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

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 681.301

«До захисту допущено»
Завідувач кафедри СПСКС

_____Віталій РОМАНКЕВИЧ
(підпис) (ім'я, прізвище)
"___"_____2020р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
на тему: Програмні засоби організації розподілених безсерверних обчислень

Виконав: студент ІІ курсу, групи КВ-91мн
Маслов Вадим Ігорович _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник доц.каф.СПСКС, к.т.н. Петрашенко А.В._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доц.каф.СПСКС, к.т.н. Заболотня Т.М. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю доцент, с.н.с.,к.т.н. Юлія БОЯРІНОВА _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

Київ – 2020 року

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

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
за освітньо-професійною програмою
Спеціальність 123 Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
«1» листопада 2020р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Маслов Вадим Ігорович
(прізвище, ім'я, по батькові)

1. Тема дисертації «Програмні засоби організації розподілених безсерверних обчислень»,
науковий керівник дисертації к.т.н., доцент Петрашенко А.В,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «12» листопада 2020 р. №3298-С

2. Термін подання студентом дисертації 10 грудня 2020 р.

3. Об'єкт дослідження: системи фонової асинхронної обробки задач на основі хмарних технологій.

4. Предмет дослідження: методи організації розподілених безсерверних обчислень на основі скінченного автомату.

5. Перелік завдань, які потрібно розробити: аналіз існуючих методів статичного аналізу зображень; обгрунтування доцільності розробки алгоритму; обгрунтування використання метрик продуктивності розробленої систем; ПЗ оркестрації хмарних функцій;

6. Перелік ілюстративного матеріалу - презентація _____

7. Перелік публікацій - 2 тези. Подання статті на конференцію «ICCSEEA2021: The Fourth International Conference on Computer Science, Engineering and Education Applications» _____

_____

8. Дата видачі завдання 5 листопада 2019 р. _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Вивчення літератури за тематикою дисертації	10.09.2019	
2.	Підготовка матеріалів першого розділу магістерської дисертації	21.11.2019	
3.	Підготовка матеріалів другого розділу магістерської дисертації	14.02.2020	
4.	Підготовка матеріалів третього розділу магістерської дисертації	19.04.2020	
5.	Підготовка матеріалів четвертого розділу магістерської дисертації	05.05.2020	
6.	Підготовка матеріалів п’ятого розділу магістерської дисертації	30.06.2020	
7.	Розробка програмної бібліотеки	19.07.2020	
8.	Розробка моделі, що демонструє приклад використання програмної бібліотеки	07.09.2020	
9.	Оформлення документації магістерської дисертації	10.10.2020	
10.	Попередній розгляд магістерської дисертації на кафедрі	26.11.2020	

Студент _____ Вадим МАСЛОВ
(підпис)

Науковий керівник дисертації _____ Андрій ПЕТРАШЕНКО

РЕФЕРАТ

Актуальність теми

Наразі використання хмарних апаратних рішень є найбільш простою та популярною стратегією формування робочої архітектури. Наслідком цього стала поява FaaS рішень, основна задача яких - це позбавлення систем часу простою, під час якого навантаження на код відсутнє або мінімальне. Такі системи вже імплементовані в більшості великих провайдерів, проте постає інша проблема - це контроль роботи цих хмарних функцій та можливість їх об’єднання у більш складні сценарії виконання. Саме такі системи нададуть можливість створювати виконання виконання складних сценаріїв роботи без значних затрат на розробку. В магістерській роботі описується програмна бібліотека та імплементація на її основі, що надає можливість кінцевому розробнику створити таку систему оркестрації без значних трудозатрат.

Об’єкт дослідження

Системи фонової асинхронної обробки задач на основі хмарних технологій

Предмет дослідження

Методи організації розподілених безсерверних обчислень на основі скінченного автомату

Мета роботи

Покращення часу відповіді на створення нового конвеєру обробки і затримкою між виконанням кожного його кроку та зменшення операційних витрат на підтримання інфраструктури

Наукова новизна

Розроблено програмну бібліотеку, що відрізняється від існуючих насамперед гнучкістю налаштування виконання окремих задач та можливістю самостійно надавати необхідні для виконання ресурси, при цьому не втрачаючи швидкодії та часу реагування.

Даний підхід дозволяє кінцевому користувачу абстрагуватися від послуг конкретного постачальника таким чином надаючи можливість не зав’язуватися на

повний пакет послуг, а лише відокремлювати необхідні частини від різних постачальників або, за необхідністю, використовувати власні апаратні можливості.

Таким чином це дозволяє користувачу значно зменшити операційні витрати на підтримання цільової інфраструктури без втрат на швидкість виконання, часу реагування та здатності до масштабування системи.

Практична цінність

На даний момент існує велика кількість сценаріїв використання даного підходу організації роботи. Найбільшу цінність такий підхід надає при використанні асинхронних та фонових обробок. У даному випадку нам не потрібна висока швидкодія для обробки та зазвичай навантаження на такі частини мінімальне, тому ми можемо утримувати тільки ті апаратні ресурси, що нам насправді потрібно, але так само, якщо раптово навантаження зросте, то ми матимемо змогу масштабувати таке рішення, оскільки підхід з графом контролю дає нам можливість запуску ізольованих контекстів у, майже, необмеженій кількості без заважання одне одному. Таким чином, такі системи зможуть адаптивно реагувати на зростання та падіння навантаження і виконувати покладені на них задачі.

Апробація роботи

Запропонований підхід був представлений та обговорений на науковій конференції магістрантів та аспірантів «Прикладна математика та комп‘ютинг» ПМК-2020 (Київ, 18 - 19 листопада 2020 р.) та на VII Міжнародній науково-технічній Internet конференції «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами», яка проводилась 26 листопада 2020 р. у Національному університеті харчових технологій

Структура та обсяг роботи

Магістерська робота складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботу, показано наукову новизну і практичну цінність роботи. Зроблено короткий огляд на технологічний стек, який використовується для створення ПЗ.

У першому розділі розглянуто і проаналізовано існуючі методи для створення FaaS архітектур та було проведено їх порівняння.

У другому розділі наведено опис теоретичної частини обраних алгоритмів та структур даних .

У третьому розділі наведено особливості практичної реалізації програмної бібліотеки, переваги використання обраної мови програмування, а також додаткових інструментів.

У четвертому розділі проведено аналіз представленої обробки у порівнянні з існуючими аналогами.

У висновках описані результати проведеної роботи.

Ключові слова: control flow graph, скінченний автомат, хмарні обчислення, IaaS, FaaS.

ABSTRACT

Actuality of theme

Currently, the use of cloud hardware solutions is the simplest and most popular strategy for building a working architecture. The result is the emergence of FaaS solutions, the main task of which is to deprive systems of downtime, during which the load on the code is absent or minimal. Most cloud providers already has those systems implemented, but there is another problem - the control of these cloud functions and the ability to combine them into scenarios that are more complex. Such systems will provide an opportunity to create execution of execution of difficult scenarios of work without considerable expenses for development. The master's thesis describes the software library and implementation based on it, which allows the final developer to create such an orchestration system without significant labor costs.

Object of study

Background asynchronous task processing systems based on cloud technologies

Subject of study

Methods of organizing distributed serverless calculations based on a finite state machine

The purpose of the work

Response time improvement for create of processing pipelines and decrease of the delay between each step and reduce operating costs for infrastructure maintenance

Scientific novelty

Developed software library differs from the existing ones primarily by the flexibility of setting up the execution of individual tasks and the ability to provide independent setup for the necessary execution resources, without performance and response time degradation.

This approach allows the end user to abstract from the services of a particular provider, thus providing the opportunity not to commit to a full package of services, but only to separate the necessary parts from different providers or, if necessary, use their own hardware capabilities.

Thus, it provides significant reduce of operating costs of maintaining the target infrastructure without loss of execution speed, response time and scalability of the system.

Practical value

Currently, there are a large number of scenarios for using this approach to work organization. This approach provides the greatest value when using asynchronous and background treatments. In this case, we do not need high speed processing and usually the load on such parts is minimal. Thus, we can keep only the hardware resources that we really need, but also, if the load suddenly increases, we will be able to scale this solution, because the control graph approach allows us to run isolated contexts in an almost unlimited number without interfering with each other. Thus, such systems will be outfitted with ability to be able for adaptive response to load spikes and perform the tasks assigned to them.

Approbation of the work

The proposed approach was presented and discussed at the scientific conference of undergraduates and graduate students "Applied Mathematics and Computing" PMK-2020 (Kyiv, November 18 - 19, 2020) and at the VII International Scientific and Technical Internet Conference "Modern methods, information, software and technical support of management systems of organizational, technical and technological complexes ", which was held on November 26, 2020 at the National University of Food Technologies

Structure and the scope of work

The master's thesis consists of an introduction, four chapters and conclusions.

The introduction presents a general description of the work, shows the scientific novelty and practical value of the work. A brief overview of the technology stack used to create software is given.

The first section discusses and analyzes existing methods for creating FaaS architectures and compares them.

The second section describes the theoretical part of the selected algorithms and data structures.

The third section presents the features of the practical implementation of the software library, the benefits of using the selected programming language, as well as additional tools.

In the fourth section, the analysis of the presented processing in comparison with existing analogues is carried out. The conclusions describe the results of the work.

Keywords: control flow graph, finite state machine, cloud computing, IaaS, FaaS.

Зміст

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Greenthread - це аналоги thread, що керуються рантаймом програми чи віртуальної машини замість операційної системи

Observability (з англ. “спостережуваність”) - у теорії керування, характеристика системи, що показує здатність відновити стани системи після її виходу

On-demand - в контексті IaaS, це принцип розгортання апаратних потужностей за запитом

On-premises - в контексті IaaS, це принцип розгортання апаратних потужностей на власних засадах

Prefork - це принцип попереднього виконання fork (або розмноження) процесу

RPS (англ. “Requests per second”) - це метрика, що характеризує потужність

Runtime - це конекст програми під час її виконання

Thread - найменша кількість програмних інструкцій, що можуть бути незалежно керовані планувальником

vCPU - це віртуальний процесор, що використовується в рамках IaaS і є абстракцією навколо реальних апаратних ресурсів

Інстанс (з англ. “примірник”) - в рамках сервісної архітектури це копія або репліка певного сервісу

Кластер - в контексті IaaS, кластер це набір серверів, що об’єднано в єдину мережу

Персистентне сховище (англ. “Persistent storage”) - це сховище або носій інформації, що здатний зберігати дані навіть після вимкнення живлення

Вступ

Прогрес розвитку хмарних технологій значно змінив підходи до розгортання сервісів. IaaS, SaaS, PaaS - всі ці підходи до налаштування та розгортання кінцевого продукту значною мірою є продуктом цього розвитку. Головним поштовхом до розвитку цих технологій стало спрощення віртуалізації - створення таких інструментів як Docker значно зменшили вимоги до апаратних ресурсів та надали можливість створювати готові образи програмних платформ, всі залежності для яких вже було включено в такі образи.

Спрощення процесу розгортання, доставки та зберігання кінцевого коду дали можливість розвитку мікросервісної архітектури, основним принципом якої є розділення громіздких монолітних систем, що поєднують в собі повний робочий цикл додатку на більш дрібні мікросервіси, що розділяють додаток на більш дрібні під-додатки, що інкапсулюють різні контекстуальні області глобального додатку. Таким чином ми отримуємо можливість використовувати більш компактні обчислювальні станції, гнучке масштабування ділянок робочого процесу, що мають більше навантаження та отримуємо доступ до поліглотного програмування, що в свою чергу дає можливість обирати найбільш підходящу мову програмування для конкретного сервісу.

Наступним кроком такого розвитку стало відокремлення окремих функцій, тобто створення FaaS архітектурного підходу. Так само як у випадку мікросервісної архітектури ми маємо можливість зменшувати ресурси тих ділянок загального додатку, що використовуються рідко, так і у випадку FaaS - ми маємо змогу відокремити мало використані, так звані “холодні”, функції від загального коду та не виділяти постійних апаратних ресурсів для них. Натомість, ми отримуємо змогу викликати ці функції лише за необхідності та надавати апаратні ресурси тільки на

час виклику. Це особливо корисно у випадках фонової та асинхронної обробки певних ресурсів системи.

При такому архітектурному підході ми маємо змогу масштабувати велику кількість кількість таких функцій та створювати аналоги SIMD та MIMD конвеєрів. Проте, в такому випадку виникає проблема з observability даної системи, тобто ми не зможемо точно визначити статус кожного такого конвеєра чи даних, що йдуть ними.

Дану проблему можна вирішити створивши зовнішній монітор або оркестратор, що матиме змогу зберігати статуси кожної події до персистентного сховища, а також контролювати навантаження та координувати всі виклики. Такий оркестратор буде надавати можливість ініціювати додавання даних на конвеєр за допомогою різних джерел та конструювати послідовності різних викликів окремих функцій і контролювати та проводити моніторинг статусів виконання на кожному етапі циклу життя коду.

РОЗДІЛ 1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1 Огляд існуючих альтернатив FaaS

На сьогодні, послуги надання хмарних обчислень пропонують велика кількість постачальників. Проте, основними постачальниками є:

1. Amazon AWS
2. Microsoft Azure
3. Google Cloud

Кожен постачальник хмарних рішень здатний до розгортання FaaS рішень, оскільки з точки зору виконання коду такі системи майже нічим не відрізняються від звичайних мікросервісів. Усі представлені постачальники надають послуги з розгортання FaaS, що мають назву, відповідно:

1. AWS Lambda
2. Azure Functions
3. Google Cloud Functions

1.2 Amazon AWS

Cloud FaaS рішення від провайдеру Amazon - це рішення, що пропонує на вибір фіксовану кількість середовищ виконань: Node.js, Python, Ruby, Java, Go, C#, та PowerShell. Написані функції мають підпорядковуватися певному контракту написання функції та мати чітко визначену сигнатуру. Доставка функцій для розгортання виконується на основі утиліти aws-cli, яка, в свою чергу виконує пересилання попередньо створеного zip архіву, що повинен містити код програми та всі потрібні залежності. Для створення контексту потрібно попередньо

визначити ім’я функції та права на доступ IAM. Додатково доступні інтеграції з іншими сервісами AWS, як-от S3, CloudWatch, IoT, EFS та інші. Таким чином можна створювати системи обробки нових даних, сповіщень, подій ітд та поєднувати їх в конвеєри різної складності. Мінусами такого рішення є:

1. Холодний старт функцій

Коли функція запускається у відповідь на подію, може бути невелика затримка між подією та часом запуску функції. Якщо ваша функція не використовувалася протягом останніх 15 хвилин, затримка може становити 5-10 секунд, що ускладнює покладання на лямбду для критично важливих програм.

2. Обмеження виконаних функцій

Функції AWS Lambda мають до них кілька обмежень:

- Час виконання. Функція лямбда закінчиться через 15 хвилин роботи. Дане обмеження змінити неможливо, тому якщо виконання функції зазвичай триває більше 15 хвилин, система контролю роботи функцій може припинити роботу вашої функції.
- Пам'ять, доступна для функції. Варіанти обсягу оперативної пам'яті, доступної для функцій AWS Lambda, варіюються від 128 МБ до 3008 МБ з кроком 64 МБ.
- Розмір пакета коду. Розмір упакованого коду функції не повинен перевищувати 50 МБ, а розпакована версія не повинна перевищувати 250 МБ.
- Паралельність. За замовчуванням паралельне виконання всіх функцій AWS Lambda в одному обліковому записі AWS обмежується до 1000. Будь-яке виконання Лямбда, ініційоване вище вашого обмеження паралельності, буде обмежено і буде змушене чекати, поки квота не звільниться.
- Розмір корисного навантаження. При використанні Amazon API Gateway для активації Lambda функцій у відповідь на HTTP-запити, максимальний розмір корисного навантаження, який API Gateway може обробити, становить 10 МБ.

3. Не завжди рентабельно

На AWS Lambda ви платите лише за використану функціональну час виконання, плюс будь-які пов'язані з цим витрати, такі як плата за обслуговування мережевого трафіку. Такий підхід дозволяє значно скоротити витрати для on-demand задач або періодичних cron викликів. Однак, коли навантаження на вашу програму зростає, вартість AWS Lambda зростає пропорційно і може виявитися вищою, ніж вартість аналогічної інфраструктури на AWS EC2 або інших хмарних постачальниках. [1]

4. Обмежена кількість підтримуваних середовищ виконання

Незважаючи на те, що AWS Lambda дозволяє додавати власні середовища виконання, їх створення може бути великою роботою. Отже, якщо версія мови програмування, яку ви використовуєте, не підтримується AWS Lambda, можливо, вам краще скористатися AWS EC2 або іншим хмарним постачальником.

1.2 Microsoft Azure

Наступним постачальником є Azure Functions від компанії Microsoft. Azure Functions також підтримує обмежену кількість середовищ, проте дещо більшу, додатково включаючи F# та TypeScript. Також є підтримка виконання в мовах, що не входять до вищезазначеного списку, проте з обмеженням - виконуваний файл має бути попередньо статично скомпільований або якимось чином бути частково зібраний з усіма динамічними залежностями (як, наприклад, PyInstaller для Python). Так само як і у AWS Lambda - Azure Functions підтримують обробку в ланцюгах за допомогою сповіщень різного роду, таких як: поява повідомлення у черзі обробки, HTTP виклик, мережа сповіщень (Event Grid) чи за допомогою таймерів. Проте такі сповіщення обов’язково мають підпорядковуватися правилам на вхід та вихід (inlet/outlet), що повинні бути попередньо описані в конфігураційному файлі. Також в рамках цього сервісу є можливість створювати функції зі збереженням стану та контексту, такі функції називаються Durable Functions - з їх допомогою можливо створювати сценарії пов’язування між собою різних stateless функцій в

конвеєри обробки, моніторинг, агрегації даних або навіть інтеракції із зовнішніми користувачами. Список підтримуваних мов дещо менший та включає: C#, JavaScript, Python, F# та PowerShell. До мінусів можна віднести:

1. Холодний старт функцій.

При кожному холодному запуску спостерігається значна затримка старту у розмірі 5с за медіаною, але може сягати і 15-17с в максимальних ситуаціях.

2. Обмеження на виконання окремих функцій:

- Паралельність. Кількість одночасно запущених примірників коду значно залежить від якості апаратного обладнання та коливається від 200 для найпростіших до 20 для найпотужніших серверів.
- Частота виклику. Створення нових примірників обробки подій може бути за допомогою HTTP викликів - в такому разі новий обробник буде створено не частіше ніж 1 на секунду, та за допомогою системи подій - в такому випадку новий обробник буде створюватися не частіше ніж 1 на 30 секунд. В залежності від апаратних можливостей - обробник може обробляти більше ніж одну подію чи виклик, але частота масштабування головним чином залежить від вартості підписки.
- Час виконання. Стандартне обмеження на час виконання функції - це 5хв, проте час виконання можна підвищити до 30 хв використовуючи on-demand варіант апаратного забезпечення чи використавши Kubernetes забезпечення
- Кількість одночасних з’єднань. В звичайній ситуації кількість одночасних з’єднань до кожного обробника обмежена 600 та має загальне обмеження в 1200 для всіх обробників разом. Всі більш дорогі on-demand рішення не мають жодних обмежень щодо кількості з’єднань.
- Параметри запиту. Кожен запит до функції обмежено в різних його параметрах: розмір запиту не повинен перевищувати 100 МБ, довжина query параметрів не повинна перевищувати 4096 символів та загальний розмір URL не може перевищувати 8192 символи.

- Апаратне забезпечення. Кожна функція підтримується апаратним забезпеченням, що обмежено певними параметрами: кожна функція не може задіювати більше ніж 100 віртуальних обчислювальних комплексів в звичайному тарифі, кількість оперативної пам’яті обмежена 1.5 ГБ та об’єм під’єднаного дискового простору обмежений 5 ТБ. Дані характеристики не є абсолютними значеннями та можуть коливатися в більшу і меншу сторону в залежності від обраного плану.
- Паралельність виконання. Кожна функція обмежена кількістю в 100 обробників - це число накладається на регіон виконання або групу ресурсів в залежності від використаного плану.
- Кількість доменних імен. Усі тарифні плани мають обмеження в 500 доменних імен для всіх функцій.

3. Неоднозначність обмежень та неочевидність документації. Значна кількість обмежень та правил тарифікації описана в різних місцях, тому фактична вартість наданих послуг може значно відрізнятися від орієнтовної. Обмеження на виконання не описано чітко та їх потрібно шукати окремо. Кількість тарифних планів хоч і надають гнучкість налаштування, проте їх опис може призвести до неоднозначного трактування.
4. Рентабельність використання. Відповідно до прикладу AWS Lambda - використовувати Azure Functions вигідно тільки до певної межі, оскільки вартість за використання нараховується за час роботи. Однак, даний провайдер надає можливість викликати так звані Durable Functions, що створені для постійної роботи та координації викликів інших функцій і тому правила їх тарифікації дещо відрізняються. Окремо варто зазначити, що Microsoft надає можливість розгортання FaaS рішень на основі IaaS архітектури за допомогою Kubernetes, що дає можливість безшовно переводити “гарячі” функції на власне забезпечення. [2]

1.3 Google Cloud

Останнім великим постачальником є Google Cloud Functions. Набір середовищ виконання є досить обмеженим та включає до себе наступні мови: Node.Js, Python, Go та Java. Для ініціювання ланцюжків обробки доступні: HTTP виклики, Pub/Sub повідомлення та набір інтеграцій з хмарними сервісами Google, такими як, наприклад, Cloud Spanner, Cloud Storage, Cloud Bigtable ітд. Створення та налаштування контекстів та доступів виконується виключно на стороні web інтерфейсу, завантаження виконуваного коду виконується за допомогою CLI.

Мінусами такого підходу можна назвати:

1. Холодний старт функцій

При довготривалій відсутності навантаження на функцію вона стає “холодною” і в такому випадку для обслуговування нових запитів може знадобитися від двократного збільшення часу запуску (в залежності від якості написаної функції) для того щоб почати обслуговувати запити.

2. Обмеження виконаних функцій

Функції Google Cloud мають низку обмежень:

- Час виконання функції обмежено часом 540с, цей ліміт не може бути підвищено.
- Пам'ять, доступна для функції не може перевищувати 2048 МБ.
- Розмір пакета коду. Розмір упакованого коду функції не повинен перевищувати 100 МБ, а розпакована версія не повинна перевищувати 250 МБ.
- Паралельність. Обмеження на паралельне виконання функцій застосовується тільки до окремої функції та становить 3000, тобто якщо кожна функція виконується 100с, то частоту викликів буде обмежено до 30 на секунду.

- Частота викликів. Окрім обмеження на паралельні виклики існує також обмеження на частоту викликів окремої функції - незалежно від часу виконання функції, кількість викликів функції буде обмежено до 1000 на секунду.
- Ширина каналу даних. Кожна функція також обмежена шириною каналу даних - одночасно функція може використати до 10 МБ, тобто якщо одна функція отримує повідомлення розміром 1 МБ та обробляє його 10с, то виклик буде обмежено до 1 на секунду, оскільки 11й виклик не може бути виконаний поки попередні виклики не завершаться.
- Частота використання каналу даних. Окрім обмеження на ширину одночасно виконаного каналу даних, існує й обмеження на частоту використаного каналу даних, тобто якщо функція обробляє повідомлення розміром 1 МБ за 100мс, то частоту виклику функції все одно буде обмежено до 10 за секунду.
- Розмір корисного навантаження. Запити за допомогою API Google Cloud на виклик функцій мають обмеження у вигляді 16 викликів кожні 100 секунд, в загальному випадку при виконанні HTTP виклику розмір запиту та відповіді не може перевищувати 10 МБ у розпакованому вигляді.

3. Не завжди рентабельно

Відповідно до прикладу AWS Lambda - використовувати Google Cloud Functions вигідно тільки до певної межі, оскільки вартість за використання нараховується за час роботи. [3]

1.4 OpenFaaS

Окремо хочеться зазначити некомерційну та відкриту розробку під назвою OpenFaaS, ціль якої - надання можливості розгортання FaaS рішень незалежно від існуючих пропрієтарних хмарних рішень. За допомогою цього хмарні функції можна розгортати на власних потужностях. Доставка коду в рамках OpenFaaS

виконується на основі Docker образів, що включають в себе всі необхідні залежності для запуску програми, кожен запуск та відключення такого контейнер відбувається на автоматичній основі. Автоматизація цього процесу відбувається за допомогою Kubernetes - системи оркерастрації контейнерів. Kubernetes - це платформа з відкритим вихідним кодом для управління контейнеризованими робочими навантаженнями та супутніми службами. Її основні характеристики - кросплатформеність, розширюваність, успішне використання декларативної конфігурації та автоматизації. Дана система підтримує розширення у вигляді так званих “операторів”, дані оператори є програмними засобами, що дозволяють керувати кластером на основі заданих користувацьких ресурсів.

Система OpenFaaS підтримує можливість налаштування різної конфігурації операторів, таких як, наприклад, Docker Swarm, Hashicorp Nomad, AWS Fargate/ECS та AWS Lambda. Найпростішим з цих рішень є стандартне, нативне рішення від OpenFaas, що називається faas-netes, який підтримує запуск функцій за допомогою мережевих запитів та реалізований на основі PLONK (Prometheus, Linux/Linkerd, OpenFaaS, NATS/Nginx,Kubernetes) стеку [4]. За допомогою даного стеку оператор отримує основні параметри життєвого циклу функцій, кожна функція експортує набір метрик, які записуються до Prometheus - на основі цих метрик faas-netes вирішує чи потрібно запустити більше екземплярів або зменшити їх кількість. До мінусів використання даного підходу можна віднести:

1. Налаштування інфраструктури. Самостійне налаштування FaaS архітектури призводить до того, що більшість інструментів адаптування та масштабування кластеру доводиться налаштовувати самому, наприклад, балансувальники трафіку, засоби оркестрації викликів та оператори масштабування.

Висновки

Усі розглянуті рішення надають можливість розгортання FaaS рішень у cloud-native середовищі, проте можливість оркестрації таких викликів функцій є досить обмеженою, то є потреба у певному інструменті, що був би здатний оркеструвати та проводити хореографію таких викликів. Таке рішення повинно бути абстраговане від конкретного провайдеру та надавати можливість об’єднувати виклики незалежно від використаного рішення cloud постачальника. Така система буде імітувати роботу реального мікросервісу з єдиною відзнакою, що кожна функціє не буде фізично поєднана з іншими. Найкращою абстракцією щодо поєднання таких викликів можна взяти з теорії компіляції, а саме - граф потоку керування.

Описавши систему викликів таким чином ми збережемо всі характеристики реальної програми та можливості її оптимізації, натомість отримавши можливість абстрагування від використаних апаратних можливостей кожного конкретного кроку, що надасть нам можливість гнучко налаштовувати ресурси всіх вузлів системи в залежності від їх навантаженості.

РОЗДІЛ 2 СПОСОБИ ПОДАННЯ ГРАФА КЕРУВАННЯ

2.1 Граф та способи його подання

Граф - це абстрактне уявлення множини об'єктів і зв'язків між ними. Графом називають пару (V, E) де V це множина вершин, а E множина пар, кожна з яких представляє собою зв'язок (ці пари називають ребрами) [5].

Граф може бути орієнтованим або неорієнтованим. В орієнтованому графі, зв'язки є спрямованими (тобто пари в E є впорядкованими, наприклад пари (a, b) і (b, a) це дві різні зв'язки). У свою чергу в неорієнтованому графі, зв'язки ненаправлені, і тому якщо існує зв'язок, (a, b) то значить, що існує зв'язок (b, a).

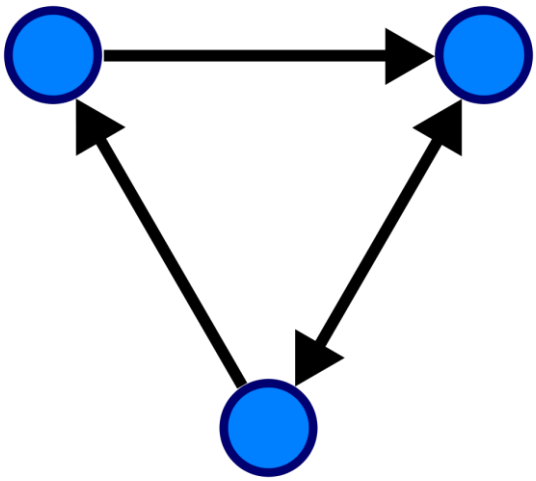


Рисунок 2.1 - Орієнтований граф

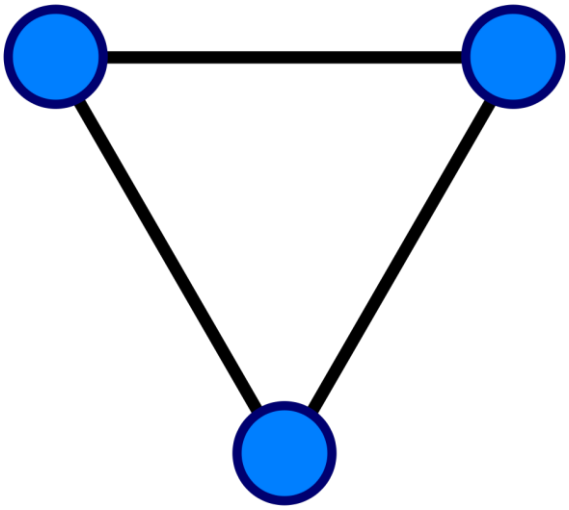


Рисунок 2.2 - Неорієнтований граф

2.2 Способи подання графу

Важливою задачею є вибір подання графу у пам’яті. У загальному випадку граф можна подати у двох видах:

- Графічно
- Таблично

Графічний спосіб подання графа насамперед полягає візуальному відображенні вузлів та зв’язків між ними. Існує велика кількість способів візуального подання графа, кожен з яких адресує різну проблему читання чи відображення графу, проте використання такого способу є недоцільним для зручного програмного чи алгоритмічного використання.

Табличний спосіб подання полягає у створенні певної структури даних, що однозначного відображає стан графу у пам’яті.

Таблично граф зручно подавати у вигляді наступних структур даних:

- Матриця суміжності
- Матриця інцидентності
- Списки суміжності
- Списки ребер

2.2.1 Подання графу за допомогою матриці суміжності

У даному випадку граф задається у вигляді матриці розмірністю $|V|x|V|$, де V - це кількість вершин графа. При такому поданні матриці A,

$A[i][j] =$ 1у випадку якщо з вершини i є ребро до вершини j

$A[i][j] =$ 0у випадку якщо зв’язку не існує

Даний спосіб підходить для орієнтованих і неорієнтованих графів. Цей спосіб є зручним для подання щільних графів, в яких кількість ребер $|E|$ приблизно дорівнює кількості вершин у квадраті $|V|^2$.

Для неорієнтованих графів матриця A є симетричною (тобто $A[i][j] == A[j][i]$, тому що якщо існує ребро між i і j, то воно є і ребром з i в j, і ребром з j в i). Завдяки цій властивості можна скоротити майже в два рази використання пам'яті, зберігаючи елементи тільки в верхній частині матриці, над головною діагоналлю). Зрозуміло що за допомогою даного способу представлення, можна швидко перевірити чи є ребро між вершинами i та j, просто подивившись в комірку $A[i][j]$. З іншого боку цей спосіб дуже громіздкий, так як вимагає $O(|V|^2)$ пам'яті для зберігання матриці.

	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	0	0
3	1	1	0	0	1
4	1	0	0	0	1
5	0	0	1	1	0

Рисунок 2.3 - Неорієнтований граф

	1	2	3	4	5
1	0	0	0	0	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	0	1	0	0
5	0	0	0	1	0

Рисунок 2.4 - Орієнтований граф

2.2.2 Матриця інцидентності

Матриця інцидентності - це також матрична форма подання графа, за якої організація даних у ній відбувається на основі інцидентності (прилягання), тобто зв'язки між ребрами (дугами) та вершинами. Стовпчики даної матриці відповідають ребрам, а рядки - вершинам. У даному випадку граф задається у вигляді матриці розмірністю $|V|x|E|$, де V - це кількість вершин графа та E- це кількість ребер графа. При такому поданні матриці А,

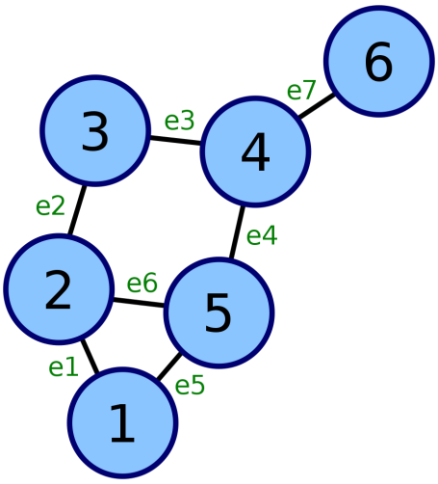
$A[i][j] = 1$ у випадку якщо вершина i поєднана з ребром j

$A[i][j] = 0$ у випадку якщо зв'язку між вершиною та ребром не існує

$A[i][j] = -1$ у випадку якщо граф орієнтований і є зв'язок між вершинами i та zза допомогою дуги j, то напрямок, обернений до напрямку дуги буде позначено таким чином

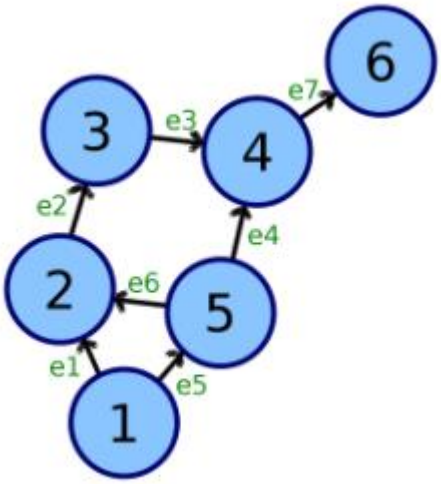
Даний спосіб підходить для орієнтованих і неорієнтованих графів. Цей спосіб є зручним для подання гіперграфів, оскільки гіперграф може містити у собі зв'язок між безліччю вершин та єдиним ребром.

Для звичайних графів завжди виконується правило, за якого сума кожного стовпчика не може бути більшою за 2, оскільки кожне ребро може поєднувати не більше ніж дві вершини.



	e1	e2	e3	e4	e5	e6	e7
1	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0
3	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0
5	0	0	0	1	1	1	0
6	0	0	0	0	0	0	1

Рисунок 2.5 - Матриця інцидентності неорієнтованого графа



	e1	e2	e3	e4	e5	e6	e7
1	-1	0	0	0	-1	0	0
2	1	-1	0	0	0	1	0
3	0	1	-1	0	0	0	0
4	0	0	1	1	0	0	-1
5	0	0	0	-1	1	-1	0
6	0	0	0	0	0	0	1

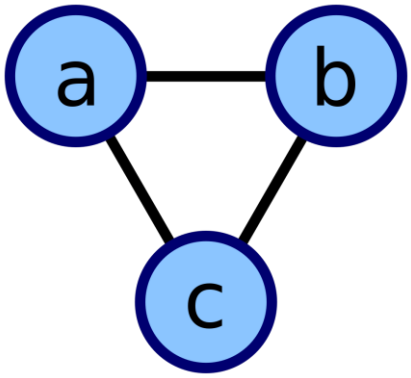
Рисунок 2.6 - Матриця інцидентності орієнтованого графа

2.2.3 Списки суміжності

Списки суміжності - це структура даних у вигляді таблиці, що містить списки вершин. Кожен ключ такої таблиці - це вершина, значення за цим ключем - список вершин, таких, щоб з вершини-ключа існувало ребро до кожної вершини зі списку-значення. Даний спосіб представлення більше підходить для розріджених графів,

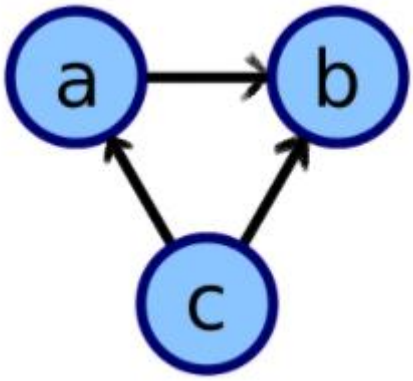
тобто графів у яких кількість ребер набагато менше ніж кількість вершин в квадраті ($|E| << |V|^2$).

Пам'ять необхідна для подання дорівнює O ($|E| + |V|$) що є найкращим показником ніж матриця суміжності для розріджених графів. Головним недоліком цього способу полягає в тому, що немає швидкого способу перевірити чи існує ребро (u, v). Даний підхід дозволяє отримати зручну реалізацію обходу графу в ширину чи глибину, оскільки дозволяє швидко отримати всіх сусідів потрібної вершини.



A	B,C
B	A,C
C	A,B

Рисунок 2.7 - Списки суміжності неорієнтованого графа

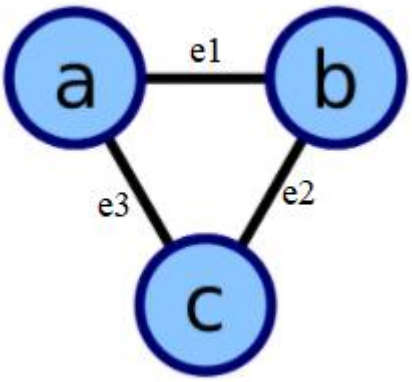


A	B
B	
C	A,B

Рисунок 2.8 - Списки суміжності орієнтованого графа

2.2.4 Списки інцидентності

Списки інцидентності - це структура даних, що містить дані про інцидентність двох вершин, за допомогою ребра і тому не залежить від направленості графа. Даний спосіб зберігання графа більше за все підходить для зовнішнього зберігання та зручної серіалізації оскільки потребує $O(|E|)$займаної пам’яті.



e1	A,B
e2	C,B
e3	A,C

Рисунок 1.9 - Списки інцидентності графа

2.3 Граф потоку керування

Граф потоку керування (Control Flow Graph) - це представлення, що використовує представлення у вигляді графу усіх шляхів, які можуть бути використані під час виконання програми [6].

У графі потоку управління кожен вузол представляє базовий блок, тобто пряму ділянку коду без будь-яких стрибків за адресою або відгалужень. Відгалуження можуть тільки починати блоки, а стрибки за адресою - завершати їх. Спрямовані ребра використовуються для представлення стрибків у потоці контролю. Більшість репрезентацій графу потоку керування мають два обов’язкові

блоки: блок входу контролю, за допомогою якого даний граф отримує контроль над виконанням та блок виходу контролю, за допомогою якого граф передає контроль над виконанням далі. Кожен блок графу потоку керування знаходиться у відношенні керування відносно інших блоків.

Таке відношення між блоками називається домінантним відношеннями [7]. Блок A домінує над блоком B, якщо кожен шлях від запису, що доходить до блоку B, повинен проходити через блок A. Блок входу домінує над усіма блоками. У зворотному напрямку блок A постдомінує блок B, якщо кожен шлях від B до виходу повинен проходити через блок A. Блок виходу контролю постдомінує всі блоки графу. Кажуть, що блок A безпосередньо домінує над блоком B, якщо A домінує над B, і немає такого проміжного блоку C, щоб A домінував над C, а C домінував над B. Іншими словами, A є останнім домінантом на всіх шляхах від входу до B. Кожен блок має унікальний безпосередній домінант. Так само існує поняття безпосереднього постдомінатора, аналогічне безпосередньому домінатору.

Граф контролю можна розкласти на дві ключові множини [8]:

1. Множина передніх граней, які разом дають направлений ациклічний граф, в рамках якого всі вузли можуть бути досягнені з вхідного вузла
2. Множина задніх граней (A, B), для яких вузол B домінуватиме над вузлом A

Саме ця характеристика є ключовою для вказаного підходу оркестрації виклику функцій, оскільки ми можемо значно спростити роботу з даним графом використавши направлений ациклічний граф в якості базового, що дасть нам змогу уникнути перевірок виходу, оскільки згідно з проблемою виходу програми [9] не існує загального алгоритму за допомогою якого було б можливо чи завершиться певна програма чи ні.

2.2 Виконання графу керування

Серед перелічених варіантів зберігання оптимальним вибором з точки зору зберігання даних під час виконання програми та зручності отримання необхідної інформації є списки суміжності, оскільки вони займають невелику кількість пам’яті та надають можливість оперативно отримувати дані щодо всіх сусідів вузла. Даний граф є частковим випадком графу потоку керування, оскільки являє собою повністю описаний набір всіх можливих переходів та варіантів стану процесу оркестрації функцій.

У випадку графу керування оркестрацією - кожен вузол системи є певною функцією, що потрібно виконати, де результат кожної такої функції - це наступний стан системи або помилка.

Одним з головних недоліків даного підходу до організації даних - це унеможливлення перевірки виходу виконання, оскільки запущена програма пройде граф від початку до кінця перераховуючи усі стани, тому важливо, щоб при ініціалізації такий граф був направлений та не мав жодного циклу та ці властивості було перевірено.

Описана вище система являє собою скінченний набір детермінованих станів та переходів між ними і тому оркестратор можна представити у вигляді скінченного автомата.

Кінцевий автомат - математична абстракція, модель дискретного пристрою, що має один вхід, один вихід і в кожен момент часу знаходиться в одному стані з множини можливих [10]. Є окремим випадком абстрактного дискретного автомата, число можливих внутрішніх станів якого скінченно. При роботі на вхід кінцевого автомату надходять послідовно вхідні сигнали, а на виході автомат формує вихідні сигнали. Зазвичай під вхідними сигналами беруть подачу на вхід автомата символів одного алфавіту, а на вихід кінцевого автомату в процесі роботи видає символи, в загальному випадку, іншого, можливо навіть не пересічного з вхідним, алфавіту. Крім кінцевих автоматів існують і нескінченні дискретні автомати - автомати з нескінченним числом внутрішніх станів, наприклад, машина Тьюринга. Перехід з

одного внутрішнього стану кінцевого автомату в інше може відбуватися не тільки від зовнішнього впливу, але й мимоволі.

Розрізняють детерміновані кінцеві автомати - автомати, в яких наступний стан однозначно визначається поточним станом і вихід залежить тільки від поточного стану і поточного входу, і недетерміновані кінцеві автомати, наступний стан у яких в загальному випадку не визначено і, відповідно, не визначений вихідний сигнал. Якщо перехід в наступні стану відбувається з деякими можливостями, то такий кінцевий автомат називають ймовірнісним кінцевим автоматом.

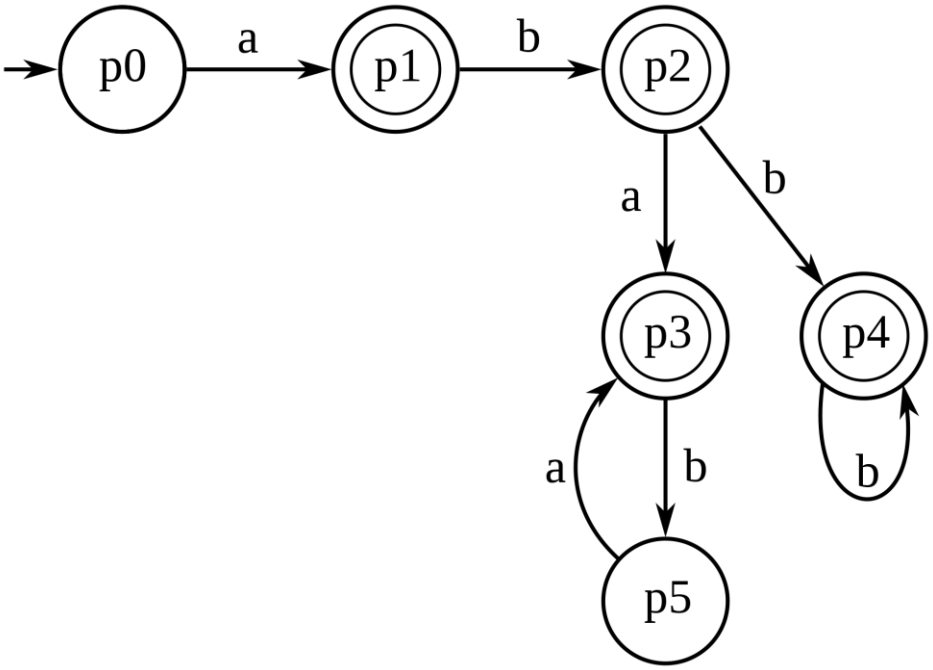


Рисунок 2.10 - Графічне подання скінченного детермінованого автомату

Представлену систему оркестрації можна віднести до класу скінченного детермінованого автомату, оскільки кількість можливих станів системи обмежена та кожен перехід між такими станами повністю визначений. Кожен такий перехід між станами є окремими, непоторюючимися символами алфавіту, оскільки, вони по своїй суті є викликами окремих функцій, що мають свої імена. Кожен виклик такої функції або вузол потрібно поєднати з вузлом обробки помилок, проте таке

поєднання не потрібно явно задавати в структурі графу - воно повинно забезпечуватися організацією автомату.

2.3 Забезпечення напрямленості та ациклічності графу

Важливим пунктом, що забезпечує стабільну роботу та виконання графу контролю є здатність однозначно та точно визначати параметри графу на кожному етапі його життя. Використання списків суміжності значно спрощують алгоритми оцінки параметрів графу, оскільки для того щоб, створити ненаправлений граф в такій конфігурації потрібно буде явно вказати двосторонній зв’язок двох вузлів графа, а для визначення циклів у напрямленому графі буде дуже зручно використати такий підхід завдяки низькій алгоритмічній складності доступу до всіх дочірніх вузлів, що становить $O(1)$.

Пошук циклів та визначення направленості графа можна виконати за допомогою єдиного проходу алгоритму пошуку в глибину (Deep First Search)[11]. Стратегія пошуку в глибину, полягає в тому, щоб йти «вглиб» графа, наскільки це можливо. Алгоритм пошуку описується рекурсивно: перебираємо всі вихідні з даної вершини ребра. Якщо ребро веде в вершину, яка не була розглянута раніше, то запускаємо алгоритм від цієї нерозглянутих вершини, а після повертаємося і продовжуємо перебирати ребра. Повернення відбувається в тому випадку, якщо в даній вершині не залишилося ребер, які ведуть в нерозглянутих вершину. Якщо після завершення алгоритму не всі вершини були розглянуті, то необхідно запустити алгоритм від однієї з нерозглянутих вершин.

Скориставшись важливою властивістю алгоритму пошуку в глибину, а саме тим, що результатом його роботи є обернений відсортований список вершин можна модифікувати роботу цього алгоритму таким чином, що даний алгоритм буде виконувати пошук графу на предмет існування циклів, а також буде проводити топологічне сортування заданого графу, що дасть нам можливість отримати всі

кореневі вузли графу. Модифікована версія даного алгоритму у вигляді псевдокоду наступна:

```
L ← порожня множина, який міститиме кореневі вузли
поки існують вузли без чорної позначки
    вибрати не позначений вузол n
    visit (n)

function visit (вузол n)
    якщо n має чорну позначку, тоді
        вийти
    якщо n має сіру позначку, тоді
        вийти (цикл не є ациклічним)

    позначити n сірим кольором

    для кожного вузла m з ребром від n до m виконати
        visit (м)

    перефарбувати n з сірого кольору на чорний

    якщо вузол не має батьківських, тоді
        додати кореневий вузол до L
```

Рисунок 2.11 - Псевдокод роботи модифікованого алгоритму пошуку в глибину

Робота даного алгоритму може повернути один з наступних результатів:

- помилка (у випадку якщо граф матиме цикли)
- множина коренів графу (у випадку якщо граф ациклічний)

Невиключеною можливістю є створення графу з декількома кореневими вузлами (наприклад якщо вказаний граф є підграфом загального графу контролю або може бути наслідком помилки введення). У такому випадку зручним до використання друга властивість алгоритму пошуку в глибину, а саме топологічне сортування - в нашому випадку, з використанням модифікованого алгоритму ми отримаємо всі корені графу та спробуємо знайти найглибший корінь графу за

допомогою простого обходу графа від вказаного вузла до останнього найглибшого постдомінатора цього кореня. Після проходження всіх коренів буде обрано найглибший, а отриману інформацію щодо графа буде записано до пам’яті програми.

Висновки

У даному розділі було розглянуто способи зберігання графа у пам’яті та підходи, що забезпечують стабільність виконання задач описаних графом під час роботи системи. Зберігання графу в пам’яті у вигляді списків суміжності надає можливість компактного зберігання без втрати гнучкості при запитах, щодо відношень даного вузла відносно інших в графі.

Вибір побудови графу на основі ацикличного напрямленого графу надає нам можливість забезпечити надійну роботи всієї системи без необхідності покрокової перевірки на виконання циклів. Також це позбавляє нас необхідності визначати нескінченні цикли, оскільки згідно до проблеми зупинки - не існує загального алгоритму, що дозволив нам би визначити наявність умови виходу з роботи алгоритму.

Ациклічність графу можна перевірити на основі алгоритму пошуку в глибину, що в поєднанні з обраною структурою представлення графу надає нам можливість визначити дотримання цієї умови. За допомогою топологічного сортування алгоритму пошуку в глибину також є можливість визначити найглибший вузол графа, що дозволить однозначно визначати умови запуску кожного окремого графа виконання.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ БІБЛІОТЕКИ ОРКЕСТРАЦІЇ

3.1 Вибір мови програмування

Для створення системи оркестрації було обрано мову Go. Мова Go є відносно новою, мультипарадигмовою мовою програмування загального призначення, що широко використовується у розробці програмного забезпечення. Вибір даної мови програмування обумовлено наступними причинами:

- Значна підтримка мови інтернет-гігантом Google
- Поширене використання та створення інструментів налаштування хмарних кластерів на основі Go
- Висока швидкість розробки
- Наявність низькорівневих абстракцій
- Невисоке споживання пам’яті завдяки малій кількості runtime залежностей

Мова програмування Go підходить для швидкого написання проектів, що мають змогу користуватися платформозалежними оптимізаціями та виконуватися швидко і ефективно у проектах різного розміру. Мультипарадигмовість Go дають можливість використовувати всі плюси як декларативного так і імперативного програмування, не даючи заганяти себе в рамки, що створюють чіткі слідування одній окремій парадигмі. Основні семантичні структури було стабілізовано, проте подальша розробка мови відбувається активно, що дозволяє отримувати важливі оптимізації мови та нові структури та семантичні конструкції, що значно покращують зручність користування мовою.

Значна підтримка серед спільноти та можливість оминати публікацію версій пакетів та бібліотек на централізоване кодове сховище, дозволяє отримати зручну децентралізовану систему доставки залежностей, що повністю незалежна від організації, що її створила.

Навідміну від існуючих мов програмування - нові мови будуються з урахуванням можливості паралелізації, тобто можливість розподіляти роботу між фізичними та логічними ядрами системи. Go дозволяє отримувати ефективну роботу програми за допомогою програмних абстракцій навколо мультитредингу, що називається green threads або в рамках мови - горутини.

3.1.1 Горутини та їх переваги над звичайними потоками

Горутини - це функції або методи, які працюють одночасно з іншими функціями або методами. Горутини можна сприймати як легкі потоки. Вартість створення окремої горутини незначна, якщо порівнювати з потоком. Отже, для програм Go нормальним є використанням та викоанням тисяч програм горутин.

Горутини надзвичайно дешеві в порівнянні з потоками. Вони мають розмір стека всього на кілька кілобайт, і стек може зростати і стискатися відповідно до потреб програми, тоді як у випадку потоків розмір стека повинен бути вказаний і фіксований. Горутини мультиплексуються до меншої кількості фізичних потоків ОС. У програмі з тисячами горутин може бути лише один фізичний потік виконання. Якщо яка-небудь горутина у цьому блоці потоці каже, що чекає введення користувачем, то в такому випадку створюється інший потік ОС, а решта горутин переміщуються до нового потоку ОС. Про все це дбає середовище виконання коду, і ми, як програмісти, маємо можливість повної абстракції від деталей роботи паралелізму та натомість отримуємо чисте API роботи з конкурентністю

Горутини спілкуються за допомогою каналів. Створені канали запобігають появі гонок при доступі до спільної пам'яті за використанням горутин. Канали можна сприймати як трубу, за допомогою якої горутини спілкуються.

При створенні нової горутини, виклик програми негайно повертається. На відміну від функцій, елемент керування горутинами не чекає закінчення виконання

програми. Елемент керування негайно повертається до наступного рядка коду після виклику горутини, а будь-які повернені значення з горутини ігноруються.

У викликах за допомогою горутин виконується наслідковість, тобто головна точка входу main неявно створює головну горутину, що відповідає цій функції, кожне створення нової горутини породжує дочірню горутину, що підпорядковується головній або тій, в якій нову горутину було створено. Тому у випадку завершення виконання батьківської горутини - всі породжені нею горутини будуть автоматично завершені.

3.1.2 Жадібний планувальник задач

Задачею планувальника Go полягає в розподілі керованих програм під кількома робочими потоками ОС, які працюють на одному або декількох процесорах. У багатопотокових обчисленнях у плануванні з’явилися дві парадигми: спільне виконання роботи та жадібне виконання роботи.

- Спільне виконання роботи: при створенні процесором нових потоків, такий планувальник намагається перенести деякі з них на інші ядра процесору, сподіваючись на те, що вони будуть виконані іншими ядрами, що простоюють чи недостатньо завантажені.
- Жадібне виконання роботи: кожне ядро процесору, що було недостатньо навантажено буде шукати можливість “вкрасти” роботу у вигляді потоків у інших процесорів.

При використанні жадібного підходу реальна міграція потоків між процесорами буде виконуватися рідше, аніж у іншому випадку, оскільки якщо кожен процесор матиме роботу, то ніякого перетягування даних між ядрами не відбуватиметься. Можливість міграції чи переносу даних можлива лише у випадку простою певного ядра.

Починаючи з версії 1.1 Go базує своє планування задач на основі жадібного виконання роботи [14]. Go має планувальник M:N, який також може використовувати кілька процесорів. У будь-який час M кількість програм горутин повинні бути заплановані на N потоків ОС, які працюють на щонайменше GOMAXPROCS кількості процесорів.

Планувальник Go використовує таку термінологію для програм, потоків і процесорів:

- G: горутина
- M: Потік ОС
- P: ядро процесора (обробник)

Існує дві контекстозалежні черги задач для P - локальна та глобальна. Кожен M повинен бути призначений до P. P. Кожен P може не мати M, якщо його було заблоковано або він обробляє системний виклик. У будь-який момент існує не більше GOMAXPROCS кількість P. У будь-який час на кожному окремому P може працювати лише одна M. В свою чергу, за необхідністю, планувальник може створити більшу кількість M.

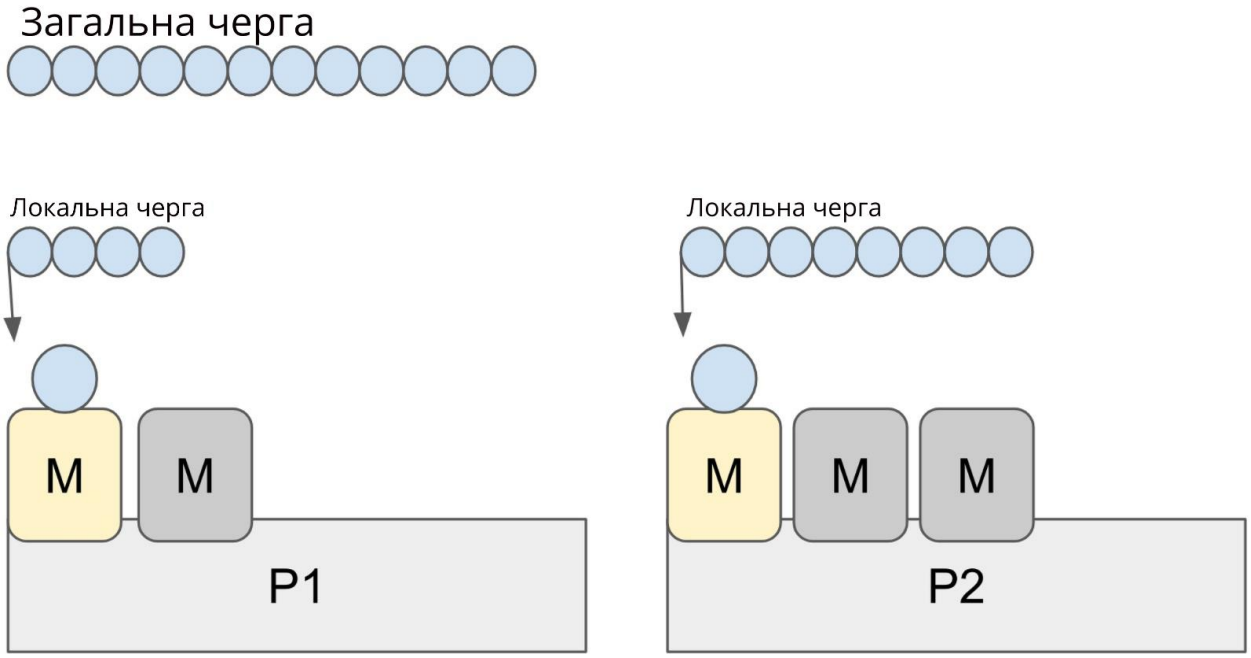


Рисунок 3.1 - Модель жадібного планувальника

Кожен раунд планування - це просто пошук виконуваної програми та її виконання. На кожному раунді планування пошук відбувається в такому порядку:

1. Один раз на 61 раз виконання перевіряємо глобальну чергу на наявність G
2. У випадку, якщо глобальних задач немає, перевіряємо локальну чергу
3. Якщо задач немає, то намагаємося перемістити роботу з інших обробників
4. Якщо переміщення неможливо - знову перевіряємо глобальну чергу
5. Якщо всі кроки були неуспішними - обслуговуємо мережеві запити

Як тільки запущений G знайдений, він виконується, доки його не заблокують. Глобальна черга має перевагу над локальною чергою, але перевірка глобальної черги час від часу є критично важливою, щоб уникнути того, що M виконує лише планування з локальної черги, поки не залишиться жодних горутин у локальній черзі.

Викрадання роботи

Коли створюється новий G або існуючий G стає придатним для роботи, він переходить до списку виконуваних горутин поточного P. Коли P закінчує виконувати G, він намагається прибрати виконаний G із власного списку виконуваних програм. Якщо список тепер порожній, P вибирає випадковий інший процесор (P) і намагається викрасти половину керованих програм із своєї черги.

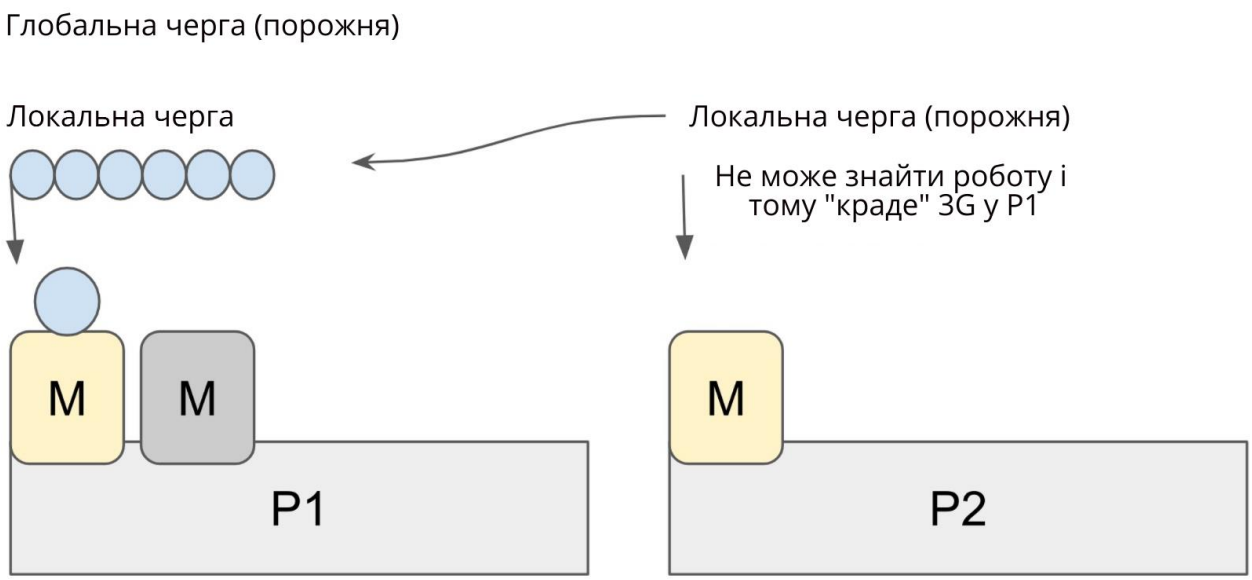


Рисунок 3.2 - Порядок “викрадення” роботи між ядрами

На рисунку 3.2 P2 не може знайти жодних виконуваних програм. Тому він випадковим чином вибирає інший процесор (P1) і викрадає три програми в свою локальну чергу. P2 зможе виконати ці задачі окремо, і робота планувальника буде більш справедливо розподілена між кількома процесорами.

Планувальник завжди хоче розподілити стільки ж, скільки виконуваних програмних програм, для використання процесорів, але в той же час нам потрібно оминати надмірну роботу для економії центрального процесора та енергії. Всупереч цьому, планувальник також повинен мати можливість масштабувати програми з високою продуктивністю та інтенсивними процесорними задачами. Постійне припинення - це водночас і дорога, і проблематична дія для високопродуктивних програм, якщо продуктивність є критичною.

Потоки ОС не повинні часто передавати виконувані програми між собою, оскільки це призводить до збільшення затримки відповіді. Крім того, за наявності системних викликів, потоки ОС потрібно постійно блокувати та розблоковувати. Це дорого і додає багато накладних витрат.

Для того, щоб мінімізувати передачу, планувальник Go реалізує постійно працюючі “обертальні” потоки (spinning threads). Такі потоки споживають

мінімальну додаткову потужність процесора, але вони мінімізують припинення потоків ОС. Потік буде обертатися, якщо:

- M з призначенням P шукає на горутину, яку можна запустити
- M без визначеного P шукає на доступні P
- Планувальник також запускає додатковий потік і обертає його, у випадку якщо є невикористана P, та немає інших обертальних потоків

У будь-який час існує щонайбільше GOMAXPROCS M, що обертаються. Коли обертальний потік знаходить роботу, вона виходить із стану обертання та стає звичайним потоком. Неактивні потоки з призначеним P не блокуються, якщо є бездіяльні M без визначеного P. Коли створюються нові горутини або блокується M, планувальник гарантує, що є хоча б один обертальний M. Такий підхід гарантує можливість запустити горутину без простою та допомагає уникнути непотрібного надлишкового блокування/розблокування M.

Даний підхід планування дуже добре підходить до поставленої задачі, оскільки виконання окремих викликів відбувається на основі паралелізації, кожен окремий запуск ініціюватиме старт виконання автомату, оскільки кожен перехід - це окремий виклик функції, тому важливо, щоб ресурси виділені на кожен виклик не аллокувались на одному й тому самому ядрі процессора, таким чином відбуватиметься рівномірне розподілення навантаження на процесорі.

3.2 Реалізація конкуретного виконання

У розробленій програмній бібліотеці було реалізовано prefork конкурентної обробки черги. Черга обробки задач наслідує спосіб роботи планувальника задач мови програмування Go - локальна черга та глобальна. Prefork локальної черги налаштовується на двох рівнях додатку - на рівні обробки та на рівні наповнення локального буферу.

3.2.1 Локальний рівень

Рівень обробки, згідно з назвою, створює чергу обробки, об’єм якої точно дорівнює кількості одночасних обробників. Внутрішні черги обробки поєднані єдиною шиною, що використовується для поєднання та передачі подій між чергами.

Другий рівень локальної обробки слугує засобом буферизації, буферизація є важливим способом оптимізації мережевих накладок, оскільки ми попередньо зберігаємо набір задач, які можна виконати навіть якщо ми тимчасово втратимо доступ до зовнішньої черги. У випадку падіння сервісу - зовнішній наглядач підтримує можливість повернення задач назад до загальної черги, тому можливість втрати практично повністю нівелюється організацією наглядача. Таким чином можна створити і обернену ситуацію - створивши дефіцит робочого навантаження ми можемо створити буферну кількість обробників для того щоб зменшити навантаження на систему та всі підсистеми, що знаходяться нижче у потоці виконання, створення дефіциту таким чином не призведе до перевантаження окремих обробників, оскільки формат читання з каналу виконується розподілено між всіма підписниками.

Шина, що поєднує два рівні локальної обробки використовує головну концепцію мови програмування Go - канали. Канали реалізують однин з головних принципів конкурентного виконання програм в Go - не дозволяти горутинам спілкуватися шляхом спільного використання пам’яті, горутини повинні ділитися даними, спілкуючись через канали.

Спілкування шляхом спільного використання пам’яті та спільне використання пам’яті шляхом спілкування - це два способи програмування при одночасному програмуванні. Коли горутини спілкуються шляхом спільного

використання пам'яті, ми використовуємо традиційні методи синхронізації паралельних операцій, такі як блокування за допомогою м’ютексів, для захисту спільної пам'яті від гонок даних. Проте, ми можемо використовувати канали для реалізації спільного використання даних на основі спілкування.

Go забезпечує унікальну техніку синхронізації паралельності - канал. Канали дають можливість горутинам ділитися пам’яттю, обмінюючись подіями. Ми можемо розглядати канал як внутрішню програмну чергу FIFO. Одні програми надсилають значення в чергу (канал), а інші програми отримують значення з черги (каналу). Разом з передачею даних через канали, володіння деякими даними також може передаватися між програмами. Коли горутина передає значення за допомогою каналу, ми можемо розглядати це, як програма передачу володіння даними. Коли інша горутина отримує значення з каналу, то ми можемо розглядати цю подію, як отримання володіння даними. Проте, передача та отримання інформації таким чином не завжди означає передачу володіння.

Дані, володіння якими було передано часто можуть бути просто посиланнями на певні ділянки пам’яті. Тому варто звернути увагу передача володіння над даними в цьому контексті визначається як логічна передача, а не на рівні рантайму програми. Таким чином логічна передача прав на дані не позбавляє розробника від ситуацій гонок даних, проте за допомогою каналів ми можемо отримати значно чистіший код з синтаксичної точки зору. Хоч мова програмування Go і підтримує всі традиційні структури синхронізації даних, тільки канали отримують положення першокласного громадянина.

Таким чином за допомогою використаного проміжного каналу ми отримуємо структуру синхронізації та спілкування кількох незалежних контекстів: контексту виконання та контексту буферизації навантаження, без необхідності використання традиційних підходів, що дозволяє нам не використовувати ніяких примітивів синхронізації та не втрачати час виконання на синхронізацію завдяки використання FIFO черги та жадібному виконанню роботи - за допомогою цього обробники будуть гарантовано рівномірно завантажені роботою.

3.2.2 Глобальний рівень

На глобальному рівні - черга обробки координується на основі Redis, усі отримані задачі на виконання передаються на зберігання до FIFO черги на її основі. Дана черга побудована на основі вбудованого типу Redis - списки. В загальному використанні списки - це структура даних дек (deque), тобто має можливість двостороннього додавання та віднімання елементів: з початку та з кінця. В рамках розробленої системи читання та додавання до списку відбувається за допомогою вбудованих функцій LPUSH та RPOP, тобто додавання зліва та читання справа [12].

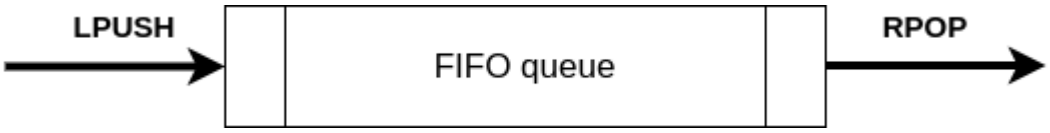


Рисунок 3.3 - Діаграма роботи з чергою задач

Міжсервісне координування роботи за допомогою цієї бази даних забезпечується набором таких черг, що в поєднанні забезпечують надійну взаємодію глобальної черги з кожним споживачем черги. Така система черг надає можливість вести облік всіх задач, взятих до роботи та їх параметри. Всі помилки під час обробки повідомлення призведуть до повтору повідомлення відповідно до його налаштувань, задача повернеться до загальної черги виконання де може бути взята будь-яким обробником. З’єднання з кожним окремим обробником перевіряється за допомогою heartbeat повідомлень, що перевіряють їх життєздатність. У разі відмови будь-якого споживача за критерієм heartbeat - усі повідомлення, що не було позначено як виконані чи помилкові будуть повернені до загальної черги відповідно до налаштувань даної черги, що дозволить не втрачати корисне навантаження у випадку втрати зв’язку з обробником.

Описана система виконує пасивне розподілення задач та по суті є лише середовищем, що зберігає дані та надає можливість швидкого атомарного додавання та вилучення, тому за своєю суттю є модифікованою версією монітору, що синхронізує конкурентні операції доступу до черги, але без використання м’ютексу на рівні коду, оскільки середовище забезпечує атомарність виконаних операцій.

Таким чином реалізоване середовище зберігання глобальної черги виконання дає можливість синхронізувати роботу будь-якої кількості обробників та розподіляти задачі між ними без значних витрат на таку систему та зменшуючи втрати на синхронізацію до мінімуму.

3.3 Способи збереження та виконання задач

Описані раніше підходи щодо обробки задач були побудовані в основному на зберіганні задач у пам’яті обробника або зовнішнього наглядача та спілкування між ними мало виключно технічний характер - такі дані не призначені для читання користувачем оскільки їх формати в першу чергу ставлять швидкість передачі та серіалізації/десеріалізації головним фактором, а також через те, що такі дані не є персистентними в загальному розумінні, оскільки вони існуються тільки в пам’яті обробників та координатора роботи. А для відновлення статусів роботи під час аварійного вимкнення координатора використовується лог задач, що при повторному запуску відновлює стан координатора просто застосовуючи всі зміни по порядку. В той самий час системою користуються не тільки обробники, але й самі користувачі - для отримання інформації щодо виконання роботи загалом чи детального огляду статусу виконання окремої задачі.

Тому важливо забезпечити зберігання даних до більш надійного сховища, що не потрібно часто опитувати та яке здатне виконувати роль “журналу” роботи, до якого будуть записуватися всі задачі та оновлюватися відповідно до статусу

виконання з можливістю довгострокового зберігання цих даних. Таким чином можна проводити як і активний аналіз виконаної роботи так і ретроспективний, переглядаючи статуси минулих задач та дані про їх виконання будуючи різні постагрегації.

3.3.1 Спосіб збереження даних

Збереження даних було побудовано на основі абстрактного інтерфейсу. В мові програмування Go інтерфейс - це набір сигнатур методів, які може імплементувати структура даних. Основною задачею інтерфейсу є надання лише сигнатур методів, що складаються з імені методу, вхідних аргументів та типів повернення. Тип (наприклад, struct) залежить від декларування методів та їх реалізації. Даний підхід дуже схожий з implements в різних мовах програмування, що слідують принципам ООП. Але в Go структура не повинна явно вказувати який тип інтерфейсу вона імплементує - це дозволяє розділити абстрактні інтерфейси та конкретні імплементації без необхідності вказувати в якості залежностей постачальника інтерфейсу, а структурі достатньо просто імплементувати правильну сигнатуру методів, тобто забезпечити контракт. Таким чином існування структур забезпечує принцип качиної типізації, що можна описати фразою “якщо воно ходить як качка і квакає як качка, то значить це і є качка” - дана фраза означає те, що можна передавати типи даних, що мають тільки загальну сигнатуру функцій, незалежно від їх імплементації.

В рамках розробленої бібліотеки було розроблено такий інтерфейс, що потребує базовий набір необхідних функцій, на основі яких можлива реалізація будь якої задачі по роботі зі сховищем даних. На основі роботи цього інтерфейсу будується робота всіх інших важливих компонентів, таких як, створення нового запису в журналі, оновлення кроку виконання роботи, сповіщення про початок та кінець роботи. Приклад описаного інтерфейсу зображено на рисунку 3.1.

```
type Storage interface {
    Create(obj ObjectDTO) (*Object, error)
    FindById(id string) (*Object, error)
    UpdateById(id string, update KV, operation OperationMap) error
}
```

Рисунок 3.1 - Інтерфейс функцій-примітивів абстрактного сховища

Гарантією коректної взаємодії бібліотеки з конкретною імплементацією сховища, окрім необхідного набору функцій є ще й набір правил виклику цих функцій, тобто створення контракту виклику. Контракт виклику описує правила, яким буде підпорядковуватися виклик функцій. В даній розробці контракт пов'язує тільки базові операції і тому кількість правил досить обмежена. Гарантом виконання контракту є система типів мови програмування Go завдяки її сильній типізації, що не дасть викликати та передати до функції в неправильному вигляді. Це дозволить не розробляти окремо систему, що повинна буде перевіряти передані параметри та значно економить час розробки бібліотеки загалом, а також дозволить зберегти однорідність розробленої системи, оскільки всі її частини будуть написані за допомогою однієї й тієї ж самої мови програмування.

```
type KV map[string]interface{}

type OperationKey string

var (
    AddOperation OperationKey = "add"
)

type OperationValue map[string]interface{}
type OperationMap map[OperationKey]OperationValue
```

Рисунок 3.2 - Набір правил контракту виконання

За допомогою розробленого набору правил фактична реалізація системи зберігання даних може бути реалізована незалежно від бібліотеки та постачатися і підключатися до системи окремо від коду системи оркестрації.

Додатковим шаром абстракції є використання так званих DTO (Data Transfer Object), що по суті є мінімально необхідною кількістю інформації необхідної для передачі - така структура даних повністю позбавлена будь-якої бізнес логіки та несе в собі лише інструменти серіалізації та десеріалізації такої структури даних. За допомогою такої абстракції ми можемо відокремити джерело надходження таких даних, в даній ситуації ініціатора обробки, від системи збереження цих даних та таким чином дасть можливість не залежати одному від іншого. Структуру об’єкту DTO наведено на рисунку 3.3

```
type ObjectDTO struct {
    CommandGraph string                 `bson:"commandGraph" json:"commandGraph"`
    Status       Status                 `bson:"status" json:"status"`
    Params       map[string]interface{} `bson:"params" json:"params"`
}
```

Рисунок 3.3 - Структура об’єкту DTO

За допомогою системи типів також було описано модель запису у журналі, що включає до себе всі дані та поля, що реалізують бізнес логіку додатку. Такі поля описують унікальний ідентифікатор задачі, її поточний статус та інші метадані. Структура такого об’єкта наведена на рисунку 3.4

```
type Object struct {
    ID           interface{}            `bson:"_id" json:"id"`
    CreatedAt    time.Time              `bson:"createdAt" json:"createdAt"`
    UpdatedAt    time.Time              `bson:"updatedAt" json:"updatedAt"`
    CompletedAt  time.Time              `bson:"completedAt" json:"completedAt"`
    Error        string                 `bson:"error" json:"error"`
    CurrentStep  string                 `bson:"currentStep" json:"currentStep"`
    CommandGraph string                 `bson:"commandGraph" json:"commandGraph"`
    Status       Status                 `bson:"status" json:"status"`
    Params       map[string]interface{} `bson:"params" json:"params"`
}
```

Рисунок 3.4 - Повна структура об’єкту виконуваної задачі

Створена система типів повністю заснована на базових типах мови програмування Go і тому не потребує додаткових налаштувань.

3.3.2 Ініціація обробки

У стандартній конфігурації бібліотеки ініціація обробки повідомлення доступна лише за допомогою мережевого HTTP запиту. Такий запит повинен включати в себе назву графа керування та параметри його запуску, що будуть передані в початкову функцію. Приклад такого запиту представлено на рисунку 3.5.

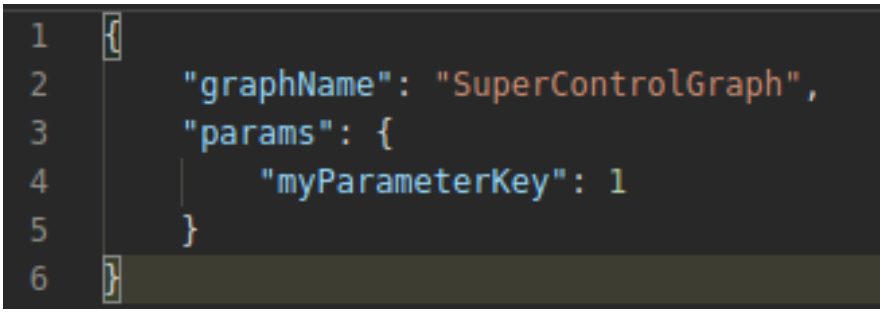


Рисунок 3.5 - Приклад тіла запиту на обробку

Кожен такий запит на обробку ініціює запит на створення запису у журналі виконуваних задач. З полів, наданих в запиті, створюється DTO, який потім використовується для запису до персистентного сховища. В стандартному налаштуванні персистентне сховище створене на основі MongoDB.

MongoDB - документоорієнтованих система управління базами даних, яка не потребує опису схеми таблиць [13]. Вважається одним з класичних прикладів NoSQL-систем, використовує JSON-подібні документи і схему бази даних. Застосовується в веб-розробці, зокрема, в рамках JavaScript-орієнтованого стека MEAN. Система підтримує ad-hoc-запити: вони можуть повертати конкретні поля документів і призначені для користувача JavaScript-функції. Підтримується пошук за регулярними виразами. Також можна налаштувати запит на повернення випадкового набору результатів. Є підтримка індексів. Система може працювати з набором реплік, тобто містити дві або більше копії даних на різних вузлах. Кожен екземпляр набору реплік може в будь-який момент виступати в ролі основної або допоміжної репліки. Всі операції запису і читання за замовчуванням здійснюються з основною реплікою. Допоміжні репліки підтримують в актуальному стані копії

даних. У разі, коли основна репліка дає збій, набір реплік проводить вибір, яка з реплік повинна стати основною. Другорядні репліки можуть додатково бути джерелом для операцій читання. Система масштабується горизонтально, використовуючи техніку сегментування (англ. Sharding) об'єктів баз даних - розподіл їх частин з різних вузлів кластера. Адміністратор вибирає ключ сегментування, який визначає, за яким критерієм дані будуть рознесені по вузлах (в залежності від значень хеша ключа сегментування). Завдяки тому, що кожен вузол кластера може приймати запити, забезпечується балансування навантаження. Система може бути використана в якості файлового сховища з балансуванням навантаження і репликацией даних (функція Grid File System; поставляється разом з драйверами MongoDB). Надаються програмні засоби для роботи з файлами і їх вмістом. GridFS використовується в плагінах для Nginx і lighttpd. GridFS розділяє файл на частини і зберігає кожну частину як окремий документ. Може працювати відповідно до парадигми MapReduce. У фреймворку для агрегації є аналог SQL-вирази GROUP BY. Оператори агрегації можуть бути пов'язані в конвеєр подібно до UNIX-Конвейер. Фреймворк так само має оператор $lookup для зв'язки документів при вивантаженні і статистичні операції такі як середньоквадратичне відхилення. Підтримується JavaScript в запитах, функціях агрегації (наприклад, в MapReduce). Підтримує колекції з фіксованим розміром. Такі колекції зберігають порядок вставки і після досягнення заданого розміру поводяться як кільцевої буфер. Починаючи з версії 4.0 була додана підтримка транзакцій, які відповідають вимогам ACID.

Після запису всієї інформації щодо виконаної задачі, унікальний ідентифікатор задачі поміщається у чергу виконання, з якої задачі будуть розподілятися між обробниками. Після чого кожен обробник при отриманні роботи виконує запит на сховище для перевірки валідності задачі та починає обробку, оновлюючи статус виконання на кожному кроці графу виконання.

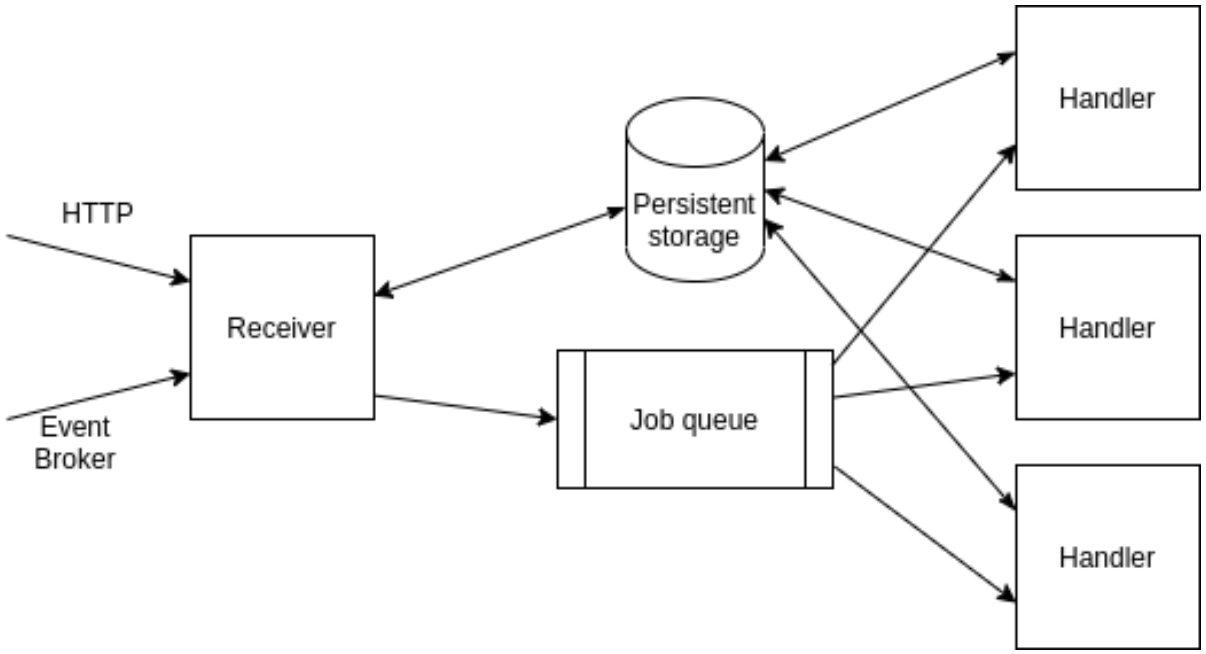


Рисунок 3.6 - Схематичне зображення ініціації та виконання заданого графу

3.3.3 Можливість розширення

Описані раніше стандартні підходи щодо ініціації та збереження даних не є єдиним способом роботи з системою, оскільки реалізований архітектурний підхід передбачає можливість розширення.

Систему ініціалізації та початку виконання можна розширити додавши читання з певних черг брокерів сповіщень, наприклад, RabbitMQ чи AWS SQS оскільки систему ініціювання було логічно відокремлено у окрему абстрактну підсистему, що поєднана з виконанням лише за допомогою глобальної черги планування задач.

Збереження до персистентного сховища можна розширити та доповнити додавши адаптери та драйвери інших систем зберігання даних, також поєднуючи їх на різний лад, що дозволить використовувати дублювання даних до різних сховищ або реалізовувати модель конкурентного читання з декількох місць - така можливість базується на основі надання абстрактного інтерфейсу, що може бути

реалізована за допомогою надання драйверу, що реалізує інтерфейс та контракт роботи.

Висновки

В даному розділі було розглянуто деталі реалізації програмної бібліотеки оркестрації. Система розподілена на набір окремих, незалежних блоків, що працюють незалежно один від одного.

Першим кроком на шляху виконання корисного навантаження є модуль отримання сповіщень за допомогою якого оркестратор отримує нові запити на виконання роботи, зберігає дані про правильні задачі до персистентного сховища та передає унікальний ідентифікатор до черги планування задач.

Об’єднуючим блоком, що передає та розподіляє роботу між обробниками є глобальна черга на основі Redis, яку читає кожен окремий обробник. Робота з цією чергою виконується на основі атомарних операцій і тому розподіл задач є безпечним в умовах конкурентного виконання

Останнім кроком є самі обробники, що проактивно забирають задачі з черги, читають статус задачі зі сховища за їх унікальним ідентифікатором та передають задані користувачем параметри до графу виконання роботи.

Окремо варто зазначити широку можливість розширення даної системи за допомогою системи інтерфейсів, що не потребують конкретної імплементації та базуються лише на сигнатурах функцій та контракті виклику даних функцій, що дозволить розширити роботу системи на різні способи запуску обробки повідомлень та використати різні системи зберігання даних. Таким чином ми отримуємо систему оркестрації з можливістю гнучкого налаштування окремих її модулів.

РОЗДІЛ 4 ТЕСТУВАННЯ РОЗРОБЛЕНОЇ БІБЛІОТЕКИ

Останнім кроком у розробці системи оркестрації є тестування системи та її порівняння з подібними існуючими аналогами. Також даний етап роботи дозволить пересвідчитися у коректній роботі всіх підсистем та зрозуміти якість наданих абстрактних інтерфейсів у використанні кінцевої задачі.

Методологія тестування полягатиме в комплексній оцінці роботи системи, тобто перевірці підлягатиме вся система загалом, а не окремі її частини. За допомогою комплексного тестування ми зможемо перевірити правильність взаємодії всіх складових системи, а також правильність логіки їх роботи у контексті оркестрації розподілених обчислень.

4.1 Вибір критеріїв оцінки якості розробленої системи

Оцінка якості роботи системи оркестрації - це комплексних набір мір та заходів, головна мета яких визначити в першу чергу саме якість написаного ПЗ, що надає можливість створювати повноцінну оркестрацію. В даному випадку головним критерієм оцінки є швидкодія саме системи виконання перехідних функцій конвеєра. Другим головним критерієм оцінки буде власне орієнтовна вартість на створення, розгортання та підтримання цільової архітектури.

Критерій швидкодії можна оцінити як здатність системи вчасно реагувати на один окремий запит та час за який система встигає перемикати етапи заданого процесу конвеєрної обробки задач. Таким чином ми можемо значною мірою абстрагуватися від конкретного апаратного забезпечення створених систем та націлимося лише на оптимальність використаних примітивів та абстракцій у реалізованій розробці. Єдиним фактором, що буде мати значний вплив на похибку

вимірювань - це мережа. Важко переоцінити вплив який мережа має на час реагування - навіть найбільш оптимізована система буде працювати зі швидкістю її найповільнішої складової і в даному випадку саме мережа буде вносити вклад у час реагування. Зменшити цей вклад можна за допомогою двох факторів: зблизити оркестратор та виконавців розташувавши їх в одній мережі, що зменшить кількість мережевих стрибків до мінімуму та за допомогою масовості тестових випадків, що дозволить отримати інформацію, щодо розподілу часу відповіді та швидкодії та не бути прив’язаними до конкретних чисел.

Критерій вартості буде однозначно описувати кількість вкладених ресурсів. За допомогою цього критерію можна буде визначити доцільність вкладення певних ресурсів в підготовку інфраструктури. Моделювання витрат буде проводитися на основі типового навантаження робочого кластеру.

Хоч типове навантаження на кластер відрізняються від кластера до кластера, проте все ж таки можна виділити певні шаблони поведінки, такі як наприклад,

1. Часте виконання швидких функцій та рідке - повільних
2. Повна побудова кластеру на основі швидких функцій
3. Повне превалювання повільних функцій над швидкими

4.2 Огляд розробленої програми тестування навантаження

Розроблена для аналізу програма моделюватиме реальний приклад з робочого середовища, що буде включати в себе послідовне виконання трьох функцій, які будуть імітувати певне навантаження. Оркестратор буде отримувати на вхід запити на виконання сталої довжини та однакового змісту. Схему організації оркестратора наведено на рисунку 4.1

Тестування реагування на навантаження можна провести з використанням кількох стратегій. Оскільки додаток розподілено на дві умовні частини, то варто

перевірити здатність приймаючої сторони працювати під різним активним навантаженням та здатність виконуючої сторони оперативно обробляти задачі без деградації потужності виконання.

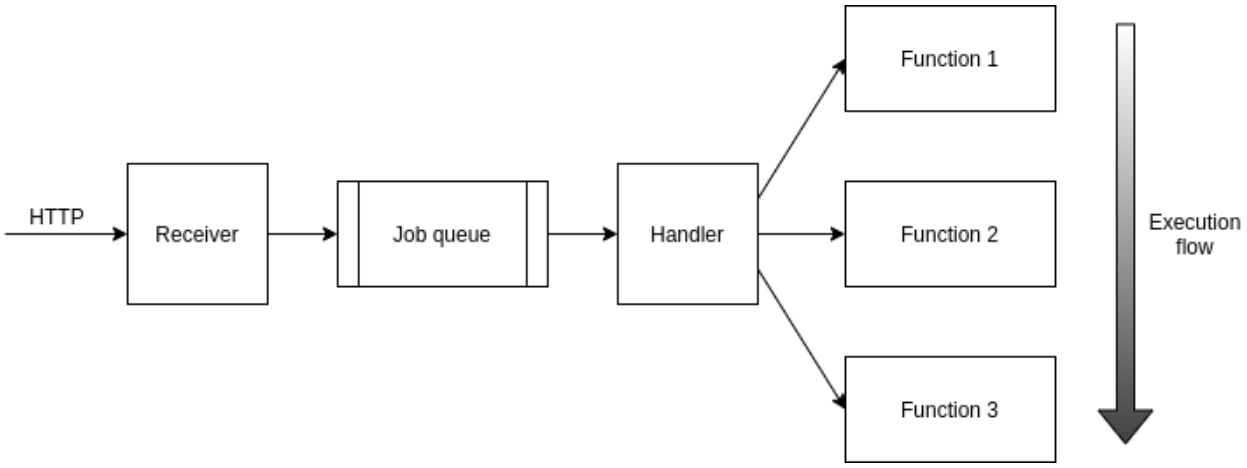


Рисунок 4.1 - Схема тестової конфігурації обробників

Для тестування навантаження використовувався програмний додаток Apache Benchmark, який дозволяє створювати фронти навантаження за заданими параметрами. На основі виконаних запитів додаток будує аналітику якості відповіді на основі часу на з’єднання, часу очікування, часу обробки та загального часу на виконання запиту. Окрім параметрів часу даний додаток дозволяє перевірити якість з’єднання за допомогою аналізу швидкості обміну даними з сервером. Додаток дозволяє налаштовувати велику кількість параметрів запитів до серверу, проте в даному контексті нас цікавить два параметри - це кількість одночасних з’єднань та загальна кількість запитів, за допомогою цих параметрів ми зможемо перевірити здатність витримувати навантаження головних компонентів системи.

4.3 Навантажувальне тестування

Першим кандидатом на тестування буде система прийому задач - для перевірки надійності цього компоненту буде використано параметр одночасних запитів - таким чином ми зможемо зрозуміти максимально допустиме навантаження, після якого додаток стає нефункціональним.

Навантаження буде перевірятися на одній репліці сервісу без додаткових налаштувань, таких як балансування навантаження чи інших. Окремо варто зазначити, що одночасна кількість обробників - це стале число, яке дорівнює 100.

Для тестування прийомної частини, загальна кількість запитів буде сталою та дорівнюватиме 500.

В рамках першого тесту, кількість одночасних запитів становитиме 20. В рамках даного тесту ми отримали швидкість часу відповіді, що становить 11мс за медіаною, 27 мс за 95 процентилем та 34мс за 99м процентилем. Ці дані свідчать, що затримок в обробці запитів немає та додаток функціонує нормально. Отримана таблиця з додатку Apache Benchmark продемонстрована на рисунку 4.2.

Наступним кроком буде збільшення кількості одночасних з’єднань до 50. При збільшенні кількості одночасних з’єднань в 2.5 раз, ми отримали менш ніж двократний приріст часу відповіді, тому можна вважати, що система продовжує функціонувати стабільно. Головні процентилі становлять - 28 мс за медіаною, 38 мс за 95м процентилем та 46 мс за 99м процентилем. Результати тестування приведено на риснунку 4.3.

При збільшенні кількості одночасних вдвічі з попереднім тестом, тобто до 100 одночасних з’єднань, ми не зіштовхнулися з критичною деградацією сервісу. За медіаною час відповіді становив 56мс, за 95 процентилем час відповіді становить 63 мс та за 99 мс процентилем час відповіді становить 66 мс. Таким чином хоч час відповіді за медіаною і зріс вдвічі, проте ми не втратили швидкодію додатку. Фактично всі наступні налаштування з більшою кількістю одночасних з’єднань належать до окремого щабелю highload. Результати цього тесту приведено на рисунку 4.4.

Завдяки тому що додаток майже не втратив у швидкодії під таким навантаженням ми можемо підняти ліміт далі - до 150 одночасних з’єднань. В такому режимі роботи ми отримуємо фактично двократний приріст часу відповіді, що становить 94 мс, 116 мс та 118 мс відповідно до головних процентилів. Хоч дані числа є і порівняно незначними - такий приріст показує, що додаток не здатний ефективно обробляти потік задач і тому зазвичай в таких ситуаціях вмикаються механізми горизонтального масштабування та встановлюється балансувальник навантаження між усіма репліками цього сервісу. Результати зображено на рисунку 4.5.

На рисунку 4.5 наведено порівняння росту часу відповіді за головними квантилями в залежності від кількості одночасних навантажень за допомогою неї можна помітити, що за 99м процентилем максимальний час відповіді було досягнуто ще в середині тестування і що зі збільшенням навантаження почалась більша деградація системи, яку стало помітно у все більшій кількості випадків.

```
Concurrency Level:      20
Time taken for tests:   0.346 seconds
Complete requests:      500
Failed requests:        99
   (Connect: 0, Receive: 0, Length: 99, Exceptions: 0)
Total transferred:      195385 bytes
Total body sent:        109000
HTML transferred:       140885 bytes
Requests per second:    1444.75 [#/sec] (mean)
Time per request:       13.843 [ms] (mean)
Time per request:       0.692 [ms] (mean, across all concurrent requests)
Transfer rate:          551.33 [Kbytes/sec] received
                        307.57 kb/s sent
                        858.91 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.8      0       7
Processing:     4   13   5.5     11      38
Waiting:        4   13   5.4     11      37
Total:          4   13   5.7     11      39

Percentage of the requests served within a certain time (ms)
  50%     11
  66%     13
  75%     15
  80%     16
  90%     21
  95%     27
  98%     32
  99%     34
 100%     39 (longest request)
```

Рисунок 4.2 - 20 одночасних з’єднань

```
Concurrency Level:      50
Time taken for tests:   0.331 seconds
Complete requests:      500
Failed requests:        86
   (Connect: 0, Receive: 0, Length: 86, Exceptions: 0)
Total transferred:      195403 bytes
Total body sent:        109000
HTML transferred:       140903 bytes
Requests per second:    1512.35 [#/sec] (mean)
Time per request:       33.061 [ms] (mean)
Time per request:       0.661 [ms] (mean, across all concurrent requests)
Transfer rate:          577.18 [Kbytes/sec] received
                        321.96 kb/s sent
                        899.15 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.8      0       4
Processing:     2   28   5.0     28      52
Waiting:        2   28   5.0     28      52
Total:          2   28   5.1     28      52

Percentage of the requests served within a certain time (ms)
  50%     28
  66%     30
  75%     31
  80%     31
  90%     33
  95%     38
  98%     43
  99%     46
 100%     52 (longest request)
```

Рисунок 4.3 - 50 одночасних з’єднань

```
Concurrency Level:      100
Time taken for tests:   0.348 seconds
Complete requests:      500
Failed requests:        98
   (Connect: 0, Receive: 0, Length: 98, Exceptions: 0)
Total transferred:      195382 bytes
Total body sent:        109000
HTML transferred:       140882 bytes
Requests per second:    1438.28 [#/sec] (mean)
Time per request:       69.528 [ms] (mean)
Time per request:       0.695 [ms] (mean, across all concurrent requests)
Transfer rate:          548.85 [Kbytes/sec] received
                        306.20 kb/s sent
                        855.05 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   5.4      0      21
Processing:     5   52  10.8     55      66
Waiting:        5   52  10.9     55      66
Total:         15   55   6.8     56      68

Percentage of the requests served within a certain time (ms)
  50%     56
  66%     58
  75%     60
  80%     61
  90%     62
  95%     63
  98%     65
  99%     66
 100%     68 (longest request)
```

Рисунок 4.4 - 100 одночасних з’єднань

```
Concurrency Level:      150
Time taken for tests:   0.369 seconds
Complete requests:      500
Failed requests:        99
   (Connect: 0, Receive: 0, Length: 99, Exceptions: 0)
Total transferred:      195382 bytes
Total body sent:        109000
HTML transferred:       140882 bytes
Requests per second:    1354.64 [#/sec] (mean)
Time per request:       110.730 [ms] (mean)
Time per request:       0.738 [ms] (mean, across all concurrent requests)
Transfer rate:          516.94 [Kbytes/sec] received
                        288.39 kb/s sent
                        805.33 kb/s total

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   4.9      0      14
Processing:     1   84  27.7     93     141
Waiting:        1   84  27.7     93     141
Total:          2   88  24.2     94     152

Percentage of the requests served within a certain time (ms)
  50%     94
  66%    101
  75%    105
  80%    107
  90%    115
  95%    116
  98%    117
  99%    118
 100%    152 (longest request)
```

Рисунок 4.5 - 150 одночасних з’єднань

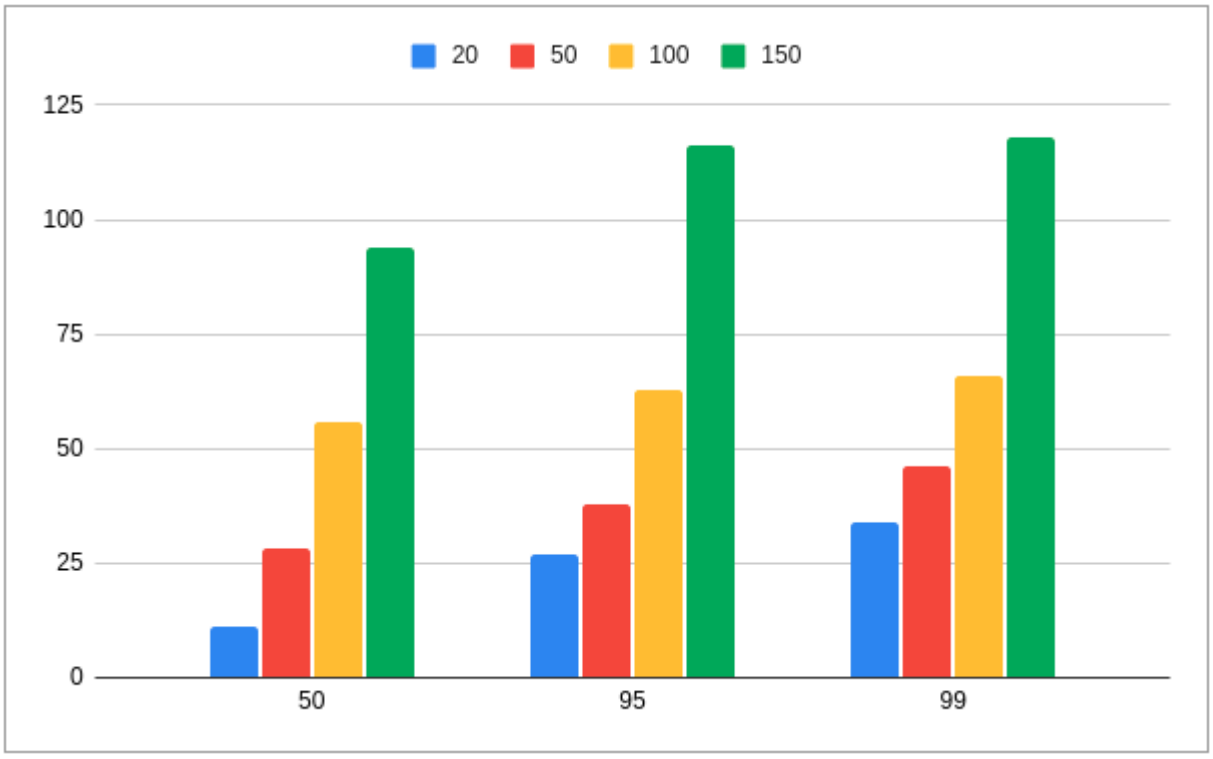


Рисунок 4.6 - Порівняння результатів тестування системи отримання

Другою частиною тестування надійності системи - є тестування здатності системи фонових обчислень підтримувати однакову якість виконання роботи без збільшення на час перемикання між задачами.

В рамках цього тестування буде використовуватися друге налаштування Apache Benchmark - це загальна кількість запитів. За допомогою цього параметру ми зможемо навантажити систему відкладених обчислень та перевіримо здатність до обробки значних об’ємів інформації. Одночасна кількість з’єднань буде встановлена на рівні 20, а загальна кількість запитів починатиметься від 500. Кожне виконання конвеєру викликає 3 взаємопов’язані функції, тому кількість даних становить 1500 викликів. Методологія виміру часу викликів наступна - на початку виклику обробки ми заміряємо часову мітку, так само ми робимо на початку виклику самої FaaS функції, також заміряємо час наприкінці функції-обробника та FaaS функції. На виході ми отримуємо для проміжки часу - час від виклику обробника і до виклику хмарної функції та час від завершення функції і до завершення обробника. Проміжки часу в наносекундах. Як можна побачити на рисунку 4.7 - проміжок часу на початок та завершення функції майже ніяк не залежить від часу виконання функції і є приблизно сталим числом.

Наступним кроком буде збільшення навантаження вдвічі - до 1000 задач загалом. Враховуючи виконання трьох функцій на кожну задачу - система виконала 3000 викликів. Графік даних викликів подано на рисунку 4.8 - з нього можна помітити, що при збільшенні кількості задач не зростає середній час на виконання окремої з них, можна помітити виражені піки - це моменти в які система OpenFaaS масштабує хмарну функцію на нові репліки і тому в такі моменти помітно ріст часу виконання.

Для закріплення теорії можна підняти кількість задач ще вдвічі - до 1000 задач загалом і 6000 викликів функцій. Результати цього тесту можна побачити на рисунку 4.9. На основі цих даних можна судити про те, що система фонової обробки може стабільно обробляти практично необмежені об’єми задач завдяки вбудованій системі обмеження на основі черги.

Всі три тести показують тимчасові зростання проміжків обробки - це можна пояснити застарілістю апаратного забезпечення на якому було виконано прикладні тести, що призводило до повільного масштабування примірників хмарних функцій. В рамках реальних системи таких ефектів не спостерігалося б.

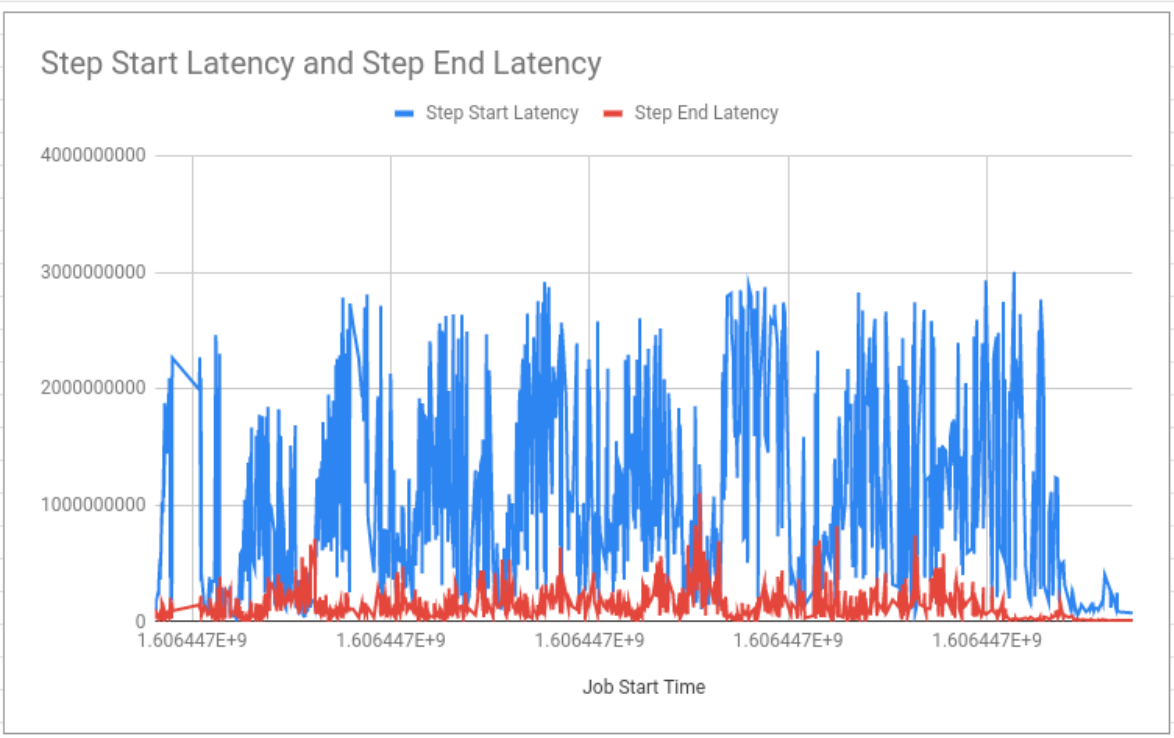


Рисунок 4.7 - 500 задач загалом

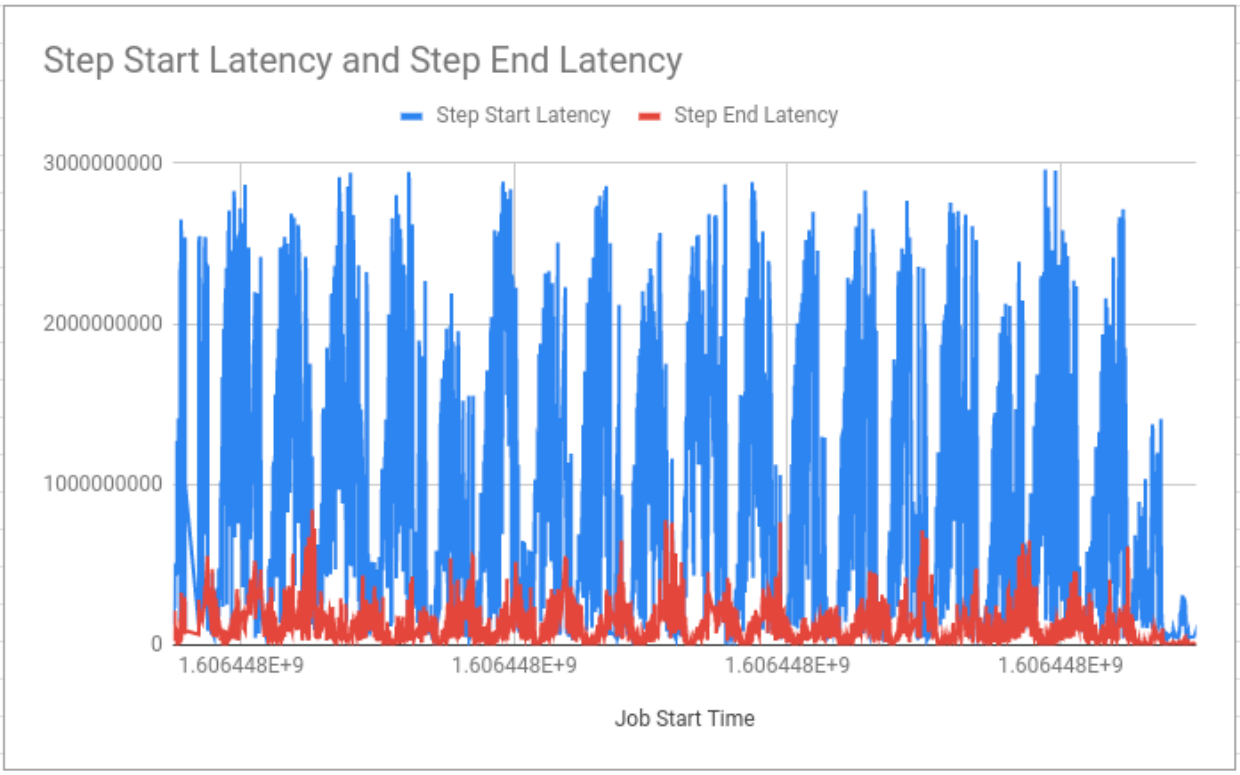


Рисунок 4.8 - 1000 задач загалом

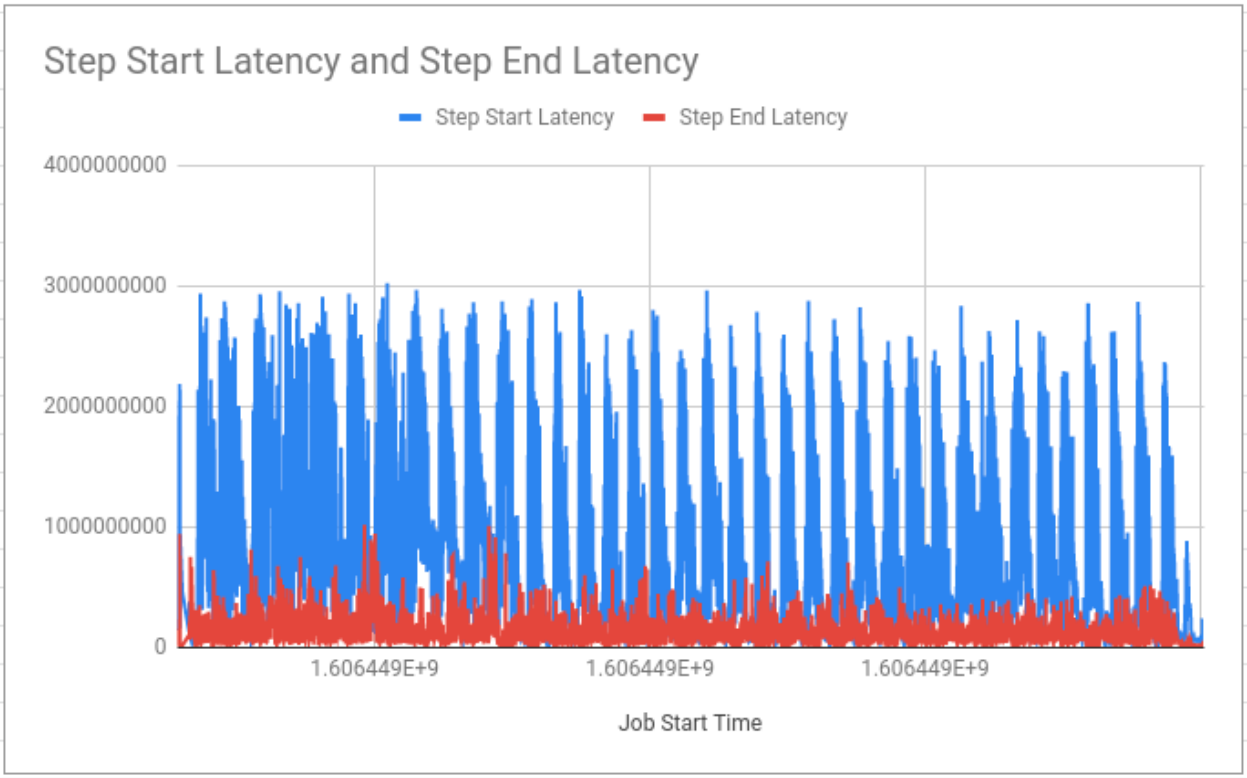


Рисунок 4.9 - 2000 задач загалом

4.4 Моделювання вартості

Для порівняння вартості на обслуговування подібного кластеру було обрано послуги хмарного провайдера AWS.

В рамках IaaS моделі було використано послугу EC2, що являє собою надання серверів для власного використання. Завдяки зручній системі конфігурації даного провайдера було обрано середній за потужністю клас серверів t3.large - це сервери загального призначення, що здатні виконувати будь-яку поставлену задачу, такі сервери обладнані двома vCPU та 8 ГБ оперативної пам’яті з під’єднаними дисками ємністю 50 ГБ. Завдяки окремій можливості EC2 можливо використовувати сервери зниженої вартості, а саме Spot Instances, що дає можливість зменшити вартість таких серверів в порівнянні з звичайними серверами мінімум вдвічі. Плата за таку конфігурацію відбувається щомісячно та сумується за кожен окремий використаний сервер. Сконфігурований таким чином сервер коштуватиме 46.38 USD щомісячно, ріст ціни на такі сервери відбувається прямопропорційно і тому два сервери будуть коштувати 92.75 USD, три - 139.12 USD ітд.

Для створення FaaS архітектури було використано послугу AWS Lambda, що являє собою послугу надання безсерверних викликів функцій. Для моделювання було обрану схожу за параметрами IaaS систему з 8 ГБ пам’яті та двома vCPU. Оплата за послуги Lambda виконується на основі сумарного часу виконання функцій за місяць з урахуванням виділених апаратних ресурсів. Для моделювання було обрано, що в місяці в середньому 30 днів, кожна функція буде оброблятися в середньому 400 мс, а середня частота запитів становитиме 50, 100 та 200 RPS. Провізіонування конкурентного виконання функцій в даній моделі було вимкнено оскільки хоч такий підхід дозволить пришвидшити виконання функцій, але вартість такої послуги буде співмірна до вартості самої Lambda.

На основі наведених даних для моделювання, за допомогою конструктора вартості AWS було отримано наступні розрахунки:

- При 15 RPS щомісячна вартість буде становити 2,081.38 USD
- При 50 RPS щомісячна вартість буде становити 6,937.93 USD
- При 100 RPS щомісячна вартість буде становити 13,875.87 USD
- При 200 RPS щомісячна вартість буде становити 27,751.74 USD

В першу чергу дані цифри показують, що при збільшенні темпів навантаження “чистий” підхід FaaS втрачає всі свої переваги та потребує перегляду архітектури. На рисунку 4.10 зображено графік порівнянь вартості FaaS рішення та OpenFaaS на основі IaaS - в другій конфігурації для адекватного підтримання навантаження кількість серверів 2, 3, 5 та 10 відповідно до градації збільшення RPS, також додатково було промодельовано вартість при збільшенні кількості серверів в 10 раз більше ніж при попередній конфігурації,тобто 20, 30, 50 та 100 серверів відповідно.

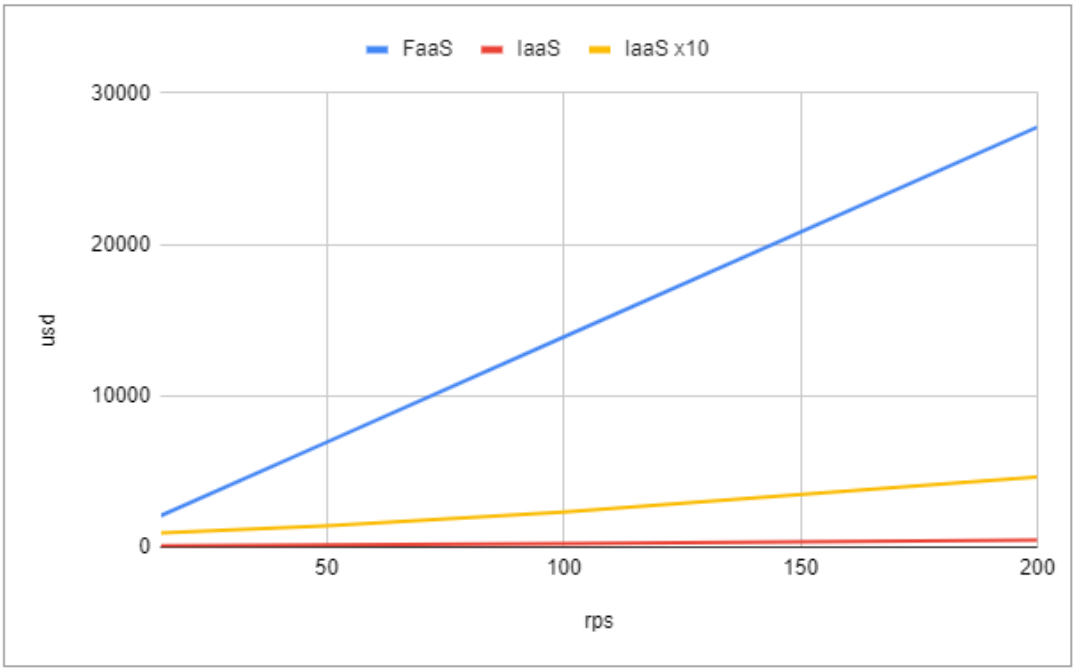


Рисунок 4.10 - Порівняння вартості FaaS та OpenFaaS на основі IaaS

Згідно з графіком помітно лінійний ріст вартості FaaS рішення та здатність IaaS рішення до економії коштів при збереженні функціональних властивостей. Таким

чином при використанні IaaS можливо значно збільшити штат серверів при збереженні відносної дешевизни всієї архітектури загалом, що неможливо у випадку з FaaS.

Висновки

Реалізована система оркестрації є сучасною системою координування окремих ізольованих FaaS функцій, що здатна проводити оркестрацію та хореографію викликів за заданими кінцевим користувачем параметрами. Результуючий продукт можна швидко доставляти в робоче середовище, завдяки тому, що система вже включає до себе всі необхідні інструменти обмеження виконання та відкладеності обчислень. За допомогою асинхронної черги можливе розділене масштабування як прийомної частини так і частини, що відповідає за обробку навантаження. Абстрактність інтерфейсу прийому та зберігання даних дозволяє розширення таких компонентів під необхідності кінцевого користувача що, наприклад, дозволить розширити канали прийому задач.

Стабільність роботи та безпека масштабування компонентів систем забезпечена використаними примітивами, такими як граф потоку керування та абстрактний детермінований автомат, що дозволяють уявляти кожен конвеєр виконання у вигляді шаблону, до якого потрібно лише додати сутність, що буде реплікувати такі шаблони - кожен з них буде контекстуально обмежено один від одного. Єдиною точкою дотику всіх компонентів буде лише черга обробки з якої кожен обробник буде намагатися забрати роботу. Такий підхід дуже добре накладається на принцип роботи планувальника обраної мови програмування Go - в рамках роботи жадібного GMP планувальника всі отримані задачі на конкретному планувальнику будуть рівномірно розподілені між усіма ядрами та потоками виконання процесора.

Перевірка життєздатності розробки виконувалася на основі навантажувального тестування в два етапи. Перший етап перевіряв здатність приймальної сторони витримувати одночасне навантаження при цьому не втрачаючи якість обслуговування кожного окремого запиту. Другим етапом була

перевірка системи обробки задач при якому цю підсистему перевіряли на здатність обробляти значні об’єми задач. Обидва етапи показали, що ані система прийому, ні система обробки не втрачала своїх функціональних властивостей навіть при критичному навантаженні системи.

Так як система була розроблена у вигляді програмної бібліотеки, то таку розробку можна буде застосувати у будь-якій необхідній ситуації, налаштувавши її під конкретні необхідні умови. Розроблена система має простий абстрактний інтерфейс та чітко визначений контракт використання тому налаштування для виконання конкретної задачі буде неважким. Окремо варто зазначити, що бібліотека постачається з базовими імплементаціями абстрактних інтерфейсів систем прийому навантаження та зберігання даних до персистентного сховища, що дозволить кінцевому користувачу не витрачати час на повторну імплементацію цих компонентів.

В майбутньому цю систему можна розширити, додавши до неї трасування кожної обробки, додаванням метрик роботи та спробувати поєднати з існуючими бібліотеками, що дозволяють працювати з хмарними ресурсами як з кодом. Таким чином ми зможемо підвищити observability кінцевої системи та надати можливість розгортання всієї необхідної інфраструктури на базі даної бібліотеки, що зменшить кількість необхідної роботи для інтеграції такої системи в цільову інфраструктуру користувача.

Список використаних джерел

1. Цінова політика Amazon AWS [Електронний ресурс]. - Режим доступу: https://aws.amazon.com/pricing
2. Цінова політика Microsoft Azure [Електронний ресурс]. - Режим доступу: https://azure.microsoft.com/en-us/pricing
3. Цінова політика Google Cloud [Електронний ресурс]. - Режим доступу: https://cloud.google.com/pricing
4. Опис архітектурного стеку OpenFaaS [Електронний ресурс]. - Режим доступу: https://docs.openfaas.com/architecture/stack/
5. What is graph [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
6. Wei Le, “Control flow analysis” [Електронний ресурс]. Режим доступу: http://web.cs.iastate.edu/~weile/cs513x/4.ControlFlowAnalysis.pdf
7. What is dominator [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Dominator_(graph_theory)
8. Michelle Strout, “Control-flow and loop detection” [Online]. Available: https://www.cs.colostate.edu/~mstrout/CS553Fall06/slides/lecture13-control.pdf
9. Alan Turing, “On computable numbers, with an application to the Entscheidungsproblem”, 1937
10. https://en.wikipedia.org/wiki/Finite-state_machine
11. Donald E. Knuth. “The art of computer programming Vol.1 3rd ed”, Boston: Addison-Wesley, 1997
12. Атомарна операція RPOPLPUSH Redis [Електронний ресурс]. Режим доступу: https://redis.io/commands/rpoplpush
13. База даних MongoDB [Електронний ресурс]. Режим доступу: https://www.mongodb.com/what-is-mongodb

14. Реалізація жадібного планувальника мови програмування Go [Електронний ресурс]. Режим доступу: https://github.com/golang/go/blob/master/misc/cgo/gmp/gmp.go