Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Фізико-технічний інститут

Проектування розподілених систем

Проект

Replicated log task

Виконала:

Студентка групи ФБ-42мп

Легойда Юлія

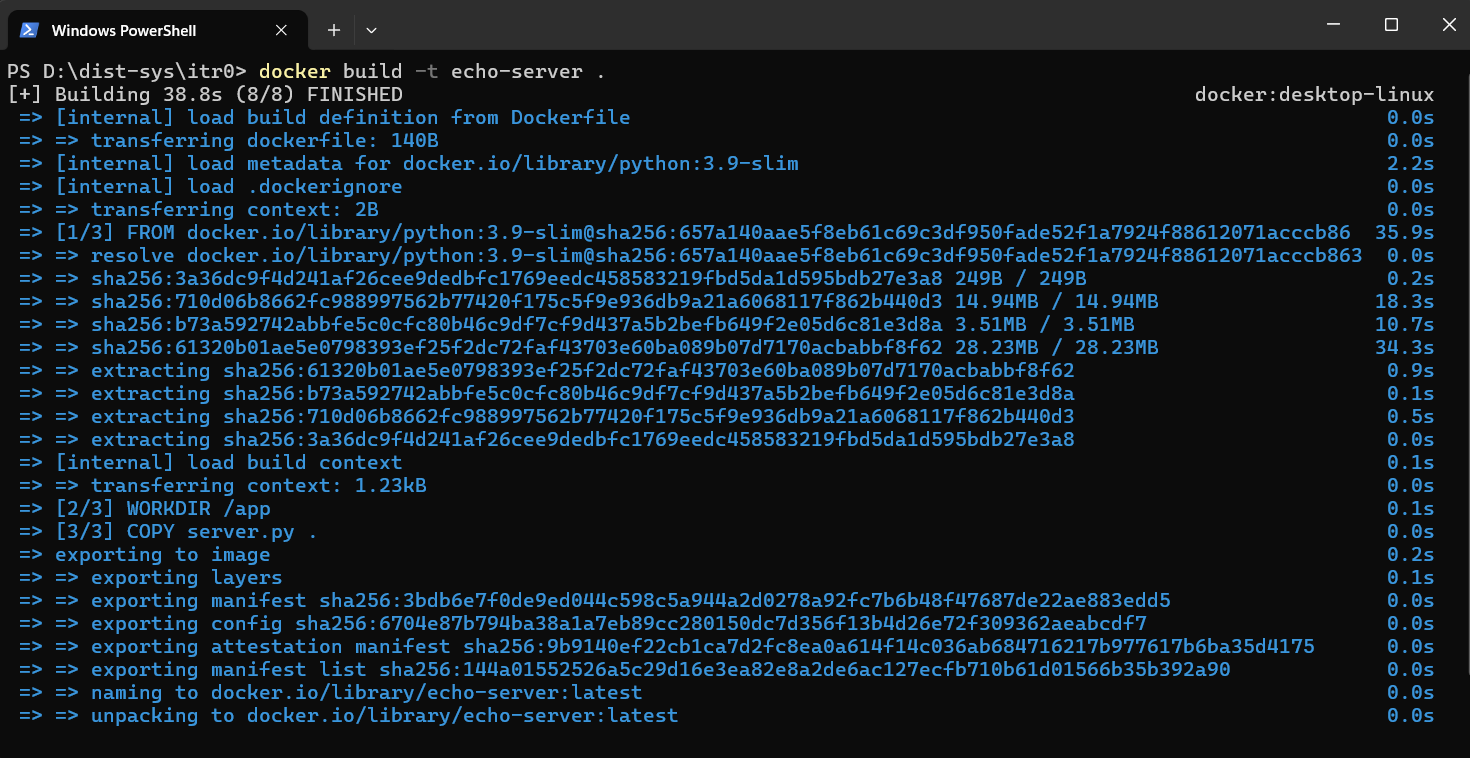
Київ – 2025

Replicated log task

**Iteration 0.**

Choose a desirable language for implementation and try to implement (or find the implementation) a simple *Echo Client-Server* application.

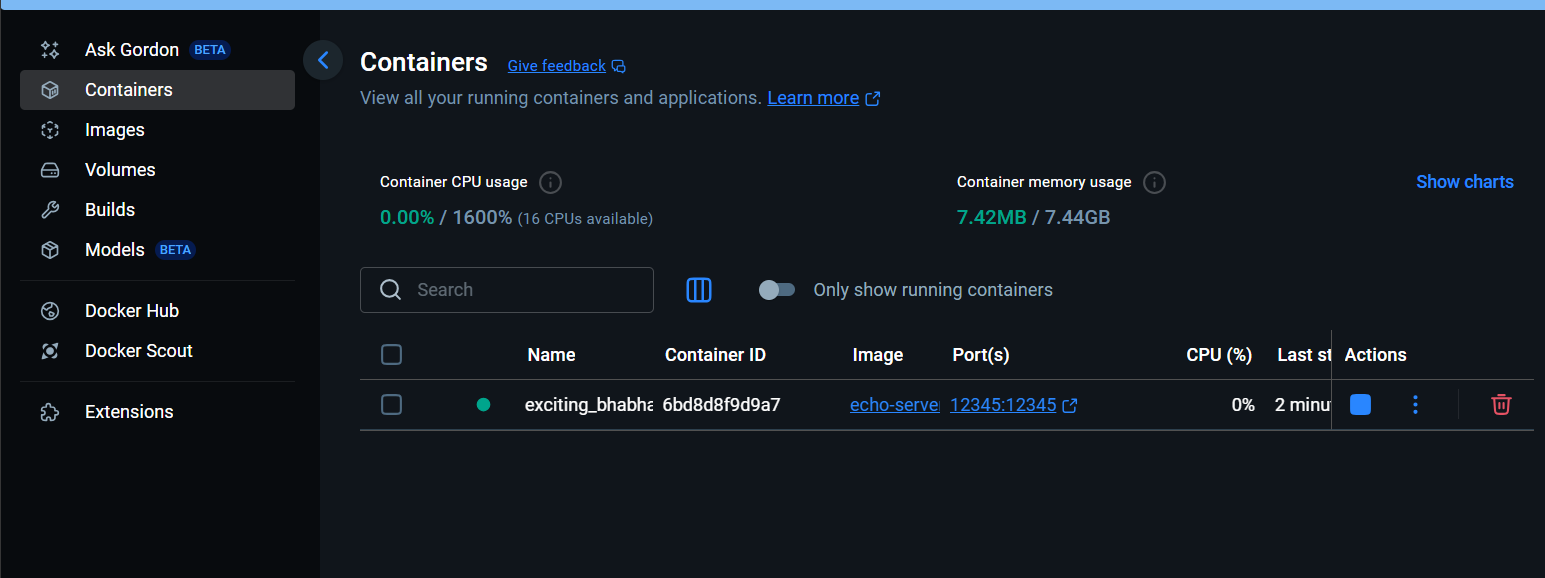
Побудуйте Docker-образ:



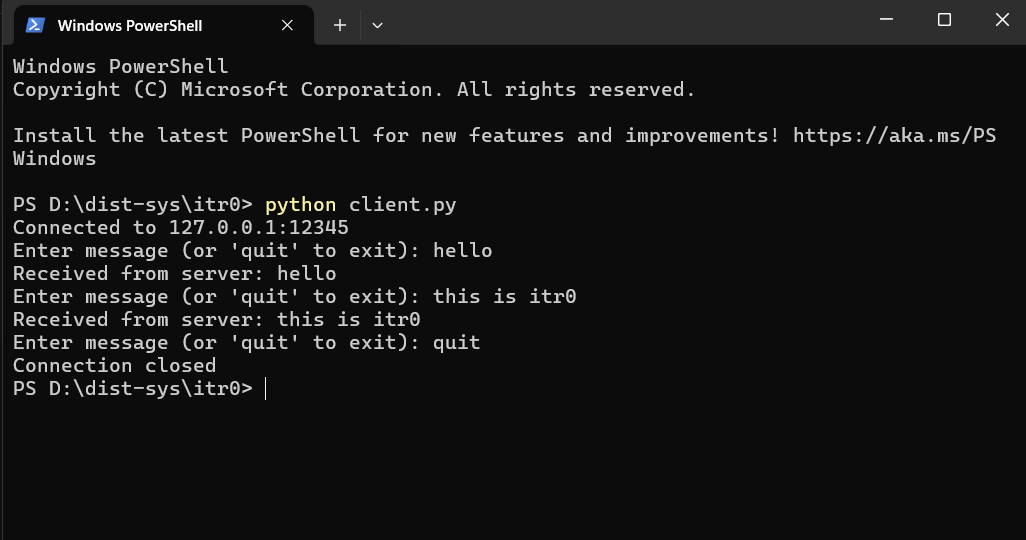
Запустила сервер у контейнері:



Це прив’яже порт 12345 на вашому комп’ютері до порту 12345 у контейнері.



Запускаю клієнт у іншому терміналі:



Сервер виводить повідомлення про підключення клієнта, отримані дані та закриття з’єднання. Клієнт надсилає повідомлення, отримує їх назад від сервера та виводить на екран.

**Iteration 1.**

* **5 points**

The Replicated Log should have the following deployment architecture: one ***Master*** and any number of ***Secondaries***.

У docker-compose.yml визначено один сервіс master і два сервіси secondary1 та secondary2, що демонструє архітектуру з одним Master і кількома Secondary.

services:

  master:

    build: ./master

    ports:

      - "5000:5000"

      - "50050:50050"

    networks:

      - replicated-net

  secondary1:

    build: ./secondary

    ports:

      - "5001:5001"

      - "50051:50051"

    environment:

      - HTTP\_PORT=5001

      - GRPC\_PORT=50051

    networks:

      - replicated-net

  secondary2:

    build: ./secondary

    ports:

      - "5002:5002"

      - "50052:50052"

    environment:

      - HTTP\_PORT=5002

      - GRPC\_PORT=50052

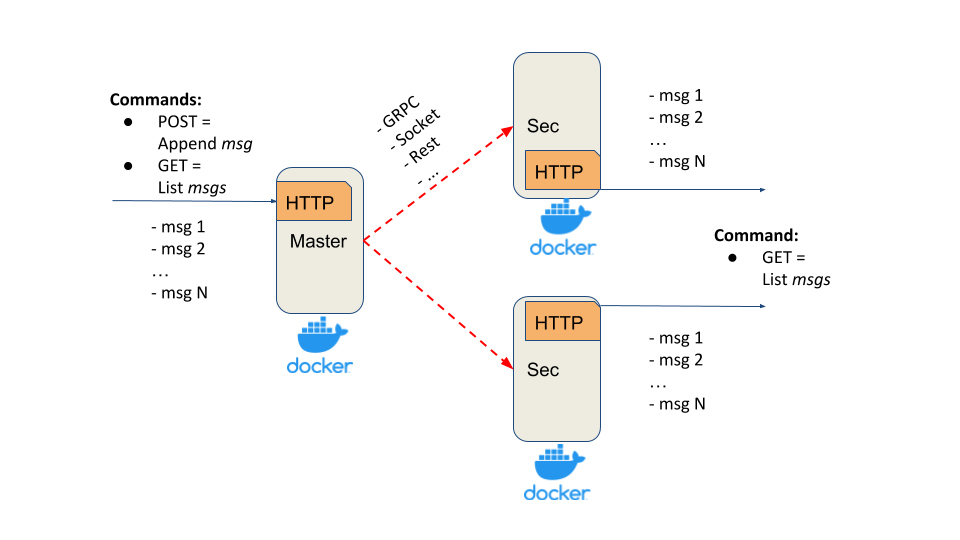
    networks:

      - replicated-net

networks:

  replicated-net:

    driver: bridge



***Master*** should expose a simple HTTP server (or alternative service with a similar API) with:

* *POST method* - appends a message into the in-memory list

У master.py, рядок @app.route("/messages", methods=["POST"]) обробляє POST-запити. Рядок messages.append(message) додає повідомлення до списку messages.

@app.route("/messages", methods=["POST"])

def append\_message():

    message = flask.request.json.get("message")

    if not message:

        return flask.jsonify({"error": "No message provided"}), 400

    messages.append(message)

    log.info(f"Appending message: {message}")

    def replicate\_to\_secondary(addr):

        try:

            with grpc.insecure\_channel(addr) as channel:

                stub = replication\_pb2\_grpc.ReplicationServiceStub(channel)

                response = stub.ReplicateMessage(replication\_pb2.MessageRequest(message=message))

                return response.success

        except grpc.RpcError as e:

            log.error(f"Failed to replicate to {addr}: {e}")

            return False

* *GET method* - returns all messages from the in-memory list

if all(replicate\_to\_secondary(addr) for addr in secondary\_addresses):

        return flask.jsonify({"status": "success", "messages": messages}), 200

    else:

        return flask.jsonify({"error": "Failed to replicate to all secondaries"}), 500

***Secondary*** should expose a simple HTTP server(or alternative service with a similar API) with:

* *GET method* - returns all replicated messages from the in-memory list

Secondary експонує простий HTTP-сервер через Flask із методом GET, який повертає всі репліковані повідомлення з in-memory списку messages.

Properties and assumptions:

* **after each POST request, the message should be replicated on every *Secondary* server**

У master.py, функція replicate\_to\_secondary надсилає повідомлення через gRPC до кожного Secondary, визначеного в secondary\_addresses. Кожен Secondary додає повідомлення до свого списку в ReplicateMessage.

* ***Master* should ensure that *Secondaries* have received a message via *ACK***

*У master.py, функція replicate\_to\_secondary повертає response.success, яке є ACK від Secondary. Ці ACK перевіряються в умові if all(replicate\_to\_secondary(addr) for addr in secondary\_addresses)*

* ***Master’s POST request* should be finished only after receiving *ACKs* from all *Secondaries* (blocking replication approach)**

У master.py, метод join() для потоків блокує виконання POST-запиту, доки всі потоки replicate\_to\_secondary не завершаться, тобто доки не отримано ACK від усіх Secondary.

* **to test that the replication is blocking, introduce the delay/sleep on the *Secondary***

У secondary.py, time.sleep(2), що додає затримку 2 секунди під час обробки кожного повідомлення в ReplicateMessage, що дозволяє перевірити блокуючу поведінку.

* **at this stage assume that the communication channel is a perfect link (no failures and messages lost)**

master.py логується помилка grpc.RpcError

* **any RPC framework can be used for *Master-Secondary* communication (Sockets, language-specific RPC, HTTP, Rest, gRPC, …)**

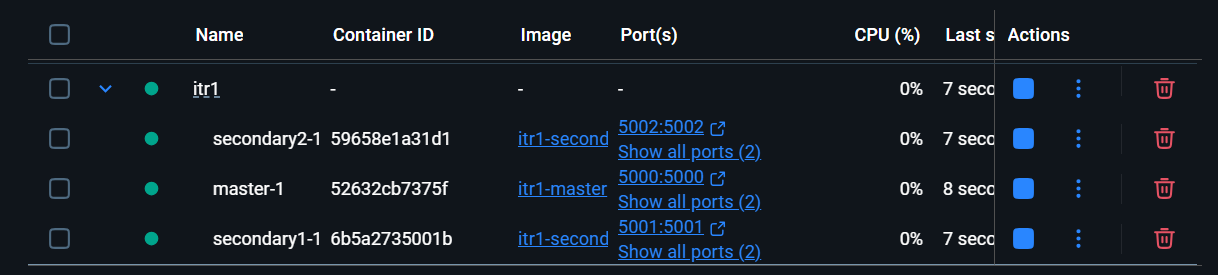
Використовується gRPC, визначений у replication.proto і реалізований через grpc у master.py і secondary.py.

* **your implementation should support logging**

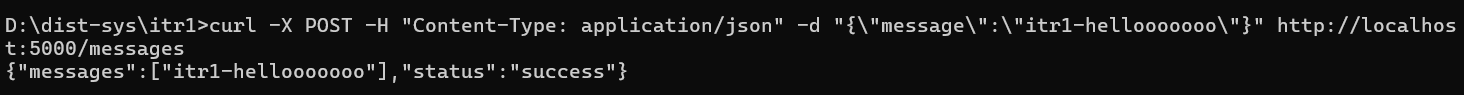
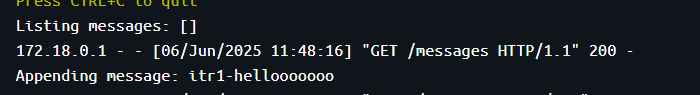
У master.py і secondary.py використовується модуль logging для запису подій

* ***Master* and *Secondaries* should run in Docker**

У docker-compose.yml усі сервіси (master, secondary1, secondary2) визначаються з використанням build, що вказує на запуск у Docker-контейнерах



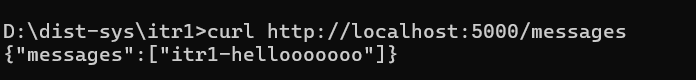
Додаємо повідомлення (POST до Master):

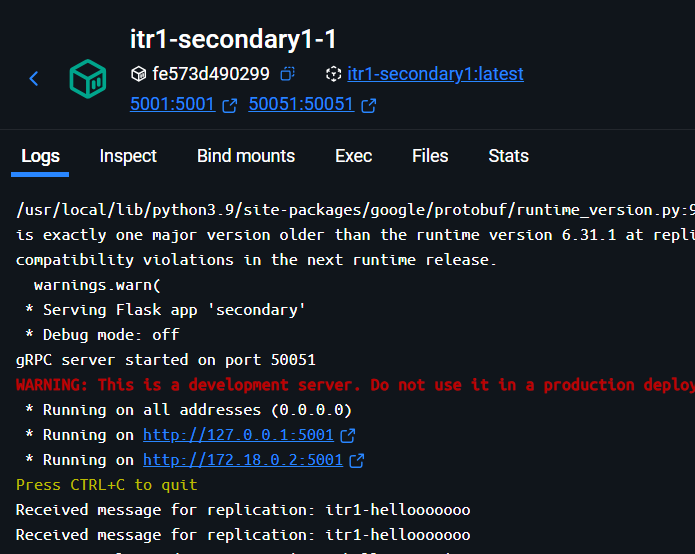
Запит заблокується на ~2 секунди через затримку в Secondary (time.sleep(2)), що підтверджує блокуючу реплікацію.

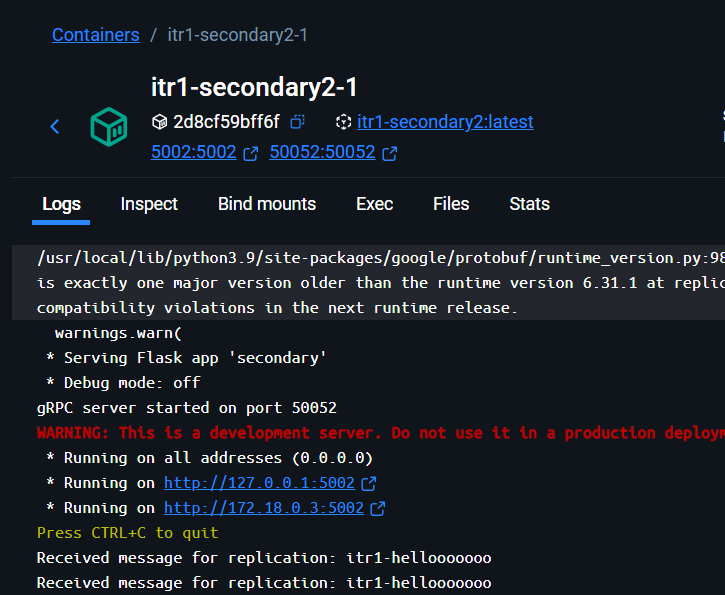
Отримання список повідомлень (GET):

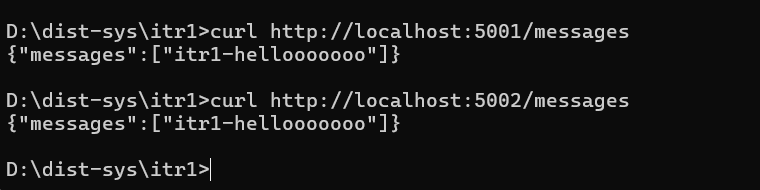
З Master:



З Secondary:



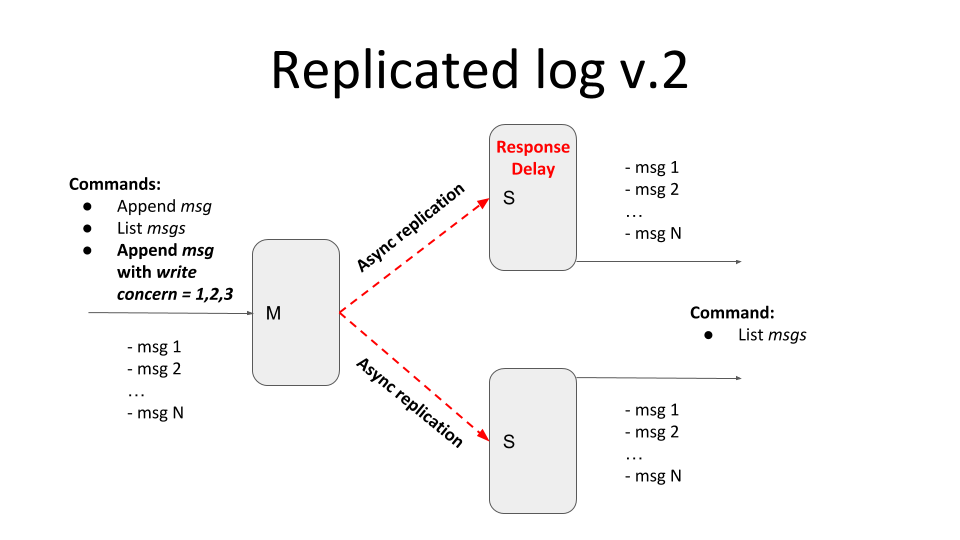




**Iteration 2.**

* **5 points**

In the previous iteration, the replication was blocking for all secondaries, i.e. to return a response to the client we should receive acknowledgements (ACK) from all secondaries.



Current iteration should provide tunable semi-synchronicity for replication, by defining *write concern* parameters.

* client POST request in addition to the message should also contain *write concern* parameter *w=1,2,3,..,n*
* *w* value specifies how many ACKs the master should receive from secondaries before responding to the client

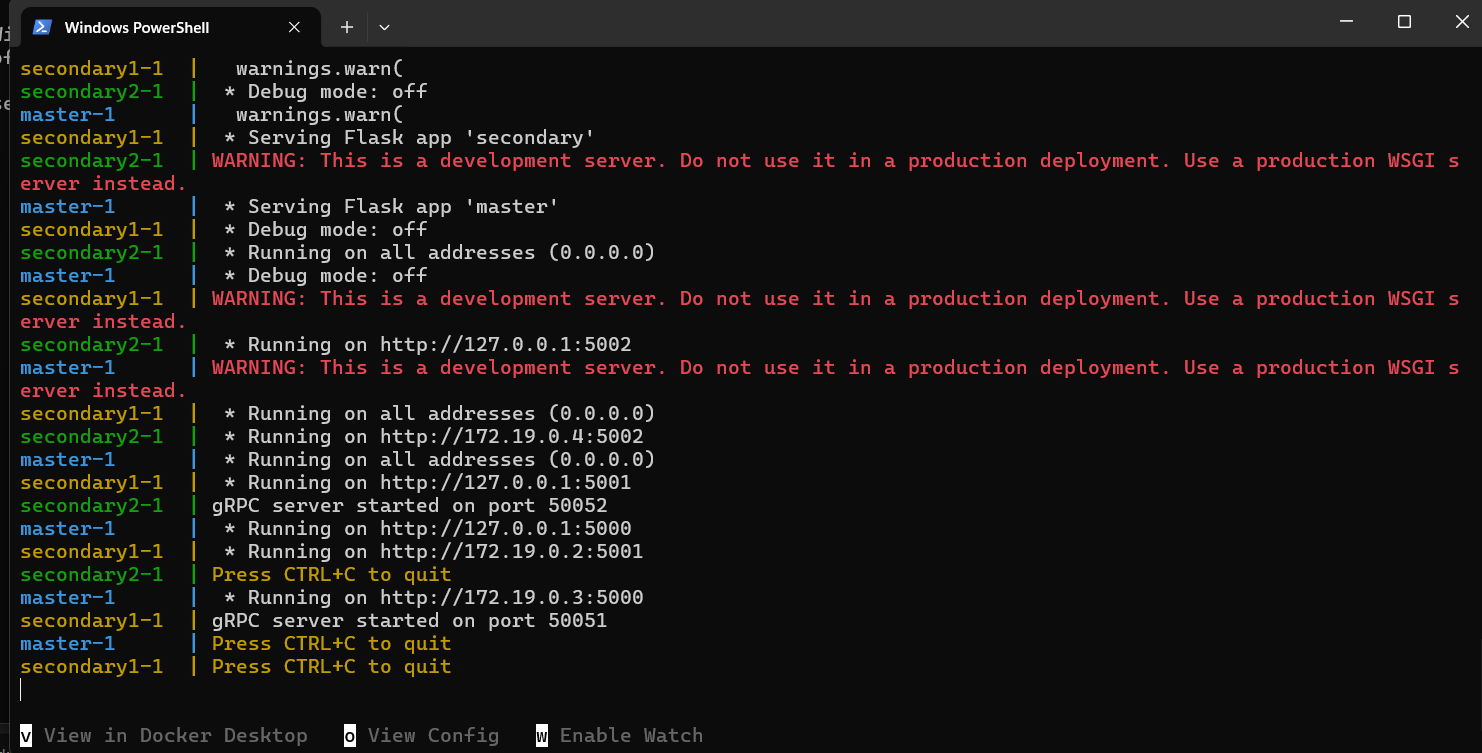
*w = 1* - only from master

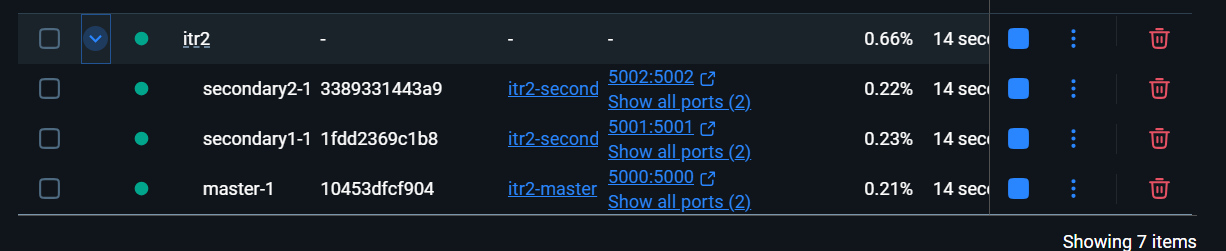
*w = 2* - from master and one secondary

*w = 3* - from master and two secondaries

Please emulate the replica’s inconsistency (and eventual consistency) with the master by introducing the artificial delay on the secondary node. In this case, the master and secondary should temporarily return different lists of messages.

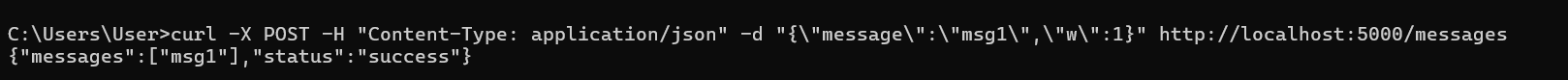
Add logic for messages deduplication and to guarantee the total ordering of messages.



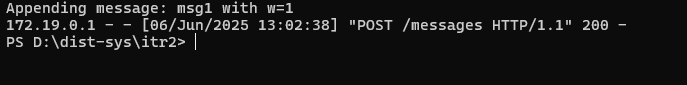


Ми виконаємо POST-запити з різними значеннями w, щоб перевірити semi-synchronicity, а потім GET-запити, щоб продемонструвати тимчасову невідповідність.

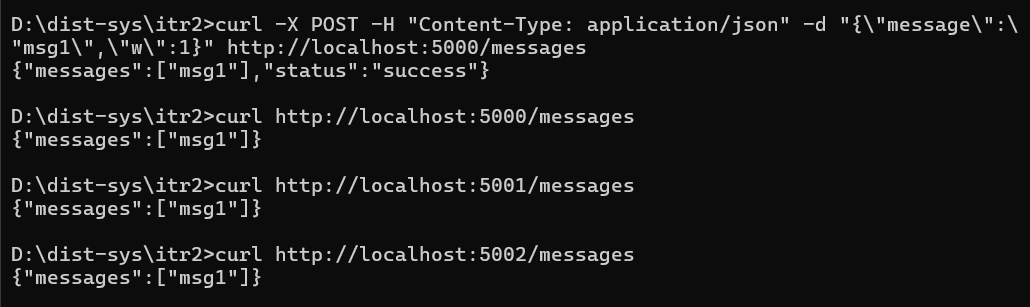
POST-запит з w=1 (тільки Master):



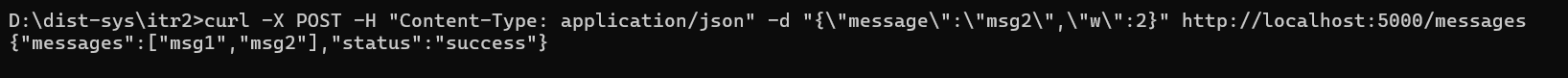
Логи:



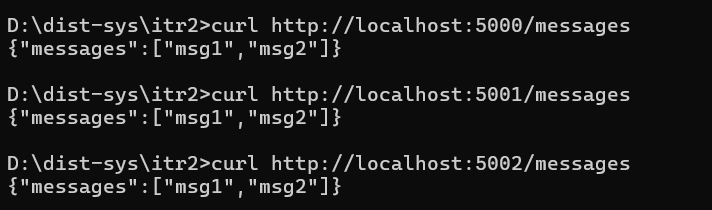
Із завершеною реплікацією:

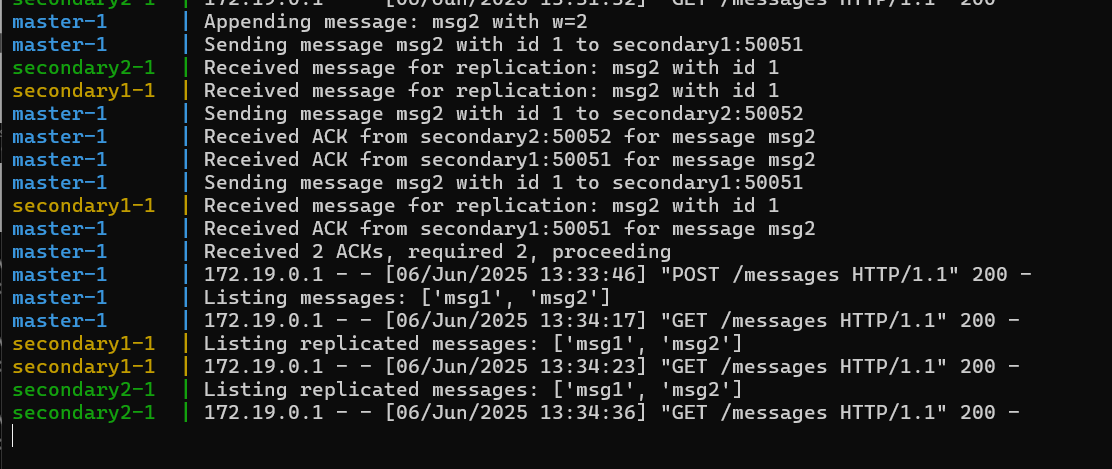


POST-запит з w=2 (Master + 1 Secondary):

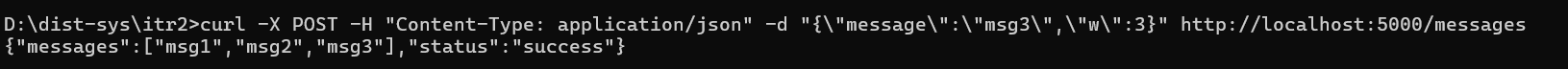


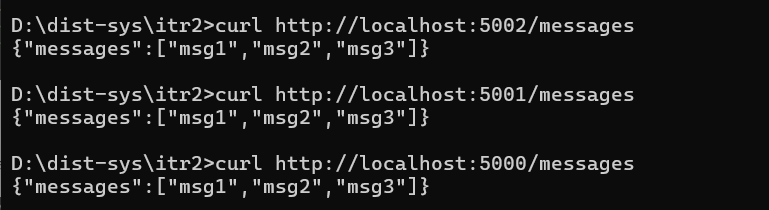
GET-запити для перевірки eventual consistency:





POST-запит з w=3 (Master + 2 Secondary):

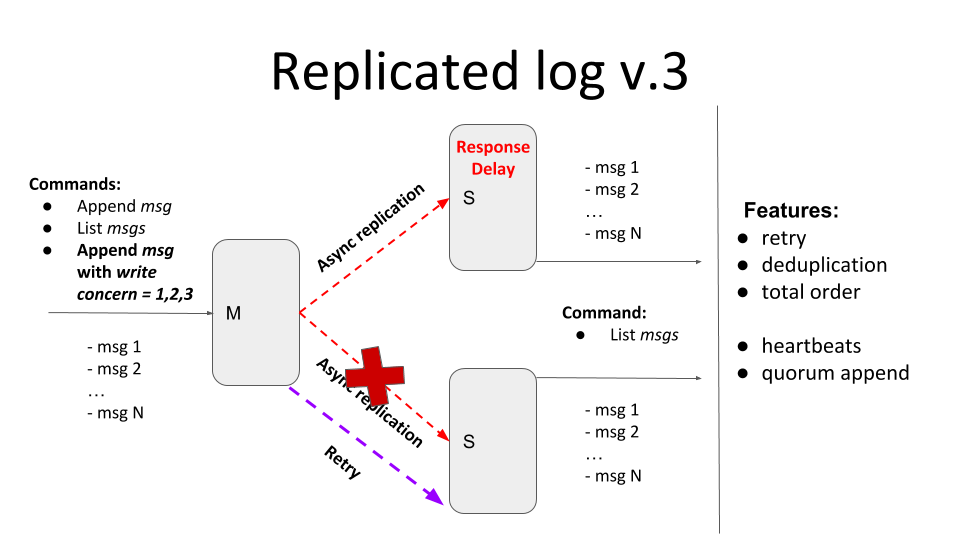






**Iteration 3.**

* **15 points**



The current iteration should provide tunable semi-synchronicity for replication with a *retry* mechanism that should deliver all messages *exactly-once* in total order.

Main features:

* If message delivery fails (due to connection, or internal server error, or secondary is unavailable) the delivery attempts should be repeated - *retry*
  + If one of the secondaries is down and *w=3*, the client should be blocked until the node becomes available. Clients running in parallel shouldn’t be blocked by the blocked one.
  + If *w>1* the client should be blocked until the message will be delivered to all secondaries required by the write concern level. Clients running in parallel shouldn’t be blocked by the blocked one.
  + All messages that secondaries have missed due to unavailability should be replicated after (re)joining the master
  + Retries can be implemented with an unlimited number of attempts but, possibly, with some “smart” delays logic
  + You can specify a *timeout* for the master in case if there is no response from the secondary
* All messages should be present exactly once in the secondary log - *deduplication*
  + To test deduplication you can generate some random internal server error response from the secondary after the message has been added to the log
* The order of messages should be the same in all nodes - *total order*
  + If secondary has received messages *[msg1, msg2, msg4]*, it shouldn’t display the message *‘msg4’* until the *‘msg3’* will bereceived
  + To test the total order, you can generate some random internal server error response from the secondaries

**У цій роботі реалізовано семі-синхронну реплікацію повідомлень між одним Master-вузлом і двома Secondary-вузлами. Ключові функції:**- Write-concern (w) для блокування відповіді клієнта до отримання потрібної кількості ACK.  
- Retry-механізм із нескінченними повторними спробами доставки до вторинок.  
- Дедуплікація у Secondary, щоб жодне повідомлення не з’явилося двічі.  
- Total-order через буферизацію «out-of-order» повідомлень до отримання пропущених.  
- Initial-sync у Secondary після запуску, щоб підтягнути всі пропущені записи із Master.

**1. master.py**

**Основні нові елементи в master.py:  
  
-** Retry‐механізм – якщо під час синхронної реплікації якийсь сервер недоступна або повернув помилку, цикл while True в sync\_replicate\_to\_secondary починає нескінченні спроби з паузою 2 сек. Поки останній сервер не «підніметься», master не відправить клієнту 200.  
  
- Метод /full\_messages – віддає увесь масив повідомлень із полями id, message, order. Це потрібно для механізму initial sync у вторинних вузлах.- Метод /clean – потрібен для коректного функціонування автоматичних тестів, цей виклик очистить messages.

**2. secondary.py**

- Initial Sync-- При старті сервіс робить GET /full\_messages до master.  
-- Отримує масив { id, message, order }, який сортує за order і додає в локальний список messages.  
-- Після цього expected\_order = len(messages) – всі нові POST /replicate будуть додаватися «з місця, де зупинилися».

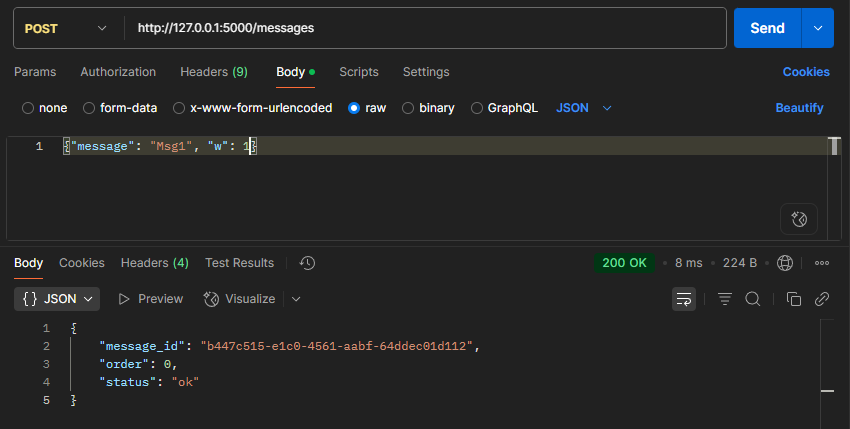
Self-check acceptance test:

* Start M + S1
* send (Msg1, W=1) - Ok
* send (Msg2, W=2) - Ok
* send (Msg3, W=3) - Wait
* send (Msg4, W=1) - Ok
* Start S2
* Check messages on S2 - [Msg1, Msg2, Msg3, Msg4]

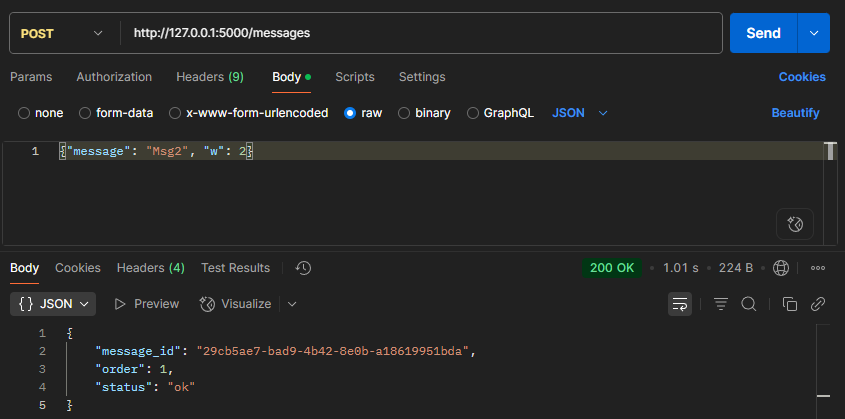
Start M + S1

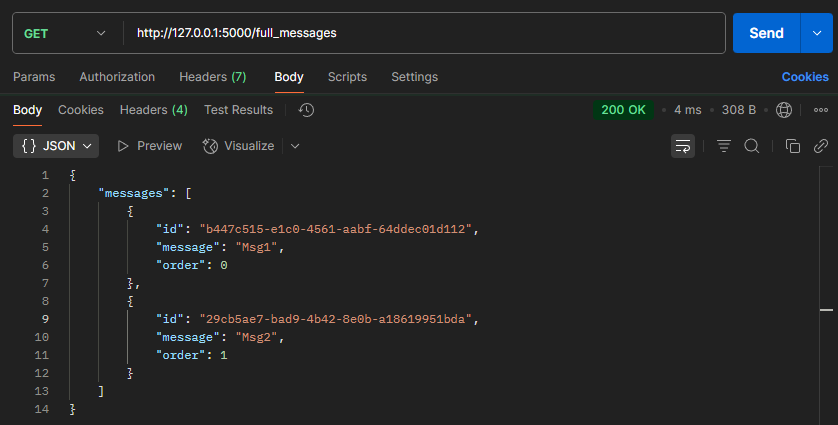


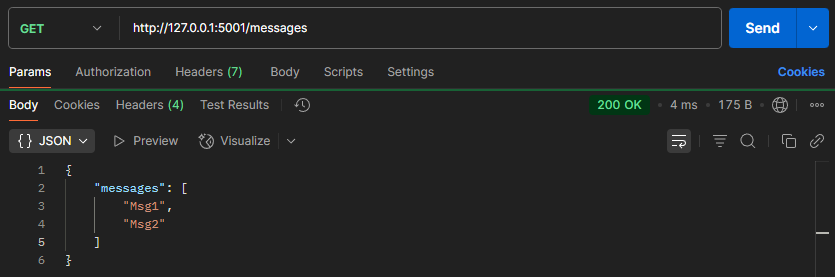
send (Msg1, W=1) — Ok



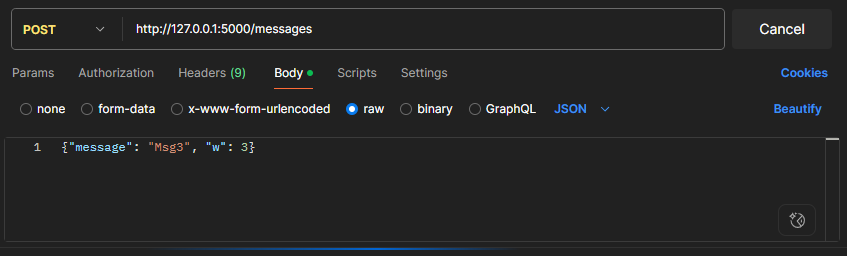
send (Msg2, W=2) — Ok



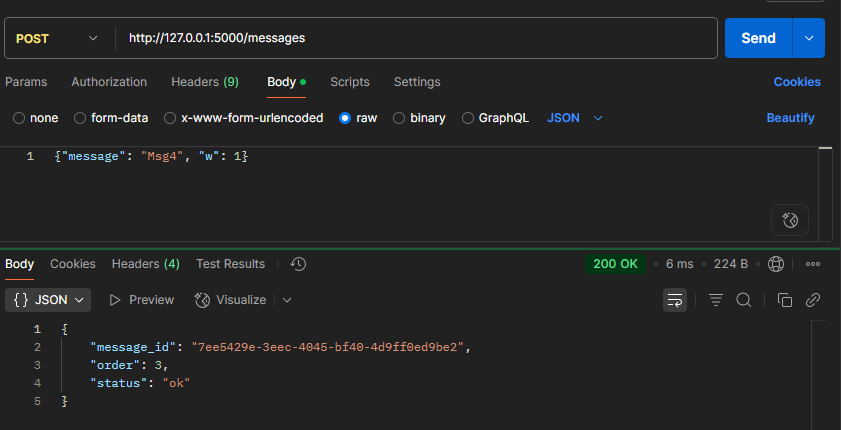


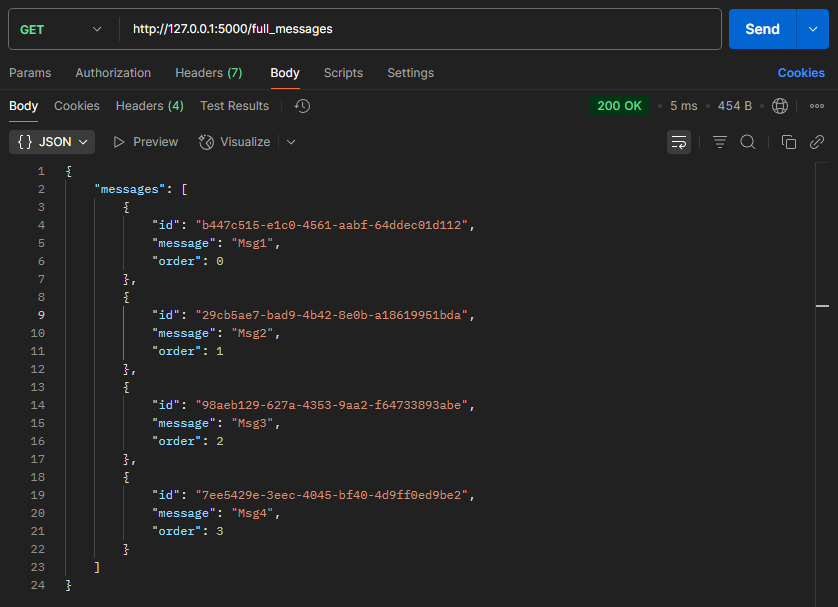


send (Msg3, W=3) - Wait



send (Msg4, W=1) - Ok

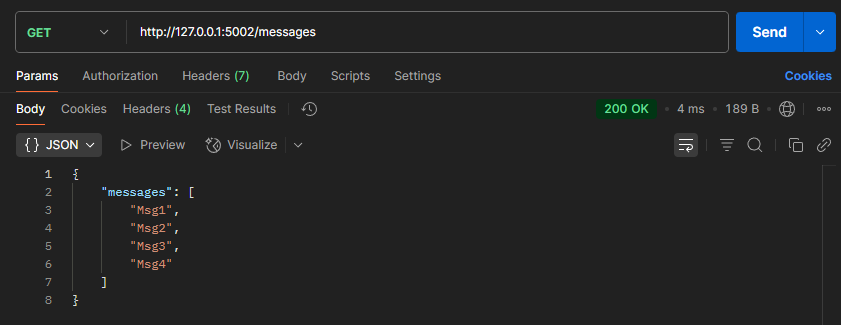




Start S2



Check messages on S2 - [Msg1, Msg2, Msg3, Msg4]



Перевіряю тепер автоматичними тестами, в них перевіряється в більшості все те саме, але автоматично.

Тести:  
1) перевірка що при w=3 master буде чекати запуску усіх secondary, перевірка доставки після запуску, перевірка що сервіс не блокується для інших запитів

2) перевірка що нода отримала всі повідомлення після запуску

3) перевірка на подвійне отримання

4) перевірка на правильний порядок при помилці в доставці

