

Programmieren für das Internet

Martin Gasbichler Eric Knauel

7. November 2005

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

Studenten der Informatik dürfen dieses Dokument für ihre persönlichen Lehrzwecke verwenden.

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skript nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Martin Gasbichler, Eric Knauel 2002, 2003, 2005

Inhaltsverzeichnis

3	Kommunikation im Web über HTTP	5
3.1	HTTP-Adressen: Uniform-Resource-Identifiers und Uniform-Resource-Locators	5
3.2	HTTP-Nachrichten	6
3.3	HTTP-Anfragen	8
3.4	HTTP-Antworten	9
	3.4.1 Beispiele	10
3.5	Persistente Verbindungen	11

Kapitel 3

Kommunikation im Web über HTTP

Die Kommunikation zwischen Webbrowser und Webserver findet über das *Hypertext Transfer Protocol (HTTP)* [FGM⁺99] statt. Dieses Kapitel gibt einen Überblick über die aktuelle Version 1.1 von HTTP. HTTP setzt voraus, daß das darunterliegende Protokoll zuverlässig ist; in aller Regel wird TCP (vgl. Abschnitt ??) verwendet. Als well-known port wird Port Nummer 80 verwendet.

Der Ablauf der Kommunikation im HTTP-Protokoll ist sehr einfach: Der Client sendet eine Nachricht, die eine Anfrage beinhaltet und der Server schickt eine Antwort-Nachricht zurück. Damit ist die Kommunikation abgeschlossen und im einfachsten Fall wird auch die entsprechende TCP-Verbindung beendet. Die aktuelle HTTP-Version erlaubt es zwar eine TCP-Verbindung für mehrere Anfragen zu benutzen (sogenannte *persistente Verbindungen* und *Pipelining*), aber aus konzeptueller Sicht stehen die Anfragen in keinerlei Kontext.

3.1 HTTP-Adressen: Uniform-Resource-Identifiers und Uniform-Resource-Locators

Ein Client kann unterschiedliche Webseiten, oder, wie es allgemeiner in der HTTP-Spezifikation genannt wird, *Ressourcen* von einem Server anfordern. Diese Ressourcen werden durch ein eigenes Adressformat, die *Uniform Resource Locators (URL)* identifiziert. Dieses Adressformat baut auf dem allgemeineren Adressformat *Uniform Resource Identifier (URI)* auf. Sowohl Uniform Resource Locator als auch Uniform Resource Identifier sind in RFC 3986 [BLFM05] spezifiziert. Dieses RFC ersetzt die ursprüngliche URL-Spezifikation in RFC 1738 [BLMM94] und deren Nachfolger RFC 2396 [BLFM98].

Abbildung 3.1 zeigt einige Beispiel-URIs. Eine generische URI ist einfach aufgebaut:

```
<scheme>:<scheme-specific-part>
```

Der <scheme>-Teil der URI benennt das für die Kommunikation benutzte Protokoll, für die Kommunikation im Web ist dies `http`. Für Ressourcen, die eine hierarchische Struktur beschreiben, sieht die URI-Spezifikation die folgende Syntax (in EBNF) vor:

```
ftp://ftp.informatik.uni-tuebingen.de/pub/wsi.ps.gz
http://www.ietf.org/rfc/rfc3986.txt
http://www.beispiel.de/cgi/;sid=12&a=%42/test?c=%2A#(kompliziert)
mailto:knauel@informatik.uni-tuebingen.de
tel:+49-7071-2970500
```

Abbildung 3.1: Einige Adressen in der Uniform-Resource-Identifier-Syntax.

```
URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
```

Die folgenden Produktionen stellen eine Auswahl der wichtigsten Regeln aus der in der Spezifikation gegebenen Syntaxdefinition dar:

```
scheme          = ALPHA ( ALPHA | DIGIT | "+" | "-" | "." ) *

hier-part       = "//" authority path-abempty
                 | path-absolute
                 | path-rootless
                 | path-empty

authority       = [ userinfo "@" ] host [ ":" port ]

path-abempty    = ( "/" segment ) *

query           = ( pchar | "/" | "?" ) *
fragment       = ( pchar | "/" | "?" ) *

segment        = pchar *
segment-nz     = pchar +

pchar           = unreserved | pct-encoded | sub-delims | ":" | "@"
pct-encoded    = "%" HEXDIG HEXDIG
unreserved     = ALPHA | DIGIT | "-" | "." | "_" | "~"
reserved       = gen-delims | sub-delims
gen-delims     = ":" | "/" | "?" | "#" | "[" | "]" | "@"
sub-delims     = "!" | "$" | "&" | "'" | "(" | ")"
                 | "*" | "+" | "," | ";" | "="

absolute-URI   = scheme ":" hier-part [ "?" query ]
path-absolute  = "/" [ segment-nz ( "/" segment ) * ]
```

3.2 HTTP-Nachrichten

Das HTTP-Protokoll kennt genau zwei Arten von Nachrichten: Anfragen des Clients (**Request**) und Antworten des Servers (**Response**):

```
HTTP-message = Request | Response
```

Die beiden Nachrichten, **Request** und **Response** haben die gleiche Struktur und unterscheiden sich nur hinsichtlich der **start-line**:

```

generic-message = start-line
                  (message-header CRLF)*
                  CRLF
                  [ message-body ]
start-line       = Request-Line | Status-Line

Request-Line     = Method " " Request-URI " " HTTP-Version CRLF
Method           = "GET"
                  | "HEAD"
                  | "POST"
                  | "PUT"
                  | "DELETE"
                  | ...
                  | extension-method
extension-method = token

Status-Line      = HTTP-Version " " Status-Code " " Reason-Phrase CRLF
Status-Code      = drei Ziffern
Reason-Phrase    = Text

HTTP-Version     = "HTTP" "/" DIGIT+ "." DIGIT+

```

Die `Request-Line` spezifiziert die *Methode* mit der auf die unter der Adresse `Request-URI` zu findenden Ressource zugegriffen werden soll (siehe Abschnitt 3.3). `HTTP-Version` gibt an, welcher Protokoll-Version die übertragenen Nachrichten entsprechen — in diesem Fall handelt es sich immer um „HTTP/1.1“. Die `Status-Line` leitet den Anfang der Antwort des Server ein und enthält einen `Status-Code` (siehe Abschnitt 3.4). Dieser Code ist eine dreistellige Zahl, die Auskunft darüber gibt, ob die Anfrage erfolgreich bearbeitet wurde. Dieser Fehlercode wird durch `Reason-Phrase` in Textform näher erläutert. `CRLF` steht für die Zeichenfolge bestehend aus einem Carriage Return (ASCII 13) und einem Linefeed (ASCII 10).

Die eigentliche Nachricht folgt auf die `start-line` und besteht aus einer Reihe von Kopfzeilen `message-header` und einem optionalen Rumpf `message-body`. Kopfzeilen und Rumpf sind durch eine leere Zeile voneinander getrennt. Die Syntax der Kopfzeilen ist in RFC 822 [Cro82] festgelegt. Zusammengefasst sieht die Syntax einer Kopfzeile so aus:

```

message-header = field-name ":" [ field-value ]
field-name     = <printable ASCII characters>
field-value    = ( field-content | LWS)*
field-content  = <any ASCII characters other than CR or LF>
LWS           = [CRLF] ( " " | ASCII 11)

```

Ob die Nachricht einen Rumpf enthalten kann hängt von der Methode ab. Ist ein Rumpf vorhanden, so gibt der `Content-Type-Header` an, wie der Rumpf codiert ist und der `Content-Length-Header`, wie lang der Rumpf ist.

`Content-Type` gehört zur Klasse der Header-Felder, die sowohl in Anfragen, als auch in Antworten vorkommen können. Andere Felder dieser Klasse sind:

`Cache-Control` Zeigt zwischengeschalteten Cache-Mechanismen an, ob und wenn

ja, für wie lange die Nachricht in einem Zwischenspeicher abgelegt werden darf.

Connection Zeigt einen Header an, der nur für diese Verbindung gilt und nicht weitergereicht werden darf.

Date Datum und Uhrzeit, an dem die Nachricht abgesendet wurde.

Pragma Kann für implementationsspezifische Erweiterungen benutzt werden.

Trailer Verweist auf zusätzliche Header, die im Trailer einer **chunked** kodierten Nachricht zu finden sind.

Upgrade Erlaubt es, das Protokoll zu wechseln.

Via Listet die zwischengeschalteten Mechanismen auf, welche die Nachricht weitergeleitet haben.

Warning Enthält Warnungen über den Status der Nachricht.

3.3 HTTP-Anfragen

Das HTTP/1.1-Protokoll sieht eine Reihe unterschiedlicher Request-Methoden vor und läßt sich aber bei Bedarf noch um eigene Methoden erweitern. In diesem Abschnitt werden nur die gängigsten Request-Methoden vorgestellt. Für Requests ist die **start-line** eine **Request-Line**:

```
Request-Line = Method " " Request-URI " " HTTP-Version CRLF
```

Der Teil **Request-URI** beschreibt die Ressource auf dem Server:

```
Request-URI = "*" | absoluteURI | abs_path
```

- "*" besagt, daß die Anfrage sich auf den Server selbst bezieht.
- **absoluteURI** wird verwendet, wenn die Anfrage an einen Proxy gesendet wird.
- **abs_path** ist der Normalfall: der Pfad bezeichnet eine Ressource auf dem Server. In diesem Fall muß der **Host-Header** den Namen des Hosts enthalten.

Die hier angegebenen Non-Terminals beziehen sich auf die in Abschnitt 3.1 zusammengefasste Syntax für URIs [BLFM05].

HTTP 1.1 ermöglicht es, mehrere Webserver unterschiedlichen Namens, sogenannte *virtuelle Hosts*, auf demselben Webserver mit nur einer IP-Adresse laufen zu lassen. Die Namen der virtuellen Server werden alle zu dieser einen IP-Adresse aufgelöst. Damit landen alle Anfragen für alle virtuelle Hosts auf dem Webserver, der an auf der einen IP-Adresse und Port 80 lauscht. Nun müssen also die Anfragen für die einzelnen Server unterschieden werden. Die im Request enthaltene URI enthält, wie eben gesehen, nicht notwendigerweise eine Information welcher Host gemeint ist. Deshalb schreibt die HTTP-Spezifikation vor, daß Anfragen immer mit einem **Host-Header** zu versehen sind. Dieser **Host-Header** enthält den Namen des betreffenden virtuellen Hosts.

Die GET-Methode

Der gängigste Anfragetyp im HTTP-Protokoll ist die **GET**-Methode. Als Reaktion auf diese Anfrage sendet der Server die mit dieser URI assoziierte Information an den Client. Wird mit der URI beispielsweise eine Datei im Dateisystem des Servers referenziert, sendet der Server deren Inhalt.

Eine **GET**-Anfrage kann zu einer bedingten **GET**-Anfrage werden, wenn die Anfrage einen der folgenden Header enthält: **If-Modified-Since**, **If-Unmodified-Since**, **If-Match** oder **If-Non-Match**. Der Server antwortet auf bedingte Anfragen mit der gewünschten Information oder einem negativem Response-Code (vgl. Abschnitt 3.4). Durch bedingte Anfragen läßt sich die Menge der übertragenen Daten reduzieren. Zum Beispiel bedienen sich Proxy-Server dieser Methode, um Webseiten nur zu übertragen wenn der Webserver eine neuere als die vom Proxy zwischengespeicherte Version hat.

Der Client hat außerdem die Möglichkeit durch einen zusätzlichen **If-Range**-Header nur einen Teil der Daten anzufordern. Dadurch lassen sich beispielsweise abgebrochene Downloads fortsetzen.

Die HEAD-Methode

Die **HEAD**-Methode funktioniert im Prinzip genauso wie die **GET**-Methode. Allerdings sendet der Server die eigentlich angeforderte Information, also den **message-body** nicht, sondern nur den HTTP-Header. Die Header, die mit **HEAD**-Anfragen mitgeschickt werden dürfen, stimmen mit den Headern der **GET**-Methode überein. Die **HEAD**-Methode wird beispielsweise benutzt, um die Gültigkeit von Hyperlinks zu überprüfen.

Die POST-Methode

Die **POST**-Methode wird verwendet, um Daten zum Server zu übertragen, die dort als Eingabe für eine durch die URI bestimmte Funktion verwendet werden. Die Daten eines ausgefüllten HTML-Formulars werden zum Beispiel üblicherweise mit einem **POST**-Request an den Server übermittelt. Die Daten werden hierbei im **message-body** übermittelt (vgl. Abschnitt ??).

Die PUT- und DELETE-Methode

Die **PUT**-Methode ist der **POST**-Methode sehr ähnlich. Allerdings werden die vom Client zum Server übertragenen Informationen unter der angegebenen URI gespeichert. Diese Methode wird zum Beispiel benutzt um ganze Dateien zum Server zu übertragen („HTTP-Upload“).

Gegenstück zur **PUT**-Methode ist die **DELETE**-Methode mit der die Daten mit denen eine URI assoziiert ist wieder gelöscht werden..

3.4 HTTP-Antworten

Die Antwort des Server besteht aus einer HTTP-Nachricht (siehe Abschnitt ??). Die **start-line** enthält die sogenannte **status-line**. Die Informationen der **status-line** geben Auskunft darüber ob die Anfrage erfolgreich beantwortet werden konnte, bzw. welcher Fehler aufgetreten ist.

status-code	Bedeutung
200	Die Anfrage wurde erfolgreich bearbeitet.
202	Die Anfrage wurde angenommen, aber die Bearbeitung dauert noch an.
204	Die Anfrage wurde erfolgreich bearbeitet, aber die HTTP-Nachricht enthält keinen <code>message-body</code> .
206	Positive Antwort auf eine GET-Anfrage, die nur einen Teil der Daten angefordert hat.
301	Die angeforderte URI ist unter einer neuen dauerhaften URI erreichbar, diese URI ist im Header <code>Location</code> enthalten.
304	Antwort auf eine bedingte GET-Anfrage: Die Ressource hat sich nicht verändert. Die Antwort-Nachricht enthält keinen <code>message-body</code> .
400	Die Anfrage enthielt einen Syntax-Fehler.
403	Für den Zugriff auf angeforderte URI ist eine Authentifizierung notwendig. Der Zugriff wurde verweigert.
404	Die angeforderte Ressource wurde nicht gefunden.
500	Interner Serverfehler.
503	Der Server bearbeitet vorübergehend keine Anfragen (z. B. wegen Überlastung).
505	Die für die Anfrage verwendete Version des HTTP-Protokolls wird vom Server nicht unterstützt.

Abbildung 3.2: Die gängigsten HTTP-Statuscodes

Status-Line = HTTP-Version " " Status-Code " " Reason-Phrase CRLF

Gültige status-lines sind zum Beispiel:

HTTP/1.1 200 OK

HTTP/1.1 400 Bad Request

Status-code ist eine dreistellige Integerzahl mit festgelegter Bedeutung, eine Kurzbeschreibung der Bedeutung dieser Zahl wird üblicherweise außerdem in der Reason-Phrase gleich mitübertragen. Abbildung 3.2 zeigt die gängigsten HTTP-Statuscodes.

3.4.1 Beispiele

Abbildung 3.3 zeigt die Kommunikation zwischen Client und Server für eine GET-Anfrage. Der Client fordert die URI `/abc.html` auf dem Webserver `www-pu.informatik.uni-tuebingen.de` an. Der Server findet unter dieser URI eine Datei, die in der Antwort übertragen wird. Die Status-Line enthält den Statuscode 200, der die erfolgreiche Bearbeitung der Anfrage signalisiert. Die Header der Nachricht geben Auskunft über den Zeitpunkt der Bearbeitung (`Date`) und die Version der Serversoftware (`Server`) sowie, wann das unter dieser URI erreichbare Dokument zuletzt geändert wurde (`Last-Modified`). Der Header `Accept-Ranges` informiert den Client, daß bei Anfragen an diese URI, die nur einen Teil der Daten anfordern, die Größe des Teils in der Einheit Bytes angegeben werden kann. `Content-Length` enthält eine Längenangabe in Bytes für den Rumpf der Nachricht. `Content-Type` gibt Auskunft darüber um welchen Typ

```
GET /abc.html HTTP/1.1
Host: www-pu.informatik.uni-tuebingen.de

HTTP/1.1 200 OK
Date: Mon, 24 Oct 2005 09:02:35 GMT
Server: Apache/1.3.33 (Unix) PHP/4.3.11
Last-Modified: Tue, 19 Jul 2005 16:35:51 GMT
Accept-Ranges: bytes
Content-Length: 7627
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
[... weiterer Inhalt des Dokumentes ...]
```

Abbildung 3.3: Beispiel für eine Anfrage mit GET mit entsprechender Antwort.

Daten es sich handelt. In diesem Fall wird ein HTML-Dokument übertragen, welches durch den Typ `text/html` gekennzeichnet ist.

Abbildung 3.4 zeigt die Kommunikation zwischen Client und Server für eine POST-Anfrage. Der Client fordert die Ressource unter der URI `/cgi-bin/all-env.scm` an. Bei dieser Ressource handelt es sich um ein externes Programm (siehe Abschnitt ??), das die im Rumpf der Anfrage-Nachricht übertragenen Daten entgegennehmen soll.

Der Server antwortet mit dem Status-Code 200 — es ist also kein Fehler aufgetreten. Der Rumpf enthält, wie schon im vorherigen Beispiel, wieder ein HTML-Dokument. Doch diesmal gibt der Server keine Auskunft über die zu erwartende Länge des Rumpfes. Dies ist häufig der Fall, wenn der Rumpf der Nachricht durch ein externes Programm berechnet wird. In diesen Fällen kann der Server den Rumpf in einzelnen Stückchen, in `chunks`, übertragen. Der Header `Transfer-Encoding` mit dem Argument `chunked` signalisiert dies. Die Übertragung erfolgt in diesem Beispiel in zwei Stücken: Zuerst wird ein Stück der Länge `C5B` Bytes, also 3163 Bytes, übertragen, dann ein Stück mit Null Bytes. Die Längen der einzelnen Stückchen werden dabei in dem Rumpf der Nachricht geschrieben. Wenn das nächsten Stück die Länge Null hat, weiß der Client, daß der Rumpf komplett ist.

Der Header `Transfer-Encoding` hat eine allgemeinere Aufgabe: Hier wird die Funktion angegeben, die auf den Rumpf der Nachricht vor der eigentlichen Übertragung angewendet wurde. Andere mögliche Transferfunktionen sind zum Beispiel Kompression — dadurch wird der Rumpf komprimiert und spart Bandbreite bei der Übertragung.

3.5 Persistente Verbindungen

Konzeptionell ist ein Kommunikationsvorgang im HTTP-Protokoll nach Senden der Response-Nachricht beendet — es gibt keine Möglichkeit sich bei der nächsten Anfrage auf die vorhergehende Anfrage zu beziehen. In älteren Versionen von HTTP [BLFF96] drückte sich das auch auf technischer Ebene aus: Die Spezifikation schrieb vor, daß der Server die TCP-Verbindung nach Übertragung

```
POST /cgi-bin/all-env.scm HTTP/1.1
Host: www-pu.informatik.uni-tuebingen.de
Content-Length: 10

Hallo=Welt
HTTP/1.1 200 OK
Date: Mon, 24 Oct 2005 09:17:21 GMT
Server: Apache/1.3.33 (Unix) PHP/4.3.11
Transfer-Encoding: chunked
Content-Type: text/html

c5b
<html><body>
[Chunk der Größe 3163 Bytes (C5B Bytes) wird übertragen]
0
```

Abbildung 3.4: Beispiel für eine Anfrage mit POST mit entsprechender Antwort.

der Response-Nachricht beendete.

Um weitere Anfragen zu stellen, mußte der Client also erstmal eine neue TCP-Verbindung aufbauen. Da Webseiten häufig Bilder enthalten, die in separaten Anfragen vom Server angefordert werden, müssen in der Regel für das Laden einer Webseite mehrere TCP-Verbindungen für die einzelnen HTTP-Anfragen aufgebaut werden. Dies belastet nicht nur Server und Netzwerk unnötig, sondern macht sich auch als Verzögerung beim Benutzer bemerkbar. Mit der Einführung von HTTP 1.1, ist es möglich eine einmal aufgebaute Verbindung für weitere Anfragen wiederzuverwenden. In der Terminologie der Spezifikation handelt es sich bei Verbindungen dieser Art um *persistente Verbindungen*.

Neben persistenten Verbindungen sieht HTTP 1.1 auch die Kommunikation im *Pipeline-Modus* vor. Ein Client stellt dabei mehrere Anfragen über eine Verbindung direkt hintereinander und wartet danach auf die Antworten des Servers. Die Antworten des Servers treffen in der Reihenfolge der Anfragen beim Clienten ein. Durch diese Methode kann die Verbindung noch einmal effizienter benutzt werden.

Literaturverzeichnis

- [BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996. [11](#)
- [BLFM98] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifiers (URI): Generic syntax. <http://www.faqs.org/ftp/rfc/rfc2396.txt>, August 1998. [5](#)
- [BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005. [5, 8](#)
- [BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), December 1994. Updated by RFCs 1808, 2368, 2396, 3986. [5](#)
- [Cro82] D. Crocker. Standard for the format of ARPA Internet text messages. RFC 822 (Standard), August 1982. Obsoleted by RFC 2822, updated by RFCs 1123, 1138, 1148, 1327, 2156. [7](#)
- [FGM⁺99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol. <http://www.faqs.org/rfcs/rfc2616.html>, June 1999. [5](#)

Index

Host, 8

Hypertext Transfer Protocol (HTTP),
5

Methode, 7

persistente Verbindungen, 5, 12

Pipeline-Modus, 12

Pipelining, 5

Ressourcen, 5

Uniform Resource Identifier, 5

Uniform Resource Locators, 5

URI, 5

URL, 5

virtuelle Hosts, 8

Webbrowser, 5

Webserver, 5

well-known port, 5