

# Client- Server- Programmierung

Client Hello

# Vorgehen & Ziele

- Theorie
  - Kursliteratur ([java] Kap. 20)
  - Literaturangaben auf Folien
- Übungen
  - online Tutorials Selbststudium
  - Hands-On im Unterricht
- Heute:
  - Grundlegende Konzepte
    - Begriffe
    - Client & Server Model
  - TCP Client & Java Server Übung
  - HTTP Protokoll Einführung
  - HTTP Client Übung
- Wo 2
  - Server Architekturen
  - Serialisierung
  - Java Server Ausbau

# Grundlegende Konzepte

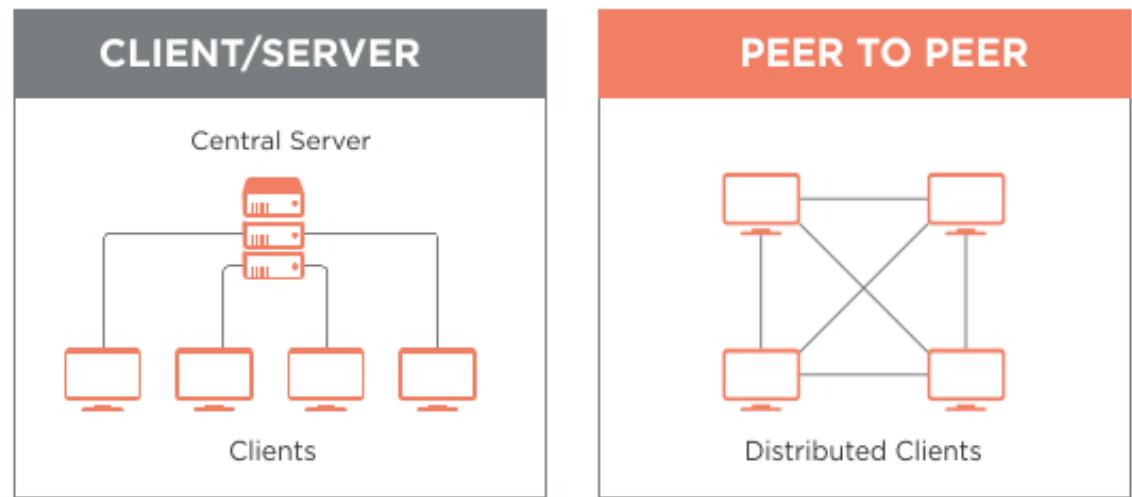
Begriffe & Modelle

# Netzwerkprogrammierung

- Oder: wie Informationen Distanz überwinden
- Anwendungsgebiete
  - Surfen im Internet
  - Temperatur Sensor
  - GPS
  - TV
  - Anlagensteuerung via Ethernet

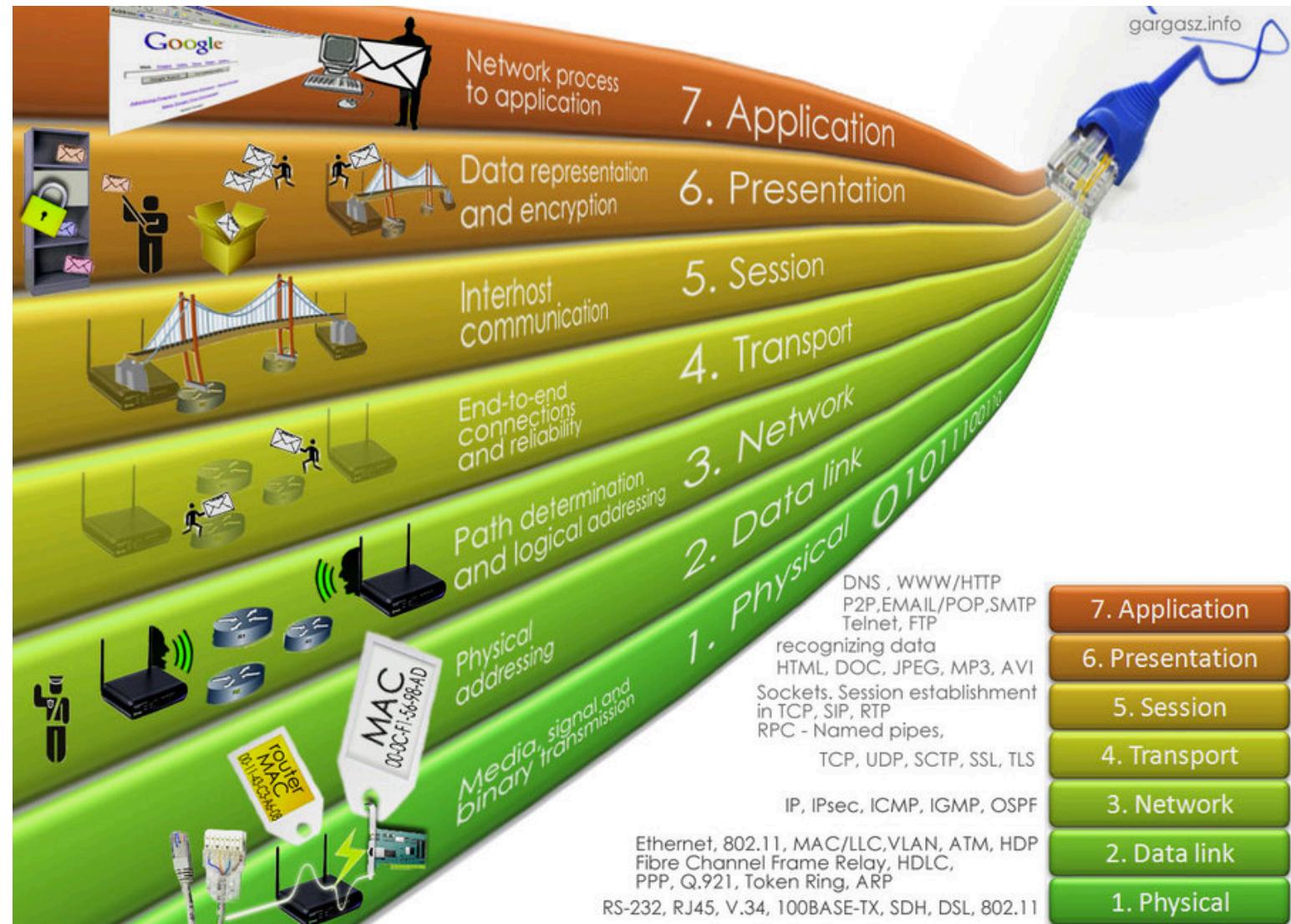
# Netzwerk Topologien

- Logischer Aufbau der Netzwerk-knoten und Verbindungen
- Client & Server
  - Client ist “dumm”
  - Server ist der Wissende
  - Client <==> Server
  - Client <!=> Client
  - Single Point of Failure: Server
- Peer to Peer
  - Alle sind Client und Server gleichzeitig
  - Client <==> Client
  - Ausfallsicherer da kein SPOF



# Protokolle

- Definiert nach welchen Regeln kommuniziert wird (Formate, Befehle, Abläufe)
- Ermöglicht Komm. über versch. Platformen & Programmiersprachen
- Schichtenmodell
  - OSI-Standard erstellt von ISO
  - Zwiebelprinzip
- Vereinfachung mit 4 Layern (TCP/IP Model)
  - Application (7, 6)
  - Transport (5,6)
  - Network (3,2)
  - Physical (1)

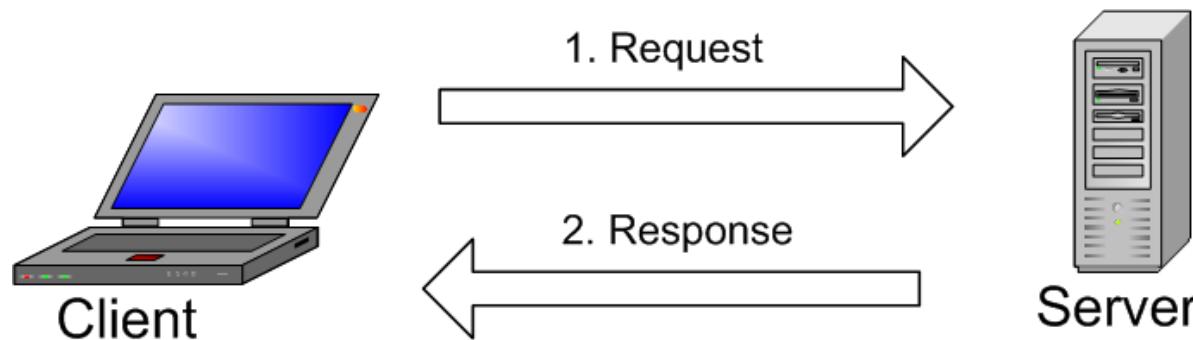


# Verbindungsorientierung

- Verbindungsorientiert
  - Zwei Netzwerknoten müssen eine Verbindung hergestellt haben, bevor Daten ausgetauscht werden.
- Beispiel TCP (Transmission Control Prot)
  - Nur Unicast Nachrichten
  - Liefersicherheit, Fehlerbehandlung, Reihenfolge
  - Session / Zustand
- **Nicht** Verbindungsorientiert
  - Jeder Netzwerknoten, kann jederzeit Daten schicken
  - An mehrere oder eine Ziel-Adresse
- Beispiel UDP (User Datagram Prot)
  - Broadcast Nachricht mit Empfänger 192.168.1.0 geht an alle Netzwerknoten im Subnetz 1.\*
  - Unicast Nachricht mit Empfänger 192.168.1.23 geht nur an den einen Empfänger

# Client & Server

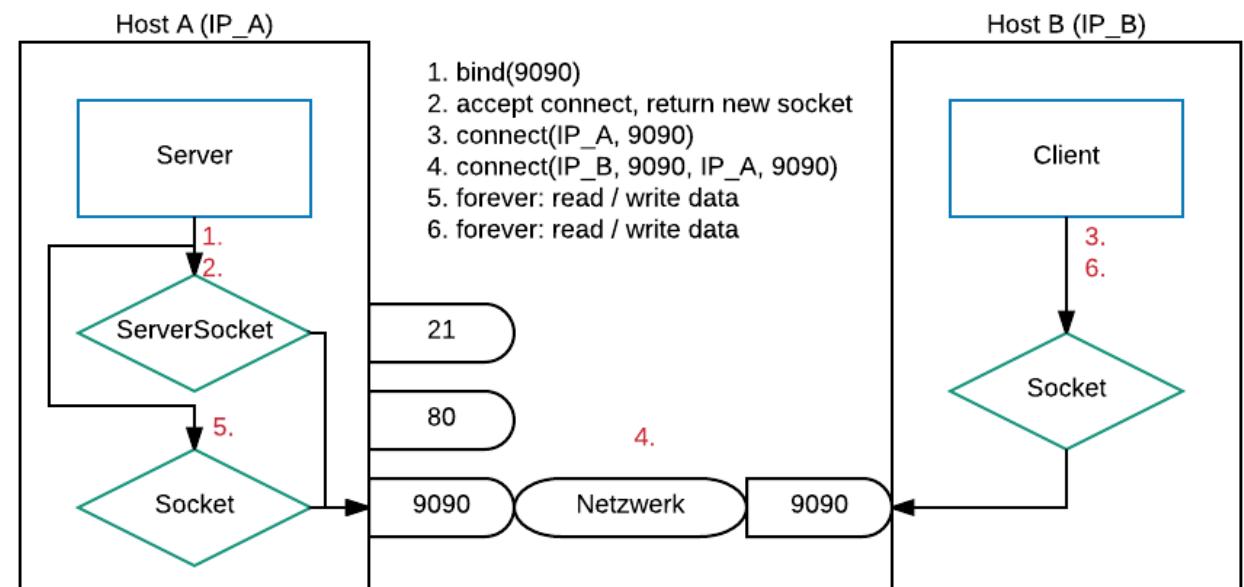
- Client:
  - Will etwas vom Server
  - Stellt Anfrage
- Server:
  - Hat etwas für Client
  - Erstellt passende Antwort



# Client & Server – Komponenten & Ablauf

## Begriffe:

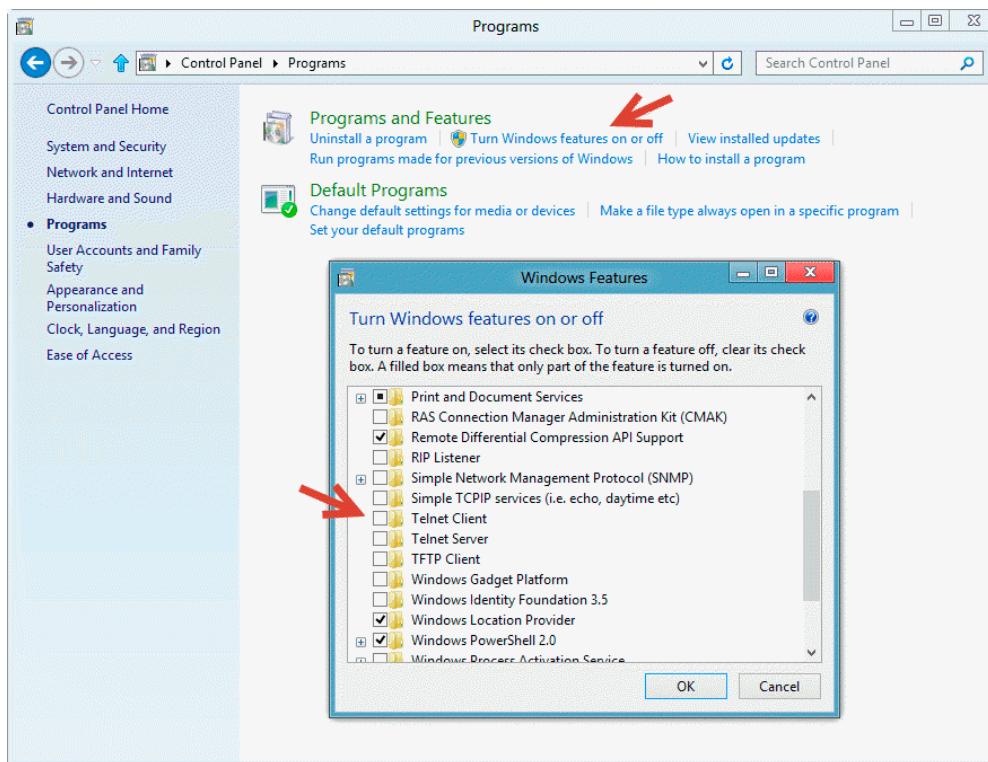
- Server  
Prozesss, welcher über das Netzwerk erreichbar ist
- Client  
Prozess, welcher Verbindung mit einem Server initiiert
- Host  
Teilnehmer im Netzwerk
- IP Address  
Eindeutig pro Netzwerkteilnehmer (Host). z.B.  
192.168.1.12
- Port  
Eine Nummer zwischen 0 und 65535, welche zusammen  
mit der IP, einen Server eindeutig identifiziert.
- Socket  
Schnittstellen, welche vom Betriebssystem verwaltet  
werden, und vom Prozess verwendet werden, um über  
das Netzwerk zu senden / empfangen.



# Übung

TCP, Java Zeitserver mit native TCP Client

# TCP Client telnet – Windows Setup

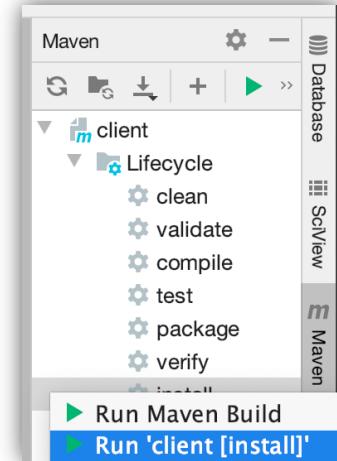


<https://www.labnol.org/software/windows-telnet-command/18222/>

Telnet nach der Übung unbedingt wieder deaktivieren!

# Java Zeitserver mit Client (TCP)

1. **DateTimeServer** Projekt in IDE öffnen  
[github.com/ibwgr/DateTimeServer](https://github.com/ibwgr/DateTimeServer)
2. Server starten. Dazu:
  1. Maven install via IDE. Siehe Screenshot.
  2. `java -cp target/*.jar ch.ibw.clientServer.server.dateReply.DateServer 6060`
3. Client starten. Dazu:
  1. MacOS: `nc 127.0.0.1 6060`  
Windows: `telnet 127.0.0.1 6060`



Variante:

- Anstelle von 127.0.0.1 die IP eines Klassenkammeraden verwenden ☺

# Rückblick

- Netzwerktopologien
- Verbindungsorientiertheit
- Schichtenmodel
- Client & Server Model

# HTTP

Einführung

# Hyper Text Transfer Protocol (HTTP)

- Anwendungsschicht
- Browser verwenden HTTP um Webseiten und deren Resourcen (Bilder, Skripte, Schriften etc.) zu laden
- Entstand 1991, mit der Erfindung des World Wide Web (Begründer Tim Berners-Lee)
- Request Response Model (Client, Server)

# HTTP Anfrage (Request)

- Was will ich (Verb)
- Von wem will ich (Pfad & Host)
- In welchem Format (accept)
- Welche Sprache
- Protokoll Version
- ...



# HTTP Antwort (Response)

- Status Code (**200 OK**)
- Format (Content-Type)
- Länge in Bytes (Content-Length)
- Protokoll Version
- ...

HTTP/1.1 200 OK

Date: Thu, 15 Jul 2004 19:20:21 GMT Headers

Server: Apache/1.3.5 (Unix)

Content-length: 46

Connection: close

Content-type: text/html

<h1>Antwort</h1>

<p>Ich bin eine Antwort</p>

Status

Leere Zeile

Body

# HTTP Verben / Methoden

- Die häufigsten
  - GET  
Datei vom Server lesen
  - PUT  
Datei auf den Server hochladen / überschreiben
  - POST  
Daten an ein Skript zur Verarbeitung übergeben
  - DELETE  
Datei vom Server löschen

# Übung

Chrome Devtools Network

# HTTP Versionshistory

## □ **HTTP/0.9**

was proposed in March 1991.

## □ **HTTP/1.0**

In May 1996 he was released the document RFC 1945, which served as the basis for the HTTP / 1.0.

## □ **HTTP/1.1**

The current version, adopted in June 1999. TCP-connection can remain opened after sending a response to the request. The client now have to send information about the host name.

## □ **HTTP/2**

February 11, 2015 published the final version of the blueprint the next version Protocol. Unlike previous versions, HTTP/2 is a binary protocol.

- In HTTP/2 hat sich einiges Grundlegend geändert. Details dazu für Interessierte auf:
  - <https://http2.github.io/>
  - <http://caniuse.com/#search=http2>

# HTTP Response Status Codes

- Immer zuoberst in der Antwort  
**HTTP/1.1 200 OK**
  - 1xx Info
  - 2xx Erfolg
  - 3xx Weiterleitung
  - 4xx Fehler beim Client
  - 5xx Fehler beim Server
  - <https://github.com/for-GET/http-decision-diagram>
- Beispiele
    - 200 OK
      - Alles gut. Geforderter Inhalt im Body.
    - 301 Moved Permanently
      - Weiterleitung. Neuer Ort steht im location Header
    - 401 Unauthorized
      - Client hat keine Berechtigung auf die Resource zuzugreifen
    - 404 Not Found
      - Resource nicht vorhanden
    - 503 Service Unavailable
      - System / Service nicht verfügbar
    - 500 Internal Server Error
      - Bei der Verarbeitung der Anfrage ist ein Fehler aufgetreten (z.B. NullPointerException)

# Übung

Simple HTTP Client

# Simple HTTP Clients

- curl
  - curl --trace-ascii debugdump.txt <https://curl.haxx.se>
  - [wttr.in](https://wttr.in)
- browser
  - Chrome, Safari, Opera, Firefox, ...
- Java
  - URL & StreamReader ([java] <https://www.grundkurs-java.de/index.php/beispiele#18.2.6>)
  - HttpClient <https://github.com/ibwgr/java-http-client>

# Rückblick

- HTTP Request & Response Syntax
- HTTP Headers
- HTTP Status Codes
- HTTP Clients

# Programm Heute

- Eigenes Protokoll für Zeitserver
- Server Architekturen (Thread vs Process)
- High Availability Zeitserver
- Serialisierung

# Übung

Client Befehle

# Java Zeitserver mit Protokoll – Native Client

1. Server starten. Dazu:
  1. `java -cp target/*.jar ch.ibw.clientServer.server.dateReply.DateServer 6060`
2. Einen TCP Client starten. z.B. nc oder telnet
  1. `nc 127.0.0.1 6060`
3. Server so erweitern, dass bei Eingabe von:
  1. `TIME`  
nur die Zeit ausgegeben wird
  2. `DATE`  
die Zeit mit Datum ausgegeben wird

Um die Eingabe an den Server zu schicken, musst du mit “Enter” den Befehl abschicken.

Lösung auf Branch date-time-protocol

# Server Architekturen

Probleme & Lösungen – Software Patterns

# Problemstellung

- Eine Webserver bekommt mehrere dutzend Requests pro Sekunde
- Clients (Menschen die am Surfen sind) erwarten Antwortzeiten von unter einer Sekunde!
- Wie beantworte ich als Server so viele Anfragen so schnell?
- Milchbüchlirechnung:
  - 20 simultane Clients \*
  - 10 Requests pro Seitenaufruf \*
  - 0.1 Seitenaufrufe / Sekunde
$$= 20 * 10 * 0.1 \text{ req/s} = 100\text{req/s}$$
- [internetlivestats.com/one-second/](http://internetlivestats.com/one-second/)

# Threads

- [youtube.com/watch?v=YB5I2w-8YQ4](https://youtube.com/watch?v=YB5I2w-8YQ4)

```
new Thread(new Runnable() {
    public void run() {
        // long running stuff here
    }
}).start();
```

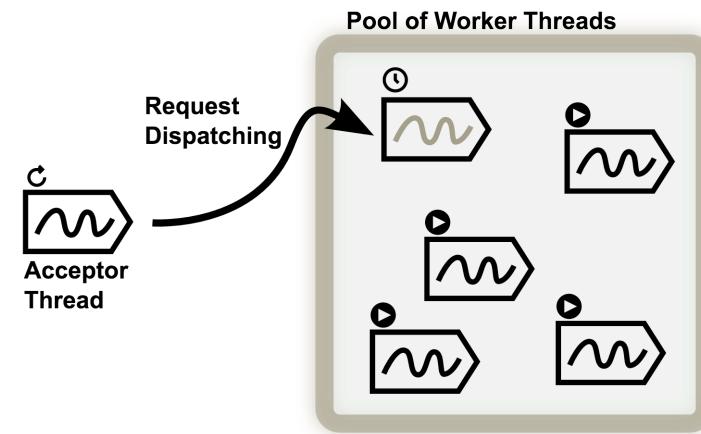
- Prozess hat 1-n Threads
- Pro CPU läuft ein Thread
- Thread hat Zugriff auf Prozessspeicher
- Quasi-Parallele Ausführung

# Lösungen

- Architekturmuster (Design Patterns)
  - Multi-Process
  - Multi-Thread
  - Event-Driven
    - nicht weiter behandelt in diesem Modul

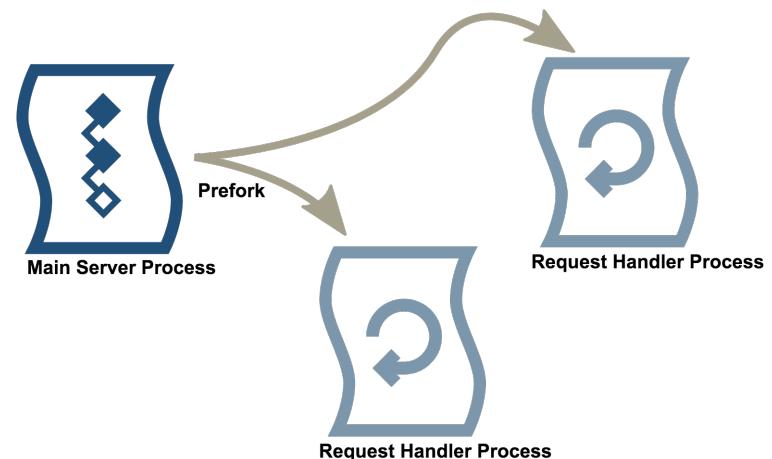
# Multi-Thread

- 1 Thread pro Verbindung
- Erstellen / Beenden / Switchen von Thread ist schnell
- Threads können auf shared Memory zugreifen (z.B. Cache)
- Verwendet von Tomcat Webserver



# Multi-Process

- 1 Prozess pro Verbindung
- Erstellen / Beenden / Switchen von Prozess ist aufwendig
- Verwendet von Apache Webserver by default



# Übung

DateTimeServer Performance Evolution

# DateTimeServer Evolution

Zur Zeit beendet der Server nachdem er die erste Response gesendet hat.

Ziel:

1. Server kann 1-n Anfragen nacheinander beantworten
2. Server kann 1-n Anfragen gleichzeitig von mehreren Clients beantworten

```
new Thread(new Runnable() {
    public void run() {
        // long running stuff here
    }
}).start();
```

# DateTimeServer Evolution – Lösung 1

1. Server kann 1-n Anfragen nacheinander beantworten

```
while(true){  
    Socket s = server.accept();    // Client-Verbindung akzeptieren  
    new DateTimeProtokoll(s).transact();    // Protokoll abwickeln  
}
```

- Eine Verbindung nach der anderen  
=> Solange eine Verbindung verarbeitet wird, kann keine neue Verbindung angenommen werden.

Wie können wir das demonstrieren / testen?

# DateTimeServer Evolution – Lösung 2

1. Server kann 1-n Anfragen gleichzeitig von mehreren Clients beantworten

```
while(true){  
    final Socket s = server.accept();      // Client-Verbindung akzeptieren  
    new Thread(new Runnable() {  
        public void run() {  
            new DateTimeProtokoll(s).transact(); // Protokoll abwickeln  
        }  
    }).start();  
}
```

- Eine Verbindung nach der anderen
- Während eine Verbindung in einem eigenen Thread behandelt wird, kann im Hauptprogramm eine neue Verbindung angenommen werden.  
Wie können wir das demonstrieren / testen?
- Musterlösung auf [github.com/ibwgr/DateTimeServer](https://github.com/ibwgr/DateTimeServer), Klasse ch.ibw.clientServer.server.javaReply.DateTimeServer

# Serialisierung

Objekte (Strukturen, Klassen) zwischen Programmen austauschen

# Serialisierung – Was, Warum?

Problem:

Server möchte ein Java Objekt an den Client schicken.

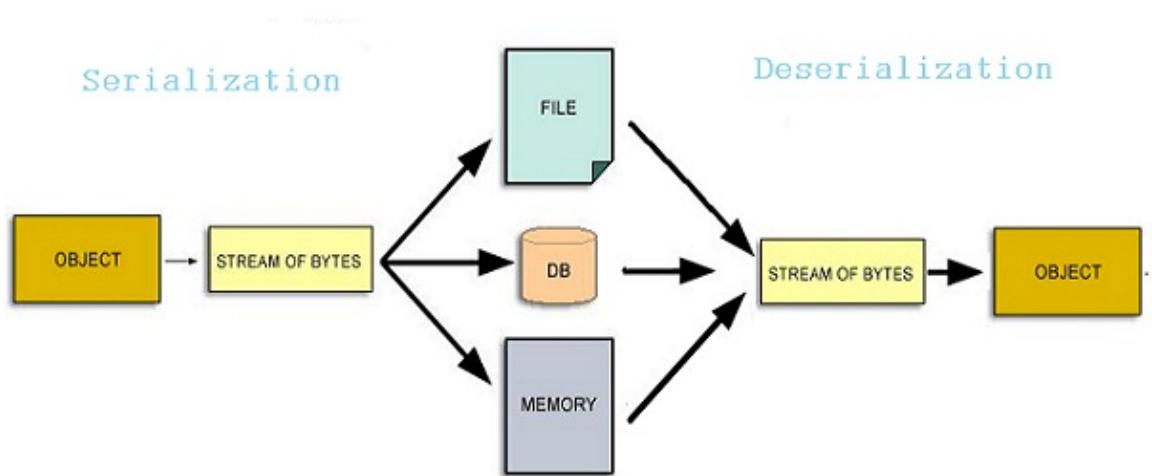
Lösung:

Server speichert alle Daten vom Objekt in eine Datei und schickt diese Datei über das Netzwerk zum Client.

Der Client liest die Datei und bestückt mit den Daten ein neu erstelltes Java Objekt.

# Serialisierung – Model

- Format ist standardisiert  
XML, JSON, CSV, Binär, ...
- Produzent und Konsument  
unterschiedliche Plattformen /  
Programmiersprachen
- Auch bekannt als marshalling  
/ unmarschalling



# Übung

Serializer Demo Walkthrough

<https://github.com/ibwgr/serializer>

# Serialisierung – mit Java Boardmitteln

- Nachteile
  - Serializable Marker-Interface
  - Binärformat – abhängig von Plattform/Sprache (nicht interoperabel)
  - Deserialize gibt Object zurück – expliziter Cast notwendig
- serialVersionUID  
Stellt sicher, dass der Produzent und Konsument die selbe Version der Klasse verwendet.

# Serialisierung – mit Java via XML / JSON / ...

- Vorteile:
  - Kein Marker-Interface
  - XML / JSON Format – Menschenlesbar, Plattformunabhängig
  - Deserialize gibt direkt gewünschten Typ zurück
  - Keine serialVersionUID notwendig
- Beispiel Jackson Library:
  - Default Konstruktor notwendig
  - public fields || public getter/setter || annotated fields
  - <https://github.com/FasterXML/jackson>

# Übung

DateTimeServer & Client mit Serialisierung

# DateTimeServer & Client mit Serialisierung

- Server schickt nicht mehr einfachen Datums-Text, sondern ein XML wo das serialisierte Java Objekt mit dem Datum drin steht
- Client empfängt XML und deserialisiert es in ein Objekt
- Idee für Klasse: `DateTimeInfo(String dateOrTime)`

```
<DateTimeInfo><info>Heute ist Saturday, der 21.12.19</info></DateTimeInfo>
```

# Java Zeitserver – Serializable

1. **DateTimeClient** Projekt in IDE öffnen  
[github.com/ibwgr/DateTimeClient](https://github.com/ibwgr/DateTimeClient)

2. Server starten. Dazu:

```
java -cp target/*.jar ch.ibw.clientServer.server.javaReply.DateTimeServer 6060
```

3. Client starten. Dazu:

1. Maven install via IDE.
2. java -cp target/\*.jar ch.ibw.clientServer.client.javaReply.DateTimeClient 127.0.0.1 6060

4. TIME oder DATE eingeben in Eingabeaufforderung. Mit Enter bestätigen.

5. Versuche, das selbe mit **nc** bzw. **telnet** als Client. Was ist das Ergebnis? Warum?

6. Ändere den Server so, dass auch telnet bzw. nc funktioniert (Hinweis: Serialisierung ☺)

7. Ändere den Client so, dass er mit dem neuen Server kompatibel ist

Tipp: Lass dich von der XmlSerializer Klasse inspirieren ...

Lösungen des Client und Server auf Branch serializable

Ergänze das jeweils im pom.xml

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.10.1</version>
</dependency>
```

# Rückblick

- Threads
- Zwei verschiedene Server Architekturen
- Multi-Thread Architektur mit Übung erarbeitet
- Serialisierung mit Java, interoperabel

# Prüfung

- $\frac{1}{2}$  A4 Zusammenfassung
- 25 Minuten
- Kein Computer

# Literatur

- [java] GRUNDKURS PROGRAMMIEREN IN JAVA  
<http://www.grundkurs-java.de/>

exit 0;