

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Розрахунково-графічна робота

на тему: «Стая дронів»

з предмету «Проектування розподілених систем»

Виконав:

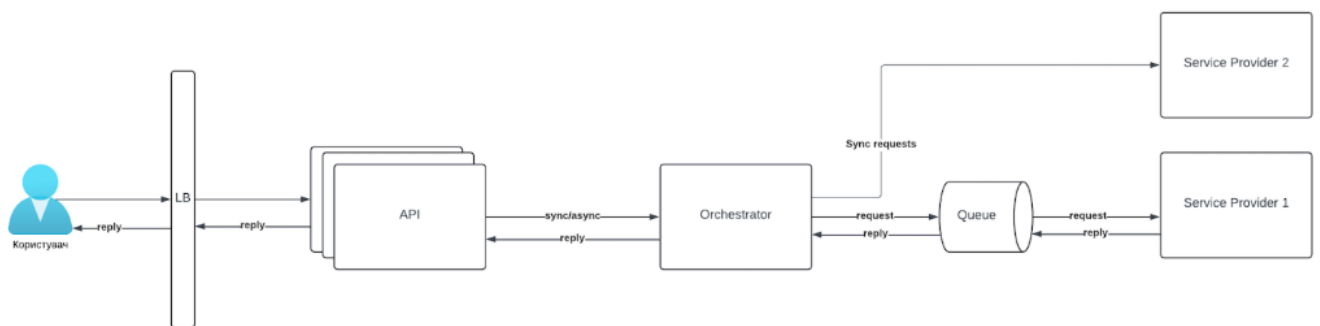
студент групи ІМ-31мн

Грибенко Єгор

Київ 2024

## Завдання:

- Реалізувати патерн оркестратор який буде керувати процесом розрахунку мат.моделі
- Побудувати математичну модель системи масового обслуговування (стая дронів) і розрахувати вплив кількості Постачальників сервісу на швидкість обробки завдань
- Зробити опис системи



## Хід роботи:

В роботі я вирішив реалізувати модель стаї дронів, і зробити систему, що буде оркеструвати обчислення поведінки цієї стаї.

Маємо наступні елементи проекту:

- Балансування навантаження над апі
- Кілька реплік апі
- Оркестратор
- Черга RabbitMQ, Асинхронний сервіс провайдер 1
- Синхронний сервіс провайдер 2, та Redis

В роботі було використано балансування навантаження перед апі, засобами docker:

```
9  ports:
10    - "8080:8080"
11    - "8081:8081"
12  depends_on:
13    - api
14  api:
15    image: ${DOCKER_REGISTRY-}api
16    build:
17      context: .
18      dockerfile: API/Dockerfile
19  deploy:
20    mode: replicated
21    replicas: 3
22  orchestrator:
23    container_name: orchestrator
24    image: ${DOCKER_REGISTRY-}orchestrator
25    build:
```

Балансування навантаження між оркестратором та асинхронними сервісами здійснюється через чергу RabbitMQ (вона вирішує, який з підписників забере наступну задачу).

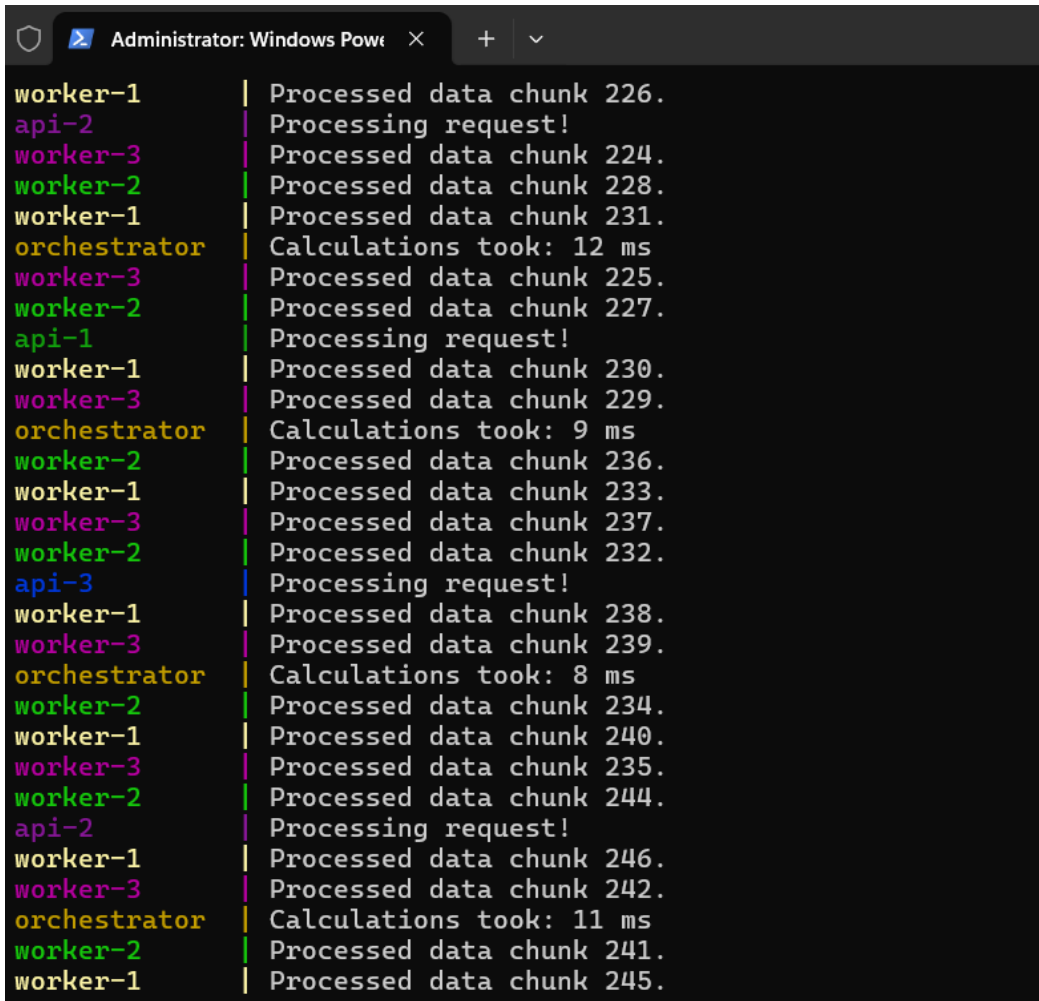
Черга, насправді, є двосторонньою, в яку з однієї сторони кладуться реквести на обчислення, а з іншої – повертаються назад результати обчислень.

## Весь docker-compose.yml:

```
name: RGR
services:
  api-lb:
    image: nginx:latest
    container_name: api-lb
    volumes:
      - ./API/nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "8080:8080"
      - "8081:8081"
    depends_on:
      - api
  api:
    image: ${DOCKER_REGISTRY-}api
    build:
      context: .
      dockerfile: API/Dockerfile
    deploy:
      mode: replicated
      replicas: 3
  orchestrator:
    container_name: orchestrator
    image: ${DOCKER_REGISTRY-}orchestrator
    build:
      context: .
      dockerfile: Orchestrator/Dockerfile
    depends_on:
      - rabbitmq
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: rabbitmquser
      RABBITMQ_DEFAULT_PASS: rabbitmqpassword
  worker:
    image: ${DOCKER_REGISTRY-}worker
    build:
      context: .
      dockerfile: Worker/Dockerfile
    deploy:
      mode: replicated
      replicas: 3
    environment:
      - "RMQ_HOST=rabbitmq"
      - "RMQ_PORT=5672"
      - "RMQ_USERNAME=rabbitmquser"
      - "RMQ_PASSWORD=rabbitmqpassword"
    depends_on:
      - rabbitmq
  workersync:
    image: ${DOCKER_REGISTRY-}workersync
    build:
      context: .
      dockerfile: WorkerSync/Dockerfile
    ports:
      - "5000:8080"
      - "5001:8081"
    depends_on:
      - redis
  redis:
    image: redis:latest
    container_name: redis-container
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    restart: unless-stopped
volumes:
  redis-data:
```

Тепер варто перевірити роботу балансування навантаження.

Бачимо, що балансування навантаження відбувається успішно, як на апі, так і на сервіс провайдерах:



```
Administrator: Windows Powe  x  +  v
worker-1 | Processed data chunk 226.
api-2    | Processing request!
worker-3 | Processed data chunk 224.
worker-2 | Processed data chunk 228.
worker-1 | Processed data chunk 231.
orchestrator | Calculations took: 12 ms
worker-3 | Processed data chunk 225.
worker-2 | Processed data chunk 227.
api-1    | Processing request!
worker-1 | Processed data chunk 230.
worker-3 | Processed data chunk 229.
orchestrator | Calculations took: 9 ms
worker-2 | Processed data chunk 236.
worker-1 | Processed data chunk 233.
worker-3 | Processed data chunk 237.
worker-2 | Processed data chunk 232.
api-3    | Processing request!
worker-1 | Processed data chunk 238.
worker-3 | Processed data chunk 239.
orchestrator | Calculations took: 8 ms
worker-2 | Processed data chunk 234.
worker-1 | Processed data chunk 240.
worker-3 | Processed data chunk 235.
worker-2 | Processed data chunk 244.
api-2    | Processing request!
worker-1 | Processed data chunk 246.
worker-3 | Processed data chunk 242.
orchestrator | Calculations took: 11 ms
worker-2 | Processed data chunk 241.
worker-1 | Processed data chunk 245.
```

Отже, в роботі були реалізовані як синхронні, так і асинхронні сервіси.

Оркестратор керує виконанням загальної задачі, яка складається з обрахунку поведінки дронів, і збереженню стану у сховище. Управління сховищем відбувається за допомогою синхронного сервісу.

Виконання асинхронних задач виконується за допомогою черги RabbitMQ, а синхронний запит в проєкті виконується для зв'язку з ще одним сервіс провайдером для збереження даних у Redis.

Синхронний запит збереження до сховища Redis виконується наступним чином:

```
35
36 app.MapPost("/save", ([FromBody] UpdateRequest request) =>
37 {
38     var dataRaw = request.data;
39
40     var batch = db.CreateBatch();
41
42     batch.SetStringAsync("data", request.data);
43     batch.SetStringAsync("target", request.target);
44     batch.SetStringAsync("droneCount", request.droneCount);
45
46     batch.Execute();
47     Console.WriteLine($"Saved data to storage.");
48 })
49 .WithName("save")
50 .WithOpenApi();
51
```

Запит виконується з тіла оркестратора:

```
int requestCounter = 0;
object requestLock = new();

app.MapPost("/process", async ([FromBody] UpdateRequest input) =>
{
    var content = new StringContent(JsonConvert.SerializeObject(input), Encoding.UTF8, "application/json");
    var response = syncServiceClient.PostAsync("/save", content);

    int totalDrones = input.droneCount;
    var ranges = SplitRange(0, totalDrones, 8);

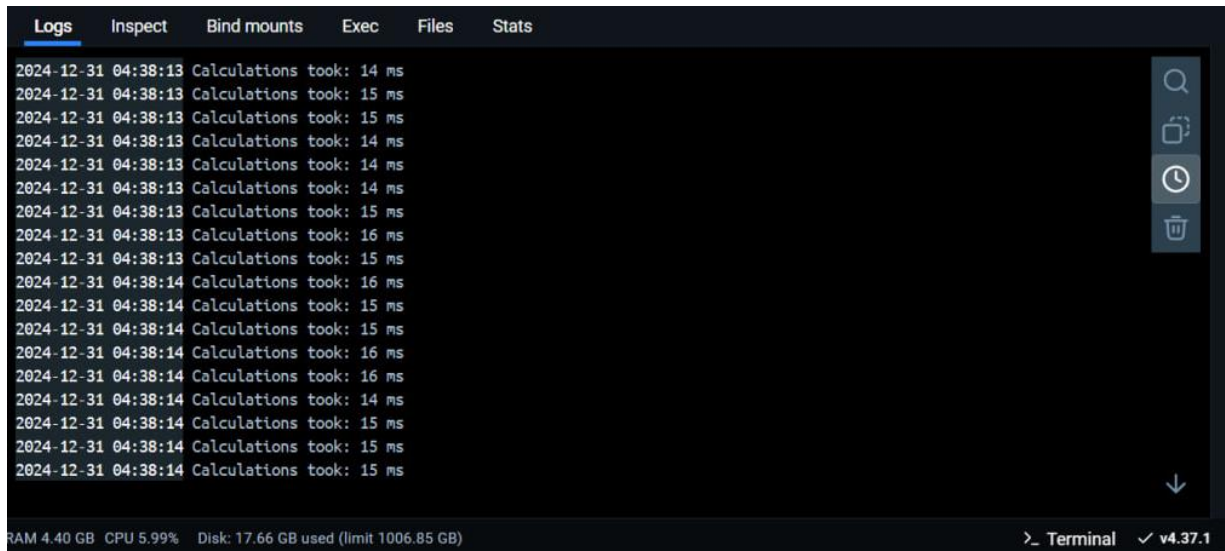
    var dataRaw = input.data;
    var sw = new Stopwatch();
    sw.Start();

    var results = new List<(int, SerializableVector)>();
}
```

Тут слід зазначити що назва методу PostAsync – не тому, що зв'язок асинхронний, а тому що запит є неблокуючим, тобто можна зробити очікування результату за допомогою await, що я і зробив далі в тілі оркестратора. Тобто це є звичайний синхронний Post запит, до якого одразу неопосередковано повертається результат, без черги. Натомість, в другому варіанті з RabbitMQ, черга використана і це є асинхронним зв'язком.

Також було записано час виконання задачі.

При синхронному обчисленні одним сервіс провайдером маємо:

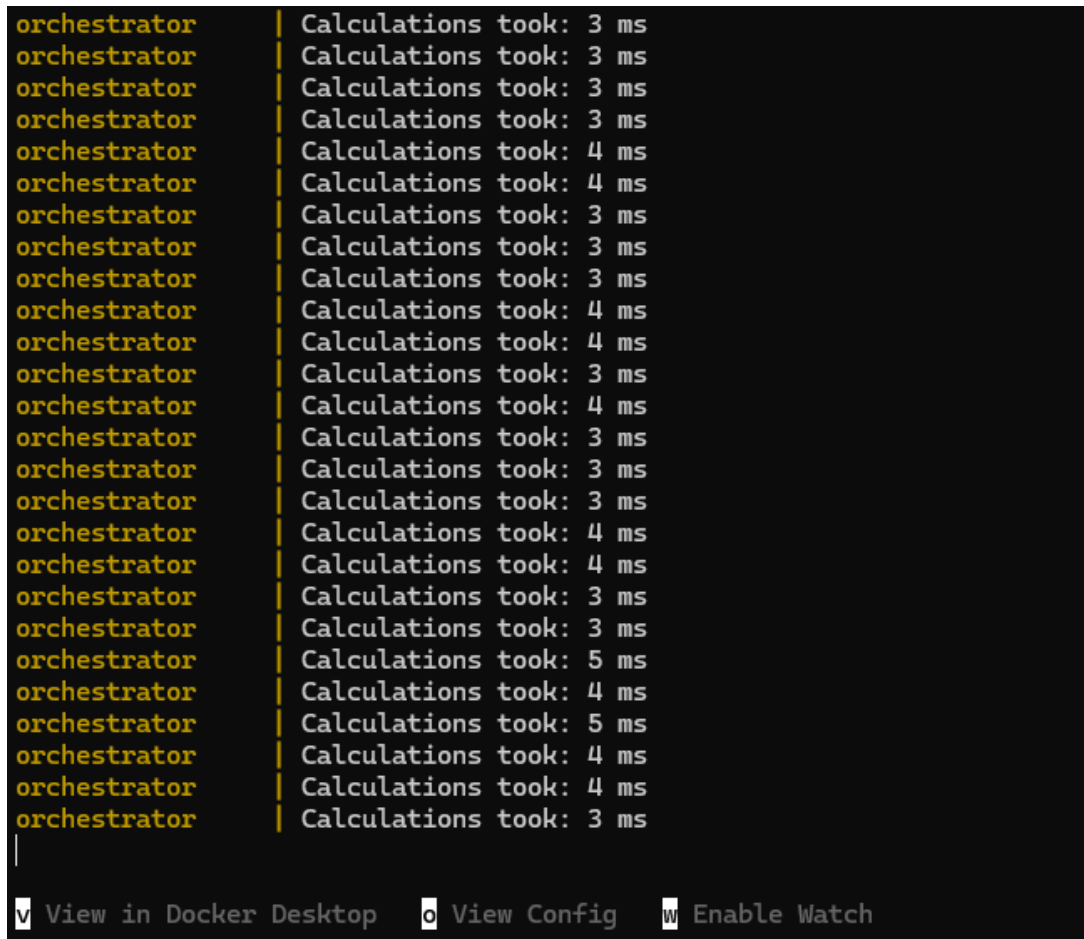


The screenshot shows the 'Logs' tab in Docker Desktop. It displays a list of log entries for a service provider. Each entry consists of a timestamp (2024-12-31 04:38:13 or 04:38:14) followed by the message 'Calculations took: [time] ms'. The times range from 14 ms to 16 ms. On the right side of the logs, there are icons for search, copy, refresh, and a trash can. At the bottom of the logs area, there is a status bar showing 'RAM 4.40 GB CPU 5.99% Disk: 17.66 GB used (limit 1006.85 GB)' and a 'Terminal' button. The version 'v4.37.1' is also visible.

```
2024-12-31 04:38:13 Calculations took: 14 ms
2024-12-31 04:38:13 Calculations took: 15 ms
2024-12-31 04:38:13 Calculations took: 15 ms
2024-12-31 04:38:13 Calculations took: 14 ms
2024-12-31 04:38:13 Calculations took: 14 ms
2024-12-31 04:38:13 Calculations took: 14 ms
2024-12-31 04:38:13 Calculations took: 15 ms
2024-12-31 04:38:13 Calculations took: 16 ms
2024-12-31 04:38:13 Calculations took: 15 ms
2024-12-31 04:38:14 Calculations took: 16 ms
2024-12-31 04:38:14 Calculations took: 15 ms
2024-12-31 04:38:14 Calculations took: 15 ms
2024-12-31 04:38:14 Calculations took: 16 ms
2024-12-31 04:38:14 Calculations took: 16 ms
2024-12-31 04:38:14 Calculations took: 16 ms
2024-12-31 04:38:14 Calculations took: 14 ms
2024-12-31 04:38:14 Calculations took: 15 ms
2024-12-31 04:38:14 Calculations took: 15 ms
2024-12-31 04:38:14 Calculations took: 15 ms
```

RAM 4.40 GB CPU 5.99% Disk: 17.66 GB used (limit 1006.85 GB) >\_ Terminal ✓ v4.37.1

При паралельному обчисленні кількома сервіс провайдерами маємо:



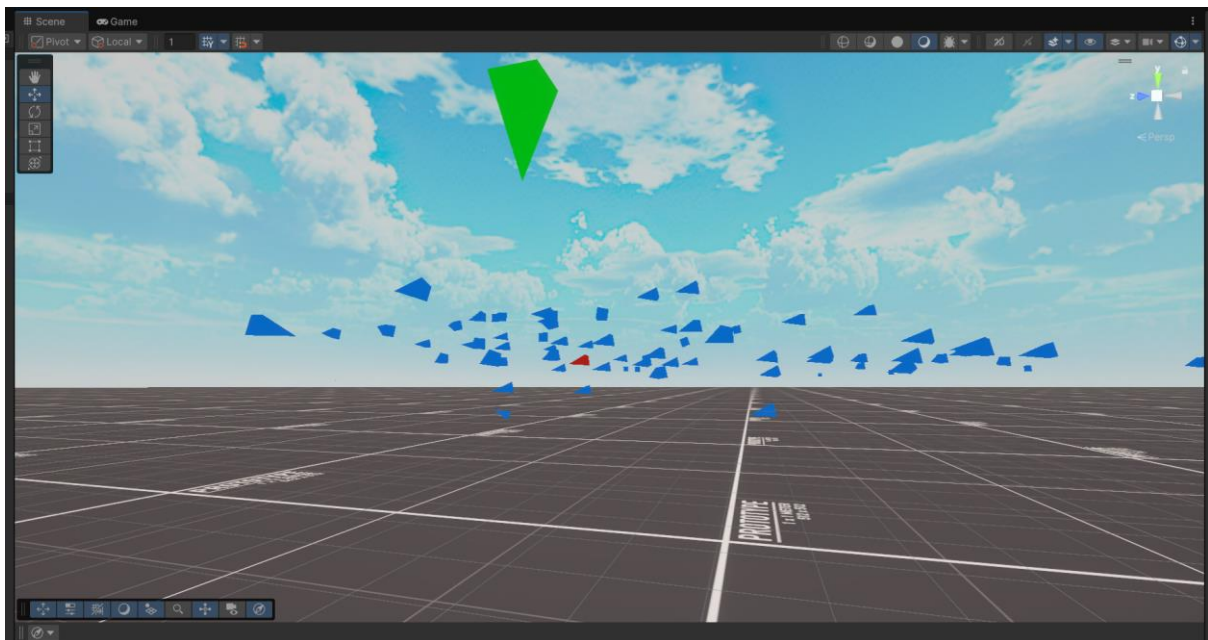
The screenshot shows a terminal window with multiple lines of output. Each line starts with the word 'orchestrator' in yellow, followed by a vertical bar and the message 'Calculations took: [time] ms'. The times range from 3 ms to 5 ms. At the bottom of the terminal, there are three buttons: 'View in Docker Desktop', 'View Config', and 'Enable Watch'.

```
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 3 ms
orchestrator | Calculations took: 5 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 5 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 4 ms
orchestrator | Calculations took: 3 ms
```

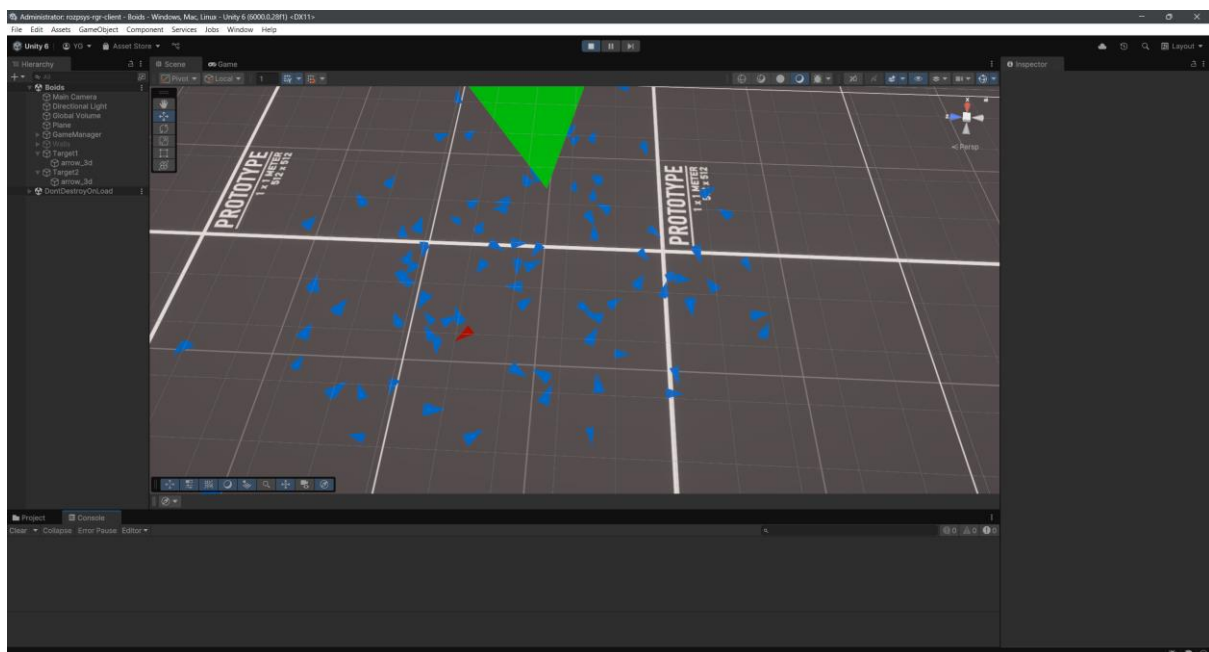
View in Docker Desktop View Config Enable Watch

Отже, кількість сервіс провайдерів суттєво впливають, на швидкість обробки (бо задачу я розпаралелив). Це відбувається тому, що задача ділиться на підзадачі, кожна з яких передається в чергу за допомогою асинхронного зв'язку та вирішується окремо. Сервіс провайдери паралельно беруть ці завдання з черги та повертають результати. Оркестратор чекає, поки придуть всі результати паралельних обчислень, та повертає результат.

Створений клієнт візуалізує результат:

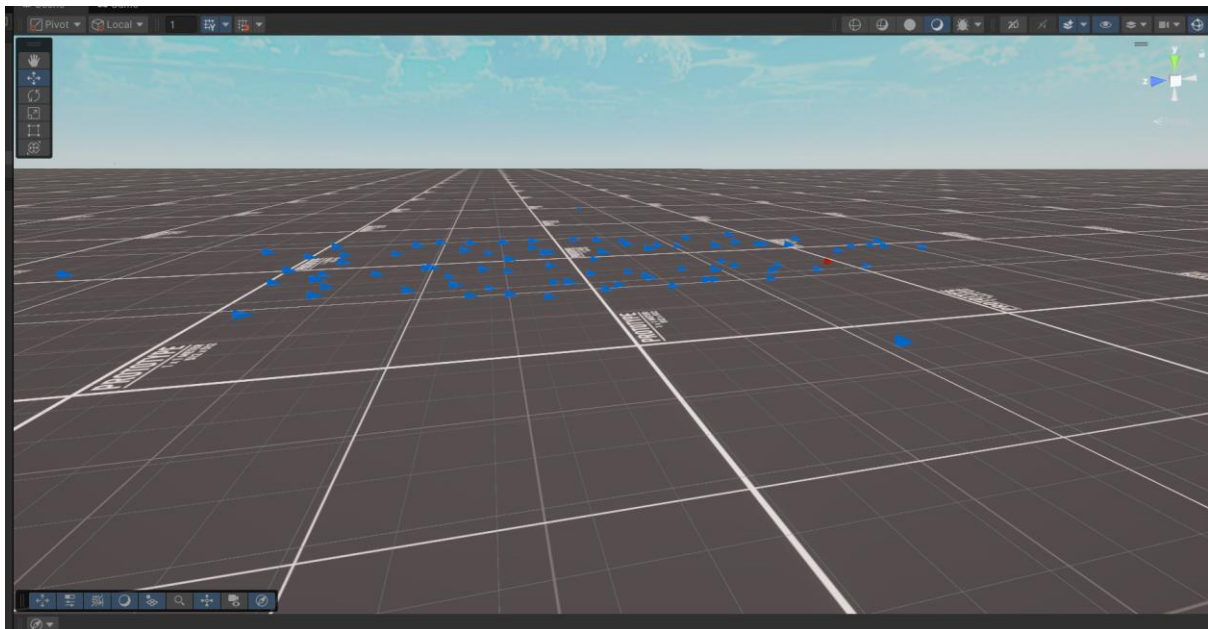


Ось так дрони кучкуються навколо точки інтересу:





Ось так дрони дружньо вирушають до точки інтересу:



Клієнт робить запити до сервера, щоб отримати керування для стаї дронів. Кожен кадр дрони керуються за вказівками сервера, реалізуючи запропоновану поведінку, що наведена у завданні роботи.

Якщо коротко, то існує дрон-лідер (червоний на малюнку), що спілкується з централізованим сервером. Всі інші дрони в групі йому підпорядковуються. Поведінка виконує три принципи стаї: cohesion, alignment, separation. При знищенні лідера, рій обирає нового лідера (реалізовано цю частину моделі на стороні клієнта, як і комунікацію всередині групи дронів). Стая дронів має певні цілі, які можуть змінюватися (встановити можна в редакторі клієнта вручну).

Також я реалізував примітивну фізику для дронів, а саме фізику квадрокоптера, що обчислюється вже на клієнті (репозиторій клієнта теж наводжу вкінці). Тобто дрони це не прості точки, що рухаються з певною швидкістю, а вони мають свою масу, вплив гравітації, і утримуються в повітрі з певною силою роботи моторів, що направлена вгору. Щоб долетіти до цілі, дрони спочатку нахиляються в потрібну сторону. Стабілізується управління за допомогою PID-контролерів. Для спрощення сприймання рою на картинці, я вирішив лишити модельки – стрілочки для дронів.

## **Висновок**

В роботі я створив систему, що реалізує розподілену архітектуру та патерн оркестратор. В проекті маємо балансування навантаження засобами докера та засобами черги. В роботі використано асинхронний та синхронний сервіс провайдер. Сервіс провайдери, що обчислюють математичну модель, масштабуються репліками контейнерів у docker-compose.

Посилання на репозиторії:

Сервер (на якому реалізована архітектура завдання)

<https://github.com/YehorHrybenko/rozpsys-rgr>

Клієнт, який робить запити на сервер та візуалізує процес:

<https://github.com/YehorHrybenko/rozpsys-rgr-client>