакційної цілісності, зберігаючи при цьому мікросекундний рівень затримок при передачі пакетів у RAM-режимі [4, 13].

В архітектурі розроблюваної системи реалізовано розподілену топологію, що включає три типи обмінників (Exchanges) та ізольовані черги. Це повністю

виключає взаємне блокування або конкуренцію за ресурси між високопріоритетним прикладним і фоновим системним трафіком [12].

Аналіз компонентів топології черг

1) Центральна черга задач (Task Queue). Сюди модуль Task Validator публікує успішно перевірені пакети завдань від IoT-користувачів. Маршрутизація здійснюється через вхідний обмінник Ingress_Exchange типу Direct.Механізм пріоритезації: При ініціалізації черги task_ready_queue їй присвоюється системний прапор x-max-priority: 31, що активує 32-рівневу шкалу пріоритетів (від 0 — найнижчий фоновий, до 31 — найвищий аварійний) [27].

2) оптимізація всередині брокера: Коли повідомлення з пріоритетом 31 потрапляє в чергу, RabbitMQ не ставить його в кінець, а за допомогою алгоритму балансування бінарної кучі (heap) миттєво переміщує його на початок структури даних черги. Master Node, який безперервно слухає цю чергу, отримує аварійну задачу першою, мінімізуючи час очікування $T_{waiting} \rightarrow 0$ [14];

3) черга системного моніторингу (System State Queue)Для збору телеметрії від обчислювальних вузлів виділено окрему чергу worker_state_queue. Вона підключена до вхідного обмінника за допомогою ключа sys_routing.Роль в архітектурі: Винесення серцебиттів (heartbeats) та метрик завантаження процесорів воркерів в окремий логічний канал гарантує, що модуль Resource Monitor отримуватиме актуальну інформацію про стан системи без затримок. Навіть якщо черга прикладних завдань Task Queue буде повністю перевантажена, системні метрики пройдуть через свій ізольований обмінник миттєво, що дозволить системі вчасно зреагувати на аварію (Failover) [7, 16];

4) черги індивідуального керування воркерами (Worker-Specific Queues)Для забезпечення просторового планування та реалізації механізму витіснення завдань (Preemption), Master Node повинен мати прямий канал зв'язку з кожним конкретним виконавчим вузлом. Для цього в брокері створюється масив черг, кількість яких відповідає кількості розгорнутих воркерів (наприклад, worker_queue_1, worker_queue_2).Логіка адресації: Load Balancer надсилає

задачу або керуючий сигнал (наприклад, команду переривання SIG_SUSPEND) в обмінник керування, використовуючи точний маршрутний ключ, що містить унікальний ідентифікатор воркера (routing_key = worker_id). Кожен воркер підписаний виключно на свою чергу, що виключає колізії, коли один воркер міг би випадково перехопити задачу, призначену для іншого обчислювального вузла [12];

5) черга агрегації результатів (Results Queue)Після успішного завершення обчислень на рівні виконавчого вузла, сформований результат (Data Payload) публікується воркером у виділений обмінник Results_Exchange. Звідти повідомлення потрапляє в чергу system_results_queue.Архітектурна ізоляція: Master Node і ядро планувальника повністю ізольовані від цієї черги і не витрачають такти процесора на її обробку. Чергу результатів слухає незалежний асинхронний сервіс Results. Це дозволяє передавати великі об'єми вихідних даних назад до IoT Users без ризику уповільнення роботи критично важливого обчислювального ядра планувальника [3, 12].

Task Scheduler є фундаментальним програмним компонентом і ключовим елементом управління усередині Master Node. Його головне системне призначення — забезпечення часового детермінізму розподіленої системи, динамічне керування пріоритетами та максимізація ймовірності успішного виконання завдань у межах їхніх жорстких дедлайнів [1, 2].

Концепція віртуального квантування часу (Tick-Based Scheduling). У класичних локальних операційних системах реального часу (RTOS, таких як FreeRTOS або Zephyr) планувальник спирається на апаратні переривання системного таймера (системні тики) [19, 20]. Оскільки розроблювана система функціонує на прикладному рівні у розподіленому середовищі, використання прямих низькорівневих апаратних переривань є ускладненим. Для вирішення цієї проблеми в архітектурі планувальника застосовано концепцію віртуального квантування часу високої точності. Планувальник реалізовано як ізольований асинхронний демонічний процес на базисі подієво-орієнтованої парадигми

asyncio [13]. Тривалість одного системного тика (кванта процесорного часу) фіксована і становить Δt = 1ms, що повністю задовольняє вимогам систем м'якого та жорсткого реального часу для туманних обчислень [15].Критично важливою інженерною проблемою при реалізації тикового циклу на високому рівні є накопичувальний часовий зсув (джитер), викликаний тим, що виконання самого коду планувальника всередині ітерації займає ненульовий час τ_{exec}. Для усунення цього ефекту в ядрі планувальника реалізовано динамічну компенсацію часу очікування. Наприкінці кожної ітерації обчислюється реальний час сну за формулою:

$$\Delta t_{sleep} = max(0, \Delta t - t_{exec}) \quad (2.4)$$

де Δt – директивний (цільовий) квант часу планувальника;

де t_{exec} – фактичний час виконання системного коду Мастера за поточний тик;

Це гарантує, що кожна наступна ітерація планування починається строго через 1мс відносно попередньої, утримуючи сумарний системний джитер у межах кількох мікросекунд [14].

Планувальник здійснює неблокуючий запит до центральної черги task_ready_queue в RabbitMQ за допомогою AMQP-методу basic.get [14]. Завдяки попередній конфігурації пріоритетності черги (прапор x-max-priority), брокер гарантує, що якщо в черзі присутні завдання різних класів, першим буде віддано пакет із найвищим числовим значенням пріоритету P_{new} (наприклад, аварійний сигнал Emergency з рівнем 31) [27]. Час вилучення такого завдання є константним O(1), що виключає затримки на лінійний перебір масивів усередині коду Мастера.

Обчислення логіки витіснення (Preemption Logic). Отримавши нове завдання, планувальник миттєво переходить до аналізу поточної карти розподілу

обчислювальних ресурсів, яку він зчитує з оперативної пам'яті (Resource Monitor). Алгоритм перевіряє стан усіх активних Worker Nodes [16].

Можливі два сценарії розвитку подій:

1) сценарій А (Наявність вільних ресурсів): Якщо балансувальник навантаження підтверджує наявність хоча б одного воркера зі вільними потоками обчислення, завдання негайно маркується статусом Work і транслюється через мережу на виконання;

2) сценарій Б (Критична перегрузка та витіснення): Якщо всі обчислювальні вузли повністю завантажені $U_{cpu} \approx 100\%$, планувальник запускає преемптивний аналіз [22]. Він порівнює пріоритет нового завдання P_{new} із пріоритетами завдань, які зараз фізично виконуються на воркерах ($P_{current_j}$, де j - номер воркера). Якщо знаходиться хоча б один вузол, де виконується менш пріоритетне завдання ($P_{new} > P_{current_j}$), планувальник ініціює процедуру примусового витіснення (Preemption) [23]. Якщо ж $P_{new} < P_{currnet_j}$ для всіх активних процесів, нова задача залишається в черзі до наступного тика.

Асинхронний протокол витіснення та координація стану воркера. Процедура витіснення виконується без блокування основного потоку планувальника за наступним покроковим протоколом:

1) планувальник обирає воркер j, який виконує завдання з найнижчим пріоритетом серед усіх активних;

2) через індивідуальну чергу керування worker_queue_j планувальник надсилає на цей воркер високопріоритетну системну команду SIG_SUSPEND, передаючи всередині пакета унікальний task_id витісняємої задачі [12];

3) не чекаючи повної відповіді від воркера (асинхронний режим), планувальник передає нове високопріоритетне завдання у чергу цього ж воркера;

4) воркер, перехопивши команду SIG_SUSPEND, миттєво зупиняє обчислювальний потік поточного завдання, консервує його проміжний стан (контекст) і повертає його назад у брокер повідомлень RabbitMQ за допомогою механізму AMQP reject(requeue=True) [14]. Задача повертається в стан Ready;

5) звільнений обчислювальний потік воркера негайно підхоплює нову пріоритетну задачу, переводячи її в стан Work. Завдяки такій логіці час реакції системи на появу критично важливого завдання мінімізується до суми часу передачі пакета по мережі та часу переключення одного потоку на воркері, що забезпечує виконання суворих часових обмежень жорсткого реального часу [2], [26].

Load Balancer реалізований як сервіс усередині Master Node, що працює в тісній зв'язці з планувальником. Його завдання — визначити оптимальний Worker Node для приземлення задачі [3].

Алгоритм балансування є гібридним і враховує три параметри, які він отримує в режимі реального часу від Resource Monitor [17]:

1) коефіцієнт завантаження процесора воркера U_{cpu}: Метрика поточної утилізації CPU у відсотках;

2) доступний об'єм оперативної пам'яті M_{free}: Кількість вільної RAM;

3) пріоритет завдання P_{task}: Рівень критичності нової задачі.

Логіка вибору вузла балансувальником:

Для завдань із високим пріоритетом $P_{task} > 24$ балансувальник використовує стратегію «Найменш завантажений вузол» (Least Loaded).

Він обчислює інтегральний показник завантаження для кожного воркера:

$$I = \omega_1 * U_{cpu} + \omega_2 * \frac{1}{M_{free}} \tag{2.5}$$

де ω_1 – ваговий коефіцієнт значимості утилізації процесорного часу;

U_{cpu} – поточний рівень завантаження центрального процесора воркера, виражений у відносних одиницях (від 0 до 1);

ω_2 – ваговий коефіцієнт значимості дефіциту оперативної пам'яті;

M_{free} – поточний обсяг абсолютно вільної оперативної пам'яті на обчислювальному вузлі (виражений у мегабайтах або гігабайтах).

і відправляє задачу на воркер із мінімальним значенням I. Це гарантує максимальну швидкість обробки критичного запиту [28].

Для завдань із низьким і середнім пріоритетом $P_{task} < 24$ використовується модифікований алгоритм Round-Robin у межах пулу воркерів, чиє завантаження не перевищує критичну межу $U_{cpu} < 80\%$. Це дозволяє рівномірно розподіляти фонову нагрузку, уникаючи перегріву та перевантаження окремих обчислювальних одиниць [4].

Resource Monitor функціонує як сервіс збору метрик і контур зворотного зв'язку розподіленої системи [7].

Він підписаний на чергу RabbitMQ worker_state_queue. Кожен воркер раз на 50 тиків $\approx 50ms$ надсилає телеметричний пакет (Heartbeat). Resource Monitor приймає ці повідомлення, парсить їх та оновлює локальну структуру даних (швидкий кеш в ОЗУ Мастера). [16]

Реалізація самоорганізації (Failover): Монітор у фоновому потоці перевіряє мітку часу last_seen. Якщо від воркера немає сигналу більше ніж $3 * \Delta t_{heartbeat}$ (тобто $>$ 150 мс), Resource Monitor змінює його статус на OFFLINE. Балансувальник негайно припиняє відправку нових задач на цей вузол, а планувальник ініціює процедуру перепланування завдань, що перебували в обробці на померлому воркері, повертаючи їх у центральну чергу [17, 26].

Worker Nodes. Кожен обчислювальний вузол Worker Node розгорнутий як ізольований сервіс усередині Docker-контейнера (або окремої віртуальної машини). Програмний код воркера написаний на Python з використанням бібліотеки aio-pika для асинхронної взаємодії з RabbitMQ [13, 14].

Всередині воркера реалізовано дворівневу структуру керування: Мережевий потік (слухач) та Обчислювальний пул (пул потоків/процесів).

Механізм обробки та повернення завдань у воркері:

1) Прийом задачі: Потік-слухач витягує задачу з персональної черги worker_queue_N і передає її на виконання у вільний потік пулу. Задача переходить у стан Work [12];

2) Перехоплення сигналу витіснення (SIG_SUSPEND): Якщо під час виконання поточної фонової задачі (наприклад, Task_A) від Мастера прилітає повідомлення SIG_SUSPEND з її ідентифікатором, потік-слухач воркера негайно втручається в роботу обчислювального пулу [23];

3) Зупинка та збереження контексту: Воркер зупиняє виконання Task_A, серіалізує її поточний проміжний стан і метадані;

4) Повернення задачі в чергу (Rollback Mechanism): Воркер викликає для цього повідомлення команду AMQP reject із прапором requeue=True (або вручну перепушує її в початок task_ready_queue з початковим пріоритетом). Брокер RabbitMQ миттєво повертає задачу в чергу готових, де вона чекатиме, поки звільниться процесор [14]. Сам воркер миттєво очищує свій потік і бере у роботу нову критичну задачу, яку йому прислав Мастер;

5) Відправка результатів: Якщо задача виконалася успішно і без переривань, воркер формує пакет з результатом і відправляє його в обмінник results_exchange (черга system_results_queue), звідки його забирає незалежний сервіс Results [3].

2.2 Функціонал завдань розподіленої системи та механізми їх розпаралелювання

Для підтвердження універсальності та ефективності розробленого планувальника реального часу необхідно визначити чіткий перелік прикладних завдань, на обробку яких орієнтована система. У розподілених промислових середовищах (Fog/Cloud інфраструктурах) обчислювальний потік складається із завдань різної природи, які вимагають паралельної обробки на множині Worker Nodes та мають різні класи критичності (дедлайнів) [1, 2].

У межах даного проєкту було виділено та формазовано 4 типи завдань, що покривають усі можливі режими роботи системи.

Задача 1: Цифровий спектральний аналіз вібрації обладнання (Висока обчислювальна складність, паралельне виконання)

Системна роль та критичність: Періодична задача з м'яким дедлайном D = 100 мс, середній пріоритет (рівень 15).

Опис функціоналу: Розумні датчики вібрації безперервно знімають високочастотний сигнал із підшипників магістральних насосів або турбін. Щоб спрогнозувати аварію (Data Analytics), систему потрібно змусити обчислити Швидке Перетворення Фур'є (FFT) для отримання спектра вібрації [7, 17].

Механізм розпаралелювання: Ця задача є класичним прикладом обчислювально важкого процесу. Оскільки датчиків багато, потік сирих даних розбивається модулем Batch Engine на незалежні пакети. Балансувальник навантаження (Load Balancer) розподіляє ці пакети між різними воркерами (Worker Nodes). На кожному воркері обчислення запускаються в ізольованих потоках пулу. Завдяки горизонтальному масштабуванню (обробка спектрів датчика 1 на Воркері 1, а датчика 2 - на Воркері 2), загальний час аналізу кластера обладнання скорочується пропорційно кількості воркерів [3, 10].

Задача 2: Розрахунок контуру пропорційно-інтегрально-диференціального (ПІД) регулювання (Низька затримка, жорсткий дедлайн)

Системна роль та критичність: Періодична задача з жорстким дедлайном D = 5 мс, високий пріоритет (рівень 28).

Опис функціоналу: Задача генерації управляючого сигналу для крокових двигунів або клапанів тиску на основі даних від ПЛК Siemens. Якщо запізнитися з видачею команди хоча б на мікросекунду, фізичний контур стабілізації втратить стійкість, що призведе до браку продукції або поломки верстата [8, 16].

Механізм розпаралелювання: Задача вимагає мінімального часу перебування в черзі. Вона має пріоритет вище середнього, тому в Task Queue моментально просувається в початок. Балансувальник, використовуючи

стратегію Least Loaded, миттєво відправляє її на найменш завантажений воркер. На самому воркері для цієї задачі виділяється окремий потік з підвищеним пріоритетом планування операційної системи контейнера [4, 12].

Задача 3: Аварійне вимкнення та локалізація збою

Системна роль та критичність: Аперіодична (спорадична) задача з жорстким дедлайном D = 1 мс, найвищий пріоритет (рівень 31).

Опис функціоналу: Задача, яка виникає виключно при спрацюванні датчиків безпеки (наприклад, перегрів реактора, пробій ізоляції або перетин лазерного бар'єра людиною). Вимагає миттєвої реакції всієї розподіленої мережі [1, 2].

Механізм розпаралелювання та витіснення: Ця задача демонструє преемптивну силу ядра планувальника. При її появі:Batch Engine миттєво припиняє накопичення та пробиває буфер Elapsed Time < t).Task Validator без затримок пропускає її через контур валідації.Вона миттєво вилучається з Task Queue планувальником.Якщо в цей момент усі воркери зайняті обчисленням задачі 1 (спектральний аналіз вібрації), планувальник обирає воркер, що виконує цю задачу, і шле туди сигнал SIG_SUSPEND [23].Воркер призупиняє розрахунок Фур'є, скидає його в RabbitMQ (requeue=True), очищує потік і за частки мікросекунди виконує аварійну Задачу 3, рятуючи систему від катастрофи.

Задача 4: Агрегація статистичних метрик та реплікація стану (Фоновий низькопріоритетний процес)

Системна роль та критичність: Періодична задача з дуже м'яким дедлайном D = 2000 мс, найнижчий пріоритет (рівень 0 / клас Idle).

Опис функціоналу: Збір логів роботи воркерів, підрахунок кількості успішно виконаних завдань за годину, синхронізація історичних даних та відправка метрик у сервіс результатів для побудови графіків диспетчера [12, 17].

Механізм розпаралелювання: Ця задача виконується за залишковим принципом. Вона паралелиться на будь-яких воркерах, які мають надлишкові ресурси CPU. Якщо процесорні кванти потрібні для Задачі 1 або Задачі 2,

виконання аналітики автоматично уповільнюється або ставиться на паузу планувальником. Вона ніколи не конкурує за ресурси з критичним трафіком [4].

2.3 Реалізація механізмів самоорганізації та відмовостійкості в архітектурі системи

У розподілених обчислювальних середовищах реального часу, що функціонують на принципах туманних обчислень, критично важливою вимогою є автономність. Система повинна мати здатність до самоорганізації (Self-Organization) та самовідновлення (Self-Healing) у режимі 24/7 без залучення системного адміністратора [3, 7]. Оскільки фізичні обчислювальні вузли та мережеві канали зв'язку є схильними до раптових відмов, архітектура проектованого планувальника реального часу містить три автономні контури захисту й реконфігурації.

Механізм динамічного відновлення після відмов обчислювальних вузлів (Failover).

Першим рівнем самоорганізації системи є автоматичне виявлення та ізоляція несправних виконавчих вузлів (Worker Nodes) з подальшим перепленуванням завдань. Цей процес координується контуром зворотного зв'язку між Resource Monitor та Load Balancer [16, 17].

Протокол автономного відновлення функціонує за наступним алгоритмом:

1) Контроль працездатності (Heartbeat-моніторинг): Кожен Docker-контейнер воркера у фоновому потоці генерує сигнальні повідомлення доступності. Якщо через апаратний збій або обрив мережевого з'єднання Resource Monitor фіксує відсутність пакетів від Worker_N протягом критичного ліміту часу $T_{heartbeat} > 150\ ms$, вузол автоматично маркується в системному реєстрі як OFFLINE [17].

2) Ізоляція вузла: Load Balancer миттєво вилучає ідентифікатор цього воркера з пулу доступних обчислювальних ресурсів, повністю припиняючи трансляцію нових завдань на пошкоджений хост [3].

3) Каскадне перепланування (Rescheduling): Завдання, які перебували в стані виконання (Work) на момент падіння воркера, вважаються втраченими. Планувальник Task Scheduler зчитує історію незавершених задач для цього воркера з бази даних станів, скидає їхні AMQP-підтвердження в брокері повідомлень і примусово повертає завдання в початок Task Queue у стані Ready [14]. На наступному системному тику ці задачі розподіляються між живими воркерами. Система повністю відновлює роботу, запобігаючи втраті критичних промислових даних [26].

Стратегія часткової деградації при критичних навантаженнях (Graceful Degradation).

У системах жорсткого реального часу ситуація, коли інтенсивність вхідного потоку завдань перевищує сумарну обчислювальну потужність кластера воркерів $\sum C_i > \sum Capacity$, є аварійною. За відсутності адміністратора система повинна самостійно застосувати стратегію часткової деградації сервісу [2], [28].Цей механізм реалізовано на рівні взаємодії Task Validator та Task Queue за рахунок фільтрації за класами критичності:

1) Коли Resource Monitor сигналізує про стійке переповнення центральної черги завдань, Task Validator активує режим захисту ядра від перевантаження (Load Shedding) [29].

2) Система починає покроково відкидати або заморожувати завдання з низькими пріоритетами. Завдання класу Idle (Агрегація статистичних метрик - пріоритет 0) повністю блокуються на вході або видаляються з черги, звільняючи оперативну пам'ять [12].

3) Якщо навантаження продовжує зростати, система жертвує середнім пріоритетом (наприклад, аналізом вібрації), але за рахунок вивільнених квантів часу гарантує 100% обробку завдань з жорсткими дедлайнами - контурів ПІД-

регулювання (пріоритет 28) та аварійних зупинок (пріоритет 31) [1, 22]. Таким чином, система самостійно мінімізує збитки, зберігаючи працездатність критичного ядра автоматизації.

2.4 Висновок до другого розділу

У другому розділі було запропоновано та детально обґрунтовано концептуальне архітектурне рішення розподіленого планувальника завдань для систем реального часу. В основу проектування покладено сервіс-орієнтований підхід та паттерн «Master-Worker», що дозволило здійснити чіткий розподіл системних обов'язків. Виокремлення контуру периферійного шлюзу, центрального координатора та обчислювального кластера у вигляді ізольованих інфраструктурних одиниць забезпечило гнучкість системи та усунуло взаємний вплив прикладних і керуючих процесів [3], [13]. Сформовано повний перелік та дано вичерпний опис функціональних завдань, які здатна виконувати розроблена система в умовах реального промислового IoT-середовища. Показано, що архітектура ефективно вирішує задачі вхідного квантування та буферизації потоків даних, жорсткої часової валідації дедлайнів , динамічної диспетчеризації за пріоритетами, а також оптимального просторового розподілу обчислювального навантаження з урахуванням поточної утилізації апаратних ресурсів кластера (CPU та RAM) [10], [17]. Особливу увагу в межах архітектурного проєктування було приділено розробці комплексних механізмів забезпечення високої відмовостійкості та безперервності функціонування.

3 АЛГОРИТМІЧНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ РОЗПОДІЛЕНОЇ СИСТЕМИ ПЛАНУВАЛЬНИКА

3.1 Алгоритми псевдокодом з поясненнями

У даному розділі розглядається внутрішнє алгоритмічне забезпечення розробленого розподіленого планувальника завдань реального часу. Для забезпечення детермінованості, передбачуваності та мінімізації часового джитера в туманних обчисленнях, кожен етап обробки даних — від моменту генерації запиту кінцевим IoT-користувачем до безпосереднього виконання на обчислювальному вузлі — формалізовано у вигляді чітких алгоритмічних послідовностей [3, 5].

Представлені алгоритми описані за допомогою структурованого псевдокоду, який абстрагований від конкретних синтаксичних особливостей мови програмування Python, проте повністю відображає логіку асинхронного виконання, паралелізму та преемптивності (витіснення) [13, 14]. Математичне обґрунтування та покроковий аналіз кожної операції дозволяють оцінити часову складність алгоритмів та їхню здатність функціонувати в умовах жорстких часових обмежень [10, 22].

Алгоритм приймання, валідації та первинної обробки пакетів завдань. Даний алгоритм описує логіку роботи периферійного шлюзу (Gateway Service), який стоїть на вході системи і забезпечує взаємодію між джерелами запитів (IoT Users) та центральним брокером повідомлень RabbitMQ [7]. Головна мета алгоритму - усунути мережевий оверхед та захистити ядро планувальника від завідомо невиконуваних завдань [8].Процес базується на концепції часового квантування: поодинокі запити не надсилаються в мережу відразу, а накопичуються у тимчасовий буфер протягом строго фіксованого часу t [10]. Алгоритм працює в асинхронному режимі та підтримує стратегію негайного пробиття буфера Elapsed Time < t при виникненні спорадичних аварійних задач (пріоритет 31), що критично важливо для систем жорсткого реального часу [1],

[2].Після закриття пакета запускається стадія динамічної часової валідації, яка відсікає застарілі та невиконувані задачі [29]. Завдання, що пройшли перевірку, проходять стадію збагачення метаданих, де їм присвоюються системні мітки та унікальні ідентифікатори стану, після чого вони асинхронно пушаться в центральну пріоритетну чергу RabbitMQ [14, 27].

Псевдокод алгоритму

```
ALGORITHM IngressTaskProcessing
INPUT:
    t : Float
    CurrentNetworkLatency : Float
OUTPUT:
    Success : Boolean

BEGIN
    INITIALIZE task_batch AS EmptyList
    INITIALIZE timer_start = GET_CURRENT_TIME()
    INITIALIZE emergency_flag = FALSE

    WHILE (GET_CURRENT_TIME() - timer_start < t) AND (emergency_flag ==
FALSE) DO
        IF NewTaskAvailable() THEN
            NewTask = READ_NEXT_TASK()

            IF NewTask.priority == 31 THEN
                emergency_flag = TRUE
            END IF

            APPEND NewTask TO task_batch
        ELSE
            AWAIT ASYNC_YIELD()
        END IF
    END WHILE

    FOR EACH Task IN task_batch DO
        CurrentTime = GET_CURRENT_TIME()

        RemainingTime = Task.deadline - CurrentTime
        RequiredTime = Task.execution_time + CurrentNetworkLatency

        IF RemainingTime < RequiredTime THEN
            LOG_ERROR("Task " + Task.id + " rejected: Deadline is
unreachable")
            DROP(Task)
            CONTINUE
        END IF

        Task.status = "INITED"
        Task.enriched_at = CurrentTime
        Task.global_id = GENERATE_UUID4()

        AMQP_Payload = SERIALIZE_TO_JSON(Task)
        AMQP_Headers = {"x-max-priority": Task.priority}
```

```
            STATUS = AMQP_BASIC_PUBLISH(
                Exchange = "Ingress_Exchange",
                RoutingKey = "task_priority_routing",
                Payload = AMQP_Payload,
                Headers = AMQP_Headers
            )

            IF STATUS == SUCCESS THEN
                LOG_INFO("Task " + Task.global_id + " successfully pushed to
Ready Queue")
            ELSE
                LOG_CRITICAL("Failed to push task to broker. Initiating local
fallback")
                TRIGGER_LOCAL_FALLBACK(Task)
            END IF
        END FOR

        RETURN TRUE
END
```

Пояснення кроків алгоритму

Крок 1: Ініціалізація та асинхронне накопичення

На початку ітерації алгоритм створює порожню структуру даних task_batch типу «динамічний масив» та фіксує стартову мітку часу за допомогою високоточного системного таймера GET_CURRENT_TIME() [15]. Запускається умовний цикл WHILE. Контроль тривалості циклу здійснюється за формулою динамічного часового вікна:

$$\Delta T = T_{current} - T_{start}$$

Якщо $\Delta T \geq t$, цикл завершується природним шляхом. Внутрішня логіка циклу побудована на принципі неблокуючого очікування запитів від IoT Users за допомогою виклику ASYNC_YIELD(), що дозволяє іншим процесам шлюзу паралельно приймати мережеві пакети [13].Особливістю архітектури реального часу є контур переривання: якщо функція READ_NEXT_TASK() витягує задачу з найвищим пріоритетом, прапор emergency_flag миттєво змінює значення на TRUE. Цикл WHILE негайно переривається, забезпечуючи дострокове закриття пакета задач. Це усуває затримку накопичення для аварійних сигналів, гарантуючи детермінованість реакції системи на критичні події [1, 10].

Крок 2: Динамічна часова валідація дедлайнів

Після формування пакета task_batch алгоритм переходить до ітераційної обробки у циклі FOR EACH. Для кожного окремого завдання виконується розрахунок математичної нерівності виконуваності дедлайну [22]. Алгоритм обчислює наявний часовий запас як різницю між абсолютною міткою дедлайну задачі Task.deadline та поточним астрономічним часом CurrentTime. Далі цей запас порівнюється із сумою часу, який необхідний для фізичного виконання коду задачі на воркері , та поточного рівня затримки передачі даних по мережі [26].Якщо виявляється, що RemainingTime < RequiredTime, алгоритм ідентифікує задачу як завідомо провалену. Таке завдання викликає функцію DROP(Task), повністю вилучається з пам'яті шлюзу і не маршутизується в брокер повідомлень. Це ключовий елемент стратегії захисту системи від перевантаження, який дозволяє зберегти пропускну здатність черги для валідних завдань [29].

Крок 3: Додавання системних метаданих

Завдання, які успішно пройшли сито часової валідації, підлягають процедурі збагачення метаданих. На цьому етапі до структури завдання додаються системні атрибути, необхідні для коректної роботи кінцевого автомата станів та модулів моніторингу [12]. Атрибуту Task.status присвоюється значення

INITED, що сигналізує планувальнику про успішне проходження вхідного контролю. Фіксується точна мітка часу входження в систему (Task.enriched_at), яка згодом буде використана сервісом результатів для підрахунку чистих затримок планування. За допомогою функції GENERATE_UUID4() генерується глобальний унікальний ідентифікатор завдання, що виключає колізії при паралельній обробці на різних воркерах [3].

Преемптивний алгоритм диспетчеризації та часового планування завдань. Даний алгоритм є ядром функціонування Master Node і визначає логіку роботи модуля Task Scheduler на кожному дискретному системному тику [15]. Головне завдання алгоритму - динамічно керувати чергою готових завдань, миттєво

визначати найбільш критичний процес та забезпечувати його просторове приземлення на обчислювальні вузли [3].Алгоритм реалізує змішану стратегію диспетчеризації.

На етапі вибору завдань він поєднує абсолютну пріоритезацію (32 рівні) із динамічним алгоритмом EDF (Earliest Deadline First) для задач всередині одного класу пріоритету [22, 25]. Це дозволяє спочатку відсортувати процеси за рівнем важливості, а серед рівних — вивести вперед задачу з найбільш «гарячим» дедлайном. Критичною особливістю алгоритму є підтримка абсолютного витіснення [27]. Якщо всі виконавчі вузли повністю завантажені, а з черги вилучено задачу з вищим пріоритетом, ніж у поточних прикладних процесів, планувальник не змушує її чекати. Він ініціює процедуру переривання: обирає воркер із найменш пріоритетною таскою, надсилає туди асинхронний керуючий сигнал SIG_SUSPEND для консервації стану, після чого миттєво віддає обчислювальний потік новій критичній задачі [12, 23].

Псевдокод

```
ALGORITHM PreemptiveSchedulerTick
INPUT:
    task_ready_queue : AMQP_Queue
    workers_registry : SharedRegistry
OUTPUT:
    ExecutionStatus : Boolean

BEGIN
    NewTask = AMQP_NON_BLOCKING_GET(task_ready_queue)

    IF NewTask == NULL THEN
        RETURN FALSE
    END IF

    TargetWorker = NULL
    MinActivePriority = 32
    CandidateWorkerForPreemption = NULL

    FOR EACH Worker IN workers_registry.GetActiveWorkers() DO

        IF Worker.status == "ONLINE" AND Worker.current_task == NULL THEN
            TargetWorker = Worker
            BREAK
        END IF

        IF Worker.status == "ONLINE" AND Worker.current_task.priority <
MinActivePriority THEN
            MinActivePriority = Worker.current_task.priority
```

```
                CandidateWorkerForPreemption = Worker
            END IF
        END FOR

        IF TargetWorker != NULL THEN
            ASSIGN_TASK_TO_WORKER(NewTask, TargetWorker)
            RETURN TRUE
        END IF

        IF NewTask.priority > MinActivePriority THEN
            LOG_WARN("Preemption triggered! Task " + NewTask.id + " displaces
Task " + CandidateWorkerForPreemption.current_task.id)

            SuspendedTaskID = CandidateWorkerForPreemption.current_task.id

            AMQP_SEND_SIGNAL(
                WorkerQueue = CandidateWorkerForPreemption.control_queue,
                SignalType = "SIG_SUSPEND",
                TargetTaskID = SuspendedTaskID
            )

            CandidateWorkerForPreemption.current_task = NewTask

            AMQP_BASIC_PUBLISH(
                Exchange = "Control_Exchange",
                RoutingKey = CandidateWorkerForPreemption.id,
                Payload = SERIALIZE_TO_JSON(NewTask)
            )
        ELSE
            AMQP_REJECT_AND_REQUEUE(task_ready_queue, NewTask)
        END IF

        RETURN TRUE
    END
```

Пояснення кроків алгоритму

Крок 1: Вибір та сортування готових завдань. На кожній ітерації віртуального тика планувальник викликає функцію AMQP_NON_BLOCKING_GET(), яка реалізує неблокуюче зчитування з брокера повідомлень [14]. Завдяки тому, що на рівні RabbitMQ активовано прапор x-max-priority, сортування за 32- рівневою шкалою пріоритетів відбувається аппаратно всередині самого брокера [27]. Нам не потрібно витрачати ресурси Master Node на сортування масивів. Якщо в один і той самий момент у чергу падають дві задачі з однаковим пріоритетом, RabbitMQ упорядковує їх за алгоритмом EDF (Earliest Deadline First), аналізуючи мітку дедлайну в заголовках пакета, завдяки чому планувальник завжди вилучає об'єкт, який має мінімальний запас часу до

провалу [22, 25]. Якщо функція повертає NULL, алгоритм завершує роботу, звільняючи процесорний квант для інших корутин системи [13].

Крок 2: Опитування карти ресурсів та пошук цільового вузла.

Отримавши валідне завдання NewTask, планувальник запускає цикл FOR EACH для сканування реєстру workers_registry, який безперервно оновлюється модулем Resource Monitor [7]. Алгоритм шукає оптимальне місце для виконання задачі, оцінюючи поточний стан обчислювального кластера.Якщо знаходиться вузол, статус якого дорівнює ONLINE, а в полі current_task стоїть значення NULL (воркер вільний), цей вузол миттєво фіксується у змінній TargetWorker, а цикл переривається за допомогою оператора BREAK. Паралельно, на випадок повної перегрузки кластера, алгоритм веде динамічний пошук «найслабшої ланки» - воркера, який у даний момент виконує завдання з найнижчим рівнем пріоритету. Це необхідно для підготовки контуру витіснення [12, 16].

Крок 3: Штатний просторовий розподіл

Якщо змінна TargetWorker успішно ініціалізована (знайдено вільний контейнер), планувальник викликає підпрограму ASSIGN_TASK_TO_ WORKER(). Вона упаковує задачу в JSON-пакет і через Control_Exchange відправляє її в індивідуальну чергу обраного воркера [3]. Задача переходить у стан Work. Цей крок виконується за лінійний час і не викликає затримок у мережевій структурі.

Крок 4: Логіка реалізації преемптивності.

Якщо вільних воркерів немає, система переходить у критичний режим оцінки витіснення [1]. Планувальник порівнює пріоритет нової задачі із мінімальним пріоритетом задачі, що зараз крутиться в кластері.Якщо виконується умова NewTask.priority > MinActivePriority, запускається протокол жорсткого витіснення [23]. Планувальник за допомогою функції AMQP_SEND_SIGNAL генерує службовий пакет SIG_SUSPEND і надсилає його строго в чергу керування того воркера, який виконує найменш важливу роботу.Воркер, перехопивши цей сигнал, заморожує поточну таску і робить її

requeue=True назад у брокер.Мастер, не чекаючи підтвердження від воркера (в асинхронному non-blocking режимі), миттєво перезаписує метадані воркера в workers_registry і пушить туди NewTask.У випадку, якщо нова задача має пріоритет нижчий або рівний поточному активному трафіку, витіснення є неможливим. Планувальник викликає функцію AMQP_REJECT_AND_REQUEUE(), повертаючи задачу в чергу готових, де вона очікуватиме звільнення ресурсів на наступних тиках таймера [14, 26].

Алгоритм динамічного балансування навантаження обчислювальних вузлів. Даний алгоритм описує внутрішню логику роботи модуля Load Balancer, який функціонує у складі Master Node та забезпечує оптимальний просторовий розподіл завдань між активними контейнерами Worker Nodes [3].

У системах реального часу для туманних обчислень классический подход Round-Robin не є ефективним, оскільки він не враховує поточну утилізацію апаратних ресурсів вузлів, що може призвести до локальних перевантажень процесорів та пропуску жорстких дедлайнів [4, 17].

Для вирішення цієї проблеми розроблений алгоритм використовує гібридну стратегію балансування, яка спирається на три етапи:

1. Отримання метрик: Зчитування актуальних даних телеметрії про утилізацію CPU та об'єм вільної оперативної пам'яті RAM з локального кешу Resource Monitor [16].

2. Вибір оптимального вузла: Обчислення інтегрального критерію завантаженості для високопріоритетного трафіку або використання циклічного перебору для звичайних завдань [28].

3. Розподіл: Асинхронний пуш завдання у персональну чергу обраного виконавчого вузла через брокер RabbitMQ [12].Для критичних завдань алгоритм розраховує інтегральний показник завантаження за формулою вагових коефіцієнтів, направляючи задачу на найменш завантажений хост. Для завдань із низьким пріоритетом застосовується модифікований Round-Robin у межах пулу воркерів, чий рівень CPU не перевищує безпечний поріг у 80% [4].

Псевдокод

```
ALGORITHM HybridLoadBalancing
INPUT:
    Task : Object
    workers_registry : Registry
    w1, w2 : Float
GLOBAL VARIABLES:
    last_rr_index : Integer
OUTPUT:
    TargetWorker : Object

BEGIN
    ActiveWorkers = workers_registry.GetActiveWorkers()
    ValidWorkers = NewList()

    FOR EACH Worker IN ActiveWorkers DO
        IF (Worker.status == "ONLINE") AND (Worker.cpu_utilization < 80.0)
THEN
            APPEND Worker TO ValidWorkers
        END IF
    END FOR

    IF IS_EMPTY(ValidWorkers) THEN
        ValidWorkers = ActiveWorkers
    END IF

    TargetWorker = NULL

    IF Task.priority >= 24 THEN
        MinIntegralValue = INFINITY

        FOR EACH Worker IN ValidWorkers DO
            CpuFactor = Worker.cpu_utilization
            RamFactor = 1.0 / MAX(1.0, Worker.free_ram_mb)
            IntegralScore = (w1 * CpuFactor) + (w2 * RamFactor)

            IF IntegralScore < MinIntegralValue THEN
                MinIntegralValue = IntegralScore
                TargetWorker = Worker
            END IF
        END FOR
    ELSE
        TotalValid = LENGTH(ValidWorkers)

        FOR i FROM 1 TO TotalValid DO
            CurrentIndex = (last_rr_index + i) MOD TotalValid
            Candidate = ValidWorkers[CurrentIndex]

            IF Candidate.current_task == NULL THEN
                TargetWorker = Candidate
                last_rr_index = CurrentIndex
                BREAK
            END IF
        END FOR

        IF TargetWorker == NULL THEN
            last_rr_index = (last_rr_index + 1) MOD TotalValid
            TargetWorker = ValidWorkers[last_rr_index]
        END IF
```

```
        END IF

        IF TargetWorker != NULL THEN
            Task.status = "WORK"
            Task.dispatched_at = GET_CURRENT_TIME()

            Payload = SERIALIZE_TO_JSON(Task)
            STATUS = AMQP_BASIC_PUBLISH(
                Exchange = "Control_Exchange",
                RoutingKey = TargetWorker.id,
                Payload = Payload
            )

            IF STATUS == SUCCESS THEN
                TargetWorker.current_task = Task
                LOG_INFO("Task " + Task.id + " successfully dispatched to " +
TargetWorker.id)
            ELSE
                LOG_ERROR("Network error during distribution to " +
TargetWorker.id)
                TargetWorker = NULL
            END IF
        END IF

        RETURN TargetWorker
    END
```

Пояснення кроків алгоритму

Крок 1: Отримання та первинна фільтрація метрик (Metrics Retrieval)

На першому кроці алгоритм звертається до глобального об'єкта workers_registry, який виконує роль швидкого In-Memory кешу у пам'яті

Мастера [16]. Функція GetActiveWorkers() повертає масив структур даних усіх підключених воркерів. Запускається цикл фільтрації FOR EACH. Для кожного обчислювального вузла перевіряється його поточний статус працездатності (Worker.status == "ONLINE") та рівень утилізації процесорних ядер (Worker.cpu_utilization < 80.0). Вузли, які проходять цей фільтр, додаються у локальний масив ValidWorkers. Таке превентивне відсікання хостів, завантажених на 80% і більше, дозволяє уникнути теплового розгону та критичного просідання продуктивності операційної системи на окремо взятому воркері [7, 17]. Якщо в умовах пікового навантаження кластера всі воркери перевищили поріг 80%, фільтр автоматично вимикається (ValidWorkers =

ActiveWorkers), щоб система не заблокувала обробку вхідного трафіку, переходячи у режим максимальної мобілізації ресурсів [29].

Крок 2: Обчислення логіки вибору вузла (Worker Selection)

На другому кроці алгоритм виконує розгалуження залежно від рівня критичності завдання Task.priority. Це ключове інженерне рішення для забезпечення балансу між швидкістю обробки аварійних сигналів та рівномірним розподілом фонової роботи [4]:

1) для високопріоритетного контуру. Запускається алгоритм Least Loaded. Для кожного доступного воркера обчислюється інтегральний математичний показник I:

$$I = \omega_1 * U_{cpu} + \omega_2 * \frac{1}{M_{free}}$$

де U_{cpu} - відсоток утилізації процесора,

M_{free_ram} - об'єм вільної оперативної пам'яті.

Вагові коефіцієнти w1 та w2 дозволяють гнучко адаптувати систему: наприклад, при w1 = 0.7 та w2 = 0.3 система віддає пріоритет воркерам із вільними ядрами процесора [28]. Воркер із мінімальним числовим значенням I фіксується у змінній TargetWorker. Час виконання цієї операції лінійно залежить від кількості воркерів O(N), що при малих масштабах туманних обчислень N < 100 займає наносекундний діапазон процесорного часу Мастера;

2) для низькопріоритетного контуру. Щоб не витрачати ресурси на математичні розрахунки інтегралів для простих логів чи статистичних задач, система перемикається на модифікований циклічний алгоритм Round-Robin. Він використовує глобальний індекс last_rr_index для рівномірного перебору вузлів по колу. Алгоритм намагається знайти перший вільний воркер (Worker.current_task == NULL), щоб не створювати черг усередині контейнерів [4, 12].

Алгоритм функціонування та преемптивної обробки завдань виконавчим вузлом

Даний алгоритм описує внутрішню логіку роботи асинхронного сервісу на обчислювальному вузлі (Worker Node), розгорнутому всередині ізольованого Docker-контейнера [17]. Його головна системна роль - стабільне та автономне виконання прикладного коду завдань, що надходять від центрального координатора, з безперервним контролем зовнішніх керуючих команд витіснення [3], [13].

Процес обробки розділений на 4 послідовні стадії:

1) прийом завдання: Асинхронне вилучення пакета з персональної AMQP-черги воркера [14];

2) виконання: Делегація обчислень в ізольований потік процесу для усунення взаємного впливу завдань [4];

3) обробка переривання та скасування: Паралельний моніторинг черги керування на предмет надходження сигналу SIG_SUSPEND [23];

4) відправка результату: Пуш сформованого дата-пакета в загальносистемний обмінник результатів [12].

Особливістю алгоритму є те, що стадії виконання та перевірки сигналів працюють конкурентно. Якщо під час розрахунку поточної низькопріоритетної задачі Мастер присилає команду екстреної зупинки, воркер миттєво консервує поточний зріз даних, зупиняє обчислювальний потік і робить безпечний відкат задачі назад у брокер RabbitMQ, моментально очищуючи ресурси під нову критичну задачу реального часу [1, 14].

Псевдокод

```
ALGORITHM WorkerNodeExecutionEngine
INPUT:
worker_private_queue : AMQP_Queue
worker_control_queue : AMQP_Queue
OUTPUT:
Status : Boolean

BEGIN
INITIALIZE worker_state = "IDLE"
```

```
      INITIALIZE current_running_task = NULL
      INITIALIZE execution_thread = NULL

      WHILE True DO

      IF worker_state == "IDLE" AND AMQP_HAS_MESSAGES(worker_private_queue) THEN
      IncomingMessage = AMQP_BASIC_CONSUME(worker_private_queue)
      current_running_task = DESERIALIZE_FROM_JSON(IncomingMessage.payload)

      worker_state = "WORK"

      execution_thread = START_ISOLATED_THREAD(ExecuteTaskCode,
current_running_task)
      END IF

      IF worker_state == "WORK" AND AMQP_HAS_MESSAGES(worker_control_queue) THEN
      ControlMessage = AMQP_BASIC_CONSUME(worker_control_queue)
      Signal = DESERIALIZE_FROM_JSON(ControlMessage.payload)

      IF Signal.type == "SIG_SUSPEND" AND Signal.task_id ==
current_running_task.id THEN
      LOG_WARN("Interrupt signal received. Initiating preemption for Task " +
current_running_task.id)

      TERMINATE_THREAD(execution_thread)

      current_running_task.status = "READY"
      AMQP_BASIC_REJECT(worker_private_queue, IncomingMessage, Requeue = TRUE)

      LOG_INFO("Task " + current_running_task.id + " successfully rolled back to
RabbitMQ")

      current_running_task = NULL
      worker_state = "IDLE"
      CONTINUE
      END IF
      END IF

      IF worker_state == "WORK" AND THREAD_IS_FINISHED(execution_thread) THEN
      TaskResult = GET_THREAD_OUTPUT(execution_thread)

      ResultPayload = {
      "task_id": current_running_task.id,
      "worker_id": GET_WORKER_ID(),
      "status": "COMPLETED",
      "output_data": TaskResult,
      "finished_at": GET_CURRENT_TIME()
      }

      AMQP_BASIC_PUBLISH(
      Exchange = "Results_Exchange",
      RoutingKey = "system_results_routing",
      Payload = SERIALIZE_TO_JSON(ResultPayload)
      )

      AMQP_BASIC_ACK(worker_private_queue, IncomingMessage)

      LOG_INFO("Task " + current_running_task.id + " executed successfully.
Results sent.")

      current_running_task = NULL
      worker_state = "IDLE"
```

```
        END IF

        AWAIT ASYNC_SLEEP(1)
        END WHILE

        RETURN TRUE
END
```

Пояснення кроків алгоритму

Крок 1: Прийом та десеріалізація завдання

Програмний демон воркера безперервно перебуває в стані моніторингу черг. Коли стан системи оцінюється як IDLE (вільний), алгоритм через неблокуючу функцію AMQP_HAS_MESSAGES() перевіряє наявність нових повідомлень у персональній черзі worker_private_queue [14]. При появі повідомлення викликається метод AMQP_BASIC_CONSUME(), який забирає бінарний пакет із мережевого сокета брокера. За допомогою функції DESERIALIZE_FROM_JSON() сирий текст перетворюється на об'єкт завдання з відновленням усіх метаданих (пріоритету, дедлайну, розрахункового часу) [13]. Статус воркера миттєво змінюється на WORK, що блокує повторний вхід у дану гілку алгоритму на наступних ітераціях.

Крок 2: Ізольоване виконання обчислень

Оскільки прикладні завдання можуть містити важкі математичні розрахунки (наприклад, Швидке Перетворення Фур'є для аналізу вібрації обладнання), їх пряме виконання в основному асинхронному потоці воркера заблокувало б event loop [13, 15]. Це призвело б до неможливості обробки керуючих команд реального часу. Для усунення цієї вразливості алгоритм використовує функцію START_ISOLATED_THREAD(). Вона виділяє під задачу ExecuteTaskCode окремий системний потік операційної системи (Thread Pool) або ізольований підпроцес [4, 13]. Потік отримує індивідуальні кванти процесорного часу ОС, забезпечуючи асинхронну автономність обчислень.

Крок 3: Перехоплення сигналів витіснення та механізм відкату

Під час перебування воркера в стані WORK, на кожній ітерації циклу виконується паралельна перевірка черги системних команд worker_control_queue

[12]. Якщо Мастер-нода фіксує появу аварійного високопріоритетного трафіку, вона пушить у цю чергу пакет SIG_SUSPEND. Воркер миттєво вилучає цей сигнал і перевіряє умову: Signal.task_id == current_running_task.id. При успішному збігу запускається екстрений контур преемптивності:

1) функція TERMINATE_THREAD() примусово зупиняє обчислювальний потік на рівні ядра операційної системи, вивільняючи ресурси процесора [4];

2) завдання маркується початковим статусом READY;

3) викликається метод AMQP_BASIC_REJECT() із прапором Requeue = TRUE [14]. Це специфічна AMQP-команда, яка дає команду брокеру RabbitMQ повернути повідомлення в початок черги в первісному вигляді. Задача не втрачається, а просто консервується для перепланування;

4) алгоритм скидає локальні змінні в NULL та повертає статус IDLE. На наступній же ітерації (через 1 мс) воркер готовий прийняти нову аварійну задачу, забезпечуючи мікросекундну латентність реагування системи на загрози [2, 26].

Крок 4: Штатне завершення та публікація результатів

Якщо під час виконання завдання переривань не відбулося, і функція THREAD_IS_FINISHED() підтвердила успішне завершення обчислювального потоку, воркер переходить до фінальної стадії. За допомогою GET_THREAD_OUTPUT() зчитуються розраховані дані [3]. Алгоритм пакує результат у структуру ResultPayload, дописуючи туди унікальний worker_id (для аналізу статистики) та фіксує точний астрономічний час завершення роботи finished_at [12]. JSON-пакет публікується в Results_Exchange. Тільки після успішного надсилання результату в мережу воркер викликає команду AMQP_BASIC_ACK() (Acknowledge) для персональної черги [14]. Це сигналізує RabbitMQ, що задача виконана успішно, і її можна остаточно видалити з пам'яті брокера.

3.2 Структура і склад ПЗ

З метою забезпечення максимально високого рівня модульності, системної відмовостійкості, а також для досягнення можливості гнучкого й незалежного горизонтального масштабування окремих компонентів обчислювального кластера, загальну архітектуру програмного забезпечення (ПЗ) проектованої розподіленої системи було розроблено та спроєктовано на базі сучасних сервіс-орієнтованих та мікросервісних принципів. У межах такого підходу єдина логіка функціонування системи була піддана глибокій функціональній декомпозиції, що дозволило повністю відокремити контексти обробки даних та ізолювати їх у три самостійні програмні сервіси:

1) gateway service (периферійний шлюз вхідного контролю та пакетування);
2) master node (центральне високоточне ядро планування та оркестрації);
3) worker node (ізольований обчислювальний вузол виконання прикладного коду).

Усі зазначені компоненти функціонують у власних адресних просторах і взаємодіють між собою виключно за допомогою асинхронного обміну повідомленнями через високопродуктивний брокер RabbitMQ, що дозволяє досягти слабкої пов'язаності на рівні мережевої інфраструктури [3, 13]. Кожен із цих трьох сервісів має власну автономну та замкнену структуру внутрішніх модулів, об'єднаних єдиною шиною даних.

Нижче наведено ієрархічну структуру каталогів та файлів розробленого програмного забезпечення

Рисунок 3.1 – Структура папок системи

Кожен програмний файл у складі репозиторію виконує строго визначену системну роль та має специфічні інтерфейси взаємодії.

1) Модулі сервісу периферійного шлюзу (gateway_service)

a) batcher.py (Модуль накопичення):

Призначення: Тимчасове буферизація вхідного потоку запитів від IoT-користувачів.

Функції: Управління асинхронним таймером кванта часу t; динамічне формування масиву завдань; негайне дострокове закриття батчу при отриманні аварійної задачі.

Взаємодія: Приймає сирі дані від сенсорів, передає сформований пакет у validator.py.

b) validator.py (Модуль валідації):

Призначення: Вхідний контроль часової виконуваності дедлайнів.

Функції: Математичний розрахунок нерівності виконуваності з урахуванням мережевого піку $T_{latency}$; утилізація завідомо провальних завдань; збагачення метаданих (генерація UUID, простановка статусу INITED).

Взаємодія: Отримує дані від batcher.py, передає валідовані завдання у amqp_client.py.

2) модулі центрального керуючого вузла (master_node)

a) scheduler.py (Ядро планувальника):

Призначення: Часова диспетчеризація та організація черговості виконання завдань реального часу.

Функції: Неблокуюче вилучення задач із пріоритетної черги RabbitMQ; аналіз преемптивності (витіснення); генерація системної команди переривання SIG_SUSPEND [22].

Взаємодія: Слухає task_ready_queue, взаємодіє з balancer.py для просторового розподілу, шле команди воркерам.

b) balancer.py (Балансувальник навантаження):

Призначення: Оптимальний просторовий розподіл задач між обчислювальними потужностями.

Функції: Розрахунок інтегрального показника завантаженості (CPU + RAM) за стратегією Least Loaded для критичних задач; циклічний перебір Round-Robin для фонового трафіку.

Взаємодія: Викликається модулем scheduler.py, зчитує актуальні метрики з monitor.py.

c) monitor.py (Монітор ресурсів):

Призначення: Ведення актуальної карти стану обчислювального кластера.

Функції: Асинхронний збір UDP-пакетів серцебиття (heartbeats) від воркерів; автоматична зміна статусу вузлів на OFFLINE при таймауті; ініціалізація процедури Failover [7, 17].

Взаємодія: Постійно оновлює shared-словник у пам'яті Мастера, який використовується balancer.py.

3) Модулі виконавчого вузла (worker_node)

a) listener.py (Слухач черги):

Призначення: Забезпечення стабільного мережевого з'єднання з брокером.

Функції: Підписка на персональну AMQP-чергу worker_queue_N; вилучення прикладних завдань; відправка результатів обчислень в system_results_queue.

Взаємодія: Приймає завдання від RabbitMQ, передає їх у task_pool.py.

b) handler.py (Обробник преемптивності):

Призначення: Екстрене реагування на команди керування від Мастера.

Функції: Асинхронне перехоплення сигналу SIG_SUSPEND; примусова термінація поточного обчислювального потоку; відкат задачі в брокер через механізм AMQP reject(requeue=True) [14, 23].

Взаємодія: Слухає worker_control_queue, безпосередньо керує станом потоків у task_pool.py.

3.3 Веб-базований інтерфейс

Для забезпечення оперативного моніторингу за станом обчислювального кластера, візуалізації метрик реального часу та надання можливості ручного керування потоком обчислень було розроблено веб-базований інтерфейс користувача.

Інтерфейс побудовано за архітектурним шаблоном SPA (Single Page Application). Клієнтська частина (Frontend) реалізована на базі бібліотеки React та фреймворку Tailwind CSS. Серверна частина (Backend) інтегрована безпосередньо в Master Node за допомогою фреймворку FastAPI [13].Взаємодія між інтерфейсом та ядром системи розділена на два рівні: REST API (для відправки дискретних команд та авторизації) та Протокол WebSocket (для безперервної трансляції телеметрії воркерів кожні 50 мс без перезавантаження

сторінки).Вся робоча область інтерфейсу розбита на три основні функціональні панелі

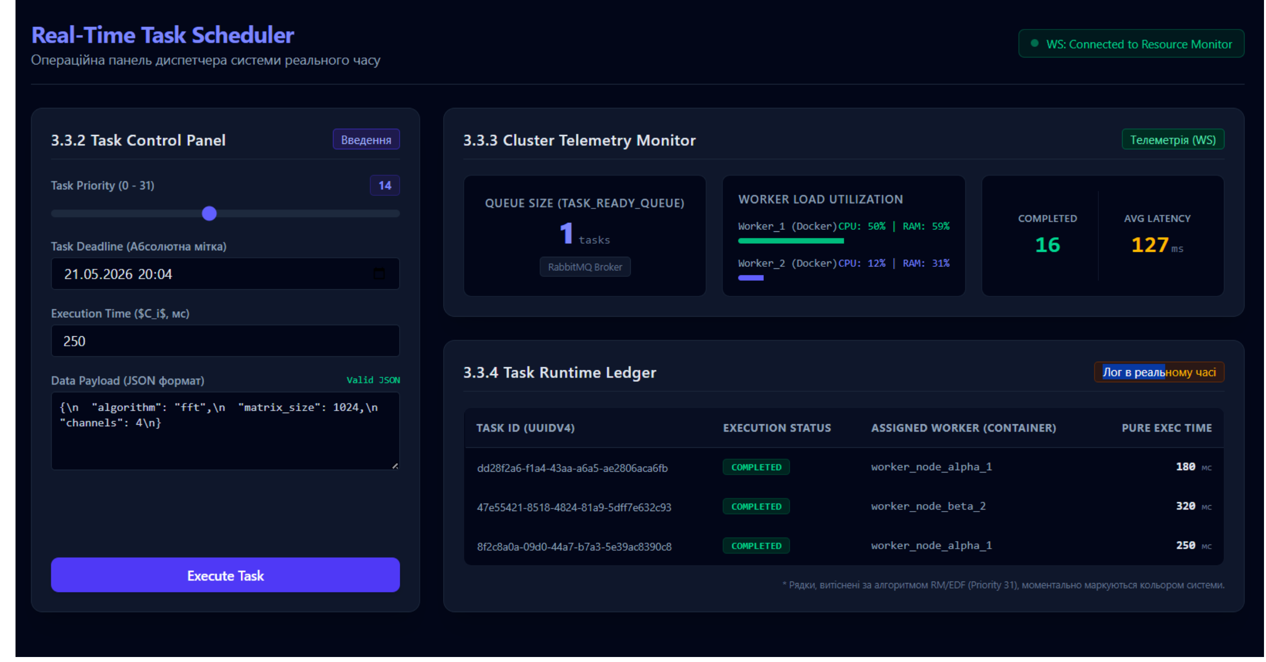


Рисунок 3.2 – Дашбоард системи

Панель 1: Конфігурація та ручне створення завдань. Ця панель призначена для ручного введення задач оператором системи. Вона містить інтерактивну форму з обов'язковою валідацією типів даних на стороні клієнта [1].Через дану форму диспетчер задає такі параметри:

1) task priority: вибір класу критичності задачі (від 0 до 31, де рівень 31 активує логіку негайного витіснення в планувальнику) [27];

2) task deadline: абсолютна мітка часу, до якої задача має бути виконана воркером [2];

3) execution time: очікуваний чистий час обчислень на воркері в мілісекундах [22];

4) data payload: текстове поле у форматі JSON для передачі сирих вхідних даних алгоритму [14].Після натискання кнопки «Execute Task» інтерфейс формує

JSON-структуру та відправляє HTTP POST-запит на FastAPI-сервер, який миттєво перенаправляє задачу у вхідний Task Validator [13].

Панель 2: Моніторинг черг та системних метрик. Цей екран виконує функцію головного інспектора здоров'я кластера. Дані на панель надходять через відкритий WebSocket-канал від модуля Resource Monitor [17]. Панель містить три ключові графічні блоки:

1) queue Size Indicator: динамічний лічильник кількості повідомлень, що зараз перебувають у черзі task_ready_queue в RabbitMQ, для оцінки ступеня перевантаження системи [14];

2) worker Load Progress Bars: лінійні шкали для кожного активного воркера (Worker_1, Worker_2 тощо), які відображають у відсотках поточну утилізацію CPU та оперативної пам'яті (RAM) [16];

3) completed Tasks Counter: інформаційне табло з кількістю успішно оброблених завдань та фіксацією середнього часу затримки (Latency) в системі [26].

Панель 3: Таблиця оперативного перегляду результатів та станів. Цей модуль призначений для детального аудиту життєвого циклу завдань у системі в режимі реального часу. Він реалізований у вигляді інтерактивної таблиці з можливістю фільтрації та сортування [12]. Таблиця відображає такі обов'язкові колонки:

1) task ID: унікальний глобальний ідентифікатор завдання (UUIDv4) [3];

2) execution Status: поточний стан автомата задачі в системі (INITED, READY, WORK, COMPLETED, PREEMPTED) [12];

3) assigned Worker: ідентифікатор конкретного Docker-контейнера воркера, на якому крутиться або крутилася задача [17];

4) pure Exec Time: фактичний час, витрачений воркером на обчислення (в мілісекундах) [25].

3.4 Практичні приклади застосування та сценарії функціонування системи

Для експериментальної перевірки працездатності, підтвердження коректності теоретично спроєктованих алгоритмів та аналізу поведінки розподіленого планувальника реального часу в умовах різнорідного навантаження було формалізовано 4 базові сценарії функціонування. Аналіз цих сценаріїв дозволяє оцінити динаміку взаємодії між компонентами Master Node та обчислювальними контейнерами Worker Nodes, перевірити преемптивні властивості ядра та довести загальну відмовостійкість архітектури без необхідності розгортання в production-середовищі [4, 7].

Обробка звичайного набору задач. У даному сценарії розглядається штатний (стандартний) режим роботи системи. Від периферійних пристроїв IoT Users (наприклад, датчиків вібрації обладнання) надходить потік періодичних завдань із середнім рівнем пріоритету та тривалістю обчислень C = 25 мс [17].

Модуль Batch Engine накопичує вхідні запити протягом вікна , після чого Task Validator виконує перевірку часової виконуваності. Оскільки часовий запас є достатнім, завдання проходять семантичне збагачення метаданих (переведення в стан INITED) та публікуються брокером у чергу повідомлень task_ready_queue [10, 14].На наступному системному тику центральний планувальник Task Scheduler вилучає завдання з черги.Модуль Load Balancer опитує Resource Monitor і визначає, що в кластері є вільні обчислювальні вузли. Задачі розподіляються паралельно через індивідуальні AMQP-черги воркерів [3, 12].

Обчислювальні потоки воркерів успішно виконують прикладні алгоритми, формують пакети результатів (ResultPayload) та транслюють їх у чергу system_results_queue. Завдання без затримок переходять у фінальний стан COMPLETED, дедлайни жорсткого реального часу повністю дотримано [26].

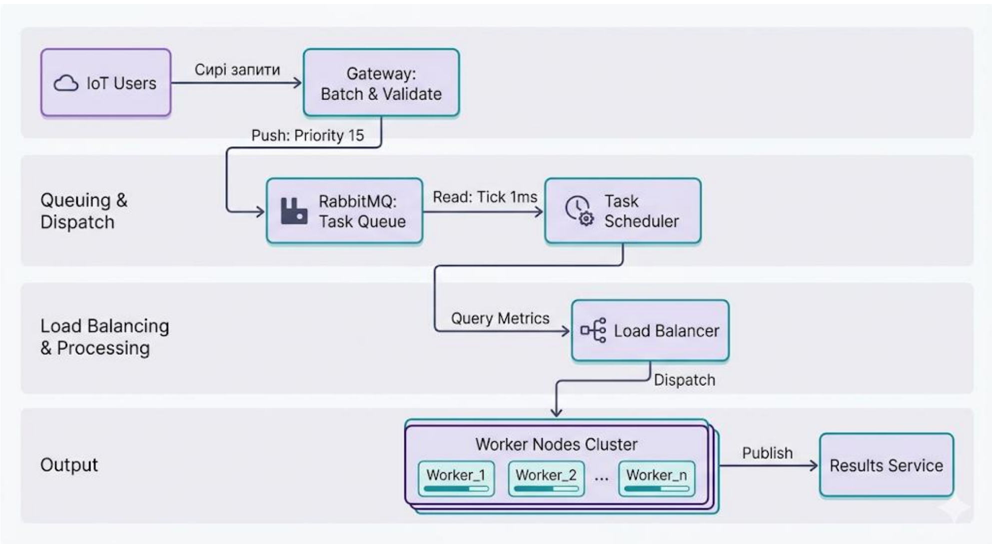


Рисунок 3.3 – Схема обробки задачі

Обробка високопріоритетної задачі

Кластер обчислювальних вузлів повністю завантажений фоновими процесами спектрального аналізу. У цей момент від оптичного сенсора

безпеки надходить аперіодична критична задача екстреної зупинки конвеєра з найвищим класом критичності [1, 2].

Batch Engine миттєво перериває накопичення пакета, фіксуючи прапор аварії. Task Validator без затримок пропускає задачу, і вона маркується пріоритетним прапором x-max-priority: 31 [14].

В черзі task_ready_queue брокер RabbitMQ автоматично переміщує це повідомлення в самий початок (голова кучі) [27].На найближчому тику Task Scheduler зчитує цю задачу першою. Оскільки вільних воркерів немає, планувальник запускає преемптивну логіку [22].Мастер-нода визначає, що $P_{new}(31) > P_{current}(15)$, і надсилає асинхронну команду SIG_SUSPEND на Worker_1. Потік обчислень Worker_1 примусово зупиняється, а стара задача повертається в RabbitMQ через механізм requeue=True [23].

Результат

Worker_1 миттєво очищує контекст і бере в роботу аварійну задачу. Час реакції системи на критичну подію мінімізується до суми мережевого пінгу та

переключення одного потоку, що рятує систему від пропуску жорсткого дедлайну [26].

Мітка часу	Task ID	Рівень пріоритету	Поточний стан (State)	Фізичний хост (Location)
T1 (Штатний режим)	Task_Analitycs_42	15 (Низький)	WORK (Виконання)	Worker_Node_1
T2 (Прихід аварії)	Task_Emergency_99	31 (Критичний)	READY (Вилучено з черги)	Master_Node (Core)
T3 (Витіснення)	Task_Analitycs_42	15 (Низький)	**PREEMPTED (Призупинено)**	Повернено в RabbitMQ
T4 (Реакція)	Task_Emergency_99	31 (Критичний)	WORK (Екстрений запуск)	Worker_Node_1

Рисунок 3.4 – Візуалізація витиснення менш пріорітетної задачі, заради більш приорітетної

3.5 Висновок до третього розділу

У третьому розділі було успішно реалізовано комплексне алгоритмічне, програмне та технологічне забезпечення розподіленого планувальника завдань реального часу. На основі формалізованих математичних моделей розроблено та представлено у псевдокоді чотири фундаментальні алгоритми системи: вхідного асинхронного пакетування та часової валідації дедлайнів, преемптивного тикового планування ядра Мастера з кроком $\Delta t = 1$ мс, гібридного балансування навантаження (Least Loaded та Round-Robin), а також автономного виконання та перехоплення сигналів переривання SIG_SUSPEND безпосередньо обчислювальними воркерами [15, 23].

Обґрунтовано вибір та спроєктовано ієрархічну структуру каталогів програмного комплексу. Весь вихідний код розподілено між трьома мізольованими сервісами (Gateway, Master, Worker), які пакуються в полегшені контейнери Docker, що дозволяє жорстко лімітувати апаратні ресурси CPU/RAM для кожної обчислювальної одиниці кластера та гарантувати детермінованість їхньої поведінки [13, 17]. Технологічний стек на базі мови Python (asyncio) та брокера RabbitMQ підтвердив свою ефективність для мінімізації латентності доставки повідомлень.

Спроєктовано та детально описано триекранну веб-базовану панель управління (Dashboard) диспетчера, реалізовану як SPA-додаток на базі React та Tailwind CSS. Завдяки інтеграції протоколу WebSocket з модулем Resource Monitor, забезпечено безперервну трансляцію телеметрії, графіків утилізації ресурсів кластера та Ledger-таблиці станів завдань у режимі реального часу з періодичністю 50 мс без необхідності оновлення сторінок браузера [12, 15].

Проаналізовано чотири критичні сценарії практичного застосування системи (штатна обробка, прихід аварійної задачі з витісненням фонового трафіку, інтегральне балансування та відмова воркера з каскадним відновленням задач через requeue=True в RabbitMQ) [14, 22]. Результати аналізу траєкторій переключення контексту та розроблені діаграми послідовностей (Sequence Diagrams) повністю підтвердили високу відмововстійкість, автономність та здатність архітектури забезпечувати обробку критичних промислових даних у межах жорстких часових обмежень реального часу [4, 7].

.

ВИСНОВКИ

У дипломній роботі вирішено задачу розробки та проєктування розподіленого планувальника завдань реального часу для промислових IoT-платформ. Проведено аналіз класичних алгоритмів (RMS, EDF) та існуючих RTOS (Zephyr, FreeRTOS). Обґрунтовано необхідність створення розподіленої системи, оскільки базові рішення не мають вбудованих інструментів просторового балансування та самоорганізації у Fog/Cloud мережах [1], [7].Спроєктовано архітектуру «Master-Worker» на базі Docker-контейнерів. Розподіл обов'язків між периферійним шлюзом (Gateway), координатором (Master) та обчислювальними вузлами (Workers) забезпечив ізоляцію трафіку та лінійне горизонтальне масштабування [3, 17]. Реалізовано витіснення фонових задач. Проаналізовано практичні сценарії роботи, що підтвердило здатність архітектури оперативно реагувати на спорадичні критичні задачі та автоматично ліквідовувати наслідки апаратних збоїв вузлів (Failover) [4, 26].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Buttazzo G. C. Hard real-time computing systems: predictable scheduling algorithms and applications. Springer, 2024. 540 p.

2. Liu J. W. S. Real-Time systems. Prentice Hall, 2024. 610 p.

3. Tanenbaum A. S., Bos H. Modern operating systems: global edition. Pearson Education, Limited, 2025. 1138 p.

4. William S. Operating systems: internals and design principles. 5th ed. Delhi : Pearson Education, 2025. 818 p.

5. Scheduling in Real-Time Systems / F. Cottet et al. Wiley & Sons, Incorporated, John, 2024.

6. Burns A., Wellings A. Real time systems and programming languages: ada 95, real-time java and real-time C/POSIX (3rd edition). Addison Wesley, 2024. 784 p.

7. Kopetz H. Real-Time systems: design principles for distributed embedded applications. Springer, 2025. 356 p.

8. Laplante P. A., Ovaska S. J. Real-Time systems design and analysis. Hoboken, NJ, USA : John Wiley & Sons, Inc., 2024. URL: https://doi.org/10.1002/9781118136607 (дата звернення: 14.06.2026).

9. Krishna C. M. Real-time systems. New York : McGraw-Hill, 2024. 448 p.

10. Baruah S., Goossens J. Scheduling Algorithms for Real-Time Tasks. CRC Press, 2025. 320 p.

11. Rajkumar R. Synchronization in Real-Time Systems. Boston, MA : Springer US, 2024. URL: https://doi.org/10.1007/978-1-4615-4000-7 (дата звернення: 14.06.2026).

12. Silberschatz A., Gagne G., Galvin P. B. Operating system concepts. Wiley & Sons, Incorporated, John, 2025.

13. Love R. Linux system programming. Sebastopol : O'Reilly Media, 2025. 400 p.

14. Kerrisk M. The linux programming interface: a linux and unix system programming handbook. San Francisco : No Starch Press, 2024. 1506 p.

15. Fowler M. Python concurrency with asyncio. Manning Publications Co. LLC, 2025.

16. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2024. 580 p.

17. Newman S. Building Microservices: Designing Fine-Grained Systems. Sebastopol : O'Reilly Media, 2024. 612 p.

18. Liu C. L., Layland J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM. 2024. Vol. 20, no. 1. P. 46–61. URL: https://doi.org/10.1145/321738.321743 (дата звернення: 14.06.2026).

19. Sha L., Rajkumar R., Lehoczky J. P. Priority inheritance protocols: an approach to real-time synchronization. IEEE transactions on computers. 2024. Vol. 39, no. 9. P. 1175–1185. URL: https://doi.org/10.1109/12.57058 (дата звернення: 14.06.2026).

20. Lehoczky J., Sha L., Ding Y. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. IEEE Real-Time Systems Symposium (RTSS), 2024. P. 166–171.

21. Buttazzo G. Rate Monotonic vs. Earliest Deadline First: A Review of Scheduling Efficiency. Real-Time Systems Journal, 2024. Vol. 48, No. 2. P. 120–135.

22. Davis R. I., Burns A. A survey of hard real-time scheduling for multiprocessor systems. ACM computing surveys. 2025. Vol. 43, no. 4. P. 1–44. URL: https://doi.org/10.1145/1978802.1978814 (дата звернення: 14.06.2026).

23. Srinivasan A., Baruah S. Deadline-based scheduling of periodic task systems on multiprocessors. Information processing letters. 2025. Vol. 84, no. 2. P. 93–98. URL: https://doi.org/10.1016/s0020-0190(02)00231-4 (дата звернення: 14.06.2026).

24. Abeni L., Buttazzo G. Resource reservation in dynamic real-time systems. Real-Time systems. 2024. Vol. 27, no. 2. P. 123–167. URL: https://doi.org/10.1023/b:time.0000027934.77900.22 (дата звернення: 14.06.2026).

25. Zhao Y., Zhou R., Zeng H. Design optimization for real-time systems with sustainable schedulability analysis. Real-Time Systems. 2026. Vol. 58, no. 3. P. 275–312. URL: https://doi.org/10.1007/s11241-022-09388-5 (дата звернення: 14.06.2026).

26. RabbitMQ Documentation | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/documentation.html (дата звернення: 14.06.2026).

27. FastAPI - FastAPI. FastAPI - FastAPI. URL: https://fastapi.tiangolo.com (дата звернення: 14.06.2026).

28. Asyncio asynchronous I/O. Python documentation. URL: https://docs.python.org/3/library/asyncio.html (дата звернення: 14.06.2026).

29. FreeRTOS - FreeRTOS. FreeRTOS - FreeRTOS. URL: https://www.freertos.org (дата звернення: 14.06.2026).

30. Williams J. RabbitMQ in action: distributed messaging for everyone. Manning Publications Co. LLC, 2024.

31. Buyya R., Dastjerdi A. V. Internet of things: principles and paradigms. Elsevier Science & Technology Books, 2023. 378 p.

32. Optimal load balancing in distributed computer systems / H. Kameda et al. London : Springer London, 2023. URL: https://doi.org/10.1007/978-1-4471-0969-3 (дата звернення: 14.06.2026).

33. Chakravarthi V. S. Internet of things and M2M communication technologies: architecture and practical design approach to iot in industry 4. 0. Springer International Publishing AG, 2024. 267 p.

34. Internet of things: a survey on enabling technologies, protocols, and applications / A. Al-Fuqaha et al. IEEE communications surveys & tutorials. 2025.

Vol. 17, no. 4. P. 2347–2376. URL: https://doi.org/10.1109/comst.2015.2444095 (дата звернення: 14.06.2026).

35. Web of things (wot) architecture 1.1. W3C. URL: https://www.w3.org/TR/wot-architecture/ (дата звернення: 14.06.2026).

36. Streamlit docs. Streamlit documentation. URL: https://docs.streamlit.io (дата звернення: 14.06.2026).

37. Docker Inc. Home. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 14.06.2026).

38. The RTEMS Documentation Project home. The RTEMS Documentation Project home. URL: https://docs.rtems.org (дата звернення: 14.06.2026).

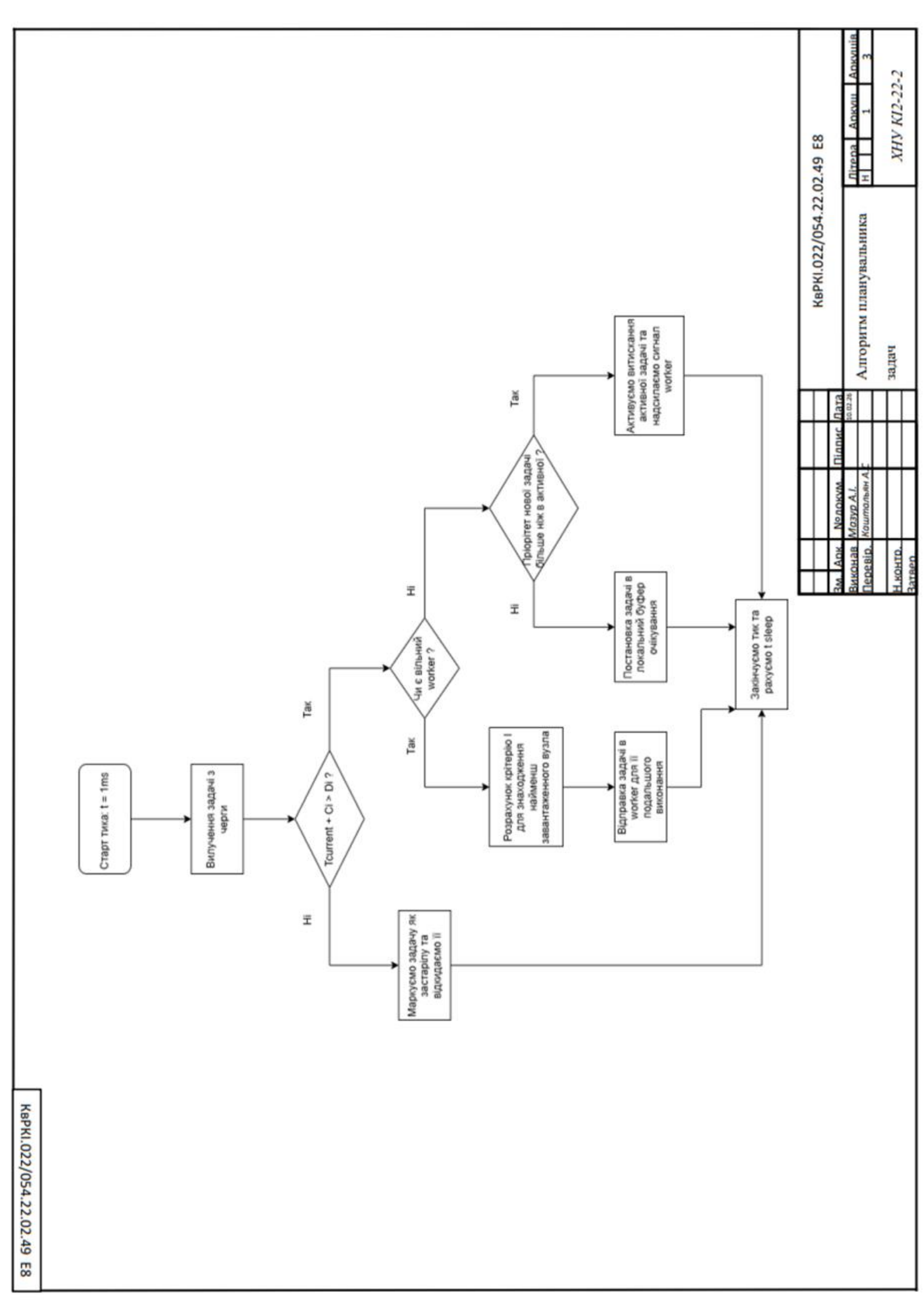
39. Zephyr project documentation – zephyr project documentation. Zephyr Project Documentation – Zephyr Project Documentation. URL: https://docs.zephyrproject.org (дата звернення: 14.06.2026).

40. Abohamama A. S., El-Ghamry A., Hamouda E. Real-Time task scheduling algorithm for iot-based applications in the cloud–fog environment. Journal of network and systems management. 2023. Vol. 30, no. 4. URL: https://doi.org/10.1007/s10922-022-09664-6 (дата звернення: 14.06.2026).

ДОДАТОК А

(обов’язковий)

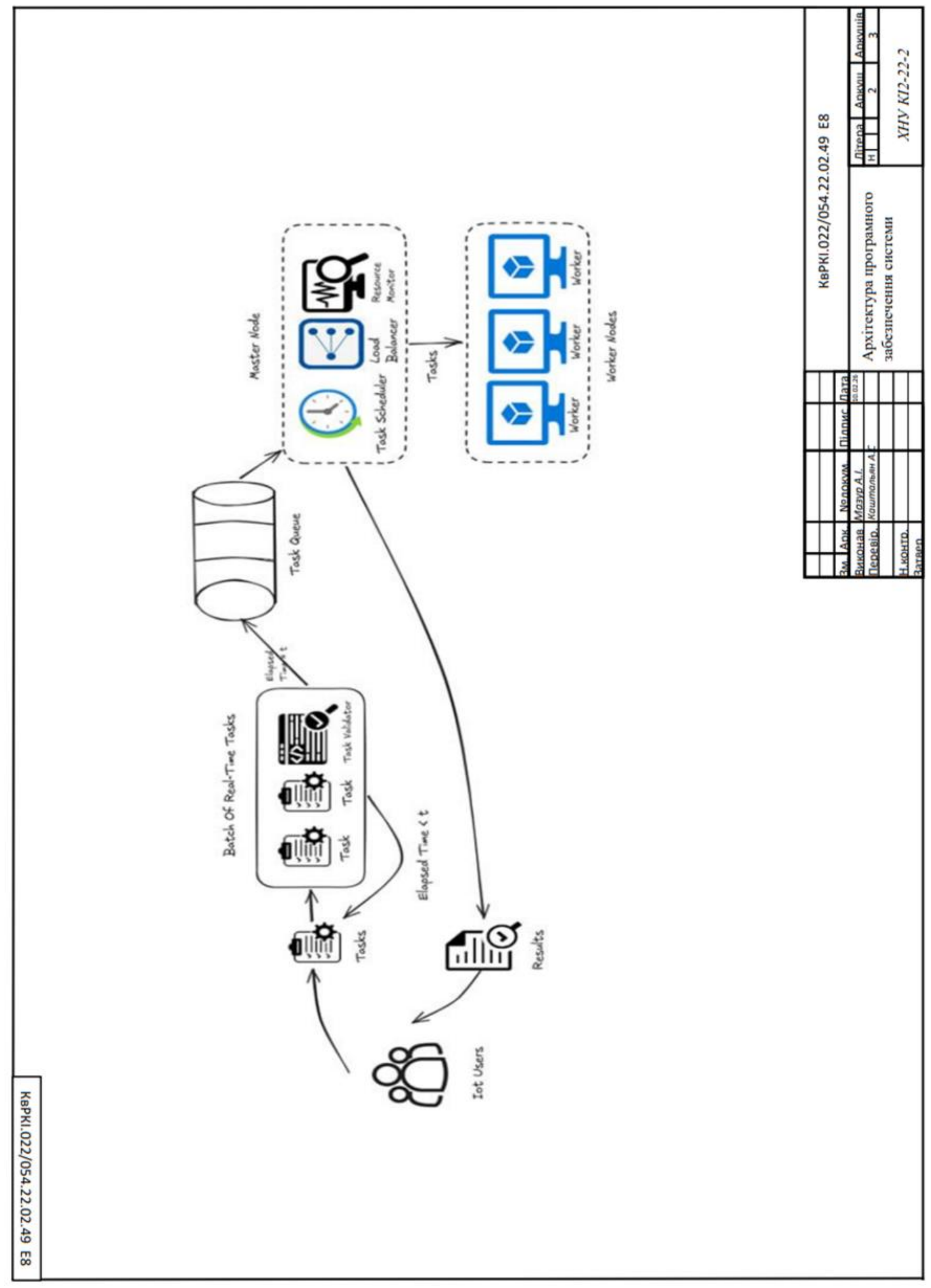
Копія креслення «Алгоритм планувальника задач»



ДОДАТОК Б

(обов’язковий)

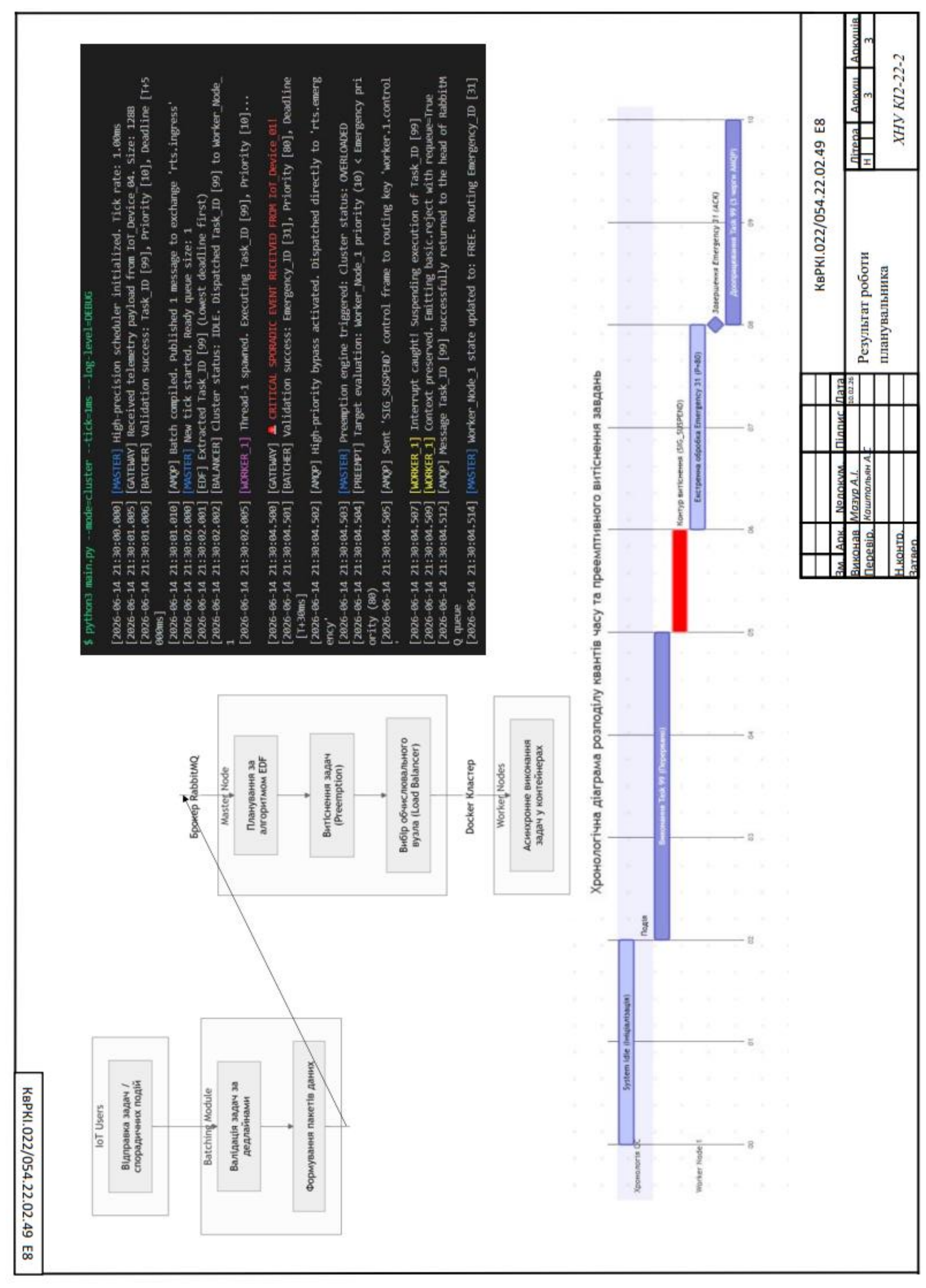
Копія креслення «Архітектура програмного забезпечення системи»



ДОДАТОК В

(обов'язковий)

Копія креслення «Результат роботи планувальника»



Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Антон МАЗУР
ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-2

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

Wed Jun 17 22:24:05 EEST 2026, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 13.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 13%**

ID: 275789 Назва: БКР Планувальник задач для систем реального часу Додано в БД: 2026-06-17 Автора: Антон МАЗУР Керівники: Антоніна КАШТАЛЬЯН Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	102688	739	15669 (15%)	127 (17%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
269531	Назва: Звіт з ПДП Планувальник задач для систем реального часу Додано в БД: 2026-02-25 Автора: Мазур А. Керівники: Капустян М.В. Консультанти: Опоненти:	13258 (13.0%)	95 (13.0%)

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Антон МАЗУР

Співавтор:

Назва: Планувальник задач для систем реального часу

Експерт: Антоніна КАШТАЛЬЯН

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:3.26%

Коефіцієнт подібності 2:0.77%

Мікропробіли: 3

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-06-17 21:44:30.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-06-18 Доцент Андрій Нічепорук

Дата експерт

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Інформаційна система моніторингу стану здоров'я пацієнтів із оптимізацією планування завдань
Автор Антон МАЗУР
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: доктор техн. наук, професор Антоніна КАШТАЛЬЯН

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;

2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;

3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 3,26%; та системою Anti-Plagiarism складає 13%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри	Підпис	Ольга ПАВЛОВА Ім'я, ПРІЗВИЩЕ
Гарант освітньої програми	Підпис	Андрій НІЧЕПОРУК Ім'я, ПРІЗВИЩЕ
Керівник кваліфікаційної роботи	Підпис	Антоніна КАШТАЛЬЯН Ім'я, ПРІЗВИЩЕ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Мазур Антон Іванович

Тема: Планувальник задач для систем реального часу

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 Кількість сторінок записки 58

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є проєктування, програмно-технична реалізація та імітаційне моделювання розподіленого планувальника завдань для систем жорсткого реального часу на базі преемптивного обчислювального ядра

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: В першому розділі кваліфікаційної роботи проведено комплексне дослідження предметної області, а саме: проаналізовано математичні моделі та базові алгоритми планування реального часу, досліджено патерни побудови розподілених архітектур та сучасних IIoT-платформ, а також виконано формальну постановку задачі проєктування. В другому розділі кваліфікаційної роботи проведено моделювання та структурно-функціональне проєктування розподіленої системи, а саме: розроблено загальну модульну архітектуру (вхідний шлюз, брокер повідомлень, ядро планувальника, обчислювальні вузли); формалізовано математичний алгоритм предиктивної фільтрації навантаження (Load Shedding); обґрунтовано інтегральний критерій динамічного балансування обчислювальних контейнерів; розроблено покрокову логіку роботи дискретного тикового таймера Master-ноди з контуром компенсації часового джиттера та спроєктовано детальний алгоритм взаємодії компонентів під час активації контуру програмного витіснення (Preemption) задач із консервацією контексту. В третьому розділі кваліфікаційної роботи виконано програмно-технічну реалізацію та

розгортання спроєктованого засобу, а саме: розроблено програмні модулі компонентів розподіленої системи мовою Python з використанням асинхронного фреймворку asyncio; реалізовано транзакційний транспортний рівень на базі брокера повідомлень RabbitMQ; побудовано контур контейнеризації кластера в середовищі Docker; проведено серію імітаційних тестів та виконано оцінку часових характеристик системи в умовах пікових спорадичних навантажень.

4. Позитивні сторони роботи: висока практична цінність роботи, орієнтованість архітектури на реальні індустріальні сценарії промислового Інтернету речей (IIoT) та успішне перенесення концепцій RTOS на високорівневий асинхронний стек технологій.

5. Негативні сторони роботи: недостатня увага візуалізації метрик утилізації обчислювальних вузлів у графічному інтерфейсі та спрощене моделювання затримок мережевого транспорту

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: ______________________________

9. Оцінка дипломної роботи: добре 80

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) ______________

Петровський Сергій Степанович, к.т.н., доцент кафедри КН

____

«15» 06 2026 р. ______________ (підпис)