HTTP und Sockets in Python

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de
Version: 1.0

Folien: [HTML] https://delors.github.io/ds-http-und-sockets-python/folien.de.rst.html
[PDF] https://delors.github.io/ds-http-und-sockets-python/folien.de.rst.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

Dieser Foliensatz basiert lose auf Folien von Prof. Dr. Henning Pagnia.

Alle Fehler sind meine eigenen.

IP

Die Vermittlungsschicht (Internet Layer)

- übernimmt das Routing
- realisiert Ende-zu-Ende-Kommunikation
- überträgt Pakete
- ist im Internet durch IP realisiert
- löst folgende Probleme:
  - Sender und Empfänger erhalten netzweit eindeutige Bezeichner (⇒ IP-Adressen)
  - die Pakete werden durch spezielle Geräte (⇒ Router) weitergeleitet

TCP und UDP

Transmission Control Protocol (TCP), RFC 793

- verbindungsorientierte Kommunikation
- ebenfalls Konzept der Ports
- Verbindungsaufbau zwischen zwei Prozessen (Dreifacher Handshake, Full-Duplex-Kommunikation)
  - geordnete Kommunikation
  - zuverlässige Kommunikation
  - Flusskontrolle
  - hoher Overhead ⇒ eher langsam
  - nur Unicasts

User Datagram Protocol (UDP), RFC 768

- verbindungslose Kommunikation
  - unzuverlässig ⇒ keine Fehlerkontrolle
  - ungeordnet ⇒ beliebige Reihenfolge
  - wenig Overhead ⇒ schnell
- Größe der Nutzdaten ist 65507 Byte
- Anwendungsfelder:
  - Anw. mit vorwiegend kurzen Nachrichten (z. B. NTP, RPC, NIS)
  - Anw. mit hohem Durchsatz, die gel. Fehler tolerieren (z. B. Multimedia)
  - Multicasts sowie Broadcasts

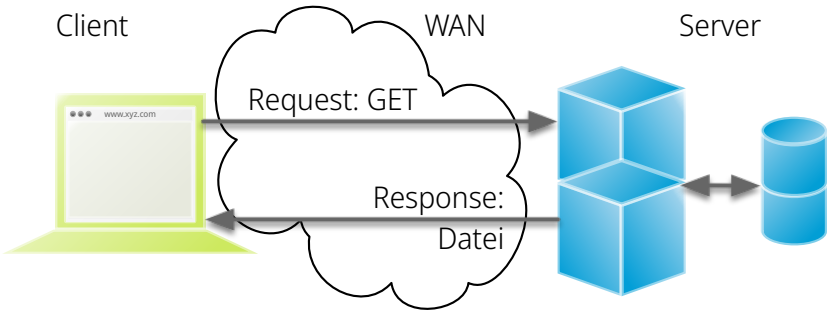
UDP nutzt für die Kommunikation "Datagramme" (Pakete). In der Praxis sin die Nutzdaten meist deutlich kleiner und orientieren sich an der Größe für IP-Pakete.

1. Hypertext Transfer Protocol (HTTP)

HTTP

- RFC 7230 – 7235: HTTP/1.1 (redigiert im Jahr 2014; urspr. 1999 RFC 2626)
- RFC 7540: HTTP/2 (seit Mai 2015 standardisiert)
- Eigenschaften:
  - Client / Server (Browser / Web-Server)
  - basierend auf TCP, i. d. R. Port 80
  - Server (meist) zustandslos
  - seit HTTP/1.1 auch persistente Verbindungen und Pipelining
  - abgesicherte Übertragung (Verschlüsselung) möglich mittels Secure Socket Layer (SSL) bzw. Transport Layer Security (TLS)

Konzeptioneller Ablauf



HTTP-Kommandos

(„Verben“)

- HEAD
- GET
- POST
- PUT
- PATCH
- DELETE
- OPTIONS
- TRACE
- CONNECT
- ...

Protokolldefinition

Aufbau der Dokumentenbezeichner *Uniform Resource Locator (URL)*

```
scheme://host[:port][abs_path[?query][#anchor]]
```

`scheme`:	Protokoll (case-insensitive) (z. B. `http`, `https` oder `ftp`)
`host`:	DNS-Name (oder IP-Adresse) des Servers (case-insensitive)
`port`:	(optional) falls leer, 80 bei `http` und 443 bei `https`
`abs_path`:	(optional) Pfadausdruck relativ zum Server-Root (case-sensitive)
`?query`:	(optional) direkte Parameterübergabe (case-sensitive) (`?from=…&to=…`)
`#anchor`:	(optional) Sprungmarke innerhalb des Dokuments

Uniform Resource Identifier (URI) sind eine Verallgemeinerung von URLs.

- definiert in RFC 1630 (im Jahr 1994)
- entweder URL (Location) oder URN (Name) (z. B. `urn:isbn:1234567890`)
- Beispiele von URIs, die keine URL sind, sind *XML Namespace Iidentifiers*

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">...</svg>
```

Das GET Kommando

- Dient dem Anfordern von HTML-Daten vom Server (Request-Methode).
- Minimale Anfrage:

Anfrage:

```
GET <Path> HTTP/1.1
Host: <Hostname>
Connection: close
<Leerzeile (CRLF)>
```

Optionen:

- Client kann zusätzlich weitere Infos über die Anfrage sowie sich selbst senden.
- Server sendet Status der Anfrage sowie Infos über sich selbst und ggf. die angeforderte HTML-Datei.

- Fehlermeldungen werden ggf. vom Server ebenfalls als HTML-Daten verpackt und als Antwort gesendet.

Beispiel Anfrage des Clients

```
GET /web/web.php HTTP/1.1
Host: archive.org
**CRLF**
```

Beispiel Antwort des Servers

```
HTTP/1.1 200 OK
Server: nginx/1.25.1
Date: Thu, 22 Feb 2024 19:47:11 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: close
**CRLF**
<!DOCTYPE html>
…
</html>**CRLF**
```

2. Sockets

Sockets in Python

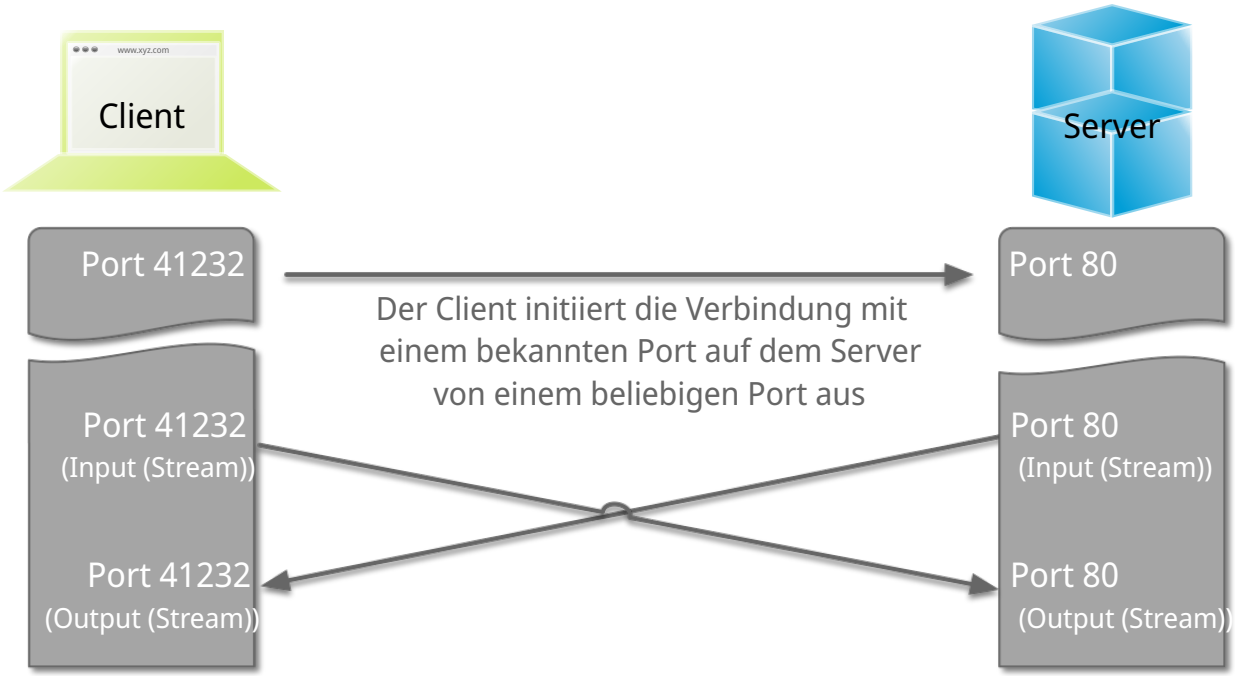
Sockets sind Kommunikationsendpunkte.

- Sockets werden adressiert über die IP-Adresse (InetAddress-Objekt) und eine interne Port-Nummer (int-Wert).
- Sockets gibt es bei TCP und auch bei UDP, allerdings mit unterschiedlichen Eigenschaften:

TCP:	verbindungsorientierte Kommunikation über *Streams*
UDP:	verbindungslose Kommunikation mittels *Datagrams*

- Das Empfangen von Daten ist in jedem Fall blockierend, d. h. der empfangende Thread bzw. Prozess wartet, falls keine Daten vorliegen.

TCP Sockets



---

1. Der Server-Prozess wartet an dem bekannten Server-Port.
2. Der Client-Prozess erzeugt einen privaten Socket.
3. Der Socket baut zum Server-Prozess eine Verbindung auf – falls der Server die Verbindung akzeptiert.
4. Die Kommunikation erfolgt Strom-orientiert: Für beide Parteien wird je ein Eingabestrom und ein Ausgabestrom eingerichtet, über den nun Daten ausgetauscht werden können.
5. Wenn alle Daten ausgetauscht wurden, schließen im Allg. beide Parteien die Verbindung.

(Ein einfacher) Portscanner

```python
import sys
import socket

def scan_port(host, port):
  try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.settimeout(0.5)  # Set a timeout to avoid hanging connections
      s.connect((host, port))
      print(f"Port {port} is open on {host}")
  except (ConnectionRefusedError, TimeoutError) as e:
    pass  # Port is likely closed, expected behavior

def main():
  host = "localhost"
  if len(sys.argv) > 1: host = sys.argv[1]
  for port in range(1, 1024): scan_port(host, port)

if __name__ == "__main__": main()
```

Austausch von Daten

- Nach erfolgtem Verbindungsaufbau können zwischen Client und Server mittels `sendall` und `recv` Daten ausgetauscht werden.
- Wir können blockierend auf Daten warten bzw. blockierend schreiben, indem wir `recv` bzw. `sendall` aufrufen. (Siehe nächstes Beispiel.)

  Sollte die Verbindung abbrechen oder die Gegenseite nicht antworten, kann es „relativ lange dauern“, bis dieser Fehler erkannt bzw. gemeldet wird.
- Wir können den Socket auch in den nicht-blockierenden Modus versetzen, indem wir `setblocking(False)` aufrufen (ggf. sinnvoll).

Ein einfacher Echo-Dienst

```python
# Client
import socket
def receive_all(conn, chunk_size=1024):
    data = b''
    while True:
        part = conn.recv(chunk_size)
        data += part
        if len(part) == 0: break # no more data
    return data

while True:
    the_line = input()
    if the_line == ".": break
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(("localhost", 5678))  # Connect to localhost on port 5678
        s.sendall(the_line.encode())
        data = receive_all(s)
    print(data.decode())
```

```python
# Server
import socket
def receive_all(conn, chunk_size=1024): # see previous example

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
  server.bind(("localhost", 5678)) # Bind to localhost on port 5678
  server.listen(1) # queue at most one connection at a time
  while True:
      conn, addr = server.accept()
      with conn:
          print(f"Connection from {addr}.")
          data = receive_all(conn, 1024)
          print(f"Received {data}.")
          if data:
              conn.sendall(data)
```

- Python erlaubt es Sockets zu Wrappen, um sie wie Dateien behandeln zu können.

  `<Socket>.makefile(mode="r?w?b?" [, encoding="utf-8"])` erzeugt ein Dateiobjekt, das (insbesondere) `readline()` und `write()` unterstützt. Dies kann insbesondere bei zeilenorientierter Kommunikation hilfreich sein.
- Es können auch ganze Dateien über Sockets basierend übertragen werden (`<Socket>.sendfile(<File>)`).

Warnung

Einige Methoden sind nur auf spezifischen Betriebssystemen (meist Unix) verfügbar.

UDP Sockets

Clientseitig

1. *Datagram-Socket* erzeugen und an Zieladresse binden
2. Nachricht erzeugen (ggf. vorher maximale Länge prüfen)
3. *Datagram* absenden
4. ggf. Antwort empfangen und verarbeiten

Serverseitig

1. *Datagram-Socket* auf festem Port erzeugen

   (Die Hostangabe bestimmt wer sich mit dem Socket verbinden darf; `localhost` bedeutet nur lokale Verbindungen sind erlaubt.)
2. Endlosschleife beginnen
3. *Datagram* empfangen (und verarbeiten)
4. ggf. Antwort erstellen und absenden

UDP basierter Echo Server

```python
import socket

HOST = "localhost"
PORT = 5678

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as server:
    server.bind((HOST, PORT))

    while True:
        data, addr = server.recvfrom(65507)  # buffer size is 65507 bytes
        print(f"received message: {data} from: {addr}")
        server.sendto(data, addr)
```

Übung

2.1. Ein einfacher HTTP-Client

a. Schreiben Sie einen HTTP-Client, der den Server `www.michael-eichberg.de` kontaktiert, die Datei `/index.html` anfordert und die Antwort des Servers auf dem Bildschirm ausgibt.

   Verwenden Sie HTTP/1.1 und eine Struktur ähnlich dem in der Vorlesung vorgestellten Echo-Client.

   Senden Sie das GET-Kommando, die Host-Zeile sowie eine Leerzeile als Strings an den Server.

b. Erweitern Sie Ihren Client um die Fähigkeit URLs auf der Kommandozeile anzugeben.

   Verwenden Sie existierende Funktionalität, um die angegebene URL zu zerlegen (`urlparse von urllib.parse`).

c. Speichern Sie die Antwort des Servers in einer lokalen Datei. Prüfen Sie, dass die Datei in einem Browser korrekt angezeigt wird.

   Kann Ihr Programm auch Bilddateien (z. B. "http://www.michael-eichberg.de/acm.svg") korrekt speichern? Falls nicht, prüfen Sie ob Sie Antwort des Servers richtig verarbeiten; analysieren Sie ggf. den Header und passen Sie Ihr Programm entsprechend an.

Übung

2.2. Protokollaggregation

Schreiben Sie einen Python basierten Server und Client, mit dem sich Protokoll-Meldungen auf einem Server zentral anzeigen lassen. Das Programm soll mehrere Clients unterstützen und UDP verwenden. Jeder Client liest von der Tastatur eine Eingabezeile in Form eines Strings ein, validiert die Eingabe und sendet diese dann ggf. sofort zum Server. Der Server wartet auf Port 5678 und empfängt die Meldungen beliebiger Clients, die er dann unmittelbar ausgibt.

Stellen Sie sicher, dass Fehler adäquat behandelt werden.

Sie können den UDP basierten Echo Server als Vorlage für Ihren Server verwenden.