УДК 004.75 **doi: 10.26906/SUNZ.2024.4.056**

О. А. Дмитренко, М. А. Скулиш

Національний технічний університет України “КПІ ім. Ігоря Сікорського”, Київ, Україна

МЕТОДИ ЗБОРУ ІНФОРМАЦІЇ ТА РЕАЛІЗАЦІЇ АЛГОРИТМУ ДОПОВНЮВАЛЬНИХ НАВАНТАЖЕНЬ

Анотація: У статті розглядається проблема ефективного використання ресурсів у хмарних обчислювальних середовищах, що стає дедалі актуальнішою через зростання попиту на обчислювальні потужності. Представлений алгоритм оптимізації розподілу навантаження створеного мікросервісами та монолітами на серверні вузли, спрямований на максимізацію використання доступних ресурсів. Стаття зосереджена на описані технологій, які допоможуть повторити алгоритм локально з метою його тестування для власних потреб. Порівнюються методи збору метрик, а також середовища, у яких можна впровадити цей алгоритм. Особливу увагу приділено Kubernetes і можливості застосування різних механізмів розподілу мікросервісів, таких як Node Affinity, Pod Affinity/Anti-Affinity, а також власні планувальники (Custom Schedulers). Для проактивного масштабування включаючи "гарячий розігрів" вузлів запропоновано використання Horizontal Pod Autoscaler з використанням метрик роботи елементів системи.

Ключові слова: Kubernetes, збір метрик, доповняльні навантаження, хмарні системи, мікросервісна архітектура, моноліти.

Вступ

Сучасне IT рухається у напрямку забезпечення сервісів в будь-який час довільній кількості людей. Часто одне програмне забезпечення необхідне людям в різних куточках світу, наприклад, сайт покупки авіабілетів. Це накладає додаткові вимоги на програмне забезпечення, а саме змушує його бути готовим до різких навантажень коли-завгодно. Це говорить про необхідність програмного забезпечення масштабуватись (scaling) та рівномірно розподіляти навантаження (load balancing). З такою метою створювались хмарні системи, а трохи згодом з’явився Kubernetes [1], котрий теж має такі можливості.

Авторами статті пропонується алгоритм [2], який влаштується у стандартні добре відтестовані процеси хмарної системи та допоможе ресурсоефективно розподілити навантаження додатків, щоб використовувалось менше серверів для забезпечення тої самої продуктивності шляхом розташування доповняльних навантажень разом.

Постановка проблеми. Запропонувати способи реалізації методу пошуку доповняльних мікросервісів локально з метою реалізації та тестування алгоритму доповняльних додатків (мікросервісів) та потенційного його використання для свого проєкту, розгорнутого на хмарній системі чи іншому сервер провайдері.

Аналіз останніх досліджень та публікацій. Коли мова йде про алгоритм, котрий має виконуватись на хмарному середовищі, та ще й самим оператором, слід мати ресурси для імітації процесу на локальних серверах, аби його протестувати та налаштувати параметри. По суті єдиним інструментом, який може зімітувати хмарну систему є Kubernetes. Це платформа з відкритим вихідним кодом для управління контейнеризованими (через Docker [3]) робочими навантаженнями та супутніми службами (наприклад, трекінг метрик). Вона характеризується кросплатформенністю та розширюваністю, тобто в разі посиленого навантаження, вона може адаптуватись та вирости, аби якісно надати послугу користувачам. Kubernetes часто асоціюється з хмарними технологіями (cloud) через те, що обидві концепції мають спільну мету — забезпечити масштабованість, автоматизацію та ефективне управління ресурсами для додатків. Kubernetes, як і хмарна система, включає балансування навантаження, розподіл ресурсів і автоматичне відновлення контейнерів після збоїв [1]. Хмарні платформи (AWS, Azure, Google Cloud) також надають ресурси автоматично і вимагають мінімального втручання користувача.

Як і в хмарних середовищах, Kubernetes дозволяє масштабувати додатки за необхідності, автоматично створюючи нові інстанси (екземпляри) контейнерів. Це схоже на масштабування ресурсів в хмарних інфраструктурах, де можна динамічно додавати або видаляти обчислювальні ресурси на основі попиту.

Хмарні сервіси надають можливість керування різними ресурсами, такими як віртуальні машини (VM), мережеві підключення, сховища і бази даних. Kubernetes забезпечує оркестрацію контейнеризованих додатків, що є аналогом такого управління на рівні додатків. І хмарні технології, і Kubernetes добре підходять для роботи з мікросервісною архітектурою. Вони дозволяють легко розгортати, масштабувати та підтримувати незалежні сервіси, з яких складається додаток, що є основним підходом у хмарних системах. Таким чином, імітація роботи алгоритму на Kubernetes є способом протестувати його, а можливо і повноцінно користуватись, якщо мова йде про сервер-провайдера, котрий не обслуговує клієнтів на рівні хмарної технології, а лиш надає локальні послуги хостінгу. Щобільше, то багато хмарних платформ використовують Kubernetes як основний механізм для управління контейнерами. Вони надають сервіси, що базуються на Kubernetes для спрощення розгортання та управління контейнеризованими додатками в хмарі. Сам Kubernetes був розроблений у Google, який відкрив доступ до вихідного коду проекту Kubernetes у 2014. Google Kubernetes Engine (GKE) — сервіс від Google Cloud, побудований на основі Kubernetes. Так само існує Amazon Elastic Kubernetes Service (EKS) — сервіс від AWS для управління Kubernetes-кластерами (імітується за допомогою інструментів kOps або EKS-D**)** та Azure Kubernetes Service (AKS) — сервіс від Microsoft Azure для розгортання та управління Kubernetes.

Зважаючи на це, головною задачею тестування алгоритму є розгортання Kubernetes на локальних серверах. Можна навіть створити приватну хмару. **Minikube** - один з найпоширеніших інструментів для локальної інсталяції Kubernetes-кластера на одному вузлі (single-node). Він підходить для тестування та розробки. Minikube дозволяє працювати з Kubernetes на локальному комп'ютері, імітуючи хмарне середовище.

Іншим інструментом є **Kind (Kubernetes in Docker)**. Він дозволяє розгортати Kubernetes кластери безпосередньо у Docker-контейнерах. Це також хороший варіант для розробників, яким потрібно тестувати додатки в контейнерах.

Kubeadm - це інструмент для створення повноцінних кластерів Kubernetes на локальних машинах або віртуальних машинах. Він забезпечує більш повну конфігурацію, що підходить для тестування або навіть продакшн-оточень.

Також існує **MicroK8s**, але оскільки це легка реалізація Kubernetes від Canonical (розробники Ubuntu), він не дуже підходить для тестування габаритного алгоритму, адже створений для розгортання Kubernetes-кластер на малопотужних пристроях, таких як Raspberry Pi, але теж і на локальних машинах.

Отже, хмарні платформи використовують Kubernetes як один з основних інструментів для оркестрації контейнерів. Локальне розгортання Kubernetes дозволяє імітувати хмарну інфраструктуру для тестування та розробки. З допомогою Kubernetes можна імітувати хмарні системи AWS, Azure, Google Cloud [4], [5], [6] та інші локально, що надає можливості для тестування додатків та оркестрації додатків у хмарному контексті без необхідності використовувати реальні хмарні ресурси. Таким чином, розгорнувши Kubernetes за допомогою Minicube можна зімітувати потрібні мікросервіси та протестувати алгоритм пошуку доповняльних особин.

Метою статті є показати можливість використання алгоритму доповняльних мікросервісів як на локальному проєкті, запущеному на серверному провайдері чи хмарній системі, так і глобально для оркестрації всіх проєктів, мікросервісів та монолітів на рівні провайдера серверів чи хмари.

Виклад основного матеріалу

Суть алгоритму пошуку доповнень. Алгоритм спрямований на оптимізацію використаних ресурсів під час роботи багатокомпонентної системи, зокрема, при масштабуванні її вузлів для забезпечення покриття навантаження всіх користувацьких запитів. Алгоритм виглядає наступним чином. Під час пошуку комплементарних навантажень (мікросервісів) алгоритм виконує кластеризацію, в результаті чого екземпляри мікросервісів групуються за схожістю робочих шаблонів у класи еквівалентності. В межах кожної групи екземпляри сортуються за амплітудою використання ресурсів. Далі групи, що визначені як доповняльні, перетинаються з метою знайти пари комплементарних мікросервісів. Екземпляри з протилежними шаблонами навантаження, але схожими амплітудами, комбінуються для ефективнішого використання ресурсів. Також можуть розглядатися пари з низькою амплітудою для заповнення "пробілів". Алгоритм використовує жадібний підхід, щоб знайти перший екземпляр мікросервісу, що відповідає критеріям доповнення. Якщо доповнення в межах першої групи не знайдено, зберігається статистика комбінацій з усіма екземплярами, і пошук триває до знаходження доповнень або вичерпання доповняльних груп. Процес повторюється до знаходження всіх можливих пар, хоча залишкові екземпляри мікросервісів є очікуваними. Для мікросервісів, що не знайшли доповнень, застосовується додатковий аналіз екстремумів з використанням середнього та стандартного відхилення, щоб підвищити ймовірність знайти комплементи. Кластеризація і пошук продовжуються до останньої успішної спроби. Якщо залишається невелика кількість мікросервісів без пари, вони комбінуються за допомогою рішення задачі про множинний рюкзак [2, 7, 8].

Методи збору вхідних даних. Алгоритм базується на статистичних даних, запропонованих в першу чергу розробниками програмного забезпечення (ПЗ). Ці дані важливі для почткової класифікації мікропроцесора, вибора йому пари. При неправильному виборі пари, результати алгоритму не будуть кращими, ніж стандартні. Вартість помилки збору початкових даних не дуже велика, адже клауд-провайдер зніматиме і свої метрики, котрі будуть застосовуватись на наступних ітераціях. Аби коректно провести збір даних, слід користуватись додатковими інструментами. Шаблони інструментів для збору та обробки метрик мікросервісів такі: система моніторингу та збору метрик, яка добре інтегрується з Kubernetes (напр., Prometheus); інструмент для візуалізації метрик (напр., Grafana, Kibana); інструменти для збору та обробки логів. В логах можуть бути додаткові метрики, які можуть бути необхідні для формування повної картини у розробників ПЗ (напр., Fluentd або Logstash, Grafana Loki, ElasticSeach). Оскільки для хмарного провайдера головне — самі метрики, а не їх візуалізація, нижче описано деталі стосовно роботи Prometheus.

Prometheus — це потужна система для моніторингу та збирання метрик, яка працює шляхом надсилання HTTP-запитів до інтерфейсу взаємодії (API, endpoints), що повертає необхідні метрики. Такий інтерфейс з локальними метриками має бути написаний в застосунках та сервісах, які потребують моніторингу. Зібрані дані зберігаються в спеціалізованій базі даних часових рядів Prometheus, що забезпечує високу ефективність і швидкість доступу до інформації, надає можливість оперативно аналізувати зміни у метриках із плином часу. Ця система є чудовим вибором для моніторингу використання ресурсів, таких як CPU, RAM і мережеві інтерфейси, оскільки містить в собі Node Exporter для експорту цих даних [9]. Prometheus підтримує кілька форматів даних, серед яких простий текстовий формат і OpenMetrics. Він здатен збирати метрики з різноманітних джерел, таких як веб-сервери, бази даних, апаратні пристрої, користувацькі застосунки.

Розглянемо основні функції Prometheus.

Для розробників при аналізі проблем продуктивності системи та статистики буде корисними PromQL (Prometheus Query Language) — потужна мова запитів, яка дозволяє гнучко запитувати та аналізувати

збережені дані. Для моніторингу системи в реальному часі в Prometheus існує механізм для створення сповіщення на основі заздалегідь визначених умов. Ці сповіщення обробляються через Alertmanager, який може інтегруватися з різними системами сповіщень, такими як електронна пошта, Slack або PagerDuty, забезпечуючи своєчасну реакцію на критичні події.

Альтернативу Prometeus в розрізі збору та збереження метрик по ресурсах для часових рядів складають **InfluxDB та Telegraf**. Telegraf - агент, який збирає дані про використання CPU, RAM, дискові ресурси та мережеві інтерфейси з різних джерел і передає їх в InfluxDB. InfluxDB - одна з найпотужніших баз даних часових рядів, оптимізована для високошвидкісного зберігання і запитів з InfluxQL - SQL-подібна мова запитів, що дозволяє виконувати складні запити для аналізу продуктивності систем у реальному часі. Також можна користуватись **Zabbix**, безплатним інструментом для збору різного роду метрик та логів, що працює через власні агенти або SNMP. Він має розвинену систему сповіщень і підтримує візуалізацію часових рядів.

Для здійснення професійного моніторингу на хмарному провайдері корисним інструментом буде комерційний хмарний сервіс **Datadog**. Він надає прості в налаштуванні агенти для збору метрик з серверів, контейнерів і хмарних платформ, таких як AWS, Azure чи GCP. Ним зручно збирати метрики, відстежувати ресурси та аналізувати продуктивність систем у реальному часі. Додатково, для старту та налаштування роботи системи, корисними будуть автоматичні дашборди - візуалізації з зібраних даних. Іншими доступними аналогами є New Relic, VictoriaMetrics та Netdata, які працюють з часовими рядами, великими даними та знімають потрібні метрики [9, 10]. Найуніверсальнішим, з відкритим кодом (opensource) та безплатним є Prometheus. З ним інтегруються багато додаткових інструментів, а головне - він співпрацює з Kubernetes за допомогою додатків Thanos або Cortex. Вони дозволяють масштабувати Prometheus для довготривалого зберігання метрик і роботи з розподіленими кластерами Kubernetes . Вони додають реплікацію, що забезпечує резервне зберігання, захист від втрат даних і централізацію метрик із декількох кластерів.

Мови програмування для написання скриптів. Обрана мова програмування має добре співпрацювати як з Kubernetes, так і з системою збору метрик. Як останню, оберемо Prometheus. До нього існують бібліотеки від виробника на шість мов програмування, а також додаткові бібліотеки для всіх поширених мов [11]. Kubernetes написаний на **Go**. Вона забезпечує високу продуктивність та інтеграцію з системою. Іншою поширеною та зручною мовою для написання скриптів, які можуть обробляти та аналізувати метрики є **Python**. Бібліотеки до цих мов написані авторами Prometheus, а отже є гарантія якості та підтримка в разі виникнення проблем. Для Python використовується бібліотека kubernetes-client, а для Go - client-go. Тому пропонується обрати одну з цих мов для виконання алгоритму пошуку доповнень для Kubernetes та хмарних систем.

Налаштування Kubernetes для виконання алгоритму. У Kubernetes можна налаштувати інтелектуальний розподіл мікросервісів між подами та вузлами, використовуючи різні механізми, такі як планувальники (schedulers) [12], політики аффініті та антиаффініті - приналежність до певних вузлів, чи навпаки, про які розказано нижче, а також метрики для масштабування і «розігріву» ресурсів. Для застосування алгоритму розподілу навантаження для вирішення, на які поди направляти додатки, базуючись на ряді факторів, таких як наявність ресурсів (CPU, RAM) та визначення кількості подів на вузлі, слід інтегрувати зовнішні інструменти та метрики з бази даних для детального планування та передбачуваного "гарячого розігріву" (hot standby) [13] вузлів.

Node Affinity і Pod Affinity/Anti-affinity - це механізми, що дозволяють задавати правила для планувальника, щоб він розміщував поди на конкретних вузлах або на основі близькості до інших подів. Наприклад, можна сказати Kubernetes розміщувати певні мікросервіси ближче один до одного або навпаки – розподіляти їх на різні вузли для зменшення навантаження.

Механізм **Custom Schedulers** дає можливість розробити власний планувальник (scheduler) [12, 13], який буде враховувати зазначені в алгоритмі фактори або для розподілу подів між вузлами. На цьому етапі, можна інтегрувати метрики з БД, щоб впливати на рішення планувальника.

За допомогою **Taints and Tolerations** можна "позначати" вузли, щоб певні поди могли бути розгорнуті тільки на них або уникати цих вузлів. Це корисно для управління подами, на яких навантаження може сильно змінюватись, аби запобігти ситуації призначення додаткового навантаження на вузол в той час, як скоро вільний простір на вузлі зникне. За умови додаткового навантаження довелось би робити додатковий небажаний скейлінг [15].

Щоб реалізувати "гарячий розігрів" [13] вузлів на основі метрик, можна інтегрувати рішення на базі **Horizontal Pod Autoscaler (HPA)** - додавання додаткових вузлів або цю задачу теж покласти на алгоритм на основі статистики метрик. Останнє допоможе запобігати піковому навантаженню і дасть більше часу для розігріву нової ноди. Горизонтальне автомасштабування використовує метрики, такі як CPU і пам’ять, для автоматичного збільшення або зменшення кількості подів на основі навантаження [16]. Витягнути метрики з бази даних і використовувати їх у HPA можна використовуючи **Custom Metrics API** або **Prometheus Adapter**.

Іншим механізмом масштабування є Vertical Pod Autoscaler (VPA) дозволяє автоматично змінювати кількість ресурсів (CPU, пам’ять) для окремих подів, але у цього підходу є обмеження. При зміні кількості ресурсу под необхідно перезавантажувати, таким чином це не може бути єдиний под для мікросервісу, щоб не припиняти обслуговування додатка [15], а також це бажаніше робити при зменшенні навантаження, а не збільшенні, щоб не переставати обслуговування пікового навантаження користувачів та щоб не спричиняти потенційних проблем обслуговування користувачів пов’язаних з перевантаженням додатка. Таким чином, алгоритм пошуку доповнень орієнтований на горизонтальне розширення, котре гарантовано даватиме необхідний результат та буде

більш передбачуване, не ставлячи під загрозу забезпечення сервісом користувачів. Додатково слід відмітити Kubernetes Event-Driven Autoscaling (KEDA), що дозволяє масштабувати поди на основі подій або зовнішніх метрик, таких як повідомлення з черг, HTTP-запитів, або аналітики з баз даних. KEDA підтримує інтеграцію з Prometheus, Azure Monitor, AWS CloudWatch та іншими системами збору метрик. В разі раптової зміни обстановки можна скористатись цим механізмом, але на цей час алгоритм доповнювальних навантажень це не передбачає.

Висновки

У статті розглянуто засоби реалізації алгоритму пошуку доповнювальних навантажень [7] локально з метою перевірити якість його роботи та за потреби скористатись ним при деплої власного програмного забезпечення на хмарну систему. Все ж, планується, що даний алгоритм застосовуватиметься більше на всій хмарі, адже його ефективність тоді буде вищою.

Основною технологією для реалізації алгоритму пропонується взяти Kubernetes. Як локальний помічник може виступати Minicube. Prometheus - пропонована система для зняття метрик. Аналіз метрик слід покласти частково на Python або Go, а частково на Kubernetes. Написавши власний планувальник (Custom Scheduler) з використанням обмежень, таких як афініті, антіафініті та толерації - характеристики прив'язаності до вузлів та подів для формування правил розгортання додаткових вузлів на етапі горизонтального масштабування, яке передбачене алгоритмом.

Впровадження алгоритму доповнювальних навантажень сприятиме зменшенню кількості необхідних серверів для хостінгу, оптимізації енергетичних ресурсів та як результат, меншому негативному впливу на екологію через антропогенний фактор.

СПИСОК ЛІТЕРАТУРИ

1. "Що таке Kubernetes?," Kubernetes. Accessed: Oct. 22, 2024. Available: https://kubernetes.io/uk/docs/concepts/overview/what-is-kubernetes/
2. O. Dmytrenko, M. Skulysh, and L. Globa, "Microservice Complimentary Groups Determination Algorithm for the Effective Resource Usage," CEUR Workshop Proc. Forthcom., no. Scientific and Practical Programming-UkrPROG 2024, p. 20.
3. Docker. Available: https://www.docker.com/resources/what-container/
4. S. Al-Raheym, S. C. Açan, and Ö. T. Pusatli, "Investigation Of Amazon And Google For Fault Tolerance Strategies In Cloud Computing Services," AJIT-E Online Acad. J. Inf. Technol., vol. 7, no. 23, pp. 7–22, Nov. 2016, doi: 10.5824/1309-1581.2016.4.001.x.
5. A. Belgacem, "Dynamic resource allocation in cloud computing: analysis and taxonomies," Computing, vol. 104, no. 3, pp. 681–710, Mar. 2022, doi: 10.1007/s00607-021-01045-2.
6. E. R. Jamzuri, R. Analia, H. Mandala, and S. Susanto, "Cloud-Based Architecture for YOLOv3 Object Detector using gRPC and Protobuf," J. Tek. Elektro, vol. Vol 14, No 1, pp. 18–23, 2022.
7. O. Dmytrenko and M. Skulysh, "Визначення мікропроцесорних груп для ефективного використання процесорних потужностей," Probl. Program. Forthcom., no. №2-3, Sep. 2024, Available: https://docs.google.com/document/d/1ocqA-e7d-OAibFVGiEE2UcxZRKgYNn_yPX6e94dHwYM/edit?usp=sharing
8. O. Dmytrenko and M. Skulysh, "Method of Grouping Complementary Microservices Using Fuzzy Lattice Theory," vol. 12, no. 1, pp. 11–18, Mar. 2024, doi: 10.25673/115636.
9. "Що таке Prometheus? | Адміністрування серверів та cloud рішень." Accessed: Oct. 22, 2024. Available: https://itfb.com.ua/uk/shho-take-prometheus/
10. Prometheus, "Overview | Prometheus." Accessed: Oct. 23, 2024. Available: https://prometheus.io/docs/introduction/overview/
11. Prometheus, "Client libraries | Prometheus.". Available: https://prometheus.io/docs/instrumenting/clientlibs/
12. P. Salot, "A Survey of Various Scheduling Algorithm in Cloud Computing Environment," Int. J. Res. Eng. Technol., vol. 02, no. 02, pp. 131–135, Feb. 2013, doi: 10.15623/ijret.2013.0202008.
13. O. Dmytrenko and M. Skulysh, "Fault Tolerance Redundancy Methods for IoT Devices," Infocommunication Comput. Technol., vol. 2(04), no. University "Ukraine," pp. 59–65, Dec. 2022.
14. Kubernetes Team, "Kubernetes Scheduler." Kubernetes Documentation, Dec. 14, 2023. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
15. "Vertical Pod autoscaling | Google Kubernetes Engine (GKE)," Google Cloud. Accessed: Jul. 14, 2024. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler
16. "Kubernetes Autoscaling: Horizontal and Vertical explained." Accessed: Oct. 23, 2024. Available: https://www.full-staq.com/knowledge-hub/blogs/autoscaling-in-kubernetes

Received (Надійшла) 20.09.2024
(Accepted for publication) Прийнята до друку 13.11.2024

Methods of data collection and implementation of accomplishing loads algorithm

O. Dmytrenko, M. Skulysh

Abstract: The article addresses the issue of efficient resource utilization in cloud computing environments, which has become increasingly important due to the growing demand for computational power. The presented algorithm focuses on optimizing the load distribution generated by microservices and monoliths across server nodes, aiming to maximize the usage of available resources. The article focuses on technologies that help replicate the algorithm locally for testing and adaptation to specific needs. It compares methods for collecting metrics, as well as environments where the algorithm can be implemented. Special attention is given to Kubernetes and the possibilities of applying different microservice distribution mechanisms, such as Node Affinity, Pod Affinity/Anti-Affinity, and custom schedulers. For proactive scaling, including "hot start" of nodes, the use of Horizontal Pod Autoscaler with system element metrics is proposed.

Keywords: Kubernetes, metric collection, complementary loads, cloud systems, microservice architecture, monoliths.