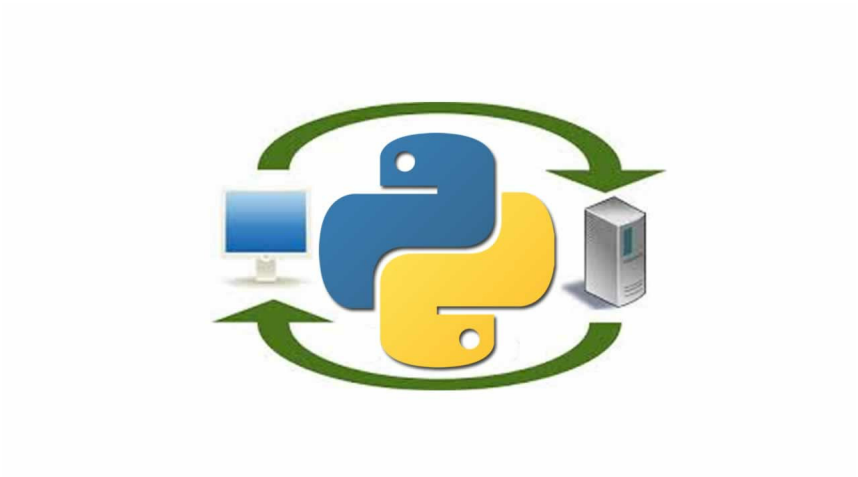
Netzwerkkommunikation in FSST



Grundlagen der Netzwerkkommunikation mit Schwerpunkt auf Sockets in Python und C.

Basierend auf dem *Openbook Python 3* von Johannes Ernesti und Peter Kaiser, erschienen im Rheinwerk Verlag, sowie diversen online Tutorials.

Version 0.1
Mai 2021

Typesetting by LaTeX

Inhaltsverzeichnis

1 Netzwerkkommunikation auf der Commandline - netcat

Netcat oder auch **nc** genannt ist das *swiss army knife* der Netzwerkkommunikation. Ursprünglich für Unix entwickelt (1999), ist es inzwischen plattformübergreifend verfügbar.
Für Unix, Linux und MacOS ist es meist schon vorinstalliert, für Windows muss es eventuell nachinstalliert werden.
Da **netcat** schon etwas in die Jahre gekommen ist, hat das Team hinter *www.nmap.org* eine neue Implementierung geschrieben: **Ncat**.
Downloadlink für Windows: `https://nmap.org/ncat/`
Installation unter Linux: `sudo apt install ncat`
Die grundsätzlich Syntax ist aber für beide Tools gleich!

1.1 Anwendungsbeispiele für netcat

Die Syntax ist recht einfach. Mit der Option `-l` für listen erzeugt man eine Server Instanz von netcat. (Die Option `-v` steht für verbose und gibt mehr Informativen output aus)
z.B. `netcat -v -l 61111`
Erzeugt eine Serverinstanz auf der lokalen Maschine und hört auf Port 61111.
Um sich darauf von einer anderen Maschine zu verbinden muss man die IP der Server Instanz angeben:
z.B. `netcat -v 10.0.0.99 61111`

1.1.1 Einfacher Netzwerk Chat

Beispiel: Erstellen einer Server Instanz von netcat auf einem Raspberry Pi mit der IP 10.0.0.99

```
markus@raspberrypi:~ $ ifconfig | head -4
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.99  netmask 255.0.0.0  broadcast 10.255.255.255
        inet6 fe80::dea6:32ff:fec6:85d0  prefixlen 64  scopeid 0x20<link>
        ether dc:a6:32:c6:85:d0  txqueuelen 1000  (Ethernet)
markus@raspberrypi:~ $ netcat -v -l 61111
Listening on [0.0.0.0] (family 2, port 61111)
```

Erstellen einer Client Instanz von netcat auf einem PC mit der IP 10.0.0.66

```
markus@pc:~$ ifconfig | head -4
enp6s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.66  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 fe80::76a8:c076:9189:ab66  prefixlen 64  scopeid 0x20<link>
        ether 2c:f0:5d:56:44:3b  txqueuelen 1000  (Ethernet)
markus@pc:~$ netcat -v 10.0.0.99 61111
Connection to 10.0.0.99 61111 port [tcp/*] succeeded!
```

Nun kann einfach in die Eingabeaufforderung geschrieben werden und am anderen Ende wird es über *stdout* angezeigt!

```
markus@pc:~$ netcat -v 10.0.0.99 61111
Connection to 10.0.0.99 61111 port [tcp/*] succeeded!
Hello Raspi
Hello PC :)
How are you? (... noch nicht mit ENTER bestätigt)


markus@raspberrypi:~ $ netcat -v -l 61111
Listening on [0.0.0.0] (family 2, port 61111)
Connection from 10.0.0.66 59104 received!
Hello Raspi
Hello PC :)
```

1.1.2 Einfaches Kopieren von Files mittels netcat

Mit *re-directs* lassen sich auch einfach Files kopieren - es wird einfach die *stdin* und *stdout* auf files umgeleitet! Beispiel:

```
#Server Instanz - soll ein file empfangen!
markus@raspberrypi:~ $ netcat -v -l 61111 > reveice.file

#Client Instanz - soll ein file zum server schicken
markus@pc:~$ netcat -v 10.0.0.99 61111 < send.file
```

1.1.3 Kopieren von mehreren Files mittels netcat

Mit Hilfe von pipes lassen sich auf der Command-Line auch mehrere Files gleichzeitig kopieren - man zipt sie einfach zuerst mittels `tar` und `gzip`!

```
#Server Instanz - soll files empfangen und entpackt den tar-ball!
markus@raspberrypi:~ $ netcat -v -l 61111 | gunzip | tar vx

#Client Instanz - soll alle files im lokalen folder zippen und zum server schicken
markus@pc:~$ tar vc * | gzip | netcat -v 10.0.0.99 61111
```

2 Sockets mit Python

Mit einem *Socket* meint man in der Netzwerktechnik die Kombination aus einer IP-Adresse und einem Port, an welchem ein Service angeboten wird.
z.B. *192.168.10.7:22* wäre ein Socket.
In der Programmierung meint man mit Socket ein Objekt in das man ähnlich wie in ein File lesen oder schreiben kann, nur wird die Information nicht auf der lokalen Festplatte gespeichert, sondern über das Netzwerk übertragen.
Es gibt in Python eine Vielzahl von Sockets, wir werden uns aber auf `INET` (IPv4) und `STREAM` (z.B. TCP) Sockets konzentrieren.

2.1 Server- VS Clientsocket

Ein **Clientsocket** ist ein klassischer *end-point* einer Netzwerkkommunikation.
Ein **Serversocket** ist mehr ein Vermittelungsservice auf einem Server, der auf eingehende Client-Verbindungen wartet, und dann für jede Verbindung mit einem Client einen eigenen Clientsocket am Server erstellt. Diser kommuniziert dann (paarweise)mit dem Clientsocket am Client.
Wenn nun mehrer Clients gleichzeitig mit einem Server kommunizieren wollen, muss der Server multitaskfähig sein - was uns zu Multithreading und Multiprocessing führt.

2.2 Socket erzeugen

2.2.1 Client-Socket

Man erzeugt ein Socket-Objekt und verbindet es dann mit einem Server.

```
# INET, STREAMing socket erzeugen
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# nun mit einem Server (IP oder URL + Portnummer) verbinden
s.connect(("www.MyServer.org", 61111))
```

Wenn die Verbindung steht kann man über den Socket etwas senden und das gleiche Socketobjekt wird auch die Antwort lesen. Überlicherweise wird der Socket danach **automatisch zerstört!**. Also standardmäßig hat man mit einem Socketobjekt nur einen einzigen Datenaustausch!
Mögliche weitere Optionen bei der Erstellung des Socket-Objekts:

Adressfamilie	Socket-Typ	Verwendungszweck
AF_INET	SOCK_STREAM	IPv4 + TCP (z. B. HTTP, Chat)
AF_INET	SOCK_DGRAM	IPv4 + UDP (z. B. DNS, Video)
AF_INET6	SOCK_STREAM	IPv6 + TCP (z. B. moderne Webdienste)
AF_INET6	SOCK_DGRAM	IPv6 + UDP (z. B. Multicast)
AF_INET	SOCK_RAW	Rohdaten (z. B. Protokollanalyse)

2.2.2 Server-Socket

Am Server ist der Server-Socket so was wie ein Vermittlungsdienst, der auf eingehende Verbindungen wartet:

```
# INET, STREAMing socket erzeugen - gleich wie am Client!
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# bind - der socket wird jetzt zu einer IP/URL und einem port gebunden
serversocket.bind((socket.gethostname(), 61111))
#  ODER
serversocket.bind(("127.0.0.1", 61111))
# ODER nimm einfach die IP der Maschine
serversocket.bind(("", 61111))

# und nun warte auf eingehende Verbindungen
serversocket.listen(5)
# der Paramter 5 bedeutet, das bis zu 5 eingehende Verbindungen
# abgearbeitet werden (in einer Warteschlange) bevor Verbindungen
# abgelehnt (refused) werden
```

Nun haben wir also ein *horchendes* Server-Socket-Objekt und können in einem Loop auf eingehende Verbindungen warten.

```
while True:
    # eingehende Verbindungen annehmen
    (clientsocket, address) = serversocket.accept()
    # nun was mit dem clientsocket machen z.B: einen eigenen Prozess
    # starten, der dann die Kommunikation mit dem Client abwickelt
    p = multiprocessing.Process(target=meine_funktion, arg=(clientsocket,))
    p.start()
```

Nachdem der neue Prozess angelegt und gestartet wurde kann der Server wieder auf eingehende Verbindungen warten. Die eigentliche Kommunikation läuft nun zwischen den zwei Client-Sockets in einem eigenen Prozess ab.

3 Python-Socket-Beispiele

3.1 Echo-Server

Ein einfacher Server, der auf Clients wartet und eingehende Nachrichten einfach wieder zum Client zurück schickt!

3.1.0.1 Echo-Server-Code

```
'''
einfacher echo server
erste Uebung fuer Sockets in Python

Mai 2021
Markus
'''

import socket

HOST = "127.0.0.1"
PORT = 61111

#IPv4 TCP Socket erzeugen und auf Clients warten (lauschen)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
#bis zu 5 clients werden im backlog gehalten!
s.listen(5)


while True:

    conn, addr = s.accept()
    print("Connected by", addr)

    while True:
        #.recv mit verschiedenen Byte-Laengen ausprobieren!
        data = conn.recv(1024)
        if not data:
           break
        #man kann auch .decode() verwenden, statt repr
        print("Received from client", repr(data))
        #.sendall ist komfortabler als .send(byte-laenge)
        conn.sendall(data)
    conn.close()
```

3.1.0.2 Echo-Client-Code

```
import socket

HOST = "127.0.0.1"
PORT = 61111
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
print("welcome to the echo server, please type what you want and 'end' to quit")
while True:
    data = input()
    if (data == "end"):
        break
    s.sendall(data.encode())
    data = s.recv(1024)
    print("Received:", data.decode())

print("cleaning up ...")
s.close()
```

3.2 Echo Server Multi-Threading

```
'''
Echo server mit multi threading - ist in der Lage
mit mehr als einem Client gleichzeitig zu sprechen

zweite Uebung fuer Sockets in Python

Mai 2021
Markus
'''

import socket
import threading

#echo function - bekommt die Adresse und den Client-Socket
def echo(conn, addr):
    print('Connected by', addr)

    while True:
        data = conn.recv(1024)
        if not data:
           break
        print('Received from client', repr(data))
        conn.sendall(data)
    conn.close()



HOST = '127.0.0.1'
PORT = 61111

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()

while True:
    conn, addr = s.accept()
    echo_thread = threading.Thread(target=echo, args=(conn, addr))
    echo_thread.start()
```