24.11.2024

Zespół 36

Mateusz Łukasiewicz

Rafał Celiński

Przemysław Walecki

Programowanie sieciowe

Zadanie 1.2

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.

Konfiguracja

Adres IP serwera – 172.21.36.2

Adres IP klienta – 172.21.36.3

Porty – 5018:5018 (UDP)

Obrazy Docker

Obraz dla Pythona - python:3-slim

Obraz dla GCC - gcc:4.9

Rozwiązanie

Do zasymulowania gubienia pakietów w środowisku użyliśmy polecenia

docker exec z36_client_container tc qdisc add dev eth0 root netem loss 40%

Pozwala ono na gubienie 40% pakietów wysłanych przez kontener klienta.

Aby móc wykonać to polecenie w kontenerach należało zainstalować paczkę *iproute2*. Wystarczyło dodać taki element do Dockerfile

RUN apt-get update && apt-get install -y iproute2 && apt-get clean

Programy klienta i serwera zrealizowaliśmy w języku Python. Powstały dwie wersje które umożliwiają zobaczenie różnic w komunikacji.

Wersja 1 plików zawiera w nazwie '_loss' i jest bardzo uproszczoną formą przesyłania pakietów. Jedyne zadanie klienta to wysłanie prostego pakiet i wypisanie tej informację w konsoli. Zadaniem serwera jest odbieranie pakietów i wypisanie otrzymanej wiadomości. W rezultacie po włączeniu gubienia pakietów otrzymujemy poniższe wyniki:

```
z36_server_container  | Received datagram: 28: Test message
z36_client_container  | Sent message: 29
z36_client_container  | Sent message: 30
z36_client_container  | Sent message: 31
z36_server_container  | Received datagram: 31: Test message
z36_server_container  | Received datagram: 32: Test message
z36_client_container  | Sent message: 32
z36_server_container  | Received datagram: 33: Test message
z36_client_container  | Sent message: 33
z36_server_container  | Received datagram: 34: Test message
z36_client_container  | Sent message: 34
z36_client_container  | Sent message: 35
z36_server_container  | Received datagram: 35: Test message
z36_client_container  | Sent message: 36
z36_client_container  | Sent message: 37
z36_server_container  | Received datagram: 38: Test message
z36_client_container  | Sent message: 38
z36_client_container  | Sent message: 39
z36_client_container  | Sent message: 40
z36_server_container  | Received datagram: 41: Test message
```

Nie ma żadnego zaskoczenia. Część pakietów po prostu nie dociera do serwera.

Wersja druga jest bardziej złożona. Zgodnie z wskazówką zaimplementowaliśmy bit alternate protocol. Jego działanie omówiliśmy poniżej wraz z wyjaśnieniem kodu.

Klient

```
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.settimeout(2)
    message_bit = 0
    for iteration in range(2000):
        message = f"{message_bit}{iteration}".encode()
        while True:
            sent_bytes = s.sendto(message, (HOST, UDP_PORT))
            print(f"Sent message: Message bit = {message_bit}, Iteration = {iteration}")
            try:
                ack_bit, address = s.recvfrom(256)
                ack_bit = int(ack_bit.decode())
                print(f'Received ACK: {ack_bit}')
                if (ack_bit == message_bit):
                    message_bit = 1 - message_bit
                    break
            except socket.timeout:
                print("No ACK. Resending.")
        time.sleep(0.2)

s.close()
```

1. Tworzymy gniazdo UDP do komunikacji z serwerem oraz ustawiamy bit sekwencyjny, który będziemy wykorzystywali do weryfikacji kolejności przesyłanych wiadomości
2. W każdej iteracji skończonej pętli staramy się wysłać wiadomość aż do skutku:
   a) Wysyłamy wiadomość z bitem sekwencyjnym
   b) Czekamy na odpowiedź od serwera
   c) Jeżeli otrzymaliśmy odpowiedź z takim samym bitem sekwencyjnym przechodzimy do kolejnej iteracji pętli, a samą wartość bitu zmieniamy tak żeby serwer wiedział że otrzymuje już kolejną wiadomość
   d) Jeżeli nie otrzymaliśmy odpowiedzi – wysyłamy wiadomość ponownie z takim samym bitem, bo prawdopodobnie przepadła
3. Po sukcesywnym wysłaniu wszystkich wiadomości kończymy program

Serwer

```python
import socket

IP = "172.21.36.2"
UDP_PORT = 5018

print(f"Server listening on {IP}:{UDP_PORT}")
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    expected_message_bit = 0
    s.bind((IP, UDP_PORT))
    while True:
        message, addr = s.recvfrom(256)
        message = message.decode()
        message_bit = int(message[0])
        iteration = int(message[1:])
        print(f'Received message: Message bit = {message_bit} Iteration = {iteration}')

        if(message_bit == expected_message_bit):
            ack_bit = str(message_bit).encode()
            s.sendto(ack_bit, addr)
            print(f'Send ACK: {message_bit}')
            expected_message_bit = 1 - expected_message_bit

        else:
            ack_bit = str(1 - expected_message_bit).encode()
            s.sendto(ack_bit, addr)
            print(f'Resending ACK: {1 - expected_message_bit}')
```

1. Tworzymy gniazdo UDP i powiązujemy je z adresem IP i portem serwera aby móc nasłuchiwać na połączenia oraz ustawiamy wartość spodziewanego bitu sekwencyjnego (**wartość startowa powinna być taka sama jak w programie klienta aby komunikacja przebiegła poprawnie**)
2. W nieskończonej pętli odbieramy wiadomości od klienta oraz odczytujemy otrzymany bit sekwencyjny
3. Jeżeli otrzymany bit jest taki sam jak spodziewana wartość to znaczy, że komunikacja przebiegła dobrze i możemy poinformować klienta o tym, że dostaliśmy wiadomość
4. Jeżeli otrzymany bit jest inny to znaczy, że poprzednie wysłane potwierdzenie nie dotarło do klienta – w związku z tym ten cały czas wysyła tą samą wiadomość. W takim wypadku wysyłamy potwierdzenie ponownie

Działanie programów

Po zbudowaniu i uruchomieniu kontenerów w Dockerze uzyskaliśmy takie wyniki

```
z36_server_container  | Server listening on 172.21.36.2:5018
z36_server_container  | Received message: Message bit = 0 Iteration = 0
z36_client_container  | Sent message: Message bit = 0, Iteration = 0
z36_server_container  | Send ACK: 0
z36_client_container  | Received ACK: 0
z36_client_container  | Sent message: Message bit = 1, Iteration = 1
z36_server_container  | Received message: Message bit = 1 Iteration = 1
z36_server_container  | Send ACK: 1
z36_client_container  | Received ACK: 1
z36_server_container  | Received message: Message bit = 0 Iteration = 2
z36_server_container  | Send ACK: 0
z36_client_container  | Sent message: Message bit = 0, Iteration = 2
z36_client_container  | Received ACK: 0
z36_server_container  | Received message: Message bit = 1 Iteration = 3
z36_client_container  | Sent message: Message bit = 1, Iteration = 3
z36_server_container  | Send ACK: 1
z36_client_container  | Received ACK: 1
z36_client_container  | Sent message: Message bit = 0, Iteration = 4
z36_client_container  | Received ACK: 0
z36_server_container  | Received message: Message bit = 0 Iteration = 4
z36_server_container  | Send ACK: 0
z36_server_container  | Received message: Message bit = 1 Iteration = 5
z36_server_container  | Send ACK: 1
z36_client_container  | Sent message: Message bit = 1, Iteration = 5
```

Na razie bez uruchomienia poleceń zakłócających połączenie nie pojawiają się wiadomości o ponownym wysyłaniu pakietów.

Po wywołaniu poniższych poleceń można zaobserwować retransmisję pakietów

docker exec z36_client_container tc qdisc add dev eth0 root netem loss 40%
docker exec z36_server_container tc qdisc add dev eth0 root netem loss 40%

W tym przypadku włączyliśmy także możliwość gubienia pakietów wysłanych przez serwer

```
z36_client_container | Sent message: Message bit = 0, Iteration = 38
z36_server_container | Received message: Message bit = 0 Iteration = 38
z36_server_container | Send ACK: 0
z36_client_container | No ACK. Resending.
z36_client_container | Sent message: Message bit = 0, Iteration = 38
z36_client_container | Received ACK: 0
z36_server_container | Received message: Message bit = 0 Iteration = 38
z36_server_container | Resending ACK: 0
z36_client_container | Sent message: Message bit = 1, Iteration = 39
z36_server_container | Received message: Message bit = 1 Iteration = 39
z36_server_container | Send ACK: 1
z36_client_container | No ACK. Resending.
z36_client_container | Sent message: Message bit = 1, Iteration = 39
z36_server_container | Received message: Message bit = 1 Iteration = 39
z36_server_container | Resending ACK: 1
z36_client_container | Received ACK: 1
z36_client_container | Sent message: Message bit = 0, Iteration = 40
z36_client_container | No ACK. Resending.
z36_client_container | Sent message: Message bit = 0, Iteration = 40
z36_client_container | No ACK. Resending.
z36_client_container | Sent message: Message bit = 0, Iteration = 40
z36_server_container | Received message: Message bit = 0 Iteration = 40
z36_server_container | Send ACK: 0
```

Jak widać występują tu sytuacje, w których klient wysyła wiadomość ponownie ponieważ nie dostał potwierdzenia, a także sytuacje, w których takie potwierdzenie zostało zgubione i serwer musi wysłać je ponownie.

Komunikacja przebiega prawidłowo pomimo gubienia pakietów.