HTTP und Sockets (in Java)

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw-mannheim.de
Version: 2024-02-26
Folien: https://delors.github.io/ds-http-and-sockets/folien.rst.html
https://delors.github.io/ds-http-and-sockets/folien.rst.html.pdf

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

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

Alle Fehler sind meine eigenen.

IP - Wiederholung

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 - Wiederholung

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 via Datagramme
  - unzuverlässig (⇒ keine Fehlerkontrolle)
  - ungeordnet (⇒ beliebige Reihenfolge)
  - wenig Overhead (⇒ schnell)
- Größe der Nutzdaten ist 65507 Byte, in der Praxis jedoch meist deutlich kleiner
- Anwendungsfelder:
  - Anwendungen mit vorwiegend kurzen Nachrichten (z.B. NTP, RPC, NIS)
  - Anwendungen mit hohem Durchsatz, die ab und zu Fehler tolerieren (z.B. Multimedia)
  - Multicasts sowie Broadcasts

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

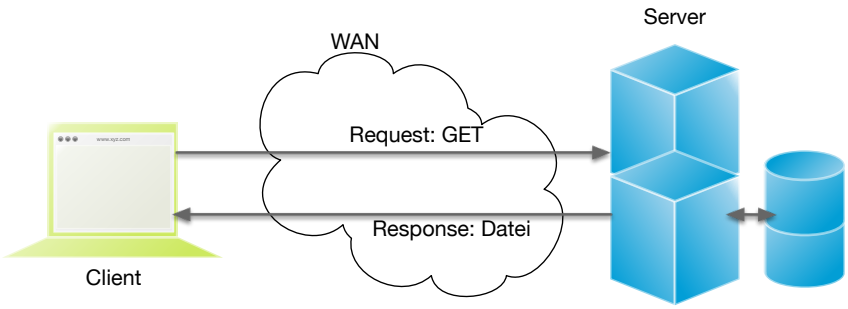
1. Hypertext Transfer Protocol (HTTP)

Prof. Dr. Michael Eichberg

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.

1

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

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

2. Sockets

Prof. Dr. Michael Eichberg

Sockets in Java

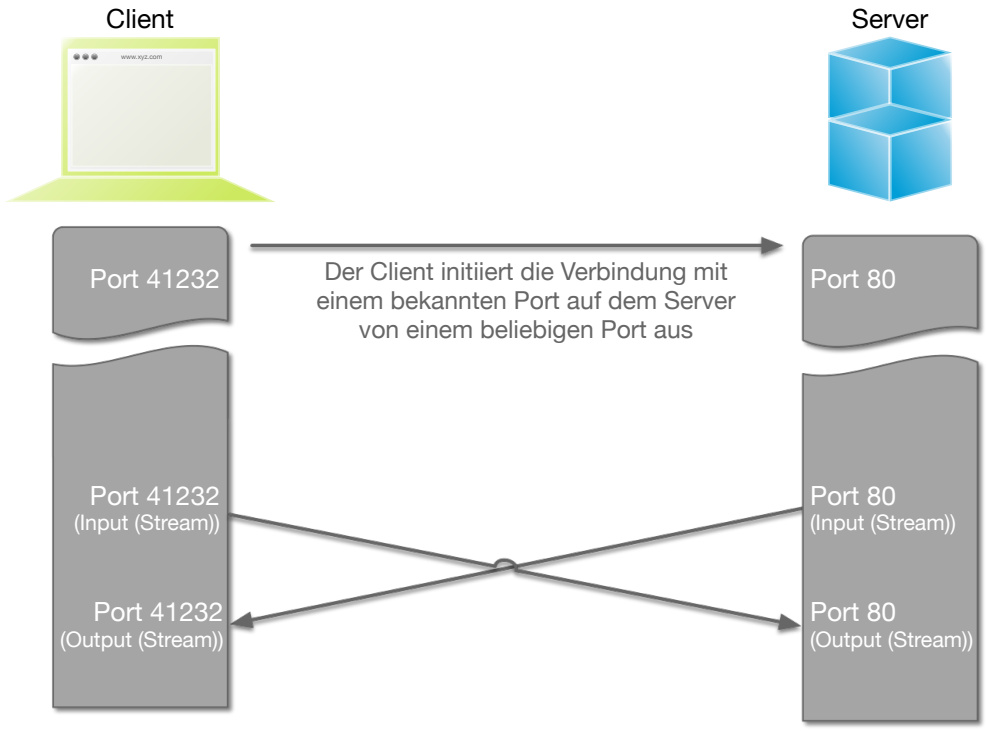
Sockets sind Kommunikationsendpunkte.

- 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 in Java

```
import java.net.*;
import java.io.*;

public class LowPortScanner {
  public static void main(String [] args) {
    String host = "localhost";
    if (args.length > 0) { host = args [0]; }
    for (int i = 1; i < 1024; i++) {
      try {
        Socket s = new Socket(host, i);
        System.out.println ("There is a server on port "+ i + "at "+host);
        s.close();
      } catch (UnknownHostException e) {
        System.err.println(e);
        break ;
      }
      catch (IOException e) {/* probably no server waiting at this port */ }
} } }
```

Austausch von Daten

- Nach erfolgtem Verbindungsaufbau können zwischen Client und Server mittels des Socket-InputStream und Socket-OutputStream Daten ausgetauscht werden.
- Hierzu leitet man die rohen Daten am besten durch geeignete Filter-Streams, um eine möglichst hohe semantische Ebene zu erreichen.
  - Beispiele: `PrintWriter`, `BufferedReader`, `BufferedInputStream`, `BufferedOutputStream`
  - Die Netzwerkkommunikation kann dann ggf. bequem über wohlbekannte und komfortable Ein- und Ausgabe-Routinen (z. B. `readLine` oder `println`) durchgeführt werden.
  - Filter-Streams werden auch für den Zugriff auf andere Geräte und Dateien verwendet.

Durch die Verwendung des *Decorater-Patterns* können die Filter-Streams beliebig geschachtelt werden und vielfältig verwendet werden. Dies macht die Anwendungsprogrammierung einfacher und erlaubt zum Beispiel das einfache Umwandeln von Zeichenketten, Datenkomprimierung, Verschlüsselung, usw.

(Schachtelung von Streams) Ein einfacher Echo-Dienst

```java
import java.net.*;
import java.io.*;

public class EchoClient {
  public static void main(String[] args) throws IOException {
    BufferedReader userIn = new BufferedReader(new InputStreamReader(System.in));
    while (true) {
      String theLine = userIn.readLine();
      if (theLine.equals(".")) break;
      try (Socket s = new Socket("localhost"/*hostname*/, 7/*serverPort*/)) {
        BufferedReader networkIn =
            new BufferedReader(new InputStreamReader(s.getInputStream()));
        PrintWriter networkOut = new PrintWriter(s.getOutputStream());
        networkOut.println(theLine);
        networkOut.flush();
        System.out.println(networkIn.readLine());
      }
} } }
```

1

```java
import java.net.*;
import java.io.*;


public class EchoServer {
  public static void main(String[] args) {
    BufferedReader in = null ;
    try {
      ServerSocket server = new ServerSocket(7 /*DEFAULT PORT*/);
      while (true) {
        try (Socket con = server.accept()) {
          in = new BufferedReader(new InputStreamReader(con.getInputStream()));
          PrintWriter out = new PrintWriter(con.getOutputStream());
          out.println(in.readLine()) ;
          out.flush() ;
        } catch (IOException e) { System.err.println(e); }
      }
    } catch (IOException e) { System.err.println(e); }
} }
```

2

UPD Sockets

Clientseitig

1. `DatagramSocket` erzeugen
2. `DatagramPacket` erzeugen
3. `DatagramPacket` absenden
4. ggf. Antwort empfangen und verarbeiten

Serverseitig

1. `DatagramSocket` auf festem Port erzeugen
2. Endlosschleife beginnen
3. `DatagramPacket` vorbereiten
4. `DatagramPacket` empfangen
5. `DatagramPacket` verarbeiten
6. ggf. Antwort erstellen und absenden

UDP basierter Echo Server

```
import java.net.*;
import java.io.*;

public class UDPEchoServer {
  public final static int DEFAULT_PORT = 7; // privileged port
  public static void main(String[] args) {
    try (DatagramSocket server = new DatagramSocket(DEFAULT_PORT)) {
      while(true) {
        try {
          byte[] buffer = new byte[65507]; // room for incoming message
          DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
          server.receive(dp) ;
          String data = new String(dp.getData(),0,dp.getLength());
          DatagramPacket dp2 =
            new DatagramPacket(data.getBytes(),
              data.getBytes().length, dp.getAddress(), dp.getPort());
          server.send(dp2) ;
        } catch (IOException e) {System.err.println(e);}
    } }
} }
```

Übung - Ein einfacher HTTP-Client

a. Schreiben Sie einen HTTP-Client der den Server `archive.org` kontaktiert, die Datei `/web/web.php` 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. Modifizieren Sie Ihren Client, so dass eine URL als Kommandozeilenparameter akzeptiert wird.

   Verwenden Sie die (existierende) Klasse URL, um die angegebene URL zu zerlegen.

c. Modifizieren Sie Ihr Programm, so dass die Antwort des Servers als lokale Datei abgespeichert wird. Laden Sie die Datei zum Anzeigen in einen Browser.

   Nutzen Sie die Klasse `FileOutputStream` oder `FileWriter` zum Speichern der Datei.

   Kann Ihr Programm auch Bilddateien (z. B. "/images/logo_wayback_210x77.png") korrekt speichern?

Übung - Protokollaggregation

Schreiben Sie ein UDP-basiertes Java-Programm mit dem sich Protokoll-Meldungen auf einem Server zentral anzeigen lassen. Das Programm soll aus mehreren Clients und einem Server bestehen. Jeder Client liest von der Tastatur eine Eingabezeile in Form eines Strings ein, der dann sofort zum Server gesendet wird. Der Server wartet auf Port 4999 und empfängt die Meldungen beliebiger Clients, die er dann unmittelbar auf den Bildschirm ausgibt.