01.12.2024

Zespół 36

Mateusz Łukasiewicz

Rafał Celiński

Przemysław Walecki

Programowanie sieciowe

Zadanie 2

Z 2 Komunikacja TCP

Napisz zestaw dwóch programów – klienta i serwera komunikujących się poprzez TCP. Transmitowany strumień danych powinien być stosunkowo duży, nie mniej niż 100 kB.

Klient powinien wysyłać do serwera strumień danych w pętli (tzn. danych powinno być „dużo", minimum rzędu kilkuset KB). Serwer powinien odbierać dane, ale między odczytami realizować sztuczne opóźnienie (np. przy pomocy funkcji sleep()). W ten sposób symulujemy zjawisko odbiorcy, który „nie nadąża" za szybkim nadawcą. Stos TCP będzie spowalniał nadawcę, aby uniknąć tracenia danych. Należy zidentyfikować objawy tego zjawiska po stronie klienta (dodając pomiar i logowanie czasu) i krótko przedstawić swoje wnioski poparte uzyskanymi statystykami czasowymi. Wskazane jest też przeprowadzenie eksperymentu z różnymi rozmiarami bufora nadawczego po stronie klienta (np. 100 B, 1 KB, 10 KB)

To zadanie należy wykonać, korzystając z kodu klienta i serwera napisanych w języku C.

Konfiguracja

Adres IP serwera – 172.21.36.2

Adres IP klienta – 172.21.36.3

Porty – 8080:8080 (TCP)

Obrazy Docker

Obraz dla Pythona - python:3-slim

Obraz dla GCC - gcc:4.9

Rozwiązanie

Programy klienta i serwera zrealizowaliśmy w języku C++.

Są to proste implementacje komunikacji za pomocą protokołu TCP.
Zadaniem klienta jest wysyłać na serwer coraz większe wiadomości najszybciej jak może. Czas wysyłania wiadomości jest mierzony i wypisywany w konsoli.
Zadaniem serwera jest odbieranie tych pakietów w sposób niewystarczająco szybki powodując przy tym opóźnienie.

Klient

Potrzebujemy funkcji podającej nam czas do logowania go

```
long long get_current_time_in_ms() {
    struct timeval time;
    gettimeofday(&time, NULL);
    return time.tv_sec * 1000LL + time.tv_usec / 1000;
}
```

Tworzymy socket i zestawiamy połączenie TCP (funkcja *connect*)

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    std::cout << "Failed to create socket" << std::endl;
    return -1;
}

struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

if(connect(sock, (struct sockaddr*)& server_addr, sizeof(server_addr)) < 0) {
    std::cout << "Failed to connect" << std::endl;
    return -1;
}
```

W pętli wysyłamy po 100 wiadomości o rozmiarach kolejno (200 B, 2kB, 20kB, 200kB).

Tuż przed wywołaniem funkcji *send* zaczynamy pomiar czasu i kończymy go po poprawnym wysłaniu wiadomości.

```
int message_size = 200;

while(message_size <= MAX_MESSAGE_SIZE)
{
    char* message = new char[message_size];
    memset(message, 'M', message_size);
    for (int i = 0; i < 100; i++)
    {
        long long start_time = get_current_time_in_ms();
        int bytes_send = send(sock, message, message_size, 0);
        if (bytes_send < 0) {
            std::cout << "Couldn't send message" << std::endl;
            close(sock);
            delete[] message;
            return -1;
        }

        long long end_time = get_current_time_in_ms();
        long long difference = end_time - start_time;
        std::cout << "Message " << i << " of size " << message_size << " B sent in " << difference << " ms" << std::endl;
    }
    delete[] message;
    message_size *= 10;
}
std::cout << "Sent all messages" << std::endl;
close(sock);
return 0;
```

Serwer

Tworzymy i konfigurujemy gniazdo oraz obsługujemy połączenie z klientem

(po kolei funkcje *bind, listen, accept)*

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    std::cout << "Failed to create socket" << std::endl;
    return -1;
}

struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

if(bind(sock, (struct sockaddr*)& server_addr, sizeof(server_addr)) < 0) {
    std::cout << "Failed to bind" << std::endl;
    return -1;
}

if(listen(sock, 0) < 0) {
    std::cout << "Failed to listen" << std::endl;
    return -1;
}

std::cout << "Server listening on " << SERVER_IP << ":" << SERVER_PORT << std::endl;
```

```
client_socket = accept(sock, (struct sockaddr*)& client_address, &addr_len);
if (client_socket < 0) {
    std::cout << "Failed to accept" << std::endl;
    close(sock);
    return -1;
}

close(sock);
std::cout << "Client connected - closing socket to prevent other connections" << std::endl;
```

Po połączeniu pierwszego klienta zamykamy gniazdo ponieważ jesteśmy przygotowani do obsługi tylko jednego klienta.

Następnie w pętli odbieramy wiadomości od klienta wprowadzając opóźnienie do komunikacji

```
char buffer[BUFFER_SIZE] = {0};
while (true) {
    int bytes_received = read(client_socket, buffer, BUFFER_SIZE);
    if (bytes_received > 0) {
        std::cout << "Received message from client of size: " << bytes_received <<std::endl;
        sleep(1);
    }
    else if (bytes_received <= 0) {
        std::cout << "Error or disconnection occurred" << std::endl;
        break;
    }
}
close(client_socket);
return 0;
```

Działanie programów

Po zbudowaniu i uruchomieniu kontenerów w Dockerze uzyskaliśmy takie wyniki

```
z36_server_container  | Server listening on 172.21.36.2:8080
z36_client_container  | Connected to server
z36_client_container  | Message 0 of size 200 B sent in 0 ms
z36_server_container  | Client connected - closing socket to prevent other connections
z36_client_container  | Message 1 of size 200 B sent in 0 ms
z36_server_container  | Received message from client of size: 200
z36_client_container  | Message 2 of size 200 B sent in 0 ms
z36_client_container  | Message 3 of size 200 B sent in 0 ms
z36_client_container  | Message 4 of size 200 B sent in 0 ms
z36_client_container  | Message 5 of size 200 B sent in 0 ms
z36_client_container  | Message 6 of size 200 B sent in 0 ms
```

Odebrana została pierwsza wiadomość – następnie serwer zawiesza się na sekundę w trakcie której klient wysyła wiadomości

```
z36_client_container  | Message 28 of size 2000 B sent in 0 ms
z36_client_container  | Message 29 of size 2000 B sent in 0 ms
z36_client_container  | Message 30 of size 2000 B sent in 22 ms
z36_client_container  | Message 31 of size 2000 B sent in 0 ms
z36_client_container  | Message 32 of size 2000 B sent in 0 ms
z36_client_container  | Message 33 of size 2000 B sent in 0 ms
z36_client_container  | Message 34 of size 2000 B sent in 0 ms
```

Pierwsze nieznaczne opóźnienie pojawia się dopiero po wysłaniu trzydziestej z kolei wiadomości o rozmiarze 2000 B (w sumie po 130 wiadomościach)

```
z36_client_container   | Message 20 of size 20000 B sent in 0 ms
z36_client_container   | Message 21 of size 20000 B sent in 0 ms
z36_client_container   | Message 22 of size 20000 B sent in 0 ms
z36_server_container   | Received message from client of size: 116280
z36_client_container   | Message 23 of size 20000 B sent in 1877 ms
z36_client_container   | Message 24 of size 20000 B sent in 0 ms
z36_server_container   | Received message from client of size: 115120
z36_client_container   | Message 25 of size 20000 B sent in 0 ms
z36_client_container   | Message 26 of size 20000 B sent in 0 ms
```

Kolejne wiadomości serwer odbiera po około 220 wiadomościach. Odebrane dane nie mają typowego rozmiaru, który był cały czas wysyłany – natomiast po zsumowaniu tych dwóch powyższych wartości otrzymamy około 230 kB (dokładnie 231400 B) co wskazywałoby na to że poprzednie pojedyncze pakiety zostały scalone (po wykonaniu matematyki wychodzi, że scalone zostały wiadomości do mniej więcej 205-tej co nie powinno nas niepokoić, ponieważ sama operacja wypisywania na ekran trwa jakiś ułamek czasu).

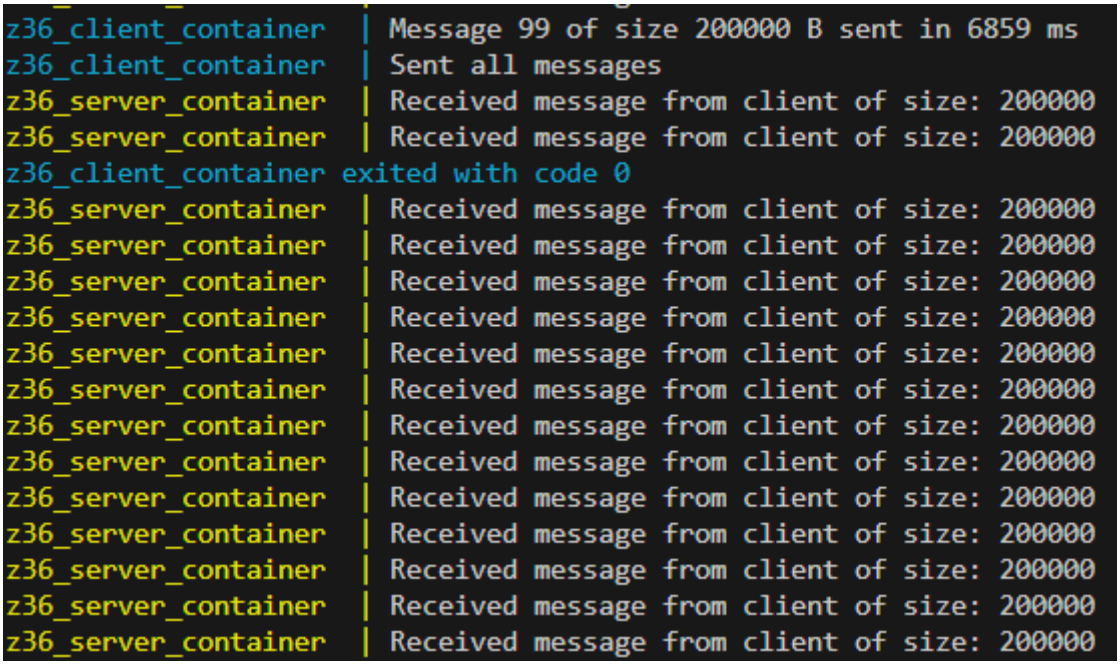
Można także zauważyć pierwsze większe opóźnienie po stronie klienta.

```
z36_client_container   | Message 32 of size 200000 B sent in 0 ms
z36_server_container   | Received message from client of size: 200000
z36_client_container   | Message 33 of size 200000 B sent in 919 ms
z36_client_container   | Message 34 of size 200000 B sent in 0 ms
z36_client_container   | Message 35 of size 200000 B sent in 0 ms
z36_client_container   | Message 36 of size 200000 B sent in 0 ms
z36_client_container   | Message 37 of size 200000 B sent in 0 ms
z36_client_container   | Message 38 of size 200000 B sent in 0 ms
z36_client_container   | Message 39 of size 200000 B sent in 0 ms
z36_client_container   | Message 40 of size 200000 B sent in 0 ms
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_server_container   | Received message from client of size: 200000
z36_client_container   | Message 41 of size 200000 B sent in 8100 ms
z36_client_container   | Message 42 of size 200000 B sent in 0 ms
z36_client_container   | Message 43 of size 200000 B sent in 0 ms
z36_client_container   | Message 44 of size 200000 B sent in 0 ms
z36_client_container   | Message 45 of size 200000 B sent in 0 ms
z36_client_container   | Message 46 of size 200000 B sent in 0 ms
z36_client_container   | Message 47 of size 200000 B sent in 0 ms
```

Przy wysyłaniu największych wiadomości da się najlepiej zauważyć działanie opóźniania klienta przez stos TCP.

Na zrzucie ekranu widać że dopiero przy 40-tej porcji danych o rozmiarze 200 kB klient zaczyna znacznie zwalniać.

Ten bufor danych będzie obsługiwany jeszcze długo po zakończeniu połączenia przez klienta.



Uzyskane wyniki były powtarzalne dla wszystkich podjętych prób.