

# ChatServer

Wir wollen einen universell verwendbaren Server programmieren, der die wichtigsten Funktionen eines Chat-Servers erfüllt:

- es soll ein 'Threaded TCP Server' sein
- Clients können sich mit Port Nummer (änderbar) 5000 anmelden
- die Clients werden einzeln begrüßt
- sie können sich mit einem Spezialkommando '/quit' wieder abmelden
- Die wichtigste Funktion: Jede Eingabezeile eines Clients wird an alle Clients weiterverbreitet.

Erweiterungen, die wir im Auge behalten:

- bei der Anmeldung wählt der Client einen Namen (Nickname), der in der Konversation verwendet wird
- durch ein Spezialkommando kann man nur einem einzigen anderen Client etwas mitteilen ('flüstern', eventuell mittels [Name] Text)

Du siehst, die Erweiterungen betreffen die Art der Konversation und die Formulierung von Anforderungen – das wird dann das 'Protokoll' unseres Chatraumes.

Als Client kommt in Frage:

- ein simpler TCP Client, der nur in einer Endlosschleife liest und anzeigt (Welches Problem kann hier auftauchen – kann dieses Programm gleichzeitig einlesen, senden und empfangen?)
- ein GUI-Programm, das ganz komfortabel ein Ausgabe-Textfenster und eine Eingabezeile besitzt
- ein beliebiges Telnet-Programm, wie es im Lieferumfang von Windows und Unix enthalten ist. (Beachte den Zeilenvorschub: "\n" versteht Python, ein Telnet-Programm mit VT-Emulation braucht aber wie Unix zumeist ein "\r\n" (return+newline))  
Am besten programmieren wir unseren Client später so, dass er für das passende Zeilenendzeichen zur Ausgabe auf dem Bildschirm sorgt.

## Unser Server

Ich möchte das Programm 'top-down' erklären, also von den großen Einheiten hinunter zu den kleinen.

### **Zuerst die Importe von Bibliotheks-Modulen**

```
from threading import *
from Queue import Queue
import SocketServer
```

**threading** enthält die Klasse Thread, die uns Methoden zur Parallel-Abarbeitung von Prozessen bereitstellt.

Die einzelnen Anforderungen an den Server werden wir in Form einer **Queue** (FIFO-Warteschlange) zwischenspeichern, damit jeder Client seine Botschaften konfliktlos ablegen kann. Da mehrere Threads gleichzeitig auf diese Daten zugreifen können, müssen wir einen lock-release Mechanismus einbauen, damit nicht ein zweiter Thread Daten während ihrer Abarbeitung verändern

kann. Der lock sollte 'reentrant' sein (der selbe Programmteil kann mehrfach aufgerufen sein). Und **SocketServer** enthält bereits einen rudimentären TCP Server als praktische Klasse, den wir unseren Anforderungen anpassen werden.

### **Der Aufruf unseres Servers:**

```
def start(port=5000):  
  
    server = MyChatServer(('localhost',port),MyRequestHandler)  
  
    try:  
        print "Server started, waiting for connections."  
        server.serve_forever()  
    except:  
        server.shutdown()  
        print "Shut down!"  
  
start(5000)
```

MyChatServer ist die oben erwähnte von uns angepasste Serverklasse, von der server eine Instanz darstellt. Die Methode `serve_forever()` lässt die Hauptschleife des Servers ewig laufen. Aber was soll unser Server eigentlich mit den Clients tun? Nun – jeden in einem eigenen Thread behandeln. Und wie? Das teilen wir durch die Angabe eines Handlers mit – `MyRequestHandler` wird als Klassenname übergeben, damit der Server dann für jeden ankommenden Client ein eigenes Objekt, das mit dieser Handler-Methode ausgestattet ist, erzeugen kann.

### **Nun zur Definition von MyChatServer:**

```
class MyChatServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):  
  
    allow_reuse_address = 1  
  
    def __init__(self,server_address,request_handler_Class):  
  
        SocketServer.TCPServer.__init__(self,server_address,  
                                         request_handler_Class)  
  
        self.chatroom = ChatRoom()  
        self.chatroom.setDaemon(1)  
        self.chatroom.start()  
  
    def shutdown(self):  
        self.chatroom.shutdown()
```

Wir leiten unseren Server natürlich von `SocketServer.TCPServer` ab. Die Behandlung der Clients soll durch ständiges Kontrollieren aller Verbindungen geschehen: `ThreadingMixIn`. Die `init`-Methode bereitet den TCP Server vor, übergibt unsere Adresse und die erwähnte Handler-Klasse.

Was der Server in seiner Hauptschleife tun soll – das Verteilen der Meldungen – wollen wir nicht hier programmieren, sondern in einer eigenen Klasse. `ChatRoom()` soll sie heißen und als Thread laufen.

### ***Nun fehlt noch die Handler-Klasse:***

```
class MyRequestHandler(SocketServer.StreamRequestHandler):

    def setup(self):
        SocketServer.StreamRequestHandler.setup(self)
        self.server.chatroom.addChatter(self.sendAnswer)
        print "%s:%d connected" % self.client_address

    def finish(self):
        self.server.chatroom.removeChatter(self.sendAnswer)
        SocketServer.StreamRequestHandler.finish(self)
        print "%s:%d disconnected" % self.client_address

    def handle(self):
        self.sayHello()
        while 1:
            line = self.rfile.readline()
            if not line:
                break
            line = line.strip()
            if line.upper().startswith("/QUIT"):
                break
            if line.upper().startswith("/SHUTDOWN"):
                self.server.shutdown()
                break
            if line != "":
                message = ("<%s:%d> " % self.client_address)+line
                self.server.chatroom.spread(message)

        def sayHello(self):
            self.wfile.write("Willkommen %s! Tippe '/quit' um
auszusteigen!\r\n"% str(self.client_address))

        def sendAnswer(self,msg):
            self.wfile.write(msg+'\r\n')
```

`setup` und `finish` überschreiben die Originalmethoden und sollen dem Server das Ankommen bzw. das Abmelden eines Clients mitteilen.

`handle` ist nun die eigentliche Bearbeitungsroutine. Am Anfang begrüßen wir den neuen Chat-Gast und erklären die wichtigsten Dinge. Dann gehen wir in eine Endlosschleife, wo wir

- mit `rfile` den Socket wartend auslesen – wir warten auf eine Textzeile dieses Chatters.
- wenn 'nichts' zurückkommt ist die Verbindung unterbrochen und wir beenden die Schleife
- dann testen wir auf Spezialkommandos
- und andernfalls basteln wir uns eine Zeile zusammen, die den Schreiber dieser Zeile und den Inhalt darstellt. (Hier wäre ein Nickname praktisch).
- `self.server.chatroom.spread(message)` ist die Verbindung zu den Chatroom-Funktionen. Auf diese Weise wird der Befehl zum Verteilen der Botschaft an die Chat-Teilnehmer erteilt.
- eine zentrale Rolle spielt die Methode **`sendAnswer`**. Sie ist die Nabelschnur, mit der unser Server den Client erreichen kann. Bei `addChatter` übergeben wir auch genau die Adresse

dieser Funktion. Will der Server dem Chatter etwas schicken, braucht er nur diese Methode aufzurufen. Da sie also den Chat-Teilnehmer aus Sicht des Servers vollständig repräsentiert, nennen wir diese Methode dann im Chatroom `chatter`. (Eingaben vom Client zum Server muss nicht der Server entschlüsseln: unser Handler untersucht die Eingabezeile und leitet nötigenfalls selbst die Server-Aktivität wie etwa `addChatter` oder `removeChatter` ein.)

### **Und jetzt nur mehr die Funktionen des Chatrooms:**

```
class ChatRoom(Thread):

    def __init__(self):
        Thread.__init__(self)
        self.events = Queue()
        self.messages = Queue()
        self.chatters = []
        self.chatters_lock = RLock()
        self._shutdown = 0
```

`events` sind Ereignisse, die der Server bearbeiten soll. `messages` sind die zugehörigen Botschaften, die an die Clients verschickt werden sollten. `chatters` ist die Liste aller aktiver Teilnehmer – tatsächlich ist es aber die Liste der entsprechenden `sendAnswer` Methoden. `chatters_lock` ist der Lock, den wir zum Ansprechen von Daten verwenden, die im Hintergrund ständig verändert werden könnten.

Eine Methode zum Schließen des Chatrooms könnte man vorbereiten:

```
def shutdown(self):
    self._shutdown = 1
    print "shutdown requested!"
    self.messages.put("Server is shutting down!")
    self.events.put("/shutdown")
```

Die folgenden beiden Funktionen nehmen einen neuen Client in die Liste auf, bzw. entfernen ihn daraus.

```
def addChatter(self, chatter):
    self.chatters_lock.acquire()
    try:
        self.chatters.append(chatter)
    except: pass
    self.chatters_lock.release()

def removeChatter(self, chatter):
    self.chatters_lock.acquire()
    try:
        self.chatters.remove(chatter)
    except: pass
    self.chatters_lock.release()
```

Nun die Hauptschleife des Threads. Wir warten ständig auf Ereignisse und liefern Botschaften an alle Teilnehmer. Für spezielle Zusatzfunktionen ('Flüstern',...) müssten wir hier oder in `deliverMessage` weiterprogrammieren.

```
def run(self):
    while 1:
        self.waitForEvent()
```

```
        if self._shutdown:
            break
        if self.messages.qsize()>0:
            self.deliverMessage()
```

`spread` erzeugt ein Ereignis und legt den Text in der Queue ab. Diese Methode wird vom Client-Handler aufgerufen!

```
def spread(self,msg):
    self.messages.put(msg)
    self.events.put("message: "+msg)
```

Warte auf das Eintreffen eines Events in der Warteschlange:

```
def waitForEvent(self):
    nextevent = self.events.get()
    print nextevent
    return
```

Die folgende Methode versucht, eine Botschaft aus der Queue an alle (augenblicklich aktiven) Teilnehmer zu verteilen.

```
def deliverMessage(self):
    try:
        msg = self.messages.get_nowait()
    except Queue.Empty:
        return

    self.chatters_lock.acquire()
    try:
        chatters_copy = self.chatters[:]
    except:
        chatters_copy = []
    self.chatters_lock.release()

    for chatter in chatters_copy:
        self.sendtoChatter(chatter,msg)
```

Und die letzte Methode führt die Versendung für einen Client tatsächlich durch, indem sie `chatter`, was ja nichts anderes als `sendAnswer` eines Clients ist, aufruft. Falls ein Client nicht mehr erreichbar ist, löschen wir ihn aus der Liste.

```
def sendtoChatter(self,chatter,msg):
    try:
        chatter(msg)
    except:
        self.removeChatter(chatter)
```

Wie wir nun einen passenden eleganten Chat-Client programmieren können, soll ein weiteres Kapitel sein.