8 Ein erster Server für Webseiten

Bislang haben wir für den ESP32 nur Clients programmiert: Diese haben Daten von einem bereits bestehenden Server abgerufen. Nun wollen wir selbst einen solchen Server programmieren (`sta_server_0.py`); Server und Client sollen sich dabei im selben Heimnetz befinden. Als Client werden wir ein Handy oder einen PC einsetzen.

Programmieren: Wlan einrichten und Server-Socket erzeugen

Der erste Schritt ist uns zur Genüge bekannt: Ein Wlan-Objekt muss erzeugt und entsprechend mit dem AP verbunden werden. Die Vorgehensweise ist genauso wie bei den letzten Kapiteln.

```
import network
from time import time

# Konfiguration des Wlans:
station = network.WLAN(network.STA_IF)
station.active(True)
station.connect('ssid', 'passwort') # hier Ihre Wlan-Daten eintragen!
time0 = time()
while (not station.isconnected()) and (time() - time0 < 5):
  pass
```

Bevor es weiter geht, prüfen wir nach, ob unser ESP32 mit dem Access Point verbunden ist. In diesem Fall geht es so weiter: Zunächst informieren wir den Benutzer darüber und lassen uns auch die IP-Konfiguration des Wlans angeben (vgl. Ende von Kapitel 3). Dann wird ein Socket `s` erzeugt; durch `s.bind(('', 80))` wird dieser Socket an den Port 80 gebunden. Das bedeutet: Alle weiteren Aktivitäten des Sockets beziehen sich auf diesen Port: Er kann nur an diesem Port Daten empfangen und auch nur über diesen Port Daten senden. Mit `s.listen(5)` sorgen wir dafür, dass sich das Socket auf die Lauer legt und darauf wartet, dass ein Client einen Verbindungsversuch startet. Wie genau dieser Verbindungsaufbau zustande kommt und welche Bedeutung die Zahl 5 hier hat, soll am Ende des Kapitels noch einmal genauer betrachtet werden.

```
if station.isconnected():
  print('Wlan connected')
  print(station.ifconfig()) # zur Information
  s = socket.socket()
  s.bind(('', 80))
  s.listen(5)
  print('Server listening')
  print('-------------------------------------')
```

Programmieren: Verbindung herstellen und HTML-Dokument senden

Mit `connection, addr = s.accept()` wartet unser Server auf eine Verbindung; dabei gibt `s.accept()` ein Tupel mit zwei Werten zurück: Zum einen ein neues Socket-Objekt `connection`, welches eine Verbindung des Servers zum Client bietet; zum Anderen ein Tupel `addr`, das aus der IP-Adresse und der Portnummer des Clients besteht. Letzteres lassen wir uns im Terminal anzeigen.

```
# Daten holen
while True:
  connection, addr = s.accept()
  print("Client connected: ", addr)
  request = connection.recv(1000)
  print(str(request, 'UTF-8'))
  html = '<html><body><h1>Hallo Welt</h1></body></html>'
  connection.send(html)
  connection.close()
  print('-------------------------------------')
```

Mit der Zeile `request = connection.recv(1000)` empfangen wir die Anfrage des Clients und lassen sie zur Kontrolle ebenfalls am Terminal ausgeben.

Die Zeichenkette `'<html><body><h1>Hallo Welt</h1></body></html>'` stellt ein sehr einfaches HTML-Dokument dar. HTML ist die Sprache, in der Webseiten beschrieben werden (mehr dazu in der HTML-Box weiter unten). In diesem Fall besagt es, dass auf der Webseite die beiden Worte “Hallo Welt” in Überschriftgröße (gekennzeichnet durch <h1> und </h1> stehen.

Genau diese Zeichenkette wird nun an den Client gesendet. Eigentlich sollten wir zuvor noch einen HTTP-Header an den Cient (Browser) senden. Der kommt hier aber auch ohne den Header aus und zeigt uns die Webseite korrekt an (vgl. Abb. 1). Abschließend wird der Socket `connection` wieder geschlossen.

Die Endlosschleife sorgt dafür, dass die Webseite unseres Servers beliebig oft aufgerufen werden kann. Der Aufruf kann auch von unterschiedlichen Clients aus erfolgen, weil für jeden Client eine eigene Verbindung aufgebaut wird.

Programm testen

Führen wir nun ein paar Experimente mit diesem Programm durch und schauen, was geschieht: Nach dem Start des Server-Programms erscheint auf dem Terminal zunächst die Meldung:

```
Wlan connected
('192.168.2.117', '255.255.255.0', '192.168.2.1', '192.168.2.1')
Server listening
-------------------------------------
```

Wie wir sehen, hat der ESP32 in meinem Wlan die IP-Adresse 192.168.2.117. Bei meinem ersten Test habe ich nun auf dem PC den Edge-Browser gestartet und dort genau diese IP-Adresse in der Adresszeile eingegeben. Jetzt ging alles sehr schnell: Zuerst erschien im Terminal

```
Client connected:  ('192.168.2.103', 49879)
```

Client und Server haben sich also verbunden; der Client hat die IP 192.168.2.103 mit der Portnummer 49879.

Fast gleichzeitig erschien auf dem Terminal der HTTP-Request, den der Server empfangen hat. Der Anfang davon sah so aus:

```
GET / HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-DE
```

Unser Server soll in jedem Fall immer nur "Hallo Welt" antworten. Deswegen reicht es aus, in die Adresszeile des Browsers lediglich die IP-Adresse zu schreiben. Der Browser fügt - wie wir bereits wissen - automatisch ein /-Zeichen an. Dieses Zeichen erscheint in dem obigen HTTP-Request als GET-Parameter.

Unmittelbar danach konnte ich beobachten, wie im Browser unsere Hallo-Welt-Webseite dargestellt wurde:

Abb. 1

Anschließend versuchte ich es mit dem Chrome-Browser auf meinem Android-Handy; im Terminal konnte ich jetzt feststellen, dass gleich zwei Requests beim Server eingegangen sind, zunächst:

```
Client connected:  ('192.168.2.170', 40309)
GET / HTTP/1.1
Host: 192.168.2.117
...
Accept: text/html...
```

und dann:

```
Client connected:  ('192.168.2.170', 40311)
GET /favicon.ico HTTP/1.1
Host: 192.168.2.117
...
Accept: image/webp
...
```

Der Chrome-Browser fragte hier beim zweiten Request nach der Datei mit dem Namen `favicon.ico`, einer Icon-Datei. Unser Server beachtete diesen Wunsch gar nicht und schickte einfach wieder die Hallo-Welt-Webseite. Ein intelligenteres Server-Programm hätte den HTTP-Request analysiert und dabei festgestellt, dass bei diesem Request ein spezielles Icon mit dem Namen `favicon.ico` angefordert wird (und auch nur Bild-Dateien akzeptiert werden). In seiner Blindheit ähnelt unser erstes Server-Programm dem bereits behandelten Daytime-Server. Übrigens zeigte die Ausgabe am Termina, dass der Chrome-Browser meines Handys sofort nach dem Erhalt der zweiten Hallo-Welt-Webseite automatisch für eine neue Verbindung sorgte.

Unser erstes Server-Programm wäre natürlich auch ohne die beiden Zeilen

```
request = connection.recv(1000)
print(str(request, 'UTF-8'))
```

ausgekommen; allerdings hätten wir dann nicht so viel über HTTP-Requests gelernt!

Beenden können wir das Server-Programm - wie jedes andere Programm - mit Ctrl-C. Beim Chrome-Browser meines Handys fand der dadurch ausgelöste Keyboard-Interrupt gerade dann statt, wenn der Server auf den Request wartete `(request = connection.recv(1000))`. Dies führte zu der Meldung:

```
Traceback (most recent call last):
  File "<stdin>", line 28, in <module>
KeyboardInterrupt:
```

Wichtig ist: Bevor das Server-Programm erneut gestartet wird, sollte das ESP32-Modul jedesmal mit dem RESET-Taster neu gebootet werden. Ansonsten gab es beim Neustart eine Fehlermeldung beim Binden des Sockets `s`; selbst ein Schließen des Sockets am Ende des Programms hat bei mir dieses Problem nicht beseitigen können. Ein Soft-Reboot reichte bei mir hier nicht aus!

Verbindungsaufbau unter der Lupe

Die folgende Abbildung 2 fast zusammen, was wir darüber gelernt haben, wie die Aktivitäten von Client und Server ineinander greifen. Zusätzlich wollen wir an dieser Stelle den Aufbau der Verbindung zwischen Client und Server noch etwas näher erläutern.

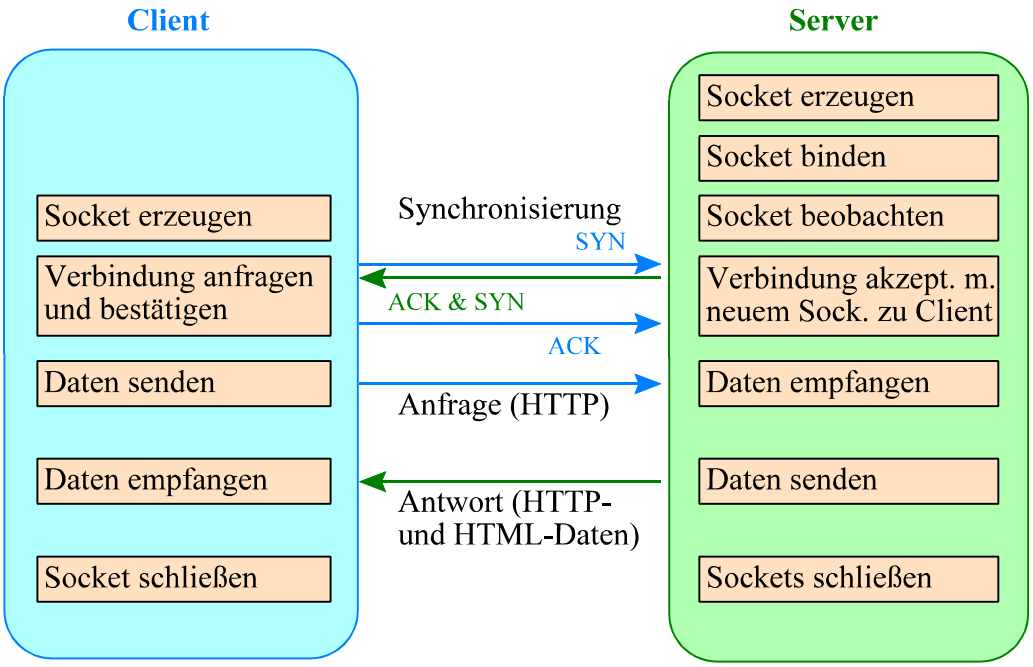

Abb. 2

In einem ersten Schritt sendet der Client ein SYN-Signal aus (SYN: Abkürzung von synchronize). Im zweiten Schritt reagiert der Server darauf, indem er dieses Signal mit einem ACK-Signal bestätigt und seinerseits ein SYN-Signal an den Client sendet (ACK: Abkürzung für acknowledge = Bestätigen). Dadurch ist der Client nun darüber informiert, dass der Server das SYN-Signal erhalten hat. In einem dritten Schritt bestätigt nun auch seinerseits der Client dem Server mit einem ACK-Signal, das Signal vom Server erhalten zu haben. Im Rahmen dieses dreistufigen Prozesses sind jetzt sowohl der Client als auch der Server sicher, dass die Verbindung etabliert ist. Nun kann der Client seinen Request absenden und der Server entsprechend antworten.

Es kann vorkommen, dass ein zweiter Client versucht, mit unserem Server in Kontakt zu treten, bevor der Request des ersten Clients abgearbeitet ist. In einem solchen Fall kommt die Verbindung des zweiten Clients in eine Warteschlange. Erst mit dem erneuten Aufruf von `s.accept()` wird diese zweit e Verbindung aus der Warteschlange geholt; damit kann dann der Request des zweiten Clients empfangen und bearbeitet werden. Wie viele Clients sich in der Warteschlange maximal befinden können, wird durch den Parameter bei der `listen`-Methode angegeben. Ein intelligenteres Programm könnte mehrere simultane Verbindungen akzeptieren und gleichzeitig die zugehörigen Requests verarbeiten.

Zum Ende dieses Kapitels noch die versprochenen Informationen zu HTML:

Ein Blick hinter die Kulissen: HTML

HTML steht für **H**yper**t**ext **M**arkup **L**anguage. Der Name spiegelt die Merkmale dieser Seitenbeschreibungssprache wieder: Zum einen ermöglicht sie die Verlinkung zu weiteren Dateien (Hypertext), zum Anderen gestattet sie es, einzelnen Textteilen eine Struktur zu geben, z. B. in Form von Schriftformatierungen oder auch Tabellen, Eingabe-Felder oder Schaltflächen. Diese Strukturgebung (Markup) erfolgt durch sogenannte **Tags**. So wird z. B. bei dem Text

```
Der Mikrocontroller <b>ESP32</b> bietet viele Möglichkeiten.
```

das Wort ESP32 im Browser fett (**b**old) ausgegeben. und werden Start- bzw. End-Tag genannt. Es gibt auch spezielle Tags, die kein End-Tag verwenden.

Eine HTML-Datei hat folgende Grundform:

```
<html>
  <head>
    <title>Titel der Webseite</title>
    Hier stehen weitere Meta-Daten zur Webseite
  </head>
  <body>
    Hier steht der Inhalt der Webseite
  </body>
</html>
```

Eine HTML-Datei ist eine Text-Datei. Entscheidende Strukturmerkmale sind:

Das HTML-Dokument steht innerhalb eines <html>-Paars. Dazwischen steht der Kopf, gekennzeichnet durch das <head>-Tag und der Körper, gekennzeichnet durch das <body>-Tag. Im Kopfteil können z. B. ein Webseitentitel, Programm-Skripte oder Meta-Daten angegeben werden. Der Kopfteil wird nicht im Browser-Fenster angezeigt; moderne Browser kommen auch ohne Kopfteil zurecht.

Im Körper steht, was im Browserfenster angezeigt werden soll. Innerhalb des Körpers tauchen i. A. weitere Tag-Paare auf, welche die Struktur festlegen (s. o.). Wichtig ist, dass diese Tag-Paare korrekt geschachtelt werden. Damit man hier nicht die Übersicht verliert, ist es sinnvoll, den HTML-Code wie oben dargestellt blockmäßig einzurücken. Für den Browser selbst ist dieses Einrücken irrelevant: Zeilenwechsel im Quellcode ignoriert er; ein Zeilenwechsel erfolgt bei der Anzeige im Browser-Fenster nur, wenn der laufende Text den rechten Fensterrand erreicht oder entsprechende Formatierungs-Tags vorliegen.