HTTP and Sockets (in Java)

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

Lecture Material: [HTML] https://delors.github.io/ds-http-and-sockets-java/folien.en.rst.html
[PDF] https://delors.github.io/ds-http-and-sockets-java/folien.en.rst.html.pdf
Reporting Errors: https://github.com/Delors/delors.github.io/issues

This set of slides is based on slides by Prof Dr Henning Pagnia.

All errors are my own.

IP

The network layer (Internet layer)

- handles the routing
- realizes end-to-end communication
- transmits packets
- is realized in the Internet through IP
- solves the following problems:
  - Sender and receiver receive network-wide unique identifiers (⇒ IP addresses)
  - the packets are forwarded by special devices (⇒ routers)

TCP and UDP

Transmission Control Protocol (TCP), RFC 793

- connection-orientated communication
- also the concept of ports
- Establishing a connection between two processes (triple handshake, full-duplex communication)
  - Ordered communication
  - reliable communication
  - Flow control
  - high overhead ⇒ rather slow
  - only unicasts

User Datagram Protocol (UDP), RFC 768

- connectionless communication
  - unreliable (⇒ no error control)
  - unordered (⇒ arbitrary order)
  - little overhead (⇒ fast)
- Size of the user data is 65507 bytes
  - Apps with predominantly short messages (e.g. NTP, RPC, NIS)
  - Apps with high throughput that tolerate some errors (e.g. multimedia)
  - Multicasts and broadcasts

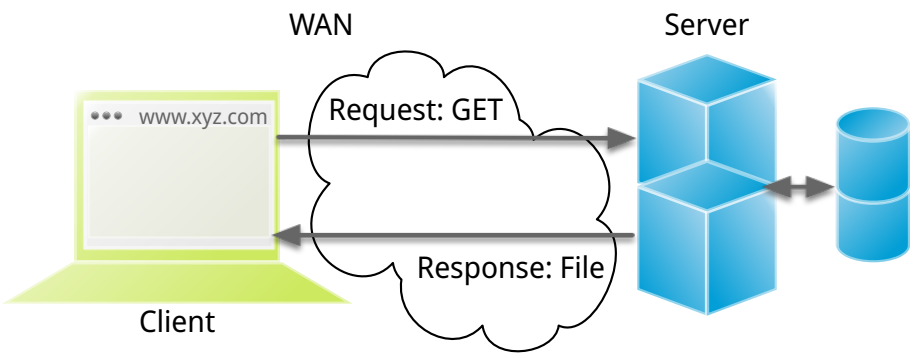
In practice datagrams (i. e. packages sent using UDP) are usually much smaller than 65507 bytes.

1. Hypertext Transfer Protocol (HTTP)

HTTP

- RFC 7230 – 7235: HTTP/1.1 (updated in 2014; orig. 1999 RFC 2626)
- RFC 7540: HTTP/2 (standardized since May 2015)
- Properties:
  - Client / server (browser / web server)
  - based on TCP, usually port 80
  - Server (mostly) stateless
  - since HTTP/1.1 also persistent connections and pipelining
  - Secure transmission (encryption) possible using Secure Socket Layer (SSL) or Transport Layer Security (TLS)

Conceptual process



HTTP-Kommandos

("Verbs")

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

Protocol definition

Structure of document identifiers *Uniform Resource Locator (URL)*

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

`scheme`:	Protocol (case-insensitive) (z. B. `http`, `https` oder `ftp`)
`host`:	DNS-Name (or IP-address) of the server (case-insensitive)
`port`:	(optional) if empty, 80 in case of `http` and 443 in case of `https`
`abs_path`:	(optional) path-expression relative to the server-root (case-sensitive)
`?query`:	(optional) direct parameter transfer (case-sensitive) (`?from=…&to=…`)
`#anchor`:	(optional) jump label within the document

Uniform Resource Identifier (URI) are a generalization URLs.

- defined in RFC 1630 in 1994
- either a URL (location) or a URN (name) (e. g. `urn:isbn:1234567890`)
- examples of URIs that are not URLs are *XML Namespace Iidentifiers*

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

Quite frequently URIs take the shape of URLs and hence are often referred to as URLs thought they do not primarily identify locations but rather names.

The GET command

- Used to request HTML data from the server (request method).
- Minimal request:

Request:

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

Options:

- Clients can also send additional information about the request and itself.
- Servers send the status of the request as well as information about itself and, if applicable, the requested HTML file.

- Error messages may also be packaged by the server as HTML data and sent as a response.

Example request

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

Example response

```
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 Java

Sockets are communication endpoints.

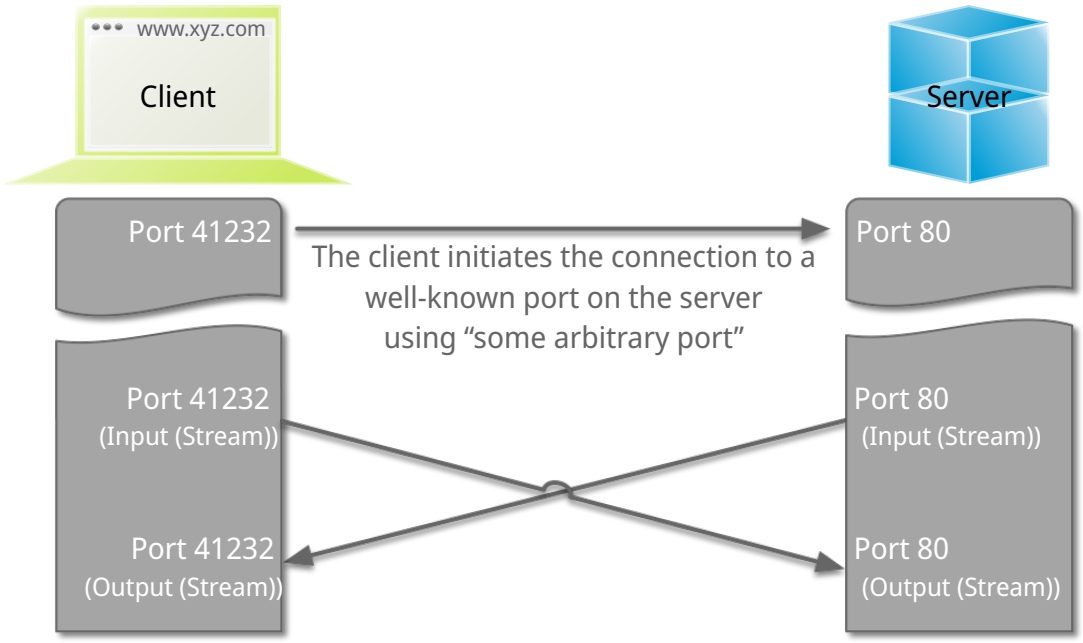
- Sockets are addressed via the IP address (InetAddress object) and an internal port number (int value).
- Sockets exist for TCP and also for UDP, but with different properties:

  TCP: connection-orientated communication via *streams*

  UDP: connectionless communication via *datagrams*

- Receiving data is always blocking, i. e. the receiving thread or process waits if no data is available.

TCP Sockets



1. The server process waits at the known server port.
2. The client process creates a private socket.
3. The socket establishes a connection to the server process - if the server accepts the connection.
4. Communication is stream-orientated: An input stream and an output stream are set up for both parties, via which data can now be exchanged.
5. When all data has been exchanged, both parties generally close the connection.

(A simple) 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 */ }
} } }
```

Exchange of Data

- Once the connection has been established, data can be exchanged between the client and server using the Socket-InputStream and Socket-OutputStream.
- The best way to do this is to pass the raw data through suitable filter streams in order to achieve the highest possible semantic level.
  - Examples: `PrintWriter`, `BufferedReader`, `BufferedInputStream`, `BufferedOutputStream`
  - Network communication can then be conveniently carried out via well-known and convenient input and output routines (e.g. `readLine` or `println`).
  - Filter streams are also used to access other devices and files.

By using the *decorater pattern*, the filter streams can be nested as required and used in a variety of ways. This makes application programming easier and allows, for example, the simple conversion of character strings, data compression, encryption, etc.

(Nesting of streams) A simple Echo service

```
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());
} } } }
```

```
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); }
} }
```

UDP Sockets

At the client side

1. create `DatagramSocket`
2. create `DatagramPacket`
3. send `DatagramPacket`
4. wait for response and process it, if needed

At the server side

1. create `DatagramSocket` with a fixed port
2. stard endless loop
3. prepare `DatagramPacket`
4. receive `DatagramPacket`
5. process `DatagramPacket`
6. create and send response if needed

UDP based Echo Server

```
1  import java.net.*; import java.io.*;
2
3  public class UDPEchoServer {
4    public final static int DEFAULT_PORT = 7; // privileged port
5    public static void main(String[] args) {
6      try (DatagramSocket server = new DatagramSocket(DEFAULT_PORT)) {
7        while(true) {
8          try {
9            byte[] buffer = new byte[65507]; // room for incoming message
10           DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
11           server.receive(dp) ;
12           String data = new String(dp.getData(),0,dp.getLength());
13           DatagramPacket dp2 =
14             new DatagramPacket(data.getBytes(),
15               data.getBytes().length, dp.getAddress(), dp.getPort());
16           server.send(dp2) ;
17         } catch (IOException e) {System.err.println(e);}
18 } } } }
```

Exercise

2.1. A simple HTTP-Client

a. Write an HTTP client that contacts the server `www.michael-eichberg.de`, requests the file `/index.html` and displays the server response on the screen.

   Use HTTP/1.1 and a structure similar to the echo client presented in the lecture.

   Send the GET command, the host line and an empty line to the server as strings.

b. Modify your client so that a URL is accepted as a command line parameter.

   Use the (existing) class URL to decompose the specified URL.

c. Modify your program so that the response from the server is saved in a local file. Load the file into a browser for display.

   Use the class `FileOutputStream` or `FileWriter` to save the file.

   Can your programme also save image files (e.g. '/exercises/star.jpg') correctly?

Exercise

2.2. Log Aggregation

Write a UDP-based Java program with which log messages can be displayed centrally on a server. The program should consist of several clients and a server. Each client reads an input line from the keyboard in the form of a string, which is then immediately sent to the server. The server waits on port 4999 and receives the messages from any client, which it then outputs immediately.