WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

PROGRAMOWANIE SIECIOWE
LABORATORIUM NR. 1
KOMUNIKACJA UDP
17.11.2024

Jan Lewandowski 325184
Krzysztof Sokół
Szymon Ryszka

1. TREŚĆ ZADANIA

Napisać zestaw dwóch programów – klienta i serwera wysyłające datagramy UDP. Klient jak i serwer powinien być napisany zarówno w C jak i Pythonie (4 programy).

Sprawdzić i przetestować działanie „między-platformowe", tj. klient w C z serwerem Python i vice versa. Klient wysyła, a serwer odbiera datagramy o stałym rozmiarze (rzędu kilkuset bajtów). Datagramy powinny posiadać ustaloną formę danych. Przykładowo: pierwsze dwa bajty datagramu mogą zawierać informację o jego długości, a kolejne bajty kolejne litery A-Z powtarzające się wymaganą liczbę razy (ale można przyjąć inne rozwiązanie). Odbiorca powinien weryfikować odebrany datagram i odsyłać odpowiedź o ustalonym formacie. Klient powinien wysyłać kolejne datagramy o przyrastającej wielkości np. 1, 100, 200, 1000, 2000… bajtów. Sprawdzić, jaki był maksymalny rozmiar wysłanego (przyjętego) datagramu. Ustalić z dokładnością do jednego bajta jak duży datagram jest obsługiwany. Wyjaśnić.

2. OPIS ROZWIĄZANIA PROBLEMU

2.1 Klient i serwer w C

Komunikacja między klientem a serwerem opiera się na protokole UDP (User Datagram Protocol). Protokół ten jest lekki, niewymagający nawiązywania trwałego połączenia między komunikującymi się urządzeniami, co sprawia, że jest szybki, ale nie gwarantuje dostarczenia pakietów. Poniżej przedstawiono krok po kroku, jak działa komunikacja w tym systemie.

2.1.1Tworzenie gniazd (sockets)

Klient i serwer tworzą swoje gniazda UDP przy użyciu funkcji socket():

- AF_INET wskazuje na użycie protokołu IPv4.
- SOCK_DGRAM oznacza użycie protokołu UDP.

Serwer przypisuje swój port (8080) i adres 172.21.32.2 do gniazda za pomocą funkcji bind()

Klient nie używa funkcji bind() do przypisania gniazda, ponieważ w Dockerze adres IP 172.21.32.3 zostanie przypisany w momencie uruchamiania kontenera.

2.1.2Wysyłanie wiadomości przez klienta

Klient generuje wiadomość za pomocą funkcji msg_generator, której treść zależy od iteracji (zaczyna się od ciągu znaków z32, a następne znaki są generowane jako kolejne litery alfabetu).Wiadomość jest wysyłana do serwera za pomocą funkcji sendto():

- Adres docelowy (struct sockaddr_in) zawiera IP (172.21.32.2) i port serwera (8080).
- UDP nie wymaga nawiązywania połączenia - klient po prostu wysyła dane.

2.1.3 Odbieranie wiadomości przez serwer

Serwer odbiera wiadomości od klienta przy użyciu funkcji recvfrom(). Ta funkcja:

- Przechwytuje dane wysłane na jego gniazdo.
- Pobiera także adres IP i port nadawcy (klienta), dzięki czemu serwer wie, gdzie wysłać odpowiedź.

- Serwer wypisuje na konsolę informacje o:
    - Rozmiarze odebranej wiadomości.
    - Pierwszych trzech znakach wiadomości, które identyfikują nadawcę (z32)

2.1.4 Wysyłanie odpowiedzi przez serwer

Serwer odpowiada klientowi, wysyłając wiadomość zwrotną (stałą wiadomość "Confirmation sent") przy użyciu sendto():

- Adres klienta (IP i port) pochodzi z odebranej wiadomości.
- Dzięki temu wiadomość trafia do odpowiedniego klienta, nawet jeśli w systemie działa wielu klientów.

2.1.5 Cykliczna wymiana danych

- Klient w pętli wysyła kolejne wiadomości o rosnącej długości (zaczyna od 1 znaku, zwiększa o 100 przy każdej iteracji).
- Serwer w pętli odbiera wiadomości i odpowiada na każdą z nich.

- Obie aplikacje działają równolegle przez 10 sekund, a następnie kończą swoje działanie.

2.2 Klient z serwer w pythonie

2.2.1Opis serwera

W przypadku użycia Dockera serwer działa w kontenerze z przypisanym statycznym adresem IP oraz nazwą sieci.

Tworzenie gniazda UDP:

- Serwer tworzy gniazdo UDP (*socket.socket(socket.AF_INET, socket.SOCK_DGRAM)*) i wiąże je z podanym adresem IP i portem (*s.bind((HOST, PORT))*).
- W środowisku Dockera HOST jest ustalany na podstawie aliasu sieciowego, np. z32_server_py

Odbieranie danych:

- Serwer odbiera wiadomości za pomocą recvfrom(size) w pętli, gdzie size to maksymalny rozmiar wiadomości.
- Jeśli wiadomość różni się od poprzedniej, serwer:
    - Sprawdza, czy pierwsze trzy znaki wiadomości to z32 (ASCII kodowanie znaków z, 3, 2).
    - Wyświetla rozmiar odebranej wiadomości.
    - Wysyła potwierdzenie do klienta, w którym przekazuje liczbę odebranych bajtów (*s.sendto(str.encode(f"{size_rec}"), ret_addr)*).

Przerwanie działania:

- Serwer kończy pracę, jeśli wystąpi błąd odczytu wiadomości lub czas oczekiwania (timeout) na dane przekroczy 10 sekund.

2.2.2 Opis klienta

Klient dynamicznie wysyła wiadomości o zwiększającym się rozmiarze, sprawdzając, jaki największy rozmiar wiadomości może być obsłużony przez serwer.

Tworzenie gniazda UDP i połączenie:

- Klient tworzy gniazdo UDP *(socket.socket(socket.AF_INET, socket.SOCK_DGRAM)*) i łączy się z serwerem (*s.connect((HOST, PORT))*).

Dynamiczne dopasowanie rozmiaru wiadomości:

- *Inicjacja zmiennych:*
  - x – aktualny rozmiar wiadomości (zaczyna od 2 bajtów).
  - x_o – poprzedni rozmiar wiadomości, dla porównania.
  - a – współczynnik wzrostu rozmiaru wiadomości (domyślnie 2).
- W pętli:
  - Generuje wiadomość o długości x. Dla wiadomości o długości ≥ 3 dodaje na początku znakowy ciąg z32.
  - Wysyła wiadomość do serwera i czeka na odpowiedź z rozmiarem odebranych danych.
  - Jeśli odebrany rozmiar różni się od wysłanego, klient zmniejsza wartość x (zmniejszenie kroku iteracji).
  - W przeciwnym razie zwiększa rozmiar wiadomości (multiplikuje przez a).
- Pętla kończy się, gdy różnica między aktualnym a poprzednim rozmiarem *(abs(x - x_o))* jest mniejsza lub równa 1.

Obsługa odpowiedzi serwera:

- Klient odbiera odpowiedź serwera za pomocą recvfrom(). Jeśli odpowiedź różni się od długości wysłanej wiadomości, zgłasza błąd weryfikacji.

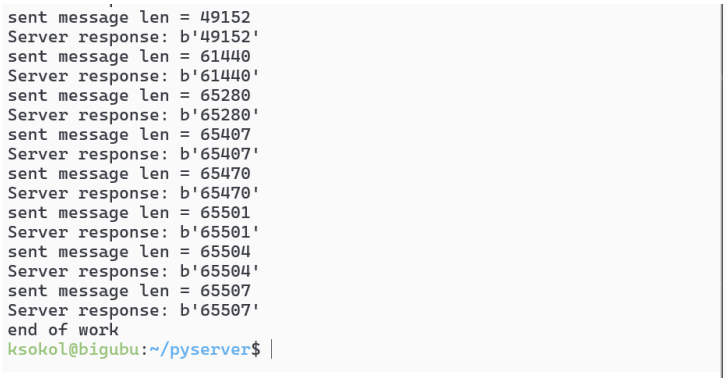
Zakończenie działania:

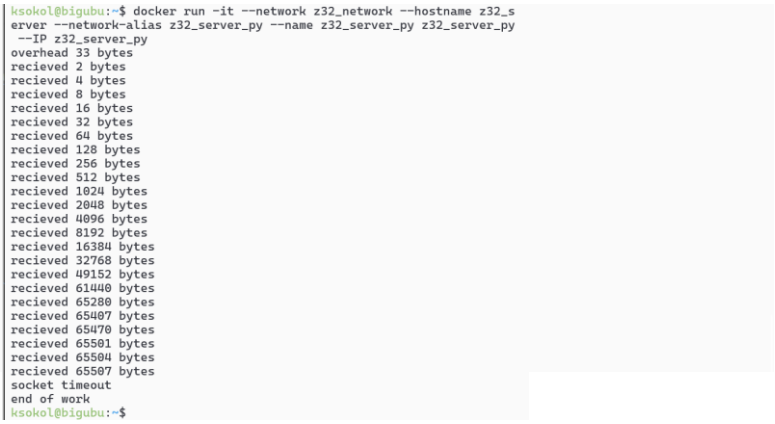
- Gniazdo jest zamykane po zakończeniu iteracji (s.close())

3.TESTOWANIE

3.1 TESTOWANIE PY Z PY

```
ksokol@bigubu:~/pyserver$ docker run -it --network z32_network --hostname z32_client_py --network-alias z32_client_py --name z32_client_py z32_
client_py --IP z32_server_py
created socket
sent message len = 2
Server response: b'2'
sent message len = 4
Server response: b'4'
sent message len = 8
Server response: b'8'
sent message len = 16
Server response: b'16'
sent message len = 32
Server response: b'32'
sent message len = 64
Server response: b'64'
sent message len = 128
Server response: b'128'
sent message len = 256
Server response: b'256'
sent message len = 512
Server response: b'512'
sent message len = 1024
Server response: b'1024'
sent message len = 2048
Server response: b'2048'
sent message len = 4096
Server response: b'4096'
sent message len = 8192
Server response: b'8192'
sent message len = 16384
Server response: b'16384'
sent message len = 32768
```



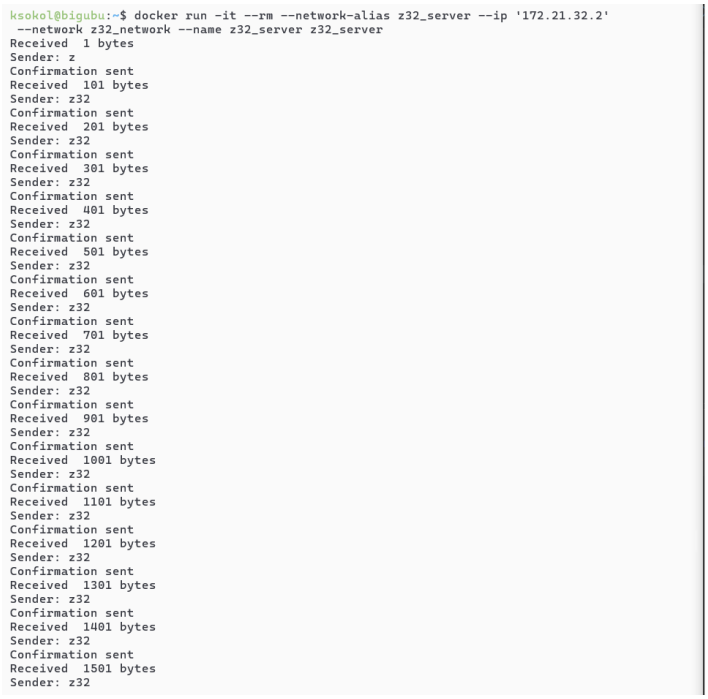


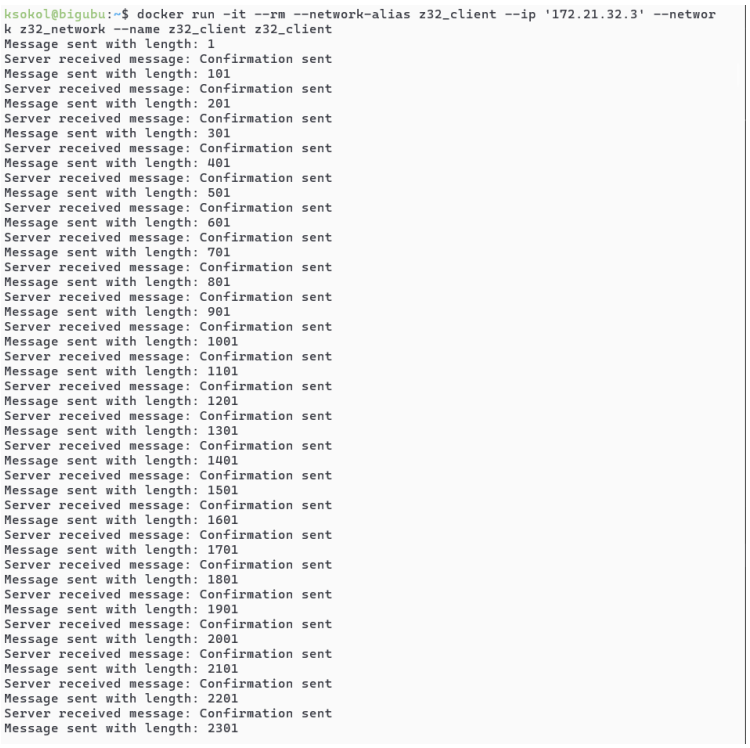
Tak wygląda efekt działania programów. Gdy nasz napis osiągnął długość 65507 bajtów, komunikacja została przerwana. Takie zachowanie serwera jest sensowne. Pakiet UDP składa się z nagłówka i danych, jego maksymalna długość to 65 515 bajtów. Po prostu przekroczyliśmy limit danych które można przesłać w pojedynczym pakiecie. Komendy dockerowe użyte podczas testów:

(docker run -it -d --network z32_network --hostname z32_server --network-alias z32_server_py --name z32_server_py z32_server_py --IP z32_server_py)"

(docker run -it -d --network z32_network --hostname z32_client_py --network-alias z32_client_py --name z32_client_py z32_client_py --IP z32_server_py)"

3.2 TESTOWANIE C Z C





Zachowanie programów było podobne jak do tego w pythonie (65501) Komendy dockerowe:

docker run -it -d --network-alias z32_server --ip '172.21.32.2' --network z32_network --name z32_server z32_server)"

docker run -it -d --network-alias z32_client --ip '172.21.32.3' --network z32_network --name z32_client z32_client)"

3.3 CROSS TESTY

```
Message sent with length: 65101
Server received message: 65101
Message sent with length: 65201
Server received message: 65201
Message sent with length: 65301
Server received message: 65301
Message sent with length: 65401
Server received message: 65401
Message sent with length: 65501
Server received message: 65501
Message sent with length: 65601
"cross_client_c.txt" [dos] 1313L, 42449B
```

Przetestowaliśmy nasze programy zarówno w wariancie klienta c i serwera w pythonie oraz odwrotnie. Wszystko zadziałało poprawie, zgodnie z oczekiwaniami.

docker run -it -d --network-alias z32_server --ip '172.21.32.2' --network z32_network --name z32_server z32_server)"

docker run -it -d --network z32_network --hostname z32_client_py --network-alias z32_client_py --name z32_client_py z32_client_py --IP '172.21.32.2' --PORT 8080)"

4.WNIOSKI KOŃCOWE

W ramach realizacji zadania zaimplementowano komunikację klient-serwer z wykorzystaniem protokołu UDP w Pythonie i C, zarówno w środowisku lokalnym, jak i w kontenerach Docker. Testy potwierdziły skuteczność protokołu UDP w przesyłaniu danych o zmiennym rozmiarze, uwzględniając minimalne opóźnienia oraz brak potrzeby nawiązywania połączenia. Zastosowanie Dockera umożliwiło izolację środowisk aplikacji, ułatwiło konfigurację sieci (statyczne IP, aliasy), a także uprościło uruchamianie i zarządzanie instancjami klienta i serwera.