RS Global Journals

Scholarly Publisher
RS Global Sp. z O.O.
ISNI: 0000 0004 8495 2390

Dolna 17, Warsaw, Poland 00-773
Tel: +48 226 0 227 03
Email: editorial_office@rsglobal.pl

JOURNAL	International Academy Journal Web of Scholar
p-ISSN	2518-167X
e-ISSN	2518-1688
PUBLISHER	RS Global Sp. z O.O., Poland

ARTICLE TITLE	ДОСЯГНЕННЯ ЕФЕКТИВНОГО РОЗПОДІЛЕНОГО ПЛАНУВАННЯ ЗА ДОПОМОГОЮ ЧЕРГ ПОВІДОМЛЕНЬ У ХМАРІ ДЛЯ БАГАТОЗАДАЧНИХ ОБЧИСЛЕНЬ ТА ВИСОКОПРОДУКТИВНИХ ОБЧИСЛЕНЬ
AUTHOR(S)	Старовойтенко Олексій Володимирович
ARTICLE INFO	Starovoitenko O. V. (2020) Achieve Efficient Distributed Scheduling with Cloud Message Queuing for Multitasking and High-Performance Computing. International Academy Journal Web of Scholar. 8(50). doi: 10.31435/rsglobal_wos/30122020/7323
DOI	https://doi.org/10.31435/rsglobal_wos/30122020/7323
RECEIVED	26 October 2020
ACCEPTED	14 December 2020
PUBLISHED	19 December 2020
LICENSE	This work is licensed under a **Creative Commons Attribution 4.0 International License.**

ДОСЯГНЕННЯ ЕФЕКТИВНОГО РОЗПОДІЛЕНОГО ПЛАНУВАННЯ ЗА ДОПОМОГОЮ ЧЕРГ ПОВІДОМЛЕНЬ У ХМАРІ ДЛЯ БАГАТОЗАДАЧНИХ ОБЧИСЛЕНЬ ТА ВИСОКОПРОДУКТИВНИХ ОБЧИСЛЕНЬ

Старовойтенко Олексій Володимирович,
Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", ORCID ID: https://orcid.org/0000-0001-7365-3940

DOI: https://doi.org/10.31435/rsglobal_wos/30122020/7323

ARTICLE INFO

Received: 26 October 2020
Accepted: 14 December 2020
Published: 19 December 2020

KEYWORDS

FlexQueue, multitasking, cloud message queues, high-performance system, data consistency.

ABSTRACT

Due to the growth of data and the number of computational tasks, it is necessary to ensure the required level of system performance. Performance can be achieved by scaling the system horizontally / vertically, but even increasing the amount of computing resources does not solve all the problems. For example, a complex computational problem should be decomposed into smaller subtasks, the computation time of which is much shorter. However, the number of such tasks may be constantly increasing, due to which the processing on the services is delayed or even certain messages will not be processed. In many cases, message processing should be coordinated, for example, message A should be processed only after messages B and C. Given the problems of processing a large number of subtasks, we aim in this work - to design a mechanism for effective distributed scheduling through message queues. As services we will choose cloud services Amazon Webservices such as Amazon EC2, SQS and DynamoDB. Our FlexQueue solution can compete with state-of-the-art systems such as Sparrow and MATRIX. Distributed systems are quite complex and require complex algorithms and control units, so the solution of this problem requires detailed research.

Citation: Starovoitenko O. V. (2020) Achieve Efficient Distributed Scheduling with Cloud Message Queuing for Multitasking and High-Performance Computing. *International Academy Journal Web of Scholar*. 8(50). doi: 10.31435/rsglobal_wos/30122020/7323

Вступ. Метою системи планування завдань є ефективне управління розподіленою обчислювальною потужністю робочих станцій, серверів та суперкомп'ютерів з метою максимізації пропускної здатності роботи та використання системи. Прогнозується, що до кінця цього десятиліття ми матимемо систему масштабування з мільйонами вузлів і мільярдами потоків виконання [1].

На жаль, сучасні планувальні системи мають централізовану архітектуру Master/Slaves (наприклад, Slurm [2], Condor [3, 4], PBS [5], SGE [6]), де централізований сервер відповідає за надання ресурсів та виконання завдань. Ця архітектура добре працювала в масштабах обчислювальних мереж та грубих гранульованих робочих навантаженнях [7], але вона має погану масштабованість при екстремальних масштабах систем із дрібнозернистими робочими навантаженнями [8, 9]. Рішенням цієї проблеми є перехід до децентралізованих архітектур, які уникають використання одного компонента в якості менеджера. Розподілені планувальні системи, як правило, реалізуються в ієрархічній [10] або повністю розподіленій архітектурі [31] для вирішення проблеми масштабованості. Використання нових архітектур може вирішити потенційну

єдину точку відмови та покращити загальну продуктивність системи до певного рівня, але можуть виникнути проблеми при розподілі завдань та балансуванні навантаження між вузлами [25].

Ідея використання хмарних сервісів для високопродуктивних обчислень існує вже кілька років, але вона не набула популярності в першу чергу через багато проблем. Маючи великі ресурси, публічні хмари можна використовувати для виконання завдань в екстремальних масштабах розподіленим способом. Наша мета у цьому проекті - забезпечити компактну та легку розподілену структуру виконання завдань, яка працює на Amazon Elastic Compute Cloud (EC2) [17], використовуючи складні розподілені будівельні блоки, такі як Amazon Simple Queuing Service (SQS) [18] та Amazon, розподілений NoSQL ключ/значення сховища (DynamoDB) [33].

Було багато дослідницьких робіт з використання загальнодоступного хмарного середовища для наукових обчислень та високопродуктивних обчислень (HPC). Більшість цих робіт показують, що хмара не могла виконувати добре працюючі наукові програми [11, 12, 13, 14]. Більшість існуючих дослідницьких робіт використовували підхід використання публічної хмари як подібний ресурс до традиційних кластерів та суперкомп'ютерів. Використання спільних ресурсів та технологій віртуалізації робить загальнодоступні хмари зовсім іншими, ніж традиційні системи HPC. Замість того, щоб запускати одні й ті ж традиційні програми на іншій інфраструктурі, ми пропонуємо використовувати публічні хмарні сервісні програми, які оптимізовані для хмарного середовища. Використання загальнодоступних хмар, таких як Amazon, як ресурсу для виконання завдань може бути складним для кінцевих користувачів, якщо воно надає лише необроблену IaaS [34]. Було б дуже корисно, якби користувачі могли лише увійти в свою систему та надіслати завдання, не турбуючись про управління ресурсами.

Ще однією перевагою хмарних служб є те, що користуючись цими службами, користувачі можуть впродовж короткого періоду впровадити порівняно складні системи з дуже короткою базою коду. Наша мета - показати докази того, що користуючись цими послугами, ми можемо надати систему, яка надає високоякісні послуги, які відповідають сучасним системам, із значно меншою базою коду. *У цій роботі ми розробляємо та впроваджуємо масштабовану структуру виконання завдань на хмарі Amazon, використовуючи різні хмарні сервіси AWS, і спрямовуємо її на підтримку обчислень із багатьма завданнями та високопродуктивних робочих навантажень.*

Найважливішим компонентом нашої системи є служба простої черги Amazon (SQS), яка діє як служба доставки вмісту для виконання завдань, дозволяючи клієнтам ефективно, асинхронно та масштабовано спілкуватися з сервісами. Amazon DynamoDB - це ще один хмарний сервіс, який використовується, щоб переконатися, що завдання виконуються рівно один раз (це потрібно, оскільки Amazon SQS не гарантує семантику доставки точно один раз). Ми також використовуємо Amazon Elastic Compute Cloud (EC2) для управління віртуальними ресурсами. Завдяки можливості SQS одночасно доставляти надзвичайно велику кількість повідомлень великій кількості користувачів, система планування може забезпечити високу пропускну здатність навіть у більших масштабах.

Сучасна аналітика даних направлена до інтерактивних коротших завдань із більшою пропускною здатністю та меншою затримкою [35][10]. Більше програм використовується до запуску більшої кількості завдань з метою покращення пропускної здатності та продуктивності програм. Хорошим прикладом для цього типу програм є багатозадачні обчислення (MTC) [15, 16, 39, 40]. Додатки MTC часто вимагають короткого часу для вирішення і можуть вимагати значних комунікацій або даних [41].

Розподілені системи управління роботою мають проблему низького використання через їх погану стратегію балансування навантаження. *Ми пропонуємо FlexQueue як систему управління робочими місцями, яка забезпечує гарне балансування навантаження та велике використання системи у великих масштабах.* Замість використання таких методів, як випадкова вибірка, FlexQueue використовує розподілені черги, щоб коректно доставити завдання сервісам, не вимагаючи при цьому системи вибору між вузлами. Розподілена черга служить великим пулом завдань, які є високодоступними. Сервіс вирішує коли отримати нове повідомлення. Такий підхід забезпечує простоту та ефективність дизайну. Більше того, застосовуючи такий підхід, компоненти системи нещільно зв'язані між собою. Тому *система буде високо масштабованою, надійною та простою в оновленні.* Хоча мотивацією цієї роботи є підтримка завдань MTC, вона також забезпечує підтримку розподіленого планування HPC. Це дозволяє FlexQueue бути ще більш гнучким, одночасно виконуючи різні типи робочих навантажень.

Основним внеском цієї роботи є:

1. Спроектувати та впровадити просту та легку структуру виконання завдань за допомогою сервісів Amazon Cloud (EC2, SQS та DynamoDB), що підтримує як навантаження MTC, так і HPC

3. Оцінка продуктивності до масштабу 1024 екземплярів порівняно з Sparrow та MATRIX: FlexQueue здатний перевершити інші дві системи після масштабування 64 екземплярів з точки зору пропускної здатності та ефективності.

Решта розділів цієї статті є такими. В Розділ II продемонстровано деталі проектування та реалізації FlexQueue. Розділ III оцінює ефективність FlexQueue у різних аспектах, використовуючи різні показники. Розділ IV вивчає відповідну роботу в галузі систем виконання завдань. Нарешті, розділ V обговорює обмеження поточної роботи та висвітлює майбутні напрямки цієї роботи.

Розробка та впровадження flexqueue.

Метою цієї роботи є впровадження системи планування/управління сервісами, яка відповідає *чотирьом основним цілям:*

• Масштаб: пропонуємо збільшити пропускну спроможність із більшими масштабами через розподілені послуги

• Баланс навантаження: пропонуємо балансування навантаження у великих масштабах при неоднорідних робочих навантаженнях

• Слабко зв'язана: вирішальне значення для того, щоб зробити систему стійкою до несправностей і простою в обслуговуванні

Для досягнення масштабованості FlexQueue використовує SQS, який є розподіленим та дуже масштабованим. Як основний елемент FlexQueue, SQS може завантажувати та завантажувати велику кількість повідомлень одночасно. Незалежність сервісів та клієнтів забезпечує зручність роботи системи в більших масштабах. Для забезпечення інших функціональних можливостей, таких як моніторинг або послідовність виконання завдань, FlexQueue також використовує такі хмарні сервіси, як DynamoDB, які є повністю розподіленими та масштабованими.

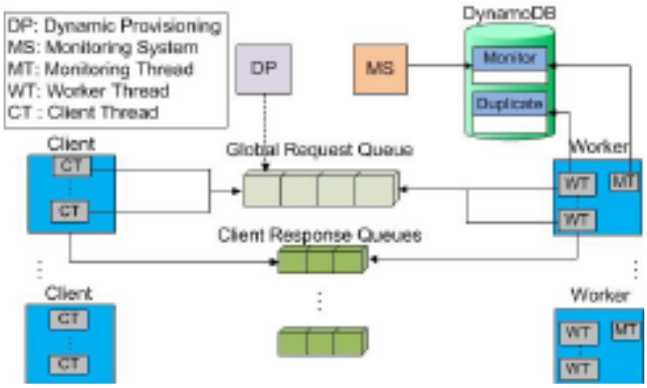

Рис. 1. Огляд архітектури FlexQueue

Через використання хмарних служб витрати на обробку FlexQueue дуже низькі. Багато програмних викликів у FlexQueue - це виклики до хмарних служб. Маючи абсолютно незалежні сервіси і клієнтів, FlexQueue не потрібно зберігати будь-яку інформацію про свої вузли, таку як IP-адреса або будь-який інший стан своїх вузлів.

Компоненти FlexQueue можуть працювати незалежно від компонента SQS посередині, щоб відокремити різні частини фреймворку один від одного. Це робить наш дизайн компактним, надійним і легко розширюваним.

Система планування може працювати в міжплатформенній системі з можливістю обслуговування в неоднорідному середовищі, яке має системи з різними типами вузлів з різними платформами та конфігураціями. Використання розподілених черг також допомагає зменшити залежність між клієнтами та сервісами. Клієнти та сервіси можуть самостійно змінювати свою швидкість pub/sub без будь-яких змін у системі.

Усі вищезазначені переваги покладаються на розподілену чергу, яка може забезпечити хорошу продуктивність у будь-якому масштабі. Amazon SQS - це високошвидкісна хмарна

служба, яка може надати всі функції, необхідні для реалізації масштабованої системи планування завдань. Використовуючи цей сервіс, ми можемо досягти мети - мати систему, яка ідеально вписується в загальнодоступне хмарне середовище та оптимально працює на своїх ресурсах.

Система полегшує користувачам розподілену роботу над хмарними ресурсами, просто використовуючи клієнтський інтерфейс, без необхідності знати деталі базових ресурсів, а також налаштовувати кластер.

A. Архітектура.

У цьому розділі пояснюється системний дизайн FlexQueue. Ми використали дизайн на основі компонентів для цього проекту з двох причин. (1) Дизайн на основі компонентів краще підходить для хмарного середовища. Це також допомагає розробити проект у вільному поєднанні. (2) Поліпшити впровадження в майбутньому буде простіше.

У наступних розділах пояснюється архітектура системи як для навантажень MTC, так і для HPC. FlexQueue має можливість запускати робочі навантаження за допомогою суміші обох типів завдань. Перший розділ показує архітектуру системи на випадок виконання виключно завдань MTC. Другий розділ описує процес у разі запуску завдань HPC.

1) Управління завданнями MTC.

На Рис. 1. показані різні компоненти FlexQueue, які беруть участь лише у запуску завдань MTC. Завдання MTC визначається як завдання, яке вимагає обчислювальних ресурсів, які може задовольнити один сервіс (наприклад, там, де сервіс керує ядром або вузлом). Клієнтський вузол працює як інтерфейс для користувачів для подання своїх завдань. SQS має обмеження в 256 КБ для розміру повідомлень, якого достатньо для розміру завдання FlexQueue. Для надсилання завдань через SQS нам потрібно використовувати ефективний протокол серіалізації з низькими накладними витратами на обробку. З цієї причини ми використовуємо буфер протоколу Google. Завдання зберігає системний журнал під час процесу, передаючи різні компоненти. Таким чином, ми можемо повністю зрозуміти різні компоненти, використовуючи докладні журнали.

Основними компонентами FlexQueue для запуску завдань MTC є Клієнт, Сервіс, Глобальна черга запитів та Черги відповідей клієнта. Система також має динамічний провайзер для управління ресурсами. Він також використовує DynamoDB для забезпечення моніторингу. Для кожного сервіса запущений потік моніторингу, який періодично повідомляє про використання кожного сервіса до сховища значень ключа DynamoDB.

Клієнтський компонент не залежить від інших частин системи. Він може почати виконувати та подавати завдання без необхідності реєструватися в системі. Наявності адреси глобальної черги достатньо, щоб компонент Клієнта приєднався до системи. Клієнтська програма багатопоточна. Тож він може подавати кілька завдань паралельно. Перед відправкою будь-яких завдань Клієнт створює для себе чергу відповідей. Усі подані завдання містять адресу черги відповідей Клієнта. Клієнт також має можливість використовувати групування завдань, щоб зменшити накладні витрати на зв'язок.

З метою підвищення продуктивності та ефективності системи ми вирішили встановити два режими. Якщо в системі виконуються завдання MTC, усі сервіси працюють як звичайні сервіси. Але у випадку запуску навантажень HPC або навантажень із комбінацією завдань HPC та MTC, крім звичайних сервісів, сервіси також можуть стати або менеджерами сервісами, які керують сервісам HPC, або субсервісами, які виконують завдання HPC.

Подібно до компонента Клієнт, компонент Сервіс самостійно працює в системі. Що стосується підтримки MTC, робоча функціональність є відносно простою і прямою. Маючи загальну чергу запитів, Сервіси можуть приєднатися до системи та вийти з неї в будь-який час під час виконання. Глобальна черга запитів діє як великий пул завдань. Клієнти можуть подавати свої завдання до цієї черги, а сервіс можуть витягувати із неї завдання. Використовуючи такий підхід, масштабованість системи залежить лише від масштабованості Глобальної черги, і це не призведе до додаткового навантаження на сервісів у більших масштабах. Код сервіса також є багатопотоковим і може паралельно отримувати кілька завдань. Кожен потік може об'єднати до 10 об'єднаних завдань. Знову ж таки, ця функція зроблена для зменшення великих накладних витрат на зв'язок. Після отримання завдання робочий потік перевіряє дублювання завдання, а потім перевіряє тип завдання. У разі запуску завдань MTC це одразу ж це зробить. Потім він поміщає результати в завдання і використовуючи заздалегідь

задану адресу всередині завдання, відправляє завдання назад в чергу відповіді Клієнта. Як тільки черга відповідей отримує завдання, відповідний потік клієнта витягує результати. Процес закінчується, коли Клієнт отримує всі результати своїх завдань.

2) Управління завданнями HPC.

На Рис. 2 показані додаткові компоненти для запуску завдань HPC. Як зазначалося вище, у разі запуску поєднання завдань HPC та MTC кожен сервіс може виконувати різні ролі. У разі отримання завдання MTC сервіс приступає до виконання завдання самостійно. DynamoDB використовується для підтримки статусу системи, щоб сервіси могли приймати рішення про життєздатність виконання завдання HPC. По суті, у DynamoDB ми зберігаємо поточну кількість діючих менеджерів та підпорядкованих сервісів, які зайняті виконанням завдань HPC, що дає іншим сервісам уявлення про те, скільки наявних ресурсів існує.

Якщо сервіс отримує роботу HPC. DynamoDB перевіряється, щоб переконатися, що в системі достатньо доступних вузлів, що імітують для виконання завдання HPC. Якщо це задоволено, сервіс (який тепер називається менеджером робочих) розміщує n повідомлень у другому SQS (Черга завдань HPC), *n* - кількість робочих місць, необхідних робочому менеджеру для виконання завдання. Якщо недостатньо доступних ресурсів, вузол не може виконувати функції менеджера сервісів; натомість цей вузол перевірить Чергу завдань HPC та виступить у ролі допоміжного сервіса. Якщо в черзі HPC є повідомлення, субсервіс повідомить менеджера, використовуючи IP-адресу робочого менеджера. Менеджер сервіса та субсервіс використовують RMI для спілкування. Після виконання менеджер сервісів надсилає результат до черги відповідей, яку повинен забрати клієнт.

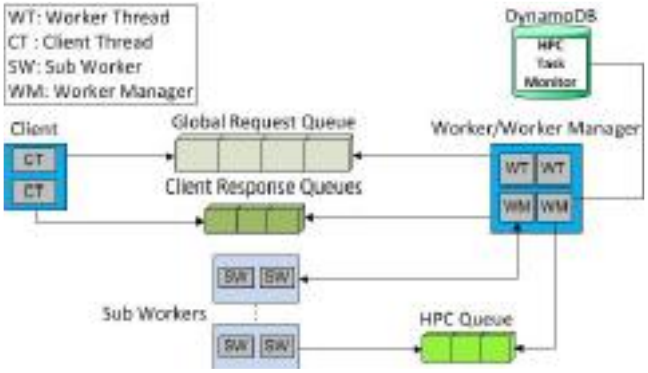

Рис. 2. Огляд архітектури FlexQueue-HPC

Б. Питання узгодженості виконання завдання.

Основним обмеженням SQS є те, що черга гарантує доставку повідомлень принаймні один раз. Це означає, що можуть існувати дублікати повідомлень, що передаються сервісам. Існування дублікатів повідомлень походить від того, що ці повідомлення копіюються на кілька серверів з метою забезпечення високої доступності та збільшення можливості паралельного доступу. Нам потрібно запропонувати методику, яка запобігає запуску дублікатів завдань, які доставляє SQS. У багатьох типах навантажень виконання завдання більше одного разу неприйнятно. Для того, щоб бути сумісним для таких типів програм, FlexQueue повинен гарантувати рівномірне виконання завдань.

Для того, щоб мати можливість перевірити дублювання, ми використовуємо DynamoDB. Отримавши завдання, робочий потік перевіряє, чи виконується завдання вперше. Робочий потік робить умовний запис у таблицю DynamoDB, додаючи унікальний ідентифікатор завдання, який є комбінацією ідентифікатора завдання та ідентифікатора клієнта. Операція завершується успішно, якщо ідентифікатор не був записаний раніше. В іншому випадку служба видає виняток для сервіса, і сервіс скидає дублікат завдання, не запускаючи його. Ця операція є атомарною операцією.

Як ми вже згадували вище, рівно один раз доставка необхідна для багатьох типів програм, таких як наукові програми. Але є деякі програми, які мають менші вимоги до послідовності і все ще можуть функціонувати без цієї вимоги. Наша програма має можливість вимкнути цю функцію для цих програм, щоб зменшити затримку та збільшити загальну

продуктивність. Ми вивчимо накладні витрати на цю функцію щодо загальної продуктивності системи в розділі оцінки.

C. Динамічне забезпечення.

Однією з головних цілей у публічному хмарному середовищі є економічна ефективність. Доступна вартість ресурсів є однією з головних особливостей публічної хмари для залучення користувачів. Для такої системи, що підтримує хмару, дуже важливо підтримувати витрати на найнижчому рівні. Для досягнення економічної ефективності ми впровадили динамічну систему забезпечення. Динамічний постачальник відповідає за призначення та запуск нових сервісів до системи, щоб не відставати від вхідного навантаження.

Компонент динамічного постачальника відповідає за запуск нових екземплярів сервісів у разі нестачі ресурсів. Додаток періодично перевіряє об'єм глобальної черги запитів та порівнює довжину черги з попереднім розміром. Якщо швидкість збільшення перевищує дозволену межу, він запускає новий сервіс. Як тільки запущений, сервіс автоматично приєднується до системи. І інтервал перевірки, і поріг розміру налаштовуються користувачем.

Для того, щоб запропонувати рішення для динамічного зменшення масштабу системи, щоб утримати низькі витрат ми додали програму, щоб сервіси могли припинити екземпляр якщо виконуються дві умови. Це трапляється лише в тому випадку, якщо сервіс на деякий час переходить у режим очікування, а також якщо екземпляр наближається до поновлення. Екземпляри в Amazon EC2 стягуються щогодини, і вони будуть поновлюватися щогодини, коли користувач не вимикає їх. Цей механізм допомагає нашій системі автоматично зменшувати масштаби без необхідності отримувати будь-який запит від компонента. Використовуючи ці механізми, система здатна динамічно масштабувати вгору і вниз.

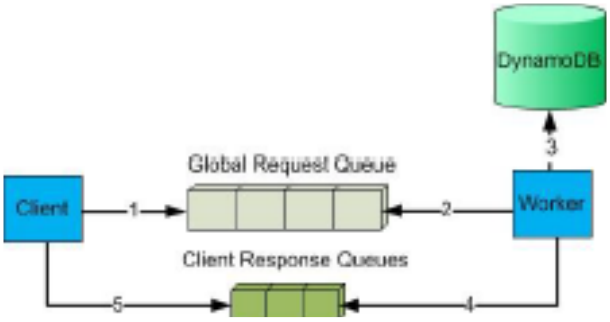

Рис. 3. Вартість зв'язку

D. Витрати на спілкування.

Затримка мережі між екземплярами в загальнодоступній хмарі порівняно висока у порівнянні з системами HPC [36, 37]. Для досягнення розумної пропускної здатності та затримки нам потрібно мінімізувати накладні витрати на зв'язок між різними компонентами системи. На Рис. 3 показано кількість зв'язків, необхідних для завершення повного циклу запуску завдання. Для виконання завдання існує 5 кроків. FlexQueue також забезпечує групування завдань під час кроків. Клієнт може надсилати кілька завдань разом. Максимальний розмір пакета повідомлень у SQS становить 256 КБ або 10 повідомлень.

E. Безпека та надійність.

Що стосується системної безпеки FlexQueue, ми покладаємось на безпеку SQS. SQS забезпечує надзвичайно безпечну систему, що використовує механізм автентифікації. Лише авторизовані користувачі можуть отримати доступ до вмісту Черг. Щоб зберегти низьку затримку, ми не додаємо жодного шифрування до повідомлень. SQS забезпечує надійність, надлишково зберігаючи повідомлення на декількох серверах і в декількох центрах обробки даних [18].

F. Деталі реалізації.

Ми впровадили всі компоненти FlexQueue в Node.js. Наша реалізація багатопотокова як в компонентах клієнта, так і в Сервісах. Багато функцій в обох цих системах, таких як моніторинг, узгодженість, кількість потоків та розмір розбиття завдань, можна налаштувати як аргумент введення програми.

Оцінка ефективності. Ми оцінюємо продуктивність FlexQueue і порівнюємо її з двома іншими розподіленими системами управління роботою, а саме Sparrow та MATRIX. Спочатку ми обговоримо їх особливості високого рівня та основні відмінності. Потім ми порівнюємо їх

ефективність з точки зору пропускної здатності та ефективності. Ми також оцінюємо затримку FlexQueue.

A. Порівняння FlexQueue з іншими системами планування.

Нам було достатньо порівняти нашу систему з Sparrow та MATRIX, оскільки ці дві системи представляють найкращі у своєму роді розподілені системи управління завданнями з відкритим кодом.

Sparrow був розроблений для досягнення мети управління мілісекундними сервісами у широкомасштабній розподіленій системі. Він використовує децентралізований підхід рандомізованої вибірки для планування завдань на робочих вузлах. У системі є кілька планувальників, кожен із яких має список сервісів і розподіляє робочі місця між сервісами, приймаючи рішення на основі довжини черги робочих місць.

У MATRIX на кожному комп'ютерному вузлі працює планувальник, виконавець та сервер ZHT. Виконавець може бути окремим ланцюжком у планувальнику. Всі планувальники повністю пов'язані з кожним, знаючи всі інші. Клієнт - це інструмент стендової розмітки, який видає запит на генерацію набору завдань і передає завдання будь-якому планувальнику. Виконавець продовжує виконувати завдання планувальника. Кожного разу, коли планувальник не має виконувати більше завдань, він ініціює адаптивний алгоритм, щоб забрати завдання у планувальників-сусідів-кандидатів. ZHT - це DKVS, який використовується для збереження метаданих задачі розподіленим, масштабованим та відмовостійким способом.

Однією з головних відмінностей між Sparrow та FlexQueue або MATRIX є те, що Sparrow розподіляє завдання, підштовхуючи їх до сервісів, тоді як FlexQueue та MATRIX використовують підтягуючий підхід. Крім того, в FlexQueue система надсилає клієнтам результати виконання завдань. Але як у Sparrow, так і в MATRIX, система не надсилає будь-який тип повідомлень назад клієнтам. Це може дозволити Sparrow та MATRIX працювати швидше, оскільки це дозволяє уникнути ще одного кроку спілкування, але також ускладнює для клієнтів з'ясування, чи успішно виконувались їх завдання.

B. Тестовий стенд.

Ми розгортаємо та запускаємо всі три системи на Amazon EC2. Ми використовували екземпляри t3.large на Amazon EC2. Ми провели всі наші експерименти в центрі обробки даних Amazon (us.eastl). Ми масштабували експерименти до 1024 вузлів. Для того, щоб зробити експерименти ефективними, клієнтський та робочий вузли запускаються на кожному вузлі. Усі екземпляри мали операційні системи Linux. Наш фреймворк повинен працювати на будь-якій ОС, що має Node12+, включаючи Ubuntu.

C. Пропускна здатність.

1) *Завдання MTC.*

Для того, щоб виміряти пропускну здатність нашої системи, ми запускаємо задачі sleep 0. Ми також порівняли пропускну здатність FlexQueue із Sparrow та MATRIX. У кожному екземплярі запущено 2 потоки клієнта та 4 робочі потоки. Кожен екземпляр подає 16000 завдань. Рис. 4 порівнює пропускну здатність FlexQueue із Sparrow та MATRIX у різних масштабах. Кожен екземпляр подає 16000 завдань із загальним обсягом до 16.38 мільйонів завдань у найбільшому масштабі.

Пропускна здатність MATRIX значно вища, ніж FlexQueue та Sparrow у 1 екземплярі. Причина полягає в тому, що MATRIX може виконувати багато детальних завдань локально, будь-яке планування або накладні витрати на мережу. Але на FlexQueue завдання повинні проходити через мережу, навіть якщо в системі запущений один вузол. Розрив між пропускною здатністю систем зменшується, оскільки накладні витрати на мережу додаються до двох інших систем. Планувальники MATRIX синхронізуються між собою, використовуючи загальний метод синхронізації. Маючи занадто багато відкритих TCP-з'єднань між сервісами та планувальниками у масштабі 256 екземплярів, MATRIX стає нестабільним. Продуктивність мережі в хмарі EC2 значно нижча, ніж у системах HPC, де MATRIX успішно запущений у масштабах 1024 вузли.

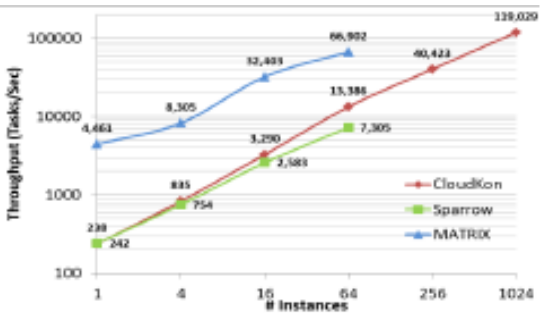

Рис. 4. Пропускна здатність FlexQueue, Sparrow та MATRIX (завдання MTC)

Sparrow є найповільнішою серед трьох систем з точки зору пропускної здатності. Вона демонструє стабільну пропускну здатність з майже лінійним прискоренням до 64 екземплярів. Оскільки кількість екземплярів збільшується більше ніж на 64, список екземплярів для вибору для кожного планувальника на Sparrow збільшується. Тому багато сервісів залишаються без роботи, а пропускна спроможність не збільшиться, як очікувалося. Ми не змогли запустити Sparrow у масштабі 128 або 256 примірників, оскільки у планувальників було відкрито занадто багато сокетів, що призвело до збоїв системи.

FlexQueue досягає хорошого прискорення в 500 разів, починаючи з 238 завдань на секунду на 1 екземплярі до 119 тис. Завдань на секунду на 1024 екземплярах. На відміну від інших двох систем, процес планування на FlexQueue не виконується екземплярами. Оскільки управління завданнями здійснює SQS, продуктивність системи в основному залежить від цієї послуги. Ми прогнозуємо, що пропускна здатність буде продовжувати масштабуватися, поки не досягне обмежень продуктивності SQS (яких ми не змогли досягти до 1024 екземплярів). Через бюджетні обмеження ми не змогли розширити наш масштаб понад 1024 екземпляри, хоча ми плануємо подати заявку на додаткові кредити Amazon AWS та перенести нашу оцінку на шкали екземплярів 10K, найбільшу допустиму кількість екземплярів на користувача без попереднього резервування.

2) *Завдання HPC.*

Ми демонструємо пропускну здатність FlexQueue під робочим навантаженням завдань HPC. Запуск завдань HPC додає системі більше накладних витрат, оскільки буде виконано більше кроків для їх виконання. Замість того, щоб виконувати завдання відразу, менеджеру сервіса потрібно пройти кілька кроків і почекати, щоб отримати достатньо ресурсів для запуску роботи. Використання DynamoDB сповільнює проботу системи та знижує ефективність робот. Але це не впливає на масштабованість. Використання FlexQueue може значно покращити час роботи робочих навантажень HPC, паралельно виконуючи завдання, яке зазвичай виконується послідовно. Ми вибрали сервіси з 4, 8 та 16 завданнями. На кожному екземплярі запущено 4 сервісні потоки. Кількість виконаних завдань за кожною шкалою для різних сервісів однакова.

Рис. 5 порівнює пропускну здатність системи у випадку запуску завдань HPC з різною кількістю підзавдань на завдання. Результати показують, що пропускна здатність імітаційних завдань із більшою кількістю завдань на одне завдання менша. На робочих місцях із більшою кількістю завдань потрібно чекати, поки більше сервісів почнуть процес вимирання. Це додає більше затримок і уповільнює роботу системи. Ми бачимо, що FlexQueue здатний досягти високої пропускної здатності в 205 сервісах в секунду. Результати також показують хорошу масштабованість, оскільки ми додаємо більше примірників.

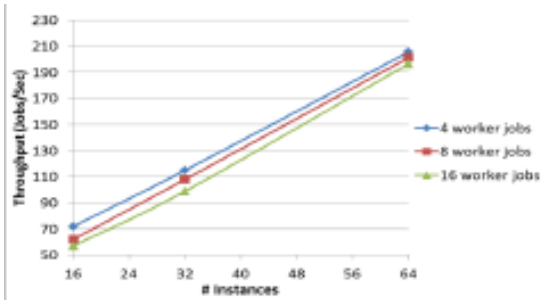

Рис. 5. Пропускна здатність FlexQueue (завдання HPC)

D. Латентність.

Для точного вимірювання затримки система повинна записати запит і відповісти на позначки часу кожного завдання. Проблема Sparrow та MATRIX полягає в тому, що в процесі виконання робочого часу сервіси не надсилають сповіщення клієнтам. Тому неможливо виміряти затримку кожного завдання, порівнюючи мітки часу з різних вузлів. У цьому розділі ми виміряли латентність FlexQueue та проаналізували латентність різних етапів процесу штампа.

На Рис. 6 показана затримка FlexQueue для режиму сну 0 мс, масштабування від 1 до 1024 екземплярів. Кожен екземпляр запускає 1 клієнтський потік і 2 робочі потоки і надсилає 16000 завдань на екземпляр.

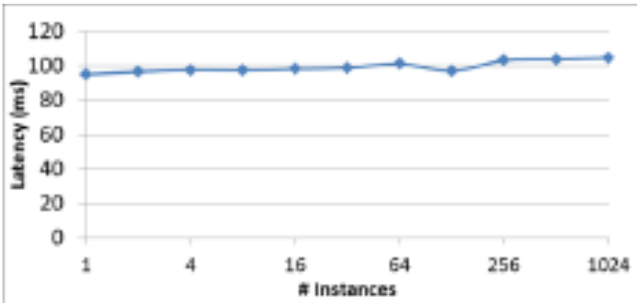

Рис. 6. Затримка завдань FlexQueue sleep 0 ms

Час затримки системи на 1 вузлі є відносно високим, показуючи накладні витрати 95 мс, додані системою. Але це буде прийнятно у більших масштабах. Той факт, що затримка не збільшується більше 10 мс при збільшенні кількості екземплярів з 1 екземпляра до 1024, свідчить про стабільність FlexQueue. SQS як пул завдань - це надзвичайно масштабований відправник, резервне копіювання якого складається з декількох серверів, що забезпечує відправника дуже масштабованим. Таким чином, масштабування системи шляхом додавання потоків та збільшення кількості завдань не впливає на продуктивність SQS. Клієнтський та сервісний вузли завжди обробляють однакову кількість завдань у різних масштабах. Тому масштабування не впливає на екземпляри. FlexQueue включає кілька компонентів, і його продуктивність та затримка залежать від різних компонентів. Результат затримки на Рис. 6 не показує нам жодних подробиць щодо продуктивності системи. Для того, щоб проаналізувати ефективність різних компонентів, ми вимірюємо час, який кожне завдання витрачає на різні компоненти системи, реєструючи час під час процесу виконання.

Рис. 7, 8 та 9 відповідно показують сукупний розподіл етапу завдання доставки, етапу результату доставки та етапу виконання завдань на FlexQueue. Кожен етап спілкування має три етапи: надсилання, чергу та отримання. Час затримки викликів API SQS, включаючи send-task та receive-task для обох, досить високий у порівнянні з часом виконання завдань на FlexQueue. Причиною цього є дорога вартість дзвінків API веб-служби, яка використовує формат XML для зв'язку. Обробка сервісом займає 16 мс більше 50% випадків. Сюди входить DynamoDB на який відводиться 8 мс у понад 50% випадків. Це свідчить, що гіпотетично затримка FlexQueue може значно покращитись, якщо ми використовуємо низьку розподілену чергу повідомлень, яка може гарантувати рівну доставку завдань. Ми розглянемо це докладніше в наступному розділі роботи.

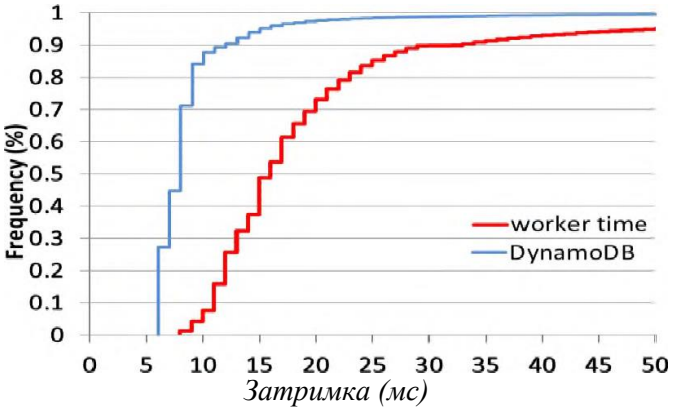

Рис. 7. Кумулятивний розподіл затримки на етапі виконання завдання

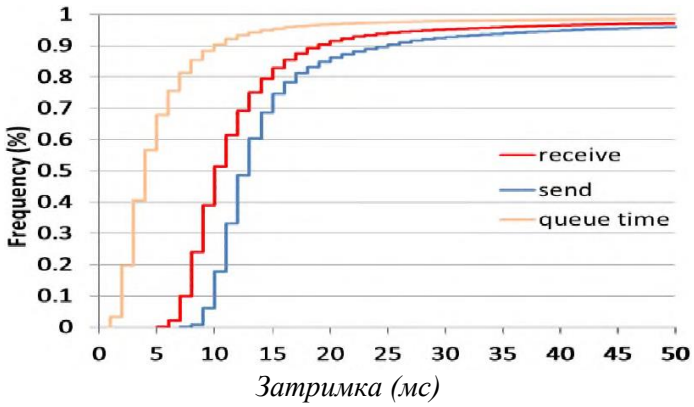

Рис. 8. Кумулятивний розподіл затримки на етапі подання завдання

Іншим помітним моментом є різниця між часом виконання завдання та результатом доставки як у черзі, так і при отриманні назад, навіть якщо вони мають однакові виклики API. Час, витрачений завданнями на чергу відповідей, перевищує час, витрачений на чергу запитів. Причина полягає в тому, що у кожному екземплярі є два потоки сервісів і лише один потік клієнта. Тому частота виконання завдань вища, коли завдання тягнуть робочі нитки.

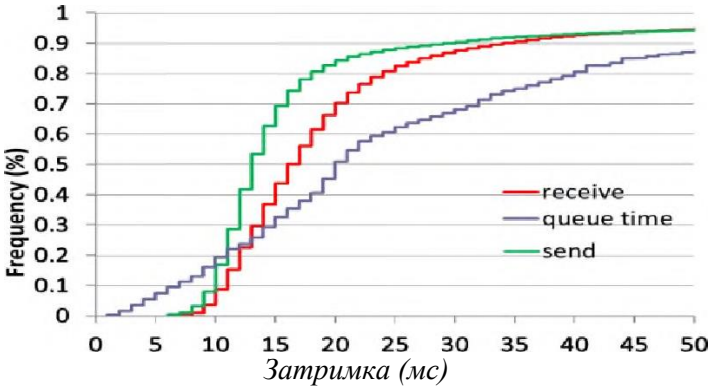

Рис. 9. Кумулятивний розподіл затримки на етапі доставки результату

E. Ефективність FlexQueue.

Для системи дуже важливо ефективно керувати системами. Досягнення високої ефективності в розподілених системах планування сервісів не є тривіальним. Важко справедливо розподілити навантаження на всіх сервсіах і зайняти всі вузли під час виконання у більших масштабах.

Для того, щоб показати ефективність системи, ми розробили два набори експериментів. Ми перевіряємо ефективність системи на випадок однорідних та неоднорідних завдань. Однорідні завдання мають певну тривалість. Тому їх легше розподілити, оскільки планувальник передбачає, що для їх запуску потрібен однаковий час. Це може дати нам хороший відгук про ефективність системи у випадку запуску різних типів завдань з різною деталізацією. Ми також можемо оцінити здатність системи виконувати дуже довгі завдання. Проблема першого експерименту полягає в тому, що для виконання всіх завдань потрібна однакова кількість часу. Це може суттєво вплинути на ефективність системи, якщо планувальник не бере до уваги об'єм завдань. Випадкове робоче навантаження може показати, як планувальник працюватиме у випадку запуску реальних додатків.

1) Однорідні навантаження.

У цьому розділі ми порівнюємо ефективність FlexQueue із Sparrow та MATRIX щодо допоміжних завдань. На Рис. 10 показана ефективність завдань на системи 1, 16 та 128 мс. Ефективність FlexQueue при виконанні завдань на 1 мс нижча, ніж у інших двох системах. Як ми вже згадували раніше, затримка FlexQueue велика для дуже коротких завдань через значні накладні витрати мережі, додані в циклі виконання. Матриця має кращу ефективність на менших масштабах, але, як показує тенденція, ефективність надзвичайно падає, доки система не вийде з ладу через занадто велику кількість компіляцій TCP на масштабах 128 екземплярів або більше. Для завдань сну 16 мс ефективність FlexQueue становить близько 40%, що є низьким (у

порівнянні з іншими системами). Ефективність MATRIX починається з більш ніж 93% на одному екземплярі, але знову ж вона падає до нижчої ефективності, ніж FlexQueue у більшій кількості екземплярів. Ми можемо помітити, що ефективність FlexQueue дуже стабільна в порівнянні з двома іншими системами в різних масштабах. Це показує, що FlexQueue досягає кращої масштабованості. У завданнях 128 мс для сну ефективність FlexQueue сягає 88%. Знову ж таки, результати показують, що ефективність MATRIX падає у більших масштабах.

Sparrow демонструє дуже хорошу та стабільну ефективність, виконуючи однорідні завдання до 64 екземплярів. Ефективність падає після цієї шкали для коротших завдань. Маючи занадто багато сервісів для розподілу завдань, планувальник не може мати ідеального балансу навантаження, а деякі сервіси залишаються без обробки. Тому система буде недостатньо використана, а ефективність знизиться. Система аварійно завершує роботу на масштабах 128 або більше через те, що в планувальниках надто багато сокетів

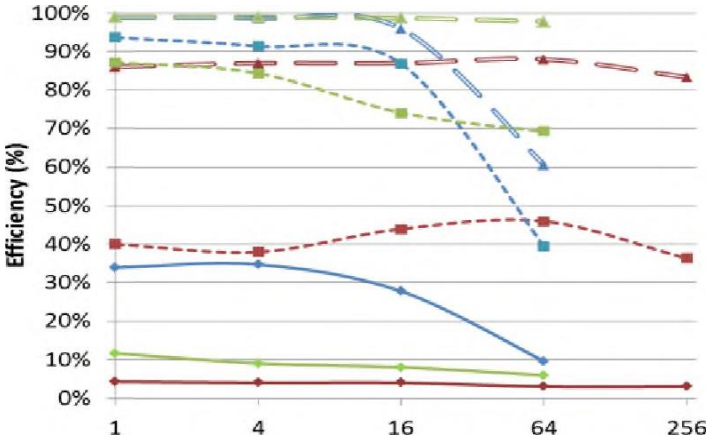

Рис. 10. Ефективність FlexQueue, Sparrow та MATRIX, що виконують однорідні робочі навантаження різної тривалості завдань (завдання 1, 16, 128 мс)

2) Неоднорідні навантаження.

Для того, щоб виміряти ефективність, ми дослідили найбільший доступний слід реальних робочих навантажень MTC [38] та відфільтрували журнали, щоб виділити лише допоміжні завдання, які об'єднали близько 2,07 млн завдань з діапазоном часу роботи від 1 мілісекунди до 1 секунди. Завдання були подані випадковим чином. Середня тривалість завдання різних екземплярів відрізняється одна від одної.

Кожен екземпляр виконує в середньому 2K завдань. Порівняння ефективності на Рис. 9 показує подібні тенденції щодо FlexQueue та MATRIX. В обох системах сервіс тягне завдання лише тоді, коли у нього є доступні ресурси для запуску завдання. Тому той факт, що тривалість виконання завдань різний, не впливає на ефективність системи. З іншого боку, у Sparrow планувальник розподіляє завдання, надсилаючи їх сервісам, у яких менше черги завдань для виконання в черзі. Той факт, що завдання мають різний час виконання, вплине на ефективність системи. Деякі з сервісів можуть мати багато довгих завдань, а багато інших сервісів можуть виконувати короткі завдання.

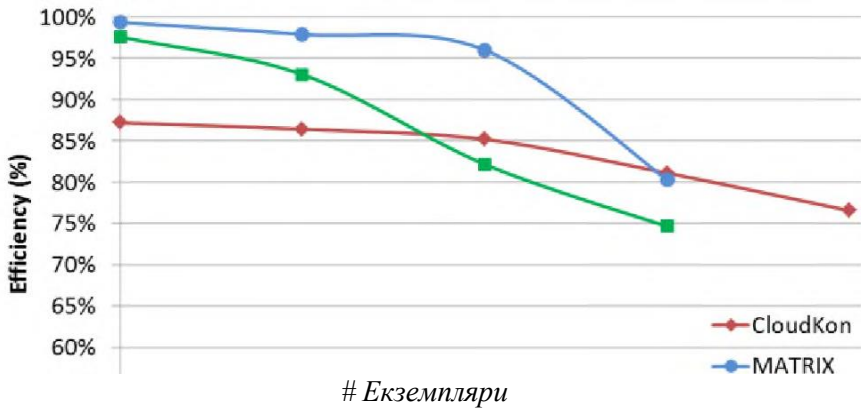

Рис. 11. Ефективність систем, що використовують неоднорідні навантаження.

Будучи недостатньо використаним, ефективність Sparrow має найбільше падіння з 1 примірника до 64 примірників. Система не працювала в 128 випадках і більше. Подібним чином ефективність MATRIX почалася з високої ефективності, але почала суттєво падати через занадто багато відкритих сокетів на TCP-з'єднаннях. Ефективність FlexQueue не така висока, як у двох інших системах, але вона є більш стабільною, оскільки зменшується лише на 6% від 1 до 64 примірників порівняно з MATRIX на 19% і Sparrow, що падає на 23%. Знову ж таки, FlexQueue була єдиною функціональною системою на 256 екземплярах із 77% ефективністю.

F. Накладні витрати на послідовність.

У цьому розділі ми оцінюємо вплив узгодженості виконання завдань на FlexQueue. На Рис. 10 показана система запуску для 16мс сну з контролером дублювання вмикається і вимикається. Витрати на інші завдання зі сну були подібні до цього експерименту. Отже, ми включили лише один із експериментів у цю роботу.

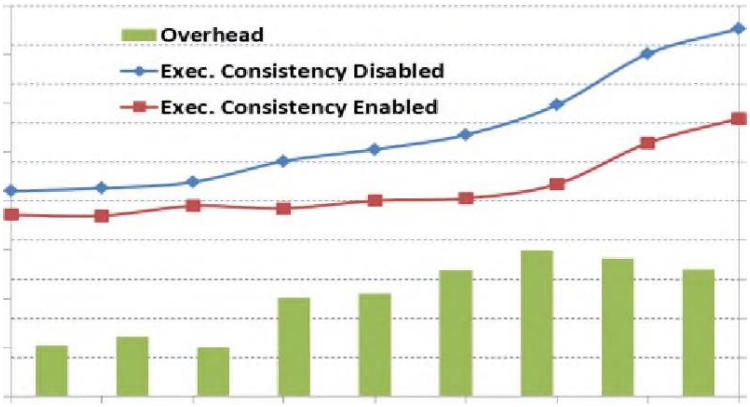

Рис. 12. Виконайте накладні витрати на послідовність виконання завдань на FlexQueue

Накладні показники консистенції збільшуються зі збільшенням масштабу. Невідповідність у різних масштабах є результатом змінної кількості повторюваних повідомлень при кожному запуску. Це призводить до більш випадкової продуктивності системи в різних експериментах, загалом накладні витрати за шкалою менше 10 становлять менше 15%. Ці накладні витрати служать здебільшого для успішних операцій запису на DynamoDB. Ймовірність отримати дублікати завдань зростає у більших масштабах. Тому винятків буде більше. Це призводить до вищих накладних витрат. Накладні витрати у більших масштабах сягають до 35%. Однак ставка накладних витрат стабільна і не перевищує цю ставку. Використання розподіленої черги повідомлень, яка гарантує доставку точно один раз, може значно покращити продуктивність.

Пов'язана робота. Планувальники завдань можуть бути централізованими, де єдиний диспетчер керує поданням роботи та оновленням стану виконання; або ієрархічний, де кілька диспетчерів організовані в топології на основі дерева; або розподілений, де кожен обчислювальний вузол підтримує власну структуру виконання завдань.

Condor [3] був реалізований для використання невикористаних циклів процесора на робочих станціях для тривалих пакетних робіт. Slurm [2] - це менеджер ресурсів, призначений для кластерів Linux усіх розмірів. Він надає користувачам ексклюзивний та/або невиключний доступ до ресурсів протягом певного періоду часу, щоб вони могли виконувати роботу, і забезпечує основу для початку, виконання та моніторинг роботи над набором виділених вузлів. Портативна пакетна система (PBS) [5] була спочатку розроблена для задоволення потреб HPC. Вона може керувати пакетними та взаємоактивними робочими місцями, а також додавати можливість сигналізації, повторного запуску та зміни завдань. LSF Batch [19] - це компонент розподілу навантаження та черги пакетів у наборі інструментів управління навантаженням.

Висновки та майбутня робота. Широкомасштабні розподілені системи вимагають ефективної системи планування завдань для досягнення високої пропускної здатності та використання системи. Важливо, щоб система планування забезпечувала високу пропускну здатність та низьку затримку на більших масштабах та додавала мінімальну накладну витрату до робочого процесу. FlexQueue - це розподілена програма виконання завдань, що підтримує хмару,

яка працює в хмарі Amazon AWS. Це унікальна система з точки зору виконання робочих навантажень HPC та MTC у загальнодоступному хмарному середовищі. Використання служби SQS дає FlexQueue перевагу масштабованості. Оцінка FlexQueue доводить, що вона є дуже масштабованою та забезпечує стабільну продуктивність у різних масштабах. Ми протестували нашу систему до 1024 екземплярів. FlexQueue зміг перевершити інші системи, такі як Sparrow та MATRIX, у масштабах 128 екземплярів або більше з точки зору пропускної здатності. FlexQueue досягає ефективності до 87%, виконуючи однорідні та неоднорідні дрібнозернисті завдання на секунду. У порівнянні з іншими системами, такими як Sparrow, він забезпечує меншу ефективність на менших масштабах. Але у більших масштабах він досягає значно вищої ефективності.

Існує багато напрямків для подальшої роботи. Один із напрямків - зробити систему повністю незалежною та протестувати її на різних державних та приватних хмарах. Ми збираємось впровадити послугу, подібну до SQS, з високою пропускною здатністю на більших масштабах доступу. За допомогою інших систем, таких як розподілена хеш-таблиця ZHT [32], ми зможемо реалізувати таку послугу. Іншим майбутнім напрямком цієї роботи є впровадження більш тісно пов'язаної версії FlexQueue і тестування її на суперкомп'ютерах та середовищах HPC під час запуску завдань HPC розподіленим способом та порівняння безпосередньо з Slurm та Slurm++ в тому ж середовищі.

ЛІТЕРАТУРА

1. P. Kogge, et. al., "Exascale computing study: Technology challenges in achieving exascale systems," 2008.
2. M. A. Jette et. al, "Slurm: Simple linux utility for resource management". In *Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for Prarallel Procesing* (JSSPP) 2003 (2002), Springer-Verlag, pp. 44-60.
3. D. Thain, T. Tannenbaum, M. Livny, "Distributed Computing in Practice: *The Condor Experience" Concurrency and Computation: Practice and Experience* 17 (2-4), pp. 323-356, 2005.
4. J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke. "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, 2002.
5. B. Bode et. al. "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters," *Usenix, 4th Annual Linux Showcase & Conference*, 2000.
6. W. Gentzsch, et. al. "Sun Grid Engine: Towards Creating a Compute Power Grid," *1st International Symposium on Cluster Computing and the Grid* (CCGRID'01), 2001.
7. C. Dumitrescu, I. Raicu, I. Foster. "Experiences in Running Workloads over Grid3", *The 4th International Conference on Grid and Cooperative Computing* (GCC 2005), 2005.
8. I. Raicu, et. al. "Toward Loosely Coupled Programming on Petascale Systems," *IEEE/ACM Super Computing Conference* (SC'08), 2008.
9. I. Raicu, et. al. "Falkon: A Fast and Light-weight tasK executiON Framework," *IEEE/ACM SC 2007*.
10. S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. "Dremel: Interactive Analysis of Web-Scale Datasets. Proc." *VLDB Endow*., 2010.
11. L. Ramakrishnan, et. al. "Evaluating Interconnect and virtualization performance for high performance computing", *ACM Performance Evaluation Review*, 40(2), 2012.
12. P. Mehrotra, et. al. "Performance evaluation of Amazon EC2 for NASA HPC applications". In *Proceedings of the 3rd workshop on Scientific Cloud Computing* (ScienceCloud '12). ACM, NY, USA, pp. 41-50, 2012.
13. Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. "Case study for running HPC applications in public clouds," In *Proc. of ACM Symposium on High Performance Distributed Computing*, 2010.
14. G. Wang and T. S. Eugene. "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center". *In IEEE INFOCOM*, 2010.
15. I. Raicu, Y. Zhao, I. Foster, "Many-Task Computing for Grids and Supercomputers," *1st IEEE Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS) 2008.
16. I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Computer Science Dept., University of Chicago, Doctorate Dissertation, March 2009
17. Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/
18. Amazon SQS, [online] 2013, Retrieved from http://aws.amazon.com/sqs/
19. LSF: http://platform.com/Products/TheLSFSuite/Batch, 2012.
20. L. V. Kal´e et. al. "Comparing the performance of two dynamic load distribution methods," *In Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, August 1988.
21. W. W. Shu and L. V. Kal´e, "A dynamic load balancing strategy for the Chare Kernel system," *In Proceedings of Supercomputing* '89, pages 389–398, November 1989.
22. A. Sinha and L.V. Kal´e, "A load balancing strategy for prioritized execution of tasks," In International Parallel Processing Symposium, pages 230–237, April 1993.

23. M.H. Willebeek-LeMair, A.P. Reeves, “Strategies for dynamic load balancing on highly parallel computers,” *In IEEE Transactions on Parallel and Distributed Systems*, volume 4, September 1993.
24. G. Zhang, et. al, “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers,” *In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW 10, pages 436-444, Washington, DC, USA, 2010.
25. K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. “Sparrow: distributed, low latency scheduling”. In *Proceedings of the TwentyFourth ACM Symposium on Operating Systems Principles* (SOSP '13). ACM, New York, NY, USA, 69-84.
26. M. Schwarzkopf, A Konwinski, M. Abd-el-malek, and J. Wilkes, Omega: Flexible, scalable schedulers for large compute clusters. *In Proc. EuroSys* (2013).
27. Frigo, et. al, “The implementation of the Cilk-5 multithreaded language,” *In Proc. Conf. on Prog. Language Design and Implementation* (PLDI), pages 212–223. ACM SIGPLAN, 1998.
28. R. D. Blumofe, et. al. “Scheduling multithreaded computations by work stealing,” *In Proc. 35th FOCS*, pages 356–368, Nov. 1994.
29. V. Kumar, et. al. “Scalable load balancing techniques for parallel computers,” *J. Parallel Distrib. Comput.,* 22(1):60–79, 1994.
30. J. Dinan et. al. “Scalable work stealing,” *In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.
31. A. Rajendran, Ioan Raicu. "MATRIX: Many-Task Computing Execution Fabric for Extreme Scales", Department of Computer Science, Illinois Institute of Technology, MS Thesis, 2013.
32. T. Li, et al., “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” *in IEEE International Parallel & Distributed Processing Symposium* (IPDPS ’13), 2013.
33. Amazon DynamoDB (beta), Amazon Web Services, [online] 2013, http://aws.amazon.com/dynamodb
34. P. Mell and T. Grance. “NIST definition of cloud computing.” National Institute of Standards and Technology. October 7, 2009.
35. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” *in Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing*, Boston, MA, June 2010.
36. P. Mehrotra, et al. 2012. “Performance evaluation of Amazon EC2 for NASA HPC applications” In (ScienceCloud '12). ACM, New York, NY, pp. 41-50.
37. I. Sadooghi, et al. “Understanding the cost of cloud computing”. Illinois Institute of Technology, Technical report. 2013.
38. I. Raicu, et al. “The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems,” *ACM HPDC* 2009.
39. I. Raicu, et al. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010.
40. Y. Zhao, et al. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, Nova Publisher 2008.
41. I. Raicu, et al. "Towards Data Intensive Many-Task Computing", book chapter in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", IGI Global Publishers, 2009.
42. Y. Zhao, et al. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE CyberC 2011.
43. M. Wilde, et al. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", SciDAC 2009.