Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних систем управління

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Мобільно-орієнтована система централізованої обробки даних розумного будинку.»

Виконав: студент 2 курсу, групи 2АКІТР-23м
спеціальності 174 – Автоматизація, комп'ютерно-інтегровані технології, та робототехніка
Богдан МЕЛЬНИК
Ім'я ПРІЗВИЩЕ
Керівник: Зав. кафедри, проф.
степінь, звання, посада
В'ячеслав КОВТУН
Ім'я ПРІЗВИЩЕ
«4» 12 2024 р.

Опонент: к.т.н., доцент
степінь, звання, посада
Юрій ІВАНОВ
Ім'я ПРІЗВИЩЕ
«4» 12 2024 р.

Допущено до захисту Зав. кафедри
КСУ В'ячеслав КОВТУН
«4» 12 2024 р.

Вінниця ВНТУ – 2024рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних систем управління
Рівень вищої освіти другий (магістерський)
Галузь знань – 17 – Електроніка, автоматизація та електронні комунікації
Спеціальність – 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка
Освітньо-професійна програма – Інтелектуальні комп'ютерні системи

ЗАТВЕРДЖУЮ
Зав. кафедри КСУ

В'ячеслав КОВТУН

"14" жовтня 2024 року

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Мельнику Богдану Олеговичу.
(прізвище, ім'я, по батькові)

1. Тема роботи: Мобільно-орієнтована система централізованої обробки даних розумного будинку.
керівник роботи Ковтун В.В.
затверджені наказом ВНТУ від "17" вересня 2024 року №310
2. Термін подання студентом роботи "1" грудня 2024 року
3. Вихідні дані до роботи: вихідні дані до роботи представлені у структурованому csv файлі. Формуючи вихідні дані автор використовував такі інформаційні джерела:

1. O. Popov, A. Yatsyshyn, and I. Ivanov, "Designing a Smart Home System Based on IoT Technologies," Electronics and Control Systems, vol. 2, no. 60, pp. 80-88, 2021. [Online]. Available: https://doi.org/10.18372/1990-5548.60.15633. (9 стор.)

2. M. Smith and K. Johnson, "Mobile-Oriented Cloud-Based Smart Home Data Processing," IEEE Access, vol. 8, pp. 24012-24022, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2973345. (11 стор.)

3. P. Borysov, "Internet of Things and Mobile Application Design for Smart Home Data Management," Scientific Bulletin of the National Technical University of Ukraine, no. 5, pp. 45-52, 2021. (8 стор.)

4. Зміст текстової частини: текстова частина включає анотацію, зміст, вступ, основну частину з першим, оглядовим, розділом, висновки, список посилань та додатки.
5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень): об'єкт, предмет та мета дослідження, структурна схема запропонованої програмно-апаратної системи, алгоритми роботи програної складової розробки, результати тестваннння створеної системи.

1. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	виконання прийняв
4	Кавецький В.В. – доцент кафедри економіки підприємства і виробничого менеджменту		

2. Дата видачі завдання "14" жовтня 2024 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1.	Аналітичний огляд переваг та недоліків, які характеризують досліджуваний технологічний процес	09.10.24	11.10.24	
2.	Дослідження стану розв'язання технічної задачі	23.10.24	29.10.24	
3.	Розробка та тестування системи	01.11.24	19.11.24	
4.	Розрахунок економічної частини	21.11.24	22.11.24	
5.	Оформлення матеріалів до захисту	26.11.24	28.11.24	

Студент (підпис) Богдан МЕЛЬНИК
(*Ім'я ПРІЗВИЩЕ*)

Керівник роботи (підпис) В'ячеслав КОВТУН
(*Ім'я ПРІЗВИЩЕ*)

АНОТАЦІЯ

УДК.669.05

Мельник Б.О. Мобільно-орієнтована система централізованої обробки даних розумного будинку.Магістерська кваліфікаційна робота зі спеціальності 174 «Автоматизація, комп’ютерно-інтегровані технології та робототехніка», освітня програма – Інтелектуальні комп’ютерні системи. Вінниця: ВНТУ, 2024. 162 с.

На укр. мові. Бібліогр.: 19 назв; рис.: 35; табл. 15.

У магістерській роботі розглянуто проблему семантичного аналізу великих обсягів XML-файлів на мобільних пристроях під платформу Android. Завдання полягало у розробці ефективних методів обробки XML-файлів з урахуванням багатопоточності для підвищення швидкості та зниження ресурсоємності процесів. Розроблено бібліотеку, що містить вісім методів обробки даних, реалізованих з використанням бібліотек Threads, Kotlin Coroutines та RxJava. Кожен з методів тестувався за кількома критеріями, такими як час обробки та витрати ресурсів. Визначено найоптимальніші методи обробки XML для різних умов роботи пристроїв. Створено алгоритм, який дозволяє вибирати найбільш ефективний метод для конкретних параметрів девайсу, таких як кількість ядер, заряд батареї та навантаженість системи. Практична значущість роботи полягає у створенні інструменту для ефективної обробки даних у мобільних додатках, що можуть працювати з великими XML-файлами без значного навантаження на пристрій.

Ключові слова: мобільна операційна система, Android, багатопоточність, XML-парсинг, Kotlin Coroutines, RxJava.

ABSTRACT

УДК.669.05

Melnyk B.O. Mobile-Oriented System for Centralized Smart Home Data Processing. Master’s Qualification Thesis in Specialty 174 "Automation, Computer-Integrated Technologies, and Robotics," Educational Program – Intelligent Computer Systems. Vinnytsia: Vinnytsia National Technical University, 2024. - 162 p.

In Ukrainian. Bibliography: 19 references; illustrations: 35; tables: 15.

The master’s thesis examines the issue of semantic analysis of large volumes of XML files on mobile devices operating on the Android platform. The objective was to develop efficient XML file processing methods considering multithreading to improve speed and reduce resource consumption. A library comprising eight data processing methods was developed using Threads, Kotlin Coroutines, and RxJava. Each method was evaluated based on several criteria, including processing time and resource consumption. The most optimal XML processing methods were identified for various device operating conditions. An algorithm was developed to select the most effective method for specific device parameters, such as the number of cores, battery level, and system load. The practical significance of the research lies in creating a tool for efficient data processing in mobile applications that can handle large XML files without imposing substantial load on the device.

Keywords: mobile operating system, Android, multithreading, XML parsing, Kotlin Coroutines, RxJava.

ЗМІСТ

ВСТУП

Мобільні операційні системи стали невід'ємною частиною сучасної реальності, оскільки важко уявити людину без мобільного пристрою, який дозволяє не лише взаємодіяти з контентом в Інтернеті, але й здійснювати безліч інших операцій, від спілкування у соціальних мережах до отримання важливої інформації в режимі реального часу. Зручність, мобільність і постійна доступність таких пристроїв — це те, що стало основою їх популярності у повсякденному житті. Смартфони завжди під рукою або в кишені, забезпечуючи безперервний доступ до необхідних даних і ресурсів.

Розробка мобільних додатків стала важливим напрямом у сфері програмування, зокрема для таких компаній, які прагнуть обробляти й використовувати інформацію в реальному часі. Однак, на ринку мобільних операційних систем домінують кілька великих компаній, які контролюють оновлення платформ і визначають умови для розробників програмного забезпечення. Вони встановлюють базові правила для додатків, серед яких одними з найбільш важливих є зручність інтерфейсу і швидкість обробки даних.

Тим не менш, є ряд завдань, вирішення яких на сучасних мобільних пристроях потребує значних обчислювальних потужностей і значного навантаження на систему. Одним з таких складних і ресурсоємних завдань є обробка даних Інтернету речей (IoT). Ці дані генеруються численними сенсорами і пристроями, що активно взаємодіють між собою та з іншими системами в режимі реального часу. Обробка і аналіз таких обсягів даних вимагає використання ефективних алгоритмів і методів, здатних забезпечити високу продуктивність при мінімальному використанні ресурсів мобільного пристрою.

Зокрема, IoT-системи створюють потоки даних, які можуть бути надзвичайно великими й непередбачуваними за своєю природою. Це вимагає розробки нових підходів до обробки таких даних, щоб забезпечити швидкий аналіз, збереження корисної інформації та прийняття рішень у реальному часі. Оскільки обсяги таких даних зростають, а час реагування на зміни у середовищі

стає критичним, ефективне використання ресурсів є одним з ключових аспектів у створенні додатків для обробки IoT-даних.

Метою цієї роботи є розробка та оптимізація методів обробки даних IoT у форматі XML, що дозволить підвищити ефективність і зменшити навантаження на мобільні пристрої. Для цього буде створена спеціалізована бібліотека, яка інтегрується в мобільний додаток на платформі Android. Мета полягає в тому, щоб оптимізувати процеси обробки великих обсягів даних, що надходять з сенсорів і пристроїв IoT, використовуючи багатопоточність і паралельне обчислення для підвищення ефективності.

У процесі дослідження буде здійснено порівняння різних підходів до обробки IoT-даних, оцінюючи їх за такими критеріями, як час виконання та споживання ресурсів. Для реалізації цього було розроблено бібліотеку з використанням різних інструментів для багатопотокової обробки даних, таких як Threads, RxJava та Kotlin Coroutines. В рамках цих технологій реалізовано кілька методів обробки: від обробки даних в один потік до використання декількох потоків для паралельного аналізу даних. Кожен метод має свої особливості у споживанні ресурсів та швидкості виконання, що дозволяє вибирати найбільш ефективне рішення в залежності від умов виконання.

Це дослідження зосереджене на розробці алгоритмів для оптимізації обробки IoT-даних, що дозволить максимально ефективно використовувати обчислювальні потужності мобільних пристроїв, враховуючи їх обмеження. Алгоритм, розроблений у межах роботи, здатний вибирати оптимальний метод обробки в залежності від характеристик конкретного пристрою і наданих умов, що дозволить забезпечити швидке і ресурсоефективне прийняття рішень у реальному часі для IoT-систем.

Об'єкт дослідження:

Процеси обробки великих обсягів даних у форматі XML на мобільних пристроях під платформу Android, зокрема в контексті Інтернету речей (IoT).

Предмет дослідження:

Методи обробки XML-даних з використанням багатопоточності, паралельних обчислень і специфічних технологій (Threads, Kotlin Coroutines, RxJava) для оптимізації обробки IoT-даних на мобільних пристроях.

Мета дослідження:

Розробка та оптимізація методів обробки даних IoT у форматі XML для підвищення ефективності та зниження навантаження на мобільні пристрої, зокрема через застосування багатопоточних та паралельних обчислень.

Завдання дослідження:

1. Розробити бібліотеку для обробки XML-даних з використанням багатопоточних технологій.

2. Порівняти різні методи обробки XML-даних за критеріями часу виконання та споживання ресурсів.

3. Визначити найоптимальніші методи обробки даних для різних умов роботи мобільних пристроїв.

4. Розробити алгоритм вибору найбільш ефективного методу обробки XML в залежності від характеристик пристрою.

5. Провести тестування розроблених методів на реальних пристроях з урахуванням їх характеристик (кількість ядер, заряд батареї, навантаженість системи).

Практична цінність дослідження:

Створення інструменту для ефективної обробки великих обсягів XML-даних у мобільних додатках, що працюють з IoT-системами, дозволяючи забезпечити високу продуктивність при мінімальному використанні обчислювальних ресурсів мобільного пристрою. Розроблені методи і алгоритм дозволяють оптимізувати процеси обробки даних у реальному часі, що має велике значення для мобільних додатків, які працюють з даними від численних сенсорів та пристроїв.

Наукова новизна дослідження:

1. Розробка нових методів обробки XML-даних для мобільних пристроїв з урахуванням багатопоточності та паралельних обчислень.

2. Створення алгоритму вибору оптимальних методів обробки даних на основі характеристик мобільного пристрою, що дозволяє підвищити ефективність та знизити витрати ресурсів.

3. Оцінка різних підходів до обробки IoT-даних на мобільних пристроях з порівнянням їх за кількома критеріями ефективності.

1. ОГЛЯД СУЧАСНИХ МЕТОДІВ СЕМАНТИЧНОГО АНАЛІЗУ XML-ФАЙЛА З IoT ДАНИМИ

1.1 Загальний огляд об’єкта дослідження

Серед сучасних мобільних операційних систем на сьогоднішній день найбільшу популярність здобула платформа Android, яка забезпечує зручну та ефективну роботу з великою кількістю додатків, спрямованих на обробку різноманітних типів даних. Одним з найбільш поширених форматів для зберігання і обміну даними є XML (eXtensible Markup Language) — універсальний та гнучкий формат для структурування і представлення даних, який активно використовується у багатьох IoT-системах. XML-файли можуть містити величезну кількість інформації, і саме їх зручність і здатність бути підтримуваними майже всіма мобільними та веб-додатками робить їх популярним інструментом для обробки даних, що надходять від різних сенсорів IoT-пристроїв.

У контексті Інтернету речей (IoT) дані, що генеруються пристроями та сенсорами, можуть бути представлені у вигляді XML-документів[3], що дозволяє зберігати та обмінювати великі обсяги інформації між пристроями, серверами та мобільними додатками [9]. Обробка таких даних є необхідною для ефективного функціонування IoT-систем, оскільки ці дані часто містять як структуровану, так і неструктуровану інформацію, яка потребує семантичного аналізу для подальшого використання.

Особливості обробки IoT-даних в XML-файлах включають необхідність швидкої і точнішої обробки великих обсягів даних[4], забезпечення високої продуктивності при обмежених ресурсах мобільних пристроїв та адаптацію до різних типів даних, що надходять від різноманітних сенсорів. Тому методи семантичного аналізу, здатні ефективно обробляти ці дані, мають велике значення для розробки додатків, які працюють в умовах обмежених ресурсів і високої динамічності інформації.

У цьому розділі буде розглянуто основні підходи до семантичного аналізу XML-файлів, які використовуються для обробки IoT-даних. Зокрема, йдеться про різні методи парсингу, багатопоточності та оптимізації процесів обробки, а також

їх вплив на продуктивність мобільних пристроїв, які мають обмежені обчислювальні потужності при роботі з великими обсягами даних.

1.1.1 Деталізований аналіз проблеми централізованої обробки даних у розумному будинку із застосуванням мобільних додатків

Розумний будинок, як технологія майбутнього, забезпечує автоматизацію та інтеграцію різноманітних побутових функцій: управління освітленням, температурою, безпекою, розважальними системами тощо. Однак зростання кількості таких пристроїв і обсягу даних, що передаються для централізованої обробки, створює численні проблеми[2], пов'язані з продуктивністю, безпекою, конфіденційністю та взаємодією з мобільними додатками. Для кращого розуміння цих проблем варто детально розглянути їх ключові аспекти.

Масштабованість та продуктивність системи. Централізована обробка даних передбачає, що всі дані з датчиків і пристроїв розумного будинку передаються на сервери або у хмарні платформи для обробки і зберігання[6]. Зі збільшенням кількості підключених пристроїв і зростанням кількості даних, які необхідно обробляти, з'являється проблема масштабованості. Центральний сервер або хмарна платформа може стикнутися з труднощами при обробці великої кількості одночасних запитів.

Зокрема, такі проблеми можуть проявлятися в наступному:

- Затримка в обробці даних: Наприклад, якщо велика кількість пристроїв передає дані одночасно, це може призвести до перевантаження каналів зв'язку та серверів, що знижує швидкість реакції системи[3].

- Обмеження пропускної здатності мережі: Централізована система може залежати від швидкості інтернет-з'єднання користувача, а також від доступної пропускної здатності каналів, що впливає на якість обробки даних у реальному часі.

Можливі рішення: Одним з підходів до вирішення цієї проблеми є перехід до децентралізованих обчислювальних архітектур, таких як edge computing[7]. У таких системах дані обробляються локально, на рівні пристроїв або мережі, що знижує навантаження на центральний сервер і дозволяє зменшити затримки.

Безпека і конфіденційність даних. Централізоване зберігання і обробка даних часто потребує передачі чутливої інформації (наприклад, відеозаписів із камер спостереження, даних про присутність людей у будинку, або їхньої активності) на зовнішні сервери. Це створює кілька загроз:

- Небезпека витоку даних: Якщо дані зберігаються на централізованих серверах, вони можуть стати ціллю для хакерських атак. У випадку компрометації серверів, зловмисники можуть отримати доступ до даних про поведінку мешканців, що становить серйозну загрозу їхній безпеці.

- Проблеми з конфіденційністю: Користувачі розумних будинків можуть мати сумніви щодо передачі своїх особистих даних на сторонні сервери. Навіть за наявності шифрування, централізовані системи можуть залишатися вразливими до зловмисників або втручання сторонніх осіб.

Можливі рішення: Одним із способів підвищення безпеки є застосування локальних обчислювальних центрів[9] у поєднанні з технологіями шифрування даних. Це дозволить мінімізувати кількість даних, що передаються на зовнішні сервери, і зберігати більшість інформації на рівні локальних пристроїв.

Залежність від інтернет-з'єднання. Централізовані рішення для розумних будинків вимагають стабільного інтернет-з'єднання для передачі даних між пристроями та сервером. У випадку нестабільного з'єднання або його відсутності виникає кілька проблем:

- Неможливість віддаленого управління: Якщо мобільний додаток не може підключитися до централізованого сервера через інтернет[8], користувач втрачає можливість дистанційного контролю за системами розумного будинку.

- Втрати у функціональності: Системи можуть бути налаштовані так, що втрачають частину своєї функціональності при відсутності підключення до інтернету, наприклад, в режимі автономної роботи.

Можливі рішення: Одним з підходів до вирішення цієї проблеми є розробка систем із автономними функціями, які можуть працювати без підключення до інтернету, а також резервні рішення, що використовують локальні мережі для базового управління будинком навіть у випадку відсутності з'єднання з глобальною мережею.

Складність інтеграції та сумісності. У розумному будинку може бути встановлено безліч пристроїв різних виробників, кожен з яких використовує власні протоколи зв'язку і стандарти. Це ускладнює процес інтеграції всіх пристроїв в єдину централізовану систему обробки даних, особливо якщо для їхнього управління використовуються мобільні додатки:

- Відсутність єдиних стандартів: Кожен виробник може використовувати різні стандарти передачі даних, що ускладнює їх інтеграцію. Деякі мобільні додатки можуть підтримувати тільки певні платформи або стандарти.

- Необхідність використання кількох додатків: Якщо користувач має пристрої від різних виробників, він може бути змушений використовувати декілька різних мобільних додатків для управління системами розумного будинку, що знижує зручність.

Можливі рішення: Важливим аспектом вирішення цієї проблеми є розробка стандартів для інтероперабельності пристроїв і систем. Застосування відкритих протоколів, таких як Zigbee або Z-Wave [10], дозволить забезпечити сумісність між різними пристроями і платформами.

Витрати на підтримку і обслуговування. Централізовані системи розумного будинку потребують постійної підтримки, оновлень програмного забезпечення і технічного обслуговування для забезпечення стабільної роботи:

- Залежність від сторонніх сервісів: Користувачі можуть бути змушені підписуватися на платні послуги хмарних платформ для забезпечення зберігання даних і їх обробки.

- Оновлення системи: Зі збільшенням кількості підключених пристроїв і появою нових технологій, необхідно забезпечувати регулярні оновлення програмного забезпечення для забезпечення сумісності та функціональності системи.

Можливі рішення: Одним із підходів до зниження витрат може бути перехід до гібридних систем обробки даних, які використовують як локальні сервери, так і хмарні технології для балансування навантаження.

Енергоспоживання. Централізована обробка даних у розумному будинку може призвести до підвищення енергоспоживання, оскільки всі пристрої повинні

постійно передавати дані на сервери для обробки. Це може впливати як на енергоспоживання самих пристроїв, так і на сервери, що здійснюють обробку цих даних.

Можливі рішення: Оптимізація енергоспоживання може бути досягнута через використання енергоефективних протоколів зв'язку і технологій на зразок Bluetooth Low Energy (BLE) або LoRaWAN, що знижують навантаження на мережу і споживання енергії пристроями.

Централізована обробка даних у розумних будинках за допомогою мобільних додатків має як переваги, так і серйозні виклики. Серед основних проблем варто відзначити масштабованість і продуктивність системи, питання безпеки і конфіденційності, залежність від інтернет-з'єднання, складність інтеграції пристроїв, витрати на підтримку, а також енергоспоживання. Можливі рішення включають перехід до децентралізованих архітектур, таких як edge computing, які дозволяють обробляти дані безпосередньо на пристроях або в межах локальної мережі, що знижує навантаження на центральні сервери і підвищує безпеку даних. Важливо також забезпечити стандартизацію протоколів для інтеграції різних пристроїв і розробку рішень для локальної автономної роботи систем у випадку відсутності доступу до мережі інтернет.

1.1.2 Мобільна платформа Android в контексті обробки даних IoT

Мобільні операційні системи стали основою для більшості сучасних пристроїв, і серед них Android займає одну з лідируючих позицій. Ця операційна система не лише поширена серед кінцевих користувачів, але й активно використовується для інтеграції з системами Інтернету речей (IoT) [1]. Android став популярною платформою для управління різноманітними пристроями IoT завдяки своїй гнучкості, широкому функціоналу та можливості інтеграції з великою кількістю зовнішніх пристроїв.

З розвитком IoT, де мільйони різноманітних пристроїв, датчиків і систем обмінюються даними, смартфони на платформі Android почали виконувати роль центральних хабів [6] для моніторингу та управління цими пристроями. Завдяки Android, користувачі можуть керувати не лише стандартними мобільними

додатками, але й обробляти потоки даних з IoT-пристроїв у реальному часі, що дозволяє забезпечити безперервний контроль і оптимізацію функціонування розумних будинків або промислових об'єктів.

Одним з головних аспектів обробки IoT-даних є збирання та аналіз інформації, отриманої від різноманітних сенсорів і пристроїв. У цьому контексті, Android пропонує потужні інструменти для інтеграції таких пристроїв через різноманітні протоколи, наприклад, Bluetooth, Wi-Fi, ZigBee, та інші. Додатки на Android можуть отримувати дані в реальному часі, здійснювати їх попередню обробку, фільтрацію та передавати для подальшої аналітики чи зберігання на сервері або в хмарі.

Важливою перевагою Android є можливість використання складних алгоритмів обробки даних безпосередньо на мобільному пристрої. Це особливо корисно для IoT-систем, де дані можуть надходити з різних джерел, і необхідно забезпечити їх швидку обробку, щоб уникнути затримок в управлінні системами. За допомогою таких технологій, як RxJava, Kotlin Coroutines або Threads, Android дає змогу реалізовувати паралельну обробку даних з кількох джерел, що особливо важливо для інтеграції з IoT-пристроями, що генерують великі обсяги інформації.

Більш того, Android підтримує використання машинного навчання для поліпшення якості обробки IoT-даних. Вбудовані в мобільну платформу інструменти для обробки даних, зокрема TensorFlow Lite, дозволяють створювати додатки, які можуть автоматично навчатися на основі наданих даних, покращуючи точність прогнозів або адаптивність системи. Це дає можливість не тільки збирати дані, але й використовувати їх для прогнозування та прийняття рішень на основі виведених закономірностей, що особливо корисно для управління розумними будинками та автоматизації виробничих процесів.

Однак не менш важливим є питання безпеки даних, оскільки IoT-системи часто передають чутливу інформацію про стан домашніх пристроїв або промислових механізмів. Android надає інструменти для забезпечення захищеного з'єднання, такі як шифрування, аутентифікація та авторизація користувачів, що гарантує безпеку передачі даних між мобільним пристроєм і IoT-обладнанням.

Впровадження цих заходів є необхідним для зменшення ризиків витоку даних або несанкціонованого доступу до інтелектуальних систем.

У результаті, завдяки платформі Android, можливості обробки та управління IoT-даними стають доступними і ефективними для кінцевих користувачів. Це дозволяє створювати інтелектуальні системи з високим рівнем адаптивності та автоматизації, що покращує комфорт і ефективність в управлінні домашніми та промисловими процесами. Крім того, Android також забезпечує гнучкість у розробці додатків, які можуть працювати з різноманітними типами пристроїв IoT, що робить її ідеальною платформою для розробки рішень в області Інтернету речей.

1.1.3 Що таке XML-файл та чому він важливий у наш час

XML (Extensible Markup Language) — це мова маркування, яка дозволяє структурувати та зберігати дані у текстовому вигляді [1]. Вона стала основним інструментом для обміну даними в Інтернеті та для інтеграції різних систем. Одна з її головних особливостей — це можливість створення власних тегів, які дозволяють адаптувати XML до специфічних потреб користувачів і завдань. Це робить XML дуже гнучким і універсальним форматом для зберігання та обміну інформацією між різними системами.

XML може бути відкритий будь-яким текстовим редактором, але для більш зручного перегляду та аналізу часто використовуються спеціалізовані інструменти, такі як XML-парсери або браузери, що дозволяють візуалізувати дані у вигляді дерева елементів. Така структура дає можливість чітко розділяти різні типи інформації, що дозволяє зберігати складні дані в упорядкованому вигляді, зберігаючи їх у форматі, зрозумілому як для людини, так і для комп'ютера.

XML є одним із основних форматів для обміну даними в Інтернеті, зокрема в таких галузях, як веб-служби, бази даних, інтеграція програмного забезпечення і, звісно, в Інтернеті речей (IoT). У контексті IoT, XML використовують для передачі даних між різними пристроями, сенсорами та серверами. Пристрої IoT часто генерують величезні об'єми даних, які потрібно передавати між різними платформами та системами для подальшої обробки і аналізу. XML дозволяє

передавати ці дані у структурованому вигляді, зберігаючи їх точність і забезпечуючи можливість інтеграції з іншими технологіями та сервісами.

Для IoT важливою є здатність XML адаптуватися під різні типи даних. Наприклад, датчики температури, вологості, руху або інші пристрої можуть генерувати різноманітні формати даних, і XML дозволяє створювати відповідні теги для кожного з них. Це дозволяє різним пристроям передавати дані в єдиному форматі, що зручно обробляти, а також зберігати їх у центральних базах даних або передавати на сервери для подальшої обробки.

Крім того, XML дозволяє інтегрувати додаткову інформацію, таку як атрибути [11], що можуть містити метадані або додаткові параметри, необхідні для правильної обробки даних. Це дає змогу більш точно описати дані і їх контекст, що в свою чергу підвищує ефективність їх аналізу. Наприклад, для IoT пристроїв важливим аспектом є зберігання часу вимірювання або ідентифікації пристрою, і XML дає можливість вбудувати ці дані безпосередньо в структуру файлу.

Оскільки в IoT можуть використовуватися різні типи пристроїв від різних виробників, XML також виступає як універсальний формат, що дозволяє здійснювати інтеграцію між ними. Незалежно від того, чи це сенсори, розумні прилади, або платформи для збору і аналізу даних, XML дозволяє забезпечити взаємодію між пристроями та забезпечити стандартизовану передачу інформації. Це особливо важливо для реалізації системи, де різні пристрої повинні працювати разом для досягнення спільної мети, наприклад, в системах «розумного дому».

Таким чином, XML є не лише способом зберігання і передачі даних, але й важливим компонентом в екосистемі Інтернету речей. Його здатність до розширення, зручність в обробці і універсальність робить цей формат ідеальним для багатьох сценаріїв IoT, від обміну даними між сенсорами до взаємодії з іншими пристроями та платформами. У майбутньому, з розвитком IoT і зростанням кількості пристроїв, XML буде залишатися ключовим інструментом для забезпечення безперебійного обміну даними, що дозволяє зберігати інформацію в структурованому вигляді та ефективно її обробляти. Наведено у таблиці 1.1

Таблиця 1.1. Відношення спеціальних елементів на їх сутностей

Символ	Заміна
<	<
>	>
&	&

У XML, щоб уникнути плутанини між символами, що використовуються для позначення тегів або атрибутів, важливо мати спеціальні сутності для представлення символів, які зазвичай не можуть бути використані у своєму прямому вигляді. Це зокрема стосується таких символів, як апостроф (') і подвійні лапки ("). В XML ці символи мають спеціальні сутності для їх коректного використання в значеннях атрибутів, щоб уникнути конфлікту з синтаксисом мови [15]. Наведено у таблиці 1.2

Таблиця 1.2. Відношення апострофів,лапок та їх сутності

Символ	**Заміна**
‘	'
“	"

Правило заміни символів, що використовуються для маркування, в XML не застосовується до символічних даних у спеціальних розділах «CDATA», оскільки в цих секціях текст може містити символи, які зазвичай трактуються як частини маркування. Це дозволяє зберегти вихідний текст без необхідності використовувати спеціальні сутності для символів, таких як амперсанд, менший знак тощо. Однак, у всіх інших частинах документа, де можуть бути розташовані атрибути або інші елементи XML, потрібно використовувати ці сутності для уникнення конфліктів з розміткою.

Що стосується імен в XML, то вони мають чіткі правила. Імена елементів, атрибутів та інших об'єктів в XML повинні починатися з літери, символу виділення

«_» або двокрапки «:». Важливо також зазначити, що імена не можуть починатися з "XML", оскільки це зарезервоване для використання стандартом, що визначає саму мову XML. Більш того, через підтримку Юнікоду у XML можна використовувати не тільки символи ASCII, а й букви з інших мов, що робить XML універсальним для використання в глобальному середовищі.

Однією з основних переваг XML є його здатність зберігати великі обсяги структурованої інформації у зручному форматі. Наприклад, у контексті «розумного будинку» можна створити XML файл, що містить дані про споживання енергії, води та інших ресурсів протягом тижня. Оскільки XML є текстовим форматом, його можна легко передавати між різними пристроями, наприклад, з сервера на смартфон. Це зручно, оскільки дозволяє зберігати і обмінювати інформацію без значних витрат часу на запис.

Для того, щоб клієнтська частина програми змогла відновити необхідну інформацію з XML файлу, використовують процес, відомий як парсинг (або семантичний аналіз). Парсинг передбачає розбір XML файлу на складові його елементи, що дозволяє програмі точно інтерпретувати і відобразити дані у відповідному вигляді на екрані користувача. Це може бути важливою частиною розробки програмного забезпечення для розумних будинків, оскільки забезпечує можливість ефективно працювати з різноманітними типами даних, що постійно змінюються та оновлюються.

Таким чином, XML відіграє важливу роль в обміні інформацією та її збереженні у багатьох сферах, зокрема в Інтернеті речей (IoT), де пристрої часто передають та отримують дані у вигляді XML-файлів для подальшого аналізу та обробки.

1.1.4 Семантичний аналіз або парсинг XML-файлів

У сучасних технологіях обміну даними XML (Extensible Markup Language) є одним із найбільш популярних форматів для зберігання та передачі структурованої інформації. Його перевагою є здатність зберігати дані в ієрархічній, гнучкій структурі, що дозволяє легко адаптувати формат під різні потреби та типи даних.

Однак XML-документи, хоча й забезпечують високу гнучкість і універсальність, вимагають спеціальних методів для їх аналізу та обробки[8]. Це завдання здійснюється за допомогою парсерів — програм, що відповідають за «розбір» XML-документів, витягнення потрібної інформації і перевірку коректності структури документа.

Парсери для XML-документів є основними інструментами, що забезпечують автоматичне виконання синтаксичного і семантичного аналізу. Вони виконують дві ключові функції: перевірку документа на відповідність певним стандартам і схемам[7] (для гарантованої правильності даних) і забезпечення доступу до інформації в зручному для користувача вигляді. Ці програми здійснюють «чорну» роботу, відокремлюючи розробників від необхідності вручну виконувати складні операції з текстовими даними, зокрема перевіряти синтаксис або шукати елементи, атрибути та інші конструкції в документах.

Парсери XML можна поділити на кілька категорій, залежно від їх функціональності та специфікацій:

1. Верифікуючі парсери здійснюють перевірку документа на відповідність схемам, які визначають його структуру. Схеми можуть бути представлені у вигляді DTD (Document Type Definition) або XML Schema. Такі парсери гарантують, що XML-документ відповідає певним стандартам, що особливо важливо в розподілених системах, де необхідно забезпечити коректний обмін даними між різними компонентами системи.

2. Неверіфікуючі парсери просто обробляють документ, не перевіряючи його відповідність схемам. Вони розбирають XML-дані в реальному часі, не виконуючи додаткових перевірок, що робить їх швидшими і менш ресурсозатратними. Такі парсери зазвичай використовуються для обробки великих обсягів даних, де критичним фактором є швидкість.

Основний принцип роботи парсерів XML полягає в аналізі структури документа, який є текстовим, і виділенні з нього окремих елементів та атрибутів. Парсер обробляє кожен символ в документі по черзі, визначаючи його роль у структурі. Це дозволяє будувати дерево документів, що складається з різних

елементів, піделементів, атрибутів, значень тощо. Після цього парсер може надавати доступ до цих даних через спеціалізований інтерфейс.

Оскільки XML-документи можуть бути великими за розміром і містити велику кількість даних, парсери повинні бути максимально ефективними. Для цього використовуються різні методи парсингу:

- DOM (Document Object Model) парсер створює в пам'яті повне дерево документа, яке зберігає всю структуру XML. Це дозволяє легко маніпулювати даними та працювати з ними в реальному часі. Однак цей метод займає значні обсяги пам'яті і може бути неефективним при роботі з великими файлами.

- SAX (Simple API for XML) парсер працює за принципом подій. Він не зберігає весь документ в пам'яті, а обробляє його по частинах, викидаючи непотрібні елементи після їх обробки. Це дозволяє значно заощадити пам'ять, але ускладнює доступ до даних, оскільки немає можливості одразу маніпулювати всіма елементами документа.

- StAX (Streaming API for XML) парсер поєднує елементи DOM і SAX. Він дозволяє отримувати доступ до XML-документів у режимі потоку, зберігаючи лише частину даних в пам'яті, що робить його дуже ефективним для роботи з великими файлами.

Існує багато різних парсерів XML, які доступні для різних мов програмування та платформ. Одним з найбільш відомих і поширених є парсер Expat, розроблений Джеймсом Кларком. Цей парсер є open-source продуктом, написаним на мові C, і використовується в багатьох платформах, таких як PHP і Perl. Він забезпечує ефективний парсинг XML і є дуже популярним у багатьох розподілених системах завдяки своїй простоті та швидкодії.

Іншим відомим парсером є Xerces, який є частиною проекту Apache XML. Xerces підтримує не тільки мови програмування C++ і Java, але й багато інших. Цей парсер підтримує всі основні стандарти XML і включає в себе велику кількість додаткових функцій для роботи з документами, включаючи перевірку на відповідність схемам і підтримку різних версій XML.

На платформі Android для роботи з XML-документами використовуються специфічні парсери, такі як XMLPullParser, DOM Parser і SAX Parser. Ці парсери

розроблені для роботи з XML на мобільних пристроях і мають свої специфічні переваги та недоліки. Наприклад, XMLPullParser є дуже швидким і зручним для обробки великих файлів, але він вимагає більш складної реалізації, в порівнянні з DOM або SAX.

Парсери XML є важливими інструментами для мобільних додатків, особливо в контексті Інтернету речей (IoT). У таких системах необхідно обробляти великі обсяги даних з різних пристроїв і сенсорів, а XML є зручним форматом для зберігання та передачі таких даних. Використання парсерів дозволяє швидко і ефективно обробляти ці дані, забезпечуючи мобільні додатки необхідною інформацією для відображення або подальшої обробки.

У мобільних додатках для Android парсери XML використовуються для обробки даних, отриманих від різних пристроїв, наприклад, для відображення статистики споживання енергії, води або інших ресурсів у розумному будинку. Використовуючи парсери, можна зберігати ці дані у вигляді XML-файлів і відновлювати їх на клієнтському пристрої, що дозволяє зберігати високий рівень інтерактивності і гнучкості у додатках.

XML є дуже потужним і універсальним форматом для зберігання та передачі даних у сучасних розподілених системах. Однак для ефективної роботи з цими даними необхідно використовувати парсери, які виконують складний процес синтаксичного та семантичного аналізу XML-документів. Розробники можуть вибирати між різними типами парсерів, залежно від вимог до продуктивності та ресурсів. У мобільних додатках, особливо в контексті Інтернету речей, парсери XML забезпечують зручний та ефективний спосіб обробки даних, що робить їх невід'ємною частиною сучасних розподілених систем.

1.2. Виділення проблеми та визначення предмета дослідження

В умовах стрімкого розвитку Інтернету речей (IoT), ефективне зберігання та обробка даних є критично важливим завданням для мобільних додатків, які взаємодіють з пристроями IoT. Одним із найбільш поширених форматів для обміну даними між пристроями IoT є XML, оскільки цей формат забезпечує зручний, стандартизований спосіб представлення структурованої інформації. Однак,

незважаючи на свою популярність, робота з XML у мобільних додатках, зокрема на платформі Android, може стати викликом, якщо не використовувати оптимальні інструменти для парсингу та обробки великих обсягів даних.

У цьому контексті важливо звернути увагу на три основні парсери XML, які використовуються для обробки таких файлів у мобільних додатках на платформі Android і мові Java: DOMParser, XMLPullParser, SAXParser. Кожен з цих парсерів має свої унікальні характеристики, що визначають їх ефективність у різних сценаріях обробки даних, зокрема у додатках IoT.

1. DOMParser — цей парсер створює повну модель документа в пам'яті, що може бути дуже зручним для обробки XML-документів середнього розміру, коли потрібно активно працювати з даними. Однак у контексті IoT, де часто маємо справу з великими обсягами даних або з постійно змінюваними потоками інформації, використання DOMParser може призвести до значних витрат пам'яті, що негативно позначиться на ефективності роботи додатку, особливо на пристроях з обмеженими ресурсами.

2. XMLPullParser — цей парсер використовує подієвий підхід до обробки XML, що робить його значно більш ефективним для обробки великих файлів або даних, що надходять в реальному часі, що є звичним для пристроїв IoT. Завдяки своїй низькій витраті пам'яті та здатності працювати в режимі подій, він є ідеальним вибором для мобільних додатків IoT, де важлива швидка обробка даних без завантаження всього документа в пам'ять. Це особливо корисно, коли необхідно обробляти інформацію від багатьох пристроїв одночасно.

3. SAXParser — схожий на XMLPullParser, цей парсер також працює за допомогою подієвого механізму, що дозволяє ефективно обробляти великі обсяги даних. Однак, на відміну від XMLPullParser, SAXParser має певні обмеження, оскільки не дозволяє «повертатись» до попередніх елементів документа, що може обмежувати його зручність при роботі з більш складними структурами даних. Для IoT додатків, де структура даних може бути змінною і багаторівневою, це може стати проблемою, якщо потрібно здійснювати складні маніпуляції з інформацією.

Враховуючи особливості IoT, де обробка даних у реальному часі та ефективне використання ресурсів є важливими вимогами, розробникам необхідно

обирати парсер XML з урахуванням не лише швидкості та пам'яті, а й специфічних потреб додатка. Для IoT-мереж, де дані можуть надходити від сотень або тисяч пристроїв одночасно, ефективне використання парсерів XML, таких як XMLPullParser або SAXParser, може значно покращити продуктивність і знизити навантаження на пристрої, що працюють у мобільних додатках.

Завдання цього дослідження полягає в тому, щоб порівняти ці парсери з точки зору їх здатності обробляти дані IoT, забезпечити оптимальну обробку великих обсягів інформації та гарантувати збереження ефективності навіть за умов обмежених ресурсів. Це дозволить розробникам вибирати найкращі інструменти для створення високопродуктивних та надійних мобільних додатків, які взаємодіють з IoT-пристроями.

1.2.1 Огляд XMLPullParser

XMLPullParser є потужним інструментом для аналізу XML-документів, який активно використовується на платформі Android, зокрема в додатках, що працюють з великими обсягами даних, таких як додатки для Інтернету речей (IoT) [3]. Цей парсер використовує подієву модель для обробки XML, що робить його ефективним у сценаріях, де необхідна обробка даних у реальному часі з мінімальними вимогами до пам'яті. У контексті IoT, де пристрої генерують потоки даних, що швидко змінюються, ефективне управління пам'яттю та швидка обробка є критичними для забезпечення високої продуктивності додатків.

Принцип роботи XMLPullParser ґрунтується на проходженні через весь документ XML поетапно, зупиняючись на важливих елементах. Це дозволяє економно використовувати пам'ять, оскільки кожен елемент документа обробляється лише за потреби, а не весь документ зберігається в оперативній пам'яті одночасно. Таке поетапне оброблення робить XMLPullParser оптимальним вибором для мобільних додатків, де ресурси можуть бути обмеженими, а документи часто великі або складні.

XMLPullParser не аналізує весь XML-документ одразу, що дозволяє скоротити витрати пам'яті, забезпечуючи лише необхідну частину даних для

роботи. Кожен крок через документ здійснюється через метод next(), який дозволяє парсеру переходити до наступної події або елементу в документі[4]. При цьому кожен елемент, такий як START_TAG, TEXT або END_TAG, має свій власний статус, що дозволяє ефективно визначати структуру і вміст документа на кожному етапі.

Програміст може викликати next() в циклі, щоб пройти через документ і обробляти кожну подію. Наприклад, для того, щоб отримати текст з певного тегу, потрібно чекати, поки парсер не зупиниться на елементі TEXT, а потім отримати значення за допомогою getText().

Ключова особливість цього парсера — це можливість контролювати швидкість та ефективність обробки XML-даних, що є важливим для додатків в сфері IoT. Наприклад, пристрій у мережі IoT може генерувати потоки даних у вигляді XML-файлів, які містять параметри стану сенсорів або інформацію про події, що виникли на пристрої. Використання XMLPullParser дозволяє мобільним пристроям обробляти ці дані, не перевантажуючи систему пам'яті і підтримуючи оптимальний рівень споживання ресурсів.

Важливим моментом є те, що XMLPullParser дозволяє працювати з великими XML-документами в реальному часі. Це особливо корисно в контексті IoT, де часто потрібно обробляти великі масиви даних, що надходять від різних пристроїв або датчиків. Наприклад, датчики в розумному домі можуть постійно надсилати дані у вигляді XML-документів, що містять показники температури, вологості, стану безпеки тощо. Оскільки IoT-пристрої зазвичай працюють в умовах обмежених ресурсів, така економія пам'яті дозволяє використовувати XMLPullParser для обробки даних навіть на мобільних пристроях або мікроконтролерах з обмеженими можливостями.

Давайте розглянемо приклад використання XMLPullParser для обробки XML, що містить інформацію про показники датчиків. Уявімо, що ми отримуємо XML-документ, який містить дані про температуру і вологість:

```xml
<sensorData>
```

```
 <temperature>22.5</temperature>
 <humidity>60</humidity>
</sensorData>
```

Код для обробки цього документа за допомогою XMLPullParser може виглядати так:

```java
XmlPullParser parser = XmlPullParserFactory.newInstance().newPullParser();
InputStream inputStream = new FileInputStream("sensorData.xml");
parser.setInput(inputStream, null);

int eventType = parser.getEventType();
while (eventType != XmlPullParser.END_DOCUMENT) {
 String tagName = parser.getName();
 switch (eventType) {
 case XmlPullParser.START_TAG:
 if ("temperature".equals(tagName)) {
 String temperature = parser.nextText();
 System.out.println("Temperature: " + temperature);
 } else if ("humidity".equals(tagName)) {
 String humidity = parser.nextText();
 System.out.println("Humidity: " + humidity);
 }
 break;
 }
 eventType = parser.next();
}
```

У цьому прикладі ми використовуємо XMLPullParser для поетапного аналізу XML-документа. Кожен тег START_TAG перевіряється, і якщо це тег temperature

або humidity, ми отримуємо значення тексту з цих елементів і виводимо їх на консоль. Це дозволяє ефективно обробляти інформацію, що надходить від датчиків у реальному часі, і може бути корисним для мобільних додатків в IoT, де постійно генеруються нові дані від різних пристроїв.

Проте, важливо зазначити, що XMLPullParser має певні обмеження щодо швидкості обробки великих XML-документів. Оскільки він проходить через весь файл кілька разів, кожен пошук елемента потребує нового проходу через документ. Це може бути не так ефективно, коли потрібно обробляти величезні файли в реальному часі або у випадках, коли швидкість обробки є критично важливою для додатка.

Загалом, XMLPullParser є відмінним вибором для додатків IoT, де важлива економія пам'яті та ефективність обробки даних в реальному часі, але для великих даних або складних документів можуть бути потрібні додаткові оптимізації або інші парсери.

1.2.2 Огляд парсеру DOMParser

DOMParser — це парсер XML, який використовує об'єктно-орієнтований підхід для створення та аналізу XML-документів у Android-додатках. У порівнянні з іншими парсерами, такими як XMLPullParser, DOM парсер завантажує весь XML-документ у пам'ять і створює його об'єктну модель, що дозволяє програмісту працювати з документом як з набором об'єктів. Це може бути корисним, коли необхідно здійснити складні операції з документа, такі як зміна структури або створення нових елементів.

DOM парсер обробляє XML від самого початку (прологу) до кінця документа, включаючи всі елементи та атрибути. Він створює дерево, що містить усі вузли документа, включаючи елементи, атрибути, текст та інші частини XML. Це дерево дозволяє програмісту звертатися до конкретних частин документа для подальшої обробки.

Таблиця 1.3 надає огляд основних компонентів XML-документа, з якими працює DOM парсер:

Таблиця 1.3. Складові DOM parser

Компонент	**Опис**
Пролог	Як правило, файл XML почнеться з прологу. Перший рядок, що містить інформацію про файл - пролог.
Події	Файл XML буде містити багато подій, які включають початок і кінець документа, початок і закінчення тегів і т.д.
Текст	Це простий текст в елементах тегів xml.
Атрибути	Атрибути - це додаткові властивості тега, такі як значення і тд, які знаходяться в тегах.

DOM парсер створює внутрішню об'єктну модель документа. Це означає, що весь XML-документ завантажується в пам'ять як дерево об'єктів, що дозволяє маніпулювати документом безпосередньо через ці об'єкти. Після завантаження документа в пам'ять програміст може виконувати операції як на всьому документі, так і на окремих елементах або атрибутах.

Проте така структура має свої переваги та недоліки. Перевага DOM парсера полягає в тому, що він забезпечує простоту роботи з XML-документом, адже всю його структуру можна легко навігувати, змінювати, додавати нові елементи або атрибути. Це робить його ідеальним для сценаріїв, де необхідна гнучка робота з XML, наприклад, в програмуванні додатків для управління складними даними.

Однак, основним недоліком є те, що DOM парсер завантажує весь XML-документ у пам'ять. Це може призвести до значного споживання пам'яті та уповільнення роботи, особливо коли мова йде про великі документи. У контексті мобільних додатків на Android, де ресурси обмежені, таке завантаження може бути недоцільним, особливо в випадках, коли необхідно обробляти великі XML-файли або документи з динамічними даними, як це часто трапляється в додатках для Інтернету речей (IoT).

У випадку IoT, коли пристрої зберігають дані у форматі XML, важливо враховувати, що великі документи, наприклад, записи стану сенсорів з великою

кількістю елементів, можуть стати проблемою для парсера DOM через його потребу у великій кількості пам'яті[5]. У таких випадках використання XMLPullParser, що не завантажує весь файл у пам'ять, може бути більш ефективним.

Розглянемо приклад використання DOMParser для читання XML-файлу, який містить дані про температуру та вологість з сенсорів:

```xml
<sensors>
 <sensor>
 <type>temperature</type>
 <value>22.5</value>
 </sensor>
 <sensor>
 <type>humidity</type>
 <value>60</value>
 </sensor>
</sensors>
```

Код для аналізу цього документа за допомогою DOMParser в Android може виглядати так:

```java
try {
 FileInputStream inputStream = new FileInputStream("sensors.xml");
 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
 DocumentBuilder builder = factory.newDocumentBuilder();
 Document document = builder.parse(inputStream);
```

```
NodeList sensorNodes = document.getElementsByTagName("sensor");

for (int i = 0; i < sensorNodes.getLength(); i++) {
Node sensorNode = sensorNodes.item(i);
if (sensorNode.getNodeType() == Node.ELEMENT_NODE) {
Element sensorElement = (Element) sensorNode;
String type = sensorElement.getElementsByTagName("type").item(0).getTextContent();
String value = sensorElement.getElementsByTagName("value").item(0).getTextContent();

System.out.println(type + ": " + value);
}
}
} catch (Exception e) {
e.printStackTrace();
}
```

У цьому прикладі ми використовуємо DOMParser для завантаження XML-документа у пам'ять і далі працюємо з елементами, витягуючи значення температури та вологості кожного сенсора. Звертаючись до тегів за допомогою методів getElementsByTagName() та getTextContent(), ми можемо легко отримати значення, необхідні для подальшої обробки.

DOMParser є потужним інструментом для роботи з XML-документами, коли потрібно здійснювати складну обробку чи маніпулювання даними, що містяться в документі. Однак, через необхідність завантаження всього документа в пам'ять, його використання може бути обмеженим у випадках, коли необхідно працювати з великими файлами або коли важлива ефективність використання пам'яті, як це часто буває в додатках IoT. У таких випадках краще використовувати інші парсери, наприклад XMLPullParser, для оптимізації ресурсів.

1.2.3 Огляд SAXParser

SAXParser (Simple API for XML) є популярним підходом для аналізу XML-документів в Android-додатках, який працює за принципом обробки подій[8]. На відміну від DOMParser, який завантажує весь документ у пам'ять, SAX використовує більш ефективний метод, обробляючи документ по черзі, символ за символом. Цей підхід дозволяє зменшити споживання пам'яті, оскільки не потребує завантаження всього XML-файлу в оперативну пам'ять. SAX "зчитує" файл по частинах, генеруючи події, які потім обробляються відповідними методами.

Основною перевагою SAX є те, що він дозволяє зупиняти обробку на будь-якому етапі документа, при цьому не втрачаючи зібраної інформації. Це робить SAX дуже корисним для великих XML-документів або файлів, що містять великий обсяг даних, адже він ефективно працює з пам'яттю, обробляючи лише поточну частину файлу.

Під час роботи з SAX, кожен елемент XML-документа обробляється у вигляді окремої події: початок елемента, кінець елемента, або текст між тегами. Цей підхід дозволяє використовувати SAX для ситуацій, коли необхідно виконати лише просту лінійну обробку даних, наприклад, зчитувати певні значення з документа, без потреби маніпулювати структурою XML. Водночас SAX не дає змогу змінювати або додавати нові елементи до документа, що робить його менш гнучким порівняно з DOMParser, який створює повну структуру документа.

Ще одним важливим аспектом є те, що хоча SAX споживає менше пам'яті, він також має свої обмеження. Оскільки SAX працює з подіями, а не з повним деревом документа, він не дозволяє легко виконувати складні маніпуляції з даними, такі як змінення або додавання елементів. Крім того, як і інші XML-парсери, SAX також має обмеження на швидкість обробки великих файлів, оскільки йому потрібно пройти через весь документ, щоб обробити кожну подію.

Для обробки XML в реальному часі або для застосувань, де потрібно мінімізувати використання пам'яті, наприклад, в IoT-системах, де відбувається збирання великих обсягів даних від різних сенсорів, SAXParser є одним з найкращих варіантів. Його можливість обробляти дані без необхідності завантажувати весь файл в пам'ять робить його ідеальним для пристроїв з обмеженими ресурсами, таких як мобільні телефони або сенсорні мережі.

У кінцевому підсумку, SAXParser є потужним інструментом для обробки XML в Android-додатках, який відрізняється високою ефективністю пам'яті та здатністю обробляти великі документи. Однак він підходить лише для ситуацій, де потрібна лінійна обробка даних без потреби змінювати саму структуру документа. Для більш складних сценаріїв, де необхідно маніпулювати XML-документом, краще використовувати DOMParser.

1.2.4 Швидкість обробки великого обсягу даних – одна з найголовніших проблем при парсингу файлів

У випадку мобільних додатків, що працюють з великими обсягами даних, особливо з файлами XML, проблема швидкості обробки стає критично важливою. Прикладом може бути система управління інтелектуальним будинком, де сервер збирає дані протягом тижня і відправляє їх на мобільний девайс у вигляді XML-файлу. Оскільки розмір таких файлів може бути значним, а також важливо, щоб користувач не відчував жодних затримок у роботі програми, потрібно вирішити проблему повільної обробки даних.

У цьому контексті важливо розуміти, що зазвичай програмне забезпечення, особливо мобільні додатки, має вимоги до того, щоб обробка даних була швидкою, без затримок та блокування інтерфейсу. Це означає, що всі операції з обробки даних повинні виконуватись асинхронно або в окремих потоках, щоб не заважати користувацькому досвіду. В іншому випадку, велике завантаження даних, особливо з XML-файлів, може призвести до того, що інтерфейс стане "замороженим", а користувач не зможе взаємодіяти з програмою.

Основною проблемою при парсингу великих XML-файлів є саме час, що витрачається на обробку. Якщо XML-файл містить велику кількість елементів і має складну структуру, це може споживати значну кількість пам'яті та часу процесора, що викликає блокування програми або навіть її аварійне завершення. Особливо це стає помітним у мобільних додатках, де ресурси пристрою обмежені.

Для вирішення цієї проблеми необхідно вибирати відповідні стратегії обробки даних. Один із способів – це використання технологій, що дозволяють обробляти XML дані асинхронно, не блокуючи основний потік програми. Наприклад, замість завантаження всього файлу XML у пам'ять можна використовувати підхід поетапної обробки, при якому дані зчитуються та обробляються частинами.

Такі технології, як SAXParser, можуть бути корисними в таких ситуаціях, оскільки вони дозволяють обробляти дані по черзі, подія за подією, що дозволяє значно зменшити навантаження на систему. Інші методи, як асинхронна обробка або багатопотоковість, можуть допомогти у виконанні операцій паралельно з іншими задачами, не блокуючи інтерфейс програми.

Ще один підхід, який можна використовувати в мобільних додатках для підвищення швидкості обробки великих XML-файлів, — це попереднє збереження або кешування часто використовуваних даних. Наприклад, деякі дані можна зберігати в структурованому вигляді (наприклад, в базі даних або в спеціальному кешованому файлі), щоб уникнути необхідності обробляти великий файл XML кожного разу при завантаженні.

Крім того, з точки зору UX (User Experience) важливо, щоб програма завжди надавала користувачеві зворотний зв'язок у разі затримок, наприклад, шляхом відображення прогрес-барів чи сповіщень про стан обробки даних, що дозволить уникнути негативного враження від програми.

Загалом, ключовим моментом є правильний вибір стратегії обробки великих XML-файлів та ефективне використання ресурсів пристрою. Використання таких методів, як асинхронна обробка, багатопоточність, або спеціалізовані парсери на основі подій, допоможе забезпечити високу швидкість та ефективність обробки даних у мобільних додатках без блокування основного інтерфейсу користувача.

1.2.5 Багатопоточне програмування для оптимізації обробки даних

З розвитком багатоядерних і багатопроцесорних систем програмістам постало питання: як найбільш ефективно розподілити навантаження і створити паралельне виконання програмного коду? Ця проблема не є новою, однак з появою багатоядерних процесорів вона набуває нового значення. Від того, як ми організуємо паралельну обробку, залежить ефективність виконання складних задач, таких як парсинг великих обсягів даних.

У найпростішому вигляді, процес у комп'ютерних системах можна уявити як набір інструкцій і даних, які виконуються у вигляді окремих потоків. Зазвичай одна програма складається з одного процесу, однак є випадки, коли розподіл на кілька процесів дозволяє досягти значних переваг, як-от у веб-браузерах, де кожна вкладка працює у своєму окремому процесі, що підвищує стабільність роботи та незалежність між вкладками.

Потоки — це одиниці виконання коду, які можуть працювати паралельно[2]. Однак є важлива особливість, пов’язана з архітектурою процесорів. Кожен процесор або його ядро може одночасно виконувати лише один потік. Отже, навіть якщо система має кілька ядер, кожен потік виконується на одному ядрі в один момент часу. Водночас, навіть у одноядерних процесорах можлива симуляція паралельного виконання завдяки так званому «псевдопаралелізму», коли система перемикається між потоками, обробляючи їх по черзі. Це дозволяє здаватися, що програма виконується одночасно на кількох потоках, хоча насправді потоки обробляються один за одним.

Застосування багатопоточності в програмуванні дає суттєві переваги, особливо при обробці великих даних. Наприклад, у випадку парсингу великих XML-файлів, багатопоточність дозволяє паралельно обробляти різні частини документа. Це суттєво скорочує час, необхідний для обробки, оскільки кожен потік може здійснювати обчислення і парсинг своєї частини файлу, не чекаючи на інші потоки. Така організація роботи дозволяє значно підвищити ефективність і

швидкість обробки даних, особливо в мобільних додатках, де обмеження на ресурси — пам'ять і процесорну потужність — є критичними.

Для цього важливо правильно налаштувати багатопоточність. Наприклад, XML-файл можна розділити на кілька частин, кожну з яких буде обробляти окремий потік. Після завершення обробки всі частини об'єднуються для отримання цілісного результату. Така схема дозволяє оптимізувати процес і максимально ефективно використовувати ресурси системи.

Застосування багатопоточного програмування в контексті парсингу XML-файлів має важливе значення для досягнення високої швидкості обробки великих обсягів даних. Особливо це важливо для мобільних пристроїв, де часто необхідно обробляти великий обсяг даних за короткий час, не перевантажуючи систему. Використовуючи багатопоточність, можна мінімізувати час очікування користувача, поліпшити взаємодію з інтерфейсом і забезпечити швидку та ефективну обробку даних. Тому багатопоточність стає важливим інструментом для розробки ефективних і швидких додатків, здатних працювати з великими обсягами інформації.

1.3 Постановка задачі

Завдання, що стоїть перед нами, полягає в розробці інструментів та методів для ефективної обробки даних Інтернету Речей (IoT) у середовищі Java та ОС Android, зокрема в контексті обробки великих обсягів XML-файлів, що містять дані від IoT-пристроїв[3]. Для цього необхідно створити програму, яка наочно продемонструє час, необхідний для обробки таких великих файлів XML, використовуючи різні методи обробки в мобільних додатках. Після цього буде проведено серію тестів, порівняння результатів і розроблено алгоритм для вибору найбільш оптимальних методів в залежності від умов, таких як розмір файлу, ресурсна потужність пристрою та типи використовуваних IoT-даних.

У цьому контексті, важливо визначити, як ефективно обробляти та аналізувати дані від IoT-пристроїв, зокрема великі потоки даних, що надходять в

режимі реального часу. Потрібно розглянути, як забезпечити зручність і швидкість обробки цих даних в мобільних додатках, не затримуючи роботу користувацького інтерфейсу та зберігаючи високу продуктивність навіть при великих обсягах даних.

Предметом дослідження є методи та інструменти обробки даних IoT з використанням багатопотоковості та оптимізації роботи з великими XML-файлами у мобільних додатках на базі Java та Android. Об'єктом дослідження стане обробка даних IoT в умовах обмежених ресурсів мобільних пристроїв і необхідність використання багатопоточних технологій для швидкої та ефективної обробки великих обсягів даних.

Проблема полягає в тому, щоб визначити найбільш ефективні методи обробки великих обсягів даних від IoT-пристроїв, які часто надходять у вигляді XML-файлів або потоків даних, з урахуванням обмежених ресурсів мобільних пристроїв (наприклад, пам'ять, обчислювальна потужність). Це вимагає інтеграції технологій для швидкої обробки таких даних без блокування користувацького інтерфейсу, з одночасним використанням багатопоточних підходів для розподілу навантаження.

У рамках цього дослідження пропонується наступний сценарій для розробників IoT-додатків:

1. Збір та передача даних IoT. IoT-пристрої протягом певного часу (наприклад, день чи тиждень) збирають різноманітні дані (наприклад, температурні показники, вологість, рух, енергоспоживання) та формують XML-файли для передачі їх на мобільний пристрій.

2. Обробка даних XML. Мобільний додаток отримує ці файли для подальшої обробки та аналізу. Через великий обсяг даних, обробка може займати значний час і навіть блокувати основний інтерфейс програми. Завдання полягає в тому, щоб ефективно обробити ці великі файли даних, не затримуючи користувача в процесі взаємодії з додатком.

3. Використання багатопоточності для обробки IoT-даних. Для обробки великих XML-файлів, що містять дані IoT, необхідно застосувати багатопоточні

технології. Це дозволить паралельно обробляти різні частини файлу, що значно зменшить час обробки та підвищить загальну ефективність.

4. Тестування та оптимізація. Після реалізації кількох методів обробки XML-документів в мобільному додатку, буде проведено порівняння ефективності кожного з них в умовах реального часу. Це дозволить вибрати оптимальний метод обробки для конкретного випадку (наприклад, залежно від кількості даних, типу пристрою та його ресурсних обмежень).

Таким чином, основною метою цього дослідження є розробка інструментів для обробки даних IoT у мобільних додатках, де великі XML-файли, що містять дані від різноманітних сенсорів і пристроїв, обробляються ефективно з використанням багатопотоковості та оптимізації часу обробки.

2. АНАЛІЗ ПРОБЛЕМ ШВИДКОГО СЕМАНТИЧНОГО АНАЛІЗУ ВЕЛИКОГО ОБСЯГУ ДАНИХ

2.1 Багатопоточність в обробці даних

Багатопоточність є важливою технологією, що дозволяє значно підвищити продуктивність при обробці великих обсягів даних, таких як дані з Інтернету речей (IoT) або великі XML-файли [3]. Вона дає змогу одночасно виконувати кілька потоків обробки даних в межах одного процесу, що значно скорочує час виконання складних або великих обчислювальних задач.

Існує кілька моделей багатопоточності, кожна з яких має свої переваги і недоліки в залежності від сценаріїв застосування. Одна з найбільш поширених моделей — це паралельне виконання, коли кожен потік обробляє свою частину даних одночасно. Цей підхід дозволяє значно зменшити час обробки великих обсягів інформації, що особливо актуально для таких завдань, як аналіз потокових даних з IoT-пристроїв. У свою чергу, для систем з обмеженими обчислювальними ресурсами, наприклад, одноядерними процесорами, може бути застосований псевдопаралелізм. У цьому випадку система чергує виконання потоків, що дає можливість ефективно використовувати ресурси, хоча справжньої паралельності не досягається. Такий підхід дозволяє обробляти кілька задач, хоча й не одночасно, а на переривах між ними.

Іншою популярною моделлю є модель акторів, де кожен потік є окремим об'єктом, що обробляє свої дані незалежно від інших. Цей підхід дозволяє уникнути проблем з синхронізацією, оскільки кожен актор працює самостійно, обробляючи свою частину завдання без взаємодії з іншими потоками[2].

Залежно від конкретного сценарію застосування, багатопоточність може бути корисною для обробки великих обсягів даних, таких як інформація з датчиків IoT. Наприклад, для обробки XML-файлів, що містять дані від різних сенсорів (температура, вологість, тиск), кожен потік може обробляти різні частини цього файлу. Це дає змогу значно прискорити процес аналізу і мінімізувати затримки у системах, де дані надходять безперервно. Крім того, багатопоточність дозволяє

ефективно управляти асинхронними завданнями, такими як завантаження даних або виконання фонових операцій, не блокуючи основний потік програми.

Однак, застосування багатопоточності має і свої виклики. Одним з основних є правильна синхронізація між потоками, адже якщо кілька потоків одночасно звертаються до спільних ресурсів, можуть виникнути конфлікти. Крім того, збільшення кількості потоків може призвести до значного навантаження на систему, особливо в умовах обмежених ресурсів, таких як пам'ять або процесорна потужність. Іншим викликом є складність налагодження багатопоточних програм, адже не завжди можна легко відтворити умови для відтворення помилок, що виникають лише в специфічних ситуаціях.

Для реалізації багатопоточності в різних програмних середовищах використовуються різні інструменти. У Java, наприклад, для створення багатопоточних програм можна використовувати класи, такі як `java.lang.Thread` або більш складні конструкції, такі як `ExecutorService` чи `ForkJoinPool`, які допомагають керувати потоками та виконувати паралельні завдання. У середовищі Android для обробки багатопоточних завдань використовуються інструменти, як-от `AsyncTask`, `HandlerThread` або `ExecutorService`, які дозволяють розподіляти завдання між різними потоками, не блокуючи основний інтерфейс користувача.

Водночас для обробки великих обсягів даних в реальному часі, зокрема при аналізі даних IoT, можуть бути застосовані спеціалізовані платформи, такі як Apache Kafka або Akka, які дозволяють управляти потоками даних і ефективно обробляти їх паралельно, знижуючи час затримки та забезпечуючи високу масштабованість системи.

Таким чином, багатопоточність є потужним інструментом для обробки великих обсягів даних, що дозволяє значно підвищити ефективність роботи систем, таких як IoT-платформи, де швидкість обробки даних має критичне значення. Однак важливо правильно вибирати модель багатопоточності та синхронізувати потоки для досягнення максимального результату при мінімальних витратах на ресурси.

2.1.1 Огляд поняття багатопоточності в контексті IoT

У сучасних програмних системах важливу роль відіграють методи багатозадачності, зокрема, багатопоточність.[3] Вона дозволяє ефективно використовувати ресурси, знижувати час затримки та підвищувати продуктивність. Особливо це важливо для Інтернету речей (IoT), де одночасно здійснюється обробка великих обсягів даних від численних пристроїв і сенсорів. Від правильного використання багатопоточності залежить здатність системи обробляти кілька запитів одночасно без затримок, що критично для своєчасного отримання й обробки інформації.

У традиційних однопоточних системах процеси виконуються по черзі, що може стати суттєвим обмеженням у швидкості виконання. Для вирішення цієї проблеми застосовуються різноманітні підходи до багатопоточності, що дозволяють одночасно виконувати кілька завдань, знижуючи загальний час виконання програми.

У синхронній однопоточній моделі, всі завдання обробляються однією одиничною програмною ланкою. Цей підхід є простим і ефективним для маленьких або однотипних завдань, однак він не є оптимальним для обробки паралельних запитів у великих системах, зокрема, для IoT, де можуть надходити численні запити від різних пристроїв одночасно. Водночас однопоточність дозволяє уникнути проблем із синхронізацією та взаємодією потоків, оскільки всі операції виконуються по черзі.

Для того щоб краще зрозуміти особливості цієї моделі в контексті IoT, розглянемо Рис. 2.1, який демонструє, як виглядає процес виконання завдань у такій моделі.

Рисунок 2.1 – Сценарій однопоточності

У такому режимі, кожен запит обробляється строго по черзі, що може призводити до затримок при великому навантаженні або великій кількості одночасних запитів від пристроїв IoT.

У багатопоточній моделі система використовує кілька потоків для одночасного виконання кількох завдань. Це дозволяє значно зменшити час, необхідний для виконання великої кількості операцій, оскільки кожен потік може бути призначений для обробки окремого завдання або операції. Для систем IoT це має велике значення, оскільки дозволяє одночасно обробляти дані з багатьох датчиків або пристроїв без затримок.

У Рис. 2.2 зображено, як виглядає багатопоточний підхід у порівнянні з однопоточним. Паралельна обробка запитів дозволяє швидше реагувати на зміни, що є важливим для пристроїв IoT, де час реакції може мати критичне значення.

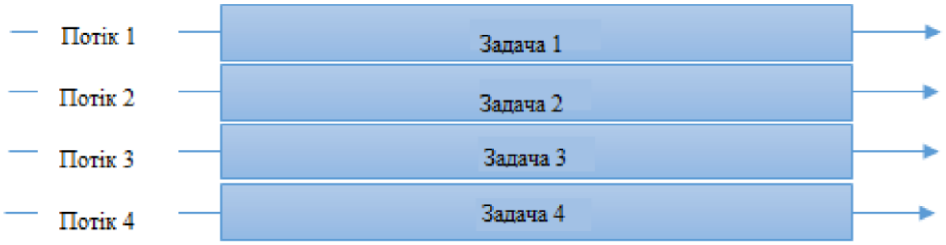


Рисунок 2.2 – Сценарій багатопоточності

Завдяки багатопоточності, система може динамічно адаптуватися до різних навантажень, адже завдання можуть виконуватися незалежно від інших. Проте варто зазначити, що за такої моделі може виникати проблема синхронізації між потоками, оскільки їх взаємодія потребує певного контролю для уникнення гонок даних.

Асинхронне програмування дозволяє значно покращити ефективність виконання задач у середовищах з обмеженими ресурсами, таких як IoT. У цьому режимі потік не чекає на завершення одного завдання перед тим, як перейти до іншого. Як тільки задача була ініційована, потік може перейти до наступного завдання, не чекаючи результатів попереднього. Це дозволяє зменшити час простоя потоку і значно підвищити загальну продуктивність системи.

На Рис. 2.3 показано, як виглядає процес асинхронного виконання завдань у однопоточному середовищі. Кожне завдання виконується паралельно, але всі вони здійснюються в одному потоці.

Рисунок 2.3 – Сценарій асинхронної однопоточної моделі

Цей підхід особливо ефективний для обробки великої кількості однотипних запитів, таких як зчитування показників із сенсорів IoT, де кожне завдання є незалежним і не потребує миттєвого завершення для переходу до наступного. Ідеальний сценарій для обробки великої кількості паралельних запитів від пристроїв IoT — це асинхронний багатопотоковий режим. У цьому випадку система використовує кілька потоків для обробки різних завдань, кожен з яких може працювати асинхронно. Цей підхід дає можливість максимізувати використання доступних ресурсів і одночасно зменшити час затримки для користувачів або зовнішніх пристроїв.

Рис. 2.4 показує, як виглядає багатопотокова асинхронна модель. Кожен потік обробляє своє завдання, і жоден потік не блокує виконання інших. Це дозволяє ефективно використовувати доступні ресурси і швидко реагувати на зовнішні запити.

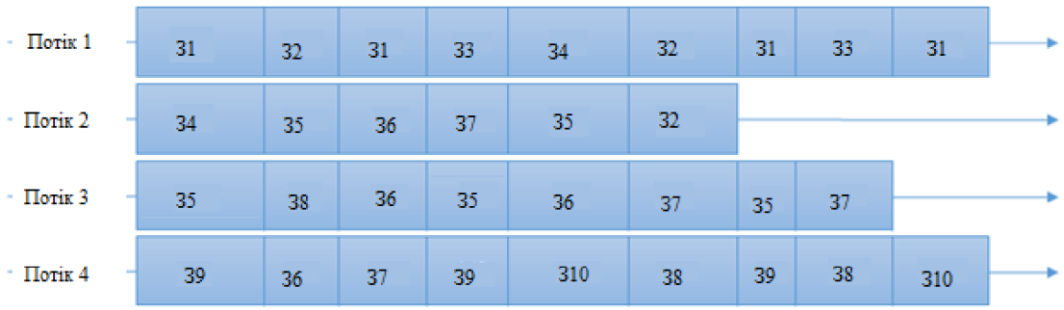


Рисунок 2.4 – Сценарій асинхронної багатопоточності

Особливо важливим є те, що за такої моделі всі потоки можуть обробляти запити в реальному часі, що є необхідним для IoT-систем, де дані надходять від численних пристроїв без затримок.

Паралелізм є ключовим елементом для забезпечення високої продуктивності в системах з великим навантаженням, таких як Інтернет речей. У таких системах численні пристрої можуть одночасно генерувати великі обсяги даних, які потребують швидкої обробки. Завдяки паралелізму можна обробляти ці дані за допомогою кількох потоків або навіть на різних фізичних пристроях.

Паралелізм дозволяє значно зменшити час затримки при виконанні задач, оскільки кожне завдання може бути оброблене незалежно від інших. Цей підхід оптимальний для систем, що мають справу з великими потоками даних, наприклад, у розумних містах, сільському господарстві або автоматизації промислових процесів, де дані збираються від численних сенсорів і пристроїв.

У контексті IoT-систем використання багатопоточності та асинхронного програмування має важливе значення для забезпечення ефективності й швидкості обробки даних. Кожен підхід має свої переваги та недоліки, але в комбінації вони дозволяють побудувати більш потужні та масштабовані системи, здатні працювати в реальному часі і обробляти величезні обсяги даних. Технології паралельної обробки, багатопоточності та асинхронних операцій є основою для ефективного функціонування сучасних IoT-систем.

2.1.2 Паралельне програмування

Паралельне програмування є основою для побудови ефективних і швидких обчислювальних систем, де кілька процесів можуть виконуватися одночасно, спільно вирішуючи одну задачу. Паралельна програма — це сукупність взаємодіючих паралельних процесів, що дозволяють обробляти різні частини задачі одночасно, значно прискорюючи виконання. Основна мета паралельних обчислень полягає в прискоренні вирішення обчислювальних проблем, що особливо актуально для систем, що обробляють великі обсяги даних, таких як IoT.[5]

Особливості паралельних програм:

• Управління роботою множини процесів: у паралельних програмах одночасно працює кілька процесів, кожен з яких виконує свою частину завдання.

• Обмін даними між процесами: для коректної роботи паралельних процесів необхідно організувати ефективний обмін даними між ними.

• Втрата детермінізму: асинхронність доступу до даних може призвести до непередбачуваних результатів , що ускладнює відлагодження паралельних систем.

• Нелокальні та динамічні помилки: в паралельних системах можуть виникати помилки, що не можна передбачити на етапі проектування.

• Тупикові ситуації: у разі некоректного планування паралельних процесів, можуть виникати ситуації, коли потоки блокуються, очікуючи один на одного (наприклад, у разі взаємного блокування).

• Масштабованість і балансування: з ростом кількості процесів у системі виникають труднощі з масштабованістю і рівномірним розподілом навантаження між обчислювальними вузлами.

У паралельних обчисленнях існують операції, які не можна виконувати паралельно, наприклад, операції введення/виведення або взаємодія з користувачем. Однак є й операції, які можна виконувати одночасно, наприклад, обробка даних із різних сенсорів у IoT-системах. Для таких задач використовується модель, в якій частка операцій, що можуть бути виконані паралельно, позначається як α (паралельність).

Час виконання послідовного алгоритму позначається як T1, а час виконання паралельної версії алгоритму на P процесорах — як T(P). Для оцінки ефективності паралельного алгоритму використовується співвідношення прискорення:

$$S(P) = \frac{T_1}{T(P)}$$

Де:

- $S(P)$ — прискорення паралельного алгоритму на P процесорах,
- T_1 — час виконання алгоритму на одному процесорі (послідовна версія),
- $T(P)$ — час виконання алгоритму на P процесорах (паралельна версія).

Це співвідношення показує, на скільки разів швидше працює паралельна версія алгоритму у порівнянні з послідовною.

Однак неефективне використання багатьох процесів може призвести до зниження ефективності паралельної програми. Ефективність паралельного алгоритму визначається як:

$$E(P) = \frac{S(P)}{P} = \frac{\frac{T_1}{T(P)}}{P} = \frac{T_1}{P \cdot T(P)}$$

Де:

- $E(P)$ — ефективність паралельного алгоритму,
- $S(P)$ — прискорення,
- P — кількість процесорів,
- T_1 — час виконання послідовного алгоритму,
- $T(P)$ — час виконання алгоритму на P процесорах.

Для оцінки максимального можливого прискорення, до якого може привести паралельний алгоритм, використовується закон Амдала:

$$S_{\max}(P) = \frac{1}{(1-f) + \frac{f}{P}}$$

Де:

- $S_{\max}(P)$ — максимальне можливе прискорення при використанні P процесорів,
- f — частка алгоритму, яка може бути виконана паралельно (від 0 до 1),
- $1 - f$ — частка алгоритму, яка виконується послідовно,
- P — кількість процесорів.

Згідно з цим законом, якщо лише 10% операцій алгоритму не можуть виконуватися паралельно, то максимальне прискорення паралельного виконання не перевищить 10 разів, навіть при дуже великій кількості процесорів. Це важливо

враховувати при розробці IoT-систем, де значна частина обробки даних повинна бути виконана в реальному часі, і надмірне використання процесів може призвести до зниження загальної продуктивності.

2.2 Інструменти та бібліотеки для багатопоточності в Android-розробці.

Для розробки програм для Android існує кілька основних бібліотек та інструментів, що допомагають працювати з багатопоточністю та асинхронними операціями [4]. Це дозволяє ефективно обробляти численні одночасні запити та процеси, що характерно для IoT-додатків.

Основні бібліотеки для роботи з багатопоточністю:

1. Java Concurrency API: у Java для роботи з багатопоточністю є потужна бібліотека java.util.concurrent, що дозволяє створювати багатопоточні додатки з високою продуктивністю. Вона включає в себе механізми для синхронізації потоків, планування задач і обміну даними між потоками. Це особливо корисно для IoT-систем, де потрібно швидко обробляти дані, що надходять від численних пристроїв.

2. AsyncTask: хоча цей клас більше не є рекомендованим в нових версіях Android, він використовувався для виконання асинхронних операцій, таких як виконання задач у фоновому режимі, що дозволяло не блокувати основний інтерфейс додатка [11]. Для IoT-додатків, які потребують частих оновлень і взаємодії з користувачем, ця бібліотека була корисною.

3. Executor Framework: цей фреймворк дозволяє керувати пулом потоків і запускати задачі в асинхронному режимі. Він підтримує різні стратегії планування задач і є основою для багатьох сучасних додатків для Android. Для IoT-систем, де часто необхідно обробляти запити від багатьох пристроїв одночасно, це є надзвичайно важливим інструментом.

Ці бібліотеки дозволяють ефективно керувати потоками і виконувати обчислення, що є необхідним для побудови масштабованих та ефективних IoT-додатків, які працюють з великою кількістю паралельних запитів від різних пристроїв.

2.2.1 Огляд Threads and Locks

У системах з одним процесором потоки виконуються по черзі, але швидке перемикання між ними створює ілюзію паралельного або одночасного виконання задач. Це досягається завдяки механізмам переключення контексту [6]. На мобільних пристроях або у IoT-системах, де є лише один фізичний процесор, кілька потоків можуть виконуватися одночасно через механізм контекстного перемикання. Це дає змогу системам з обмеженими ресурсами виконувати кілька задач паралельно, ефективно використовуючи їх потенціал.

Рисунок 2.5 ілюструє схему виконання трьох потоків в одному процесі, де кожен потік виконує свою частину роботи, при цьому система швидко перемикається між ними.

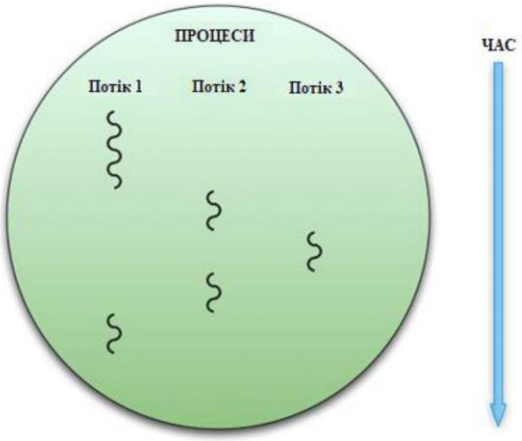


Рисунок 2.5 – Виконання 3-х потоків, схема

Потоки дозволяють ефективно використовувати ресурси системи, виконуючи кілька задач одночасно без блокування однієї задачі іншою. У IoT-системах, де одночасно обробляються численні запити від різних пристроїв, використання потоків є критично важливим для досягнення високої ефективності.

Наприклад, у IoT-додатках, таких як система моніторингу здоров'я, можуть одночасно здійснюватися кілька операцій: збирання даних з різних сенсорів, їх обробка, передачу на сервер і відображення інформації на екрані. У такій ситуації використання потоків дозволяє виконувати ці операції паралельно, що значно покращує відгук системи і зменшує час очікування.

Потоки можна використовувати в різних сценаріях, де тривалі операції не повинні блокувати виконання інших задач. Наприклад, у музичному плеєрі з кнопками відтворення і паузи, ви можете натискати кнопку паузи навіть тоді, коли музика відтворюється у фоновому потоці. Якщо для кожної операції створити окремий потік, ви зможете забезпечити одночасне виконання кількох задач без блокування одна одної.

У IoT-системах, де пристрої можуть генерувати великі обсяги даних і потребують швидкої обробки, наявність потоків дозволяє оптимізувати процеси обробки інформації. Наприклад, датчики можуть відправляти дані в реальному часі в декілька потоків, що дозволяє швидко реагувати на зміни стану середовища чи обладнання. Це особливо важливо у системах реального часу, де затримка може призвести до серйозних наслідків.

Однак при роботі з потоками можуть виникати конкурентні проблеми [8]. Це ситуації, коли кілька потоків намагаються одночасно змінити одні й ті самі дані, що може призвести до непередбачуваних результатів. У таких випадках для синхронізації потоків і уникнення помилок використовуються замки (Locks).

Замки дозволяють заблокувати частину коду або ресурс, щоб тільки один потік міг виконувати певну операцію в певний момент часу. Наприклад, у IoT-додатках, де кілька пристроїв одночасно звертаються до спільної бази даних, замки можуть забезпечити, щоб лише один потік одночасно вносив зміни в базу, запобігаючи конфліктам. Це може бути особливо корисно, коли кілька потоків намагаються одночасно змінювати налаштування пристроїв або дані, що можуть бути критичними для стабільної роботи всієї системи.

У IoT-системах замки використовуються для забезпечення цілісності даних та належної синхронізації потоків. Наприклад, у медичних пристроях, що збирають дані про стан пацієнта, важливо, щоб жоден з потоків не змінив важливу інформацію, поки інший потік її обробляє або передає на сервер.

2.2.2 Огляд RxJava

Перед тим, як заглибитися в механізми RxJava, варто згадати основи моделі спостерігача [5]. В цій моделі є об'єкт, який генерує події, та інші об'єкти, які підписуються на ці події і отримують їх. Цю модель ви, ймовірно, використовуєте щодня на роботі. Простим прикладом є кнопка з функцією натискання, де натискання на кнопку викликає певну подію, яку можуть спостерігати інші елементи системи.

В Java існують вбудовані інструменти для реалізації цієї моделі, такі як клас Observable та інтерфейс Observer. Реалізація інтерфейсу Observer визначає об'єкт, який чекає події від об'єкта Observable, що генерує події.

Те ж саме відбувається і в RxJava, де використовується схожа термінологія, але концепція «події» була значно розширена. У RxJava Observable — це об'єкт, який генерує потік даних, а Observer — це об'єкт, який підписується на цей потік, отримуючи всі події від Observable. У цьому випадку подія не обмежується простим сповіщенням, а включає в себе передачу даних через потік.

У RxJava події, які передає Observable і отримує Observer, можуть мати три основні типи:

1. Next — нова порція даних, яка передається в потік.
2. Error — повідомлення про помилку, яке виникло під час обробки потоку.
3. Completed — сигнал, що потік завершений і більше не буде нових даних.

У IoT-системах RxJava можна використовувати для обробки потоків даних від сенсорів або пристроїв, що працюють в реальному часі. Кожен сенсор або пристрій може бути представлений як Observable, який генерує потік даних. Інші компоненти системи, наприклад, сервіс збору або обробки даних, підписуються на ці потоки як Observer.

Це дозволяє зручно обробляти дані в асинхронному режимі, підтримуючи масштабованість і ефективність обробки численних подій і даних. Наприклад, у розумних будинках, де багато сенсорів збирають інформацію про температуру, вологість, рух або рівень освітленості, RxJava дозволяє ефективно обробляти ці потоки даних у реальному часі. Кожен сенсор може генерувати потік подій, який потім обробляється іншими частинами системи без блокування основних процесів.

Переваги використання RxJava в IoT:

1. Асиметрична обробка подій: Використання потоків дозволяє обробляти події незалежно від інших операцій, що критично важливо для ефективності в реальному часі.

2. Масштабованість: RxJava дозволяє створювати масштабовані додатки, які здатні ефективно обробляти численні події, зберігаючи високу продуктивність навіть при великій кількості датчиків або пристроїв.

3. Чистота коду: Завдяки декларативному підходу RxJava дозволяє писати чистий і зрозумілий код для обробки складних потоків даних без необхідності вручну управляти асинхронними операціями, що спрощує розробку і тестування додатків для IoT.

4. Обробка помилок: Оскільки у RxJava є механізм обробки помилок через події типу Error, система може оперативно реагувати на несправності або помилки у потоці даних, що критично для систем реального часу, де відмови або затримки можуть призвести до серйозних наслідків.

Отже, RxJava — це потужний інструмент для розробки асинхронних, реактівних додатків у контексті IoT, що дозволяє обробляти складні потоки даних з різних джерел, одночасно зберігаючи ефективність і масштабованість системи.

2.2.3 Огляд Kotlin Coroutines

Коротіни (coroutines) — це потужний інструмент для спрощення асинхронного програмування, який дозволяє працювати з асинхронними задачами без необхідності управляти потоками вручну [9]. Вони значно спрощують реалізацію асинхронного коду, приховуючи складність у бібліотеках, що забезпечує більш простий і чистий синтаксис порівняно з традиційними методами.

Ідея коротинів бере свій початок ще в 1967 році в мові програмування Симула, де було запропоновано кілька методів для зупинки та відновлення виконання задач. З часом ідея була забута через поширення використання потоків, але, як виявилося, коротіни мають значні переваги в певних випадках, особливо при виконанні асинхронних операцій. Вони дозволяють зберігати високу

ефективність, не блокуючи потоки, на відміну від традиційного підходу з потоками, де кожен асинхронний процес запускається в окремому потоці.

Розробники описують коротіни так: "коротіни — це обчислення, які можуть бути зупинені та відновлені без блокування потоку". Це означає, що замість того, щоб блокувати потік до завершення певної операції, коротіна може бути тимчасово зупинена, а потім відновлена без втрати контексту виконання. В результаті, система не втрачає ресурсів, і потік не залишається заблокованим, що значно підвищує ефективність і зменшує навантаження на систему.

Що означає «не блокувати потік»? Простіше кажучи, це означає, що коротіна може бути зупинена в процесі виконання і відновлена з того ж самого місця без того, щоб весь потік зупинявся. Таким чином, асинхронні операції можуть виконуватись без необхідності створення окремих потоків для кожної операції, що робить програму більш економною та ефективною.

Переваги використання Kotlin Coroutines:

1. Менше витрат на управління потоками: Створення та використання коротинів значно легше, ніж створення нових потоків для кожної асинхронної задачі, що зменшує витрати на ресурси системи.

2. Неблокуюче виконання: Коротіни дозволяють виконувати асинхронні операції без блокування основного потоку, що є важливим для програм, що потребують високої відгукності.

3. Простота коду: Використання коротинів дозволяє писати асинхронний код у лінійній, синхронній манері, що значно полегшує розуміння та підтримку коду порівняно з традиційними методами обробки асинхронних задач через зворотні виклики чи інтерфейси.

4. Ідеально підходить для багатозадачності: Коротіни дозволяють одночасно виконувати кілька асинхронних задач без необхідності створювати нові потоки для кожної задачі, що робить їх ідеальними для обробки численних операцій у фоновому режимі.

У IoT-додатках, де часто потрібно працювати з великою кількістю пристроїв або сенсорів, що генерують дані в реальному часі, коротіни можуть бути корисними для ефективної обробки таких даних. Наприклад, обробка даних з

сенсорів може відбуватися асинхронно, дозволяючи системі обробляти нові дані, не блокуючи виконання інших задач. Коротіни дозволяють легко управляти таким обробленням без зайвих витрат на управління потоками, що важливо для мобільних пристроїв або пристроїв з обмеженими ресурсами. Завдяки своїй гнучкості та простоті, Kotlin Coroutines є ефективним інструментом для розробки додатків, що працюють з великою кількістю паралельних завдань, зокрема в контексті IoT, де важлива висока ефективність і низьке споживання ресурсів.

2.3 Деталізація постановки задачі

Аналізуючи існуючі бібліотеки для роботи з багатопотоковою обробкою в контексті Інтернету речей (IoT), а також розглядаючи концепції паралельного програмування та асинхронності, постановка задачі стає ще більш актуальною. Оскільки пристрої в екосистемах IoT часто мають обмежені ресурси, а також потребують обробки великих обсягів даних в реальному часі, ефективна реалізація багатопоточності та асинхронних операцій є ключовим аспектом для забезпечення продуктивності і зниження затримок в таких системах.

1. Аналіз бібліотек для багатопотокової роботи в IoT

Потрібно провести детальний аналіз бібліотек, таких як RxJava, Kotlin Coroutines та Threads, з урахуванням специфіки IoT-систем. Оскільки IoT-пристрої можуть мати обмежені обчислювальні потужності та пам'ять, важливо визначити, як кожна з цих бібліотек може бути адаптована для ефективної обробки багатопоточних задач, наприклад, обробки великих потоків даних з сенсорів або виконання асинхронних операцій для інтеграції з хмарними сервісами.

2. Розробка уніфікованої бібліотеки для обробки даних в IoT-системах

У рамках цього завдання слід розробити бібліотеку, яка об'єднує всі основні методи обробки даних у контексті IoT. Вона повинна включати механізми для ефективної роботи з потоками даних, обробки інформації від сенсорів в реальному часі, а також оптимізації використання мережевих ресурсів для передачі даних між пристроями IoT та хмарними або локальними серверами. Важливою частиною цієї бібліотеки стане оптимізація енергоспоживання для малопотужних пристроїв.

3. Розробка бібліотеки для збору даних з IoT-сенсорів

Власну бібліотеку буде розроблено для збору та обробки даних, що надходять від IoT-сенсорів. Це включатиме інтерфейси для підключення різних типів сенсорів, таких як температурні датчики, датчики вологості, датчики руху тощо. Особлива увага буде приділена підтримці багатопоточності та асинхронних операцій для обробки та передачі великих обсягів даних в реальному часі, що є критично важливим для високопродуктивних IoT-систем.

4. Апробація бібліотеки через створення прикладного IoT-додатку

Для перевірки розробленої бібліотеки буде створено прикладний додаток, який демонструє її використання в реальних IoT-сценаріях. Це може бути додаток для моніторингу даних від сенсорів, управління пристроями, або інтеграції з хмарними платформами для обробки та зберігання великих обсягів даних. Апробація дозволить оцінити ефективність бібліотеки в умовах реального часу і тестувати її на масштабованість в мережах IoT.

5. Аналіз методів обробки даних і розробка алгоритму для оптимізації в IoT

Після апробації бібліотеки та аналізу всіх методів обробки даних буде розроблено алгоритм, який дозволяє вибирати найефективніший метод обробки даних в залежності від конкретних обставин. Це може включати адаптацію до різних умов, таких як обмеження пропускної здатності мережі, енергоспоживання пристроїв, або потреби в реальному часі обробки даних.

Цей розділ описує передові бібліотеки для багатопотокової роботи в контексті IoT, зокрема RxJava, Threads, Kotlin Coroutines, а також концепції багатопоточності, асинхронності і паралельності. Задача полягає в тому, щоб визначити найкращі підходи для обробки даних в системах Інтернету речей, що включає як обробку сенсорних даних в реальному часі, так і ефективну передачу даних між пристроями і хмарними сервісами. Розробка уніфікованої бібліотеки для IoT та створення алгоритму оптимізації методів обробки дозволить досягти ефективної роботи та зниження затримок у реальних IoT-системах.

3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТА

3.1. Склад та ієрархія створюваного додатку

Мобільний додаток розробляється для платформи Android з урахуванням особливостей роботи з Інтернетом Речей (IoT). Оскільки IoT передбачає використання великої кількості пристроїв, підключених до інтернету, задача полягає в тому, щоб розроблений додаток міг обробляти дані, що надходять від цих пристроїв у форматі XML, і забезпечити ефективну взаємодію з ними. Додаток має використовувати різні підходи до багатопоточності та асинхронного виконання, щоб забезпечити швидку обробку та мінімальні затримки при роботі з даними в реальному часі.

Перш за все, необхідно розробити кілька алгоритмів для обробки XML-файлів, що містять дані з IoT-пристроїв. Оскільки дані можуть надходити в різних форматах і обсягах, кожен з алгоритмів, що використовуються в додатку, повинен бути оптимізований для роботи з великими наборами даних. Використання різних підходів до багатопоточності, таких як Thread and Locks, RxJava та Kotlin Coroutines, дозволяє забезпечити паралельну обробку кількох потоків даних одночасно, що є критично важливим для ефективної роботи з IoT-даними.

Важливою частиною додатку є інтерфейс користувача, який має бути зручним для взаємодії з даними, що надходять від IoT-пристроїв. Користувач повинен мати можливість вибирати, який саме метод обробки даних з XML-файлів він хоче використати, і відстежувати результат на екрані. При цьому додаток має показувати прогрес обробки в реальному часі, блокуючи основний потік лише на час обробки даних, щоб не впливати на інші процеси на пристрої.

Програма повинна також інтегруватися з зовнішнім носієм, таким як SD-карта або інші засоби зберігання, щоб мати можливість читати дані з пристроїв, підключених до IoT. Для цього важливо забезпечити належні дозволи на доступ до цих файлів і коректну обробку їхнього вмісту, оскільки XML-файли з великими даними можуть значно відрізнятися за розміром і форматом. Оскільки IoT-пристрої можуть працювати в умовах нестабільного з'єднання з мережею, додаток

має бути здатний працювати в оффлайн-режимі, а також забезпечувати стійкість до збоїв під час обробки даних.

З урахуванням змінних умов роботи IoT-пристроїв, таких як їхня взаємодія з іншими пристроями в мережі або зміни в навантаженні на систему, додаток повинен коригувати свою роботу відповідно до поточних умов. Зокрема, це означає, що час обробки даних може змінюватися в залежності від багатьох факторів, таких як доступність мережі, обробка даних іншими програмами чи сервісами, тому кожен запуск програми має демонструвати трохи змінений час виконання.

Важливим аспектом є також реалізація тестування додатка на різних типах Android-пристроїв, включаючи ті, що взаємодіють з IoT-пристроями через Bluetooth або Wi-Fi. Це дозволить забезпечити сумісність і стабільність роботи програми при різних налаштуваннях мережі та підключених пристроях.

Таким чином, розробка додатку повинна враховувати особливості роботи з IoT, де важливими аспектами є обробка великих обсягів даних у реальному часі, ефективне використання багатопоточних обчислень для оптимізації часу обробки, інтеграція з різними пристроями та забезпечення зручного інтерфейсу для кінцевого користувача.

3.2 Вхідні та вихідні інформаційні потоки для кожної задачі та підзадачі

У розробленій системі управління для мобільного додатку на платформі Android, вхідні та вихідні інформаційні потоки визначають, як дані надходять від користувача, пристроїв IoT, зовнішньої пам'яті і як вони обробляються. Кожен етап роботи системи управління (СУ) включає певні інформаційні потоки, які відповідають за отримання і передачу даних між компонентами додатку.

1. Прийом повідомлень від користувача

Користувач вибирає метод обробки даних з XML-файлів. Це відбувається через інтерфейс користувача, де представлені різні алгоритми обробки (Thread and Locks, RxJava, Kotlin Coroutines). Після вибору методу, вхідний потік включає сигнал від користувача для ініціалізації обробки даних за вибраним методом. Цей сигнал є першочерговим повідомленням, що запускає весь процес обробки:

Вхідний потік:

- Повідомлення від користувача про вибір методу обробки.

- Алгоритм обробки даних залежно від вибору користувача.

Вихідний потік:

- Запуск вибраного алгоритму з відповідними параметрами обробки.

2. Обробка даних з зовнішнього носія

Система процесу повинна отримати повідомлення про наявність доступу до зовнішньої пам'яті, наприклад, SD-карти або іншого пристрою зберігання, з якого будуть завантажені файли XML. Цей етап передбачає отримання даних з зовнішнього носія через відповідний інтерфейс Android API для доступу до файлів.

Вхідний потік:

- Повідомлення про дозвіл доступу до зовнішньої пам'яті (наприклад, через користувацький дозвіл або автоматичну перевірку наявності з'єднання з носієм).

- Зчитування XML-файлів з зовнішнього носія.

Вихідний потік:

- Дані XML, передані з зовнішнього носія до програми.

- Підтвердження успішного зчитування файлів або повідомлення про помилку у разі відсутності доступу.

3. Обробка даних з використанням вибраного методу

Після того, як система отримає необхідні дані з XML-файлів, вони повинні бути оброблені за допомогою вибраного методу: Thread and Locks, RxJava, або Kotlin Coroutines. Вхідний потік для кожного з методів включає запуск відповідних процесів для обробки даних у фонових потоках.

Вхідний потік:

- Передача XML-даних у вибраний метод обробки.

- Ініціалізація багатопоточних чи асинхронних задач, які обробляють дані.

Вихідний потік:

- Результат обробки даних (наприклад, час обробки, зміни у форматі файлів або інші вихідні дані).

- Оновлені дані, готові для виведення на екран користувача.

4. Інтерфейс користувача і відображення результатів

Інтерфейс користувача відображає інформацію про прогрес виконання алгоритму, а також результат обробки даних. Вхідний потік включає отримання інформації про хід виконання алгоритму, а вихідний потік — виведення результатів на екран та оновлення інтерфейсу.

Вхідний потік:

- Запит на відображення прогресу виконання алгоритму.
- Отримання результату обробки для відображення на екрані.

Вихідний потік:

- Виведення повідомлення про завершення обробки.
- Оновлення інтерфейсу користувача з результатами виконання.

5. Завершення обробки і виведення результатів

Після виконання всіх етапів обробки даних, додаток надає користувачеві підсумкові результати — час обробки, відомості про успішність виконання задачі, можливі помилки тощо.

Вхідний потік:

- Останні дані про час виконання та стан обробки.
- Вихідні дані після завершення обробки.

Вихідний потік:

- Повідомлення про успішне завершення операції та результати.
- Завершення процесу із відображенням результатів на екрані.

Візуалізація потоків на рисунку 3.1

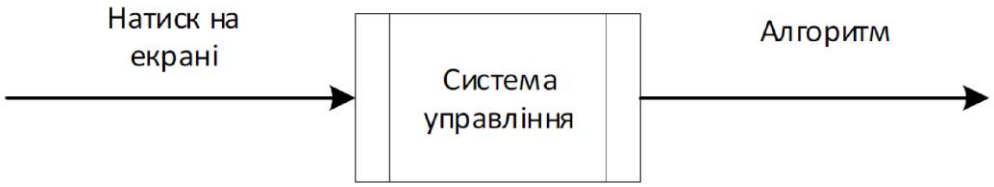


Рисунок 3.1 – Потоки процесу СУ, інформаційні

У процесі роботи з додатком відображаються наступні етапи:

- Користувач вибирає метод обробки.

- Система перевіряє доступність зовнішньої пам'яті.
- Вибраний метод обробки зчитує дані і запускає процес.
- Після завершення користувач отримує результат обробки.

Потоки процесу оновлення наведено на рисунку 3.2

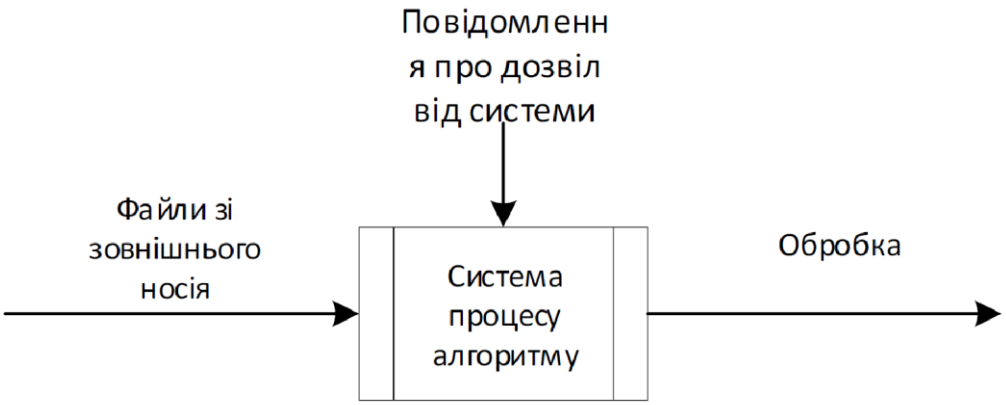


Рисунок 3.2 – Потоки процесу оновлення, інформаційні

Інформаційні потоки під час оновлення додатку включають такі етапи:

- Вхідне повідомлення про вибір методу обробки.
- Передача даних із зовнішнього носія.
- Обробка даних методом багатопоточного виконання.
- Оновлення інтерфейсу користувача з результатами.

Загалом, інформаційні потоки охоплюють всі етапи взаємодії користувача з додатком, починаючи з вибору методу обробки і закінчуючи виведенням результату. Кожен етап містить як вхідні, так і вихідні потоки, які визначають порядок і методи обробки інформації в системі управління.

3.3 Послідовність вирішення задач та підзадач

Розробка мобільного додатку для Android вимагає чіткої послідовності виконання задач, кожна з яких залежить від попередньої. Першочерговим завданням є створення алгоритмів парсингу даних, оскільки без них подальша робота додатку буде неможливою. Після цього, для забезпечення ефективного функціонування програми, розробляється інтерфейс користувача та організовується коректна обробка даних.

1. Створення алгоритмів парсингу даних

Першим кроком є розробка алгоритмів парсингу XML-файлів, оскільки саме ці файли будуть використовуватися для подальшої обробки. Алгоритми парсингу повинні ефективно обробляти великі файли, зважаючи на можливі зовнішні впливи, такі як зайнятість пам'яті чи інші програми, що працюють на девайсі. Для цього будуть використані стандартні бібліотеки Android для роботи з XML та додаткові інструменти для оптимізації процесу.

2. Розробка алгоритмів обробки даних

Після налаштування парсингу, необхідно перейти до розробки основних методів обробки даних. Це включає в себе створення кількох обраних методів, таких як Thread and Locks, RxJava, та Kotlin Coroutines. Кожен з методів повинен забезпечити обробку даних у фоновому режимі, не блокуючи основний потік програми.

3. Розробка інтерфейсу користувача

Одним із важливих етапів є створення інтерфейсу користувача, який дозволяє легко вибирати метод обробки і спостерігати за процесом виконання. Інтерфейс повинен бути інтуїтивно зрозумілим, щоб користувач без зусиль міг вибрати метод обробки, бачити хід процесу, а також отримувати підсумкові результати.

4. Реалізація блокування основного потоку

Для забезпечення коректної роботи додатку необхідно реалізувати механізм блокування основного потоку під час виконання операцій, пов'язаних з обробкою даних. Під час виконання кожного з методів обробки має бути активне вікно завантаження, яке інформує користувача про стан процесу. Вікно повинно бути невідключним до тих пір, поки обробка не буде завершена. Це гарантує, що користувач не зможе взаємодіяти з додатком під час виконання важких операцій.

5. Виведення результатів обробки

Після завершення обробки користувач має отримати підсумкові результати. Це включає в себе час обробки, який залежить від вибраного методу, а також будь-які інші важливі параметри, як-от статус успішності чи помилки. Виведення

результатів на екран відбувається через інтерфейс користувача, що дає змогу користувачу ознайомитись з результатами кожної операції.

6. Надання можливості вибору методу обробки

Кожен користувач може вибрати найбільш підходящий для себе метод обробки, виходячи з власних уподобань або потреб. Всі методи повинні мати однакову можливість результативно обробляти вхідні дані, і кожен із них повинен забезпечувати певні варіації в часі обробки, що дозволяє користувачу порівнювати ефективність різних методів.

Загалом, послідовність вирішення задач полягає в тому, щоб спершу створити основні компоненти для парсингу та обробки даних, а потім поступово додавати інтерфейс користувача та механізми, які дозволяють ефективно керувати потоком даних, зберігаючи при цьому оптимальну продуктивність програми.

3.4 Алгоритм функціонування додатку

Алгоритм функціонування додатку можна подати в декілька основних етапів, кожен з яких виконується по черзі після запуску програми. Ось як відбувається взаємодія користувача з додатком:

1. Запуск програми. Після запуску програми користувач бачить на екрані вітальне повідомлення, яке містить коротке привітання або пояснення щодо функціональності додатку. Цей етап не вимагає активних дій з боку користувача.

2. Перехід до активності вибору методів. На головному екрані користувач бачить єдину кнопку, яка виконуватиме роль переходу до наступного екрану – активності методів обробки. Кнопка повинна бути стилізована та привертати увагу, що дозволить користувачу зрозуміти, як діяти далі.

3. Вибір методу обробки. Після натискання на кнопку користувач потрапляє до активності, де відображається список доступних методів обробки даних. Кожен метод позначений відповідним описом чи назвою, що дає користувачу уявлення про те, який метод вибрати для обробки даних. Користувач має можливість вибрати один з методів для запуску обробки.

4. Операція завантаження та блокування основного потоку. Після вибору методу, на екрані з'являється діалогове вікно завантаження, яке інформує

користувача про те, що обраний метод обробки запускається. Це вікно має блокувати основний потік додатку до завершення операції. Таким чином, користувач не зможе взаємодіяти з іншими елементами програми, поки триває обробка даних. Завдяки цьому, програма не дасть змоги користувачу виконувати інші дії, поки виконуються ресурсоємні операції.

5. Обробка даних. Паралельно з відображенням діалогового вікна завантаження, програма починає обробляти дані з використанням обраного методу. Це може бути один з методів, таких як Thread and Locks, RxJava, або Kotlin Coroutines, що обробляють XML-файли в фоновому режимі. Кожен з цих методів обробки має своє специфічне функціонування, але всі вони взаємодіють із зовнішніми файлами та обчислюють час обробки, який з кожним запуском може варіюватися через зовнішні фактори (наприклад, доступ до пам'яті або інші програми).

6. Завершення обробки та відображення результату. Після того, як обраний метод обробки завершить свою роботу, вікно завантаження зникає, і користувач отримує результат виконання алгоритму на екрані. Результат може містити час обробки, успішність виконання, а також інші параметри, які допоможуть користувачу оцінити ефективність вибраного методу.

7. Повторний вибір або вихід. Після того, як користувач отримав результат обробки, у нього є можливість вибрати інший метод обробки або повернутися до головного меню. Програма має бути зручною для навігації, тому користувач повинен легко орієнтуватися, як перейти до наступного етапу або вийти з програми.

8. Завершення роботи програми. Після того, як всі вибори зроблені і результати продемонстровані, програма може повернути користувача до основного екрану або запропонувати йому завершити роботу, якщо це необхідно.

Таким чином, алгоритм функціонування додатку забезпечує зручний і послідовний процес вибору методів обробки, їх виконання з блокуванням основного потоку, а також надання результатів, що дозволяють користувачеві оцінити ефективність кожного методу.

3.5 Алгоритми вирішення задач

Додаток містить 7 методів обробки XML-файлів за допомогою трьох інструментів: Threads, RxJava та Kotlin Coroutines. Кожен з інструментів передбачає два варіанти виконання – обробка одного або кількох файлів у різних потоках. Останній інструмент, Kotlin Coroutines, також підтримує обробку в чотирьох нитках, що значно підвищує ефективність алгоритму.

1. Threads. Інструмент Threads дозволяє виконувати обробку файлів у фонових потоках, що запобігає блокуванню основного потоку програми. Для кожного методу обробки надається два варіанти:

- Обробка двох файлів у одному потоці: Файли будуть завантажуватися та оброблятися по черзі в одному потоці.

- Обробка двох файлів у різних потоках: Файли обробляються паралельно, що дозволяє зменшити час виконання, якщо ресурси системи дозволяють таку обробку.

2. RxJava – це потужна бібліотека для реактивного програмування, яка підтримує асинхронні обчислення. Вона надає можливість обробляти дані паралельно за допомогою таких функцій:

- Паралельна обробка через метод parallelStream(): Це дозволяє здійснювати паралельну обробку даних у кількох потоках, використовуючи потоки реактивного програмування. Це значно покращує ефективність обробки великого обсягу даних.

3. Kotlin Coroutines дозволяє ефективно працювати з асинхронними задачами, не блокуючи основний потік. У цьому випадку також передбачено два методи обробки:

- Обробка даних у одному потоці: Використовується для менш ресурсоємних операцій, коли необхідно обробити файли в один потік.

- Обробка даних у чотирьох потоках: Цей метод дозволяє обробляти файли паралельно в чотирьох окремих нитках, що істотно підвищує ефективність обробки великих файлів XML.

Алгоритм функціонування:

1. Запит на доступ до зовнішнього носія. Перш ніж почати обробку файлів XML, система повинна отримати дозвіл на роботу з зовнішнім носієм. Це дозволить додатку зчитувати файли з SD-карти або іншого зовнішнього пристрою. Користувач повинен надати доступ один раз при першому запуску додатку. Якщо доступ не надано, додаток закриється після відмови.

2. Вибір методу обробки. Після отримання дозволу, на головному екрані з'являється кнопка для вибору методу обробки. Якщо користувач погоджується надати доступ до зовнішнього носія, йому буде запропоновано вибрати один з методів обробки даних, що включає варіанти, які згадані вище (Threads, RxJava, Kotlin Coroutines).

3. Обробка даних за обраним методом. Після вибору методу обробки, система використовує обраний інструмент для обробки XML-файлів. Це може бути обробка в одному або кількох потоках, залежно від вибраного методу:

- Для Threads буде виконуватися обробка в одному або декількох потоках.
- Для RxJava можна застосувати методи для паралельної обробки даних.
- Для Kotlin Coroutines передбачено обробку даних в одному потоці або з використанням чотирьох потоків для більш ефективної обробки великих файлів.

4. Завершення обробки. Після завершення обробки файлів на екрані з'явиться результат (час обробки, успішність виконання). Користувач отримає можливість переглянути результати, а також вибрати наступний метод обробки або повернутися до головного екрану.

4. Перевірка на доступ до зовнішнього носія. Якщо користувач відмовляється надати доступ до зовнішнього носія, додаток не буде мати змоги працювати з файлами. У такому випадку система видасть повідомлення про помилку і запропонує повторно надати дозвіл при наступному запуску.

5. Автоматичне повторне запитування дозволу. Якщо користувач відмовляється від надання доступу, програма буде повторно запитувати дозвіл лише після того, як користувач натисне на відповідну кнопку на головному екрані.

Таким чином, кожен метод вирішення задачі обробки XML-файлів реалізовано за допомогою відповідних інструментів, що забезпечують ефективне

та асинхронне виконання задач у фоновому режимі з мінімальним впливом на основний потік програми.

3.6 Склад головного меню

Головне меню додатку є простим і інтуїтивно зрозумілим, оскільки воно складається лише з одного елемента — кнопки, яка переводить користувача в меню вибору методів обробки. Це меню служить першим кроком у роботі з додатком, надаючи користувачу можливість почати обробку даних.

Головне меню:

- Одна кнопка: Кнопка, яка ініціює перехід до меню вибору методу обробки даних.

- Дія: При натисканні на кнопку користувач потрапляє в меню, де може вибрати один з доступних методів обробки файлів.

Меню вибору методів обробки:

Меню вибору методів обробки складається з 7 кнопок, кожна з яких відповідає за окремий алгоритм обробки XML-файлів. Крім того, кожен метод має супроводжуючий опис, який пояснює його суть і особливості. Ось як виглядає склад меню:

1. Кнопка для вибору методу обробки з використанням Threads:

- Опис: Алгоритм обробки даних за допомогою простих потоків (Threads), що дозволяє обробляти файли по черзі або паралельно.

- Особливості: Цей метод оптимальний для менш ресурсномістких операцій, коли немає необхідності в розподілі процесів на кілька потоків.

2. Кнопка для вибору методу обробки з використанням RxJava:

- Опис: Метод обробки даних з використанням бібліотеки RxJava для асинхронної обробки за допомогою реактивних потоків.

- Особливості: Забезпечує паралельну обробку даних, що може значно зменшити час виконання завдань при обробці великих файлів.

3. Кнопка для вибору методу обробки з використанням Kotlin Coroutines:

- Опис: Використовує Kotlin Coroutines для ефективної обробки даних без блокування основного потоку.

- Особливості: Підтримує обробку даних як в одному потоці, так і в чотирьох потоках, що підвищує ефективність обробки великих файлів.

4. Кнопка для вибору методу обробки з одним потоком для двох файлів:

- Опис: Цей метод дозволяє обробляти два XML-файли в одному потоці. Підходить для простих завдань, де не потрібно паралельне виконання.

5. Кнопка для вибору методу обробки з декількома потоками для двох файлів:

- Опис: Метод обробки двох XML-файлів паралельно в кількох потоках. Це прискорює обробку при роботі з великими файлами.

6. Кнопка для вибору методу обробки з чотирма потоками в Kotlin Coroutines:

- Опис: Обробка даних з чотирма потоками в Kotlin Coroutines для більш складних завдань, де потрібно обробити великі обсяги даних.

- Особливості: Це дозволяє максимально використовувати ресурси системи для паралельної обробки даних.

7. Кнопка для вибору методу паралельної обробки за допомогою parallelStream() в RxJava:

- Опис: Використовує метод parallelStream() для паралельної обробки даних. Це дозволяє ефективно розподіляти навантаження на кілька потоків та зменшити час виконання.

Загальний принцип роботи меню:

- Користувач вибирає один із методів обробки, натискаючи відповідну кнопку.

- Після вибору методу, програма перенаправляє користувача до діалогового вікна, де він може завантажити файли та спостерігати за виконанням обраного алгоритму.

- У кожному методі користувач може переглядати опис та налаштування, що дозволяє йому краще зрозуміти, як працює той чи інший інструмент обробки.

Це меню дозволяє користувачеві зручно вибрати відповідний метод для обробки XML-файлів і здійснювати необхідні дії в залежності від своїх потреб.

3.7 XML файли для прикладу

На зовнішньому носії для демонстрації процесу обробки даних у системах Інтернету Речей (IoT) будуть зберігатися два великі XML-файли, розміри яких становлять 28 і 24 мегабайти відповідно. Ці файли були завантажені з ресурсів Вікіпедії з різних сторінок і містять великі обсяги даних, що є типовими для сценаріїв IoT, де системи часто мають справу з великими потоками інформації.

Ці файли містять загалом 196681 слово, що дозволяє наочно демонструвати відмінності у продуктивності різних алгоритмів обробки даних. В умовах IoT обробка великих наборів даних вимагає високої ефективності, оскільки пристрої часто працюють в реальному часі, обробляючи інформацію з численних датчиків і сенсорів, що передають великі обсяги даних.

Кожен алгоритм, використаний для обробки цих файлів, дозволяє побачити, як система реагує на обробку великих даних, що є важливим аспектом при розробці IoT рішень. У результаті виконання обробки даних користувач отримує кількість слів, оброблених в кожному методі, що може служити індикатором ефективності конкретного алгоритму в реальних умовах IoT, де швидкість обробки і масштабованість відіграють критичну роль.

3.8 Запити до системи для отримання дозволів

Google надає особливу увагу безпеці операційної системи Android, що є важливим аспектом не тільки для звичайних мобільних додатків, але й для додатків, які взаємодіють із Інтернетом Речей (IoT). Користувачі повинні надавати дозволи на доступ до певних ресурсів на своєму пристрої для забезпечення коректної роботи IoT-пристроїв, які підключаються через мобільні додатки.

У випадку IoT-додатків, які працюють із зовнішніми датчиками, пристроями або зберігають дані на зовнішніх носіях (наприклад, SD-картах чи інших пристроях зберігання), необхідність у доступі до зовнішньої пам’яті стає критично важливою. Більшість IoT-систем передбачають обмін даними через XML файли або інші формати даних, що зберігаються на цих носіях. Для забезпечення цієї взаємодії додаток повинен отримати дозвіл на доступ до зовнішнього середовища пам'яті.

У таких додатках зазвичай запитуються дозволи WRITE_EXTERNAL_STORAGE (для запису даних на зовнішнє сховище) та READ_EXTERNAL_STORAGE (для зчитування даних з цього сховища). Це дозволяє додаткам обробляти великі обсяги даних, отриманих від різних пристроїв або датчиків, які часто використовуються в системах Інтернету Речей (IoT). Наприклад, пристрої можуть збирати дані у реальному часі та записувати їх у XML файли на зовнішній носій для подальшої обробки.

Якщо ці дозволи не будуть надані, додаток не зможе здійснювати доступ до необхідних даних, що серйозно обмежить його можливості щодо управління IoT-пристроями або обробки отриманих даних. Крім того, відсутність доступу до зовнішньої пам'яті може призвести до збоїв у зборі, зберіганні або передачі даних, що для IoT-систем, де дані можуть бути критичними для роботи пристроїв, є неприпустимим.

Запит на надання дозволів у вигляді діалогового вікна дозволяє користувачу вибрати, чи надасть він додатку доступ до зовнішнього сховища. Важливо, щоб цей запит був чітким і зрозумілим, щоб користувач не відмовлявся від необхідних дозволів. Якщо запит буде занадто складним або невизначеним, користувач може відмовити в наданні дозволу, що зробить подальшу роботу додатку неможливою. Це важливо особливо для IoT-додатків, де взаємодія між різними пристроями вимагає точного та безперебійного зчитування й запису даних.

Процес запиту дозволів відображається на рисунках 3.3 і 3.4, де на кожному етапі користувачу надається можливість дозволити або відхилити запит на доступ до зовнішнього носія. Для IoT-систем, цей доступ критично важливий, оскільки пристрої можуть не лише зчитувати з носіїв дані, але й записувати нові показники або конфігураційні файли, що необхідно для підтримки функціональності всієї системи.

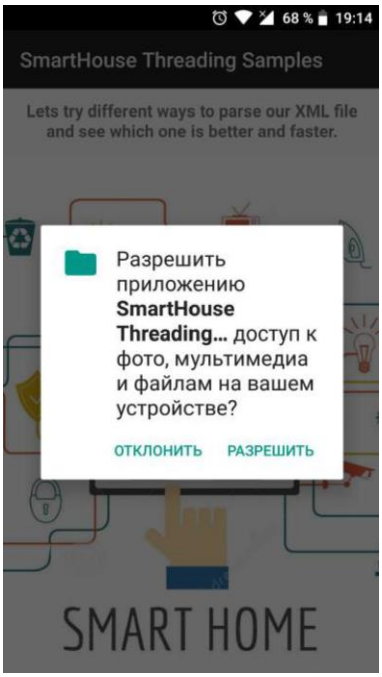


Рисунок 3.3 – Запит системи для користувача щодо доступу до додатка до зовнішнього носія в контексті IoT

На рисунку 3.3 показано стандартний запит системи до користувача щодо надання доступу до зовнішнього носія для IoT-пристроїв, що зчитують і записують дані в реальному часі.

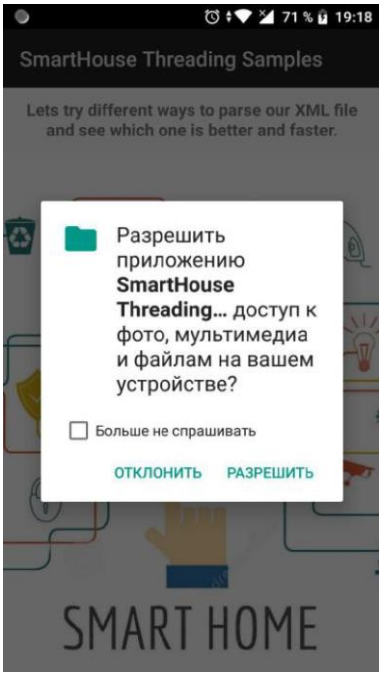


Рисунок 3.4 – Запит системи щодо доступу до зовнішнього носія у випадку, якщо доступ був відмовлений раніше

Рисунок 3.4 демонструє ситуацію, коли користувач раніше відмовився надати дозвіл на доступ до зовнішнього носія, і система знову запитує доступ для IoT-додатку.

Отже, ці дозволи не тільки дозволяють працювати з XML-файлами, але й забезпечують необхідну основу для функціонування системи Інтернету Речей, де обробка даних та взаємодія з пристроями є невід'ємною частиною повсякденних операцій.

3.9 Запити до зовнішнього середовища пам’яті для подальшого отримання файлів (IoT контекст)

У випадку додатків для Інтернету Речей (IoT), що працюють із великими обсягами даних, важливим є ефективне зберігання і обробка даних, отриманих з різних пристроїв. Одним з популярних підходів є зберігання даних у форматах, таких як XML, на зовнішньому носії. Однак з огляду на обмеження розміру APK-файлу та можливу високу потребу в обсязі пам'яті, розміщення таких файлів безпосередньо в додатку може значно збільшити його розмір (наприклад, на 50 МБ). Тому було вирішено переміщати ці файли на зовнішній носій, що дає змогу економити місце на мобільному пристрої та зберігати великі обсяги даних для подальшої обробки.

У додатках для IoT файли XML часто містять налаштування, дані з пристроїв, показники датчиків, інформацію про стан мережі та інші параметри, які постійно змінюються. Таким чином, важливо забезпечити безперешкодний доступ до цих файлів для подальшого зчитування та обробки. Для цього в коді було реалізовано механізм, що дозволяє зчитувати файли з конкретного каталогу зовнішнього середовища пам'яті, визначеного статичними змінними в класі Source.

У даному контексті було розроблено алгоритм для читання вмісту XML файлів з зовнішнього носія. Оскільки файли можуть бути великими, а доступ до них потребує високої ефективності, алгоритм побудований на принципах ітераційної обробки даних. Це дає змогу не тільки зчитувати великі файли по частинам, але й реалізувати оптимальне використання пам'яті.

1. Ітераційне зчитування XML файлів

На самому початку об'єкт класу Pages створюється для обробки XML файлів. Клас Pages успадковує інтерфейс Iterator, що дозволяє поетапно зчитувати вміст файлу. У параметри конструктора передається раніше зчитаний файл із зовнішнього носія, що дозволяє почати обробку даних.

2. Перехід по тегах і вимірювання часу

В середині класу Pages було створено клас PageIterator, що також успадковує інтерфейс Iterator. Цей клас відповідає за перебір всіх символів у файлі, перехід між тегами, а також за вимірювання часу, що дозволяє оцінити продуктивність обробки великих файлів. Зазвичай для IoT-систем це важливо, оскільки час затримки в обробці даних може впливати на швидкість реакції пристроїв у мережі.

3. Читання даних на рівні слів

Окрім класу Pages, були також створені класи Words та WordIterator, які спеціалізуються на зчитуванні вмісту сторінки файлу XML на рівні окремих слів. Це дозволяє точно обробляти окремі елементи даних, отримані з пристроїв чи датчиків. Ітераційний процес дає можливість обробляти великі обсяги даних поетапно, не перевантажуючи пам'ять.

Цей підхід є важливим для ефективної роботи IoT-додатків, що мають справу з великими даними. Наприклад, дані про стан різних датчиків, вимірювання температури, вологості, тиску тощо, можуть бути збережені у великих XML файлах і потребують швидкої обробки для прийняття рішень в реальному часі.

Ітераційні методи зчитування дозволяють не тільки ефективно працювати з великими файлами, але й забезпечують більш високу продуктивність та надійність системи, особливо коли обробка даних відбувається без затримок, що критично важливо для IoT-пристроїв.

Докладний програмний код для роботи з XML файлами, а також опис створених класів можна знайти в Додатку Б. Ці класи є основою для обробки даних у середовищі IoT, де кожен елемент системи, будь то пристрій, сенсор чи додаток, взаємодіє через ці дані для досягнення оптимальної продуктивності та надійності.

3.10 Звіт про завершення парсингу файлів (IoT контекст)

У додатках для Інтернету Речей (IoT), що працюють з великими обсягами даних, важливим етапом є ефективна обробка та аналіз зібраної інформації. Після завершення парсингу (розбір і обробка даних) файлів, користувач отримує звіт, який включає детальну інформацію про результат обробки. Зазвичай це включає час виконання операції, кількість зчитаних або оброблених елементів (наприклад, кількість слів, символів, чи тегів) у кожному файлі або наборі файлів.

Основні параметри звіту:

1. Час виконання парсингу. Оскільки парсинг великих файлів, таких як XML, є важливою частиною роботи IoT-систем, час обробки є ключовим параметром для моніторингу ефективності роботи додатка. Звіт надає точний час, який був витрачений на парсинг файлів, що допомагає оцінити швидкість обробки даних і виявити можливі вузькі місця в системі.

2. Кількість слів (елементів) в файлі. Важливою частиною звіту є також кількість слів або інших елементів (тегів, символів, даних), зчитаних із файлів. У контексті IoT, ці елементи можуть містити вимірювання датчиків, статуси пристроїв, налаштування мережі або інші значення, що передаються між пристроями. Звіт містить загальну кількість таких елементів у файлі, що дозволяє здійснити оцінку обсягу оброблених даних.

3. Загальна статистика. Звіт може також включати іншу корисну інформацію, таку як середній час парсингу для кожного елемента, кількість знайдених помилок чи відхилень в даних, тощо. Це дає можливість користувачу або системі оцінити, чи працює додаток ефективно або чи є необхідність в оптимізації процесу обробки даних.

Після виконання парсингу XML файлів додаток виводить результат, що містить час виконання обробки та кількість зчитаних елементів. Це може бути у вигляді текстового звіту або відображено графічно на інтерфейсі користувача. Приклад звіту:

```
Звіт про завершення парсингу файлів:
```

---

1. Час парсингу: 4.35 секунди
2. Загальна кількість слів: 1524
3. Загальна кількість тегів: 342
4. Кількість оброблених файлів: 2
5. Помилки: немає

```

Такий звіт є важливим не лише для оцінки ефективності обробки даних, але й для моніторингу роботи пристроїв у реальному часі в контексті IoT. Наприклад, в системах, що обробляють інформацію з великої кількості датчиків, такий звіт дозволяє швидко визначити, чи не перевантажено систему, чи правильно працюють пристрої, і які саме дані були отримані з мережі. Результат роботи додатку після завершення обробки наведено на рисунку 3.5

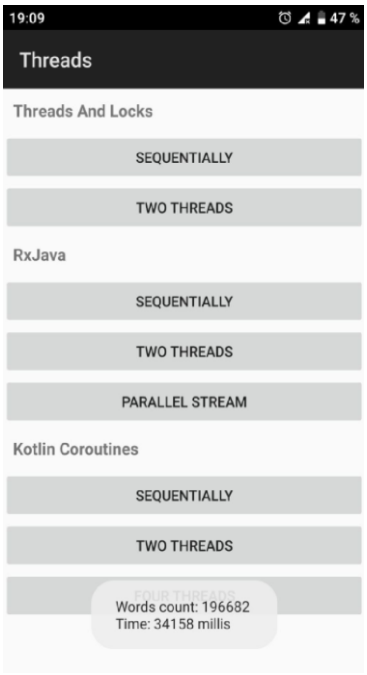



Рисунок 3.5 – Результат роботи додатку після завершення обробки

Зображення, яке може бути прикріплене до цього розділу, зазвичай містить інтерфейс користувача, на якому відображено результати парсингу: час виконання,
```

кількість оброблених елементів та інші дані. Це дозволяє користувачеві отримати швидку та зручну інформацію про стан обробки даних.

3.11 Форми меню (IoT контекст)

Додаток для Інтернету Речей (IoT), що працює з великими обсягами даних або обробкою файлів, має бути зручним і зрозумілим для користувача. Важливо, щоб інтерфейс користувача (UI) забезпечував швидкий доступ до необхідних функцій і можливість контролювати процеси збору, обробки і зберігання даних.

У нашому випадку, додаток складається з двох основних екранів:

1. Екран привітання (Рисунок 3.6). На першому екрані користувач зустрічає вітальне повідомлення або логотип програми. Це може бути перший крок у процесі взаємодії з додатком. Оскільки додаток працює з зовнішніми даними (XML файли, наприклад, для вимірювань або налаштувань IoT пристроїв), цей екран має бути простим і зрозумілим. Всі складні налаштування або додаткові функції доступні на наступних екранах.

Особливості екрана привітання:

- Кнопка переходу до наступного екрана: Єдина кнопка, яка є на цьому екрані, дозволяє користувачеві перейти до наступного екрана меню для вибору методу обробки даних.

- Логотип або вітальне повідомлення: Інтерфейс може містити логотип компанії або коротке вітання, яке дозволяє користувачу одразу зрозуміти, з яким додатком він взаємодіє.

2. Екран методів обробки (Рисунок 3.7). На другому екрані користувач може вибирати різні методи обробки даних. Оскільки додаток для IoT може працювати з великими наборами даних, важливо мати кілька варіантів обробки, щоб зручніше адаптувати систему під різні вимоги користувача або пристрою.

Особливості екрана методів обробки:

- 8 кнопок, кожна з яких відповідає певному методу обробки: Це може бути вибір способу парсингу XML файлів, фільтрації даних, або вибору алгоритмів для аналізу отриманих даних з IoT пристроїв. Кожна кнопка запускає відповідний процес.

- Зображення або іконки на кнопках: Кожна кнопка може містити відповідну іконку або текст, що ясно показує, яку саме функцію виконує вибір.

- Інтуїтивно зрозумілий інтерфейс: Користувачі повинні швидко орієнтуватися в доступних варіантах і вибирати потрібний метод обробки без зайвих зусиль.

Рисунок 3.6 – Екран привітання

Рисунок демонструє стартовий екран додатку, на якому є лише одна кнопка для переходу до наступного етапу. Наведено на рисунку 3.7.

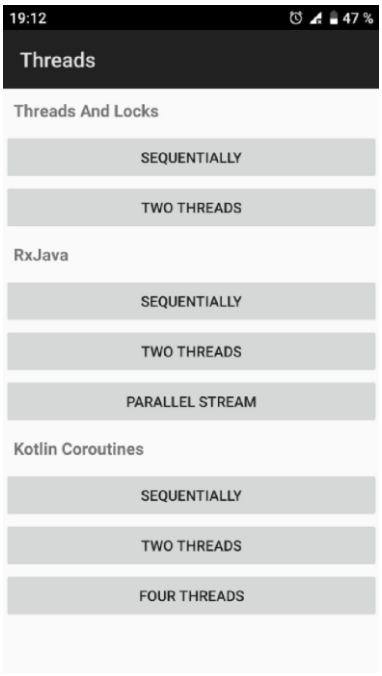


Рисунок 3.7 – Екран методів обробки

Цей екран містить кілька кнопок, кожна з яких відповідає конкретному методу обробки або функціональності додатку.

Важливо зазначити, що всі ці екранні форми повинні бути оптимізовані для мобільних пристроїв і забезпечувати зручне управління навіть при взаємодії з великою кількістю даних, що типово для IoT додатків.

3.12 Принципи проектування інтерфейсу користувача

Для додатку Інтернету Речей (IoT), графічний інтерфейс користувача (ГІК) та меню мають бути спроектовані з урахуванням особливостей взаємодії з пристроями IoT, які можуть вимагати обробки великих обсягів даних, відображення складної інформації та надання користувачеві можливості швидкого прийняття рішень. У цьому контексті особливу увагу слід приділити зручності використання, інтуїтивно зрозумілому управлінню та ефективному відображенню даних.

Графічний інтерфейс користувача є основним способом взаємодії користувача з додатком. ГІК має бути простим, зрозумілим та наглядним, щоб користувач міг швидко орієнтуватися і ефективно використовувати додаток навіть без попереднього досвіду.

Особливості ГІК для IoT додатків:

1. Метафори для зручності користувача: Оскільки додаток може включати складні функції, важливо використовувати метафори, які допомагають користувачам швидко зрозуміти функціональність. Наприклад, іконки, що позначають різні пристрої або методи обробки даних, можуть бути зображені у вигляді інтуїтивно зрозумілих символів (наприклад, зображення датчика для вимірювальних пристроїв або графіка для аналітичних функцій).

2. Інтерактивність: Кожен елемент інтерфейсу, будь то кнопка або меню, має бути інтерактивним, з можливістю взаємодії через сенсорний екран або інші пристрої введення. Інтерактивність дозволяє користувачеві миттєво реагувати на зміни і дає можливість гнучко налаштовувати роботу додатку під потреби.

3. Візуальне відображення даних: Оскільки IoT пристрої генерують великий обсяг інформації (наприклад, дані сенсорів, показники температури, вологості тощо), інтерфейс має бути здатний ефективно відображати ці дані. Важливо використовувати графічні елементи, такі як графіки, діаграми або карти, для наочного представлення інформації.

Меню в додатках IoT повинні бути продуманими для швидкого доступу до ключових функцій додатку. Оскільки додатки IoT часто включають кілька етапів взаємодії (збір даних, обробка, налаштування параметрів тощо), меню повинне бути організоване в ієрархічній формі, що дає змогу легко переміщатися між різними етапами роботи.

Особливості меню в додатках IoT:

1. Ієрархічна структура: Меню має мати чітку структуру, де кожна категорія має підкатегорії, що дозволяють користувачеві швидко знайти потрібну функцію або налаштування. Наприклад, користувач може спочатку вибирати тип пристрою, потім переходити до налаштувань для конкретного сенсора, а потім вибирати метод обробки або аналізу даних.

2. Повноекранне меню: Додаток може використовувати повноекранне меню для забезпечення доступу до всіх функцій. Такі меню можуть бути представлені у вигляді списків або плиток з іконками, що відповідають за певні категорії, як,

наприклад, налаштування сенсорів, перегляд результатів вимірювань або запуск аналізу даних.

3. Швидкий доступ до основних функцій: Важливо, щоб користувач міг швидко отримати доступ до основних функцій, таких як запуск збору даних або запуск процесу обробки та аналізу, не витрачаючи багато часу на пошук цих функцій у меню.

На Рисунку 3.8 і Рисунку 3.9 представлені схематичні зображення екрану привітання та екрану методів обробки, що допомагають зрозуміти, як буде виглядати інтерфейс користувача додатку:

- Рисунок 3.8. Це перший екран, який вітає користувача та пропонує перейти до подальшого налаштування або взаємодії з додатком. Тут є лише одна кнопка, що дозволяє перейти до наступного етапу.

- Рисунок 3.9. На цьому екрані користувач може вибрати одну з восьми кнопок, кожна з яких відповідає за запуск певного методу обробки даних. Це може бути вибір методу парсингу, фільтрації даних або інші операції, що характерні для роботи з даними, отриманими з пристроїв IoT.

Рисунок 3.8 – Схема екрану привітання

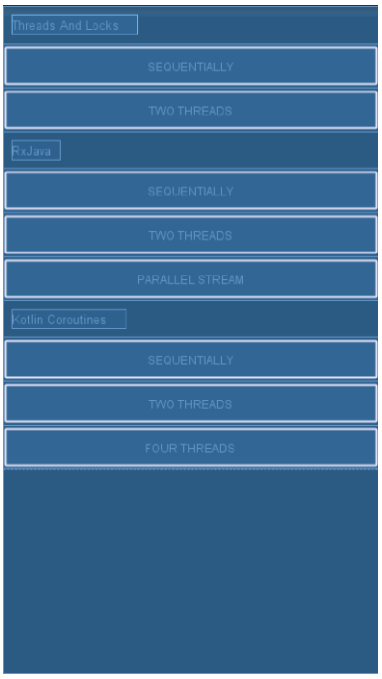


Рисунок 3.9 – Схема екрану методів обробки

Проектування інтерфейсу користувача в додатку для IoT вимагає використання принципів, які забезпечують інтуїтивно зрозумілу взаємодію з додатком, можливість швидкого доступу до потрібних функцій і ефективне відображення результатів обробки даних. Ієрархічні меню та графічні елементи мають бути адаптовані до специфіки IoT додатків, де часто потрібно працювати з великими обсягами даних і здійснювати складні операції обробки інформації.

3.13 Структура комплексу програми

Каталог створеної програми побудовано за принципом організації програмного коду та ресурсів, що забезпечують чітке розмежування між основними компонентами додатку. Нижче наведено загальну структуру каталогу з поясненням файлів та їхнього призначення. Також UML-діаграма комплексу додатку подана у додатку Д.

Основні компоненти каталогу:

1. .2 Activity:

- MainActivity: Головний клас активності, який відповідає за запуск програми та керування головним екраном користувацького інтерфейсу. Саме тут

відбувається початкове налаштування додатку та ініціалізація основних елементів інтерфейсу.

- ThreadsActivity: Клас активності, який керує багатопоточними процесами, що використовуються для паралельної обробки даних. Цей компонент відповідає за налаштування й запуск потоків для ефективної роботи з великими обсягами текстової інформації.

2. 8 файлів (методи обробки):

Кожен з цих файлів відповідає за різні методи обробки даних, які реалізовані в програмі. Методи використовують різні підходи до обробки даних, зокрема багатопоточність, реактивні потоки (Rx) та корутини. Нижче наведено список методів:

- ThreadBetterWordCount: Метод багатопоточної обробки, що покращує швидкість підрахунку слів у тексті за рахунок оптимізованого розподілу ресурсів між потоками.

- ThreadSequentialWordCount: Метод підрахунку слів, що виконується послідовно в одному потоці без паралельної обробки.

- RxTwoThreadsWordCount: Метод реактивного програмування з використанням двох потоків для підрахунку слів, що дозволяє одночасно обробляти різні частини тексту.

- RxParallelStreamWordCount: Реалізація паралельної обробки тексту з використанням потоків на базі бібліотеки RxJava, що дозволяє підвищити ефективність обробки великих обсягів даних.

- RxSequentialWordCount: Метод підрахунку слів, реалізований на базі RxJava, але виконується послідовно, що дозволяє порівняти продуктивність з іншими методами.

- CoroutinesTwoThreadsWordCount: Використання корутин для багатопоточної обробки тексту з використанням двох потоків, що дозволяє поєднувати простоту і асинхронність.

- CoroutinesSequentialWordCount: Підрахунок слів з використанням корутин в одному потоці, що дозволяє порівняти ефективність корутин із класичною багатопоточністю.

- CoroutinesTwoThreadsWordCount (повтор): Повторне використання корутин для паралельного підрахунку з оптимізованим розподілом ресурсів.

3. Моделі та утиліти:

- Моделі: Файли моделей містять класи та структури даних, що використовуються для зберігання та передачі даних між різними компонентами програми. Це можуть бути структури для представлення результатів обробки тексту, статистичних даних тощо.

- Утиліти: Допоміжні файли, які включають функції та класи для спрощення виконання певних завдань, наприклад, обробки тексту, роботи з файлами чи керування потоками.

4. Ресурси та екрани:

- Ресурси: Каталог з графічними файлами, стилями, макетами екрану та іншими елементами, що використовуються для відображення інтерфейсу додатку.

- Екрани: XML-файли, які описують структуру екранів програми, такі як екран привітання, меню обробки та інші. Це дозволяє чітко розмежувати логіку програми та її графічний інтерфейс.

Структура комплексу програми чітко розділена на компоненти, що полегшує підтримку та подальший розвиток додатку. Багатопоточні методи, реактивні потоки та корутини забезпечують високу ефективність обробки текстових даних. UML-діаграма у додатку Д допоможе краще зрозуміти архітектуру програми та взаємодію між її компонентами.

3.14 Програмні інтерфейси

Програмні інтерфейси додатку побудовані навколо його основних компонентів, які взаємодіють між собою та з системою Android через точки входу і механізми керування процесами. Основним класом, що визначає точку входу в додаток, є MainActivity, оголошений у файлі Manifest.xml, а інтерфейси з іншими компонентами забезпечують взаємодію між класами обробки даних та графічним інтерфейсом користувача.

Точки входу.

1. MainActivity:

- MainActivity є головним компонентом програми, що запускається після ініціалізації. Він визначає базову логіку роботи додатку та реалізує перехід між різними активностями.

- У Manifest.xml визначено основну активність програми – MainActivity, що відповідає за первинне завантаження інтерфейсу.

- MainActivity утримує activity_layout, що є графічним шаром, у якому присутня кнопка з ідентифікатором «bthThreads».

- При натисканні на цю кнопку виконується дві основні дії:

1. Перевірка наявності дозволу системи Android на користування зовнішнім носієм. Це є важливим кроком для забезпечення безпеки та коректного функціонування додатку при доступі до зовнішніх ресурсів.

2. Після перевірки дозволу викликається активність ThreadsActivity, що відповідає за виконання обчислювальних процесів у потоках.

2. ThreadsActivity:

- ThreadsActivity відповідає за реалізацію багатопоточної обробки та містить основні методи для паралельної роботи з текстовими даними.

- Ця активність завантажує інтерфейс із макета activity_threads, який включає в себе текстові поля з описом методів обробки даних і 8 кнопок, кожна з яких відповідає за виклик відповідного методу обробки.

- Після натискання на одну з кнопок відбувається виклик діалогового вікна завантаження, яке сигналізує користувачу про початок процесу обробки.

Методи обробки.

Додаток підтримує кілька підходів до багатопоточної обробки, які реалізовані у вигляді окремих файлів. Для кожного підходу розроблено специфічні програмні інтерфейси для керування обробкою даних.

1. Threads:

- Перші два методи використовують класичну багатопоточність через Threads:

- ThreadSequentialWordCount – обробка даних у послідовному режимі в одному потоці.

- ThreadTwoThreadsWordCount – обробка даних у паралельному режимі з використанням двох потоків.

2. RxJava:

- Наступні три методи використовують бібліотеку RxJava для реактивної обробки даних:

- RxSequentialWordCount – послідовна обробка в одному потоці з використанням RxJava.

- RxTwoThreadsWordCount – паралельна обробка у двох потоках через RxJava.

- RxParallelStreamWordCount – паралельна обробка даних з використанням потоків та потокової обробки RxJava.

3. Kotlin Coroutines:

- Останні три методи використовують Kotlin Coroutines для асинхронної багатопоточної обробки:

- CoroutinesBetterWordCount – обробка з використанням чотирьох потоків для покращеної продуктивності.

- CoroutinesSequentialWordCount – послідовна обробка у одному потоці з використанням корутин.

- CoroutinesTwoThreadsWordCount – паралельна обробка в двох потоках з використанням корутин.

Утіліти.

Для покращення користувацького досвіду і забезпечення безперервності інтерфейсу було розроблено утіліту LoadingViewMaterial. Цей компонент відповідає за відображення завантажувального екрана, що блокує користувацький інтерфейс під час виконання складних обчислювальних процесів. Такий підхід дозволяє уникнути зупинки або зависання інтерфейсу до моменту завершення обробки.

Основні програмні інтерфейси додатку базуються на активностях MainActivity і ThreadsActivity, які забезпечують навігацію між екранами та обробку запитів користувача. Різні методи обробки даних (Threads, RxJava, Kotlin

Coroutines) забезпечують гнучкість у роботі з потоками та ефективне керування обчислювальними ресурсами.

3.15 Інструментарій розробки, мова програмування

Для розробки додатку використовувалися мови програмування Java, Kotlin та XML. Java застосовувалася для базових компонентів додатку, обробки потоків та взаємодії з Android API, зокрема для реалізації багатопоточних обчислень через створення і керування потоками. Kotlin, як основна мова для більш сучасних рішень в Android-розробці, використовувалася для асинхронних обчислень із застосуванням корутин, що значно спрощує роботу з багатопоточністю та підвищує ефективність обробки. XML слугував для визначення макетів інтерфейсу користувача, структури екранів та взаємодії з елементами програми, а також для читання та обробки даних з зовнішніх файлів.

Серед бібліотек використовувалася RxJava, яка є реактивною бібліотекою для керування асинхронними операціями, що дозволило реалізувати багатопоточні обчислення з потоковою обробкою. Крім того, застосовувалася бібліотека EventBus для подієвого зв'язку між компонентами додатку, що забезпечило ефективну передачу подій між різними частинами програми. Бібліотека Support-Compat використовувалася для забезпечення сумісності додатку з різними версіями Android, що дозволило використовувати нові функції та підтримувати старіші версії операційної системи.

Інструментом для розробки додатку була Android Studio, яка забезпечувала повний цикл розробки, включно з написанням коду, створенням інтерфейсів, тестуванням та налагодженням. Android Studio також надавала зручні засоби для управління проєктом, інтеграції з Android SDK, запуску емуляторів пристроїв та відлагоджування додатку на різних конфігураціях Android.

Окрім того, у додатку були реалізовані методи для отримання системних дозволів, необхідних для доступу до зовнішніх носіїв. Для цього використовувалися стандартні засоби Android SDK, які забезпечували перевірку дозволів та, за потреби, їх запит у користувача. Також були реалізовані функції для

читання XML-файлів, що зберігаються на зовнішньому носії, з використанням стандартних інструментів Java та Android для обробки даних.

Розробка інтерфейсу користувача базувалася на принципах простоти та інтуїтивності. Було дотримано стандартів дизайну Android для забезпечення зручної навігації, а також використано сучасні підходи, такі як матеріальний дизайн, що надало додатку сучасний вигляд та спрощувало взаємодію користувачів із програмою.

Отже, для створення додатку використовувалися мови Java, Kotlin та XML, бібліотеки RxJava, EventBus та Support-Compat, а інструментом для розробки слугувала Android Studio. Це забезпечило ефективну реалізацію багатопоточних обчислень, асинхронності та інтуїтивного інтерфейсу користувача, водночас підтримуючи сумісність із різними версіями Android.

4. ОЦІНКА СТВОРЕНИХ МЕТОДІВ ПАРСИНГУ ТА ЇХ ПОРІВНЯННЯ

4.1 Аналізу навантаженності на систему від реалізованих методів

У середовищі розробки Android Studio доступ до Profiler дозволяє розробникам детально аналізувати споживання ресурсів додатком у реальному часі. Цей інструмент відображає ключові параметри системи, що дозволяє краще зрозуміти, як додаток впливає на продуктивність пристрою під час виконання різних операцій.

Параметри, що відстежуються за допомогою Profiler, включають:

1. Центральний процесор (CPU) – інструмент показує відсоткове навантаження на процесор під час виконання різних операцій. Це дає змогу розробникам визначити, наскільки обчислювально інтенсивними є процеси додатка. Наприклад, якщо додаток працює з багатопотоковими обчисленнями або інтенсивною графікою, можна відстежити, чи перевантажує це процесор. Це важливо, оскільки надмірне навантаження на CPU може призводити до затримок, зависань або підвищеного енергоспоживання.

2. Оперативна пам'ять (RAM) – Profiler дозволяє відстежувати, скільки пам'яті використовує додаток. Відображається як обсяг пам'яті в мегабайтах, який споживається під час виконання різних задач. Оскільки перевищення обсягу доступної пам'яті може призводити до збоїв у роботі додатка або навіть до закриття додатка системою для звільнення пам'яті, такий моніторинг допомагає оптимізувати використання ресурсів.

3. Енергоспоживання – для кожного сценарію роботи додатка система може показувати рівень споживання енергії. Profiler дає оцінку енергоспоживання, розподіляючи його на три рівні: низький, середній і високий. Цей параметр особливо важливий для мобільних додатків, які працюють на пристроях із обмеженим зарядом батареї, таких як смартфони та планшети. Занадто велике енергоспоживання може негативно вплинути на досвід користувача, оскільки додаток швидко виснажує батарею.

4. Мережеві ресурси – хоча в даному дослідженні цей параметр не є актуальним через відсутність інтернет-запитів у додатку, Android Profiler також дозволяє відстежувати трафік, пов'язаний із завантаженням і передачею даних через мережу. Це важливо для додатків, які активно працюють з онлайн-сервісами або потребують постійного підключення до мережі, щоб забезпечити плавну роботу з даними без надмірних затримок.

Для запуску аналізу необхідно спочатку підключити фізичний пристрій (смартфон або планшет) до комп'ютера, запустити додаток, увімкнути інструмент Profiler в Android Studio та вибрати процес, який слід моніторити. Після цього Profiler почне відображати в реальному часі інформацію про споживання ресурсів додатком у вигляді графіків. Спостерігаючи за цими графіками протягом певного періоду часу, розробники можуть робити висновки про ефективність реалізованих методів у додатку та визначати можливі шляхи для їх оптимізації.

Таким чином, Profiler є ключовим інструментом для оцінки продуктивності додатків на платформі Android. Він допомагає не тільки виявити проблемні місця в коді, але й оптимізувати додаток для кращого використання системних ресурсів.

На рисунку 4.1 показано приклад використання системних ресурсів додатком під час його запуску.

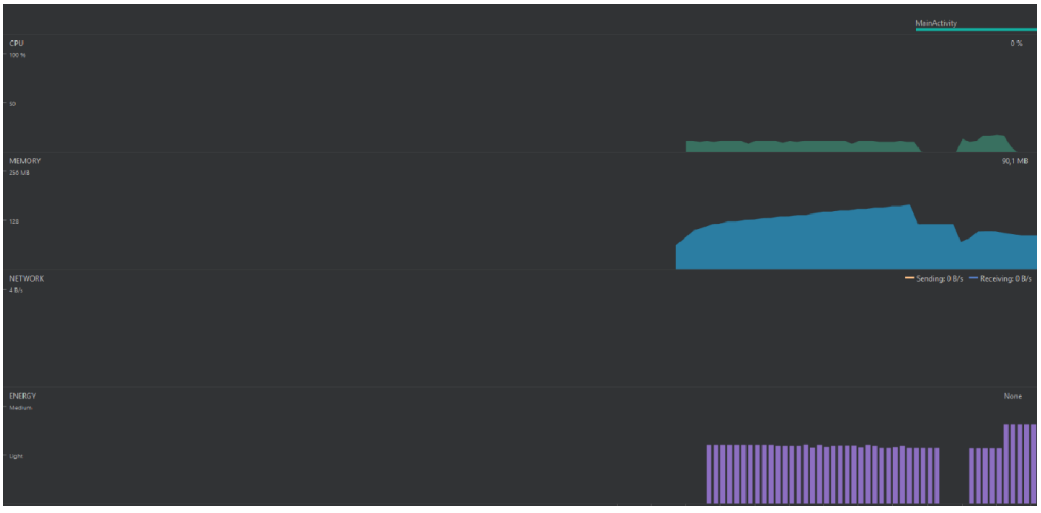


Рисунок 4.1 – Використання ресурсів додатком під час його запуску

4.1.1 Аналіз навантаженності на систему від реалізованих методів за допомогою Threads

Метод Threads and Locks демонструє різний рівень навантаження на систему в залежності від кількості потоків (див. рис. 4.2), які використовуються для виконання задачі. Це відображено в результатах аналізу навантаження на систему за допомогою інструменту Profiler в Android Studio, де відстежується використання ключових ресурсів, таких як процесор (CPU), оперативна пам'ять (RAM), а також енергоспоживання.

Один потік.

При виконанні методу Threads and Locks в один потік (див. рис. 4.2), система використовує наступні ресурси:

- ЦПУ: навантаження на центральний процесор становить 12%, що є відносно низьким показником;

- Оперативна пам'ять: споживання пам'яті дорівнює 180.7 мегабайт, що також вважається незначним;

- Енергоспоживання: на рівні середнього.

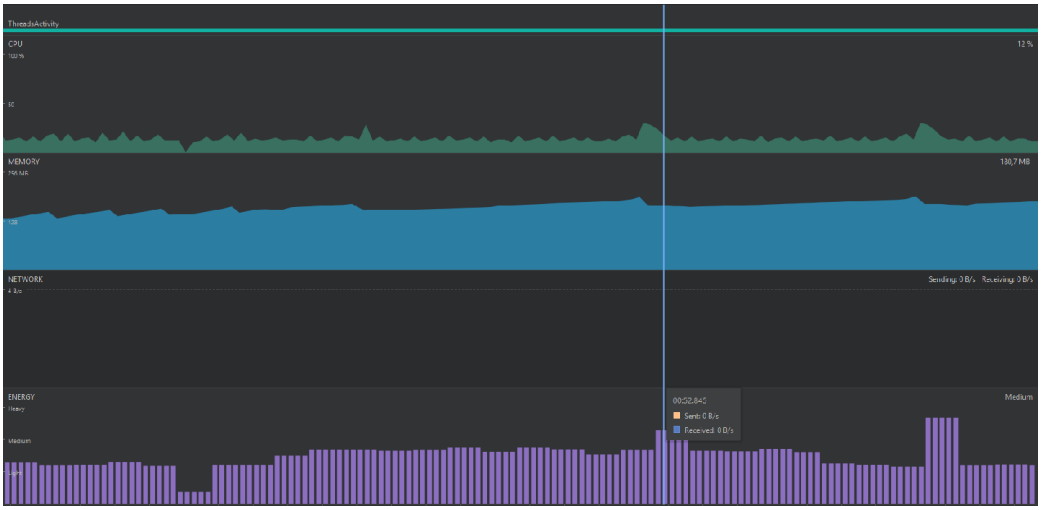


Рисунок 4.2 – Використання методу Threads в один потік

З наведених даних можна зробити висновок, що цей підхід з використанням одного потоку є досить ефективним з точки зору споживання ресурсів. Таке низьке

навантаження робить цей метод придатним для використання у широкому спектрі мобільних додатків, особливо у тих випадках, коли ресурси обмежені або додаток виконується на пристроях із невисокою продуктивністю.

Два потоки.

Однак при використанні методу Threads and Locks у два потоки (див. рис. 4.3) споживання ресурсів суттєво зростає:

- ЦПУ: навантаження на процесор збільшується до 23%, що майже вдвічі більше порівняно з одним потоком;

- Оперативна пам'ять: споживання пам'яті також підвищується до 233.1 мегабайт;

- Енергоспоживання: енергія витрачається на важкому рівні, що вказує на більш інтенсивне використання ресурсів пристрою.

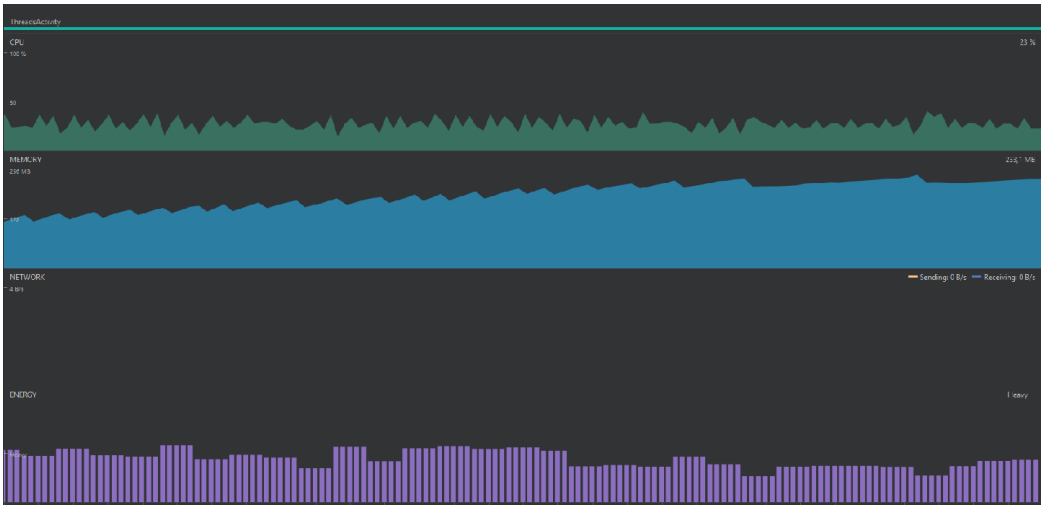


Рисунок 4.3 – Використання методу Threads в два потоки

Ці показники вказують на те, що використання двох потоків у цьому методі створює значне навантаження на систему. Таке навантаження може призвести до підвищення витрат ресурсів пристрою, що важливо враховувати при розробці додатків. Особливо це стосується додатків, які будуть працювати на пристроях з обмеженою обчислювальною потужністю або батареєю, оскільки підвищене енергоспоживання може негативно вплинути на тривалість роботи пристрою.

Метод Threads and Locks у два потоки може бути ефективним для виконання завдань, які вимагають високої продуктивності, але при цьому користувачам слід заздалегідь визначати обставини, за яких цей метод буде використовуватися. Важливо зважити можливі компроміси між продуктивністю та споживанням системних ресурсів, щоб не перевантажувати пристрій і не знижувати його загальну ефективність.

4.1.2 Аналіз навантаженності на систему від реалізованих методів за допомогою RxJava

При обробці метода RxJava в один потік споживаються певні ресурси (див. рис. 4.4):

- ЦПУ: навантаження на центральний процесор становить 15%, що є відносно низьким показником. Це вказує на те, що в цьому випадку програма ефективно використовує обмежені ресурси, не навантажуючи процесор.

- Оперативна пам'ять: споживання пам'яті дорівнює 204.6 мегабайт, що є порівняно помірним для такого типу операцій, і може бути допустимим у більшості випадків.

- Енергоспоживання: на рівні середнього, що свідчить про помірне використання енергетичних ресурсів пристрою, під час виконання цього методу.

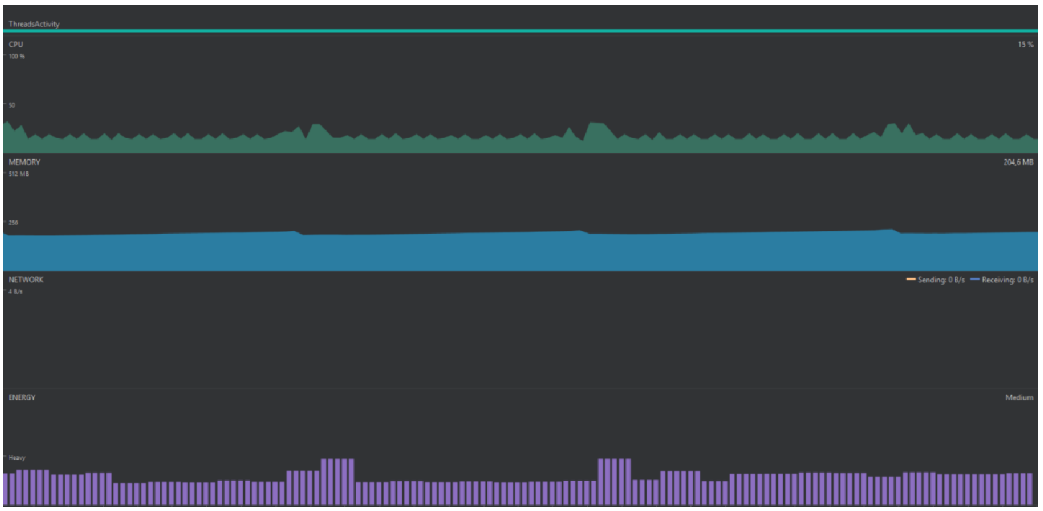


Рисунок 4.4 – Використання методу RxJava в один потік_

З аналізу видно, що метод RxJava в один потік має помірне навантаження на систему. Це робить його ідеальним вибором для додатків, де важливе ефективне використання ресурсів. Такий підхід забезпечує швидку та стабільну роботу на пристроях з обмеженими ресурсами або на старих моделях смартфонів. З огляду на помірне енергоспоживання та відносно низьке навантаження на процесор і пам'ять, цей метод можна вважати універсальним для широкого кола застосувань.

При обробці метода RxJava в два потоки споживаються певні ресурси (див. рис. 4.5):

- ЦПУ: навантаження на процесор збільшується до 39%, що вказує на підвищене використання обчислювальних потужностей пристрою. Це може мати негативний вплив на продуктивність пристроїв з менш потужними процесорами.

- Оперативна пам'ять: споживання пам'яті зростає до 228.6 мегабайт, що є суттєвим збільшенням порівняно з одно поточним виконанням. Це підвищене споживання може вплинути на швидкодію програми, особливо при запуску кількох додатків одночасно.

- Енергоспоживання: на рівні важкого, що вказує на високий рівень використання батареї, що може бути суттєвим недоліком для додатків, які планують працювати в середовищі з обмеженою батареєю.

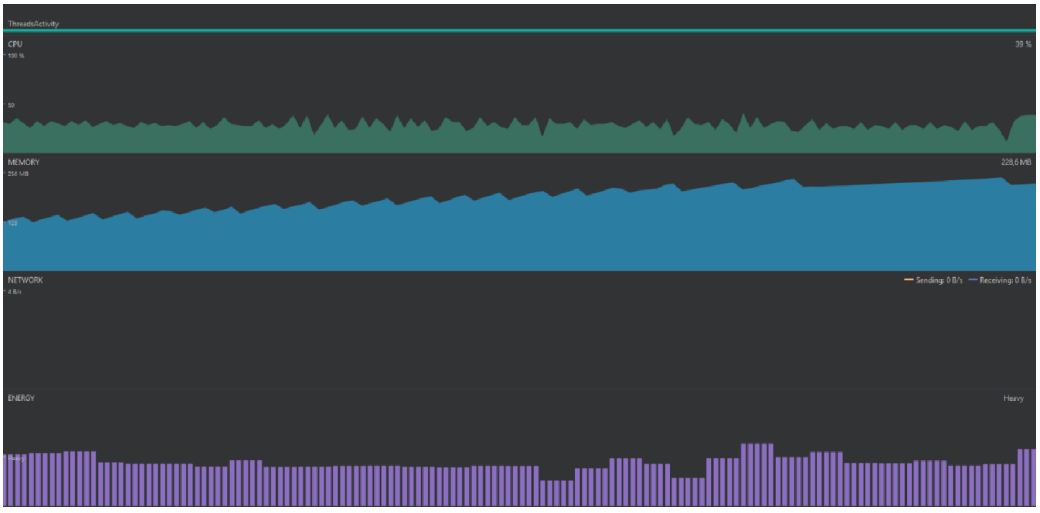


Рисунок 4.5 – Використання методу RxJava в два потоки_

З цієї інформації можна зробити висновок, що метод RxJava в два потоки створює значне навантаження на систему. Це підвищене навантаження на процесор і пам'ять може стати проблемою на старих пристроях або в сценаріях, де батарея є обмеженим ресурсом. Тому цей метод доцільно використовувати тільки в тих випадках, коли потрібна висока обчислювальна потужність, і коли користувачі можуть дозволити собі більші витрати енергії. Наприклад, для додатків, що вимагають обробки великих обсягів даних або складних операцій, цей метод може бути виправданим, але для легких додатків, де важлива економія енергії, він може бути надмірним.

При обробці метода RxJava за паралельними потоками споживаються певні ресурси (див. рис. 4.6):

- ЦПУ: навантаження на процесор становить 20%, що є помірним і дозволяє уникнути надмірного навантаження на систему. Це оптимальний варіант для багатьох застосувань, де продуктивність має бути збалансована з використанням ресурсів.

- Оперативна пам'ять: споживання пам'яті становить 179.7 мегабайт, що є нижчим порівняно з обробкою в два потоки, і дозволяє працювати ефективно навіть на менш потужних пристроях.

- Енергоспоживання: на рівні середнього, що є перевагою для додатків, де енергоспоживання є важливим фактором. Це дозволяє зберегти баланс між продуктивністю та енергоефективністю.

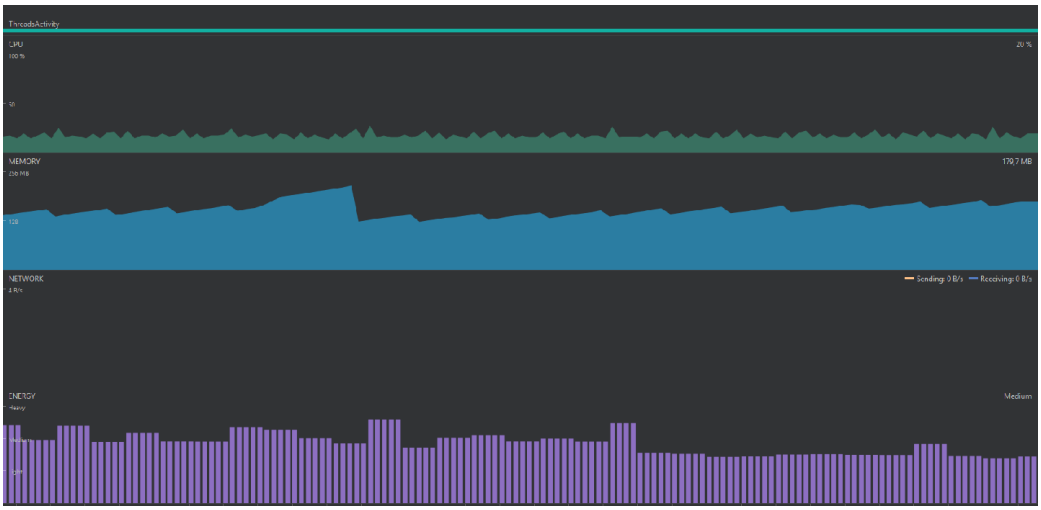


Рисунок 4.6 – Використання методу RxJava з паралельними потоками_

З огляду на результати, метод RxJava з паралельними потоками є оптимальним для більшості сценаріїв використання, оскільки він поєднує помірне навантаження на систему з хорошою продуктивністю. Він є чудовим варіантом для додатків, де необхідно обробляти дані ефективно без зайвих витрат енергії та пам'яті. Цей метод буде ефективним як для потужних пристроїв, так і для тих, що мають обмеження по ресурсах, зокрема для мобільних додатків, де важлива енергоефективність.

4.1.3 Аналіз навантаженності на систему від реалізованих методів за допомогою Kotlin Coroutines

При обробці метода Kotlin Coroutines в один потік споживаються певні ресурси див. рис. 4.7):

- ЦПУ: навантаження на процесор становить 21%, що є помірним і вказує на те, що використання одного потоку не створює суттєвого навантаження на центральний процесор.

- Оперативна пам'ять: споживання пам'яті дорівнює 163.7 мегабайт, що є досить помірним для цього методу і дозволяє йому ефективно працювати без значного впливу на доступні ресурси пристрою.

- Енергоспоживання: на рівні важкого, що вказує на високий рівень енергоспоживання під час роботи цього методу. Це може бути значним недоліком, особливо для мобільних пристроїв, де енергетичні ресурси є обмеженими.

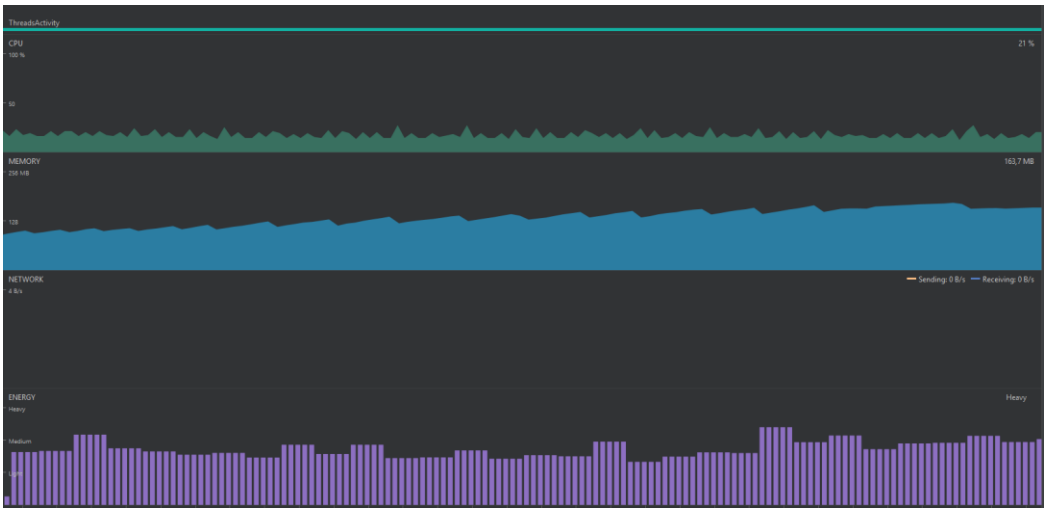


Рисунок 4.7 – Використання методу Kotlin Coroutines в один потік_

Метод Kotlin Coroutines в один потік, хоч і використовує помірно ресурси ЦПУ та пам'яті, все ж таки має високий рівень енергоспоживання, що може стати обмеженням для додатків, де важлива тривалість роботи від батареї. Тому для пристроїв з обмеженим енергетичним ресурсом або коли енергоспоживання має критичне значення, використання цього методу потребує ретельного обміркування. Він буде оптимальним для задач, де критично важлива простота обробки, але де використання великої кількості енергії може бути виправданим лише у виняткових випадках.

При обробці метода Kotlin Coroutines в два потоки споживаються певні ресурси (див. рис. 4.8):

- ЦПУ: навантаження на процесор збільшується до 31%, що є підвищеним порівняно з обробкою в один потік. Це може призвести до меншої ефективності на пристроях з обмеженими обчислювальними потужностями.

- Оперативна пам'ять: споживання пам'яті досягає 238.4 мегабайт, що є значним збільшенням порівняно з попереднім варіантом. Це може призвести до навантаження на систему, особливо на мобільних пристроях або старих моделях.

- Енергоспоживання: на рівні важкого, що зберігається на такому ж рівні, як і в методі з одним потоком. Це підвищене енергоспоживання є значним фактором, особливо для мобільних пристроїв, що потребують оптимізації енергії.

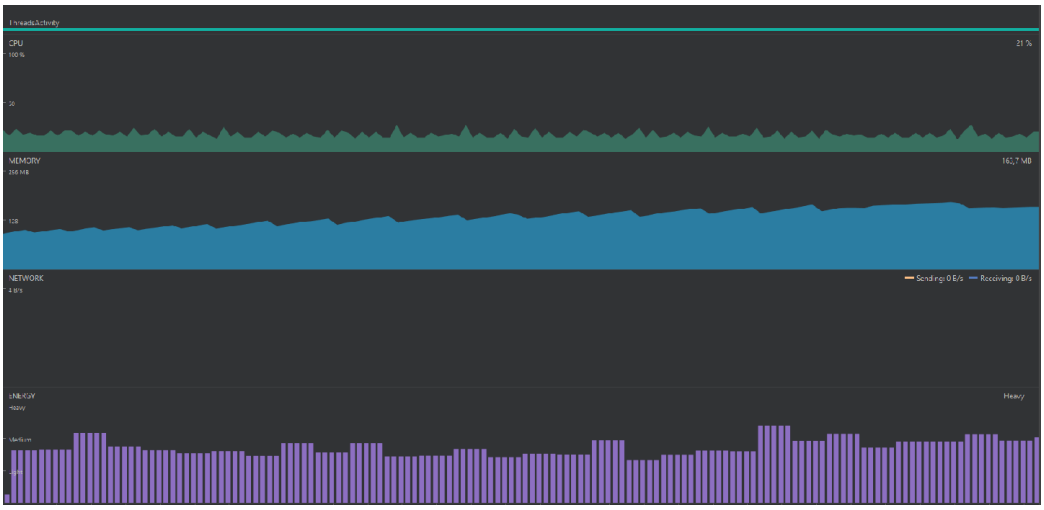


Рисунок 4.8 – Використання методу Kotlin Coroutines в два потоки_

З аналізу видно, що метод Kotlin Coroutines в два потоки створює ще більше навантаження на систему в порівнянні з одно-поточним варіантом. Хоча збільшена кількість потоків дозволяє зменшити час виконання обробки, це також збільшує використання ресурсів системи. Тому цей метод більше підходить для потужніших пристроїв, які можуть ефективно обробляти додаткові потоки та витримувати високий рівень енергоспоживання. Однак на старих або менш потужних пристроях це може стати проблемою, оскільки споживання пам'яті та ЦПУ збільшується, що може привести до уповільнення або навіть до нестабільної роботи програми.

При обробці метода Kotlin Coroutines в чотири потоки споживаються певні ресурси (див. рис. 4.9):

- ЦПУ: навантаження на процесор досягає 63%, що є значним підвищенням порівняно з попередніми варіантами. Це означає, що обробка в чотири потоки створює велике навантаження на систему і може призвести до істотного зниження продуктивності на пристроях з обмеженими ресурсами.

- Оперативна пам'ять: споживання пам'яті досягає 316.4 мегабайт, що є значним збільшенням в порівнянні з обробкою в два потоки. Це може суттєво вплинути на роботу системи, особливо при одночасному запуску декількох додатків.

- Енергоспоживання: на рівні важкого, що також вказує на дуже високий рівень енергоспоживання. Це може бути неприязно для мобільних додатків, оскільки підвищене енергоспоживання призводить до швидкого розряду батареї.

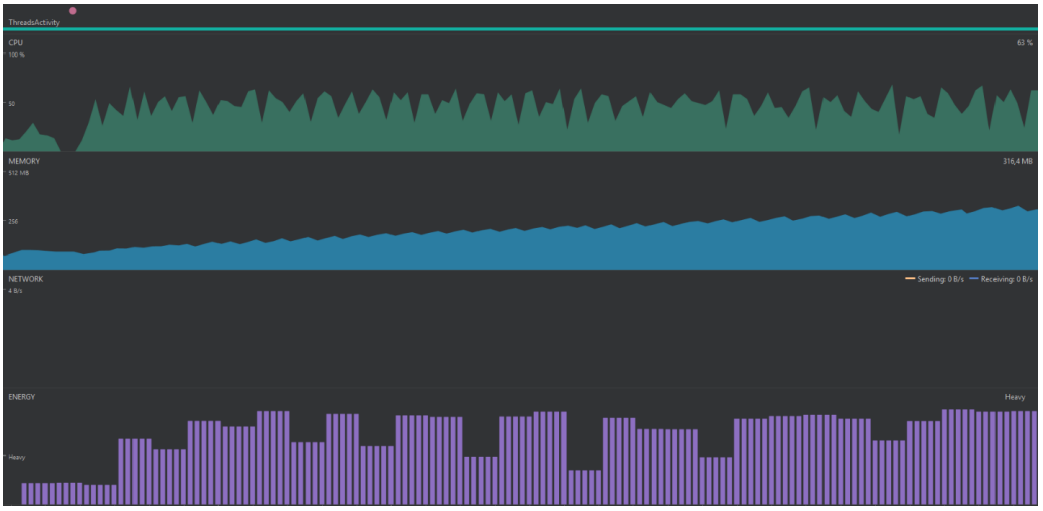


Рисунок 4.9 – Використання методу Kotlin Coroutines в чотири потоки_

Метод Kotlin Coroutines в чотири потоки значно збільшує навантаження на систему, як за рівнем використання ЦПУ, так і за споживанням пам'яті. Це робить його підходящим для використання лише на потужних пристроях з високими обчислювальними можливостями, таких як сучасні смартфони або пристрої, що мають велику кількість ядер і достатню кількість оперативної пам'яті. Для звичайних мобільних пристроїв або старіших гаджетів цей метод може виявитися занадто ресурсомістким і привести до погіршення загальної продуктивності та швидкого розряду батареї.

4.2 Аналіз витраченого часу на завершення парсингу створеними методами

Для отримання всебічної оцінки часових витрат на обробку XML-файлів реалізованими методами семантичного аналізу, необхідно провести серію тестів, що дозволяють оцінити ефективність роботи кожного методу в різних умовах. Алгоритм для збору та обчислення цих даних виглядає наступним чином:

1. Тестування без навантаження на систему. У цьому випадку проводиться вимірювання часу на обробку файлів XML одразу після перезавантаження системи, коли вона ще не задіяна іншими ресурсоємними процесами. Тестування у таких умовах дозволяє отримати показники роботи системи в найбільш оптимальних умовах, тобто коли всі ресурси доступні для виконання поставленої задачі, а фонові служби та додатки ще не почали своєчасно працювати. Цей тест дає можливість оцінити, скільки часу необхідно для обробки файлів за умов мінімального навантаження на ресурси системи.

2. Тестування з навантаженням на систему. У цьому випадку тестування проводиться після запуску кількох ресурсоємних додатків або виконання фонових процесів, які можуть значно знизити доступні системні ресурси. Під «завантаженням» мається на увазі вплив різних додатків, що працюють одночасно, і можуть використовувати значні частини процесора, пам'яті чи енергоспоживання, що є реалістичним сценарієм для багатьох мобільних та десктопних систем у реальному використанні. Тестування з навантаженням дозволяє оцінити, як змінюється час обробки XML-файлів, коли система вже зайнята іншими завданнями.

3. Обчислення середнього значення. Після проведення тестів за двома умовами (без навантаження та з навантаженням) проводиться підсумковий розрахунок середнього значення для кожного тесту. Це дає можливість отримати два часових інтервали для кожного методу: один при мінімальному навантаженні на систему, і інший — коли система вже завантажена іншими процесами. Середнє значення дозволяє створити більш реалістичну картину щодо того, скільки часу буде потрібно для виконання парсингу XML-файлів у реальних умовах.

Зібрані дані дозволяють зробити висновки про ефективність кожного методу в залежності від навантаження на систему. Таким чином, проведення таких тестів дозволяє не тільки оцінити швидкість виконання кожного методу, але й зрозуміти, як система поводиться при різних рівнях навантаження, що важливо для вибору оптимального методу для конкретних умов експлуатації. Тестування в реальних умовах, коли система вже працює з іншими додатками, дає змогу визначити,

наскільки вибраний метод буде ефективним у повсякденному використанні та забезпечить коректну роботу навіть при високому навантаженні.

4.2.1 Аналіз потрібного часу на завершення методів, реалізованих за допомогою Threads

У цьому підрозділі буде проведено аналіз часу, який необхідний для виконання парсингу XML-файлів за допомогою методу обробки через потоки (Threads)(див. рис.4.10). Оцінка часу виконання допоможе визначити, який варіант обробки є найбільш оптимальним в умовах різного навантаження на систему.

Тестування було здійснене для різних варіантів кількості потоків, і зафіксовані середні значення часу, необхідного для обробки даних:

- Один потік: Час обробки складає 30,67 секунд. Це базовий варіант, де система використовує лише один потік для обробки файлів XML. Цей метод може бути підходящим для ситуацій, коли не потрібно надмірного навантаження на систему і коли інші додатки не займають значних ресурсів.

- Два потоки: Час обробки зменшується до 24,50 секунд. Використання двох потоків дозволяє дещо прискорити процес обробки за рахунок паралельного виконання деяких завдань. Це може бути корисно в умовах, коли потрібно швидше обробити великі файли, і система може дозволити собі додаткові потоки.

- Один потік з додатковим навантаженням: У випадку додаткового навантаження (коли інші ресурсоємні програми або процеси займають значну частину системних ресурсів) час обробки збільшується до 33,37 секунд. Це демонструє, як додаткові навантаження можуть сповільнити роботу навіть для простого методу з одним потоком.

- Два потоки з додатковим навантаженням: Коли система знаходиться під навантаженням і використовується два потоки, час обробки дещо зменшується до 25,56 секунд, але все одно не досягає значного прискорення в порівнянні з обробкою в одну нитку в умовах навантаження. Збільшення кількості потоків може мати сенс, але це також залежить від того, скільки ресурсів система може виділити для таких потоків у момент тестування.

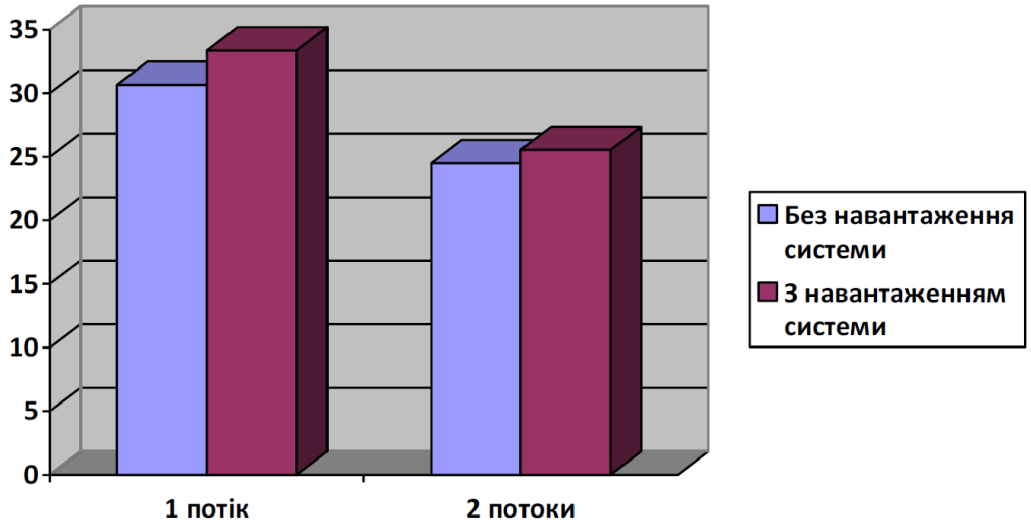


Рисунок 4.10 – Діаграма часу при обробці за допомогою Threads

Висновки:

1. Один потік є найбільш «легким» для системи, проте його обробка займає найбільше часу в порівнянні з двома потоками.

2. Два потоки дозволяють значно скоротити час обробки, особливо коли система не має додаткових навантажень, що робить цей метод більш привабливим для використання на більш потужних пристроях.

3. У умовах навантаження на систему, обробка в один потік стає менш ефективною, оскільки кожен додатковий ресурс (як пам'ять, так і CPU) займається іншими процесами, що підвищує час обробки.

4. За наявності додаткового навантаження на систему, збільшення кількості потоків до двох потужностей може зменшити час обробки, хоча й не значно.

Ці дані дозволяють зробити висновок, що для забезпечення ефективності обробки XML-файлів потрібно враховувати не лише кількість потоків, але й стан системи, включаючи навантаження від інших додатків.

4.2.2 Аналіз потрібного часу на завершення методів, реалізованих за допомогою RxJava

У цьому підрозділі проведено аналіз часу, необхідного для обробки XML-файлів за допомогою бібліотеки RxJava(див. рис. 4.11), яка реалізує асинхронне програмування та підтримує реактивні потоки. Оцінка часу виконання допомагає зрозуміти, як різні стратегії обробки з використанням RxJava можуть вплинути на ефективність виконання завдання.

Тестування було здійснене для різних варіантів кількості потоків, і зафіксовані середні значення часу, який необхідний для обробки даних:

- Один потік: Час обробки складає 30,26 секунд. У цьому варіанті використовується один потік, і обробка файлів проходить за допомогою RxJava без паралельного виконання завдань.

- Два потоки: Час обробки зменшується до 23,23 секунд. Використання двох потоків дає можливість обробляти дані паралельно, що знижує час виконання.

- Паралельні потоки: Час обробки знижується значно до 13,05 секунд. Це стало можливим завдяки використанню паралельних потоків в RxJava, що дозволяє досягти максимального прискорення при обробці великих обсягів даних.

- Один потік з додатковим навантаженням: Коли система перебуває під додатковим навантаженням, час обробки збільшується до 32,14 секунд. Важливо відзначити, що додаткові ресурсоємні процеси негативно впливають на продуктивність навіть при використанні одного потоку.

- Два потоки з додатковим навантаженням: Час обробки зменшується до 24,53 секунд. Два потоки все ще дозволяють дещо прискорити обробку, хоча й не досягають таких результатів, як у випадку без навантаження.

- Паралельні потоки з додатковим навантаженням: Час обробки складає 14,44 секунди, що є трохи повільніше, ніж при обробці без навантаження, але все одно значно швидше за інші варіанти.

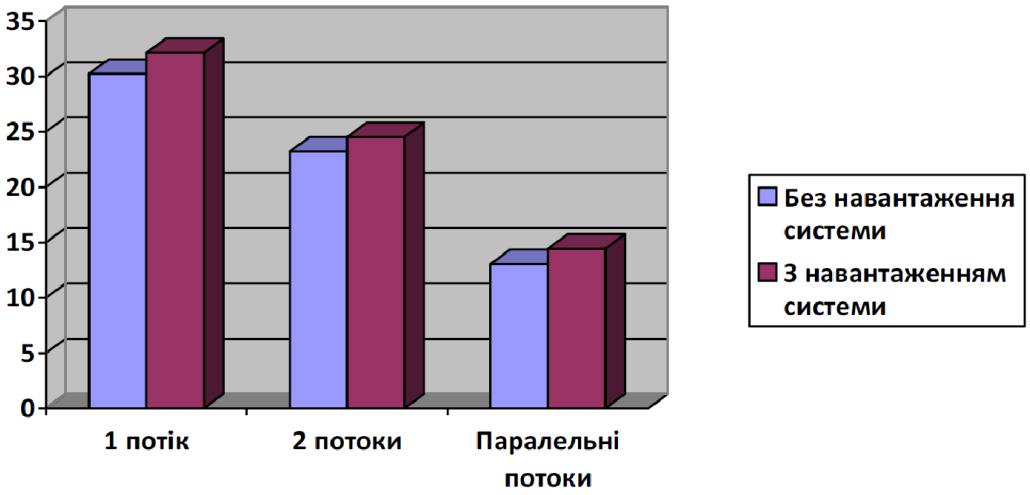


Рисунок 4.11 – Діаграма часу при обробці за допомогою RxJava

Висновки:

1. Один потік забезпечує базову обробку, але має найвищий час виконання серед усіх варіантів, оскільки немає паралелізму.

2. Два потоки та паралельні потоки дозволяють суттєво прискорити обробку, при цьому паралельні потоки демонструють найкращу ефективність.

3. У умовах додаткового навантаження на систему, всі варіанти обробки з додатковими потоками показують дещо повільніші результати, але в цілому RxJava з паралельними потоками залишаються найбільш ефективним методом.

4. RxJava демонструє високу ефективність завдяки паралельному виконанню завдань, і, незважаючи на додаткове навантаження на систему, цей метод залишається оптимальним для обробки великих XML-файлів, порівняно з іншими методами.

Ці результати допомагають визначити, що для швидкої обробки даних у реальних умовах, особливо при наявності навантаження на систему, метод з паралельними потоками у RxJava буде найбільш ефективним.

4.2.3 Аналіз потрібного часу на завершення методів, реалізованих за допомогою Kotlin Coroutines

Для аналізу методів обробки, реалізованих з використанням Kotlin Coroutines, було зафіксовано середні значення часу, необхідного для обробки XML-файлів з різною кількістю потоків. Kotlin Coroutines дозволяє організувати асинхронну обробку завдань із мінімальними витратами на управління потоками. Результати тестування представлені нижче (рис. 4.12):

- Один потік: Час обробки складає 30,09 секунд. Це базовий випадок, де використовується лише один потік для обробки всіх файлів, без паралелізму.

- Два потоки: Час обробки зменшується до 22,57 секунд. Використання двох потоків дозволяє зменшити час виконання завдяки паралельній обробці частини завдання.

- Чотири потоки: Час обробки становить 16,47 секунд. Чотири потоки значно знижують час виконання завдяки кращому використанню багатоядерних процесорів.

- Один потік з додатковим навантаженням: Коли система працює під додатковим навантаженням, час обробки збільшується до 31,99 секунд. Як і в інших методах, додаткові ресурсоємні процеси негативно впливають на загальну продуктивність.

- Два потоки з додатковим навантаженням: Час обробки зменшується до 23,45 секунд, що показує деяке зниження продуктивності через навантаження на систему, але все ж таки значне прискорення у порівнянні з одним потоком.

- Чотири потоки з додатковим навантаженням: Час обробки становить 18,57 секунд, що є оптимальним результатом серед варіантів з додатковим навантаженням. Вигоди від паралельної обробки зберігаються навіть при зменшеній системній продуктивності.

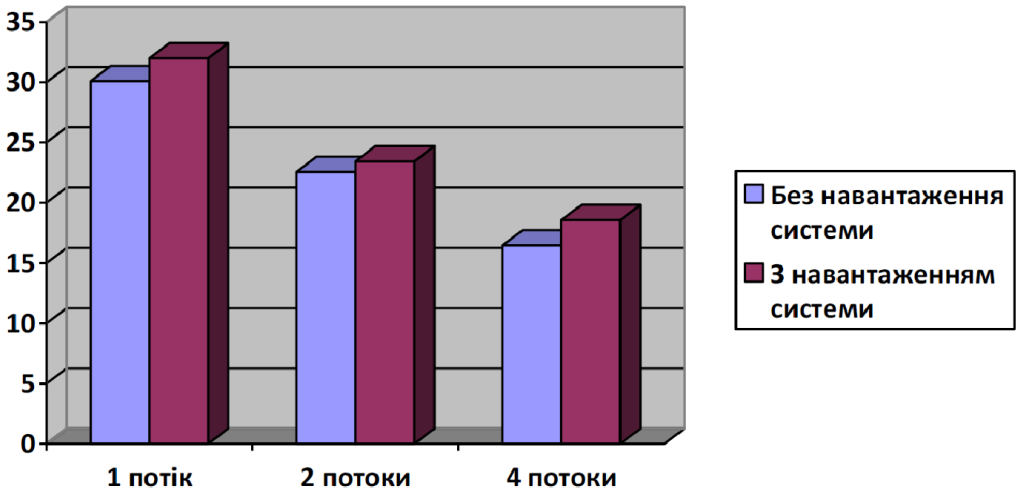


Рисунок 4.12 – Діаграма часу при обробці за допомогою Kotlin Coroutines

Висновки:

1. Один потік показує найбільший час виконання, що є очікуваним для однониткового виконання завдань.

2. Два потоки та чотири потоки значно покращують ефективність обробки, зокрема чотири потоки дають найбільше зменшення часу виконання.

3. Навіть у умовах додаткового навантаження на систему методи з більшою кількістю потоків (особливо чотири потоки) показують високу ефективність і суттєво знижують час обробки.

4. Використання Kotlin Coroutines дозволяє ефективно працювати з багатопоточністю, а також забезпечує гарну підтримку асинхронного виконання завдань при роботі з великими обсягами даних.

Ці результати підтверджують, що використання корутин є вигідним для обробки даних, коли потрібно зменшити час виконання завдань, а також забезпечити стійкість системи при високому навантаженні.

4.2.4 Загальне порівняння усіх методів за часом

З використанням всіх фіксованих тестових значень, що включають як завантажену систему, так і без навантаження, можна здійснити повне порівняння всіх реалізованих методів (див. рис. 4.13), а також трьох раніше обраних бібліотек: XMLPullParser, SAXParser, DOMParser. Це порівняння дозволяє оцінити, як кожен метод справляється з обробкою XML-файлів при різних умовах навантаження на систему.

Згідно з аналізом тимчасових витрат, що здійснюються кожним методом, можна зробити такі висновки:

- Чим більше потоків розподіляється, тим більше навантаження на систему, що може призвести до зменшення ефективності в умовах обмежених системних ресурсів.

- Проте, для методу RxJava з паралельними потоками спостерігається інша ситуація. Цей метод використовує функцію `parallelStream()`, яка дозволяє самій системі створювати необхідну кількість потоків в залежності від багатоядерності процесора. Це дає певні переваги на потужних пристроях, але не дає вигод на слабких пристроях, де багатоядерність процесора може бути обмежена.

- Вибір методу обробки XML повинен базуватися на конкретних умовах використання, таких як характеристики пристрою та вимоги до часу обробки даних.

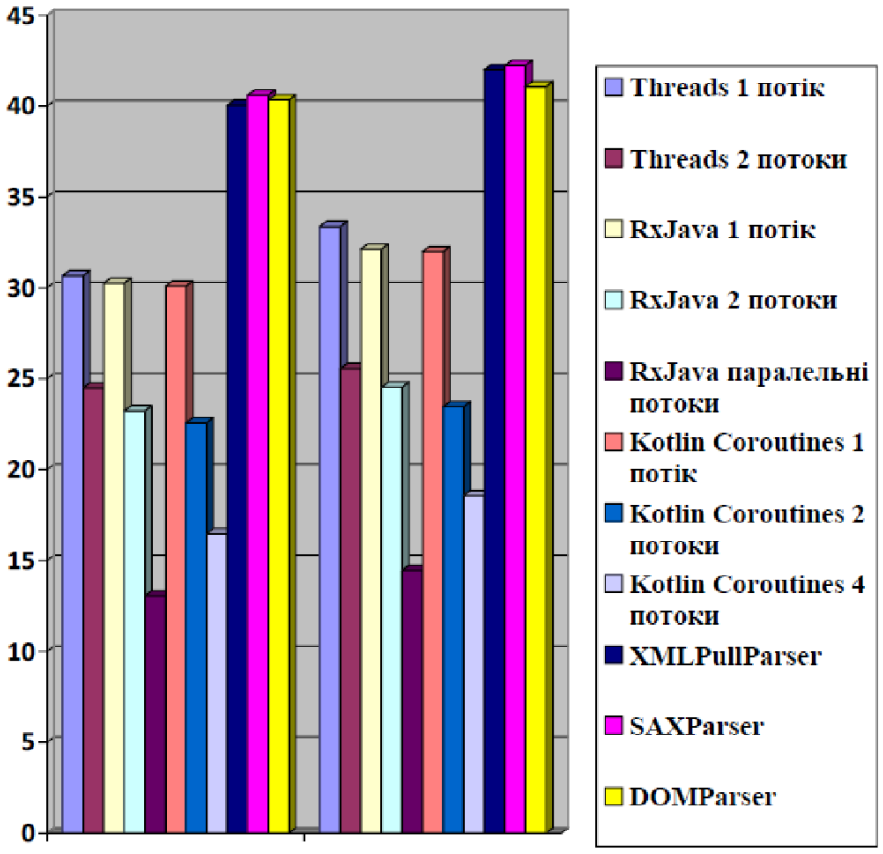


Рисунок 4.13 – Діаграма порівняння всіх реалізованих методів та трьох обраних існуючих бібліотек

Висновки:

1. Методи з меншим числом потоків забезпечують менше навантаження на систему і підходять для менш потужних пристроїв.

2. RxJava з паралельними потоками може бути дуже ефективним на потужних багатоядерних пристроях, але не дає суттєвих переваг на слабких пристроях, де можливості багатоядерності обмежені.

3. Для кожного конкретного випадку парсингу XML у додатку необхідно вибрати метод, який найкраще відповідає вимогам щодо швидкості обробки і доступних системних ресурсів.

Це дозволяє зробити вибір найбільш підходящого методу для парсингу XML-файлів залежно від вимог до продуктивності та обмежень апаратних засобів.

4.3 Створення алгоритму для визначення найоптимальнішого методу обробки даних

Після дослідження розроблених методів семантичного аналізу, збору всіх даних та порівняння їх з існуючими бібліотеками, можна створити алгоритм для пошуку найоптимальнішого способу обробки даних в додатку. Спершу варто визначити умови, які будуть застосовуватися для побудови алгоритму, і представити їх у вигляді блок-схеми.

Перший крок у алгоритмі — це питання: «Чи може алгоритм обробки навантажувати пристрій?» (див. рис. 4.14).

Рисунок 4.14 – Вхідна умова «Чи може алгоритм обробки навантажувати пристрій?»

Якщо відповідь «Ні», переходимо до наступної умови: «Чи є в пристрої багатоядерний процесор?» (див. рис. 4.15). Якщо відповідь «Так», можна використовувати метод RxParallelStreamWordCount, оскільки цей метод оптимально працює на багатоядерних пристроях, не навантажуючи систему.

Рисунок 4.15 – Умова «Чи наявний в девайсі багатоядерний процесор?»

Якщо відповідь на умову «Чи може алгоритм обробки навантажувати пристрій?» — «Ні», наступне питання буде: «Чи цікавить нас найшвидший

метод?» (див. рис. 4.16). Якщо відповідь «Так», вибираємо метод RxSequentialWordCount, оскільки він ідеально підходить для одноядерних процесорів та швидко виконує парсинг. Якщо відповідь «Ні», переходимо до питання «Чи великий за об'ємом файл?» (див. рис. 4.17). Якщо файл більше 200 МБ, використовуємо метод ThreadsSequentialWordCount. Для менших файлів можна обирати будь-яку з існуючих бібліотек: XMLPullParser, DOMParser, SAXParser.

Рисунок 4.16 – Умова «Чи цікавить нас найшвидший метод?»

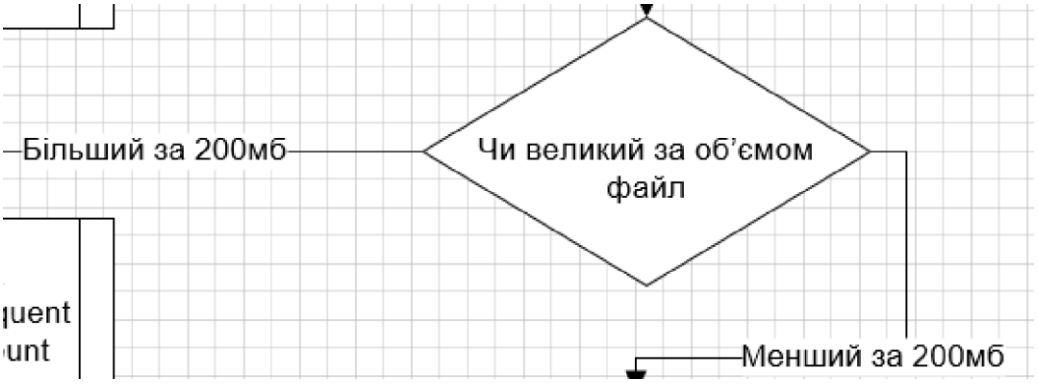


Рисунок 4.17 – Умова «Чи великий за об’ємом файл?»

Повертаємося до початкового питання «Чи може алгоритм обробки навантажувати пристрій?». Якщо відповідь «Так», далі проходимо через більш складний цикл:

Наступне питання — «Чи є в пристрої багатоядерний процесор?» (ця умова була раніше — див. рис. 4.15). Якщо відповідь «Ні», використовуємо метод

CoroutinesSequentialWordCount, який підходить для одноядерних процесорів, де система може бути навантажена.

Якщо відповідь «Так», далі з'ясовуємо: «Скільки пам'яті може виділити пристрій на один процес?» (див. рис. 4.18). Якщо більше 300 МБ, використовуємо метод CoroutinesBetterWordCount, який споживає багато ресурсів, але швидко виконує завдання. Якщо пам'яті менше — переходимо до наступної умови: «Кількість ядер» (див. рис. 4.19). Якщо кількість ядер більше 4, метод RxParallelStreamWordCount підходить, оскільки він може ефективно використовувати велику кількість ядер.

Рисунок 4.18 – Умова «Скільки пристрій може виділити пам’яті на один процес?»

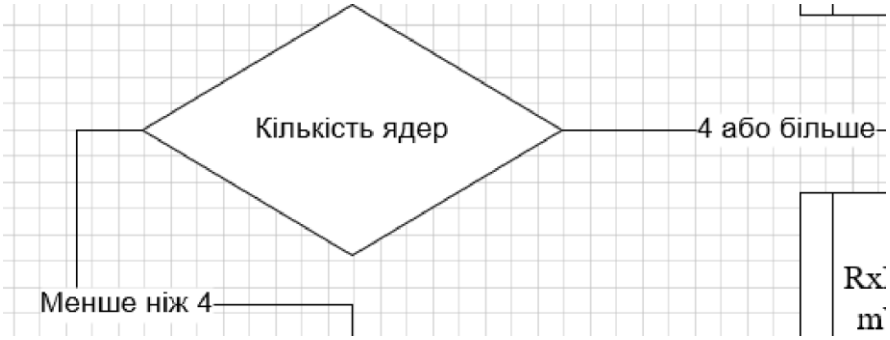


Рисунок 4.19 – Умова «Кількість ядер»

Якщо кількість ядер менше або дорівнює 4, наступне питання: «Чи навантажена система в даний момент?» (див. рис. 4.20). Якщо відповідь «Так», застосовуємо метод CoroutinesTwoThreadsWordCount, який краще працює при додатковому навантаженні на систему. Якщо відповідь «Ні», переходимо до питання «Заряд батареї» (див. рис. 4.21). Якщо заряд батареї менший за 30%,

використовуємо метод ThreadTwoThreadsWordCount, інакше — RxTwoThreadsWordCount, який споживає багато енергії, але виконує свою роботу дуже ефективно.

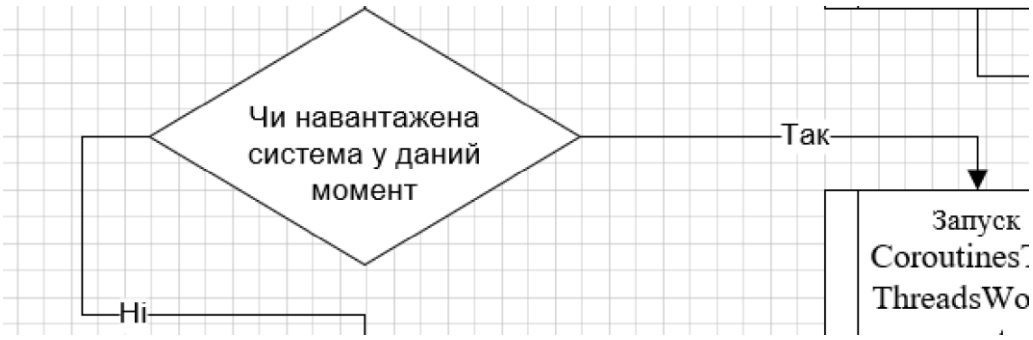


Рисунок 4.20 – Умова «Чи навантажена система у даний момент?»

Рисунок 4.21 – Умова «Заряд батареї»

Таким чином, створено алгоритм, який, проходячи через кілька умов, дозволяє визначити найоптимальніший метод для конкретного пристрою при конкретних умовах, забезпечуючи ефективну обробку даних з урахуванням усіх факторів, що впливають на продуктивність пристрою.

5. ЕКОНОМІЧНА ЧАСТИНА

Науково-технічна розробка має право на існування та впровадження, якщо вона відповідає вимогам часу, як в напрямку науково-технічного прогресу та і в плані економіки. Тому для науково-дослідної роботи необхідно оцінювати економічну ефективність результатів виконаної роботи.

Магістерська кваліфікаційна робота з розробки та дослідження «Мобільно-орієнтована система централізованої обробки даних розумного будинку» відноситься до науково-технічних робіт, які орієнтовані на виведення на ринок (або рішення про виведення науково-технічної розробки на ринок може бути прийнято у процесі проведення самої роботи), тобто коли відбувається так звана комерціалізація науково-технічної розробки. Цей напрямок є пріоритетним, оскільки результатами розробки можуть користуватися інші споживачі, отримуючи при цьому певний економічний ефект. Але для цього потрібно знайти потенційного інвестора, який би взявся за реалізацію цього проекту і переконати його в економічній доцільності такого кроку.

Для наведеного випадку нами мають бути виконані такі етапи робіт:

1) проведено комерційний аудит науково-технічної розробки, тобто встановлення її науково-технічного рівня та комерційного потенціалу;

2) розраховано витрати на здійснення науково-технічної розробки;

3) розрахована економічна ефективність науково-технічної розробки у випадку її впровадження і комерціалізації потенційним інвестором і проведено обґрунтування економічної доцільності комерціалізації потенційним інвестором.

5.1 Проведення комерційного та технологічного аудиту науково-технічної розробки

Метою проведення комерційного і технологічного аудиту дослідження за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку» є оцінювання науково-технічного рівня та рівня комерційного потенціалу розробки, створеної в результаті науково-технічної діяльності.

Оцінювання науково-технічного рівня розробки та її комерційного потенціалу рекомендується здійснювати із застосуванням 5-ти бальної системи оцінювання за 12-ма критеріями, наведеними в табл. 5.1 **[18]**.

Таблиця 5.1 – Рекомендовані критерії оцінювання науково-технічного рівня і комерційного потенціалу розробки та бальна оцінка

Бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Технічна здійсненність концепції					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено працездатність продукту в реальних умовах
Ринкові переваги (недоліки)					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає

Продовження таблиці 5.1					
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуютьс я у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь- які регламентні обмеження на виробництво та реалізацію продукту

Результати оцінювання науково-технічного рівня та комерційного потенціалу науково-технічної розробки потрібно звести до таблиці 5.2.

Таблиця 5.2 – Результати оцінювання науково-технічного рівня і комерційного потенціалу розробки експертами

Критерії	Експерт (ПІБ, посада)		
	1	2	3
	Бали:		
1. Технічна здійсненність концепції	5	4	4
2. Ринкові переваги (наявність аналогів)	4	3	3
3. Ринкові переваги (ціна продукту)	2	2	3
Таблиця 4.2 - продовження			
4. Ринкові переваги (технічні властивості)	3	3	3
5. Ринкові переваги (експлуатаційні витрати)	3	3	3
6. Ринкові перспективи (розмір ринку)	2	2	2
7. Ринкові перспективи (конкуренція)	3	3	3
8. Практична здійсненність (наявність фахівців)	4	4	4
9. Практична здійсненність (наявність фінансів)	2	3	2
10. Практична здійсненність (необхідність нових матеріалів)	2	2	2
11. Практична здійсненність (термін реалізації)	3	4	4
12. Практична здійсненність (розробка документів)	4	4	4
Сума балів	37	37	37
Середньоарифметична сума балів $СБ_с$	37,0		

За результатами розрахунків, наведених в таблиці 5.2, зробимо висновок щодо науково-технічного рівня і рівня комерційного потенціалу розробки. При цьому використаємо рекомендації, наведені в табл. 5.3 **[18]**.

Таблиця 5.3 – Науково-технічні рівні та комерційні потенціали розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Науково-технічний рівень та комерційний потенціал розробки
41…48	Високий
31…40	Вище середнього
21…30	Середній
11…20	Нижче середнього
0…10	Низький

Згідно проведених досліджень рівень комерційного потенціалу розробки за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку» становить 37,0 бала, що, відповідно до таблиці 5.3, свідчить про

комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вище середнього).

5.2 Розрахунок узагальненого коефіцієнта якості розробки

Окрім комерційного аудиту розробки доцільно також розглянути технічний рівень якості розробки, розглянувши її основні технічні показники. Ці показники по-різному впливають на загальну якість проектної розробки.

Узагальнений коефіцієнт якості ($B_н$) для нового технічного рішення розрахуємо за формулою **[19]**:

$$B_н = \sum_{i=1}^{k} \alpha_i \cdot \beta_i \quad , \tag{5.1}$$

де k – кількість найбільш важливих технічних показників, які впливають на якість нового технічного рішення;

α_i – коефіцієнт, який враховує питому вагу *і-го* технічного показника в загальній якості розробки. Коефіцієнт α_i визначається експертним шляхом і при цьому має виконуватись умова $\sum_{i=1}^{k} \alpha_i = 1$;

β_i – відносне значення *і-го* технічного показника якості нової розробки.

Відносні значення β_i для різних випадків розраховуємо за такими формулами:

- для показників, зростання яких вказує на підвищення в лінійній залежності якості нової розробки:

$$\beta_i = \frac{I_{нi}}{I_{ai}} \quad , \tag{5.2}$$

де $I_{нi}$ та $I_{на}$ – чисельні значення конкретного *і*-го технічного показника якості відповідно для нової розробки та аналога;

- для показників, зростання яких вказує на погіршення в лінійній залежності якості нової розробки:

$$\beta_i = \frac{I_{ai}}{I_{ni}}; \quad (5.3)$$

Використовуючи наведені залежності можемо проаналізувати та порівняти техніко-економічні характеристики аналогу та розробки на основі отриманих наявних та проектних показників, а результати порівняння зведемо до таблиці 5.4.

Таблиця 5.4 – Порівняння основних параметрів розробки та аналога.

Показники (параметри)	Одиниця вимірювання	Аналог	Проектований пристрій	Відношення параметрів нової розробки до аналога	Питома вага показника
Швидкодія	с	0,8	0,4	2	0,1
Швидкість доступу до даних	мкс	210	140	1,5	0,4
Кількість обробки запитів на хвилину	шт.	60	100	1,67	0,3
Місце на диску	Мб	1200	200	6	0,1
Потреба в оперативній пам’яті	Гб	6	4	1,5	0,1

Узагальнений коефіцієнт якості ($B_н$) для нового технічного рішення складе:

$$B_н = \sum_{i=1}^{k} \alpha_i \cdot \beta_i = 2 \cdot 0,1 + 1,5 \cdot 0,4 + 1,67 \cdot 0,3 + 6 \cdot 0,1 + 1,5 \cdot 0,1 = 2,05.$$

Отже за технічними параметрами, згідно узагальненого коефіцієнту якості розробки, науково-технічна розробка переважає існуючі аналоги приблизно в 2,05 рази.

5.3 Розрахунок витрат на проведення науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи на тему «Мобільно-орієнтована система централізованої обробки даних розумного будинку», під час планування, обліку і калькулювання собівартості науково-дослідної роботи групуємо за відповідними статтями.

5.3.1 Витрати на оплату праці

До статті «Витрати на оплату праці» належать витрати на виплату основної та додаткової заробітної плати керівникам відділів, лабораторій, секторів і груп, науковим, інженерно-технічним працівникам, конструкторам, технологам, креслярам, копіювальникам, лаборантам, робітникам, студентам, аспірантам та іншим працівникам, безпосередньо зайнятим виконанням конкретної теми, обчисленої за посадовими окладами, відрядними розцінками, тарифними ставками згідно з чинними в організаціях системами оплати праці.

Основна заробітна плата дослідників

Витрати на основну заробітну плату дослідників ($З_о$) розраховуємо у відповідності до посадових окладів працівників, за формулою **[18]**:

$$З_о = \sum_{i=1}^{k} \frac{M_{ni} \cdot t_i}{T_p}, \quad (5.4)$$

де k – кількість посад дослідників залучених до процесу досліджень;

M_{ni} – місячний посадовий оклад конкретного дослідника, грн;

t_i – число днів роботи конкретного дослідника, дн.;

T_p – середнє число робочих днів в місяці, T_p=21 дні.

$$З_о = 22000{,}00 \cdot 10 / 21 = 10476{,}19 \ грн.$$

Проведені розрахунки зведемо до таблиці 5.5.

Таблиця 5.5 – Витрати на заробітну плату дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на заробітну плату, грн
Проектний менеджер (координатор)	22000,00	1047,62	10	10476,19
Інженер-розробник автоматизованих систем управління	17650,00	840,48	15	12607,14
Інженер-розробник програмного забезпечення	18000,00	857,14	12	10285,71
Технік	8150,00	388,10	10	3880,95
Всього				37250,00

Основна заробітна плата робітників

Витрати на основну заробітну плату робітників (див. таб. 5.6) ($З_р$) за відповідними найменуваннями робіт НДР на тему «Мобільно-орієнтована система централізованої обробки даних розумного будинку» розраховуємо за формулою:

$$З_р = \sum_{i=1}^{n} C_i \cdot t_i \quad , \tag{5.5}$$

де C_i – погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

t_i – час роботи робітника при виконанні визначеної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду C_i можна визначити за формулою:

$$C_i = \frac{M_M \cdot K_i \cdot K_c}{T_p \cdot t_{зм}} \quad , \tag{5.6}$$

де M_M – розмір прожиткового мінімуму працездатної особи, або мінімальної місячної заробітної плати (в залежності від діючого законодавства), приймемо *Мм*=8000,00 грн;

K_i – коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду **[18]**;

K_c – мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих об'єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати.

T_p – середнє число робочих днів в місяці, приблизно T_p = 21 дн;

$t_{зм}$ – тривалість зміни, год.

$C_1 = 8000{,}00 \cdot 1{,}10 \cdot 1{,}15 / (21 \cdot 8) = 60{,}24$ *грн.*
$З_{р1} = 60{,}24 \cdot 7{,}90 = 475{,}88$ *грн.*

Таблиця 5.6 – Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Тарифний коефіцієнт	Погодинна тарифна ставка, грн	Величина оплати на робітника грн
Установка обладнання для моделювання та проектування обчислювальної	7,90	2	1,10	60,24	475,88

Продовження таблиці 5.6					
Підготовка робочого місця розробника програмного	4,50	3	1,35	73,93	332,68
Продовження 4.6					
Підготовка серверного обладнання	3,15	4	1,50	82,14	258,75
Інсталяція програмного забезпечення для моделювання та розробки обчислювальної системи	6,44	4	1,50	82,14	529,00
Компіляція програмних блоків	8,12	5	1,70	93,10	755,93
Налагодження програмних блоків	3,60	6	2,00	109,52	394,29
Тестування системи	9,50	2	1,10	60,24	572,26
Всього					3318,79

Додаткова заробітна плата дослідників та робітників

Додаткову заробітну плату розраховуємо як 10 … 12% від суми основної заробітної плати дослідників та робітників за формулою:

$$З_{дод} = (З_о + З_р) \cdot \frac{Н_{дод}}{100\%}, \quad (5.7)$$

де $Н_{дод}$ – норма нарахування додаткової заробітної плати. Приймемо 11%.

$З_{дод}$ = (37250,00 + 3318,79) · 11 / 100% = 4462,57 грн.

5.3.2 Відрахування на соціальні заходи

Нарахування на заробітну плату дослідників та робітників розраховуємо як 22% від суми основної та додаткової заробітної плати дослідників і робітників за формулою:

$$З_н = (З_о + З_р + З_{дод}) \cdot \frac{Н_{зп}}{100\%} \quad (5.8)$$

де $Н_{зп}$ – норма нарахування на заробітну плату. Приймаємо 22%.

Зн = (37250,00 + 3318,79 + 4462,57) · 22 / 100% = 9906,90 грн.

5.3.3 Сировина та матеріали

До статті «Сировина та матеріали» належать витрати на сировину, основні та допоміжні матеріали, інструменти, пристрої та інші засоби і предмети праці (див. таб. 5.7), які придбані у сторонніх підприємств, установ і організацій та витрачені на проведення досліджень за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку».

Витрати на матеріали (*М*), у вартісному вираженні розраховуються окремо по кожному виду матеріалів за формулою:

$$M = \sum_{j=1}^{n} H_{вj} \cdot Ц_j \cdot K_j - \sum_{j=1}^{n} B_j \cdot Ц_{вj}, \quad (5.9)$$

де H_j – норма витрат матеріалу *j*-го найменування, кг;

n – кількість видів матеріалів;

$Ц_j$ – вартість матеріалу *j*-го найменування, грн/кг;

K_j – коефіцієнт транспортних витрат, (K_j = *1,1 ... 1,15*);

B_j – маса відходів *j*-го найменування, кг;

$Ц_{вj}$ – вартість відходів *j*-го найменування, грн/кг.

$М_1$ = 1 · 202,00 · 1,02 - 0 · 0 = 206,04 грн.

Проведені розрахунки зведемо до таблиці.

Таблиця 5.7 – Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за 1 кг, грн	Норма витрат, кг	Величина відходів, кг	Ціна відходів, грн/кг	Вартість витраченого матеріалу, грн
Офісний папір Canon 500/80 A4	202,00	1	0	0	206,04
Папір для записів Canon Parers A5	92,00	2	0	0	187,68
Органайзер офісний BUROMAX	205,00	2	0	0	418,20
Канцелярське приладдя	215,00	2	0	0	438,60

Продовження таблиці 5.7					
Картридж для принтера Canon i-sensys LBP6500	1129,00	1	0	0	1151,58
Тека для паперів BUROMAX	127,00	1	0	0	129,54
Flesh-пам'ять Kingston Data Traveler 128 GB	295,00	1	0	0	300,90
Тека для паперів	45,00	6	0	0	275,40
Інші матеріали	148,00	1	0	0	150,96
Всього					3258,90

5.3.4 Розрахунок витрат на комплектуючі

Витрати на комплектуючі *(К$_в$)*,(див. таб 5.8) які використовують при проведенні НДР на тему «Мобільно-орієнтована система централізованої обробки даних розумного будинку», розраховуємо, згідно з їхньою номенклатурою, за формулою:

$$К_в = \sum_{j=1}^{n} Н_j \cdot Ц_j \cdot К_j \tag{5.10}$$

де $Н_j$ – кількість комплектуючих *j*-го виду, шт.;

$Ц_j$ – покупна ціна комплектуючих *j*-го виду, грн;

$К_j$ – коефіцієнт транспортних витрат, ($К_j$ = *1,1 ... 1,15*).

$К_в = 1 \cdot 4420,00 \cdot 1,02 = 4508,40$ грн.

Проведені розрахунки зведемо до таблиці.

Таблиця 5.8 – Витрати на комплектуючі

Найменування комплектуючих	Кількість, шт.	Ціна за штуку, грн	Сума, грн
RouterBOARD TP-Link230	1	4420,00	4508,40
База даних (тестувальна)	1	3250,00	3315,00
Всього			7823,40

5.3.5 Спецустаткування для наукових (експериментальних) робіт

До статті «Спецустаткування для наукових (експериментальних) робіт» належать витрати на виготовлення та придбання спецустаткування необхідного для проведення досліджень (див. таб. 5.9), також витрати на їх проектування, виготовлення, транспортування, монтаж та встановлення.

Балансову вартість спецустаткування розраховуємо за формулою:

$$B_{спец} = \sum_{i=1}^{k} Ц_i \cdot C_{пр.i} \cdot K_i \quad , \tag{5.11}$$

де $Ц_i$ – ціна придбання одиниці спецустаткування даного виду, марки, грн;

$C_{пр.i}$ –кількість одиниць устаткування відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує доставку, монтаж, налагодження устаткування тощо, (K_i = *1,10…1,12);*

k – кількість найменувань устаткування.

$B_{спец} = 35899{,}00 \cdot 1 \cdot 1{,}02 = 36616{,}98$ *грн.*

Отримані результати зведемо до таблиці 5.9:

Таблиця 5.9 – Витрати на придбання спецустаткування по кожному виду

Найменування устаткування	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Серверне обладнання на основі ПК ZEVS PC 13430U i5 9400F + GTX 1060 3GB	1	35899,00	36616,98
Засоби передачі даних	1	6230,00	6354,60
Всього			42971,58

5.3.6 Програмне забезпечення для наукових (експериментальних) робіт

До статті «Програмне забезпечення для наукових (експериментальних) робіт» належать витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення, (програм, алгоритмів, баз даних) необхідних для проведення досліджень, також витрати на їх проектування, формування та встановлення.

Балансову вартість програмного забезпечення розраховуємо за формулою:

$$B_{прг} = \sum_{i=1}^{k} Ц_{iпрг} \cdot C_{прг.i} \cdot K_i \quad , \tag{5.12}$$

де $Ц_{iпрг}$ – ціна придбання одиниці програмного засобу даного виду, грн;

$C_{прг.i}$ – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо, (K_i = *1,10…1,12);*

k – кількість найменувань програмних засобів.

$B_{прг} = 5320,00 \cdot 1 \cdot 1,02 = 5426,40$ *грн.*

Отримані результати зведемо до таблиці 5.10:

Таблиця 5.10 – Витрати на придбання програмних засобів по кожному виду

Найменування програмного засобу	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Прикладне програмне забезпечення системи управління реляційними базами даних MySQL Workbench	1	5320,00	5426,40
Емулятор серверної інтернет-платформи для моделювання обробки даних	1	6520,00	6650,40
SQL Server 2016	1	45,00	45,90
Всього			12122,70

5.3.7 Амортизація обладнання, програмних засобів та приміщень

В спрощеному вигляді амортизаційні відрахування по кожному виду обладнання, приміщень та програмному забезпеченню тощо, розраховуємо з використанням прямолінійного методу амортизації за формулою:

$$A_{обл} = \frac{Ц_б}{T_в} \cdot \frac{t_{вик}}{12} \quad , \tag{5.13}$$

де $Ц_б$ – балансова вартість обладнання, програмних засобів, приміщень тощо, які використовувались для проведення досліджень, грн;

$t_{вик}$ – термін використання обладнання, програмних засобів, приміщень під час досліджень, місяців;

$T_в$ – строк корисного використання обладнання, програмних засобів, приміщень тощо, років.

$A_{обл} = (36999,00 \cdot 1) / (3 \cdot 12) = 1027,75$ *грн.*

Проведені розрахунки зведемо до таблиці 5.11.

Таблиця 5.11 – Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Персональний комп'ютер розробника ПЗ (Ноутбук Acer Nitro 5 AN515-58 (NH.QMZEU.006) Obsidian Black)	36999,00	3	1	1027,75
Смартфон Xiaomi Redmi 12 8/256Gb Midnight Black	6799,00	3	1	188,86
Пристрої виводу графічної інформації	7770,00	4	1	161,88
Оргтехніка	7320,00	5	1	122,00
Приміщення лабораторії розробки програмного забезпечення	410000,0 0	25	1	1366,67
Робоче місце розробника	7320,00	4	1	152,50
Пристрої передачі цифрової інформації (Маршрутизатор інтернет WiFi6 TP-Link Archer AX12)	6500,00	5	1	108,33
ОС Windows 10	4260,00	3	1	118,33
Прикладний пакет Microsoft Office 2019	4800,00	3	1	133,33
Всього				3379,65

5.3.8 Паливо та енергія для науково-виробничих цілей

Витрати на силову електроенергію (B_e) розраховуємо за формулою:

$$B_e = \sum_{i=1}^{n} \frac{W_{yi} \cdot t_i \cdot Ц_e \cdot K_{вni}}{\eta_i}, \quad (5.14)$$

де W_{yi} – встановлена потужність обладнання на визначеному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_e$ – вартість 1 кВт-години електроенергії, грн; (вартість електроенергії визначається за даними енергопостачальної компанії), приймемо $Ц_e$ = 10,98 грн;

$K_{вni}$ – коефіцієнт, що враховує використання потужності, $K_{вni} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

$B_e = 0,25 \cdot 160,0 \cdot 10,98 \cdot 0,95 / 0,97 = 439,20$ *грн.*

Проведені розрахунки зведемо до таблиці 5.12

Таблиця 5.12 – Витрати на електроенергію

Найменування обладнання	Встановлена потужність, кВт	Тривалість роботи, год	Сума, грн
Персональний комп'ютер розробника ПЗ (Ноутбук Acer Nitro 5 AN515-58 (NH.QMZEU.006) Obsidian Black)	0,25	160,0	439,20
Смартфон Xiaomi Redmi 12 8/256Gb Midnight Black	0,01	20,0	2,20
Пристрої виводу графічної інформації	0,10	5,7	6,26
Оргтехніка	0,32	1,5	5,27
Робоче місце розробника	0,02	160,0	35,14
Пристрої передачі цифрової інформації (Маршрутизатор інтернет WiFi6 TP-Link Archer AX12)	0,02	160,0	35,14
Серверне обладнання на основі ПК ZEVS PC 13430U i5 9400F + GTX	0,32	160,0	562,18

Продовженя таблиці 5.12			
Засоби передачі даних	0,02	160,0	35,14
Всього			1120,51

5.3.9 Службові відрядження

До статті «Службові відрядження» дослідної роботи на тему «Мобільно-орієнтована система централізованої обробки даних розумного будинку» належать витрати на відрядження штатних працівників, працівників організацій, які працюють за договорами цивільно-правового характеру, аспірантів, зайнятих розробленням досліджень, відрядження, пов'язані з проведенням випробувань машин та приладів, а також витрати на відрядження на наукові з'їзди, конференції, наради, пов'язані з виконанням конкретних досліджень.

Витрати за статтею «Службові відрядження» розраховуємо як 20…25% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{св} = (З_о + З_р) \cdot \frac{H_{св}}{100\%}, \quad (5.15)$$

де $H_{св}$ – норма нарахування за статтею «Службові відрядження», приймемо $Н_{св}$ = 20%.

$B_{св} = (37250,00 + 3318,79) \cdot 20 / 100\% = 8113,76$ *грн.*

5.3.10 Витрати на роботи, які виконують сторонні підприємства, установи і організації

Витрати за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації» розраховуємо як 30…45% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{сп} = (З_о + З_р) \cdot \frac{H_{сп}}{100\%}, \quad (5.16)$$

де $Н_{сп}$ – норма нарахування за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації», приймемо $Н_{сп}$= 30%.

$B_{сп} = (37250,00 + 3318,79) \cdot 30 / 100\% = 12170,64$ *грн.*

5.3.11 Інші витрати

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками.

Витрати за статтею «Інші витрати» розраховуємо як 50…100% від суми основної заробітної плати дослідників та робітників за формулою:

$$I_в = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%}, \quad (5.17)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати», приймемо $Н_{ів}$ = 50%.

$I_в$ = (37250,00 + 3318,79) · 50 / 100% = 20284,40 грн.

5.3.12 Накладні (загальновиробничі) витрати

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов'язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов'язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов'язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін.

Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуємо як 100…150% від суми основної заробітної плати дослідників та робітників за формулою:

$$В_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (5.18)$$

де $Н_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати», приймемо $Н_{нзв}$ = 100%.

$В_{нзв}$ = (37250,00 + 3318,79) · 100 / 100% = 40568,79 грн.

Витрати на проведення науково-дослідної роботи на тему «Мобільно-орієнтована система централізованої обробки даних розумного будинку» розраховуємо як суму всіх попередніх статей витрат за формулою:

$$В_{заг} = З_о + З_р + З_{дод} + З_н + М + К_в + В_{спец} + В_{прг} + А_{обл} + В_е + В_{св} + В_{сп} + І_в + В_{нзв}. \quad (5.19)$$

$B_{заг}$= *37250,00 +3318,79 +4462,57 +9906,90 +3258,90 +7823,40 +42971,58 + 12122,70 + 3379,65 +1120,51 +8113,76 +12170,64 +20284,40 +40568,79 = 206752,58 грн.*

Загальні витрати *ЗВ* на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів розраховується за формулою:

$$ЗВ = \frac{B_{заг}}{\eta}, \quad (5.20)$$

де η - коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи, приймемо η=0,9.

ЗВ = 206752,58 / 0,9 = 229725,09 грн.

5.4 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором

В ринкових умовах узагальнюючим позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки, є збільшення у потенційного інвестора величини чистого прибутку.

Результати дослідження проведені за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку» передбачають комерціалізацію протягом 4-х років реалізації на ринку.

В цьому випадку майбутній економічний ефект буде формуватися на основі таких даних:

ΔN – збільшення кількості споживачів продукту, у періоди часу, що аналізуються, від покращення його певних характеристик;

Показник	1-й рік	2-й рік	3-й рік	4-й рік
Збільшення кількості споживачів, осіб	500	1000	1200	900

N – кількість споживачів які використовували аналогічний продукт у році до впровадження результатів нової науково-технічної розробки, приймемо 10000 осіб;

$Ц_б$ – вартість програмного продукту у році до впровадження результатів розробки, приймемо 1250,00 грн;

$\pm\Delta Ц_о$ – зміна вартості програмного продукту від впровадження результатів науково-технічної розробки, приймемо 313,75 грн.

Можливе збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ для кожного із 4-х років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховуємо за формулою **[18]:**

$$\Delta П_i = (\pm\Delta Ц_о \cdot N + Ц_о \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \quad (5.21)$$

де λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2024 році ставка податку на додану вартість складає 20%, а коефіцієнт $\lambda = 0,8333$;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту). Приймемо ρ =38%;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2024 році $\vartheta = 18\%$;

Збільшення чистого прибутку 1-го року:

$\Delta П_1$ =(313,75·10000,00+1563,75·500)·0,83·0,38·(1-0,18/100%)=1013660,12 грн.

Збільшення чистого прибутку 2-го року:

$\Delta П_2$ =(313,75·10000,00+1563,75·1500)·0,83·0,38·(1-0,18/100%)=1418089,65 грн.

Збільшення чистого прибутку 3-го року:

$\Delta П_3 =$(313,75·10000,00+1563,75·2700)·0,83·0,38·(1-0,18/100%)=1903405,09 грн.

Збільшення чистого прибутку 4-го року:

$\Delta П_4 =$(313,75·10000,00+1563,75·3600)·0,83·0,38·(1-0,18/100%)=2267391,68 грн.

Приведена вартість збільшення всіх чистих прибутків *ПП*, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.22)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн;

T – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ =0,15;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

ПП =1013660,12/(1+0,15)^1+1418089,65/(1+0,15)^2+1903405,09/(1+0,15)^3+
+2267391,68/(1+0,15)^4=881443,58+1072279,51+1251519,75+1296388,55=4501631,39 грн.

Величина початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки:

$$PV = k_{інв} \cdot \qquad , \qquad (5.23)$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, приймаємо $k_{інв}$=2;

$ЗВ$ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, приймаємо 229725,09 грн.

$ВВ = k_{інв} \cdot$ $= 2 \cdot 229725{,}09 = 459450{,}18$ грн.

Абсолютний економічний ефект $E_{абс}$ для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV \tag{5.24}$$

де $ПП$ – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, 4501631,39 грн;

PV – теперішня вартість початкових інвестицій, 459450,18 грн.

$E_{абс} = ПП - PV$ = 4501631,39 - 459450,18 = 4042181,21 грн.

Внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$E_в = \sqrt[Tж]{1 + \frac{E_{абс}}{PV}} - 1, \tag{5.25}$$

де $E_{абс}$ – абсолютний економічний ефект вкладених інвестицій, 4042181,21 грн;

PV – теперішня вартість початкових інвестицій, 459450,18 грн;

$T_{ж}$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, 4 роки.

$$E_в = \sqrt[Tж]{1 + \frac{E_{абс}}{PV}} - 1 = (1+4042181,21/459450,18)^{1/4} = 0,77.$$

Мінімальна внутрішня економічна дохідність вкладених інвестицій $\tau_{мін}$:

$$\tau_{мін} = d + f, \quad (5.26)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2024 році в Україні d =0,12;

f – показник, що характеризує ризикованість вкладення інвестицій, приймемо 0,15.

$\tau_{мін}$ = 0,12+0,15 = 0,27 < 0,77 свідчить про те, що внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки вища мінімальної внутрішньої дохідності. Тобто інвестувати в науково-дослідну роботу за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку» доцільно.

Період окупності інвестицій $T_{ок}$ які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$T_{ок} = \frac{1}{E_в}, \quad (5.27)$$

де $E_в$ – внутрішня економічна дохідність вкладених інвестицій.

$T_{ок}$ = 1 / 0,77 = 1,30 р.

$T_{ок} <$ 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

Висновки до розділу

Згідно проведених досліджень рівень комерційного потенціалу розробки за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку» становить 37,0 бала, що, свідчить про комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вище середнього).

При оцінюванні за технічними параметрами, згідно узагальненого коефіцієнту якості розробки, науково-технічна розробка переважає існуючі аналоги приблизно в 2,05 рази.

Також термін окупності становить 1,30 р., що менше 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

Отже можна зробити висновок про доцільність проведення науково-дослідної роботи за темою «Мобільно-орієнтована система централізованої обробки даних розумного будинку».

ВИСНОВКИ

У цій магістерській роботі розглянуто важливість та перспективи використання XML-файлів у контексті мобільних операційних систем, зокрема на платформі Android, а також роль таких технологій у розвитку інтернету речей (IoT). XML файли є важливим інструментом для зберігання та передачі структурованих даних, що широко використовуються в мобільних додатках, особливо в IoT-системах, де часто потрібна обробка великих обсягів даних у реальному часі. З огляду на обмежені ресурси мобільних пристроїв, особливо в IoT-гаджетах, обробка таких файлів повинна бути ефективною і мінімізувати навантаження на систему.

Однією з головних проблем, що виникають при роботі з великими XML-документами на мобільних пристроях, є їх парсинг, оскільки стандартні методи обробки можуть бути надто ресурсоємними для пристроїв з обмеженими можливостями. Для вирішення цієї проблеми було запропоновано використання багатопоточних технологій, таких як RxJava та Kotlin Coroutines, що дозволяє ефективно розподіляти обчислювальні ресурси між кількома потоками та знижувати навантаження на систему.

Розроблений SDK для семантичного аналізу XML-файлів на платформі Android дозволяє використовувати різні методи багатопотокової обробки, що допомагає оптимізувати продуктивність мобільних додатків у контексті IoT. Зокрема, цей SDK містить методи для обробки XML-файлів в одному, двох та чотирьох потоках, а також методи на основі RxJava та parallelStream, що дає можливість вибору найбільш підходящого варіанту в залежності від конкретних умов.

У результаті проведених тестів було встановлено, що кожен з методів обробки має свої переваги і підходить для різних сценаріїв використання. Наприклад, методи з меншим числом потоків (один або два потоки) показали кращі результати на пристроях з обмеженими ресурсами, таких як IoT-гаджети з низькою потужністю або маленьким обсягом пам'яті. У той же час, методи з більшими

кількостями потоків дозволяють значно зменшити час обробки при наявності більш потужних пристроїв, таких як сучасні смартфони або планшети.

Особливу увагу в роботі було приділено розробці алгоритму для визначення оптимального методу обробки в залежності від таких факторів, як навантаження на систему, кількість доступних ядер процесора, обсяг пам'яті та заряд батареї. Це дозволяє вибрати найбільш ефективний метод в кожному конкретному випадку, що особливо важливо для мобільних додатків в екосистемі IoT, де пристрої часто працюють у змінних умовах, з різними параметрами ресурсу та обмеженнями на батарею.

В результаті, розроблений алгоритм дозволяє оптимізувати використання системних ресурсів, зменшити час обробки даних та покращити загальну ефективність мобільних додатків, що працюють з великими XML-файлами в контексті IoT. Таким чином, робота демонструє важливість правильної організації багатопотокової обробки для досягнення ефективної роботи з великими обсягами даних в обмежених умовах мобільних пристроїв, що є критично важливим для інтернету речей та його розвитку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Popov O., Yatsyshyn A., and Ivanov I., "Designing a Smart Home System Based on IoT Technologies," Electronics and Control Systems, vol. 2, no. 60, pp. 80-88, 2021. [Online]. Available: https://doi.org/10.18372/1990-5548.60.15633. (9 стор.)

2. Smith M. and Johnson K., "Mobile-Oriented Cloud-Based Smart Home Data Processing," IEEE Access, vol. 8, pp. 24012-24022, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2973345. (11 стор.)

3. Borysov P. , "Internet of Things and Mobile Application Design for Smart Home Data Management," Scientific Bulletin of the National Technical University of Ukraine, no. 5, pp. 45-52, 2021. (8 стор.)

4. Kumar S. and Verma A., "Energy Efficient Data Processing in IoT-Based Smart Homes," Journal of Cleaner Production, vol. 271, pp. 122-132, 2021. [Online]. Available: https://doi.org/10.1016/j.jclepro.2020.122385. (11 стор.)

5. Ivashko A., "The Development of Mobile Applications for Smart Home Automation," Proceedings of the International Conference on Advanced Information and Communication Technologies, pp. 78-83, 2020. (6 стор.)

6. Zhou Y. , Xie C., and Chen Y. , "Cloud Computing for Smart Home Data Processing: A Mobile-Centric Approach," IEEE Transactions on Cloud Computing, vol. 9, no. 1, pp. 65-75, 2021. [Online]. Available: https://doi.org/10.1109/TCC.2020.3012332. (11 стор.)

7. Козлова О., Селезньов О., та Шевченко І., "Інтернет речей та централізована обробка даних у системах розумного будинку," Вісник Київського національного університету технологій та дизайну, вип. 4, с. 40-45, 2022. (6 стор.)

8. Rybalko V. , "Mobile App Development for IoT-Based Smart Home Systems," Journal of Automation and Information Sciences, vol. 52, no. 2, pp. 95-104, 2020. [Online]. Available: https://doi.org/10.1615/JAutomatInfScien.v52.i2.9020. (10 стор.)

9. Чернов А., "Централізоване управління та обробка даних у системах розумного будинку на основі мобільних додатків," Науковий журнал НТУУ "КПІ", вип. 3, с. 25-32, 2021. (8 стор.)

10. Singh M. K., "Centralized Data Management in Smart Homes with Cloud Integration," Sensors, vol. 21, no. 8, pp. 254-266, 2021. [Online]. Available: https://doi.org/10.3390/s2108254. (13 стор.)

11. Shvets O., "Mobile-Oriented Cloud Systems for Smart Home Monitoring," Modern Information Systems, no. 4, pp. 112-121, 2020. (10 стор.)

12. Patel T. and Patel P., "Smart Home Automation Using Mobile-Based Cloud Systems," Proceedings of the 2022 International Conference on Data Science and Engineering, pp. 95-101, 2022. (7 стор.)

13. Gonchar M., "IoT and Mobile-Centric Data Processing in Modern Smart Homes," Intelligent Systems and Applications, vol. 3, no. 1, pp. 43-54, 2023. [Online]. Available: https://doi.org/10.1007/s00542-023-06756-w. (12 стор.)

14. Петренко О. та Зайцев В. , "Мобільно-орієнтовані системи обробки даних у контексті автоматизації розумного будинку," Комп'ютерні системи та мережі, т. 8, с. 58-64, 2022. (7 стор.)

15. Brown A., "Data Processing Models for IoT-Enabled Smart Homes: A Mobile Approach," IEEE Internet of Things Journal, vol. 8, no. 3, pp. 550-560, 2021. [Online]. Available: https://doi.org/10.1109/JIOT.2020.3022334. (12 стор.)

16. Voronov Y. , "Mobile-First Design for Smart Home Data Centralization," Proceedings of the 2021 IEEE 5th International Conference on Energy and Smart Cities, pp. 144-151, 2021. (8 стор.)

17. Мельник О., "Централізовані системи обробки даних у розумних будинках: перспективи використання мобільних додатків," Інформаційні технології та системи управління, вип. 1, с. 33-39, 2023. (7 стор.)

18. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

19. Кавецький В. В**.** Економічне обґрунтування інноваційних рішень: практикум / В. В. Кавецький, В. О. Козловський, І. В. Причепа – Вінниця : ВНТУ, 2016. – 113 с.

Додаток А
(обов'язковий)

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: « Мобільно-орієнтована система централізованої обробкиданих розумного будинку »

Тип роботи: магістерська кваліфікаційна робота

(кваліфікаційна робота, курсовий проект (робота), реферат, аналітичний огляд, інше (зазначити))

Підрозділ кафедра комп'ютерних систем управління

(кафедра, факультет (інститут), навчальна група)

Керівник Ковтун В. В. професор, зав. кафедри КСУ

(прізвище, ініціали, посада)

Показники звіту подібності

Turnitin	
Оригінальність	96%
Загальна схожість	4%

Аналіз звіту подібності (відмітити потрібне)

√ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлений (-на) з повним звітом подібності, який був згенерований Системою щодо роботи (додається)

Автор _____ Богдан МЕЛЬНИК
(підпис) (прізвище, ініціали)

Опис прийнятого рішення

_____

_____

Особа, відповідальна за перевірку _____ Володимир ДУБОВОЙ
(підпис) (прізвище, ініціали)

Експерт _____ _____.
(за потреби) (підпис) (прізвище, ініціали, посада)

Додаток Б
(обов'язковий)

ВНТУ

ЗАТВЕРДЖЕНО
Зав. кафедри КСУ ВНТУ,
д.т.н., проф.
_____ В'ячеслав КОВТУН
«14» 10 2024 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи
«МОБІЛЬНО-ОРІЄНТОВАНА СИСТЕМА ЦЕНТРАЛІЗОВАНОЇ ОБРОБКИ ДАНИХ РОЗУМНОГО БУДИНКУ»
08-01.МКР.002.00.000 ТЗ

Студент групи 2АКІТР-23М
_____ Богдан МЕЛЬНИК
Підпис *Ім'я ПРІЗВИЩЕ*

Керівник Зав. кафедри КСУ ВНТУ,
д.т.н., проф.
_____ В'ячеслав КОВТУН
Підпис *Ім'я ПРІЗВИЩЕ*

Вінниця 2024

1 Назва та галузь застосування

1.1 Назва – Мобільно-орієнтована система централізованої обробки даних розумного будинку.

1.2 Галузь застосування – сфера розумних будинків.

2 Підстава для проведення розробки.

Тема магістерської кваліфікаційної роботи затверджена наказом по ВНТУ № 310 від «17» вересня 2024р.

3 Мета та призначення розробки.

Метою магістерської кваліфікаційної роботи є розробка мобільно-орієнтованої системи централізованної обробки даних для розумного будинку.

4 Джерела розробки.

Магістерська кваліфікаційна робота виконується вперше. В ході проведення розробки повинні використовуватись такі документи:

1. O. Popov, A. Yatsyshyn, and I. Ivanov, "Designing a Smart Home System Based on IoT Technologies," Electronics and Control Systems, vol. 2, no. 60, pp. 80-88, 2021. [Online]. Available: https://doi.org/10.18372/1990-5548.60.15633. (9 стор.)

2. M. Smith and K. Johnson, "Mobile-Oriented Cloud-Based Smart Home Data Processing," IEEE Access, vol. 8, pp. 24012-24022, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2973345. (11 стор.)

3. P. Borysov, "Internet of Things and Mobile Application Design for Smart Home Data Management," Scientific Bulletin of the National Technical University of Ukraine, no. 5, pp. 45-52, 2021. (8 стор.)

5 Вимоги до розробки.

5.1 Перелік головних функцій:

1. Централізоване управління
2. Мобільний доступ
3. Збір та обробка даних

5.2 Основні технічні вимоги до розробки.

5.2.1 Вимоги до програмної платформи:

– WINDOWS 10;
– Visual Studio 2019.
– Android 13

5.2.2 Умови експлуатації системи:

– робота на мобільних платформах;
– можливість цілодобового функціонування системи;

– дані оновлюються і є актуальними.

6 Стадії та етапи розробки.

6.1 Пояснювальна записка:

1 Аналіз методів, принципів, підходів і засобів реалізації задачі автоматизації процесами в об’єкті управління відповідно до теми кваліфікаційної роботи. Постановка задач дослідження «1» жовтня 2024р.

2 Удосконалення технології прийняття рішень при автоматизації об’єкту управління «2» жовтня 2024р.

3 Визначення технічних характеристик системи «5» жовтня 2024р.

4 Розробка програмного забезпечення системи «6» жовтня 2024р.

6.2 Графічні матеріали:

1 Розробка UML-діаграм системи «11» листопада 2024р.

2 Розробка моделі бази даних системи «14» листопада 2024р.

3 Тестування програмного забезпечення «18» листопада 2024р.

7 Порядок контролю і приймання.

7.1 Хід виконання роботи контролюється керівником роботи. Рубіжний контроль провести до «29» листопада 2024р.

7.2 Атестація МКР здійснюється на попередньому захисті. Попередній захист магістерської кваліфікаційної роботи провести до «1» грудня 2024р.

7.3 Підсумкове рішення щодо оцінки якості виконання роботи приймається на засіданні ЕК. Захист магістерської кваліфікаційної роботи провести до «10» грудня 2024

Додаток В
(Довідниковий)
Лістинг програмного коду

```
class MainActivity : Activity() {

val RC_ALL_PERMISIONS = 100

override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState) setContentView(R.layout.activity_main)

btnThreads.setOnClickListener { if (checkPermission()) {
navigateTo(ThreadsActivity::class.java)
}
}
}

private fun navigateTo(activity: Class<ThreadsActivity>) = startActivity(Intent(this, activity))

private fun checkPermission(): Boolean {
var result = false

val sdCardWrite = Manifest.permission.WRITE_EXTERNAL_STORAGE val sdCardRead = Manifest.permission.READ_EXTERNAL_STORAGE val permissionSdCardWriteCheck =
ContextCompat.checkSelfPermission(this, sdCardWrite)
val permissionSdCardReadCheck = ContextCompat.checkSelfPermission(this, sdCardRead)
if (permissionSdCardWriteCheck == PackageManager.PERMISSION_GRANTED
&& permissionSdCardReadCheck == PackageManager.PERMISSION_GRANTED) { result = true
} else {
requestPermissions(arrayOf(sdCardWrite, sdCardRead),
RC_ALL_PERMISIONS)
result = false
}

return result
}

override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>?, grantResults: IntArray?) {
super.onRequestPermissionsResult(requestCode, permissions, grantResults)
if (requestCode == RC_ALL_PERMISIONS) {
```

```
        if (grantResults!![1] == PackageManager.PERMISSION_GRANTED) {
navigateTo(ThreadsActivity::class.java)
        }
        }
        }
        }


        class ThreadsActivity : Activity() {

        override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState)
setContentView(R.layout.activity_threads) btnRun1.setOnClickListener {

        EventBus.getDefault().post(StartMessage()) Handler().postDelayed({
        threadsSequentialWordCount()
        }, 500)
        }
        btnRun2.setOnClickListener { EventBus.getDefault().post(StartMessage()) Handler().postDelayed({
        threadsTwoThreadsWordCount()
        }, 500)
        }
        btnRun3.setOnClickListener { EventBus.getDefault().post(StartMessage()) rxJavaSequentialWordCount()
        }
        btnRun4.setOnClickListener { EventBus.getDefault().post(StartMessage()) rxJavaTwoThreadsWordCount()
        }
        btnRun5.setOnClickListener { EventBus.getDefault().post(StartMessage()) coroutinesSequentialWordCount()
        }
        btnRun6.setOnClickListener { EventBus.getDefault().post(StartMessage())
coroutinesTwoThreadsWordCount()
        }
        btnRun7.setOnClickListener { EventBus.getDefault().post(StartMessage()) coroutinesBetterWordCount()
        }

        btnRun8.setOnClickListener { EventBus.getDefault().post(StartMessage()) rxJavaParallelStreamWordCount()
        }
        }

        override fun onStart() {
        super.onStart() EventBus.getDefault().register(this)
        }

        override fun onStop() {
```

```
super.onStop() EventBus.getDefault().unregister(this)
}

@Subscribe(sticky = false, threadMode = ThreadMode.MAIN)
fun onEvent(message: StartMessage) { LoadingViewMaterial.show(this)
}

@Subscribe(sticky = false, threadMode = ThreadMode.MAIN)
fun onEvent(message: FinishMessage) { LoadingViewMaterial.hide()
}

@Subscribe(sticky = false, threadMode = ThreadMode.MAIN)
fun onEvent(message: StaticticMessage) {
Toast.makeText(this, "Words count: ${message.count} \nTime:
${message.time} millis", Toast.LENGTH_LONG).show()
}

//Threads and Locks Examples
private fun threadsSequentialWordCount() { ThreadSequentialWordCount().run()
}

private fun threadsTwoThreadsWordCount() { ThreadTwoThreadsWordCount().run()
}


//RxJava 2 Examples
private fun rxJavaSequentialWordCount() { RxSequentialWordCount().run()
}

private fun rxJavaTwoThreadsWordCount() { RxTwoThreadsWordCount().run()
}

private fun rxJavaParallelStreamWordCount() { RxParallelStreamWordCount().run()
}


//Kotlin Coroutines Examples
private fun coroutinesSequentialWordCount() { CoroutinesSequentialWordCount().run()
}

private fun coroutinesTwoThreadsWordCount() { CoroutinesTwoThreadsWordCount().run()
}
```

```
private fun coroutinesBetterWordCount() { CoroutinesBetterWordCount(Source()).run()
}
}


class ThreadSequentialWordCount {
private val LOG_TAG = ThreadSequentialWordCount::class.java.canonicalName

private val counts: HashMap<String, Int?> = HashMap()

fun run() {

val thread = Thread {
val pagesOne = Pages(0, 700, Source().wikiPagesBatchOne()) pagesOne.forEach { page ->
Words(page.text).forEach {
countWord(it) } }

val pagesTwo = Pages(0, 700, Source().wikiPagesBatchTwo()) pagesTwo.forEach { page ->
Words(page.text).forEach {
countWord(it) } }
}

val time = measureTimeMillis { thread.start() thread.join()
}

Log.d(LOG_TAG, "Number of elements: ${counts.size}") Log.d(LOG_TAG, "Execution Time: $time ms")

EventBus.getDefault().post(FinishMessage()) EventBus.getDefault().post(StaticticMessage(time, counts.size))
}

private fun countWord(word: String) {
when (counts.containsKey(word)) {
true -> counts[word] = counts[word]?.plus(1) false -> counts[word] = 1
}
}
}


class ThreadTwoThreadsWordCount {
private val LOG_TAG = ThreadTwoThreadsWordCount::class.java.canonicalName
```

```
private val counts: ConcurrentHashMap<String, Int?> = ConcurrentHashMap()

fun run() {
val threadOne = Thread {
val pagesOne = Pages(0, 700, Source().wikiPagesBatchOne()) pagesOne.forEach { page -> Words(page.text).forEach {
countWord(it) } }
}

val threadTwo = Thread {
val pagesTwo = Pages(0, 700, Source().wikiPagesBatchTwo()) pagesTwo.forEach { page -> Words(page.text).forEach {
countWord(it) } }
}

val time = measureTimeMillis { threadOne.start() threadTwo.start() threadOne.join() threadTwo.join()
}

Log.d(LOG_TAG, "Number of elements: ${counts.size}") Log.d(LOG_TAG, "Execution Time: $time ms") EventBus.getDefault().post(FinishMessage()) EventBus.getDefault().post(StaticticMessage(time, counts.size))
}

private fun countWord(word: String) =
counts.merge(word, 1) { oldValue, _ -> oldValue.plus(1) }
}



class RxSequentialWordCount {
private val LOG_TAG = RxSequentialWordCount::class.java.canonicalName

private val counts: HashMap<String, Int?> = HashMap()

fun run() {
val startTime = System.currentTimeMillis()

val observable = Observable.fromCallable { var pagesOne = Pages.empty()
val pagesOneCreationTime = measureTimeMillis {
pagesOne = Pages(0, 700, Source().wikiPagesBatchOne())
}
pagesOne.forEach { page -> Words(page.text).forEach {

countWord(it) } }
```

```
var pagesTwo = Pages.empty()
val pagesTwoCreationTime = measureTimeMillis {
pagesTwo = Pages(0, 700, Source().wikiPagesBatchTwo())
}
pagesTwo.forEach { page -> Words(page.text).forEach {
countWord(it) } }

logPagesCreationTime(pagesOneCreationTime, pagesTwoCreationTime)
}

observable
.doOnComplete { logExecutionData(System.currentTimeMillis() -

startTime) }



}



.subscribeOn(Schedulers.single())
.subscribe()



private fun countWord(word: String) {
when (counts.containsKey(word)) {
true -> counts[word] = counts[word]?.plus(1) false -> counts[word] = 1
}
}

private fun logPagesCreationTime(timePagesOne: Long, timePagesTwo: Long)
{
Log.d(LOG_TAG, "PageOne creation time: $timePagesOne ms") Log.d(LOG_TAG, "PageOne creation time:
$timePagesTwo ms") Log.d(LOG_TAG, "Total Execution Pages Creation:
${timePagesOne.plus(timePagesTwo)} ms")
}

private fun logExecutionData(time: Long) { Log.d(LOG_TAG, "Number of elements: ${counts.size}")
Log.d(LOG_TAG, "Execution Time: $time ms") EventBus.getDefault().post(FinishMessage())
EventBus.getDefault().post(StaticticMessage(time, counts.size))
}
}
```

```
class RxTwoThreadsWordCount {
private val LOG_TAG = RxTwoThreadsWordCount::class.java.canonicalName

private val counts: ConcurrentHashMap<String, Int?> = ConcurrentHashMap()

fun run() {
val startTime = System.currentTimeMillis()

val observablePagesOne = Observable.fromCallable {
val pagesOne = Pages(0, 700, Source().wikiPagesBatchOne()) pagesOne.forEach { page ->
Words(page.text).forEach {
countWord(it) } }
}.subscribeOn(Schedulers.newThread())

val observablePagesTwo = Observable.fromCallable {
val pagesTwo = Pages(0, 700, Source().wikiPagesBatchTwo()) pagesTwo.forEach { page ->
Words(page.text).forEach {
countWord(it) } }
}.subscribeOn(Schedulers.newThread())

observablePagesOne
.mergeWith(observablePagesTwo)
.doOnComplete { logData(System.currentTimeMillis() -

startTime) }

}


.subscribe()


private fun countWord(word: String) =
counts.merge(word, 1) { oldValue, _ -> oldValue.plus(1) }

private fun logData(time: Long) {
Log.d(LOG_TAG, "Number of elements: ${counts.size}") Log.d(LOG_TAG, "Execution Time: $time ms")
EventBus.getDefault().post(FinishMessage()) EventBus.getDefault().post(StaticticMessage(time, counts.size))
}
}
```

```
class RxParallelStreamWordCount {
private val LOG_TAG = RxParallelStreamWordCount::class.java.canonicalName

fun run() {
val startTime = System.currentTimeMillis()

val singleLettersCount = Single.fromCallable {
Pages(0, 700, Source().wikiPagesBatchOne())
.plus(Pages(0, 700, Source().wikiPagesBatchTwo()))
.parallelStream()
.flatMap {
StreamSupport.stream(Words(it.text).spliterator(), false) }
.collect(Collectors.groupingBy({ it.toString() }, Collectors.counting()))
}

singleLettersCount
.subscribeOn(Schedulers.io())
.subscribe { data ->
logData(System.currentTimeMillis() - startTime,
data.size)
}
}

private fun logData(time: Long, count: Int) { Log.d(LOG_TAG, "Number of elements: $count")
Log.d(LOG_TAG, "Execution Time: $time ms") EventBus.getDefault().post(FinishMessage())
EventBus.getDefault().post(StaticticMessage(time, count))
}
}


class CoroutinesSequentialWordCount {
private val counts: HashMap<String, Int?> = HashMap()

private val LOG_TAG =
CoroutinesSequentialWordCount::class.java.canonicalName

fun run() {
launch(newSingleThreadContext("myThread")) { val startTime = System.currentTimeMillis() counter()
logData(System.currentTimeMillis() - startTime)
```

```
}
}

private suspend fun counter() {
val pagesOne = Pages(0, 700, Source().wikiPagesBatchOne()) pagesOne.forEach { page -> Words(page.text).forEach { countWord(it) }
}

val pagesTwo = Pages(0, 700, Source().wikiPagesBatchTwo()) pagesTwo.forEach { page -> Words(page.text).forEach { countWord(it) }
}
}

private fun countWord(word: String) {
when (counts.containsKey(word)) {
true -> counts[word] = counts[word]?.plus(1) false -> counts[word] = 1
}
}

private suspend fun logData(time: Long) {
Log.d(LOG_TAG, "Number of elements: ${counts.size}") Log.d(LOG_TAG, "Execution Time: $time ms") EventBus.getDefault().post(FinishMessage()) EventBus.getDefault().post(StaticticMessage(time, counts.size))
}
}



class CoroutinesTwoThreadsWordCount {
private val counts: ConcurrentHashMap<String, Int?> = ConcurrentHashMap()

private val LOG_TAG =
CoroutinesTwoThreadsWordCount::class.java.canonicalName

fun run() {
val poolContext = newFixedThreadPoolContext(2, "ThreadPool")
launch(poolContext) {
val time = measureTimeMillis {
val one = async(poolContext) { counterPages1() } val two = async(poolContext) { counterPages2() } one.await()
two.await()
}
logData(time)
}
}
```

```
private fun counterPages1() {
val pagesOne = Pages(0, 700, Source().wikiPagesBatchOne()) pagesOne.forEach { page -> Words(page.text).forEach { countWord(it) }
}
}

private fun counterPages2() {
val pagesTwo = Pages(0, 700, Source().wikiPagesBatchTwo()) pagesTwo.forEach { page -> Words(page.text).forEach { countWord(it) }
}
}

private fun countWord(word: String) =
counts.merge(word, 1) { oldValue, _ -> oldValue.plus(1) }

private fun logData(time: Long) {
Log.d(LOG_TAG, "Number of elements: ${counts.size}") Log.d(LOG_TAG, "Execution Time: $time ms") EventBus.getDefault().post(FinishMessage()) EventBus.getDefault().post(StaticticMessage(time, counts.size))
}
}



class CoroutinesBetterWordCount(source: Source) {
private val LOG_TAG = CoroutinesBetterWordCount::class.java.canonicalName

private val filePagesOne = source.wikiPagesBatchOne()
private val filePagesTwo = source.wikiPagesBatchTwo()

fun run() {
launch(CommonPool) {
val time = measureTimeMillis {
val one = async(CommonPool) { counter(0.rangeTo(2000),

filePagesOne) } filePagesOne) } filePagesTwo) } filePagesTwo) }
four.await())
}



val two = async(CommonPool) { counter(2001.rangeTo(3000), val three = async(CommonPool) { counter(0.rangeTo(2000), val four = async(CommonPool) { counter(2001.rangeTo(3000), mergeResults(one.await(), two.await(), three.await(),
```

```
logData(time)
}
}

private fun counter(range: IntRange, file: File): HashMap<String, Int?> {
val counts: HashMap<String, Int?> = HashMap()
val pages = Pages(range.start, range.endInclusive, file) pages.forEach { page -> Words(page.text).forEach { countWord(counts,

it) } }

}


return counts


private fun countWord(counts: HashMap<String, Int?>, word: String) = counts.merge(word, 1) { oldValue, _ -> oldValue.plus(1) }

private fun logData(time: Long) { Log.d(LOG_TAG, "Execution Time: $time ms") EventBus.getDefault().post(FinishMessage())
EventBus.getDefault().post(StaticticMessage(time, 196681))
}

@TargetApi(VERSION_CODES.N)
fun <K, V> Map<K, V>.mergeReduce(other: Map<K, V>, reduce: (V, V) -> V =
{ _, b -> b }): Map<K, V> =
this.toMutableMap().apply { other.forEach { merge(it.key,
it.value, reduce) } }
}
```

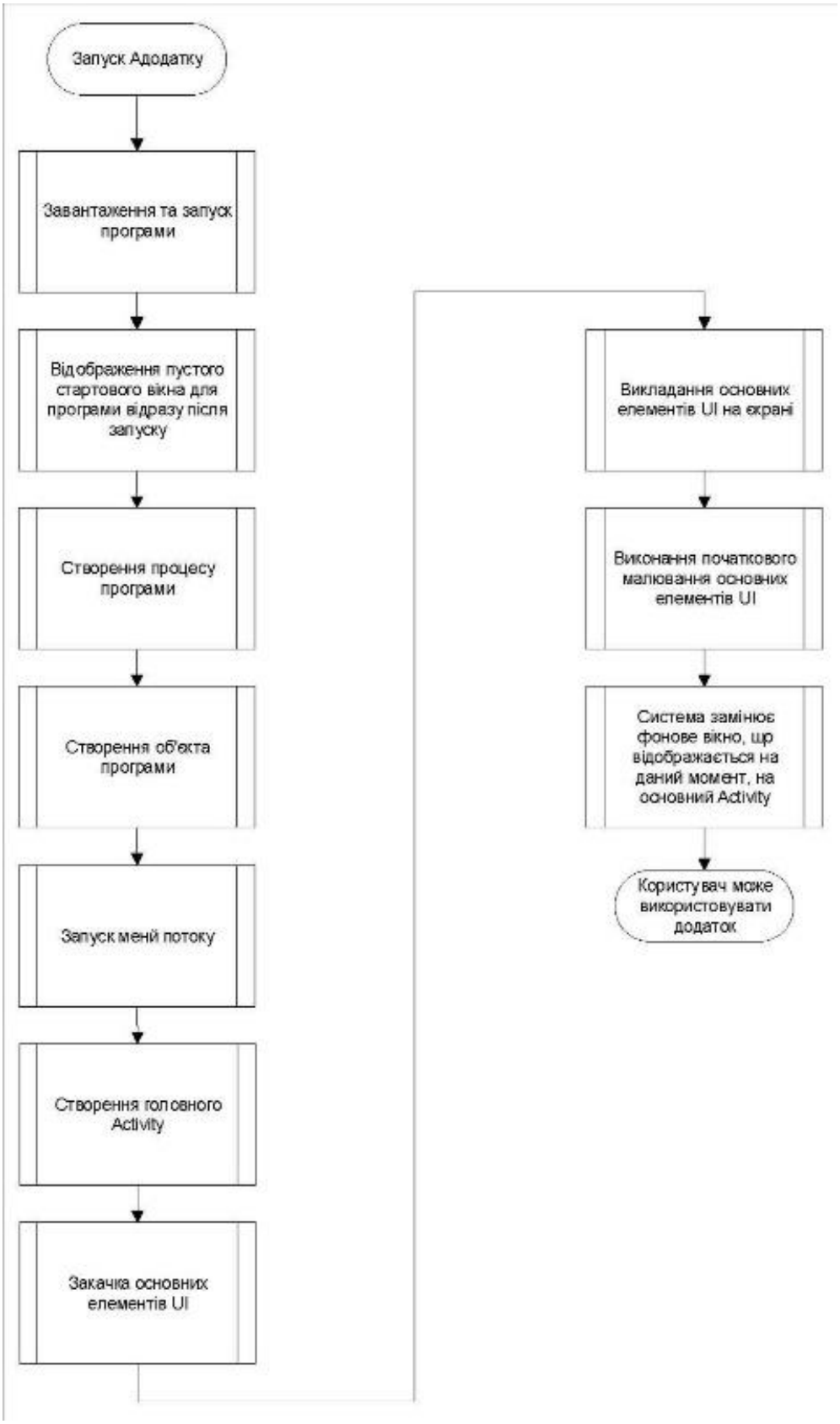
Додаток Д
(обов'язковий)

ІЛЮСТРАТИВНА ЧАСТИНА
«Мобільно-орієнтована система централізованої обробки даних розумного будинку.»

Студент групи 2АКІТР-23М
Підпис Богдан МЕЛЬНИК
Ім'я ПРІЗВИЩЕ

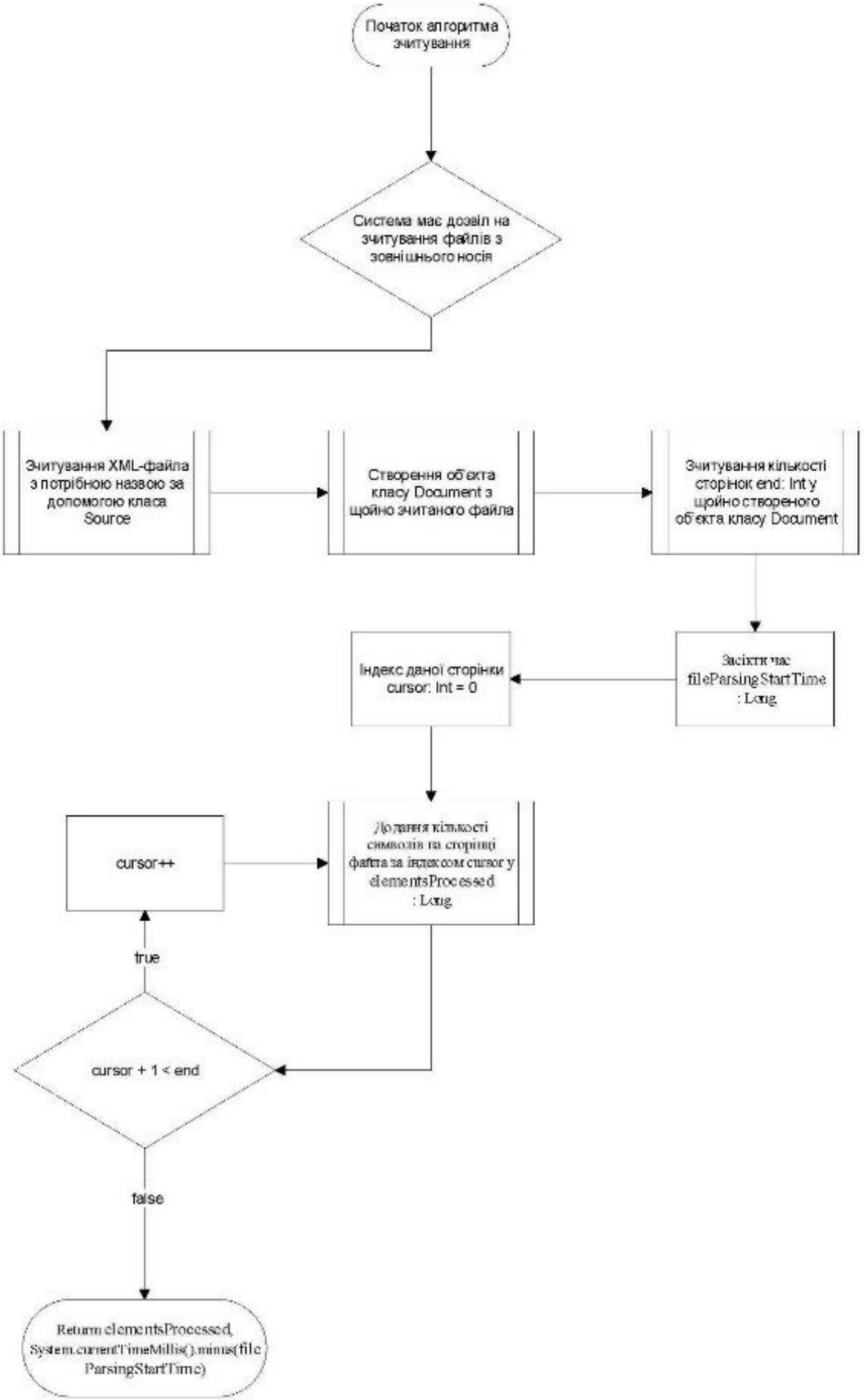
Керівник Зав. кафедри КСУ ВНТУ, д.т.н., проф.
Підпис В'ячеслав КОВТУН
Ім'я ПРІЗВИЩЕ

Вінниця 2024

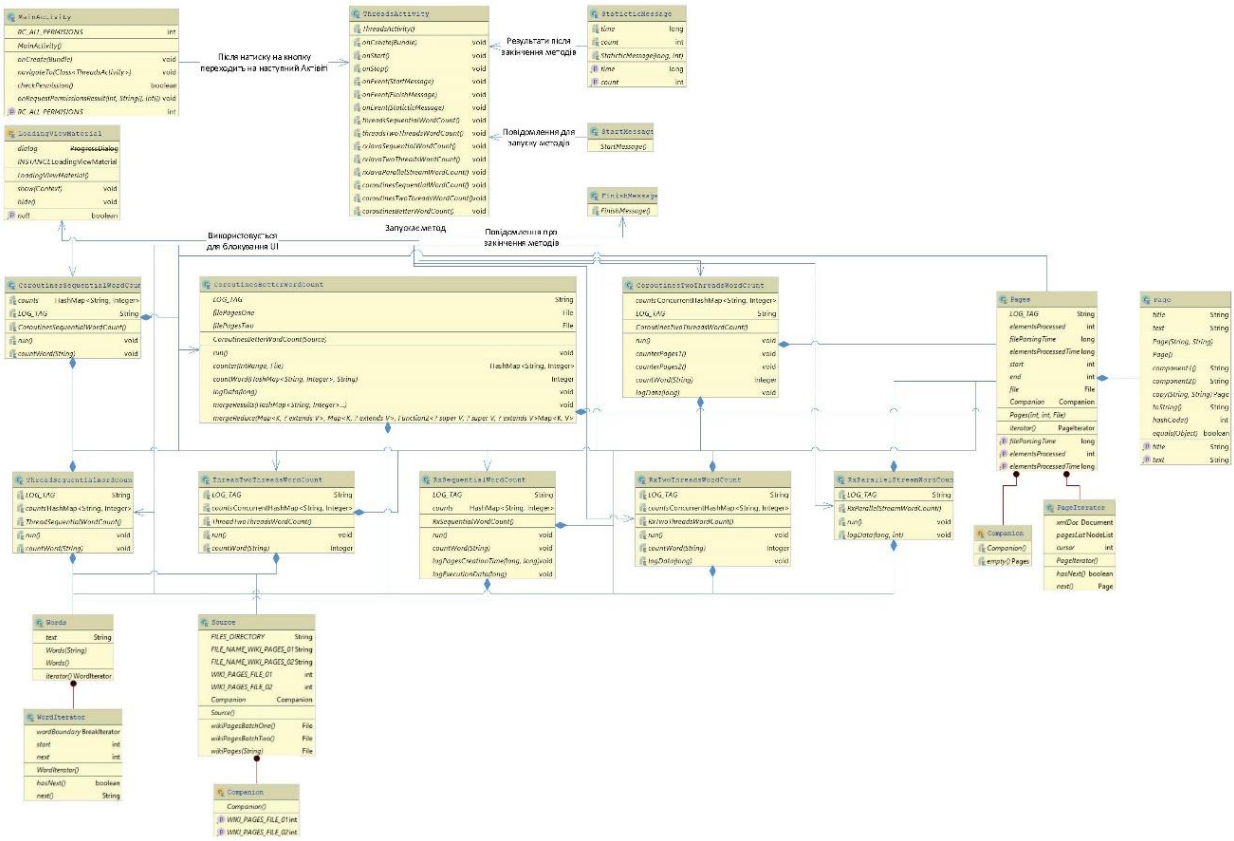


Процес холодного Android додатку

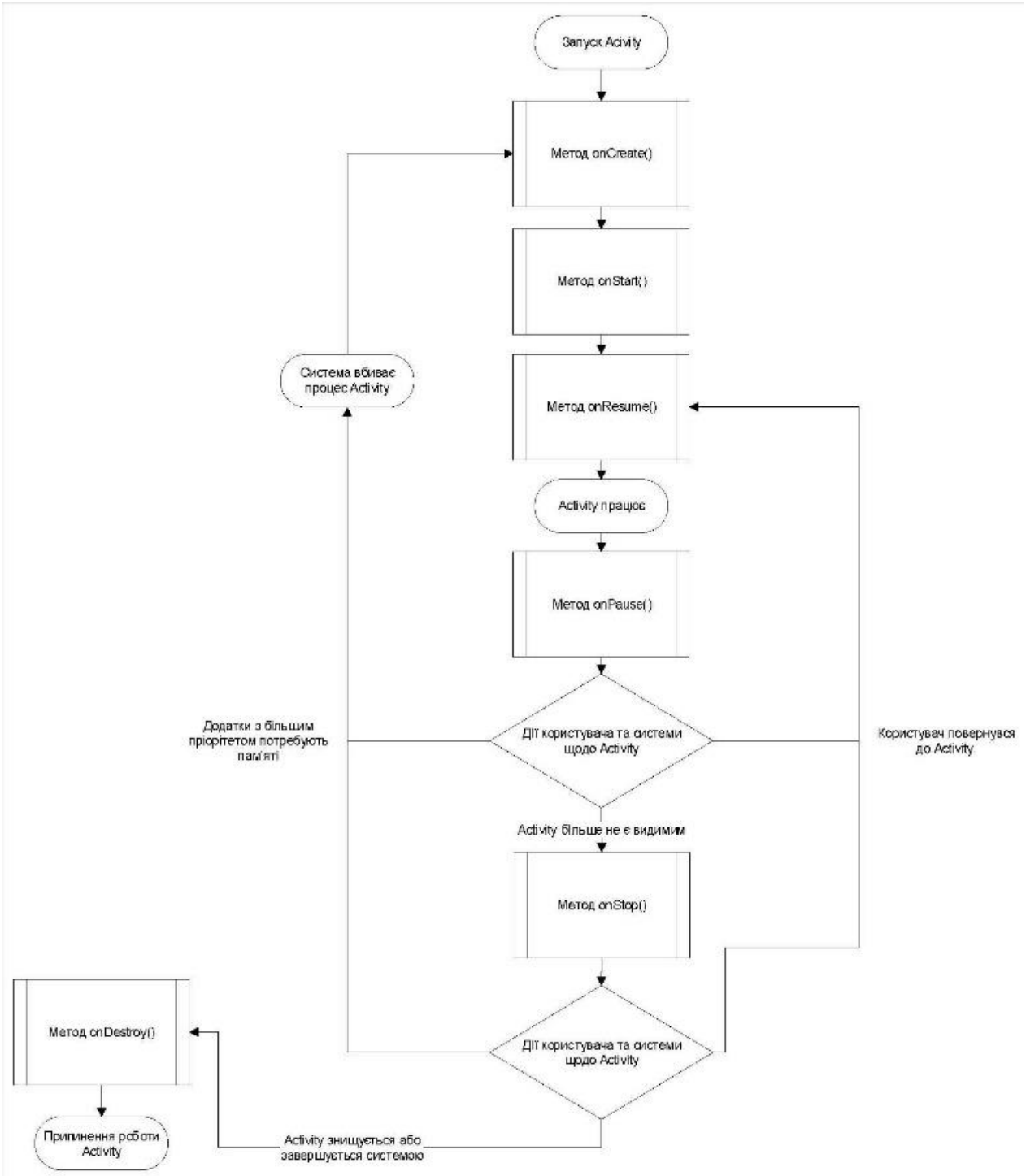
Алгоритм функціонування додатку



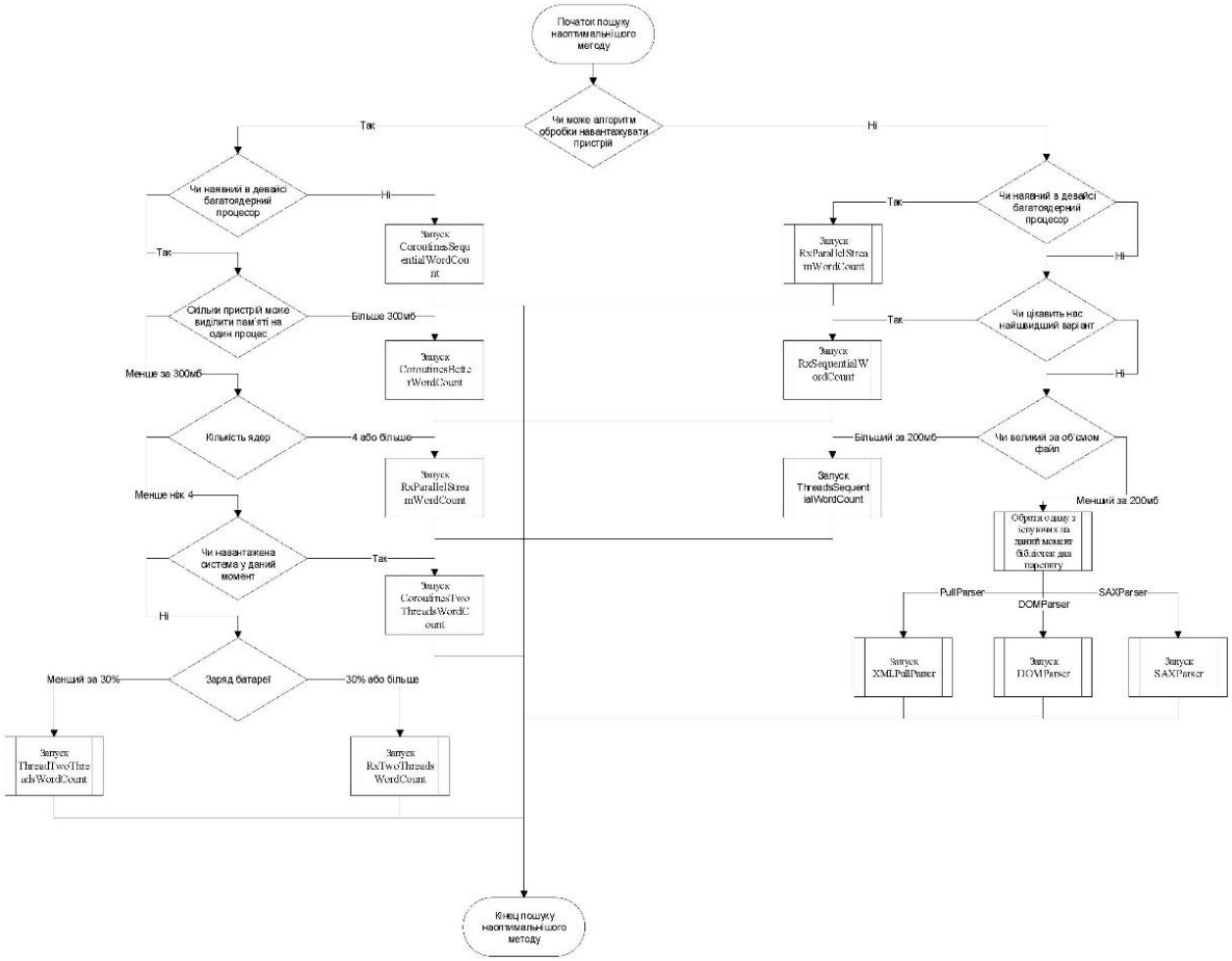
Алгоритм зчитування XML-фай ла додатком



Діаграма класів додатку

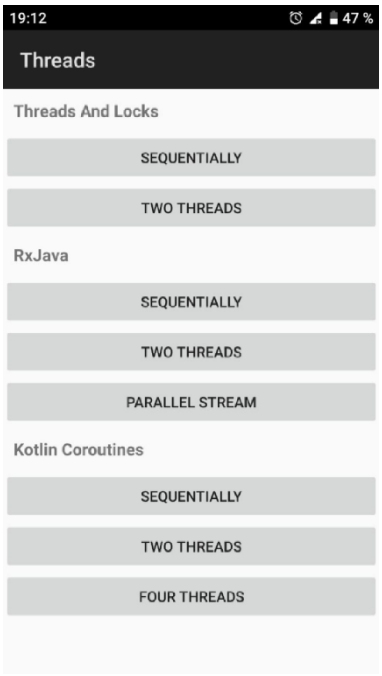


Життєвий цикл додатку

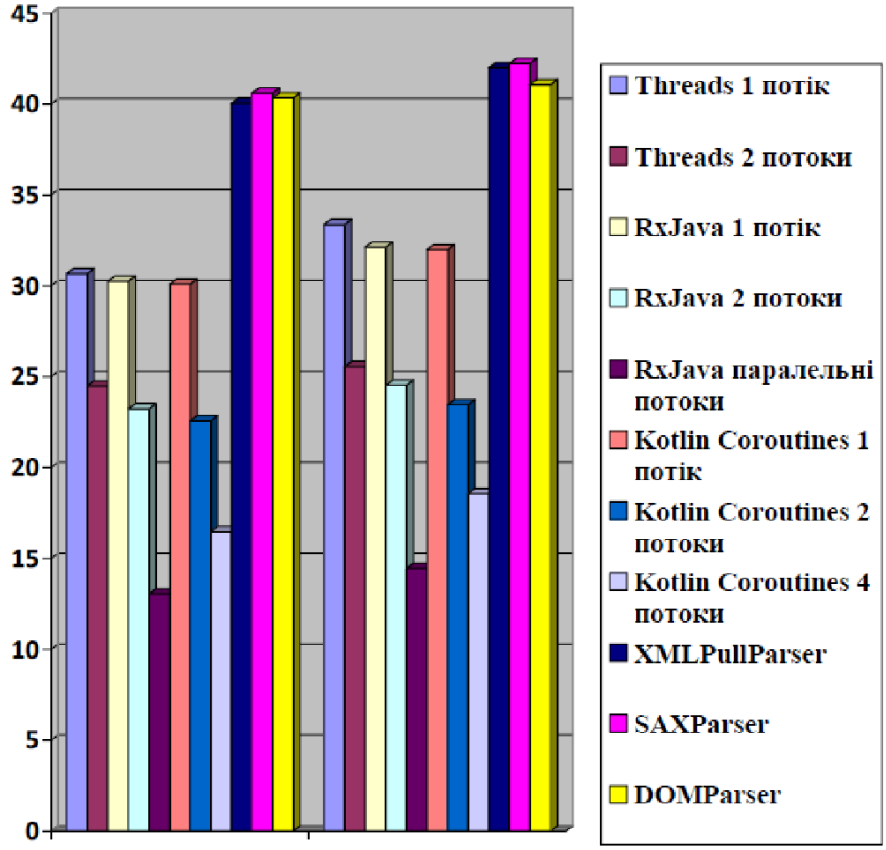


Алгоритм пошуку найоптимальнішого методу парсингу

Екран привітання



Екран методів обробки



Діаграма порівняння всіх реалізованих.