technische universität
dortmund

Fakultät für Informatik
Lehrstühle 4, 11, 12

Alexander Lochmann, Nils Kriege
Nils Dunker, Felix Homa, Simon Koschel, Sebastian Lau, Benedikt Maus

Wintersemester 2018/19

Rechnernetze und verteilte Systeme
Programmieraufgabe

Ausgabe: 18. Dezember 2018 **Abgabe:** 13. Januar 2019 **Besprechung:** 29. Januar – 1. Februar 2019

1 Problem

A. Nonymous hat sich für das neue Jahr vorgenommen, seine seit Jahrzehnten geführten Tagebücher zu digitalisieren und in Form eines Blogs der Öffentlichkeit zur Verfügung zu stellen. Hierzu möchte er einen eigenen Webserver betreiben. Leider genügt keines der existieren Software-Pakete, die unter einer Open-Source-Lizenz stehen, seinen hohen Ansprüchen. Kommerzielle Produkte kommen für ihn ebenfalls nicht in Frage.

Daher möchte er einen eigenen rudimentären Webserver entwickeln, der die für seine Zwecke notwendige Funktionalität zur Verfügung stellt.

2 Anforderungen

Der Webserver soll Teile des *Hypertext Transfer Protocol* in der Version 1.0 entsprechend des RFC 1945 unterstützen. Er soll unter einer IP-Adresse, z. B. `127.0.0.1`, und einem konfigurierbaren Port, z. B. 80, erreichbar sein (siehe Abschnitt 5.1) und die Dokumente sowie den Verzeichnisbaum unterhalb eines vorgegebenen Ordners zur Verfügung stellen. Wenn der Server auf einem System mit der IP-Adresse *`IP-Adresse`* aus dem Verzeichnis *`Ordner`* gestartet wurde, sollen die lokalen Dateien im Unterverzeichnis `wwwroot` folgendermaßen erreichbar gemacht werden:

`Ordner``/wwwroot/page.html`	erreichbar unter	`http://`*`IP-Adresse`*`/page.html`
`Ordner``/wwwroot/index.html`	erreichbar unter	`http://`*`IP-Adresse`*`/index.html`
	und	`http://`*`IP-Adresse`*
`Ordner``/wwwroot/subfolder/subpage.html`	erreichbar unter	`http://`*`IP-Adresse`*`/subfolder/subpage.html`

Aus Sicherheitsgründen muss verhindert werden, dass der Webserver Dateien außerhalb des Verzeichnis `wwwroot` zur Verfügung stellt. Auf Anfragen der Form `http://`*`IP-Adresse`*`/../../passwoerter.txt` soll daher angemessen reagiert werden. Wenn die angefragte URL keinen Dateinamen enthält, soll geprüft werden, ob der angefragte Ordner eine Datei `index.???` beinhaltet (z. B. `index.html`, `index.htm`, `index.txt`) und diese zurückgegeben werden.

Auf eine HTTP/1.1-Anfrage soll genauso geantwortet werden wie auf eine HTTP/1.0-Anfrage. Der Server soll die folgenden MIME-Typen unterstützen und an der Dateiendung erkennen:

.txt	`Content-Type: text/plain; charset=utf-8`
.html, .htm	`Content-Type: text/html; charset=utf-8`
.css	`Content-Type: text/css; charset=utf-8`
.ico	`Content-Type: image/x-icon`
.pdf	`Content-Type: application/pdf`
sonst	`Content-Type: application/octet-stream`

Alle Dateien vom Typ `text` sollen stets in der Kodierung UTF-8 ausgeliefert werden. Hierzu müssen die aus der lokalen Datei gelesenen Daten ggf. konvertiert werden (siehe Abschnitt 5.3). Im RFC 1945 sind auch sogenannte Multipart-Datentypen definiert, die der Server jedoch nicht unterstützen muss.

Der Webserver soll die Anfragemethoden `GET`, `HEAD` und `POST` unterstützen. Bei `GET`-Anfragen soll die Implementierung auch das Header-Feld `If-Modified-Since` beachten und entsprechend reagieren (*conditional GET*).

Die mit `POST`-Anfragen gesendeten Informationen sollen ignoriert werden und es kann genau so wie auf eine `GET`-Anfrage reagiert werden. Die Anfragemethoden `PUT`, `DELETE`, `LINK` und `UNLINK` sollen nicht implementiert werden, entsprechende Anfragen aber sinnvoll beantwortet werden.

Es sollen die folgenden Status-Codes unterstützt werden:

Code	Nachricht	Verwendung
200	OK	
204	No Content	Valide URL, aber kein Inhalt, hier: Ersatz für Verzeichnislisting (also URL zeigt auf Ordner anstatt auf Datei und es existiert keine `index`-Datei)
304	Not Modified	Mögliche Antwort auf *conditional GET*-Anfragen
400	Bad Request	Bei HTTP/2.0 anfragen (alternativ Code 501)
403	Forbidden	Keine Zugriffsrechte auf die angefragte Datei
404	Not Found	Die angefragte Datei existiert nicht
500	Internal Server Error	Unerwartete Fehler, z. B. bei Exceptions
501	Not Implemented	Reaktion auf `PUT`, `DELETE`, `LINK`, `UNLINK` Anfragen

Alle weiteren im RFC 1945 definierten Codes (z. B. 1xx, 201 Created, 202 Accepted, 300 Multiple Choices, 301 Moved Permanently, 302 Moved Temporarily, 401 Unauthorized) müssen nicht implementiert werden.

Der Webserver soll in der Lage sein, mehrere unterschiedliche Clients gleichzeitig zu bedienen und ihre Anfragen parallel zu bearbeiten. Hierzu sollen Threads verwendet werden (siehe Abschnitt 5.2).

3 Aufgabe

Zur Lösung der Aufgaben sollen Sie in Gruppen mit maximal drei Studierenden arbeiten. Der Webserver soll in der Programmiersprache Java implementiert werden. Machen Sie sich zunächst mit HTTP/1.0 mit Hilfe des RFC 1945 und der Programmierung in Java (siehe Abschnitt 5 und offizielle Dokumentation der Java API) vertraut. Gehen Sie folgendermaßen vor:

1. Laden Sie die Datei `Programmieraufgabe_src.zip` herunter und entpacken Sie diese auf Ihrer Festplatte. Das Archiv enthält einen Ordner `wwwroot` mit einer Webseite als Beispiel sowie ein vorgegebenes Rahmenprogramm bestehend aus zwei Java-Dateien:

   Die in der Datei `src/edu/udo/cs/rvs/Main.java` definierte Klasse startet den Webserver, wobei der Port als Kommandozeilenparameter übergeben wird.

   Die Klasse `HttpServer` ist von Ihnen in der Datei `src/edu/udo/cs/rvs/HttpServer.java` vollständig zu implementieren. Sie können ggf. weitere Dateien und eigene Klassen hinzufügen.

2. Bevor Sie mit der Implementierung beginnen, ist es ratsam, zunächst ein passendes Modell zu erstellen, das alle nötigen Elemente enthält, die für die Funktion des Webservers nötig sind.

3. Implementieren Sie den Webserver anschließend so, dass er die oben genannten Anforderungen erfüllt. Testen Sie Ihre Implementierung zunächst, indem Sie die zurückgegebenen Antworten ausgeben lassen und mit der erwarteten Antwort abgleichen. Anschließend sollen Sie den Webserver mit einem Webbrowser Ihrer Wahl testen.

3.1 Bibliotheken

Zur Lösung der Aufgabe soll ausschließlich auf die durch die Java-Standardbibliothek zur Verfügung gestellten Klassen zurückgegriffen werden, die sich in den folgenden Paketen (oder Subpaketen davon) befinden:

- `java.io.*`
- `java.lang.*`
- `java.net.*`
- `java.nio.*`
- `java.security.*`
- `java.util.*`

Klassen aus den Pakten `com.sun.net.*` bzw. `sun.net.*` oder Bibliotheken von Drittanbietern dürfen nicht eingebunden werden. Ebenfalls ist das Kopieren von Quellcode aus Bibliotheken oder anderen Fremdquellen nicht erlaubt. **Verstößen können zum Nichtbestehen der Studienleistung führen!**

3.2 Fehlerbehandlung

Im Programm sollten Exceptions (bzw. Fehlerzustände), die beispielsweise beim Verbindungsaufbau oder Versenden von Nachrichten entstehen können, verarbeitet werden. Eine unzureichende Fehlerbehandlung führt zu Punktabzug.

3.3 Kommentare und Dokumentation

Kommentieren Sie Ihren Quelltext ordentlich! Für Abgaben ohne ausreichende Kommentare werden Punkte abgezogen. Zusätzlich sollten Sie alle über die Aufgabenstellung hinausgehenden Bedienungsaspekte der Applikation (Befehle, Kommandozeilenparameter, ...) dokumentieren, um eine reibungslose Korrektur zu ermöglichen. Neben einer Textdatei ist es auch zulässig, diese Dokumentation von der Anwendung selbst (z. B. direkt nach dem Starten) ausgeben zu lassen.

3.4 Abgabe

Die Abgabe findet über AsSESS statt. Sie kann bis zum Abgabetermin beliebig oft durchgeführt werden; es wird dann die zuletzt abgegebene Version gewertet.

Abzugeben ist ein Zip-Archiv mit allen zum Projekt gehörenden Quelltexten sowie ein ausführbares Jar-Archiv mit dem fertigen Programm (siehe Abschnitt 5.4). Beim Start des Programms müssen die oben angegebenen Parameter akzeptiert werden.

4 Beispiel zu HTTP

Das folgende Beispiel zeigt eine Anfrage an den Server der TU-Dortmund und seine Antwort. Zeilenumbrüche im HTTP-Header werden stets mit `\r\n` (Carriage return und Line feed) erzeugt.

Anfrage:

```
GET / HTTP/1.0
Host: www.tu-dortmund.de

```

Beachten Sie, dass die abschließende Leerzeile notwendig ist, da sie das Ende der Anfrage signalisiert.

Antwort:

```
HTTP/1.0 302 Found
Server: nginx
Date: Tue, 27 Nov 2018 11:11:56 GMT
Content-Type: text/html; charset=iso-8859-1
Content-Length: 279
Location: https://www.tu-dortmund.de/

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://www.tu-dortmund.de/">here</a>.</p>
<hr>
<address>Apache Server at www.tu-dortmund.de Port 80</address>
</body></html>
```

5 Hinweise zur Implementierung

Die folgenden Hinweise zu Java-Komponenten und Techniken können für diese Aufgabe nützlich sein.

5.1 Sockets

Um zu vermeiden, dass jeder Programmierer, der über eine Netzwerkschnittstelle kommunizieren will, alle Protokolle, die dafür nötig sind (z. B. Ethernet, IP, TCP, UDP, etc.) selbst implementieren muss, gibt es eine Software-Abstraktion mit dem Namen *Socket*.

In Java werden zwei unterschiedliche Arten von Sockets unterschieden: *Sockets* und *ServerSockets*. Beide verhalten sich unterschiedlich.

Sockets sind diejenigen Sockets, über die tatsächlich Nachrichten verschickt werden können. Um eine Verbindung aufzubauen dient folgender Code:

```
Socket mySocket = new Socket();
mySocket.connect(
  new InetSocketAddress( "123.234.111.100", 12345 )
);
```

Dadurch wird eine Verbindung mit dem Rechner aufgebaut, der die Adresse `123.234.111.100` hat. Zur Adresse gehört noch ein Port. Da ein Rechner mehrere Verbindungen verwalten kann, kann mit Hilfe des Ports auf der Gegenstelle das zugehörige Programm ausgewählt werden.

Um textbasiert zu kommunizieren, werden in Java Helferobjekte benötigt. Das ist zum Senden der *PrintWriter* und für den Empfang der *BufferedReader*. Die *read*-Funktionen des *BufferedReader*s blockieren. Das heißt, dass sie warten, bis zu lesende Inhalte verfügbar sind. Dazu wieder ein Beispiel:

```
Socket mySocket = new Socket();
mySocket.connect(
  new InetSocketAddress( "123.234.111.100", 12345 )
);
PrintWriter out = new PrintWriter(
  mySocket.getOutputStream(), true
);
BufferedReader in = new BufferedReader(
  new InputStreamReader(
    mySocket.getInputStream()
  )
);
```

ServerSockets dienen dazu auf ankommende Verbindungen zu warten. Ein ServerSocket muss an einen lokalen Port gebunden werden:

```
ServerSocket ssocket = new ServerSocket();
ssocket.bind( new InetSocketAddress( 12345 ) );
```

Die Methode *accept* ist eine blockierende Methode, die wartet, bis eine Verbindung aufgebaut wird. Wird eine Verbindung aufgebaut, so kehrt die Methode zurück und übergibt als Ergebnis einen neuen Socket. Mit Hilfe dieses neuen Sockets kann dann mit der Gegenstelle kommuniziert werden.

```
Socket client = ssocket.accept();
```

Da die Netzwerkschnittstelle ein Bauteil des Rechners ist, ist auch die Interaktion mit dem Betriebssystem nötig. Daher „lebt“ jeder Socket im Betriebssystem und es muss dem Betriebssystem mitgeteilt werden, dass man die Verbindung nicht länger benötigt. Alle Sockets – also auch ServerSockets – müssen daher geschlossen werden. Ansonsten bleiben sie auch nach Ende des Programms offen. Ein erneutes Öffnen ist dann nicht mehr möglich – auch dann nicht, wenn das Programm neu gestartet wird. Denken Sie also immer an

```
mySocket.close();
```

beziehungsweise

```
ssocket.close();
```

Weitere Informationen über Sockets finden Sie in “The Java Tutorials”, Abschnitt “All About Sockets”, oder weiterer Java-Literatur Ihrer Wahl.

5.2 Threads

Threads sind parallel ablaufende Kontrollflüsse. Sie werden benötigt, wenn das Programm beispielsweise an einem Socket lauscht, ob Nachrichten eingehen oder auf eingehende Verbindungen wartet. Während gewartet wird, kann nichts anderes getan werden. Dieses Verhalten wird *blockieren* genannt. Ein Thread kann nun Abhilfe schaffen, indem ein solches Warten *parallel* ausgeführt wird. Einen Thread erzeugt man in Java wie folgt.

```
public class MyThread extends Thread
{
  @Override
  public void run{
    /* do something useful */
  }
}
```

Die Methode run() ist dabei die "main"-Methode des Threads. Ansonsten verhalten sich Threads wie ganz normale Klassen. Der folgende Codeschnipsel erzeugt einen Thread und startet ihn.

```
public class Program
{
  public static void main( String[] args ){
    MyThread t;
    t = new MyThread();
    t.start();
  }
}
```

Beachten Sie, dass ein Thread gestartet werden muss, damit er mit seiner Arbeit beginnt. Eine weitere wichtige Erweiterung von "MyThread" ist das Einfügen einer Hauptschleife.

```
public class MyThread extends Thread
{

  private boolean _terminate;

  MyThread()
  {
    _terminate = false;
  }

  public void terminate()
  {
    _terminate = true;
  }

  @Override
  public void run
  {
    while ( !_terminate )
    {
      /* do something useful */
    }
  }
}
```

Wichtig ist es zu beachten, dass der Zugriff auf den Thread aus einem anderen Thread heraus dazu führen kann, dass Daten inkonsistent sind. Beispielsweise könnte ein Thread einen String ändern, während ein anderer Thread diesen gerade lesen möchte. In dem Fall könnte vom zweiten Thread ein unsinniges Wort gelesen werden. Dieses Verhalten nennt man *konkurrierend*, da zwei Threads um den Zugriff auf eine Variable konkurrieren. In den Java-Paketen java.util.concurrent.* finden Sie Datenstrukturen, die gegenüber solchen konkurrierenden Zugriffen geschützt sind.

Weitere Informationen über Threads finden Sie in "The Java Tutorials", Abschnitt "Defining and Starting a Thread", oder weiterer Java-Literatur Ihrer Wahl.

5.3 Zeichenkodierung

Für Textdaten existieren unterschiedliche Zeichenkodierungen wie ASCII oder UTF-8. Java unterstützt verschiedene Kodierungen und kann diese ebenfalls konvertieren. Weitere Informationen hierzu finden Sie in "The Java Tutorials" unter den Abschnitten Byte Encodings and Strings und Character and Byte Streams.

5.4 Erstellen von Jar-Archiven

Sie können eine ausführbare Jar-Datei Ihres Programms mit folgendem Befehl erstellen:

```
jar cfe Webserver.jar edu.udo.cs.rvs.Main input-file(s)
```

Hierbei ist `input-file(s)` eine durch Leerzeichen getrennte Liste von Dateien und Verzeichnissen, die für das Programm benötigt werden. Bitte beachten Sie, dass die Ordnerstruktur im Jar-Archiv die Paketstruktur der Java-Klassen widerspiegeln muss. Wenn sich die `.class`-Dateien Ihres Programms im Unterordner `bin/edu/udo/cs/rvs` befinden, kann mit folgendem Befehl ein geeignetes Jar-Archiv erzeugt werden:

```
jar cfe Webserver.jar edu.udo.cs.rvs.Main -C bin .
```

Anschließend kann der Server mit dem Befehl `java -jar Webserver.jar` gestartet werden.

Weitere Informationen zu Jar-Archiven finden Sie in "The Java Tutorials", Abschnitt "Packaging Programs in JAR Files", oder weiterer Java-Literatur Ihrer Wahl.

Achtung!

Bitte nur selbst erstellte Lösungen abgeben. Plagiate und mehrfache Abgaben derselben Lösung werden mit 0 Punkten gewertet und führen zum Nichtbestehen der Studienleistung!

Die Abgabe der Aufgabe erfolgt über AsSESS unter https://ess.cs.tu-dortmund.de/ASSESS/.

Die Aufgabe soll in Gruppen mit 3 Studierenden bearbeitet werden! Kleinere Gruppengrößen müssen mit dem jeweiligen Übungsgruppenleiter abgesprochen werden.