НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.75

«До захисту допущено»
Завідувач кафедри
_____ Едуард ЖАРІКОВ
«__»_____ 2024р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Інженерія програмного забезпечення інформаційних систем»
зі спеціальності 121 «Інженерія програмного забезпечення»
на тему: «Методи та програмні засоби надання програмно-визначеної віддаленої пам'яті у розподілених системах»

Виконав:
студент ІІ курсу, групи ІП-22мп
Волобуєв Нікіта Олександрович _____

Керівник:
д.т.н., проф., засл.діяч
Павлов Олександр Анатолійович _____

Рецензент:
декан ФІОТ, професор кафедри ІСТ, д.т.н,
Корнага Ярослав Ігоревич _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2024 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____ 2023р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Волобуєву Нікіті Олександровичу

1. Тема дисертації «Методи та програмні засоби надання програмно-визначеної віддаленої пам’яті у розподілених системах», науковий керівник дисертації Павлов Олександр Анатолійович д.т.н., проф., засл.діяч, затверджені наказом по університету від «11» листопада 2023 р. № 5168-с.

2. Термін подання студентом дисертації «19» січня 2024 р.

3. Об’єкт дослідження – віддалена пам’ять у розподілених інформаційних системах.

4. Предмет дослідження – процес створення архітектури програмних засобів що реалізують методи забезпечення швидкого доступу до блоків даних у віддаленій пам’яті, їх реплікація, розгортання та інтеграція віддаленої пам’яті у програмне забезпечення.

5. Перелік завдань, які потрібно розробити – провести аналіз існуючих реалізацій та методів надання віддаленої пам’яті; розробити методи інтеграції віддаленої пам’яті у нове та існуюче програмне забезпечення; розробити архітектуру, структуру та взаємодію компонентів віддаленої пам’яті; знизити в середньому затримку доступу до блоків у віддаленій пам’яті за рахунок використання алгоритму заміщення, що спирається на статистику доступу до пам’яті та використання прогнозних моделей; розробити методи забезпечення відмовостійкості віддаленої пам’яті; провести оцінку ефективності запропонованого рішення.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 3 плакати

7. Орієнтовний перелік публікацій – одна публікація

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання «01» вересня 2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Огляд наявної літератури	01.09.2023	
2	Аналіз існуючих методів та реалізацій віддаленої пам’яті	10.09.2023	
3	Постановка та формалізація задачі	20.09.2023	
4	Розробка методів надання віддаленої пам’яті	03.10.2023	
5	Розробка методів забезпечення відмовостійкості та швидкодії віддаленої пам’яті	10.10.2023	
6	Розробка програмного забезпечення	15.10.2023	
7	Проведення експериментальних досліджень	01.11.2023	
8	Оформлення пояснювальної записки	10.11.2023	
9	Подання дисертації на попередній захист	19.12.2023	
10	Подання дисертації на захист	19.01.2024	

Студент Нікіта ВОЛОБУЄВ

Науковий керівник Павлов ОЛЕКСАНДР

РЕФЕРАТ

Розмір пояснювальної записки – 135 аркушів, містить 17 ілюстрацій, 23 таблиці, 7 додатків, 75 посилань на джерела.

Актуальність теми. У сучасних центрах обробки даних актуальним є збільшення ефективності використання ресурсів, зокрема оперативної пам’яті серверів. Навіть при наявності сучасних планувальників задач та віртуалізації, використання пам’яті є нерівномірним між вузлами у обчислювальному кластері. Застосування віддаленої пам’яті дозволяє використовувати оперативну пам’ять більш оптимально, знижуючи нерівномірність використання цього ресурсу, а також мати доступ до більшого об’єму пам’яті ніж є доступним на одному вузлі. При цьому, існуючі реалізації та методи надання віддаленої пам’яті мають обмежену область застосування та невисоку ефективність, зумовлену особливостями задачі. Через це, підвищення ефективності за рахунок модифікації існуючих методів надання віддаленої пам’яті є актуальним.

Мета дослідження. Підвищення ефективності використання оперативної пам’яті за рахунок розробки ефективного методу надання віддаленої пам’яті в інформаційному забезпеченні сучасних центрів обробки даних.

Об’єкт дослідження: віддалена пам’ять у розподілених інформаційних системах.

Предмет дослідження: процес створення архітектури програмних засобів що реалізують методи забезпечення швидкого доступу до блоків даних у віддаленій пам’яті, їх реплікація, розгортання та інтеграція віддаленої пам’яті у програмне забезпечення.

Для реалізації поставленої мети **сформульовані наступні завдання**:

– провести аналіз існуючих реалізацій та методів надання віддаленої пам’яті;

– розробити методи інтеграції віддаленої пам’яті у нове та існуюче програмне забезпечення;

– розробити архітектуру, структуру та взаємодію компонентів віддаленої пам’яті;

– знизити в середньому затримку доступу до блоків у віддаленій пам’яті за рахунок використання алгоритму заміщення, що спирається на статистику доступу до пам’яті та використання прогнозних моделей;

– розробити методи забезпечення відмовостійкості віддаленої пам’яті;

– провести оцінку ефективності запропонованого рішення.

Наукова новизна: на відміну від існуючих методів, задача заміщення проміжків вирішена статистично більш ефективно за рахунок реалізації адаптації параметрів моделі прогнозування доступу на основі використання статистики, що неперервно збирається в процесі роботи програмного забезпечення.

Практичне значення отриманих результатів полягає в тому, що розроблене програмне забезпечення для надання віддаленої пам’яті є простим для розгортання, не вимагає значних змін у програмне забезпечення при інтеграції. Дане програмне забезпечення може бути використане для підвищення ефективності використання ресурсів центру обробки даних у програмному забезпеченні параметри роботи якого дозволяють використання такого класу пам’яті як віддалена пам’ять.

Зв’язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Апробація. Наукові положення дисертації пройшли апробацію на V Міжнародній науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології SoftTech-2023».

Публікації. Наукові положення дисертації опубліковані в:

1) Methods and software for providing software-defined far memory in distributed systems/ Н.О. Волобуєв, О.А. Павлов, М.М. Головченко // Матеріали V Міжнародної науково-практичної конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології

SoftTech-2023» – м. Київ: НТУУ «КПІ ім. Ігоря Сікорського», 12-21 грудня 2023 р.

Ключові слова: ВІДДАЛЕНА ПАМ'ЯТЬ, РОЗПОДІЛЕНІ СИСТЕМИ, КОМП'ЮТЕРНІ МЕРЕЖІ, ЗАМІЩЕННЯ СТОРІНОК, LINUX, RUST

ABSTRACT

Explanatory note size – 135 pages, contains 17 illustrations, 23 tables, 7 applications, 75 references.

Topicality. Increasing resource utilization efficiency (random access memory in particular) is an important problem which arises in modern data centers. Memory utilization is uneven between nodes in a computing cluster even with modern schedulers and virtualization. Far memory allows using random-access memory more efficiently and evenly, while also allowing to access more memory than available on a compute node. At the same time, existing implementations and methods of providing far memory have a limited application scope and low efficiency, which are defined by the specifics of the task. Because of that, increasing efficiency by modifying existing methods of providing far memory is relevant.

The aim of the study. Improving the efficiency of using far memory by developing an effective method for providing far memory to the information software of modern datacenters.

Object of research: far memory in distributed information systems.

Subject of research: the process of creation of the architecture of software tools that implement methods for providing fast access to span in far memory, their replication, deployment and integration of far memory into the software.

To achieve this goal, the **following tasks** were formulated:

– perform analysis of existing far memory implementations and methods;

– develop far memory integration methods into new and existing software;

– develop architecture, structure and interaction between far memory components;

– decrease average latency of far memory spans access by using span replacement algorithm that relies on memory access statistics and predictive models;

– develop methods to ensure far memory resiliency;

– assess solution efficiency.

The scientific novelty unlike existing methods, span replacement problem is solved statistically more efficiently by implementing the adaptation of span access

prediction model parameters based on statistics that are continously collected during runtime of software.

The practical value of the obtained results lies in the fact that the developed software for providing far memory is easy to deploy and does not require significant changes to the software during integration. This software can be used to enhance the efficiency of resource utilization in datacenter for software which operating parameters allow using such memory class as far memory.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Informatics and Software Engineering of the National Technical University of Ukraine «Kyiv Polytechnic Institute. Igor Sikorsky».

Approbation. The scientific provisions of the dissertation were tested at the V International Scientific and Practical Conference for Young Scientists and Students "Software Engineering and Advanced Information Technologies SoftTech-2023".

Publications. The scientific provisions of the dissertation published in:

1) Methods and software for providing software-defined far memory in distributed systems/ N.O. Volobuev, O.A. Pavlov, M.M. Holovchenko // Proceedings of the V International Scientific and Practical Conference for Young Scientists and Students "Software Engineering and Advanced Information Technologies SoftTech-2023" - Kyiv: National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», December 19-21, 2023.

Keywords: FAR MEMORY, DISTRIBUTED SYSTEMS, COMPUTER NETWORKS, PAGE REPLACEMENT, LINUX, RUST

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

TCP – Transmission Control Protocol, мережевий протокол транспортного рівня;

RDMA – Remote Direct Memory Access, технологія яка дозволяє вузам в системі працювати з даними у пам'яті інших вузлів, не використовуючи ресурси їх процесору;

HPC – High Performance Computing, підхід у вирішенні задач, які потребують багато обчислень за допомогою суперкомп'ютерів або обчислювальних кластерів;

MPI – Message Passing Interface, стандарт обміну інформацією в програмному забезпеченні, яке працює на обчислювальних кластерах;

zswap – функціонал ядра операційної системи Linux, що дозволяє стискати пам'ять у файлах підкачки;

RAM - random-access memory, оперативна пам'ять, у якій зберігаються код та дані програм;

latency - затримка часу між запитом до даних та поверненням системою даних у відповідь;

span - сторінка або блок пам'яті, що є одиницею, з якою працює віддалена пам'ять;

LRU - least recently used, алгоритм заміщення найменш нещодавно використаних елементів у пам'яті.

ВСТУП

У сучасному світі дуже поширеним є хмарне програмне забезпечення, яке з кожним днем замінює собою або інтегрується у вигляді нового функціоналу у існуюче програмне забезпечення в усіх галузях використання. Центральним компонентом такого програмного забезпечення є його серверна частина, що обслуговує запити багатьох користувачів. Цей компонент обробляє велику кількість запитів від різних користувачів зазвичай виконуючи найбільш ресурсоємну частину роботи у порівнянні з частиною розміщеною на пристрої кінцевого користувача. Оскільки ці ресурси зазвичай обмежені можливостями обладнання, що використовується (чи бюджетом на оренду такого обладнання), то будь-яка оптимізація використання ресурсів призводить до можливості обробляти більшу кількість запитів та тому ж самому обладнанні (що в результаті знижує витрати).

Оператори великих центрів обробки даних вже великий час застосовують різні методи для підвищення ефективності використання ресурсів серверного обладнання. Так, наприклад, для ефективного використання ресурсів процесору використовується підхід “надмірної підписки” (oversubscription) обчислювального часу. Схожий метод використовується і при організації інфраструктури сховищ даних в додачу до компресії та дедублікації даних.

Якщо перейти до ефективності використання оперативної пам'яті, то оператори найбільших у світі центрів обробки даних зазначають, що середнє використання пам'яті знаходиться на рівні близько 60% [1]. Для того, щоб покращити цей показник розробляються різні методи. Одним з цих методів є використання віддаленої пам'яті (Far Memory).

Сервери у центрі обробки даних (і програмне забезпечення, що на них розгорнуте) можна поділити на два типи:

– сервери, на яких більша частина пам’яті є вільною;

– сервери, які могли б цю пам’ять використовувати, якщо мали би до неї доступ.

Суть методу віддаленої пам'яті полягає в тому, що сервери з вільною пам'яттю можуть надавати доступ до неї по мережі тому програмному забезпеченню, яке могло б її використовувати для зберігання тієї частини даних, що підходить для зберігання за умов та обмежень, що накладає віддалена пам'ять.

У даній роботі розглянуто методи надання програмно-визначеної віддаленої пам'яті у розподілених системах, а також способи зниження затримки доступу до даних у віддаленій пам'яті та забезпечення відмовостійкості.

1 ОГЛЯД ІСНУЮЧИХ МЕТОДІВ НАДАННЯ ВІДДАЛЕНОЇ ПАМ’ЯТІ

1.1 Ресурси обладнання у розподілених системах та проблема їх ефективного використання

Будь-який сучасний центр обробки даних складається з великої кількості серверного та мережевого обладнання. На цьому обладнанні виконується програмне забезпечення, що обробляє запити від користувачів та може бути частинами розподілених систем.

Під час своєї роботи на цьому обладнанні, програмне забезпечення може використовувати наступні його ресурси:

– процесорний час;

– оперативна пам’ять;

– постійна пам’ять на різних типах сховища, таких як жорсткі диски, твердотільні накопичувачі на ін.;

– спеціалізовані пристрої, такі як графічні прискорювачі.

Для кожного з цих ресурсів існує проблема їх ефективного використання та різні рішення для досягнення такої мети.

Один з методів який дозволяє підвищити ефективність використання ресурсів процесору є “надмінна підписка” (oversubscription) [2] його обчислювального часу. Це означає що на одному процесорі запускається декілька різних програм або віртуальних машин, кожна з яких використовує його частину часу, а разом всі - використовують процесор майже весь час, при цьому розрахунок йде на те, що піки завантаженості цих програм не збігаються.

Через особливості того, як програмне забезпечення працює з оперативною пам'яттю, вона є найбільш складним ресурсом, ефективність використання якого можна було б підвищити. Одним з підходів, що останнім часом багато досліджується та розглядається операторами великих центрів обробки даних для інтеграції є віддалена пам'ять (Far Memory) [3].

Суть цього методу полягає в тому, що сервери у центрі обробки даних (і програмне забезпечення, що на них розгорнуте) можна поділити на два типи:

– сервери, на яких більша частина пам’яті є вільною;

– сервери, які могли б цю пам’ять використовувати, якщо мали би до неї доступ.

Програмне забезпечення першого типу зазвичай має “вузьке місце” у ресурсах процесору (наприклад, виконує задачі кодування даних, або простого обміну даними), програмне забезпечення другого - у ресурсах пам'яті (зазвичай це аналіз великих масивів даних або просто у програмного забезпечення є деякий великий набір даних, який йому потрібен для роботи). Використання пам'яті диску для розширення основної пам'яті не є оптимальним - через великий час доступу (а в хмарній інфраструктурі в додаток до цього зазвичай диски не є локальними, а розміщені віддалено на окремій інфраструктурі [4]). У порівнянні з часом доступу до диску час доступу до даних у пам'яті іншого серверу є значно меншим [5] (хоча все ще більшим за той випадок, коли дані доступні локально).

На рисунку 1.1 схематично показано принцип роботи віддаленої пам'яті при навявності серверів з різним рівнем використання оперативної пам'яті.

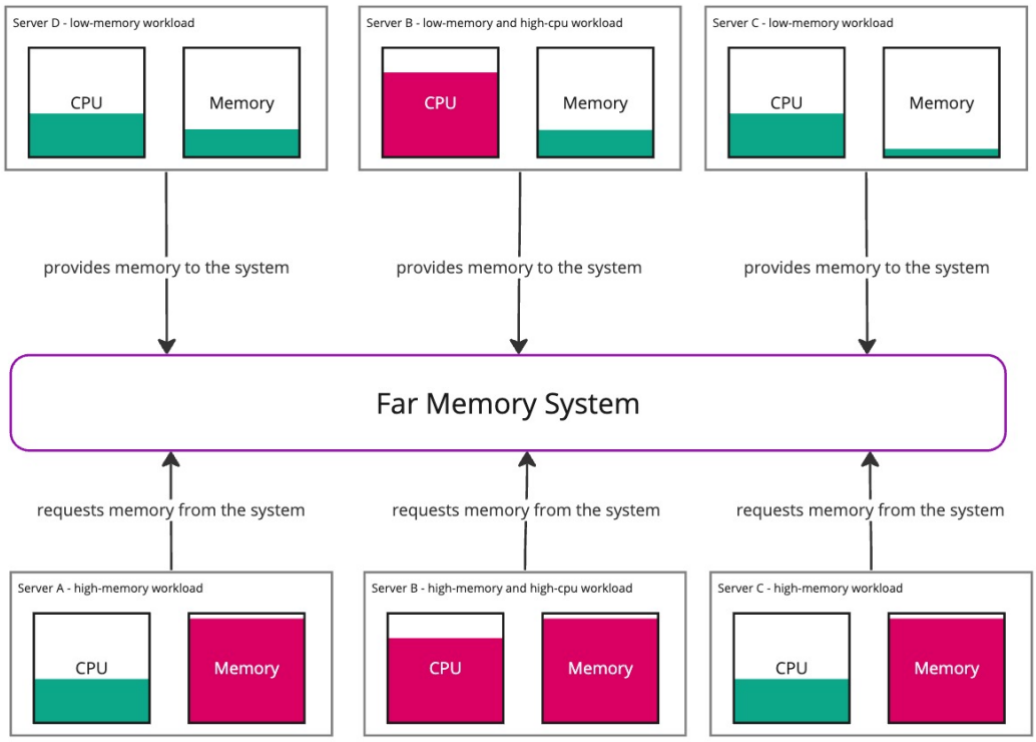


Рисунок 1.1 – Схематичне зображення принципу роботи віддаленої пам’яті

Це все робить використання такої віддаленої пам'яті привабливим для випадків, коли можна знайти сторінки пам'яті, доступ до яких відбувається

порівняно не часто, перемістити їх у віддалену пам'ять та звільнити місце для даних, доступ до яких відбувається частіше.

1.2 Огляд існуючих реалізацій віддаленої пам’яті

Існуючі методи надання віддаленої пам'яті можна проаналізувати у контексті критеріїв, що впливають на область застосування такої системи та розглядаються у межах цієї роботи.

Важливою характеристикою цього класу пам'яті є залежність від спеціалізованого апаратного та програмного забезпечення у середовищі, де вона розгортається. Хоча використання таких компонентів (наприклад, мережевих карт з підтримкою технологій прямого доступу до пам'яті віддалених вузлів) дозволяє значно знизити затримку доступу до даних, воно також значно звужує область застосування на практиці, або робить використання недоцільним [6] чи неможливим.

Іншим важливим аспектом є метод інтеграції у програмне забезпечення. Це є складною задачею, оскільки доступ до даних у віддаленій пам'яті повинен бути прозорим та схожим на роботу з даними у локальній пам'яті, не вимагаючи значних змін у код програмного забезпечення. В іншому випадку, це робить використання віддаленої пам'яті менш привабливим за зберігання даних у віддалених базах даних чи використання інструментів для розподіленої обробки даних, таких як Spark [7].

На можливість застосування віддаленої пам'яті також впливає швидкість доступу до даних, що забезпечується реалізацією. Чим більшим є негативний вплив на швидкодію, тим менше область застосування (наприклад, програмне забезпечення що виконує фонову обробку даних є менш чутливим до затримок, а програмне забезпечення що обробляє запити користувача у інтерактивному режимі є більш чутливим). Зниження затримки може забезпечуватись через оптимізацію реалізації переміщення даних, а також завдяки прогнозуванню доступу до даних, та завчасному переміщенню у локальну пам'ять.

Оскільки зберігання даних на інших вузлах робить систему менш відмовостійкою, то важливим є те, як реалізація віддаленої пам'яті знижує ризик втрати даних і який рівень надлишковості та додаткового використання ресурсів є необхідним для цього.

1.3 Remote Direct Memory Access та її реалізації

Технологія віддаленого прямого доступу до пам'яті (Remote Direct Memory Access) полягає в використанні спеціальних апаратних засобів, що дозволяють вузлам в системі отримувати дані з невеликою затримкою з інших вузлів без витрачання ресурсів процесору цих вузлів для обробки запитів.

Однією з найбільш розповсюджених реалізацій RDMA є InfiniBand [8]. Ця реалізація забезпечує пропускну здатність до 50Гбіт/с на кожен кабель, при цьому має затримку що є в десятки раз меншою (в залежності від розміру повідомлення) ніж за умов пересилання даних по TCP [9] через Gigabit Ethernet [10].

RDMA використовується і є виправданим для використання в середовищах високопродуктивних обчислень (High Performance Computing - HPC). Завдяки низькій затримці, програмне забезпечення може ефективно працювати з наборами даних, які розподілені у оперативній пам'яті багатьох вузлів.

Головним недоліком цієї реалізації віддаленої пам'яті є те, що вона потребує додаткового спеціалізованого обладнання. Для задач та середовища що розглядаються в цій роботі не є підхожим рішенням, тому що використання додаткових пристроїв потребує додаткових ресурсів і не вирішує проблему більш ефективного використання наявних ресурсів без змін в апаратну платформу. Крім цього, RDMA вирішує задачу передачі даних з низькою затримкою між вузлами, а забезпечення відмовостійкості та керування переміщенням даних між локальною пам'яттю та пам'яттю віддалених вузлів забезпечується розробником прикладного програмного забезпечення. Для інтеграції у програмне забезпечення зазвичай потрібні значні зміни, наприклад використання MPI (Message Passing Interface) [11].

1.4 Hydra: Resilient and Highly Available Remote Memory

Hydra [12] це реалізація віддаленої пам'яті, яка використовує RDMA для передачі даних між вузлами, забезпечує відмовостійкість та вирішує задачу інтеграції у програмне забезпечення.

Hydra інтегрується у програмне забезпечення через використання механізму підкачки у операційній системі Linux [13]. Компонент системи (resource monitor) що працює у просторі користувача надає віртуальний блоковий пристрій, на якому розміщується розділ підкачки пам'яті. Це надає можливість інтегруватись у нове та існуюче програмне забезпечення без змін у код.

Для забезпечення відмовостійкості, у цій роботі пропонується підхід під назвою CodingSets. Під час перенесення даних у пам'ять інших вузлів, виконується кодування кодом Ріда-Соломона [14], що дає декілька частин даних (data splits) та парності (parity splits). Підхід CodingSets полягає у тому, щоб оптимально розподілити частини між вузлами таким чином, щоб не тільки розподілити навантаження, а й максимально знизити ймовірність втрати даних у разі одночасного виходу з ладу декількох вузлів. Операції кодування та розподілення частин виконуються компонентом resilience manager який реалізований як модуль ядра операційної системи. Коли один з вузлів виходить з ладу, операції читання та запису перенаправляються на інші вузли, доки відбувається відновлення даних на новий вузол.

Низький рівень затримки доступу до даних у віддаленій пам'яті забезпечується використанням RDMA, в тому числі тим фактом що малі повідомлення (як результат кодування) швидко передаються. Крім цього, під час перенесення даних з віддаленої пам'яті у локальну системі достатньо отримати частини даних лише з частки вузлів (число залежить від параметрів кодування стиранням, які були обрані користувачем). Для зниження затримки під час запису даних, ця реалізація віддаленої пам'яті відправляє дані парності асинхронно, очікуючи на успішний запис лише основної частини даних.

Головним недоліком цієї реалізації віддаленої пам'яті є залежність від RDMA, що вимагає спеціального апаратного забезпечення (мережевих карт, що

її підтримують). Оскільки RDMA не є розповсюдженим у сучасних центрах обробки даних (за винятком спеціалізованих HPC кластерів), то можливість використання Hydra на практиці є обмеженим.

1.5 AIFM: High-Performance Application-Integrated Far Memory

Система Application-Integrated Far Memory (AIFM) [15] на відміну від попередніх реалізацій віддаленої пам'яті не потребує спеціалізованого апаратного забезпечення, для передачі даних використовується TCP.

Головною відмінною рисою цієї реалізації є інтеграція з клієнтським програмним забезпеченням за допомогою клієнтської бібліотеки на C++ що надає розумні покажчики та структури даних оптимізовані для використання з віддаленою пам'яттю.

Для спілкування між сервером що надає пам'ять та сервером що її потребує використовується TCP/IP з'єднання.

Для виявлення ділянок пам'яті, що можна перенести у віддалену пам'ять, використовується механізм відслідковування рівня гарячості сторінок пам'яті. Також, для підвищення ефективності, структури даних визначені бібліотекою AIFM реалізують предзавантаження наступних ділянок пам'яті.

Для цієї реалізації є доступним програмний код. Недоліком є те, що без додаткової доробки він підтримує лише один сервер що надає віддалену пам'ять. Система не містить компоненту для розподілення та керування пам'яттю по кластеру вузлів. Також реалізація не є придатною для розгортання в середовищах що обробляють справжнє навантаження без додаткової доробки. Система має складний процес розгортання, залежить від зовнішніх компонентів, не має простих механізмів конфігурування та моніторингу.

1.6 Carbink: Fault-Tolerant Far Memory

Carbink [16] це система віддаленої пам'яті розроблена та протестована компанією Google в своїх центрах обробки даних. Ця реалізація фокусується на покращені відмовостійкості та зниженні рівня затримок при роботі з віддаленою пам'яттю.

Система складається з декількох компонентів: memory nodes (вузли, що зберігають дані), compute nodes (програмне забезпечення, в яке інтегрується віддалена пам'ять) та memory manager (менеджер пам'яті). Менеджер пам'яті керує розподілом фрагментів пам'яті по вузлах, що їх зберігають та перевіряє стан роботи цих вузлів. Важливим припущенням є те, що вважається що менеджер пам'яті завжди залишається в робочому стані. Мережевий зв'язок теж вважається стабільним.

Для інтеграції у програмне забезпечення використовується бібліотека яка надає застосунку доступ до даних у віддаленій пам'яті через розумні покажчики. При розміщенні даних у віддаленій пам'яті вони групуються у сторінки (spans).

Групування сторінок відбувається за їх розміром - такий самий підхід, який використовується у алокаторах пам'яті, наприклад у TCMalloc [17]. Окремий потік у фоновому режимі переміщує об'єкти, доступ до яких відбувається частіше у спільні сторінки. Для того, щоб відслідковувати частоту доступу до об'єктів, використовується підхід схожий на бар'єри запису та читання у прибиральниках сміття. Для роботи з байтом активності (hotness byte) використовується алгоритм CLOCK [18]. Також слід зазначити, що переміщення даних між локальною та віддаленою пам'яттю відбувається на рівні гранулярності сторінки (span). Коли системі потрібно звільнити пам'ять, то переміщуються у першу чергу сторінки з найбільшою кількістю холодних об'єктів.

Для забезпечення відмовостійкості, менеджер пам'яті постійно перевіряє стан інших вузлів (як тих, що зберігають дані, так і тих, на яких розміщене програмне забезпечення, в яке інтегрується віддалена пам'ять) за допомогою healthcheck запитів. Якщо вузол обчислень (compute node) виходить з ладу, то вузлам що зберігають дані, надається команда на звільнення пам'яті. Якщо вузол зберігання даних (storage node) виходить з ладу, то дані відновляться з використанням частин парності (parity shards) які були розміщені на інших вузлах після кодування стиранням (кодування відбувається одразу на рівні декількох сторінок). На час відновлення даних та переміщення на новий вузол

очікується більш високий рівень затримки операції читання. Також менеджер пам'яті підтримує планове обслуговування вузлів, коли дані завчасно переміщуються.

Головним недоліком цієї реалізації віддаленої пам'яті є те, що програмний код не є публічно доступним, а також його прив'язаність до програмної та апаратної інфраструктури, що використовується в компанії Google. Крім цього, ця реалізація не має механізмів оптимізації параметрів роботи з урахуванням закономірностей у доступі до даних програмним забезпеченням.

1.7 Software-Defined Far Memory in Warehouse-Scale Computers

Ця [19] реалізація віддаленої пам'яті зберігає дані не на віддалених вузлах, а у локальній пам'яті, але у стиснутому стані. Для цього, реалізація спирається на функціонал zswap [20] у операційній системі Linux.

Найбільший ефект на швидкодію програмного забезпечення та кількість звільненої пам'яті має алгоритм ідентифікації холодних сторінок (тобто сторінок пам'яті з низькою частотою доступу). Ціллю алгоритму є пошук такого рівню відносно якого сторінки вважаються холодними, щоб максимізувати кількість сторінок у віддаленій пам'яті при цьому задовольняючи вимоги програмного забезпечення щодо швидкодії. Цей рівень виражається у частці від об'єму пам'яті, доступ до якої програма виконує за хвилину. Для встановлення значення цього рівня будується гістограма за попередньо зібраною статистикою та обирається той рівень, який задовлльняє вимогам по швидкодії.

Виявлення сторінок які задовольняють критерію відбувається у фоновому режимі, після чого zswap виконує стиснення цих даних для звільнення пам'яті.

Окремий компонент збирає статистику з усіх вузлів у кластері та за допомогою алгоритмів машинного навчання оптимізує значення гіперпараметрів. Це дозволяє оптимізувати агресивність алгоритму перенесення даних у віддалену пам'ять з урахуванням різниці у закономірностях доступу до пам'яті між обчислювальними вузлами у кластері, різними датацентрами та різним програмним забезпеченням. Дані збираються за допомогою існуючої

інфраструктури збору телеметрії. Для оптимізації використовується алгоритм машинного навчання Gaussian Process (GP) Bandit [21], який є оптимізацією чорного ящику (black-box optimization [22]). У роботі зазначається, що це підвищує ефективність системи на 30% у порівнянні з евристичними підходами. Алгоритм генерує декілька можливих конфігурацій параметрів системи, які оцінюються за допомогою симуляції роботи за статистикою, що була зібрана протягом тижня. Найкраща конфігурація (за показником звільненої пам'яті) передається для використання на всьому кластері.

Недоліками цієї реалізації є те, що вона використовує локальну пам'ять для збереження даних, а не пам'ять віддалених вузлів. Крім цього, код реалізації є закритим, недоступним для використання ззовні та прив'язаним до інфраструктури що використовується у компанії Google.

1.8 Порівняння існуючих рішень

У таблиці 1.1 наведено порівняння основних властивостей існуючих рішень, що було розглянуто.

Таблиця 1.1 - Порівняльна таблиця існуючих методів надання віддаленої пам’яті

	Carbink	**AIFM**	**Hydra**	**Software-Defined Far Memory in Warehouse-Scale Computers**
Відкритий вихідний код та доступність для зовнішнього використання	-	+	+	-
Не залежить від спеціалізованого апаратного забезпечення	+	RDMA NIC	RDMA NIC	+
Зберігання даних на багатьох віддалених вузлах	+	-	+	-

Продовження таблиці 1.1

Підтримка інтеграції без зміни коду	-	-	+	+
Зниження затримки за рахунок керування заміщенням сторінок	прості евристики	prefetching	прості евристики	адаптація гіперпараметрів механізму підкачки

1.9 Постановка задачі

Метою роботи є покращення програмних засобів та методів, що можуть використовуватись операторами центрів обробки даних та розробниками програмного забезпечення для розгортання та використання віддаленої пам'яті. Для досягнення мети необхідно вирішити наступні задачі:

– провести аналіз існуючих реалізацій та методів надання віддаленої пам’яті;

– розробити методи інтеграції віддаленої пам’яті у нове та існуюче програмне забезпечення;

– розробити архітектуру, структуру та взаємодію компонентів віддаленої пам’яті;

– знизити в середньому затримку доступу до блоків у віддаленій пам’яті за рахунок використання алгоритму заміщення, що спирається на статистику доступу до пам’яті та використання прогнозних моделей;

– розробити методи забезпечення відмовостійкості віддаленої пам’яті;

– провести оцінку ефективності запропонованого рішення.

Створене програмне забезпечення повинно відповідати наступним вимогам:

– реалізація віддаленої памя’ті повинна містити компонент, який користувачі системи можуть розгорнути на вузлах системи (під управлінням операційної системи Linux), що мають вільну пам’ять для її використання по мережі. Цей компонент повинен використовувати невелику кількість ресурсів, та

для зберігання даних використовувати кількість пам’яті задану користувачем або визначену автоматично;

– реалізація повинна мати варіанти інтеграції як в нове програмне забезпечення (де є можливість змінювати програмний код) так і в існуюче (де змінювати код не є можливим);

– показники швидкодії віддаленої пам’яті в операціях читання та запису повинні бути кращими за типові показники при роботі з даними у постійному сховищі (наприклад, жорсткі диски та твердотільні накопичувачі), що зазвичай використовуються у середовищах з дезагрегованими ресурсами.;

– наявність центрального компоненту, який налаштовує конфігурацію та дозволяє керувати усією системою;

– реалізація повинна мати автоматичну зміну параметрів з урахуванням особливостей програмного забезпечення, що використовується.;

– повинна забезпечуватись відмовостійкість та збереження даних що зберігаються у разі апаратних чи програмних збоїв у кластері;

– програмне забезпечення повинне бути простим у розгортанні, адмініструванні а також у інтеграції в клієнтське програмне забезпечення.

Призначенням цієї розробки є надання програмних засобів для розгортання віддаленої пам'яті та інструментів для її використання в існуючому та новому програмному забезпеченні.

Висновки до розділу

У цьому розділі розглянуто проблему надання віддаленої пам'яті у розподілених системах та виконано аналіз існуючих реалізацій. Визначено які підходи використовуються для інтеграції віддаленої пам'яті у нове та існуюче програмне забезпечення, зниження затримки доступу та забезпечення відмовостійкості. Розглянуто переваги та недоліки існуючих реалізацій.

За результатами розгляду існуючих реалізацій, можна зробити висновок що Carbink та AIFM є найбільш підходящими для використання на практиці, але вони, як і інші реалізації мають певні обмеження. З цього випливає потреба у

розробці нової реалізації, яка б була розвитком цих реалізацій, при цьому б не залежала б від спеціалізованого апаратного та програмного забезпечення, підтримувала б оптимізацію параметрів роботи для підвищення ефективності, мала б відкритий програмний код, тим самим усуваючи недоліки існуючих рішень.

В результаті проведеного аналізу сформульована постановка задачі, наведене призначення, цілі та задачі розробки.

2 МЕТОДИ ТА ЗАСОБИ НАДАННЯ ВІДДАЛЕНОЇ ПАМ’ЯТІ

Загалом, принцип роботи віддаленої пам'яті полягає в тому, що програмне забезпечення передає у віддалену пам'ять дані для зберігання, а коли до цих даних потрібен доступ, то реалізація віддаленої пам'яті за запитом від застосунку переміщує дані з інших більш повільних пристроїв зберігання даних у локальну оперативну пам'ять. Після переміщення, програмне забезпечення працює з даними так само, як і з будь-якими іншими даними у оперативній пам'яті. Після того, як робота з даними закінчена, вони знов переміщуються на зберігання у інший клас пам'яті. Саме це надає можливість зменшити використання оперативної пам'яті вузла та обробляти більше даних ніж об'єм пам'яті прозоро для програмного забезпечення (тобто без значних змін у код, та те, як застосунок працює з даними).

2.1 Компоненти системи

Перший компонент, з якого варто почати розглядати систему це вузли обчислення (compute nodes). Ці вузли є програмним забезпеченням, у яке інтегрується віддалена пам'ять. Слід зазначити, що вузли розглядаються не з точки зору фізичного обчислювального вузла, а як екземпляр програмного забезпечення, що виконується. Користуватися віддаленою пам'яттю може одночасно різне програмне забезпечення, різна кількість його екземплярів, різні версії з різними налаштуваннями. При цьому, як і у звичайному програмному забезпеченні, що працює з локальною оперативною пам'яттю, програма що зберігає дані у віддаленій пам'яті не має доступу до даних інших програм.

Вузол обчислення складається з програмного забезпечення у яке інтегрована віддалена пам'ять та клієнта віддаленої пам'яті. Програмне забезпечення працює з клієнтом віддаленої пам'яті для розміщення даних у ній та для отримання доступу до цих даних. Крім цього, про властивості програмного забезпечення у яке інтегрується віддалена пам'ять не робиться додаткових припущень.

Основною сутністю, з якою працює клієнт віддаленої пам'яті, є проміжок пам'яті (span). Проміжок пам'яті можна вважати аналогічним сторінці пам'яті у операційній системі, тобто це безперервним блоком пам'яті. Проміжок пам'яті має ідентифікатор (64-бітне число), яке ключем у будь-яких операціях пов'язаних з проміжком. Пам'ять, яка пов'язана з проміжком, має фіксовану довжину, тобто не змінюється після створення проміжку, але одночасно можуть існувати проміжки різної довжини. Дані (послідовність байтів) можуть знаходитись на різних пристроях зберігання, проте вони залишаються прив'язаними до цього проміжку.

Для зберігання даних за межами оперативної пам'яті, клієнт віддаленої пам'яті використовує задану користувачем конфігурацію бекенду (backend). В цій роботі розглядаються різні реалізації бекенду, такі як локальна пам'ять, SSD диски, пам'ять одного чи декількох віддалених вузлів, але головна увага приділяється зберіганню даних у пам'яті багатьох віддалених вузлів. Саме переміщення даних у пам'ять багатьох віддалених вузлів надає цій розробці практичної цінності.

Тому вузли зберігання даних (storage nodes) є іншим компонентом системи. Як бекенд віддаленої пам'яті, цей вузол можна розглядати як сховище у форматі ключ-значення, де ключем є ідентифікатор проміжку пам'яті, а значенням - дані, що зберігаються у цьому проміжку. Вузли зберігання даних обробляють запити на запис та читання проміжків (при цьому читання видаляє дані, так як немає сенсу одночасно зберігати одні й ті самі дані як на вузлах зберігання, так і на вузлах обчислення). Таке формулювання призначення цих вузлів може зробити привабливим використання сховищ даних які працюють за схожим принципом, таких як наприклад Redis [23]. Однак, використання такого сховища даних наклало б обмеження на можливості оптимізації протоколу, необхідних для мінімізації затримки з урахуванням особливостей задачі, що вирішується.

Наявність багатьох вузлів зберігання та обчислення створює необхідність у компоненті, який керував би роботою кластеру - вузла керування (manager

node). Цей компонент повинен вирішувати декілька задач. По-перше, за запитом від вузлів обчислення надавати доступ до вузлів зберігання згідно з кількістю пам'яті необхідної для програмного забезпечення. Для цього, вузол керування відслідковує рівень завантаженості вузлів у кластері, та пріоритизує ті вузли, де найбільше вільної пам'яті. Крім цього, вузол керування відслідковує стан інших компонентів та обробляє випадки виходу з ладу інших вузлів, так і їх запланованого виводу на обслуговування. Також цей компонент надає інструменти для моніторингу стану віддаленої пам'яті. У випадку використання локальної пам'яті (як наприклад SSD диску) чи лише одного віддаленого вузла зберігання наявність вузла керування не є необхідним.

Вузол керування також збирає статистику про роботу системи та визначає параметри її роботи (наприклад, умови за яких проміжки пам'яті переміщуються між локальною та віддаленою пам'яттю). Ці обчислення відбуваються на цьому вузлі через наявність більшої інформації про роботу системі ніж та, що є доступною на окремих вузлах. Також, централізоване керування параметрами роботи віддаленої пам'яті спрощує її використання для користувачів (розробників програмного забезпечення, у яке вона інтегрується) - це дозволяє уникнути проблеми задавати параметри на кожному вузлі окремо. В архітектурі системи, що була обрана, компоненти віддаленої пам'яті отримують параметри роботи від вузла керування під час роботи системи. Початкові параметри задаються в момент ініціалізації віддаленої пам'яті та можуть змінюватись під час її роботи. Зміна параметрів під час роботи є необхідною, так як поведінка та закономірності роботи з пам'яттю програмного забезпечення як правило змінюються у часі і система повинна адаптувати параметри з урахуванням нової інформації.

На рисунку 2.1 схематично зображено компоненти з яких складається програмне забезпечення для надання віддаленої пам'ять та зв'язки між ними.

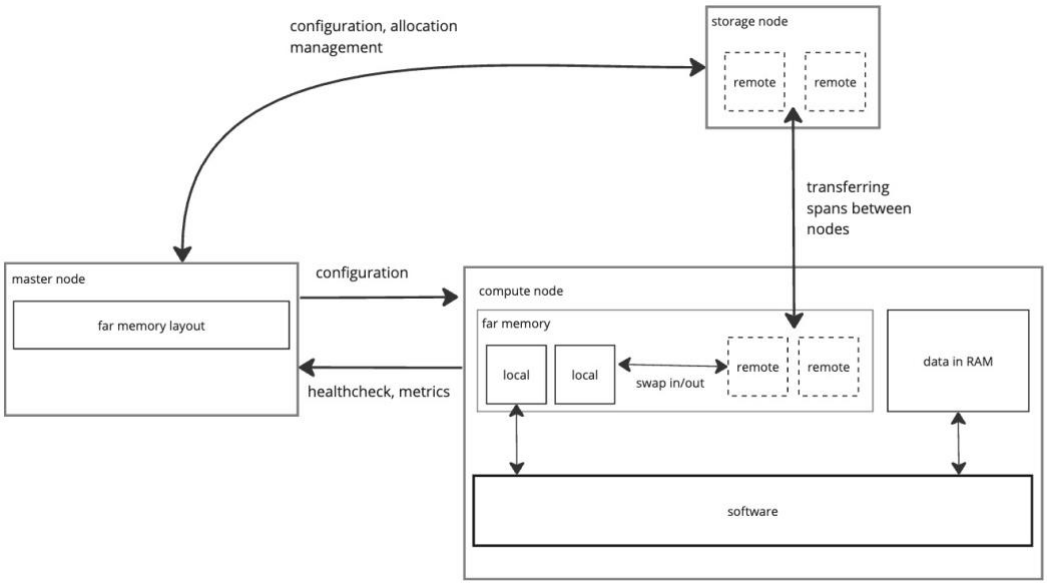


Рисунок 2.1 – Схематичне зображення компонентів віддаленої пам’яті, зв’язків між ними та переміщення проміжків

Кожен з цих компонентів надається користувачу як виконуваний файл для операційної системи Linux (за винятком клієнту віддаленої пам'яті який може інтегруватися у вузол обчислення за допомогою бібліотеки або окремого сервісу, що взаємодіє з механізмами керування пам'яттю операційної системи). Також надаються Docker [24] контейнери та конфігурація Kubernetes [25], що спрощує розгортання у кластері. До середовища, в якому розгортається система, ставиться вимога що всі вузли доступні всім іншим вузлам за мережею.

При використанні реалізації відаленної пам'яті на практиці виникає потреба у інтеграції з зовнішніми системами моніторингу для того, щоб одночасно контролювати як роботу віддаленої пам'яті, так і програмного забезпечення у яке вона інтегрована. Для цього, у кожному з компонентів віддаленої пам'яті передбачено відправку метрик у окремому потоці через регулярні проміжки часу у Prometheus [26] або інші сховища метрик сумісні з ним. Для відправки метрик використовується push-механізм, так як це робить комунікацію з зовнішньою системою моніторингу більш простою, прибираючи необхідність у використанні механізму Service Discovery.

Так як вузли обчислення підключаються одночасно до декількох вузлів зберігання та до вузла керування, то виникає потреба в можливості ідентифікації

сесії роботи з віддаленою пам'яттю. Це також потрібно для того, щоб у разі закриття та повторного відкриття з'єднання сервер міг встановити що клієнт є тим самим екземпляром програмного забезпечення. Для вирішення цієї проблеми, у протокол передачі інформації між вузлами додається ідентифікатор сесії (run id), котрий генерується на вузлі обчислення під час запуску програми та ініціалізації клієнту віддаленої пам'яті і залишається незмінним увесь час роботи програми.

Крім цього, вузол обчислення при встановленні з'єднання з вузлом керування крім run id також передає таку інформацію як тип та версія програмного забезпечення. Це потрібно для моніторингу та збору статистики для автоматичної оптимізації параметрів віддаленої пам'яті.

2.2 Інтеграція у програмне забезпечення

Складність інтеграції віддаленої пам'яті у програмне забезпечення полягає у тому, що зазвичай у сучасних мовах програмування доступ до даних відбувається з припущенням що всі дані розміщені в локальній оперативній пам'яті. Тобто структури даних, з якими працює програма, використовують покажчики на пам'ять які були видані алокатором пам'яті. Це робить неможливим прозору роботу з даними у віддаленій пам'яті для програми, так як не є можливим створити покажчик на пам'ять у іншому пристрої. В операційних системах звісно існує віртуальна пам'ять і, наприклад, механізм memory mapping, але сам по собі він не підходить для того, щоб прозоро (тобто без значних змін у код) та швидко працювати з даними у пам'яті віддалених вузлів.

Можливим рішенням було б внести зміни в механізми роботи операційної системи Linux з пам'яттю. Але значним недоліком цього підходу є необхідність користувачів компілювати та встановлювати ядро операційної системи з необіхними змінами - що робить використання віддаленої пам'яті з такою реалізацією на практиці надто складним. Навіть якщо потрібний функціонал реалізувати як окремий модуль ядра, це достатньо щоб зменшити область застосування на практиці. Крім цього, недоліком такого підходу є те, що на рівні

операційної системи є менше інформації про контекст запиту до пам'яті, який є на рівні програми. Це робить більш складним обмеження використання більш повільного типу пам'яті лише для окремих структур даних. Також це робить неможливими різні оптимізації які базуються на додатковій інформації яка доступна на рівні програми (наприклад, до якого саме елементу масиву чи хеш-таблиці відбувається доступ).

Через це найбільш привабливим варіантом інтеграції є абстракція у коді програмного забезпечення, яка б слугувала обгорткою над даними, що зберігаються у віддаленій пам'яті і при цьому дозволяла б працювати з ними так само, як це відбувається з даними у локальній оперативній пам'яті. Зручною реалізацією цього є реалізація розумного покажчика (smart pointer), який би надавався бібліотекою клієнта віддаленої пам'яті і використовується програмним забезпеченням у яке вона інтегрується. В цій роботі розглядається створення такої бібліотеки на мові програмування Rust [27]. Розумні покажчики у таких мовах як Rust чи C++ [28] імітують звичайні покажчики, при цьому дають додаткові можливості: реалізація такого покажчика може, наприклад, виконати додатковий код у момент, коли програма виконує доступ до нього.

Бібліотека клієнту віддаленої пам'яті, яка розробляється в цій роботі, надає розробникам прикладного програмного забезпечення розумний покажчик. При створенні цього покажчика через конструктор передається об'єкт, розміщення пам'яті якого с цього моменту керується клієнтом віддаленої пам'яті.

Використання бібліотеки віддаленої пам'яті є найбільш оптимальним у програмному забезпеченні написаному на мові програмування Rust. Створення інтеграцій для інших мов програмування знаходиться за межами того, що рогляадається в цій роботі. Програмне забезпечення що написано на інших мовах програмування може використовувати біндинги (bindings) для того, щоб користуватись реалізацією цієї бібліотеки з мов програмування що підтримують створення таких біндингів. Недоліком цього підходу є необхідність додаткових змін і менш зручний інтерфейс використання.

При створенні розумного покажчика (FarMemory<T>), він прив'язується до одного з проміжків пам'яті (span). Приклад використання розумного покажчика для зберігання даних у віддаленій пам'яті показано на рисунку 2.2. В залежності від розміру об'єкта, створюється або новий проміжок з необхідним розміром, або об'єкт разом з іншими об'єктами розміщується в межах одного проміжку, з різними значеннями зміщення відносно початку проміжку. Розумний покажчик FarMemory<T> зберігає або ідентифікатор проміжку (span) якщо це великий об'єкт, або ідентифікатор об'єкта якщо це малий об'єкт. У клієнті віддаленої пам'яті (FarMemoryClient) зберігається таблиця, для ключем є ідентифікатор об'єкту, а значенням - ідентифікатор проміжку та зсув у ньому. Необхідність цієї таблиці зумовлена тим, що вона дозволяє переміщувати об'єкти між проміжками у разі необхідності. Це потрібно наприклад для розміщення об'єктів доступ до яких зазвичай відбувається одночасно у межах одного проміжку для зниження кількості операцій переміщення проміжків між пристроями пам'яті. Це також робить можливим реалізацію дефрагментації (compaction) яка знижує загальну кількість проміжків, що використається програмним забезпеченням.

```rust
fn using_far_memory_in_application() {
  // this is how application data is normally accessed in application.
  let local_data: ApplicationData = ApplicationData::new();
  local_data.do_something_with_data();

  // application data placed in far memory, under FarMemory smart pointer. From now on, this data
  // is managed by far memory and swapped between local and remote memory transparently.
  let far_memory_data: FarMemory<ApplicationData> = FarMemory::new(inner: local_data);

  // when data is accessed, it is automatically moved into local memory.
  far_memory_data.do_something_with_data();

  // when data is not accessed, it may be moved back to remote memory.
}
```

Рисунок 2.2 – Приклад використання бібліотеки клієнту віддаленої пам'яті

Коли розумний покажчик отримає запит на доступ до даних (у мові програмування Rust це реалізується операцією Deref [29]), то він звертається до клієнту віддаленої пам'яті для отримання адреси проміжку пам'яті в оперативній пам'яті. Якщо потрібний проміжок в цей час вже знаходиться в оперативній пам'яті, то клієнт просто повертає його адресу (*mut u8). Якщо потрібного проміжку в пам'яті немає, то клієнт переміщує його з пам'яті пристроїв зберігання (бекенду) у локальну пам'ять, після чого повертається та адреса, де було розміщено цей проміжок пам'яті. Прикладне програмне забезпечення після

цього працює з даними що розміщені у цій адресі так само, як працювало б з будь-якими іншими даними у оперативній пам'яті.

Суть роботи клієнту віддаленої пам'яті полягає в тому, що проміжки пам'яті (spans) які не використовуються у певний момент часу програмним забезпеченням можуть бути переміщені частково або повністю у більш повільну пам'ять віддаленого пристрою (бекенду). Це створює необхідність відслідковувати які об'єкти та проміжки використовуються програмним забезпеченням у певний момент часу, а які - ні. Для цього можна використати наступний підхід: розумний покажчик FarMemory<T> після операції розіменування (dereferencing - функція deref у Rust) повертає інший розумний покажчик (FarMemoryLocal<T>), який у результаті розіменування повертає вже сам об'єкт T. Цей другий розумний покажчик потрібен для контролю використання даних з віддаленої пам'яті. Мова програмування Rust та механізм borrow checking у ній гарантує що посилання на T (&T) не буде більше використовуватись після того, як розумний покажчик FarMemoryLocal<T> вийшов з зони видимості. Це дає можливість використовувати метод підрахунку посилань для проміжків пам'яті (spans). При розіменуванні першого покажчика (FarMemory<T>) лічильник посилань для проміжку пам'яті збільшується. При виході з видимості другого розумного покажчика (FarMemoryLocal<T>) - лічильник посилань зменшується. Клієнт віддаленої пам'яті переміщує проміжки пам'яті для зберігання у бекенд тільки коли лічильник посилань дорівнює нулю.

При переміщенні об'єкту за покажчик у віддаленій пам'яті виникає проблема його перетворення на послідовність байт для представлення у пам'яті та передачі мережею чи запису на диск. Найбільш простим методом зробити це є взяти адресу у якій розміщено об'єкт T та перемістити у віддалену пам'ять з тієї адреси стільки байт, скільки є розміром типу T. Цей підхід використовується у FarMemory<T> і для більшості сценаріїв використання цього методу достатньо. Але є структури даних для яких використання такого підходу не є оптимальним. Це, наприклад, структури які мають покажчики на інші структури у пам'яті. Якщо перекопіювати дані з пам'яті де знаходиться ця структура даних, то

скопіюється не значення поля, а покажчик на структуру даних, яка там знаходиться. Для користувача зазвичай необхідно щоб об'єкт був переміщений у віддалену пам'ять повністю, з усіма вкладеними полями, так як це дозволяє звільнити максимальну кількість пам'яті. Вирішенням цієї проблеми є створення розумного покажчика (FarMemorySerialized<T>) який не копіює дані з пам'яті, а використовує механізм серіалізації. Для цього на тип T накладається вимога реалізувати ознаки (trait) Serialize та Deserialize з бібліотеки serde [30]. Під час роботи десеріалізація відбувається у момент доступу до даних за розумним покажчиком. Негативним наслідком цього підходу є додаткове використання ресурсів, яке викликане цим. Для операцій серіалізації та десеріалізації використовується бібліотека bincode [31], так як для більшості типів вона копіює дані з пам'яті без змін, уникаючи зайвих операцій. Іншими словами, цей підхід максимально близький до рекурсивного обходу структури даних.

Можливість розміщувати об'єкти у віддаленій пам'яті за допомогою розумних покажчиків може бути достатньо для багатьох випадків, але у прикладному програмному забезпеченні часто використовуються так звані "великі буфери" що створюються програмою за допомогою алокатора, та використовуються як великий послідовний обсяг пам'яті для збереження різних даних в оптимальному вигляді для конкретної задачі. При цьому такий буфер надає можливість читання та запису байтів з довільного зсуву та довжини. Для того, щоб зробити роботу з таким буфером більш оптимальною у випадку віддаленої пам'яті, у бібліотеку клієнту віддаленої пам'яті додається FarMemoryBuffer. Він має такий самий інтерфейс та зберігає дані розбиваючи їх на необхідну кількість проміжків (spans). Кожен проміжок має однаковий розмір вказаний користувачем або 2 мегабайти за замовчуванням. Таким чином, лише частина буфера знаходиться у локальній пам'яті (ті, до яких відбувається доступ), тоді як всі інші дані - у віддаленій. Це дозволяє знизити використання пам'яті та створювати буфери для зберігання даних розмір яких перевищує кількість оперативної пам'яті обчислювального вузла, що є дуже зручним у відповідних сценаріях використання. Цей тип можна розглядати як аналогічний звичайній

області пам'яті отриманій від алокатора та використовувати розробниками як основу при реалізації власних структур даних, адаптованих під використання віддаленої пам'яті.

Крім цього, FarMemoryBuffer як і деякі інші реалізації структур даних вирішують одну з проблем використання розумних покажчиків: у випадку великого об'єкта вони надають можливість розміщувати у локальній оперативній пам'яті лише його частину. Ця особливість надається завдяки тому, що доступ до частини даних надається через визначений інтерфейс, де є контекст до якої саме частини даних відбувається доступ, на відміну від покажчиків, коли прикладному програмному забезпеченню надається адреса у пам'яті та довжина даних і клієнту віддаленої пам'яті з цього моменту невідомо як саме програма працює з даними у цій ділянці оперативної пам'яті.

Розвиваючи цю ідею далі, у бібліотеку клієнту віддаленої пам'яті було додано реалізацію вектора (Vec<T> у Rust) адаптованого для роботи з віддаленою пам'яттю: FarMemoryVec<T>. Він передбачає доступ як до частини даних, так і до усіх даних одночасно перетворивши їх на звичайний вектор (Vec<T>) за допомогою розумного покажчика. Особливістю цієї реалізації є те, що вона використовує один проміжок пам'яті (span). В деяких сценаріях використання (наприклад великий масив, доступ до якого буде відбуватися завжди у вигляді доступу до конкретного елементу або невеликої частини масиву), більш оптимальним є використання вектору який розбиває дані на багато невеликих проміжків, тому для таких цілей клієнтом надається FarMemoryBufferedVec<T>.

Крім цього, структури даних адаптовані для роботи з віддаленою пам'яттю дозволяють більш ефективно реалізувати деякі операції. Наприклад, це дає можливість для вектору реалізувати ітератор, що повертає елементи з урахуванням того, в якому класі пам'яті вони знаходяться. Повертання в першу чергу тих елементів що знаходяться в локальній пам'яті дозволяє значно підвищити ефективність ітерації при задачах де порядок елементів не має значення (наприклад, при виконанні пошуку) за рахунок меншої кількості

блокувань основного потоку виконання шляхом зменшення зайвих переміщень даних між локальною пам'яттю та пам'яттю віддалених вузлів.

Альтернативним підходом для інтеграції віддаленої пам'яті у тих випадках, коли будь-які зміни в код програмного забезпечення не є бажаними є інтеграція за допомогою механізму підкачки у операційній системі Linux. Як було зазначено раніше, у цього підходу є недоліки у вигляді меншої гнучкості та додаткового контексту, який можна було б використати для підвищення швидкодії віддаленої пам'яті. Для використання віддаленої пам'яті для підкачки у Linux використовується підхід подібний тому, що використовують деякі існуючі реалізації методів надання віддаленої пам'яті (наприклад InfiniSwap [32]), а саме реалізацію клієнтом віддаленої пам'яті віртуального блокового пристрою [33]. Користувач налаштовує розміщення розділу підкачки у операційній системі Linux на цьому пристрої. В результаті, сторінки пам'яті які операційна система вважає рідко вживаними переміщуються з основної пам'яті у розділ підкачки (тобто, в цьому випадку у віддалену пам'ять) у випадку коли використання локальної оперативної пам'яті досягає високого рівня. В такому режимі роботи кожен блок віртуального блокового пристрою відповідає проміжку віддаленої пам'яті і тому лише частина блоків цього пристрою знаходяться в локальній пам'яті.

2.3 Забезпечення відмовостійкості

Однією з проблем використання віддаленої пам'яті є те, що зберігання даних на інших пристроях має значний негативний вплив на розмір домену збою (failure domain). Тобто якщо зазвичай ключову роль у відмовостійкості програмного забезпечення грає деяка ймовірність що вузол обчислення вийде з ладу, то при використанні віддаленої пам'яті додається ймовірність що SSD диск (якщо він використовується як бекенд для зберігання даних) чи віддалений вузол (у випадку коли дані розміщуються на віддалених вузлах) вийде з ладу, що приведе до втрати частини даних. У випадку віддалених вузлів ризик збою стає значно високим, тому що з великою кількістю вузлів для втрати даних достатньо

програмного чи апаратного збою на будь-якому з них, тобто ризик втрати даних збільшується пропорційно.

На жаль, не є можливим знизити ризик втрати даних при використанні віддаленої пам'яті до того ж самого рівня, як коли використовується тільки локальна оперативна пам'ять. Але, можна мінімізувати ризик події коли дані будуть втрачені без можливості відновлення до такого рівня, який буде прийнятним для більшості сценаріїв використання на практиці.

Найбільш простим методом забезпечення відмовостійкості є для кожного проміжку пам'яті (span) що було переміщено у пам'ять віддаленого вузла, зберігати ще одну копію даних на локальному SSD диску. Недоліком цього підходу є високий рівень затримки у разі відмови вузла зберігання: читання даних з диску є набагато більш повільним ніж отримання даних по мережі, тому у разі відмови це негативно вплине на швидкодію програмного забезпечення, у яке інтегрована віддалена пам'ять. Крім цього, це призводить до використання додаткового ресурсу - дискового простору, в результаті баланс між більш ефективним використанням оперативної пам'яті та додатковим використанням диску буде мати негативний вплив на розмір простору можливого застосування віддаленої пам'яті на практиці. Використання дискового простору можна зменшити використовуючи алгоритми стискання даних, такі як zstd, але це зробить систему більш повільною (найбільш швидкі алгоритми стискання даних обробляють на рівні сотень мегабайт у секунду) та збільшить використання ресурсів процесору, що не є бажаним при використанні віддаленої пам'яті. Оскільки можуть існувати середовища, де такий підхід є вигідним, то в реалізацію, яка розглядається в цій роботі, додано LocalDiskBackend який у поєднанні з ReplicationBackend (який забезпечує реплікацію на декілька бекендів) можна використовувати для реалізації цього підходу.

Іншим методом є реплікація за фактором n, тобто збереження декількох (n) копій даних на різних вузлах зберігання. Цей підхід на відміну від попереднього забезпечує кращий рівень швидкодії у разі відновлення даних при відмові одного з вузлів. Недоліком цього методу є високий рівень використання ресурсів по

кластеру: пам'яті використовується в n разів більше. В цій роботі цей метод забезпечення відмовостійкості реалізовано за допомогою ReplicationBackend, який може бути за конфігурацією користувача використаний разом з NetworkNodeBackend.

Найбільш ефективним методом забезпечення відмовостійкості є кодування стиранням (erasure coding) [34]. У цій роботі розглядається кодування за допомогою коду Ріда-Соломона, який є поширеним методом виправлення помилок у блоках даних. Суть цього методу полягає в тому, що дані поділяються на N частин та M додаткових, які обчислюються через лінійне перетворення. При втраті будь-яких M частин дані можна відновити через зворотнє лінійне перетворення. N та M задаються користувачем, приклади значень які можна взяти: (3, 2), (4, 3) та ін. - в залежності від середовища та вимог. Перевагою цього методу є швидке відновлення даних (так як час відновлення з пам'яті інших вузлів є нижчим за час відновлення з більш повільних пристроїв таких як SSD диски) та відносно невелике використання додаткової пам'яті за рахунок того, що додаткові частини даних мають у сумі менший розмір ніж розмір додаткової копії даних у разі використанні реплікації. В цій роботі це реалізовано у ErasureCodingBackend, який виконує кодування, а для власне зберігання даних передає шарди (частини) даних до існуючих реалізацій бекендів (наприклад NetworkNodeBackend).

На рисунку 2.3 схематично показано підхід з використанням кодів Ріда-Соломона для розділення даних на частини, що зберігаються на різних вузлах.

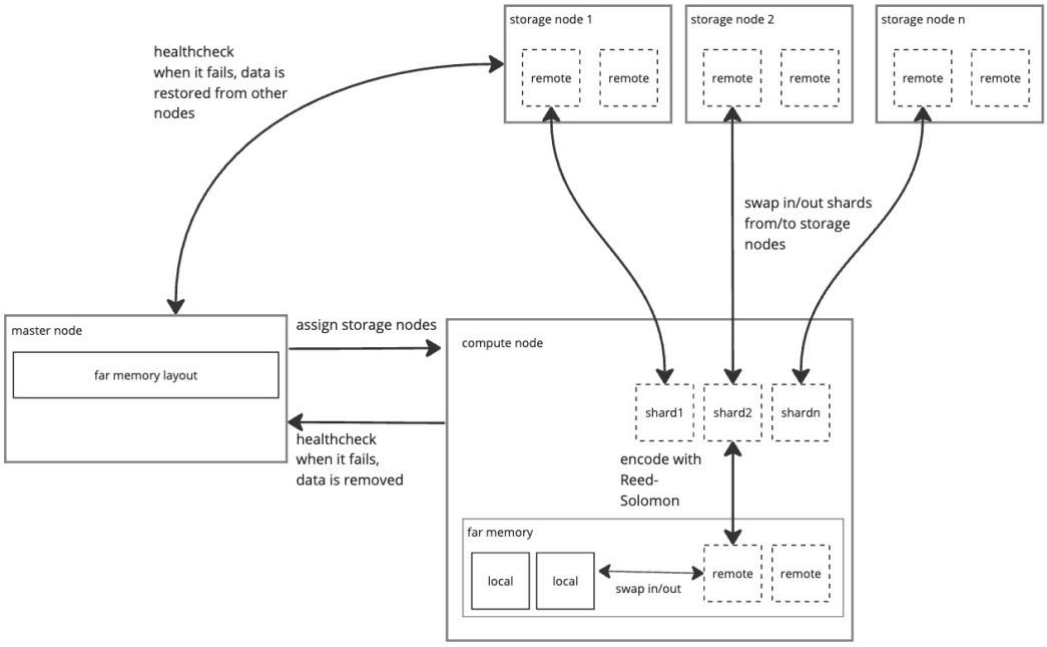


Рисунок 2.3 – Схематичне зображення кодування проміжків кодами Ріда-Соломона та зберігання частин даних на різних вузлах

2.4 Забезпечення швидкодії віддаленої пам’яті

Головним недоліком використання віддаленої пам'яті є негативний ефект на швидкодію програмного забезпечення, у яке вона інтегрована. Погіршення швидкодії виникає через більш повільний доступ до пам'яті який зумовлений в першу чергу використанням пристроїв зберігання (таких як пам'ять інших вузлів, доступ до якої виконується за мережею, дисків, та ін.) затримка (latency) та пропускна здатність (throughput) яких є значно більшою за такі параметри для оперативної пам'яті. Крім цього, під час доступу до проміжку даних у віддаленій пам'яті виконуються додаткові перевірки та дії, які не є потрібними при роботі зі звичайною пам'яттю.

Неможливо зробити доступ до віддаленої пам'яті таким саме швидким як доступ до оперативної пам'яті (у умовах, коли пристрій зберігання є повільним). Але має сенс мінімізувати затримку доступу до даних до того рівня, який є прийнятним для використання на практиці. Існує баланс між тим наскільки активно віддалена пам'ять використовується програмним забезпеченням (скільки даних та якого типу в ній зберігається) та негативним ефектом на швидкодію програмного забезпечення. Доцільність використання віддаленої

пам'яті та параметрів її роботи є відповідальністю розробника прикладного програмного забезпечення. Конфігурація віддаленої пам'яті обирається розробником базуючись на вимогах щодо швидкодії програмного забезпечення, його особливостях роботи з пам'яттю та характеристиках апаратного забезпечення (наприклад, швидкість мережі).

Як було зазначено раніше, кожен проміжок (span) у віддаленій пам'яті може в певний момент часу знаходитись у локальній пам'яті чи пам'яті віддаленого вузла (бекенду). В той час як доступ до даних у локальній пам'яті майже не відрізняється від часу доступу до даних у оперативній пам'яті (зазвичай декілька мікросекунд), час доступу до даних у пам'яті віддалених вузлів є значно більшим (десятки-сотні мілісекунд). Цей фактор швидкодії віддаленої пам'яті схематично показано на рисунку 2.4. Коли програмне забезпечення запитує дані які знаходяться у віддаленій пам'яті, то доки вони не будуть переміщенні у локальну пам'ять, виконання програми не можна продовжувати. Це блокування є головною причиною негативного впливу на швидкодію. Для зниження часу доступу до даних, кількість а також довжину таких блокувань потрібно зменшувати.

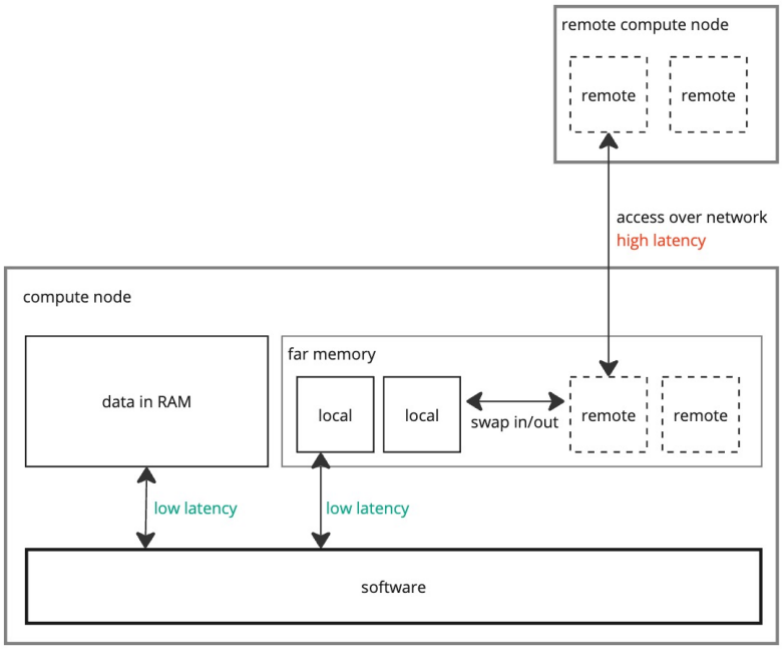


Рисунок 2.4 – Рівень часу доступу до даних в залежності від їх розміщення

На швидкість переміщення даних між пам'яттю віддаленого вузла та локальною пам'яттю в першу чергу впливає швидкість переміщення даних мережею. Через це важливо ефективно використовувати ресурси мережевого обладнання. Для передачі даних було розроблено легкий бінарний протокол, який працює поверх TCP. В сучасному програмному забезпеченні для передачі даних часто використовуються протоколи віддаленого виклику процедур (remote procedure call), такі як gRPC [35] чи HTTP [36] запити. Використання таких протоколів не є оптимальним для клієнту віддаленої пам'яті, оскільки їх реалізації покладаються на складні алгоритми серіалізації запитів, що виконують зайві копіювання та перетворення даних. Розробка легкого протоколу поверх TCP дозволяє уникнути цих недоліків високорівневих протоколів. TCP був обраний так як цей протокол на відміну від UDP [37] забезпечує надійність та реалізовує необхідні механізми для максимізації ефективності використання каналу передачі даних. QUIC [38] як альтернатива TCP в цій роботі не розглядається оскільки реалізації з використанням TCP вже достатньо для того, щоб швидкість передачі даних була близькою до пропускної здатності мережі, через що немає причин ускладнювати систему використанням QUIC.

Для того, щоб знизити затримку (latency) передачі даних по TCP, клієнт віддаленої пам'яті не використовує алгоритм Нейгла (Nagle's algorithm) [39]. Його задачею є підвищення швидкодії мережі через уникання невеликих пакетів даних під час передачі. Додаткове очікування наступних пакетів не має сенсу, оскільки клієнт віддаленої пам'яті пише увесь запит за раз і навіть якщо він невеликий (наприклад, запит на отримання проміжку пам'яті), клієнт очікує відповідь одразу після запису запиту. Алгоритм Нейгла відключено через використання опції nodelay під час налаштування TCP сокету.

Для того, щоб знизити час обробки при декількох послідовних запитах (наприклад, переміщення декількох проміжків одночасно), клієнт віддаленої пам'яті може об'єднувати декілька запитів у один (batching) для більш ефективної обробки. Це дозволяє уникнути завершення попереднього запиту для надсилання наступного.

В програмному забезпеченні, швидкодія якого залежить від швидкодії мережі для зниження затримки часто використовується підхід реалізації роботи з мережею у просторі користувача (userspace networking [40] або kernel bypass networking). В тому числі цей підхід використовується у реалізації віддаленої пам'яті AIFM. В цій роботі kernel bypass networking не використовується, тому що зниження затримки (latency) передачі пакетів незначно вплине на час доступу до даних, тому що під час переміщення великих проміжків пам'яті обмежуючим фактором в першу чергу є пропускна здатність мережі (throughput). Використання kernel bypass networking також прив'яже реалізацію до конкретних моделей мережевих карт, що суперечить одній з вимог до системи. Отже, використання цього підходу та теоретично відносно невелике покращення швидкодії не є того вартим.

Слід зазначити, що сучасне мережеве обладнання зазвичай підтримує одночасну передачу і отримання даних на максимальній швидкості, тобто підтримує двосторонню (duplex) передачу. Наприклад, стандартом 10 Gigabit Ethernet [41] це є єдиним режимом з'єднання який визначено, тобто програмне забезпечення може одночасно передавати та отримувати дані на швидкості 10 Гігабіт у секунду. У цій роботі, ця можливість апаратного забезпечення використовується клієнтом віддаленої пам'яті для одночасного переміщення сторінок у пам'ять віддалених вузлів та з неї. Це дозволяє максимально ефективно використовувати канал з'єднання.

Для того, щоб зменшити кількість зайвих перетворень та копіювань даних, дані проміжків пам'яті передаються мережею окремо від тіла запиту чи відповіді. Тіло запита чи відповіді містить поле, яке зберігає довжину проміжку пам'яті. Це прискорює серіалізацію запитів, оскільки тіло запиту стає значно меншим. Зайве копіювання (яке зазвичай займає порівняно великий проміжок часу) уникається, так як читання даних проміжку пам'яті з TCP сокету виконується напряму у ту адресу оперативної пам'яті де дані після цього будуть зберігатися. Для серіалізації запитів обрано формат bincode, так як це одна з найбільш швидких реалізацій серіалізацій у мові програмування Rust.

Часто використання стиснення даних дозволяє знизити час передачі даних по мережі за рахунок використання додаткових ресурсів процесору для зменшення кількості даних, які потрібно передати. Це є ефективним при обмежених ресурсах пропускної здатності мережі. Однак, швидкість обробки найбільш швидких сучасних алгоритмів стиснення даних (таких як zstandard [42], snappy [43] чи lz4 [44]) на сучасному обладнанні становить близько 800 Мегабайт у секунду (6,4 Гбіт у секунду), що у більшості випадків є меншим за пропускну здатність мережі (у центрах обробки даних пропускна здатність мережі між серверами зазвичай становить 10 Гбіт/сек та вище). Це робить використання стиснення даних недоцільним. Через це за замовчуванням клієнт віддаленої пам'яті не використовує стиснення, але залишає можливість його опціально увімкнути користувачу (використовується алгоритм lz4).

З такої самої причини (повільна обробка), за замовчуванням шифрування даних не виконується.

Клієнт віддаленої пам'яті перевіряє рівень вільної локальної пам'яті під час переміщення проміжків з пам'яті віддалених вузлів. Можливо що для переміщення даних потрібно додатково звільнити певний обсяг локальної пам'яті через переміщення проміжків у зворотньому напрямку для зниження використання локальної пам'яті. Розміри проміжків пам'яті можуть перевищувати розмір пам'яті, яку потрібно звільнити. Це робить переміщення таких проміжків неефективним: переміщується більше даних ніж потрібно. Це можна було б вирішити комбінуванням невеликих проміжків пам'яті, які б в сумі були близькими до того обсягу пам'яті, який потрібно звільнити. Але це теж не є ефективним: це вимагає вирішення задачі рюкзака, що потребує значних обчислень, при цьому вільних (тобто тих, які не використовуються програмою та не будуть використані у найближчий час) невеликих проміжків може не бути. Через це, в цій роботі реалізовано часткове переміщення проміжків пам'яті: дані можуть бути розділені між локальною пам'яттю та пам'яттю віддалених вузлів у будь-якій пропорції, яка може змінюватись за необхідністю. Це дозволяє переміщувати у віддалену пам'ять рівно стільки даних, скільки потрібно щоб

звільнити локальну пам'ять до достатнього рівня. У локальній пам'яті залишається частина з початку проміжку, у віддаленій - з кінця, це робить повторні операції часткового переміщення більш дешевими за рахунок того, що дані не потрібно копіювати або зсувати. В більшості випадків достатньо виклику алокатора, який зменшує чи збільшує розмір цієї ділянки пам'яті. Зберігання частини даних також залишає можливість обмежувати час доступу до великого проміжку пам'яті у гіршому випадку (частина даних що вже є в локальній пам'яті зменшує кількість даних що потрібно перемістити та відповідний час на це). Іншими словами, переміщення частин великої кількості проміжків замість повного переміщення невеликої кількості проміжків дозволяє знизити час доступу у гіршому випадку (tail latency) за рахунок більшої кількості операцій переміщення проміжків.

Як було зазначено раніше, невеликі об'єкти вимагають розташування декількох об'єктів у межах одного проміжку пам'яті, так як створення великої кількості малих проміжків підвищує витрати ресурсів необхідних для керування віддаленою пам'яттю. При цьому виникає проблема, схожа на ту, з якою стикаються алокатори пам'яті: фрагментація пам'яті призводить до неефективного використання ресурсів. У випадку віддаленої пам'яті це не тільки призводить до зайвого використання пам'яті, а й збільшує кількість операцій доступу до проміжків пам'яті та їх переміщення, що має негативний ефект на швидкодію. Для вирішення цієї проблеми у цій роботі використовується підхід класів розміру (size classes), подібний до того, що використовується у Carbink чи TCMalloc. Цей підхід полягає у тому, що кожен з проміжків зберігає об'єкти одного розміру (можливі розміри задані завчасно, при додаванні об'єкту його розмір округляється до найближчого класу розміру). Коли об'єкт прибирається з віддаленої пам'яті, то на його місці можна розмістити об'єкт такого ж розміру. В результаті, фрагментація знижується і переміщення об'єктів між локальною та віддаленою пам'яттю становиться більш ефективним.

Незважаючи на важливість швидкої передачі проміжків по мережі та зниження фрагментації, ключем до забезпечення низької затримки є розміщення

проміжків пам'яті таким чином, щоб потрібні дані в найбільшій кількості випадків знаходились у локальній пам'яті, доступ до якої є швидким. Як зазначалось раніше, важливим є зменшення часу, коли програмне забезпечення блокується очікуючи даних по мережі. Одним зі способів як цього досягти є створення фонового потоку виконання, який виконує переміщення проміжків пам'яті уникаючи блокування основного потоку виконання програмного забезпечення. Цей потік у циклі слідкує за рівнем використання локальної пам'яті та у разі якщо він є високим, переміщує проміжки у пам'ять віддалених вузлів. Це дозволяє основному потіку виконання не витрачати час на звільнення пам'яті, а одразу перемішувати проміжки з пам'яті віддалених вузлів у локальну пам'ять. Крім цього, фоновий потік завчасно переміщує проміжки до яких з високою ймовірністю буде виконано доступ у найближчий час (для вибору проміжків використовується алгоритм заміщення проміжків, що буде розглянуто більш детально далі). Аналогічно, це дозволяє уникнути блокування основного потоку для переміщення даних у локальну пам'ять. Якщо припустити що фоновий потік працює максимально швидко та обирає проміжки для переміщення максимально правильно, то основний потік не буде блокуватися ніколи, що наблизить швидкодію програмного забезпечення до рівня що є максимально близьким до того, коли віддалена пам'ять не використовується.

Легко помітити, що метод вибору проміжків пам'яті для переміщення між локальною пам'яттю та пам'яттю віддалених вузлів має значний вплив на ефективність роботи віддаленої пам'яті. Якщо у локальну пам'ять переміщуються саме ті проміжки, доступ до яких очікується в першу чергу, а на зберігання у віддалені вузли передаються проміжки, які у найближчий час виконання не будуть потрібні, то зменшується кількість блокувань для очікування переміщення потрібних проміжків. Як зазначалось раніше, зменшення кількості блокувань покращує швидкодію.

Таке формулювання вказує на те, що вибір проміжків пам'яті для переміщення є відомою задачею заміщення сторінок (page replacement algorithm) [45]. Ця задача формулюється як задача керування пам'яттю комп'ютера, у якій

потрібно обирати яка сторінка буде переміщена у більш повільну пам'ять з більш швидкої замість тієї сторінки пам'яті на яку надійшов запит. В схожому вигляді ця задача вирішується, наприклад, у операційних системах: операційній системі потрібно обирати які сторінки пам'яті будуть переміщенні у файл підкачки у першу чергу коли потрібно звільнити оперативну пам'ять.

У випадку клієнта віддаленої пам'яті ця задача, як зазначалось раніше, розширюється також тим, що необхідно обирати проміжки пам'яті які навпаки будуть завчасно переміщенні з більш повільної пам'яті у більш швидку. Вхідними даними є інформація про проміжки пам'яті до яких відбувався доступ під час поточного запуску програми. Кожен з проміжків ідентифікується номером, як зазначалось раніше. Надається інформація також про те, які проміжки пам'яті знаходяться у локальній пам'яті, а які - у пам'яті віддалених вузлів. Базуючись на цих даних, реалізація алгоритму заміщення сторінок (у коді - ReplacementPolicy) обробляє запити на вибір проміжка для переміщення з локальної пам'яті до віддалених вузлів і навпаки. Цей компонент може зберігати внутрішній стан між запитами.

На рисунку 2.5 показано використання алгоритму заміщення проміжків для вибору проміжків для переміщення між локальною пам'яттю та пам'яттю віддалених вузлів (сам алгоритм схематично позначено символом f). Як можна побачити з рисунку, алгоритм заміщення проміжків є частиною процесу переміщення проміжків між типами пам'яті та використовується при кожній такій операції.

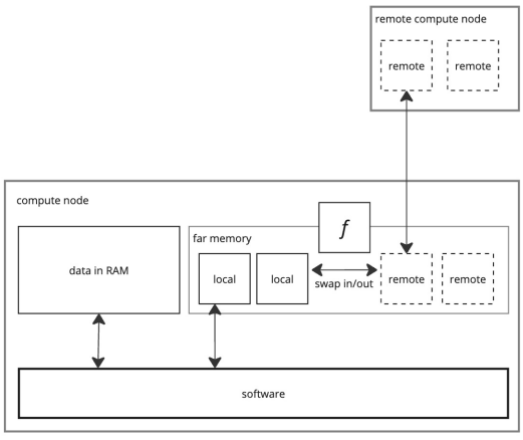


Рисунок 2.5 – Використання алгоритму заміщення проміжків

Для збору статистики про доступ про проміжків пам'яті клієнт віддаленої пам'яті повідомляє алгоритму заміщення сторінок про кожен випадок розіменування розумного покажчика, за яким зберігаються дані у віддаленій пам'яті. Окремо також повідомляється про кожне переміщення проміжків між типами пам'яті. В залежності від реалізації (в цій роботі розглядається декілька), цей компонент може передавати статистику в агрегованому вигляді на вузол керування для збереження статистики між запусками програми, агрегування даних між декількома обчислювальними вузлами та отримувати оновлені параметри моделі прогнозування що використовується алгоритмом заміщення проміжків.

Найбільш проста реалізація алгоритму заміщення сторінок яка розглядається це заміщення випадковим чином (RandomReplacementPolicy). Це дуже простий алгоритм, але він є зручним для встановлення базового рівня для подальшого аналізу. На кожен запит на вибір проміжку для переміщення у віддалену пам'ять, цей алгоритм випадковим чином обирає проміжок серед усіх які знаходяться у локальній пам'яті. Недоліком цього методу є те, що він обирає проміжки з однаковою ймовірністю, що приводить до того, що проміжки які будуть використані найближчим часом під час подальшого виконання програмного забезпечення переміщуються так само, як і проміжки які не будуть потрібні. Це призводить до великої кількості випадків коли потрібного проміжку немає в локальній пам'яті, що погіршує швидкодію програмного забезпечення.

Альтернативою є використання алгоритму заміщення найменш нещодавно використаних (least recently used [46]) проміжків (LeastRecentlyUsedReplacementPolicy). Цей алгоритм часто використовується в операційних системах для заміщення сторінок під час використання файлу підкачки, а також у системах кешування - в багатьох випадках використання цього алгоритму є виправданим. Так само цей алгоритм може бути виправданим при інтеграції віддаленої пам'яті у програмне забезпечення яке має закономірності доступу до пам'яті які роблять цей алгоритм ефективним: програмне забезпечення, у якому до деякої частини даних приходить набагато

більше запитів ніж до інших даних. Легко помітити неефективність цього алгоритму для програмного забезпечення яке регулярно виконує доступ до проміжків пам'яті у певному порядку. Прикладами програмного забезпечення з цього класу може бути програмне забезпечення яке виконує обчислення де значний обсяг даних використовується у певному порядку, наприклад, нейронні мережі чи бази даних які виконують сканування таблиці у певному порядку. Для такого програмного забезпечення під час використання віддаленої пам'яті алгоритм заміщення найменш нещодавно використаних проміжків буде призводити до низького рівня швидкодії. Це пояснюється тим, що у віддалену пам'ять будуть переміщатися саме ті проміжки, доступ до яких буде відбуватися в першу чергу під час подальшого виконання програми.

Очікувано, що поки алгоритм заміщення найменш нещодавно використаних проміжків є неефективним для зазначеного вище класу програмного забезпечення, "зворотній" алгоритм є ефективним для нього. Заміщення найбільш нещодавно використаних (most recently used) проміжків (MostRecentlyUsedReplacementPolicy) дозволяє отримати низький рівень випадків коли потрібного проміжку немає в локальній пам'яті, що має позитивний ефект на швидкодію програмного забезпечення. Так само, недоліком цього алгоритму є його неефективність для програмного забезпечення яке має інші закономірності роботи з пам'яттю.

На рисунку 2.6 наведено приклад простої закономірності доступу до проміжків пам'яті у типового програмного забезпечення, де два проміжки (292 та 291) мають властивість декілька разів зустрічаються послідовно. Алгоритм заміщення проміжків міг би використовувати такі закономірності для більш ефективної роботи, наприклад в цьому випадку перемістивши проміжок з ідентифікатором 291 одразу після того, як програмне забезпечення запросило доступ до проміжку з ідентифікатором 291.

Рисунок 2.6 – Приклад закономірностей доступу до пам'яті у програмного забезпеченні

Оскільки у прикладному програмному забезпеченні закономірності роботи з пам'яттю можуть бути складними та залежати від багатьох факторів, то пошук алгоритму заміщення проміжків оптимального для задачі може бути складним для розробників, які використовують віддалену пам'ять у своєму програмного забезпеченні. Це створює необхідність у алгоритмах заміщення проміжків, які можуть адаптуватися під особливості роботи програмного забезпечення з пам'яттю використовуючи статистику доступу до проміжків пам'яті.

Такий тип алгоритмів заміщення проміжків працює за наступним принципом. Під час роботи програмного забезпечення, клієнт віддаленої пам'яті повідомляє реалізації алгоритму заміщення проміжків про кожен випадок доступу до проміжку пам'яті одразу під час кожного доступу до проміжку, надаючи інформацію про ідентифікатор проміжку. Доступ до проміжку віддаленої пам'яті відбувається під час розіменування розумного покажчика на об'єкт у віддаленій пам'яті (такий як FarMemoryLocal<T>) чи при роботі зі структурою даних адаптованої для роботи з віддаленою пам'яттю (наприклад операція get для FarMemoryHashMap<K, V>). Ця інформація в незмінному чи агрегованому вигляді (в залежності від конкретного алгоритму заміщення

проміжків що використовується) безперервно передається з усіх вузлів обчислення на вузол керування. На вузлі керування, ця інформація зберігається та неперервно обробляється у фоновому потоці виконання. Зібрана статистика доступу до пам'яті з усіх вузлів використовується для адаптації параметрів моделі прогнозування. Цей процес може займати значний час, тому адаптація параметрів відбувається у фоновому потоці та не блокує роботу ні вузла керування, ні вузлів обчислень. Як тільки процес підбору параметрів завершується, то нові параметри моделі прогнозування надсилаються вузлам обчислення для використанні у алгоритмі заміщення проміжків. Після цього, вузол керування одразу починає нову ітерацію адаптації параметрів з використанням нової статистики і продовжує робити це у циклі весь час своєї роботи.

Процес збору статистики та її використання для адаптації параметрів моделі прогнозування що використовується алгоритмом заміщення проміжків схематично зображено на рисунку 2.7.

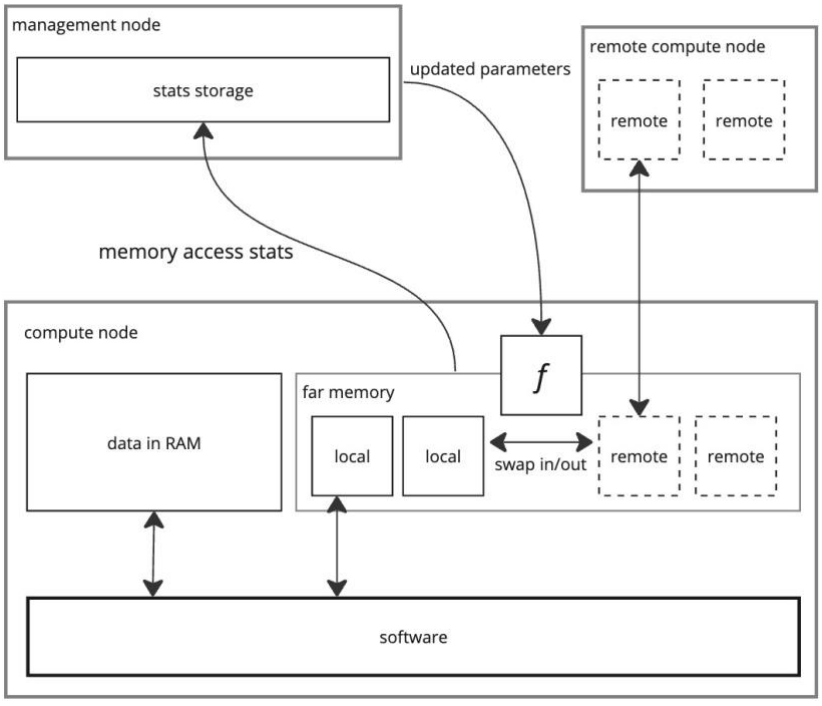


Рисунок 2.7 – Збір статистики доступу до проміжків та її використання для адаптації параметрів алгоритму заміщення

Окремо слід розглянути випадок коли до алгоритму заміщення проміжків надходять запити на вибір проміжків для заміщення, але параметри моделі

прогнозування ще не були отримані. Такий сценарій виникає на початку роботи віддаленої пам'яті. В такому випадку клієнт віддаленої пам'яті тимчасово використовує більш простий алгоритм заміщення проміжків обраний користувачем (наприклад, LRU). Статистика при цьому продовжується збиратись і як тільки будуть отримані параметри моделі прогнозування, то клієнт віддаленої пам'яті переключиться на використання алгоритму заміщення проміжків що базується на цій моделі.

Прогнозна модель (зображено на рисунку 2.8) що використовується такими алгоритмами заміщення проміжків приймає на вхід історію доступів до проміжків пам'яті. На виході така модель має значення що відповідають очікуваному часу доступу для кожного проміжку, з яким працює клієнт віддаленої пам'яті. Параметри моделі зберігаються у локальній оперативній пам'яті вузла обчислення та є однаковими для усіх вузлів обчислення. Прогнозування з використанням моделі відбувається на кожен запит до алгоритму заміщення проміжків на вибір проміжків для переміщення (як зазначалось раніше, це відбувається в основному потоці виконання при необхідності звільнення пам'яті або в фоновому потоці виконання для завчасного звільнення пам'яті або переміщення проміжків у локальну пам'ять).

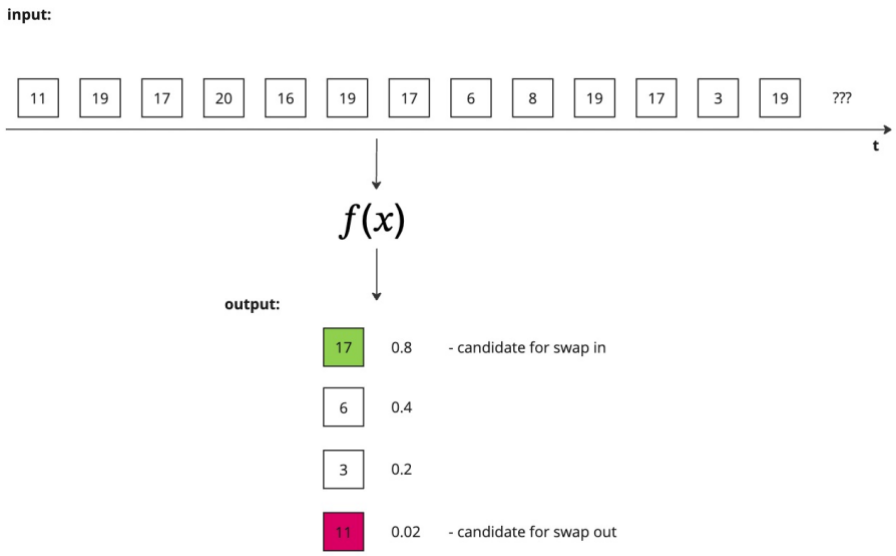


Рисунок 2.8 – Схематичне зображення принципу роботи моделі прогнозування доступу до проміжків

Одним з алгоритмів заміщення проміжків що розглядаються (та реалізовані) в цій роботі є алгоритм що базується на "оптимальній моделі" (використовує алгоритм Беладі [47]). Цей алгоритм заміщення проміжків неперервно збирає статистику доступу до проміжків та надсилає її на вузол керування, який в свою чергу зберігає статистику з усіх вузлів обчислень та попередніх запусків програмного забезпечення, серед зібраних даних обирає найдовшу послідовність та надсилає її усім вузлам обчислення як параметр для алгоритму Беладі. Алгоритм заміщення проміжків обирає зміщення в послідовності що відповідає часу роботи програмного забезпечення, та обробляє запити на заміщення проміжків наступним чином: для переміщення проміжків у віддалену пам'ять обираються ті проміжки, що знаходяться у локальній пам'яті та зустрічаються у послідовності в останню чергу у порівнянні з іншими проміжками відносно поточного зсуву. Аналогічно, для завчасного переміщення проміжків з віддаленої пам'яті у локальну обираються ті проміжки, які є найближчими у послідовності до поточного зсуву та знаходяться у пам'яті віддалених вузлів. Перевагою цього алгоритму заміщення проміжків є його простота у реалізації та використанні. Недоліком є те, що цей алгоритм є ефективним лише для програмного забезпечення що має постійні схеми доступу до пам'яті (наприклад, база даних що сканує увесь набір даних у однаковій послідовності кожного разу або нейронна мережа робота з вагами якої відбувається завжди однаковими чином).

Для ефективної роботи з програмним забезпеченням що має динамічні схеми доступу до пам'яті, в цій роботі пропонується використовувати (і реалізовано) алгоритм заміщення проміжків що базується на використанні рекурентної нейронної мережі (recurrent neural network [48]) як моделі прогнозування доступу до проміжків. Рекурентні нейронні мережі є розповсюдженим рішенням у задачах де необхідно обробляти послідовності даних. Заміщення сторінок у віддаленій пам'яті є такою задачею. Послідовністю в цьому випадку є події доступу до проміжків пам'яті. Як і у попередньому алгоритму заміщення проміжків, цей алгоритм неперервно збирає та надсилає на

вузол керування статистику доступів до проміжків. Вузол керування у фоновому потоці адаптації параметрів, агрегує статистику зібрану з усіх вузлів та попередніх запусків програмного забезпечення, розбиває на послідовності з заданою довжиною вікна (за замовчуванням, 100 останніх проміжків). Після цього, зібрана статистика використовується для тренування нейронної мережі LSTM [49] з лінійним шаром на виході розмір якого відповідає кількості проміжків, та має значення наступного проміжка до якого відбудеться доступ закодований унітарним кодом [50]. Після завершення тренування нейронної мережі, значення її вагів передаються на вузли обчислення для використання для прогнозування у алгоритмі заміщення проміжків. На кожен запит на переміщення проміжків у віддалену пам'ять, цей алгоритм обирає проміжок з найнижчим значенням коефіцієнту на виході нейронної мережі. Для завчасного переміщення у локальну пам'ять навпаки обираються проміжки з найвищим значенням.

Слід зазначити, що наведені алгоритми заміщення проміжків є лише демонстрацією використання статистики доступу до пам'яті що неперервно збирається для адаптації параметрів моделі прогнозування, що використовується у алгоритмі заміщення проміжків. Наступним розвитком алгоритмів заміщення проміжків може бути побудова скінченних автоматів, використання моделей глибинного навчання такі як трансформер [51] та інші моделі що можуть бути більш ефективними.

Висновки до розділу

В даному розділі було розглянуто проблему надання віддаленої пам'яті у розподілених інформаційних системах. Описано середовище та принцип роботи віддаленої пам'яті у ньому.

З урахуванням специфіки середовищ, для інтеграції в які призначена ця реалізація віддаленої пам'яті, визначено компоненти системи. Це вузли обчислення, вузли зберігання та вузол керування. Визначено які дані та як

передаються між цими вузлами, як виконується збір статистики про роботу кластеру та керування ним.

З формулювання проблеми що вирішується, середовища та структури компонентів випливають підзадачі, які необхідно розглянути для створення програмного засобу надання віддаленої пам'яті у розподіленій системі.

Розглянуто проблему інтеграції віддаленої пам'яті у програмне забезпечення. Запропоновано інтеграцію у програмне забезпечення за допомогою бібліотеки та віртуального блокового пристрою. Розглянуто підходи та структури даних які в визначених випадках роблять віддалену пам'ять більш ефективною.

Для забезпечення відмовостійкості, проаналізовано існуючі підходи до відновлення даних у разі відмови віддалених вузлів у існуючих реалізаціях віддаленої пам'яті. Зазначено їх переваги та недоліки. Підхід з використанням кодування стиранням як найбільш ефективний обрано для використання в цій роботі. Інші розглянуті методи надаються до використання опціонально, за конфігурацією від користувача.

Крім цього, розглянуто проблему забезпечення високого рівня швидкодії під час доступу до даних у віддаленій пам'яті. Визначено які фактори впливають на затримку доступу і принципи, на яких ґрунтуються підходи для її зниження. Описано особливості реалізації клієнта віддаленої пам'яті а також протоколу передачі даних, які є важливими для забезпечення високого рівня швидкодії. Обґрунтовано необхідність використання переміщення проміжків пам'яті у фоновому потоці, вплив алгоритму заміщення проміжків на ефективність роботи віддаленої пам'яті. Розглянуто можливі алгоритми заміщення проміжків, описано механізм збору статистики доступу до проміжків пам'яті та її використання для неперевної адаптації параметрів моделі прогнозування у більш ефективних алгоритмах заміщення проміжків.

Реалізація віддаленої пам'яті, яка розглядається в цій роботі, складається з методів вирішення кожної з підзадач, які було сформульовано та проаналізовано в цьому розділі.

3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вимоги до програмного забезпечення

Як було зазначено в попередніх розділах, до реалізації віддаленої пам'яті, що розглядається в цій роботі, висуваються вимоги які визначені розглянутими особливостями середовища та програмного забезпечення, у яке вона інтегрується.

Віддалена пам'ять повинна інтегруватися у програмне забезпечення за допомогою бібліотеки або віртуального блокового пристрою. У разі використання бібліотеки, розробнику прикладного програмного забезпечення повинен надаватись клієнт віддаленої пам'яті, який надає засоби для зберігання послідовностей байт, об'єктів наданих користувачем та структур даних оптимізованих для роботи з віддаленою пам'яттю.

Реалізація віддаленої пам'яті повинна коректно обробляти події виходу з ладу віддалених вузлів зберігання та підтримувати запланований вивід вузлів на обслуговування, що є типовою вимогою для програмного забезпечення що працює у розподіленій системі. У разі виходу вузла зберігання з ладу, клієнт віддаленої пам'яті повинен мінімізувати вірогідність втрати даних через їх відновлення з пам'яті інших вузлів.

Так як рівень швидкодії віддаленої пам'яі напряму впливає на доцільність її використання для різних типів програмного забезпечення, то має сенс визначити вимоги щодо часу доступу до даних у віддаленій пам'яті. Мінімальним рівнем що робить використання віддаленої пам'яті виправданим є той рівень, де час доступу до даних у віддаленій пам'яті є меншим ніж час доступу до даних такого самого розміру розміщених на дисковому сховищі у сучасній інфраструктурі з дізагрегованими ресурсами. Якщо віддалена пам'ять буде більш повільною у порівнянні, то її використання не є доцільним, так як програмне забезпечення буде працювати швидше у разі зберігання даних на диску.

До апаратної платформи, на якій розгортаються вузли зберігання, обчислення та керування висуваються наступні вимоги:

– процесор архітектури x86 або ARM з тактовою частотою не менше 1 ГГц, з одним ядром чи більше;

– оперативна пам’ять об’ємом не менше 1 Гб;

– вільний дисковий простір об’ємом не менше 10Гб (тільки для вузла керування);

– всі вузли мають доступ до всіх інших вузлів по мережі, можуть відкрити з’єднання та передавати та отримувати дані;

– пропускна здатність мережі не менше 100Мбіт/сек, затримка між вузлами - не більше 10 мілісекунд.

Усі вузли розгортаються на операційній системі Linux.

3.2 Засоби розробки

Для реалізації усіх компонентів віддаленої пам'яті було обрано мову програмування Rust. Rust є популярною мовою у сфері системного програмування та є ефективною для цієї задачі так як ця мова є компільованою, не має збирача сміття, використовує абстракції з нульовою ціною що дозволяє отримати рівень швидкодії що є подібним до рівня який забезпечують такі мови програмування як C чи C++. При цьому, Rust гарантує безпечну роботу з пам'яттю завдяки системі статичної перевірки посилань (borrow checker), що спрощує написання безпечного програмного забезпечення та паралельних обчислень. Крім цього, Rust є сучасною мовою програмування що підтримує функціональну парадигму програмування, має строгу типізацію, що спрощує розробку програмного забезпечення. Розвинена екосистема цієї мови дає можливість використовувати якісні бібліотеки для типових задач, де це необхідно.

Клієнт віддаленої пам'яті використовує декілька потоків виконання, то виникає проблема у їх синхронізації з використанням спеціалізованих структур даних та інструментів синхронізації виконання. Оскільки стандартної бібліотеки мови програмування Rust в деяких випадках може бути недостатньо, то

використовується бібліотека crossbeam [52] що містить додаткові інструменти для паралельного програмування.

В деяких компонентах виникає потреба у використанні асинхронних функцій (наприклад, у сервері зберігання та сервері керування). Це дозволяє спростити реалізацію коду, що паралельно працює з декількома мережевими з'єднаннями (чи іншими IO операціями), та зробити її більш ефективною у порівнянні з кодом що використовує окремі потоки рівня операційної системи. В мові програмування Rust для виконання асинхронних функцій потрібно використати середовище виконання. В цій роботі використовується tokio [53] який є найбільш популярною бібліотекою що надає середовище виконання та реалізації асинхронних функцій.

Для того, щоб мати можливість детально аналізувати роботу віддаленої пам'яті та шукати вузькі місця які вимагають оптимізацій необхідним є фреймворк інструментування програмування забезпечення з підтримкою трасування, оскільки звичайного логування недостатньо. У цій роботі для цього використовується бібліотека tracing [54]. У поєднанні з tracing-chrome [55] вона дозволяє зберігати інформацію про події під час роботи клієнту віддаленої пам'яті та аналізувати їх у вигляді діаграми з часовою шкалою за допомогою chrome developer tools [56].

Бібліотека vblk [57] використана для реалізації віртуального блокового пристрою. Для цього ця бібліотека взаємодіє з модулем NBD [58] у операційній системі Linux.

Для серіалізації даних використовується бібліотека serde [59] разом з bincode, що реалізує компактне кодування даних у набір байт. За замовчуванням serde працює неефективно з векторами байт (Vec<u8>): серіалізує кожен елемент окремо замість того, щоб скопіювати усю ділянку пам'яті за одну операцію. Для усунення цього недоліку використовується бібліотека serde-bytes [60].

Для реалізацій бекендів клієнта віддаленої пам'яті використані наступні бібліотеки: reed-solomon-erasure [61] (для кодування та відновлення даних

кодами Ріда-Соломона), aes-gcm [62] (для шифрування), lz4 [63] (для стиснення даних).

Деякі реалізації алгоритмів заміщення проміжків пам'яті, що розглядаються в цій роботі, використовують алгоритми машинного навчання, наприклад рекурентні нейронні мережі. Для цього використана бібліотека candle [64]. Перевагою використання цієї бібліотеки є те, що вона повністю реалізована на мові програмування Rust та не містить зовнішніх залежностей (в тому числі не використовує динамічні бібліотеки, такі як libtorch). Це дозволяє зробити процес зборки та розгортання програмного забезпечення більш простим, так як результатом компіляції є виконуваний файл з усіма залежностями залінкованими статично.

Бібліотека thiserror [65] використовується для простого визначення типів помилок, що використовуються у клієнті та іншних компонентах віддаленої пам'яті.

Для моніторингу використовується бібліотека prometheus, за допомогою якої важливі показники, такі як кількість та тип проміжків пам'яті, швидкість їх передачі відслідковуються у лічильних. Отримані значення передаються раз на 10 секунд у базу даних сумісну з Prometheus методом push.

Для того, щоб зробити розгортання системи більш простим для кінцевого користувача, збираються Docker зображення (в одному зображенні є всі компоненти віддаленої пам'яті, так як вони всі доступні у вигляді одного виконуваного файла що запускається з різними параметрами). Крім цього, реалізовані файли-визначення для Kubernetes, що дає можливість у разі необхідності розгорнути усі компоненти системи у Kubernetes кластері.

3.3 Архітектура програмного забезпечення

3.3.1 Компоненти програмного забезпечення що надає віддалену пам’ять

Як було зазначено раніше, цей метод надання віддаленої пам'яті використовує три компоненти: інтеграція у програмне забезпечення на стороні вузлів обчислення, вузли зберігання та вузол керування.

Схема структурна розгортання цих компонентів наведена у додатку А.

Інтеграція у програмне забезпечення представлена клієнтською бібліотекою або віртуальним блоковим пристроєм. Інтеграція налаштовується розробником інформаційної системи яка використовує віддалену пам'ять. У разі використання бібліотеки, цей компонент розгортається у кількості екземплярів рівній кількості екземплярів програмного забезпечення інформаційної системи. При використанні блокового пристрою - в залежності від кількості серверів на яких розміщена інформаційна система та кількості блокових пристроїв на кожному з них. Очікується що під час роботи інформаційної системи кількість вузлів обчислень може змінюватись.

Вузол керування та вузли зберігання розгортаються на інших серверах. Вузол керування завжди один, а кількість вузлів зберігання може змінюватись адміністратором інформаційної системи або планувальником задач, що використовується, в залежності від обсягу вільних ресурсів. На одному сервері може одночасно бути розгорнуто декілька вузлів зберігання.

3.3.2 Взаємодія компонентів

Схема структурна компонентів програмного забезпечення що надає віддалену пам'ять наведено у додатку Б. Ця схема окрім компонентів також показує зв'язки між ними, а саме:

– вузли зберігання передають на вузол керування інформацію про кількість вільної пам’яті;

– вузол керування передає вузлам обчислення призначені їм вузли зберігання та кількість доступної пам’яті на них;

– вузли зберігання та обчислення передають інформацію про свій стан через регулярні інтервали часу та вузол керування;

– вузли обчислення передають проміжки пам’яті на зберігання вузлам обчислення та за запитом отримують ці дані назад.

Передача даних між вузлами виконується по протоколу TCP.

3.3.3 Структура клієнта віддаленої пам’яті

Структура бібліотеки клієнта віддаленої пам'яті представлена у вигляді діаграми класів у додатку В. На діаграмі представлені:

– клієнт віддаленої пам’яті (FarMemoryClient), з яким взаємодіє інформаційна система;

– проміжки пам’яті (FarMemorySpan) та набори байтів що вони використовують (FarMemoryData);

– бекенд зберігання на вузлі зберігання (NetworkNodeBackend);

– бекенд зберігання на SSD диску (LocalDiskBackend);

– бекенд зберігання у локальній пам’яті (InMemoryBackend);

– кодування стиранням (ErasureCodingBackend);

– реплікація (ReplicationBackend);

– стиснення даних (CompressionBackend);

– шифрування даних (EncryptionBackend);

– розумні покажчики FarMemory<T> і FarMemoryLocal<T> для зберігання об’єктів;

– розумні покажчики FarMemorySerialized<T> і FarMemorySerializedLocal<T> для зберігання об’єктів з використанням серіалізації;

– буфер байтів (FarMemoryBuffer);

– реалізації структури даних вектор, адаптованих для роботи з віддаленою пам’яттю (FarMemoryVec<T>, FarMemoryBufferedVec<T>, FarMemorySerializedVec<T>);

– реалізація структури даних хеш-таблиця, адаптована для роботи з віддаленою пам’яттю (FarMemoryHashMap);

– реалізація алгоритмів заміщення проміжків: RandomReplacementPolicy, LeastRecentlyUsedreplacementPolicy, MostRecentlyUsedReplacementPolicy, ReplayReplacementPolicy.

3.3.4 Послідовність доступу до даних у віддаленій пам’яті

Схема структурна послідовності доступу до об'єкту що зберігається у віддаленій пам'яті інформаційною системою представлена у додатку Г. На схемі показано випадок коли об'єкт зберігається за допомогою FarMemory<T>, знаходиться у віддаленій пам'яті та при доступі до нього виникає потреба у звільненні додаткової локальної пам'яті.

3.3.5 Послідовність роботи фонового потоку клієнта віддаленої пам’яті

Схема структурна послідовності роботи фонового потоку переміщення проміжків представлена у додатку Д. На схемі показано випадок коли фоновий потік звільняє локальну пам'ять через переміщення проміжків у віддалену пам'ять, а після цього - переміщує проміжок у локальну пам'ять для зменшення блокування основного потоку виконання. Проміжки обираються через використання алгоритму заміщення що спирається на статистику доступів до пам'яті що була зібрана під час роботи програмного забезпечення.

3.4 Інструкція користувача

Користувачем програмного забезпечення що надає віддалену пам'ять є розробник інформаційної системи у яку віддалена пам'ять інтегрується. Рекомендованим методом інтеграції віддаленої пам'яті у програмне забезпечення є використання клієнтської бібліотеки. Саме цей спосіб розглядається далі в цій інструкції.

Першим кроком є розгортання вузла керування. Для цього, користувач повинен на сервері де буде розгорнуто цей компонент виконати команду git clone https://github.com/nikitavbv/far-memory.git для завантаження вихідного коду програмного забезпечення для надання віддаленої пам'яті після чого 'наступні дії

виконуються у папці far-memory. Користувач повинен самостійно створити токен (будь-яка послідовність символів, використовується як пароль) та зберегти у файл config/.token . Після цього, вузол керування запускається командою cargo run --release -- --manager. Після цього, вузол керування є готовим для роботи з іншими компонентами системи.

Наступним кроком є розгортання вузлів зберігання. На кожному з серверів, де розгортаються вузли зберігання користувач повинен виконати дії що описані далі. Так само як і для вузла керування, вихідний код завантажується командою git clone https://github.com/nikitavbv/far-memory.git . Токен що було використано під час налаштування вузла керування, потрібно так само зберегти (використати існуючий токен, не створювати нові) у файл config/.token . Вузол зберігання після цього запускається командою cargo run --release -- --storage --manager-endpoint 'MANAGER_ENDPOINT' --endpoint 'LOCAL_ENDPOINT', де MANAGER_ENDPOINT - адреса та порт (наприклад: 192.168.254.30:14000) на якому працює вузол керування, а LOCAL_ENDPOINT - це адреса та порт вузла зберігання що розгортається. Адреса що вказується для вузла зберігання повинна бути доступною для з'єднання з інших вузлів. Після виконання цієї команди, вузол зберігання є готовим до роботи.

Інтеграція клієнтської бібліотеки у програмний код інформаційної системи що написано на мові програмування Rust відбувається шляхом додавання бібліотеки у список залежностей. Для цього в Cargo.toml потрібно додати рядок far-memory = { git = "https://github.com/nikitavbv/far-memory.git" }. Після цього, клієнт віддаленої пам'яті створюється викликом FarMemoryClient::new (приклад наведено на рисунку 3.1).

```
// create a client when software is initilized:
let client: FarMemoryClient = FarMemoryClient::connect_to(manager_endpoint: "192.168.254.30:14000", token).unwrap();
```

Рисунок 3.1 – Ініціалізація клієнта віддаленої пам'яті

Після цього клієнт віддаленої пам'яті використовується програмним забезпеченням для зберігання даних у цьому класі пам'яті та автоматичного переміщення даних між локальною та віддаленою пам'яттю. Наприклад, для

переміщення об'єкта у віддалену пам'ять, розробник може використати функцію FarMemoryClient::object<T>, яка повертає розумний покажчик з яким можна взаємодіяти так само як і з самим об'єктом, без внесення значних змін у вихідний код програмного забезпечення (приклад наведено на рисунку 3.2).

```
let object: SomeApplicationData = SomeApplicationData::new();
object.do_something();

let object: FarMemory<SomeApplicationData> = client.object(object);
object.do_something();
```

Рисунок 3.2 – Використання об'єктів у віддаленій пам’яті

Використання структур даних адаптованих для роботи з віддаленою пам'яттю відбувається схожим чином. Наприклад, для переміщення списку з елементів у віддалену пам'ять з використанням серіалізації (FarMemorySerializedObjectVec<T>), розробник повинен використати функцію FarMemoryClient::serialized_object_vec<T>, яка повертає розумний покажчик на структуру даних типу вектор адаптовану для роботи з віддаленою пам'яттю (приклад наведено на рисунку 3.3).

```
let data: Vec<i32> = vec![1, 2, 3, 4, 5];

let data: FarMemorySerializedObjectVec<...> = client.serialized_object_vec(objects: data);
for entry: i32 in data.iter() {
    println!("entry: {:?}", entry);
}
```

Рисунок 3.3 – Використання структури даних адаптованої для роботи з віддаленою пам'яттю

Висновки до розділу

В даному розділі було розглянуто реалізацію програмного продукту що надає віддалену пам'ять у розподіленій системі.

Описано вимоги до програмного продукту що розглядається. До цих вимог входять набір фунціоналу, що є необхідним для використання клієнту віддаленої пам'яті у типовому програмному забезпеченні інформаційної системи. Крім цього, описані вимоги до апаратного забезпечення.

Розглянуто використання мови програмування Rust як найбільш ефективної для цієї задачі та визначено набір бібліотек що використовуються.

Описано компоненти з яких складається програмний продукт, зв'язки між ними. Розглянуто компоненти клієнтської бібліотеки віддаленої пам'яті. Наведено діаграми послідовностей її роботи під час взаємодії з даними що знаходяться у віддаленій пам'яті.

Було наведено інструкцію з розгортання віддаленої пам'яті та її інтеграцію у програмне забезпечення інформаційної системи.

4 СТАТИСТИЧНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО МЕТОДУ

Статистичне дослідження ефективності запропонованого методу має на меті отримати відповіді на наступні запитання:

– Яким є рівень швидкодії типового програмного забезпечення з різними схемами доступу до оперативної пам’яті при викори-станні віддаленої пам’яті?

– Як залежить швидкодія програмного забезпечення що використовує віддалену пам’ять від розподілу запитів до об’єктів у пам’яті?

– Яким є рівень швидкодії при використанні різних алгоритмів заміщення проміжків?

4.1 Програмне та апаратне забезпечення, що використовується для дослідження ефективності

Для того, щоб оцінити рівень швидкодії для різних типів програмного забезпечення, конфігурацій клієнта віддаленої пам'яті та алгоритмів заміщення проміжків, була проведена серія статистичних досліджень. Для цих досліджень було використано два сервери з наступним апаратним забезпеченням: процесор AMD Ryzen 5 3600 [66], 64 Гб оперативної пам'яті, мережева карта Intel 82599 [67] пропускною здатністю 10 Гбіт/с. При цьому, між серверами встановлено пряме мережеве з'єднання: мережеві карти з'єднані напряму, без використання додаткового мережевого обладнання, таких як маршрутизатори чи комутатори. На цих серверах встановлена операційна система ArchLinux [68] з версією ядра 6.5.

Віддалена пам'ять була інтегрована у три синтетичні інформаційні системи, що були розроблені для цього дослідження.

Першою такою інформаційною системою є програмне забезпечення для генерації тексту за допомогою великої мовленнєвої моделі Llama2 [69]. За основу була взята існуюча реалізація з відкритим кодом цієї програми на мові програмування Rust (llama2.rs [70]). У віддалену пам'ять було переміщено ваги

нейронної мережі. Ця програма в циклі обробляє запити на генерацію наступного токена тексту. Показником швидкодії для цього програмного забезпечення є кількість згенерованих токенів у хвилину. Особливістю цієї програми є те, що під час генерації тексту відбувається циклічний доступ до усіх параметрів нейронної мережі у незмінному порядку. Це означає що цю програму можна віднести до класу програмного забезпечення, що багато разів сканує весь робочий набір даних у незмінному порядку.

Крім цього, було реалізовано інформаційну систему веб сервісу, що обробляє запити від користувача. Ця програма є подібною до тієї, що використовувалась для оцінки ефективності у AIFM. Ця програма на початку своєї роботи генерує великий масив зображень, кожне з яких має розмір 8Кб. Крім цього, генерується набір користувачів, кожному з яких ставиться у відповідність одне з зображень. Обидві структури даних (масив та хеш-таблиця) зберігаються у віддаленій пам'яті. Під час своєї роботи, ця інформаційна система обробляє запити, кожен з яких складається з 32 випадкових ідентифікаторів користувачів. За кожним ідентифікатором користувача, веб сервіс отримує ідентифікатор зображення з хеш-таблиці. Серед усіх ідентифікаторів зображень у запиті обирається випадковий ідентифікатор, по якому отримується зображення з масиву зображень. Після цього, отримане зображення шифрується за допомогою AES GCM [71], стискається алгоритмом Snappy, після чого результат надсилається у відповідь на запит. Слід зазначити, що розподіл ідентифікаторів користувачів, а також відповідність зображень користувачам слідують розподіленню Ципфа [72] з параметром s (за замовчуванням дорівнює 0.8). Показником швидкодії для цієї програми є кількість запитів що в середньому оброблено за секунду. Ця інформаційна система є прикладом програми з класу програмного забезпечення, що використовує сховище типу ключ-значення для зберігання даних запити до якого відбуваються з деяким нерівномірним розподілом.

Іншою інформаційною системою у яку було інтегровано віддалену пам'ять є застосунок, що обробляє запити до таблиці (dataframe) з даними подібно до

типової системи аналізу даних чи бази даних. На початку своєї роботи, ця програма завантажує у пам'ять набір даних з рейсами літаків взятий з Kaggle [73]. Набор даних з 61 колонки розміщується у структурі даних вектор адаптованої для роботи з віддаленою пам'яттю. Ця інформаційна система обробляє запити згенеровані випадковим чином для фільтрації по даті, аеропорту та авіакомпанії. Для кожного запиту з набору даних обираються 10000 рейсів що відповідають критеріям та рахується середня затримка прибуття літака у аеропорт призначення. Значення затримки повертається як відповідь на запит. Ця інформаційна система належить до класу програм що використовує структури даних адаптовані для роботи з віддаленою пам'яттю, обробляє дані у потоковому режимі та може обробляти дані (у цьому випадку - рядки) в будь-якому порядку.

4.2 Пропускна здатність різних класів програмного забезпечення

Для усіх трьох інформаційних систем що розглядаються було проведено вимірювання пропускної здатності в залежності від частки локальної пам'яті. Іншими словами, розмір набору даних, з яким працює програмне забезпечення є однаковим для усіх запусків програми одного типу, але розмір простору відведений для зберігання даних у локальній пам'яті обмежується певним процентом від розміру набору даних, інша частина зберігається у пам'яті віддалених вузлів. Значення пропускної здатності нормалізуються максимальною пропускною здатністю отриманою для типу програмного забезпечення з усіх проведених вимірювань.

Результати вимірювання пропускної здатності наведено на графіку, що зображено на рисунку 4.1.

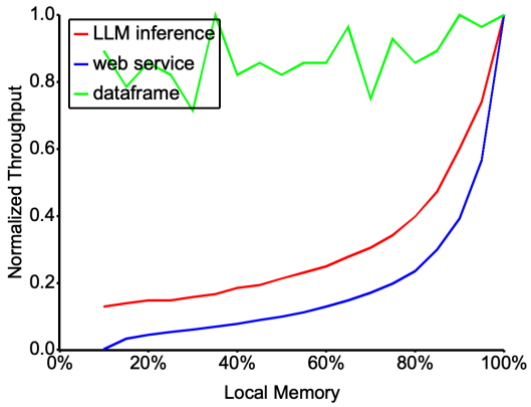


Рисунок 4.1 – Пропускна здатність програмного забезпечення в залежності від його типу та обсягу локальної пам'яті

З цього графіку помітно, що зі зменшенням частки локальної пам'яті зменшується пропускна здатність програмного забезпечення. При цьому, інформаційна система що обробляє запити до таблиці (dataframe) з усіх програм що розглядалися мала найменше зниження швидкодії. Це пояснюється використанням структур даних адаптованих для роботи з віддаленою пам'яттю, що дозволило знизити кількість переміщень проміжків між локальною пам'яттю та пам'яттю віддалених вузлів.

Порівняння рівня швидкодії для програми що генерує текст та веб сервісу приводить до висновку що віддалена пам'ять є більш ефективною для програмного забезпечення що працює з великими об'єктами та незмінними схемами доступу до пам'яті.

Зі зниженням частки локальної пам'яті рівень швидкодії програмного забезпечення наближується до 15% від максимального.

4.3 Вплив розподілу запитів на пропускну здатність

Для вимірювання впливу розподілу запитів на пропускну здатність, було зібрано статистику пропускної здатності інформаційної системи веб сервісу з 80% локальної пам'яті та різним значенням параметру s розподілу Ципфа для розподілу ідентифікаторів користувачів у запитах та збережених ідентифікаторів зображень. При s = 0 розподіл є рівномірним, при s = 1 всі запити мають однакові ідентифікатори.

Результати вимірювання пропускної здатності при різних значеннях параметру розподілу наведено на графіку, що зображено на рисунку 4.2.

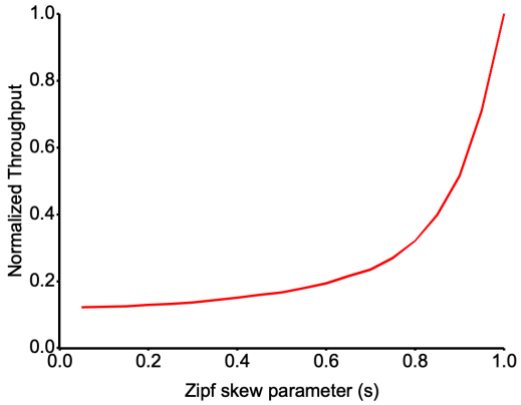


Рисунок 4.2 – Пропускна здатність веб сервісу в залежності від розподілу запитів

З результатів цього дослідження можна зробити висновок що чим більш рівномірним є розподіл запитів до даних у віддаленій пам'яті, тим більшим є негативний ефект на швидкодію програмного забезпечення.

На практиці, типове сховище типу ключ-значення має розподіл запитів що є близьким до розподілу Ципфа з параметром s = 0.88. Для такого програмного забезпечення при частці локальної пам'яті рівної 80% від загального обсягу пам'яті, пропускна здатність згідно з графіком дорівнює 63% від звичайного рівня.

4.4 Порівняння ефективності алгоритмів заміщення проміжків

Для порівняння ефективності різних алгоритмів заміщення проміжків було використано програмне забезпечення для генерації тексту з часткою локальної пам'яті рівної 80% від усієї пам'яті що використовується для зберігання вагів нейронної мережі.

Були порівняні наступні алгоритми заміщення проміжків:

– заміщення проміжків випадковим чином (random), тобто у разі необхідності переміщення проміжків з локальної пам'яті у пам’ять віддалених вузлів проміжки обираються випадковим чином зі списку проміжків що знаходяться у локальній пам’яті. Цей алгоритм є зручним для визначення

базового рівня, з яким можна порівнювати інші алгоритми заміщення проміжків.;

– заміщення найменш нещодавно використаних проміжків (least recently used), тобто при переміщенні проміжків з локальною пам’яті у пам’ять віддалених вузлів обираються ті проміжки доступ до яких не виконувався найбільший інтервал часу. Цей алгоритм часто використовується для заміщення сторінок у операційних системах, у системах кешування та інших схожих задачах. Але програмне забезпечення що розглядається в цьому випадку є гарним прикладом програмного забезпечення з схемами доступу до пам’яті для яких використання LRU є неефективним. Оскільки доступ до вагів кожного шару нейронної мережі є незмінним і повторюється в циклі, то найменш нещодавно використаним проміжком буде той, доступ до якого відбудеться одним з наступних.;

– заміщення проміжків з використанням "оптимальної моделі" (алгоритм Беладі) - для переміщення з локальної пам’яті у віддалену обираються ті проміжки, доступ до яких не буде потрібен довше всього (базуючись на попередньо зібраній статистиці про роботу програмного забезпечення). Оскільки ефективна робота цього алгоритму заміщення проміжків залежить від зібраної статистики, то перед виконанням вимірювання програмне забезпечення було запущено один раз для збору статистики (без цього кроку, перше вимірювання буде нижчим за дійсний показник швидкодії та не буде коректним)..

Результати вимірювання пропускної здатності інформаційної системи що розглядається з різними алгоритмами заміщення проміжків наведено на графіку, що зображено на рисунку 4.3.

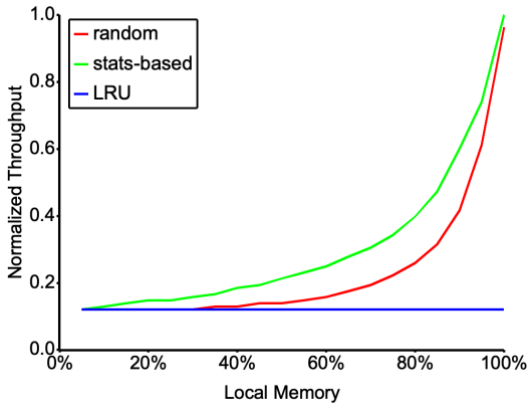


Рисунок 4.3 – Пропускна здатність програмного забезпечення для генерації тексту в залежності від алгоритму заміщення проміжків що використовується

З графіку видно, що для цього типу програмного забезпечення алгоритм LRU є неефективним та призводить до низького рівня швидкодії при будь-якому рівні локальної пам'яті.

Алгоритм заміщення проміжків що адаптує параметри моделі за зібраною статистикою є найбільш ефективним для цієї інформаційної системи. При 40% локальної пам'яті цей алгоритм дозволяє досягти на 53% більшу пропускну здатність у порівнянні з LRU.

Однак, слід зазначити що програмне забезпечення що розглядається є кращим випадком: це програмне забезпечення має незмінну схему доступу до пам'яті для якої можна використати "оптимальну модель" та через особливості роботи з пам'яттю алгоритм LRU який часто використовується для заміщення сторінок не є ефективним.

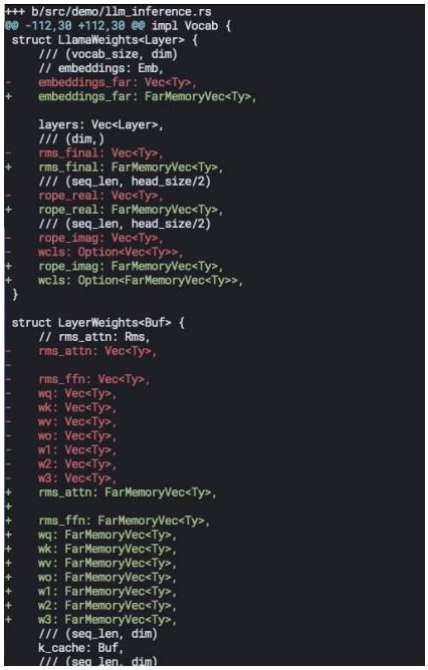
4.5 Зміни у програмний код при інтеграції віддаленої пам’яті

Оскільки при інтеграції віддаленої пам'яті у програмне забезпечення що генерує текст за допомогою великої мовленнєвої моделі за основу було взято програмний код вже існуючого програмного забезпечення, це дає можливість оцінити кількість необхідних змін у код програмного забезпечення для інтеграції у нього віддаленої пам'яті.

В цьому випадку, для інтеграції віддаленої пам'яті було потрібно додати клієнтську бібліотеку у список залежностей та ініціалізувати клієнт вказавши

адресу вузла керування (один рядок коду). Для переміщення вагів нейронної мережі у віддалену пам'ять знадобилося додати у список аргументів функції завантаження вагів клієнт віддаленої пам'яті. Крім цього, було необхідно змінити тип змінних (з Vec<T> на FarMemoryVec<T>) в усіх структурах даних та функціях що працюють з цими даними.

Приклад змін що було виконано наведено на рисунку 4.4 та рисунку 4.5.



```
+++ b/src/demo/llm_inference.rs
@@ -112,30 +112,30 @@ impl Vocab {
 struct LlamaWeights<Layer> {
     /// (vocab_size, dim)
     // embeddings: Emb,
-    embeddings_far: Vec<Ty>,
+    embeddings_far: FarMemoryVec<Ty>,

     layers: Vec<Layer>,
     /// (dim,)
-    rms_final: Vec<Ty>,
+    rms_final: FarMemoryVec<Ty>,
     /// (seq_len, head_size/2)
-    rope_real: Vec<Ty>,
+    rope_real: FarMemoryVec<Ty>,
     /// (seq_len, head_size/2)
-    rope_imag: Vec<Ty>,
-    wcls: Option<Vec<Ty>>,
+    rope_imag: FarMemoryVec<Ty>,
+    wcls: Option<FarMemoryVec<Ty>>,
 }

 struct LayerWeights<Buf> {
     // rms_attn: Rms,
-    rms_attn: Vec<Ty>,
-
-    rms_ffn: Vec<Ty>,
-    wq: Vec<Ty>,
-    wk: Vec<Ty>,
-    wv: Vec<Ty>,
-    wo: Vec<Ty>,
-    w1: Vec<Ty>,
-    w2: Vec<Ty>,
-    w3: Vec<Ty>,
+    rms_attn: FarMemoryVec<Ty>,
+
+    rms_ffn: FarMemoryVec<Ty>,
+    wq: FarMemoryVec<Ty>,
+    wk: FarMemoryVec<Ty>,
+    wv: FarMemoryVec<Ty>,
+    wo: FarMemoryVec<Ty>,
+    w1: FarMemoryVec<Ty>,
+    w2: FarMemoryVec<Ty>,
+    w3: FarMemoryVec<Ty>,
     /// (seq_len, dim)
     k_cache: Buf,
     /// (seq_len, dim)
```

Рисунок 4.4 – Зміни у визначення структур даних для використання віддаленої пам'яті

```
@@ -147,7 +147,7 @@ type CPULayerFloat = LayerWeights<Vec<Ty>>;
 type Llama2CPUFloat = LlamaWeights<CPULayerFloat>;

 impl Llama2CPUFloat {
-    fn load_weights(cfg: &Config, path: &str) -> Self {
+    fn load_weights(client: FarMemoryClient, cfg: &Config, path: &str) -> Self {
         let (weights, wcls) = load_raw_karpathy(cfg, path);
         let embeddings = weights[0].clone();

@@ -162,32 +162,32 @@ impl Llama2CPUFloat {

         let layers = (0..cfg.n_layers)
             .map(|l| LayerWeights::<Vec<Ty>> {
-                rms_attn: w_layer_iters[0].next().unwrap(),
-                wq: w_layer_iters[1].next().unwrap(),
-                wk: w_layer_iters[2].next().unwrap(),
-                wv: w_layer_iters[3].next().unwrap(),
-                wo: w_layer_iters[4].next().unwrap(),
-                rms_ffn: w_layer_iters[5].next().unwrap(),
-                w1: w_layer_iters[6].next().unwrap(),
-                w2: w_layer_iters[7].next().unwrap(),
-                w3: w_layer_iters[8].next().unwrap(),
+                rms_attn: client.vec(w_layer_iters[0].next().unwrap()),
+                wq: client.vec(w_layer_iters[1].next().unwrap()),
+                wk: client.vec(w_layer_iters[2].next().unwrap()),
+                wv: client.vec(w_layer_iters[3].next().unwrap()),
+                wo: client.vec(w_layer_iters[4].next().unwrap()),
+                rms_ffn: client.vec(w_layer_iters[5].next().unwrap()),
+                w1: client.vec(w_layer_iters[6].next().unwrap()),
+                w2: client.vec(w_layer_iters[7].next().unwrap()),
+                w3: client.vec(w_layer_iters[8].next().unwrap()),
                 k_cache: vec![0 as Ty; cfg.seq_len * cfg.dim],
                 v_cache: vec![0 as Ty; cfg.seq_len * cfg.dim],
             })
             .collect();

-        let rms_final = weights[10].clone();
-        let rope_real =  weights[11].clone();
-        let rope_imag = weights[12].clone();
+        let rms_final = client.vec(weights[10].clone());
+        let rope_real = client.vec(weights[11].clone());
+        let rope_imag = client.vec(weights[12].clone());

         Self {
-            embeddings_far: embeddings.clone(),
+            embeddings_far: client.vec(embeddings),

             layers,
             rms_final,
             rope_real,
             rope_imag,
-            wcls,
+            wcls: wcls.map(|v| client.vec(v)),
         }
     }
 }
```

Рисунок 4.5 – Зміни у визначення функцій для використання клієнту віддаленої пам'яті

Така кількість необхідних змін у програмний код є допустимою, оскільки у типовому програмному забезпеченні інформаційної системи лише невелика частина структур даних мають великий розмір. Саме такі структури можна розглядати для перенесення у віддалену пам'ять, перенесення усіх даних що використовується програмним забезпеченням не є виправданим.

Висновки до розділу

В даному розділі було проведено статистичне дослідження ефективності запропонованого методу надання віддаленої пам'яті у розподілених системах.

Було проведено вимірювання пропускної здатності програмного забезпечення з різними схемами доступу до пам'яті при різних рівнях локальної пам'яті. Було встановлено що запропонований метод надання віддаленої пам'яті є найбільш ефективним для програмного забезпечення що використовує високорівневі структури даних що адаптовані для використання з віддаленою пам'яттю, для програмного забезпечення що працює з великими об'єктами, має незмінні схеми доступу до пам'яті. Крім цього, віддалена пам'ять є більш ефективною для програмного забезпечення що має зміщений розподіл запитів, тобто такий де невелика частина об'єктів отримає значно більше запитів за інші об'єкти.

Було порівняно різні алгоритми заміщення проміжків та встановлено що для розглянутого програмного забезпечення алгоритм заміни проміжків що використовує статистику доступу до пам'яті є більш ефективним за прості евристики.

Віддалену пам'ять було інтегровано у існуюче програмне забезпечення. Це дозволило перевірити що для інтеграції віддаленої пам'яті у програмне забезпечення потрібні лише незначні зміни у програмний код.

5 МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЕКТУ

5.1 Опис ідеї проекту

Для опису ідеї проекту проаналізуємо зміст ідеї що пропонується, можливі напрямки застосування, та основні вигоди що може отримати користувач цього програмного продукту. Результати аналізу наведено у таблиці 5.1.

Таблиця 5.1 - Опис ідеї стартап-проекту

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
Розробка програмного продукту, що надає віддалену пам'ять у розподілених системах	Зменшення використання локальної пам'яті за рахунок перенесення даних у пам'ять віддалених вузлів	Більш ефективне використання ресурсів центру обробки даних, зниження витрат на обладнання
	Використання віддаленої пам'яті для роботи з наборами даних що є більшими за обсяг локальної пам'яті	Підвищення розміру набору даних з яким може працювати програмне забезпечення без значних змін у програмний код

Аналіз техніко-економічних переваг ідеї наведено в таблиці 5.2, де властивості програмного продукту що розглядається порівнюється з конкурентами: Carbink та AIFM. Ці програмні продукти є найбільш близькими з технічного боку, але жоден з них не призначений для широкого використання у зовнішніх центрах обробки даних і не пропонується як продукт чи послуга.

Таблиця 5.2 - Аналіз сильних, слабких та нейтральних сторін запропонованої ідеї

№	**Техніко-економічні характеристики ідеї**	**Продукція конкурентів**			**W (слабка сторона)**	**N (нейтральна сторона)**	**S (сильна сторона)**
		Проект	Carbink	AIFM			
1	Відкритий вихідний код та доступність для зовнішнього використання	+	-	+			+

Продовження таблиці 5.2

2	Не залежить від спеціалізованого апаратного забезпечення	+	+	-		+	
3	Зберігання даних на багатьох віддалених вузлах	+	+	-		+	
4	Підтримка інтеграції без зміни коду	+	-	-			+
5	Зниження затримки за рахунок керування заміщенням сторінок	+	+	+			+

Згідно з наведеною таблицею, програмний продукт що розглядається має переваги перед потенційними конкурентами, а саме: відкритий програмний код та доступність для зовнішнього використання, легка інтеграція у програмне забезпечення та зниження затримки доступу за рахунок більш ефективного алгоритму заміщення проміжків.

5.2 Технологійчний аудит ідеї проекту

Для проведення технологічного аудиту ідеї проекту було проаналізовано які технології є необхідними для реалізації ідеї цього проекту. У таблиці 5.3 проведено аналіз технологічної здійсненності ідеї проекту.

Таблиця 5.3 - Технологічна здійсненність ідеї проекту

№	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Передача даних між вузлами без використання спеціалізованого апаратного забезпечення	Бінарний протокол на основі TCP	+	+

Продовження таблиці 5.3

2	Відстеження доступу до даних за допомогою розумних покажчиків та лічильника посилань	Властивності AsRef, Drop у мові програмування Rust, а також структура AtomicU64	+	+
3	Надання віддаленої пам'яті за допомогою віртуального блокового пристрою	Модуль Network Block Device ядра операційної системи Linux	+	+
4	Кодування стиранням проміжків у віддаленій пам'яті	Бібліотека reed-solomon-erasure у мові програмування Rust	+	+
5	Прогнозування доступу до проміжків даних за допомогою рекурентної нейаронної мережі	Бібліотека машинного навчання candle у мові програмування Rust	+	+
Обрані технології для реалізації ідеї проекту: 1, 2, 3, 4, 5.				

Виходячи з проведеного аналізу, технологічна здійсненність проекту можлива, найбільш прийнятні технології, що було обрані, є наявними та доступними.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Першим кроком визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, є аналіз попиту. У таблиці 5.4 проведено попередню характеристику потенційного ринку стратап-проекту.

Таблиця 5.4 - Попередня характеристика потенційного ринку стратап-проекту

№	Показники стану ринку	Характеристика
1	Кількість головних гравців, од.	0
2	Загальний обсяг продаж, грн/ум.од	Точна статистика відсутня. Розмір ринку програмного забезпечення для надання віддаленої пам'яті може приблизно дорівнювати розміру ринку віртуалізації ($86 мільярдів), оскільки потреба в використанні віддаленої пам'яті виникає у схожих середовищах.
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузлі (або по ринку), %	Невідома

Незважаючи на відсутність повної інформації про потенційний ринок (зумовлену відсутністю існуючих комерційних програмних продуктів у цій сфері), можна зробити висновок що цей ринок є привабливим через велику кількість клієнтів (розмір ринку) які зацікавлені в збільшенні ефективності використання ресурсів у центрі обробки даних, відсутність конкурентів у цій сфері та відсутність обмежень для входу.

Основна характеристика майбутніх клієнтів наведена у таблиці 5.5.

Таблиця 5.5 - Характеристика потенційних клієнтів стартап-проекту

№	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Збільшення ефективності використання ресурсів центру обробки даних	Оператори великих центрів обробки даних	відсутні	Легкість у розгортанні Легкість у інтеграції у програмне забезпечення Рівень відмовостійкості що є допустимим для інформаційної системи, в яку інтегрується віддалена пам'ять Рівень швидкодії що дозволяє інформаційній системі, в яку інтегрується віддалена пам'ять, дотримуватись цільового рівня обслуговування (SLO)

Важливо розглянути фактори що можуть перешкоджати ринковому впровадженню проекту. Такі фактори розглянуто у таблиці 5.6.

Таблиця 5.6 - Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	Конкурентне середовище	Вирішення задачі збільшення ефективності використання оперативної пам'яті на рівні віртуалізації або інших компонентів хмарної інфраструктури.	Підвищення ефективності програмного продукту (рівня швидкодії), оскільки інтеграція на рівні програмного забезпечення повинна бути більш ефективною через більшу кількість наявної інформації.

Продовження таблиці 5.6

2	Незацікавленість клієнтів у використанні програмного продукту	Клієнти зацікавлені у використанні інших підходів до підвищення ефективності використання ресурсів та вважають використання віддаленої пам'яті недоцільним.	Спрощення розгортання компонентів віддаленої пам'яті, вдосконалення методу інтеграції у програмне забезпечення, покращення рівня швидкодії що збільшить область застосування.
3	Відповідність продукту очікуванням та потребам сегменту	Рівень швидкодії віддаленої пам'яті не дозволяє інформаційним системам, в які віддалена пам'ять інтегрується, дотримуватись цільового рівня обслуговування (SLO)	Зниження затримки доступу до даних у віддаленій пам'яті за рахунок збільшення ефективності клієнта віддаленої пам'яті, використання більш ефективних алгоритмів заміщення проміжків, і т.д.

Крім цього, існують фактори що сприяють ринковому впровадженню проекту. Ці фактори розглянуто у таблиці 5.7.

Таблиця 5.7 - Фактори можливостей

№	**Фактор**	**Зміст можливості**	**Можлива реакція компанії**
1	Підвищення цін на серверне обладнання, виникнення складнощів з його покупкою у операторів центрів обробки даних	Необхідність оптимізації використання ресурсів стає більш критичною для клієнтів	Рекламна компанія спрямована на ознайомлення клієнтів з можливістю використання віддаленої пам'яті.

Продовження таблиці 5.7

2	Значні зміни на ринку віртуалізації/хмарних обчислень	Умови в яких розробники інформаційних систем переглядають архітектуру програмного забезпечення, інтеграція віддаленої пам'яті може бути однією зі змін що виконуються одночасно.	Надання послуг консультування для таких міграцій
3	Можливості демонстрації програмного продукту	Популярне програмне забезпечення використовує багато оперативної пам'яті (наприклад, бази даних)	Для програмного забезпечення з відкритим кодом створювати версії в які інтегрована віддалена пам'ять та яке є простим у розгортанні. Виконати вимірювання швидкодії для демонстрації привабливості використання віддаленої пам'яті.

Для більш ефективного розвитку програмного продукту на ринку необхідно провести аналіз рис конкуренції на ринку. Ступеневий аналіз конкуренції на ринку розглянуто у таблиці 5.8.

Таблиця 5.8 - Ступеневий аналіз конкуренції на ринку

№	Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1	Тип конкуренції: чиста	Програмний продукт для забезпечення віддаленої пам'яті не може повністю замінити інше програмне забезпечення для надання віддаленої пам'яті, можуть співіснувати навіть у межах одного центру обробки даних; Ринок має умови для входу і виходу; Ціна відрізняється між конкурентами.	Розробка програмного продукту з властивостями, яких немає у конкурентів. Встановлення ціни що відображає рівень переваг над іншими програмними продуктами.
2	За рівнем конкурентної боротьби: мультинаціональний бізнес	Програмний продукт не потребує значних змін для використання у закордонних центрах обробки даних.	Використання великої кількості програмного забезпечення для інтеграції з усього світу. Орієнтація на інтеграцію у найбільші світові інформаційні системи та центри обробки даних та використання цього для подальшого розвитку продукту.

Продовження таблиці 5.8

3	За галузевою ознакою: міжгалузева	Конкуренція з іншими типами рішень: на рівні інфраструктури чи апаратних компонентів які також могли б вирішувати проблему більш ефективного використання оперативної пам'яті	Адаптація програмного забезпечення під зміни у інфраструктурі та компоненти інформаційних систем. Підвищення ефективності надання віддаленої пам'яті за рахунок використання покращень інших продуктів та технологій (наприклад, нові можливості апаратної платформи які можна використати для більш ефективного переміщення даних між вузлами).
4	Конкуренція за видами товарів: товарно-родова	Програмні продукти що конкурують спираються на різні підходи для надання віддаленої пам'яті. Конкурентну перевагу надає не більш ефективна реалізація підходу, а використання більш ефективних підходів.	Постійний аналіз впливу різних факторів на рівень швидкодії віддаленої пам'яті, внесення значних змін у методи її надання якщо це необхідно для покращення швидкодії.
5	За характером конкурентних переваг: нецінова	Найбільшим фактором є кількість оперативної пам'яті що було перерозподілено для більш ефективного використання у інформаційній системі у порівнянні з затратами на інтеграцію віддаленої пам'яті у програмне забезпечення.	Постійний моніторинг ефективності віддаленої пам'яті для різних типів програмного забезпечення, пошук шляхів зменшення затримки доступу до даних та збільшення відношення обсягу даних у віддаленій пам'яті до обсягу даних у локальній пам'яті
6	За інтенсивністю: марочна	Створення бренду програмного продукту що підвищує ефективність використання ресурсів у центрі обробки даних	Робота над розвитком бренду

Для більш детального аналізу умов конкуренції в галузі використовується модель М. Портера [74]. Детальний аналіз умов конкуренції на ринку наведено у таблиці 5.9.

Таблиця 5.9 - Аналіз конкуренції в галузі за М. Портером

Складові аналізу		**Висновки**
Прямі конкуренти в галузі	Відсутні	В даний момент відсутні інші програмні продукти для надання віддаленої пам'яті що є доступними для практичного використання.
Потенційні конкуренти в галузі	Хмарна інфраструктура	Альтернативою збільшення ефективності використання серверного обладнання є використання хмарної інфраструктури, де інформаційній системі надається рівно стільки ресурсів скільки є потрібним для її ресурсів. Слід аналізувати типові випадки клієнтів для демонстрації, що збільшення ефективності використання серверного обладнання за допомогою віддаленої пам'яті є більш економічно вигідним у порівнянні з використанням хмарної інфраструктури.
Постачальники	Постачальники серверного обладнання	Незважаючи на те, що цей програмний продукт не залежить від спеціалізованого апаратного забезпечення, особливості апаратної платформи (серверного обладнання) мають значний вплив на ефективність роботи віддаленої пам'яті.

Продовження таблиці 5.9

Клієнти	Оператори великих центрів обробки даних	Важливим є те, що оператори великих центрів обробки даних зазвичай мають можливість розробити власний програмний продукт для надання віддаленої пам'яті. Це означає, що програмний продукт що розробляється у цьому проекті повинен мати властивості що роблять його використання більш привабливим за розробку власного рішення окремими операторами центрів обробки даних.
Товари-замінники	Програмні продукти що виконують функції віртуалізації або що є компонентами хмарної інфраструктури.	Програмні продукти такого типу можуть підвищувати ефективність використання оперативної пам'яті іншими методами, наприклад за допомогою зміни механізмів керування пам'яттю на рівні операційної системи чи віртуальної машини. Однак, такий підхід є менш ефективним у порівнянні з інтеграцією віддаленої пам'яті на рівні програмного забезпечення, що зумовлено меншою кількістю доступної інформації про доступ до даних у віддаленій пам'яті.

З проведеного аналізу можна зробити висновок що з огляду на конкурентну ситуацію існує можливість роботи на ринку. Для того, щоб проект був конкурентоспроможним на ринку він повинен мати більшу ефективність у порівнянні з іншими методами збільшення ефективності використання оперативної пам'яті.

Фактори конкурентоспроможності докладніше розглянуто в таблиці 5.10.

Таблиця 5.10 - Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування
1	До і після продажне обслуговування	Надання консультацій перед інтеграцією віддаленої пам'яті у інформаційну систему надає можливість клієнту використовувати віддалену пам'ять більш ефективно у специфічних для клієнта умовах (ціль: правильно обрати дані що будуть зберігатися у віддаленій пам'яті, та компоненти програмного забезпечення що будуть з ними працювати, правильно обрати параметри налаштування віддаленої пам'яті для більш високої ефективності). Консультації після інтеграції віддаленої пам'яті дає можливості змінювати налаштування для більш ефективної роботи та встановити напрямки розвитку технології. Все це приводить до більш широкого впровадження програмного продукту.
2	Модифікація по замовленню	Модифікація програмного продукту для надання віддаленої пам'яті по замовленню від клієнта дає можливість адаптувати програмний продукт для більш ефективної роботи з програмною та апаратною платформою клієнта. Це дозволяє отримати перевагу над продукцією конкурентів завдяки більшій ефективності.
3	Легкість розгортання та інтеграції	Розробка простих методів розгортання віддаленої пам'яті та її інтеграції у програмне забезпечення робить програмний продукт більш конкурентоспроможнім за програмні продукти конкурентів що мають більш складне розгортання та налаштування чи вимоги до програмної та апаратної платформи.

Продовження таблиці 5.10

4	Рівень швидкодії	Більш високий рівень швидкодії (більш низька затримка доступу до даних у віддаленій пам'яті) дає можливість використовувати віддалену пам'ять у програмному забезпеченні що є більш чутливим до додаткових затримок. Це дає перевагу над програмним забезпеченням для надання віддаленої пам'яті що має більш низьку ефективність, що обмежує область застосування.

Базуючись на визначених факторах конкурентоспроможності було проведено аналіз слабких сторін стартап-проекту. Порівняльний аналіз наведено у таблиці 5.11.

Таблиця 5.11 - Порівняльний аналіз сильних та слабких сторін програмного забезпечення для надання віддаленої пам’яті у розподілених системах

№	**Фактор конкурентоспроможності**	**Бали 1-20**	**Рейтинг товарів-конкурентів у порівнянні з запропонованим**						
			-3	**-2**	**-1**	**0**	**+1**	**+2**	**+3**
1	До і після продажне обслуговування	16					+		
2	Модифікація по замовленню	18						+	
3	Легкість розгортання та інтеграції	19							+
4	Рівень швидкодії	15					+		

Фінальним етапом ринкового аналізу можливостей впровадження програмного продукту для надання віддаленої пам'яті є проведення SWOT-аналізу [75]. Сильні та слабкі сторони проекти, а також можливості та загрози наведені у таблиці 5.12.

Таблиця 5.12 - SWOT-аналіз стартап-проекту

Сильні сторони (S):	**Слабкі сторони (W):**
- зниження витрат на серверне обладнання, оскільки його ресурси використовуються більш ефективно; - можливість обробляти набори даних розмір яких є більшим за обсяг локальної пам'яті, без значних змін у програмний код.	- негативний ефект на швидкодію програмного забезпечення, що обмежує область застосування (типи програмного забезпечення для якого використання віддаленої пам'яті є допустимим); - потенційна наявність альтернатив у вигляді відповідного функціоналу у програмному забезпеченні віртуалізації та хмарної інфраструктури.
Можливості (O):	**Загрози (T):**
- інтеграція у популярне програмне забезпечення з відкритим кодом що широко використовується (наприклад, бази даних); - зміни на ринку віртуалізації, хмарних обчислень, зміна підходів у архітектурі типового програмного забезпечення інформаційних систем що можуть створити привід для інтеграції віддаленої пам'яті у ці системи.	- незацікавленість клієнтів використанням віддаленої пам'яті на користь інших методів підвищення ефективності використання ресурсів або зниження витрат на серверне обладнання; - поява альтернатив що дозволяють іншими методами знизити використання оперативної пам'яті.

В результаті проведеного SWOT-аналізу можна зробити висновок про необхідність роботи над легкістю інтеграції та розгортання віддаленої пам'яті, покращення рівня швидкодії для того щоб робити її використання більш виправданим, робити інтеграцію у існуюче програмне забезпечення що широко використовується різними клієнтами. Це дозволить зробити потенційних користувачів більш зацікавленими у використанні віддаленої пам'яті, знизити рівень витрат необхідних щоб почати роботу з цим програмним продуктом, розширити область застосування.

У таблиці 5.13 наведено альтернативи ринкового впровадження стартап-проекту.

Таблиця 5.13 - Альтернативи ринкового впровадження стартап-проекту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Інтеграція віддаленої пам'яті у демонстраційне програмне забезпечення (популярне програмне забезпечення з відкритим програмним кодом), дослідження ефективності у кожному випадку, створення звітів	Час	3 місяці
2	Статті, відео-огляди, сторінки в соціальних мережах	Час та фінансові ресурси	6 місяців
3	Демонстрація програмного продукту на тематичних конференціях та заходах	Власні кошти	місяць

Обраною альтернативою обрано інтеграцію віддаленої пам'яті у демонстраційне програмне забезпечення, оскільки отримання ресурсів для цього є найбільш ймовірним, строки відносно стислі, а результат у вигляді звітів по зниження використання ресурсів для демонстраційного програмного забезпечення є ефективним для підвищення зацікавленості серед потенційних користувачів.

5.4 Розроблення ринкової стратегії проекту

Першим кроком розроблення ринкової стратегії проекту є визначення стратегії охоплення ринку. Опис цільових груп потенційних користувачів наведено у таблиці 5.14.

Таблиця 5.14 - Вибір цільових груп потенційних користувачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Індивідуальні розробники програмного забезпечення що працюють над невеликими проектами	Низька, оскільки підвищення ефективності використання ресурсів не буде мати значного результату через невелику кількість обладнання що використовується для розгортання інформаційної системи.	1%	Низька	Середня, оскільки ця категорія користувачів зацікавлена в ознайомленні з віддаленою пам'яттю як альтернативою або доповненням інших методів підвищення ефективності використання ресурсів, але при цьому зазвичай не мають середовища де використання віддаленої пам'яті було б виправданим.

Продовження таблиці 5.14

2	Підприємства середнього розміру що є операторами інформаційних систем	Середня, оскільки такі підприємства зацікавлені у зниженні затрат на серверне обладнання, при цьому можуть мати обмежені ресурси для впровадження віддаленої пам'яті.	60%	Низька	Легка, оскільки в більшості випадків для цієї категорії єдиною умовою є вартість впровадження що є меншою за вартість ресурсів що було використано більш ефективно.
3	Великі підприємства що є операторами великих інформаційних систем та центрів обробки даних	Висока, оскільки такі підприємства зацікавлені в збільшенні ефективності використання ресурсів що має прямий вплив на одну з головних їх категорії витрат.	90%	Середня, зумовлена тим що таке підприємство має ресурси та зацікавленість у розробці власного рішення	Середня, через специфічні у кожному випадку вимоги, складністю інтеграції через складність програмного забезпечення інформаційної системи, конкуренцією з власними розробками підприємства.

Продовження таблиці 5.14

4	Оператори публічних хмарних платформ	Середня, оскільки такі підприємства не завжди мають можливість змінювати програмний код програмного забезпечення що працює у їх центрах обробки даних (така можливість є лише для високорівневого програмного забезпечення що вони самі розробляють та надають як послугу). Незважаючи на це, зацікавлені в збільшенні ефективності використання ресурсів оскільки це одна з їх конкурентних переваг над іншими операторами хмарних платформ.	90%	Середня, зумовлена тим що таке підприємство має ресурси та зацікавленість у розробці власного рішення	Висока складність входу через високі вимоги до програмного продукту що надає віддалену пам'ять та значної зацікавленості у розробці власного рішення.
Які цільові групи обрано: підприємства середнього (головний фокус) та великого розміру.					

На основі проведеного аналізу цільових груп потенційних користувачів визначається стратегія охоплення ринку. Базову стратегію розвитку сформовано у таблиці 5.15.

Таблиця 5.15 - Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	**Стратегія охоплення ринку**	**Ключові конкурентоспроможні позиції відповідно до обраної альтернативи**	**Базова стратегія розвитку**
Надання функціональності що відсутня у товарів-замінників	Інтеграція віддаленої пам'яті у популярне програмне забезпечення з відкритим програмним кодом, дослідження ефективності, повідомлення про результати та рекламувати в соціальних мережах, на конференціях і т.д.	Легкість інтеграції та наявність підтверджень високої ефективності програмного продукту та можливості застосування у різних сценаріях, на відміну від конкурентів.	Стратегія диференціації

Наступним важливим кроком є вибір стратегії конкурентної поведінки. Визначення базової стратегії конкурентної поведінки наведено у таблиці 5.16.

Таблиця 5.16 - Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	**Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?**	**Чи буде компанія копіювати основні характеристики товару конкурента, і які?**	**Стратегія конкурентної поведінки**
Так, тому що існуючи програмні продукти не є доступними для використання або мають характеристики що заважають їх використанню на практиці	Компанія буде шукати як нових користувачів зацікавлених у підвищенні ефективності використання ресурсів, так і переконувати користувачів конкурентів виконати міграцію на програмний продукт компанії	Так, копіювання особливостей методу інтеграції у програмне забезпечення (інтерфейс клієнтської бібліотеки) є необхідним щоб користувачі програмного продукту конкурентів розглядали можливість міграції	Стратегія лідера

Стратегією конкурентної поведінки було обрано стратегію лідера, оскільки проект є «першопрохідцем» на ринку і головною метою є переконати потенційних користувачів використовувати цей програмний продукт для підвищення ефективності використання ресурсів, а не перейти з рішення конкурентів що вони вже використовують для цієї задачі.

Наступним кроком є розробка стратегії позиціонування на основі вимог до продукту та обраної базової стратегії розвитку. Визначення стратегії позиціонування наведено у таблиці 5.17.

Таблиця 5.17 - Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
1	Легкість у розгортанні	Стратегія диференціації	Технологічні властивості програмного продукту що знижують витрати на розгортання віддаленої пам'яті	Розгортання за допомогою однієї команди чи кліку
2	Легкість у інтеграції у програмне забезпечення	Стратегія диференціації	Клієнт віддаленої пам'яті що у зручним у використанні у різних типах програмного забезпечення	Працювати з даними у віддаленій пам'яті настільки ж легко, як з даними у локальній пам'яті
3	Рівень відмовостійкості що є допустимим для інформаційної системи, в яку інтегрується віддалена пам'ять	Стратегія спеціалізації	Технологічні властивості програмного продукту що дозоляють відновлювати дані у випадку відмови вузлів розподіленої системи	Розміщення даних на декількох вузлах

Продовження таблиці 5.17

4	Рівень швидкодії що дозволяє інформаційній системі, в яку інтегрується віддалена пам'ять, дотримуватись цільового рівня обслуговування (SLO)	Стратегія диференціації	Технологічні властивості програмного продукту що дозволяють мати низький рівень затримки доступу до даних у віддаленій пам'яті	Інформаційна система зберігає прийнятний рівень швидкодії після інтеграції віддаленої пам'яті

Отже, основою ринкової стратегії буде залучення уваги підприємств середнього та великого розміру що є зацікавленими у збільшенні ефективності використання ресурсів центрів обробки даних. Стратегією позиціонування обрано стратегію диференціації для задоволення вимог цільової аудиторії шляхом розвитку технологічних властивостей програмного продукту. Оскільки програмний продукт є «першопрохідцем» на ринку, то стратегією конкурентної поведінки обрано стратегію лідера.

5.5 Розроблення маркетингової програми стартап-проекту

Першим кроком розроблення маркетингової програми стартап-проекту є формування маркетингової концепції товару. У таблиці 5.18 визначено ключові переваги концепції потенційного товару.

Таблиця 5.18 - Визначення ключових переваг концепції потенційного товару

Потреба	**Вигода, яку пропонує товар**	**Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)**
Збільшення ефективності використання ресурсів центру обробки даних	Зниження витрат на додаткові сервери або розширення оперативної пам'яті в існуючому обладнанні за рахунок перенесення частин даних з пам'яті більш завантажених серверів у пам'ять менш завантажених (тобто, за рахунок використання віддаленої пам'яті)	Більш легке розгортання, більш легка інтеграція у програмне забезпечення інформаційної системи, більш високий рівень швидкодії (нижча затримка доступу до даних у віддаленій пам'яті).

Наступним кроком є розробка трирівневої маркетингової моделі товару. Опис трьох рівнів моделі товару наведено у таблиці 5.19.

Таблиця 5.19 - Опис трьох рівнів моделі товару

Рівні товару	**Сутність та складові**		
1. Товар за задумом	Збільшення ефективності використання ресурсів центру обробки даних, а саме більш ефективне використання оперативної пам'яті у розподіленій системі		
2. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	Розгортання	Нм	Вр/Тх/Е
	Інтеграція у програмне забезпечення	М	Вр/Тх/Тл/Е
	Забезпечення відмовостійкості	Нм	Вр/Тх
	Забезпечення високого рівня швидкодії	М	Вр/Тх/Тл
	Інтерфейс для моніторингу та керування	Нм	Вр/Тх/Е
	Адаптація до запитів клієнта	М	Вр/Тх/Ор
	Якість: покриття усіх компонентів автоматизованими тестами		

Продовження таблиці 5.19

2. Товар у реальному виконанні	Пакування: компоненти віддаленої пам'яті надаються у вигляді Docker контейнерів, конфігурації Kubernetes на Helm для більш простого розгортання
	Марка: назва організації-розробника та назва програмного продукту
3. Товар із підкріпленням	До продажу: наявність документації для усіх компонентів програмного продукту для надання віддаленої пам'яті, надання послуг консультації по розгортанню та інтеграції віддаленої пам'яті
	Після продажу: моніторинг стану віддаленої пам'яті, надання послуг консультацій по обслуговуванню та налаштуванню, випуск оновлень
За рахунок чого потенційний товар буде захищено від копіювання: запатентовані алгоритми заміщення проміжків що є фактором швидкодії віддаленої пам'яті, послуги консультації базуються на знаннях та досвіді яких немає у конкурентів	

М/Нм – монотонні/немонотонні; Вр/Тх/Тл/Е/Ор – вартісні/ технічні/ технологічні/ ергономічні/ органолептичні.

Далі слід визначити цінові межі, якими необхідно керуватись при встановленні ціни на потенційний товар. Визначення меж встановлення ціни наведено у таблиці 5.20.

Таблиця 5.20 - Визначення меж встановлення ціни

Рівень цін на товари-замінники	**Рівень цін на товари-аналоги**	**Рівень доходів цільової групи споживачів**	**Верхня та нижня межі встановлення ціни на товар/послугу**
Програмне забезпечення віртуалізації для центрів обробки даних: 0-$10тис за кожен процесор (одноразова витрата)	На момент проведення аналізу комерційних товарів-аналогів не існує на ринку	Важко визначити через наявність великої кількості різних підприємств у цільовій групі користувачів. Але, можна орієнтуватися на рівень витрат на підтримку центрів обробки даних такими підприємствами: $500тис-$50млн на рік	Ліцензія на використання: $0-$1000 (одноразово); Консультації та обслуговування: $100-$20тис (щомісячно)

Наступним кроком є визначення оптимальної системи збуту. У таблиці 5.21 наведено формування системи збуту.

Таблиця 5.21 - Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	**Функції збуту, які має виконувати постачальник товару**	**Глибина каналу збуту**	**Оптимальна система збуту**
За допомогою купленого програмного продукту збільшити ефективність використання ресурсів на суму більше ніж вартість програмного забезпечення для надання віддаленої пам'яті	Постійний доступ до виконуваних файлів компонентів віддаленої пам'яті, документації та інших ресурсів	Однорівневний або нульовий	Власноруч, посередники тільки у вигляді оператора приватної хмарної платформи на стороні клієнта

Останньою складовою маркетингової програми є розроблення концепції маркетингових комунікацій. Концепцію маркетингових комунікацій наведено у таблиці 5.22.

Таблиця 5.22 - Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	**Канали комунікацій, якими користуються цільові клієнти**	**Ключові позиції, обрані для позиціонування**	**Завдання рекламного повідомлення**	**Концепція рекламного звернення**
Цільові клієнти зацікавлені в збільшенні ефективності використання ресурсів центру обробки даних шляхом внесення змін в інформаційну систему вартість яких є нижчою за вартість ресурсів що було використано більш ефективно	Веб-сайти в інтернеті, соціальні мережі, тематичні конференції	Легкість у інтеграції та використанні, значний позитивний вплив на ефективність використання оперативної пам'яті серверів у центрі обробки даних	Звернути увагу на використання віддаленої пам'яті як інструменту зниження витрат на обслуговування та розширення центру обробки даних.	Прозоре збільшення обсягу оперативної пам'яті у сервері клієнта

Отже, було створено маркетингову програму що включає в себе концепції програмного продукту, збуту, просування та попередній аналіз можливостей ціноутворення.

Висновки до розділу

В цьому розділі було проведено маркетинговий аналіз стратап-проекту. Для цього було описано ідею проекту, проведено її технологічний аудит, аналіз ринкових можливостей. Було проаналізовано конкуренцію на ринку, можливості та загрози. Була розроблена ринкова стратегія проекту, обрана стратегія розвитку та стратегія конкурентної поведінки. Для стартап-проекту було розроблено маркетингову програму.

З огляду на наявність попиту, його зростання та рентабельність роботи на ринку, можна зробити висновок що можливість ринкової комерціалізації проекту існує. Конкурентоспроможність проекту є високою що робить його перспективним для впровадження, існує заінтересована група клієнтів для якої бар'єр входження не є високим.

Варіантом впровадження для ринкової реалізації проекту доцільно обрати інтеграцію віддаленої пам'яті у демонстраційне програмне забезпечення (популярне програмне забезпечення з відкритим кодом що широко використовується, таке як бази даних), дослідження ефективності та створення звітів, які потом можна використати у маркетинговій програмі.

Проведений аналіз підтверджує що подальша імплементація проекту є доцільною.

ВИСНОВКИ

У роботі було розв'язано проблему підвищення ефективності використання оперативної пам'яті у центрах обробки даних за рахунок розробки ефективного методу надання віддаленої пам'яті в розподілених інформаційних системах.

При цьому отримано наступні наукові та практичні результати:

– розроблено архітектуру програмного засобу для надання віддаленої пам’яті;

– визначено метод інтеграції у програмне забезпечення за допомогою клієнтської бібліотеки що надає розумні покажчики для зберігання об’єктів у віддаленій пам’яті та структури даних адаптовані для роботи з нею;

– розроблено альтернативний метод інтеграції за рахунок надання віртуального блокового пристрою;

– задача забезпечення відмовостійкості віддаленої пам’яті була вирішена за рахунок використання кодів Ріда-Соломона для кодування та розміщення частин даних на різних вузлах;

– знижено в середньому затримку доступу до блоків у віддаленій пам’яті за рахунок використання фонового потоку виконання для завчасного переміщення проміжків та використання алгоритму заміщення проміжків що спирається на статистику доступу до пам’яті яка неперервно збирається під час роботи програмного забезпечення;

– було створено програмний продукт для надання віддаленої пам’яті що реалізує запропонований метод;

– ефективність запропонованого методу надання віддаленої пам’яті було доведено статистично. Було встановлено що для програмного забезпечення що має схеми доступу до пам’яті що дозволяють ефективно використовувати прогнозні моделі, при 40% локальної пам’яті запронований метод дозволяє досягти на 53% більшу пропускну здатність у порівняні з більш простими алгоритмами заміщення проміжків;

– проведено маркетинговий аналіз стартап-проекту, що включає в себе опис ідеї, її технологічний аудит та аналіз ринкових можливостей запуску

стартап-проекту. Розроблено ринкову стратегію проекту, маркетингову програму. Проведений аналіз показує що подальша імплементація проекту є доцільною.

На основі матеріалів магістерської дисертації було опубліковано тези доповіді на V науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології SoftTech-2023».

Наукова новизна запропонованого методу надання віддаленої пам'яті полягає в тому, що, на відміну від існуючих методів, задача заміщення проміжків вирішена статистично більш ефективно за рахунок реалізації адаптації параметрів моделі прогнозування доступу на основі використання статистики, що неперервно збирається в процесі роботи програмного забезпечення.

Практичне значення отриманих результатів полягає в тому, що розроблене програмне забезпечення для надання віддаленої пам'яті є простим для розгортання, не вимагає значних змін у програмне забезпечення при інтеграції. Дане програмне забезпечення може бути використане для підвищення ефективності використання ресурсів центру обробки даних у програмному забезпеченні параметри роботи якого дозволяють використання такого класу пам'яті як віддалена пам'ять.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, John Wilkes. Borg: the Next Generation. Proceedings of ACM EuroSys. 2020.

2) Salman A. Baset, Long Wang, Chunqiang Tang. Towards an understanding of oversubscription in cloud. Proceedings of the 2nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services. 2012.

3) Amaro, E., Branner-Augmon C., Luo Z., Ousterhout A., Aguilera M. K., Panda A., Ratnasamy S., & Shenker S. J.. Can far memory improve job throughput?. EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems. 2020.

4) Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, Sanjeev Kumar. Flash Storage Disaggregation. Proceedings of the 11th European Conference on Computer Systems (EuroSys). 2016.

5) Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, John K. Ousterhout. It's s Time for Low Latency. 13th Workshop on Hot Topics in Operating Systems (HotOS XIII). 2011.

6) Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Abdul Kabbani, Moray McLaren, Steve Scott. Datacenter Ethernet and RDMA: Issues at Hyperscale. IEEE Computer. 2023.

7) Apache Spark. URL: https://spark.apache.org/ (дата звернення 07.01.2024).

8) Understanding InfiniBand and RDMA // Red Hat Customer Portal. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_infiniband_and_rdma_networks/understanding-infiniband-and-rdma_configuring-infiniband-and-rdma-networks (дата звернення 07.01.2024).

9) RFC 9293 - Transmission Control Protocol. URL: https://datatracker.ietf.org/doc/html/rfc9293 (дата звернення 07.01.2024).

10) IEEE 802.3 ETHERNET. URL: https://www.ieee802.org/3/ (дата звернення 07.01.2024).

11) MPI Forum. URL: https://www.mpi-forum.org/ (дата звернення 07.01.2024).

12) Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, Kang G. Shin. Hydra: Resilient and Highly Available Remote Memory. 20th USENIX Conference on File and Storage Technologies (FAST 22). 2022.

13) The Linux Kernel. URL: kernel.org (дата звернення 07.01.2024).

14) An introduction to Reed-Solomon codes: principles, architecture and implementation. URL: https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html (дата звернення 07.01.2024).

15) Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020.

16) Yang Zhou, Hassan Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Jack Turner, David E Culler, Hank Levy, Amin Vahdat. Carbink: Fault-tolerant Far Memory. Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation. 2022.

17) Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, Tipp James Moseley. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). 2021.

18) Clock Algorithm, Second Chance List Algorithm, and Intro to I/O // CS162. URL: https://inst.eecs.berkeley.edu/~cs162/sp20/static/sections/section8-sol.pdf (дата звернення 07.01.2024).

19) Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid

Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. International Conference on Architectural Support for Programming Languages and Operating Systems. 2019.

20) zswap // ArchWiki. URL: https://wiki.archlinux.org/title/zswap (дата звернення 07.01.2024).

21) Gaussian Process (GP) Bandits. URL: https://acsweb.ucsd.edu/~shshekha/GPBandits.html (дата звернення 07.01.2024).

22) Andrew R. Conn, Katya Scheinberg, Luis N. Vicente. Introduction to Derivative-Free Optimization. University City, 2009. 289 p.

23) Redis. URL: https://redis.io/ (дата звернення 07.01.2024).

24) Docker. URL: https://www.docker.com/ (дата звернення 07.01.2024).

25) Kubernetes. URL: https://kubernetes.io/ (дата звернення 07.01.2024).

26) Prometheus - Monitoring system & time series database. URL: https://prometheus.io/ (дата звернення 07.01.2024).

27) Rust Programming Language. URL: https://www.rust-lang.org/ (дата звернення 07.01.2024).

28) Standard C++. URL: https://isocpp.org/ (дата звернення 07.01.2024).

29) Deref in std::ops // Rust. URL: https://doc.rust-lang.org/std/ops/trait.Deref.html (дата звернення 07.01.2024).

30) Serialize in serde // Rust. URL: https://docs.rs/serde/latest/serde/trait.Serialize.html (дата звернення 07.01.2024).

31) bincode // Rust Package Registry. URL: https://crates.io/crates/bincode (дата звернення 07.01.2024).

32) Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). 2017.

33) Block Device Driver // The Linux Kernel documentation. URL: https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html (дата звернення 07.01.2024).

34) Kythe D.K., Kythe P.K.. lgebraic and Stochastic Coding Theory. Boca Raton, 212. 512 p.

35) gRPC. URL: https://grpc.io/ (дата звернення 07.01.2024).

36) RFC 9910 - Http Semantics. URL: https://datatracker.ietf.org/doc/html/rfc9110 (дата звернення 07.01.2024).

37) RFC 768 - User Datagram Protocol. URL: https://datatracker.ietf.org/doc/html/rfc768 (дата звернення 07.01.2024).

38) QUIC Working Group. URL: https://quicwg.org/ (дата звернення 07.01.2024).

39) RFC 896 - Congestion Control in IP/TCP Internetworks. URL: https://datatracker.ietf.org/doc/html/rfc896 (дата звернення 07.01.2024).

40) User-Space Networking // Lecture 14, Computer Networks (198:552). URL: https://people.cs.rutgers.edu/~sn624/552-F19/lectures/14-userspace-networking.pdf (дата звернення 07.01.2024).

41) IEEE SA - IEEE 802.3ae-2002. URL: https://standards.ieee.org/ieee/802.3ae/1089/ (дата звернення 07.01.2024).

42) RFC 8878 - Zstandard Compression and the 'application/zstd' Media Type. URL: https://datatracker.ietf.org/doc/html/rfc8878 (дата звернення 07.01.2024).

43) Snappy | A fast compressor/decompressor. URL: https://google.github.io/snappy/ (дата звернення 07.01.2024).

44) LZ4 - Extremely fast compression. URL: https://lz4.org/ (дата звернення 07.01.2024).

45) Andrew S. Tanenbaum. Page Replacement Algorithms. Modern Operating Systems, 2nd Edition / Andrew S. Tanenbaum. Hoboken, 2001. P. 273-275

46) Andrew S. Tanenbaum. The Least Recently Used (LRU) Page Replacement Algorithm. Modern Operating Systems, 2nd Edition / Andrew S. Tanenbaum. Hoboken, 2001. P. 278-281

47) Page replacement (CS4410) // Cornell University. URL: https://www.cs.cornell.edu/courses/cs4410/2015su/lectures/lec15-replacement.html (дата звернення 07.01.2024).

48) S. Dupond. A thorough review on the current advance of neural network structures. Annual Reviews in Control, vol. 14. 2019.

49) S. Hochreiter, J. Schmidhuber. Long short-term memory. Neural computation, vol. 9, no. 8. 1997.

50) David Harris, Sarah Harris. Digital design and computer architecture (2nd ed.). San Francisco, 2012. 712 p.

51) Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention is All You Need. Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017.

52) crossbeam // Rust Package Registry. URL: https://crates.io/crates/crossbeam (дата звернення 07.01.2024).

53) tokio // Rust Package Registry. URL: https://crates.io/crates/tokio (дата звернення 07.01.2024).

54) tracing // Rust Package Registry. URL: https://crates.io/crates/tracing (дата звернення 07.01.2024).

55) tracing-chrome // Rust Package Registry. URL: https://crates.io/crates/tracing-chrome (дата звернення 07.01.2024).

56) The Trace Event Profiling Tool (about:tracing). URL: https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/ (дата звернення 07.01.2024).

57) vblk // Rust Package Registry. URL: https://crates.io/crates/vblk (дата звернення 07.01.2024).

58) Network Block Device // The Linux Kernel documentation. URL: https://docs.kernel.org/admin-guide/blockdev/nbd.html (дата звернення 07.01.2024).

59) serde // Rust Package Registry. URL: https://crates.io/crates/serde (дата звернення 07.01.2024).

60) serde-bytes // Rust Package Registry. URL: https://crates.io/crates/serde-bytes (дата звернення 07.01.2024).

61) reed-solomon-erasure // Rust Package Registry. URL: https://crates.io/crates/reed-solomon-erasure (дата звернення 07.01.2024).

62) aes-gcm // Rust Package Registry. URL: https://crates.io/crates/aes-gcm (дата звернення 07.01.2024).

63) lz4 // Rust Package Registry. URL: https://crates.io/crates/lz4 (дата звернення 07.01.2024).

64) candle - Minimalist ML framework for Rust // Github. URL: https://github.com/huggingface/candle (дата звернення 07.01.2024).

65) thiserror // Rust Package Registry. URL: https://crates.io/crates/thiserror (дата звернення 07.01.2024).

66) AMD Ryzen 5 3600 // AMD. URL: https://www.amd.com/en/product/8456 (дата звернення 07.01.2024).

67) Intel® 82599ES 10 Gigabit Ethernet Controller. URL: https://www.intel.com/content/www/us/en/products/sku/41282/intel-82599es-10-gigabit-ethernet-controller/specifications.html (дата звернення 07.01.2024).

68) Arch Linux. URL: https://archlinux.org/ (дата звернення 07.01.2024).

69) Llama 2 // Meta AI. URL: https://ai.meta.com/llama/ (дата звернення 07.01.2024).

70) gaxler/llama2.rs // GitHub. URL: https://github.com/gaxler/llama2.rs (дата звернення 07.01.2024).

71) RFC 5288 - AES Galois Counter Mode (GCM) Cipher Suites for TLS. URL: https://datatracker.ietf.org/doc/html/rfc5288 (дата звернення 07.01.2024).

72) Steven T. Piantadosi. Zipf’s word frequency law in natural language: A critical review and future directions. Psychon Bull. 2014.

73) Flight Status Prediction - Kaggle. URL: https://www.kaggle.com/datasets/robikscube/flight-delay-dataset-20182022 (дата звернення 07.01.2024).

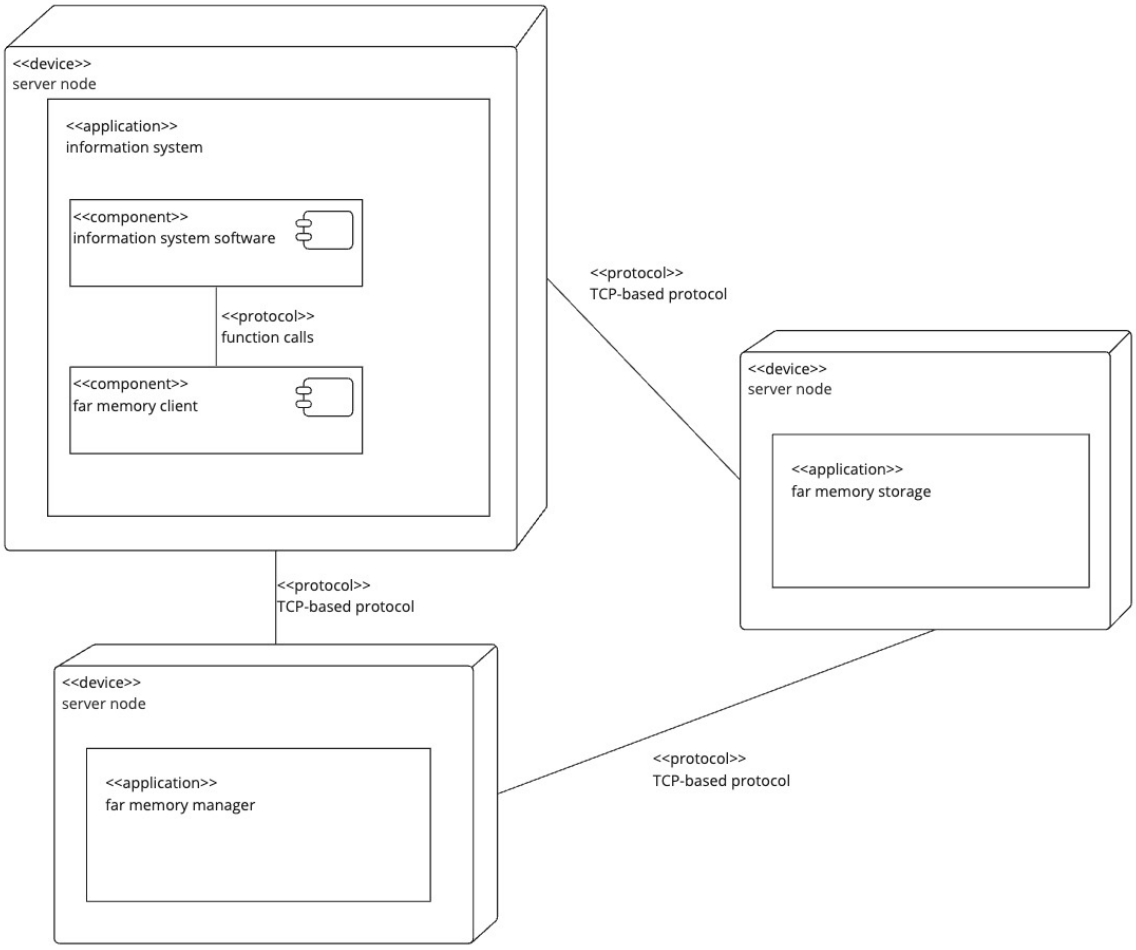
74) How Competitive Forces Shape Strategy. Michael E. Porter. Harvard Business Review. 1979.

75) Heinz Weihrich. The TOWS matrix—a tool for situational analysis. Long Range Planning. 1982.

ДОДАТКИ

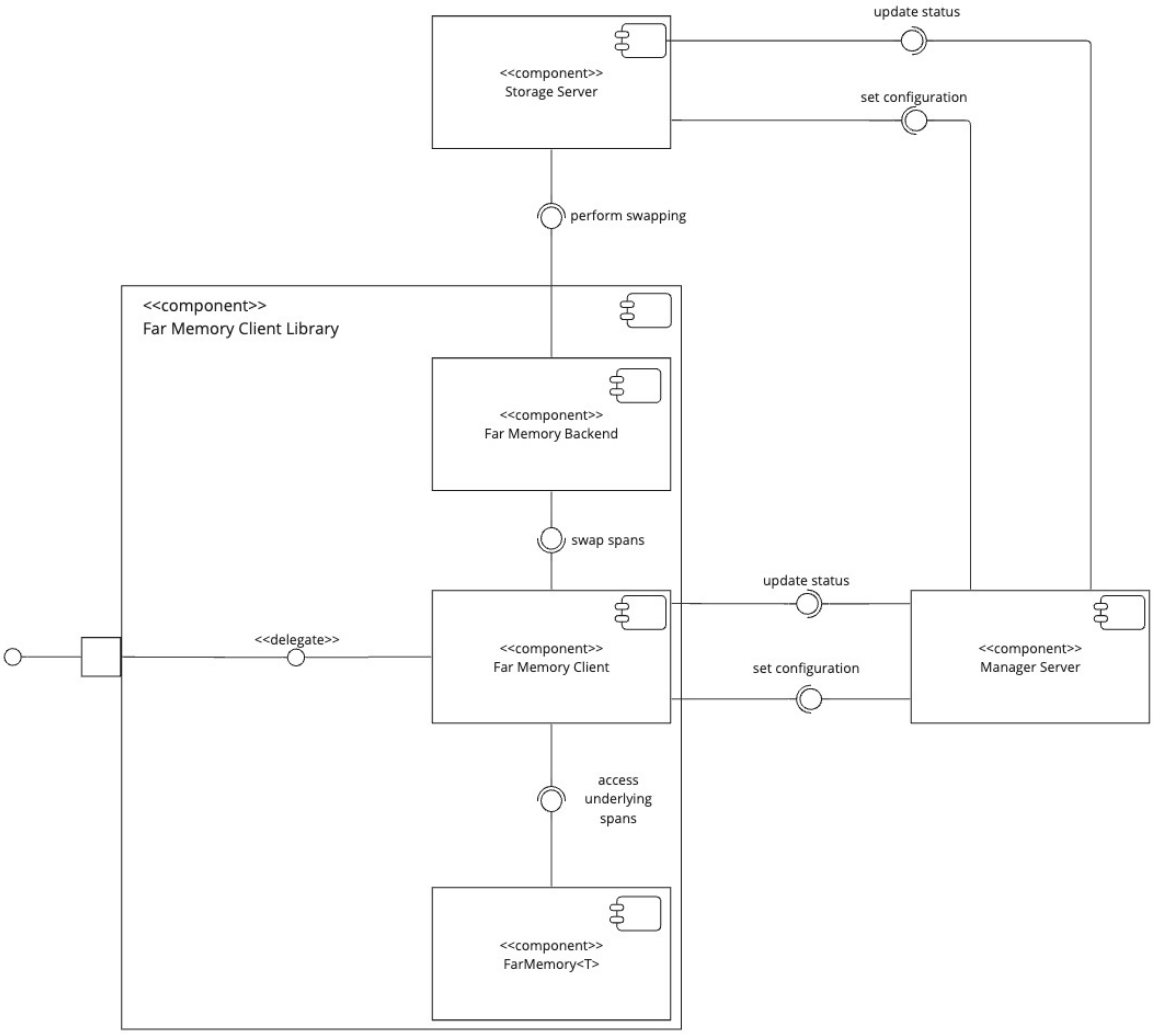
ДОДАТОК А

Схема структурна розгортання



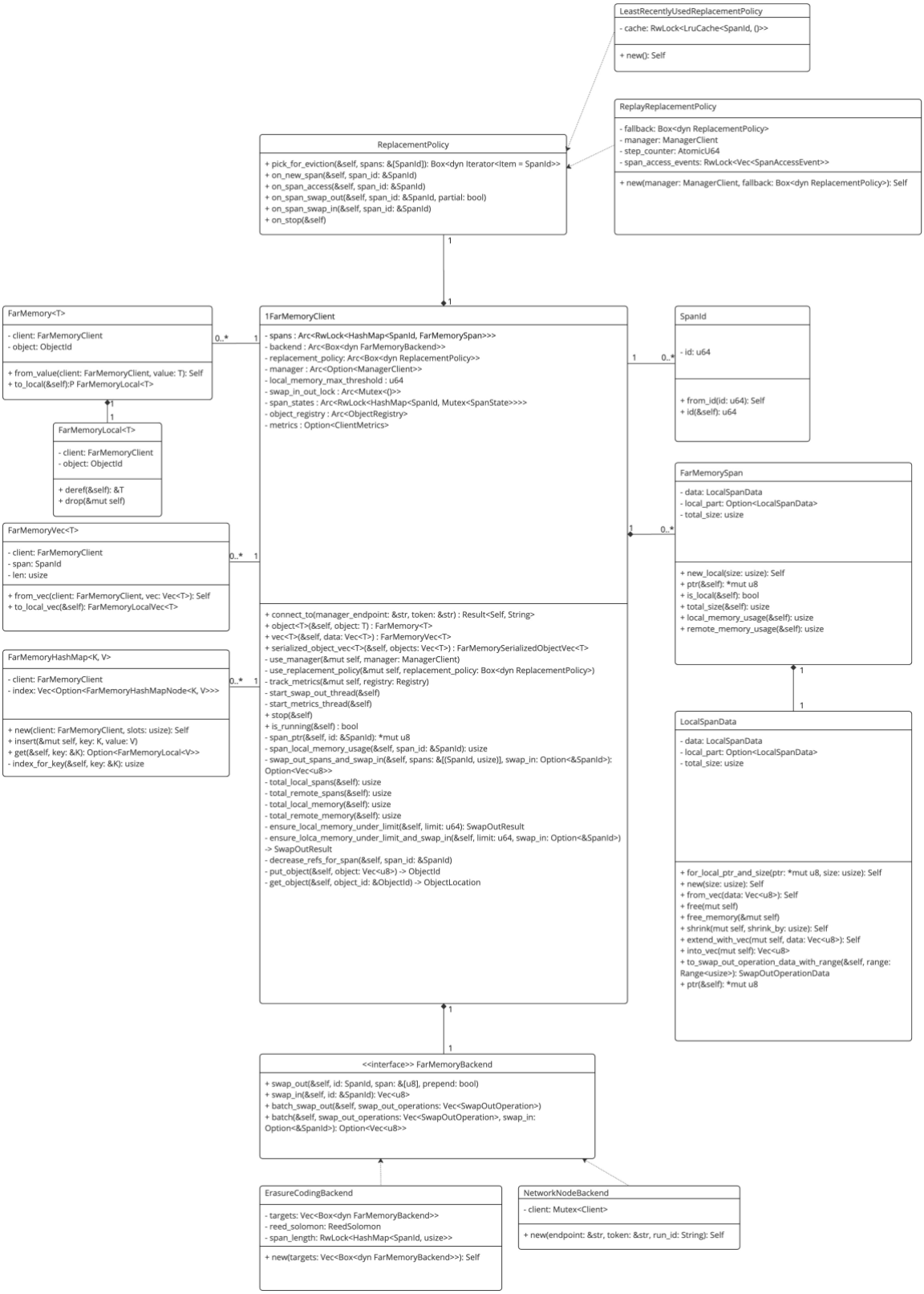
ДОДАТОК Б

Схема структурна компонентів



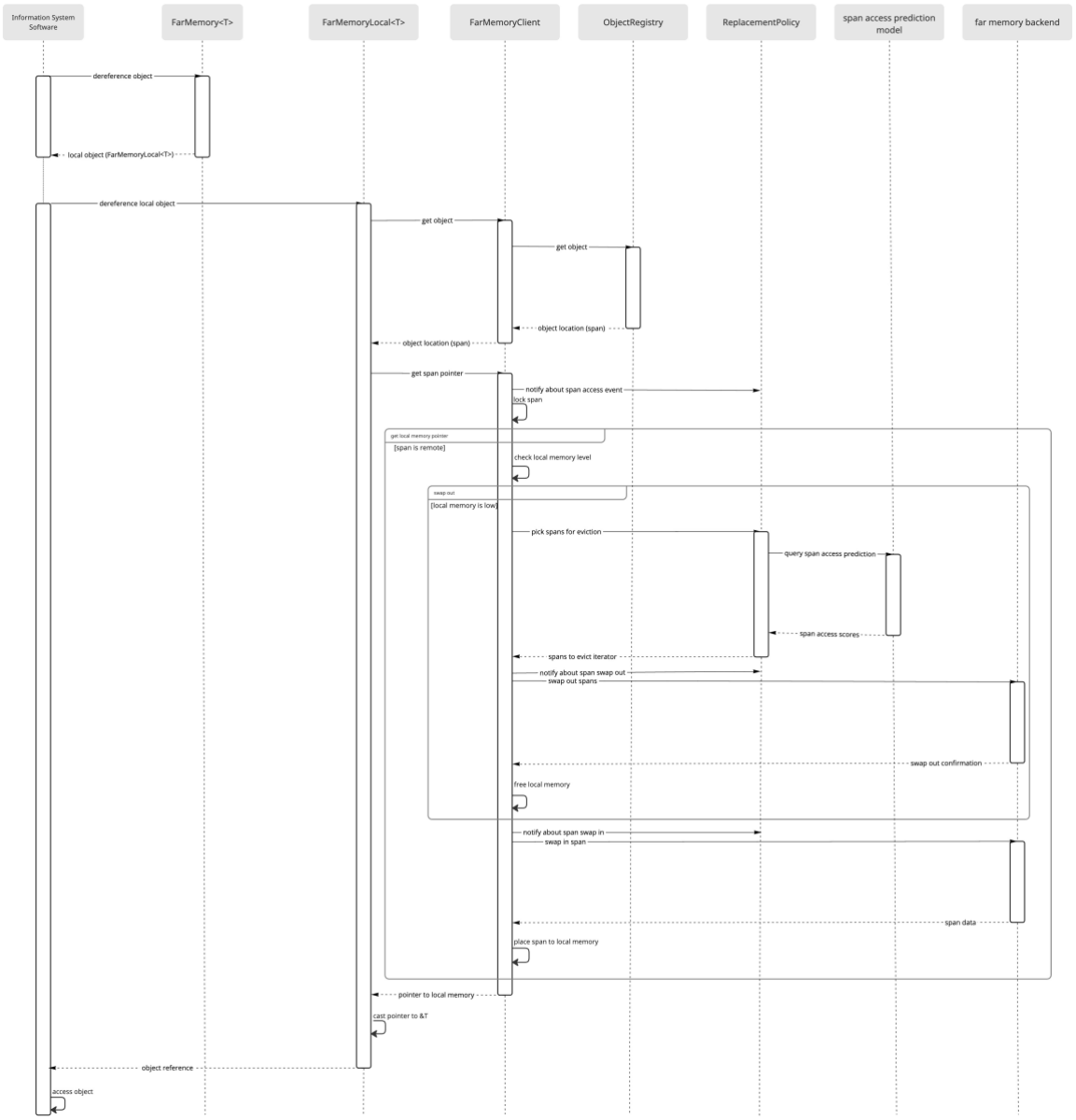
ДОДАТОК В

Структура бібліотеки клієнта віддаленої пам’яті



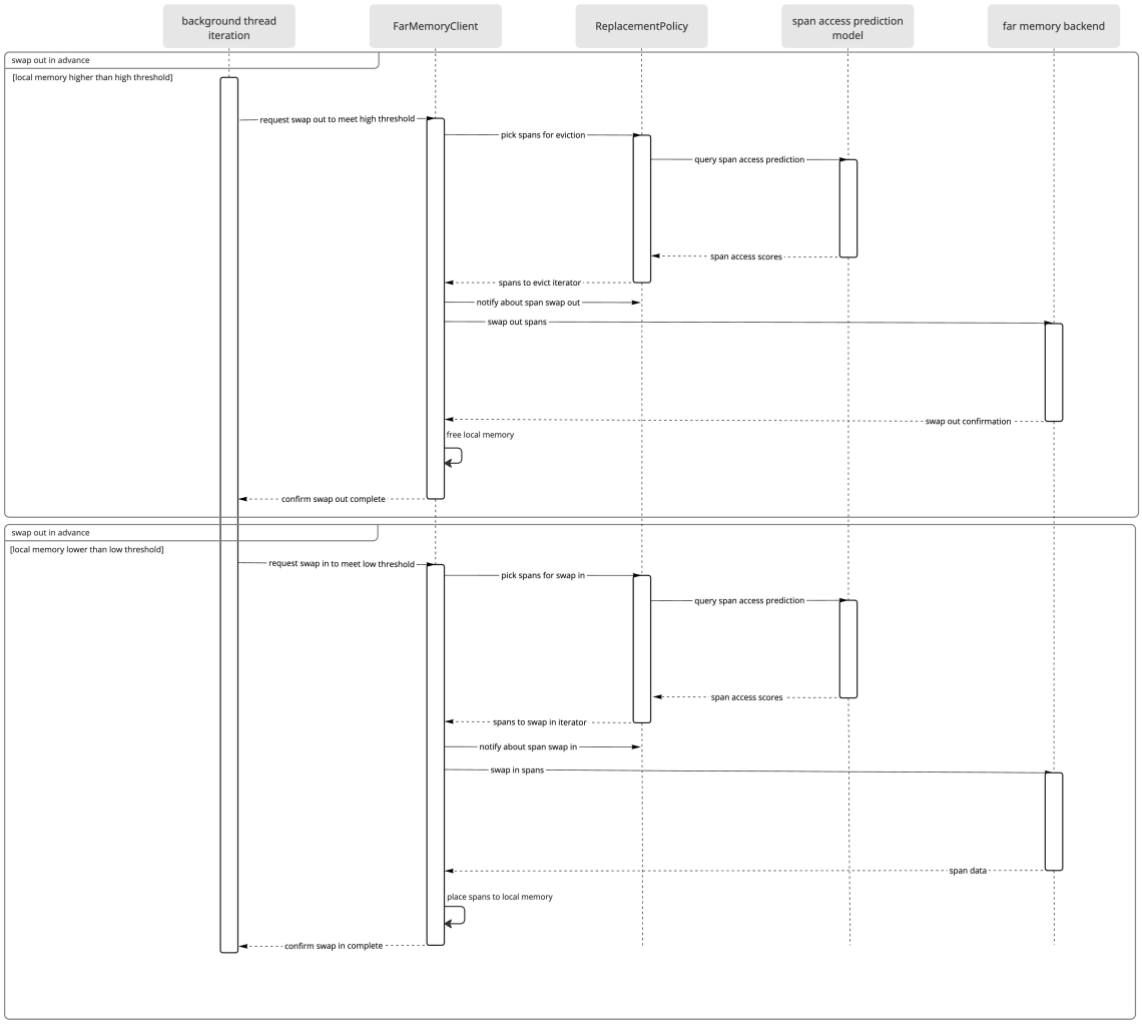
ДОДАТОК Г

Схема структурна послідовності доступу до об’єкту що зберігається у віддаленій пам’яті інформаційною системою



ДОДАТОК Д

Схема структурна послідовності роботи фонового потоку переміщення проміжків



ДОДАТОК Е

Лістинг коду

Файл ./Cargo.toml:

```
[package]
name = "far-memory"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1.25.0", features = ["full"] }
tracing = "0.1.37"
tracing-subscriber = "0.3.16"
vblk = "0.1.2"
toml = "0.7.6"
serde = { version = "1.0.179", features = ["derive"] }
bincode = "1.3.3"
html-escape = "0.2.13"
clap = { version = "4.3.21", features = ["derive"] }
markdown = "0.3.0"
lopdf = "0.31.0"
image = "0.24.7"

# supports tab leader customization for table of contents, see https://github.com/bokuweb/docx-rs/issues/652
# also supports sections for paragraphs (for example, to have paragraphs with multiple columns), see https://github.com/bokuweb/docx-rs/issues/664
docx-rs = { git = "https://github.com/nikitavbv/docx-rs" }

thiserror = "1.0.48"
rand = { version = "0.8.5", features = ["small_rng"] }
rand_distr = "0.4.3"
quantiles = "0.7.1"
tracing-chrome = "0.7.1"
ctrlc = "3.4.1"
crossbeam = "0.8.2"
serde_json = "1.0.107"
reed-solomon-erasure = "6.0.0"
prometheus = "0.13.3"
reqwest = { version = "0.11.22", features = ["blocking"] }
hostname = "0.3.1"
serde_bytes = "0.11.12"
aes-gcm = "0.10.3"
lz4 = "1.24.0"
candle-core = { git = "https://github.com/huggingface/candle.git" }
candle-nn = { git = "https://github.com/huggingface/candle.git" }
itertools = "0.12.0"
snap = "1.1.1"
indicatif = "0.17.7"
chrono = { version = "0.4.31", features = ["serde"] }
csv = "1.3.0"
plotters = "0.3.5"
human_bytes = "0.4.3"
lru = "0.12.1"
```

Файл ./src/client/client.rs:

```
use {
    std::{sync::{Arc, atomic::{AtomicU64, Ordering, AtomicBool}, RwLock, Mutex}, collections::HashMap, thread, time::{Instant, Duration}},
    tracing::{Level, span, info, debug, warn},
    crossbeam::utils::Backoff,
    prometheus::{Registry, register_int_gauge_with_registry, IntGauge, IntCounter, register_int_counter_with_registry},
    crate::manager::ManagerClient,
    super::{
        backend::{FarMemoryBackend, SwapOutOperation, SwapOutOperationData},
        replacement::{ReplacementPolicy, MostRecentlyUsedReplacementPolicy, PreferRemoteSpansReplacementPolicy, ReplayReplacementPolicy},
        span::{SpanId, FarMemorySpan, LocalSpanData},
        object::{ObjectId, ObjectRegistry, ObjectLocation, FarMemory},
        serialized_object_vec::FarMemorySerializedObjectVec,
        vec::FarMemoryVec,
    },
};

#[derive(Clone)]
```

```
pub struct FarMemoryClient {
    span_id_counter: Arc<AtomicU64>,
    spans: Arc<RwLock<HashMap<SpanId, FarMemorySpan>>>,
    is_running: Arc<AtomicBool>,

    backend: Arc<Box<dyn FarMemoryBackend>>,
    replacement_policy: Arc<Box<dyn ReplacementPolicy>>,
    manager: Arc<Option<ManagerClient>>,

    local_memory_max_threshold: u64,

    swap_in_out_lock: Arc<Mutex<()>>,
    span_states: Arc<RwLock<HashMap<SpanId, Mutex<SpanState>>>>,

    object_registry: Arc<ObjectRegistry>,

    metrics: Option<ClientMetrics>,
}

#[derive(Eq, PartialEq)]
enum SpanState {
    Free, // can be local or remote
    InUse(usize), // span is in use, it is local
    Swapping, // swapping in or swapping out
}

struct SwapOutResult {
    spans: usize,
    bytes: usize,

    swap_in_span_data: Option<Vec<u8>>, // data of a span that was swapped in during the same request
}

impl FarMemoryClient {
    // higher level API
    pub fn connect_to(manager_endpoint: &str, token: &str) -> Result<Self, String> {
        unimplemented!()
    }

    pub fn object<T>(&self, object: T) -> FarMemory<T> {
        unimplemented!()
    }

    pub fn vec<T>(&self, data: Vec<T>) -> FarMemoryVec<T> {
        FarMemoryVec::from_vec(self.clone(), data)
    }

    pub fn serialized_object_vec<T>(&self, objects: Vec<T>) -> FarMemorySerializedObjectVec<T> {
        unimplemented!()
    }

    // lower level API
    pub fn new(backend: Box<dyn FarMemoryBackend>, local_memory_max_threshold: u64) -> Self {
        Self {
            span_id_counter: Arc::new(AtomicU64::new(0)),
            spans: Arc::new(RwLock::new(HashMap::new())),
            is_running: Arc::new(AtomicBool::new(true)),

            backend: Arc::new(backend),
            replacement_policy:
Arc::new(Box::new(ReplayReplacementPolicy::new(Box::new(PreferRemoteSpansReplacementPolicy::new(Box::new(MostRecentlyUsedReplacem
entPolicy::new()))))),
            manager: Arc::new(None),
            local_memory_max_threshold,

            swap_in_out_lock: Arc::new(Mutex::new(())),
            span_states: Arc::new(RwLock::new(HashMap::new())),

            object_registry: Arc::new(ObjectRegistry::new()),

            metrics: None,
        }
    }

    pub fn use_manager(&mut self, manager: ManagerClient) {
        self.manager = Arc::new(Some(manager));
    }
```

```
pub fn use_replacement_policy(&mut self, replacement_policy: Box<dyn ReplacementPolicy>) {
    self.replacement_policy = Arc::new(replacement_policy);
}

pub fn track_metrics(&mut self, registry: Registry) {
    self.metrics = Some(ClientMetrics::new(registry));
    self.start_metrics_thread();
}

pub fn start_swap_out_thread(&self) {
    thread::Builder::new().name("swap-out".to_owned())
        .spawn(swap_out_thread(
            self.clone(),
            self.local_memory_max_threshold - 256 * 1024 * 1024
        )).unwrap();
}

pub fn start_metrics_thread(&self) {
    thread::Builder::new().name("metrics".to_owned())
        .spawn(report_metrics_thread(self.clone()))
        .unwrap();
}

pub fn stop(&self) {
    self.is_running.store(false, Ordering::Relaxed);
    self.replacement_policy.on_stop();
    self.backend.on_stop();
    if let Some(metrics) = self.metrics.as_ref() {
        metrics.unregister();
    }
    if let Some(manager) = self.manager.as_ref() {
        manager.on_stop();
    }
}

pub fn is_running(&self) -> bool {
    self.is_running.load(Ordering::Relaxed)
}

pub fn allocate_span(&self, span_size: usize) -> SpanId {
    span!(Level::DEBUG, "allocate_span - ensure local memory limit").in_scope(|| {
        self.ensure_local_memory_under_limit(self.local_memory_max_threshold - span_size as u64, true);
    });

    let _guard = span!(Level::DEBUG, "waiting for lock").in_scope(|| self.swap_in_out_lock.lock().unwrap());
    let id = SpanId::from_id(self.span_id_counter.fetch_add(1, Ordering::Relaxed));
    self.replacement_policy.on_new_span(&id);
    self.spans.write().unwrap().insert(id.clone(), FarMemorySpan::new_local(span_size));
    self.span_states.write().unwrap().insert(id.clone(), Mutex::new(SpanState::Free));
    id
}

pub fn span_ptr(&self, id: &SpanId) -> *mut u8 {
    let started_at = Instant::now();

    self.replacement_policy.on_span_access(id);
    if let Some(metrics) = self.metrics.as_ref() {
        metrics.span_access_ops.inc();
    }

    let span_remote_size = {
        let backoff = Backoff::new();
        let waiting_for_span_lock = span!(Level::DEBUG, "waiting for span lock");
        let mut waiting_for_span_lock_guard = None;
        loop {
            let span_states = self.span_states.read().unwrap();
            let mut span_state = span_states[id].lock().unwrap();
            match &*span_state {
                SpanState::Free => {
                    drop(waiting_for_span_lock_guard);

                    let span = &self.spans.read().unwrap()[id];
                    if span.is_local() {
                        // span is local already, so no need to swap it in
                        // marking it as in use:
                        *span_state = SpanState::InUse(1);

                        if let Some(metrics) = &self.metrics {
```

```
                        metrics.access_latency_micros_local.inc_by((Instant::now() - started_at).as_micros() as u64);
                    }
                    return span.ptr();
                } else {
                    // span is not local, so will need to swap it in
                    // marking it as in swapping state
                    *span_state = SpanState::Swapping;
                    break span.remote_memory_usage();
                };
            },
            SpanState::InUse(refs) => {
                drop(waiting_for_span_lock_guard);

                *span_state = SpanState::InUse(refs + 1);

                let span = &self.spans.read().unwrap()[id];
                if let Some(metrics) = &self.metrics {
                    metrics.access_latency_micros_local.inc_by((Instant::now() - started_at).as_micros() as u64);
                }
                return span.ptr();
            },
            SpanState::Swapping => {
                // waiting for swap out to finish to swap back in again
                // or waiting for it to finish swapping in
                if waiting_for_span_lock_guard.is_none() {
                    waiting_for_span_lock_guard = Some(waiting_for_span_lock.enter());
                }
                backoff.spin();
            },
        };
    }
};

let data = span!(Level::DEBUG, "swap out and swap in").in_scope(|| {
    // only need to free as much memory as remote part will take. There is already memory for local part of span
    let result = self.ensure_local_memory_under_limit_and_swap_in(self.local_memory_max_threshold - span_remote_size as u64, Some(id),
true);
    if let Some(metrics) = &self.metrics {
        metrics.span_swap_out_on_access_ops.inc_by(result.spans as u64);
    }
    result.swap_in_span_data.unwrap()
});

// swap in
span!(Level::DEBUG, "span_ptr - swap_in", span_id = id.id(), span_remote_size).in_scope(|| {
    let span = self.spans.write().unwrap().remove(id).unwrap();

    // new swap in with support for partial
    let local_data = match span {
        FarMemorySpan::Local { .. } => panic!("didn't expect span that is being swapped in to be marked as local"),
        FarMemorySpan::Remote { local_part, total_size: _ } => local_part,
    };

    let local_data = span!(Level::DEBUG, "creating local data").in_scope(|| if let Some(local_data) = local_data {
        span!(Level::DEBUG, "extending local data").in_scope(|| local_data.extend_with_vec(data))
    } else {
        span!(Level::DEBUG, "creating local data from vec").in_scope(|| LocalSpanData::from_vec(data))
    });

    let ptr = local_data.ptr();
    self.spans.write().unwrap().insert(id.clone(), FarMemorySpan::Local {
        data: local_data,
    });

    {
        let span_states = self.span_states.read().unwrap();
        let mut span_state = span_states[id].lock().unwrap();
        match &*span_state {
            SpanState::Free => panic!("did not expect span state to be free when finishing swapping in"),
            SpanState::InUse(_) => panic!("did not expect span state to be in use when finishing swapping in"),
            SpanState::Swapping => *span_state = SpanState::InUse(1),
        };
    }

    self.replacement_policy.on_span_swap_in(id);
    if let Some(metrics) = self.metrics.as_ref() {
        metrics.span_swap_in_ops.inc();
        metrics.access_latency_micros_swap_in.inc_by((Instant::now() - started_at).as_micros() as u64);
```

```
            }

            ptr
        })
    }

    pub fn span_local_memory_usage(&self, span_id: &SpanId) -> usize {
        self.spans.read().unwrap().get(&span_id).unwrap().local_memory_usage()
    }

    pub fn swap_out_spans_fully(&self, spans: &[SpanId]) {
        for span in spans {
            self.swap_out_span(span, self.spans.read().unwrap().get(span).unwrap().local_memory_usage());
        }
    }

    pub fn swap_out_spans(&self, spans: &[(SpanId, usize)]) {
        self.swap_out_spans_and_swap_in(spans, None);
    }

    fn swap_out_spans_and_swap_in(&self, spans: &[(SpanId, usize)], swap_in: Option<&SpanId>) -> Option<Vec<u8>> {
        struct SwapOutFinalizeOperation {
            span_id: SpanId,
            local_part: Option<LocalSpanData>,
            full_swap_out: bool,
            total_size: usize,
            swap_out_size: usize,
        }

        let mut swap_out_ops = Vec::new();
        let mut finalize_ops: Vec<SwapOutFinalizeOperation> = Vec::new();

        // (span, how much memory to swap out - can be partial or full swap out)
        for (span_id, swap_out_size) in spans {
            span!(Level::DEBUG, "creating swap op").in_scope(|| {
                let span = self.spans.write().unwrap().remove(&span_id).unwrap();
                let total_size = span.total_size();
                let (local_part, prepend_to_backend) = match span {
                    FarMemorySpan::Local { data } => {
                        (data, false) // not prepending to remote, because span is local
                    },
                    FarMemorySpan::Remote { local_part, total_size: _ } => (
                        local_part.expect("expected span to contain local part when swapping out"),
                        true, // prepending, because this span already contains a remote part
                    ),
                };
                if *swap_out_size > local_part.size() {
                    panic!("swap out size cannot be larger than local part size");
                }
                let remaining_local_part = local_part.size() - swap_out_size;
                let full_swap_out = remaining_local_part == 0;

                let (data, local_part) = span!(Level::DEBUG, "reading local part").in_scope(|| if full_swap_out {
                    (SwapOutOperationData::Owned(local_part.into_vec()), None)
                } else {
                    // read from end
                    (local_part.to_swap_out_operation_data_with_range(remaining_local_part..local_part.size()), Some(local_part))
                });

                let push_ops_span = span!(Level::DEBUG, "push ops");
                let _push_ops_span_guard = push_ops_span.enter();
                swap_out_ops.push(SwapOutOperation::new(span_id.clone(), span!(Level::DEBUG, "data to vec", full_swap_out).in_scope(|| data),
prepend_to_backend));
                finalize_ops.push(SwapOutFinalizeOperation { span_id: span_id.clone(), local_part, full_swap_out, total_size, swap_out_size:
*swap_out_size })
            });
        }

        let swap_in_data = span!(Level::DEBUG, "backend batch swap").in_scope(|| {
            self.backend.batch(swap_out_ops, swap_in)
        });

        span!(Level::DEBUG, "swap out finalize ops").in_scope(|| {
            for op in finalize_ops {
                span!(Level::DEBUG, "finalize op").in_scope(|| {
                    if op.full_swap_out {
                        self.spans.write().unwrap().insert(op.span_id.clone(), FarMemorySpan::Remote { local_part: None, total_size: op.total_size });
                    } else {
```

```
                self.spans.write().unwrap().insert(
                    op.span_id.clone(),
                    FarMemorySpan::Remote {
                        local_part: Some(span!(Level::DEBUG, "shrinking local part").in_scope(|| op.local_part.unwrap().shrink(op.swap_out_size))),
                        total_size: op.total_size
                    }
                );
            }

            let span_states = self.span_states.read().unwrap();
            let mut span_state = span_states[&op.span_id].lock().unwrap();
            if *span_state != SpanState::Swapping {
                panic!("expected span to be in swapping state when actually swapping out");
            }
            *span_state = SpanState::Free;
            self.replacement_policy.on_span_swap_out(&op.span_id, !op.full_swap_out);

            if let Some(metrics) = self.metrics.as_ref() {
                metrics.span_swap_out_ops.inc();
            }
        });
    }
});

swap_in_data
}

fn swap_out_span(&self, span_id: &SpanId, swap_out_size: usize) {
    let span = self.spans.write().unwrap().remove(&span_id).unwrap();

    let total_size = span.total_size();
    let (local_part, prepend_to_backend) = match span {
        FarMemorySpan::Local { data } => {
            (data, false) // not prepending to remote, because span is local
        },
        FarMemorySpan::Remote { local_part, total_size: _ } => (
            local_part.expect("expected span to contain local part when swapping out"),
            true, // prepending, because this span already contains a remote part
        ),
    };
    if swap_out_size > local_part.size() {
        panic!("swap out size cannot be larger than local part size");
    }
    let remaining_local_part = local_part.size() - swap_out_size;
    let full_swap_out = remaining_local_part == 0;

    let data = if full_swap_out {
        local_part.read_to_slice()
    } else {
        // read from end
        local_part.read_to_slice_with_range(remaining_local_part..local_part.size())
    };

    span!(Level::DEBUG, "backend swap out", size = data.len()).in_scope(|| {
        self.backend.swap_out(span_id.clone(), data, prepend_to_backend);
    });

    if full_swap_out {
        self.spans.write().unwrap().insert(span_id.clone(), FarMemorySpan::Remote { local_part: None, total_size });
        local_part.free();
    } else {
        self.spans.write().unwrap().insert(span_id.clone(), FarMemorySpan::Remote { local_part: Some(local_part.shrink(swap_out_size)), total_size
});
    }

    let span_states = self.span_states.read().unwrap();
    let mut span_state = span_states[&span_id].lock().unwrap();
    if *span_state != SpanState::Swapping {
        panic!("expected span to be in swapping out state when actually swapping out");
    }
    *span_state = SpanState::Free;
    self.replacement_policy.on_span_swap_out(span_id, !full_swap_out);

    if let Some(metrics) = self.metrics.as_ref() {
        metrics.span_swap_out_ops.inc();
    }
}
```

```
    pub fn total_local_spans(&self) -> usize {
        self.spans.read().unwrap().iter().filter(|v| v.1.is_local()).count()
    }

    pub fn total_remote_spans(&self) -> usize {
        self.spans.read().unwrap().len() - self.total_local_spans()
    }

    pub fn total_local_memory(&self) -> usize {
        self.spans.read().unwrap().iter().map(|v| v.1.local_memory_usage()).sum()
    }

    pub fn total_remote_memory(&self) -> usize {
        self.spans.read().unwrap().iter().map(|v| v.1.remote_memory_usage()).sum()
    }

    fn ensure_local_memory_under_limit(&self, limit: u64, strict: bool) -> SwapOutResult {
        self.ensure_local_memory_under_limit_and_swap_in(limit, None, strict)
    }

    /// strict: whether to wait if there are no enough spans to swap out to fulfill memory limit request
    fn ensure_local_memory_under_limit_and_swap_in(&self, limit: u64, swap_in: Option<&SpanId>, strict: bool) -> SwapOutResult {
        let current_local_memory = self.total_local_memory() as u64;
        if current_local_memory < limit {
            return SwapOutResult {
                spans: 0,
                bytes: 0,
                swap_in_span_data: swap_in.map(|v| self.backend.swap_in(v)),
            };
        }

        let _swap_ops_lock_guard = span!(Level::DEBUG, "waiting for lock").in_scope(|| self.swap_in_out_lock.lock().unwrap());
        let memory_to_swap_out = current_local_memory - limit;
        let mut spans_to_swap_out = Vec::new(); // (span, how much memory to swap out - can be partial or full swap out)

        let mut total_memory = 0;
        let possible_swap_out_spans: Vec<SpanId> = self.spans.read().unwrap().keys().cloned().collect(); // TODO: remove this completely

        let mut spans_for_eviction = span!(Level::DEBUG, "querying replacement policy").in_scope(||
self.replacement_policy.pick_for_eviction(&possible_swap_out_spans));

        span!(Level::DEBUG, "picking spans for eviction", total_spans=possible_swap_out_spans.len()).in_scope(|| {
            'spans_picking: loop {
                if total_memory >= memory_to_swap_out {
                    break;
                }

                let span_id = loop {
                    if let Some(span_id) = spans_for_eviction.next() {
                        break span_id;
                    } else {
                        warn!("there are no spans to evict remaining that can be picked");
                        if strict {
                            spans_for_eviction = span!(Level::DEBUG, "querying replacement policy").in_scope(||
self.replacement_policy.pick_for_eviction(&possible_swap_out_spans));
                            continue;
                        } else {
                            break 'spans_picking;
                        }
                    }
                };

                let spans = self.spans.read().unwrap();
                let span = spans.get(&span_id).unwrap();
                {
                    let span_states = self.span_states.read().unwrap();
                    let mut span_state = span_states[&span_id].lock().unwrap();
                    match &*span_state {
                        SpanState::Free => {
                            let span_local_memory_size = span.local_memory_usage();
                            if span_local_memory_size == 0 {
                                debug!(span_id=span_id.id(), "skipping span that does not have local memory");
                                continue;
                            }

                            // marking swap as in swapping state so anyone else who needs it has to wait until it is fully swapped out.
                            *span_state = SpanState::Swapping;
```

```
                        let span_swap_out_len = span_local_memory_size.min((memory_to_swap_out - total_memory) as usize);
                        spans_to_swap_out.push((span_id.clone(), span_swap_out_len));
                        total_memory += span_swap_out_len as u64;
                    },
                    SpanState::InUse(_) => {
                        // cannot swap out span that is in use
                        debug!(span_id=span_id.id(), "skipping span that is in use");
                        continue;
                    },
                    SpanState::Swapping => {
                        // cannot swap out span that is already being swapped out or is in progress of being swapped in
                        debug!(span_id=span_id.id(), "skipping span that is in process of swapping");
                        continue;
                    },
                }
            }
        }
    });

    let swap_in_span_data = span!(Level::DEBUG, "perform swapping", needed = memory_to_swap_out, swap_out_req_size =
total_memory).in_scope(|| {
        self.swap_out_spans_and_swap_in(&spans_to_swap_out, swap_in)
    });

    drop(_swap_ops_lock_guard);

    SwapOutResult {
        spans: spans_to_swap_out.len(),
        bytes: total_memory as usize,
        swap_in_span_data,
    }
}

pub fn decrease_refs_for_span(&self, span_id: &SpanId) {
    let span_states = self.span_states.read().unwrap();
    let mut span_state = span_states[span_id].lock().unwrap();
    match &*span_state {
        SpanState::Free => panic!("span is already free!"),
        SpanState::InUse(refs) => *span_state = if *refs == 1 {
            SpanState::Free
        } else {
            SpanState::InUse(refs - 1)
        },
        SpanState::Swapping => panic!("cannot decrease refs for span that is being swapped out or swapped in")
    }
}

// objects
pub fn put_object(&self, object: Vec<u8>) -> ObjectId {
    let object_id = self.object_registry.next_object_id();
    let object_location = self.object_registry.put_object(object_id.clone(), object.len());
    let object_location = if let Some(object_location) = object_location {
        // append object to existing span
        object_location
    } else {
        // create new span for this object
        let size_class = self.object_registry.size_class_for_object(object.len());
        let span_size = 2 * 1024 * 1024;
        let span_size = span_size + (size_class - span_size % size_class);
        let span = self.allocate_span(span_size);
        self.object_registry.add_span_for_object(span.clone(), span_size, object_id.clone(), object.len())
    };

    let span_ptr = self.span_ptr(&object_location.span_id);
    unsafe {
        std::ptr::copy_nonoverlapping(object.as_ptr(), span_ptr.add(object_location.offset), object.len());
    }
    self.decrease_refs_for_span(&object_location.span_id);

    object_id
}

pub fn get_object(&self, object_id: &ObjectId) -> ObjectLocation {
    self.object_registry.get_object(object_id)
}

pub fn is_object_local(&self, object_id: &ObjectId) -> bool {
    let location = self.object_registry.get_object(object_id);
```

```
        self.is_span_local(&location.span_id)
    }

    fn is_span_local(&self, span_id: &SpanId) -> bool {
        self.spans.read().unwrap().get(span_id).map(|v| v.is_local()).unwrap_or(false)
    }
}

#[derive(Clone)]
struct ClientMetrics {
    registry: Registry,

    local_memory: IntGauge,
    remote_memory: IntGauge,
    local_spans: IntGauge,
    remote_spans: IntGauge,

    span_access_ops: IntCounter,
    span_swap_in_ops: IntCounter,
    span_swap_out_ops: IntCounter,
    span_swap_out_on_access_ops: IntCounter,

    background_swap_out_spans: IntCounter,
    background_swap_out_bytes: IntCounter,

    access_latency_micros_local: IntCounter,
    access_latency_micros_swap_in: IntCounter,
}

impl ClientMetrics {
    pub fn new(registry: Registry) -> Self {
        Self {
            registry: registry.clone(),

            local_memory: register_int_gauge_with_registry!(
                "client_local_memory",
                "local memory in bytes",
                registry
            ).unwrap(),
            remote_memory: register_int_gauge_with_registry!(
                "client_remote_memory",
                "remote memory in bytes",
                registry
            ).unwrap(),
            local_spans: register_int_gauge_with_registry!(
                "client_local_spans",
                "number of local spans",
                registry
            ).unwrap(),
            remote_spans: register_int_gauge_with_registry!(
                "client_remote_spans",
                "number of remote spans",
                registry
            ).unwrap(),

            span_access_ops: register_int_counter_with_registry!(
                "client_span_access_ops",
                "total span access operations",
                registry
            ).unwrap(),
            span_swap_in_ops: register_int_counter_with_registry!(
                "client_swap_in_ops",
                "total span swap in operations",
                registry
            ).unwrap(),
            span_swap_out_ops: register_int_counter_with_registry!(
                "client_swap_out_ops",
                "total span swap out operations",
                registry
            ).unwrap(),
            span_swap_out_on_access_ops: register_int_counter_with_registry!(
                "client_swap_out_on_access_ops",
                "total swap out ops to free memory when accessing span",
                registry
            ).unwrap(),

            background_swap_out_spans: register_int_counter_with_registry!(
                "client_background_swap_out_spans",
```

```
            "swapped out spans by background thread",
            registry
        ).unwrap(),
        background_swap_out_bytes: register_int_counter_with_registry!(
            "client_background_swap_out_bytes",
            "swapped out bytes by background thread",
            registry
        ).unwrap(),

        access_latency_micros_local: register_int_counter_with_registry!(
            "client_access_latency_local",
            "local span access latency in microseconds",
            registry
        ).unwrap(),
        access_latency_micros_swap_in: register_int_counter_with_registry!(
            "client_access_latency_swap_in",
            "remote span access latency in microseconds",
            registry
        ).unwrap(),
    }
  }

  pub fn unregister(&self) {
    self.registry.unregister(Box::new(self.local_memory.clone())).unwrap();
    self.registry.unregister(Box::new(self.remote_memory.clone())).unwrap();
    self.registry.unregister(Box::new(self.local_spans.clone())).unwrap();
    self.registry.unregister(Box::new(self.remote_spans.clone())).unwrap();

    self.registry.unregister(Box::new(self.span_access_ops.clone())).unwrap();
    self.registry.unregister(Box::new(self.span_swap_in_ops.clone())).unwrap();
    self.registry.unregister(Box::new(self.span_swap_out_ops.clone())).unwrap();
    self.registry.unregister(Box::new(self.span_swap_out_on_access_ops.clone())).unwrap();

    self.registry.unregister(Box::new(self.background_swap_out_spans.clone())).unwrap();
    self.registry.unregister(Box::new(self.background_swap_out_bytes.clone())).unwrap();

    self.registry.unregister(Box::new(self.access_latency_micros_local.clone())).unwrap();
    self.registry.unregister(Box::new(self.access_latency_micros_swap_in.clone())).unwrap();
  }
}

fn swap_out_thread(client: FarMemoryClient, target_memory_usage: u64) -> impl FnOnce() -> () {
  move || {
    info!("starting swap out thread");
    span!(Level::DEBUG, "swap out thread").in_scope(|| {
      while client.is_running() {
        thread::sleep(Duration::from_millis(16));

        let swap_out_result = span!(Level::DEBUG, "swap out iteration").in_scope(|| {
          client.ensure_local_memory_under_limit(target_memory_usage, false)
        });

        if let Some(metrics) = client.metrics.as_ref() {
          metrics.background_swap_out_spans.inc_by(swap_out_result.spans as u64);
          metrics.background_swap_out_bytes.inc_by(swap_out_result.bytes as u64);
        }
      }
    });
  }
}

fn report_metrics_thread(client: FarMemoryClient) -> impl FnOnce() -> () {
  move || {
    while client.is_running() {
      let metrics = client.metrics.as_ref().unwrap();
      metrics.local_memory.set(client.total_local_memory() as i64);
      metrics.remote_memory.set(client.total_remote_memory() as i64);
      metrics.local_spans.set(client.total_local_spans() as i64);
      metrics.remote_spans.set(client.total_remote_spans() as i64);

      thread::sleep(Duration::from_secs(10));
    }
  }
}

#[cfg(test)]
mod tests {
  use {
```

```
    crate::client::InMemoryBackend,
    super::*,
  };

  #[test]
  fn partial_swap_out() {
    let client = FarMemoryClient::new(Box::new(InMemoryBackend::new()), 30);
    let span = client.allocate_span(20);

    assert_eq!(20, client.total_local_memory());
    assert_eq!(0, client.total_remote_memory());

    client.ensure_local_memory_under_limit(15, true);
    assert_eq!(15, client.total_local_memory());
    assert_eq!(5, client.total_remote_memory());

    let _ptr = client.span_ptr(&span);
    assert_eq!(20, client.total_local_memory());
    assert_eq!(0, client.total_remote_memory());
  }

  #[test]
  fn partial_swap_out_multiple_parts() {
    let client = FarMemoryClient::new(Box::new(InMemoryBackend::new()), 30);
    let span = client.allocate_span(20);

    client.ensure_local_memory_under_limit(15, true);
    assert_eq!(15, client.total_local_memory());
    assert_eq!(5, client.total_remote_memory());

    client.ensure_local_memory_under_limit(10, true);
    assert_eq!(10, client.total_local_memory());
    assert_eq!(10, client.total_remote_memory());

    let _ptr = client.span_ptr(&span);
    assert_eq!(20, client.total_local_memory()); // first part (5) and second (5) were both swapped, so +10.
    assert_eq!(0, client.total_remote_memory());
  }
}
```

Файл ./src/client/replacement/mod.rs:

```
use {
  std::{sync::{atomic::{AtomicU64, Ordering}, RwLock}, collections::{HashMap, HashSet}, path::Path, fs},
  rand::seq::SliceRandom,
  itertools::Itertools,
  super::SpanId,
};

pub use {
  lru::LeastRecentlyUsedReplacementPolicy,
  replay::RemoteReplayReplacementPolicy,
  rnn::RnnReplacementPolicy,
  tracking::TrackingReplacementPolicy,
};

mod lru;
mod replay;
mod rnn;
mod tracking;

pub trait ReplacementPolicy: Send + Sync {
  fn pick_for_eviction(&self, spans: &[SpanId]) -> Box<dyn Iterator<Item = SpanId>>;

  fn on_new_span(&self, span_id: &SpanId) {}
  fn on_span_access(&self, span_id: &SpanId) {}
  fn on_span_swap_out(&self, span_id: &SpanId, partial: bool) {}
  fn on_span_swap_in(&self, span_id: &SpanId) {}
  fn on_stop(&self) {}
}

// 6.01 per token (for 25700)
pub struct RandomReplacementPolicy {
}

impl RandomReplacementPolicy {
  pub fn new() -> Self {
```

```
        Self {
        }
    }
}

impl ReplacementPolicy for RandomReplacementPolicy {
    fn pick_for_eviction<'a>(&self, spans: &[SpanId]) -> Box<dyn Iterator<Item = SpanId>> {
        let mut spans = spans.to_vec();
        spans.shuffle(&mut rand::thread_rng());
        Box::new(spans.into_iter())
    }
}

// 5.42 per token (for 25700)
// 12.31 per token (for 25600)
pub struct MostRecentlyUsedReplacementPolicy {
    counter: AtomicU64,
    history: RwLock<HashMap<SpanId, u64>>,
}

impl MostRecentlyUsedReplacementPolicy {
    pub fn new() -> Self {
        Self {
            counter: AtomicU64::new(0),
            history: RwLock::new(HashMap::new()),
        }
    }
}

impl ReplacementPolicy for MostRecentlyUsedReplacementPolicy {
    fn pick_for_eviction(&self, spans: &[SpanId]) -> Box<dyn Iterator<Item = SpanId>> {
        let history = self.history.read().unwrap();
        let spans: Vec<_> = spans.iter()
            .map(|v| (v, history.get(v).unwrap_or(&0)))
            .sorted_by_key(|v| 0 - v.1)
            .map(|a| a.0.clone())
            .collect();
        Box::new(spans.into_iter())
    }

    fn on_span_access(&self, span_id: &SpanId) {
        self.history.write().unwrap().insert(span_id.clone(), self.counter.fetch_add(1, Ordering::Relaxed));;
    }
}

// current best when combined with MostRecentlyUsedReplacementPolicy
// 8.34 per token (for 25600)
pub struct PreferRemoteSpansReplacementPolicy {
    remote_spans: RwLock<HashSet<SpanId>>,
    inner: Box<dyn ReplacementPolicy>,
}

impl PreferRemoteSpansReplacementPolicy {
    pub fn new(inner: Box<dyn ReplacementPolicy>) -> Self {
        Self {
            remote_spans: RwLock::new(HashSet::new()),
            inner,
        }
    }
}

impl ReplacementPolicy for PreferRemoteSpansReplacementPolicy {
    fn pick_for_eviction(&self, spans: &[SpanId]) -> Box<dyn Iterator<Item = SpanId>> {
        let inner_result: Vec<_> = self.inner.pick_for_eviction(spans).collect();

        let remote_spans: Vec<_> = {
            let remote_spans = self.remote_spans.try_read().unwrap();
            inner_result.iter().filter(|s| remote_spans.contains(s)).cloned().collect()
        };

        let local_spans: Vec<_> = {
            let remote_spans = self.remote_spans.try_read().unwrap();
            inner_result.iter().filter(|s| !remote_spans.contains(s)).cloned().collect()
        };

        Box::new(remote_spans.into_iter().chain(local_spans.into_iter()))
    }
```

```
    fn on_span_access(&self, span_id: &SpanId) {
        self.inner.on_span_access(span_id)
    }

    fn on_span_swap_in(&self, span_id: &SpanId) {
        self.inner.on_span_swap_in(span_id);
        self.remote_spans.write().unwrap().remove(span_id);
    }

    fn on_span_swap_out(&self, span_id: &SpanId, partial: bool) {
        self.inner.on_span_swap_out(span_id, partial);
        self.remote_spans.write().unwrap().insert(span_id.clone());
    }
}

pub struct ReplayReplacementPolicy {
    history_file_path: String,
    record_mode: bool,
    history: RwLock<Vec<SpanId>>,
    access_counter: AtomicU64,
    fallback: Box<dyn ReplacementPolicy>,
}

impl ReplayReplacementPolicy {
    pub fn new(fallback: Box<dyn ReplacementPolicy>) -> Self {
        let history_file_path = "./data/eviction_history.json".to_owned();

        let path = Path::new(&history_file_path);
        let record_mode = !path.exists();

        let history = if record_mode {
            Vec::new()
        } else {
            serde_json::from_slice(&fs::read(path).unwrap()).unwrap()
        };

        Self {
            history_file_path,
            record_mode,
            history: RwLock::new(history),
            access_counter: AtomicU64::new(0),
            fallback,
        }
    }
}

impl ReplacementPolicy for ReplayReplacementPolicy {
    fn pick_for_eviction(&self, spans: &[SpanId]) -> Box<dyn Iterator<Item = SpanId>> {
        if self.record_mode {
            return self.fallback.pick_for_eviction(spans);
        }

        let position = self.access_counter.load(Ordering::Relaxed);
        let history = self.history.read().unwrap();
        if position >= history.len() as u64 {
            return self.fallback.pick_for_eviction(spans);
        }

        // pick based on history
        let mut span_pos = vec![usize::MAX; spans.len()];
        for i in 0..spans.len() {
            for k in (position as usize)..history.len() {
                if history[k] == spans[i] {
                    span_pos[i] = k;
                    break;
                }
            }
        }

        let max_pos = span_pos.iter()
            .enumerate()
            .max_by(|(_, a), (_, b)| a.cmp(b))
            .map(|(index, _)| index)
            .unwrap();

        Box::new(vec![spans[max_pos].clone()].into_iter()) // todo: replace with proper iterator
    }
```

```
    fn on_span_access(&self, span_id: &SpanId) {
        if self.record_mode {
            self.history.write().unwrap().push(span_id.clone());
        } else {
            self.access_counter.fetch_add(1, Ordering::Relaxed);
        }

        self.fallback.on_span_access(span_id)
    }

    fn on_span_swap_out(&self, span_id: &SpanId, partial: bool) {
        self.fallback.on_span_swap_out(span_id, partial)
    }

    fn on_span_swap_in(&self, span_id: &SpanId) {
        self.fallback.on_span_swap_in(span_id)
    }

    fn on_stop(&self) {
        if self.record_mode {
            fs::write(&self.history_file_path, &serde_json::to_vec(&*self.history.read().unwrap()).unwrap()).unwrap();
        }
    }
}

pub fn run_replacement_policies_demo() {
    rnn::rnn_training_test();
}
```

Файл ./src/client/replacement/replay.rs:

```
use {
    std::sync::{atomic::{AtomicU64, Ordering}, RwLock},
    itertools::Itertools,
    crate::{
        manager::{ManagerClient, SpanAccessEvent, ReplacementPolicyType},
        client::SpanId,
    },
    super::ReplacementPolicy,
};

pub struct RemoteReplayReplacementPolicy {
    fallback: Box<dyn ReplacementPolicy>,
    manager: ManagerClient,

    step_counter: AtomicU64,
    span_access_events: RwLock<Vec<SpanAccessEvent>>,
}

impl RemoteReplayReplacementPolicy {
    pub fn new(manager: ManagerClient, fallback: Box<dyn ReplacementPolicy>) -> Self {
        let params = manager.get_replacement_policy_params(ReplacementPolicyType::Replay).span_access_history;

        Self {
            manager,
            fallback,

            step_counter: AtomicU64::new(0),
            span_access_events: RwLock::new(params.unwrap_or(Vec::new())),
        }
    }
}

impl ReplacementPolicy for RemoteReplayReplacementPolicy {
    fn pick_for_eviction(&self, spans: &[SpanId]) -> Box<dyn Iterator<Item = SpanId>> {
        {
            let span_access_events = self.span_access_events.read().unwrap();
            if !span_access_events.is_empty() {
                return pick_based_on_history(spans, &span_access_events);
            }
        }
        self.fallback.pick_for_eviction(spans)
    }

    fn on_span_access(&self, span_id: &SpanId) {
        {
            let mut span_access_events = self.span_access_events.write().unwrap();
```

```
            if !span_access_events.is_empty() {
                span_access_events.remove(0);
            }
        }

        self.manager.on_span_access(span_id, self.step_counter.fetch_add(1, Ordering::Relaxed));
        self.fallback.on_span_access(span_id)
    }

    fn on_span_swap_out(&self, span_id: &SpanId, partial: bool) {
        self.fallback.on_span_swap_out(span_id, partial)
    }

    fn on_span_swap_in(&self, span_id: &SpanId) {
        self.fallback.on_span_swap_in(span_id)
    }

    fn on_stop(&self) {
        // TODO: flush state
        self.fallback.on_stop()
    }
}

fn pick_based_on_history(spans: &[SpanId], events: &[SpanAccessEvent]) -> Box<dyn Iterator<Item = SpanId>> {
    let mut span_pos = vec![usize::MAX; spans.len()];
    for i in 0..spans.len() {
        for k in 0..events.len() {
            if events[k].span_id == spans[i].id() {
                span_pos[i] = k;
                break;
            }
        }
    }

    let spans = spans.to_vec();
    return Box::new(span_pos.into_iter()
        .enumerate()
        .sorted_by_key(|(_index, value)| *value)
        .map(move |(index, _value)| spans[index].clone()));
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn simple() {
        let events = vec![
            SpanAccessEvent {
                time_step: 0,
                span_id: 2,
            },
            SpanAccessEvent {
                time_step: 1,
                span_id: 0,
            },
            SpanAccessEvent {
                time_step: 2,
                span_id: 1,
            }
        ];

        let spans = vec![
            SpanId::from_id(0),
            SpanId::from_id(1),
            SpanId::from_id(2),
        ];

        let mut spans_to_swap_out = pick_based_on_history(&spans, &events);
        assert_eq!(Some(SpanId::from_id(2)), spans_to_swap_out.next());
        assert_eq!(Some(SpanId::from_id(0)), spans_to_swap_out.next());
        assert_eq!(Some(SpanId::from_id(1)), spans_to_swap_out.next());
        assert_eq!(None, spans_to_swap_out.next());
    }
}
```

Файл ./src/client/object.rs:

```
use {
    std::{
        sync::{atomic::{AtomicU64, Ordering}, RwLock, Mutex},
        collections::HashMap,
        marker::PhantomData,
        ops::Deref,
    },
    super::{
        span::SpanId,
        client::FarMemoryClient,
    },
};

#[derive(Clone, Eq, PartialEq, Hash)]
pub struct ObjectId(u64);

#[derive(Clone)]
pub struct ObjectLocation {
    pub span_id: SpanId,
    pub offset: usize,
    pub len: usize,
}

impl ObjectLocation {
    pub fn new(span_id: SpanId, offset: usize, len: usize) -> Self {
        Self {
            span_id,
            offset,
            len,
        }
    }
}

pub struct ObjectSlot {
    span_id: SpanId,
    offset: usize,
    len: usize,
}

pub struct ObjectRegistry {
    object_id_counter: AtomicU64,

    object_mapping: RwLock<HashMap<ObjectId, ObjectLocation>>,
    slots_by_size_class: Mutex<HashMap<usize, Vec<ObjectSlot>>>,
}

impl ObjectRegistry {
    pub fn new() -> Self {
        Self {
            object_id_counter: AtomicU64::new(0),
            object_mapping: RwLock::new(HashMap::new()),
            slots_by_size_class: Mutex::new(HashMap::new()),
        }
    }

    pub fn next_object_id(&self) -> ObjectId {
        ObjectId(self.object_id_counter.fetch_add(1, Ordering::Relaxed))
    }

    pub fn put_object(&self, object_id: ObjectId, object_size: usize) -> Option<ObjectLocation> {
        let size_class = self.size_class_for_object(object_size);
        let mut slots_map = self.slots_by_size_class.lock().unwrap();
        let slots_by_size_class = match slots_map.get_mut(&size_class) {
            Some(v) => v,
            None => return None, // no spans were allocated for this size class yet
        };
        if slots_by_size_class.is_empty() {
            // no free space left in this size class
            return None;
        }

        let slot = if slots_by_size_class[0].len > object_size {
            let remaining_len = slots_by_size_class[0].len - size_class;

            if remaining_len > 0 {
                let remaining = ObjectSlot {
                    span_id: slots_by_size_class[0].span_id.clone(),
                    offset: slots_by_size_class[0].offset + size_class,
```

```
                len: remaining_len,
            };
            std::mem::replace(&mut slots_by_size_class[0], remaining)
        } else {
            slots_by_size_class.remove(0)
        }
    } else if slots_by_size_class[0].len == object_size {
        slots_by_size_class.remove(0)
    } else {
        panic!("it is not expected that slot size is smaller than object size: {}, size class is {}", slots_by_size_class[0].len, size_class);
    };

    let location = ObjectLocation {
        span_id: slot.span_id,
        offset: slot.offset,
        len: object_size,
    };
    self.object_mapping.write().unwrap().insert(object_id, location.clone());

    Some(location)
}

pub fn add_span_for_object(&self, span_id: SpanId, span_size: usize, object_id: ObjectId, object_size: usize) -> ObjectLocation {
    let size_class = self.size_class_for_object(object_size);
    {
        let mut slots_by_size_class = self.slots_by_size_class.lock().unwrap();

        if !slots_by_size_class.contains_key(&size_class) {
            slots_by_size_class.insert(size_class, vec![]);
        }

        slots_by_size_class.get_mut(&size_class).unwrap().push(ObjectSlot {
            span_id,
            offset: 0,
            len: span_size,
        });
    }

    self.put_object(object_id, object_size).unwrap()
}

pub fn get_object(&self, object_id: &ObjectId) -> ObjectLocation {
    self.object_mapping.read().unwrap().get(object_id).unwrap().clone()
}

pub fn size_class_for_object(&self, object_size: usize) -> usize {
    // TODO: implement actual size classes

    // for flights in dataframe demo
    if object_size >= 110 && object_size <= 120 {
        return 120;
    } else if object_size >= 310 && object_size <= 400 {
        return 400;
    } else if object_size >= 400 && object_size <= 500 {
        return 500;
    }

    if object_size != 8200 && object_size != 8 {
        panic!("this object size is not supported: {:?}", object_size);
    }

    object_size
}
}

pub struct FarMemory<T> {
    client: FarMemoryClient,
    object: ObjectId,
    _phantom: PhantomData<T>,
}

impl<T> FarMemory<T> {
    pub fn from_value(client: FarMemoryClient, value: T) -> Self {
        let object = client.put_object(unsafe {
            std::slice::from_raw_parts(
                (&value as *const _) as *const u8,
                std::mem::size_of::<T>()
            ).to_vec()
```

```
        });

        Self {
            client,
            object,
            _phantom: PhantomData,
        }
    }

    pub fn to_local(&self) -> FarMemoryLocal<T> {
        FarMemoryLocal {
            client: self.client.clone(),
            object: self.object.clone(),
            _phantom: PhantomData,
        }
    }
}

impl<T> Deref for FarMemory<T> {
    type Target = FarMemoryLocal<T>;

    fn deref(&self) -> &Self::Target {
        unimplemented!()
    }
}

pub struct FarMemoryLocal<T> {
    client: FarMemoryClient,
    object: ObjectId,
    _phantom: PhantomData<T>,
}

impl<T> Deref for FarMemoryLocal<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        let location = self.client.get_object(&self.object);
        unsafe {
            &*(self.client.span_ptr(&location.span_id).add(location.offset) as *const T)
        }
    }
}

impl<T> Drop for FarMemoryLocal<T> {
    fn drop(&mut self) {
        self.client.decrease_refs_for_span(&self.client.get_object(&self.object).span_id);
    }
}

#[cfg(test)]
mod tests {
    use {
        crate::client::InMemoryBackend,
        super::*,
    };

    struct TestValue {
        v: u64,
    }

    #[test]
    fn simple() {
        let client = FarMemoryClient::new(Box::new(InMemoryBackend::new()), 10 * 1024 * 1024);
        let object = FarMemory::from_value(client, TestValue { v: 42 });

        assert_eq!(42, object.to_local().v);
    }
}
```

Файл ./src/client/vec.rs:

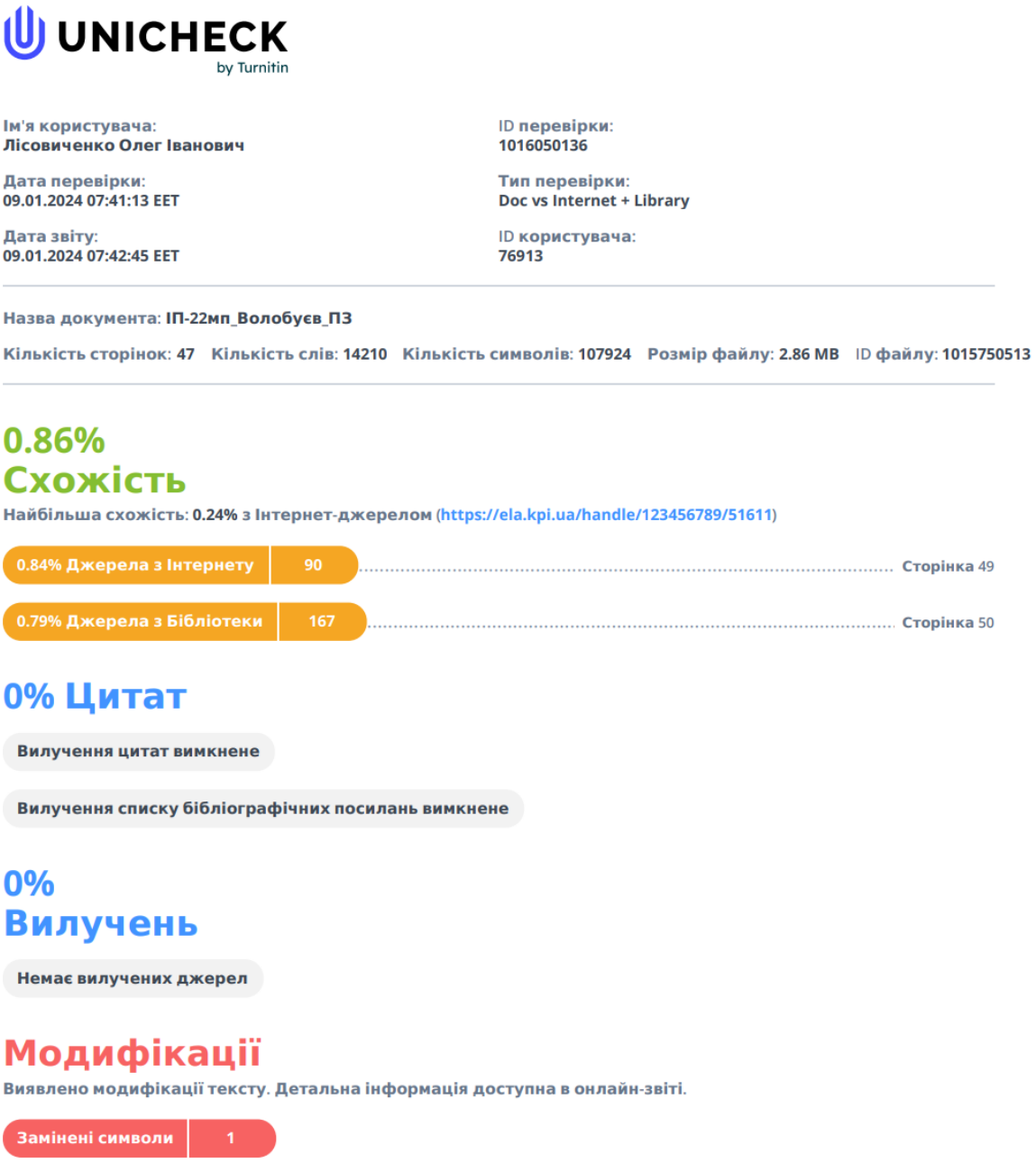
```
use {
    std::{ops::Deref, fmt::Debug, marker::PhantomData},
    tracing::{span, Level},
    super::{FarMemoryClient, span::SpanId},
};
```

```
pub struct FarMemoryVec<T> {
    client: FarMemoryClient,
    span: SpanId,
    len: usize,

    _phantom: PhantomData<T>,
}

pub struct FarMemoryLocalVec<T> {
    client: FarMemoryClient,
    span: SpanId,
    vec: Vec<T>,
}

impl <T> FarMemoryVec<T> {
    pub fn from_vec(client: FarMemoryClient, vec: Vec<T>) -> Self {
        let size = std::mem::size_of::<T>() * vec.len();
        let span = client.allocate_span(size);
        let ptr = client.span_ptr(&span);
        // this can probably be optimized by taking ptr and giving it to client, instead of
        // allocation and copy
        unsafe {
            std::ptr::copy_nonoverlapping(vec.as_ptr() as *const _, ptr, size);
        }
        client.decrease_refs_for_span(&span);

        Self {
            client,
            span,
            len: vec.len(),

            _phantom: PhantomData,
        }
    }

    pub fn to_local_vec(&self) -> FarMemoryLocalVec<T> {
        span!(Level::DEBUG, "FarMemoryVec::to_local_vec", span_id=self.span.id()).in_scope(|| {
            let ptr = self.client.span_ptr(&self.span) as *const T;
            if self.len * std::mem::size_of::<T>() != self.client.span_local_memory_usage(&self.span) {
                panic!("memory needed for mem does not match size of memory allocated");
            }

            FarMemoryLocalVec {
                client: self.client.clone(),
                span: self.span.clone(),
                vec: unsafe { Vec::from_raw_parts(ptr as *mut T, self.len, self.len) },
            }
        })
    }
}

impl<T> Deref for FarMemoryLocalVec<T> {
    type Target = Vec<T>;

    fn deref(&self) -> &Self::Target {
        &self.vec
    }
}

impl<T: Debug> Debug for FarMemoryLocalVec<T> {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.deref().fmt(f)
    }
}

impl<T: std::cmp::PartialEq> PartialEq<Vec<T>> for FarMemoryLocalVec<T> {
    fn eq(&self, other: &Vec<T>) -> bool {
        self.deref() == other
    }
}

impl<T: std::cmp::PartialEq> PartialEq<FarMemoryLocalVec<T>> for Vec<T> {
    fn eq(&self, other: &FarMemoryLocalVec<T>) -> bool {
        self == other.deref()
    }
}

impl<T> Drop for FarMemoryLocalVec<T> {
```

```
    fn drop(&mut self) {
        let mut v = Vec::new();
        std::mem::swap(&mut self.vec, &mut v);
        std::mem::forget(v);
        self.client.decrease_refs_for_span(&self.span);
    }
}

#[cfg(test)]
mod tests {
    use {
        crate::client::InMemoryBackend,
        super::*,
    };

    #[test]
    fn to_local_vec() {
        let vec = FarMemoryVec::from_vec(
            FarMemoryClient::new(Box::new(InMemoryBackend::new()), 1000),
            vec![10.02, 9.02, 8.02, 7.02, 6.02, 5.02, 4.02, 3.02, 2.02, 1.02]
        );

        assert_eq!(
            vec![10.02, 9.02, 8.02, 7.02, 6.02, 5.02, 4.02, 3.02, 2.02, 1.02],
            vec.to_local_vec()
        );
    }

    #[test]
    fn to_local_vec_no_double_free() {
        let vec = FarMemoryVec::from_vec(
            FarMemoryClient::new(Box::new(InMemoryBackend::new()), 1000),
            vec![10.02, 9.02, 8.02, 7.02, 6.02, 5.02, 4.02, 3.02, 2.02, 1.02]
        );

        assert_eq!(
            vec![10.02, 9.02, 8.02, 7.02, 6.02, 5.02, 4.02, 3.02, 2.02, 1.02],
            vec.to_local_vec()
        );

        assert_eq!(
            vec![10.02, 9.02, 8.02, 7.02, 6.02, 5.02, 4.02, 3.02, 2.02, 1.02],
            vec.to_local_vec()
        );
    }
}
```

ДОДАТОК Ж

Результати перевірки роботи на співпадіння



UNICHECK
by Turnitin

Ім'я користувача:
Лісовиченко Олег Іванович

ID перевірки:
1016050136

Дата перевірки:
09.01.2024 07:41:13 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
09.01.2024 07:42:45 EET

ID користувача:
76913

Назва документа: **ІП-22мп_Волобуєв_ПЗ**

Кількість сторінок: **47** Кількість слів: **14210** Кількість символів: **107924** Розмір файлу: **2.86 MB** ID файлу: **1015750513**

0.86%
Схожість

Найбільша схожість: **0.24%** з Інтернет-джерелом (https://ela.kpi.ua/handle/123456789/51611)

0.84% Джерела з Інтернету 90 ..... Сторінка 49

0.79% Джерела з Бібліотеки 167 ..... Сторінка 50

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи 1