Programowanie sieciowe

Miłosz Cieśla, Filip Ryniewicz, Aleksander Szymczyk

Sprawozdanie z zadania 1.2

22.11.2024

1 Treść zadania

Z 1 - Komunikacja UDP

Napisz zestaw dwóch programów – klienta i serwera wysyłających datagramy UDP. Wykonaj ćwiczenie w kolejnych inkrementalnych wariantach (rozszerzając kod z poprzedniej wersji). Zarówno klient, jak i serwer powinien być napisany w językach C oraz Python (4 programy). Sprawdź i przetestuj działanie międzyplatformowe, tj. klient w C z serwerem w Pythonie i odwrotnie.

Z 1.2

Wychodzimy z kodu z zadania 1.1, tym razem pakiety datagramu mają stałą wielkość, można przyjąć np. 512B. Należy zaimplementować prosty protokół niezawodnej transmisji, uwzględniający możliwość gubienia datagramów. Rozszerzyć protokół i program tak, aby gubione pakiety były wykrywane i retransmitowane. Wskazówka – „Bit alternate protocol". Należy uruchomić program w środowisku symulującym błędy gubienia pakietów. (Informacja o tym, jak to zrobić znajduje się w skrypcie opisującym środowisko Dockera).

To zadanie można wykonać, korzystając z kodu klienta i serwera napisanych w C lub w Pythonie (do wyboru). Nie trzeba tworzyć wersji w obydwu językach.

2 Opis rozwiązania

Ustalony format datagramu:

- pierwszy bajt: alternating bit
- drugi i trzeci bajt: długość przesyłanego datagramu
- czwarty i piąty bajt: suma kontrolna obliczana algorytmem fletcher16
- kolejne bajty: zawierają kolejne litery A-Z powtarzające się wymaganą liczbę razy

Ustalony format odpowiedzi:

- pierwszy bajt: alternating bit, ten sam co otrzymany jeśli kolejność się zgadza, zanegowana wartość bitu w przypadku błędnej sumy kontrolnej.
- drugi i trzeci bajt: długość przesyłanego datagramu

W celu symulacji retransmisji, przeprowadziliśmy eksperyment polegający na symulacji zaburzeń w przesyłaniu zgodnie ze skryptem do dockera. Przyjeliśmy rozmiar buffera równy 512B.

2.1 Schemat działania

- Serwer i klient tworzą własne sockety UDP.
- Serwer podpina swój socket pod wybrany przez argument wywołania port.
- Serwer oczekuje na nadejście datagramu.
- Klient generuje wiadomość.
- Na podstawie wiadomości klient generuje sumę kontrolną.
- Tworzymy datagram w założonym formacie 2.
- Klient przesyła datagram.
- Serwer odbiera datagram.
- Serwer oblicza dla wiadomości z datagramu sumę kontrolną i porównuje ją z otrzymaną od serwera. Korzystając z tego może on potwierdzić poprawność danych.
- Jeśli suma kontrolna się nie zgadza, serwer zmienia wartość alternating bit. W przeciwnym przypadku pozostawia ją niezmienioną.
- Serwer tworzy datagram w założonym formacie odpowiedzi 2

- Serwer wysyła datagram do adresu otrzymanego wcześniej dzięki funkcji recvfrom.
- Klient odbiera datagram z odpowiedzią, jeśli nie odbierze go w zadanym czasie (timeout) przesyła datagram ponownie.
- Klient sprawdza czy alternating bit odpowiedzi jest taki sam jak wysyłanego datagramu. Jeśli tak wysyła następną wiadomość, w przeciwnym wypadku ponownie wysyła tą samą wiadomość.

Struktura projektu została opisana w README.md.
Kod zawiera komentarze opisujące jego działanie.

3 Problemy

Podczas podawania rozmiaru datagramu do funkcji zapomnieliśmy uwzględnić zmieniony rozmiar nagłówka (z 4 na 5 bajtów), przez co generowaliśmy za dużą wiadomość, co powodowało błędne obliczenie sumy kontrolnej przez klienta oraz niezgodność sprawdzenia przez serwer.

4 Konfiguracja testowa

Rozwiązanie było testowane w dwóch wariantach:

4.1 Lokalnie

Utworzono sieć docker typu bridge o nazwie `z40_network`. Serwer przyjmował adres IP: `172.19.0.2`, a klient: `172.19.0.3`. Serwer UDP uruchamiany był na porcie `8000`, który to port był następnie podawany do klienta wraz z nazwą sieci.

4.2 Na serwerze `bigubu.ii.pw.edu.pl`

Do testów wykorzystywano utworzoną wcześniej sieć o nazwie `z40_network`. Serwer przyjmował adres IP: `172.21.40.2`, a klient: `172.21.40.3`. Serwer UDP uruchamiany był na porcie `8000`, który to port był następnie podawany do klienta wraz z nazwą sieci.

5 Testy

Testy polegały na nawiązaniu komunikacji klienta z serwerem i przesyłaniu datagramów o stałych rozmiarach z wprowadzeniem opóźnień za pomocą komendy:
`docker exec z40_client_container tc qdisc add dev eth0 root netem delay 1000ms 500ms loss 50%` . Przyjęty został timeout 5 sekund.

5.1 Testy lokalne

Brak opóźnienia.

```
2024-11-22 15:00:18 Client communicating with servername: z40_server_container and server port: 8000
2024-11-22 15:00:18 Sending message #0
2024-11-22 15:00:18 recv OK, communication time: 0.0009245420000070226
2024-11-22 15:00:18 Sending message #1
2024-11-22 15:00:18 recv OK, communication time: 0.0005739510000921655
2024-11-22 15:00:18 Sending message #2
2024-11-22 15:00:18 recv OK, communication time: 0.0003967339999973853
2024-11-22 15:00:18 Sending message #3
2024-11-22 15:00:18 recv OK, communication time: 0.0003113530000291576
2024-11-22 15:00:18 Sending message #4
2024-11-22 15:00:18 recv OK, communication time: 0.0002804199999673074
2024-11-22 15:00:18 Client finished.
```

Rysunek 1: Klient

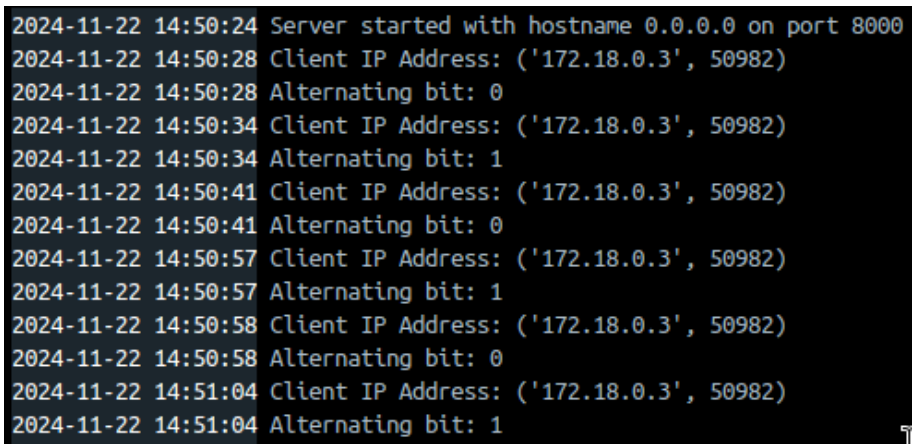
```
2024-11-22 15:00:18 Server started with hostname 0.0.0.0 on port 8000
2024-11-22 15:00:18 Client IP Address: ('172.20.0.3', 54354)
2024-11-22 15:00:18 Alternating bit: 0
2024-11-22 15:00:18 Client IP Address: ('172.20.0.3', 54354)
2024-11-22 15:00:18 Alternating bit: 1
2024-11-22 15:00:18 Client IP Address: ('172.20.0.3', 54354)
2024-11-22 15:00:18 Alternating bit: 0
2024-11-22 15:00:18 Client IP Address: ('172.20.0.3', 54354)
2024-11-22 15:00:18 Alternating bit: 1
2024-11-22 15:00:18 Client IP Address: ('172.20.0.3', 54354)
2024-11-22 15:00:18 Alternating bit: 0
```

Rysunek 2: Serwer

Średnie opóźnienie 1000ms z wariancją 500ms. Strata pakietów na poziomie 50%.

```
2024-11-22 14:50:24 Client communicating with servername: z40_server_container and server port: 8000
2024-11-22 14:50:24 Sending message #0
2024-11-22 14:50:28 recv OK, communication time: 3.9496572630000006
2024-11-22 14:50:28 Sending message #1
2024-11-22 14:50:33 Timeout occurred while waiting for response.
2024-11-22 14:50:33 Resending message #1
2024-11-22 14:50:34 recv OK, communication time: 1.0152178350000014
2024-11-22 14:50:34 Sending message #2
2024-11-22 14:50:39 Timeout occurred while waiting for response.
2024-11-22 14:50:39 Resending message #2
2024-11-22 14:50:41 recv OK, communication time: 1.1936771740000012
2024-11-22 14:50:41 Sending message #3
2024-11-22 14:50:46 Timeout occurred while waiting for response.
2024-11-22 14:50:46 Resending message #3
2024-11-22 14:50:51 Timeout occurred while waiting for response.
2024-11-22 14:50:51 Resending message #3
2024-11-22 14:50:56 Timeout occurred while waiting for response.
2024-11-22 14:50:56 Resending message #3
2024-11-22 14:50:57 recv OK, communication time: 1.3828157319999974
2024-11-22 14:50:57 Sending message #4
2024-11-22 14:50:58 recv OK, communication time: 0.9704220180000007
2024-11-22 14:50:58 Sending message #5
2024-11-22 14:51:03 Timeout occurred while waiting for response.
2024-11-22 14:51:03 Resending message #5
2024-11-22 14:51:04 recv OK, communication time: 1.4437276809999986
```

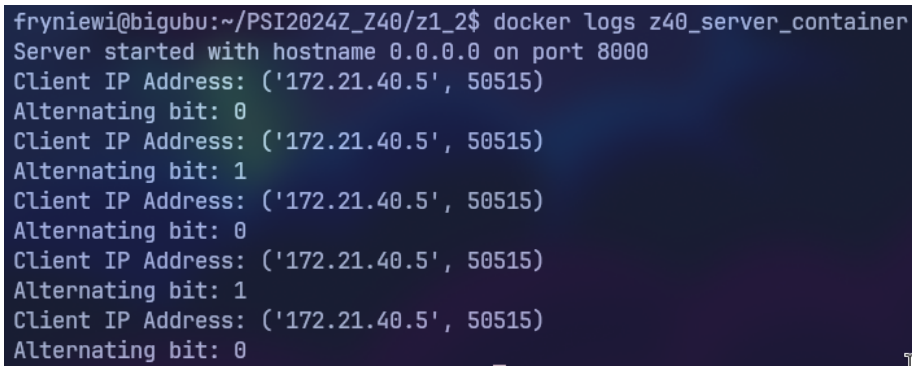
Rysunek 3: Klient



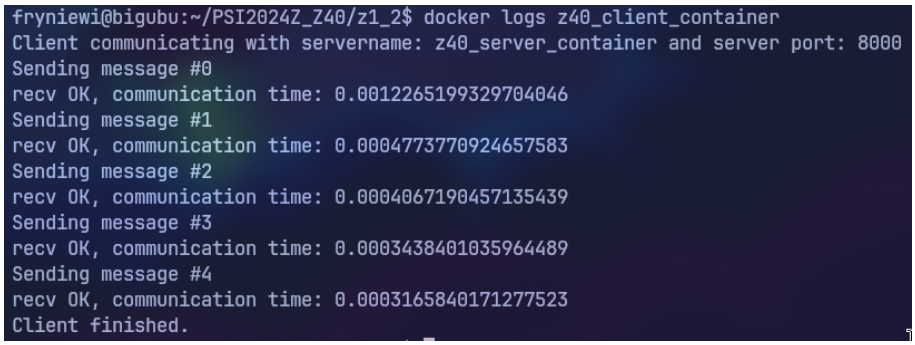
Rysunek 4: Serwer

5.2 Bigubu

Brak opóźnienia.

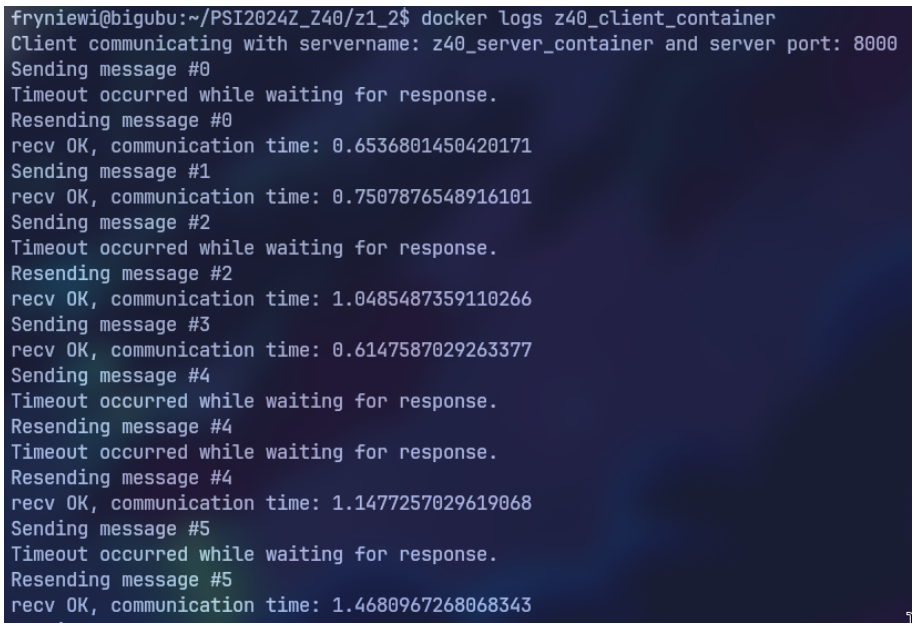


Rysunek 5: Klient

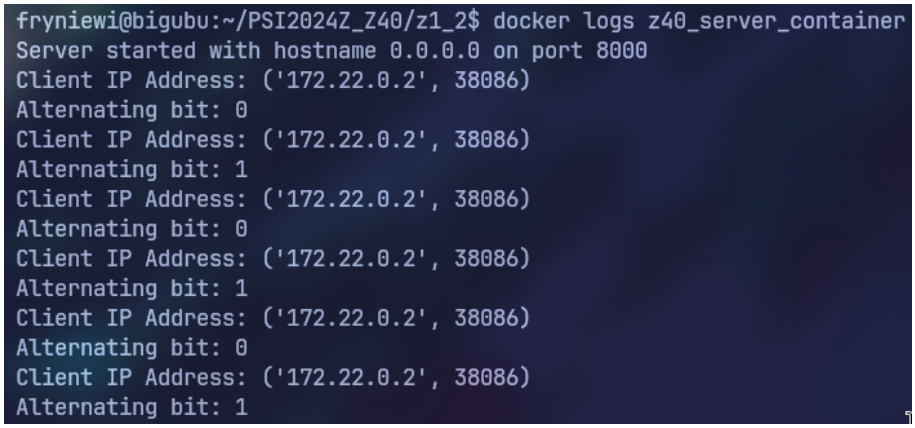


Rysunek 6: Enter Caption

Średnie opóźnienie 1000ms z wariancją 500ms. Strata pakietów na poziomie 50%.



Rysunek 7: Klient



Rysunek 8: Serwer

6 Wnioski

Przy niestabilnej sieci potrzebne są protokoły wskazujące na utratę pakietów. Protokół ABP jest skuteczny w wykrywaniu zgubionych pakietów dzięki mechanizmowi naprzemiennego bitu (alternating bit) oraz potwierdzeń (ACK). Każdy pakiet musi być potwierdzony przed wysłaniem kolejnego, co gwarantuje poprawność transmisji, nawet w niestabilnej sieci.