Socket Programmierung in Python

Suman Kafle, Beruk

14.06.2024

Lab-Anleitung

Bevor Sie mit dem Labor beginnen, stellen Sie sicher, dass Sie die folgenden Schritte ausgeführt haben:

1. Klonen Sie das Git-Repository mit dem Laborcode auf Ihren lokalen Computer. `https://github.com/Master-of-Communication/KNJ--Socket-Programming.git`
2. Stellen Sie sicher, dass Sie Python auf Ihrem Computer installiert haben. Dieses Labor erfordert Python-Kenntnisse.
3. Lesen Sie im README.md Datei für Python Socket API Module.

1 Entwicklung eines Webservers in Python

Aufgabenstellung

In dieser Übung werden Sie die Grundlagen der Socket-Programmierung für TCP-Verbindungen in Python kennenlernen. Ihr Ziel ist es, einen Webserver zu erstellen, der HTTP-Anfragen empfängt, verarbeitet und entsprechende Antworten zurücksendet. Diese Aufgabe umfasst das Parsen von HTTP-Anfragen, das Lesen von Dateien aus dem Dateisystem und das Erstellen von HTTP-Antwortnachrichten.

Zielsetzung

Entwickeln Sie einen Webserver, der folgende Aufgaben erfüllt:

1. Empfang und Parsing von HTTP-Anfragen.
2. Abrufen der angeforderten Datei aus dem Dateisystem des Servers.
3. Erstellen und Zurücksenden einer HTTP-Antwortnachricht, die den Inhalt der angeforderten Datei oder eine Fehlermeldung enthält, falls die Datei nicht existiert.

Anforderungen

1. Verarbeitung der HTTP-Anfrage:
    - Der Server soll HTTP-Anfragen über einen bestimmten Port empfangen und die Anfrage parsen, um den angeforderten Dateinamen zu ermitteln.

2. Dateiabruf und Antworterstellung:
    - Der Server soll die angeforderte Datei aus dem Dateisystem lesen.
    - Wenn die Datei vorhanden ist, soll der Server eine HTTP 200 OK-Antwort zusammen mit dem Dateiinhalten zurücksenden.
    - Wenn die Datei nicht vorhanden ist, soll der Server eine HTTP 404 Not Found-Antwort zurücksenden.
3. Fehlerbehandlung:
    - Der Server soll in der Lage sein, Fehler zu erkennen und entsprechend zu reagieren, beispielsweise bei ungültigen Anfragen oder nicht vorhandenen Dateien.

Code

Unten finden Sie den Skeleton-Code für den Webserver. Sie sollen den Skeleton-Code vervollständigen. Die Stellen, an denen Sie Code einfügen müssen, sind mit `#Your code comes here` und `#Your code ends here` markiert. Jede Stelle kann eine oder mehrere Zeilen Code erfordern. Der Skeleton-Code ist verfügbar unter folgendem Link:
https://github.com/Master-of-Communication/KNJ--Socket-Programming/blob/main/assigments/http_server/http_server.py

```
#import socket module
from socket import *
server_socket = socket(AF_INET, SOCK_STREAM)
'''
Prepare a server socket, e.g address and port then bind the server socket
    to the address and port dont forgent to listen for incoming connections
'''

HOST = '127.0.0.1'
# Your code comes here
# port number
#associate the server port number with this socket
#wait and listen for some client to knock on the door, the number of
    clients that can wait is N>0
# Your code ends here

while True:
    #Establish the connection
    print('Server is running.....')

    #accept the connection
    connection_socket, addr =  # Your code comes here
    print("Request accepted from (address, port) tuple: %s \n" % (addr,))

    try:
        message = connection_socket.recv(1024) #receive the message from
    the client
        print('the message is : \n', message)
        filename = message.split()[1]
        print('the filename is: ', filename)
        f = open(filename[1:])
        outputdata = f.read() #read the file
        #Send one HTTP header line into socket
        connection_socket.send('HTTP/1.1 200 OK\n\n'.encode())
        #Send the content of the requested file to the client
        print('the length is \n\n', len(outputdata))

        for i in range(0, len(outputdata)):
            connection_socket.send(outputdata[i].encode())
        connection_socket.send("\r\n".encode())
        connection_socket.close()
    except IOError:
        #Send response message for file not found
        connection_socket.send('404 Not Found'.encode())
        #Close client socket
        # Your code comes here

    #Close server  socket
    # Your code comes here

#Terminate the program after sending the corresponding data
```

Listing 1: Skeleton-Code für den Http Webserver

Hinweise

- Legen Sie eine HTML-Datei (z.B. `helloworld.html`) in dasselbe Verzeichnis, in dem sich der Server befindet.
- Navigieren Sie zu dem Verzeichnis, in dem sich Ihre Python-Datei befindet um Server zu starten.

```
cd pfad/zu/ihrem/verzeichnis
python ihr_script.py oder
python3 ihr_script.py
```

- Bestimmen Sie die IP-Adresse des Hosts, der den Server ausführt (z.B. `127.0.1.1`). Öffnen Sie einen Browser und geben Sie die entsprechende URL ein. Zum Beispiel:

  `http://127.0.1.1:7001/helloworld.html`

  `helloworld.html` ist der Name der Datei, die Sie im Serververzeichnis abgelegt haben. Beachten Sie auch die Verwendung der Portnummer nach dem Doppelpunkt. Sie müssen diese Portnummer durch diejenige ersetzen, die Sie im Servercode verwendet haben. Im obigen Beispiel haben wir die Portnummer `7001` verwendet. Der Browser sollte dann den Inhalt von `HelloWorld.html` anzeigen. Wenn Sie `:7001` weglassen, nimmt der Browser Port 80 an und Sie erhalten die Webseite vom Server nur, wenn Ihr Server auf Port 80 hört.
- Versuchen Sie dann, eine Datei abzurufen, die nicht auf dem Server vorhanden ist. Sie sollten eine „404 Not Found“-Nachricht erhalten.
- Sehen Sie im README.md für Python Socket API Module.

Einzureichende Materialien

- Den vollständigen, gut dokumentierten Quellcode Ihres Webservers.
- Screenshot

2 Entwicklung eines Multi-Connection Servers mit mehreren Clients

Aufgabenstellung

In dieser Aufgabe testen Sie einen Multi-Connection Chat Server und Client in Python. Der Server kann mehrere Clients gleichzeitig bedienen, und der Client ermöglicht es Benutzern, sich mit dem Server zu verbinden, Nachrichten zu senden und zu empfangen.

Voraussetzungen

Ein Rechner mit installiertem Python (mindestens Version 3.6).
Ein Terminal oder eine Konsole zum Ausführen der Python-Skripte. (siehe Vorbereitung.)

Dateien

`server.py`: Die Server-Seite des Chat-Systems.
`client.py`: Die Client-Seite des Chat-Systems.

Zielsetzung

Ausführen der vorliegenden Quellcoden.
Verstehen der Quellcoden.
Kommentieren der Quellcoden.
Verstehen der Zusammenhang mit der Socket-Programmierung

- Der Server zeigt an, dass er gestartet wurde und auf Verbindungen wartet.
- Mehrere Clients verbinden sich mit dem Server, senden Nachrichten und erhalten die Nachrichten zurück.
- Der Server verarbeitet jede Verbindung in einem separaten Thread, sodass mehrere Clients gleichzeitig bedient werden können.

Durchführung

1. Ausführen der `server.py`
2. Ausführen der `client.py` (Mehrfach in unterschiedlichen Terminals ausführen)
3. Fehlerbehandlung beobachten:
    - Der Server/Client soll in der Lage sein, Fehler zu erkennen und entsprechend zu reagieren.

Hilfe

```
$cd pfad/zu/ihrem/aufgabe 2 verzeichnis
$python ihr_script_server.py oder
$python3 ihr_script_server.py
$python ihr_script_client.py oder
$python3 ihr_script_client.py
```

Einzureichende Materialien

- server.py: Ausführlicher kommentierter Quellcode
- client.py: Ausführlicher kommentierter Quellcode
- Screenshot

3 UDP Pinger

Einführung

In diesem Labor werden Sie die Grundlagen der Socket-Programmierung für UDP in Python kennenlernen. Sie werden lernen, wie man Datagram-Pakete mit UDP-Sockets sendet und empfängt und wie man eine ordnungsgemäße Socket-Timeout-Einstellung vornimmt. Im Verlauf des Labors werden Sie mit einer Ping-Anwendung vertraut gemacht und deren Nützlichkeit zur Berechnung von Statistiken wie der Paketverlustquote verstehen. Der Skeleton-Code ist verfügbar unter folgendem Link: `https://github.com/Master-of-Communication/KNJ--Socket-Programming/tree/main/assigments/udppinger`

Server Code

Der vollständige Code für den Ping-Server ist im udppingerserver im git angegeben. Sie müssen diesen Code kompilieren und ausführen.

In Servercode werden 30% der Pakete des Clients als verloren simuliert. Der Server befindet sich in einer Endlosschleife und wartet auf eingehende UDP-Pakete. Wenn ein Paket eingeht und eine randomisierte Ganzzahl größer oder gleich 4 ist, kapitalisiert der Server einfach die enthaltenen Daten und sendet sie an den Client zurück.

Paketverlust

UDP bietet Anwendungen einen unzuverlässigen Transportdienst. Nachrichten können im Netzwerk aufgrund von Router-Überlauf, fehlerhafter Hardware oder aus anderen Gründen verloren gehen. Der Server erstellt eine variable randomisierte Ganzzahl, die bestimmt, ob ein bestimmtes eingehendes Paket verloren geht oder nicht.

Client-Code

In das gegebene `udppingerclient.py` muss folgendes Client-Programm implementiert werden.
Der Client sollte 10 Pings an den Server senden. Da UDP ein unzuverlässiges Protokoll ist, kann ein vom Client an den Server gesendetes Paket im Netzwerk verloren gehen oder umgekehrt. Aus diesem Grund kann der Client nicht unbegrenzt auf eine Antwort auf eine Ping-Nachricht warten. Der Client sollte bis zu eine Sekunde auf eine Antwort warten; wenn innerhalb einer Sekunde keine Antwort empfangen wird, sollte Ihr Client-Programm davon ausgehen, dass das Paket während der Übertragung im Netzwerk verloren ging.

Insbesondere sollte Ihr Client-Programm:

1. die Ping-Nachricht mit UDP senden (Hinweis: Im Gegensatz zu TCP müssen Sie keine Verbindung herstellen, da UDP ein verbindungsloses Protokoll ist.)
2. die Antwortnachricht des Servers, falls vorhanden, ausgeben
3. die Round-Trip-Zeit (RTT) in Sekunden für jedes Paket berechnen und ausgeben, falls der Server antwortet
4. andernfalls "Request timed outäusgeben

```
from socket import *
from datetime import datetime
import time

# Create a client UDP socket
# Your code comes inside the parenthesis
clientSocket = socket( ) # See readme for the correct parameters in our git

# Server details
serverIP =   #Your code comes, see server adress in server output
serverPort =  #Your code comes, see server port number in server output

rtts = []
lost = 0

# Send ? pings to server, see question 3
for i in range(0, ): # Your code comes inside the parenthesis

    # Ping message to be sent
    time_string = datetime.now().isoformat(sep=' ')

    #msg to server
    message = "Ping " + str(i+1) + " " + time_string
    print(f"\n{message}")
    message = message.encode()

    # Start time to calculate RTT
    start = time.process_time()

    # Send the message
    clientSocket.sendto(message.encode(), (serverIP, serverPort))

    try:
        # Set timeout for any response from the Ping server
        # Set timeout to ... second according to question
        clientSocket.settimeout( ) # Your code comes inside the parenthesis

        # your code comes here
        response = # receive the message from the server
        # your code ends here

        # End time to calculate RTT
        end = time.process_time()

        # Your code comes here,
        rtt =   # calculate the round trip time, end time - start time
        # Your code ends here

        rtts.append(rtt)

        # Remove timeout
        print(f"\t{response[0].decode()}") # message from server to client
        print(f"\tCalculated Round Trip Time = {rtt:.6f} seconds")
        clientSocket.settimeout(None)

    except timeout:
        # Packet has been lost
        lost += 1
```

```
        print("\tRequest timed out")
    # Close the socket
clientSocket.close()

# Print report just for fun
print("\nRTT Report:")
if rtts:
    print(f"Maximum RTT = {max(rtts):.6f} seconds")
    print(f"Minimum RTT = {min(rtts):.6f} seconds")
    print(f"Average RTT = {sum(rtts) / len(rtts):.6f} seconds")
    print(f"Packet Loss Percentage = {lost / 10 * 100:.2f}%")
```

Listing 2: Skeleton-Code für den UDP Pinger Client