PSI Zadanie 2

Z37 - Aleksandra Szymańska, Angelika Ostrowska, Jakub Bąba

1.12.2024

1. Treść zadania

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.

Zmodyfikować serwer tak, aby miał konstrukcję współbieżną, tj. obsługiwał każdego klienta w osobnym procesie. Przy czym:

• Dla C należy posłużyć się funkcjami fork() oraz (obowiązkowo) wait().

• Dla Pythona należy posłużyć się wątkami, do wyboru: wariant podstawowy lub skorzystanie z ThreadPoolExecutor

Przetestować dla kliku równolegle działających klientów.

2. Opis rozwiązania problemu

Zapoznaliśmy się z treścią zadania i wybraliśmy realizowanie go w języku Python.

Bazowaliśmy na kodach z poprzednich zadań i przykładowych kodach komunikacji TCP w Pythonie dostarczonych przez wykładowcę.

Najpierw zmodyfikowaliśmy kod klienta, aby korzystał z komunikacji TCP zamiast UDP, następnie zrobiliśmy to samo dla serwera. Upewniliśmy się, żeby stworzyć odpowienio duży strumień danych. Wymagany w zadaniu strumień ma wielkość co najmniej 100kB. Oznacza to 100 * 1000 B = 100 000 B = 100 000 liter zapisanych w ASCII (czyli gdzie na zakodowanie litery przypada 1 bajt). Zdecydowaliśmy się więc na wysłanie 100 000 zapętlonych liter A-Z

Przetestowaliśmy działanie standardowej komunikacji TCP. Napotkaliśmy problem z odpowiedziami serwera, który opisaliśmy w odpowiedniej sekcji.

Następnie zaimplementowaliśmy wielowątkowość, aby serwer mógł obsługiwać wielu klientów równocześnie. Użyliśmy do tego podejścia z ThreadPoolExecutor.

3. Napotkane problemy

W pierwszej wersji programu próbowaliśmy na każdy odebrany fragment zwracać odpowiedź o całkowitym rozmiarze dotychczas odebranych danych. Powodowało to jednak taki problem, że czasem kolejna odpowiedź się wysyłała, zanim pierwsza z nich została odebrana, przez co odbierany przez klienta łączny rozmiar był sklejeniem kilku cyfr stając się bardzo dużym. Zdecydowaliśmy się w związku z tym na dodanie komunikatu “DONE” pod koniec każdej wiadomości od klienta oraz wysyłanie tylko jednej odpowiedzi, pod koniec całej przesłanej wiadomości.

4. Konfiguracja testowa

Nasza sieć testowa typu bridge nazywa się „z37_network”

Adres IP podsieci to 172.21.37.0/24

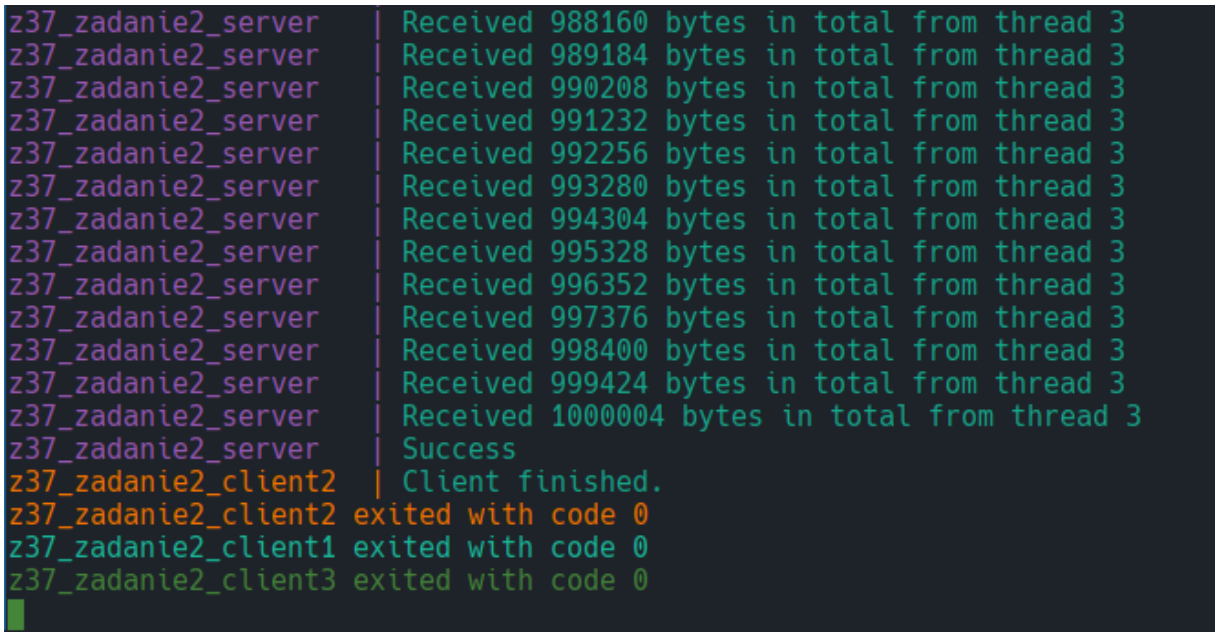
W tej sieci serwer jest lokalny i ma adres 127.0.0.1 i łaczymy się z nim na porcie 8000.

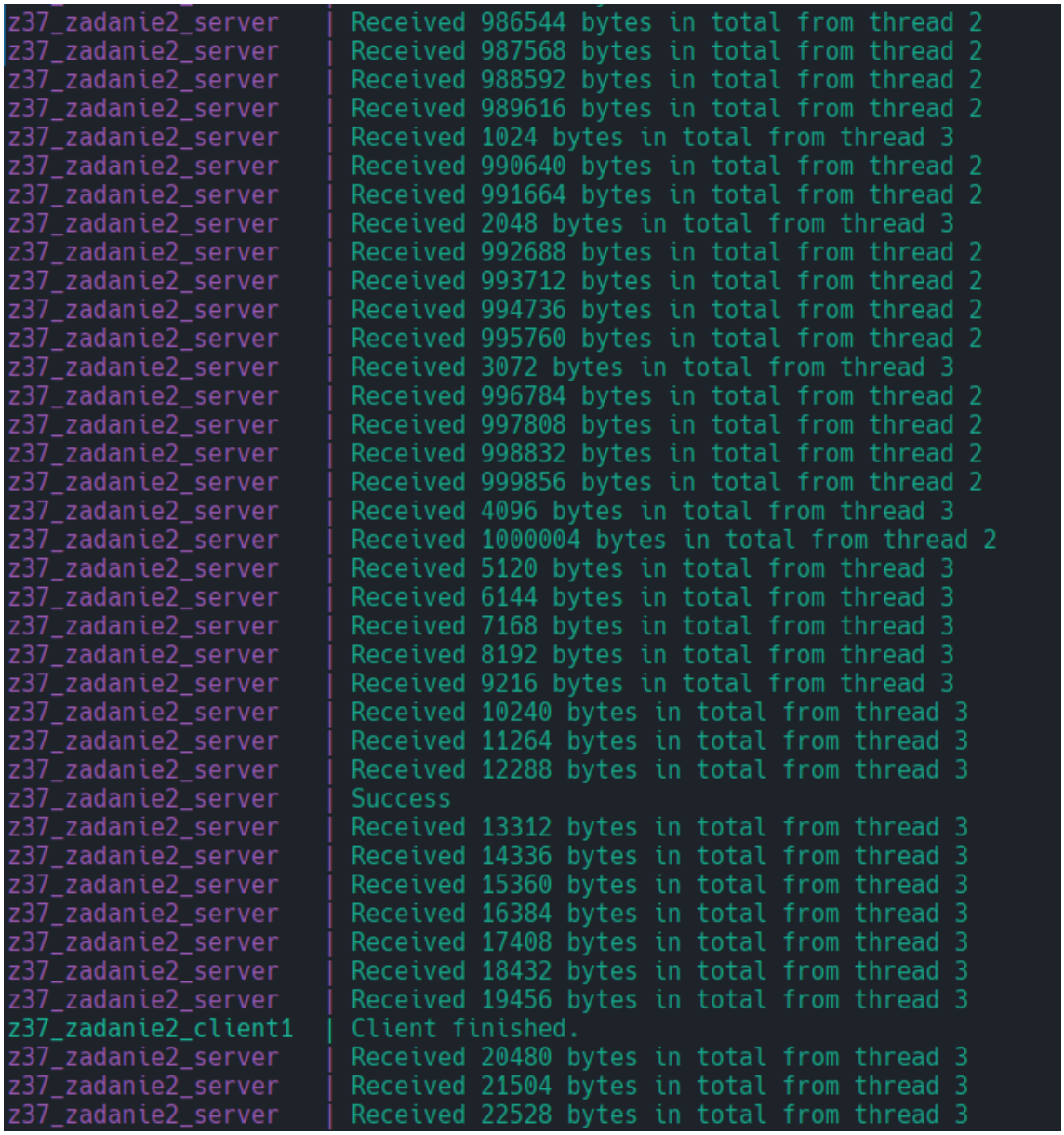
5. Testy

Po napisaniu kodu sprawdzaliśmy, czy działa on z zamierzonym efektem. Uruchamialiśmy serwer i klientów za pomocą „docker compose up --build” i testowaliśmy komunikację.

Zauważyliśmy, że program działał tak jak należy, odbierał wiadomości w kawałkach po 1024 bajty (BUFSIZE), a pod koniec zwracał informację klientowi o zakończeniu transmisji.

Thread jest zmienną zwiększaną za każdym razem, gdy nowy klient się połączy z serwerem (nie jest powiązana z nazwą kontenera dockerowego). Jak można się domyślić, w tym wypadku najpierw uruchomił się client3, później client1, a na końcu client2.





6. Wnioski końcowe

Udało nam się osiągnąć komunikację między klientem a serwerem, która odbywa się przez TCP. Zaimplementowaliśmy jednocześnie serwer tak, aby miał konstrukcję współbieżną. Udało nam się to sprawdzić za pomocą pliku docker-compose.yml, który zadbał o uruchomienie wszystkich klientów równolegle.