Programowanie Sieciowe – Laboratorium 2

1. Informacje ogólne

Nr zespołu: 4

Prowadzący: Grzegorz Blinowski

Skład: Łukasz Jaremek, Daniel Kobiałka, Radosław Kostrzewski, Hubert Soroka

Data sporządzenia: 15.12.2022

Wersja: 1

Realizowane zadania: Z2.1, Z2.2, Z2.5

Kod zespołu na bigubu: z14

2. Treść zadania

- Z 2

Napisz zestaw dwóch programów – klienta i serwera wysyłające dane w protokole TCP. Wykonaj ćwiczenie w kolejnych inkrementalnych wariantach (rozszerzając kod z poprzedniej wersji). W wariancie Python należy początkowo w kodzie klienta i serwera użyć funkcji sendall().

- Z 2.1

Klient wysyła, serwer odbiera porcje danych o stałym, niewielkim rozmiarze (rzędu kilkudziesięciu bajtów). Mogą one zawierać ustalony „na sztywno" lub generowany napis – np. „abcde....", „bcdef...", itd. Po wysłaniu danych klient powinien kończyć pracę. Serwer raz uruchomiony pracuje aż do zabicia procesu.

Wykonać program w dwóch wariantach: C oraz Python.

Sprawdzić i przetestować działanie „między platformowe", tj. klient w C z serwerem Python i vice versa.

- Z 2.2

Zmodyfikować program serwera tak, aby bufor odbiorczy był mniejszy od wysyłanej jednorazowo przez klienta porcji danych. W wariancie Python wykonać eksperymenty z funkcjami send() i sendall(). Jak powinien zostać zmodyfikowany program klienta i serwera, aby poprawnie obsłużyć komunikację? (uwaga – w zależności od wykonanej implementacji programu z punktu 2.1 mogą ale nie muszą poprawnie obsługiwać wariant transmisji z punktu 2.2)

- Z 2.5

Na bazie wersji 2.1 – 2.2 zmodyfikować serwer tak, aby miał konstrukcję współbieżną, tj. obsługiwał każdego klienta w osobnym procesie. Przy czym:

- Dla C. Należy posłużyć się funkcjami fork() oraz (obowiązkowo) wait().

- Dla Pythona należy posłużyć się wątkami, do wyboru: wariant podstawowy lub skorzystanie z ThreadPoolExecutor

3. Opis rozwiązania problemu

- Z 2.1
    - Klient:
        - Python – z14_client_21_py:

W przypadku pythonowej implementacji, rozwiązanie jest bardzo proste. Klient tworzy gniazdo SOCK_STREAM, wywołuje na nim connect() dla podanego adresu ip i portu, co zestawia połączenie z serwerem, po czym używa sendall() by wysłać serwerowi komunikat „hello from python TCP client" zakodowany w UTF-8.

Klient kończy działanie po wysłaniu komunikatu jeden raz.

Nie występuje to żadna większa logika poza wykorzystaniem typowych funkcji gniazd w sposób przewidywalny i typowy.

- C – z14_client_21_c:

W przypadku C, istnieje większy narzut związany z używaniem funkcji gniazd, lecz logika pozostaje taka sama.

Klient tworzy gniazdo SOCK_STREAM, po czym parsuje parametry (adres i port serwera) i przygotowuje strukturę potrzebną do zlokalizowania serwera. Pola struktury typu sockaddr_in są uzupełniane adresem serwera, portem serwera, długością adresu serwera i rodziną adresów (w tym przypadku AF_INET). Następnie klient wywołuje connect() dla przygotowanej struktury adresowej i wysyła dane z użyciem write. Zaletą zestawionego połączenia TCP jest właśnie możliwość korzystania z funkcji do obsługi plików, jak write().

Logowany jest wysłany komunikat, czyli „hello from TCP C client".

Potem program zamyka gniazdo i kończy działanie.

- Serwer:

W przypadku serwerów, ich adres jest ustalany przez skrypt kontenera (omówiony później), więc tylko port jest parametrem wywołania.

- Python – z14_server_21_py:

Serwer tworzy gniazdo, po czym wywołuje bind na adres 0.0.0.0 oraz podany port. Pozwala to połączyć się do serwera na podanym porcie i adresie wyspecyfikowanym w skrypcie tworzącym kontener (gniazdo nasłuchuje na wszystkich [w tym przypadku jednym] interfejsach sieciowych). Następnie, program wchodzi w nieskończoną pętle i czeka na dane na połączenie wywołując funkcję accept(). Następnie na deskryptorze gniazda zwróconego przez accept() wywoływane jest recv() i na konsoli wypisywane są otrzymane dane. Maksymalny rozmiar to 1024 bajty, po otrzymaniu serwer dekoduje z UTF-8. Po wypisaniu deskryptor gniazda zwrócony przez accept() jest zamykany.

Program nie przestaje działać, ciągle czeka na dane. Aby wyłączyć, należy użyć CTRL+C lub „docker kill z14_server_21_py".

- C – z14_server_21_c:

Podobnie, jak wcześniej, tworzone jest gniazdo i uzupełniane pola struktury sockadd_in. Jako adres podajemy INADDR_ANY, co spowoduje nasłuchiwanie na każdym interfejsie. Następnie wołane jest

bind dla podanego jako parametr portu, po czym, w nieskończonej pętli, serwer czeka na połączenia wywołując accept(). Po otrzymaniu gniazda z accept(), czytane są z niego dane dopóki będzie co czytać Warto pamiętać o czyszczeniu bufora na odbierane dane, ponieważ dane odebrane z gniazda nie zawsze muszą zapełnić cały bufor.

Odebrane dane są logowane na konsoli w porcjach, w jakich zostały odebrane.

Program będzie działał wiecznie, o ile nie zawołamy CTRL+C lub „docker kill z14_server_21_c".

- Z 2.2
    - Klient
        - Python - z14_client_22_py:

Od klienta python 2.1 różni się tylko i wyłącznie użyciem send() zamiast sendall() do wysyłania danych.

    - Serwer
        - Python - z14_server_22_py:

Serwer 2.2 od serwera 2.1 różni się rozmiarem bufora na dane (10 bajtów zamiast 1024, jest to mniej niż potrzebne do odebrania całego komunikatu naraz) oraz tym, że odbiera dane w pętli, zamiast jednorazowo. Dzięki temu, klienci z send() i sendall() działają tak samo – przesyłają cały komunikat podzielony na części. Bez pętli, klient z send() przesłałby tylko pierwszy kawałek komunikatu.

Otrzymane odkodowane dane są wypisywane na konsoli.

        - C – z14_server_22_c:

Od serwera C z 2.2 różni się tylko i wyłącznie rozmiarem bufora (10 zamiast 1024 bajtów).

- Z 2.5
    - Serwer
        - Python - z14_server_25_py:

Od serwera z 2.1 różni się tym, że po otrzymaniu połączenia, wywołuje nowy wątek, który otrzymuje jako parametry zwrócony przez accept() deskryptor gniazda i adres połączenia. Wątek w pętli odbiera dane z gniazda, dopóki są, i wypisuje je na konsolę. Na koniec zamyka gniazdo.

        - C - z14_server_25_c:

Wersja w C inaczej operuje współbieżnością. Zamiast tworzyć wątek wykonujący podaną funkcję która obsługuje połączenie, wołane jest fork(), które klonuje obecnie wykonywany proces i następnie proces potomny obsługuje gniazdo identycznie jak inne serwery w C, a proces nadrzędny oczekuje na zakończenie pracy procesu potomnego.

4. Uwagi dotyczące problemów podczas realizacji

Podczas realizacji nie natrafiliśmy na duże problemy.

Jedyne zagadnienie nad którym musieliśmy przysiąść to realizacja współbieżnego serwera w C. Wynika to z mniej intuicyjnej obsługi wątków i procesów w C niż w pythonie.

5. Konfiguracja testowa

Każdy folder „server” i „client” zawiera w sobie pliki potrzebne do uruchomienia kontenera realizującego odpowiednią funkcjonalność. Są to:

- Dockerfile – zawiera instrukcje do tworzenia kontenera, jest bardzo prosty, uruchamia program z obecnego folderu

- build.sh – skrypt usuwający obraz kontenera i budujący go na nowo – wymagane uruchomienie przy każdej zmianie w kodzie

- run.sh – skrypt uruchamiający kontener w przydzielonej podsieci z odpowiednią nazwą. Dla serwerów przydziela też IP następująco:

172.21.14.7 – 2.5 C

172.21.14.6 – 2.5 python

172.21.14.5 – 2.2 C

172.21.14.4 – 2.2 python

172.21.14.3 – 2.1 C

172.21.14.2 – 2.1 python

Dodatkowo, skrypt przekazuje argumenty do programu serwera / klienta, więc poprawne wywołanie np. klienta C 2.1 wygląda następująco:

sh run.sh 172.21.14.3 1414

Po wywołaniu, prosta wiadomość tekstowa zostanie wysłana do serwera C z zadania 2.1 na port 1414 (może być inny efemeryczny port, 1414 był używany do testów).

6. Opis testów

- Z 2.1

Sprawdzono, czy klienci mogą wysłać dane do obu serwerów oraz czy otrzymane dane się zgadzają.

Uruchomienie klienta wymagało użycia poniższych komend w katalogu odpowiedniego klienta:

sh build.sh

sh run.sh 172.21.14.2 / 172.21.14.3 1414

Uruchomienie serwera wymagało użycia poniższych komend w odpowiednim katalogu serwera:

sh build.sh

sh run.sh 1414

- Wyniki testów:

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/C_implementation/client$ sh run.sh 172.21.14.3 1414
Client sent : hello from TCP C client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.3 1414
Client sent : hello from python TCP client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/C_implementation/server$ sh run.sh 1414
C server received : hello from TCP C client
Connection ended
C server received : hello from python TCP client
Connection ended
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.2 1414
Client sent : hello from python TCP client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/C_implementation/client$ sh run.sh 172.21.14.2 1414
Client sent : hello from TCP C client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/server$ sh run.sh 1414
Python server received : hello from python TCP client
Python server received : hello from TCP C client
```

Jak widać, oba serwery przyjmują dane od obu klientów.

- Z 2.2
    - Wyniki testów

```
hsoroka@bigubu:~/python-c-socketing-psi/2_2/python_implementation/client$ sh run.sh 172.21.14.4 1414
Client sent : hello from python TCP client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.4 1414
Client sent : hello from python TCP client
```

(wariant z send() i sendall())

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/C_implementation/client$ sh run.sh 172.21.14.4 1414
Client sent : hello from TCP C client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_2/python_implementation/server$ sh run.sh 1414
Python server received : hello from
Python server received :  python TC
Python server received : P client
Python server received : hello from
Python server received :  python TC
Python server received : P client
Python server received : hello from
Python server received :  TCP C cli
Python server received : ent
```

Jak widać serwer otrzymał trzy komunikaty, każdy obsłużył w ten sam sposób. Ponieważ bufor gniazda jest zbyt mały, aby pomieścić w nim cały napis, napis jest odbierany w częściach. Jest to spodziewany efekt.

Serwery 2.2 przerabiają dane z bufora w pętli (pythonowy serwer 2.1 odbiera dane jednokrotnie, ale ma dostatecznie duży bufor), dzięki czemu nie ma różnicy pomiędzy działaniem funkcji send i sendall. Gdyby serwer odbierał dane jednokrotnie ze zbyt małym buforem, otrzymywałby tylko pierwszą część komunikatu.

```
hsoroka@bigubu:~/python-c-socketing-psi/2_2/C_implementation/server$ sh run.sh 1414
C server received : hello from@
C server received :  TCP C cli@
C server received : ent
Connection ended
C server received : hello from@
C server received :  python TC@
C server received : P client
Connection ended
C server received : hello from@
C server received :  python TC@
C server received : P client
Connection ended
```

Dla serwera w C, sytuacja wygląda analogicznie. Warto zwrócić uwagę na znak @ na końcu linii, wynikający z konieczności pamiętania o znaku zakończenia łańcucha znaków w C.

- Z 2.5
    - Wyniki testów:

```
hsoroka@bigubu:~/python-c-socketing-psi/2_2/python_implementation/client$ sh run.sh 172.21.14.6 1414
Client sent : hello from python TCP client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/C_implementation/client$ sh run.sh 172.21.14.6 1414
Client sent : hello from TCP C client
```

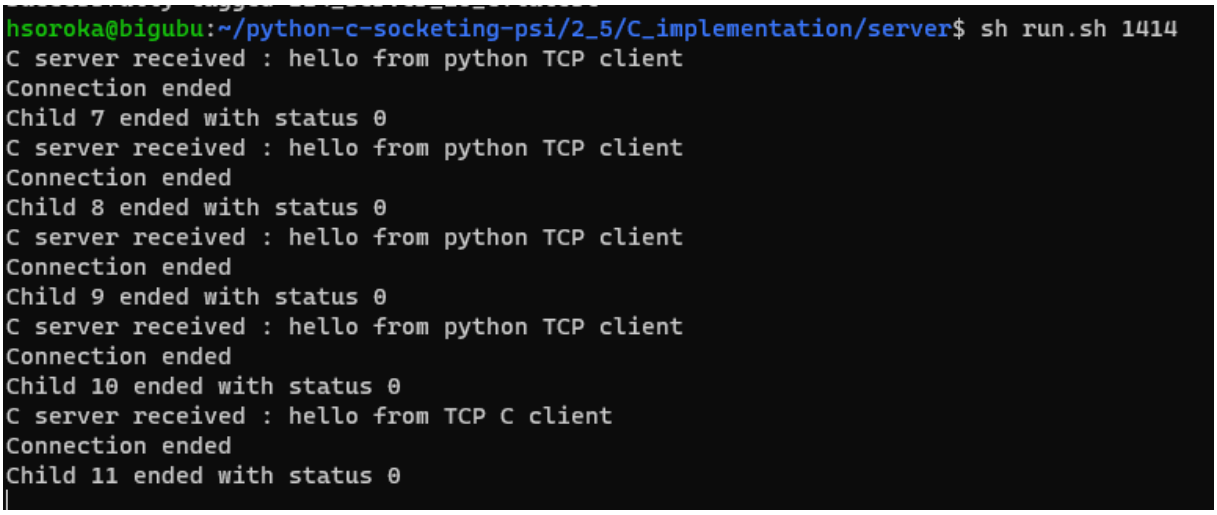
```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.6 1414
Client sent : hello from python TCP client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_5/python_implementation/server$ sh run.sh 1414
Python server received : hello from python TCP client
Python server received : hello from TCP C client
Python server received : hello from python TCP client
```

Jak widać, wielowątkowy serwer w pythonie poprawie odbiera połączenia od klientów.

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.7 1414
Client sent : hello from python TCP client
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.7 1414
Client sent : hello from python TCP client
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.7 1414
Client sent : hello from python TCP client
hsoroka@bigubu:~/python-c-socketing-psi/2_1/python_implementation/client$ sh run.sh 172.21.14.7 1414
Client sent : hello from python TCP client
```

```
hsoroka@bigubu:~/python-c-socketing-psi/2_1/C_implementation/client$ sh run.sh 172.21.14.7 1414
Client sent : hello from TCP C client
```



Jak widać, współbieżny serwer w C też poprawnie odbiera połączenia od klientów, procesy potomne kończą się ze statusem 0, a więc sukcesem.

7. Wnioski i uwagi

Na pewno należy zwrócić uwagę na zalety przetwarzania danych z bufora w pętli, której warunkiem jest jego niepustość. W ten sposób umożliwiamy odczytywanie danych większych niż rozmiar bufora niezależnie od funkcji użytej po stronie klienta.

Ponownie zwraca na siebie uwagę komfort i szybkość pisania tego samego kodu w C i pythonie, pisząc tak prosty program, zdecydowanie wygodniej jest korzystać z pythona.

Warto też zwrócić uwagę, że w wypadku serwera współbieżnego można używać zarówno wątków jak i procesów. Dla prostych przykładów nie było większej różnicy, lecz przy większym systemie należy zastanowić się nad kwestiami zasobów współdzielonych czy komunikacji, które mogą wyglądać inaczej dla wątków i procesów potomnych.