Programowanie Sieciowe – Laboratorium 1

1. Informacje ogólne

Nr zespołu: 4

Prowadzący: Grzegorz Blinowski

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

Data sporządzenia: 02.12.2022

Wersja: 1

Realizowane zadania: Z1.1, Z1.3

Kod zespołu na bigubu: z14

2. Treść zadania

- Z 1

Napisz zestaw dwóch programów – klienta i serwera wysyłające datagramy UDP. Wykonaj ćwiczenie w kolejnych inkrementalnych wariantach (rozszerzając kod z poprzedniej wersji).

- Z 1.1

Klient wysyła, a serwer odbiera datagramy o stałym, niewielkim rozmiarze (rzędu kilkudziesięciu bajtów). Datagramy mogą zawierać ustalony „na sztywno” lub generowany napis – np. „abcde....”, „bcdef...”, itd. Powinno być wysyłanych kilka datagramów, po czym 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 1.3

Na bazie wersji zadania 1.1 napisać w C klienta, który wysyła datagram zawierający strukturę zawierającą kilka liczb oraz napis: np. „struct {long int a; short int b; char c[10];}”. Serwer napisany w Pythonie powinien odebrać i dokonać poprawnego „odpakowania” tej struktury i wydrukowania jej pól. (Można też napisać klienta w Pythonie, a serwer w C)

Wskazówka: wykorzystać moduły Python-a: struct i io.

3. Opis rozwiązania problemu

- Z 1.1
  - Klient:
    - Python – z14_client_11_py:

W przypadku pythonowej implementacji, rozwiązanie jest bardzo proste. Klient tworzy gniazdo, po czym 5 razy wywołuje na nim sendto, wysyłając zakodowany w UTF-8 komunikat „hello from python client” na podany jako argument wywołania adres i port. Na konsoli jest wypisywany komunikat informujący o wysłaniu przez klienta takiego komunikatu.

Klient kończy działanie po wysłaniu komunikatu 5 razy.

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

- C – z14_client_11_c:

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

Klient tworzy gniazdo, po czym parsuje parametry 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, podobnie jak w kliencie python, pięciokrotnie wysyłane są dane na podany jako argument wywołania adres i port.

Logowane są komunikaty o wysyłanej wiadomości, czyli „hello from 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_11_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 gnieździe. Maksymalny rozmiar to 1024 bajty, po otrzymaniu, dekoduje z UTF-8 i wypisuje je na konsole.

Program nie przestaje działać, ciągle czeka na dane. Aby wyłączyć, należy użyć CTRL+C lub „docker kill z14_server_11_py".

- C – z14_server_11_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, odbierane są dane przez recvfrom.

Dane są logowane na konsoli, warto pamiętać o dodaniu '/0' na koniec łańcucha znaków.

Program będzie działał wiecznie, o ile nie zawołamy CTRL+C lub „docker kill z14_server_11_c".

Z 1.3

Klient - z14_client_13_c:

Zadaniem klienta w C jest wysłanie struktury składającej się z różnych, prostych typów danych do serwera w pythonie.

Podobnie jak klienci z 1.1, przyjmuje jako parametr adres i port serwera. Parsowanie argumentów, tworzenie gniazda i uzupełnianie struktury adresowej są identyczne jak dla klienta w 1.1.

Różnica polega na utworzeniu obiektu test_struct, który składa się z jednego pola typu long, jednego typu short i 10 charów. Uzupełniamy te pola testowymi danymi, które prześlemy do serwera, z zaznaczeniem, że typy liczbowe należy jeszcze zmodyfikować funkcjami htons i htonl, aby zapewnić odczytywalność po stronie serwera.

Następnie wysyłamy strukturę przez sendto i logujemy fakt wysłania na konsolę.

- Serwer - z14_server_13_py:

Serwer 1.3 od serwera 1.1 różni się tylko odkodowaniem otrzymanych danych. Jest tworzony w przejrzysty sposób string formatujący dane. Serwer musi być świadom formatu wysyłanych danych, string formatujący musi odwzorowywać strukturę test_struct w kodzie klienta.

Otrzymane odkodowane dane są wypisywane na konsoli.

4. Uwagi dotyczące problemów podczas realizacji

Podczas realizacji nie natrafiliśmy na duże problemy.

Jednym z mniejszych na pewno było umieszczenie adresów i portów w kodzie, bez możliwości zmiany bez modyfikacji kodu. Okazało się to dosyć uciążliwe podczas pracy na bigubu i zostało naprawione.

Innym mniejszym problemem były lakoniczne informacje zwrotne, podczas testowania na bigubu zaistniała potrzeba bardziej szczegółowego opisu, dlatego serwery i klienci wymieniają się danymi zawierającymi język implementacji.

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:

172.21.14.2 – serwer python 1.1

172.21.14.3 – serwer c 1.1

172.21.14.4 – serwer python 1.3

Dodatkowo, skrypt przekazuje argumenty do programu serwera / klienta, więc poprawne wywołanie np. klienta C 1.1 wygląda następująco:

sh run.sh 172.21.14.3 1414

Po wywołaniu, 5 prostych wiadomości tekstowych zostanie wysłanych do serwera C z zadania 1.1 na port 1414 (może być inny efemeryczny, 1414 był używany do testów).

6. Opis testów

- Z 1.1

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

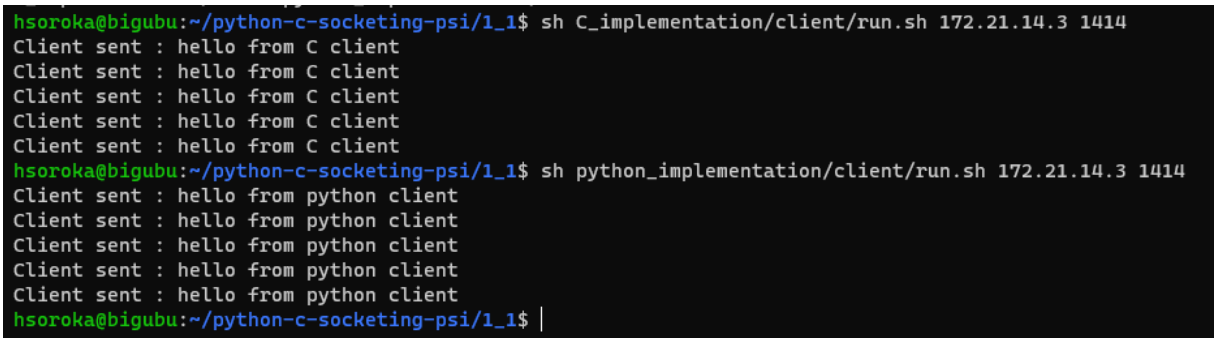
sh run.sh 172.21.14.2 / 172.21.14.3 1414

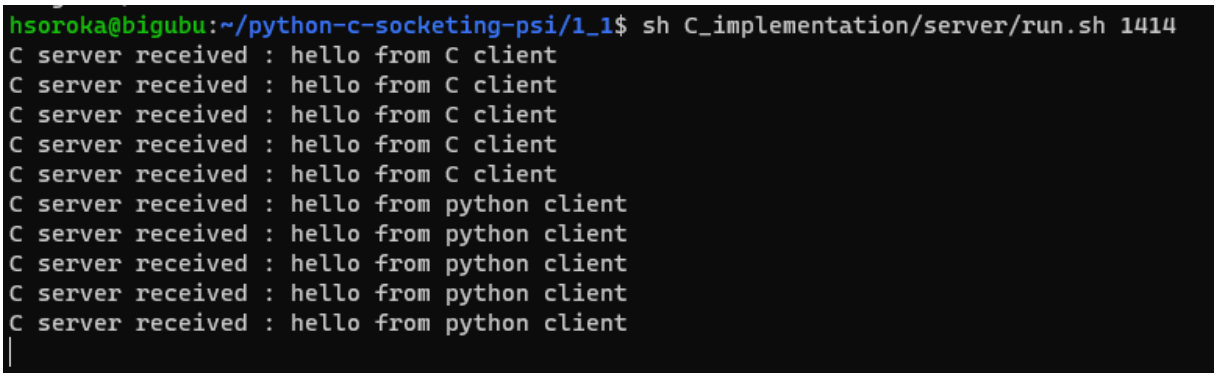
Uruchomienie serwera wymagało użycia poniższych komend w odpowiednim katalogu serwera:

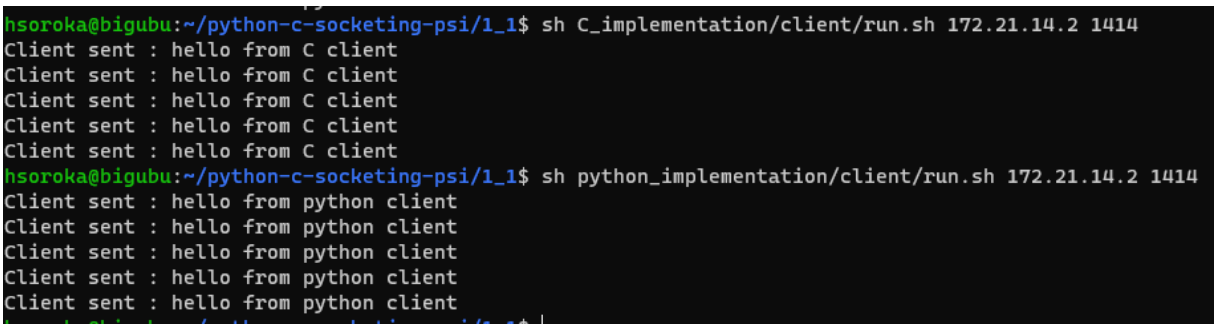
sh build.sh

sh run.sh 1414

- Wyniki testów:







```
hsoroka@bigubu:~/python-c-socketing-psi/1_1$ sh python_implementation/server/run.sh 1414
Python server received : hello from C client
Python server received : hello from C client
Python server received : hello from C client
Python server received : hello from C client
Python server received : hello from C client
Python server received : hello from python client
Python server received : hello from python client
Python server received : hello from python client
Python server received : hello from python client
Python server received : hello from python client
```

Jak widać, oba serwery przyjmują dane od obu klientów.

- Z 1.3

Dla 1.3 sprawdzono, czy dane przesyłane przez klienta w C, są odbierane i poprawnie interpretowane przez serwer w pythonie. Uruchomienie wygląda analogicznie jak w 1.1, z wyjątkiem adresu serwera, który jest równy 172.21.14.4.

- Wyniki testów

```
hsoroka@bigubu:~/python-c-socketing-psi/1_3$ sh C_implementation/client/run.sh 172.21.14.4 1414
Client sent struct with string : Zadanie13
hsoroka@bigubu:~/python-c-socketing-psi/1_3$
```

```
hsoroka@bigubu:~/python-c-socketing-psi/1_3$ sh python_implementation/server/run.sh 1414
Python server received :        long: 123456789, short: 123, string: b'Zadanie13\x00'
```

7. Wnioski i uwagi

Laboratorium na pewno zwiększyło naszą wiedzę nt. programowania z użyciem gniazd, nawet jeśli logika wykonywana przez programy jest nikła.

Możliwość porównania kodu wykonującego to samo w C i w pythonie dobrze pokazuje jak duży jest dodatkowy nakład wymagany do implementacji przesyłania danych w C, nawet tak prosty kod jest około 2 razy dłuższy niż w pythonie.