УДК 004.08
DOI https://doi.org/10.32782/2663-5941/2024.1.1/21

Киричек Г.Г.
Національний університет «Запорізька політехніка»

Тягунова М.Ю.
Національний університет «Запорізька політехніка»

Братчиков В.В.
Національний університет «Запорізька політехніка»

СИСТЕМА КЕШУВАННЯ ДАНИХ В РОЗГАЛУЖЕНІЙ МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

На даний час технологія кешування даних є ключовою в інформаційних системах при забезпеченні їх швидкодії та ефективності. Вона включає збереження копій даних у сховищах швидкої дії, таких як оперативна пам'ять, з метою подальшого доступу до них, без необхідності повторного запиту первинних джерел інформації. Метою роботи є дослідження методів та засобів автоматизації кешування даних у розгалуженій мікросервісній архітектурі та реалізація системи автоматизованого кешування для веб-платформ з метою впровадження етапів кешування даних. Об'єктом дослідження є процес реалізації високопродуктивної системи кешування даних для розгалуженої мікросервісної архітектури із використанням архітектури token ring та інструментів синхронізації між нодами. Предметом дослідження є моделі, методи та засоби автоматизації кешування даних у розгалуженій мікросервісній архітектурі. Система реалізована із використанням архітектури token ring та має механізми синхронізації між вузлами. Для створення моделі системи та реалізації кешування обрано фреймворк Spring Boot та інструмент Maven у середовищі Intelijj IDEA. Процес розгалуженого кешування виконаний за допомогою Maven, Docker та із використанням власного реалізованого механізму кешування та синхронізації даних для розгалуженої мікросервісної архітектури. Для реалізації системи проаналізовані три популярні мови програмування: Java, Phyton, C# за чотирма основними критеріями: швидкодія, екосистема, підтримка розподілених систем, масштабованість та підтримка операцій. Враховуючи переваги в швидкості, можливості масштабування та наявність великої кількості бібліотек обрана мова програмування Java. Авторами проведено дослідження продуктивності системи, в порівнянні з існуючими рішеннями на основі аналізу швидкісних характеристик використання механізму кешування. Реалізована високопродуктивна система кешування даних в розгалуженій мікросервісній архітектурі істотно підвищує якість послуг, які дозволяють зберігати та отримувати доступ до даних швидше, зменшуючи навантаження на сервери та оптимізуючи мережеві ресурси.

Ключові слова: *docker, maven, redis, spring, кластер, оптимізація.*

Постановка проблеми. Сучасний світ інформаційних технологій відзначається надзвичайною динамікою розвитку та нескінченним потоком даних, які потребують обробки та надійного збереження. Переваги мікросервісної архітектури, що дозволяє будувати розподілені та масштабовані системи є незаперечними [1]. Проте, разом із зростанням об'ємів додатків і збільшенням потоку даних, виникає необхідність в ефективному управлінні цими даними та доступі до них. Тому важливою складовою для забезпечення високої продуктивності в мікросервісній архітектурі є системи кешування даних. Такі системи визначаються як критичний аспект в забезпеченні швидкодії мікросервісних додатків, особливо в умовах високого навантаження та даних, які надзвичайно швидко змінюються [2]. Кешування дозволяє зберігати ці дані та мати до них доступ швидше, зменшуючи навантаження на сервери та оптимізуючи ресурси [3]. Memcached використовує клієнт-серверну архітектуру та зберігає дані у форматі ключ-значення, де кожен ключ пов'язаний з певним значенням. Для реалізації системи кешування найкраще використовувати перевірені та надійні мови програмування і фреймворки [4], а також потрібно добре розумітися в побудові архітектур та знати декілька мов програмування: Java, Javascript, Phyton, або C# а також вміти конфігурувати кластер із серверів, наприклад використовуючи docker-compose [5].

Середовище IntelliJ IDEA, при реалізації та виконанні коду, аналізує його, виявляє помилки та пропонує належні рішення, а також будує синтаксичне дерево відразу ж при вводі коду [4].

Аналіз останніх досліджень та публікацій. Системи збереження та кешування даних відіграють важливу роль у підвищенні продуктивності та швидкості доступу до інформації [2]. Маємо основні поняття та визначення, пов'язані із подібними системами, включаючи Redis (Remote Dictionary Server), Apache Ignite та Memcached. Redis є дистриб'ютивною системою кешування та збереження даних у пам'яті, яка використовує ключ-значення при збереженні даних та доступу до інформації [6]. Вона використовує клієнт-серверну архітектуру і підтримує різні типи даних, включаючи рядки, хеші (асоціативні масиви), списки, множини та сортовані множини [7]. Apache Ignite є розподіленою системою для збереження даних, яка поєднує в собі функції кешування та обчислення. Архітектура Apache Ignite робить його потужним інструментом при реалізації додатків, які вимагають збільшення швидкості доступу до даних, проведення розподілених операцій і аналітики, дозволяючи покращити продуктивність та надійність додатків, які обробляють великі обсяги даних [6]. Memcached (Memory Cache Daemon) є простою та ефективною системою кешування даних, яка базується на розподіленій пам'яті та відома швидкістю доступу до інформації і легкістю використання [2].

При необхідності синхронізувати дані між нодами, в мікросервісних архітектурах застосовується декілька базових алгоритмів [8]. Алгоритм Лівайна для LSM-дерев (Log-Structured Merge Tree) застосовується в розподілених системах, використовуючи підходи мультиверсійності для оптимізації читання та запису в розподіленому середовищі. Основною перевагою є висока продуктивність та швидкий доступ до даних, що робить цей алгоритм ефективним для розподілених систем з великим обсягом даних. Алгоритм Шутдайна (Conflict-Free Replicated Data Types (CRDT)) є класом алгоритмів з відсутністю конфліктів, які дозволяють синхронізувати дані без необхідності вирішення конфліктів в системах, де можуть виникати асинхронні зміни [9]. І алгоритм Vector Clocks застосовується для визначення порядку подій в розподіленому середовищі. Кожна нода має свій вектор часу, який використовується для визначення, які зміни вже враховані в системі. Перевагами є визначення порядку подій та зменшення конфліктів [10].

Метою роботи є дослідження методів та засобів автоматизації кешування даних у розгалуженій мікросервісній архітектурі та реалізація системи автоматизованого кешування для веб-платформ з метою впровадження етапів кешування даних. Об'єктом дослідження є процес реалізації високопродуктивної системи кешування даних для розгалуженої мікросервісної архітектури з використанням архітектури token ring та інструментів синхронізації між нодами. Предметом дослідження є моделі, методи, інструментальні та програмні засоби автоматизації кешування даних у розгалуженій мікросервісній архітектурі.

У рамках дослідження запропоновано логічну модель системи автоматизованого кешування даних. Процес реалізації високопродуктивної та надійної системи, спрямований на вирішення конкретних завдань із синхронізацією на основі архітектури token ring. Архітектура дозволяє підтримувати актуальність даних та забезпечувати консистентність системи навіть в розподіленому середовищі. Під час аналізу існуючих систем та мов програмування враховані найкращі практики та рекомендації із використання Redis, Apache Ignite та Memcached [7–9]. Реалізована система повинна відповідати вимогам надійності та продуктивності і бути інструментом для застосувань в розгалуженій мікросервісній архітектурі, дозволяючи отримувати швидкий доступ на запис та зчитування даних з кешу.

Виклад основного матеріалу. Система кешування даних між мікросервісами використовує вдосконалений алгоритм синхронізації для ефективного управління і підтримки актуальності даних між різними кеш-нодами. Основна мета цього алгоритму полягає в зниженні затримок, пов'язаних із мережею, і забезпеченні консистентності інформації, розподіленої між різними складовими системи. Алгоритм реалізований таким чином, щоб забезпечити ефективне керування синхронізацією даних і уникнути надмірної завантаженості мережі [11]. Це сприяє підвищенню ефективності системи кешування та забезпечує мінімізацію затримок при отриманні актуальних даних. Такий підхід дозволяє системі оперативно реагувати на зміни в даних та забезпечує їх узгодженість в усьому середовищі мікросервісної архітектури [12]. Основні кроки алгоритму включають такі дії: визначення сусідніх нод; ініціалізація синхронізації; обмін даними; підтвердження, оновлення та актуалізація [10].

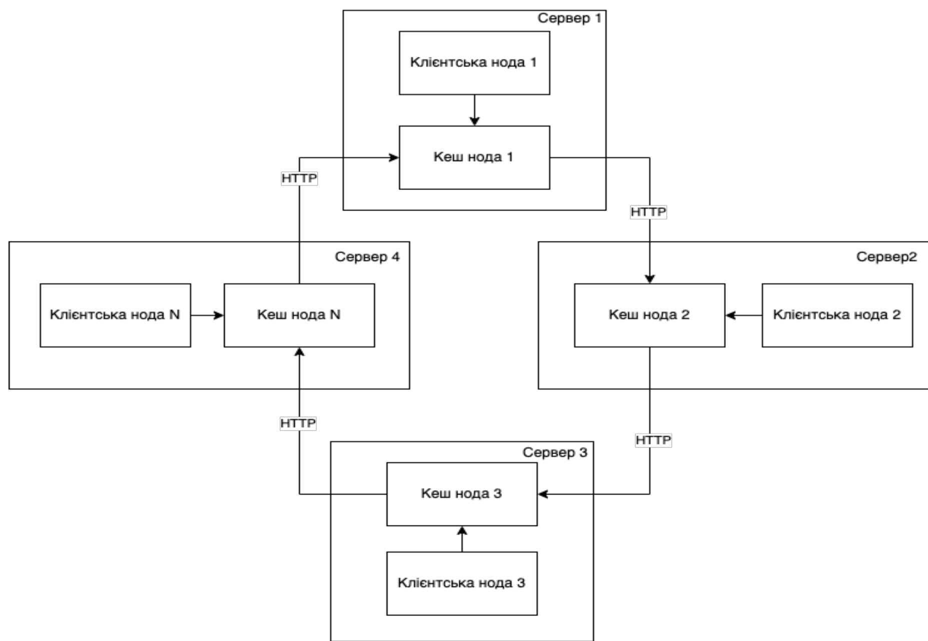

Рис. 1. Логічна схема системи

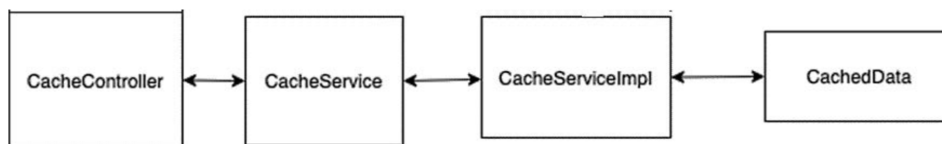

Рис. 2. Логічна схема модулів підсистеми «кеш нода»

Логічна схема системи зображена на рисунку 1.

Реалізація системи застосовує Java [13], Spring Boot та використовує архітектуру MVC, наведемо структуру її основних модулів (рис. 2).

Для створення кеш-ноди, яка передає дані по запиту клієнта та синхронізується із сусідньою нодою за принципом token-ring, реалізуємо модулі для кеш-ноди. CacheController.java є класом контролера, що відповідає за обробку HTTP-запитів від клієнтів та за відповідь на них . Він містить логіку для обміну даними з сусідніми нодами за принципом token-ring. Загалом, CacheController виступає як посередник між клієнтами та кеш-сервісом:

```
@RestController
@RequestMapping("node")
@RequiredArgsConstructor
public class CacheController {
private final CacheService cacheService;
@Value("${cache.node.force.delay}")
private Long forceDelayTime;
@SneakyThrows
@PostMapping("{initiator}")
public ResponseEntity<Object> sync(@RequestBody
CachedData data, @PathVariable String initiator) {
sleep(forceDelayTime);
cacheService.update(initiator, data);
return ResponseEntity.ok("");     }     }
```

CacheStorageService.java, клас зберігає локальні дані кешу в оперативній пам'яті та надає доступ до цього сховища:

```
@Service
@Getter
public class CacheStorageService {
private final Map<String, CachedData> DATA
= new HashMap<>();     }
```

CacheService.java визначає методи для роботи з кешем, включаючи збереження, отримання та оновлення даних [14]. В ньому визначена логіка для синхронізації з сусідніми нодами:

```
public interface CacheService {
 void sync(String initiator, CachedData data);
 void update(String initiator, CachedData data);
}
```

CacheServiceImpl.java, клас реалізує інтерфейс 'CacheService' та містить логіку для роботи з кешем, включаючи логіку синхронізації з сусідніми нодами за принципом token-ring. Для формування дистрибутиву та керування залежностями використовуємо фреймворк Maven.

Модуль тестування даних запускається окремо і налаштовується для кожної ноди. Цей модуль здатен проводити тестування з навантаженням, створювати нові записи у кеші та зчитувати їх,

а також показувати середній час операцій. Для більшої гнучкості тестування та можливості аналізу даних додані спеціальні параметри для симуляції затримки мережі. Для комплексної перевірки реалізовано тестовий стенд, на базі docker контейнерів [5], які об'єднанні у спільну мережу за допомогою технології docker-compose.

Маючи дані тестувань наведемо показники швидкості систем у вигляді таблиці. Для цього проведено по три тестування, для кожної з трьох нод (табл. 1), записів 1000 генерованих даних і 1000 генерованих даних зі штучною затримкою та по три зчитування, для кожної з трьох нод (табл. 2), 1000 тестових даних і 1000 тестових даних зі штучною затримкою.

Таблиця 1

Результат з тестування запису 1000 генерованих даних

№	Нода	Середній час обробки одного запису, мс	Штучна затримка, мс	Середній час обробки одного запису, мс
1	Нода 1	186	60	241
2	Нода 1	182	60	244
3	Нода 1	182	60	250
4	Нода 2	184	60	243
5	Нода 2	188	60	246
6	Нода 2	185	60	244
7	Нода 3	182	60	342
8	Нода 3	184	60	346
9	Нода 3	182	60	248

Таблиця 2

Результат з тестування зчитування 1000 тестових даних

№	Нода	Середній час обробки одного запису, мс	Штучна затримка, мс	Середній час обробки одного запису, мс
1	Нода 1	2	60	62
2	Нода 1	2	60	62
3	Нода 1	3	60	63
4	Нода 2	4	60	62
5	Нода 2	2	60	64
6	Нода 2	2	60	62
7	Нода 3	3	60	64
8	Нода 3	2	60	66
9	Нода 3	4	60	62

Після отримання результатів впровадженої системи стенд змінено на стенд на базі Redis і проведені аналогічні тести.

Результати тестів кеш системи на базі Redis наведено в таблицях 3 та 4.

Таблиця 3

Результат запису 1000 генерованих даних на базі Redis

№	Нода	Середній час обробки 1 запису, мс	Штучна затримка, мс	Середній час обробки 1 запису, мс
1	Нода 1	24	60	90
2	Нода 1	30	60	88
3	Нода 1	28	60	96
4	Нода 2	40	60	102
5	Нода 2	36	60	94
6	Нода 2	38	60	88
7	Нода 3	42	60	92
8	Нода 3	48	60	94
9	Нода 3	48	60	100

Таблиця 4

Результат зчитування 1000 тестових даних на базі Redis

№	Нода	Середній час обробки 1 запису, мс	Штучна затримка, мс	Середній час обробки 1 запису, мс
1	Нода 1	10	60	78
2	Нода 1	12	60	72
3	Нода 1	8	60	76
4	Нода 2	16	60	76
5	Нода 2	14	60	72
6	Нода 2	10	60	80
7	Нода 3	16	60	76
8	Нода 3	14	60	78
9	Нода 3	10	60	78

З метою агрегації результатів і наведення висновків, середні значення по всім операціям наведені у таблиці 5.

Таблиця 5

Таблиця зі зведеним результатом вимірювань

№	Нода	Тип операції	Середній час (реалізація), мс	Середній час (Redis), мс
1	Нода 1	Запис	215	47
2	Нода 2	Запис	216	45
3	Нода 3	Запис	212	48
1	Нода 1	Зчитування	32	40
2	Нода 2	Зчитування	34	44
3	Нода 3	Зчитування	32	46

На рисунках 3 і 4 наведено порівняння результатів запису та зчитування.

Наведені результати вимірювань дозволяють зробити висновки, що попри зменшення швидкості запису нових даних в впровадженій системі досягається приріст в швидкості зчитування даних близько 25%, що є вагомою перевагою для систем з високим навантаженням.

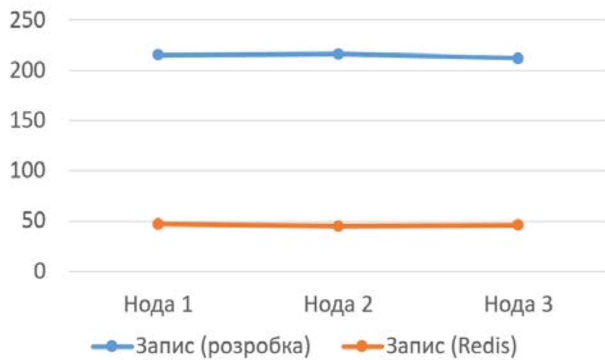

Рис. 3. Графік порівняння результатів запису

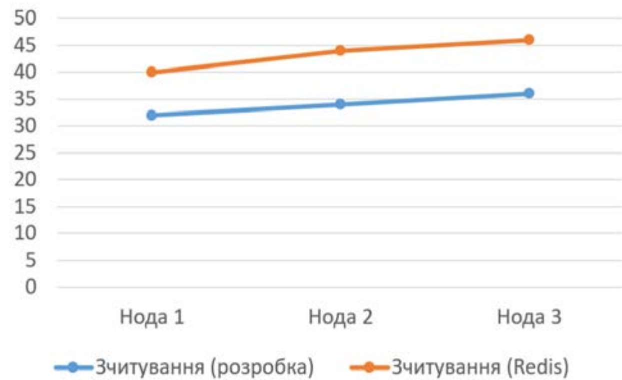

Рис. 4. Графік порівняння результатів зчитування

Висновки. У роботі проведено дослідження актуальних на сьогодні методів та алгоритмів реалізації систем кешування даних з використанням різних архітектур та інструментів синхронізації між нодами. У процесі дослідження реалізовано функціональну та високопродуктивну систему, спрямовану на вирішення конкретних завдань. Синхронізація в системі реалізована на основі архітектури token ring та працює надійно. Архітектура дозволяє підтримувати актуальність даних та забезпечувати консистентність системи навіть в розподіленому середовищі. Під час аналізу існуючих систем та мов програмування враховані найкращі практики та рекомендації з використання Redis, Apache Ignite та Memcached. Реалізована система відповідає вимогам, забезпечує надійність та продуктивність і є цінним інструментом для застосувань в розгалуженій мікросервісній архітектурі. Вона дозволяє отримати швидкий доступ на зчитування даних з кешу, так як дані зберігаються локально в кожній ноді і немає необхідності витрачати ресурси на звернення до іншого ресурсу в мережі. Авторами проведено тестування системи та дослідження швидкісних характеристик реалізованої системи у порівнянні з іншим аналогом. Отримані часові характеристики у результаті тестування: для запису даних t = 214 мс для реалізованої системи та t = 45 мс для системи на базі Redis, а також для зчитування даних t = 32 мс для реалізованої системи та t = 44 мс для системи на базі Redis.

Список літератури:

1. Киричек Г.Г., Щетінін М.О. Конфігурація серверів з використанням Ansible. *Modern scientific research: achievements, innovations and development prospects: International scientific conference. Riga*, 2021. P. 15–17.
2. Bazylevych R., Burtnyk R. Algorithms for software clustering and modularization. *In CSIT-2015,* P. 30–33.
3. Киричек Г.Г., Гаркуша В.Ю. Віртуалізація хостів на основі Proxmox VE в умовах надлишкового використання ресурсів. *Вчені записки ТНУ імені В.І. Вернадського. Серія «Технічні науки»*. 2021. Вип. 32 (71). № 1. С. 78–84.
4. Рудьковський О.Р., Киричек Г.Г. Програмний комплекс з підтримки розподіленої взаємодії мережевих пристроїв та додатків. *Вчені записки ТНУ ім. В.І. Вернадського. Серія «Технічні науки»*. 2021. Вип. 32(71). № 2. С. 229–234.
5. Nickoloff J., Kuenzli S. Docker in action. *Simon and Schuster,* 2019.
6. Acharya S. Apache Ignite Quick Start Guide: Distributed data caching and processing made easy. *Packt Publishing Ltd*, 2018.
7. Nelson J. Mastering redis. *Packt Publishing Ltd*, 2016.
8. Richardson C. Microservices patterns: with examples in Java. *Simon and Schuster*, 2018.
9. Walls C. Spring in action. Simon and Schuster, 2022.
10. Newman S. Building microservices. *O'Reilly Media, Inc*, 2021.
11. Kirichek, G., Kyrychek, D., Hrushko, S., Timenko, A. Implementation the Protection Method of Data Transmission in Network. *In: ATIT-2019*, P. 29–132.
12. Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, DA, Lynn, T. Microservices migration patterns. *Software: Practice and Experience*, 48 (11), 2018. P. 2019–2042.
13. Horstmann C. S. Core Java SE 9 for the impatient. *Addison-Wesley Professional,* 2017.
14. Bloch J. Effective java. *Addison-Wesley Professional*, 2017.

Kyrychek H.H., Tiahunova M.Yu., Bratchykov V.V. **DATA CACHING SYSTEM IN DISTRIBUTED MICROSERVICE ARCHITECTURE**

Currently, data caching technology is key in information systems while ensuring their speed and efficiency. It involves saving copies of data in high-speed storage, such as RAM, in order to access them later, without the need to re-request the original sources of information. The purpose of the work is to research the methods and data caching automation tools in a branched microservice architecture and the implementation of an automated caching system for web platforms with the aim of implementing stages of data caching. The object of research is the process of implementing a high-performance data caching system for a branched microservice architecture using the token ring architecture and synchronization tools between nodes. The subject of research are models, methods and data caching automation tools in a branched microservice architecture. The system is implemented using the token ring architecture and has synchronization mechanisms between nodes. To create a system model and implement caching, the Spring Boot framework and the Maven tool in the Intelijj IDEA environment were chosen. The distributed caching process is implemented using Maven, Docker, and using our own implemented data caching and synchronization mechanism for distributed microservice architecture. To implement the system, three popular programming languages: Java, Python, C# were analyzed according to four main criteria: speed, ecosystem, support for distributed systems, scalability and support for operations. Given the advantages in speed, scalability and the availability of a large number of libraries, the Java programming language was chosen. The authors conducted a study of the system performance in comparison with existing solutions based on the analysis of the speed characteristics of the caching mechanism use. The implemented high-performance data caching system in a branched microservice architecture significantly improves the quality of services that allow you to store and access data faster, reducing the load on servers and optimizing network resources.

Key words: *docker, maven, redis, spring, cluster, optimization.*