

# Vorlesung Rechnernetze

## Laborübung

### Einstieg in die Socketprogrammierung

Prof. Dr. Dirk Staehle

Die Abgabe erfolgt durch Hochladen der bearbeiteten Word-Datei in Moodle.

#### **Bearbeitung in Zweier-Teams**

**Team-Mitglied 1: Yasmin Hoffmann**

**Team-Mitglied 2: Chris Jakob**

**Team-Mitglied 3: Axel Schwarz**

#### **1 Einleitung**

In dieser Übung lernen Sie die Programmierung mit Sockets kennen. Die Funktionen zum Umgang mit Sockets sind in jeder Programmiersprache ähnlich, da Sockets vom Betriebssystem angeboten werden. In der Laborübung wird als Programmiersprache Python verwendet. Python ist eine moderne Interpreter-Sprache, die als einfach zu erlernen gilt. Python hat sich in den

letzten Jahren zu einer der populärsten Sprache für die Entwicklung von Netzwerk- und Webanwendungen entwickelt.

Dieser Laborversuch besteht aus drei Aufgaben, die in zwei Laborstunden durchzuführen sind. Im ersten Versuch lernen Sie „telnet“ kennen. Telnet ist ein Tool mit dem Sie über die Kommandozeile in einen Socket schreiben bzw. aus einem Socket lesen können. Als Beispiel für telnet dient die tägliche Mail, die Sie dann auch mit einem Python-Skript schreiben dürfen. Der zweite Versuch dient zum Einstieg in die Socket-Programmierung mit Python. Sie implementieren hier einen einfachen Rechenserver. Das dritte Beispiel ist das Erstellen eines Port Scanners, um den Umgang mit Sockets, Threads und Timeouts weiter zu üben.

## 2 Vorbereitung

Wenn Sie mit Python noch nicht vertraut sind, arbeiten Sie sich in Python ein, indem Sie ein Python-Tutorial durchgehen. Python ist auf den Laborrechnern bereits installiert. Sie können beispielsweise Spyder oder IDLE als IDE nutzen.

Sockets werden in Python von der Bibliothek „socket“ unterstützt. Beschreibungen der Socket-Programmierung in Python finden Sie unter

1. <https://docs.python.org/3/howto/sockets.html>
2. <https://docs.python.org/3/library/socket.html>

## 3 Mail

In diesem Versuch verstehen Sie am Beispiel eines Mail-Clients, wie Anwendungen über Sockets kommunizieren. Sie lernen das Tool „telnet“ kennen, mit dem ein Socket geöffnet werden kann, um über die Kommandozeile mit einem Server zu kommunizieren. Im ersten Versuch, bauen Sie eine Verbindung zu einem Mail-Server, um über SMTP eine Mail zu schreiben und über IMAP Mails zu lesen. In einem zweiten Versuch implementieren Sie einen Mail-Client in Python, um Mails zu versenden oder abzurufen.

### 3.1 SMTP über telnet

Mit telnet können Sie einen Socket zu einem Server auf einem bestimmten Port öffnen und dann über Kommandozeilen-Eingabe mit dem Server kommunizieren. In diesem Beispiel sollen Sie über telnet mit dem SMTP Protokoll Mails verschicken. Öffnen Sie dazu mit telnet einen Socket zum (A)SMTP-Port (587) des Mail-Servers `asmtpt.htwg-konstanz.de` und melden Sie sich mit dem Account (Login: `rnetin`, Password: `Ueben8fuer8RN`) an. Schreiben Sie eine Email an einen ihrer Mail-Accounts und prüfen Sie, ob die Mail angekommen ist.

Hinweise:

1. Achten Sie darauf, dass Sie sich im VPN befinden.
2. Zeichnen Sie ihre SMTP Session mit WireShark auf.
3. Sie können „telnet“ entweder in der Windowskommandozeile oder über „Putty“ öffnen.

4. Gehen Sie vor wie beispielsweise auf Wikipedia beschrieben:

[https://de.wikipedia.org/wiki/Simple\\_Mail\\_Transfer\\_Protocol](https://de.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol)

<https://de.wikipedia.org/wiki/SMTP-Auth>

5. Login und Passwort müssen als Base64-kodierte Zeichenketten übertragen werden. Sie können die Konvertierung mit Hilfe eines Online-Tools durchführen oder die PythonBibliothek `base64` verwenden. Importieren Sie dazu die `base64`-Bibliothek in der Python Shell (`import base64`) und verwenden Sie folgende Befehle zum Konvertieren zwischen ASCII- und Base64-Zeichenketten:

```
(base64.b64encode('ASCII-String'.encode('utf-8'))).decode('utf-8')
(base64.b64decode('Base64-String')).decode('utf-8')
```

3. Die Antwort-Mail können alle Labor-Teilnehmer lesen, da alle den gleichen Mail-Account für ping asmtip die Laborübung nutzen.
4. Achten Sie bei der Eingabe von Text in telnet darauf, dass Sie sich nicht vertippen. Eine Korrektur ist nicht mehr möglich. Ein praktikabler Umgang mit diesem „Problem“ besteht darin, die Kommandos in einem Editor zu entwerfen und dann in das „telnet-Fenster“ zu kopieren. Das ist auch hilfreich für die nächste Aufgabe, wenn Sie den Mail-Client in Python implementieren.

→ Verwendete Befehle in telnet:

```
telnet asmtip.htwg-konstanz.de 587

EHLO example.net

AUTH LOGIN

cm5ldGlu //username: rnetin

VWViZW44ZnVlcjhSTg== //passwort: Ueben8fuer8RN

MAIL FROM: <yasminhoffmann99@web.de>

RCPT TO:<yasmin.hoffmann@htwg-konstanz.de>

DATA

From: <yasminhoffmann99@web.de>
To: <yasminhoffmann@htwg-konstanz.de>
Subject: Testmail1

Hallo,
das ist eine neue Testmail fuer RN.
.
```

## 5. Betrachten Sie die Aufzeichnung in WireShark. Was fällt Ihnen auf?

No.	Time	Source	Destination	Protocol	Length	Info
12	1.715919	141.37.168.39	141.37.11.129	TCP	66	53628 → 587 [SYN] Seq=0 Win=0 Len=0 MSS=1460 WS=256 SACK_PERM=1
13	1.716484	141.37.11.129	141.37.168.39	TCP	66	587 → 53628 [SYN, ACK] Seq=0 Ack=1 Win=1460 Len=0 MSS=1380 SACK_PERM=1 WS=16
14	1.716453	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=1 Ack=1 Win=2097408 Len=0
15	1.718372	141.37.11.129	141.37.168.39	SMTP	189	S: 220 asmtg.konstanz.de ESMTP Postfix (Debian/GNU)
16	1.726248	141.37.168.39	141.37.6.164	TCP	54	51553 → 445 [ACK] Seq=455 Ack=1866 Win=8287 Len=0
17	1.773884	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=1 Ack=56 Win=2097408 Len=0
31	6.366598	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=1 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
32	6.367137	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=2 Win=14608 Len=0
33	6.368588	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=2 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
34	6.368916	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=3 Win=14608 Len=0
35	6.369804	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=3 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
36	6.369289	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=4 Win=14608 Len=0
37	6.369394	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=4 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
38	6.369587	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=5 Win=14608 Len=0
39	6.369662	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=5 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
40	6.369858	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=6 Win=14608 Len=0
41	6.369987	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=6 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
42	6.370099	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=7 Win=14608 Len=0
43	6.370156	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=7 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
44	6.370358	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=8 Win=14608 Len=0
45	6.370487	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=8 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
46	6.370599	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=9 Win=14608 Len=0
47	6.370657	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=9 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
48	6.370852	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=10 Win=14608 Len=0
49	6.370987	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=10 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
50	6.371098	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=11 Win=14608 Len=0
51	6.371156	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=11 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
52	6.371345	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=12 Win=14608 Len=0
53	6.371482	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=12 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
54	6.371603	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=13 Win=14608 Len=0
55	6.371642	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=13 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
56	6.371861	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=14 Win=14608 Len=0
57	6.371881	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=14 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
58	6.372136	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=15 Win=14608 Len=0
59	6.372145	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=15 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
60	6.372481	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=16 Win=14608 Len=0
61	6.372411	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=16 Ack=56 Win=2097408 Len=1 [TCP segment of a reassembled PDU]
62	6.372756	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=17 Win=14608 Len=0
64	6.744613	34.107.148.139	141.37.168.39	TLShv.2	127	Application Data
65	6.744659	141.37.168.39	34.107.148.139	TCP	54	53464 → 443 [FIN, ACK] Seq=1 Ack=74 Win=1023 Len=0
66	6.751987	34.107.148.139	141.37.168.39	TCP	60	443 → 53464 [FIN, ACK] Seq=74 Ack=2 Win=1060 Len=0
67	6.752842	141.37.168.39	34.107.148.139	TCP	54	53464 → 443 [ACK] Seq=2 Ack=75 Win=1023 Len=0
70	7.140438	141.37.168.39	141.37.11.129	SMTP	56	C: DRO example.net
71	7.140962	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=56 Ack=19 Win=14608 Len=0
72	7.140962	141.37.11.129	141.37.168.39	SMTP	233	S: 250-asmtg.konstanz.de [250-PIPELINING] [250-SIZE 54525952] [250-ETRN] [250-STARTTLS] [250-AUTH PLAIN LOGIN] [250-ENHANCEDSTATUSCODES] [250-8BITMIME] [250-...
73	7.194849	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=19 Ack=235 Win=2097152 Len=0
91	11.654338	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=19 Ack=235 Win=2097152 Len=1 [TCP segment of a reassembled PDU]
92	11.693408	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=235 Ack=20 Win=14608 Len=0
93	11.693563	141.37.168.39	141.37.11.129	TCP	63	53628 → 587 [PSH, ACK] Seq=20 Ack=235 Win=2097152 Len=9 [TCP segment of a reassembled PDU]
94	11.694069	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=235 Ack=29 Win=14608 Len=0
99	13.004298	141.37.168.39	141.37.11.129	SMTP	56	C: AUTH LOGIN
100	13.004619	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=235 Ack=31 Win=14608 Len=0
101	13.004820	141.37.11.129	141.37.168.39	SMTP	72	S: 334 VbWlcmShbM6U
102	13.045566	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=31 Ack=253 Win=2097152 Len=0
118	17.813944	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=31 Ack=253 Win=2097152 Len=1 [TCP segment of a reassembled PDU]
119	17.853394	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=253 Ack=32 Win=14608 Len=0
120	17.853441	141.37.168.39	141.37.11.129	TCP	61	53628 → 587 [PSH, ACK] Seq=32 Ack=253 Win=2097152 Len=7 [TCP segment of a reassembled PDU]
121	17.853846	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=253 Ack=39 Win=14608 Len=0
125	18.628291	141.37.168.39	141.37.11.129	SMTP	56	C: User: cm01061u
126	18.628756	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=253 Ack=41 Win=14608 Len=0
127	18.628944	141.37.11.129	141.37.168.39	SMTP	72	S: 334 UGfc3hcvQm6
128	18.683533	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=41 Ack=271 Win=2097152 Len=0
144	23.118614	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=41 Ack=271 Win=2097152 Len=1 [TCP segment of a reassembled PDU]
145	23.157308	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=271 Ack=42 Win=14608 Len=0
146	23.157420	141.37.11.129	141.37.11.129	TCP	73	53628 → 587 [PSH, ACK] Seq=42 Ack=271 Win=2097152 Len=19 [TCP segment of a reassembled PDU]
147	23.157849	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=271 Ack=61 Win=14608 Len=0
148	23.900367	141.37.168.39	141.37.11.129	SMTP	56	C: Pass: VbWl2M44znVlcjh5tG==
149	23.900634	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=271 Ack=63 Win=14608 Len=0
150	23.902895	141.37.11.129	141.37.168.39	SMTP	91	S: 235 2.7.0 Authentication successful
151	23.957847	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=63 Ack=388 Win=2097152 Len=0
171	29.080953	141.37.168.39	141.37.11.129	TCP	55	53628 → 587 [PSH, ACK] Seq=63 Ack=388 Win=2097152 Len=1 [TCP segment of a reassembled PDU]
172	29.109258	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=388 Ack=64 Win=14608 Len=0
173	29.189284	141.37.168.39	141.37.11.129	TCP	60	53628 → 587 [PSH, ACK] Seq=64 Ack=388 Win=2097152 Len=35 [TCP segment of a reassembled PDU]
174	29.189615	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=388 Ack=99 Win=14608 Len=0
177	29.772284	141.37.168.39	141.37.11.129	SMTP	56	C: MAIL FROM: cyassinhoffmann9@web.de
178	29.772755	141.37.11.129	141.37.168.39	TCP	60	587 → 53628 [ACK] Seq=388 Ack=101 Win=14608 Len=0
179	29.773547	141.37.11.129	141.37.168.39	SMTP	68	S: 250 2.1.0 OK
180	29.822533	141.37.168.39	141.37.11.129	TCP	54	53628 → 587 [ACK] Seq=101 Ack=322 Win=2097152 Len=0
188	31.679205	141.37.168.39	141.37.6.164	TCP	55	[TCP Keep-Alive] 51553 → 445 [ACK] Seq=454 Ack=1866 Win=8287 Len=1
189	31.679355	141.37.6.164	141.37.168.39	TCP	60	[TCP Keep-Alive ACK] 445 → 51553 [ACK] Seq=1866 Ack=455 Win=257 Len=0

No.	Time	Source	Destination	Protocol	Length	Info
15	1.718372	141.37.11.129	141.37.168.39	SMTP	109	S: 220 asmtplib.htwg-konstanz.de ESMTP Postfix (Debian/GNU)
70	7.140430	141.37.168.39	141.37.11.129	SMTP	56	C: EHLO example.net
72	7.140962	141.37.11.129	141.37.168.39	SMTP	233	S: 250-asmtplib.htwg-konstanz.de   250-PIPELINING   250-SIZE 545259
99	13.004298	141.37.168.39	141.37.11.129	SMTP	56	C: AUTH LOGIN
101	13.004820	141.37.11.129	141.37.168.39	SMTP	72	S: 334 VXNlcm5hbnU6
125	18.628291	141.37.168.39	141.37.11.129	SMTP	56	C: User: cm5ld6lu
127	18.628944	141.37.11.129	141.37.168.39	SMTP	72	S: 334 UGFzc3dvcmQ6
148	23.980367	141.37.168.39	141.37.11.129	SMTP	56	C: Pass: VwViZw44ZnVicjhSTg==
150	23.982695	141.37.11.129	141.37.168.39	SMTP	91	S: 235 2.7.0 Authentication successful
177	29.772284	141.37.168.39	141.37.11.129	SMTP	56	C: MAIL FROM: <yasminhoffmann99@web.de>
179	29.773547	141.37.11.129	141.37.168.39	SMTP	68	S: 250 2.1.0 Ok
206	35.436412	141.37.168.39	141.37.11.129	SMTP	56	C: RCPT TO: <yasmin.hoffmann@htwg-konstanz.de>
208	35.438805	141.37.11.129	141.37.168.39	SMTP	68	S: 250 2.1.5 Ok
228	38.844605	141.37.168.39	141.37.11.129	SMTP	56	C: DATA
230	38.851397	141.37.11.129	141.37.168.39	SMTP	91	S: 354 End data with <CR><LF>.<CR><LF>
255	43.725230	141.37.168.39	141.37.11.129	SMTP	194	C: DATA fragment, 141 bytes
260	44.710805	141.37.168.39	141.37.11.129	SMTP	56	C: DATA fragment, 2 bytes
262	44.728053	141.37.11.129	141.37.168.39	SMTP	90	S: 250 2.0.0 Ok: queued as 0D4C882A1D

→ SMTP Pakete in Klartext

6. Schreiben Sie noch eine Mail an ihren Email-Account. Verwenden Sie willkürliche EmailAdressen für MAIL-FROM sowie für das „from“-Feld in der Mail. Siehe auch Wikipedia Beispiel. Lesen Sie die Mail in ihrem Postfach. Was fällt Ihnen auf?

→ MAIL-FROM , „from“ und „to“ können beliebig gewählt werden. Lediglich RCPT-TO muss die richtige Zieladresse sein. Beim Empfänger wird schlussendlich nur das „from“ Feld angezeigt, dieses kann man somit auch ganz einfach auf **service@paypal.de** oder Ähnliches setzen und gezielte Spam oder Phishing Mails verschicken.

## 3.2 SMTP in Python

Implementieren sie einen SMTP-Client in Python. Benutzen Sie aber nicht die in Python vorhandene Mail-Bibliotheken (smtplib) sondern programmieren Sie direkt auf Sockets.

Hinweise:

1. Verwenden Sie die Python Bibliothek base64, um Login und Passwort in Base64-Code zu konvertieren.
2. Alle Daten werden im UTF-8-Format übertragen.
3. Alle gesendeten Zeilen müssen mit "\r\n" enden, damit der Mail-Server das Zeilenende erkennt. Auch diese Zeichen müssen „UTF-8“-codiert gesendet werden.

→ Siehe 3\_smtp.py Datei.

## 4 Rechen-Server

Implementieren Sie einen einfachen Rechen-Server in Python. Der Rechen-Server soll in der Lage sein, die Summe, das Produkt, das Minimum und das Maximum von n Zahlen zu bestimmen. Die Kommunikation mit dem Server erfolgt über Sockets. Eine Anfrage hat das folgende Format:

<ID><Rechenoperation><N><z1><z2>...<zN>

ID ist ein unsigned Integer (4 Bytes) und dient als Identifikator der Aufgabe. Rechenoperation ist eine der Zeichenketten „Summe“, „Produkt“, „Minimum“, „Maximum“ und wird „UTF-8 kodiert“ übertragen. Die Zahl N wird als Unsigned-Char-Wert (1 Byte) übertragen und gibt an, wie viele Zahlen folgen. Die Zahlen z1 bis zN werden als signed Integer (4 Bytes) übertragen. Verwenden Sie die Funktionen `pack` und `unpack` aus dem Modul `struct`, um die Nachrichten zu erzeugen. Sowohl `struct.pack` als auch `.encode()` liefern Bytes-Objekte, die einfach konkateniert werden können.

Das Ergebnis der Rechnung soll dem Client in folgendem Format zurückgeliefert werden:

<ID><Ergebnis>

Dabei ist ID definiert wie oben und Ergebnis soll ein signed Integer-Wert sein.

Implementieren Sie die Aufgabe mit Stream- und Datagram Sockets.

→ Siehe `4_rechenserver.py`, `4_client_tcp.py`, `4_client_udp.py`

### 4.1 Lokale Kommunikation

Testen Sie die Skripte zunächst lokal, indem Sie für Client und Server die IP-Adresse 127.0.0.1 (localhost) verwenden. Starten sie dazu erst das Server-Skript und dann das Client-Skript jeweils in einem Windows-Cmd-Fenster. Nutzen Sie außerdem das Tool CurrPorts (<https://www.nirsoft.net/utils/cports.html>), um die aktiven Sockets zu überwachen.

Erklären Sie den Zusammenhang von ausgetauschten Paketen und Python-Code, indem Sie

1. für jedes gesendete Paket bestimmen, welcher Befehl in welchem Skript (Client/Server) dafür verantwortlich ist, dass das Paket gesendet wird
2. für jeden blockierenden Befehl bestimmen, die Ankunft welches Pakets dafür verantwortlich ist, dass die Ausführung des Befehls vervollständigt wird

→

No.	Time	Source	Destination	Protocol	Length	Info
45	15.281959	127.0.0.1	127.0.0.1	TCP	56	53676 → 50000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
46	15.282038	127.0.0.1	127.0.0.1	TCP	56	50000 → 53676 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
47	15.282085	127.0.0.1	127.0.0.1	TCP	44	53676 → 50000 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
60	19.571068	127.0.0.1	127.0.0.1	TCP	62	53676 → 50000 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=18
61	19.571101	127.0.0.1	127.0.0.1	TCP	44	50000 → 53676 [ACK] Seq=1 Ack=19 Win=2619648 Len=0
62	19.571385	127.0.0.1	127.0.0.1	TCP	52	50000 → 53676 [PSH, ACK] Seq=1 Ack=19 Win=2619648 Len=8
63	19.571403	127.0.0.1	127.0.0.1	TCP	44	53676 → 50000 [ACK] Seq=19 Ack=9 Win=2619648 Len=0
64	19.571811	127.0.0.1	127.0.0.1	TCP	44	53676 → 50000 [FIN, ACK] Seq=19 Ack=9 Win=2619648 Len=0
65	19.571832	127.0.0.1	127.0.0.1	TCP	44	50000 → 53676 [ACK] Seq=9 Ack=20 Win=2619648 Len=0
66	19.572333	127.0.0.1	127.0.0.1	TCP	44	50000 → 53676 [FIN, ACK] Seq=9 Ack=20 Win=2619648 Len=0
67	19.572367	127.0.0.1	127.0.0.1	TCP	44	53676 → 50000 [ACK] Seq=20 Ack=10 Win=2619648 Len=0

- No.45: CLIENT - sock.connect((Server\_IP,Server\_Port)) sendet Paket
- No.47: CLIENT - sock.connect((Server\_IP,Server\_Port)) beendet
- No.60: CLIENT - sock.send(build.msg()) sendet Paket
- No.61: CLIENT - sock.send(build.msg()) beendet
- No.62: SERVER - sock.send(getAnswer(data)) sendet Paket
- No.63: SERVER - sock.send(getAnswer(data)) beendet
- No.64: CLIENT - sock.close() sendet Paket
- No.65: CLIENT - sock.close() beendet
- No.66: SERVER - sock.close() sendet Paket
- No.67: SERVER - sock.close() beendet

## 4.2 Netzwerk-Kommunikation

Gehen Sie mit ihren Rechnern ins VPN. Konfigurieren Sie die IP-Adressen jetzt so, dass die Kommunikation im lokalen Netzwerk (im VPN) möglich ist.

Beantworten Sie die folgenden Fragen durch Experimente und unter Verwendung der Python-Hilfe:

1. Wie können Sie im Client Python-Skript die IP-Adresse und Port-Nummer des verwendeten lokalen Sockets bestimmen („bestimmen“ im Sinne von herausfinden)?  
→ `ip, port = sock.getsockname()`
2. Wann (in welcher Code-Zeile) und woher erhält ein Client seine IP-Adresse und Port-Nummer?  
→ `sock.connect((Server_IP, Server_PORT))` #vom Server
3. Wie können Sie im Client-Skript die IP-Adresse und Port-Nummer des Sockets setzen?  
→ `sock.bind((IP,port))` #wenn es festgelegt werden muss
4. Warum müssen Sie Timeouts verwenden und wie funktioniert `try ... except`? Mit welchem Befehl können Sie einen gemeinsamen Timeout für alle Sockets setzen?  
→ `timeout: zeit die ein vorgang(z.B. threads) brauchen darf bis er mit einem fehler abgebrochen wird`  
→ `socket.setdefaulttimeout(10)`

5. Finden Sie experimentell heraus, ob Sie einen Server betreiben können, der ECHO-Anfragen auf dem gleichen Port für UDP und TCP beantwortet?

→ nicht möglich

### 4.3 Unterstützung für mehrere Clients

Erweitern sie den Server mit Threads, so dass dieser mit mehreren Clients gleichzeitig verbunden sein kann.

→ Siehe `rechen_server.py`

## 5 Port Scan

### 5.1 Beschreibung

In diesem Versuch führen wir einen Port-Scan durch, um herauszufinden, welche Ports auf einem Server geöffnet sind. Wir scannen zum einen die standardisierten Ports von 1-50 nach offenen TCP Ports. Wenn wir einen offenen Port finden, versuchen wir, eine Nachricht an diesen Port zu schicken. Weiterhin vermuten wir, dass auf dem Server ECHO-Dienste für die Übertragung mit TCP und UDP laufen.

Ist ein TCP-Port auf einem Server geöffnet, dann wird ein Verbindungsaufbau auf diesem Port akzeptiert. Ist der TCP-Port nicht offen, so antwortet der Server entweder nicht (Windows FehlerCode 10060) oder mit einem RST+ACK, in dem der Verbindungsaufbau zurückgewiesen wird (Windows Fehler-Code 10061).

Ist ein UDP Port auf einem Server geöffnet, so antwortet der Server entweder mit einer Nachricht oder überhaupt nicht. Die Reaktion hängt sowohl vom empfangenden Dienst als auch von der Nachricht selbst ab. Ist der UDP Port nicht geöffnet, so antwortet der Server entweder nicht oder mit einem ICMP Paket vom Typ 3, mit dem er mitteilt, dass das Ziel nicht erreichbar ist (Windows Fehler-Code 10054).

### 5.2 Versuch

#### 5.2.1 TCP Port Scanner

Implementieren Sie ein Skript, das eine gegebene Anzahl von TCP-Ports auf einem gegebenen Server scannt und die offenen Ports zurückliefert. Führen Sie das Script für den Labor-Server 141.37.168.26 und Ports zwischen 1 und 50 durch. Zeichnen Sie die Kommunikation mit WireShark auf.



Starten Sie die einzelnen Port-Anfragen als Threads, um den Scan-Vorgang zu beschleunigen. Sie können einfach eine Funktion mit

```
t=Thread(target=<function>,args=(<arg>,))
```

starten. Achten Sie darauf, dass der Thread beendet werden kann. Einfach geht dies mittels eines global Flags `Continue`, das der Thread kontinuierlich abfragt und sich bei `Continue==False` beendet.

Hinweis: <https://docs.python.org/3.4/library/threading.html>

→ siehe `5_portscan_tcp.py`

### 5.2.2 UDP Port Scanner

Erweitern Sie das Script, um auch UDP Ports zu scannen. Führen Sie das Script für den LaborServer 141.37.168.26 und Ports zwischen 1 und 50 durch. Zeichnen Sie die Kommunikation mit WireShark auf. Unterscheiden Sie Ports, auf denen Sie keine Antwort bekommen und Ports, auf denen Sie Fehlermeldung 10054 erhalten.

→ siehe `5_portscan_udp.py`

## 5.3 Fragen

1. Geben Sie die Liste der offenen TCP und UDP Ports an.
  - TCP: offene Ports: [7, 9, 13, 17, 19]
  - UDP: offene Ports: [7, 13, 17]
2. Wählen Sie für TCP und UDP jeweils einen offenen und einen geschlossenen Port und erklären Sie die entsprechende Paketsequenz, die Sie in WireShark aufgezeichnet haben.
3. Auf Port 7 des Servers läuft ein ECHO-Dienst. Testen Sie ihr Client-Script mit dem ECHO Server. Versuchen Sie das TCP und das UDP Script.
  - Port 7 beim TCP script liefert keine Antwort zurück, da wir keine Nachricht schicken sondern nur eine Verbindung aufbauen
  - Port 7 beim UDP script liefert den Inhalt der gesendeten Nachricht zurück