Präsentationsbericht SWE

Philipp Andert
if18b072

Aufgabenstellung

Im Rahmen der Lehrveranstaltung Softwareengineering galt es, einen HTTP Webserver in Java oder C# zu implementieren. Weiters sollte dieser Server das dynamische Laden von - ebenfalls selbst entwickelten - Plugins unterstützen. Diese Plugins stellen teils Grundfunktionen zur Verfügung, teils simulieren sie auch echte Anwendungszwecke, dazu später mehr.

Der geschätzte Arbeitsaufwand pro Studenten belief sich auf etwa 21 Stunden, wobei ich aufgrund von Überarbeitungen diverser integraler Prozeduren auf etwa 28 Stunden gekommen bin. Darauf werde ich ebenfalls später eingehen.

Ich habe mich für die Programmierung in Java entschieden.

Konzept

Im Grunde genommen baut ein Server (üblicherweise TCP) Verbindungen zu anfragenden Client-Programmen auf und verarbeitet deren Anfragen, „Requests“. In diesem Fall sind die Clients Internetbrowser, welche Requests gemäß von HTTP verschicken. Typischerweise werden Websites und die dafür relevanten Daten angefragt, um diese anschließend in den Browser zu laden und anzuzeigen.

HTTP (Abk. für Hypertext Transfer Protocol) ist ein Protokoll, dass die Übertragung von Daten definiert. Trotz seines Namens können und werden auch nicht-textuelle Dateien übertragen, etwa binäre Bilddateien.

Wichtig ist auch, wo und vor allem wie angegeben wird, welche Dateien man erhalten möchte. Das geschieht über die URL (Uniform Resource Locator), die in der Adressleiste eines Browsers nach der IP-Adresse bzw. dem Hostnamen folgt.

Beispiel: *de.wikipedia.org/wiki/Java_(Programmiersprache)*

In diesem Beispiel lautet die URL „/wiki/Java_(Programmiersprache)“, welche dem zuständigen Wikipedia-Server mitteilt, dass der Artikel „Java (Programmiersprache) aus dem Verzeichnis „wiki“ übertragen werden soll.

Umsetzung im Detail

Verbindungsaufbau und Client-Handling

Startet man den Server, wird auf dem mitgegebenen Port ein neuer ServerSocket erstellt, der dann im Zustand ServerSocket.accept() auf eingehende Verbindungen wartet. Beim Verbindungsaufbau mit einem Client wird der von accept() zurückgegebenen Socket an ClientHandler in einem neuen Thread, der auch gleichzeitig gestartet wird, übergeben.

```
Server server = new Server(Integer.parseInt(args[0]), args[1]);
```

Argument 0 ist der Port und Argument 1 das Verzeichnis, in dem sich die Website befindet.

Im Konstruktor des Servers:

```
try {
    this.listener = new ServerSocket(port);
    System.out.println("Server started");
} catch (IOException e) {
    e.printStackTrace();
}
```

In der Main-Funktion wird in einer while-Schleife auf eingehende Verbindungen gewartet und bei jedem Verbindungsaufbau das Handling in einen neuen Thread verlagert, um paralleles Bearbeiten von Requests zu ermöglichen.

```
Socket socket = new Socket();
while(true) {
    System.out.println("Waiting for a client...");

    try {
        if(server.listener != null)
        socket = server.listener.accept();
    } catch (IOException e) {
        e.printStackTrace();
    }

    System.out.println("Client accepted");
    Thread thread = new Thread(new ClientHandler(socket));
    thread.start();
}
```

In der run-Methode von ClientHandler werden DataInputStream und OutputStream des zugehörigen Sockets geladen. Der DataInputStream wird an die Methode getWebRequests() der RequestFactory-Klasse übergeben, um ein Request-Objekt zu bekommen. Dieser Request wiederum wird an die statische Methode getSuitablePluginForRequest() von WebPluginManager übergeben, um das best-geeignete Plugin für die Aufgabe zu erhalten.

Das Plugin verarbeitet den Request in handle() und gibt einen Response zurück, mit dessen send-Methode der OutputStream beschrieben wird, was bedeutet, dass der Client ein Ergebnis erhält.

Der Socket wird im Anschluss geschlossen.

```
try {
    this.in = this.getDataInputStream();
    this.out = this.getOutputStream();

    Request req = new RequestFactory().getWebRequest(this.in);
    Plugin plugin = WebPluginManager.getSuitablePluginForRequest(req);

    Response resp = plugin.handle(req);
    resp.send(out);

    this.socket.close();

} catch (IOException e) {
    e.printStackTrace();
}
```

Parsen des Requests und der URL

Die Klasse RequestFactory kümmert sich um das Zerlegen des HTTP Request-Strings, der vom Client verschickt wird.

Zunächst wird ein BufferedReader-Objekt über einen InputStreamReader instanziert, der wiederum über den InputStream instanziert wird. Das ermöglicht überhaupt erst, die Daten des InputStreams als String zu interpretieren.

Darauf wird der BufferedReader Zeile für Zeile gelesen und der Inhalt direkt an die Methode parseRequestLine() übergeben. Der Inhalt von parseRequestLine() ist nicht relevant, wichtig ist bloß zu wissen, dass die Informationen des Requests in einzelnen Variablen gespeichert werden.

```
BufferedReader reader = new BufferedReader(new InputStreamReader(stream));
System.out.println("Parsing Request Lines...");
String line;

try {
    while (reader.ready() && !(line = reader.readLine()).equals("")) {
        this.parseRequestLine(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Nun ist ein kurzer Exkurs ins HTTP notwendig. Hier ist ein Beispiel eines korrekten Requests:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Wichtig ist die erste Zeile: GET beschreibt die Methode dieses Requests, gefolgt von der URL und der HTTP Version, der dieser Request entspricht. Auf die erste Zeile folgen diverse Header, die für die Verarbeitung relevant sein könnten. Handelt es sich um einen Request mit Methode POST, folgt noch nach den Headern der sogenannte „content body“ mit dem Inhalt des POSTs.

GET Requests fordern Dateien an, wohingegen POST Requests beliebige Daten an den Server übertragen, sie erwarten allerdings selbstverständlich auch eine Response vom Server.

Der POST content muss bytewise gelesen werden, was in der Methode parsePostContent() passiert, ebenfalls wird das Ergebnis in einer Variable gespeichert.

Gespeichert wird der Inhalt in der WebRequest-Klasse mithilfe der Setter-Methoden. Der URL-String muss aber noch geparst werden, weswegen er in setUrl von WebRequest an die Methode getWebUrl() von UrlFactory übergeben wird, die ein Objekt der Klasse WebUrl mit den Daten der URL zurückliefert.

```
public void setUrl(String url) {
    this.url = new UrlFactory().getWebUrl(url);
}
```

Die UrlFactory stellt zunächst fest, um welche Art von URL es sich handelt, sprich ob Parameter übergeben werden, ob ein Fragment vorkommt, welcher Inhalt überhaupt angefragt wird, etc. Dazu nochmal ein Beispiel, wie eine komplexere URL aussehen könnte:

somehost.com/directory/somepage.html?x=1&y=2#somefragment

Diese URL besteht neben des Dateipfades aus zwei Parametern (die im Kontext der Website eine beliebige Bedeutung haben können), x mit Wert 1 und y mit Wert 2 und einem Fragment somefragment, das ebenfalls eine bestimmte Funktion einnimmt. Oft handelt es sich bei Fragmenten um Sprungmarken, wie etwa auf *wikipedia.org*.

Nach Feststellen des URL-Typs wird der rohe String Stück für Stück in seine einzelnen Komponenten zerlegt, mithilfe von Funktionen wie split() und substring().

So kann man beispielsweise die URL zerlegen, um an das Fragment zu kommen:

```
String[] result = url.split("#");
this.url_without_fragment = result[0];
this.fragment = result[1];
```

Wichtige Methoden der WebResponse-Klasse

Die WebResponse-Klasse besteht aus den Grundmethoden des Response-Interfaces für das Setzen, Auslesen und Senden des Responses und seinen Komponenten und wurde von mir um ein paar Helfermethoden erweitert.

Das Setzen der Komponenten ist ebenfalls nicht nennenswert, dafür aber das Konstruieren eines korrekt beschriebenen OutputStreams mit den Daten des Responses. Dazu ein Beispiel eines Responses:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Die erste Zeile beinhaltet die HTTP-Version, den Statuscode des Responses und die dazugehörige Bezeichnung, in diesem Fall „OK“. Darauf folgen wieder diverse Header und am Schluss folgt bei Responses immer ein content body, hier mit Inhalt einer HTML-Datei, die im Browser dargestellt wird.

Wie vorhin bereits erwähnt, wird der OutputStream eines Sockets in der send-Methode beschrieben. Diese Methode wiederum greift auf Helfermethoden zu, die jeweils Stück für Stück die relevanten Informationen des Responses (die bereits gesetzt wurden) zusammentragen und in den Stream schreiben.

Hier die Methode, die einen ByteArrayOutputStream entgegennimmt und diesen mit den Statuscode-Informationen beschreibt:

```
private void writeStatusCodeToStream(ByteArrayOutputStream out) {
    // get status code
    int code = this.getStatusCode();
    // get code and text as string
    String codeStr = String.valueOf(code);
    String statusText = String.valueOf(validStatusCodes.get(code));

    for(int i = 0; i < codeStr.length(); i++) {
        out.write(codeStr.charAt(i));
    }
    // write space
    out.write(' ');
    // write status text
    for(int i = 0; i < statusText.length(); i++) {
        out.write(statusText.charAt(i));
    }
    // write newline
    out.write('\r');
    out.write('\n');
}
```

Methoden ähnlich wie diese werden nacheinander von der constructResponseStream-Methode aufgerufen, welche wiederum von der send-Methode aufgerufen wird.

Die send-Methode wandelt dann den ByteArrayOutputStream in ein Bytearray um, um den OutputStream des Sockets zu beschreiben. Die Nachricht gilt nun als gesendet, der OutputStream wird mit flush() wieder geleert.

```
byte[] arr = out.toByteArray();
try {
    network.write(arr);
    network.flush();
} catch (IOException e) {
    e.printStackTrace();
}
```

Plugins und Plugin-Manager

Die Plugins sind Erweiterungen des Servers, die einen Request entgegennehmen, ihre Fähigkeit, den Request zu bearbeiten, bewerten und je nach Implementierung verschiedene Aufgaben bearbeiten, für die sie auch geeignet sind. Daraufhin senden sie ihr Ergebnis mittels der Response-Klasse.

Zu implementieren galten:

- Ein statisches Plugin zur Beschaffung von Dateien, genannt StaticGetPlugin
- Ein Plugin, dass POST Requests mit Texten entgegennimmt und den Text in Kleinbuchstaben zurückgibt, genannt ToLowerPlugin
- Ein Plugin, dass einen Temperatursensor simuliert und das Auslesen der Daten im Browser erlaubt, genannt TemperaturePlugin
- Ein Plugin für die Navigation und Bedienung von OpenStreetMap, genannt NaviPlugin

Hilfreich ist auch die Implementation des ErrorPlugins, welches immer einen Score größer 0 hat und daher immer aufgerufen wird, wenn keines der Plugins den Request bearbeiten kann. Dieses Plugin sendet eine Response mit Status 500 Internal Server Error an den Client.

Ebenso haben auch alle anderen Plugins Zugriff auf die statische Methode constructErrorResponse von WebResponse, die, wie der Name schon verrät, eine Error-Response basierend auf dem übergebenen Statuscode konstruiert. Das ist notwendig, wenn beispielsweise StaticGetPlugin eine nicht vorhandene Datei senden soll.

Der Plugin-Manager, implementiert in WebPluginManager, lädt dynamisch die Plugins, die er im plugins Verzeichnis des Source-Verzeichnisses findet.

Zuerst wird jedes gefundene Plugin in die Plugin-Collection plugins vom Typen CopyOnWriteArrayList aufgenommen, mit Paketnamen vorangestellt.

Für jedes Plugin wird zuerst die Klassenbezeichnung geladen, mit der wiederum der passende Konstruktor geladen werden kann, um schließlich eine neue Instanz dieses Plugins zu erzeugen.

Zuletzt wird das Objekt zu plugins hinzugefügt.

```
Class pluginClass = Class.forName(plugin);

Plugin object = (Plugin) pluginClass.getDeclaredConstructor().newInstance();

plugins.add(object);
```

Die vorhin bereits erwähnte Methode getSuitablePluginForRequest() probt alle Plugins mithilfe deren Methode canHandle(), die den Bearbeitungs-Score setzt, und gibt das best-geeignete Plugin zurück.

WebPluginManager selbst wurde nach Singleton-Prinzip implementiert, sprich beim Aufruf der public Methode getPluginManager wird immer dieselbe, statisch angelegte, Instanz von WebPluginManager zurückgegeben. Existiert noch kein Objekt, wird es hier instanziert.

```
public static PluginManager getPluginManager(){
    if(manager == null){
        manager = new WebPluginManager();
    }
    return manager;
}
```

Reflexion und Erkenntnisse

Das Projekt war für mich bisher definitiv das Umfangreichste im Studium und durchaus sinnvoll, wie ich finde. Umfangreich sowohl im Hinblick auf die Funktionalität als auch die Stunden an Nachforschungen, die dafür nötig waren.

Am Anfang war ich noch sehr überfordert, da es in den zuvorgekommenen zwei Semestern nichts Vergleichbares gab, aber das hat sich mit der Zeit stark gebessert, je mehr ich mich damit – vor allem mit Java – auseinandergesetzt habe.

Besonders gut hat mir gefallen, dass sich mein Kreis des Verständnisses nun quasi geschlossen hat, was Webentwicklung angeht. Als mein selbst programmierter Webserver meine selbst programmierte Website aus dem ersten Semester geladen hat, war das definitiv ein Wow-Moment.

Im Nachhinein wünsche ich mir bloß, mich schon früher im Semester mit der Thematik auseinandergesetzt zu haben, ansonsten bin ich aber zufrieden mit dem, was ich zu leisten gelernt habe.