Kommunikation im Netzwerk

Die Client-Server Beziehung

Der Client stellt eine Frage (**Request**) an den Server, dieser sendet eine Antworte (**Response**) zurück. Grundlage ist die Verfügbarkeit von **Sockets**, die eine Kommunikation ähnlich dem Datenaustausch mit einer Diskette erlauben: man schreibt auf den Socket oder liest von ihm, als ob es sich um eine einfache Text-Datei handeln würde. Die tatsächliche Umsetzung nehmen Bibliotheken des Betriebssystems vor. Windows beherrscht diesen Mechanismus seit Win3.11, Unix kennt ihn schon seit der Entwicklung des BSD-Dialektes (wurde an der Universität Berkeley erfunden).

Die Beziehung zwischen den Computern wird durch 2 Kennungen festgelegt:
1.) die **IP-Adresse** (oder der Name) des Computers
2.) eine gemeinsame **Portnummer,** da ein Computer mehrere Verbindungen unterhalten kann. Oft spricht man einfach nur von 'Port'. 'Der Server wartet auf dem Port auf Verbindungen'.

Ein Vergleich aus dem täglichen Leben: die IP-Adresse entspricht der Hausnummer eines Bürogebäudes, der Port ist die Türnummer einer speziellen Abteilung. Ein Gebäude kann viele einzelne Dienstleistungen (Abteilungen) zur Verfügung stellen, eine Abteilung kann auch mehrere Arbeitsaufträge annehmen.

Die Port-Nummer ist eine beliebige 16-bit Zahl, die beiden Teilnehmern bekannt sein muss. Die Werte bis 1024 sind für besondere Dienste reserviert, alle übrigen dürfen prinzipiell verwendet werden.

Beispiele für Adressen und Ports:
("www.hib-wien.ac.at",80) verbindet mit dem http-Server der Schulhomepage
("127.0.0.1",21) verbindet mit dem FTP-Kontrollkanal des lokalen Gerätes
("mail.myserver",25) für den SMTP Mailversand

Bei Sockets unterscheiden wir die
1.) Familie: AF_UNIX für Prozesskommunikation innerhalb des Computers und AF_INET für Kommunikation nach außen
2.) Typ: SOCK_STREAM für 'Streaming Sockets' des TCP und SOCK_DGRAM für 'Datagram Sockets'

Diese zwei Typen möchte ich anhand der beiden wichtigsten Protokolle erläutern:
1.) **TCP (Transmission Control Protocol)**: (SOCK_STREAM) hier wird eine dauerhafte Verbindung zwischen zwei Computern hergestellt. Der Datenverkehr zwischen den Partnern ist fest gebunden. Beispiel: FTP, HTTP. Typischerweise wartet der Server auf Aufträge durch den Client.
2.) **UDP (User Datagram Protocol):** (SOCK_DGRAM) hier wird eine 'lose' Verbindung geschaffen. Es werden einzelne Datenpakete zwischen den Computern ausgetauscht, deren korrekte Übermittlung jedoch nicht überprüft wird. Es können also Datenpakete 'verloren gehen', doppelt beim Empfänger ankommen oder in falscher Reihenfolge eintreffen. Praktische Anwendung findet dieses Protokoll etwa bei der Multimediapräsentation im Internet: gehen bei der Übertragung eines Films ein paar Frames verloren, ist dies leicht zu verschmerzen

('Streaming Media' wie RealMovie,..).

Erzeugung und Benutzung eines Socket-Objekts

Dies ist mit dem Modul socket ganz einfach:
Wir erzeugen ein Socket-Objekt und verbinden es mit einem Adress-Tupel (IP-Adresse,Port)

```
import socket
mysocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Nun würde mysocket eine TCP-Verbindung ermöglichen, und

```
mysocket.connect("www.google.com",80)
```

bereitet die Kommunikation mit dem WWW-Server der Suchmaschine Google vor.

Damit steht nun eine Verbindung zu einem Partnercomputer, die nun nach Belieben benutzt werden kann:

```
mysocket.send("GET /\r\n")
mysocket.recv(1000)
```

fordert 1000 Byte der Startseite von Google an und liefert sie zurück. Es handelt sich dabei klarerweise um den HTML-Quellcode der Seite. Dass die Methode receive als 'recv' abgekürzt wird ist keine Python-Spezialität, die vierbuchstabigen Abkürzungen haben sich allgemein eingebürgert (Programmierer sind tippfaul).

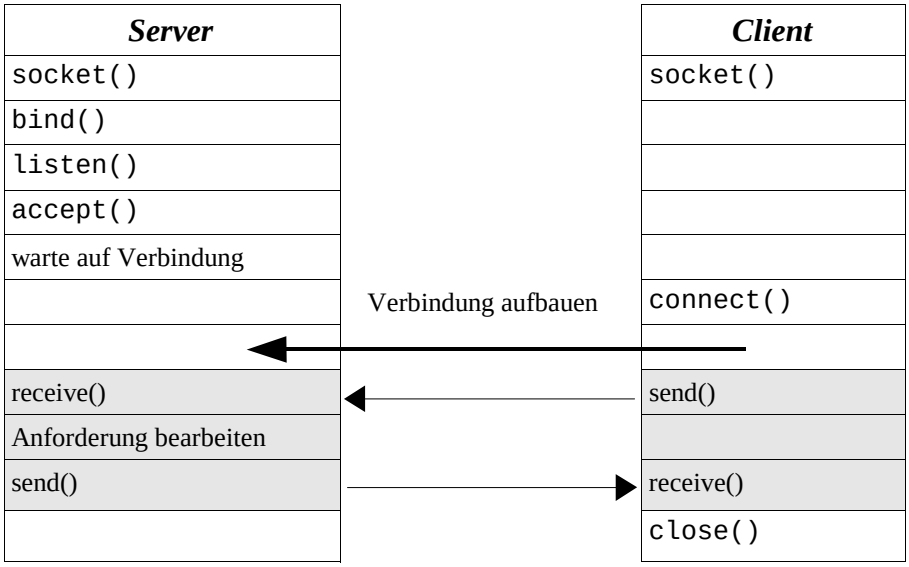
Beispiel: **Portscanner**
Wir wollen untersuchen, ob ein gewisser Port auf einem Computer offen (empfangsbereit) oder geschlossen ist. Dies ist etwa beim Untersuchen eines Wurm-infizierten Rechners praktisch.

```
# Beispiel: scan("www.schule.at",80)

from socket import *

def scan(host,port):
    s = socket(AF_INET,SOCK_STREAM)
    try:
        s.connect((host,port))
        print "Port",port,"ist offen"
    except:
        print "Port",port,"ist zu"
```

Kommunikation

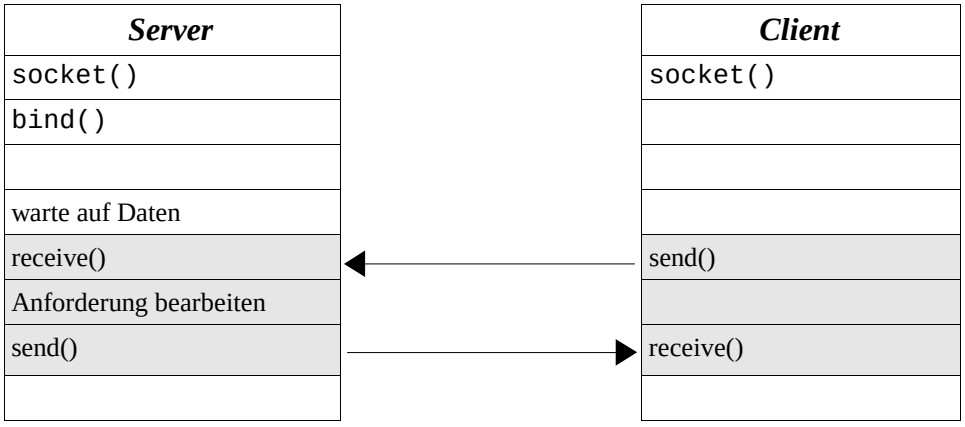
Hier müssen wir zwei Unterscheidungen treffen:
1.)Welches Protokoll benutzen wir – TCP oder UDP
2.)Welche Rolle spielt dieses Programm – Client oder Server

TCP – Kommunikation



Bemerkung: Das Socket-Objekt, das der Server zum Empfang einer konkreten Verbindung verwendet, ist nicht der Socket, mit dem der Server startet. Die accept-Methode erzeugt einen neuen Socket (mit einer neuen internen Port-Nummer), der für die folgende Kommunikation mit diesem speziellen Client verwendet wird.

UDP – Kommunikation



TCP Server/Client

Der Server soll die von den Clients gesendeten Strings alphabetisch sortiert zurücksenden.

Das Server-Programm

```
from socket import *

myHostName = ''                                  localhost
myPort = 5007

def answer(s):                                   als Antwort die sortierte Eingabe
    l = list(s.upper())
    l.sort()
    return "".join(l)

mainsocket = socket(AF_INET, SOCK_STREAM)        Hauptsocket
mainsocket.bind((myHostName, myPort))            binden
mainsocket.listen(1)                             max. eine Verbindung gleichzeitig

while 1:                                         die Hauptschleife
    connection,address = mainsocket.accept()     Socket für nächsten Client
    print "Server connected by", address         Wer ist hier?
    while 1:
        data = connection.recv(1024)             max. 1024 Byte lesen
        if not data: break                       wenn keine Daten mehr
        print "received:",data                   Request anzeigen
        connection.send(answer(data))            sortierten String zurücksenden
    connection.close()                           Verbindung schließen
```

Das Client-Programm

```
from socket import *

serverHostName = '192.168.1.1'                   IP-Adresse des Servers
serverPort = 5007                                Port

clientsocket = socket(AF_INET, SOCK_STREAM)      Socket erzeugen
clientsocket.connect((serverHostName, serverPort))  Verbindung herstellen

clientsocket.send("Hello Server")                String senden
data = clientsocket.recv(1024)                   Antwort empfangen
print "Client received:", data                   und anzeigen

clientsocket.send("How are you")
data = clientsocket.recv(1024)
print "Client received:", data

clientsocket.close()                             Verbindung trennen

raw_input()                                      Auf Taste warten
```

Zum Experimentieren starte den Server auf Deinem Computer in einem DOS-Fenster (einfach Datei doppelclicken), setze im Clientprogramm die IP-Adresse auf Deinen eigenen Rechner (127.0.0.1) und starte (eventuell) mehrere Clients durch Doppelclick. Nicht in der IDLE ausführen!

Beim Start mehrerer Clients erkennst Du, dass auf der Serverseite jeder eine eigene interne Portnummer erhält! Damit kannst Du die einzelnen Clients voneinander unterscheiden.

Aufgaben:
- programmiere im Client eine Schleife, die mittels raw_input Strings von der Tastatur entgegennimmt, diese an den Server schickt und seine Antwort anzeigt.
- Der Client soll bei Eingabe des Wortes 'STOP' seine Arbeit beenden
- Der Server soll, wenn er von einem Client das Wort 'SHUTDOWN' empfängt, seine Dienste einstellen.

UDP Server/Client

Der Server soll die von den Clients gesendeten Strings einfach nur anzeigen.

Das Server-Programm

`from socket import *`	
`myHostName = ""`	
`myPort = 50007`	
`mysocket = socket(AF_INET, SOCK_DGRAM)`	Datagramm Protokoll
`mysocket.bind((myHostName, myPort))`	
`while 1:`	
`    data, address = mysocket.recvfrom(1024)`	warte auf Empfang von max. 1024 Byte
`    print address,"sagte:", data`	und zeige sie an
`    if data=="KILL": break`	dieses Wort schließt den Server
`mysocket.close()`	Schließen
`raw_input()`	auf <RETURN> warten

Das Client-Programm

`from socket import *`	
`theHostName = "192.168.1.1"`	Adresse des Servers
`thePort = 50007`	
`mysocket = socket(AF_INET, SOCK_DGRAM)`	Socket erzeugen
`while 1:`	
`    msg = raw_input("Sag etwas: ")`	Zeile einlesen
`    if msg:`	
`        mysocket.sendto(msg,(theHostName, thePort))`	an Server senden
`    else:`	
`        break`	Leerzeile beendet
`mysocket.close()`	

Komfort

Python hält einige weitere Module bereit, die das Netzwerk-Programmieren erleichtern. Keines dieser Module *muss* man verwenden, alles lässt sich auch direkt über die Sockets erledigen. Doch oft viel umständlicher (und nur mit detaillierter Kenntnis der Protokolle).

Wir haben die Module `smtplib` zum Versand und `poplib` zum Abholen von E-Mails bereits kennengelernt.

Um beliebige Dateien (HTML-Seiten, Bilder, Mediendateien auf ftp-Servern,...) abholen zu können, ohne sich um Protokolldetails kümmern zu müssen, gibt es das Modul `urllib`. Man braucht nur die Adresse des Servers und den Pfad zur Datei:

```
import urllib
filmseite = "http://www.hib-wien.at/aktiv/HIB-Filme.html"
datei = urllib.urlopen(filmseite).readlines()
print datei
```

holt eine html-Datei als Liste von einzelnen Zeilen und zeigt sie an.

Oder wir speichern die Datei ab:

```
import urllib
film = "http://www.hib-wien.at/aktiv/HIB-Film.divX"
urllib.urlretrieve(film,"H:\meineFilme\HIB-demo.divX")
```

Applikationen

Für unsere Zwecke ist das TCP Protokoll praktisch. Wollen wir allerdings einen Server programmieren, der zu mehreren Clients gleichzeitig Verbindungen unterhält und diese auch alle 'parallel' und ohne merkliche Zeitverzögerungen unterstützt, stehen wir vor einem Problem. Die Clients sollen ja gleichzeitig zugreifen dürfen, und nicht immer nur einer nach dem anderen. Wir müssen also das Serverprogramm in parallel laufende Unterprogramme aufteilen, sogenannte **Threads**.
Der Server arbeitet dann so: In seiner Hauptschleife wartet er auf Clients und erzeugt mittels der accept-Methode für jeden ankommenden Client einen eigenen Socket, der in einem eigenen Thread bearbeitet wird.

Was brauchen wir für einen Multi-Client Server:
- er soll TCP verwenden und Threads beherrschen
- er soll die Clients quasi-parallel bearbeiten

Was wollen wir zur Verfügung stellen:
- eine Funktion, die die Anfragen der Clients bearbeitet

Wie wir das mit Objekten, Methoden und dem praktischen Modul `SocketServer` (das erledigt alle TCP-Grundfunktionen automatisch) bewerkstelligen können, wollen wir bei der Programmierung eines Chat-Servers lernen.