

PSI zad. 1.2

Andrii Rybachok

Anna Tanelo

Ihor Garvylenko

Data sporządzenia: 04.12.2025

Wersja: 1.0

### **Treść:**

Klient ma za zadanie odczytać plik z dysku (proszę wygenerować plik z losowymi 10000B) i wysłać do serwera jego zawartość w paczkach po 100B. Serwer ma zrekonstruować cały plik i obliczyć jego hash. Jako dowód działania proszę m.in. porównać hash obliczony przez serwer z haszem obliczonym przez klienta (może to być wydrukowane w konsoli klienta/serwera, hashe muszą być identyczne). Należy zaimplementować prosty protokół niezawodnej transmisji, uwzględniający możliwość gubienia datagramów. Gubione pakiety muszą być wykrywane i retransmitowane aby serwer mógł odtworzyć cały plik. Należy uruchomić program w środowisku symulującym błędy gubienia pakietów. (Informacja o tym, jak to zrobić znajduje się w skrypcie opisującym środowisko Dockera).

### **Opis rozwiązania:**

#### **1. Konfiguracja językowa**

W naszym rozwiązaniu przyjęto konfigurację:

- Serwer: C (UDP, odbiór pliku, liczenie SHA-256)
- Klient: Python (UDP, wysyłanie pliku, porównanie hashy)

Oba programy są uruchamiane w oddzielnych kontenerach Dockera, połączonych we wspólnej sieci.

#### **2. Protokół aplikacyjny**

Protokół jest binarny i działa nad UDP w schemacie stop-and-wait.

Wykorzystujemy cztery typy wiadomości:

- START ('S') - inicjalizacja transmisji:
  - zawiera `file_size` i `chunk_size`.
- DATA ('D') - fragment pliku:

- zawiera numer sekwencyjny seq, długość danych data\_len oraz dane.
- ACK ('A') - potwierdzenie odebrania:
  - zawiera numer seq pakietu, który został poprawnie odebrany.
- HASH ('H') - końcowy hash SHA-256:
  - zawiera 32 bajty hash-a policzonego przez serwer.

Przebieg:

1. Klient wysyła START(file\_size=10000, chunk\_size=100) i czeka na ACK(0).
2. Klient dzieli plik na 100 fragmentów po 100 B i dla każdego seq od 0 do 99:
  - wysyła DATA(seq, dane),
  - czeka na ACK(seq) z timeoutem,
  - w razie niewłaściwego ACK powtarza wysyłkę.
3. Serwer zapisuje nowe pakiety w buforze i zawsze odsyła ACK(seq).
4. Po odebraniu 10000B serwer liczy SHA-256 i wysyła hash.
5. Klient porównuje lokalny SHA-256 z hashem serwera.

Wszystkie liczby są przesyłane w porządku sieciowym (big-endian).

### **3. Serwer (C) opis działania**

Główne kroki po stronie serwera:

1. Utworzenie gniazda UDP: socket(AF\_INET, SOCK\_DGRAM, 0) i bind() na porcie (np. 9000).
2. Inicjalizacja struktury FileContext (bufor 10000 B, expected\_seq, received\_bytes).
3. Pętla główna:
  - recvfrom() odbiera datagram,
  - na podstawie pierwszego bajtu serwer rozpoznaje typ wiadomości.
4. Dla START:
  - odczyt file\_size i chunk\_size, sprawdzenie czy to 10000 i 100,
  - wyzerowanie bufora, ustawienie expected\_seq = 0, received\_bytes = 0,
  - odesłanie ACK(0).
5. Dla DATA:

- odczyt seq i data\_len, sprawdzenie poprawności długości,
  - jeśli seq == expected\_seq, zapis danych w buforze w odpowiednie miejsce, zwiększenie expected\_seq i received\_bytes,
  - jeśli seq < expected\_seq, traktowanie jako duplikatu (danych nie nadpisujemy),
  - w każdym przypadku wysłanie ACK(seq).
6. Po każdym DATA serwer sprawdza:
- jeśli received\_bytes == file\_size, liczy SHA-256 na buforze,
  - wysyła wiadomość HASH z hashem do klienta,
  - wypisuje hash na konsolę i resetuje kontekst (gotowy na kolejną transmisję).

#### **4. Klient (Python) opis działania**

Główne kroki po stronie klienta:

1. Przygotowanie danych:
  - wygenerowanie pliku 10000 B (lub odczyt z dysku),
  - sprawdzenie, że rozmiar pliku to dokładnie 10000 bajtów,
  - policzenie lokalnego SHA-256.
2. Utworzenie gniazda UDP: socket(AF\_INET, SOCK\_DGRAM) oraz ustawienie timeoutu na odbiór.
3. Wysłanie START:
  - zbudowanie pakietu START i wysłanie go,
  - oczekiwanie na ACK(0) z ponawianiem w razie timeoutu.
4. Wysyłanie pliku:
  - podział pliku na bloki po 100 bajtów,
  - dla każdego seq:
    - zbudowanie DATA(seq, chunk),
    - wysłanie pakietu,
    - oczekiwanie na ACK(seq) (stop-and-wait, powtórki w razie braku ACK).
5. Odbiór hash:
  - po wysłaniu ostatniego bloku klient czeka na pakiet HASH z serwera,
  - po odebraniu wyciąga 32 bajty hash-a.
6. Porównanie hashy:
  - wypisuje lokalny i serwerowy hash w formie hex,

- jeśli hashe są identyczne wypisuje informację, że wynik jest poprawny,
  - w przeciwnym razie sygnalizuje błąd.
7. Zamknięcie gniazda i zakończenie programu.

## **Napotkane problemy i sposoby ich rozwiązania:**

### **1. Błędny rozmiar ostatniego pakietu DATA**

Na początku klient dzielił plik na fragmenty w pętli z trochę źle policzonym offsetem i warunkiem końca. W efekcie ostatni pakiet DATA miał inną długość niż 100 bajtów, a serwer odrzucał go przy sprawdzaniu. Objawem było to, że klient wysyłał wszystkie pakiety, ale na końcu hashe klienta i serwera się nie zgadzały. Problem został rozwiązany przez ustawienie stałych oraz poprawienie pętli tak, aby zawsze wysyłała dokładnie 100 pakietów po 100 bajtów.

### **2. Logi serwera zatrzymujące się na 8200 bajtach**

Przy pierwszym uruchomieniu w Dockerze w logach serwera `received_bytes` zatrzymywało się na wartości 8200, chociaż klient otrzymywał poprawny hash i program działał prawidłowo. Wyglądało to tak, jakby serwer nie odbierał całego pliku, co wprowadzało spore zamieszanie przy debugowaniu. Okazało się, że problem wynika z buforowania `printf` w C w środowisku kontenera. Po wyłączeniu buforowania standardowego wyjścia logi zaczęły pokazywać wszystkie pakiety aż do 10000 bajtów oraz komunikat o odebraniu pełnego pliku i wyliczonym hashu.

### **3. Adres serwera w sieci Dockera**

Dodatkowym drobnym problemem była konfiguracja adresu serwera przy uruchamianiu w Dockerze. Na początku klient próbował łączyć się z 127.0.0.1, co wewnątrz kontenera oznaczało połączenie z samym sobą, a nie z serwerem w drugim kontenerze. Objawem było to, że klient nie dostawał żadnych odpowiedzi, a protokół wchodził w kolejne retransmisje, aż do błędu. Problem został rozwiązany przez utworzenie wspólnej sieci `psi-net` w pliku `docker-compose.yml` oraz użycie nazwy usługi `server` jako hosta. Dzięki temu klient i serwer poprawnie się widzą i komunikacja działa zgodnie z założeniami.

## Testy:

### Test w środowisku Docker

test polegał na uruchomieniu całego rozwiązania przy pomocy docker compose up --build. Sprawdzaliśmy, czy kontenery klienta i serwera poprawnie się uruchamiają, klient generuje plik wewnątrz kontenera, a następnie łączy się z usługą server po UDP. W logach klienta oczekiwaliśmy identycznych hashy oraz kodu wyjścia 0, a w logach serwera - komunikatu o odebraniu pełnego pliku i poprawnej obsłudze wszystkich pakietów DATA.

```
[+] Running 4/4
✓ zadanie_1_2-server      Built
✓ zadanie_1_2-client      Built
✓ Container psi_server    Recreated
✓ Container psi_client    Recreated
Attaching to psi_client, psi_server
psi_server | Serwer nasłuchuje na porcie UDP 9000
psi_client | [client] Generuję plik /tmp/random.bin...
psi_client | Utworzono plik /tmp/random.bin o rozmiarze 10000 bajtów.
psi_client | [client] Start klienta UDP...
psi_server | Odebrano START: file_size=10000 chunk_size=100
psi_server | Kontekst zainicjalizowany, oczekuję danych...
psi_server | DATA: seq=0, data_len=100, received_bytes=100
psi_server | DATA: seq=1, data_len=100, received_bytes=200
psi_server | DATA: seq=2, data_len=100, received_bytes=300
psi_server | DATA: seq=3, data_len=100, received_bytes=400
psi_server | DATA: seq=4, data_len=100, received_bytes=500
psi_server | DATA: seq=5, data_len=100, received_bytes=600
psi_server | DATA: seq=6, data_len=100, received_bytes=700
psi_server | DATA: seq=7, data_len=100, received_bytes=800
psi_server | DATA: seq=8, data_len=100, received_bytes=900
psi_server | DATA: seq=9, data_len=100, received_bytes=1000
psi_server | DATA: seq=10, data_len=100, received_bytes=1100
psi_server | DATA: seq=11, data_len=100, received_bytes=1200
psi_server | DATA: seq=12, data_len=100, received_bytes=1300
psi_server | DATA: seq=13, data_len=100, received_bytes=1400
psi_server | DATA: seq=14, data_len=100, received_bytes=1500
psi_server | DATA: seq=15, data_len=100, received_bytes=1600
psi_server | DATA: seq=16, data_len=100, received_bytes=1700
psi_server | DATA: seq=17, data_len=100, received_bytes=1800
psi_server | DATA: seq=18, data_len=100, received_bytes=1900
psi_server | DATA: seq=19, data_len=100, received_bytes=2000
psi_server | DATA: seq=20, data_len=100, received_bytes=2100
psi_server | DATA: seq=21, data_len=100, received_bytes=2200
```

[illegible]

```

psi_server | DATA: seq=70, data_len=100, received_bytes=7100
psi_server | DATA: seq=71, data_len=100, received_bytes=7200
psi_server | DATA: seq=72, data_len=100, received_bytes=7300
psi_server | DATA: seq=73, data_len=100, received_bytes=7400
psi_server | DATA: seq=74, data_len=100, received_bytes=7500
psi_server | DATA: seq=75, data_len=100, received_bytes=7600
psi_server | DATA: seq=76, data_len=100, received_bytes=7700
psi_server | DATA: seq=77, data_len=100, received_bytes=7800
psi_server | DATA: seq=78, data_len=100, received_bytes=7900
psi_server | DATA: seq=79, data_len=100, received_bytes=8000
psi_server | DATA: seq=80, data_len=100, received_bytes=8100
psi_server | DATA: seq=81, data_len=100, received_bytes=8200
psi_server | DATA: seq=82, data_len=100, received_bytes=8300
psi_server | DATA: seq=83, data_len=100, received_bytes=8400
psi_server | DATA: seq=84, data_len=100, received_bytes=8500
psi_server | DATA: seq=85, data_len=100, received_bytes=8600
psi_server | DATA: seq=86, data_len=100, received_bytes=8700
psi_server | DATA: seq=87, data_len=100, received_bytes=8800
psi_server | DATA: seq=88, data_len=100, received_bytes=8900
psi_server | DATA: seq=89, data_len=100, received_bytes=9000
psi_server | DATA: seq=90, data_len=100, received_bytes=9100
psi_server | DATA: seq=91, data_len=100, received_bytes=9200
psi_server | DATA: seq=92, data_len=100, received_bytes=9300
psi_server | DATA: seq=93, data_len=100, received_bytes=9400
psi_server | DATA: seq=94, data_len=100, received_bytes=9500
psi_server | DATA: seq=95, data_len=100, received_bytes=9600
psi_server | DATA: seq=96, data_len=100, received_bytes=9700
psi_server | DATA: seq=97, data_len=100, received_bytes=9800
psi_server | DATA: seq=98, data_len=100, received_bytes=9900
psi_server | DATA: seq=99, data_len=100, received_bytes=10000
psi_server | Odebrano pełny plik 10000 bajtów
psi_server | Server hash: ea7a89f89263bdf76d51a28293f6bd6bdcd949ae782bf9f5b0cd3af95395a22d
psi_client | Local hash : ea7a89f89263bdf76d51a28293f6bd6bdcd949ae782bf9f5b0cd3af95395a22d
psi_client | Server hash: ea7a89f89263bdf76d51a28293f6bd6bdcd949ae782bf9f5b0cd3af95395a22d
psi_client | Wynik: hashe są identyczne (OK).
psi_client | exited with code 0

```

## Testy zachowania w sytuacjach błędnych

Dodatkowo wykonaliśmy proste testy błędów. Najpierw uruchomiliśmy klienta z plikiem o innym rozmiarze niż 10000 bajtów, co zgodnie z założeniem zakończyło się komunikatem o nieprawidłowej długości i przerwaniem programu. Następnie uruchomiliśmy klienta z błędnym portem lub bez działającego serwera – w tym przypadku klient próbował wysyłać pakiety, wchodził w retransmisje i na końcu zgłaszał błąd protokołu (brak ACK lub HASH).

```

psi_server | Serwer nasłuchuje na porcie UDP 9000
psi_client | [client] Generuję plik /tmp/random.bin...
psi_client | Utworzono plik /tmp/random.bin o rozmiarze 9000 bajtów.
psi_client | [client] Start klienta UDP...
psi_client | Błąd: plik musi mieć dokładnie 10000 bajtów, a ma 9000.
psi_client exited with code 1

```

```

[+] Running 4/4
✓ zadanie_1_2-server      Built
✓ zadanie_1_2-client      Built
✓ Container psi_server    Recreated
✓ Container psi_client    Recreated
Attaching to psi_client, psi_server
psi_server | Serwer nasłuchuje na porcie UDP 9000
psi_client | [client] Generuję plik /tmp/random.bin...
psi_client | Utworzono plik /tmp/random.bin o rozmiarze 10000 bajtów.
psi_client | [client] Start klienta UDP...
psi_client | Błąd protokołu: Za dużo prób wysłania pakietu seq=0
psi_client exited with code 1

```

## Wnioski:

- Realizacja zadania pokazała w praktyce, że standardowy UDP jest niezawodny i żeby przesłać coś sensownego (np. plik 10000 B), trzeba samemu zadbać o numerację pakietów, potwierdzenia i retransmisję.
- Podział logiki na warstwy bardzo ułatwił debugowanie. Przy błędach było od razu widać, czy problem jest w formacie pakietu, w pętli wysyłania, czy w samej komunikacji sieciowej.
- Użycie Dockera i docker-compose pozwoliło zamknąć cały projekt w jednym środowisku i uprościć uruchamianie do jednej komendy. Wyszły przy tym typowe problemy z siecią i buforowaniem logów, ale po ich rozwiązaniu konfiguracja stała się powtarzalna i wygodna do testów.
- Testy pokazały, że programy zachowują się przewidywalnie: przy poprawnych danych hashe się zgadzają, a przy problemach klient zgłasza czytelny błąd zamiast się wieszać.