Sprawozdanie z Laboratorium PSI
Zadanie 1.2

Zespół nr 39
Skład zespołu:
Łukasz Szydlik (lider)
Jan Szwagierczak
Dominik Śledziewski

1 Treść zadania

Celem zadania było zaimplementowanie protokołu niezawodnej transmisji danych (RDT) na bazie protokołu UDP. System składa się z klienta (C) i serwera (Python).

Polecenia:

- Stworzenie pliku binarnego o rozmiarze 10 000 bajtów z losowymi danymi.
- Przesłanie pliku o rozmiarze 10 000 bajtów w paczkach po 100 bajtów.
- Weryfikacja integralności danych poprzez porównanie skrótu (hash) pliku przed wysłaniem i po odebraniu.
- Implementacja mechanizmu potwierdzeń (ACK) i retransmisji w przypadku utraty pakietów.
- Przetestowanie rozwiązania w środowisku symulującym utratę pakietów.

2 Opis rozwiązania

2.1 Architektura i Protokół

Zastosowano mechanizm **Stop-and-Wait ARQ**. Klient wysyła jeden pakiet i oczekuje na potwierdzenie (ACK) od serwera przed wysłaniem kolejnego. W przypadku braku ACK w określonym czasie (*timeout*), pakiet jest retransmitowany.

Zdefiniowano własny nagłówek protokołu aplikacyjnego, wspólny dla C i Pythona:

- **Payload Size (4B):** Rozmiar danych w pakiecie.
- **Seq ID (4B):** Numer sekwencyjny pakietu.
- **Status (2B):** Flaga operacji (1=START, 0=IN_PROGRESS, 2=FIN, 3=ACK).

2.2 Implementacja Klienta (C)

Klient w C odczytuje wygenerowany plik `random.bin`, dzieli go na 100 fragmentów po 100 bajtów i wysyła je sekwencyjnie do serwera UDP. Przed wysyłką wylicza hash DJB2 całego pliku i wypisuje go na standardowe wyjście jako `CLIENT HASH`.

Dla każdego pakietu przygotowywany jest nagłówek `udp_header_t`, w którym pola liczbowe są zamieniane na porządek sieciowy za pomocą funkcji `htonl`/`htons`. Ważnym elementem jest konfiguracja timeoutu dla gniazda za pomocą opcji `SO_RCVTIMEO` (2 sekundy), co pozwala wykrywać zgubione pakiety lub potwierdzenia.

Fragment implementacji pętli retransmisji po stronie klienta:

```
for (int i = 0; i < N_CHUNKS; i++) {
    char packet[HEADER_SIZE + CHUNK_SIZE];
    prepare_header((udp_header_t*)packet, CHUNK_SIZE, i,
                   TRANSMISSION_IN_PROGRESS);
    memcpy(packet + HEADER_SIZE,
           file_data + (i * CHUNK_SIZE),
           CHUNK_SIZE);

    while (1) {
        sendto(sockfd, packet, HEADER_SIZE + CHUNK_SIZE, 0,
               (const struct sockaddr *)&servaddr, len);

        int n = recvfrom(sockfd, buffer, sizeof(buffer), 0, NULL, NULL);
        if (n >= HEADER_SIZE) {
            memcpy(&recv_hdr, buffer, HEADER_SIZE);
            uint32_t ack_seq = ntohl(recv_hdr.seq_id);
            uint16_t ack_flag = ntohs(recv_hdr.status_flag);

            if (ack_flag == ACK_FLAG && ack_seq == i) {
                printf("INFO: Got ACK form %dth chunk\n", i);
                break; // Sukces, przejscie do nastepnego pakietu
            }
        }
        printf("Timeout or invalid ACK for chunk %d\n", i);
    }
}
```

Listing 1: Pętla retransmisji w kliencie C

Na początku oraz na końcu transmisji klient wysyła dodatkowe pakiety sterujące z flagami `START_TRANSMISSION` oraz `END_TRANSMISSION`, oczekując odpowiednich ACK od serwera.

2.3 Implementacja Serwera (Python)

Serwer w Pythonie nasłuchuje na porcie UDP 8000. Dane odbierane są funkcją `recvfrom` do bufora, a następnie z pierwszych bajtów pakietu parsowany jest nagłówek w formacie `!IIH` (moduł `struct`). Serwer utrzymuje:

- zmienną `expected_seq` – numer oczekiwanego pakietu,
- flagę `started` – czy transmisja została rozpoczęta,
- bufor `file_buf` typu `bytearray` do rekonstrukcji pliku.

Logika przetwarzania pakietów:

- **START:** zainicjalizowanie transmisji, wyczyszczenie bufora, opcjonalne zapisanie pierwszych danych i odesłanie ACK.

- **IN_PROGRESS:** jeśli `seq_id == expected_seq`, dane są dopisywane do bufora, licznik jest zwiększany i wysyłany jest ACK; jeśli `seq_id < expected_seq`, traktowane jest to jako duplikat i serwer ponownie wysyła ACK bez zapisu danych.
- **FIN:** zakończenie transmisji, dopisanie ewentualnych końcowych danych, obliczenie hashu DJB2 dla całego zrekonstruowanego pliku (drukowane jako `SERVER HASH`) i odesłanie finalnego ACK.

Przykładowy fragment pętli serwera:

```
while True:
    data, client_addr = server_socket.recvfrom(MAX_BUFFER_SIZE)

    if len(data) < HEADER_SIZE:
        continue

    payload_size, seq_id, status = struct.unpack(
        HEADER_FORMAT, data[:HEADER_SIZE]
    )
    payload = data[HEADER_SIZE:]

    # ... logika dla FLAG_START, FLAG_IN_PROGRESS, FLAG_FIN ...

    if send_ack:
        ack_packet = struct.pack(HEADER_FORMAT, 0, seq_id, FLAG_ACK)
        server_socket.sendto(ack_packet, client_addr)
```

Listing 2: Główna pętla serwera UDP w Pythonie

Funkcja `djb2_hash` implementuje 64-bitową wersję algorytmu DJB2, a wynik wypisywany jest w logach serwera po otrzymaniu pakietu z flagą `FLAG_FIN`.

Struktura nagłówka po stronie klienta w C (zgodna z formatem `!IIH` po stronie serwera w Pythonie):

```
#pragma pack(push, 1)
typedef struct {
    uint32_t payload_size;  // Network byte order
    uint32_t seq_id;
    uint16_t status_flag;
} udp_header_t;
#pragma pack(pop)
```

Listing 3: Struktura nagłówka w C

3 Konfiguracja środowiska testowego

Środowisko uruchomieniowe oparto na kontenerach Docker.

- **Serwer:** Obraz bazowy `python:3`.
- **Klient:** Obraz bazowy `gcc:4.9` z zainstalowanym pakietem `iproute2` (do symulacji błędów).
- **Symulacja błędów:** Wykorzystano narzędzie `tc` (Traffic Control) do wprowadzenia zadanej procentowej utraty pakietów na interfejsie sieciowym klienta.

Komenda symulująca 30% utraty pakietów:

```
tc qdisc add dev eth0 root netem loss 30%
```

4 Przebieg testów i wyniki

4.1 Test 1: Transmisja idealna (0% strat)

W warunkach idealnych wszystkie 100 pakietów zostało przesłanych i potwierdzonych natychmiastowo. Hash pliku zgodny:

```
CLIENT HASH: 9227577407390417097
SERVER HASH: 9227577407390417097
```

Logi klienta pokazują kolejno odebrane potwierdzenia:

```
INFO: Got ACK form 0th chunk
...
INFO: Got ACK form 99th chunk
INFO: Finalized connection with z39_server_app2:8000
```

4.2 Test 2: Transmisja z zakłóceniami (30% strat)

Test ten weryfikuje działanie mechanizmu ARQ przy umiarkowanej utracie pakietów. W logach widoczne są liczne komunikaty o timeoutach, po których następuje ponowna wysyłka i (ostatecznie) otrzymanie ACK:

```
INFO: Got ACK form 0th chunk
Timeout Error.
Timeout Error.
INFO: Got ACK form 1th chunk
...
INFO: Got ACK form 32th chunk
Timeout Error.
Timeout Error.
INFO: Got ACK form 33th chunk
```

Listing 4: Fragment logów przy 30% utracie pakietów

Pomimo wielokrotnych utrat pakietów (zarówno danych, jak i potwierdzeń), plik został przesłany w całości poprawnie, co potwierdzają identyczne skróty DJB2 obliczone przez klienta i serwer:

- **CLIENT HASH:** 9227577407390417097
- **SERVER HASH:** 9227577407390417097

4.3 Test 3: Transmisja z bardzo dużą liczbą strat (60% strat)

Dla 60% symulowanej utraty pakietów liczba timeoutów znacząco wzrasta, jednak zastosowany protokół nadal umożliwia ukończenie transmisji. W logach widać liczne sekwencje:

```
Timeout Error.
Timeout Error.
...
INFO: Got ACK form Nth chunk
```

dla kolejnych numerów pakietów. Ostatecznie klient otrzymuje ACK dla wszystkich 100 segmentów oraz finalne potwierdzenie zakończenia transmisji:

```
INFO: Got ACK form 99th chunk
INFO: Finalized connection with z39_server_app2:8000
```

Hash po stronie serwera pozostaje identyczny z hashem klienta:

```
CLIENT HASH: 9227577407390417097
SERVER HASH: 9227577407390417097
```

5 Wnioski

1. Zaimplementowany protokół typu *Stop-and-Wait* skutecznie gwarantuje dostarczenie danych w środowisku z dużą utratą pakietów (testy dla 30% i 60% strat).

2. Mechanizm timeoutów po stronie klienta w C (oparty o `SO_RCVTIMEO`) poprawnie wykrywa brak potwierdzenia i inicjuje retransmisję danego segmentu.

3. Klient w C poprawnie przygotowuje nagłówki w porządku sieciowym (funkcje `htonl`/`htons`), a serwer w Pythonie interpretuje je za pomocą formatu `'!IIH'` modułu `struct`, dzięki czemu wymiana danych binarnych jest spójna między oboma językami.

4. Obliczone hashe DJB2 są identyczne po stronie klienta i serwera we wszystkich scenariuszach testowych, co dowodzi integralności przesłanych danych mimo występujących strat pakietów.