# Programowanie Sieciowe
# Zadanie 1.2

Radosława Żukowska - Lider Zespołu
Aleksandra Szczypawka
Małgorzata Grzanka

04.12.2025r
Wersja sprawozdania: 1

## Zadanie 1 - Komunikacja UDP

Napisz zestaw dwóch programów – klienta i serwera wysyłające datagramy UDP. Proszę napisać jedno zadanie w konfiguracji klient/server Python/C, a drugie w konfiguracji klient/server C/Python – do wyboru.

### 1.2

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 hashem 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).

## Rozwiązanie

Link do Repozytorium

Do realizacji zadania powstał: serwer UDP w języku Python oraz klient UDP w języku C. Pomimo wykorzystania protokołu bezpołączeniowego (UDP), zarówno klient, jak i serwer posiadają zaimplementowany mechanizm zapewniający niezawodność transmisji.

Logika zaimplementowanego protokołu opiera się na mechanizmie ARQ (Automatic Repeat Request) typu Stop-and-Wait. Dane dzielone są na mniejsze pakiety. Każdy pakiet posiada strukturę [seq_num][data], gdzie seq_num to numer sekwencyjny (indeks) danego pakietu.

```
char packet_data_buf[PACKAGE_SIZE];
char packet_with_ack[PACKAGE_SIZE + ACK_SIZE];
char received_ack_buf[ACK_SIZE];
. . .
int netseq = htonl(seq_num);
memcpy(packet_with_ack, &netseq, ACK_SIZE);
```

```
nread = fread(packet_data_buf, 1, PACKAGE_SIZE, fptr);
. . .
memcpy(file_data + seq_num * PACKAGE_SIZE, packet_data_buf, nread);
memcpy(packet_with_ack + ACK_SIZE, packet_data_buf, nread);
```

## Klient

Komunikację rozpoczyna klient. Dzieli on plik na mniejsze fragmenty i z każdego z nich tworzy pakiet o opisanej wyżej strukturze. Klient wysyła pakiet do serwera i oczekuje na zwrotne potwierdzenie ACK (acknowledgement) przez określony czas (TIMEOUT). Jeśli w tym czasie potwierdzenie nie nadejdzie, klient uznaje pakiet za utracony i dokonuje jego retransmisji.

```
// Ustawienie TIMEOUT
tv.tv_sec = TIMEOUT_SEC;
tv.tv_usec = 0;
setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
. . .
while (1) { // Główna pętla klienta
  if (nread == 0) {
    printf("Sending EOF package to the server...\n");
  } else {
    printf("Sending %d package to the server...\n", seq_num);
  }

  if (send(sock, packet_with_ack, nread + ACK_SIZE, 0) < 0) {
    perror("send");
    continue;
  }
  if (recv(sock, received_ack_buf, ACK_SIZE, 0) < 0) {
    printf("Timeout, packet lost. Trying once again...\n");
    continue;
  }
  int ack_seq;
  memcpy(&ack_seq, received_ack_buf, ACK_SIZE);
  if (ntohl(ack_seq) == seq_num) {
    break;
  } else {
    printf("Wrong ACK. Trying once again...\n");
  }
}
```

## Serwer

Serwer utrzymuje licznik oczekiwanego numeru sekwencyjnego. Po otrzymaniu pakietu sprawdza, czy zawarty w nim numer zgadza się z wartością oczekiwaną.

- Jeśli numer sekwencyjny jest zgodny z oczekiwanym, serwer inkrementuje licznik, dopisuje otrzymane dane do bufora i wysyła sygnał ACK potwierdzający odebranie pakietu.
- Jeśli otrzymany numer jest mniejszy niż oczekiwany, oznacza to, że wysłane wcześniej przez serwer potwierdzenie (ACK) nie dotarło do klienta lub opóźniło się, co wymusiło retransmisję. W takiej sytuacji serwer ponownie wysyła potwierdzenie dla tego pakietu (nie zapisując jednak dublujących się danych).
- Sytuacja, w której numer sekwencyjny od klienta jest większy niż oczekiwany, jest niemożliwa – w przyjętym protokole klient nie wyśle kolejnego pakietu bez otrzymania potwierdzenia poprzedniego.

```
if seq_num == expected_seq_num:
    s.sendto(struct.pack('!I', seq_num), address)
    if len(packet_data) == 0:
        print("EOF received")
        break
    received_data.extend(packet_data)
    expected_seq_num += 1
elif seq_num < expected_seq_num:
    s.sendto(struct.pack('!I', seq_num), address)
```

# Konfiguracja testowa

- Plik o wielkości 10000B zostaje utworzony z losowych bajtów z /dev/urandom.
- Dla klienta dodano 200ms czekania pomiędzy wysyłaniem kolejnych wiadomości.
- Serwer nasłuchuje na porcie 8080. Klient łączy się na ten port.
- Każdy kontener w sieci z36_network dostaje dynamicznie przydzielony prywatny adres IP z podsieci Dockera. Kontenery mogą komunikować się między sobą zarówno po tym adresie, jak i po aliasie nadanym przez —network-alias.
- Zostały wprowadzone następujące opóźnienia: opóźnienie 1000 ms z rozrzutem 500 ms i prawdopodobieństwem zagubienia pakietu równym 50%. Są one wprowadzone po stronie klienta: `docker exec z36_cclient1 tc qdisc add dev eth0 root netem delay 1000ms 500ms loss 50%`
- Do określenia poprawności przesłania pliku użyto funkcji hashującej SHA256.

# Napotkane problemy

- **Brak konfirmacji EOF:** Początkowa wersja komunikacji zakładała, że po wysłaniu EOF klient wylicza hash i kończy działanie, a serwer kończy działanie, kiedy otrzyma pakiet z EOF. Przy uruchomieniu wraz z gubieniem pakietów okazało się, że to nie działa, kiedy zostaje zgubiony pakiet EOF. Dlatego wprowadzono zmianę, że klient wysyła EOF do skutku, aż serwer wyśle potwierdzenie otrzymania takiego pakietu i wtedy dopiero kończy swoje działania.

# Wyniki testów

Poniżej zostały przedstawione fragmenty wydruków klienta i serwera.

```
z36_cclient1  | Sending 96 package to the server...
z36_cclient1  | Timeout, packet lost. Trying once again...
z36_cclient1  | Sending 96 package to the server...
z36_cclient1  | Timeout, packet lost. Trying once again...
z36_cclient1  | Sending 96 package to the server...
z36_cclient1  | Timeout, packet lost. Trying once again...
z36_cclient1  | Sending 96 package to the server...
z36_cclient1  | Timeout, packet lost. Trying once again...
z36_cclient1  | Sending 96 package to the server...
z36_pserver1  | Received seq=96 from ('172.21.36.3', 58733), payload_len=100
z36_cclient1  | Sending 97 package to the server...
z36_cclient1  | Timeout, packet lost. Trying once again...
z36_cclient1  | Sending 97 package to the server...
z36_pserver1  | Received seq=97 from ('172.21.36.3', 58733), payload_len=100
z36_cclient1  | Sending 98 package to the server...
z36_pserver1  | Received seq=98 from ('172.21.36.3', 58733), payload_len=100
```

Rysunek 1: Komunikacja serwera-klienta podczas przesyłania pakietów

```
z36_cclient1  | Sending 99 package to the server...
z36_pserver1  | Received seq=99 from ('172.21.36.3', 58733), payload_len=100
z36_cclient1  | Sending EOF package to the server...
z36_cclient1  | Timeout, packet lost. Trying once again...
z36_cclient1  | Sending EOF package to the server...
z36_pserver1  | Received seq=100 from ('172.21.36.3', 58733), payload_len=0
z36_pserver1  | EOF received
z36_pserver1  | Server reading ended
```

Rysunek 2: Komunikacja serwera-klienta podczas zakończenia przesyłania pakietów

```
z36_pserver1  | Server's hash: 61ce4e80d2b5b19a98de25095b88468bafaca0c0732fbae108c5e8ad5c8d9fd6
z36_cclient1  | Client streaming ended
z36_cclient1  | Client's hash: 61ce4e80d2b5b19a98de25095b88468bafaca0c0732fbae108c5e8ad5c8d9fd6
z36_pserver1 exited with code 0
z36_cclient1 exited with code 0
```

Rysunek 3: Hashe przesyłanego pliku otrzymane odpowiednio przez serwer i klienta

# Wnioski i uwagi

- Z Rysunku 1 widać, że poprawnie działa powtórne wysyłanie zgubionych pakietów, klient wysyła pakiet o tym samym numerze do momentu uzyskania potwierdzenia jego odczytanie przez serwer. Dopiero wtedy przesyła pakiet o kolejnym numerze.
- Rysunek 2 pokazuje poprawne przesłanie sygnału o końcu pliku, nawet jeśli ten pakiet zostanie "zgubiony" jest on powtórnie przesyłany, aż dotrze i wtedy serwer kończy czytanie.
- Rysunek 3 pokazuje, że poprawnie działa rekonstrukcja pliku po stronie serwera jak i niezawodna transmisja. Wszystkie pakiety zostały poprawnie przesłane i zrekonstruowany został oryginalny plik, co pokazują takie same hashe zarówno obliczone przez klienta jak i serwer.