НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ
Кафедра автоматизованих систем обробки інформації та управління

УДК 04.089

«До захисту допущено»
В.о. завідувача кафедри
_____ *О.А.Павлов*
(підпис) (ініціали, прізвище)
“___”_____2019 р.

Дипломний проект
на здобуття ступеня бакалавра

з напряму підготовки *6.050101 «Комп’ютерні науки»*

на тему: «*Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js*»

Виконав: студент *4* курсу, групи *ІС-51*

Дробот Даніїл Борисович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник *ст. вик. Олійник Ю.О.* _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з графічної документації *доц., к.т.н., доц. Тєлишева Т.О.* _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____
(підпис)

Київ – 2019 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) *інформатики та обчислювальної техніки*
(повна назва)

Кафедра *автоматизованих систем обробки інформації та управління*
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки (програма професійного спрямування) *6.050101* .

«Комп’ютерні науки» («Інформаційні управляючі системи та технології») .

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ *О.А. Павлов*
(підпис) (ініціали, прізвище)
“___”_____2019 р.

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЕКТ СТУДЕНТУ

Дроботу Даніїлу Борисовичу
(прізвище, ім’я, по батькові)

1. Тема проекту «*Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js* »

керівник проекту *Олійник Ю.О., ст. в*
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від “*23*”*квітня* 2019 р. №1181-с

2. Термін подання студентом проекту “*03*”*червня* 2019 року

3. Вихідні дані до проекту

Технічне завдання

4. Зміст пояснювальної записки

1. Загальні положення: основні визначення та терміни, опис предметного середовища, огляд ринку програмних продуктів, постановка задачі

2. Інформаційне забезпечення: вхідні дані, вихідні дані, опис структури бази даних

3. Математичне забезпечення: змістовна та математична постановки задачі, обґрунтування та опис методу розв’язання

4. Програмне та технічне забезпечення: засоби розробки, вимоги до технічного забезпечення, архітектура програмного забезпечення, побудова звітів

5. Технологічний розділ: керівництво користувача, методика випробувань

програмного продукту

5. Перелік графічного матеріалу

1. Схема структурна варіантів використань

2. Схема структурна класів програмного забезпечення

3. Схема структурна розгортання

4. Схема структурна послідовності

5. Схема структурна діяльності

6. Рішення з математичного забезпечення

6. Консультанти розділів проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *«15» лютого 2019 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1.	*Вивчення рекомендованої літератури*	*01.03.2019*	
2.	*Аналіз існуючих методів розв'язання задачі*	*29.03.2019*	
3.	*Постановка та формалізація задачі*	*16.04.2019*	
4.	*Розробка інформаційного забезпечення*	*23.04.2019*	
5.	*Алгоритмізація задачі*	*30.04.2019*	
6.	*Обґрунтування використовуваних технічних засобів*	*05.05.2019*	
7.	*Розробка програмного забезпечення*	*10.05.2019*	
8.	*Налагодження програми*	*20.05.2019*	
9.	*Виконання графічних документів*	*22.05.2019*	
10.	*Оформлення пояснювальної записки*	*27.05.2019*	
11.	*Подання ДП на попередній захист*	*30.05.2019*	
12.	*Подання ДП на основний захист*	*03.06.2019*	
13.	*Подання ДП рецензенту*	*05.06.2019*	

Студент _____ Д.Б. Дробот
(підпис)

Керівник проекту _____ Ю.О. Олійник
(підпис)

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
			Документи текстові		
1	А4	ДП ІС-5106.1181-с.ПЗ	Пояснювальна записка	58	
2	А4	ДП ІС-5106.1181-с.ТЗ	Технічне завдання	10	
3	А4	ДП ІС-5106.1181-с.ПМВ	Програма та методика випробувань	11	
			Матеріали графічні		
4	А4	ДП ІС-5106.1181-с.ССВ	Схема структурна варіантів використань	1	
5	А4	ДП ІС-5106.1181-с.ССК	Схема структурна класів програмного забезпечення	1	
6	А4	ДП ІС-5106.1181-с.ССР	Схема структурна розгортання	1	
7	А4	ДП ІС-5106.1181-с.ССП	Схема структурна послідовності	1	
8	А4	ДП ІС-5106.1181-с.ССП	Схема структурна діяльності	1	
6	А3	ДП ІС-5106.1181-с.МЗ	Рішення з математичного забезпечення	1	

Зм.	Арк.	ПІП	Підп.	Дата	ДП ІС-5106.1181-с.ВДП		
Розробн.		Дробот Д.Б.			Відомість дипломного проекту	Літ.	Лист 1 / Листів 1
Керівн.		Олійник Ю.О.					
Консульт.						КПІ ім. Ігоря Сікорського кафедра АСОІУ гр. ІС-51	
Н/контр.		Тєлишева Т.О.					
В.о.зав.каф.		Павлов О.А.					

Пояснювальна записка
до дипломного проекту

на тему: Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js

Київ – 2019 року

АНОТАЦІЯ

Структура та обсяг роботи. Пояснювальна записка дипломного проекту складається з п’яти розділів, містить 11 рисунків, 14 таблиць, 1 додаток, 24 посилання.

В дипломному проекті реалізована тема «Комплекс задач розподіленої обробки реактивного потоку даних на Node.js» метою якої є полегшення розробки розподілених систем для обробки даних та підвищення швидкості обробки даних.

Для пошуку частих елементів потоку було використано метод повної обробки.

В результаті виконання роботи було створено систему, що дозволяє надавати підтримку користувачам у більш зручному форматі та з використанням сентиментального аналізу тексту.

РЕАКТИВНЕ ПРОГРАМУВАННЯ, РЕАКТИВНІ ПОТОКИ, РОЗПОДІЛЕНІ СИСТЕМИ, NODE.JS

ABSTRACT

Structure and scope of work. The explanatory note of the diploma project consists of five sections, containing 11 figures, 14 tables, 1 application, 24 sources.

The diploma project implemented the theme «Software Solution for Distributed Reactive Data Stream Processing using Node.js » aimed at improving quality and speed of customer's support process.

The complete processing method was used to find frequent stream members.

The result of this work is system that allows to give users support in more comfortable way with the usage of sentimental data analysis.

REACTIVE PROGRAMMING, REACTIVE STREAMS, DISTRIBUTED SYSTEMS, NODE.JS

ЗМІСТ

ВСТУП

Сучасні застосунки часто мають працювати з об’ємом даних завеликим для обробки на одній машині. Це може бути як і Data Mining, так і серверний застосунок з великою кількістю запитів. З кожним роком збільшується кількість активних користувачів мережі інтернет, а тому збільшується і навантаження яке серверні застосунки мають витримувати, так найбільш популярний пошуковий сайт Google в середньому оброблює 40 000 запитів у секунду[1]. Біржа NASDAQ оброблює в середньому 100 000 запитів на секунду з середнім часом відгуку 40 мікро секунд[2].

Відео гігант Netflix має налаштовану інфраструктуру для обробки пікового навантаження у 1 000 000 запитів на секунду[3]. Для обробки трафіку такого обсягу сервіси Netflix використовують комбінацію реактивних потоків Java та веб-серверів на Node.js[4]

Реактивні потоки є широко вживаною технологією у світі Java. Вони використовуються при побудові високо навантажених, масштабованих систем, які мають вимоги то надійності[5].

Основні визначення наведені в специфікації реактивних потоків. Вона описує такі сутності:

- Subscriber
- Publisher
- Subscription
- Processor [6]

Дипломний проект присвячений розробці комплексу задач розподіленої обробки реактивного потоку даних на Node.js.

Перелік умовних позначень

JS – мова програмування JavaScript

Практичне значення одержаних результатів. Розроблено алгоритм для розподіленої обробки реактивного потоку даних на Node.js

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

Реактивні потоки – стандарт для асинхронної обробки потоків з неблокуючим зворотнім тиском. Стандарт та його сутності визначені в специфікації реактивних потоків[6]. Зворотній тиск є ключовою частиною концепції реактивних потоків, він дозволяє споживачу потоку контролювати максимальну частоту отримання нових елементів, що дозволяє розподіляти пікові навантаження впродовж часу. Основними сутностями реактивних потоків є Publisher, Subscriber, Subscription, Processor. Publisher – відповідає за наповнення потоку даних, Subscriber – приймає вхідний потік даних, та проводить над ним необхідні операції, Subscription – встановлює відповідність між Publisher та Subscriber, оскільки в одного Publisher може бути декілька Subscriber, що знаходяться на різних рівні споживання елементів. Processor – комбінація Publisher та Subscriber в одну сутність[6].

Специфікація вводить свої терміни, які наведені у таблиці 1.1 також специфікація накладає обмеження на сутності, що наведені у таблицях 1.2 – 1.5

Таблиця 1.1 – Терміни специфікації

Термін	Означення
Сигнал	Один з методів *onSubscribe, onNext, onComplete, onError, request(n), cancel*
Запит	Кількість елементів яку запросив Subscriber і яку ще не надав *Publisher*
Синхронно	Виконується в тому ж потоці та тому ж стеку викликів, що викликав
Завершитись нормально	Тільки повертати значення обумовленого типу. Єдиний спосіб повідомити про помилку – метод *onError*
Відзивчивість	Готовність реагувати
Не перешкождаючий	Метод, який виконується якомога швидше у викликаючому потоці. Це означає, наприклад, уникання важких обчислень
Термінальний стан	Для *Publisher* – коли сигналізовано *onComplete* або *onError*. Для *Subscriber* – коли було отримано сигнал *onComplete* або *onError*

Продовження таблиці 1.1

NOP	Застосування, виконання якого не має видимого ефекту на потік, що його викликав
Зовнішня синхронізація	Використання методів синхронізації доступу до ресурсів, які не описані у специфікації, такі як *atomics, monitors, locks*
Потоко-безпечний	Може бути безпечно виконаний синхронно чи асинхронно без зовнішньої синхронізації

Таблиця 1.2 - Обмеження для Publisher

# Правила	Правило
1	Загальна кількість сигналів onNext що надійшли до Subscriber має бути не більшою за кількість елементів яку він запитав
2	Publisher може передати менше сигналів onNext, ніж було запитано, та завершити Subscription викликавши onComplete або onError
3	onSubscribe, onNext, onError, onComplete мають бути надіслані у потоко-безпечному в разі надходження з різних потоків – використовувати зовнішню синхронізацію
4	Якщо Publisher не виконується коректно – він має надіслати onError

Продовження таблиці 1.2

5	Якщо Publisher успішно завершився він має надіслати onComplete
6	Якщо Publisher надсилає Subscriber'у сигнал onError, onComplete, Subscription що йому відповідає вважатиметься скасованим
7	Після переходу до термінального стану, не повинно передаватись сигналів
8	Якщо Subscription був скасований, він має зрештою перестати надсилати сигнали
9	Publisher.subscribe повинен викликати onSubscribe перед тим як надсилати сигнали
10	Publisher.subscribe може бути викликан безліч разів, але кожен раз з різним Subscriber
11	Publisher може підтримувати декілька Subscriber та вирішує чи є Subscription одно адресною чи багато адресною

Таблиця 1.3 – Обмеження для Subscriber

# Правила	Правило
1	Subscriber має надіслати сигнал запиту через Subscription.request(n: number) щоб одержати сигнали onNext
2	Якщо Subscriber підозрює, що процессинг сигналів негативно впливає на відзивчивість Publisher'а, рекомендується асинхронно оброблювати сигнали

Продовження таблиці 1.3

3	Subscriber.onComplete Subscriber.onError не можуть викликати методи Subscription чи Publisher
4	Subscriber.onComplete та Subscriber.onError повинен вважати Subscription скасованим після отримання сигналу
5	Subscriber повинен викликати Subscription.cancel на Subscription після onSubscribe, якщо у ньго вже є активний Subscription
6	Subscriber повинен викликати Subscription.cancel якщо Subscription більше не потрібен
7	Subscriber повинен переконатись, що усі виклики Subscription та методи скасування виконуються послідовно або з використанням зовнішньої синхронізації
8	Subscriber має бути готовий оброблювати onNext після виклику Subscription.cancel
9	Subscriber має бути готовий отримати сигнал onComplete до виклику Subscription.request
10	Subscriber має бути готовий отримати сигнал onError до виклику Subscription.request
11	Subscriber повинен переконатись, що всі виклики методів сигналів відбуваються до обробки сигналу.
12	Subscriber.onSubscribe повинен викликатись не більше одного разу.
13	Виклик onSubscribe, onNext, onError, onComplete повинен завершитись нормально, окрім випадку, коли передано

	параметр null, в таких випадках має бути викинута помилка.

Таблиця 1.4 – Обмеження для Subscription

# Правила	Правило
1	Subscription.request та Subscription.cancel повинні викликатись лише з контексту відповідного Subscriber
2	Subscription повинен дозволяти Subscriber викликати Subscription.request синхронно з onNext, onSubscribe
3	Subscription.request повинен накладати граничне обмеження на глибину рекурсії синхронних викликів
4	Subscription.request бажано не має містити трудомістких операцій, для уникнення затримок у викликаючого
5	Subscription.cancel не повинен містити трудомістких операцій, має бути ідемпотентним та потоко-безпечним
6	Після того, як Subscription був скасованим, додаткові виклики Subscription.request мають бути NOP
7	Після того, як Subscription був скасованим, додаткові виклики Subscription.cancel мають бути NOP
8	Поки Subscription не був скасованим, виклики Subscription.request повинні реєструвати кількість елементів, що мають бути передані
9	Доки Subcription не був скасованим, виклики Subscription.request мають повертати сигнал onError, якщо аргумент <= 0
10	Доки Subcription не був скасованим, виклики Subscription.request

	можуть синхронно викликати onComplete або onError

Продовження таблиці 1.4

11	Доки Subcription не був скасованим, виклик Subscription.cancel повинен запитати Publisher зрештою перестати надсилати сигнали
12	Доки Subcription не був скасованим, виклик Subscription.cancel повинен запитати Publisher видалити посилання на відповідного Subscriber
13	Доки Subcription не був скасованим, виклики Subscription.cancel, може перевести Publisher у термінальний стан, якщо немає інших Subscription
14	Виклик Subscription.cancel повинен завершитись нормально
15	Виклик Subscription.request повинен завершитись нормально
16	Subscription повинен підтримувати необмежені виклики request та повинен підтримувати запит до 2^63 – 1, запити вище – вважати не обмеженими.

Таблиця 1.5 – Обмеження Processor

# Правила	Правило
1	Processor відображає етап обробки, він є одночасно і Subscriber, і Publisher, та має виконувати правила обох
2	Processor може відновити роботу після отримання onError, але в

	такому випадку має вважати Subscription скасованим

Розподілена обробка – спосіб розв’язання задач в якому задачу розбивають на під-задачу, що розв’язуються одним чи більше комп’ютером[7]. Розподілена обробка даних використовується при роботі з великими масивами даних чи з трудомісткими задачами, оскільки допомагає знизити загальний час обробки та дозволяє збільшити кількість даних чи задач, що оброблюються в один момент часу.

Node.js – платформа для виконання JavaScript-ового коду[8]. Що часто використовується для написання розподілених систем.

Поєднання цих технологій спростить розробку надійних систем розподіленої обробки потоків даних.

1.1.1 Опис процесу діяльності

Процес – налаштування системи обробки та ввід потоку даних на обробку. Схема структурна діяльності наведена у графічному матеріалі.

Розподілена обробка даних - одночасне розв’язання різних частин одного завдання декількома процесорами одного або декількох комп’ютерів[7].

Типова архітектура системи розподіленої обробки даних підпадає в одну з категорії:

- клієнт-сервер архітектура - клієнтські вузли отримують задачі від головного серверу, оброблюють дані та повертають серверу для агрегації результатів
- три-рівнева архітектура - виносить клієнтську логіку у окремий компонент, дозволяючи використовувати stateless клієнтів.

- N-рівнева архітектура - виносить клієнтську логіку в окремі сервіси
- Peer-to-peer - архітектура в якій не має вузлів відповідальних за комунікацію між сервісами, а задачі строго розбиті між машинами[9]

1.1.2 Опис функціональної моделі

Специфікацію функціональної поведінки системи представимо у вигляді діаграми варіантів використання. На рисунку 1.1 наведено всі функції системи та описано акторів, які будуть їх використовувати.

Специфікацію функціональної поведінки системи представимо у вигляді діаграми варіантів використання. Структурну схему варіантів використання наведено у графічному матеріалі.

Безпосередньо із системою будуть взаємодіяти один актор:

- Користувач – це особа, якій була надана система

Відповідно до визначених варіантів використання виявлено функціональні вимоги та встановлено їх пріоритетність. Результат наведено в таблиці 1.6.

Таблиця 1.6 – Функціональні вимоги

Варіант використання	**Функціональна вимога**	**Пріоритет**
Конфігурування системи	1. Система надає можливість задати конфігурацію	Високий
Завантаження даних	2. Система надає можливість завантажити дані у вигляді потоку	Високий

Перегляд результатів	2.1. Система надає можливість переглянути результат обробки	Високий
Моніторинг системи	3. Система дає можливість переглянути актуальний стан її компонентів	Високий

1.2 Огляд наявних аналогів

Перед початком роботи над дипломним проектом було здійснено пошук застосунків зі схожою функціональністю.

Нині існують програмні продукти, які здійснюють розподілену обробку потоків. У таблиці 1.7 наведено назви наявних аналогів та режими доступу до них.

Таблиця 1.7

Назва	**Режим доступу**
Apache Flink	https://flink.apache.org/ [20]
Apache Spark	https://spark.apache.org/ [21]
Project Reactor	https://projectreactor.io/ [22]

Проаналізуємо функції, які виконують зазначені вище програмні продукти. Дані по функціям наведено в таблиці 1.8.

Таблиця 1.8

Аналоги / Функції	Apache Flink	Apache Spark	Project Reactor
Збреження стану обробки потоку	+	+	+
Пакетна обробка елементів	–	+	+
Можливість масштабування	+	+	-
Підтримка зворотного тиску	-	-	+

Із таблиці 1.8 видно, що ті аналоги, які представлені нині на ринку, не охоплюють увесь спектр функцій, необхідних для розподіленої обробки потоків, тому необхідно розробити новий засіб який би задовольняв сучасним вимогам.

1.3 Постановка задачі

1.3.1 Призначення розробки

Запропонований програмний продукт призначений для розподіленої обробки потоків даних. Під розподіленою обробкою мається на увазі розбиття основного потоку на під-потоки, виконання певних трансформації цих під-потоків в рамках інших процесів / машин та агрегація результатів. Результатом обробки буде трансформований реактивний потік даних. Реактивні потоки спрощують побудову важко-навантажених систем та додають надійності у їх роботу.

1.3.2 Цілі та задачі розробки

Мета роботи – удосконалення розподіленої обробки потоків, за рахунок використання реактивних потоків даних.

Для реалізації поставленої мети необхідно розв’язати наступні задачі:

- створення засобів для прийому потоку даних;
- створення засобів для розподілення потоку та агрегації результатів;
- створення засобів виводу даних;

Висновок до розділу

У даному розділі було розглянуто предметне середовище. Дипломний проект присвячений розробці системи для розподілення обробки реактивного потоку даних.

При описі предметного середовища було визначено та описано процеси діяльності, що представлені у вигляді двох незалежних процесів, для кожного з яких наведено структурні схеми. Також було створено функціональну модель системи, а саме: побудовано структурну схему варіантів використання, описано акторів системи та визначено функціональні вимоги з пріоритетами виконання відповідно до варіантів використання.

Наступним етапом роботи над дипломним проектом став пошук аналогів запропонованого застосунку з подібним функціоналом. Було порівняно функції, які виконують знайдені програмні продукти. На даний момент було виявлено декілька систем зі схожим функціоналом, але вони не охоплюють увесь спектр функцій, які запропоновані в даному дипломному проекті. Жоден з аналогів продуктів не використовує сентиментальний аналіз текстових даних.

Також у одному з підрозділів було сформовано постановку задачі, визначено призначення, мету та задачі розробки.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Вхідні дані

Вхідними даними є потік з чисел, який розбивається на під-потоки, що оброблюються без збереження даних та агрегуються у вихідний потік. Структурна схема потоку даних зображена на рис. 2.1

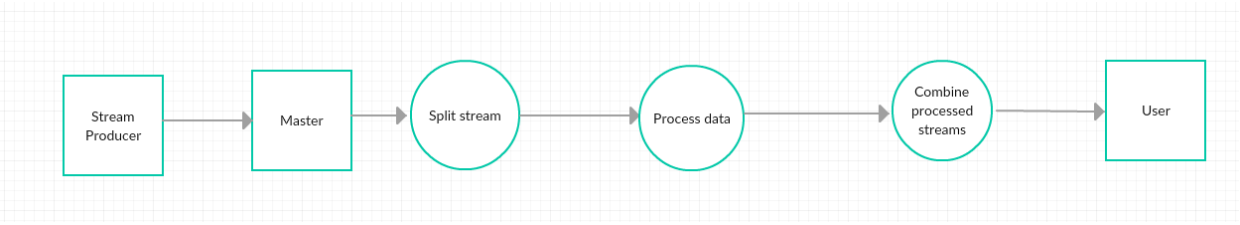


Рис.2.1 Схема структурна потоку даних

Оскільки на вхід очікується реактивний потік, то для операцій з мережею було обрано протокол передачі даних Rsocket. Rsocket - бінарний протокол прикладного рівня з підтримкою реактивної семантики. Для серіалізації повідомлень в Rsocket використовується механізм Protocol Buffers, що дозволяє описати формат даних та згенерувати програмний код для найбільш популярних мов програмування[10].

Для зручності потік буде містити в собі лише числа, без інших даних

2.2 Вихідні дані

Вихідними даними системи є оброблений потік. Тобто потік в якому кожен елемент є результатом обробки елементів вхідного потоку. Цей потік так само буде відкритий для доступу через Rsocket та використовуватиме Protocol Buffer для серіалізації повідомлень.

2.3 Опис структури бази даних

Оскільки під час обробки потоку, данні не зберігаються, а лише трансформуються, БД використовувати не має необхідності.

2.4 Структура масивів інформації

Оскільки потік має бути розподіленим між вузлами необхідно зберігати інформацію про наявні вузли, так званий “Service registry”. Для наших цілей буде достатньо масиву з адрес наявних сервісів

Висновок до розділу

У розділі з описом інформаційного забезпечення було описано вхідні дані комплексу задач: потік даних з повідомленнями у вигляді серіалізованих чисел, вихідні дані: потік даних з повідомленнями у вигляді серіалізованих результатів обробки та структура масивів інформації.

3 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Змістовна постановка задачі

Багато потокове оброблення даних – парадигма програмування, що поєднує в собі елементи паралельного програмування, подійно-орієнтованого програмування та реактивного програмування. Дозволяє підвищити пропускну здатність систем використовуючи декілька обчислювальних вузлів[11].

Одна з задач обробки потоку знаходження частих елементів, тобто таких елементів у яких, відносна частота входження в потік вище заданої. Це може використовуватись для в моніторингу мережевого трафіку чи дата майнингу[12]. Оскільки зазвичай об’єм даних в таких задачах величезний, важливу ролі відіграє час роботи та додаткове місце використане алгоритмом. Бажано, щоб такий алгоритм міг працювати в режимі реального-часу не потребуючи додаткового проходження по всьому масиву даних. В такому випадку можемо виділити критерії оцінювання алгоритму:

1. Загальний час обробки потоку
2. Час обробки в гіршому випадку
3. Додаткові місце використане

Оскільки обробка буде проходити розподіллено, також треба враховувати ресурси витрачені на комунікацію вузлів[11].

3.2 Математична постановка задачі

Є послідовність чисел

$$x = (x_0, \dots, x_N) \in \sum{}^*$$

В якій $\sum^*$ - абетка з n символів та дійсним числом θ у проміжку (0, 1]. Для задачі нашого масштабу можемо вважати дійсною нерівність[12]

$$N \gg n \gg \frac{1}{\theta}$$

Нехай існує функція $f_x(a)$, що відображає кількість входження a у x тоді задача зводиться до знаходження таких $x_i, i \in$ n для яких $\frac{f_x(x_i)}{N} > \theta$, ці елементи і будуть частими[12]

3.3 Обґрунтування методу розв'язання

Найпростішим методом розв'язання є підрахунок входження кожного елементу та збереження відповідності між елементом на кількістю у хеш-мапі. Очевидно, що в такому методі потребуватиметься не менше ніж n додаткового місця для збереження елементів у мапі. Такий метод може не задовольняти умовам оскільки можливі ситуації, коли n є занадто великим для збереження у пам'яті застосування. Також існує ризик колізій при використані хеш-мапи на занадто велику кількість елементів.

Інший алгоритм запропонований Річардом Карпом та Скоттом Шенкером[12] дозволяє вирішити проблему з використанням пам'яті але потребує більше часу для підрахунку. Алгоритм використовує перероблений. добре-відомий метод пошуку найбільш частого елементу. Суть алгоритму наведена нижче.

Нехай $x[1] \ldots . x[N]$ вхідна послідовність (потік)

K – сет з символів, на початку пустий

count – масив чисел індексований по K

тоді для кожного числа рахують входження до послідовності, якщо частота входження більша за тету елемент додається до K, якщо він вже наявний в K, то збільшує відповідне значення з count на одиницю. Якщо розмір K став більшим від $\frac{1}{\theta}$ то всім елементам з count декрементують значення, а негативні елементи видаляють. В кінці роботи у K будуть знаходитись символи, що можливо виявляться частими[12].

3.4 Опис методів розв’язання

Перший метод розв’язання можливо розподілити якщо розбити вхідний потік на m під-потоків, по одному на обчислювальний вузол, та групувати дані на вузлах відповідно до розбиття. Таким чином кожен вузол матиме інформацію про 1/m частину потоку, а головний вузол матиме інформацію про частіші надходження кожної з частин, і після кожного нового елементу матиме змогу підраховувати найбільш часті з потоку.

Другий метод так само можливо розподілити розбиваючи вхідний потік на під-потоки, але на відміну від першого методу у випадку з незліченним потоком даних можливі хибно-позитивні результати. Для запобігання яких необхідно час від часу перебирати сет К

В рамках дипломного проекту буде реалізовано перший метод розв’язання задачі, оскільки він має більшу точність.

Крок1 –проініціалізувати пусті хеш-мапи у Master та Worker’ах

Крок2 – Встановити відповідність між проміжками чисел та Worker, що буде їх обробляти.

Крок3 – Отримати число

Крок4 – Відправити його на обробку у відповідний Worker

Крок5 – Всередині Worker підвищити у хеш-мапі значення відповідне числу, та надіслати до Worker числа, для яких присутність більша граничного значення.

Крок6 – Всередині Master оновити значення хеш-мапи.

Висновок до розділу

В даному розділі було детально розібрано та порівняно різні методи розв’язання класичних задач з обробки потоків. Було описано варіанти розподіленої реалізації, та обґрунтовано вибір методу вирішення даної задачі.

4 ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

4.1 Засоби розробки

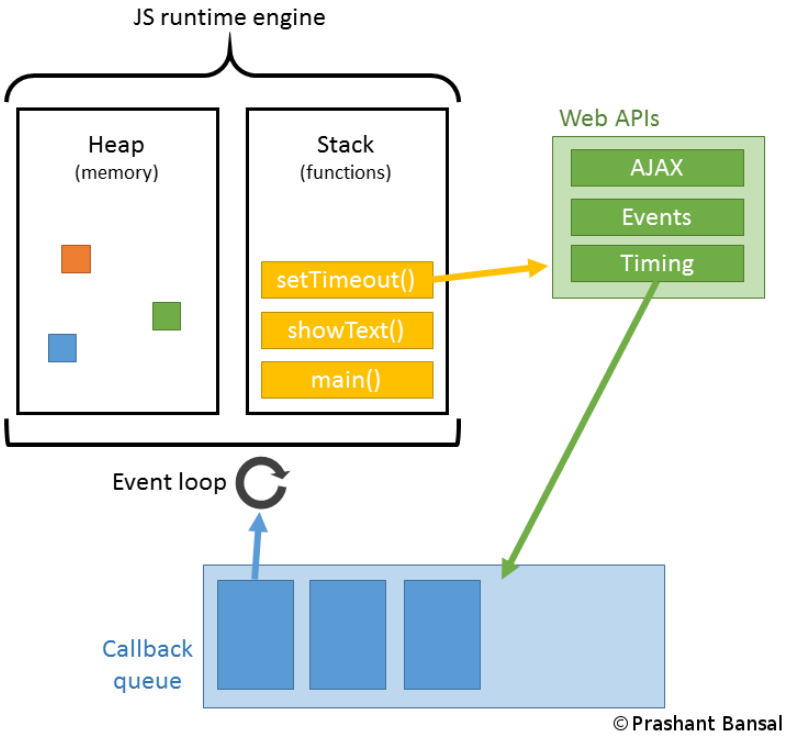
Мовою програмування було обрано Typescript, так як її використання додає типізацію до JavaScript та загалом покращує developer experience. Як протокол для передачі даних було обрано Rsocket - оскільки він бінарний та підтримує семантику реактивних потоків та надає готову реалізацію основних сутностей. Для серіалізації даних було обрано інструмент Protocol Buffer, що серіалізує повідомлення в бінарний формат для збільшення ефективності передачі даних. Для розгортання системи було обрано контейнеризаційний метод та Docker оскільки він дозволяє емулювати багато-вузлову роботу в рамках одної машини. Для моніторингу роботи системи було обрано комбінацію засобів Prometheus та Grafana – що дозволяють збирати та візуалізовувати данні.

JavaScript (JS) – асинхронна, одно-потокова, мова програмування з слабкою, динамічною типізацією. JS є гнучкою у використані мовою програмування, яка дає можливість використовувати об’єктно-орієнтовану, функціональну чи імперативну парадигми програмування. JS є мовою високого рівня, оскільки робота з пам’яттю виконується автоматично. Основним призначенням програм написаних використовуючи JavaScript є програмування веб-сторінок для відображення у браузері. Перевагами JavaScript є:

- Низький поріг входу
- Велика кількість бібліотек з відкритим кодом
- Активна спільнота

У JavaScript існує механізм «Цикл подій», що включає в деякому ступені нагадує паттерн Reactor. Цикл подій містить такі сутності як «Стек викликів» та «Черга функцій зворотного виклику» та “Зовнішній API”. Під час виконання коду, виклики потрапляють до стеку викликів та синхронно

оброблюються. При виклику асинхронного методу з зовнішнього API в нього передається функція зворотного виклику, котра буде викликана з результатом виконання методу. Зовнішній метод виконуватиметься в окремому потоці та по закінченню додасть виклик функцій зворотного



виклику у чергу. Коли стек викликів звільниться, та якщо черга функцій зовнішнього виклику не пуста, буде виконана одна з функцій з черги[13]. Схематичне зображення наведене на рисунку 4.1

Рисунок 4.1 - Схематичне зображення роботи циклу подій

Typescript мова програмування розроблена компанією Microsoft для полегшення розробки на JavaScript. Код написаний на Typescript транспайлюється у Javascript. Тому такі характеристики JavaScript'а як одно-потоковість та асинхронність передаються і Typescript'а. Головною причиною використання Typescript є додавання статичної типізації, що полегшує розуміння коду. Для додавання нових типів використовуються файли декларації. Оскільки Typescript транспайлюється у JavaScript, він зберігає звортню сумісність з JavaScript[14].

V8 – ядро для запуску JS коду, було розроблено командою Chromium для браузеру Google Chrome. Використовується у багатьох продуктах компанії. V8 замість інтерпретування, компілює JS код у машинний. Додатково згенерований код оптимізується під час виконання[15].

Node.js – платформа для виконання JS коду заснована на V8, яка дозволяє виконувати JS код за межами браузеру, наприклад на сервері. Оскільки JS є одно-потоковою мовою ефективність застосунку обмежена, тому прийнято писати серверні програми таким чином, щоб їх було можливо масштабувати горизонтально. Для цього сервер має бути без-становим, тобто не зберігати у пам’яті жодного контексту. Таким чином Node.js нав’язує використання розподіленої обробки. Також Node.js дає можливість використовувати дочірні потоки для паралелізації роботи[13].

Rsocket - бінарний протокол передачі данних прикладного рівня. Підтримує семантику реактивних потоків, має декілька моделей комунікації та може працювати незалежно від обраного протоколу транспортного рівня. Окрім підтримки реактивної семантики, важливими властивостями Rsocket є:

- Мультиплексація з’єднання, тобто навіть якщо використовується декілька логічних каналів, вони знаходитимуться всередині одного з’єднання
- Двох-направленість, тобто після відкриття з’єднання данні можуть бути надіслані як і на сервер, так і з серверу
- Можливість відновити перерване з’єднання

На разі Rsocket використовується компаніями Facebook та Alibaba Cloud[16-17]. Для транспортного рівня було обрано WebSocket - бінарний протокол побудований поверх TCP.

Protocol Buffer (Protobuf) - інструмент серіалізації повідомлень до байтового вигляду. Після узгодження структури вхідних, вихідних даних та RPC інтерфейсів пишуть відповідні .proto файли, з яких може бути згенерований код для роботи з такими структурами та інтерфейсами різними

мовами програмування[10]. В порівнянні з JSON серіалізацією Protobuf дає пришвидшення більше 5 разів у мовах де JSON не є нативним форматом, та більше 20% при використані у JavaScript, де JSON є вбудованим форматом[18].

Docker - засіб контейнеризації застосунків. Дозволяє виконувати різні компоненти системи у ізольованих середовищах. На відміну від віртуальних машин, контейнери не потребують фіксовану кількість ресурсів, а можуть динамічно її змінювати. Також у Docker вбудовано засоби для роботи з систем, що складаються з декількох контейнерів - Docker-compose налаштування звязку між контейнерами системи, та Docker-swarm для кластеризації застосунку[19].

Prometheus – засіб для моніторингу з відкритим вихідним кодом. З 2016 член Cloud Native Computing Foundation. Має гнучку архітектуру та дозволяє підключати різні засоби візуалізації. Використовує pull архітектуру для збору даних[23].

Grafana – засіб для візуалізації даних за часовими рядами з відкритим вихідним кодом. Дозволяє аналізувати дані за допомогою різних графіків та гнучких запитів[24].

4.2 Вимоги до технічного забезпечення

4.2.1 Загальні вимоги

Для коректної роботи систем технічні засоби мають відповідати наступним вимогам

- Сервер для Master вузлу з наступною конфігурацією
  1. 64-бітна операційна система
  2. Процесор з тактовою частотою не нижче 2 ГГц
  3. Об’єм оперативної пам’яті не нижчий від 8 ГБ
  4. Не менше 3ГБ вільного дискового простору
- Додаткові сервери для Worker вузлів

1. 64-бітна операційна система
2. Процесор з тактовою частотою не нижче 2 ГГц
3. Об’єм оперативної пам’яті не нижчий від 4 ГБ
4. Не менше 3ГБ вільного дискового простору

- Додатково на серверах має бути встановлене наступне програмне забезпечення:
  1. Засіб контейнеризації Docker версії 18, або вище
  2. Docker-compose версії 3, або вище
  3. Операційна система з ядром Linux kernel версії 3.10, або вище

4.2.2 Опис локальної обчислювальної мережі

Комунікація між користувачем та системою, а також між компонентами системи відбувається через мережеві інтерфейси. При наявності підключення до мережі інтернет, можливе використання комп’ютерів з зовнішніх мереж, інакше можливо лише використання комп’ютерів під’єднаних до однієї локальної мережі. Швидкість з’єднання має відповідати очікуваному навантаженню.

4.3 Архітектура програмного забезпечення

Програмне забезпечення матиме типову Master-slave архітектуру, в якій master – буде розбивати потік на під-потоки та агрегувати результати обчислень.

Структурну схему розгортання системи наведено у графічному матеріалі. Локальна мережа міститиме:

1. Веб-сервер, який містить в собі ServiceRegistry та Master та також можливі Worker
2. Веб-сервер, який складається лише з Worker сервісів.

Кількість Worker сервісів зображених на рисунку є прикладом, і може конфігуруватись при старті системи.

4.3.1 Діаграма класів

Структурну схему класів системи наведено у графічному матеріалі. На ній зображені сутності наявні в системі, та їх методи. Сутність ServiceRegistry відповідає за реєстр Worker сервісів. Та має методи для додання, видалення та отримання інформації про сервіс. Також в рамках ServiceRegistry відбувається перевірка стану Worker сервісів, та при необхідності видалення неактивних сервісів. Сутність Master відповідає за налаштування системи та роботу з користувацькими даними. Метод config задає приблизну кількість унікальних чисел у потоці, екстремуми та граничне значення частоти, це необхідно для більш ефективного розбиття потоку на під-потоки. Також Master має метод для прийому та обробки потоку даних. Цей метод приймає потік з чисел та повертає потік з частих елементів та їх відносних частот. Сутність Worker відображає Worker сервіси, що оброблюють під-потоки та повертають найбільш часті елементи з них. Також Worker сервіси мають методи для реєстрації сервісу у ServiceRegistry

4.3.2 Діаграма послідовності

Схема структурна послідовності роботи користувача з системою наведена у графічному матеріалі. Сутність User відображає користувача системи, що проводить попередню конфігурацію та використовує систему. Після того як Master було сконфігуровано, він може обролювати потік даних, розбиваючи його на під-потоки та направляючи у відповідний Worker сервіс з кластеру Worker сервісів. Після отримання результату обробки елементу, Master сервіс перевіряє найбільш часті числа задовільнення умовам, та в разі необхідності надсилає користувачу нові значення частих елементів.

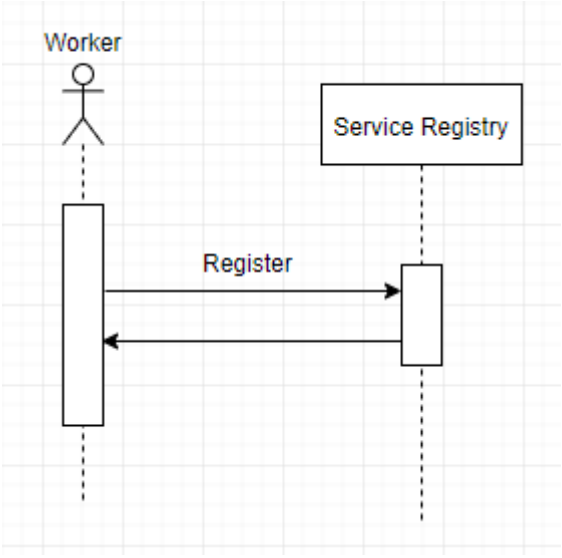


Рисунок 4.2 - Діаграма послідовності реєстрації сервісу в реєстрі

На рисунку 4.5 зображена схема структурна послідовності реєстрації сервісу в реєстрі. Сутність Worker сервіс який виконуватиме обробку інформації. Для початку його роботи, Worker сервіс має сповістити ServiceRegistry про своє існування. Після додання сервісу до реєстру він стане доступним Master сервісу для надіслання відповідного під-потоку.

4.3.3 Специфікація функцій

У таблиці 4.1 наведено основні функції системи та їх опис

Таблиця 4.1

Master.config	Задає кількість символів які будуть у вхідному потоку, мінімальне, максимальне значення з чисел потоку та шукану частоту
Master.process	Приймає потік з чисел та повертає потік з частими елементами
Master.getNumberFrequency	Повертає відносну частоту присутності числа у потоці.
Master.status	Повертає статус в якому знаходиться система

ServiceRegistry.addService	Реєструє новий сервіс
ServiceRegistry.getService	Повертає адресу сервісу
ServiceRegistry.deleteService	Видаляє сервіс з реєстру
ServiceRegistry.healthCheck	Перевіряє статус сервісу
Worker.register	Реєструє Worker у реєстрі
Worker.process	Оброблює під-потік надісланий до Worker

Висновок до розділу

В даному розділі було розглянуто програми, мови програмування, інструменти за допомогою яких було реалізовано систему, та їх роль у ній. Також наведено діаграми класів, розгортання та послідовності з детальним поясненням до них.

Було розглянуто загальні вимоги до технічних засобів та показано опис дій у модулях системи.

5 ТЕХНОЛОГІЧНИЙ РОЗДІЛ

5.1 Керівництво користувача

Для запуску системи користувач має перейти в директорію проекта та виконати команду docker-compose up -d, після чого будуть створені необхідні сервіси. Результат роботи зображений на рисунку 5.1

```
MacBook-Pro-Daniil:diplom daniildrobot$ docker-compose up -d
Creating network "diplom_default" with the default driver
Building master
Step 1/5 : FROM node:10.13.0
 ---> f09e7c96b6de
Step 2/5 : ADD . /opt/interactions
 ---> d8cf4db64256
Step 3/5 : WORKDIR /opt/interactions
 ---> Running in 67a1cc4645b9
Removing intermediate container 67a1cc4645b9
 ---> 7fc374d5482a
Step 4/5 : RUN npm install
 ---> Running in 740e93825926
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN master@1.0.0 No description
npm WARN master@1.0.0 No repository field.

up to date in 1.483s
found 0 vulnerabilities

Removing intermediate container 740e93825926
 ---> d1a69483c3d3
Step 5/5 : CMD ["npm", "start"]
 ---> Running in aa1a1aba9199
Removing intermediate container aa1a1aba9199
 ---> 56efad8ba2ec
Successfully built 56efad8ba2ec
Successfully tagged diplom_master:latest
WARNING: Image for service master was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
Building worker
Step 1/5 : FROM node:10.13.0
 ---> f09e7c96b6de
Step 2/5 : ADD . /opt/interactions
 ---> Using cache
 ---> d8cf4db64256
Step 3/5 : WORKDIR /opt/interactions
 ---> Using cache
 ---> 7fc374d5482a
Step 4/5 : RUN npm install
 ---> Using cache
 ---> d1a69483c3d3
Step 5/5 : CMD ["npm", "start"]
 ---> Using cache
 ---> 56efad8ba2ec
Successfully built 56efad8ba2ec
Successfully tagged diplom_worker:latest
WARNING: Image for service worker was built because it did not already exist. To rebuild this image you must use `docker-compose build` or `docker-compose up --build`.
Creating diplom_master_1 ... done
Creating diplom_worker_1 ... done
```

Рисунок 5.1 - Результат виконання команди розгортання системи

Коли створення сервісів завершиться, користувач має задати приблизну кількість унікальних чисел, мінімальне та максимальне число з потоку. Для цього необхідно відправвити HTTP запит до Master сервісу POST /api/v1/config. Результат виконання зображений на рисунку 5.2.

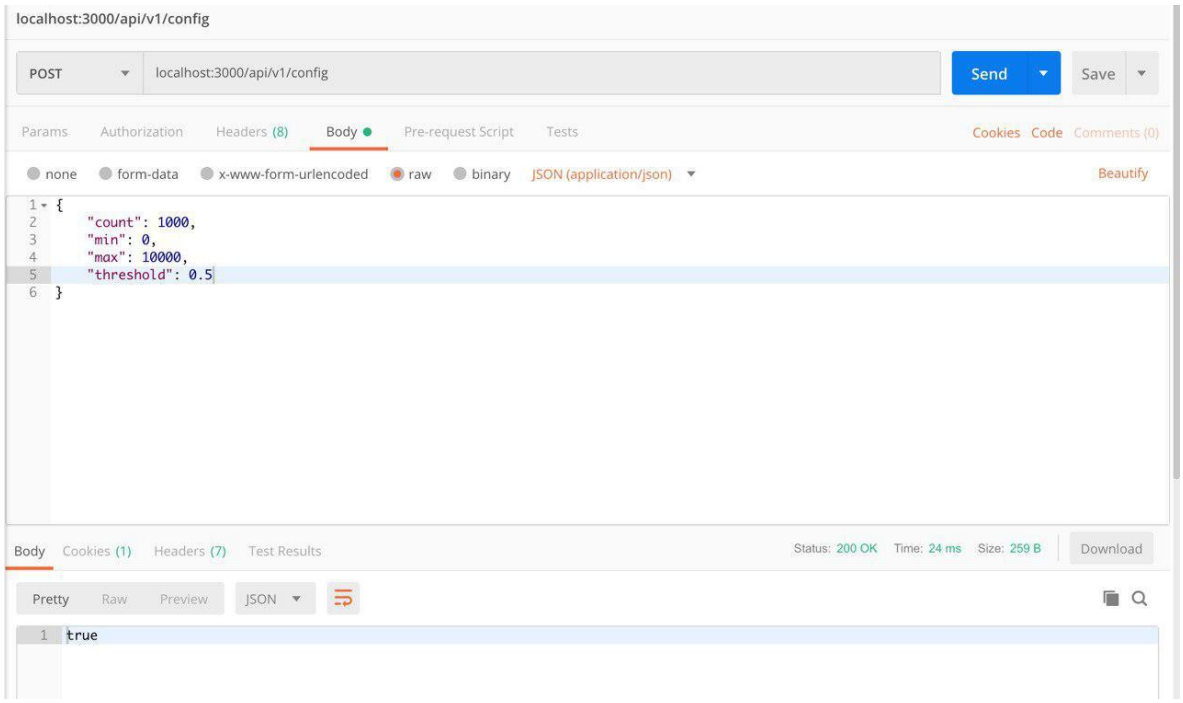


Рисунок 5.2 - Результат виконання запиту конфігурації

Після чого, користувач може надсилати данні до потоку через rsocket з'єднання /ws/v1/stream дані потоку мають бути серіалізовані використовуючи Protocol Buffers та мати структуру відповідну .proto файлам. Очікується, що користувач буде використовувати систему згідно SaaS моделі, та взаємодіятиме з API зі своєї системи. Без використання UI. При змінні частих чисел, чи їх частот відповідне повідомлення буде надіслано через те саме rsocket з'єднання. Результат відправки першого елементу потоку зображений на рисунку 5.3

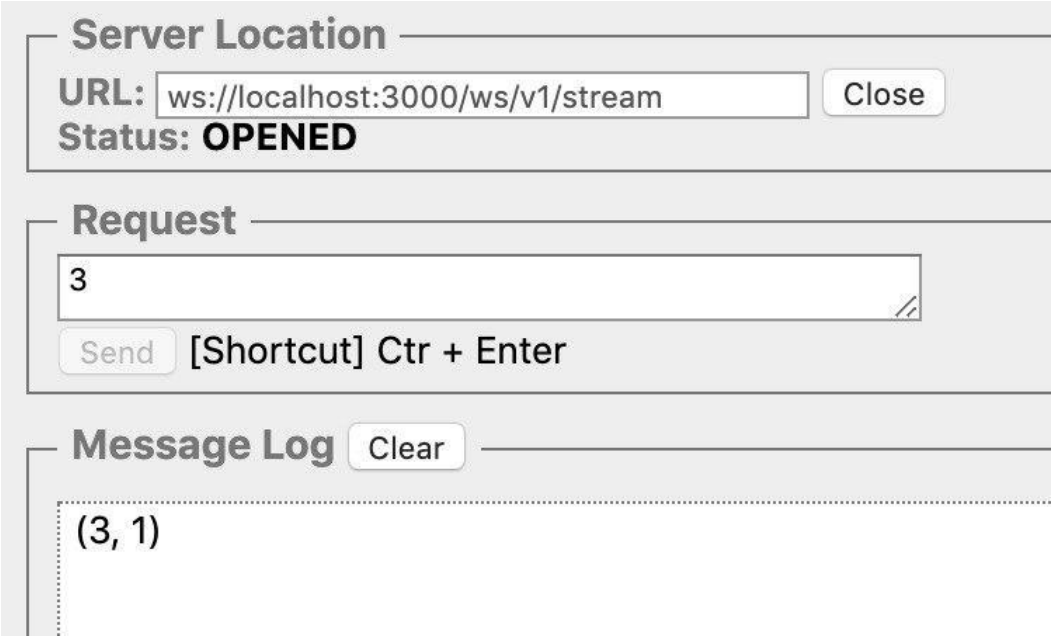


Рисунок 5.3 - Результат обробки першого елементу потоку

Для моніторингу кількості сервісів користувач може використати ServiceRegistry та HTTP метод GET /api/v1/registry. Результат виконання запиту зображений на рисунку 5.4

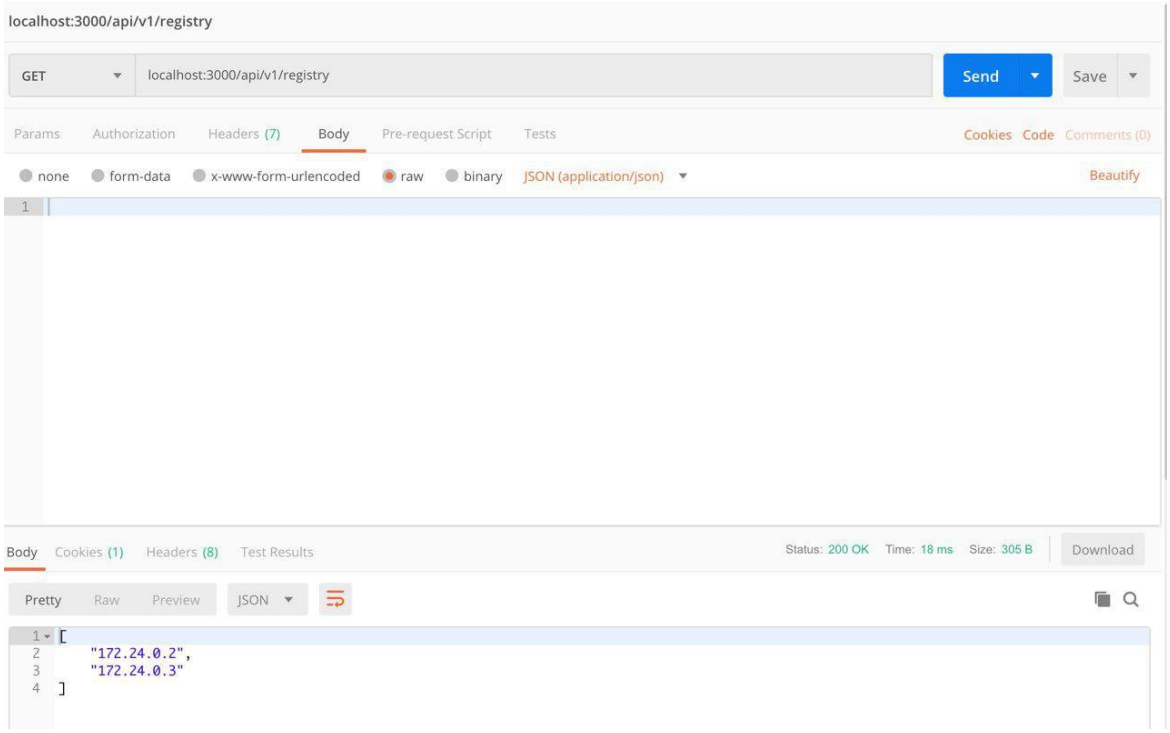


Рисунок 5.4 - Результат виконання запиту на реєстр

Також для отримання інформації про розгорнену систему можна використовувати docker cli, наприклад для отримання списку всіх

контейнерів та базові інформації про них достатньо виконати команду docker ps. Результат виконання якої зображено на рисунку 5.5

```
MacBook-Pro-Daniil:diplom daniildrobot$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
e8e1c0f05217        diplom_master       "npm run start:conta…"   15 seconds ago      Up 14 seconds       0.0.0.0:3000->3000/tcp   diplom_master_1
308eaa79ad04        diplom_worker       "npm run start:conta…"   15 seconds ago      Up 14 seconds       0.0.0.0:3001->3001/tcp   diplom_worker_1
MacBook-Pro-Daniil:diplom daniildrobot$
```

Рисунок 5.5 - Список запущених контейнерів

Для отриманні детальної інформації щодо конкретного контейнера docker inspect <container_id>. Фрагмент результату якої зображено на рисунку 5.6


```
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID": "96ba884833b6571193a4b55c2fe47ef0cb2fa8343530abe71a4f192905763f96",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
            "SandboxKey": "/var/run/docker/netns/96ba884833b6",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID": "",
            "Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "",
            "IPPrefixLen": 0,
            "IPv6Gateway": "",
            "MacAddress": "",
            "Networks": {
                "diplom_default": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": [
                        "master",
                        "e8e1c0f05217"
                    ],
                    "NetworkID": "f27871b861b7d9d1deb99ee54285b32dd6049a243a9036abd8eb200805c0ce2d"
                    "EndpointID": "",
                    "Gateway": "",
                    "IPAddress": "",
                    "IPPrefixLen": 0,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "",
                    "DriverOpts": null
                }
            }
        }
    }
]
MacBook-Pro-Daniil:diplom daniildrobot$
```


Рисунок 5.6 - Фрагмент мережевої інформації про контейнер

Для перегляду системних логів контейнера docker logs -f <container_id>. Результат виконання якої зображений на рисунку 5.7

```
MacBook-Pro-Daniil:diplom daniildrobot$ docker logs -f 517ccdf73444

> pum-content-service@1.0.0 start:container /opt/interactions
> npm run migrate ; npm start


> pum-content-service@1.0.0 migrate /opt/interactions
> npm run build ; npx typeorm migration:run -t=false


> pum-content-service@1.0.0 build /opt/interactions
> rm -rf ./dist/ ; tsc -p .

> pum-content-service@1.0.0 start /opt/interactions
> npm run build ; node dist/index.js


> pum-content-service@1.0.0 build /opt/interactions
> rm -rf ./dist/ ; tsc -p .
```

Рисунок 5.7 - Логи виконання одного з контейнерів

Для зупинення системи необхідно виконати команду docker-compose stop. Результат виконання якої наведено на рисунку 5.8.

```
MacBook-Pro-Daniil:diplom daniildrobot$ docker-compose stop
Stopping diplom_worker_1          ... done
Stopping diplom_master_1          ... done
```

Рисунок 5.8 - Результат зупинення контейнерів

Система надає можливість переглядати метрики використовуючи Prometheus та Grafana. Приклад візуалізації метрик зображено на рисунку 5.9

Рисунок 5.10 – Приклад візуалізації метрик

5.2 Випробування програмного продукту

5.2.1 Мета випробувань

Метою випробувань являється перевірка відповідності функцій комплексу задач розподіленої обробки реактивного потоку даних вимогам

технічного завдання.

5.2.2 Загальні положення

Випробування проводяться на основі наступних документів:

– ГОСТ 34.603–92. Інформаційна технологія. Види випробувань автоматизованих систем;
– ГОСТ РД 50-34.698-90. Автоматизовані системи вимог до змісту документів.

5.2.3 Результати випробувань

В результаті тестування була перевірена уся функціональність моделі. У наступних таблицях наведено перелік випробувань основних функціональних можливостей.

Таблиця 5.1 Розгортання системи

Мета тесту	Перевірка розгортання системи
Початковий стан моделі	Модель не активна
Вхідні дані	Команда з docker cli
Схема проведення тесту	Ввести команду docker-compose up -d
Очікуваний результат	Модель була розгорнута
Стан моделі після проведення випробувань	Модель очікує конфігурації

Таблиця 5.2 Конфігурування системи

Мета тесту	Перевірка можливості конфігурування системи
Початковий стан моделі	Модель очікує конфігурації
Вхідні дані	Кількість унікальних

	елементів, мінімальниий / максимальний елемент, граничне значення частоти
Схема проведення тесту	Відправити HTTP запит на POST «<master_host>/api/v1/config» з тілом запиту, що містить необхідні параметри
Очікуваний результат	Модель була зконфігурована
Стан моделі після проведення випробувань	Модель очікує вхідний потік даних

Таблиця 5.3 Отримання реєстру сервісів

Мета тесту	Перевірка можливості отримання реєстру сервісів
Початковий стан моделі	Модель розгорнута
Вхідні дані	HTTP запит
Схема проведення тесту	Відправити HTTP запит на GET «<master_host>/api/v1/registry»
Очікуваний результат	В відповіді на запит було повернуто реєстр сервісів
Стан моделі після проведення випробувань	Модель розгорнута

Таблиця 5.4 Додавання до реєстру

Мета тесту	Перевірка можливості додання сервісу до реєстру
Початковий стан моделі	Модель очікує конфігурацію
Вхідні дані	Адреса сервісу
Схема проведення тесту	Відправити HTTP запит на POST

	<master_host>/api/v1/registry
Очікуваний результат	Запит завершився нормально, сервіс було додано до реєстру
Стан моделі після проведення випробувань	Модель очікує конфігурацію

Таблиця 5.5 Ввод потоку даних

Мета тесту	Перевірка можливості вводу потоку даних
Початковий стан моделі	Модель очікує вхідний потік даних
Вхідні дані	Потік з цілих чисел

Продовження таблиці 5.5

Схема проведення тесту	Відкрити rsocket з’єднання та почати надсилати данні
Очікуваний результат	Модель почала оброблювати вхідний потік
Стан моделі після проведення випробувань	Модель оброблює вхідний потік

Висновок до розділу

В даному розділі було детального описано інструкцію користувача з використання даної системи, було перелічено список функціоналу, який присутній у системі, за допомогою знімків екрану та пояснень до них.

Було проведене тестування застосунку із детальним поясненням до нього.

ЗАГАЛЬНІ ВИСНОВКИ

ПЕРЕЛІК ПОСИЛАНЬ

1. Google Search Statistics [Електронний ресурс]: https://www.internetlivestats.com/google-search-statistics/
2. Nasdaq’s matching technology [Електронний ресурс]: https://business.nasdaq.com/market-tech/marketplaces/trading
3. Revisiting 1 Million Writes [Електронний ресурс]: https://medium.com/netflix-techblog/revisiting-1-million-writes-per-second-c191a84864cc
4. Netflix technology stack [Електронний ресурс]: https://stackshare.io/netflix/netflix
5. Reactive Streams Java [Електронний ресурс]: https://www.lightbend.com/blog/reactive-streams-java
6. Reactive Streams Specification [Електронний ресурс]: https://github.com/reactive-streams/reactive-streams-js
7. Distributed Systems: principles and paradigms [Електронний ресурс]: https://www.distributed-systems.net/index.php/books/distributed-systems-3rd-edition-2017
8. About Node.js [Електронний ресурс]: https://nodejs.org/en/about/
9. Peer-to-peer computing : principles and applications, ст 16
10. Protocol Buffers [Електронний ресурс]:https://developers.google.com/protocol-buffers/
11. Microservice Performance Patterns [Електронний ресурс]: https://dzone.com/articles/performance-patterns-in-microservices-based-integr
12. A simple Algoritgm for Finding Frequent Elements in Streams and Bags [Електронний ресурс]: https://www.cs.bgu.ac.il/~dinitz/Course/SS-12/Karp-frequent-el.pdf

13. Node.js Event Loop, Timers, and process.nextTick() [Електронний ресурс]: https://nodejs.org/de/docs/guides/event-loop-timers-and-nexttick/
14. Typescript Documentation [Електронний ресурс]: https://www.typescriptlang.org/docs/home.html
15. About V8 [Електронний ресурс]: https://v8.dev/docs
16. Rsocket [Електронний ресурс]: http://rsocket.io/
17. Rsocket, a New Application Network Protocol for Reactive Applications [Електронний ресурс]: https://www.infoq.com/news/2018/10/rsocket-facebook/
18. Beating JSON performance with Protobuf [Електронний ресурс]: https://auth0.com/blog/beating-json-performance-with-protobuf/
19. Docker [Електронний ресурс]: https://www.docker.com/
20. Apache Flink [Електронний ресурс]: https://flink.apache.org/
21. Apache Spark [Електронний ресурс]: https://spark.apache.org
22. Project Reactor [Електронний ресурс]: https://projectreactor.io/
23. Prometheus [Електронний ресурс]: https://prometheus.io/docs/introduction/overview/
24. Grafana [Електронний ресурс]: https://grafana.com/docs/

Додаток А

Тексти програмного коду

Комплексу задач з розподіленої обробки реактивного потоку даних на Node..js

(Найменування програми (документа))

DVD-R

(Вид носія даних)

(Обсяг програми (документа) , арк.,) Кб)

Київ – 2019 року

```
declare
module
'rsocket-
flowable' {

                  import { IPublisher, ISubscriber, IPartialSubscriber, ISubject,
                ISubscription } from 'rsocket-types'


                  export type CancelCallback = () => void;
                  export interface IPartialFutureSubscriber<T> {
                      readonly onComplete?: (value: T) => void,
                      readonly onError?: (error: Error) => void,
                      readonly onSubscribe?: (cancel: CancelCallback) => void,
                  }
                  export interface IFutureSubscriber<T> {
                      readonly onComplete: (value: T) => void,
                      readonly onError: (error: Error) => void,
                      readonly onSubscribe: (cancel: CancelCallback) => void,
                  }
                  export interface IFutureSubject<T> {
                      readonly onComplete: (value: T) => void,
                      readonly onError: (error: Error) => void,
                      readonly onSubscribe: (cancel?: CancelCallback) => void,
                  }


                  export class Single<T> {
                      static of<U>(value: U): Single<U>;

                      static error<U>(error: Error): Single<U>;

                      constructor(source: (subject: IFutureSubject<T>) => void);

                      subscribe(partialSubscriber?: IPartialFutureSubscriber<T>): void;

                      flatMap<R>(fn: (data: T) => Single<R>): Single<R>;

                      /**
                       * Return a new Single that resolves to the value of this Single applied
                to
                       * the given mapping function.
                       */
                      map<R>(fn: (data: T) => R): Single<R>;

                      then(successFn?: (data: T) => void, errorFn?: (error: Error) => void):
                void;
```

```
  }



  export class Flowable<T> implements IPublisher<T> {
      static just<U>(...values: Array<U>): Flowable<U>;

      static error<U>(error: Error): Flowable<U>;

      static never<U>(): Flowable<U>;

      constructor(source: (subscriber: ISubscriber<T>) => void, max?: number);

      subscribe(subscriberOrCallback?: (IPartialSubscriber<T> | ((e: T) =>
void)),): void;

      lift<R>(onSubscribeLift: (subscriber: ISubscriber<R>) =>
ISubscriber<T>,): Flowable<R>;

      map<R>(fn: (data: T) => R): Flowable<R>;

      take(toTake: number): Flowable<T>;
  }
}
declare
module
'rsocket-
types' {
    import { Single, Flowable } from "rsocket-flowable";

    export interface Responder<D, M> {
        /**
         * Fire and Forget interaction model of `ReactiveSocket`. The returned
         * Publisher resolves when the passed `payload` is successfully
handled.
         */
        fireAndForget(payload: Payload<D, M>): void,

        /**
```

```
         * Request-Response interaction model of `ReactiveSocket`. The
returned
         * Publisher resolves with the response.
         */
        requestResponse(payload: Payload<D, M>): Single<Payload<D, M>>,

        /**
         * Request-Stream interaction model of `ReactiveSocket`. The returned
         * Publisher returns values representing the response(s).
         */
        requestStream(payload: Payload<D, M>): Flowable<Payload<D, M>>,

        /**
         * Request-Channel interaction model of `ReactiveSocket`. The returned
         * Publisher returns values representing the response(s).
         */
        requestChannel(payloads: Flowable<Payload<D, M>>): Flowable<Payload<D,
M>>,

        /**
         * Metadata-Push interaction model of `ReactiveSocket`. The returned
Publisher
         * resolves when the passed `payload` is successfully handled.
         */
        metadataPush(payload: Payload<D, M>): Single<void>,
    }


    export interface PartialResponder<D, M> {
        readonly fireAndForget?: (payload: Payload<D, M>) => void,
        readonly requestResponse?: (payload: Payload<D, M>) =>
Single<Payload<D, M>>,
        readonly requestStream?: (payload: Payload<D, M>) =>
Flowable<Payload<D, M>>,
        readonly requestChannel?: (payloads: Flowable<Payload<D, M>>,) =>
Flowable<Payload<D, M>>,
        readonly metadataPush?: (payload: Payload<D, M>) => Single<void>,
    }

    /**
     * A contract providing different interaction models per the
[ReactiveSocket protocol]

(https://github.com/ReactiveSocket/reactivesocket/blob/master/Protocol.md).
    */
    export interface ReactiveSocket<D, M> extends Responder<D, M> {
        /**
         * Close this `ReactiveSocket` and the underlying transport
```

```
connection.
         */
        close(): void,

        /**
         * Returns a Flowable that immediately publishes the current
connection
         * status and thereafter updates as it changes. Once a connection is
in
         * the CLOSED or ERROR state, it may not be connected again.
         * Implementations must publish values per the comments on
ConnectionStatus.
         */
        connectionStatus(): Flowable<ConnectionStatus>,
    }

    /**
     * Represents a network connection with input/output used by a
ReactiveSocket to
     * send/receive data.
     */
    export interface DuplexConnection {
        /**
         * Send a single frame on the connection.
         */
        sendOne(frame: Frame): void,

        /**
         * Send all the `input` frames on this connection.
         *
         * Notes:
         * - Implementations must not cancel the subscription.
         * - Implementations must signal any errors by calling `onError` on
the
         *   `receive()` Publisher.
         */
        send(input: Flowable<Frame>): void,

        /**
         * Returns a stream of all `Frame`s received on this connection.
         *
         * Notes:
         * - Implementations must call `onComplete` if the underlying
connection is
         *   closed by the peer or by calling `close()`.
         * - Implementations must call `onError` if there are any errors
         *   sending/receiving frames.
         * - Implemenations may optionally support multi-cast receivers. Those
```

```
that do
         *   not should throw if `receive` is called more than once.
         */
        receive(): Flowable<Frame>,

        /**
         * Close the underlying connection, emitting `onComplete` on the
receive()
         * Publisher.
         */
        close(): void,

        /**
         * Open the underlying connection. Throws if the connection is already
in
         * the CLOSED or ERROR state.
         */
        connect(): void,

        /**
         * Returns a Flowable that immediately publishes the current
connection
         * status and thereafter updates as it changes. Once a connection is
in
         * the CLOSED or ERROR state, it may not be connected again.
         * Implementations must publish values per the comments on
ConnectionStatus.
         */
        connectionStatus(): Flowable<ConnectionStatus>,
    }

    /**
     * Describes the connection status of a ReactiveSocket/DuplexConnection.
     * - NOT_CONNECTED: no connection established or pending.
     * - CONNECTING: when `connect()` has been called but a connection is not
yet
     *   established.
     * - CONNECTED: when a connection is established.
     * - CLOSED: when the connection has been explicitly closed via `close()`.
     * - ERROR: when the connection has been closed for any other reason.
     */
    export type ConnectionStatus =
        | {kind: 'NOT_CONNECTED'}
        | {kind: 'CONNECTING'}
        | {kind: 'CONNECTED'}
        | {kind: 'CLOSED'}
        | {kind: 'ERROR', error: Error};
```

```
    export enum CONNECTION_STATUS {
        CLOSED,
        CONNECTED,
        CONNECTING,
        NOT_CONNECTED,
    }

    /**
     * A type that can be written to a buffer.
     */
    export type Encodable = string | Buffer | Uint8Array;

    /**
     * A single unit of data exchanged between the peers of a
`ReactiveSocket`.
     */
    export type Payload<D, M> = {
        data: D,
        metadata: M,
    };

    export type Frame =
        | CancelFrame
        | ErrorFrame
        | KeepAliveFrame
        | LeaseFrame
        | PayloadFrame
        | RequestChannelFrame
        | RequestFnfFrame
        | RequestNFrame
        | RequestResponseFrame
        | RequestStreamFrame
        | ResumeFrame
        | ResumeOkFrame
        | SetupFrame
        | UnsupportedFrame;

    export type FrameWithData = {
        data: Encodable,
        metadata: Encodable,
    };

    // prettier-ignore
    export type CancelFrame = {
        type: 0x09,
        flags: number,
        streamId: number,
    };
```

```
// prettier-ignore
export type ErrorFrame = {
    type: 0x0B,
    flags: number,
    code: number,
    message: string,
    streamId: number,
};
// prettier-ignore
export type KeepAliveFrame = {
    type: 0x03,
    flags: number,
    data: Encodable,
    lastReceivedPosition: number,
    streamId: 0,
};
// prettier-ignore
export type LeaseFrame = {
    type: 0x02,
    flags: number,
    ttl: number,
    requestCount: number,
    metadata: Encodable,
    streamId: 0,
};
// prettier-ignore
export type PayloadFrame = {
    type: 0x0A,
    flags: number,
    data: Encodable,
    metadata: Encodable,
    streamId: number,
};
// prettier-ignore
export type RequestChannelFrame = {
    type: 0x07,
    data: Encodable,
    metadata: Encodable,
    flags: number,
    requestN: number,
    streamId: number,
};
// prettier-ignore
export type RequestFnfFrame = {
    type: 0x05,
    data: Encodable,
    metadata: Encodable,
    flags: number,
```

```
    streamId: number,
};
// prettier-ignore
export type RequestNFrame = {
    type: 0x08,
    flags: number,
    requestN: number,
    streamId: number,
};
// prettier-ignore
export type RequestResponseFrame = {
    type: 0x04,
    data: Encodable,
    metadata: Encodable,
    flags: number,
    streamId: number,
};
// prettier-ignore
export type RequestStreamFrame = {
    type: 0x06,
    data: Encodable,
    metadata: Encodable,
    flags: number,
    requestN: number,
    streamId: number,
};
// prettier-ignore
export type ResumeFrame = {
    type: 0x0d,
    clientPosition: number,
    flags: number,
    majorVersion: number,
    minorVersion: number,
    resumeToken: Encodable,
    serverPosition: number,
    streamId: 0,
};
// prettier-ignore
export type ResumeOkFrame = {
    type: 0x0e,
    clientPosition: number,
    flags: number,
    streamId: 0,
};
// prettier-ignore
export type SetupFrame = {
    type: 0x01,
    data: Encodable,
```

```
    dataMimeType: string,
    flags: number,
    keepAlive: number,
    lifetime: number,
    metadata: Encodable,
    metadataMimeType: string,
    resumeToken: Encodable,
    streamId: 0,
    majorVersion: number,
    minorVersion: number,
};
// prettier-ignore
export type UnsupportedFrame = {
    type: 0x3f | 0x0c | 0x00,
    streamId: 0,
    flags: number,
};


export interface IPublisher<T> {
    subscribe(partialSubscriber?: IPartialSubscriber<T>): void,
    map<R>(fn: (data: T) => R): IPublisher<R>,
}

/**
 * An underlying source of values for a Publisher.
 */
export interface ISubscription {
    cancel(): void,
    request(n: number): void,
}

/**
 * A handler for values provided by a Publisher.
 */
export interface ISubscriber<T> {
    readonly onComplete: () => void,
    readonly onError: (error: Error) => void,
    readonly onNext: (value: T) => void,
    readonly onSubscribe: (subscription: ISubscription) => void,
}

/**
 * Similar to Subscriber, except that methods are optional. This is a
 * convenience type, allowing subscribers to only specify callbacks for
the
 * events they are interested in.
 */
```

```
    export interface IPartialSubscriber<T> {
        readonly onComplete?: () => void,
        readonly onError?: (error: Error) => void,
        readonly onNext?: (value: T) => void,
        readonly onSubscribe?: (subscription: ISubscription) => void,
    }

    /**
     * Similar to Subscriber, but without onSubscribe.
     */
    export interface ISubject<T> {
        readonly onComplete: () => void,
        readonly onError: (error: Error) => void,
        readonly onNext: (value: T) => void,
    }
}
// package:
io.rsocket.rp
c

// file: rsocket/options.proto


/* tslint:disable */


import * as jspb from "google-protobuf";
import * as google_protobuf_descriptor_pb from "google-
protobuf/google/protobuf/descriptor_pb";


export class RSocketMethodOptions extends jspb.Message {
    getFireAndForget(): boolean;
    setFireAndForget(value: boolean): void;




    serializeBinary(): Uint8Array;
    toObject(includeInstance?: boolean): RSocketMethodOptions.AsObject;
    static toObject(includeInstance: boolean, msg: RSocketMethodOptions):
RSocketMethodOptions.AsObject;
    static extensions: {[key: number]: jspb.ExtensionFieldInfo<jspb.Message>};
    static extensionsBinary: {[key: number]:
jspb.ExtensionFieldBinaryInfo<jspb.Message>};
    static serializeBinaryToWriter(message: RSocketMethodOptions, writer:
jspb.BinaryWriter): void;
    static deserializeBinary(bytes: Uint8Array): RSocketMethodOptions;
    static deserializeBinaryFromReader(message: RSocketMethodOptions, reader:
```

```
jspb.BinaryReader): RSocketMethodOptions;
}


export namespace RSocketMethodOptions {
    export type AsObject = {
        fireAndForget: boolean,
    }
}


export const options: jspb.ExtensionFieldInfo<RSocketMethodOptions>;

 /**
         * @fileoverview
         * @enhanceable
         * @suppress {messageConventions} JS Compiler reports an error if a
        variable or
         *     field starts with 'MSG_' and isn't a translatable message.
         * @public
         */
        // GENERATED CODE -- DO NOT EDIT!


        var jspb = require('google-protobuf');
        var goog = jspb;
        var global = Function('return this')();


        var google_protobuf_descriptor_pb = require('google-
        protobuf/google/protobuf/descriptor_pb.js');
        goog.exportSymbol('proto.io.rsocket.rpc.RSocketMethodOptions', null,
        global);
        goog.exportSymbol('proto.io.rsocket.rpc.options', null, global);


        /**
         * Generated by JsPbCodeGenerator.
         * @param {Array=} opt_data Optional initial data array, typically
        from a
         * server response, or constructed directly in Javascript. The array
        is used
         * in place and becomes part of the constructed object. It is not
        cloned.
         * If no data is provided, the constructed object will be empty, but
        still
         * valid.
         * @extends {jspb.Message}
```

```
 * @constructor
 */
proto.io.rsocket.rpc.RSocketMethodOptions = function(opt_data) {
  jspb.Message.initialize(this, opt_data, 0, -1, null, null);
};
goog.inherits(proto.io.rsocket.rpc.RSocketMethodOptions,
jspb.Message);
if (goog.DEBUG && !COMPILED) {
  proto.io.rsocket.rpc.RSocketMethodOptions.displayName =
'proto.io.rsocket.rpc.RSocketMethodOptions';
}
```

```
if (jspb.Message.GENERATE_TO_OBJECT) {
/**
 * Creates an object representation of this proto suitable for use
in Soy templates.
 * Field names that are reserved in JavaScript and will be renamed
to pb_name.
 * To access a reserved field use, foo.pb_<name>, eg,
foo.pb_default.
 * For the list of reserved names please see:
 *     com.google.apps.jspb.JsClassTemplate.JS_RESERVED_WORDS.
 * @param {boolean=} opt_includeInstance Whether to include the JSPB
instance
 *     for transitional soy proto support: http://goto/soy-param-
migration
 * @return {!Object}
 */
proto.io.rsocket.rpc.RSocketMethodOptions.prototype.toObject =
function(opt_includeInstance) {
  return
proto.io.rsocket.rpc.RSocketMethodOptions.toObject(opt_includeInstan
ce, this);
};
```

```
/**
 * Static version of the {@see toObject} method.
 * @param {boolean|undefined} includeInstance Whether to include the
JSPB
 *     instance for transitional soy proto support:
 *     http://goto/soy-param-migration
 * @param {!proto.io.rsocket.rpc.RSocketMethodOptions} msg The msg
```

```
instance to transform.
 * @return {!Object}
 * @suppress {unusedLocalVariables} f is only used for nested
messages
 */
proto.io.rsocket.rpc.RSocketMethodOptions.toObject =
function(includeInstance, msg) {
  var f, obj = {
    fireAndForget: jspb.Message.getFieldWithDefault(msg, 1, false)
  };


  if (includeInstance) {
    obj.$jspbMessageInstance = msg;
  }
  return obj;
};
}




/**
 * Deserializes binary data (in protobuf wire format).
 * @param {jspb.ByteSource} bytes The bytes to deserialize.
 * @return {!proto.io.rsocket.rpc.RSocketMethodOptions}
 */
proto.io.rsocket.rpc.RSocketMethodOptions.deserializeBinary =
function(bytes) {
  var reader = new jspb.BinaryReader(bytes);
  var msg = new proto.io.rsocket.rpc.RSocketMethodOptions;
  return
proto.io.rsocket.rpc.RSocketMethodOptions.deserializeBinaryFromReade
r(msg, reader);
};




/**
 * Deserializes binary data (in protobuf wire format) from the
 * given reader into the given message object.
 * @param {!proto.io.rsocket.rpc.RSocketMethodOptions} msg The
message object to deserialize into.
 * @param {!jspb.BinaryReader} reader The BinaryReader to use.
 * @return {!proto.io.rsocket.rpc.RSocketMethodOptions}
 */
proto.io.rsocket.rpc.RSocketMethodOptions.deserializeBinaryFromReade
```

```
r = function(msg, reader) {
  while (reader.nextField()) {
    if (reader.isEndGroup()) {
      break;
    }
    var field = reader.getFieldNumber();
    switch (field) {
    case 1:
      var value = /** @type {boolean} */ (reader.readBool());
      msg.setFireAndForget(value);
      break;
    default:
      reader.skipField();
      break;
    }
  }
  return msg;
};




/**
 * Serializes the message to binary data (in protobuf wire format).
 * @return {!Uint8Array}
 */
proto.io.rsocket.rpc.RSocketMethodOptions.prototype.serializeBinary
= function() {
  var writer = new jspb.BinaryWriter();

proto.io.rsocket.rpc.RSocketMethodOptions.serializeBinaryToWriter(th
is, writer);
  return writer.getResultBuffer();
};




/**
 * Serializes the given message to binary data (in protobuf wire
 * format), writing to the given BinaryWriter.
 * @param {!proto.io.rsocket.rpc.RSocketMethodOptions} message
 * @param {!jspb.BinaryWriter} writer
 * @suppress {unusedLocalVariables} f is only used for nested
messages
 */
proto.io.rsocket.rpc.RSocketMethodOptions.serializeBinaryToWriter =
function(message, writer) {
```

```
  var f = undefined;
  f = message.getFireAndForget();
  if (f) {
    writer.writeBool(
      1,
      f
    );
  }
};



/**
 * optional bool fire_and_forget = 1;
 * Note that Boolean fields may be set to 0/1 when serialized from a
Java server.
 * You should avoid comparisons like {@code val === true/false} in
those cases.
 * @return {boolean}
 */
proto.io.rsocket.rpc.RSocketMethodOptions.prototype.getFireAndForget
= function() {
  return /** @type {boolean} */
(jspb.Message.getFieldWithDefault(this, 1, false));
};



/** @param {boolean} value */
proto.io.rsocket.rpc.RSocketMethodOptions.prototype.setFireAndForget
= function(value) {
  jspb.Message.setProto3BooleanField(this, 1, value);
};



/**
 * A tuple of {field number, class constructor} for the extension
 * field named `options`.
 * @type
{!jspb.ExtensionFieldInfo<!proto.io.rsocket.rpc.RSocketMethodOptions
>}
 */
```

```
proto.io.rsocket.rpc.options = new jspb.ExtensionFieldInfo(
    1057,
    {options: 0},
    proto.io.rsocket.rpc.RSocketMethodOptions,
     /** @type {?function((boolean|undefined),!jspb.Message=):
!Object} */ (
         proto.io.rsocket.rpc.RSocketMethodOptions.toObject),
    0);


google_protobuf_descriptor_pb.MethodOptions.extensionsBinary[1057] =
new jspb.ExtensionFieldBinaryInfo(
    proto.io.rsocket.rpc.options,
    jspb.BinaryReader.prototype.readMessage,
    jspb.BinaryWriter.prototype.writeMessage,

proto.io.rsocket.rpc.RSocketMethodOptions.serializeBinaryToWriter,

proto.io.rsocket.rpc.RSocketMethodOptions.deserializeBinaryFromReade
r,
    false);
// This registers the extension field with the extended class, so
that
// toObject() will function correctly.
google_protobuf_descriptor_pb.MethodOptions.extensions[1057] =
proto.io.rsocket.rpc.options;


goog.object.extend(exports, proto.io.rsocket.rpc);
```

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
Кафедра автоматизованих систем обробки інформації та управління

УЗГОДЖЕНО
Керівник проекту
_____ *Олійник Ю.О.*
(підпис) (ініціали, прізвище)
“15” квітня 2019 р.

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ *О.А. Павлов*
(підпис) (ініціали, прізвище)
“16” квітня 2019 р.

Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js

ТЕХНІЧНЕ ЗАВДАННЯ

Шифр *ДП ІС-5106.1181-с.ТЗ*

на 10 сторінках

Київ – 2019 року

ЗМІСТ

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Повне найменування системи та її умовне позначення

Повна назва системи: *Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js*

1.2 Найменування організації-замовника та організацій-учасників робіт

Генеральним замовником проекту являється кафедра Автоматизованих систем обробки інформації та управління НТУУ "КПІ". Представником замовника є Олійник Юрій Олександрович.

Розробником системи є студент групи ІС-51 факультету інформатики та обчислювальної техніки НТУУ «КПІ ім. Ігоря Сікорського» Дробот Даніїл Борисович

1.3 Перелік документів, на підставі яких створюється система

При розробці системи і створення проектно-експлуатаційної документації Виконавець повинен керуватися вимогами наступних нормативних документів:

- ДСТУ 19.201-78. Технічне завдання. Вимоги до змісту і оформлення;
- ДСТУ 34.601-90. Комплекс стандартів на автоматизовані системи. Автоматизовані системи. Стадії створення;
- ДСТУ 34.201-89. Інформаційні технології. Комплекс стандартів на автоматизовані системи. Види, комплексність і позначення документів при створенні автоматизованих систем.

1.4 Планові терміни початку і закінчення роботи зі створення системи

Плановий термін початку роботи над створенням системи підтримки користувачів з використанням сентиментального аналізу даних – 15 квітня 2019 рік.

Плановий термін по закінченню роботи над створенням системи формування асортименту торгівельної організації – не пізніше 19 травня 2019 року.

2 ПРИЗНАЧЕННЯ І ЦІЛІ СТВОРЕННЯ СЕРВІСУ

2.1 Призначення розробки

Призначенням розробки є розподілення, обробка та агрегація потоків даних.

2.2 Цілі створення застосування

Цілями розробки є збільшення ефективності та полегшення розробки алгоритмів розподіленої обробки потоків за рахунок:
- Покращення масштабування за рахунок використання горизонтального та вертикального масштабування
- Полегшення розробки за рахунок розробки інтуїтивного прикладного інтерфейсу для обробки потоку.

Для досягнення поставлених цілей необхідно вирішити наступні задачі:

- розробити засіб для реєстрації вузлів системи;
- розробити засіб розподілення потоку на під-потоки;
- розробити засіб для обробки потоку;
- розробити засіб агрегації вихідних під-потоків у єдиний потік;

3 ХАРАКТЕРИСТИКА ОБ'ЄКТА АВТОМАТИЗАЦІЇ

Працювати з застосуванням можуть розробники ПЗ.

Об’єктом автоматизації є процес розподілення даних, що потребують обробки.

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

Застосування має створювати умови для підтримки комунікації і повинне задовольняти потреби користувачів. Застосування має виконувати наступні функції:

- Система повинна надати користувачу можливість задати конфігурацію
- Система повинна надати користувачу завантажити дані у вигляді потоку.
- Система повинна надати користувачу можливість переглянути результати обробки.
- Система повинна надати користувачу переглянути актуальний стан її компонентів.

4.2 Вимоги до надійності

Програма повинна зберігати працездатність і забезпечувати відновлення своїх функцій при виникненні наступних позаштатних ситуацій:

- при помилках в роботі апаратних засобів (крім носіїв даних і програм).

Програмний продукт повинен поєднувати надійність та функціональність. У разі виникнення аварійних ситуацій необхідно сповіщати користувача та надавати інструкцію для подальших дій. Будь-які аварійні ситуації мають бути задокументовані у звіті, який при необхідності надсилається розробнику для визначення причини збою в роботі та усуненні помилок, які могли привести до нестабільної роботи програмного продукту.

4.3 Вимоги до складу і параметрів технічних засобів

Склад, структура і способи організації даних в системі повинні бути визначені на етапі технічного проектування.

Структура технічних засобів визначається виходячи із можливості їх забезпечити виконання встановлених операцій процесу технічного обслуговування.

Для правильної роботи розробленої системи до складу технічних засобів повинен входити комп’ютер, що має конфігурацію наведену нижче:

- Сервер для Master вузлу з наступною конфігурацією
  1. 64-бітна операційна система
  2. Процесор з тактовою частотою не нижче 2 ГГц
  3. Об’єм оперативної пам’яті не нижчий від 8 ГБ
  4. Не менше 3ГБ вільного дискового простору
- Додаткові сервери для Worker вузлів
  1. 64-бітна операційна система
  2. Процесор з тактовою частотою не нижче 2 ГГц
  3. Об’єм оперативної пам’яті не нижчий від 4 ГБ
  4. Не менше 3ГБ вільного дискового простору
- Додатково на серверах має бути встановлене наступне програмне забезпечення:
  1. Засіб контейнеризації Docker версії 18, або вище
  2. Docker-compose версії 3, або вище
  3. Операційна система з ядром Linux kernel версії 3.10, або вище

5 СТАДІЇ І ЕТАПИ РОЗРОБКИ

Основні етапи виконання робіт з розробки системи ведення наукової роботи.

№ п/п	Назва етапу роботи	Термін виконання етапу	Результат виконання
1.	Підготовка технічного завдання на розробку програмного продукту	07.02.2019	
2.	Розробка сценарію роботи	12.02.2019	
3.	Технічне проектування – функціональність, модулі, задачі, цілі тощо	20.02.2019	
4.	Узгодження з керівником інтерфейсу користувача	02.03.2019	
5.	Розробка інформаційного забезпечення	17.03.2019	
6.	Розробка програмного забезпечення	29.03.2019	
7.	Налагодження програми	13.04.2019	
8.	Тестування програми	27.04.2019	
9.	Здача готового програмного продукту замовнику	12.05.2019	

6 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

6.1 Види випробувань

Для контролю правильності роботи програмного забезпечення буде проведено модульне тестування (Unit Test). В ході тестування буде проведено випробування основних елементів системи, які представлені бізнес-логікою.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
Кафедра автоматизованих систем обробки інформації та управління

УЗГОДЖЕНО

Керівник проекту

_____ *Ю.О. Олійник*
(підпис) (ініціали, прізвище)

“16” травня 2019 р.

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ *О.А.Павлов*
(підпис) (ініціали, прізвище)

“17” травня 2019 р.

КОМПЛЕКС ЗАДАЧ З РОЗПОДІЛЕНОЇ ОБРОБКИ РЕАКТИВНОГО ПОТОКУ ДАНИХ НА NODE.JS

ПРОГРАМА ТА МЕТОДИКА ВИПРОБУВАНЬ

Шифр *ДП ІС-5106.1181-с.ПМВ*

на 11 сторінках

Київ – 2019 року

ЗМІСТ

1 ОБ'ЄКТ ВИПРОБУВАННЯ

1.1 Найменування програми

Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js

1.2 Область застосування

Автоматизація розподілення обробки реактивного потоку даних на Node.js

1.3 Умовне позначення програми

Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js

2 МЕТА ВИПРОБУВАНЬ

Мета проведення випробувань – перевірка відповідних характеристик розробленої програми (програмного вироби) функціональним і окремим іншим видам вимог, викладених в документі технічного завдання.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Таблиця 3.1

Варіант використання	Функціональна вимога	Пріоритет
Конфігурування системи	1. Система надає можливість задати конфігурацію	Високий
Завантаження даних	2. Система надає можливість завантажити дані у вигляді потоку	Високий
Перегляд результатів	2.1. Система надає можливість переглянути результат обробки	Високий
Моніторинг системи	3. Система дає можливість переглянути актуальний стан її компонентів	Високий

3.1.1 Вимоги до складу виконуваних функцій

Таблиця 3.2

Master.config	Задає кількість символів які будуть у вхідному потоку, мінімальне, максимальне значення з чисел потоку та шукану частоту
Master.process	Приймає потік з чисел та повертає потік з частими елементами

Продовження таблиці 3.2

Master.getNumberFrequency	Повертає відносну частоту присутності числа у потоці.
Master.status	Повертає статус в якому знаходиться система
ServiceRegistry.addService	Реєструє новий сервіс
ServiceRegistry.getService	Повертає адресу сервісу
ServiceRegistry.deleteService	Видаляє сервіс з реєстру
ServiceRegistry.healthCheck	Перевіряє статус сервісу
Worker.register	Реєструє Worker у реєстрі
Worker.process	Оброблює під-потік надісланий до Worker

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

Програмний продукти розробляється на основі Технічного Завдання.

Випробування проводяться на основі наступних документів:

– ГОСТ 34.603–92. Інформаційна технологія. Види випробувань автоматизованих систем;
– ГОСТ РД 50-34.698-90. Автоматизовані системи вимог до змісту документів.

5 СКЛАД І ПОРЯДОК ВИПРОБУВАНЬ

Умови проведення випробувань: програма з готовими модулями та зв’язками між ними.

Умови початку та завершення окремих етапів тестування: тестування кожних елементів програми має відбуватися з урахуванням всіх можливих виключних ситуацій в залежності від функціональних можливостей програмного продукту.

Обмеження щодо умов проведення тестування: тестування має проводитися в рамках функціонального апарату програмного забезпечення.

Вимоги до технічного обслуговування системи: система має бути з ядром Linux kernel 3.10 або вище, та мати встановлений засіб контейнеризації Docker та утиліту Docker-compose

Міри, забезпечуючі безпеку та безаварійність проведення тестування: тестування системи не може визвати аварійних ситуацій.

Порядок взаємодій організацій, які беруть участь у тестуванні: тестування проводить один студент КПІ групи ІС-51 Дробот Даніїл Борисович

6 МЕТОДИ ВИПРОБУВАНЬ

У таблицях 6.1 – 6.5 наведено методи випробувань системи

Таблиця 6.1 Розгортання системи

Мета тесту	Перевірка розгортання системи
Початковий стан моделі	Модель не активна
Вхідні дані	Команда з docker cli
Схема проведення тесту	Ввести команду docker-compose up -d
Очікуваний результат	Модель була розгорнута
Стан моделі після проведення випробувань	Модель очікує конфігурації

Таблиця 6.2 Конфігурування системи

Мета тесту	Перевірка можливості конфігурування системи
Початковий стан моделі	Модель очікує конфігурації
Вхідні дані	Кількість унікальних елементів, мінімальниий / максимальний елемент, граничне значення частоти
Схема проведення тесту	Відправити HTTP запит на POST «<master_host>/api/v1/config» з тілом запиту, що містить необхідні параметри
Очікуваний результат	Модель була зконфігурована

Стан моделі після проведення випробувань	Модель очікує вхідний потік даних

Таблиця 6.3 Отримання реєстру сервісів

Мета тесту	Перевірка можливості отримання реєстру сервісів
Початковий стан моделі	Модель розгорнута
Вхідні дані	HTTP запит
Схема проведення тесту	Відправити HTTP запит на GET «<master_host>/api/v1/registry»
Очікуваний результат	В відповіді на запит було повернуто реєстр сервісів
Стан моделі після проведення випробувань	Модель розгорнута

Таблиця 6.4 Додавання до реєстру

Мета тесту	Перевірка можливості додання сервісу до реєстру
Початковий стан моделі	Модель очікує конфігурацію
Вхідні дані	Адреса сервісу
Схема проведення тесту	Відправити HTTP запит на POST <master_host>/api/v1/registry
Очікуваний результат	Запит завершився нормально, сервіс було додано до реєстру
Стан моделі після проведення випробувань	Модель очікує конфігурацію

Таблиця 6.5 Ввод потоку даних

Мета тесту	Перевірка можливості вводу потоку даних
Початковий стан моделі	Модель очікує вхідний потік даних
Вхідні дані	Потік з цілих чисел
Схема проведення тесту	Відкрити rsocket з’єднання та почати надсилати данні
Очікуваний результат	Модель почала оброблювати вхідний потік
Стан моделі після проведення випробувань	Модель оброблює вхідний потік

Графічний матеріал
до дипломного проекту

на тему: Комплекс задач з розподіленої обробки реактивного потоку даних на Node.js

Київ – 2019 року