Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Підвищення продуктивності обробки даних за допомогою периферійних обчислень на пристроях iOS

Текстова частина до кваліфікаційної роботи
за спеціальністю „Комп'ютерні науки ” 122

Керівник кваліфікаційної роботи
с.в. Франків О.О.
(прізвище та ініціали)

_____
(підпис)
“____” _____ 2025 р.

Виконала студентка БП КН-4
Кучина Є.В.
(прізвище та ініціали)
“____” _____ 2025 р.

Київ 2025

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 122 «Комп'ютерні Науки»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

"__" _____ 2024 року

ЗАВДАННЯ
ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТЦІ

Кучиній Єлізавєті

1. Тема роботи **«Підвищення продуктивності обробки даних за допомогою периферійних обчислень на пристроях iOS»**, керівник роботи Франків Олександр Олександрович, магістр комп'ютерних наук, старший викладач

2. Строк подання студентом роботи 1 червня 2025

3. План роботи

    Анотація

    Вступ

    Розділ 1. Периферійні обчислення

        2.1. Проблематика розподілу навантаження між пристроями
        2.2. Аналіз застосування периферійних обчислень в існуючих рішеннях

Розділ 2. Застосування периферійних обчислень для підвищення продуктивності обчислень на мобільних пристроях

- 2.1. Обмежені можливості локальних та хмарних обчислень для рішення певних видів задач
- 2.2. Архітектура периферійних обчислень для мобільних застосунків
- 2.3. Підходи до розподілу обчислень між пристроями

Розділ 3. Реалізація розподіленої системи

- 3.1. Обмежені можливості локальних та хмарних обчислень для рішення певних видів задач
- 3.2. Реалізація задачі сканування на пристроях iOS
- 3.3. Реалізація реактивного механізму прийняття рішення
- 3.4. Реалізація передачі даних від пристрою до периферії
- 3.5. Опис принципів роботи результуючої системи

Висновки

Список використаних джерел

Додатки

Дата видачі „ __” _____ 2024 р.

Керівник _____

Завдання отримано _____

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	7 жовтня 2024			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	10 жовтня 2024 – 2 листопада 2024			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	2 листопада 2024			
4.	Написання розділів роботи	2 листопада 2024 – 10 квітня 2025			
5.	Проміжний контроль виконання роботи	20 лютого 2025			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	11 січня 2025 – квітня 2025			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	25 січня 2025			
	Розділ 2 (аналітично-дослідницька частина)	01 березня 2025			
	Розділ 3 (проектно-рекомендаційна частина)	10 квітня 2025			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	25 квітня 2025 – 27 травня 2025			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	1 червня 2025			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2024 р.

Науковий керівник Франків Олександр Олександрович

Виконавець кваліфікаційної роботи Кучина Єлізавєта

ЗМІСТ

АНОТАЦІЯ

В даній роботі досліджено ефективність використання периферійних обчислень для задачі обробки даних й запропоновано механізм прийняття рішення про перенесення обчислень з локального мобільного пристрою на периферію. Для демонстрації успішної роботи стратегії обрано задачу сканування об’єктів, для якої визначено два типи обчислень для пересилання, а також емпірично досліджено пороги для стратегії, які дають найбільший виграш в контексті швидкодії. У результаті тестувань визначено, що вдалося зменшити час сканування на 78.4% й тим самим доведено, що периферійні обчислення підвищують продуктивність обробки даних для мобільних пристроїв.

ВСТУП

Важливим спостереженням при аналізі технологічного стану сучасного світу є факт того, що все будується на даних. Безумовно, будь-який користувач цифрових технологій лишає за собою інформаційний слід. Відповідно, через зростання населення – зростає й кількість даних, які люди продукують. За своєю суттю дані є структурованою або неструктурованою інформацією, з якою можна взаємодіяти: отримувати, обробляти та оновлювати.

Через причини зростання кількості даних й необхідності їх аналізувати, постало питання про варіанти їх зберігання та продуктивної обробки. Першим кроком були так звані хмарні обчислення. Зберігати дані на фізичних серверах (on-premise модель) ставало неефективно й дорого, до того ж з появою соціальних мереж та Великих даних – сервери просто не витримували навантаження. Ці причини стали передумовами виникнення хмарних обчислень, які й стали вирішенням проблем. Можливість масштабування, глобальна доступність через Інтернет та автоматизація певний час задовольняла потреби користувачів. Проте технології продовжували стрімко розвиватися й з появою IoT пристроїв, розумних девайсів (smart devices), відеокамер тощо виникла вже нова низка проблем, які хмарні обчислення не могли вирішити. Кількість пристроїв, підключених до IoT зростала як і продукування ними даних. Це призвело до високих затримок у реальному часі й зниженню швидкості. До того ж проблемою стали залежність від інтернету, високе споживання енергії та збільшення рівня загрози приватності та безпеці даних.[1] У відповідь на ці виклики виникають периферійні обчислення, суть яких полягає у вирішенні цих проблем шляхом переносу обробки даних ближче до користувача.

Об’єктом дослідження даної роботи є процес збільшення ефективності обробки даних на мобільних пристроях за допомогою периферійних обчислень.

Предметом дослідження є способи передачі даних від пристрою до периферії, сканування об’єктів у реальному світі, та інструменти реалізації стратегії для системи з метою ефективної взаємодії між основним пристроєм та периферією.

Метою роботи є дослідити ефективність використання периферійних обчислень для задачі обробки даних та запропонувати стратегію, згідно якої потрібно пересилати обчислення на периферію.

Для реалізації мети кваліфікаційної роботи необхідно виконати наступні кроки:

- зробити огляд існуючих рішень застосування периферійних обчислень, які підвищують швидкодію та обробку інформації
- розглянути й визначити ефективну стратегію розподілення обчислень, відповідно до цього побудувати механізм прийняття рішень
- обрати задачу, на прикладі якої буде показано роботу стратегії
- дослідити способи передачі даних між пристроєм та периферією
- провести тестування створеного демонстраційного застосунку й емпірично дослідити параметри, при яких стратегія вивантаження обчислень працюватиме найкраще

Результатом дослідження стала стратегія, яка спостерігатиме за ресурсами локального пристрою й на основі цього здійснюватиметься рішення про пересилання обчислень. Успішне спрацювання даної стратегії продемонстровано на прикладі створеного застосунку.

РОЗДІЛ 1. Периферійні обчислення

1.1 Проблематика розподілу навантаження між пристроями

Мобільні пристрої мають обмежені обчислювальні ресурси, це може ускладнювати їх можливість ефективно виконувати обчислення та аналізувати дані. Це особливо актуально для задач, виконання яких вимагає низької затримки у реальному часі та інтенсивне використання RAM. Їх обчислення можуть негативно впливати на паралельне виконання інших задач на пристрої й мати проблеми збереження великої кількості даних в оперативній пам’яті. Саме на прикладі такої задачі доцільно продемонструвати ефективність периферійних обчислень саме в контексті підтримки обчислювальних процесів на мобільному пристрої. Проте не завжди використання периферії може бути виправданим й надавати перевагу в контексті покращення швидкодії. Тож при побудові додатку потрібно врахувати низку аспектів.

По-перше, будь-які обчислення – це передусім дані. Питання конфіденційності користувача завжди стоїть на першому місці. Периферійні обчислення здійснюють обробку даних локально, й це відповідно дозволяє уникнути ризиків, пов’язаних з передачею сирих даних через мережу – немає загрози їх втрати чи витоку. Також якщо враховувати можливість кібератаки, то загроза буде стосуватися лише локальних даних, а не повного об’єму.[1]

По-друге, як вже було згадано, не всі задачі потребують додаткової периферії для обробки. Навпаки, є випадки, коли це призводить до більших складнощів й затримок. Це може бути обумовлено складним процесом передачі даних – у певних випадках він є тривалим, ресурсоємним або ж не виправданим в контексті складності обчислень – може бути достатньо ресурсів локального пристрою для їх виконання. З огляду на це, потрібно коректно

обрати задачу, для якої використання периферії буде виправданим та взяти спосіб передачі даних такий, який буде найбільш логічний та зручний вже в контексті обраної задачі.

Останнім важливим аспектом є обрання стратегії передачі даних. Оскільки задача може змінювати свою складність — від легкого рівня, де ресурсів мобільного пристрою достатньо, до більш складного, що потребує додаткових обчислювальних потужностей, — необхідно визначати критерії, за якими ухвалюється рішення про перенесення обчислень на периферію. Такими критеріями можуть бути обсяг даних, складність обробки, рівень завантаженості процесора, заряд батареї тощо. Водночас важливо враховувати ризики, пов’язані з передачею даних. Серед них – обмежена пропускна здатність каналу чи нестабільність з’єднання, які можуть знизити ефективність такого підходу. Тому стратегія має адаптуватися під умови виконання, балансуючи між продуктивністю, енерговитратами та надійністю передачі. Тож потрібно визначати коли є доцільним переносити обчислення на периферію.

1.2 Аналіз застосування периферійних обчислень в існуючих рішеннях

У період, коли хмарні обчислення вважалися провідним досягненням у сфері цифрових технологій, виникає концепція Інтернету речей (IoT), що передбачає об’єднання пристроїв у єдину мережу — комп’ютерів, сенсорів, телефонів, телевізорів тощо, які взаємодіють між собою, обмінюються інформацією, й дана система доповнена вбудованою аналітикою для обробки цих даних. На тлі розвитку цієї технології, периферійні обчислення стають популярним напрямом досліджень та експериментів, адже потенційно концепція обіцяє покращити швидкість виконання у реальному часі та

конфіденційність користувачів. Детальний огляд двох існуючих рішень ефективного застосування периферійних обчислень наведено у наступних підпунктах.

1.2.1. EdgeIoT

Першим прикладом рішення, яке ефективно показує переваги периферії й доводить що дійсно можна знизити затримку обробки в реальному часі та забезпечити конфіденційність є EdgeIoT – нова архітектура мобільних периферійних обчислень для IoT девайсів. Автори цього підходу поєднують концепції туманних(fog computing) та периферійних обчислень. Ці поняття часто ототожнюють або ж помилково вважають еквівалентними. Проте у цих підходів є певні відмінності. Туманні обчислення частину даних обробляють на фізичних серверах близько до мережі локального пристрою, але за її межами – це принципово важливо. Сама концепція є ширшою з точки зору архітектури за периферійні обчислення, які є, строго кажучи, її складовою. Туманні обчислення збиратимуть інформацію з багатьох IoT девайсів й оброблятимуть на краю мережі. Отже, у представленій моделі обробка даних, які надходять з IoT пристроїв, відбувається на локальних вузлах (fog nodes), розміщених ближче до джерела даних. Це зменшує навантаження на хмару та мережу, забезпечує меншу затримку й швидшу обробку великих обсягів даних IoT у реальному часі. Тож автори не передають всі дані на обробку одразу на хмару, а роблять проміжний шар, розташований близько до джерела даних.[2] Проте постає питання ефективного розташування fog nodes, адже згідно концепції, вони повинні бути близько до IoT девайсів та не перевантажувати мережу. Аби задовольнити ці вимоги, було вирішено розташовувати їх поруч з базовими станціями(Base Stations) – саме вони є місцями, де концентрується трафік від багатьох пристроїв. Тож розподілені BS мають потенціал

приєднатися до всіх IoT девайсів, не важливо статичні вони чи ні. Це дозволяє перехоплювати й обробляти дані одразу, не відправляючи їх далі у мережу, обробляти на периферії та використати існуючу архітектуру. Переваги цього підходу є наступними – менше навантаження на хмару, менші затримки та економічно вигідно відповідно. Після агрегації даних на BS, вони передаються до fog nodes.

Fog вузол може бути напряму підключеним до BS, або ж периферійні обчислення можуть бути розгорнуті на межі ядра стільникової мережі, що дозволяє кільком базовим станціям спільно використовувати один вузол для обробки своїх локальних потоків даних.[2] Також EdgeIoT використовує певну стратегію передачі даних – fog вузол може обробляти дані локально, або ж якщо не вистачає ресурсів, передавати це на хмару.

Окремо автори даного підходу вирішують питання конфіденційності користувача. Кожен користувач має свою проксі віртуальну машину(VM), яка буде його приватною VM, розташованою в сусідньому туманному вузлі. Дана віртуальна машина буде збирати дані з IoT пристроїв користувача, аналізувати й створювати метадані, які й будуть відправлятися на хмару, таким чином надавати безпеку чутливим даним користувача. Важливим та цікавим моментом є те, що віртуальні машини можуть мігрувати – якщо користувач переміщається з місця на місце – VM переміщається ближче до нових пристроїв. Дана архітектура відображена на рисунку (див. Додаток 1).

Архітектура edgeIoT поєднує переваги edge computing та fog computing, забезпечуючи обробку даних ближче до джерела їхнього створення, з одночасною можливістю масштабованого управління та гнучкого розгортання сервісів.

Отже, edgeIoT є ефективною архітектурою для периферійних обчислень, яка в контексті IoT забезпечує обробляння даних ближче до пристроїв, знижуючи затримки й навантаження на мережу. До того ж, дана архітектура поєднує периферійні та туманні обчислення, що допомагає швидше реагувати та підтримувати різного типу IoT девайси.

1.2.2. GigaSight

Розглянемо додаток, який дозволяє транслювати відео на хмару. Велика швидкість передач відео з багатьох камер створює серйозне навантаження для хмарної інфраструктури й в цьому й полягав виклик цієї задачі. Навіть невелика частина користувачів середнього міста може створити настільки високий сумарний потік даних, що це швидко призведе до перевантаження мережі метрополітенського рівня (MAN).[4] Наприклад, для 12 000 користувачів, що одночасно транслюють відео у форматі 1080p, потрібно з'єднання пропускною здатністю 100 Гбіт/с. А для мільйона користувачів необхідно вже 8,5 Тбіт/с, що значно перевищує можливості сучасних мереж.[4] Тож, головне питання полягає в тому, як передавати, зберігати та обробляти дані відео з мільйонів камер, при цьому не перевантажуючи мережу та хмарні центри, які їх обробляють.

Відповіддю став фреймворк GigaSight – гібридна хмарна архітектура, яка використовує децентралізовані хмарні обчислення, реалізовуючи клаудлети (cloudlets) у формі віртуальних машин.[4] Клаудлет – це елемент архітектури, невеликий локальний дата центр, який розташований близько до джерела даних, себто до користувача. Клаудлет буде забезпечувати обчислювальні ресурси, зберігання даних та мережеві сервіси на “краю”

мережі (на edge), з метою зниження затримки, оптимізації використання пропускної здатності мережі та підвищення швидкодії при обробці даних. По суті, клаудлет буде посередником між мобільним пристроєм та хмарою. Як кажуть самі автори фрейморку, клаудлет може розглядатися як міні дата центр, який приносить хмару ближче до користувача[3].

Тож, в запропонованій архітектурі фреймворку, відео дані передаються до найближчого від користувача клаудлету. На цьому `middle men` дані аналізуються та обробляються й на хмару вже надсилаються результати (теги, розпізнані обличчя і тд з метаданими)[3]. Це дозволяє зменшити обсяг вхідного трафіку в хмару у тисячі або навіть мільйони разів. Крім того, GigaSight демонструє, як використання тегів і метаданих у хмарі може допомогти здійснювати більш глибокий і персоналізований пошук вмісту відео фрагментів протягом обмеженого часу їх зберігання на клаудлеті. До того ж, варто зазначити, що клаудлети, як форма периферійних обчислень, дійсно вирішують проблему конфіденційності користувачів. Як вже було згадано, традиційні IoT відправляють весь пул сирих даних на хмару, що є погано з точки зору приватності. Клаудлети як раз таки дозволять обробити дані й лише потім відправити їх на хмару – автоматична модифікація відео. Конкретніше це може означати видалення кадрів, розмиття зображення об’єктів, специфічних саме для цього користувача. Цікаво, що цей процес має свою назву – денатурація (denaturing).

Отже, GigaSight є прикладом інноваційного рішення, що використовує периферійні обчислення через клаудлети для обробки відеоданих. Завдяки локальній обробці відео на клаудлетах система дозволяє зменшити навантаження на мережу та хмарну інфраструктуру, зберігаючи при цьому конфіденційність даних шляхом попередньої анонімізації. Даний фреймворк є

яскравим прикладом того, наскільки периферійні обчислення можуть грати ключову роль у контексті ефективності обробки даних та забезпечення конфіденційності користувача. Архітектура GigaSight відображена на рисунку (див. Додаток 2).

РОЗДІЛ 2. Застосування периферійних обчислень для підвищення продуктивності обчислень на мобільних пристроях

2.1 Обмежені можливості локальних та хмарних обчислень для рішення певних видів задач

Для того, аби чітко розуміти, яке архітектурне рішення використовувати та які можливості периферійні обчислення пропонують як альтернатива існуючим рішенням, розглянемо детальніше й порівняємо on-premise системи, й інфраструктуру з хмарними та периферійними обчисленнями.

Локальні (on premise) системи є традиційною моделлю розгортання інфраструктури. Суть випливає з назви – компанії зберігають свої дані та сервери локально. Відповідно, вся підтримка архітектури проводиться локально й лежить на відповідальності власників. Головні мінуси даної моделі, які й підштовхнули до розвитку хмарної інфраструктури, є високі фінансові витрати на підтримку фізичної інфраструктури, складність масштабування й модернізації (тобто якщо потреби розширюються, й потрібно більше ресурсів, потрібно докуповувати нове обладнання й при оновленні потрібно зупиняти роботу серверів). Проте варто зазначити, що перевагою on premise систем є як раз таки безпека даних, бо нічого не виходить за межі компанії.[5] З огляду на описане, для наступних задач локальна інфраструктура буде лише мінусом: розподілені системи з користувачами, які знаходяться в різних точках світу, також IoT та задачі з передбачуваними піками навантаженнями.

Наступним етапом розвитку, який вирішував проблеми локальної інфраструктури є хмарні системи. Це до сьогодні залишається інноваційним рішенням, яким користуються велика кількість людей та компаній. По суті, хмарна інфраструктура забезпечує аналіз, обробку та збереження даних. Така концепція дозволяє вигідно працювати усім користувачам системи – від

власника до розробника. Головними перевагами є масштабованість (можливість розширювати або зменшувати ресурси відповідно до задачі), швидка адаптація до трафіку та навантаження на систему, оплата лише за те, що використовуєш та порівняно мінімальні потребами в обслуговуванні(low maintenance).[5] Проте з появою IoT, mobile computing та Big Data, проявилися відчутні проблеми, які хмарні обчислення не могли вирішити. Перш за все, велике навантаження на пропускну здатність мережі, коли ми маємо потребу в обробці даних у режимі реального часу.[1] Тож для задач, у яких ми потребуємо швидкий результат, даний підхід не підходить, бо будуть затримки відповідей, що неприйнятно для роботи з медичним обладнанням, керування дронами, машинами, аналізу відео чи сканування зображень. Тобто будь які задачі, які вимагають відповідь на запит зараз. Іншим мінусом є як раз таки, те що є перевагою в on premise системах – конфіденційність користувача. Через роботу з даними юзера на хмарі, є вищий ризик витоку даних чи їх втрати. Через це хмарні обчислення є не найкращим рішенням для задач, які генеруватимуть сирі дані по типу необроблених відео фрагментів чи сканування. Третім недоліком є проблема енергоефективної роботи. Попри факт того, що енергоефективність хмарних обчислень поступово зростає, цього не вистачає, щоб відповідати швидко зростаючим енергетичним потребам обробки даних, бо суспільство зростає як і кількість продукованих даних.[1]

Отож з огляду на потреби, які мали нові технології, виникає нова парадигма “обчислень на краю мережі”. Ключовою відмінністю є те, що периферійні обчислення знаходяться й відбуваються близько до джерела даних й концепція близькості грає важливу роль. Ресурси такі як клаудлети, мікро датацентри, туманні вузли (fog nodes) будуть розташовані на краю мережі, близько до джерела даних. Отже, периферійні обчислення за своєю

семантикою є хмарними обчисленнями, перенесеними ближче до джерела даних.

Обчислення на периферії працюють локально, не завантажуючи нічого на хмару. Через цей аспект можна виділити наступні переваги. Перш за все, на відміну від хмарних обчислень, периферійні буквально створені для задач, які вимагають роботи й реагування у реальному часі. На периферійному вузлі буде відбуватися обробка й зберігання даних, що дає швидкість проміжного процесу передачі даних й дає мінімальну затримку. До того ж, різноманіття способів обробки інформації на периферії є значним. Це можуть бути як різні методи агрегації даних, так і використання алгоритмів штучного інтелекту (ШІ), зокрема машинного навчання (МН), безпосередньо на периферійних пристроях. Такий підхід дозволяє здійснювати попередню обробку, фільтрацію, класифікацію чи прогнозування на місці, зменшуючи обсяг даних, що передаються в хмару, та прискорюючи прийняття рішень у реальному часі. По- друге, як вже було згадано, перевагою є безпека даних, бо на відміну від хмари, дані обробляються локально й не передаються мережею. В разі атаки під загрозою лише невелика частина даних. Третьою перевагою, яку варто згадати, є порівняно низьке енергоспоживання. Периферійні обчислення зменшують вимоги до пропускної здатності мережі, адже обробка відбувається локально. До того ж, це фінансово вигідно, можна контролювати кошти, які витрачати на пристрої на краю.

У підсумку, периферійні обчислення є таким собі непрямим поєднанням on premise систем та хмарних обчислень, оскільки поєднують локальну обробку даних, характерну для on-premise, із можливістю взаємодії та обміну даними з хмарою, як у хмарних обчисленнях. Це дозволяє, наприклад,

обробляти дані від IoT-пристроїв чи мобільних телефонів локально, а результати зберігати або аналізувати в хмарі.

2.2 Архітектура периферійних обчислень для мобільних застосунків

Якщо розглядати архітектуру периферійних обчислень, то варто зрозуміти одне – концепція хмарних обчислень переноситься ближче до користувача. Тож загальна архітектура периферії буквально відображає дану ідею й має три шари.[11]

Першим є термінальний рівень. Це шар пристроїв, які є джерелами даних. Це може бути будь який тип пристрою, який приєднаний до периферійної системи – IoT чи мобільні девайси. Основна функція термінального рівня — генерація початкових даних про навколишнє середовище, користувача або об’єкт, що перебуває під моніторингом. Пристрої цього рівня зазвичай мають обмежені обчислювальні ресурси та обмежену енергоємність, тому вони рідко виконують складну обробку, натомість передають зібрану інформацію на вищий рівень для подальшої обробки.[11]

Наступним шаром є периферійний рівень – те, що якраз відрізняє периферійну парадигму від будь яких інших концепцій. Він розташований між пристроями термінального рівня та хмарою. Даний рівень складається з вузлів – базові станції, клаудлети, шлюзи тощо. Цей рівень комунікує з термінальним рівнем, приймає від нього дані й займається обробкою та аналізом. Залежно від подальшої логіки, дані можуть бути передані на хмару для обробки, або ж результат буде повернуто назад на термінальний рівень.

Останнім рівнем є хмара. Вона має потужні сервери та велику кількість ресурсів для зберігання, аналітики та централізованого управління даними. На

даному шарі можна обробляти дані на більш глибокому рівні й здійснювати складний аналіз та обчислення, на які немає ресурсу у периферійних пристроїв. Хмарний рівень отримує вже попередньо оброблені та агреговані дані від периферійного рівня, що дозволяє зменшити трафік і затримки в мережі.

Тож доволі ефективним є поєднання периферійних обчислень та хмарних обчислень – периферійні добре працюють з задачами в реальному часі, хмара має більші обчислювальні потужності.

За описаною вище загальною архітектурою можна будувати кастомізовані архітектурні рішення вже для конкретних задач. На рисунку (див. Додаток 3) відображено загальну схему архітектури. Розгляньмо різні варіанти побудови периферійної системи.

Перш за все, Client-Edge архітектура. Клієнтська сторона збирає всі необхідні дані для обчислень й оформлює це в певну структуру, передбачену дизайном додатку, яка буде надсилатися (offloading) до `Edge` сторони. На периферії обчислюється результат на основі отриманих даних й надсилається назад до клієнта.[6] Прикладом такої архітектури може бути мобільний пристрій, який збирає дані про положення тіла у просторі й надсилає на периферійний вузол ці дані для обрахунку покриття об’єкту у просторі, щоб після того як мобільний пристрій отримає результат – могти різним чином використовувати ці дані залежно від призначення додатку.

Іншим варіантом є Client–Edge–Cloud Architecture. По суті, у цьому дизайні додається ще сторона хмари. Периферійний вузол у даному випадку слугуватиме місцем, де відбувається попередня обробка даних з метою зменшити розмір, лишити лише необхідну інформацію. Далі агреговані дані передаються до хмари для більш глибокої аналітики чи обробки, або ж для зберігання. Прикладом даної архітектури може бути передавання потоку

кадрів (відео) від мобільного застосунку до периферійного пристрою. Буде відбуватися попередня обробка – замилювання облич людей для безпеки й надсилання результатів на хмару для зберігання.

Ще однією варіацією є архітектура Edge-Cloud. У даному підході клієнт не буде грати ніякої активної ролі за винятком надсилання даних. На периферії відбувається кешування, обробка даних, їх оптимізація. На хмарі знову ж таки може бути навчання моделі машинного навчання, глибша аналітика або ж збереження результатів. Прикладом є IoT сенсор, який збирає інформацію з навколишнього середовища, надсилає на периферійний рівень, який час від часу синхронізується з хмарою, завантажуючи туди дані.

Також можна зазначити, що периферійний рівень може бути не одним, натомість периферія може складатися з багатьох рівнів. Дана архітектура має назву Hierarchical Edge (ієрархічна периферійна архітектура). Підхід представляє собою модель багаторівневих обчислень. Дані проходять через шари периферійних вузлів, а потім надсилаються на хмару. На першому етапі – пристрої, які є джерелами даних. Далі йтиме близький периферійний пристрій, який, як правило, робить фільтрацію/агрегацію. Потім оброблені дані надсилаються на потужніший рівень представлений шлюзами, який збиратиме дані й агрегуватиме їх.[7] Кінцевим рівнем є хмара. Звичайно, кількість периферійних рівнів може варіюватися, це залежить від вимог конкретної задачі. Прикладом застосування даної архітектури може бути медична система, яка передбачатиме передачу даних з датчиків пацієнта до периферійного пристрою лікарні – якийсь конкретний комп'ютер прив'язаний до датчику. Далі центральний периферійний вузол, який збиратиме всі дані з вузлів й після агрегації зберігатиметься на хмарі для подальшого аналізу.

Останньою архітектурою є `Mobile Edge computing`. Можна сказати, що це ще більш швидкий та ефективний підхід, аніж поширена на сьогодні

класична `Client-Edge-Cloud`. `Mobile Edge computing` є архітектурою, яка передбачає обчислення та зберігання даних на базових станціях мобільного зв'язку. Ця архітектура ще краще підтримує низьку затримку з хорошою роботою в реальному часі та високу пропускну здатність[7]. Цікавим є той факт, що `Mobile Edge` може представляти собою окремий рівень у ієрархічній периферійній архітектурі. Прикладом такої архітектури може бути оперативне виявлення дронів у реальному часі.[14] `Mobile Edge` буде працювати з даними ближче до джерела, зменшуючи затримку й збільшуючи швидкість реагування.

Для даної кваліфікаційній роботі обрано задачу, для якої є метою підвищити продуктивність обробки даних за допомогою периферійних обчислень. Суть полягає у реалізації задачі сканування зображення на мобільному пристрої й знаходження зісканованого об'єкту у просторі. З огляду на вищеперечислені архітектури, для даної задачі обрано Client-Edge підхід, де клієнтом виступатиме мобільний пристрій, а периферійним вузлом – комп'ютер. Це обґрунтовано наступним чином: мобільний пристрій має обмежені ресурси для обробки великої кількості даних. Сканування вимагає складних обчислень, які здійснюють помітне навантаження на CPU телефону. Вивантаження частини обчислень на периферійний вузол в теорії підвищить швидкодію, мінімізує затримки й не буде забирати багато ресурсів мобільного пристрою. Також використання обчислень на краю дозволить підвищити конфіденційність користувача.

В представленій задачі, `Client-Edge` архітектура є балансом між швидкодією, безпекою та ефективним використанням ресурсів телефону.

2.3 Підходи до розподілу обчислень між пристроями

Одним з ключових моментів, які визначають суть периферійних обчислень, є питання, коли і куди пересилати дані. Суть вивантаження даних полягає в тому, щоб перенести обчислення з пристрою, який має обмежені ресурси (батарея, пам'ять наприклад) на потужніший периферійний вузол – комп’ютер, сервер, MEC (mobile edge) сервер тощо. Вивантаження має два етапи: рішення яка частина обчислень пересилається на периферію, та визначення, куди конкретно дані надійдуть. Розглянемо детальніше кожен з етапів.

Спершу, потрібно прийняти рішення, яку частину обчислень надсилати, базуючись на трьох аспектах: спосіб вивантажити обчислювальну задачу, кількість даних, та які конкретно задачі/дані пересилати. Традиційно система прийняття рішення складається з трьох модулів: парсер, системний моніторинг та кастомний механізм прийняття рішення.[12] Парсер буде аналізувати сам код й визначати яку його частину вигідно пересилати. Зазвичай це будуть частини, де відбувається складні з точки зору навантаження на систему обчислення. Далі системний моніторинг відповідає за рішення з точки зору стану системи. Важливим аспектом є те, скільки ресурсів є локально – заряд батареї, кількість вільної пам’яті та оцінка завантаженості процесора, також співставлення кількості даних, які необхідно обробити та якості зв’язку (і взагалі чи він наразі стабільний). Останнім блоком є механізм прийняття рішення, логіка якого буде залежати від конкретної задачі. По суті, він збирає висновки, отримані від попередніх двох модулів й приймає зважене рішення.[12] Інформація, на основі якого робиться вибір, така: є частини коду, які можна вивантажувати, стан пристрою (батарея, cpu, ram), стан зв’язку з периферією та критерії задачі (кількість даних,

критичність). Можливі три сценарії: вивантажувати всю задачу, її частину або ж не переносити взагалі, бо в цьому немає сенсу й краще лишити обчислення на локальному пристрої. Механізм прийняття рішень може бути реалізований різними способами. Перш за все, він може бути `rule-based`, а саме прописані правила, які кажуть що робити залежно від стану системи. Також це може бути `cost function`, яка буде враховувати ваги для кожного з параметрів, а також модель машинного навчання, яка класифікуватиме задачу. На сьогодні є багато робіт, які досліджують різні моделі машинного навчання та використання штучного інтелекту для ефективного підходу прийняття стратегії вивантаження задачі. Це має сенс для мобільних периферійних обчислень та ієрархічних архітектур.

Іншим важливим аспектом є вибір типу зв’язку, який буде встановлено між локальним пристроєм та периферією. Від цього напряму буде залежати логіка механізму прийняття рішень, адже технологія, яка буде встановлювати зв'язок, буде впливати на затримки, пропускну здатність та залежність від інтернету. Тож було проаналізовано декілька підходів до встановлення зв’язку.

Класичним підходом для IoT девайсів є протокол MQTT. Він спеціально спроектований для менш потужних пристроїв та нестабільних мереж, працює поверх TCP протоколу. Перш за все, даний протокол має модель клієнта – брокера. Клієнтом буде виступати пристрій, який підписується на певну тему, по якій хоче отримувати повідомлення. Інший клієнт буде видавцем, тобто публікуватиме ці повідомлення. Брокер – це центральний сервер, який передає повідомлення між видавцями та підписниками. Також він відповідальний за встановлення зв’язку, авторизацію та автентифікацію MQTT клієнтів та передавання повідомлень до інших систем (бази даних, аналітичні сервіси

тощо). Даний протокол підтримує три рівні QoS – рівні впевненості доставлення повідомлення до клієнта. QoS 0 використовується для повідомлень, які будуть втрачені при недоступному клієнті, QoS 1 гарантує доставку, проте можуть траплятися дублікати. QoS 2, відповідно, є найбільш надійним й використовується для критично важливих даних. Іншим механізмом, який підтримується даним протоколом, є постійні сесії, які дозволяють не втратити підписки та невідправлені повідомлення. Це особливо є плюсом при нестабільному Інтернет з'єднанні. Працює наступним чином: при від'єднанні клієнта від брокера, брокер зберігає його підписки та повідомлення, які на них приходили.

Іншим способом комунікації між периферією та локальним пристроєм є Web Socket. Це мережевий протокол, який працює поверх TCP та забезпечує двостороннє з'єднання між сервером та клієнтом. Це протокол, який має стан, тож його канал комунікації між сервером та клієнтом не закриється доки цього не зроблять вручну. Працює підхід наступним чином: клієнт буде ініціатором з'єднання, й подасть запит, який в цьому контексті називається рукостискання(handshake).[10] Сервер приймає та підтверджує запит. Після цього канал зв'язку встановлено – клієнт та сервер можуть обмінюватися повідомленнями. У рамках архітектури периферійних обчислень, клієнтом є локальний пристрій, а сервером – периферія. Даний протокол відповідає вимогам швидкодії та роботи в реальному часі, додатково можна додати безпеку у формі шифрування чи токенів.

Останнім способом встановлення зв'язку, який був досліджений, є Multipeer Connecivity, який працює лише на Apple продуктах. Саме він був обраний для інтеграції у додаток для сканування зображень, як технологія для комунікації мобільного пристрою та периферії. Multipeer Connecivity є peer-to-

peer технологією Apple, яка дозволяє створювати локальну мережу між пристроями, й вони будуть обмінюватися даними.[9] Peer-to-peer означає, що з’єднання не потребує маршрутизатора й працює напряму. Інтернет з’єднання не потрібне, бо Multipeer Connecivity використовує Bluetooth та peer-to-peer Wi-Fi. Безпека передачі даних також продумана – присутній алгоритм шифрування з’єднань AES.

Враховуючи те, що практичне дослідження базується на створенні периферійної системи саме для пристроїв IOS, вирішено обрати саме Multipeer Connecivity технологію, яка буде забезпечувати комунікацію локального пристрою та периферійного вузла. Вона задовольняє всі вимоги для мережевого зв’язку, адже забезпечує швидкодію та має можливість передавати великі за обсягом дані. Детальніше про реалізацію у наступному розділі.

РОЗДІЛ 3. Реалізація розподіленої системи

3.1 Порівняння двох різних задач й оцінка ефективності реалізації периферії для кожної з них

Як уже згадувалося, не для всіх типів задач ефективно переносити частину обчислень на периферію, бо це може бути невиправдано й не принести ніякої вигоди в контексті ефективності. У даному підпункті буде проведено дослідження введення периферійних обчислень до архітектури двох різних задач та порівняння їх результатів. Отже, головний аспект, який визначає доцільність вивантаження – чи обсяг переданих даних невеликий порівняно з обсягом обчислень й чи витримає обраний канал зв’язку відповідне навантаження.

Тож першою задачею є реалізація розпізнавання людей з відео потоку у реальному часі. Реалізований модуль передає стиснутий потік кадрів відео від мобільного IOS пристрою до периферійного – MacBook. Способом комунікації обрано протокол MQTT. Архітектура системи є наступною: iPhone виступає клієнтом, який здійснює захоплення кадрів за допомогою AVFoundation й розпізнає людей за допомогою YOLOv3 CoreML моделі. Здійснюється моніторинг ресурсів, якщо показник занизький – пересилаємо стиснені з використанням H.264 кадри через протокол до MQTT підписника й за сумісністю периферійного пристрою. Edge отримує кадри, здійснює розпізнавання людей через OpenCV та передає виконану аналітику назад на мобільний пристрій. Здавалося б, така архітектура мала би лише збільшити ефективність здійснення аналізу відео, адже вона поєднує ефективне стиснення даних, локальну обробку та вивантаження даних до периферії, що зменшує навантаження на мережу й дозволяє виконувати розпізнавання на

більш потужному пристрої. Проте при реалізації виникли наступні проблеми: були суттєві затримки між моментом надсилання кадру й його отриманням на периферії. Це обумовлено тим, що протокол MQTT не оптимізований для передачі мультимедійного контенту у реальному часі. У даному протоколі більший акцент на гарантію доставки повідомлення, а не швидкість передачі. Іншою суттєвою проблемою було те, що обраний брокер HiveMQ був перевантажений й досить швидко досягав критичного рівня навантаження, що призводило до проблеми втрати пакетів даних. Була здійснена спроба пересилати по одному кадру, це знизило навантаження на брокер, проте унеможливлює швидку аналітику в реальному часі, здійснюючи затримки.

Тож дана архітектура виявилася невигідною з точки зору затримки повідомлень та стабільної роботи. В даному випадку немає необхідності вивантажувати обчислення на периферію, адже оптимізовані для платформи Apple технології, такі як AVFoundation та Vision, ефективно виконують роботу, а процес пересилання даних на периферію лише обтяжує та сповільнює виконання. А обсяг даних не виправдовує складність застосованих обчислень.

На противагу цій задачі, було розглянуто задачу сканування об’єктів та їх розпізнавання за допомогою створеного reference object на мобільному пристрої – iPhone з Lidar (технологія, яка дозволяє вимірювати відстань до об’єкту за допомогою променів). Потенційно, Lidar дає багато необроблених точок й мобільний пристрій не завжди має ресурси створити повноцінну модель. Технічна суть задачі полягає в наступному: використання ARKit для сканування обраного об’єкту на сцені: отримання сирих точок, формування bounding box, створення та оновлення об’єкту для сканування. Оновлення точок відбуватиметься на кожному кадрі доки не буде повного покриття щоб мати змогу створити reference object – результат сканування. Далі буде

можливість зберегти зіскановану модель – можна надсилати її через соціальні мережі або ж здійснювати пошук цього об’єкту в просторі у додатку. Архітектура даної задачі є знову ж таки `Client-Edge`, де клієнтом виступає мобільний пристрій, який трасує та збирає промені й надсилатиме їх за допомогою `Mutlipeer Connectivity` до периферійного пристрою – MacBook для обчислення покриття. Присутній механізм прийняття рішень, який працює реактивно, й реагує на зміну показників ресурсів мобільного пристрою.

У даному розділі буде продемонстровано, що дійсно задача сканування буде вигравати в контексті швидкодії, якщо ввести периферійні обчислення. Теоретично це пояснюється тим, що обсяг даних та обчислень виправданий – невелика кількість даних у поєднанні зі складністю та частотою обчислень робить доцільним їх виконання на периферії. Mutlipeer Connectivity не потребує Інтернету та підтримує стабільний зв'язок між пристроями.

Отже, не всі задачі вигідно переносити на периферію. При рішенні варто чітко визначити, які частини обчислень доцільно делегувати периферії, чи розмір переданих даних пропорційний до складних та громіздких обчислень, а також враховувати якість каналу зв’язку та технічні можливості передачі даних.

3.2 Реалізація задачі сканування на пристроях iOS

Для демонстрації підвищення продуктивності обробки даних за допомогою периферійних обчислень при використанні стратегії обрано задачу сканування зображень. У даному підпункті буде описано технологічний аспект реалізації механізму сканування. За основу взятий проект з документації Apple - Scanning and Detecting 3D Objects[8]. Сканування реалізоване на основі

технології ARKit. Головні можливості фреймворку є наступними: відслідковує пристрій у просторі, враховує дані з акселератору та гіроскопу для визначення його позиції у просторі, визначає поверхні та створює якорі, функція яких – фіксувати певні точки в реальному світі. ARKit, як і багато технологій у Apple, працює за допомогою патерну делегування. Головний клас ViewController наслідує ARSessionDelegate й таким чином є делегатом AR сесії. Цей контролер реагуватиме на всі зміни у сесії. AR сесія є основою роботи ARKit, яка відповідає за обробку вхідних даних з камери, аналіз навколишнього середовища відбувається безперервно. Визначення делегату показано на рисунку (див. Додаток 4)

Програма матиме чотири стани роботи: initializing, boundingBoxReady, scanning та completed, рис. 3.2.2.

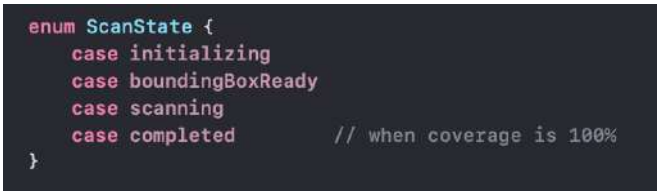


Рис. 3.2.2

Залежно від стану, програма перебуватиме на різних етапах. Initializing: додаток ініалізує AR сесію та якщо камера знаходиться у .normal положенні, додаток переходить у стан boundingBoxReady й повідомляє користувача здійснити натиск на екран по об’єкту, який хочеться відсканувати – ця дія дозволить перейти до scanning стану. Функція обробки натиску на екран відображена на рисунку (див. Додаток 4). Далі завдяки важливому методу делегату, який здійснює рендеринг сцени й викликається на кожному кадрі, перевіряється наявність умови стану scanning, й викликається метод, який працює на кожному фреймі - `updateOnEveryFrame (_ frame: ARFrame)`, рис. 3.2.3.

```
func renderer(_ renderer: SCNSceneRenderer, updateAtTime time: TimeInterval) {
    guard let frame = sceneView.session.currentFrame else { return }
    scan?.updateOnEveryFrame(frame)
}
```

Рис. 3.2.3

Він є центром програми, де працює вся основна логіка. Саме тут беруться нові сирі точки фрейму й оновлюється bounding box, періодично відбувається спроба створити об’єкт-референс. Даний метод відображено на рисунку (див. Додаток 5). Якщо його створено, переходимо до стану completed. Якщо ні, сканування продовжується викликом ще одного ключового методу – `updateCapturingProgress()`. Він слугує оцінкою прогресу сканування: оновлення інформації про покриті камерою частини об’єкту й обчислює відсоток зісканованості. Стандартний метод буде додавати нові промені та точки їх перетину з об’єктом, позначати зіскановані плити й сповіщати інші частини програми про відсоток завершеності сканування. При переході на completed уже створено об’єкт-референс, який користувач може зберегти або надіслати комусь.

Отже, при побудові архітектури `Client-Edge` постає питання як реалізувати систему прийняття рішення вивантаження даних: що, куди, коли. Спершу визначимо що пересилати. Як вже було описано у попередньому розділі, традиційно цей механізм складається з трьох модулів: парсер, системний моніторинг та кастомний механізм прийняття рішення. Після аналізу базового коду сканування, було прийнято рішення пересилати дані для всіх основних обчислень, які відбувалися у методі updateCapturingProgress. Саме він має основне обчислювальне навантаження в контексті обрахунків, які не залежать від дій користувача чи UI програми. І що важливо, цей метод викликається на мобільному пристрої щокадрово, що може значно впливати на ефективність роботи.

Тож вирішено реалізувати окремий метод updateCapturingProgressOnMac, який буде збирати дані у масив та пересилати на MacBook, відображено на рисунку (див. Додаток 6) – SIMD3<Float> position та SIMD3<Float> extent надсилаються для того, щоб периферійний пристрій розумів положення bounding box у просторі та cameraRaysAndHitLocations (origin, direction) – кортеж, який зберігає промінь та точку його перетину з об'єктом. Коли Edge пристрій отримує та десеріалізує дані, відбувається логіка визначення поточного покриття об'єкту: поточний переданий bounding box ділитиметься на менші частинки (куби) шляхом циклічного створення плиток з розділенням 4 – оптимальний розмір для малих та середніх bounding box, для кожної з яких перевіряється чи попадає у неї хоча би один промінь (відхилення – 3 см) й підраховується відсоток плиток, які покриті. Суть полягає в обрахуванні загальної кількості плиток, та плиток, які перетнулися з променем. Ми обраховуємо сам вектор напряму променю й вираховуємо відстань до поточної координати куба (частини bounding box). Ми можемо обрахувати відстань на основі евклідової метрики: $\text{simd\_distance}(\text{a}, \text{b}) = \sqrt{(\text{a.x} - \text{b.x})^2 + (\text{a.}y - \text{b.y})^2 + (\text{a.z} - \text{b.z})^2)}$. Якщо обрахована відстань менша за похибку 3 см, даний промінь зараховується до числа влучених. Відсоток обраховується звичайним чином, кількість влучених ділиться на загальну кількість. Коли покриття досягає > 98%, запаковуємо результат шляхом серіалізації й передаємо результат до мобільного пристрою. Даний метод відображений на рисунку (див. Додаток 7).

Інші обчислення, які є менш складними та ресурсозатратними порівняно з обрахунком покриття, але тим не менше мають сенс у перенесенні обчислень на периферію – обчислення нових координат bounding box на основі нових сирих точок, отриманих з рендерингу сцени. Тож залежно від обраної стратегії,

ми будемо пересилати точки на периферію. Ідеєю є профільтрувати точки, які не дають ніякої інформації й просто створюють шум. Ми у циклі обраховуємо відстань від поточної точки до всіх інших, й якщо вона менша за мінімальну – 2 см, то це вважається сусідньою точкою й за умови наявності більше двох сусідів, ми зараховуємо точку до відфільтрованих точок. Даний підхід є фільтруванням на основі щільності. Знову ж таки, відстань обраховується на основі евклідової метрики. Для обрахунку нового центру bounding box, ми беремо середнє між мінімальними та максимальними координатами відфільтрованих точок, для розміру – різницю. Результат серіалізується й надсилається на локальний мобільний пристрій. Даний метод відображений на рисунку (див. Додаток 8).

Отже, реалізовано базову спрощену задачу сканування, де за основу взятий проект з офіційної Apple документації. Створено архітектуру Client-Edge, де периферія – MacBook, а клієнт – iPhone. На периферію надсилається два різних шматки обчислень, які не залежать від UI та дій користувача: обчислення покриття сканування, та нових координат/розміру bounding box.

3.3 Реалізація реактивного механізму прийняття рішення

Важливо реалізувати механізм прийняття рішення. У пункті 3.2 було визначено, які частини обчислень ми будемо передавати. Тепер, аби програма дійсно мала дієву периферійну архітектуру, варто реалізувати блок моніторингу системи й механізм прийняття рішення, яке визначатиме, коли дані варто пересилати. Було вирішено доцільним реалізувати це за допомогою реактивної парадигми програмування – це парадигма, побудована на потоках даних і розповсюдженні змін. Обрано бібліотеку RxSwift для реалізації реактивної частини на стороні клієнта.

RxSwift реалізує реактивну модель за патерном Observable, де дані та події передаються потоками, на які можна реагувати. Основними поняттями є суб'єкт(буде змінювати свій стан, Observable) та спостерігач (реагуватиме на зміну стану суб'єкту, Observer). Спостерігач здійснює слідкування за допомогою механізму підписки. Результатом підписки є об'єкт типу Disposable, який є обгорткою навколо підписки. Також дана бібліотека має такий тип об'єкта як Subject, який є одночасно і observer, і observable, що означає можливість на них як підписуватись так і передавати події.

Отже, було реалізовано клас SystemMonitor, який здійснює моніторинг трьох системних ресурсів: CPU, RAM та рівень заряду батереї. Для кожного з показників створено змінну типу BehaviourSubject<Double>, тобто Subject, який зберігатиме значення останньої події. Фокальною функцією є startMonitoring(), суттю якої є оновлювати значення системних ресурсів. Працює це наступним чином: здійснюється підписка на Observable, який спрацьовує кожні 5 секунд й буде надсилати через .onNext нові обчислені значення кожному з системних ресурсів. Окремо реалізовані функції обчислення поточного стану ресурсів. Даний клас відображено на рисунку (див. Додаток 10)

Головним класом є TaskManager, де є екземпляр класу SystemMonitor та реле strategy, яке має тип BehaviorRelay<ExecutionStrategy>(value: .local) – тобто задає початкову стратегію – обраховувати локально. ExecutionStrategy є структурою даних Enum, де маємо два кейси: local та offloading. У конструкторі класу за допомогою combineLatest оператору об'єднуються потоки значень із cpuUsage, ramUsage та batteryLevel, щоб отримувати останнє значення з кожного джерела при будь-якій зміні. Коли зміниться хоча б одне значення – блок буде виконуватися. Далі використовується оператор throttle, щоб

обмежити частоту обробки об’єднаних значень: нове значення буде приходити не частіше ніж раз на 3 секунди. У замиканні викликається функція обчислення стратегії. Отримане значення надсилається до strategy реле через .accept.

Ідея адаптивного механізму прийняття рішення базується на порівнянні поточного стану пристрою з відповідними граничними значеннями, що визначає стратегію виконання – локальну чи вивантаження. Для того аби підхід працював, необхідно мати порогові значення для CPU, RAM та рівня заряду батареї, при яких вивантаження даних буде давати найкращі результати в контексті швидкодії. З метою знайти ці значення було проведено низку експериментів. Для задачі сканування об’єктів емпірично досліджено комбінації різних порогів для трьох показників: для кожного був заданий поріг від 0 до 100% з кроком 20%. Стратегія прийняття рішення є параметризованою, тож тестування було проведено комбінуванням варіантів показників між собою.

Для кожної конфігурації фіксувалися три пороги:

- CPU threshold – максимальний рівень навантаження процесора
- RAM threshold – максимальний рівень використання оперативної пам’яті
- Battery threshold – мінімальний рівень зарядженості батареї, при якому виконуються локальні обчислення

Протягом проведення тестування фіксувалися: поточна стратегія виконання (.local чи .offload), момент перемикання стратегії та загальний час виконання у хвилинах.

Вплив CPU є наступним: аналіз показав нелінійний вплив порогу на час виконання. При переході від 20% до 40% та фіксованою оперативною

пам’яттю 40%, тривалість зросла з приблизних 2-ох до 3-ох хвилин, бо довше утримується виконання на локальному пристрої, незважаючи на зростаюче навантаження. Ідентичною є тенденція при високому порозі 80% - тривалість роботи збільшується (приблизно 2 хвилини). Найоптимальнішою виявилась конфігурація з CPU 60%, час виконання дорівнює 41 секунді – при суттєвому збільшенні навантаження відбувається offloading. При фіксуванні інших відсоткових значень для RAM (60%, 70%, 80%) тривалість роботи коливається в інтервалі від одної до двох хвилин.

Попри початкове рішення тестувати пороги RAM із кроком 20% (тобто 40%, 60%, 80%), результати експериментів вказали на підвищену чутливість системи в межах 60-80%. Зокрема, для порогу RAM = 70% було зафіксовано найнижчий час виконання задачі у відповідних комбінаціях. У зв’язку з цим було додатково включено поріг у 70% до аналізу, аби точніше виявити оптимальні умови для обрання стратегії.

Тож щодо впливу RAM, можна виділити наступні ключові закономірності: із зафіксованим CPU = 40%, чітко простежується вплив рівня використання оперативної пам’яті на час виконання: зростання порогу призводить до суттєвого зменшення показника часу (від приблизних 3-ох хвилин до однієї). Це пояснюється тим, що при вищих значеннях RAM, механізм прийняття рішення обирає стратегію вивантаження й відповідно обрахунок відбувається швидше. Для фіксованого значення CPU = 80% поведінка є ідентичною. Водночас, при CPU = 60% результати мають нелінійний характер, оскільки оперативна пам'ять зі значенням 40 та 70% стабільно демонструє швидший результат на приблизно 40 секунд порівняно зі значеннями 60 та 80%.

Поєднання високих порогів для CPU та RAM у більшості випадків перевантажує локальний пристрій, що призводить до збільшення часу

виконання програми. На противагу високим значенням, низькі також не дають найкращі результати, що можна пояснити постійним пересиланням даних через мережу, що сповільнює час виконання. Найефективнішими є результати для середніх значень CPU та оперативної пам’яті – стратегія є гнучкою та активно перемикається між вивантаженням та локальними обчисленнями. Це чітко видно з наступних комбінацій: при CPU = 40% та RAM = 70% механізм прийняття рішень перемикається між стратегіями при цьому з тенденцією до переважного пересилання обрахунків на периферію, при 60/80 стабільне чергування локальних та периферійних обчислень, й при 80/70 це переважно локальний пристрій.

Обчислені результати відображені на графіку (див. Додаток 12). Як можна побачити, найкращими показниками порогів є наступні: CPU – 40%, RAM – 70%, батарея – 20%. Також зроблено висновок, що значення рівня зарядженості батареї найменше впливає на швидкодію. Аналіз зібраних даних показав, що за умов порогового значення battery > 20%, яке переважно використовувалось у тестах, стратегія перемикання практично не змінювалась. Це пояснюється тим, що механізм прийняття рішень активніше реагує на CPU та RAM, оскільки вони є критичними в контексті навантаження обчислювальних процесів.

Тож ідея умови вивантаження даних наступна, якщо процесор використовується більше ніж на 40%, ram перевищує 70%, а заряд менше за 40% - змінюємо стратегію на offloading, якщо присутній канал зв’язку з периферійним пристроєм. Клас, який реалізує дану стратегію, відображено на рисунку (див. Додаток 11).

Отже, реалізовано механізм прийняття рішень за допомогою реактивної бібліотеки RxSwift. Відбувається моніторинг системних ресурсів й при змінні

значення одного з них, стратегія переглядається з метою оптимізувати розподіл обчислень.

3.4 Реалізація передачі даних від пристрою до периферії

У даному підпункті буде описано реалізацію технології Multipeer Connectivity для встановлення каналу зв'язку між периферією та мобільним пристроєм[9]. Потрібно створити наступну систему, аби налагодити взаємодію: створюється сесія MCSession, яка підтримує канал зв'язку та зберігає ідентифікатори об'єктів (peer), які приєднані до сесії. Далі необхідний об'єкт класу MCNearbyServiceAdvertiser, який буде сповіщати пристрої навколо, що він готовий приєднуватись до відкритих сесій з зазначеним serviceType. У випадку задачі сканування, рекламодавцем виступатиме мобільний пристрій. Периферія ж навпаки виступатиме об'єктом класу MCNearbyServiceBrowser, який буде шукати пристрої, які відкриті до під'єднання. При з'єднанні створиться сесія й кожен пристрій має свій PeerID. Важливо, що serviceType, який є параметром як браузера, так і рекламодавця, має бути ідентичним, якщо метою є об'єднати їх у сесію. На рисунку (див. Додаток 9) зображено ініціалізацію рекламодавця та браузера.

Дана технологія працює за патерном делегування. Головним є MCSessionDelegate, який повідомлятиме про стан з'єднання та отримання/відправку даних. Також маємо MCNearbyServiceBrowserDelegate, який забезпечує знаходження всіх peers та MCNearbyServiceAdvetiserDelegate, який відповідає за прийняття й відхилення запитів на з'єднання.

Для організації передачі даних між пристроями обрано формат JSON – відбувається серіалізація структури, й надсилається формат типу Data, який десеріалізується на стороні отримувача.

3.5 Опис принципів роботи результуючої системи

Додаток Esan призначений для сканування невеликих об’єктів у просторі з можливістю подальшої взаємодії із зісканованим зображенням – зберегти на телефон, надіслати через соціальні мережі та здійснювати пошук цього об’єкту у просторі у рамках додатку. Коли користувач вперше відкриває ESan, йому потрібно надати дозвіл до камери та використання локальної мережі. Далі слідувати інструкціям, які будуть відображатися угорі екрану.

Додаток працює наступним чином:

1) При відкритті застосунку, користувач надає дозвіл на використання локальної мережі та камери.
2) Відбувається ініціалізація AR сесії, додаток переходить зі стану initializing до boundingBoxReady. Користувачу висвітлюється інструкція натиснути на екрані на об’єкт, який би він хотів зісканувати – створюється boundingBox. На цьому ж етапі iPhone починає рекламувати себе з метою знайти периферійний пристрій, з яким встановити зв'язок.
3) Далі користувачу потрібно натиснути на кнопку Scan – створюється об’єкт класу Scan й Esan переходить в стан scanning й створюються екземпляри відповідних класів, які представлятимуть зісканований предмет та хмару точок. Також тут же відбувається підписка на моніторинг стратегії у класі TaskManager.

4) Починається сам процес сканування, де відбувається три фокальні дії – оновлення boundingBox відносно нових точок, рахується покриття зісканованих частин й повторюються спроби створити об’єкт-референс. При цьому, якщо якийсь з системних ресурсів перевантажений чи низька батарея, а саме одне зі значень більше за порогове - CPU – 40%, RAM – 70%, або ж батарея має рівень зарядженості менше 20%, обчислення (покриття + розрахунок bounding box) пересилаються на MacBook, якщо встановлений канал зв’язку – використання технології Multipeer Connectivity.
5) Комп’ютер обчислює й надсилає результати назад. Користувач бачить повідомлення, що сканування завершене. Він має можливість надіслати скан через месенджери та зберегти. Також можна завантажити скан на сцену й додаток виявить цей об’єкт у просторі, якщо він є.

Для фінальної оцінки ефективності перенесення обчислень на периферійний пристрій було проведено профілювання застосунку й заміряно час його роботи із застосуванням периферійної складової та без. При пересиланні даних на периферійний пристрій використано розроблений механізм прийняття рішення, який керується оптимальними порогами показників ресурсів для конкретної задачі обробки даних. В обох випадках відбувалося сканування одного й того ж об’єкту – чашки, профілювання зображено на рисунку (див. Додаток 13). Тривалість виконання задачі при локальних обчисленнях становить 2 хвилини та 70 секунд, тоді як з використанням стратегії – 28 секунд. Отже, маємо зменшення роботи додатку без втрати якості скану на 78.46%, що є показовим результатом того, що дійсно є доцільним використовувати периферійні обчислення для підвищення продуктивності обчислень даних для мобільних застосунків.

ВИСНОВКИ

Результатом даної кваліфікаційної роботи є розроблена стратегія, яка аналізує ресурси локального пристрою й базуючись на цьому приймає рішення про пересилання обчислень. Для того аби продемонструвати ефективність даної стратегії, створено мобільний застосунок Esan для сканування об’єктів у просторі, який має розподілену архітектуру й використовує периферійні обчислення для вивантаження частини процесів з метою пришвидшити процес роботи та підвищити ефективність обробки даних. Дійсно експериментально показано, що обчислення на периферії зменшили час роботи застосунку на 78.46% без втрати якості сканування.

В рамках дослідження було розглянуто два приклади існуючих рішень впровадження периферійних обчислень. Першою моделлю обробки даних, яка була розглянута, є EdgeIoT. Дана архітектура була спрямована на ефективну роботу з IoT девайсами – отримані дані обробляються на локальних туманних вузлах (fog node), які розміщені близько до джерела. Таким чином немає навантаження на мережеву інфраструктуру, що забезпечує швидку обробку даних в реальному часі. Іншим розглянутим прикладом є фреймворк GigaSight, який вирішує проблему обробки відео потоку з мільйонів камер, таким чином, щоб не перевантажувати мережу та хмарний центр обробки даних. Він використовує клаудлети як форму периферійних обчислень. Попередня обробка відео дозволяє забезпечити конфіденційність користувача та зменшити навантаження на мережу та хмарну інфраструктуру.

Як результат, обрано задачу сканування, де чітко виділено обчислення, які теоретично було б ефективно перенести. Емпірично досліджено, при яких порогах значень ресурсів найвигідніше переносити обчислення: CPU – 40%, RAM – 70%, батарея – 20%.

Також було розглянуто різні види архітектур периферійних обчислень й досліджено загальні принципи роботи. З огляду на це, при проектуванні застосунку було обрано Client-Edge архітектуру, де клієнтом є iPhone, а периферією – MacBook. Підібрано зручний механізм створення зв’язку між складовими архітектури та розглянуто принципи роботи реактивного програмування з метою створити підхід для створення модулю для прийняття рішення про поточну стратегію.

Підсумовуючи, проведено тестування створеного застосунку у двох режимах: повний цикл роботи обчислено локально та з перенесенням на периферію. Результат профілювання наступний: крайові обчислення зменшили час роботи сканування на 78.46%. З цього випливає, що мета кваліфікаційної роботи – продемонструвати успішність виконання стратегії переносу на прикладі конкретної задачі сканування та показати ефективність периферії для роботи мобільних застосунків, досягнута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. An Overview on Edge Computing Research / K. CAO та ін. *IEEE Access: Multidisciplinary, Rapid Review, Open Access Journal*. 2020. Т. 8 : Volume 8. С. 85715-85716.

2. Sun X., Ansari N. EdgeIoT: Mobile Edge Computing for the Internet of Things. *IEEE Communications Magazine*. 2016. Т. 16. С. 22–24.

3. Satyanarayanan M., Simoens P., Xiao Y. Edge Analytics in the Internet of Things. *IEEE Computer Society*. 2015. ISSN 1536-1268. С. 26-27

4. Satyanarayanan M., Simoens P., Xiao Y. Edge Analytics in the Internet of Things. *IEEE Computer Society*. 2015. ISSN 1536-1268. С. 25

5. Rudenko D. Edge vs Cloud vs On-Premises: Choosing the Right Model for your Solution. *The manifest*. *Blog \ Development*. 13.01.2023. URL: https://themanifest.com/it-services/blog/edge-computing-vs-cloud-computing-vs-on-premises#author-section.

6. Feiwei L., Yang J., Wenming W. Client-Edge-Cloud coordination Use Cases and Requirements. *W3C*. 26.09.2023. URL: https://www.w3.org/TR/edge-cloud-reqs/.

7. Cannaday B. Hierarchical Edge Computing - A Practical Edge Architecture for IIoT. *Losant*. 11.03.2020. URL: https://www.losant.com/blog/hierarchical-edge-computing-a-practical-edge-architecture-for-iiot.

8. Apple Inc. Scanning and Detecting 3D Objects. *Apple Developer*. *Documentation*. URL: https://developer.apple.com/documentation/arkit/scanning-and-detecting-3d-objects.

9. Apple Inc. Multipeer Connectivity. *Apple Developer*. *Documentation*. URL: https://developer.apple.com/documentation/multipeerconnectivity/.

10. What is web socket and how it is different from the HTTP?. *Geeks for Geeks*. 04.04.2025. URL: https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/.

11. An Overview on Edge Computing Research / K. CAO та ін. *IEEE Access: Multidisciplinary, Rapid Review, Open Access Journal*. 2020. Т. 8 : Volume 8. С. 85717.

12. An Overview on Edge Computing Research / K. CAO та ін. *IEEE Access: Multidisciplinary, Rapid Review, Open Access Journal*. 2020. Т. 8 : Volume 8. С. 85719-85720.

13. Satyanarayanan M. The Emergence of Edge Computing. *THE IEEE COMPUTER SOCIETY*. 2017. С. 30–39. ISSN 0018-9162.

14. Adib D. Mobile Edge Computing: What is Mobile Edge Computing?. Partners. URL: https://stlpartners.com/articles/edge-computing/mobile-edge-computing/.

ДОДАТКИ

Додаток 1

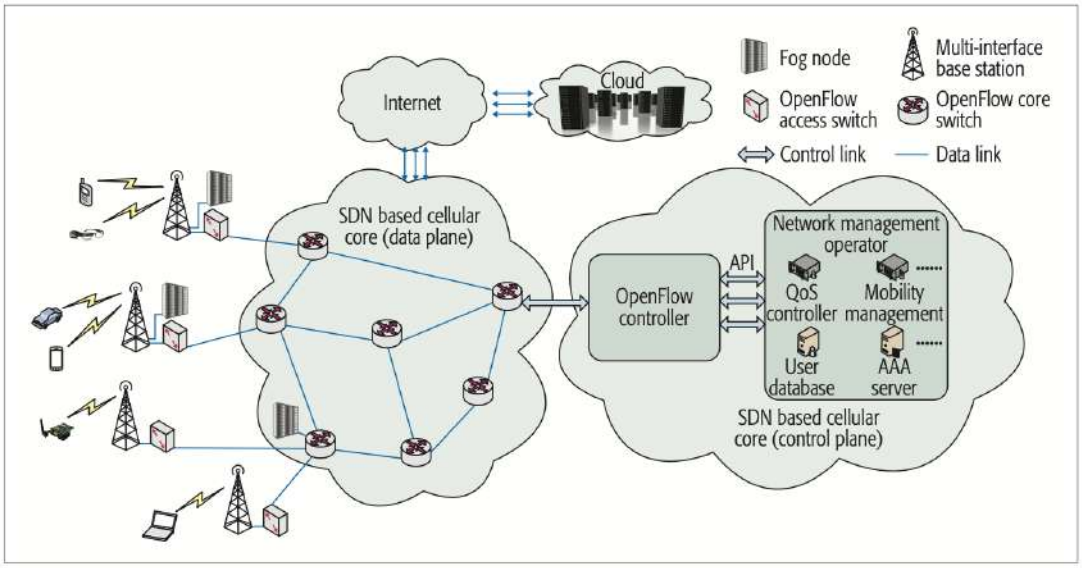


FIGURE 3. The edgeIoT architecture.

Додаток 2

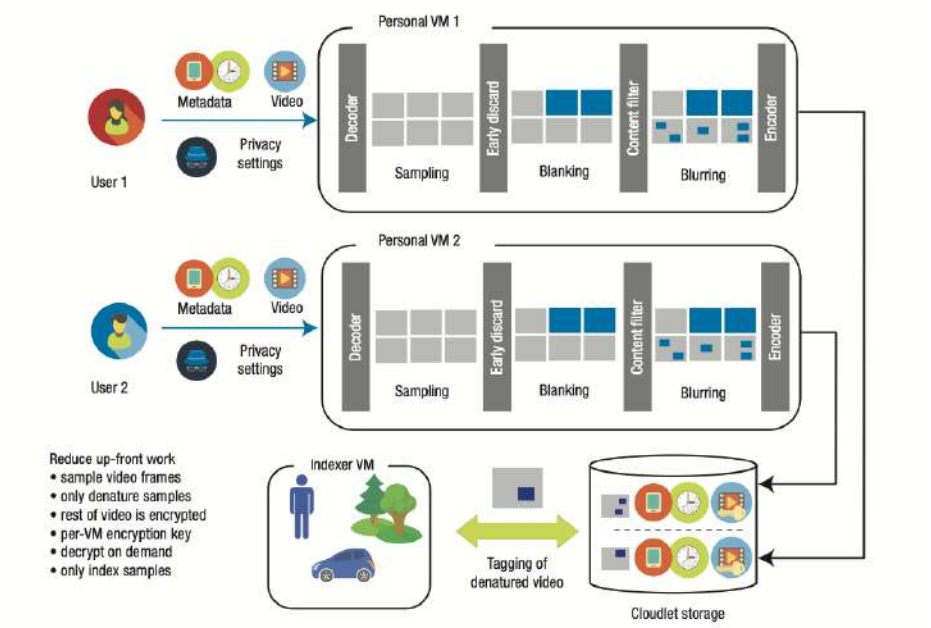
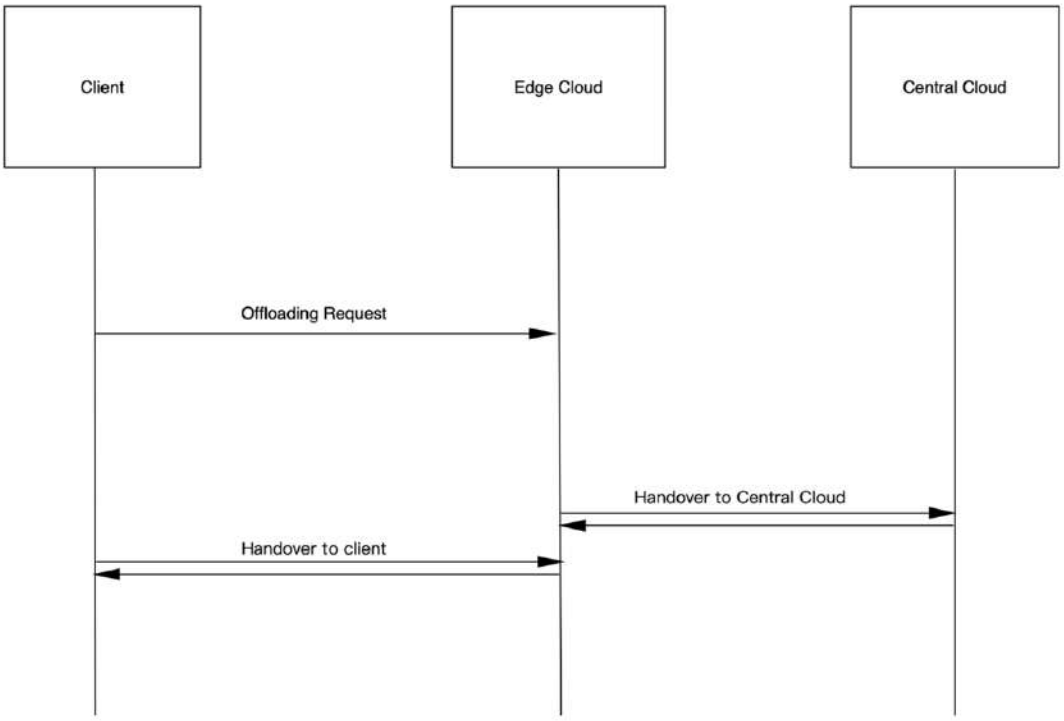


FIGURE 2. GigaSight framework. A cloudlet performs computer vision analytics on video from mobile devices in near real time and only sends the results along with metadata to the cloud, sharply reducing ingress bandwidth into the cloud. VM: Virtual machine.

Додаток 3



Додаток 4

```
override func viewDidLoad() {
    super.viewDidLoad()

    sceneView.delegate = self
    sceneView.session.delegate = self

    UIApplication.shared.isIdleTimerDisabled = true

    let tapGesture = UITapGestureRecognizer(target: self, action: #selector(handleTap(_:)))
    sceneView.addGestureRecognizer(tapGesture)

    startState = .initializing

    let notificationCenter = NotificationCenter.default
    notificationCenter.addObserver(self, selector: #selector(scanPercentageChanged),
                                   name: BoundingBox.scanPercentageChangedNotification, object: nil)

}
```

Рис. 1 Присвоєння делегату сесії

```swift
@objc func handleTap(_ gesture: UITapGestureRecognizer) {
    guard scanState == .initializing || scanState == .boundingBoxReady else { return }

    let tapLocation = gesture.location(in: sceneView)

    if scan == nil {
        scan = Scan(sceneView)
        scan?.state = .ready
    }

    scan?.scannedWrap.createBoundingBox(screenPos: tapLocation)

    scanState = .boundingBoxReady
    print("Bounding box is created")
}
```

Рис. 2 Метод обробки натиску на екран

Додаток 5

```swift
func updateOnEveryFrame(_ frame: ARFrame) {
    switch state {
    case .ready, .defineBoundingBox:
        if let points = frame.rawFeaturePoints {
            scannedWrap.fitOverPointCloud(points)

            frameCount += 1
            if frameCount >= 50 {

                switch currentStrategy {
                    case .offloadToMac:
                        print("iphone send point cloud to Mac")
                        ConnectivityManager.shared.sendPointCloud(points)
                    case .local:
                        break
                    }

                frameCount = 0
            }
        }

        if let boundingBox = scannedWrap.boundingBox,
           CACurrentMediaTime() - timeOfLastReferenceObjectCreation > 1.0,
           !isRefStillCreating {

            isRefStillCreating = true
            timeOfLastReferenceObjectCreation = CACurrentMediaTime()

            sceneView.session.createReferenceObject(
                transform: boundingBox.simdWorldTransform,
                center: SIMD3<Float>(),
                extent: boundingBox.extent
            ) { object, _ in
                if let reference = object {
                    self.pointCloud.update(with: reference.rawFeaturePoints, localFor: boundingBox)
                }
                self.isRefStillCreating = false
            }

            if let currentPoints = frame.rawFeaturePoints {
                pointCloud.update(with: currentPoints)
            }
        }
```

```
    case .scanning:
        scannedWrap.boundingBox?.highlightCurrentTile()
        frameOffloadTestCount += 1;
        if frameOffloadTestCount > 1000 { // тестування
            TaskManager.shared.monitor.simulateHighLoad()
            frameOffloadTestCount = 0
        }
        switch currentStrategy {
            case .local:
                scannedWrap.boundingBox?.updateCapturingProgress()
            case .offloadToMac:
                scannedWrap.boundingBox?.updateCapturingProgressOnMac()
            }

    case .adjustingOrigin:
        break
    }

    scannedWrap.updateOnEveryFrame()
    pointCloud.updateOnEveryFrame()
}
```

Додаток 6

```
func updateCapturingProgressOnMac() {
    let startTime = CFAbsoluteTimeGetCurrent()
    guard let camera = sceneView.pointOfView, !self.contains(camera.simdWorldPosition) else { return }

    frameCounter += 1

    // Додаємо промінь кожні 20 кадрів
    if frameCounter % 20 == 0 {
        frameCounter = 0

        let currentRay = Ray(normalFrom: camera, length: 5.0)

        /**
         Викликається метод tile(hitBy: currentRay) для визначення, чи перетинається цей промінь з якимось об'єктом (tile). Якщо знаходиться точка перетину (hitLocation) і вона відрізняється від попередніх результатів (перевіряється методом isHitLocationDifferentFromPreviousRayHitTests(hitLocation)), то пара (ray, hitLocation) додається до масиву cameraRaysAndHitLocations.
         */
        if let (_, hitLocation) = tile(hitBy: currentRay),
           isHitLocationDifferentFromPreviousRayHitTests(hitLocation) {
            cameraRaysAndHitLocations.append((ray: currentRay, hitLocation: hitLocation))
        }
    }

    // Надсилаємо coverage update кожні 10 кадрів
    guard frameCounter % 10 == 0 else { return }

    let localCenter = SCNVector3(0, 0, 0)
    let worldCenterSCN = self.convertPosition(localCenter, to: nil)
    let worldCenter = SIMD3<Float>(worldCenterSCN)

    let simdExt = SIMD3<Float>(x: self.extent.x, y: self.extent.y, z: self.extent.z)

    ConnectivityManager.shared.sendCoverageUpdate(
        position: worldCenter,
        extent: simdExt,
        rays: self.cameraRaysAndHitLocations.map { ($0.ray.origin, $0.ray.direction) }
    )

    let timeElapsed = CFAbsoluteTimeGetCurrent() - startTime
    print("! Coverage update надіслано за \(timeElapsed) секунд")
}
```

Додаток 7

```
func processPointCloud(_ points: [[Float]]) {

    guard !points.isEmpty else {
        print("Empty point cloud")
        return
    }

    // Фільтруємо точки
    let minNeighborDistance: Float = 0.02
    let requiredNeighbors = 2
    var filteredPoints: [[Float]] = []

    for point in points {
        let x = point[0], y = point[1], z = point[2]

        var neighborCount = 0
        for otherPoint in points {
            let dx = otherPoint[0] - x
            let dy = otherPoint[1] - y
            let dz = otherPoint[2] - z
            let distance = sqrt(dx*dx + dy*dy + dz*dz)

            if distance < minNeighborDistance {
                neighborCount += 1
                if neighborCount >= requiredNeighbors {
                    filteredPoints.append(point)
                    break
                }
            }
        }
    }

    print("After filtration \(filteredPoints.count)")

    guard !filteredPoints.isEmpty else {
        print("All points are filtered")
        return
    }
```

```
    var minX = Float.greatestFiniteMagnitude, minY = Float.greatestFiniteMagnitude, minZ = Float.greatestFiniteMagnitude
    var maxX = -Float.greatestFiniteMagnitude, maxY = -Float.greatestFiniteMagnitude, maxZ = -Float.greatestFiniteMagnitude

    for point in filteredPoints {
        minX = min(minX, point[0])
        minY = min(minY, point[1])
        minZ = min(minZ, point[2])

        maxX = max(maxX, point[0])
        maxY = max(maxY, point[1])
        maxZ = max(maxZ, point[2])
    }

    let newCenter = [(minX + maxX) / 2, (minY + maxY) / 2, (minZ + maxZ) / 2]
    let newExtent = [maxX - minX, maxY - minY, maxZ - minZ]

    print("New bb:")
    print("Center : \(newCenter)")
    print("Extent : \(newExtent)")

    sendBoundingBoxUpdate(newCenter: newCenter, newExtent: newExtent)
}
```

Додаток 8

```
func processCoverage(
    position: SIMD3<Float>,
    extent: SIMD3<Float>,
    rays: [(origin: SIMD3<Float>, direction: SIMD3<Float>)]
) {
    print("Analyzing coverage on Mac")

    let resolution = 4
    let distanceThreshold: Float = 0.03

    var totalTiles = 0
    var hitTiles = 0

    for x in 0..<resolution {
        for y in 0..<resolution {
            for z in 0..<resolution {
                totalTiles += 1

                let local = SIMD3<Float>(
                    Float(x) / Float(resolution - 1) - 0.5,
                    Float(y) / Float(resolution - 1) - 0.5,
                    Float(z) / Float(resolution - 1) - 0.5
                )
                let worldPos = position + local * extent

                let isHit = rays.contains { ray in
                    let toTile = worldPos - ray.origin
                    let projectedLength = simd_dot(toTile, ray.direction)
                    let projectedPoint = ray.origin + ray.direction * projectedLength
                    let distance = simd_distance(projectedPoint, worldPos)
                    return distance < distanceThreshold
                }

                if isHit {
                    hitTiles += 1
                }
            }
        }
    }

    let coverage = Float(hitTiles) / Float(totalTiles)
    print("Coverage: \(Int(coverage * 100))%")

    if coverage >= 0.98 {
        sendFinishScanSignalToiPhone()
    }
}
```

Додаток 9

```swift
class ConnectivityBrowser: NSObject, MCNearbyServiceBrowserDelegate, MCSessionDelegate {

    private let serviceType = "ep123"
    private var peerID: MCPeerID!
    private var session: MCSession!
    private var browser: MCNearbyServiceBrowser!

    override init() {
        super.init()

        peerID = MCPeerID(displayName: "MyMacBook")

        session = MCSession(peer: peerID, securityIdentity: nil, encryptionPreference: .required)
        session.delegate = self

        browser = MCNearbyServiceBrowser(peer: peerID, serviceType: serviceType)
        browser.delegate = self
    }

    func startBrowsing() {
        browser.startBrowsingForPeers()
        print("Mac is looking for iPhone to connect...")
    }

    func stopBrowsing() {
        browser.stopBrowsingForPeers()
    }

    func browser(_ browser: MCNearbyServiceBrowser, foundPeer peerID: MCPeerID, withDiscoveryInfo info: [String : String]?) {
        print("Iphone found: \(peerID.displayName)")

        browser.invitePeer(peerID, to: session, withContext: nil, timeout: 10)
    }
```

```swift
class ConnectivityManager: NSObject, MCNearbyServiceAdvertiserDelegate, MCSessionDelegate {

    static let shared = ConnectivityManager()
    var boundingBox: BoundingBox?

    private let serviceType = "ep123"

    private var peerID: MCPeerID!
    private var session: MCSession!
    private var advertiser: MCNearbyServiceAdvertiser?
    private var isAdvertising = false
    private(set) var isConnectedToMac = false
    private var scanPercentage = 0

    override init() {
        super.init()
        peerID = MCPeerID(displayName:  UIDevice.current.name + "-" + UUID().uuidString.prefix(4))
        session = MCSession(peer: peerID, securityIdentity: nil, encryptionPreference: .required)
        session.delegate = self

        NotificationCenter.default.post(name: BoundingBox.scanPercentageChangedNotification,
                                        object: self,
                                        userInfo: [BoundingBox.scanPercentageKey: scanPercentage])
    }

    func startAdvertising() {
        guard !isAdvertising else {
            print("Already advertising.")
            return
        }
        advertiser = MCNearbyServiceAdvertiser(peer: peerID, discoveryInfo: nil, serviceType: serviceType)
        advertiser?.delegate = self
        advertiser?.startAdvertisingPeer()
        isAdvertising = true
        print("Advertising started on iPhone")
    }
```

Додаток 10

```
class SystemMonitor {
    private let disposeBag = DisposeBag()

    let cpuUsage: BehaviorSubject<Double>
    let ramUsage: BehaviorSubject<Double>
    let batteryLevel: BehaviorSubject<Double>


    init() {
        UIDevice.current.isBatteryMonitoringEnabled = true
        let level = UIDevice.current.batteryLevel
        let handleLevel = level >= 0 ? level : 1.0
        let currRam = SystemMonitor.getRAMUsage()
        let currCPU = SystemMonitor.getCPUUsage()

        ramUsage = BehaviorSubject<Double>(value: Double(currRam))
        cpuUsage = BehaviorSubject<Double>(value: Double(currCPU))
        batteryLevel = BehaviorSubject<Double>(value: Double(handleLevel))

        startMonitoring()
    }

    private func startMonitoring() {
        Observable<Int>.interval(.seconds(5), scheduler: MainScheduler.instance)
            .subscribe(onNext: { [weak self] _ in
                self?.cpuUsage.onNext(SystemMonitor.getCPUUsage())
                self?.ramUsage.onNext(SystemMonitor.getRAMUsage())
                self?.batteryLevel.onNext(self?.getBatteryLevel() ?? -1)
            })
            .disposed(by: disposeBag)
    }

}
```

Додаток 11

```swift
import Foundation
import RxSwift
import RxRelay
import os

enum ExecutionStrategy {
    case local
    case offloadToMac
}

class TaskManager: ObservableObject {
    let monitor = SystemMonitor()  // для тестування забрала private
    private let disposeBag = DisposeBag()

    static let shared = TaskManager()
    let strategy = BehaviorRelay<ExecutionStrategy>(value: .local)

    let cpuThresholds: [Double] = [20, 40, 60, 80]
    let ramThresholds: [Double] = [40, 70, 80]
    let batteryThresholds: [Double] = [20, 40, 60, 80]
    struct StrategyThresholds {
        var cpuThreshold: Double
        var ramThreshold: Double
        var batteryThreshold: Double
    }

    var thresholds: StrategyThresholds
    let log = Logger(subsystem: "com.esanapp", category: "performance")
```

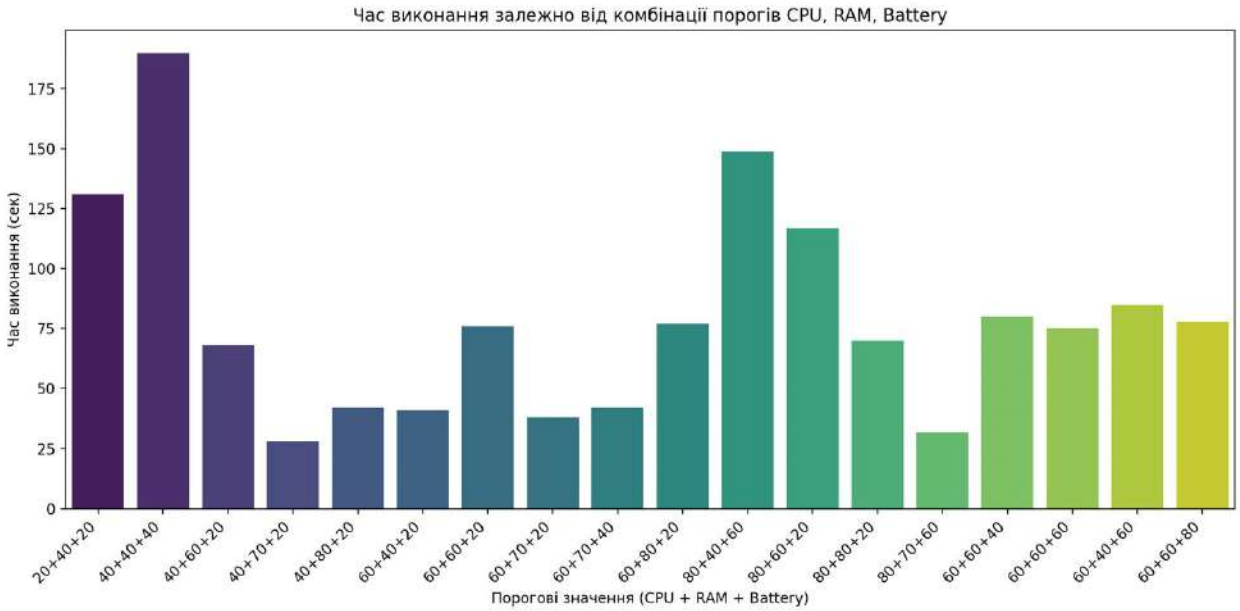
```swift
    init() {

        thresholds = StrategyThresholds(cpuThreshold: cpuThresholds[2], ramThreshold: ramThresholds[0], batteryThreshold: batteryThresholds[0])

        Observable.combineLatest(
            monitor.cpuUsage,
            monitor.ramUsage,
            monitor.batteryLevel
        )
        .throttle(.seconds(3), scheduler: MainScheduler.instance)
        .subscribe(onNext: { [weak self] cpu, ram, battery in
            self?.log.info("CPU: \(cpu)")
            self?.log.info("RAM: \(ram)")
            self?.log.info("Battery: \(battery)")
            let newStrategy = self?.calculateStrategy(cpu: cpu, ram: ram, battery: battery)
            if let newStrategy = newStrategy {
                self?.strategy.accept(newStrategy)
            }
        })
        .disposed(by: disposeBag)
    }

    private func calculateStrategy(cpu: Double, ram: Double, battery: Double) -> ExecutionStrategy {
        if cpu > thresholds.cpuThreshold || ram > thresholds.ramThreshold || battery < thresholds.batteryThreshold {
            if ConnectivityManager.shared.isConnectedToMac {
                log.info("Strategy changed to offload")
                return .offloadToMac
            } else {
                log.info("Strategy changed to local")
                return .local
            }
        } else {
            log.info("Strategy changed to local")
            return .local
        }
    }
}
```

Додаток 12


Час виконання залежно від комбінації порогів CPU, RAM, Battery
Час виконання (сек)
0
25
50
75
100
125
150
175
20+40+20
40+40+40
40+60+20
40+70+20
40+80+20
60+40+20
60+60+20
60+70+20
60+70+40
60+80+20
80+40+60
80+60+20
80+80+20
80+70+60
60+60+40
60+60+60
60+40+60
60+60+80
Порогові значення (CPU + RAM + Battery)

Додаток 13

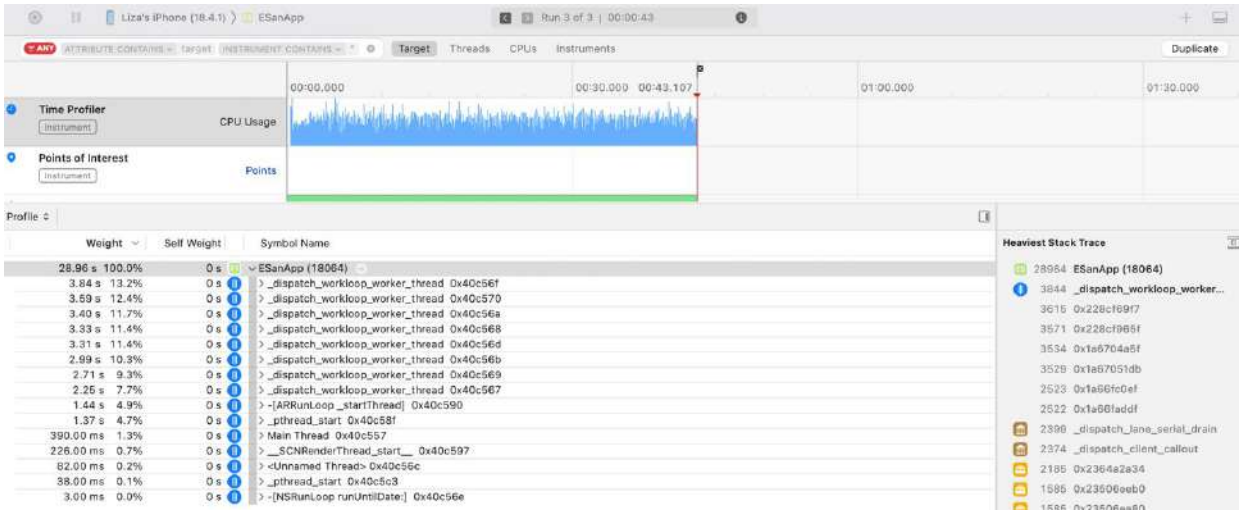


Рис. 1 Профілювання виконання сканування з використанням стратегії

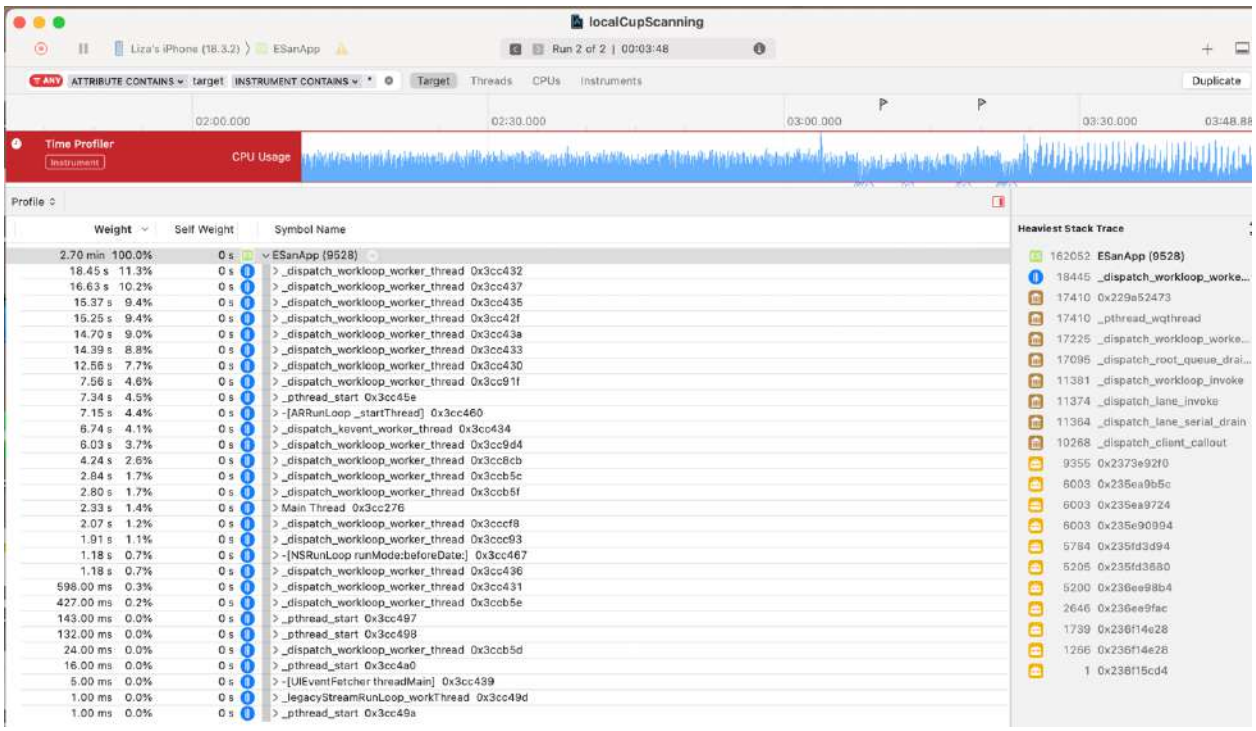


Рис. 2 Профілювання виконання сканування локально