PSI zad. 2
Andrii Rybachok
Anna Tamelo
Ihor Garvylenko
Data sporządzenia: 27.11.2025
Wersja: 1.0

Treść:

Należało napisać zestaw dwóch programów - klienta i serwera - komunikujących się poprzez protokół TCP. Jeden z programów ma być napisany w języku C, a drugi w Pythonie, w dowolnej konfiguracji.

Zadanie klienta:

- Klient po uruchomieniu działa w trybie interaktywnym, użytkownik wprowadza działanie w trzech krokach:
    1. liczba1, Enter
    2. znak *, Enter
    3. liczba2, Enter
- Klient wysyła do serwera trzy osobne wiadomości: liczba1, *, liczba2.
- Odbiera z serwera wynik mnożenia i wypisuje w konsoli całe działanie oraz wynik.

Zadanie serwera:

- Oczekuje na połączenie TCP od klienta.
- Po utrzymaniu trzech wiadomości (liczba1, *, liczba 2) oblicza liczba1 * liczba2.
- Odsyła wynik do klienta.
- Wypisuje w konsoli całe działanie oraz wynik.

Można założyć, że użytkownik zawsze wpisuje poprawne liczby i poprawny znak działania (*), więc nie jest wymagana walidacja danych wejściowych.

Opis rozwiązania:

1. **Konfiguracja językowa**
   W naszym rozwiązaniu przyjęto konfigurację:
    - Serwer: Python
    - Klient: C
2. **Protokół aplikacyjny**
   Protokół na poziomie aplikacji jest bardzo prosty i opiera się na wiadomościach tekstowych zakończonych znakiem nowej linii ‘\n’.
    1. Klient wysyła do serwera:
        - Linię z pierwszą liczbą: liczba1\n.
        - Linię z operatorem: *\n.
        - Linię z drugą liczbą: liczba2\n.

2. Serwer:
    - Odczytuje trzy linie.
    - Konwertuje liczba1 i liczba2 na typ liczbowy.
    - Wykonuje mnożenie.
    - Wysyła jedną linię z wynikiem: wynik\n.
    - Wyświetla działanie w postaci: liczba1 * liczba2 = wynik.
3. Klient:
    - Odczytuje wynik.
    - Wyświetla działanie w postaci: liczba1 * liczba2 = wynik.
      Użycie '\n' jako separatora pozwala wygodnie czytać dane po stronie serwera (readline() w Pythonie) oraz upraszcza obsługę "ramkowania" wiadomości w TCP, który jest strumieniem bajtów, a nie protokołem wiadomościowym.

3. Serwer (Python) - opis działania

Główne kroki po stronie serwera:

1. Utworzenie gniazda TCP (socket(AF_INET, SOCK_STREAM).
2. Ustawienie opcji SO_REUSEADDR, zbindowanie gniazda na HOST:PORT i przejście w tryb nasłuchu (listen()).
3. Akceptacja połączenia od klienta (accept()).
4. Objęcie połączenia w obiekt plikowy (makefile("r")), aby wygodnie używać readline().
5. Kolejno:
    - line1 = readline()
    - line_op = readline()
    - line2 = readline()
6. Usunięcie znaków końca linii (strip()), konwersja line1 i line2 na int.
7. Obliczenie wyniku: result = a * b.
8. Wypisanie w konsoli serwera: a * b = result
9. Wysłanie wyniku do klienta jako tekst: f"{result}\n".encode("utf-8")
10. Zamknięcie połączenia z klientem.

4. Klient (C) - opis działania

Główne kroki po stronie klienta:

1. Utworzenie gniazda TCP (socket(AF_INET, SOCK_STREAM, 0).
2. Wypełnienie struktury sockaddr_in adresem serwera (127.0.0.1) i portem.
3. Nawiązanie połączenia z serwerem (connect()).
4. Tryb interaktywny:
    - Odczyt z stdin trzech linii: liczba1, operator *, liczba2.
    - Wysyłanie trzech wiadomości tekstowych.
5. Odbiór wyniku za pomocą recv():
    - Dane zbierane są do bufora, aż pojawi się '\n' lub połączenie zostanie zamknięte.
6. Wypisanie z konsoli klienta: liczba1 * liczba2 = wynik.
7. Zamknięcie gniazda (close()).

Napotkane problemy i sposoby ich rozwiązania:

1. **Oczekiwanie serwera na dane (zablokowane recv)**
   Na początku serwer w Pythonie używał trzech wywołań recv(), zakładając, że każde send() z klienta w C będzie osobną “wiadomością” (osobny recv). Okazało się, że w TCP dane przychodzą jako ciągły strumień i wszystkie trzy linie (a, operator, b) często trafiały do pierwszego recv(), a kolejne recv() blokowały się, czekając na dalsze dane, których już nie było. Objawem było to, że klient nie dostawał żadnego wyniku. Problem został rozwiązany przez zmianę logiki: serwer teraz zbiera dane w buforze dopóki nie pojawią się trzy znaki końca linii (\n), a następnie dzieli całość na trzy linie i dopiero wtedy parsuje liczby.
2. **Obsługa znaków końca linii \n**
   Na początku w kliencie i serwerze mieszały się “wbudowane” znaki \n z fgets() z ręcznie dokładanymi \n przy wysyłaniu, co powodowało powstawanie pustych linii (np. "1\n\n") i błędny odczyt danych po stronie serwera. Ostatecznie przyjęto następne rozwiązanie: klient po wczytaniu danych obcina końcowy znak \n, a przy wysyłaniu każdej części wyrażenia dopisuje dokładnie jeden znak \n. Serwer zlicza wystąpienia \n i dopiero po otrzymaniu trzech linii (liczba, operator, druga liczba) przetwarza dane.
3. **Konfiguracja portu i uruchamianie programu**
   Dodatkowym drobnym problemem było uzgodnienie numeru portu między klientem a serwerem oraz obsługa przypadku, gdy użytkownik nie poda portu w argumentach programu. Zostało to rozwiązane przez wprowadzenie domyślnej wartości portu (8000) oraz możliwość przekazania portu jako pierwszego argumentu linii komend zarówno w kliencie, jak i w serwerze. Dzięki temu aplikacje są łatwiejsze do uruchomienia i zgodne z przykładowym stylem z materiałów do laboratorium.

Konfiguracja testowa:

- Adres serwera: 127.0.0.1
- Port serwera TCP: 8000
- Interfejs: loopback
- Organizacja pakietów: linie tekstowe zakończone \n.
- System operacyjny: Linux
- Języki: serwer - Python 3, klient - C (kompilacja gcc).

Testy:

1. Małe liczby dodatnie
   liczba1: 6, liczba2: 7, wartość oczekiwana: 42
   Klient:

```
MacBookPro:psi-lab2 admin$ ./client_tcp
brak portu, uzywam 8000
polaczono 127.0.0.1:8000
a: 6
op: *
b: 7
wynik: 42
expr: 6 * 7 = 42
MacBookPro:psi-lab2 admin$
```

Serwer:

```
polaczenie ('127.0.0.1', 50965)
wyr: 6 * 7 = 42
```

2. Duże liczby dodatnie
   liczba1: 12345, liczba2: 6789, wartość oczekiwana: 83810205
   Klient:

```
MacBookPro:psi-lab2 admin$ ./client_tcp
brak portu, uzywam 8000
polaczono 127.0.0.1:8000
a: 12345
op: *
b: 6789
wynik: 83810205
expr: 12345 * 6789 = 83810205
MacBookPro:psi-lab2 admin$
```

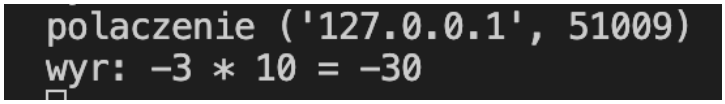
   Serwer:

```
polaczenie ('127.0.0.1', 51084)
wyr: 12345 * 6789 = 83810205
```

3. Liczby ujemne
   liczba1: -3, liczba2: 10, wartość oczekiwana: -30
   Klient:

```
MacBookPro:psi-lab2 admin$ ./client_tcp
brak portu, uzywam 8000
polaczono 127.0.0.1:8000
a: -3
op: *
b: 10
wynik: -30
expr: -3 * 10 = -30
MacBookPro:psi-lab2 admin$
```

   Serwer:



4. Test wielokrotnego uruchomienia
    - Serwer uruchamiany kolejno kilka razy pod rząd.
    - Dzięki SO_REUSEADDR nie występują problemy z ponownym bindowaniem gniazda.

Wnioski:

- TCP znacząco upraszcza kwestię niezawodności (brak konieczności implementowania numeracji pakietów, retransmisji), ale nie zwalnia z myślenia o strukturze danych w strumieniu - konieczne było wprowadzenie separatora ‘\n’ i odpowiedniego odczytu po stronie serwera/klienta.
- Rozdzielenie implementacji na klienta w C i serwera w Pythonie pokazuje, że TCP zapewnia interoperacyjność między różnymi językami programowania - istotne jest tylko trzymanie się jednolitego formatu danych.
- Interaktywny charakter klienta spełnia wymagania zadania: użytkownik w naturalny sposób wprowadza działanie, a wynik jest natychmiast prezentowany w konsoli.