Лабораторна робота №1

Web-counter

Одінцов Валерій

ФБ-51мп

1. Мета роботи

Метою роботи є порівняння пропускної здатності (throughput) Web-застосунку в залежності від:

- навантаження (кількість одночасних клієнтів),
- способу зберігання даних (в оперативній пам’яті та в реляційній базі даних PostgreSQL).

2. Задача

Реалізувати Web-застосунок з двома ендпоінтами:

- /inc – інкрементує внутрішній лічильник,
- /count – повертає поточне значення лічильника.

Провести експерименти для двох реалізацій:

1. Лічильник зберігається в оперативній пам’яті.
2. Лічильник зберігається в БД PostgreSQL.

Для кожної реалізації виконати серію тестів, коли:

- 1 клієнт виконує 10 000 запитів /inc послідовно;
- 2 клієнти паралельно виконують по 10 000 запитів кожен;
- 5 клієнтів паралельно виконують по 10 000 запитів кожен;
- 10 клієнтів паралельно виконують по 10 000 запитів кожен.

Для кожного сценарію виміряти час виконання та обчислити throughput (кількість запитів в секунду).

3. Опис реалізації

3.1. Використані технології

- Мова програмування: **Python**
- Web-фреймворк: **Flask**
- Бібліотеки:
  - requests – HTTP-клієнт
  - psycopg2-binary – підключення до PostgreSQL
- СУБД: **PostgreSQL**
- Операційна система: (твоя)

3.2. Web-застосунок з лічильником в оперативній пам'яті

У першій реалізації лічильник зберігається у глобальній змінній Python. Для забезпечення потокобезпеки використовується об'єкт threading.Lock.

Ключові моменти:

- глобальна змінна counter;
- counter_lock = threading.Lock() для синхронізації;
- всі операції змінення та читання лічильника виконуються в критичній секції with counter_lock:;
- Flask запускатиметься з параметром threaded=True, що дозволяє обробляти кілька запитів паралельно.

Основний фрагмент коду (app_memory.py) та результат:

```python
from flask import Flask, jsonify
import threading

app = Flask(__name__)

counter = 0
counter_lock = threading.Lock()

@app.route("/inc", methods=["GET"])
def inc():
    global counter
    with counter_lock:
        counter += 1
        value = counter
    return jsonify({"count": value})

@app.route("/count", methods=["GET"])
def get_count():
    with counter_lock:
        value = counter
    return jsonify({"count": value})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080, threaded=True)
```

```
┌──(kali㉿kali)-[~/Downloads/prog]
└─$ python load_client.py -c 1 -n 10000

Starting load: 1 clients x 10000 requests
Elapsed time: 24.8751 s
Total requests: 10000
/count returned: 10000
Throughput: 402.01 req/s

┌──(kali㉿kali)-[~/Downloads/prog]
└─$ python load_client.py -c 2 -n 10000

Starting load: 2 clients x 10000 requests
Elapsed time: 35.0551 s
Total requests: 20000
/count returned: 30000
Throughput: 570.53 req/s

┌──(kali㉿kali)-[~/Downloads/prog]
└─$ python load_client.py -c 5 -n 10000

Starting load: 5 clients x 10000 requests
Elapsed time: 76.1701 s
Total requests: 50000
/count returned: 80000
Throughput: 656.43 req/s

┌──(kali㉿kali)-[~/Downloads/prog]
└─$ python load_client.py -c 10 -n 10000

Starting load: 10 clients x 10000 requests
Elapsed time: 158.5592 s
Total requests: 100000
/count returned: 180000
Throughput: 630.68 req/s
```

3.3. Web-застосунок з лічильником у PostgreSQL

У другій реалізації лічильник зберігається в таблиці counter бази даних webcounter.
Структура таблиці:

```sql
CREATE TABLE counter (
    id    integer PRIMARY KEY,
    value bigint NOT NULL
);

INSERT INTO counter (id, value) VALUES (1, 0);
```

Для підключення до БД використовується пул з'єднань SimpleConnectionPool з бібліотеки psycopg2.
Операція інкременту виконується одним SQL-запитом:

```sql
UPDATE counter SET value = value + 1 WHERE id = 1 RETURNING value;
```

Це забезпечує атомарність операції інкремента та відсутність lost update на рівні БД.

Також у БД був створений користувач (наприклад, webuser) і видані права на таблицю:

```sql
GRANT ALL PRIVILEGES ON TABLE counter TO webuser;
ALTER TABLE counter OWNER TO webuser;
```

Основний фрагмент коду (app_db.py):

```python
from flask import Flask, jsonify
import os
from psycopg2.pool import SimpleConnectionPool

app = Flask(__name__)

DB_HOST = os.getenv("DB_HOST", "localhost")
DB_PORT = int(os.getenv("DB_PORT", "5432"))
DB_NAME = os.getenv("DB_NAME", "webcounter")
DB_USER = os.getenv("DB_USER", "webuser")
DB_PASSWORD = os.getenv("DB_PASSWORD", "strong_password")

db_pool = SimpleConnectionPool(
    minconn=1,
    maxconn=20,
    host=DB_HOST,
    port=DB_PORT,
    dbname=DB_NAME,
    user=DB_USER,
    password=DB_PASSWORD,
)

def get_conn():
    return db_pool.getconn()

def put_conn(conn):
    db_pool.putconn(conn)
```

```python
@app.route("/inc", methods=["GET"])
def inc_db():
    conn = get_conn()
    try:
        with conn:
            with conn.cursor() as cur:
                cur.execute(
                    "UPDATE counter SET value = value + 1 WHERE id = 1 RETURNING value;"
                )
                (value,) = cur.fetchone()
    finally:
        put_conn(conn)

    return jsonify({"count": value})
```

```python
@app.route("/count", methods=["GET"])
def count_db():
    conn = get_conn()
    try:
        with conn:
            with conn.cursor() as cur:
                cur.execute("SELECT value FROM counter WHERE id = 1;")
                (value,) = cur.fetchone()
    finally:
        put_conn(conn)

    return jsonify({"count": value})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080, threaded=True)
```

3.4. Клієнт для навантаження (Python)

Клієнт реалізований на Python з використанням ThreadPoolExecutor. Ідея:

- параметр -c/--clients – кількість паралельних клієнтів;
- параметр -n/--requests-per-client – кількість /inc на кожного клієнта;
- кожен клієнт послідовно викликає /inc у своєму потоці;
- вимірюється загальний час від старту до завершення всіх потоків;
- після цього викликається /count та обчислюється throughput.

```python
import argparse
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
import requests

def worker(client_id: int, base_url: str, requests_per_client: int):
    session = requests.Session()
    inc_url = base_url.rstrip("/") + "/inc"

    for _ in range(requests_per_client):
        resp = session.get(inc_url)
        resp.raise_for_status()

    return client_id

def main():
    parser = argparse.ArgumentParser(description="Web-counter load client")
    parser.add_argument("-c", "--clients", type=int, default=1)
    parser.add_argument("-n", "--requests-per-client", type=int, default=10000)
    parser.add_argument("-u", "--url", type=str, default="http://localhost:8080")
    args = parser.parse_args()
```

```
total_requests = args.clients * args.requests_per_client
print(f"Starting load: {args.clients} clients x {args.requests_per_client} requests")

start = time.perf_counter()

with ThreadPoolExecutor(max_workers=args.clients) as executor:
    futures = [
        executor.submit(
            worker, client_id=i, base_url=args.url,
            requests_per_client=args.requests_per_client
        )
        for i in range(args.clients)
    ]
    for f in as_completed(futures):
        f.result()

end = time.perf_counter()
elapsed = end - start

count_resp = requests.get(args.url.rstrip("/") + "/count")
count_resp.raise_for_status()
count_value = count_resp.json().get("count")

throughput = total_requests / elapsed if elapsed > 0 else 0.0
```

```
    print(f"Elapsed time: {elapsed:.4f} s")
    print(f"Total requests: {total_requests}")
    print(f"/count returned: {count_value}")
    print(f"Throughput: {throughput:.2f} req/s")

if __name__ == "__main__":
    main()
```

4. Результати експериментів

4.1. Реалізація з лічильником в пам'яті

Для кожного сценарію сервер app_memory.py було запущено з нульовим значенням лічильника.

Клієнт load_client.py запускався з такими параметрами:

1. python load_client.py -c 1 -n 10000
2. python load_client.py -c 2 -n 10000
3. python load_client.py -c 5 -n 10000
4. python load_client.py -c 10 -n 10000

Отримані результати:

Таблиця 1 — Throughput для in-memory реалізації

# клієнтів	Запитів на клієнта	Загальна к-сть запитів	Час, s	Черезпускна здатність, req/s
1	10000	10000	24s	402
2	10000	20000	35s	570
5	10000	50000	76s	656
10	10000	100000	158s	630

4.2. Реалізація з лічильником у PostgreSQL

Аналогічно проводилися експерименти для сервера app_db.py:

1. python load_client.py -c 1 -n 10000
2. python load_client.py -c 2 -n 10000
3. python load_client.py -c 5 -n 10000
4. python load_client.py -c 10 -n 10000

Таблиця 2 – Throughput для реалізації з PostgreSQL

# клієнтів	Запитів на клієнта	Загальна к-сть запитів	Час, s	Черезпускна здатність, req/s
1	10000	10000	46	216
2	10000	20000	106	187
5	10000	50000	261	191
10	10000	100000	539	185

5. Аналіз результатів

Отримані результати показали очікувану картину: реалізація з лічильником в оперативній пам’яті демонструє вищу пропускну здатність порівняно з варіантом, де лічильник зберігається в базі даних PostgreSQL. Це логічно, оскільки доступ до глобальної змінної в пам’яті значно дешевший за повноцінне виконання SQL-запитів, використання пулу з’єднань, мережевого стека та механізмів журналювання транзакцій у СУБД.

Для реалізації з PostgreSQL (таблиця 2) throughput знаходився в межах приблизно 185–216 запитів за секунду:
– для 1 клієнта: 10 000 запитів виконуються за 46 секунд (≈216 req/s);
– для 2 клієнтів: 20 000 запитів за 106 секунд (≈187 req/s);
– для 5 клієнтів: 50 000 запитів за 261 секунду (≈191 req/s);
– для 10 клієнтів: 100 000 запитів за 539 секунд (≈185 req/s).

Час виконання зростає майже лінійно із загальною кількістю запитів, а пропускна здатність залишається приблизно сталою. Це означає, що головним «вузьким місцем» у даному випадку є саме база даних: кожен інкремент вимагає виконання транзакції з оновленням рядка в таблиці counter. Збільшення кількості паралельних клієнтів (з 1 до 10) не призводить до суттєвого збільшення throughput, а навпаки, дає невеликі коливання навколо ~190 req/s через конкуренцію за ресурси БД, блокування одного й того ж рядка та обмеження пулу з’єднань.

У in-memory реалізації масштабування з кількістю клієнтів проявляється краще: завдяки багатопоточності і відсутності звернень до зовнішніх ресурсів (БД) пропускна здатність зростає до певної межі, поки не починають домінувати накладні витрати на перемикання потоків і блокування threading.Lock. У результаті in-memory варіант виявляється швидшим, але менш універсальним з точки зору збереження стану та горизонтального масштабування.

З точки зору потокобезпеки обидві реалізації поводяться коректно. У in-memory варіанті використання threading.Lock гарантує відсутність втрат оновлень (lost update) при конкурентному доступі до змінної counter. У варіанті з PostgreSQL атомарний SQL-запит UPDATE counter SET value = value + 1 WHERE id = 1 всередині транзакції забезпечує коректний інкремент навіть за одночасних оновлень з боку кількох клієнтів. Під час експериментів значення, що поверталося ендпоінтом /count, відповідало очікуваному: 10 000 для 1 клієнта, 20 000 для 2 клієнтів, 50 000 для 5 клієнтів та 100 000 для 10 клієнтів, що підтверджує відсутність помилок синхронізації.