

SKRYPT DO LABORATORIUM

Systemy internetowe i rozproszone

LABORATORIUM 1:

Praktyczne wykorzystanie architektur systemów przetwarzania rozproszonego (klient-serwer, TCP vs. UDP, wielowątkowość)

Natalia Głowacka

1. Opis ćwiczenia

Wymagania wstępne:

Wymagania w odniesieniu do studenta:

Właściwe przygotowanie się studenta do zajęć pozwoli na osiągnięcie celów ćwiczenia. Przed przystąpieniem do ćwiczenia student powinien:

- powtórzyć wiedzę nabytą w czasie wykładów,
- zapoznać się z instrukcją do ćwiczenia ilustrującą podstawowe zagadnienia z zakresu tematyki ćwiczenia.

Dodatkowo wymagana jest od studenta:

- podstawowa umiejętność programowania,
- podstawowa znajomość języka Python.

Wymagania w odniesieniu do stanowiska laboratoryjnego:

Stanowisko laboratoryjne powinno być wyposażone w komputer z dostępem do sieci komputerowej oraz z następującymi zasobami:

- przykładowe kody programów (załącznik),
- Python 3,
- inne: przeglądarka WWW.

Cele ćwiczenia:

Celem ćwiczenia jest praktyczne przedstawienie wiedzy zdobytej podczas wykładów. Realizacja ćwiczenia laboratoryjnego pozwoli na zdobycie wiedzy w zakresie wykorzystania języków programowania do tworzenia oprogramowania w architekturze klient-serwer – tworzenie aplikacji klienta i serwera w sieci TCP/IP oraz UDP, tworzenie systemów obsługujących wielu klientów poprzez wykorzystanie wątków.

Spodziewane efekty kształcenia - umiejętności i kompetencje:

Po zakończeniu ćwiczenia laboratoryjnego student będzie posiadał umiejętność w zakresie tworzenia systemów o architekturze klient-serwer z wykorzystaniem różnych protokołów komunikacji, a także wykorzystania wątków do obsługi przez serwer wielu klientów.

Metody dydaktyczne:

Na początku student realizuje zadania przykładowe, które prezentują poszczególne etapy tworzenia i wykorzystania oprogramowania w architekturze klient-serwer. W kolejnym kroku realizuje zadania według wytycznych, na podstawie zdobytych umiejętności i wiedzy. Kody źródłowe i wyniki wykonywanych programów student powinien umieszczać w dokumencie elektronicznym, który stanie się sprawozdaniem z ćwiczenia laboratoryjnego.

Materiały wprowadzające i pomocnicze:

- Python 3 - dokumentacja języka <https://docs.python.org/3/>.

Zasady oceniania/warunek zaliczenia ćwiczenia

Każde z realizowanych podczas ćwiczenia laboratoryjnego zadań będzie podlegało ocenie. W trakcie realizacji ćwiczenia, mogą zostać przydzielone dodatkowe zadania do realizacji. Maksymalna liczba punktów do zdobycia wynosi 12.

Wykaz literatury podstawowej do ćwiczenia:

1.	Treści wykładowe do przedmiotu „Systemy internetowe i rozproszone”
2.	Distributed Systems: Principles and Paradigms 2nd Edition, Andrew S. Tanenbaum, Maarten van Steen, ISBN: 978-1530281756
3.	Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies, Ali Sunyaev, Springer, 2020, ISBN: 9783030349561
4.	M. Ben-Ari, "Podstawy programowania współbieżnego i rozproszonego", WNT 2009

2. Przebieg ćwiczenia

L.p.	Zadanie
1.	Zapoznanie się z instrukcją laboratoryjną (przed ćwiczeniem)
2.	Tworzenie programów po stronie klienta (30 min.)
3.	Tworzenie programów świadczących usługi (30 min.)
4.	Wykorzystanie wątków do obsługi wielu klientów (35 min.)
5.	Program klienta i serwera z wykorzystaniem protokołu UDP (30 min.)
6.	Przesłanie sprawozdania z zajęć (10 min.)

3. Wprowadzenie do ćwiczenia

Zadanie 1. Tworzenie programów po stronie klienta

W języku Python do utworzenia socketu (gniazda), wykorzystuje się wbudowany moduł o nazwie *socket*, w którym znajdują się niezbędne definicje i funkcje pozwalające na otwieranie gniazd i komunikację sieciową między nimi.

Do utworzenia gniazda wykorzystuje się funkcję:

- `socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0)` - gdzie family określa typ adresacji (AF_INET – IPv4 (domyślnie)), type – typ przesyłania danych (domyślnie strumieniowo), proto - typ protokołu (0 – TCP (domyślnie), 1 – UDP itd.),

a do nawiązania połączenia z serwerem:

- `socket.connect(address)` - dla IPv4 jest to para (host, port).

Przy próbie nawiązywania połączenia należy zadbać o obsługę wyjątków, ze względu na możliwość braku powodzenia tej operacji (np. brak serwera nasłuchującego na danym porcie, zerwane połączenie).

Podczas komunikacji z serwerem, potrzebna jest realizacja wymiany danych – przesyłania i ich odbierania. Wykorzystując moduł `socket`, realizację wymiany danych można stworzyć z wykorzystaniem metod obiektów `socket`: **`send()`** – do wysyłania i **`recv()`** – do odbierania danych.



Aby stworzyć klienta, należy: stworzyć gniazdo, połączyć się z serwerem i wymieniać dane między klientem a serwerem.

Poniżej zamieszczony został kod programu **`client.py`**, który pozwala na realizację dwukierunkowej wymiany danych między utworzonym klientem oraz wybranym serwerem.

Kod programu **`client.py`**:

```
import socket
import sys

# tworzymy gniazdo (adresacja IPv4, przesyłanie danych - strumieniowo)
try:
    soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error:
    print("Błąd przy tworzeniu gniazda")
    sys.exit()
print("Utworzono socket poprawnie!")

#definicja adresu oraz portu hosta
host_name = "www.pg.edu.pl"
port = 80

# próba utworzenia połączenia
```

```
try:
    soc.connect((host_name, port))
    print("Nawiązano połączenie z %s na porcie %s" %(host_name, port))
except socket.gaierror:
    print("Błąd związany z adresem podczas łączenia z serwerem")
    sys.exit()
except socket.error:
    print("Błąd podczas nawiązywania połączenia")
    sys.exit()

# próba przesłania danych do serwera
try:
    text_to_send = "GET / HTTP/1.1\r\nHost:%s\r\n\r\n" % host_name
    soc.send(text_to_send.encode())
    print("Przesłano do serwera: %s" %(text_to_send))
except socket.error:
    print("Błąd podczas przesyłania danych")
    sys.exit()

# próba odebrania danych przychodzących
try:
    data = soc.recv(4096)
    print("Otrzymano od serwera: %s" %(data.decode('utf-8')))
except socket.error:
    print("Błąd podczas odbierania danych")
    sys.exit()

# zakończenie połączenia
print("Zakończono połączenie")
soc.close()
```

W powyższym przykładzie realizującym program klienta, tworzymy socket, a następnie łączymy się z hostem o podanym adresie i numerze portu. Po nawiązaniu połączenia następuje próba przesłania danych – żądania GET / HTTP/1.1. W kolejnym etapie odbieramy dane przesłane przez serwer.

Przeprowadź proste testy połączenia wykorzystując kod programu **client.py**, którego kod źródłowy podany jest w załączniku. Przeprowadź testy łączenia się z różnymi serwerami. Rozbuduj kod programu o zapisywanie otrzymywanych z serwera danych do pliku tekstowego o wybranej nazwie i lokalizacji.

Zadanie 2. Tworzenie programów świadczących usługi

Tworzenie serwera, który ma świadczyć usługi opiera się na:

- stworzeniu gniazda (socketu),
- przypisaniu gniazda do portu i adresu IP, na którym serwer będzie nasłuchiwał i oczekiwał na połączenia od klientów,
- zaakceptowaniu klienta (accept()),
- wymianie i odbieraniu danych.

Do realizacji połączenia z klientami najczęściej wykorzystuje się pętlę, która je obsługuje oraz obsługuje wystąpienie ewentualnych błędów i niepowodzeń.

Poniżej przedstawiono przykład kodu serwera, obsługujący jednego klienta w danym czasie, przesyłającego do klienta wiadomość „Witaj, tutaj serwer!”.

Kod programu **server.py**:

```
import socket
import sys

# tworzymy gniazdo i otwieramy port 1234, do którego gniazdo zostaje podpięte
try:
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 1234))
    # serwer nasłuchuje danych, które będą na port 1234 docierać (1 równoczesne połączenie do
```

```
obslużenia)
    server_socket.listen(1)
except:
    print("Nie udało się otworzyć gniazda..")
    sys.exit()

while True:
    print("Serwer oczekuje na połączenie..")
    try:
        client_socket, address = server_socket.accept()
        print("Nawiązano połączenie z klientem: %s" %(str(address)))

        # przesłanie wiadomości do klienta
        client_socket.send(bytes("Witaj, tutaj serwer!", 'utf-8'))
        client_socket.close()
        print("Zakończono połączenie z klientem. ")

    except:
        print("Zerwano połączenie z klientem..")

server_socket.close()
```

Przeprowadź testy wykorzystując program **server.py** oraz program **client.py**, który należy zmodyfikować tak, aby mógł odbierać dane z utworzonego i uruchomionego serwera. Następnie zmodyfikuj kod serwera tak, aby przysyłał użytkownikowi wiadomość zawierającą aktualną datę i czas.

Zadanie 3. Wykorzystanie wątków do obsługi wielu klientów

W poprzednich zadaniach rozważaliśmy przykład wymiany danych między klientem a serwerem, jednak serwer obsługiwał w danym momencie tylko jednego klienta. Rzeczywiste serwery mają

możliwość obsługi więcej niż jednego klienta, w szczególności wielu klientów jednocześnie. Do rozwiązania tego zadania wykorzystuje się wątki, jako dobrą metodę i praktykę obsługi takich zdarzeń. Do obsługi każdego połączenia wykorzystywany jest oddzielny wątek.

Kod programu **MuliClientServer.py**:

```
import socket
from _thread import *
import sys
import threading

# tworzymy gniazdo i otwieramy port 2500, do którego gniazdo zostaje podpięte
try:
    multiserver_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    multiserver_socket.bind(('localhost', 2500))

    multiserver_socket.listen(5)
except:
    print("Nie udało się otworzyć gniazda..")
    sys.exit()

disconnect_msg = 'koniec'

def client_handler(conn, addr):
    msg = "Witaj %s na serwerze obsługującym wielu klientów!" %(str(addr))
    conn.send(msg.encode('utf-8'))
    while True:
        data = conn.recv(4096)
        data = data.decode('utf-8')

        if data:
            if data == disconnect_msg:
                break
            else:
                msg_to_send = "Echo: " + data
                conn.send(msg_to_send.encode('utf-8'))
    print("Zakończono połączenie z klientem %s" %(addr))
```

```
conn.close()

while True:
    print("Serwer oczekuje na połączenie..")
    try:
        client_socket, address = multiserwer_socket.accept()

        print("Nawiązano połączenie z klientem: %s" %(str(address)))

        thread = threading.Thread(target=client_handler, args=(client_socket, address))
        thread.start()
        print("Liczba watków: " + str(threading.active_count()-1))

    except:
        print("Zerwano połączenie z klientem..")

multiserwer_socket.close()
```

Kod programu **MuliClient.py**:

```
import socket
import sys

# tworzymy gniazdo (adresacja IPv4, przesyłanie danych - strumieniowo)
try:
    soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error:
    print("Błąd przy tworzeniu gniazda")
    sys.exit()
print("Utworzono socket poprawnie!")

#definicja adresu oraz portu hosta
host_name = "localhost"
port = 2500

# próba utworzenia połączenia
```

```
try:
    soc.connect((host_name, port))
    print("Nawiązano połączenie z %s na porcie %s" %(host_name, port))
except socket.gaierror:
    print("Błąd związany z adresem podczas łączenia z serwerem")
    sys.exit()
except socket.error:
    print("Błąd podczas nawiązywania połączenia")
    sys.exit()

# próba odebrania i przesyłania danych
try:
    data = soc.recv(4096)
    print("Otrzymano od serwera: %s" %(str(data.decode('utf-8'))))

    message_to_server = input("Prześlij komunikat do serwera..")

    while True:
        print("Przesyłam do serwera dane: " + message_to_server)
        soc.send(message_to_server.encode('utf-8'))

        data = soc.recv(4096)
        print("Otrzymano od serwera dane: " + data.decode())

        message_to_server = input("Wpisz kolejną wiadomość do serwera. Jeśli chcesz zakończyć
komunikację wpisz: 'koniec'")
        if message_to_server.lower() == 'koniec':
            break

        soc.send(message_to_server.encode())
        print("Przesłano do serwera: %s" %(message_to_server))

        response = soc.recv(4096)
        if response:
            print("Otrzymano od serwera: %s" % (str(response.decode('utf-8'))))
except socket.error:
```

```
print("Błąd podczas komunikacji z serwerem")
sys.exit()
```

```
# zakończenie połączenia
print("Zakończono połączenie")
soc.close()
```

Powyżej zamieszczony kod programu *MuliClientServer.py*, pozwala na jednoczesną obsługę wielu klientów. Do komunikacji wykorzystano protokół TCP. Program *MultiClient.py* pozwala na wymianę komunikatów między serwerem i klientem tak długo, aż nie wpisemy słowa klucz - "koniec", które pozwoli to połączenie zakończyć. Serwer odsyła klientowi echo wiadomości, którą otrzymał.

Przeanalizuj kod programu *MuliClientServer.py* i uruchom usługę. Uruchom wielu klientów z wykorzystaniem programu *MuliClient.py* i wymieniaj komunikaty z serwerem. Sprawdź jakie porty są otwierane dla kolejnych połączeń z klientami.

Zmodyfikuj program *MuliClient.py* tak, aby mierzyć czas między wysłaniem wiadomości do serwera a odebraniem odpowiedzi od niego.

Zadanie 4. Program klienta i serwera z wykorzystaniem protokołu UDP

Zazwyczaj w komunikacji między klientem a serwerem wykorzystuje się protokół TCP. Możliwa jest również realizacja tej komunikacji z wykorzystaniem protokołu UDP. Gniazdo wykorzystujące ten protokół oferuje dwukierunkowy przepływ informacji, jednak nie zapewnia niezawodności przesłania danych, jak w przypadku gniazd TCP obsługujących strumienie.

Gniazda UDP cechuje beipołączeniowość oraz symetryczna wymiana wiadomości. W tym przypadku klient i serwer będą wymieniali datagramy.

Przy tworzeniu serwera nie wykorzystuje się metod *listen()* oraz *accept()* - taki serwer nie nasłuchuje i nie akceptuje połączeń.

Przesyłanie pakietów realizuje się poprzez wykorzystywanie metod *sendto(data, address)*, a odbieranie poprzez metodę *recvfrom(buffer [, flags])*.

Poniżej zamieszczono kod prostego serwera oraz klienta, wykorzystujące protokół UDP.

Kod programu **clientUDP.py**:

```
import socket
import sys

# tworzymy gniazdo (adresacja IPv4, przesyłanie danych - datagramy)
try:
    udp_client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
except socket.error:
    print("Błąd przy tworzeniu gniazda")
    sys.exit()
print("Utworzono socket poprawnie!")

#definicja adresu oraz portu hosta
udp_server_host = 'localhost'
udp_server_port = 2000

# wiadomość do przesłania do serwera
message_to_server = "Witaj, tutaj klient!"

try:
    print("Przesyłam do serwera dane: " + message_to_server)
    udp_client_socket.sendto(message_to_server.encode('utf-8'), (udp_server_host,
udp_server_port))

    data, address = udp_client_socket.recvfrom(16)
    print("Otrzymano od serwera dane: " + data.decode())

except socket.error:
    print("Komunikacja z serwerem się nie powiodła..")

udp_client_socket.close()
```

Kod programu **serverUDP.py**:

```
import socket
import sys
```

```
# tworzymy gniazdo i otwieramy port 2000, do którego gniazdo zostaje podpięte
try:
    udp_server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_server_socket.bind(('localhost', 2000))
except:
    print("Nie udało się otworzyć gniazda..")
    sys.exit()

while True:
    print("Serwer oczekuje na połączenie..")

    try:
        data, address = udp_server_socket.recvfrom(4096)
        print("Otrzymano od klienta dane: ")
        print(str(data.decode()))

        message_to_send = "Witam, tutaj serwer!"
        print("Wysyłam do klienta następujące dane: " + message_to_send)

        udp_server_socket.sendto(message_to_send.encode('utf-8'), address)
    except socket.error:
        print("Komunikacja z klientem się nie powiodła..")

udp_server_socket.close()
```

Uruchom i przeanalizuj działanie zamieszczonych powyżej kodów programów. Wykorzystując programy clientUDP.py oraz serverUDP.py, rozbuduj kod klienta o wpisywanie danych do przesłania do serwera przez użytkownika. Serwer niech przesyła do klienta różne (losowe) odpowiedzi.

4. Forma i zawartość sprawozdania

Sprawozdanie powinno zawierać kopie ekranu stworzonych kodów i wyników ich działania dla zadań, które w instrukcji zostały oznaczone kolorem zielonym oraz stosowne komentarze, jeśli zadanie tego

wymaga. Dokument powinien zostać przesłany na serwer wskazany przez prowadzącego ćwiczenie w formie PDF.

Dodatki

Załącznik 1: Kod programu **client.py**

Załącznik 2: Kod programu **server.py**

Załącznik 3: Kod programu **MuliClientServer.py**

Załącznik 4: Kod programu **MuliClient.py**

Załącznik 5: Kod programu **clientUDP.py**

Załącznik 6: Kod programu **serverUDP.py**