УДК 004.7 **doi: 10.26906/SUNZ.2025.2.161-168**

С. В. Носко, С. С. Бульба, О. П. Чорних, В. В. Скородєлов, В. І. Панченко

Національний технічний університет «Харківський політехнічний інститут», Харків, Україна

СПОСІБ ВИСОКОЕФЕКТИВНОЇ РЕАЛІЗАЦІЇ САЙДКАР КОМПОНЕНТА З МІНІМАЛЬНИМИ ВИТРАТАМИ СИСТЕМНИХ РЕСУРСІВ

Анотація. У статті представлено спосіб щодо реалізації високоефективного сайдкара з мінімальними витратами системних ресурсів у контексті мікросервісної архітектури. Проведено аналіз існуючих рішень для реалізації сайдкару, а також виконано порівняння різних технологій з точки зору використання оперативної пам’яті та часу на запуск, що є критично важливими аспектами для сайдкар-компонентів. На основі отриманих результатів аналізу обрано Quarkus у зв’язку з тим, що його можливості компіляції ahead-of-time (AOT) на базі GraalVM забезпечують високу швидкість запуску та низьке споживання об’ємів пам’яті та центрального процесору (CPU). Для вирішення завдань із витоками пам’яті (memory leaks) запропоновано проведення оптимізації управління потоками даних, зокрема, шляхом усунення блокуючих викликів та використання асинхронних підходів. Заміна високорівневого WebClient на низькорівневий HttpClient для перенаправлення HTTP-запитів дозволило істотно зменшити використання оперативної пам'яті та уникнути помилок OutOfMemory. Вибір оптимального Garbage Collector у JVM середовищі забезпечує високу стійкість до навантажень. Розроблений та запропонований спосіб дозволяє мінімізувати додаткові витрати на інфраструктуру, зберегти переваги сайдкар-архітектури – незалежність основних мікросервісів, простоту масштабування та високу ефективність. Розроблений сайдкар стійкий до навантажень та забезпечує мінімальне використання ресурсів. Він може ефективно інтегруватися у сучасні мікросервісні системи.

Ключові слова: сайдкар, мікросервісна архітектура, оптимізація оперативної пам’яті, витік пам’яті, асинхронне програмування, мінімальне використання ресурсів, потокова передача даних, масштабованість, збірник сміття.

Вступ

Постановка проблеми. У мікросервісних системах, де використовується сайдкар-компонент, він працює поруч із основним мікросервісом та бере на себе виконання додаткових завдань, таких як: моніторинг, управління конфігурацією, забезпечення безпеки, відновлення після збоїв, комунікація з іншими компонентами системи тощо. Однак, для його розгортання та функціонування потрібні додаткові ресурси, включаючи процесорний час та оперативну пам’ять (ОП, ОЗП). У випадку мікросервісної системи, яка складається зі 100 мікросервісів, необхідно додатково розгорнути 100 сайдкарів. Це створює виклик для системи, оскільки додаткові компоненти не лише збільшують навантаження на інфраструктуру, але й можуть негативно вплинути на економічну доцільність рішення.

Проблема полягає у тому, що без ефективної реалізації сайдкар може стати значним споживачем системних ресурсів, що ускладнює масштабованість та стабільність мікросервісної архітектури. Високий рівень споживання ОЗП, зокрема через потенційні витоки пам'яті та неконтрольоване створення короткоживучих потоків, може призвести до зниження продуктивності системи та виникнення помилок OutOfMemory (OOM) при обробці великих даних (високих навантаженнях).

Ще одним викликом є необхідність у інтеграції додаткового компонента, який не повинен збільшувати складність або вартості інфраструктури, але, водночас, забезпечувати максимальну функціональність. На приклад, передача HTTP-запитів через сайдкар потребує низького використання пам’яті, швидкого виконання та мінімізації витрат на серіалізацію та десеріалізацію даних.

Таким чином, актуальним науковим завданням є розробка сайдкару, який буде відповідати наступним характеристикам: мати оптимальні параметрами для роботи в умовах високого навантаження; мати мінімальне споживання ОЗП та центрального процесору (CPU); бути здатним до потокової передачі даних для усунення витрат на створення проміжних буферів; бути безпечним з точки зору управління потоками та повністю позбавленим від витоків пам’яті.

Аналіз останніх досліджень і публікацій. Сайдкар патерн широко використовується у мікросервісній архітектурі.

У [1] детально описані області застосування сайдкару, його переваги і особливості, а також процес розгортання. Особлива увага приділяється високорівневим аспектам та опису мікросервісних систем на вищому рівні абстракції.

У [2] описана доцільність реалізації авторизації у сайдкарі, розглянуто популярний сервер аутентифікації і авторизації Keycloak, а також описані архітектурні діаграми взаємодії з API Gateway Kong. Розкрито проблему частого виклику Keycloak, що може стати вузьким місцем в архітектурі. Наведено ефективні і високопродуктивні рішення розв’язання проблеми з використанням Caffeine кешу.

У [3] акцентовано увагу на продуктивність сайдкарів у середовищах з високим навантаженням. Підкреслюється вплив на витік пом’яті та тривалі GC (Garbage Collection) паузи на стабільність системи. Вказується необхідність у використанні сучасних збірників сміття таких, як: ZGC та G1 GC, які забезпечують короткі паузи та ефективне очищення пам'яті.

У [4] проведено аналіз підходів до запуску сайдкарів та їх інтеграція у контейнеризованих середовищах, таких як Kubernetes.

Документація Quarkus [5] надає інструкції до розгортання швидкісних застосунків на основі цієї платформи.

Крім фреймворку Quarkus розглядалися й інші альтернативи, на приклад, Spring Boot [6]. Фреймворк демонструє значно нижчі показники ефективності у продуктивності та споживанню системних ресурсів, що є визначальним фактором при виборі технології для ефективної реалізації зазначеного функціоналу. Тому, для подальшого аналізу розглядаються модифікації стандартного Spring Boot, зокрема Spring Boot Native Image, який оптимізує продуктивність та ефективність використання ресурсів через компіляцію у нативний код.

У [7] досліджують Envoy, як високопродуктивний proxy сервер, який використовується у багатьох мікросервісних рішеннях та часто виступає як стандарт де-факто для сайдкар компонентів. Основним недоліком є його складність у налаштуванні та потенційно високе споживання ресурсів.

У [8] розглядається Micronaut, який подібно до Quarkus забезпечує швидку відповідь через зниження часу на старт. Micronaut використовує Ahead-of-Time (AOT) компіляцію для вирішення залежностей та конфігурацій, що дозволяє уникнути використання рефлексії та проксі, які є основними концепціями у Spring. Основними недоліками є необхідність вивчення нового API та архітектурних підходів, зокрема, після роботи зі Spring, а також менша спільнота, що може вплинути на підтримку у майбутньому.

Мета статті – розробка способу для оптимізаціЇ ефективного сайдкар-компонента до мікросервісної архітектури, який забезпечить мінімальне споживання системних ресурсів, стійкість до високих навантажень та економічну доцільність.

Виклад основного матеріалу

За головну мету реалізації сайдкара у рамках роботи є забезпечення його високої продуктивності та економічної ефективності. Для досягнення мети використано фреймворк Quarkus у режимі native з функцією GraalVM для випереджувальної компіляції AOT, яка безпосередньо компілює код у нативний. Це дозволяє забезпечити надзвичайно швидкий час на запуск – лише кілька мілісекунд.

На відміну від традиційних Java-додатків, які компілюються у байткод Java і потім інтерпретуються або компілюються під час виконання віртуальною машиною Java (JVM), AOT-компіляція переводить додаток у нативний машинний код ще на етапі збірки. Це означає, що сайдкар не залежить від JVM для запуску та не страждає від типової затримки запуску JVM, пов’язаної з оптимізацією коду JIT-компілятором під час виконання. Натомість він стартує вже оптимізованим, забезпечуючи значно швидший час на запуск.

У режимі native додатки Quarkus компілюються у нативний виконуваний файл. Такий файл може взаємодіяти з операційною системою напряму, без посередництва JVM. У результаті, додаток починає виконання одразу після запуску, що забезпечує практично миттєвий старт.

На офіційному сайті GraalVM [9] надана діаграма, яка порівнює час на запуск додатків, написаних різними мовами програмування (рис. 1).

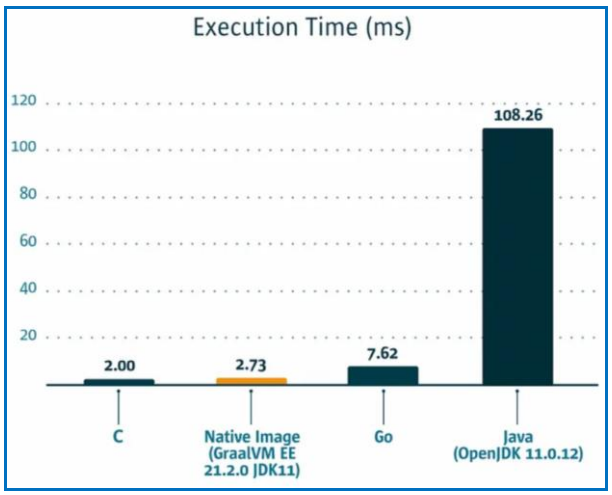

Рис. 1. Використання ОЗП різними платформами

За результатами порівняння, використання C та GraalVM показують ідентичні показники продуктивності. Однак, використання GraalVM у поєднанні з Quarkus дозволяє використовувати усі переваги високорівневої мови Java, зокрема, широкий спектр абстракцій у порівнянні з кодом на C, забезпечуючи аналогічну продуктивність до коду на C.

Для об'єктивної оцінки зазначених результатів проведено експериментальні запуски локального REST-застосунку, реалізованого за допомогою Quarkus у двох режимах роботи. Під час запуску застосунку у режимі JVM за допомогою Quarkus час на старт склав 1,07 с (рис. 2). У нативному режимі з використанням GraalVM час на запуск значно знизився до 13 мс (рис. 3), що демонструє вражаючу ефективність.

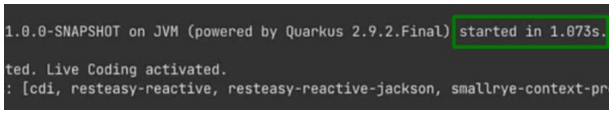

Рис. 2. Запуск Quarkus локально у JVM режимі

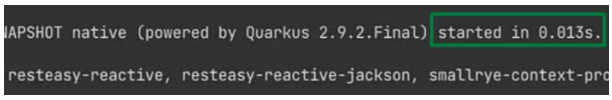

Рис. 3. Запуск Quarkus у нативному режимі

Таким чином, комбінація Quarkus та GraalVM з написанням коду на Java є оптимальним вибором, який використаний як основа для розробки високопродуктивного сайдкара.

Quarkus пропонує різноманітні внутрішні оптимізації для зменшення споживання ОЗП, що робить його дуже економічно вигідним рішенням. Серед таких оптимізацій можливо виділити такі:

– дотримання мінімальної моделі часу на виконання. Тобто, у ОЗП завантажуються компоненти, необхідні для виконання програми, що дозволяє уникнути накладних витрат на завантаження та підтримку непотрібних ресурсів у ОЗП, зменшуючи загальне споживання;

– використання стратегії вибіркового завантаження класів, де у ОЗП завантажуються лише кла-

си, які фактично використовуються програмою, що дозволяє уникнути завантаження невикористаних бібліотек та класів, що призводить до зменшення об'єм ОЗП;

– Quarkus розроблено таким чином, щоб займати мінімальний об'єм ОЗП. Він використовує такі методи оптимізації, як: стиснення метаданих JVM та видалення непотрібної інформації про налагодження, зменшуючи споживання ОЗП, необхідної для запуску програми.

На рис. 4 представлено діаграму споживання пам'яті для Quarkus, яка зображена на офіційному веб-сайті [5]. Діаграма ілюструє порівняння споживання ОЗП між різними технологічними рішеннями для розробки REST застосунків. Значна різниця у споживанні ОЗП спостерігається при використанні Quarkus у native режимі з GraalVM порівняно з традиційними підходами, що вказує на високу ефективність Quarkus для оптимізації ресурсів ОЗП.

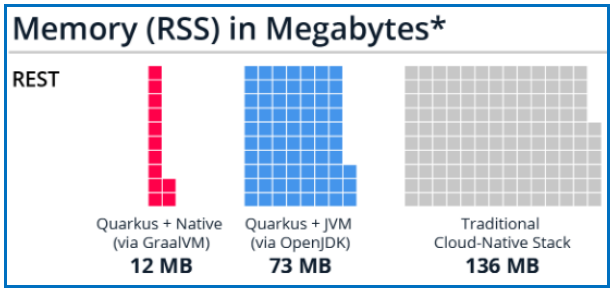

Рис. 4. Порівняння використання ОЗП у різних рішеннях

Реактивний підхід, який впроваджений у Quarkus з використанням підсистеми Vert.x, надає значні переваги для розробки легкого та високо оптимізованого сайдкара у мікросервісній архітектурі.

Сутність реактивного програмування полягає у асинхронному та неблокуючому стилі виконання, що забезпечує більш ефективне використання ресурсів порівняно з традиційним блокуючим підходом. Традиційний блокуючий підхід зображено на рис. 5.

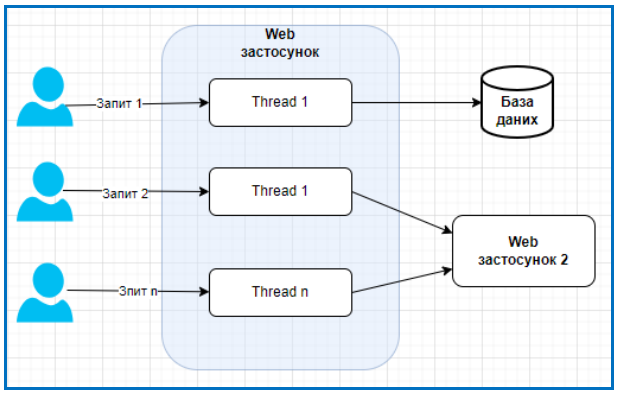

Рис. 5. Традиційний блокуючий підхід

Коли надходить запит, веб-сервер призначає створює новий потік, який буде зайнятий цим запитом, доки він не завершить процес. Процес може викликати базу даних або сторонній API, що потребує часу. За умови, якщо надходить інший запит, то він буде призначений іншому доступному потоку, який також буде зайнятий, доки процес не завершиться. Слід зазначити, що для певного веб-сервера існує максимальна кількість потоків, які він може обробити одночасно.

На приклад, при використанні Spring Boot за замовчуванням Tomcat контейнер використовує пул на 200 потоків, розмір пулу обмежує паралельність програми. Недоліками такого підходу є те, що за умови, якщо запиту призначено свій потік, то він буде недоступним, поки запит не завершить обробку. За умови, якщо усі наявні потоки зайняті, то наступні запити, які надходять на сервер, будуть чекати поки не звільниться хоча б один потік. Крім того, кожен потік має свою вартість, використовуючи ОЗП та CPU. Великі пули потоків призводять до не легковісних застосунків, що не підходить для реалізації високоефективного та швидкого сайдкара.

Для вирішення висвітленого завдання використовується неблокуючий асинхронний підхід, де компоненти системи реагують на події замість очікування результатів операцій, що знижує потребу у блокуванні ресурсів. На рис. 6 продемонстровано один і той самий потік (Thread), який переключається на виконання Запиту 1, Запиту 2 та Запиту *n*. Отже, потік завжди у роботі, а ресурси використовуються найбільш ефективно без простоїв.

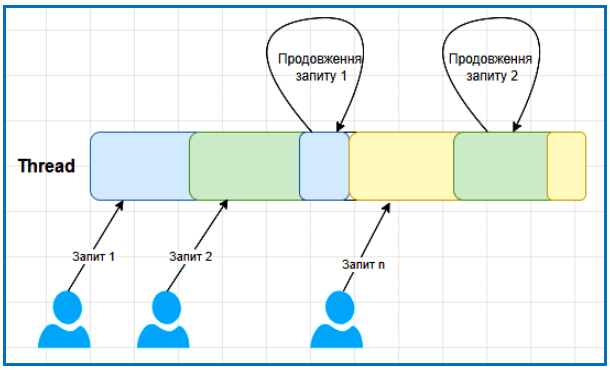

Рис. 6. Робота окремого потоку у асинхронному підході

Додатково, реалізація сайдкара з використанням фреймворку Quarkus застосовує підхід реактивного програмування, який дозволяє ефективно керувати та використовувати ресурси, на відміну від традиційного підходу із блокуючими потоками.

Реактивне програмування – це парадигма, яка спрямована на створення неблокуючих асинхронних додатків, що здатні ефективно обробляти велику кількість паралельних потоків даних. Такий реактивний підхід сприяє кращій ідентифікації та використанню усіх виділених ресурсів, забезпечуючи максимальну ефективність.

Основа Quarkus-Vert.x – це реактивний інструментарій, який дозволяє створювати легкі і швидкодіючі системи та може обробляти великі об'єми трафіку з меншою кількістю потоків ОС, що забезпечує вищий паралелізм, а навантаження на CPU і ОЗП знижується. Сайдкар, реалізований на базі Quarkus та Vert.x з транспортом Netty, не буде вимагати інтенсивного використання ресурсів для виконання своїх задач, дозволяючи головному мікросервісу

ефективно обробляти бізнес-логіку, що є особливо важливим для реалізації ефективного сайдкара.

Використання неблокуючого вводу-виводу має значні переваги, але воно не є безкоштовним. Неблокуючий ввід-вивід вводить альтернативну модель виконання, яка радикально відрізняється від традиційної блокувальної моделі у класичних фреймворків. Однак, реалізація у Quarkus зменшує таку складність, оскільки фреймворк абстрагує базовий рівень управління неблокуючими операціями за допомогою вже розробленого набору інструментів та API. Отже, розробникам не потрібно докладати додаткових зусиль для впровадження цієї моделі виконання, що сприяє більш швидкій адаптації та імплементації реактивних систем на основі Quarkus.

Проведено моніторинг роботи сайдкара під навантаженням. Для вимірювання споживання ресурсів та інших відповідних показників за різних умов/конфігурацій використано інструмент JDK Mission Control (JMC) [10]. Метою дослідження є ефективний розподіл часу та зусиль щодо оптимізації витрат та продуктивності. З метою об'єктивного порівняння роботи з GC [4], виявлення потенційних витоків пам'яті (memory leak) та оптимізації використання ОЗП, – сайдкар розгорнуто у звичайному JVM режимі замість нативного з використанням GraalVM за такими основними причинами:

– у режимі JVM надається доступ до розширених інструментів Java для профілювання, аналізу ОЗП та моніторингу таких, як: JMC, VisualVM для аналізу споживання ресурсів та ОЗП, зокрема. Така стратегія дозволяє знаходити потенційні витоки ОЗП. Нативні образи на основі GraalVM не підтримують традиційні JVM профайлерні інструменти. Оскільки GraalVM у native режимі компілює код до машинного, робота типових JVM-засобів діагностики є обмеженою;

– у JVM режимі можливо переключатися між різними збірниками сміття (Garbage Collectors) – на приклад, G1 GC, Parallel GC, ZGC або Shenandoah GC – використовуючи відповідні опції JVM (на приклад, -XX:+UseG1GC або -XX:+UseZGC). У GraalVM native mode така можливість значно обмежена, оскільки GraalVM native образи включають лише базові GC механізми, такі як SubstrateVM garbage collector, які є більш простими та менш настроюваними.

Тестування у нативному GraalVM режимі актуальне для оцінки продуктивності та часу на запуск фінальної версії високооптимізованого сайдкара, який буде розгорнуто у production середовищі.

У дослідженні протестовано три різні типи збірників сміття для сайдкарів модулів системи: Serial GC, G1 GC та ZGC.

Метою експерименту є визначення ефективності роботи збірників сміття в умовах різних навантажень, а також – оцінки їх впливу на продуктивність та використання системних ресурсів, де вимоги до швидкодії та масштабованості постійно зростають.

Опис параметрів для оцінки наведено нижче, а детальне порівняння GC наведено у табл. 1:

– найдовша пауза – вказує на максимальну затримку, викликану збором сміття за усі спостереження. Менше – краще, оскільки це знижує ризик "фризів" додатка;

– більшість пауз – вказує типові (середні або найчастіші) затримки. Менше значення означає більш стабільну продуктивність;

– сума пауз за певний період – узагальнює загальний час, який витрачений на збір сміття за визначений проміжок часу (5 хв або 10 хв). Менші значення дозволяють приділяти більше часу на корисне виконання;

– середнє використання CPU – відображає ресурси процесора, які GC використовує, що впливає на загальну продуктивність системи.

Таблиця 1 – **Тест №1 для визначення оптимального GC**

Garbage Collector	Найдовша пауза	Більшість пауз	Сума пауз за певний період	Середнє використання CPU	Особливості
Serial GC	59 – 89 мс	10 – 40 мс	300 мс–3 с (на 5 хв)	3 – 5%	1. Не підходить для високих навантажень. 2. Тривалі паузи, особливо у модулях з великим memory heap. 3. Високі затримки.
G1 GC	7,5 – 45 мс	4 – 10 мс	0,1 с – 4 с (на 10 хв)	7 – 15%	1. Баланс між продуктивністю і ресурсами. 2. Помірні паузи навіть за високого навантаження. 3. Добре працює з обмеженим memory heap (64 – 160 МБ).
ZGC	78 мкс–0,5 мс (мікро-с.!)	40 – 100 мкс	2 – 100 мс (на 5 – 10 хв)	20 – 30%	1. Найшвидший збірник завдяки мінімальним паузам. 2. Ідеальний для низьколатентних систем. 3. Високе використання CPU може впливати на масштабованість.

За результатами аналізу табл. 1 можливо стверджувати, що Serial GC підходить лише для систем з малим хіпом та низькими вимогами до продуктивності. G1 GC – найкращий варіант для більшості систем, завдяки збалансованому споживанню ресурсів та невеликим затримкам. ZGC підходить для додатків із критично важливими вимогами до низької затримки, але варто враховувати підвищене споживання CPU.

Наступним етапом дослідження виконувались бізнес запити, які емулюють процес здійсення покупок у великонавантеженій системі протягом 6-х годин із 100 запитами за секунду, а також проведено аналіз поведінки сайдкара під таким навантаженням. Сайдкари продемонстрували постійно зростаючий об'єм ОЗП heap, що використовується Java-процесом під час усіх запусків, що вказує на потенційний витік пам'яті рис. 7.

Зафіксовано створення великої кількості короткоживучих потоків для екземплярів сайдкарів, що показано на рис. 8.

На рис. 9 спостерігається те, що майже уся ОЗП використовувалася об’єктами ThreadLocalMap.

За результатами проведеного аналізу виявлено, що сайдкар використовує надмірну кількість викликів блокуючого коду для доступу до AWS SSM, використовуючи метод vertx.executeBlocking.

Даний метод створює додаткові worker потоки, що належать до пулу потоків Vert.x. Такі виклики створюються за короткий проміжок часу.

Дана обставина пояснює створення нових потоків, які мають короткий термін існування так часто, що не встигають звільнятися так швидко.

Як рішення – усунуто надмірний виклик методу vertx.executeBlocking завдяки використанню кешування.

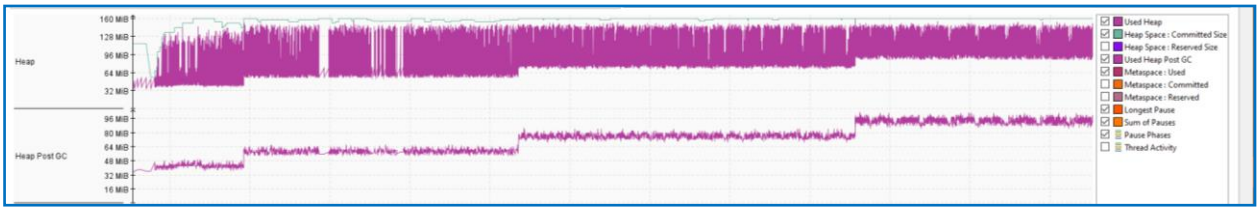

Рис. 7. Графік збільшення кількості одночасних запитів під час тесту

Рис. 8. Велика кількість короткоживучих потоків

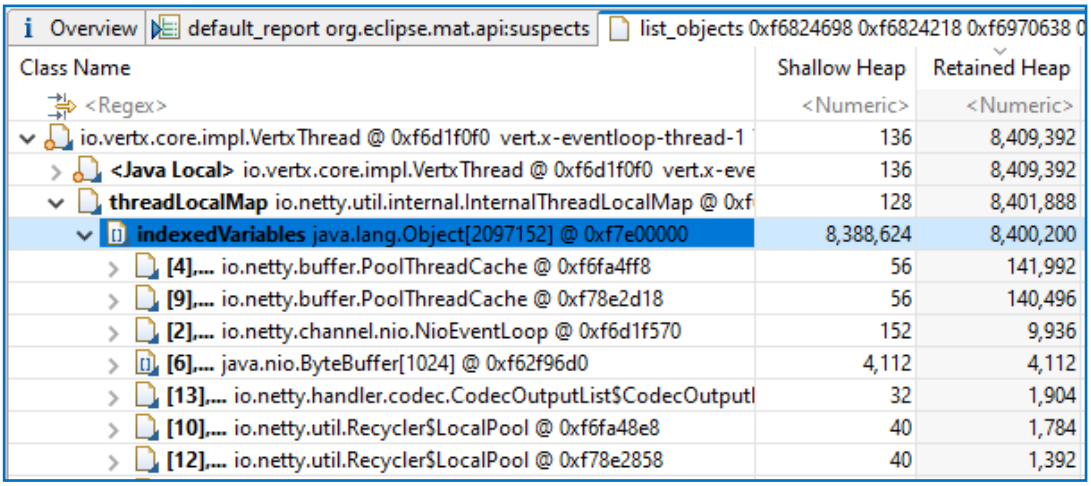

Рис. 9. Велика кількість короткоживучих потоків

Наступною перевіркою вископродуктивної реалізації сайдкара є інтенсивна кількість операцій читання та запису даних.

Використано csv файл на 500 000 записів. У бізнес процесі імпорту такого великого файлу мікросервіси під час обробки частин файлу використовують велику кількість операцій серіалізації та десеріалізації json об’єктів, які передаються через мережу.

Початкова реалізація сайдкару використовувала високорівневу реалізацію Vert.x WebClient для

усіх вхідних та вихідних викликів. До деяких переваг використання даної реалізації для комунікації між різними компонентами системи можливо віднести наступні:

1. Неблокуючий вхід-вихід. Оскільки WebClient використовує асинхронні неблокуючі I/O операції, він дозволяє сайдкару обробляти велику кількість запитів одночасно без створення вузьких місць або потреби у достатній кількості потоків для кожного запиту. Це підвищує продуктивність та зменшує загальне навантаження на систему.

2. Швидкість та ефективність реакції. Асинхронна природа WebClient, також, дозволяє сайдкару швидко реагувати на вхідні запити без затримок, призначених для обробки інших операцій.

3. Толерантність до збоїв. Використання асинхронних операцій дозволяє краще управляти помилками та збоями. WebClient може надавати механізми для повторних спроб, таймаутів та другорядних обробників помилок, що допомагає забезпечити стійкість сайдкара до збоїв у зовнішніх сервісах або мережі.

При обробці великих даних, таких як імпорт великих файлів, виникали помилки Out Of Memory (OOM). Це стало поштовхом для перегляду підходу до реалізації вихідних HTTP-запитів з використанням Vert.x WebClient та внесення оптимізацій. Vert.x WebClient забезпечує можливість сереалізації та десереалізації об'єктів, що є зручним для більшості REST-інтеграцій.

Однак, у випадку сайдкара такі абстракції зайві, оскільки основна функція сайдкара полягає у перенаправленні HTTP-запитів без будь-якої бізнес-логіки або додаткової обробки даних.

Проблема виникла через те, що у сценаріях із великими файлами Vert.x WebClient мав більш високі вимоги до ОЗП через створення буферів даних та проміжне зберігання. Це призводило до витрат ОЗП, що були неприпустимими у високонавантаженому середовищі.

З метою зменшення споживання ОЗП та забезпечення потокової передачі даних, Vert.x WebClient частково замінено на Vert.x HttpClient. HttpClient є низькорівневим інструментом, який дозволяє напряму управляти потоками даних між серверними вхідними (server requests) та клієнтськими вихідними запитами (client requests). Зміна такого підходу базується на наступних критеріях:

1. Потреба у потоковій передачі даних (streaming): HttpClient дозволяє передавати дані напряму між запитами, оминаючи створення проміжних буферів та збільшує використання ОЗП.

2. Мінімізація витрат ресурсів: Vert.xHttp Client краще підходить для сценаріїв, де не потрібно використовувати функціонал сереалізації/десереалізації.

3. Жорстке управління ОЗП: HttpClient дозволяє уникнути витрат пам’яті на проміжну обробку, яка необхідна для WebClient.

Vert.x WebClient використовується для деяких REST-запитів таких, як: аутентифікація через Keycloak, отримання різних даних про конфігурації тощо, оскільки у даних випадках сайдкар не виступає як Proxy.

Після повторного тестування витоку пам’яті не виявлено, а збирання сміття (garbage collection) рівномірно очищає ОЗП, що представлено на рис. 10.

На рис. 11 наведено навантаження на CPU сайдкара у відсотках при 100 паралельних викликах.

Обрані технології, застосовані підходи, а також проведений аналіз продуктивності та оптимізації дозволили створити високоефективну реалізацію сайдкара з мінімальними витратами системних ресурсів.

Висновки

Таким чином, розроблено спосіб високоефективної реалізації сайдкар компонента з мінімальними витратами системних ресурсів. Оптимізовано сайдкар, який спрямований на забезпечення мінімального використання системних ресурсів.

Проведено детальний аналіз, усунення потенційних проблем з ресурсозатратністю, а також впровадження оптимізацій, які знижують навантаження на інфраструктуру, залишаючи усі переваги сайдкар-архітектури.

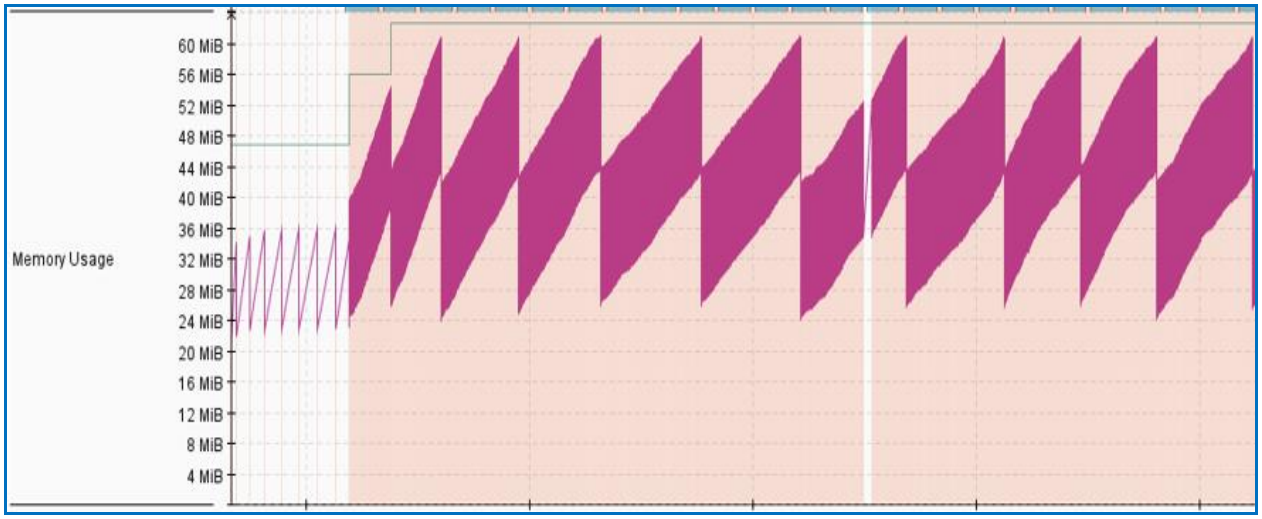

Рис. 10. Витоки пам’яті (memory leaks) не виявлено

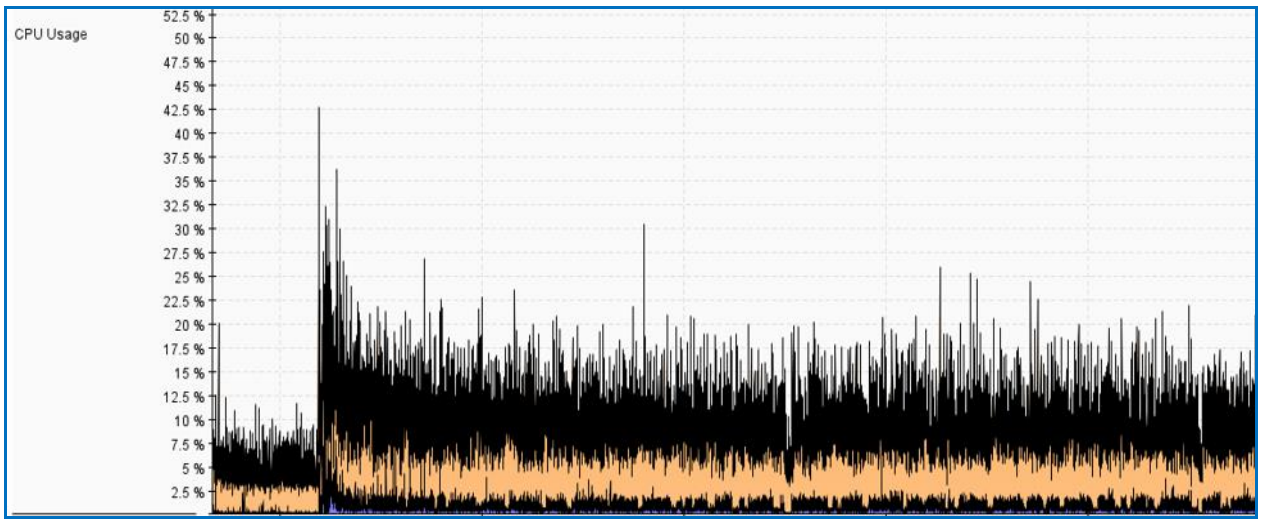

Рис. 11. CPU навантаження при 100 паралельних викликах до сайдкару

Результати показали, що завдяки обраним підходам додавання сайдкару як окремого компонента майже не вплинуло на вартість інфраструктури, але забезпечило значний функціональний та технологічний виграши для системи.

На етапах розробки способу виявлені проблеми, які пов’язані із витоком пам'яті (memory leak) у сайдкарі. Проведено низку експериментів із використанням інструментів профілювання JVM, що підтвердило накопичення об’єктів у ОЗП, зокрема, у ThreadLocalMap та звернуло увагу на необхідність оптимізації управління потоками даних та використання ОЗП. Використання блокуючих операцій, таких як vertx.executeBlocking, у критичних місцях коду замінено на більш ефективні асинхронні виклики, що зменшило кількість короткоживучих потоків та оптимізувало використання ресурсів.

У процесі профілювання виявлено помилки OOM під час обробки файлів за великим об’ємом. Подальший аналіз показав, що використання високорівневого WebClient, який потребує проміжних буферів та підтримує серіалізацію/десеріалізацію даних, суттєво впливало на ресурсозатратність сайдкару. Для усунення виявленої проблеми замінено WebClient на більш низькорівневий HttpClient для усіх вихідних запитів, які не потребують додаткової обробки даних. Такий підхід реалізує повноцінний стримінг даних між сервером та клієнтом без утворення зайвих копій у ОЗП, що суттєво зменшує споживання пам'яті й уникнення помилок OOM.

Попередньо обрані технології, зокрема, використання фреймворку Quarkus у нативному режимі із застосуванням AOT compilation на основі GraalVM, забезпечили швидкий старт сайдкара та мінімізували використання ресурсів CPU та ОЗП.

За результатами проведеного тестування різних збірників сміття таких, як: Serial GC, G1 GC та ZGC, обрані оптимальні параметри роботи JVM для різних сценаріїв навантаження. Після усунення memory leaks збірник сміття стабільно очищав ОЗП, не створюючи тривалих пауз навіть за умов роботи із великим навантаженням.

Запропонований спосіб, як сукупність проведених оптимізацій продемонстрував, що сучасні підходи до реалізації сайдкара дозволяють значно зменшити потребу у ресурсах інфраструктури. Обрані технології, інструменти та методики забезпечують створення високоефективного, стійкого до навантажень сайдкара, який демонструє мінімальні витрати ресурсів та позитивно впливає на роботу системи.

Таким чином, новий спосіб дозволяє здійснити оптимізацію управління ОЗП та потоками даних, усуває блокуючі виклики, використовує більш низькорівневі інструменти для обробки HTTP-запитів, а також впроваджує потоковий механізм передачі даних.

Проведене дослідження способу демонструє, що за умови правильної реалізації сайдкар додатково покращує функціональність мікросервісної системи, майже не додаючи вартості її інфраструктурі.

СПИСОК ЛІТЕРАТУРИ

1. . С. С. Бульба, О. В. Коломійцев, О. І. Соловйова, С. В. Носко. Засоби побудови додаткового рівня системи комунікацій у мікро-сервісній архітектурі. Грааль науки : міжнар. наук. журнал. – Вінниця : ГО «Європейська наукова платформа», 2024. – No 46. – С. 651-659. – URL: DOI 10.36074/grail-of-science.29.11.2024.084.
2. Носко С. В., Бульба С. С., Коломійцев О. В., Лисиця Д. О., Молчанов Г. І. Пропозиції щодо авторизації в сайдкар компоненті мікросервісної архітектури. *Системи управління, навігації та зв’язку*. Полтава: НУ «ПП», 2025. № 1(7). С. 116–123. – URL: https://journals.nupp.edu.ua/sunz/issue/view/127/68.
3. Meadows, C., Hounsinou, S., Wood, T., & Bloom, G. (2023). Sidecar-based Path-aware Security for Microservices. Proceedings of the 28th ACM Symposium on Access Control Models and Technologies (SACMAT '23), P. 157–162. https://doi.org/10.1145/3589608.3594742.
4. Araldo, A., Di Stefano, A., & Di Stefano, A. (2020). Resource allocation for edge computing with multiple tenant configurations. *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20)*, P. 1190–1199. https://doi.org/10.1145/3341105.3374026

5. Quarkus: офіційний веб-сайт. URL: https://quarkus.io/.
6. Parola, F., Qi, S., Narappa, A. B., Ramakrishnan, K. K., & Risso, F. (2024). SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient. *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*, 668–688. https://doi.org/10.1145/3698038.3698558
7. Poudel, A., Niroula, P., MacDonald, C., Gloudemans, L., & Herwig, S. (2025). Mazu: A Zero Trust Architecture for Service Mesh Control Planes. *Proceedings of the 18th European Workshop on Systems Security (EuroSec '25)*, 49–55. https://doi.org/10.1145/3722041.3723100
8. Basso, M., Prokopec, A., Rosà, A., & Binder, W. (2025). Improving Native-Image Startup Performance. *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*, 689–703. https://doi.org/10.1145/3696443.3708927
9. Graalvm: офіційний веб-сайт. URL: https://www.graalvm.org/native-image/
10. JDK Mission Control: веб-сайт. URL: https://www.oracle.com/java/technologies/jdk-mission-control.html

Надійшла до редколегії 01.04.2025
Схвалена до друку 11.06.2025

ВІДОМОСТІ ПРО АВТОРІВ/ ABOUT THE AUTHORS

Носко Сергій Вікторович – аспірант кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Serhii Nosko – Postgraduate of the Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: Serhii .Nosko@khpi.edu.ua; ORCID Author ID: https://orcid.org/0009-0009-5398-8650.

Бульба Сергій Сергійович – кандидат технічних наук, доцент, доцент кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Serhii Bulba – Candidate of Technical Sciences, Associate Professor, Associate Professor of the Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: bssserega@gmail.com; ORCID Author ID: https://orcid.org/0000-0003-0358-7516;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=57216350034.

Черних Олена Петрівна – кандидат фізико-математичних наук, доцент, професор кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Olena Chernykh – Candidate of Physical and Mathematical Sciences, Associate Professor, Professor of the Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: chernikhlena@ukr.net, Olena.Chernykh@khpi.edu.ua; ORCID Author ID: https://orcid.org/0000-0002-4389-2446;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=42461252100.

Скородєлов Володимир Васильович – кандидат технічних наук, доцент, професор кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Volodymyr Skorodielov – Candidate of Technical Sciences, Associate Professor, Professor of the Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: Volodymyr.Skorodielov@khpi.edu.ua; ORCID Author ID: https://orcid.org/0009-0008-1728-9089.

Панченко Володимир Іванович – старший викладач кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Volodymyr Skorodielov – Senior Lecturer of the Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: Volodymyr.Panchenko@khpi.edu.ua; ORCID Author ID: https://orcid.org/0000-0003-3364-3398.

Method of the high-efficiency implementation of a sidecar component with minimal system resource consumption

Serhii Nosko, Serhii Bulba, Oleksii Kolomiitsev, Olena Chernykh, Volodymyr Panchenko

Abstract. An analysis of existing solutions for implementing a sidekick is carried out, and a comparison of different technologies is made in terms of RAM usage and startup time, which are critical aspects for sidekick components. Based on the results of the analysis, Quarkus was chosen because its GraalVM-based advance-of-time (AOT) compilation capabilities provide high startup speed and low consumption of RAM and central processing unit (CPU). To solve problems with memory leaks, it is proposed to optimize data flow management, in particular, by eliminating blocking calls and using asynchronous approaches. Replacing the high-level WebClient with the low-level HttpClient for redirecting HTTP requests significantly reduced the use of RAM and avoided OutOfMemory errors. Choosing the optimal Garbage Collector in the JVM environment provides high load resistance. Elimination of memory leaks allows the garbage collector to steadily clear RAM without creating long pauses even under heavy load. The developed and proposed method, as a set of optimizations, demonstrated that modern approaches to the implementation of sidekick can significantly reduce the need for infrastructure resources. The selected technologies, tools, and techniques ensure the creation of a highly efficient, load-resistant sidekick that demonstrates minimal resource consumption and has a positive impact on system performance. The new method allows optimizing the management of RAM and data flows, eliminates blocking calls, uses lower-level tools for processing HTTP requests, and implements a streaming data transfer mechanism. The conducted research of the method demonstrates that, if implemented correctly, the sidekick allows minimizing additional infrastructure costs, preserving the advantages of the sidekick architecture - independence of the main microservices, ease of scaling, and high efficiency. The developed sidekick is resistant to loads and ensures minimal resource utilization. It can be effectively integrated into modern microservice systems.

Keywords: sidekick, microservice architecture, RAM optimization, memory leakage, asynchronous programming, minimal resource usage, streaming data, scalability, garbage collector.