Johannes Gutenberg-Universität Mainz
Institut für Informatik
Einführung in die Softwareentwicklung
Prof. Dr. Michael Wand
Dr. Christian Ali Mehmeti-Göpel
Visual Computing Group

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Einführung in die Softwareentwicklung

ÜBUNGSBLATT 07: Client-Server Programmierung

30. Juni 2025

Über dieses Übungsblatt

Auf diesem letzten Übungsblatt schreiben wir unsere eigene Client-Server Applikation: Einen minimalen Webserver und einen minimalen Webbrowser. Als Programmiersprache sind Python oder Scala möglich; der Beispielcode wird in Python bereitgestellt (eine Umstellung in C/C++, JAVA, Scala o.ä. besteht hier im Wesentlichen nur aus minimalen syntaktischen Anpassungen/Umbenennungen).

Hinweise zu Sockets und IP: Beachten Sie bei der Programmierung mit Sockets, dass das Internet voller zwielichtiger Gestalten und Portscanner ist. Je nach Implementation gibt es – zumindest ein hypothetisches Risiko – dass der Code angreifbar ist. Es ist daher zu empfehlen, die Beispielprogramme und eigenen Lösungen nur auf dem lokalen Rechner zu testen. Wenn man vorsichtig sein will, sollte man alles hinter einer Firewall (z.B. aus dem OS[1] oder dem Heimrouter; wer wirklich besorgt ist schaltet noch das LAN/WLAN ab) ausführen. Das Beispielprogramm nutzt Parameter (local-host/127.0.0.1 restriction), die laut Python-Socket-Doku nur lokale Verbindungen erlauben sollten – dies aber ohne Garantie, dass das allen Betriebssystemen zuverlässig funktioniert.

Abgabe: (Abgabe 14. SW)

Für dieses letzte Übungsblatt ist zwei Wochen Bearbeitungszeit vorgesehen. Das Übungsblatt muss bis zum **13. Juli 2025, 23:59h** in LMS abgegeben werden. Laden Sie dazu den Quellcode aller Aufgaben hoch. Die Ergebnisse müssen in den Übungen zwischen dem 14.-18. Juli 2025 vorgestellt werden (also in der letzten Vorlesungswoche).

[1] Manche Betriebssysteme wie Windows erfordern ggf. die OS-eigene Firewall so zu konfigurieren, dass lokale IP-Verbindungen zum eigenen Programm erlaubt werden.

Aufgabe 1: Einstieg in die Client-Server Programmierung in Python (5+5 = 10 Punkte)

In dieser Aufgabe soll *lediglich fertiger Beispielcode ausprobiert* werden. In Aufgabe 2 soll dieser dann zu einem lauffähigen Client-Server-Programm erweitert werden.

a) Schreiben Sie einen einfachen „Server" in Python (oder Scala), der einen TCP-Server-Socket öffnet und auf Verbindungen wartet. Sobald eine aufgebaut wird, sollen die eingehenden Daten einfach als UTF-8 String interpretiert und auf der Console ausgegeben werden. Versuchen Sie dann, sich mit einem Webbrowser via http mit dem Server zu verbinden und schauen Sie, was auf der Konsole passiert.

Listing 1 (s.u. / am Ende dieses Dokumentes) zeigt den *fertigen Code* mit Kommentaren, die dabei helfen sollen, das Socket-Interface für Python zu verstehen[2] (siehe auch https://docs.python.org/3/howto/sockets.html für mehr Infos).

Auf meinem System liefert Firefox mit dem Request „`http://localhost:41337/`" in der Adresszeile z.B. folgende Ausgabe:

```
GET / HTTP/1.1
Host: localhost:41337
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:127.0) Gecko/20100101
Firefox/127.0
Accept:  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,im-
age/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.7,de-DE;q=0.3
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Priority: u=1
```

Danach wartet der Browser vergebens auf eine Antwort und gibt frustriert auf (klar, unser Programm reagiert ja nicht). Schreiben wir daher nun einen eigenen Client.

b) Schreiben Sie einen einfachen Client, der sich mit dem Server verbindet und im Sekundentakt die einige Worte an Daten schickt.

Listing 2 (s.u./Ende des Dokuments) zeigt den fertigen Code, um das Client-Interface von Python-Sockets zu illustrieren.

Hinweis: Falls Sie eine vollständige Lösung für Aufgabe 2 in der Übungsabgabe demonstrieren, wird stillschweigend angenommen, dass Sie Aufgabe 1 richtig gelöst haben. Sollte es noch Fehler geben, so können mit Aufgabe 1 aber noch Punkte gesammelt werden.

[2] In anderen Programmiersprachen ist dies ähnlich gelöst, da fast immer Standard BSD-Sockets genutzt werden, wie in der Vorlesung diskutiert.. Um Beispielcode für Scala zu finden, suchen Sie z.B. einfach nach „java socket tutorial" mit der WWW-Suchmaschine Ihrer Wahl.

Aufgabe 2: Ein minimalistischer Webserver und Webbrowser (35+5 = 40 Punkte)

a) Erweitern Sie den minimalistischen Server und Client Code so, dass wir einen „richtigen" Webserver und Webbrowser erhalten. Diese beiden Programme sollten folgende Funktionen unterstützen:

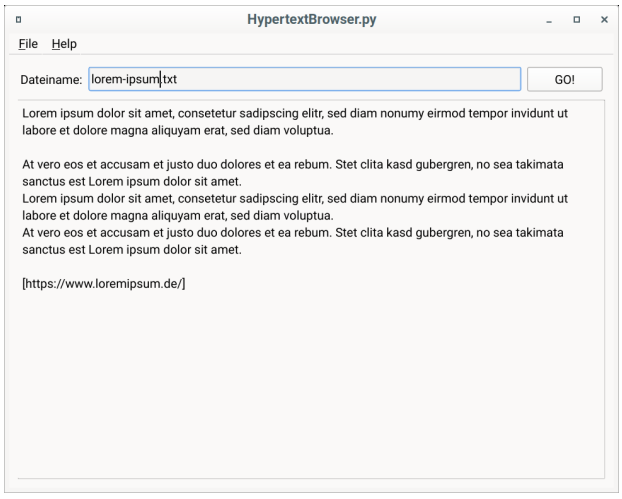
Server: Wenn man dem Server (in einem geeigneten Format/Protokoll, dass Sie selbst frei definieren dürfen) eine Anfrage schickt, in der ein Dateiname kodiert ist, so liest der Server aus einem Unterverzeichnis (z.B. „`www`" im Verzeichnis des Programms) die Datei mit diesem Namen aus und schickt den Inhalt an den Client. In diesem Verzeichnis sollten nur Textdateien liegen. Es ist nicht erforderlich, dass der Server große Dateien (über dem Limit eines `recv`-Befehls) verschicken kann.

Client: Der Client sollte ermöglichen, sich mit dem Server (ausreichend: nur auf dem gleichen Rechner mit fest kodiertem Port) zu verbinden und Dateien abzufragen. Der Inhalt der Datei soll dann als Text ausgegeben werden. Es ist nicht erforderlich, dass der Client große Dateien (über dem Limit eines `recv`-Befehls) empfangen kann.

Dies kann beispielsweise als simple Konsolenanwendung geschehen, bei der man einen Dateinamen wie „Testdokument.txt" eingibt, und man dann den Inhalt der Datei angezeigt bekommt, gefolgt von der nächsten Anforderung. Hier eine Beispielsession (Benutzereingaben fett, Inhalt der Textdateien kursiv gedruckt):

```
Which document should I get for you? test-text.txt
Dies ist ein Test!
Dies ist ein Test!
Which document should I get for you? python-food.txt
SPAM
SPAM SPAM
SPAM SPAM SPAM
Which document should I get for you? <return>
goodbye!
```

b) Um die volle Punktzahl zu erreichen, sollten Sie für den Client ein einfaches QT-GUI mit einer Adresszeile (dort den Dateinamen eingeben) und einem Widget für die Textausgabe (z.B. `QTextEdit` oder `QLabel`) sowie geeigneten Knöpfen zum Start der Übertragung (z.B. `<return>` in der Adresszeile) schreiben. Siehe Screenshot der Musterlösung unten.



Tipp: `QTextEdit` ist voll HTML-fähig; sie können also, wenn Sie möchten, „richtige" HTML-Dateien anzeigen (und sogar Bilder u.ä., wenn man das voll ausreizt). Dies alles ist natürlich optional und ohne Punkte.

```python
# Minimaler Server

import socket

# 127.0.0.1 ist "localhost" - Verbindungen werden NUR vom eigenen Rechner akzeptiert.
# Das ist ein wichtiges Sicherheitsfeature; so kann die ganze böse Welt da draußen
# unser Spielzeug nicht so leicht angreifen. Um alle Verbindungen zu erlauben,
# nutzt man hier einen leeren String (nicht empfohlen).
#
IP: str = "127.0.0.1"

# Als Portnummer für den Server sollte man eine Zahl > 1024 nehmen, da man sonst
# u.U. Root-Rechte braucht
#
PORT: int = 41337

# Die "with"-Anweisung in Python sorgt dafür, dass bei Exceptions der Socket trotzdem
# wieder richtig geschlossen wird (über magic-methods "__enter__" und "__exit__", die
# die Socket-Klasse unterstützt). Hier ist das kosmetisch, vereinfacht aber Syntax
# und Nutzung (man könnte auch einen "try...finally"-Block nutzen, oder Exceptions
# einfach ignorieren, da das nur ein Spielzeugprogramm ist und das OS am Ende aufräumt).
#
# with...as: in "sock" steht die Rückgabe der Funktion socket.socket(), die sockets erzeugt.
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Binden des Sockets an IP-Adresse und Port
    sock.bind((IP, PORT))
    # Warten, bis jemand anklopft...
    sock.listen()
    # Wir bekommen einen neuen Socket "connection" zurück und dessen IP-Adresse und Port
    connection, address = sock.accept()
    # Das Gleiche wieder: die Verbindung soll im Fehlerfall zuverlässig geschlossen werden
    # (auch hier optional)
    with connection:
        # Unser Server läuft forever!
        while True:
            # Wir empfangen ein "raw" Byte-Array.
            # Man muss die maximale Zahl von Bytes, die auf einmal empfangen werden
            #  können, angeben; laut Doku darf die wg. technischer Limits nicht beliebig groß
            # sein. Ich habe die Info gefunden, das hier 4K eine sinnvolle Obergrenze sind.
            # Will man mehr empfangen, muss man die Daten nacheinander empfangen
            # und zusammenstückeln.
            data: bytes = connection.recv(4096)
            # Dies beendet den Server falls der (simple) Client sich verabschiedet
            if len(data) == 0:
                break
            # Wir convertieren das ganze voller Vertrauen (Hallo liebe Hacker!) in Python
            # UTF-8 Strings und geben das ganze auf der Konsole aus.
            print(data.decode("utf-8"))
```

Listing 1: Fertiger Code für Aufgabe 1a)

```python
# Minimal Client

import socket
from time import sleep

IP: str = "127.0.0.1"  # **Server** IP
PORT: int = 41337      # **Server** port

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Binden des Sockets an IP-Adresse und Port
    sock.connect((IP, PORT))
    # Wir senden Blödsinn an die Gegenstelle...
    for i in range(3):
        sock.sendall(bytes("SPAM! SPAM! SPAM!\n", "utf-8"))
        sleep(1) # eine Sekunde warten
```

Listing 2: Fertiger Code für Aufgabe 1b)