PSI LAB 2- Protokół TCP

13.12.2023, v1.0

Autorzy:

Igor Matynia, Andrii Gamalii, Wiktor Topolski, Bartłomiej Pełka

Treść zadania

Z 2 Komunikacja TCP

Napisać zestaw programów – klienta i serwera – komunikujących się poprzez TCP. Wykonać ćwiczenie w kolejnych inkrementalnych wariantach, rozszerzając kod z poprzedniej wersji.

Z 2.1
Napisać w języku C/Python klienta TCP, który wysyła złożoną strukturę danych, np. uttworzoną w pamięci listę jednokierunkową lub drzewo binarne struktur zawierających (oprócz danych organizacyjnych) pewne dane dodatkowe, np. liczbę całkowitą 16-bitową, liczbę całkowitą 32-bitową oraz napis zmiennej i ograniczonej długości. Serwer napisany w Pythonie/C powinien te dane odebrać, dokonać poprawnego „odpakowania" tej struktury i wydrukować jej pola (być może w skróconej postaci, aby uniknąć nadmiaru wyświetlanych danych). Klient oraz serwer powinny być napisane w różnych językach.
Wskazówka: można wykorzystać moduły Pythona: struct i io.

Z 2.2

Zmodyfikować programy z zadania 2.1 tak, aby posługiwały się IPv6.

Z 2.3a

Zmodyfikować programy z zad. 2.1 w następujący sposób:

Klient powinien wysyłać do serwera strumień danych w pętli (danych powinno być przynajmniej kilkaset kB). Serwer powinien odbierać dane, ale między odczytami realizować sztuczne opóźnienie (np. za pomocą funkcji sleep()). W ten sposób symulowane będzie przeciążenie odbiorcy. Stos TCP będzie spowalniał nadawcę, aby uniknąć tracenia danych. Zidentyfikować objawy tego zjawiska po stronie klienta (dodając pomiar i logowanie czasu, monitorując ruch sieciowy np. za pomocą narzędzi tcpdump lub Wireshark) i krótko przedstawić swoje wnioski poparte uzyskanymi obserwacjami i statystykami czasowymi. Przeprowadzić eksperymenty z różnymi rozmiarami bufora nadawczego po stronie klienta (np. 100 B, 1 kB, 10 kB).
Zadanie należy wykonać korzystając z kodu klienta i serwera napisanych w języku C.

Przesyłana struktura

Opis

Przesyłaną strukturą jest lista jednokierunkowa, której elementy składają się po kolei z pól rating, price, title, content oraz pól związanych z organizacją struktury. Pole rating jest 16-bitową liczbą całkowitą, price 32-bitową liczbą całkowitą. Title jest ciągiem o stałej wielkości 255 znaków. Content natomiast jest ciągiem znaków o zmiennej długości.

Struktura węzła w C:

```c
struct node {
    struct node *next;
    int id;
    int next_id;
    short rating;
    unsigned int price;
    char title[MAX_TEXT_LEN];
    unsigned int content_length;
    char* content;
};
```

Serializacja

Wysyłany bufor danych zawiera liczbę węzłów na początku (int - 4 bajty) i kolejno wszystkie zserializowane węzły listy.

Struktura zserializowanego węzła w C:

```c
struct serialized_node{
    int id;
    int next_id;
    short rating;
    unsigned int price;
    char title[MAX_TEXT_LEN];
    unsigned int content_length;
};
```

Deserializacja

Najpierw węzły są zapisywane, a następnie łączone ze sobą w prawidłową listę poprzez referencje, za pomocą id węzłów.

Klient (python/c)

Serwer (c)

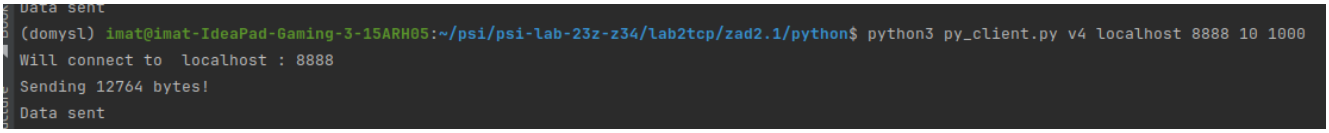
Konfiguracja testowa

Domyślnie serwer zostaje postawiony pod adresem localhost/::1 na porcie 8888. Ilość przesyłanych węzłów struktury to 10, a ilość danych w każdym z węzłów to 1000B. Ostatecznie przesyłanych jest 12764B danych. Każdy i-ty węzeł ma jako “title” ciąg znaków “Book by {i}" a zawartością content jest 1000 znaków “a”.

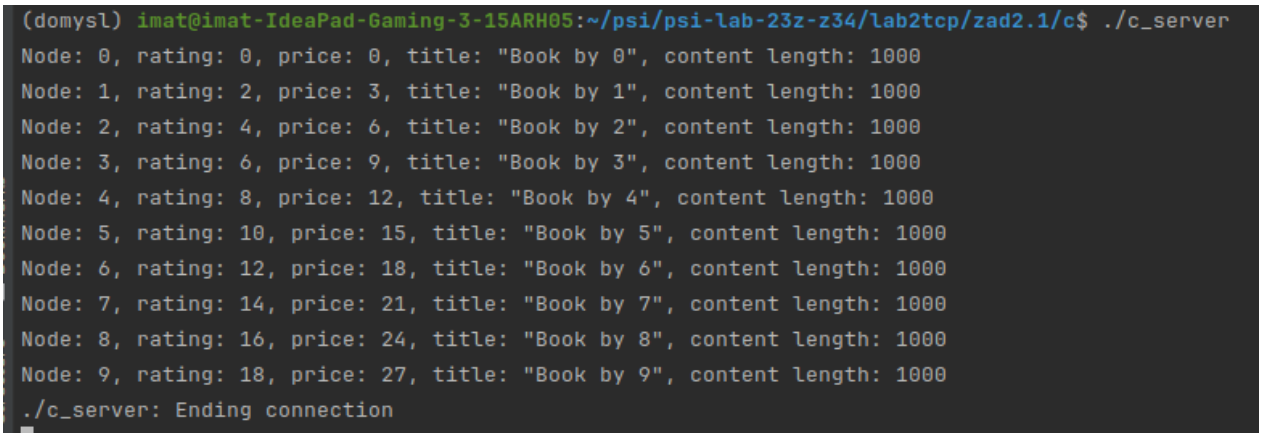
Testy

Zadanie 2.1

Uruchomiony zostaje program c_serwer po czym uruchomiony zostaje klient napisany w pythonie.



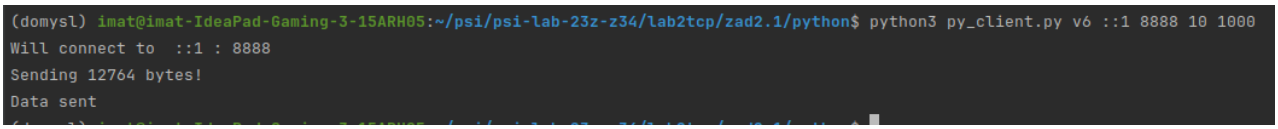
Po stronie serwera ukazuje się komunikat:



Dane zostały wysłane i odebrane pomyślnie.

Zadanie 2.2

Uruchomiony zostaje program c_serwer zadania 2.2, po czym uruchomiony zostaje klient napisany w pythonie z poniższymi argumentami wywołania.



W tym wypadku pierwszym argumentem wywołania było v6, pozwalające na obsługę IPv6. Po pomyślnym wysłaniu, w logach serwera ukazuje się lista:

```
(domysl) imat@imat-IdeaPad-Gaming-3-15ARH05:~/psi/psi-lab-23z-z34/lab2tcp/zad2.2/c$ ./c_server
Socket port #8888
Node: 0, rating: 0, price: 0, title: "Book by 0", content length: 1000
Node: 1, rating: 2, price: 3, title: "Book by 1", content length: 1000
Node: 2, rating: 4, price: 6, title: "Book by 2", content length: 1000
Node: 3, rating: 6, price: 9, title: "Book by 3", content length: 1000
Node: 4, rating: 8, price: 12, title: "Book by 4", content length: 1000
Node: 5, rating: 10, price: 15, title: "Book by 5", content length: 1000
Node: 6, rating: 12, price: 18, title: "Book by 6", content length: 1000
Node: 7, rating: 14, price: 21, title: "Book by 7", content length: 1000
Node: 8, rating: 16, price: 24, title: "Book by 8", content length: 1000
Node: 9, rating: 18, price: 27, title: "Book by 9", content length: 1000
```

Dane zostały wysłane i odebrane pomyślnie.

Zmiany wprowadzone dla zadania 2.2

Klient python:

Dodany został argument wywołania pozwalający na wybór wersji protokołu ip. Wybranie wersji decyduje o tym czy przy inicjalizacji socketu użyjemy wartości AF_INET czy AF_INET6. Przy połączeniu w przypadku IPv6 do krotki adresu należy dodać 2 dodatkowe argumenty: flow info oraz scope id, w naszym wypadku równe 0.

```python
if ip == "v4":
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        s.sendall(buffer)
elif ip == "v6":
    with socket.socket(socket.AF_INET6, socket.SOCK_STREAM) as s:
        s.connect((host, port, 0, 0))
        s.sendall(buffer)
```

Serwer c:

Zmieniona została część kody odpowiedzialna za utworzenie gniazda. Użyte zostało AF_INET6 zamiast AF_INET.

```c
int main(int argc, char **argv) {
    int sock, msgsock, length, rval, ListenQueueSize = 5, read_bytes = 0;
    struct sockaddr_in6 server;
    char buf[BSIZE], bufbuf[65356];
    linked_list_t ll;

    sock = socket( domain: AF_INET6, type: SOCK_STREAM, protocol: 0);
    if (sock<0)  bailout("opening stream socket");

/* dowiaz adres IPv6 do gniazda */
    server.sin6_family = AF_INET6;
//    inet_pton(AF_INET6, DEF_ADDR, &(server.sin6_addr));
    server.sin6_addr = in6addr_any;
    server.sin6_port = htons( hostshort: DEF_PORT);
    if (bind( fd: sock,  addr: (struct sockaddr *) &server,  len: sizeof server) == -1)
    bailout("binding stream socket");

    /* wydrukuj na konsoli przydzielony port */
    length = sizeof(server);
    if (getsockname( fd: sock, addr: (struct sockaddr *) &server, len: &length) == -1)
    bailout("getting socket name");
    printf( format: "Socket port #%d\n", ntohs( netshort: server.sin6_port));
//    char str[INET6_ADDRSTRLEN];
//    inet_ntop(server.sin6_family, &server.sin6_addr, str, INET6_ADDRSTRLEN);
//    printf("Socket address: %s\n", str);
    /* zacznij przyjmować polaczenia... */
    listen( fd: sock,  n: ListenQueueSize);
```

Zadanie 2.3

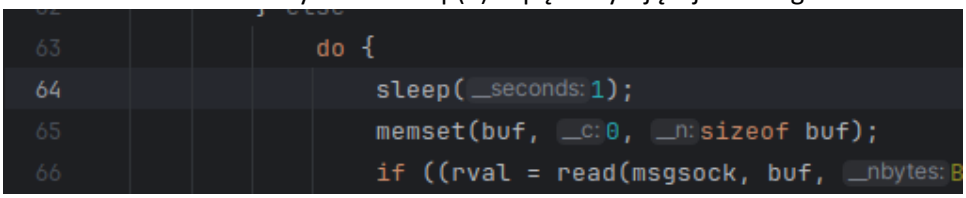
Uruchomiony zostaje program c_serwer zadania 2.3, po czym uruchomiony zostaje klient napisany w c zadania 2.3 z argumentami wywołania “8888 666”, co powoduje połączenie z serwerem nasłuchującym na porcie 8888 i przesłanie listy z 666 elementami (której wielkość to kilkaset kilobajtów)

Zmiany wprowadzone dla zadania 2.3

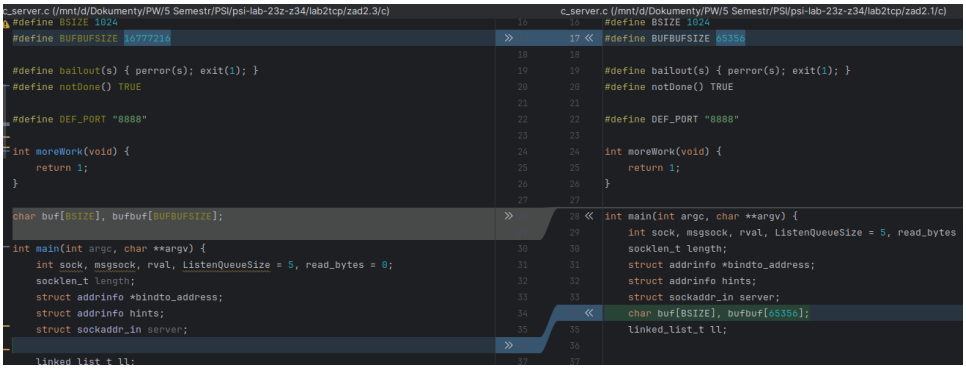
W kodzie klienta dane są teraz wysyłane porcjami po 100 (lub inną zdefiniowaną ilość) bajtów naraz:

```c
  char part_buffer[SEND_BUF_SIZE];
  unsigned int sent = 0, to_send = 0;
  while (sent < buffer_len) {
      to_send = buffer_len - sent > SEND_BUF_SIZE ? SEND_BUF_SIZE : buffer_len - sent;
      memcpy(part_buffer,  __src: buffer_to_send + sent, to_send);
      if (write(sock, part_buffer, to_send) == -1) {
          bailout("writing on stream socket");
      }
      long long send_time = timeInMilliseconds();
      long long delta = send_time - start_time;
      printf( __format: "%d B sent after %lld ms\n", sent, delta);
      sent += to_send;
  }
```

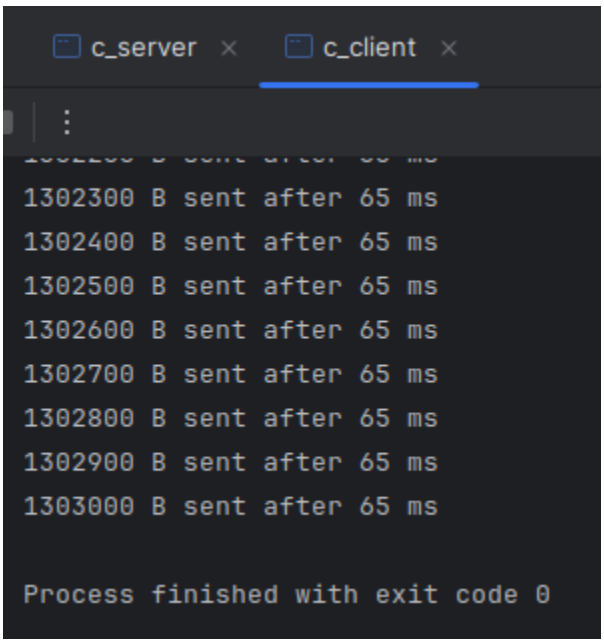
W kodzie serwera dodany został sleep(1) w pętli czytającej dane z gniazda:



Rozmiar bufora również został znacznie zwiększony

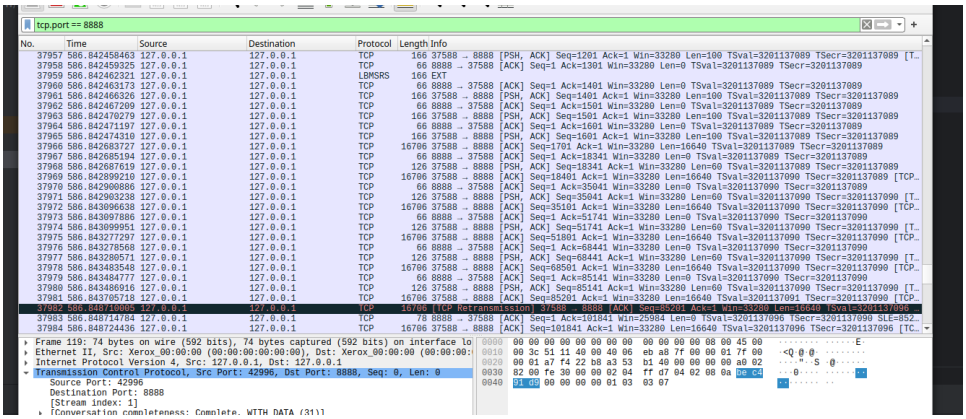


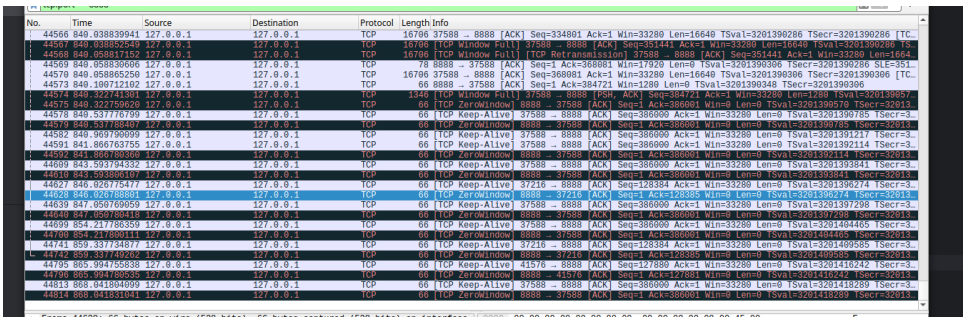
Wyniki zadania 2.3



Klientowi przy każdej testowanej wielkości wysyłanego pakietu udaje się od razu wysłać całą listę w bardzo krótkim czasie.

Z perspektywy Wiresharka przesłane pakiety wyglądają tak:





Występuje sklejanie pakietów tcp oraz ich ewentualna retransmisja.
Serwer ostrzega że bufor mu się skończył, a klient już dawno przestał nadawać