

РОЗВИТОК ІГРОВОЇ МОДЕЛІ ОПТИМІЗАЦІЇ МІКРОСЕРВІСІВ ДЛЯ ВЕЛИКИХ СИСТЕМ ІЗ УРАХУВАННЯМ ЗАЛЕЖНОСТЕЙ МІЖ СЕРВІСАМИ

В. Кулагін, здобувач третього рівня вищої освіти
Наук. керівник: **О. Улічев**, кандидат технічних наук, доцент
Центральноукраїнський національний технічний університет

Мікросервісна архітектура є одним із провідних підходів до побудови сучасних програмних систем завдяки можливості незалежного розгортання компонентів, гнучкого масштабування та підвищення керованості складних застосунків [1, 2]. Однак із переходом до мікросервісного стилю виникає новий клас задач, пов'язаних не лише з розподілом обчислювальних ресурсів, а й із координацією великої кількості сервісів, що одночасно співпрацюють у межах бізнес-процесів і конкурують за CPU, пам'ять, мережу та пропускну здатність інфраструктури [1, 3, 4]. Особливої складності ці задачі набувають у великих системах, де один користувацький запит проходить через багато сервісів, а деградація навіть одного вузла може різко збільшити наскрізну затримку [5, 6].

У статті [1] запропоновано підхід до оптимізації роботи мікросервісів на основі теорії ігор, у якому кожний мікросервіс розглядається як гравець, що прагне оптимізувати власну вигоду при спільному використанні обмежених ресурсів. У роботі побудовано модель некооперативної гри, використано ідею рівноваги Неша та запропоновано Game Theory Controller, який на основі метрик навантаження обчислює новий розподіл ресурсів між сервісами. За результатами моделювання, наведений підхід демонструє кращий середній час відповіді, ніж евристичні Binpack і Spread. Водночас автори прямо зазначають, що подальші дослідження мають бути спрямовані на поширення підходу на великі системи та на врахування взаємозв'язків між сервісами у вигляді гри на графі викликів [1].

Першою проблемою є масштабування моделі на великі системи. У невеликих експериментальних конфігураціях кількість сервісів і стратегій є відносно обмеженою, тому ітеративний пошук кращих відповідей є обчислюваним. Проте у великому мікросервісному застосунку конфігураційний простір різко зростає: якщо застосунок має m сервісів, а кожен сервіс може працювати з n варіантами виділення CPU, то кількість можливих конфігурацій становить n^m . Додатково проблему ускладнює той факт, що мікросервіси мають складну топологію взаємодій, а однаковий обсяг ресурсів, розподілений по-різному між сервісами, може давати істотно різний результат за кінцевою латентністю [5]. Це означає, що модель [1] потребує не просто збільшення числа агентів у симуляції, а зміни самої процедури прийняття рішень.

Одним із практичних шляхів розв'язання цієї проблеми є ієрархічна декомпозиція системи. Замість однієї глобальної гри для всіх сервісів доцільно формувати дворівневу схему. На верхньому рівні ресурси розподіляються між доменами, підсистемами або групами мікросервісів, наприклад між контуром автентифікації, контуром каталогу, контуром платежів і контуром аналітики. На нижньому рівні вже всередині кожної групи виконується локальна гра між сервісами, що входять до відповідного bounded context. Така декомпозиція зменшує розмір простору стратегій, дозволяє запускати оптимізацію паралельно і краще відповідає реальній структурі великих платформ, де сервіси організовані в домени, а не існують як повністю однорідна множина. Цей підхід логічно розвиває ідею [1] без втрати її ігрової природи.

Другим варіантом є обмеження множини стратегій до практично значущих профілів масштабування. Замість безперервного або надто широкого вибору ресурсів для кожного сервісу можна використовувати скінченний набір профілів, наприклад: “мінімальний”, “номінальний”, “піковий”, “критичний”. У такому разі гра виконується не над усіма можливими

значеннями CPU та пам'яті, а над невеликим каталогом допустимих конфігурацій. Для великих систем це є важливим, тому що дозволяє поєднати теоретико-ігровий механізм із практикою контейнерної оркестрації, де ресурси все одно задаються дискретно через requests/limits або профілі інстансів. Такий підхід також спрощує інтеграцію з Kubernetes HorizontalPodAutoscaler, який підтримує масштабування за ресурсними та користувацькими метриками, працює як контролер з періодичним циклом і може використовувати кілька метрик одночасно [7].

Третій варіант розв'язання проблеми масштабування полягає в поєднанні ігрової моделі з предиктивним відбором “кандидатів на масштабування”. У цьому разі теоретико-ігровий контролер не перераховує нову стратегію для всіх сервісів кластера, а спершу визначає лише ті сервіси, які найбільш імовірно спричиняють або успадковують порушення SLO. Такий відбір може виконуватися за телеметрією: p95/p99 latency, queue depth, saturation, error rate, throttling, concurrency, а також за ознаками критичного шляху. У сучасних роботах підкреслюється, що вузьке місце на критичному шляху запиту має непропорційно великий вплив на end-to-end latency, а отже саме такі сервіси повинні ставати пріоритетними для перерозподілу ресурсів [5, 6]. Це зменшує обсяг обчислень і робить алгоритм придатнішим для великих середовищ.

Четвертий шлях - гібридна схема “теорія ігор + класичні механізми autoscaling”. У такій архітектурі стандартний НРА або інший штатний контролер забезпечує базову реакцію на зростання навантаження, тоді як ігровий модуль працює як надбудова для більш “дорогих” рішень: зміни пріоритетів сервісів, перерозподілу бюджету ресурсів між доменами, тимчасового зменшення ресурсів у не критичних сервісів або вибору порядку масштабування. Перевага цього підходу в тому, що не потрібно повністю замінювати існуючу оркестрацію; навпаки, ігрова модель може підсилювати стандартні механізми Kubernetes, які вже підтримують scaling on custom metrics, stabilization windows і configurable scaling behavior [7]. Для практичної експлуатації це значно реалістичніше, ніж повна заміна на кастомний контролер.

Другою ключовою проблемою є врахування залежностей між сервісами. У базовій моделі [1] мікросервіси розглядаються як агенти, що конкурентно взаємодіють за ресурси, але в реальній системі зв'язки між сервісами мають структурний характер. Один сервіс може бути вхідною точкою, інший може виконувати оркестрацію бізнес-процесу, третій працювати з базою даних, четвертий забезпечувати кешування, а п'ятий виконувати асинхронну обробку. Через це затримка сервісу має різну ціну залежно від його місця в графі викликів. Дослідження критичних шляхів у великих мікросервісних архітектурах показують, що зменшення довжини критичного шляху є необхідним для зменшення наскрізної латентності, а тому пріоритезація сервісів на критичному шляху є більш виправданою, ніж рівномірний розподіл ресурсів між усіма вузлами [6].

Перший конкретний варіант розв'язання цієї проблеми - перехід до гри на графі викликів. У такій моделі вершини графа відповідають мікросервісам, а ребра - фактам виклику або залежностям даних. Тоді функція корисності сервісу визначається не лише його власною затримкою, а й затримкою сусідів по графу, числом вхідних і вихідних залежностей, розташуванням на критичному шляху та внеском у кінцевий час відповіді. Наприклад, штраф за дефіцит ресурсів може множитися на вагу центральності сервісу або на частку критичних бізнес-запитів, що проходять через нього. Це дає можливість відмовитися від припущення, що всі сервіси “рівноцінні”, і перейти до ресурсної політики, у якій найбільш критичні вузли отримують пріоритет. Саме такий перехід прямо анонсований у висновках базової статті [1].

Другий варіант - це використання критичного шляху як керуючого сигналу для ігрового контролера. Замість того щоб оптимізувати середні метрики по всіх сервісах, контролер може будувати або оновлювати критичний шлях із трасувань Jaeger/OpenTelemetry і на кожному циклі орієнтуватися на сервіси, які найчастіше входять у критичний ланцюг. У такому разі масштабування або додатковий ресурсний бюджет спочатку отримують ті сервіси, що найбільше впливають на tail latency. Це добре узгоджується з практичними висновками робіт з critical path analysis: у великих розподілених системах оптимізація поза критичним шляхом часто дає значно менший ефект, ніж навіть невелике прискорення сервісів, які регулярно

формують bottleneck [6].

Третій спосіб - ввести в модель класи сервісів за критичністю. Наприклад, сервіси можна поділити на:

- а) критичні синхронні сервіси на шляху користувацького запиту;
- б) важливі, але не критичні транзакційні сервіси;
- в) допоміжні сервіси;
- г) асинхронні фонові сервіси.

Для кожного класу можна визначити власну форму функції корисності. Для критичних сервісів штраф за ріст латентності має бути вищим. Для асинхронних сервісів частина дефіциту ресурсів може бути допустимою. Для допоміжних сервісів акцент можна робити на економії ресурсів. Це практичний спосіб врахувати залежності навіть без повної побудови детального графа викликів, і він добре підходить для перших етапів експериментальної перевірки на великих системах.

Четвертий варіант - поєднати ігрову оптимізацію з механізмами ізоляції від каскадних збоїв. Якщо сервіс є критичною залежністю для великої кількості інших сервісів, ресурсна модель має враховувати не тільки його latency, а й ризик поширення відмов. Тут доцільно використовувати circuit breakers, timeouts, retries з обмеженнями та traffic management на рівні service mesh. Офіційна документація Azure прямо вказує, що circuit breaker слід застосовувати для запобігання cascading failures і для захисту від повільних залежностей, а Istio надає вбудовані механізми timeout, retry та circuit breaker без зміни прикладного коду [8, 9]. Отже, один із практичних шляхів розширення моделі [1] - вбудувати в функцію виграшу штраф не лише за локальну деградацію, а й за підвищений ризик каскадного зниження доступності залежних сервісів.

Для експериментальної перевірки запропонованих рішень доцільно використовувати не тільки малі тестові конфігурації, а сценарії з десятками або сотнями сервісів, різною глибиною графа викликів, fan-out/fan-in структурами, змішаним синхронним та асинхронним навантаженням, а також штучно створеними “гарячими” сервісами. Набір метрик теж слід розширити: окрім average response time, необхідно оцінювати p95/p99 latency, throughput, rate of SLA violations, convergence time і накладні витрати самого контролера. Такий підхід дозволить перевірити, чи зберігає ігрова модель ефективність не тільки в лабораторному сценарії, а й у режимах, ближчих до експлуатації великих платформ. Саме потреба в додатковому аналізі для поширення методу на великі системи прямо зазначена в статті [1].

Список літератури

1. Улічев О. С., Кулагін В. П. Оптимізація роботи мікросервісів на основі теорії ігор. Центральноукраїнський науковий вісник. Технічні науки. 2025. Вип. 12(43), ч. 1. С. 44–57. DOI: 10.32515/2664-262X.2025.12(43).1.44-57.
2. Dragoni N., Lanese I., Larsen S. T., Mazzara M., Mustafin R., Safina L. Microservices: How to make your application scale. Lecture Notes in Computer Science. 2018. Vol. 10742. P. 95–104. DOI: 10.1007/978-3-319-74313-4_8.
3. Altman E., Boulogne T., El-Azouzi R., Jiménez T., Wynter L. A survey on networking games in telecommunications. Computers & Operations Research. 2006. Vol. 33, № 2. P. 286–311. DOI: 10.1016/j.cor.2004.06.005.
4. Ardagna D., Panicucci B., Passacantando M., Generalized Nash equilibria for the service provisioning problem in cloud systems. IEEE Transactions on Services Computing. 2013. Vol. 6, № 4. P. 429–442. DOI: 10.1109/TSC.2012.14.
5. M. R. Hossen, M. A. Islam, and K. Ahmed, “Practical Efficient Microservice Autoscaling with QoS Assurance,” in Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), 2022. doi: 10.1145/3502181.3531460.
6. Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, CRISP: Critical Path Analysis of Large-Scale Microservice Architectures, in USENIX ATC 22, 2022. [Online]. Available: https://www.researchgate.net/publication/378708480_CRISP_Critical_Path_Analysis_of_Large-Scale_Microservice_Architectures_CRISP_Critical_Path_Analysis_of_Large-Scale_Microservice_Architectures
7. Kubernetes, Horizontal Pod Autoscaling, Kubernetes Documentation. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale>
8. Microsoft, Circuit Breaker Pattern, Azure Architecture Center. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker>
9. Istio, Traffic Management, Istio Documentation. [Online]. Available: <https://istio.io/latest/docs/concepts/traffic-management/>