PSI Zadanie 1.2

Z37 - Aleksandra Szymańska, Angelika Ostrowska, Jakub Bąba

23.11.2024

1. Treść zadania

Z 1 Komunikacja UDP

Napisz zestaw dwóch programów – klienta i serwera wysyłające datagramy UDP. Wykonaj ćwiczenie w kolejnych inkrementalnych wariantach (rozszerzając kod z poprzedniej wersji). 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.

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 problemu

Zapoznaliśmy się z treścią zadania, zdecydowaliśmy się używać języka Python, bazując na serwerze i kliencie z zadania 1.1.

Za stały rozmiar datagramu wysyłanego przez klienta przyjęliśmy 512 bajtów. Pozostawia to 509 bajtów na wiadomość (zapętlone kolejne wielkie litery alfabetu), 2 bajty na rozmiar datagramu i 1 bajt na zaprezentowanie ACK.

Zmodyfikowaliśmy program klienta z poprzedniego zadania, aby przesyłał owy stały rozmiar wiadomości. Zaimplementowaliśmy używanie 1 bajtu datagramu jako ACK i przesyłanie tej samej wiadomości, dopóki nie otrzymamy potwierdzenia odebrania od serwera (wiadomości zwrotnej ze spodziewaną wartością bajtu ACK).

Następnie zmodyfikowaliśmy program serwera, aby był zgodny z przyjętymi założeniami – po odebraniu danych wysyłał wiadomość zawierającą ACK i komunikat tekstowy z numerem i długością wiadomości.

Oba pliki wymagały pewnych poprawek, na bieżąco testowaliśmy działanie kolejnych wersji programów.

Testowaliśmy również ich działanie w środowisku symulującym błędy gubienia pakietów. Wprowadzenie zakłóceń w kontenerze klienta zaimplementowaliśmy zgodnie z dostarczonym przez wykładowcę skryptem opisującym środowisko Dockera. Implementacja znajduje się w pliku „disrupt.sh", uruchomienie tego skryptu wprowadza zakłócenia.

3. Napotkane problemy

3.1. Zaimplementowanie niezawodnej transmisji

UDP, w przeciwieństwie do TCP nie gwarantuje niezawodności. Datagramy mogą ginąć, ulegać zwielokrotnieniu. Musieliśmy wymyślić jak własnoręcznie tą niezawodność zaimplementować.

Jednym z elementów jest dodanie do datagramu bajtu ACK. Przy kodowaniu wiadomości do postaci bajtowej, jako pierwszy bajt ustawiamy ACK równe 0 lub 1.

Drugim elementem są zmienne reprezentujące wysłane wiadmości i wysłane wiadomości, które dotarły do odbiorcy z sukcesem. Po wysłaniu wiadomości zwiększamy o 1 zmienną „message_sent".

Łącząc obie rzeczy – po wysłaniu wiadomości z bajtem ACK o pewnej wartości, klient czeka na odebranie danych od serwera.

> Jeśli pierwszy bajt danych jest taki jak aktualne ACK – wiadomość została dostarczona i odebrana z sukcesem. Licznik dobrze wysłanych wiadomości „message_successfully_sent" zwiększamy o 1, a wartość ACK operacją XOR zmieniana jest z 0 na 1 / z 1 na 0. Nowa wartość ACK będzie dodana do kolejnej wiadomości.
>
> Jeśli pierwszy bajt danych nie równa się aktualnemu ACK, lub serwer w ogóle nie odesłał danych w oczekiwanym czasie, oznacza to, że na którymś etapie transmisji wystąpił błąd. Nie zwiększamy więc „message_successfully_sent" i nie zmianiamy wartości ACK. W kolejnym obiegu pętli wyślemy wiadomość z tym samym ACK, co symuluje ponowne wysłanie wiadomości.

Nasze podejście przypomina działanie algorytmu „Alternating bit protocol".

3.2. Stworzenie środowiska symulującego błędy transmisji

Bazując na instrukcji dostarczonej przez wykładowcę stworzyliśmy skrypt „disrupt.sh".

Treść skryptu:

```
docker exec z37_zadanie1_2_python_client tc qdisc add dev eth0 root
netem delay 1000ms loss 50%
```

Wykorzystuje on mechanizm „netem", polecenie tc i qdisc. Aby mógł zadziałać, stworzyliśmy plik „docker-compose.yaml" opisujący konfigurację aplikacji i ustawiający uprawnienie NET_ADMIN.

Ustawiamy opóźnienie 1000 ms i prawdopodobieństwo zagubienia pakietu równe 50%.

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

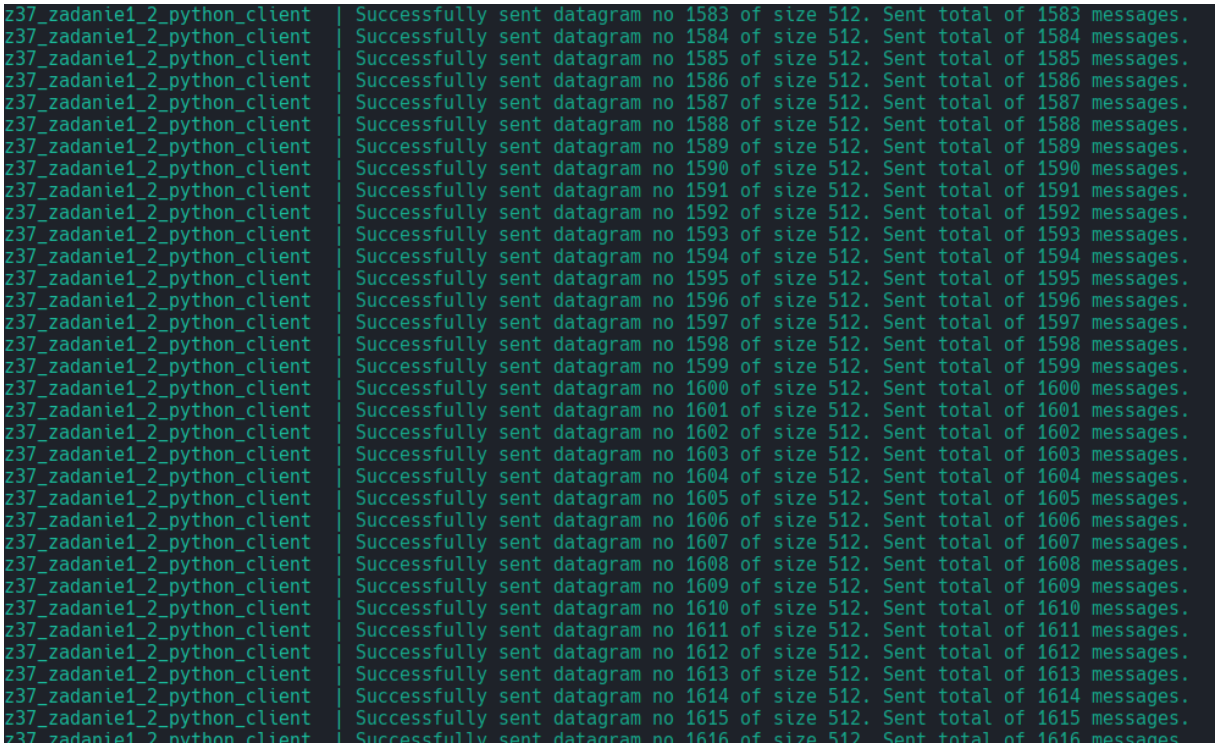
Po napisaniu kodu sprawdzaliśmy, czy działa on z zamierzonym efektem. Uruchamialiśmy serwer i klienta za pomocą „docker compose build”, „docker compose up” i testowaliśmy komunikację.

Wykryliśmy pewne błędy w kodzie, naprawialiśmy je więc i testowaliśmy ponownie, żeby upewnić się, że efekt jest zgodny z zamierzonym.

Wykonywaliśmy testy zarówno w środowisku „zwykłym” jak i w takim z symulacją zakłóceń.

Zrzuty ekranu z uruchomień:

Klient bez zakłóceń:



Widzimy, że wszystkie wysłane wiadomości docierają do odbiorcy bez zakłóceń. Można zaobserwować to po fakcie, że numer wysłanego z sukcesem datagramu jest taki sam jak numer wszystkich wysłanych datagramów.

Przykładowo komunikat „Successfully sent datagram no 1612 of size 512. Sent total of 1612 messages.” wyświetlony był po otrzymaniu od serwera potwierdzenia otrzymania tysiąc sześćset dwunastej wiadomości. Oznacza, że w sumie wysłano 1612 wiadomości, a wysłanych i z sukcesem odebranych wiadomości też jest 1612. Nie było więc wiadomości, która została wysłana i nie zakończyła się sukcesem.

Klient z zakłóceniami:

```
z37_zadanie1_2_python_client  | Timeout for message no 1940.
z37_zadanie1_2_python_client  | Timeout for message no 1941.
z37_zadanie1_2_python_client  | Timeout for message no 1942.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1900 of size 512. Sent total of 1943 messages.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1901 of size 512. Sent total of 1944 messages.
z37_zadanie1_2_python_client  | Timeout for message no 1945.
z37_zadanie1_2_python_client  | Timeout for message no 1946.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1902 of size 512. Sent total of 1947 messages.
z37_zadanie1_2_python_client  | Timeout for message no 1948.
z37_zadanie1_2_python_client  | Timeout for message no 1949.
z37_zadanie1_2_python_client  | Timeout for message no 1950.
z37_zadanie1_2_python_client  | Timeout for message no 1951.
z37_zadanie1_2_python_client  | Timeout for message no 1952.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1903 of size 512. Sent total of 1953 messages.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1904 of size 512. Sent total of 1954 messages.
z37_zadanie1_2_python_client  | Timeout for message no 1955.
z37_zadanie1_2_python_client  | Timeout for message no 1956.
z37_zadanie1_2_python_client  | Timeout for message no 1957.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1905 of size 512. Sent total of 1958 messages.
z37_zadanie1_2_python_client  | Timeout for message no 1959.
z37_zadanie1_2_python_client  | Timeout for message no 1960.
z37_zadanie1_2_python_client  | Timeout for message no 1961.
z37_zadanie1_2_python_client  | Timeout for message no 1962.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1906 of size 512. Sent total of 1963 messages.
z37_zadanie1_2_python_client  | Timeout for message no 1964.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1907 of size 512. Sent total of 1965 messages.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1908 of size 512. Sent total of 1966 messages.
z37_zadanie1_2_python_client  | Successfully sent datagram no 1909 of size 512. Sent total of 1967 messages.
z37_zadanie1_2_python_client  | Timeout for message no 1968.
z37_zadanie1_2_python_client  | Timeout for message no 1969.
z37_zadanie1_2_python_client  | Timeout for message no 1970.
z37_zadanie1_2_python_client  | Timeout for message no 1971.
z37_zadanie1_2_python_client  | Timeout for message no 1972.
```

Tutaj widzimy, że tylko część wysłanych wiadomości otrzymuje komunikat potwierdzająct od serwera. Komunikaty „Successfully sent ...” przeplatane są informacjami o timeoutach. Taka wiadomość oznacza, że albo serwer nie otrzymał wiadomości (została zgubiona), albo nie zdążył na czas (w przeciągu 3 sekund) odpowiedzieć – łącze było powolne.

Czytając przykładowy komunikat o sukcesie widzimy, że np. w sumie wysłano 1967 wiadomości, ale z sukcesem wysłanych było tylko 1909.

Instrukcja uruchomienia znajduje się w pliku README.md.

6. Wnioski końcowe

Udało nam się osiągnąć komunikację między klientem a serwerem, która odbywa się przez UDP (protokół warstwy transportowej który nie gwarantuje niezawodności), ale i tak jest niezawodna.

Udało się to dzięki zaimplementowaniu algorytmu przypominającego „Alternating bit protocol”. Przetestowaliśmy rozwiązanie zarówno w dobrze działającej sieci jak i w takiej symulującej zakłócenia.

Osiągnęliśmy niezawodną transmisję – wiadomość będzie wysyłana przez klienta, dopóki nie otrzyma on od serwera potwierdzenia jej odebrania. Gwarantuje to, że wszystkie wiadomości wysłane przez klienta zostaną w końcu odebrane przez serwer.