

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ  
 НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ   
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу  
Кафедра системного проектування**

**Звіт**

**про виконання практичної роботи №5  
з дисципліни «Паралельні обчислення»**

Виконав:  
студент III курсу, групи ДА-22  
Байдюк Антон Олексійович

Прийняв:

асистент Яременко В. С.

Київ – 2025

**ПОСТАНОВКА ЗАДАЧІ**

**Тема:** Розробка HTTP-сервера для хостингу статичного веб-сайту

**Мета:** Вивчити принцип роботи http веб-серверів. Створити простий веб-сервер для багатопотокової обробки GET-запитів. Засвоїти інструментарій для навантажувального тестування.

Хід роботи:

1. Спроектувати веб сервер для хостингу статичного контенту. Можна обирати довільну технологією, вимогою є програмування на транспортному (четвертому) рівні стеку OSI (з допомогою сокетів).

2. Веб-сервер повинен опрацьовувати http запити з багатьох клієнтів. Реалізуємо GET відповідно до протоколу HTTP 1.1. RFC9112 RFC9110 Root-запит веде на index.html (приклади запитів “GET http://localhost:8080”, “GET http://localhost:8080/second\_page.html”).

3. Захостити щонайменше 2 веб-сторінки (текст файлів наведено в Додатку 1, або можете додати власні). При цьому повинні оброблятися неіснуючі сторінки. (404 відповідь)

4. Провести навантажувальне тестування з допомогою сервісів jMeter або Locust. Визначити точку відмови в кількості паралельних клієнтів.

5. Зробити висновки, припустити яким чином можна зробити сервер більш ефективним і підвищити його здатність тримати навантаження.

6. (\*) Додаткове завдання на плюс 1 бал – оптимізувати веб-сервер, зробивши його неблокуючим.

**ХІД РОБОТИ**

Код для серверної частини:

#include <iostream>

#include <fstream>

#include <string>

#include <winsock2.h>

#include <ws2tcpip.h>

#pragma comment(lib, "ws2\_32.lib")

#define PORT 8080

void sendResponse(SOCKET clientSocket, const std::string& content, const std::string& contentType = "text/html") {

std::string httpResponse =

"HTTP/1.1 200 OK\r\n"

"Content-Length: " + std::to\_string(content.size()) + "\r\n"

"Content-Type: " + contentType + "\r\n"

"\r\n" + content;

send(clientSocket, httpResponse.c\_str(), httpResponse.size(), 0);

}

void sendNotFound(SOCKET clientSocket) {

std::string notFound =

"HTTP/1.1 404 Not Found\r\n"

"Content-Length: 13\r\n"

"Content-Type: text/plain\r\n"

"\r\n404 Not Found";

send(clientSocket, notFound.c\_str(), notFound.size(), 0);

}

std::string loadFileContent(const std::string& filePath) {

std::ifstream file(filePath, std::ios::binary);

if (!file) return "";

return std::string((std::istreambuf\_iterator<char>(file)), std::istreambuf\_iterator<char>());

}

void handleRequest(SOCKET clientSocket) {

char buffer[2048];

int received = recv(clientSocket, buffer, sizeof(buffer) - 1, 0);

if (received <= 0) {

closesocket(clientSocket);

return;

}

buffer[received] = '\0';

std::string request(buffer);

std::string requestedFile = "index.html"; // default

if (request.find("GET / ") != std::string::npos || request.find("GET /index.html") != std::string::npos) {

requestedFile = "index.html";

}

else if (request.find("GET /page2.html") != std::string::npos) {

requestedFile = "page2.html";

}

else {

sendNotFound(clientSocket);

closesocket(clientSocket);

return;

}

std::string content = loadFileContent(requestedFile);

if (content.empty()) {

sendNotFound(clientSocket);

}

else {

sendResponse(clientSocket, content);

}

closesocket(clientSocket);

}

int main() {

WSADATA wsaData;

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

std::cerr << "WSAStartup failed.\n";

return 1;

}

SOCKET serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (serverSocket == INVALID\_SOCKET) {

std::cerr << "Socket creation failed\n";

WSACleanup();

return 1;

}

sockaddr\_in serverAddr{};

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_addr.s\_addr = INADDR\_ANY;

serverAddr.sin\_port = htons(PORT);

if (bind(serverSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {

std::cerr << "Bind failed\n";

closesocket(serverSocket);

WSACleanup();

return 1;

}

if (listen(serverSocket, 5) == SOCKET\_ERROR) {

std::cerr << "Listen failed\n";

closesocket(serverSocket);

WSACleanup();

return 1;

}

std::cout << "Server listening on port " << PORT << "...\n";

while (true) {

sockaddr\_in clientAddr{};

int clientAddrLen = sizeof(clientAddr);

SOCKET clientSocket = accept(serverSocket, (sockaddr\*)&clientAddr, &clientAddrLen);

if (clientSocket == INVALID\_SOCKET) {

std::cerr << "Accept failed\n";

continue;

}

char clientIP[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &(clientAddr.sin\_addr), clientIP, INET\_ADDRSTRLEN);

std::cout << "Connection from " << clientIP << ":" << ntohs(clientAddr.sin\_port) << "\n";

handleRequest(clientSocket);

}

closesocket(serverSocket);

WSACleanup();

return 0;

}

Код для веб сторінок:

index.html:

<!DOCTYPE html>

<html>

<head>

<title>Page 1</title>

</head>

<body>

<h1>Welcome to Page 1</h1>

<p>Click <a href="page2.html">here</a> to go to Page 2.</p>

</body>

</html>

page2.html:

<!DOCTYPE html>

<html>

<head>

<title>Page 2</title>

</head>

<body>

<h1>Welcome to Page 2</h1>

<p>Click <a href="index.html">here</a> to go back to Page 1.</p>

</body>

Встановлення locust:

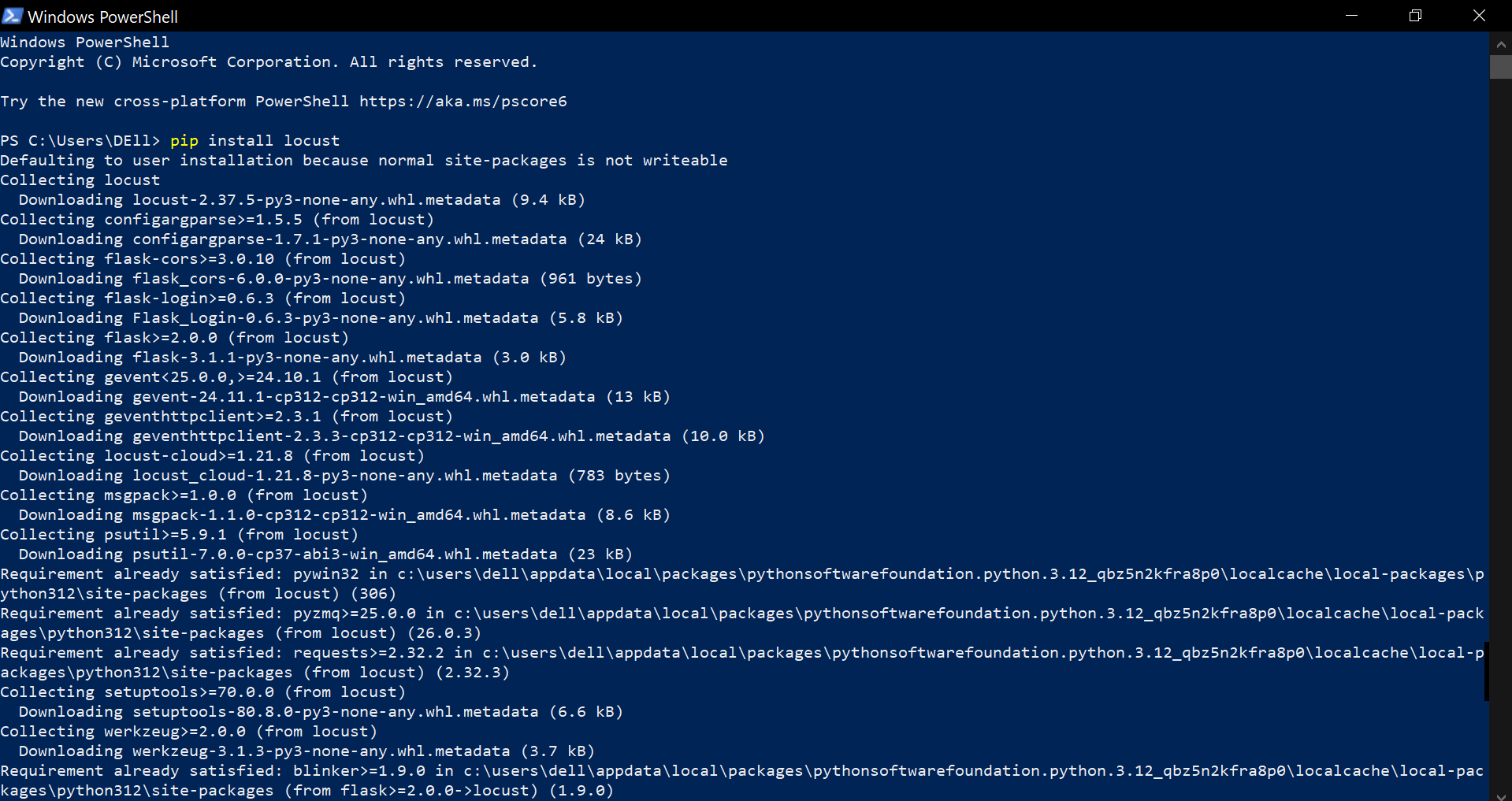


Рис. 1 - Встановлення locust

Код для тестування в locust

from locust import HttpUser, task

class StaticWebUser(HttpUser):

@task

def load\_index(self):

self.client.get("/index.html")

@task

def load\_page2(self):

self.client.get("/page2.html")

@task

def load\_404(self):

self.client.get("/404.html")

Для запуску тестування потрібно у терміналі зайти у локацію папки із файлами та ввести команду запуску, у моєму випадку:

cd C:\Users\DEll\Desktop\Lab5\_server

python -m locust -f locustfile.py --host=http://localhost:8080

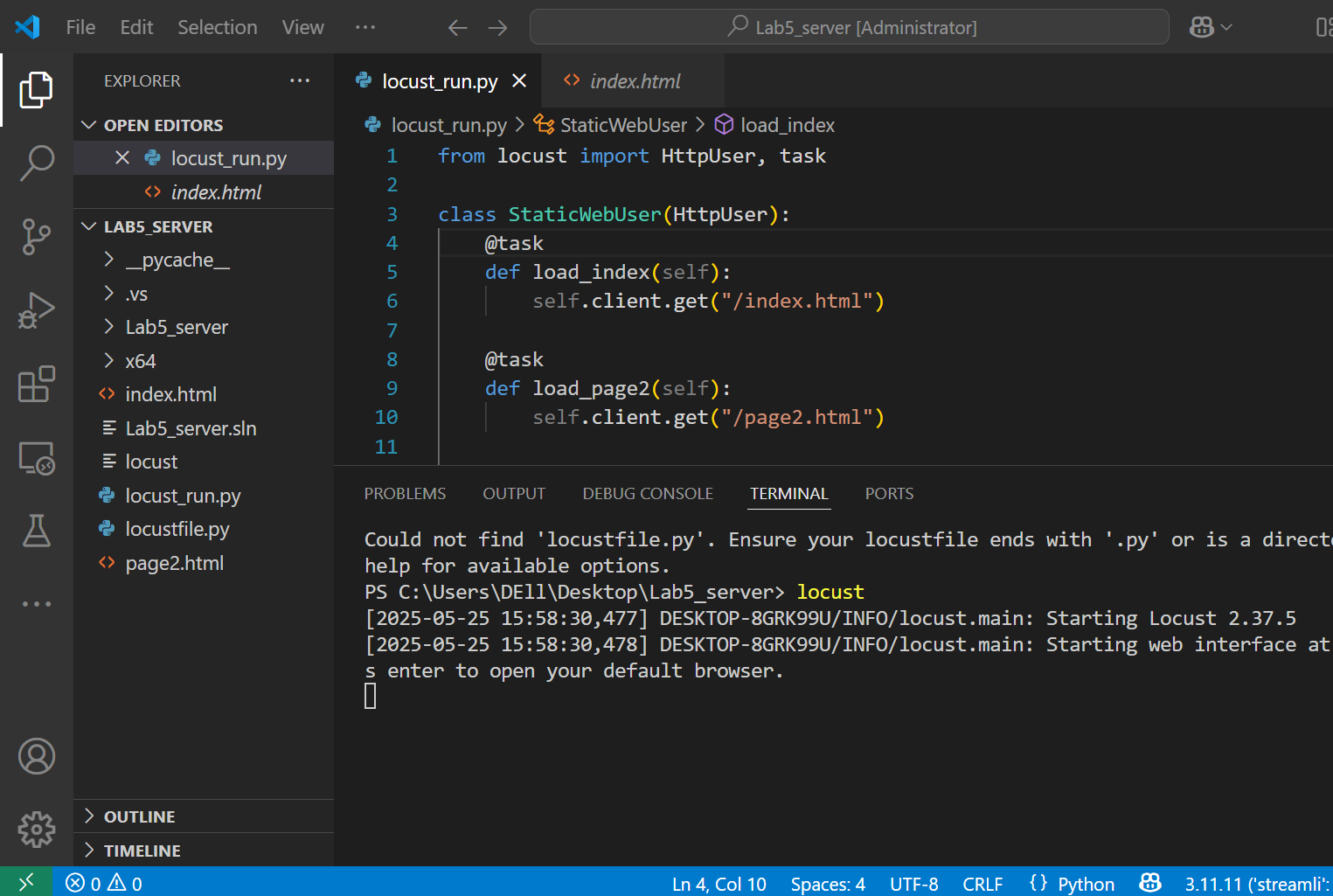


Рис. 2 - Запуск locust в терміналі

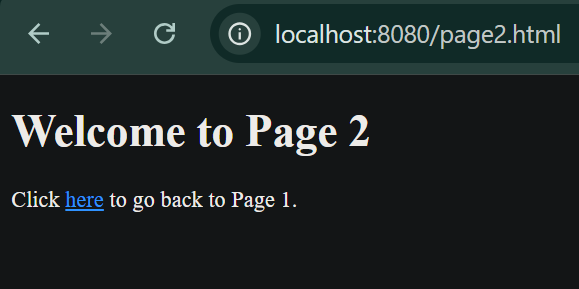


Рис. 3 - Запуск локальної веб-сторінки page2

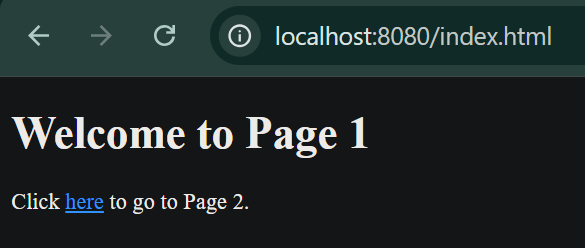


Рис. 4 - Запуск локальної веб-сторінки index

Виконаємо перший запуск із стандартними налаштуваннями: 1 підключення щосекунди

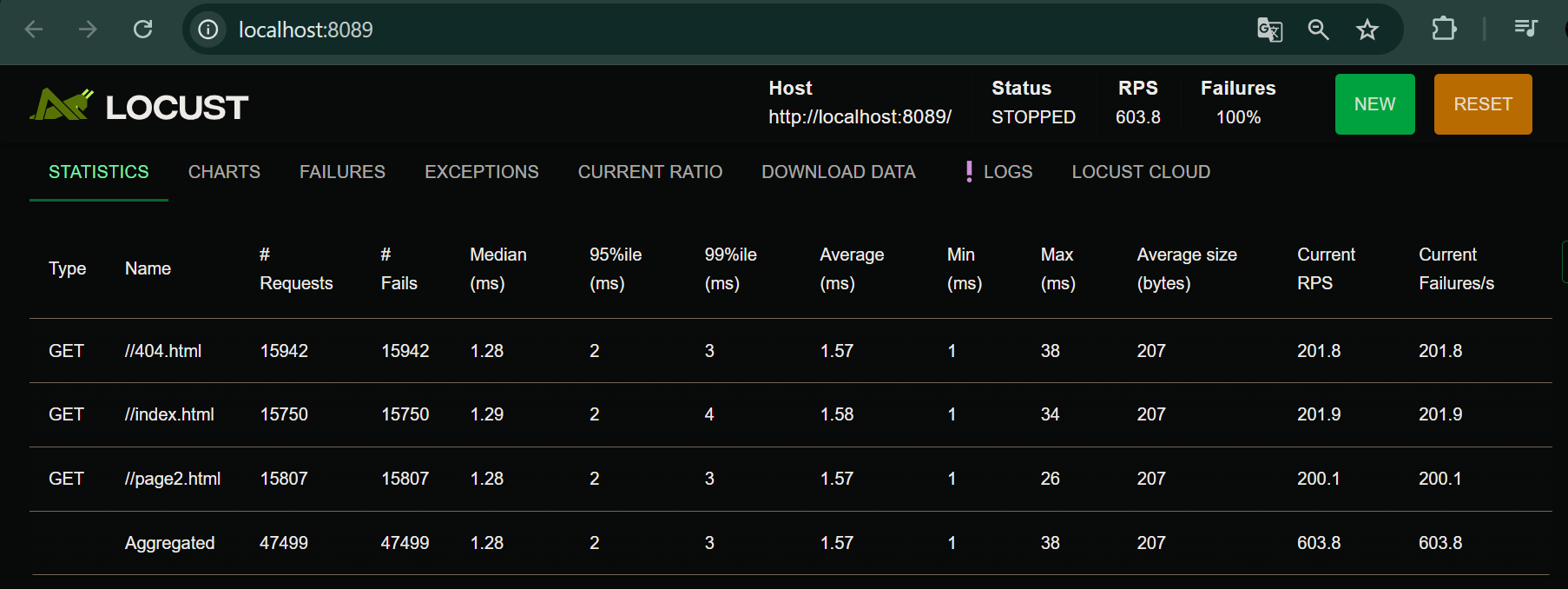


Рис. 5 - Статистика першого запуску locust

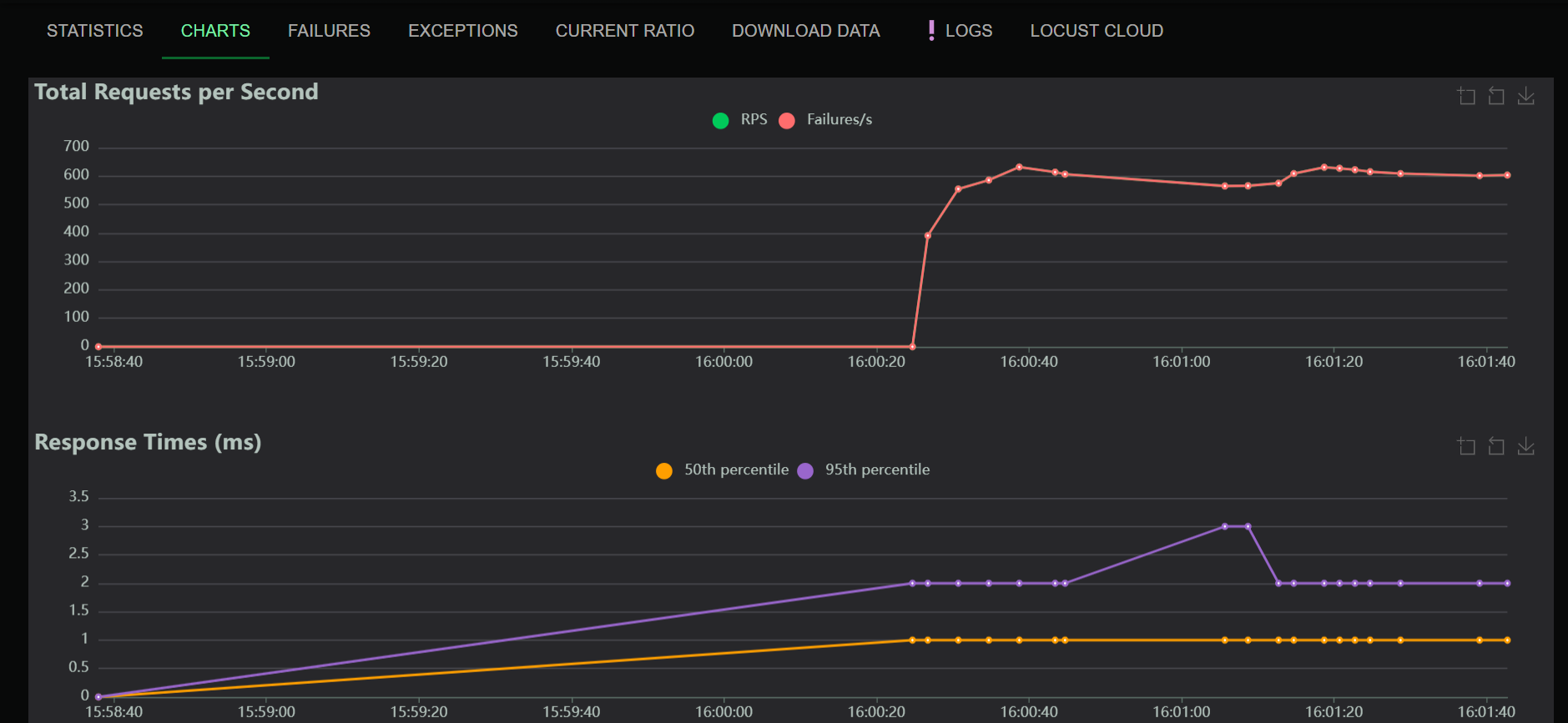


Рис. 6 - Статистика першого запуску locust

Бачимо, що при попередньому запиті 100% fail підключення. Як виявилося, це відбулося через некоректну локацію розміщення файлів веб сторінок. Змінимо шлях, а саме помістимо у папку запуску сервера і запустимо нове тестування:

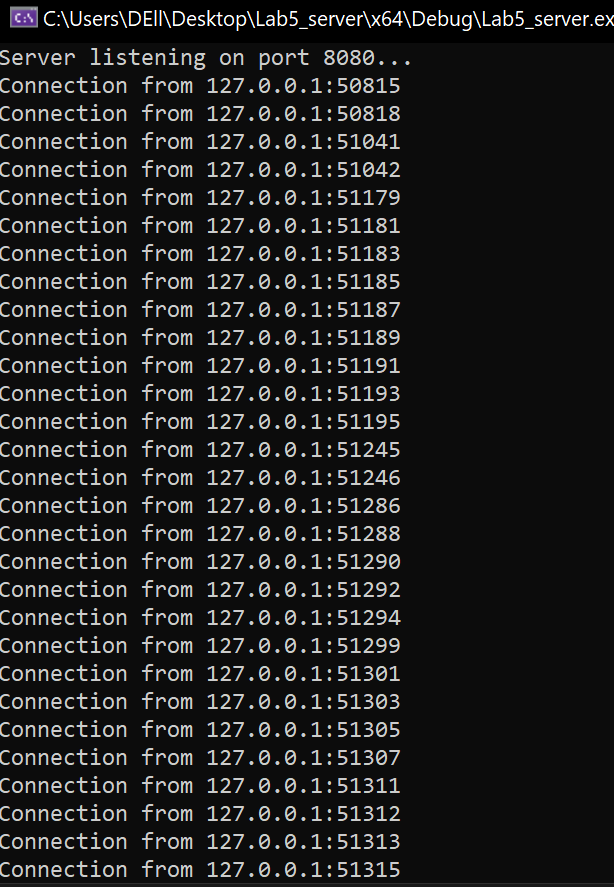


Рис. 7 - Консоль локального серверу із підключеннями locust

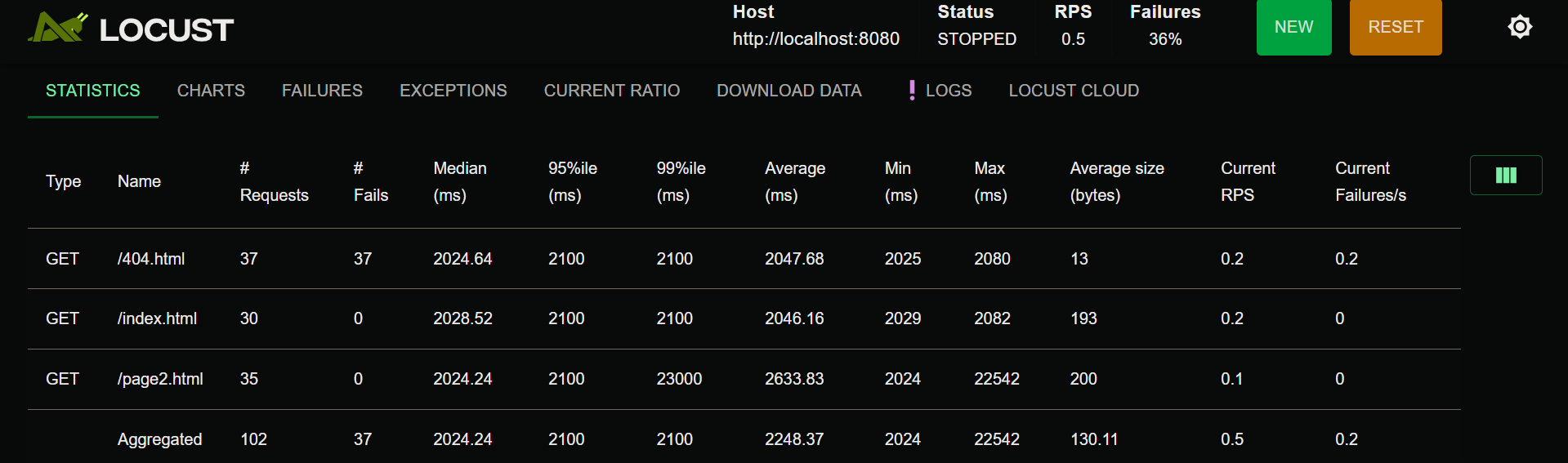


Рис. 8 - Статистика другого запуску locust

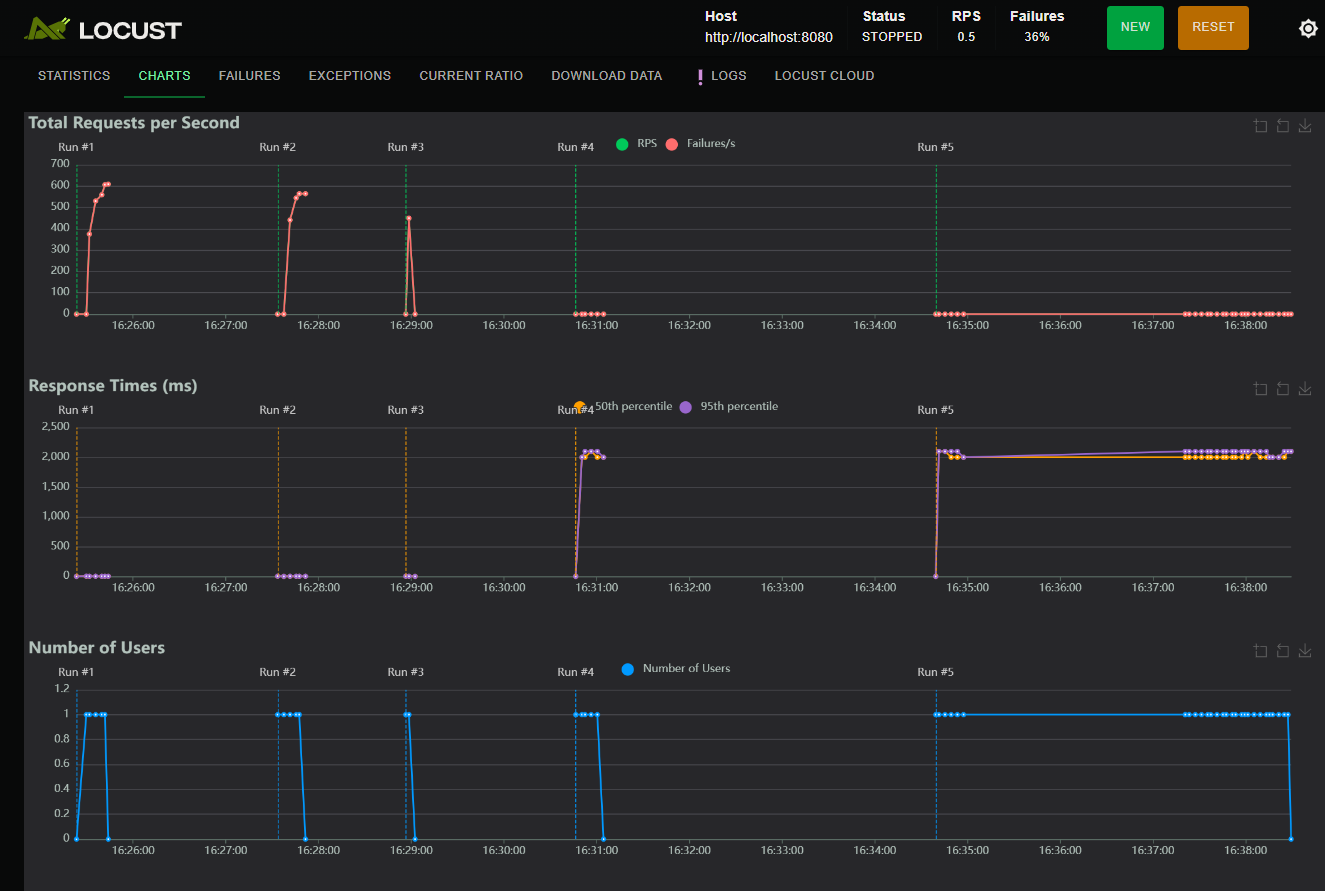


Рис. 9 - Статистика другого запуску locust

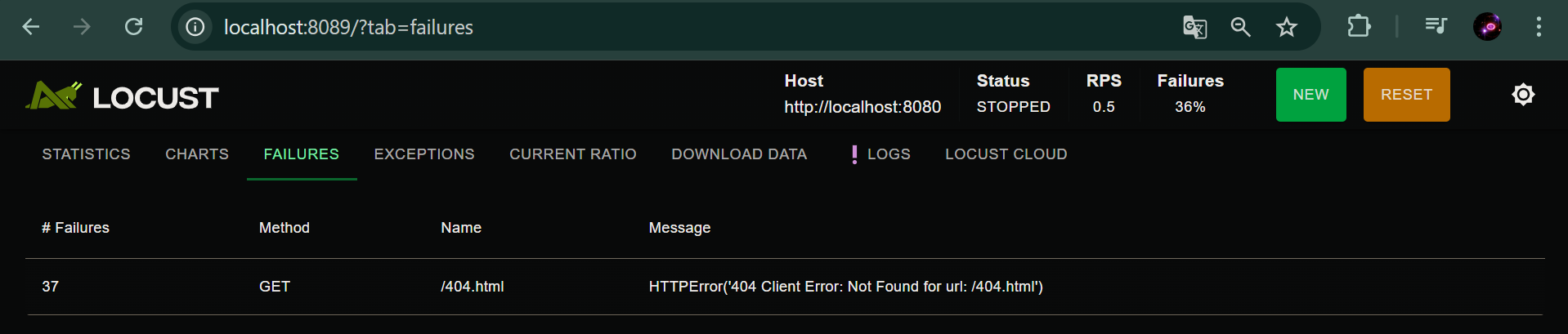


Рис. 10 - Статистика другого запуску locust

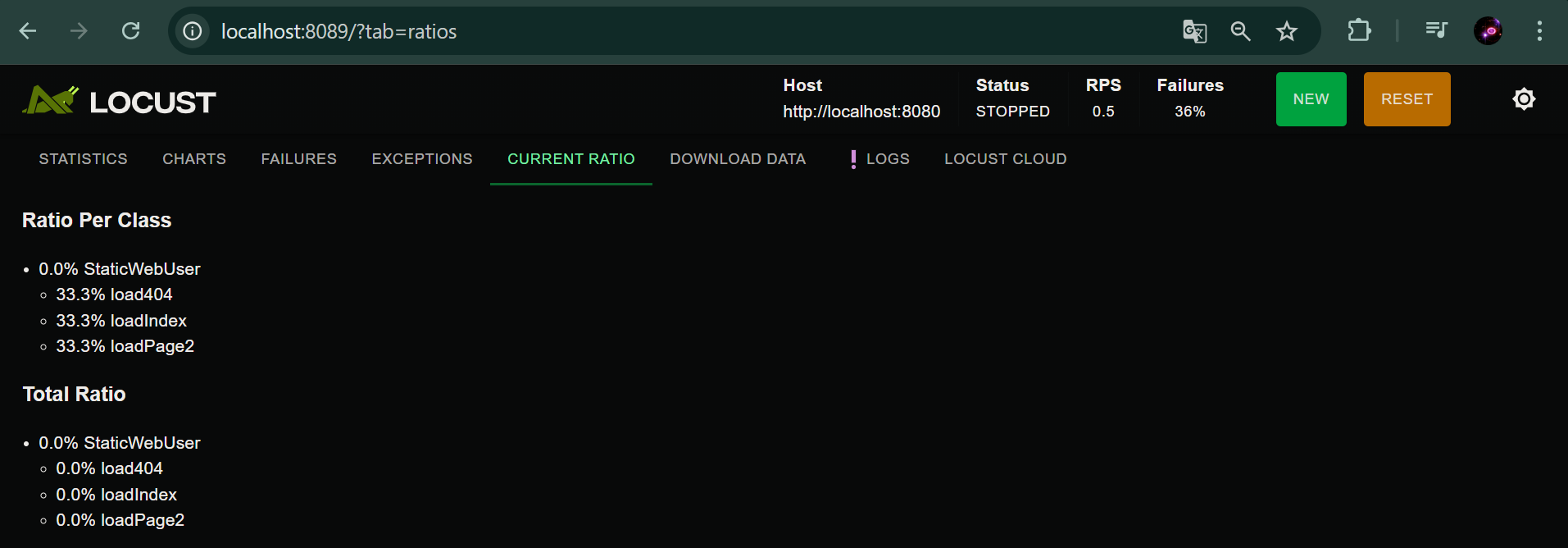


Рис. 11 - Статистика другого запуску locust

Виконаємо третій тест із великою кількістю підключень:

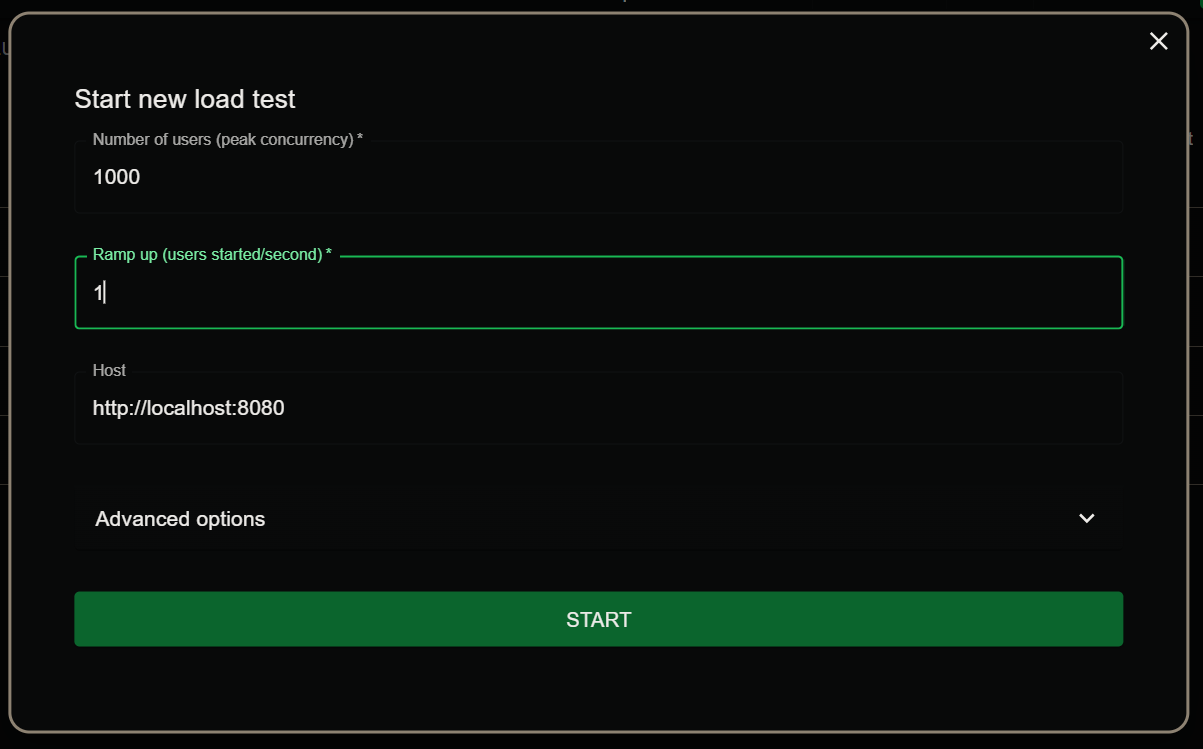


Рис. 12 - Налаштування третього запуску locust

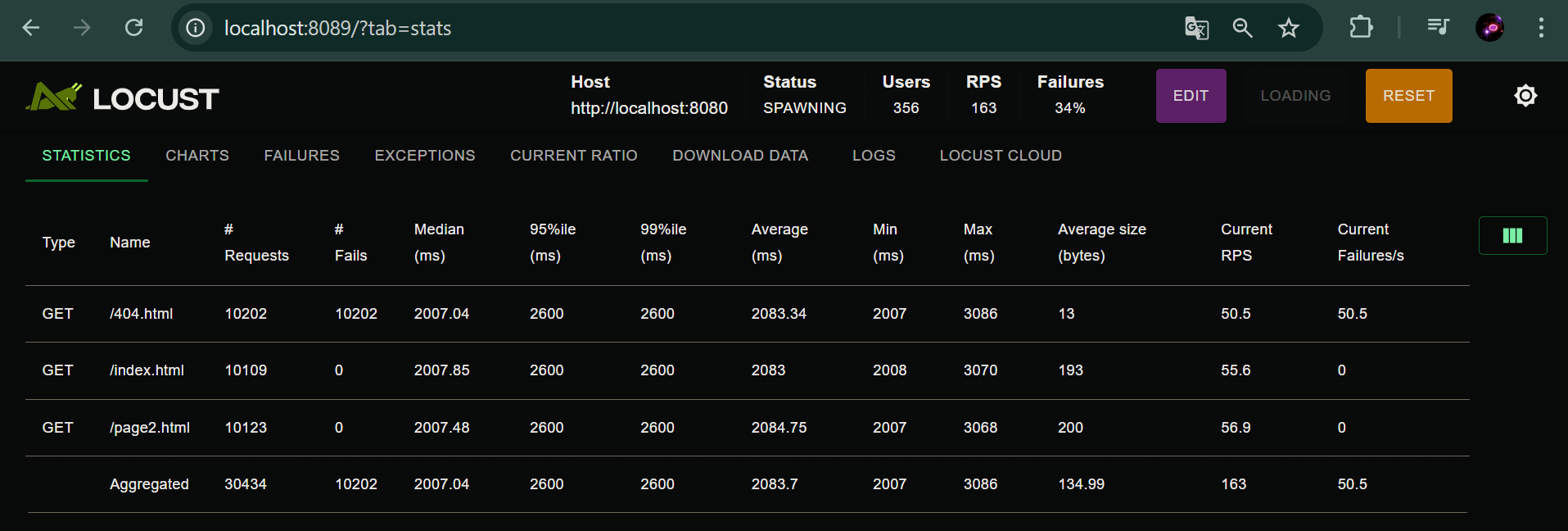


Рис. 13 - Статистика третього запуску locust

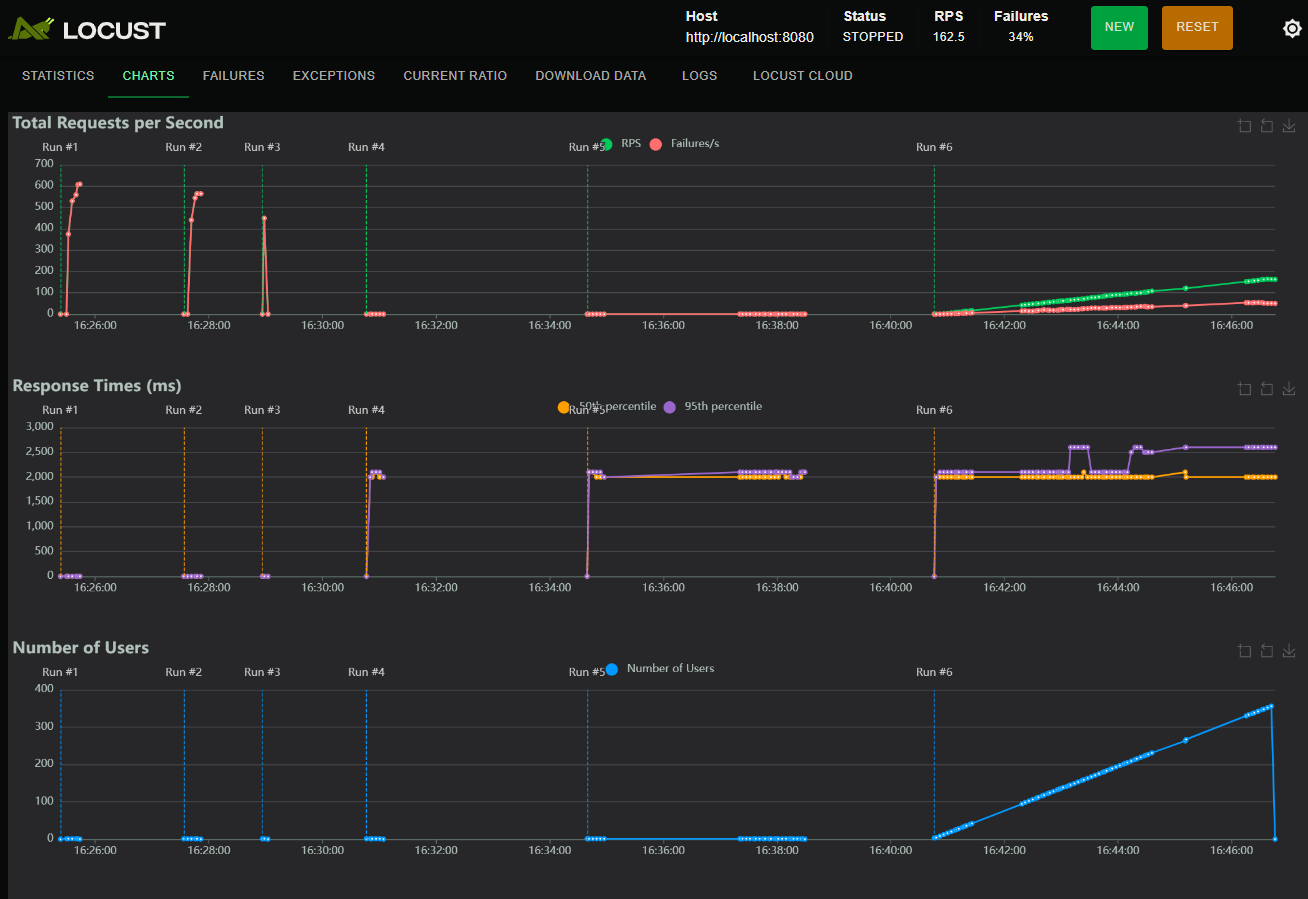


Рис. 14 - Статистика третього запуску locust

**Реалізація неблокуючого серверу:**

### Що таке неблокуючий сервер?

У звичайному (блокуючому) сервері виклик accept(), recv(), send() "блокує" потік — тобто програма чекає, поки з'єднання не прийде або поки не отримають дані, і не може робити нічого іншого в цей час. Це погано, якщо треба одночасно обробляти багато клієнтів.

Неблокуючий сервер дозволяє програмі продовжувати виконання, не чекаючи на завершення операції введення/виводу. Це робить сервер більш продуктивним і масштабованим.

Для цього існує спеціальна технологія — **Overlapped I/O** ("перекривне" введення/виведення)

* Замість звичайного socket(), створюємо сокет через WSASocket() з прапором WSA\_FLAG\_OVERLAPPED
* Використовуємо функції WSARecv() та WSASend() з додатковими параметрами для асинхронної роботи
* Передаємо в ці функції структуру OVERLAPPED — це як "маркери" для операції, які відслідковують її стан
* Коли операція завершується, Windows викликає спеціальну callback-функцію (completion routine), наприклад, WorkerRoutine, куди приходять результати операції
* Тобто сервер НЕ чекає (не блокується), а одразу може працювати з іншими клієнтами

Код для неблокуючого серверу:

#include <iostream>

#include <fstream>

#include <string>

#include <winsock2.h>

#include <ws2tcpip.h>

#pragma comment(lib, "ws2\_32.lib")

#define PORT 8080

#define MAX\_BUFFER\_SIZE 4096

struct ClientContext {

SOCKET socket;

OVERLAPPED overlapped;

WSABUF dataBuffer;

char buffer[MAX\_BUFFER\_SIZE];

};

void printLastError(const char\* msg) {

int errCode = WSAGetLastError();

std::cerr << msg << " Error code: " << errCode << std::endl;

}

void sendResponse(SOCKET clientSocket, const std::string& content, const std::string& contentType = "text/html") {

std::string httpResponse =

"HTTP/1.1 200 OK\r\n"

"Content-Length: " + std::to\_string(content.size()) + "\r\n"

"Content-Type: " + contentType + "\r\n"

"Connection: close\r\n"

"\r\n" + content;

send(clientSocket, httpResponse.c\_str(), static\_cast<int>(httpResponse.size()), 0);

}

void sendNotFound(SOCKET clientSocket) {

std::string notFound =

"HTTP/1.1 404 Not Found\r\n"

"Content-Length: 13\r\n"

"Content-Type: text/plain\r\n"

"Connection: close\r\n"

"\r\n404 Not Found";

send(clientSocket, notFound.c\_str(), static\_cast<int>(notFound.size()), 0);

}

std::string loadFileContent(const std::string& filePath) {

std::ifstream file(filePath, std::ios::binary);

if (!file) return "";

return std::string((std::istreambuf\_iterator<char>(file)), std::istreambuf\_iterator<char>());

}

std::string getRequestPath(const std::string& request) {

size\_t methodPos = request.find("GET ");

if (methodPos == std::string::npos)

return "";

size\_t pathStart = methodPos + 4;

size\_t pathEnd = request.find(' ', pathStart);

if (pathEnd == std::string::npos)

return "";

return request.substr(pathStart, pathEnd - pathStart);

}

void handleRequest(ClientContext\* clientContext) {

std::string request(clientContext->buffer);

std::string path = getRequestPath(request);

if (path.empty()) {

sendNotFound(clientContext->socket);

closesocket(clientContext->socket);

delete clientContext;

return;

}

while (path.find("//") != std::string::npos) {

path.replace(path.find("//"), 2, "/");

}

std::string requestedFile;

if (path == "/" || path == "/index.html") {

requestedFile = "index.html";

}

else if (path == "/page2.html") {

requestedFile = "page2.html";

}

else {

sendNotFound(clientContext->socket);

closesocket(clientContext->socket);

delete clientContext;

return;

}

std::string content = loadFileContent(requestedFile);

if (content.empty()) {

sendNotFound(clientContext->socket);

}

else {

sendResponse(clientContext->socket, content);

}

closesocket(clientContext->socket);

delete clientContext;

}

void CALLBACK WorkerRoutine(DWORD errorCode, DWORD bytesTransferred, LPWSAOVERLAPPED overlapped, DWORD flags) {

ClientContext\* clientContext = reinterpret\_cast<ClientContext\*>(overlapped);

if (errorCode != 0 || bytesTransferred == 0) {

closesocket(clientContext->socket);

delete clientContext;

return;

}

clientContext->buffer[bytesTransferred] = '\0';

handleRequest(clientContext);

}

int main() {

WSADATA wsaData;

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

std::cerr << "WSAStartup failed.\n";

return 1;

}

SOCKET serverSocket = WSASocket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP, NULL, 0, WSA\_FLAG\_OVERLAPPED);

if (serverSocket == INVALID\_SOCKET) {

printLastError("Socket creation failed.");

WSACleanup();

return 1;

}

sockaddr\_in serverAddr{};

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_addr.s\_addr = INADDR\_ANY;

serverAddr.sin\_port = htons(PORT);

if (bind(serverSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {

printLastError("Bind failed.");

closesocket(serverSocket);

WSACleanup();

return 1;

}

if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR) {

printLastError("Listen failed.");

closesocket(serverSocket);

WSACleanup();

return 1;

}

std::cout << "Server listening on port " << PORT << "...\n";

while (true) {

sockaddr\_in clientAddr{};

int clientAddrLen = sizeof(clientAddr);

ClientContext\* clientContext = new ClientContext{};

clientContext->socket = accept(serverSocket, (sockaddr\*)&clientAddr, &clientAddrLen);

if (clientContext->socket == INVALID\_SOCKET) {

printLastError("Accept failed.");

delete clientContext;

continue;

}

char clientIP[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &(clientAddr.sin\_addr), clientIP, INET\_ADDRSTRLEN);

std::cout << "Connection from " << clientIP << ":" << ntohs(clientAddr.sin\_port) << "\n";

ZeroMemory(&clientContext->overlapped, sizeof(OVERLAPPED));

clientContext->dataBuffer.len = MAX\_BUFFER\_SIZE - 1;

clientContext->dataBuffer.buf = clientContext->buffer;

DWORD flags = 0;

DWORD bytesReceived = 0;

int res = WSARecv(clientContext->socket, &clientContext->dataBuffer, 1, &bytesReceived, &flags, &clientContext->overlapped, WorkerRoutine);

if (res == SOCKET\_ERROR) {

int lastErr = WSAGetLastError();

if (lastErr != WSA\_IO\_PENDING) {

printLastError("WSARecv failed.");

closesocket(clientContext->socket);

delete clientContext;

}

}

}

closesocket(serverSocket);

WSACleanup();

return 0;

}

**Висновок**

В ході виконання лабораторної роботи було успішно розроблено простий HTTP-сервер для хостингу статичного веб-контенту, що підтримує багатопотокову обробку GET-запитів відповідно до протоколу HTTP/1.1. Сервер коректно обробляє запити на існуючі сторінки (index.html, page2.html) та видає правильну відповідь 404 для неіснуючих ресурсів.

За допомогою інструменту навантажувального тестування Locust було проведено тестування серверу під різним числом одночасних клієнтів. Визначено максимальну кількість одночасних підключень, при якій сервер стабільно працює без помилок. Виявлено, що подальше збільшення навантаження призводить до зростання кількості помилок (fail) через обмеження блокуючої природи базової реалізації.

Для підвищення продуктивності та масштабованості було розглянуто варіант реалізації неблокуючого сервера з використанням асинхронних викликів (Overlapped I/O) на платформі Windows. Такий підхід дозволяє обробляти більше одночасних клієнтів, оскільки потік не блокується при очікуванні операцій введення/виведення.

Подальші шляхи покращення роботи сервера можуть включати:

* Перехід на повністю асинхронний або мультипотоковий дизайн із використанням пулу потоків (як у лабі 4)
* Використання ефективних механізмів кешування статичних файлів.
* Оптимізацію обробки мережевих подій за допомогою бібліотек високого рівня (asio, libuv).
* Використання сучасних протоколів, таких як HTTP/2, для підвищення швидкодії.

Посилання на репозиторій із матеріалами - [GitHub](https://github.com/anton265463/Labs_Parellel_Computing)