Data sporządzenia - 26.11.2025

PSI – KOMUNIKACJA TCP (Zad 2.)

Zespół 32: **Katarzyna Kanicka,** Jan Mizera, Andrii-Stepan Pryimak

Treść zadania

Napisz zestaw dwóch programów – klienta i serwera komunikujących się poprzez TCP. Klient oraz serwer musi być napisany w konfiguracji C + Python (do wyboru co w czym).

Klient wysyła do serwera działanie, które serwer ma obliczyć i odesłać odpowiedź do klienta. Dla uproszczenia proszę przyjąć, że klient wysyła 3 wiadomości w ramach działania: liczba1, znak „*”, liczba2. Po otrzymaniu tych 3 wiadomości serwer oblicza „liczba1 * liczba2” i odsyła wynik do klienta. Klient i serwer wypisują działanie oraz wynik w konsoli. Program klienta ma być interaktywny, tzn. po uruchomieniu to użytkownik wpisuje działanie na zasadzie: liczba1, enter, „*”, enter, liczba2, enter → powinien pokazać się wynik. Proszę założyć, że użytkownik zawsze wpisze dobre liczby i znak działania (nie tracić czasu na walidacje).

Opis rozwiązania problemu

Server:

Został napisany w języku Python. Otwiera gniazdo TCP, adresem jest (0.0.0.0, 5000).

Po uzyskaniu połączenia z klientem czeka na trzy wiadomości: liczba 1, znak, liczba 2 (w tej konkretnej kolejności).

Po otrzymaniu każdej z nich, przetwarza liczby i mapuje znak, uzyskując ostateczne działanie. Po jego obliczeniu wysyła wynik końcowy do klienta.

```python
def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()

        print(f"Server {HOST} listening on port {PORT}")

        while True:
            conn, addr = s.accept()
            with conn:
                print(f"Connected to {addr}\n")

                num1 = recv_message(conn)
                if not num1:
                    continue

                operation = recv_message(conn)
                if not operation:
                    continue

                num2 = recv_message(conn)
                if not num2:
                    continue

                print("-" * 50)
                print(f"Operation to perform: {num1} {operation} {num2}")
                sum_result = perform_operation(float(num1), float(num2), operation)
                send_message(conn, f"Result: {sum_result}")
```

```python
def send_message(conn, message):
    try:
        conn.sendall(message.encode())
        print(f"Sent message: {message}")
    except Exception as e:
        print(f"Error sending message: {e}")
        return False
    return True

def recv_message(conn):
    data = conn.recv(BUFSIZE).decode().strip()
    print(f"Received message: {data}")
    if not data:
        print("No data received, closing connection.")
    return data
```

```python
def perform_operation(num1, num2, operation):
    if operation == '+':
        return num1 + num2
    elif operation == '-':
        return num1 - num2
    elif operation == '*':
        return num1 * num2
    elif operation == '/':
        if num2 != 0:
            return num1 / num2
        else:
            return "Error: Division by zero"
    else:
        return "Error: Unknown operation"
```

Klient:

Implementacja w języku C.

Po połączeniu się z serwerem, oczekuje od użytkownika poszczególnych elementów działania: liczby 1, znak, liczby 2. Wszystkie te wartości są przechowywane jako tekst ASCII. Liczby są umieszczane w buforach o rozmiarze 64 bajty, znak w buforze 8-bajtowym (wraz z \0 na końcu).

Wprowadzenie każdej zmiennej kończy się po naciśnięciu klawisza „enter”, następnie jest on wysyłany do serwera.

Po wprowadzeniu pełnego działania, klient otrzymuje wynik końcowy od serwera.

```c
printf("Connected to server %s:%d\n", HOST, PORT);

char num1[64], num2[64], op[8];
printf("Enter first number: ");
scanf("%63s", num1);
if (send_message(sock, num1, strlen(num1)) < 0) { close(sock); return 1; }

printf("Enter operator (+ - * /): ");
scanf("%7s", op);
if (send_message(sock, op, strlen(op)) < 0) { close(sock); return 1; }

printf("Enter second number: ");
scanf("%63s", num2);
if (send_message(sock, num2, strlen(num2)) < 0) { close(sock); return 1; }

printf("%s %s %s\n", num1, op, num2);
if (recv_message(sock, response, sizeof response) < 0) { close(sock); return 1; }

close(sock);
return 0;
```

```c
static int send_message(int sock, const char *msg, size_t len)
{
    ssize_t sent = send(sock, msg, strlen(msg), 0);
    if (sent < 0) {
        perror("Error sending");
        return -1;
    }
    printf("Sent message to server: %s\n\n", msg);
    return (int)sent;
}
```

```c
static int recv_message(int sock, char *buf, size_t buf_size)
{
    ssize_t nread = recv(sock, buf, buf_size - 1, 0);
    if (nread < 0) {
        perror("Error receiving");
        return -1;
    }
    if (nread == 0) {
        perror("Server closed connection");
        buf[0] = '\0';
        return -1;
    }

    buf[nread] = '\0';
    printf("Received message from server: %s\n", buf);
    return (int)nread;
}
```

Konfiguracja testowa

```bash
#!/bin/bash

docker rm -f z32-client-c
docker build -t z32-client-docker .
docker run -it --network z32_network --name z32-client-c z32-client-docker
```

```c
#define HOST "z32-server-python"
#define PORT 5000
#define BUFSIZE 256
```

```bash
#!/bin/bash

docker rm -f z32-server-python
docker build -t z32-server-docker .
docker run -it --network-alias z32-server-python --network z32_network --name z32-server-python z32-server-docker
```

```
HOST = "0.0.0.0"
PORT = 5000
BUFSIZE = 1024
```

Serwer: z32-server-python (Docker) **Klient:** z32-client-c (Docker)

Sieć: Docker network z32_network **Port:** 5000

Bufor serwera: 1024 bajty (dla odpowiedzi)

Bufor klienta: 256 bajtów (dla odpowiedzi)

Testowanie i wyniki

Proces uruchomienia odbywa się za pomocą skryptów bash. Sposób jest opisany w pliku README.md. Po uruchomieniu program zrobi test dla konfiguracji testowej.

Przykładowe wydruki z konsoli:

Client:

```
Connected to server z32-server-python:5000
Enter first number: 34
Sent message to server: 34

Enter operator (+ - * /): /
Sent message to server: /

Enter second number: 2
Sent message to server: 2

34 / 2
Received message from server: Result: 17.0
```

Opis: Klient prosi użytkownika o wprowadzenie kolejnych wartości danych. Wysłanie wiadomości odbywa się po naciśnięciu klawisza „enter”. Na końcu klient wyświetla pełne działanie i otrzymane od serwera rozwiązanie.

Server:

```
Server 0.0.0.0 listening on port 5000
Connected to ('172.21.32.3', 43770)

Received message: 34
Received message: /
Received message: 2
------------------------------------------------
Operation to perform: 34 / 2
Sent message: Result: 17.0
```

Opis: Serwer loguje otrzymanie poszczególnych wartości, a następnie odtwarza działanie w całości. Po obliczeniu pokazuje wynik końcowy wysyłany do klienta.

Problemy napotkane i rozwiązania

W trakcie realizacji zadania nie wystąpiły istotne problemy techniczne ani trudności implementacyjne. Kod serwera w Pythonie oraz klienta w C działały poprawnie.

Wnioski

Protokół TCP poprawnie utrzymywał komunikację między klientem a serwerem. Wszystkie wiadomości były poprawne, dotarły w odpowiedniej kolejności i zostały utrzymane w tej samej sesji, co pozwoliło odtworzyć działanie.

TCP ma automatyczny mechanizm potwierdzania otrzymania wiadomości, odsyłając pakiet ACK, dlatego nie było potrzeby implementacji wysyłania własnych potwierdzeń przesłanych danych dla każdej krótkiej odpowiedzi (liczba1, znak, liczba2), w przeciwieństwie do protokołu UDP.

Ten protokół jest wydajny do wysyłania kilku krótkich wiadomości i w tym zadaniu nie zaobserwowano opóźnień ani utraty danych.