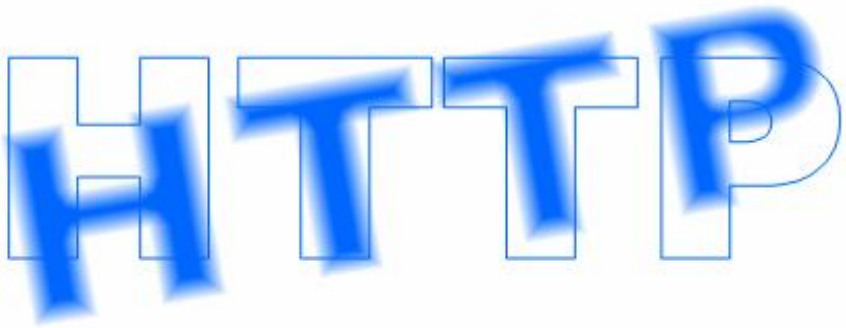


Hypertext Transfer Protocol

Ausarbeitung und Präsentation

Fach: Multimedia- und Webtechnologien

Zeitraum: WS 2005

Präsentation: 10. Januar 2006

Von:

Jens Franke
Josef Jaschkowski
Alex Miller

Inhaltsverzeichniss

1. Algemein

Das Hypertext Transfer Protocol (HTTP) ist das wichtigste und am häufigsten verwendete auf TCP/IP basierende Protokoll im World Wide Web (WWW). Allgemein kann man HTTP als ein Protokoll beschreiben, mit dem man Daten, hauptsächlich Webseiten, aus dem WWW in einem Webbrowser (Internet Explorer, Firefox,...) laden kann. Allerdings ist HTTP nicht nur auf Hypertexte beschränkt, sondern kann auch zum Austausch von Grafiken, Videos, und anderen beliebigen Dateien eingesetzt werden. Das HTTP Protokoll ist zustandslos, dass heißt der HTTP-Server kann keinen Zusammenhang zwischen einzelnen Anfragen eines Browsers herstellen. Jede Anfrage ist somit eigenständig und unabhängig von vorangegangenen Anfragen.

HTTP ist ein recht einfaches Protokoll, welches die Schnittstelle zum Benutzter darstellt. Im ISO/OSI-Schichtenmodell entspricht dies der Anwendungsschicht (Schicht 7). Jedoch wird dieses Modell heute kaum noch angewendet. Um die Netzwerkkommunikation im Allgemeinen darstellen zu können wird das TCP/IP-Referenzmodell angewendet. Auch hier befindet sich HTTP auf der Anwendungsschicht.

TCP/IP-Schicht	» OSI-Schicht	Beispiel
Anwendungsschicht	5-7	HTTP
Transportschicht	4	TCP
Internetschicht	3	Ipv4, Ipv6
Netzzugangsschicht	1-2	Ethernet

Diese Modell soll an dieser Stelle nicht näher erläutert werden, da es ähnlich dem bekannten ISO/OSI-Referenzmodell ist.

2. Geschichte

Die Geschichte des HTTP begann lange nach dem Start des Internet. 1989/90 ersann der britische Informatiker Tim Berners-Lee die Idee eines World-Wide-Web. Die Idee war, dass jeder Computer der innerhalb des WWW Daten anbieten konnte und dass man von diesem dann auch bestimmte Daten beziehen konnte. Zur Umsetzung eben dieser Idee, war es nötig, dass der Datenerfrager (der Client) seinen Wunsch nach Daten ausdrücken konnte und ebenso dass der Datenlieferer (der Server) verstand, welche Daten gewünscht waren und diese dann auch in geeigneter Form liefern konnte. Für eben diese Kommunikation benötigte man sodann ein Protokoll: HTTP. HTTP gibt es seitdem in der dritten Version: Nach HTTP/0.9 und HTTP/1.0 ist nun HTTP/1.1 die aktuelle Version.

Die meisten zahlreichen standardisierte Protokolle haben ihren Ursprung zumeist im Werdegang des RFC. Der Begriff RFC bedeutet Request for Comments, was auf Deutsch „Bitte um Kommentare“ heißt. Die Idee stammt von dem Student Steve

Crocker, der 1969 einen Artikel namens „Host Software“ verfasste, in dem er zur öffentlichen Diskussion zu seinen Vorschlag aufrief. Er nannte die Dokumentation Request for Comments und schuf eine bis dahin nicht gekannte Plattform des wissenschaftlichen Meinungsaustausches.

Nach seinen Beitrag folgten Diskussionen über verschiedenste Protokollvorschläge für Netzwerke, die Anfangs anhand ihres Erscheinungsdatum durchnummeriert wurden. Im Internet Activities Board (IAB) schuf man 1983 die Stelle eines „RFC-Editors“, der die eingehenden Diskussionsbeiträge erst nach Begutachtung zur Veröffentlichung als RFC freigab.

Das RFC 2616 behandelt das Hypertext Transfer Protocol 1.1 , welches als Standard festgelegt ist. Im Anhang befindet sich eine Auflistung der wichtigsten RFC’s.

Vor HTTP wurde hauptsächlich FTP für Datenübertragung angewendet. Dieses Verfahren hat jedoch folgende Nachteile:
man muss genau wissen was und wo man sucht
keine erklärende grafische Oberfläche
keine weiterführende Hinweise
FTP wurde in vielen Bereichen von dem Duo HTTP und HTML abgelöst.

Hauptgewicht wurde auf ein einfach zu implementierendes und eine auf das Abrufen von einfachen textbasierten Dokumenten von ein Server ermöglichendes Protokoll gelegt.
Haupziele des Entwurfs:

- Einfachheit des Protokolls:	- es sollte sich problemlos auf Servern und Clients implementieren lassen - es sollte dort nicht viele Ressourcen für sich beanspruchen
- Schnelligkeit des Protokolls:	das Protokol sollte so schnell wie möglich sein, um das schnelle Abrufen von Informationen zu erleichtern

Effizienz und Verwaltung bezüglich der zu übertragene Informationsmenge mit Hinblick auf das Wachsum des Web.

3. Versionen

3.1 HTTP/0.9

Dieses Protokoll wurde vom Tim Berners-Lee am CERN entwickelt um HTML-Dokumente (also Textbasierte Dateien) Weltweit zu versenden.

Tim Berners Lee wurde am 8. Juni 1955 in London geboren. Er hatte die HTTP-Idee seinen Arbeitgeber CERN vorgeschlagen. Die HTTP-Idee beruhte auf dem Prinzip des Hypertextes, und sollte den Datenaustausch und –aktualisierung zwischen Wissenschaftlern vereinfachen.
Das CERN ist das Europäische Kernforschungslabor, die Abkürzung stand für Conseil Européen pour la Recherche Nucléaire. Heute heißt es European Organisation for Nuclear Research.

Durch dieses Protokoll war es nur Möglich, ein Dokument anzufragen, es war nicht möglich (persönliche) Daten zum Server zu senden.
Eine HTTP/0.9-Request-Nachricht besteht nicht wie die andere Versionen aus einen Header (Kopf) und ein Body (Rumpf), sondern nur aus einen Header. In diesen Header steht allein die Start-Linie. Hier unterscheidet sich HTTP/0.9 von HTTP/1.0, bei HTTP/0.9 wird die Versionsnummer nicht angegeben.
Die einzige Möglichkeit Daten bzw. Dokumente anzufragen ist die methode `GET`.
Bsp.:
Request: `GET <URL>`
Response: `<Daten>`

Demgegenüber hat die Response-Nachricht auch nur ein Teil, nämlich nur ein Body. Dies bedeutet das keine Metainformationen gesendet werden können.
Direkt nach der Übertragung der Datei wird die Verbindung zum Server unterbrochen.

Hinweis: Metainformationen oder Metadaten sind Daten die Informationen über andere Daten enthalten, Beispielsweise die Eigenschaften eines Objekts.
Im Falle der Response-Nachricht können keine Angaben über Inhalt und Zustand usw... gesendet werden.

Der Vorteil dieses Protokolls liegt in seiner Einfachheit und die Möglichkeit beliebige Art von Dokumenten zu übertragen.

Durch das Fehlen einiger wesentlichen Eigenschaften und das Vorhandensein von nachfolgend beschriebene Nachteile wird von dieser Version nur noch selten gebrauch gemacht.

Ein erster Nachteil besteht darin das der Server nach jede Verbindung „dichtmacht“. Das hat folgende Konsequenzen:

- Wenn in ein Dokument z.B. Bilder enthalten sind, muss der Client das Dokument und jedes einzelne Bild separat anfordern.
- Lange Wartezeiten für den Benutzer
- Das Netzwerk wird durch Requests gestaut
- Wenn WebBrowser verschiedene Verbindungen gleichzeitig öffnen (bei Netscape bis zu 4 Verbindungen), wird der Server ebenfalls gestaut

Als zweiter Nachteil kann gewertet werden das keine verschlüsselte Daten gesendet werden können, die Menge der Daten auf diese Weise reduziert ist. Ebenfalls besteht hierdurch die Gefahr der Verletzung der Vertraulichkeit der Daten.

Der Benutzer kann eine fehlerhafte Webseite sehen, wo die Fehler für ihm Sichtbar sind, der Browser ist nicht in der Lage diesenFehler zu bemerken.

Die Eigenschaften des Protokolls sind im RFC 1123 hinterlegt. (Requirements for Internet Hosts - Application and Support)

3.2 HTTP/1.0

Der verbesserte Nachfolger von HTTP 0.9 ist HTTP 1.0. Es wurde in Mai 1996 als RFC veröffentlicht und vereinfacht das surfen im Web: bei Betrachtung der HTTP-Nachricht fällt auf das mehr Informationen übertragen werden.
Die Versionsnummer 1.0 wird mit übertragen, ebenfalls wird mit den Request den verwendeten Browser mit übertragen. Unter verwendung von Anweisungen werden verschiedenartige Informationen in beide Richtungen übertragen. Als letztes wird dan das benötigte Dokument oder die Datei (Entity) übertragen.

Ein Request besteht aus 3 Elementen: eine Methode (method), gefolgt von Headers und anschließend den Entity-Körper. In HTTP/1.0 wurde als erstes die Trennung zwischen Head und Body vollzogen.
Die Methode beschreibt was überhaupt gemacht werden soll. Dafür gibt es drei Befehle: GET, HEAD und POST.

Der Header besteht ebenfalls aus 3 Teile: einen General Header, einen Request Header, einen Entity-Header.
Diese Elemente werden in einen späteren Abschnitt erklärt.

Die Antwort vom Server unterscheidet sich vom Request durch eine Status-Linie am Anfang der Nachricht, bestehend aus 3 Teile: die Version des Protokolls, eine Statuscode und eine lesbare Statusinformation. Dies ist auch eine wesentliche Verbesserung gegenüber HTTP/0.9.

Eine letzt Neuerung ist die Authentifizierung, die es ermöglicht, geschützte Quellen vor fremden Zugriff zu bewahren. Es haben dann nur authorisierte Benutzer zugriff auf diese Resourcen.

Allerdings sind nicht alle Probleme die bei HTTP 0.9 vorhanden waren gelöst worden: das Problem der Mehrfachverbindungen ist noch vorhanden, trotz der "keep-alive"-Anweisung.
Diese "keep-alive"-Anweisung ist in keiner offiziellen Spezifikation bezüglich HTTP/1.0 beschrieben.
Wenn der Browser keep-alive unterstützt, fügt er ein zusätzlichen Header zur HTTP-Nachricht (CONNECTION: KEEP-ALIVE). Wenn der Server diesen Request empfängt, hängt dieser ebenfalls einen Header an seiner Antwort an (CONNECTION: KEEP-ALIVE).
Die Verbindung bleibt dann geöffnet bis der Client oder der Server sich für ein Ende der Verbindung entscheiden.
Weiter kann diese Version nicht gut mit relative URL's umgehen: es ist dan immer Notwendig die vollständige (absolute) Pfadangabe einzugeben wenn ein Server verschiedene virtuelle Server verwaltet.
Die Authentifizierungsmethode ist zu allgemein gehalten worden, und das Cache-Handling ist zu einfach gehalten worden.

In Verband mit HTTP/1.0 gibt es folgende RFC's:

- RFC 1945: HTTP/1.0
- RFC 2617: Basic and Digest Authentication Methods

4. HTTP/1.1

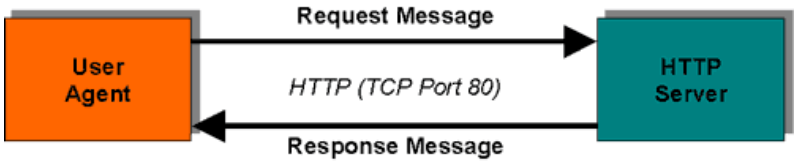
HTTP/1.1 bringt eine große Anzahl an Verbesserungen mit sich. Die folgende Liste soll nur einen kurzen Überblick über die wichtigsten Neuerungen geben, da später genauer auf diese Themen eingegangen wird.

- Persistente Verbindung
- Unterstützung von nicht IP-basierten virtuellen Hosts
- Neue Request-Methoden: DELETE, OPTIONS, PUT und TRACE
- Content Negotioation
- Chuncked Encoding
- Weiterentwicklung des Caching
- Verbesserung der Sicherheit des Authentifizierungsschemas

4.1 Funktionsweise

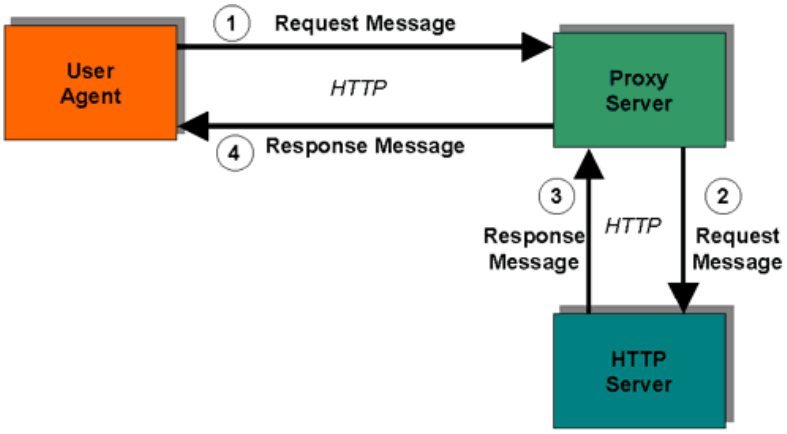
Die grundlegende Funktionsweise besteht darin, dass ein Client (meistens ein Webbrowser) eine Anfrage (Request) an einen Server stellt. Dafür muss der Client eine TCP-Verbindung aufbauen. Der Server antwortet darauf mit einem Response. Diese Antwort enthält bei HTTP/1.1 immer einen Status-Code, der zum Beispiel besagt, ob die Transaktion erfolgreich abgeschlossen werden konnte. Zusätzlich werden Informationen über den Server übertragen und gegebenenfalls die angeforderten Daten. Danach beendet der Server die Verbindung.

Wenn man z.B. auf einer Website den Link *http://www.beispiel.de/hompage.html* anklickt, so wird an den Computer mit dem Namen *www.beispiel.de* die Anfrage gerichtet, die Datei *homepage.html* zurückzusenden. Der Hostname *www.beispiel.de* wird dabei zuerst über das DNS-Protokoll in eine IP-Adresse umgewandelt. Zur Übertragung wird über das TCP-Protokoll auf (meist) Port 80 eine *HTTP*-GET-Anforderung gesendet. (siehe Kapitel 4.2.3.1)



Jedoch ist auch eine Kommunikation über Zwischenstationen möglich. So kann beispielsweise auch ein Proxy zwischen Client und Server arbeiten. In diesem Fall ist

der Proxy sowohl Server als auch Client. Er nimmt den Request an und sendet ihn als Client an den HTTP-Server. Dieser wiederum sendet die Response Message an den Proxy, der diese dann zum Client zurücksendet.



4.2 Aufbau

Der Aufbau dieser HTTP Nachrichten ist relativ einfach. Das Format ist im RFC 822 definiert.

Eine Nachricht besteht aus einer *Start-Line*, null oder mehr *Message-Header* – Feldern, einer Leerzeile sowie dem optionalen *Message-Body*.

Die Start-Line ist entweder eine *Request-Line*, wenn die Nachricht ein Request ist, oder eine *Status-Line* bei einem Response. (siehe Kapitel Request und Response)

Im Message-Body wird das eigentliche Entity (Daten) übertragen also beispielsweise der Quellcode der angeforderten Website.

Es gibt 4 verschiedene Message-Header (auch Header-Felder genannt), die Informationen enthalten, welche für die Kommunikation wichtig sind. Sie werden immer in folgendem Format angegeben: [Header Name]: [Wert]

- **General Header**: Einsatz bei Request- und Response-Nachrichten, enthält grundsätzliche Informationen
- **Entity Header**: Einsatz bei Request- und Response-Nachrichten; enthält Informationen über das zu versendende Entity
- **Request Header**: Einsatz in Request Nachrichten, enthält Informationen über den Request und den Client
- **Response Header**: Einsatz in Response Nachrichten, enthält Informationen über den Server (z.B. Versionsnummer)

Da es weit mehr als 40 Header gibt, soll an dieser Stelle nur ein kurzer Überblick über die wichtigsten Header gegeben werden.

4.2.1 General Header

Cache–control: Die Caching Information dient dazu, allen Caching-Systemen mitzuteilen, wie sie mit einer bestimmten Nachricht verfahren sollen. Da dieses Thema sehr wichtig ist, wird dies später detaillierter behandelt.

Connection: Bei HTTP/1.0 wurde `connection:keep-alive` verwendet um eine persistente zu erstellen. Bei HTTP/1.1 ist dies nicht mehr nötig, da alle Verbindungen persistent sind. Falls ein Client jedoch die Art von Verbindung nicht mehr wünscht, muss `connection:close` gesendet werden.

Date: Dieser Header dient der Übertragung von Zeitstempeln, die beispielsweise für Caching-Funktionen benötigt werden. Der Zeitpunkt sollte in diesem Format angegeben werden:

DDDD, DD, MMMM JJJJ HH:MM:SS ZONE

DDDD: Englische Abkürzung des Wochentages
MMMM: Englischer Name des Monats
ZONE: Internationale Abkürzung der Zeitzone

Bsp.: `Date: Sun, 25 March 2001 20:05:17 GMT`

4.2.2 Entity Header

Content-lenght: Gibt die Größe (in Byte) der Daten im Body an. So ist es möglich festzustellen, wann die Übertragung abgeschlossen ist.

Bsp.: `Content-lenght: 91`

Content-language: Wenn Entities übertragen werden sollen, die eine bestimmte Sprache enthalten, kann dieser Header angegeben werden. Nach RFC 1766 werden Sprachen mit Hilfe von Sprach-Tags abgekürzt, die aus einem primären Tag (Sprachenabkürzung nach ISO 693) und einem optionalen Subtag (Ländercode nach ISO 3166) bestehen.

Beispiele:
`Content-Language: de/DE` (Deutsch/Deutschland)
`Content-Language: el/GR` (Griechisch/Griechenland)

Last-Modified: Dieses Header-Feld gibt das Datum und die Uhrzeit an, an dem das Entity zum letzten mal geändert wurde.

4.2.3 Request

Der folgende Request zeigt eine typische Anfrage eines Browsers an einen Webserver.

```
GET /homepage.html HTTP/1.1
Accept: text/html, image/jpeg, image/gif, */*
Accept-Charset: ISO-8859-1
User-Agent: Mozilla/5.0 (Windows; Win 9x 4.9; de-DE;
rv:1.7.12) Gecko/20050919 Firefox/1.0.7
Host: www.beispiel.de
```

Jeder Request beginnt mit einer Request-Line. Diese Zeile beinhaltet zum ersten die Methode, also das was der Client vom Server möchte. In diesem Beispiel wird die GET Methode, die am häufigsten eingesetzte Methode, verwendet. Sie wird eingesetzt um Daten von einem Server anzufordern. (siehe 4.2.3.1)

Der zweite Teil der Request-Line (hier: *„/homepage.html“*) ist die Request-URL. Hier gibt der Client an, welche Website oder Datei er aufrufen will. Der Host (Name des Servers) muss in den nachfolgenden Header-Feldern definiert werden. In HTTP/0.9 und HTTP/1.0 war dies nicht nötig, so hat der Request `GET /hompage.html HTTP/1.0` ausgereicht. Dies hat folgenden Grund:
Bevor ein Request gesendet werden kann muss eine Verbindung zum Server hergestellt werden. Dies geschieht vom Prinzip her so, dass der Hostname (z.B. *www.beispiel.de*) durch das **D**omain **N**ame **S**ystem (verteiltes Datenbanksystem im Internet zur Umwandlung von Hostnamen in IP-Adressen) in eine IP-Adresse umwandelt wird. Durch diese IP-Adresse konnte der Server genau lokalisiert werden und es konnte eine Verbindung hergestellt und der Request gesendet werden.
Dies funktioniert heute immer noch so, jedoch durch den rasanten Wachstum des Internets sind IP-Adressen knapp geworden. Deshalb verwendet man heute oft pro IP-Adresse mehrere Hostnamen (virtual Hosts). Man muss also unbedingt das Header-Feld *Host* in den Request einfügen, damit der HTTP-Server weiß an welchen Host er die Anfrage senden soll.

Der dritte Teil der Request-Line („HTTP/1.1“) gibt die verwendetet HTTP-Version an.

Nach der Request-Line folgen die Request Header. Hier kann der Client zusätzliche Informationen über den Request und den Client selbst an den Server senden. So gibt es Header-Felder die lediglich zusätzliche Parameter übergeben, die der Server ignorieren kann (wie z.B. Informationen über die vom Client verwendete Software). Andere Header-Felder müssen vom Server konkret interpretiert werden, so kann beispielsweise auch nur ein Teil eines Dokuments angefordert werden.
(siehe Kapitel Request Header)

4.2.3.1 Request Methoden

OPTIONS: Die Methode OPTIONS fragt die verfügbaren Kommunikations-Optionen ab, die für eine Ressource existieren, oder informiert sich über die Fähigkeiten eines Servers.

GET: Die Methode **GET** wird von einem Client eingesetzt , um die durch eine URL angegebenen Informationen (HTML-Dokumente, Bilder, Videos,...) anzufordern.

Ein **bedingtes GET** erhält man, wenn in den Header-Feldern eine Bedingung angegeben wird. Hierzu gehören unter anderem

> **If-Modified-Since**-Headerfelder: es werden nur Daten übermittelt, die nach einem bestimmten Datum geändert wurden
>
> **If-Unmodified-Since**-Headerfelder: ausführen der Methode nur wenn die Daten nach einem bestimmten Datum nicht mehr verändert wurden

Diese Bedingungen sollen den Netzwerkverkehr verringern, da Cache-Server, welche Dateien zwischenspeichern, auf diese Weise nur dann Daten erhalten, wenn die angefragten Dateien die Bedingungen erfüllen.

Ein **partielles GET** überträgt nur einen Teil eines Dokuments, dies wird im Header-Feld Range angegeben.

Beispiel:

Range: bytes= 0-499	(übertrage die ersten 500 Bytes)
Range: bytes –10	(übertrage die letzten 10 Bytes)

Die Antwort ist eine 206-Status (Partial Content).

HEAD: Die Methode HEAD hat die gleiche Aufgabe wie die Methode GET. Allerdings darf hier der Server keine Daten zurücksenden, sondern nur den Antwort-Header. Dadurch kann ein Client etwa die Größe einer Datei oder die Verfügbarkeit einer Ressource abfragen, ohne das wirklich Daten übertragen werden.

POST: ähnelt der GET-Methode, nur dass ein zusätzlicher Datenblock übermittelt wird. Dieser besteht üblicherweise aus Name/Wert-Paaren, die aus einem HTML-Formular stammen. Grundsätzlich können Daten auch mittels GET übertragen werden (als Parameter im URL), aber die zulässige Datenmenge ist bei POST deutlich größer.

PUT: Die Methode PUT verlangt die Speicherung eines Dokuments unter der URL. Dabei können bestehende Dateien ersetzt werden, oder neue Dateien hinzugeschrieben werden, die unter einer neuen, bisher nicht existenten URL angelegt werden. Die Antwort ist 201 (Created) für die Erstellung einer neuen URL oder 204 (No Content) für die Modifikation eines Dokuments. PUT dient also im wesentlichen dem Aufbau einer Web-Site; die URL beschreibt die jeweiligen

Adressen. Im Gegensatz dazu beschreibt die URL in einem POST-Befehl die Adresse einer Instanz, welche das mitgesandte Dokument bearbeiten soll.

DELETE: Die Methode DELETE löscht die Ressource mit der angegebenen URL. Der Server wird in der Regel die Ressource löschen oder an einen nicht mehr zugreifbaren Ort verschieben und kann hierauf mit 200 (OK) antworten; durch 202 (Accepted) wird angedeutet, dass der Prozess später ausgeführt werden kann.

TRACE: liefert die Anfrage so zurück, wie der Server sie empfangen hat. So kann überprüft werden, ob und wie die Anfrage auf dem Weg zum Server verändert worden ist - sinnvoll für das Debugging von Verbindungen.

4.2.3.2. Request Header

Auch hier nur ein kurzer Überblick der wichtigsten Request-Header:

Accept: Dieser Header beinhaltet alle Formate der Daten, die er akzeptiert.

Dieses Format wird auch als **MIME-Typ** bezeichnet. MIME steht für *Multipurpose Internet Mail Extensions*.

Der MIME-Typ besteht aus zwei Teilen: 1. der Medientyp und 2. der Subtyp (z.B. die Art der Grafik). Beide Typen werden durch einen Schrägstrich voneinander getrennt.

Es gibt folgende Medientypen:

- *text*: für Text
- *image*: für Grafiken
- *video*: für Videomaterial
- *audio*: für Audiodaten
- *application*: für uninterpretierte binäre Daten, Mischformate (z. B. Textdokumente mit eingebetteten nichttextuellen Daten) oder Informationen, die von einem bestimmten Programm verarbeitet werden sollen
- *multipart*: für mehrteilige Daten
- *message*: für Nachrichten, beispielsweise *message/rfc822*
- *model*: für Daten, die mehrdimensionale Strukturen repräsentieren

Beispiel:

Accept: text/html, image/*, image/gif, */*

Durch Angabe des MIME-Types */* werden alle Formate akzeptiert. Falls der Browser Formate nicht anzeigen kann, so werden die Daten zum Download angeboten.

Accept-Charset: Mit diesem Header-Feld wird die Zeichenkodierung (verschiedene Sprachen benötigen unterschiedliche Schriftzeichen) angegeben, die der Client verstehen kann.

Beispiel:

Accept-Charset: ISO-8859-1

ISO-8859-1 gibt beispielsweise die Zeichenkodierungen für alle europäischen Sprachen an.

Accept-Encoding: Hiermit wird angegeben welche Kodierungen der Client kennt. Wenn dieser Header fehlt, dann werden die Daten in reiner Form (so wie sie sind) übermittelt. Andernfalls kann der Server die zu übertragenden Daten z.B. komprimieren um somit die Übertragung zu beschleunigen.
Es gibt verschiedene Komprimierungsalgorithmen, wie z.B.

- x-gzip (HTTP/1.0)
- x-compress (HTTP/1.0)
- gzip (HTTP/1.1)
- compress (HTTP/1.1)
- deflate (HTTP/1.1)

Wenn ein Server eine dieser Codierung anwendet, dann liefert er in seiner Response Nachricht den Content-Encoding-Header, der die Art der Kodierung enthält.

Host: Wie schon im Kapitel 4.2.3 beschrieben ist es zwingend notwendig diesen Header in die Request-Message einzufügen.

Beispiel:
Host: www.beispiel.de:7512

Wenn der Port vom Standart HTTP-Port (80) abweicht muss auch dieser angegeben werden. (in diesem Beispiel Port 7512)

User-Agent: Mit diesem Request-Header werden Informationen über den Client übertragen. Häufig ist dies der verwendete Browser mit Versionsnummer, es wird sogar das installierte Betriebssystem und die Sprache übertragen. Diese Daten können von Server protokolliert werden und somit feststellen welche Browser ihre Benutzer verwenden oder welche Suchmaschinen auf ihren Seiten waren. Dies stellt eine Gefahr dar, da der Benutzer ausspioniert werden kann. Häufig werden die gesammelten Daten für Werbezwecke verwendet.

4.2.4 Response

Eine typische Antwort auf ein Request könnte folgendermaßen aussehen:

```
HTTP/1.1 200 OK
Date: Sat, 17-March-01 11:45:13 GMT
Server: Apache/1.3
Content-type: text/html
Content-length: 91
```

```
<html>
  <title>Hello</title>
  <body>
    Welcome to my world.
  </body>
</html>
```

Die erste Zeile ist hier die Status-Line. Sie enthält zum einen die HTTP-Version und den Status Code. Dieser gibt das Ergebnis der Transaktion wieder und setzt sich zusammen aus einer dreistelligen Zahl und einer kurzen Beschreibung.

Die HTTP-Status-Codes unterteilt man in die Bereiche:

- Codebereich 1xx: Allgemeine Informationen
- Codebereich 2xx: Anfrage des Clients verstanden und erfüllt
- Codebereich 3xx: Anfrage des Clients verstanden, jedoch nicht erfüllbar
- Codebereich 4xx: Anfrage des Clients unvollständig oder fehlerhaft
- Codebereich 5xx: Fehler im Server

Eine Liste mit den definierten Status-Codes befindet sich im Anhang.

Nach der Status-Line folgen die Response-Header–Felder. Diese enthalten Informationen über den Response, die nicht in der ersten Zeile untergebracht werden können. Dies können Informationen über den Server und die angeforderten Daten sein und zwar in der Form [Header]:[Wert des Headers]. Pro Zeile steht immer nur ein Header. Als Ende der Header-Felder steht genau eine Leerzeile.
(siehe Kapitel 4.2.4.1)

Nach der Leerzeile, die dem letzten Header-Feld folgt, überträgt der Server im Body die eigentlichen Informationen, beispielsweise den HTML Code oder die Bytes eines Bildes. Eine Response-Nachricht liefert Daten immer im Textformat. Wenn der Server Daten sendet, die einem anderen Format entsprechen, z.B. Bilder, dann wird das Format im Header (siehe Kapitel 4.2.4.1) angegeben, und der Body erhält dann diese Daten im angekündigten Format.

4.2.4.1 Response Header

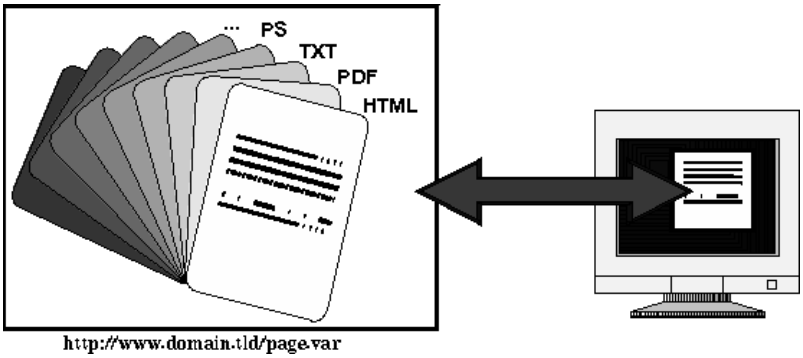
Content-Type: Mit diesem Header-Feld wird der Medientyp des enthaltenen Entity angegeben.

Beispiel:
Content-Type: text/html (Übertragen einer Website)
Content-Type: image/jpeg (Übertragen eines Bildes)

Server: Dieser Header enthält Informationen über den Server, der den Response sendet. Meist sind das der Typ und die Versionsnummer der Serversoftware. Diese Informationen kann der Client beispielsweise für statistische Zwecke nutzen.

Location: Wenn der Client den Auftrag an den Server gibt eine neue Ressource zu erstellen und dabei kein Fehler auftritt, antwortet der Server mit dem Statuscode 201 (created) und gibt die URL der neu erstellten Ressource mit Hilfe dieses Header-Feldes an.

4.2.5 Content Negotiation



Es ist möglich, dass eine Ressource (z.B eine „*.pdf“ Datei) in verschiedenen Varianten auf dem Server vorliegt, jedoch immer den gleichen Inhalt besitzt.
Wann ist es vorteilhaft verschiedene Varianten einer Ressource auf einer Webseite vorzuhalten?

Bei:

- Sprachspezifischen Varianten
  Eine Inhaltlich gleiche Ressource kann in verschiedenen Sprachversionen vorliegen. Zum Beispiel: eine mehrsprachige Bedienungsanleitung oder ein Lehr-Video auf einer internationalen Webseite.

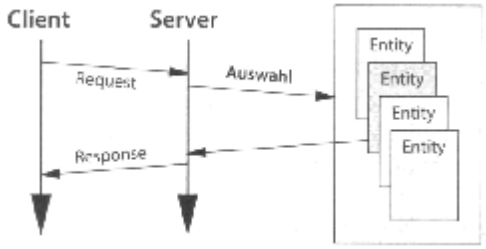
- Qualitätsspezifischen Varianten
  Abhängig von der Kapazität der Internetverbindung des Clients können unterschiedliche Datenmengen dem Benutzer zumutbar sein z. B. 10MB für ISDN-Nutzer, 200MB für DSL-Nutzer. So kann beispielsweise der DSL-Nutzer ein Video mit einer höherer Auflösung betrachten.

- Codierungspezifischen Varianten
  Anhängig von den Fähigkeiten des Client-Computers kann die Ressource in verschiedenen Datenformaten dargestellt werden. So können „DivX“ Videos nur mit einem speziellen Codec dargestellt werden.

Das Prinzip der Content Negotiation beruht darauf, dass es verschiedenen Varianten einer Ressource vorhanden sein können, auf die mit nur einer Referenz verwiesen wird, da diese vom Inhalt und nicht von Sprache, Darstellung oder Codierung abhängt. HTTP stellt Methoden zur Verfügung, die eine Auswahl aus den verschiedenen Varianten ermöglichen. Es existieren 3 Möglichkeiten von Content Negotiation.

4.2.5.1 Server-Driven Negotiation

Die Auswahl der Ressourcenvariante wird vom Server vorgenommen. Er kann dabei auf die folgenden Informationsquellen zurückgreifen:



- Verfügbare Darstellungsformen
  Der Server kennt alle Varianten der Ressource, ihre Dimensionen und Eigenschaften

- Request-Header-Felder
  Der Client kann im Request-Header Informationen über seine Ressourcen übermitteln. Accept, Accept-Language, Accept-Charset, Accept-Encoding, User-Agent
  Beispiel-Header:
  Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6, image/jpeg; q=0.6, image/*; q=0.5, */*; q=0.1

- Andere Informationen
  z.B. Netzwerkadresse

Wenn solch eine Ressource im Cache abgelegt wird, muss sie das Header-Feld „Vary“ enthalten, welches die vom Server zur Auswahl verwendeten Kriterien enthält.

Die Nachteile des Verfahrens sind:

- Begrenztes Wissen des Servers
  Die Kenntnisse des Servers reichen oft nicht aus, da er nur wenige Informationen über den Client verfügt. Der Client weiß nicht welche Informationen der Server benötigt und schickt nur allgemein definierte Informationen an den Server.

- Ineffizienz
  Der Client sendet immer die vom Server zur Content Negotiation benötigten Header-Daten. Obwohl Content Negotiation nur bei einem kleinen Teil der Requests verwendet wird.

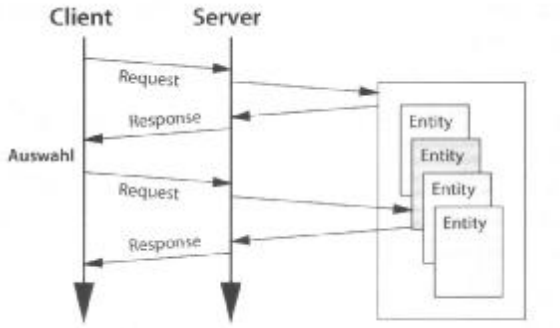
- Komplizierte Server-Implementierung
  Dem Server müssen, um Server-Driven Content Negotiation bearbeiten zu können, entsprechende Fähigkeiten implementiert werden. Darüber hinaus beansprucht Content Negotiation auf dem Server Rechenleistung. Dies kann bei stark belasteten Servern (im Vergleich zur Clientbelastung) zu Leistungsabfall führen.

- Caching
  Da der Response abhängig von einem Server-internem Auswahlprozess ist, kann er häufig nicht im Cache zwischengespeichert werden, da er nur eine

Variante des Response speichert und nicht alle, die zur einer erneuten Auswahl vorhanden seim müssen.

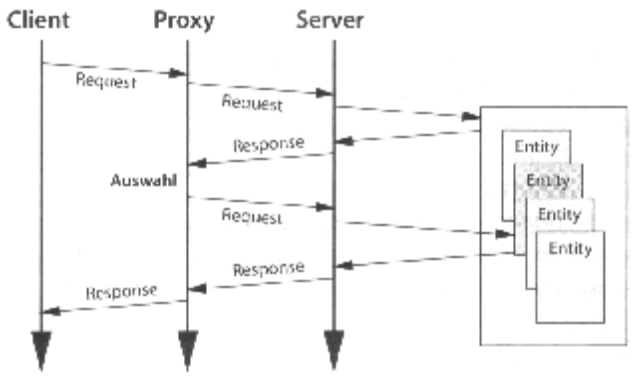
4.2.5.2 Agent-Driven Negotiation

Bei der Agent-Driven Negotiation antwortet der Server auf den Request mit einem Statuscode 300 (multiple choices) und einer Liste aller Varianten mit ihren URIs der angeforderten Ressource. Der Client wählt eine Ressourcenvariante aus der Liste aus und fordert sie in einem zweiten Request vom Server an. In der aktuellen HTTP Spezifikation existiert keine Möglichkeit eine Variante automatisch auszuwählen. Die Auswahl muss manuell vom Clientbenutzer getroffen werden, indem ihm z.B. die übermittelte Liste mit den Varianten gezeigt wird und er durch „onClick“ die Variante auswählt. Für eine zukünftige HTTP-Spezifikation ist das Feld „Alternatives“ reserviert, um ein Format für die vom Server gesendeten Varianten zu definieren. Wird vom Server eine Variante bevorzugt, kann diese im Feld „Location“ angegeben werden. Der "Location"-Header kann auch die Adresse der vom Server bevorzugten Variante enthalten. Der Client kann dann trotzdem die Auswahl anzeigen, oder direkt die, in „Location“ angegebene, Variante aufrufen.



4.2.5.3 Transparent Negotiation

Transparent Negotiation ist eine Mischung aus der Server- und der Agent-Driven Content Negotiation. Dabei schickt der Client über ein Proxy seinen Request an den Server. Dieser antwortet, wie bei der Agent- Driven Content Negotiation, mit einer Liste der Variationen. Der Response wird vom Proxy empfangen und es wird von ihm eine Variante ausgewählt. Dabei hat er die gleichen Informationen wie der Server bei der Server-Driven Content Negotiation. Er verschickt einen zweiten Request und leitet den Response mit der ausgewählten Entity an der Client weiter.



Die Vorteile dieses Verfahrens sind:

- Die Serverlast wird verringert, da der Proxy die Auswahl übernimmt.

- Der Netzwerkverkehr und die Reaktionszeit auf den ersten Request ist kleiner, da der erste Response nicht bis zum Client durchgeleitet werden muss.

Der Nachteil ist:

- In der aktuellen HTTP Version ist dafür kein allgemeingültiges Verfahren implementiert. Das bedeuten dass die bisher realisierten Lösungen proprietär sind.

4.2.7 Persistente Verbindungen

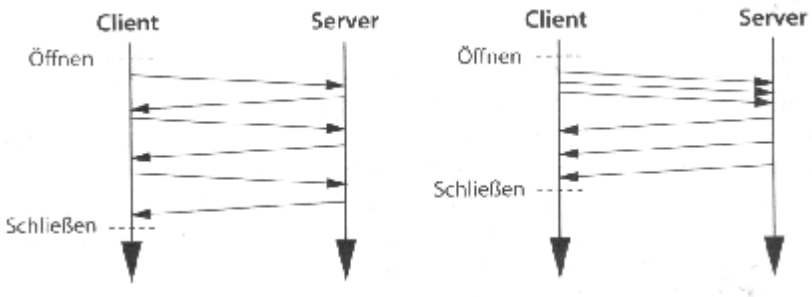
Bei HTTP 1.0 waren nur ein Request und ein Response pro Client/Server-Verbindung erlaubt. Dies führte dazu, dass die Clients viele HTTP-Verbindungen zur Beschleunigung der Datenübertragung gleichzeitig aufbauten. Das hatte zur Folge, dass die Netzwerklast stark anstieg, da das Aufbauen und Schließen einer TCP-Verbindung viel Datenverkehr erzeugt und Wartepausen beinhaltet. Außerdem wurden andere Protokolle, wie FTP, oder Telnet, die nur eine Verbindung benutzen, benachteiligt. Um diese Nachteile auszumerzen, wurden in HTTP 1.1 persistente Verbindungen eingeführt. Sie erlauben maximal zwei Verbindungen gleichzeitig pro Client, im Gegenzug muss nach einem Request/Response Datenwechsel die Verbindung nicht geschlossen werden, sondern kann für weitere Datenwechsel genutzt werden.

Da das Ende einer Nachricht und den Anfang der nächsten nicht durch den zwischenzeitliche Verbindungsabbau gekennzeichnet sind, muss die Länge des Entitys im Header-Feld „Content-Length“ angegeben werden. Zusätzlich muss das Ende der Verbindung nicht automatisch mit dem Ende des Response eintreten, denn eine persistente Verbindung kann theoretisch unendlich lange dauern. Deswegen müssen sowohl dem Client, als auch der Server das Ende der Verbindung kennen. Dies wird mir dem Header-Feld „Connection“ und seiner Option „close“ angegeben. Wenn diese Information von einem Teilnehmer erhalten wird, wird nach dem nächsten Response die Verbindung abgebaut.

Die Vorteile, die sich aus persistenten Verbindungen ergeben sind:

- Betriebssystemressourcen
  Das Auf- und Abbauen einer TCP-Verbindung erfordert zusätzliche Rechenzeit, die eingespart werden kann und eingespart werden können auch die TCP Control Blocks, die nach dem Abbauen der TCP-Verbinding noch aufrechterhalten werden müssen.

- Pipelining
  Es ist möglich mehrere Requests abzusenden, ohne die Responses des vorhergehenden Requests abzuwarten, dies

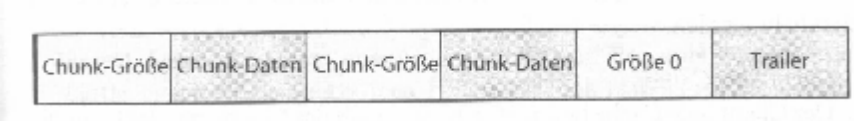


lastet die Verbindung optimal aus, da die Wartezeit auf die Responses entfällt. Es wäre auch wünschenswert, dass die Responses in einer beliebigen Reihenfolge zurückkommen könnten, da z.B. einer von ihnen im Server Side Cache liegt und damit viel schneller bearbeitet werden würde als die vor ihm losgeschickten Requests. Dies ist jedoch in HTTP 1.1 nicht möglich, alle Responses müssen in der Reihenfolge der abgesandten Requests ankommen.

- Weniger Pakete
  Jeder Verbindungsauf- und abbau verursacht zusätzlichen Datenverkehr. Durch persistente Verbindungen sind weniger Verbindungen nötig, was diesen Datenverkehr minimiert.

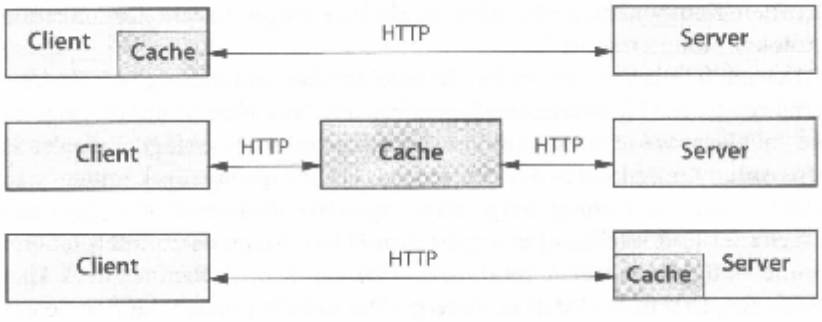
- Weniger Leistungsverlust bei Unverträglichen HTTP-Versionen
  Wenn der Client eine neuere HTTP-Version verwendet als der Server, führt dies zwar zu einer Störung, aber die Verbindung wird nicht gleich abgebrochen. Der Client kann darauffolgend den Request mit einer älteren Semantik (z.B HTTP 1.0) versenden.

4.2.8 Chunced Encoding



Bei persistenten Verbindungen wurde das Ende der Verbindung durch das Header-Feld „Content-Length“ mitgeteilt. Manchmal ist es aber nicht von vornherein bekannt, wie lang die Entity ist. Dann wird das Chunked Encoding nach dem oberen Prinzip benutzt. In dem Header-Feld „Transfer-Encoding“ wird die Länge des nächsten Teilstückes angegeben. Danach wird wieder die Chunk-Größe angegeben, bis das Entity zu Ende ist, und die Chunk-Größe = 0 ist. Danach folgt optional ein Trailer, der z.B. MD5-Codiertes Entity um die Integrität dessen zu überprüfen enthält.

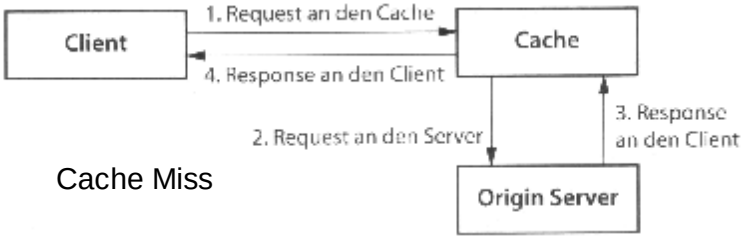
4.2.9 Caching



Durch das Caching versucht man den Zugriff auf Internetinhalte zu Beschleunigen und gleichzeitig die Verkehrsaufkommen im Internet zu verringern. Ein Cache ist ein relativ schneller Speicher zur temporären Speicherung von Responses.

Man unterscheidet dabei drei Arten von Caching:

- Client Side Cache
  Der Cache befindet sich innerhalb des Clients. Meistens kann er im Browser durch den „Zurück“ Button, oder durch den Aufruf des „Verlaufs“ aufgerufen werden.



Cache Miss

- Server Side Cache
  Der Cache innerhalb des Server und speichert die versendeten Requests. Er ist jedoch wesentlich schneller als der Server und sein Datensystem und kann einen Teil der Requests abfangen, bearbeiten, und so den Server entlasten.

- Cache zwischen Client und Server
  Der Cache befindet sich irgendwo zwischen Client und Server und arbeitet autonom. Zur Benutzung des Caches muss der Client die Requests an den Cache und nicht an den Server senden. Dieser entscheidet, ob er den Request an den Server weiterleitet (Cache Miss), da er das benötigte Entity nicht gespeichert hat. Ob er den Request an einen anderen Cache weiterleitet, oder ob er das benötigte Entity hat, und selber den Request beantworten kann (Cache Hit).

Cache Hit

Das Client- und das Server-Side Caching ist durch direkte Implementierung in das jeweilige System realisierbar und ist beim Betrachten des HTTP Protokolls weniger interessant. Hier soll der zwischen Client und Server angesiedelte Cache näher betrachtet werden.

Anhand eines Beispiels werden hier die Vorteile als auch die Probleme beim Caching erläutert:

Ein fiktives, kleineres Maschinenbau-Unternehmen hat einen Cache-Computer eingerichtet, über den alle Requests laufen, die von Mitarbeitern aus dem Firmennetzwerk uns Internet gehen. Vor der Einrichtung gingen alle Requests an den Server im Internet und wurden von dem zuständigen Server beantwortet. Dies hat hohe Kosten verursacht da häufig auf die Datenbanken bestimmter Zulieferer zugegriffen wurde. Die Anfragen konnten nur zu einem kleinen Teil aus dem Client Side Cache beantwortet werden, da jeder Client für sich nur selten mehrmals auf die gleiche Entity zugriffen hat. Nach dem Einrichten des oben genannten Caches müssen weit weniger Requests ins Internet „durchgestellt“ werden, da die

Wahrscheinlichkeit, dass einer der Mitarbeiter vor kurzem ein Request gestellt und dieser dann im Cache gespeichert wurde, höher liegt. Ruft ein anderer Mitarbeiter, dessen Requests ja über den gleichen Cache-Computer laufen, die gleiche Seite auf, so kann sie aus dem Cache bedient werden. So wird nicht nur das WWW-Netzwerk entlastet und Internetkosten gespart, sondern die Geschwindigkeit des Response wird erhöht, da der Cache-Computer unmittelbar im Unternehmen angesiedelt ist und der Datenweg somit wesentlich kürzer ist.

Doch was passiert wenn der letzte Zugriff auf eine Entity Tage oder Wochen zurückliegt und diese Version im Cache abgespeichert ist? Würde der Mitarbeiter dann eine veraltete Datenbank angezeigt bekommen? Eine, die ein Produkt als „Lieferfähig“ anzeigt, obwohl es mittlerweile ausverkauft ist? Um diese Probleme beim Caching zu verhindern, werden das Fälligkeits- und das Gültigkeitsmodell verwendet.

4.2.9.1 Fälligkeitsmodell

Beim Fälligkeitsmodell unterscheidet der Cache zwischen einem „fresh“-en und einem „veralteten“ Response. Der Cache überwacht diese Lebensdauer der Responses und benutzt nur solche, die „fresh“ sind, außer, wenn der Server mal nicht erreichbar ist, oder der Client ausdrücklich ein veralteten Response haben will. Der Zustand des Response kann dabei auf zwei Wegen bestimmt werden

- Server-bestimmte Fälligkeit
  Der Server definiert in dem Header-Feld „Expires“(Zeitpunkt), oder „Cache-Control“(Zeitraum) und der Anweisung „max-age“ einen Zeitpunkt ab dem der Response nicht mehr “fresh“ ist.
  Will der Server dass der Response nie „fresh“ ist, z.B. bei Börsenkursen, dann kann er einen Zeitpunkt in der Vergangenheit angeben, oder in das Header-Feld „Cache-Control“ die Anweisung „must-revalidate“ einsetzen. Dann wird nur in den beiden, oben genannten Ausnahmen der „verbrauchte“ Response verwendet.

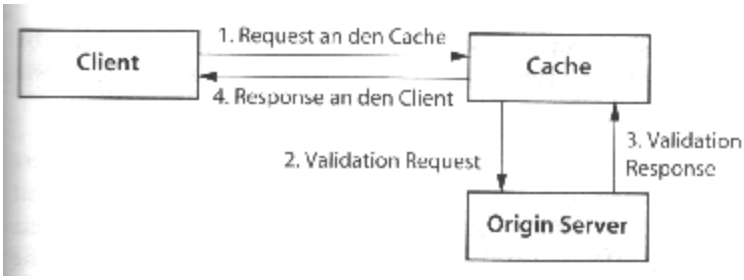
- Heuristische Fälligkeit
  Es ist nicht zwingend notwendig, dass der Server die Fälligkeit bestimmt. Falls der Cache keine Informationen über die Lebensdauer des Response erhält, dann bestimmt er mit Hilfe eines heuristischen Verfahrens sie selber. Als Grundlage kann dazu das Header-Feld „Last-Modified“ benutzt werden. Bei einer erwarteten „fresh“-ness von mehr als 24 Stunden wird aber dem Client der Warncode 113 (heuristic expiration) gezeigt. Dieses Verfahren birgt das Risiko, dass entgegen aller Wahrscheinlichkeit das Entity auf dem Server sich in der Zwischenzeit verändert hat und der Cache-Request nicht mehr aktuell ist.

Die Fälligkeit eines Response wird ermittelt indem das Alter mit der „fresh“-ness-Dauer verglichen wird. Dabei sollte Client, Server und der Cache über eine einheitliche, festgelegte Zeitbasis verfügen. Diese kann durch das Network Time Protokoll (NTP) ermittelt werden.

Das Alter des Response kann durch das Header-Feld „Date“ vom Server definiert werden, oder durch das Feld „Age“, indem der/die Cache(s) die verstrichene Zeit aufsummieren.
Die „fresh“-ness-Dauer entspricht dann entweder dem im „Cache Control“ angegebenen Zeitraum, oder „Expires“ - „Date“.

4.2.9.2 Gültigkeitsmodell

Was passiert mit einem Response, der „Verbraucht“ ist? Muss beim einem neuen Request zwangsläufig ein neuer, ein „fresh“-er Response vom Server geholt werden?
= Nein



In vielen Fällen ist das Entity auf dem Server beim Ablaufen der “fresh“-ness auf dem Server immer noch unverändert, da sowohl der Server bei der Server-bestimmten Fälligkeit, als auch der Cache bei der heuristischen Fälligkeit, den Zeitraum bis zur Veränderung nur schätzen. Wenn das so ist, ist es manchmal gar nicht notwendig die Entity immer zu übertragen. Es muss nur festgestellt werden, ob sich die Entity auf dem Server verändert hat. Dabei geht der Cache folgendermaßen vor: Er schickt einen sogenannten Validator zum Server zusammen mit einem neuen Request, weil der im Cache liegende Response nicht mehr „fresh“ ist und der Client einen neuen anfordert. Diesen Validator hat er mit dem mittlerweile „verbrauchten“ Response erhalten. Es gibt zwei Arten von Validatoren:

- Header-Feld: „Last-Modified”
  Dort speichert der Server den letzten Zeitpunkt, wann das Dokument zuletzt verändert wurde.

- Header-Feld: „ETag“
  Im ETag wird eine einzigartige Kennung des Entity gespeichert. Sie verändert sich entweder bei der kleinsten Veränderung der Entity („Strong Validator“), oder bei jeder semantisch relevanten Änderung („Weak Validator“). Dabei sollte beachtet werden, dass sowohl der Cache, als auch der Server möglichst den selben Validatortyp verwenden.

Der Server vergleicht entweder das Feld „Last-Modified“, oder das Feld „ETag“ des Requests mit den Feldern der Entitys, die aktuell bei ihm gespeichert sind. Wenn diese identisch sind, sendet er einen Response mit dem Statuscode 304 (not modified), aber ohne das Entity. Wenn sich eins der beiden Felder verändert hat, sendet er einen normalen Response mit dem Entity.

Ein Cache mit einem Gültigkeits- und Fälligkeitsmodell können die Leistungsfähigkeit wesentlich verbessern. Dazu bedarf es jedoch einer Mitarbeit des Servers und des Clients, ohne die kein Caching möglich ist.

4.2.9.3 Cache-Steuerung

Das wichtigste Header-Feld zur Cache-Steuerung ist „Cache-Control“, in ihm können unterschiedliche Informationen, sowohl für den Request, als auch für den Response abgespeichert werden. Diese Informationen können in folgende Gruppen unterteilt werden:

- Wo lässt sich der Response zwischenspeichern?
  - Public, in jedem Cache
  - Private, in einem Cache, auf den nur ein Client Zugriff hat (Client Side Cache)
  - no-cache, der Cache darf die angegebenen Feldnahmen nicht speichern
  - no-store, der Response darf auf keinen Fall in Cache gespeichert werden

- Veränderung der Fälligkeit
  - max-age, maximales „fresh“-ness-Alter des Response
  - min-fresh, Bedingung wenn der Response für den Client noch „fresh“ sein soll
  - max-stale, Zeitdauer des „veraltet“-seins eines Response, der vom Cliennt trotzdem noch akzeptiert wird

- Veränderung der Revalidierung
  - no-cache, der Response muss, wenn keine Feldnamen angegeben werden, auch Revalidiert werden, wenn er “fresh“ ist
  - must-revalidate, der Cache muss bei Ablauf der „fresh“-ness auch ohne neuen Request den Response revalidieren
  - proxy-revalidate, wie must-revalidate, aber der Client Side Cache darf „veraltete“ Responses benutzen
  - only-if-cached, der Cache gibt, wenn vorhanden, den Response ohne Revalidierung oder einem neuem Response vom Server, den gespeicherten Response an den Client, ansonsten schickt er 504 (gateway time-out)

- Verbieten der Transformation
  - no-transform, der Cache darf auf keinen Fall den Response verändern, um ihn z.B. zu komprimieren

4.2.9.4 Caching-Implementierung

Die im vorhergehenden Abschnitt erläuterten HTTP Funktionen definieren zwar Regeln, die sicherstellen dass Informationen ausgetauscht und verstanden werden können. Die Implementierung des Cache beeinflusst aber entscheidend das erzielte Ergebnis.

Es existieren daher zusätzliche Arten den Cache einzusetzen.

- Kein Caching
  Die HTTP Spezifikationen ermöglichen zwar ein Speichern von Caches und verhindern das Benutzen von „verbrauchten“ Requests, aber ein Cache muss nicht Responses speichern. Er kann dann trotzdem als Zwischenstation fungieren. Solch ein Cache bringt jedoch keine Vorteile mit sich, da er weder die Netzwerklast senkt, noch die Antwortzeit auf Requests verringert. Er hat aber den Nachteil, dass er für die Verarbeitung der Requests und Responses Zeit benötigt und damit die Antwortzeit auf Requests verlängert.

- Aktive Caches
  Die oben aufgeführten Cache sind meistens reaktiv. Das heißt sie reagieren nur auf Requests vom Client und sind von sich aus nicht tätig. Aktive Caches reagieren von sich aus, sie ersetzen oder revalidieren „veraltete“ Responses ohne einen Request. Das hat den Vorteil, dass der Response immer beim eintreffen eines Requests aus dem Cache bedient werden kann, aber auch den Nachteil dass, wenn der passende Request nicht abgerufen wird, der Cache beim Re-“fresh“-en des Responses unnötigen Datenverkehr erzeugt hat.

- Propietäre Inter-Cache-Protokolle
  Ein Cache kann auch aus einer Reihe miteinander verschalteter Caches erzeugt werden. Diese Caches müssen sich bei der Kommunikation untereinander nicht an das HTTP-Schema halten und können ein fortschrittlicheres propietäres Protokoll einsetzen und somit ein effektiveres Caching-System bilden. Nach außen hin kommunizieren sie mit dem HTTP-Protokoll

4.3 Sicherheit bei HTTP

Bei der Sicherheit kann zwischen zwei Aspekten unterschieden werden. Zum einen die
Authentifizierung des Clients, damit er bestimmte Informationen vom Server erhalten oder Daten an den Server übertragen darf. Der zweite Aspekt bei der Sicherheit ist die, vor feindlicher Einflussnahme sichere, Übertragung der Daten zwischen Client und Server.

4.3.1 Authentifizierung

Das Web ist in der Regel anonym, dass bedeutet dass der Client ohne Identifizierung ein Request aussendet, oder per „PUT“ oder „POST“ Daten zum Server überträgt. Manchmal ist es jedoch notwendig, dass die Clients und/oder die dahinterliegenden Personen identifiziert werden. z.B. bei „PUT“/“POST“ dass nur der Administrator des Servers neue Inhalte einstellen darf oder dass nur der Besitzer eines Kontos Zugriff auf das Konto übers Internet hat.

Es gibt dabei 3 Stufen der Authentifizierung:

- Identifikation:
  Identifikation einer Person oder Personengruppe. Es wird darauf vertraut, dass sich hinter der Identität der assoziierte Benutzer verbirgt. Dies wird meistens durch einen Benutzernamen realisiert.

- Authentifizierung:
  Sicherstellung dass kein Benutzer unberechtigterweise die Identität eines anderen Benutzers annimmt. Dies wird meistens durch ein Passwort realisiert.

- Authorisierung:
  Definieren eins Rechte-Katalogs für einen Identifizierten und Authentifizierten Benutzer. z.B. Benutzer XY hat das Recht Datei A zu lesen, Datei B zu verändern und aus Datei C zu kopieren.

4.3.1.1 Basic Authentication

Die Basic Authentication ist ein sehr einfaches, in HTTP1.0 verwendetes, Authentifizierungs-schema. Wenn ein Client eine Ressource anfordert, zu deren Zugriff er nicht berechtigt ist, antwortet der Server mit dem Fehlercode 401 (unathorized) und dem Header-Feld „WWW-Authenticate“ der ein Schema der Authentifizierung und ein Realm Value angibt. Der Realm Value legt die Protection Space des Servers fest, von dem der Client-Request betroffen ist. Der Client sollte diesen Realm Value anzeigen, wenn er den Benutzer zur Eingabe des Passworts auffordert. Vom Client wird dann erwartet, dass er die ihm mitgeteilten Parameter verwendet, um in einen neuen Request, mit dem Header-Feld „Autorisation“ mit den Autorisierungsangaben des Requests, der Benutzeridentifikation und dem Passwort des Benutzers zu antworten. Die größte Schwäche der Basic Authentication ist, dass die Identifikation und das Passwort des Benutzers im Klartext über das Netzwerk übertragen werden.

4.3.1.2 Digest Access Authentication

Bei diesem Authetifizierungsschema als Bestandteil der HTTP-Version 1.1, wird vermieden, dass das Benutzerpasswort in Klartext über das Netzwerk übertragen wird. Die Grundlage der Digest Access Authentication basiert auf Funktionen, welche die Berechnung der Eingabe auch dann unmöglich machen, wenn die Ausgabe vorliegt. Dazu wird das Passwort mit dem MD5-Algorithmus codiert. Um zu

verhindern, dass das verschlüsselt übertragene Passwort zu einem Angriff auf den Server verwendet wird, werden die folgenden Informationen codiert:

- Benutzername
- Passwort
- Realm,
  hilft dem Client bei der Eingabe des Benutzernamens und des Passwortes
- Nonce,
  enthält einen die Client-IP-Adresse, einen Zeitstempel und einen persönlichen Server-Schlüssel, um einen „Massive Dictionary Attack“, bei dem alle Wörter aus einem Wörterbuch als Passwort bei einem Benutzerkonto automatisch ausprobiert werden, zu verhindern.
- HTTP-Methode,
  z. B. “GET”
- Angeforderte URI,
  * = keine bestimmte URI, oder eine absolute URI/Pfad

Der Server verschlüsselt die gleichen, ihm vorliegenden Informationen und vergleicht sie dann mit den empfangenen. Sind die beiden verschlüsselten Informationen identisch, dann ist die Authentifizierung erfolgreich.
Eine weitere Möglichkeit ist es die Entity neben der Klartextübertragung auch verschlüsselt zu übertragen, das ermöglicht es die Fehlerfreiheit der Entity bei speziellen, kritischen Übertragungen zu gewährleisten.

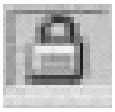
4.3.2 Geheimhaltung

Neben der Authentifizierung ist es bei der Sicherheit wichtig, ein Abhören oder ein Abfangen des Datenstroms zu verhindern. Dabei sollte abgewogen werden, welcher Angriff zu erwarten ist. Beim Abhören (Lauschangriff) kann der Angreifer die übertragenen Informationen erfassen, aber nicht verändern. Beim Abfangen des Datenstroms (Man-In-The-Middle-Angriff) kann er dagegen die Daten gezielt verändern und dann weiterschicken (Pinching). Um zu verhindern dass dies passiert können zwei Verfahren verwendet werden.

- Sichere Transportarchitektur (HTTPS)
- Sicheres Protokoll auf Anwendungsebene (S-HTTP)

4.3.2.1 HTTP über SSL (HTTPS)

Bei der Suche nach einer Möglichkeit zum Bereitstellen sicherer Transaktionen im Web beschloss Netscape, eine eigene Lösung zu implementieren HTTPS. Bei HTTPS wird davon ausgegangen, dass die Transportinfrastruktur sicher ist. HTTPS wird in Form einer zusätzlichen Schicht, die zwischen der verwendeten TCP/IP-Schicht und HTTP als Anwendungsprotokoll liegt, realisiert. Der Vorteil dieses Verfahrens ist es, dass es dann die



HTTPS-Symbol

Sicherheit für alle Anwendungen bietet, die von ihr benutzt werden. SSL besitzt dabei drei grundliegende Verfahrensschritte, die die Sicherheit der Verbindung gewährleisten sollen.

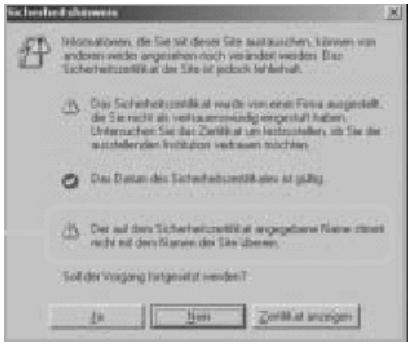
- Verbindungssicherheit,
  nach einem Hand-Shake, wird ein geheimer, symmetrischer Schlüssel vereinbart
- Optionale Authetifizierung,
  die Identität kann mit einem öffentlichem, asymmetrischem Schlüssel vereinbart werden
- Zuverlässigkeit einer Verbindung,
  Eine Nachrichtenübertragung wird anschließend mit Hilfe eines Hash-Wertes auf Integrität geprüft.

Wenn diese drei Vorraussetzungen erfüllt sind, kann SSL (Secure Socket Layer) die folgenden Aufgaben erfüllen:

- Kryptografische Sicherheit,
  Die Verbindung ist ohne die Schlüssel nicht entschlüsselbar.
- Interoperabilität,
  SSL soll auch zwischen Client und Server funktionieren, die nicht die SSL-Software der Gegenstelle kennt.
- Erweiterbarkeit,
  SSL soll erweiterbar und anpassbar an neue Anforderungen sein.
- Relative Wirksamkeit,
  SSL erzeugt eine hohe Rechen- und Netzwerklast, deswegen sollten durch Caching diese Belastungen vermindert werden.

Es lassen sich mit Hilfe von SSL drei unterschiedliche Arten von Verbindungen zwischen Client und Server aufbauen, die sich bezüglich des jeweils eingesetzten Authentifizierungs-verfahren unterscheiden.

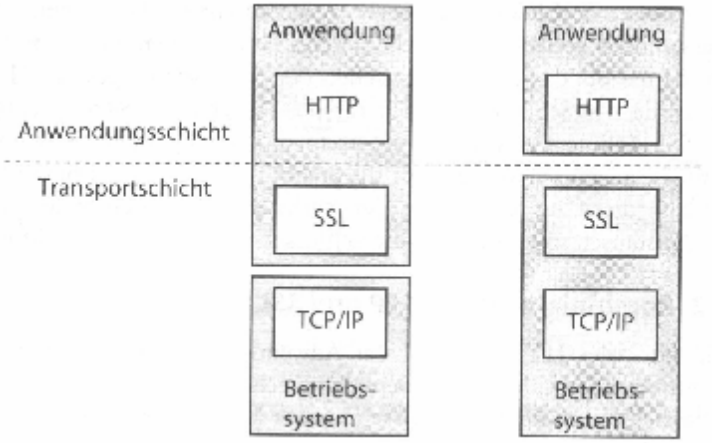
- Anonymität,
  Weder Client, noch der Server sind authentifiziert, dieses Verfahren bietet Schutz vor einem Lauschangriff, während ein Man-In-The-Middle-Angriff immer noch möglich sind.
- Server-Authentifizierung,
  der Server ist durch ein Zertifikat authentifiziert, dies erhöht die Sicherheit für den Client gegenüber einem Man-In-The-Middle-Angriff
- Authentifizierung beider Parteien,
  beide Partner sind authentifiziert und können sich sicher über die Identität des anderen sein.



Zertifikat

Entscheidend für die Sicherheit ist dabei die Echtheit der ausgetauschten, signierten Zertifikate. Bei Nichtbenutzung oder Fälschung eines Zertifikats kann sich ein Angreifer als die Gegenstelle ausgeben und einen Man-In-The-Middle-Angriff durchführen.

Obwohl normales HTTP das auf der Anwenderschicht zwischen einem HTTP einsetzenden Client und Server eingesetzte Protokoll darstellt, muss der Client wissen, dass er anstelle einer normalen (unsicheren) TCP-Verbindung eine SSL-Verbindung zu einem Server aufbauen muss. Dies wird mit Hilfe eines neuen Naming Schemes für HTTPS erreicht, in dem das Präfix "https" für URLs definiert ist.



HTTPS kann als Protokoll zwischen der Transport- und der Anwendungsschicht sowohl vom Betriebssystem, als auch von der Anwendung implementiert werden. Sinnvoller ist es in das Betriebssystem zu Implementieren da es sonst seinen Vorteil gegenüber S-HTTP, dass es Anwendungs-übergreifend zur Verfügung steht, verloren geht.

4.3.2.2 Secure HTTP (S-HTTP)

S-HTTP ist ein um Sicherheitsmerkmale erweitertes HTTP, es kapselt die HTTP-Nachrichten gegen unberechtigte Zugriffe und ist kaum verbreitet.

4.4 Cookies

HTTP ist ein zustandsloses Protokoll, d.h. das der Austausch von HTTP-Nachrichten nicht in einen größeren Rahmen eingebettet ist, z.B.: wie bei einer Sitzung wo mehrere Request / Response-Interaktionen stattfinden. Vorangegangene Interaktionen haben keinen Einfluss auf die zwischen Client und Server stattfindenden Request / Response-Interaktionen. Da dies bei bestimmten Anwendungen sinnvoll wäre, entstand eine von Netscape erstellte Lösung, welche ohne ersichtlichen Grund Cookie genannt wurde. Der Begriff stammt aus dem Amerikanischem Englisch und heißt übersetzt Plätzchen oder Keks.
Es handelt sich bei ein Cookie um einen Austausch von Informationen über den Zustand der Sitzung die nicht von HTTP abgedeckt werden. Die Eigenschaften eines Cookies sind im veralteten RFC 2109, HTTP State Management Mechanism, hinterlegt. Die neuere Festlegung ist der RFC 2965.

Anhand des Cookie-Mechanismus wird eine logische, nicht physikalische Verbindung zwischen Client und Server hergestellt, nicht physikalisch, weil es keinen Bezug zu einer physikalischen Entsprechung (z.B. eine persistenten Verbindung) gibt.

Cookies lassen sich unterscheiden in persistente Cookies, die über einen festgelegten Zeitraum auf der Festplatte gespeichert werden, und Session Cookies, die nur für die Länge einer Sitzung gespeichert werden.

Der Prozess des State Management wird in nachstehendes Schema dargestellt:

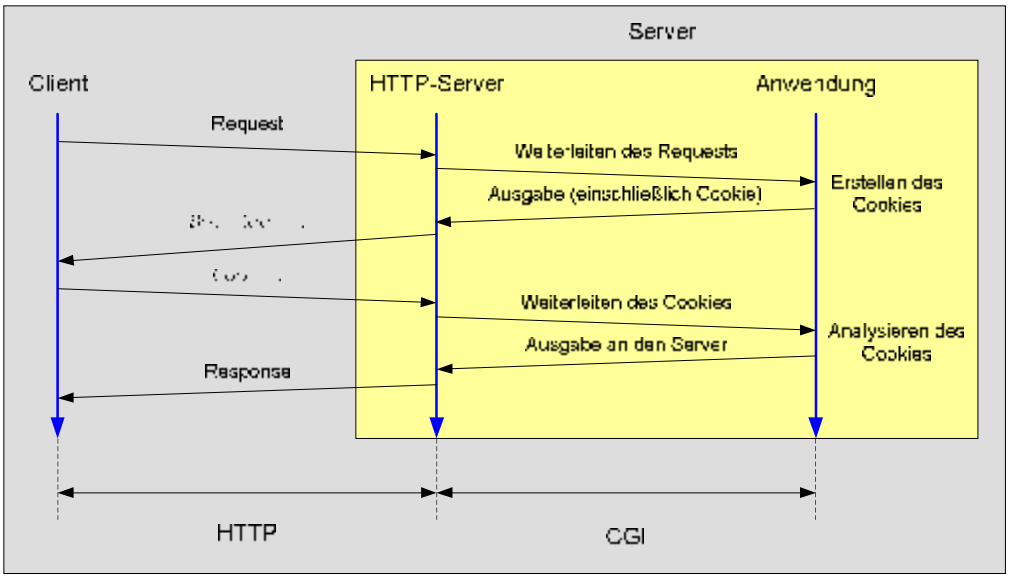


Bild 4.4: State Management mit Cookies

Bei der Darstellung wird davon ausgegangen das die Verarbeitung des Cookies auf den Server von eine separate Anwendung vorgenommen wird.

`- set cookie`

Der Response-Header `set cookie` wird vom Server verwendet, um ein Cookie auf ein Client einzurichten. Mit anderen Worten, dem Client wird ein Cookie gesendet, welches er speichern soll, sofern er die Funktion unterstützt, andernfalls wird das Headerfeld `set cookie` einfach ignoriert.

`- cookie`

Ein Client sendet Cookies mit Hilfe des Request-Headers `cookie` an einem Server. Welches Cookie an welchen Server gesendet werden soll, basiert auf eine entscheidung des Clients, welche vom Namen des Servers, der angeforderten URI und dem Alter des Cookies abhängig ist. Bei Erfüllung aller Entscheidungskriterien schließt der Client ein oder mehrere Cookies mit Hilfe des Request-Headers `cookie` in seinen Request ein.

Jeder Cookie muss folgende Felder enthalten:

- `<name>`
  Beliebiger Wert vom Server, welcher oft in ASCII-Kode dargestellt wird
- `<version >`
  Gibt die Cookie Management Specification in einer Dezimalzahl an

Weitere optionale Möglichkeiten:

- `<expires>`
  Ablaufdatum, Zeitpunkt der automatischen Löschung nach GMT (HTTP/0.9)
- `<max-age>`
  Ablaufzeit in Sekunden (HTTP/1.0)
- `<domain>`

Domain oder Bestandteil des Domainnamens, für den der Cookie gilt
- `<path>`
  Teil der Anfrage-URI, um die Gültigkeit des Cookies auf einen bestimten Pfad zu beschränken
- `<valid-from>`
  Zeitpunkt ab dem der Cookie gültig ist in GMT (HTTP/0.9)
- `<comment>`
  Kommentar für die nähere Beschreibung des Cookies
- `<secure>`
  Empfehlung für einen die Privatsphäre schützenden Umgang, wenn der Inhalt des Cookies vom Browser über HTTPS statt über HTTP zurückgesandt werden soll

Verwendung finden Cookies in Foren, Online-Shops. Bei Foren müssen dann persönliche Einstellungen nicht wieder von Neuem eingegeben werden, bei Online-Shops ermöglichen sie den virtuellen Einkaufskorb.
Die Gefahr besteht allerdings das Benutzerprofile über das Surfverhalten der jeweiligen Clients erstellt werden können, Marketingfirmen können über mehrere Seiten hinweg Benutzer verfolgen mit sog. "serverfremden" Cookies.

Beispiel:
```
Set-Cookie: letzteSuche="cookie aufbau"; expires=Tue, 29-Mar-
2005 19:30:42 GMT; Max-Age=2592000; Path=/cgi/suche.py;
Version="1";
```

Erwähnenswert ist das das Cookie-Konzept eher eine Möglichkeit ist zum Identifizieren der tatsächlichen Zustandsinformationen auf dem Server wo das Cookie erstellt wurde (dem sog. Origin Server), als ein Verfahren zum Übertragen dieser Informationen.

Nach RFC 2065 soll ein Browser folgendes unterstützen:
- es sollen insgesamt 300 Cookies gespeichert werden können
- ein Cookie soll mindestens 4096 Bytes enthalten können
- es sollen mindestens 20 Cookies pro Domain gespeichert werden können

5. Benutzung von HTTP

Trotz seiner Einfachheit lässt sich die tatsächliche Verwendung des HTTP-Protokolls in manchen Fällen nicht so leicht herausfinden.
Anhand einer in England durchgeführte Studie die sich mit der Zählung der Header und er Häufigkeit der Header-Felder der HTTP-Reponses befasste ließen sich Hinweise über die tatsächliche Verwendung des HTTP-Protokolls schließen.
Es ließ sich zeigen das HTTP-Nachrichten mit 5, 6 und 8 Headern die größte Häufigkeit hatten, d.h. das als Normalfall eine Anzahl zwischen 5 und 8 Header angenommen werden kann.
Die Anzahl 0 bis 2 Header-Felder hatte eine Häufigkeit zwischen 0% und 0,18%.
Die meisten Responses waren zudem vertreten in den Header-Feldern `<Content-Type>`, `<Server>`, `<Date>`, `<Last-Modified>` und `<Content-Length>`, wobei die ersten 3 Felder fast immer vertreten waren.

6. Nicht zum Standard gehörende HTTP-Extensions

Trotz der Erhöhung der Anzahl der Header-Felder von HTTP/1.0 bis HTTP/1.1 gibt es noch einige nicht zum Standard gehörende Header-Felder. Diese werden aber trotzdem häufig verwendet und unterstützt.

6.1 Neuladen von Webseiten

Das Header-Feld Refresh dient dazu Web-Seiten nach einer bestimmten Zeit automatisch zu aktualisieren. Es stellt formal gesehen einen HTTP-Header innerhalb einer HTML-Seite dar.
Beispiel:

```
<META HTTP-EQUIV="Refresh" CONTENT="300">
```

Ein erster Zweck liegt in der Möglichkeit, eine Seite in regelmäßigen Abständen neu zu laden, beispielsweise um regelmäßig vorgenommene Änderungen auf Web-Seite eines Servers neu zu laden.
Wenn es nicht erforderlich ist, dasselbe Dokument neu zu laden, wird auch dieser Befehl verwendet, meistens wenn ein Dokument auf eine neue URI verschoben wurde.

6.2 Übergänge zwischen Seiten

Microsoft hat einen nicht mit freier Software implementierbaren Mechanismus vorgestellt um die Übergänge zwischen Seiten anzugeben. Diese Übergänge werden mit Hilfe von vordefinierten Effekten visuell gestaltet. Die 4 Übergängen `Page-Ente`, `Page-Exit`, `Site-Enter` und `Site-Exit`, werden mit den Attributen `HTTP-EQUIV` und `CONTENT` des `<META>`-Elements festgelegt.

7. Die Zukunft von HTTP

7.1 Verbesserungen

Es lässt sich momentan eine Reihe von Gebieten nennen auf denen Verbesserungen von HTTP/1.1 möglich wären:
- Verbesserung der Caching-Algorithmen
- Hit Count Reporting: die Origin Server haben Heutzutage keine Möglichkeit verlässliche Daten über die Verwendung ihre Seiten zu erheben. Ein häufig verwendetes Verfahren ist das Cach-Busting, bei dem mit cach-spezifischen Header-Felder alle Versuche zur Zwischenspeicherung abgewehrt werden. Dies verbraucht sehr viele Ressourcen und ist nicht zuverlässig, weil es von der Mitarbeit der Caches abhängig ist.
- Es gibt vorhandene Bereiche, auf denen sich die Komprimierung auswirken würde:
    - Datenkomprimierung: die Wirksamkeit er Komprimierungs-Algorithmen wird wesentlich von der Art der zu komprimierenden Daten beeinflusst.

- Verwendung von Delta Encoding zum Aktualisieren des Cache: anstatt ein neues Dokument vom Origin Server um Cache zu senden, genügt die Übersendung des Teils, in dem sich die neue und im Cache abgelegte Version unterscheiden.

Des Weiteren haben sich 3 zukünftige HTTP-Initiative herausgebildet, welche im Folgenden kurz erläutert werden.

7.2 Web-based Distributed Authoring and Versioning (WebDAV)

WebDAV ist ein offener Standard zur Bereitstellung von Dateien im Internet. Benutzer können auf Dateien wie auf eine Online-Festplatte zugreifen.
Es arbeiten 3 Arbeitsgruppen der IETF (Internet Engineering Task Force) an diesen Standard. Das ETF ist eine große internationale Gemeinschaft welche sich mit der Entwicklung von Protokolle befasst. Diese Gruppen sind die WebDAV Working Group, die DASL Working Group und die Delta-V Working Group. Diese wollen auf der Basis von HTTP Netzwerkstandards schaffen, mit denen Dateien und Dokumente im Netzwerk verändert und geschrieben werden können.

Technisch gesehen ist das WebDAV-Protokoll eine Erweiterung des Protokolls HTTP/1.1. Es besteht aus einem Satz neuer Methoden und Header für das HTTP, und ist fast mit Sicherheit das erste Protokoll das XML benutzt.

7.3 Protocol Extension Protocol (PEP)

Das Protocol Extesion Protocol ist ein Protokoll, das speziell zur Erweiterung des HTTP-Protokolls entwickelt wurde um den Zwiespalt zwischen einem unorganisierten Erweiterungswildwuchs und gezielter Spezifikation zu überbrücken.
Dafür nutzt das PEP den Header des HTTP-Protokolls um dort spezielle Platzhalter für neue Anweisungen und Statuscodes einzufügen.
Wenn HTTP-Server diese Informationen nicht verarbeiten können, werden diese entweder ignoriert oder mit einen speziellen Fehlerkode abgewiesen.
Falls eine Erweiterung überall ingesetzt wird, kann sie in eine neue Version des Protokolls eingearbeitet werden und sich so von eine dynamische in eine statische Erweiterung wandeln.

7.4 HTTP Next Generation (HTTP-ng)

Das Hypertext Tranfer Protocol - Next Generation ist ein Protokollvorschlag von Simon Spero. Erste Implementierungen werden zur Zeit entwickelt. Neben einer Beschleunigung der Nachrichtenübertragung soll dieses Protokoll auch bessere Voraussetzungen für den elektronischen Handel bieten.

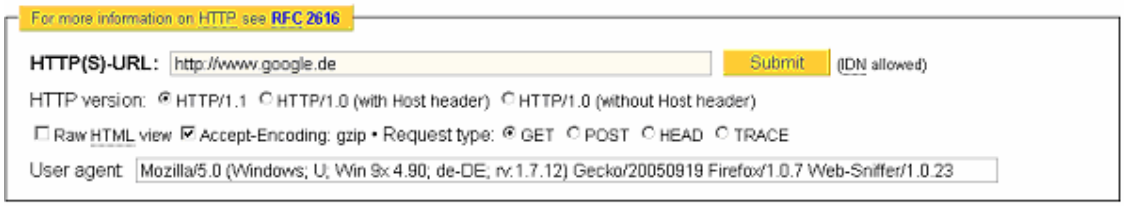
HTTP-ng ist weniger kompakt und besteht aus 3 Schichten: die obere Schicht steuert das grafische Benutzerinterface, die mittlere verteilt die Informationen auf Pakete und die untere versendet diese mittels TCP/IP über das Netz. Dabei wird das Protokoll SMUX verwendet, das eine permanente Verbindung zwischen Client und Server aufbaut.

Sicherheitsdienste lassen sich wesentlich leichter in das Protokoll integrieren. HTTP-ng verwendet ein modifiziertes Transaktionsmodell, welches die Abwicklung verschiedener Anfragen eines Clients über eine einzige Verbindung ermöglicht. Dazu ist diese Verbindung in mehrere vituelle Kanäle aufgeteilt, von dem ein Kanal die Kontrollinformationen überträgt, die anderen Kanäle die angeforderten Objekte. Die Kommunikation verläuft Asynchron, der Client muss nicht auf eine Antwort des Servers warten, bevor er eine neue Anfrage zum Server schickt.

Das Protokoll unterstützt auch die Übertragung von verschlüsselten Objekten, gegenseitige Authentifizierung aller beteiligten Parteien und Zahlungsmöglichkeiten. Von der Vielfalt der Zahlungssysteme kann gebrauch gemacht werden, da bereits auf Protokoll-Ebene des Schichtenmodells über die Voraussetzung verhandelt wird.

8. Anleitung zum Testen

Beim Surfen im Internet kommt man glücklicherweise nur indirekt mit dem HTTP-Protokoll in Berührung, da man nur die Internetadresse eingeben oder einen Link anklicken muss. Jedoch gibt es einige Seiten im Web, auf denen man sich die Datenkommunikation anschauen kann.



Eine sehr gute Seite ist **http://www.web-sniffer.net**

Die Bedienung ist sehr einfach. In dem Textfeld HTTP(S)-URL trägt man die gewünschte Webadresse ein. Zusätzlich kann man aus den zwei aktuellen HTTP-Versionen wählen. Man kann auch ausprobieren was passiert, wenn man den Host-Header nicht angibt.
Mit „Raw HTML view“ kann man sich das gesendete Entity in reinem HTML-Code ansehen, oder den Code farblich unterlegen lassen. Weiterhin kann man noch den Komprimierungsalgorithmus „gzip“ anwenden.
Auf dieser Seite kann man nur zwischen vier Request Methoden wählen (GET, POST, HEAD, TRACE), was zum testen aber völlig ausreicht.
Jetzt muss man nur noch auf den Button „Submit“ klicken und die Anfrage wird an den Server geschickt. Danach wird eine neue Seite geöffnet in dem der HTTP-Request angezeigt wird und auch die Antwort des Servers inklusive des Entities.

HTTP Request Header

```
Connect to 66.249.93.99 on port 80 ... ok

GET / HTTP/1.1[CRLF]
Host: www.google.de[CRLF]
Connection: close[CRLF]
Accept-Encoding: gzip[CRLF]
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5[CRLF]
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3[CRLF]
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7[CRLF]
User-Agent: Mozilla/5.0 (Windows; U; Win 9x 4.90; de-DE; rv:1.7.12) Gecko/20050919 Firefox/1.0.7 Web-Sniffer/1.0.23[CRLF]
Referer: http://web-sniffer.net/[CRLF]
[CRLF]
```

HTTP Response Header

Name	Value	Delim
HTTP Status Code: HTTP/1.1 200 OK		
Cache-Control:	private	CRLF
Content-Type:	text/html	CRLF
Set-Cookie:	PREF=ID=b0:9007ed0001[illegible] FF=1:1064:5:000 :id=1106:5:000:St:[illegible] qL4j:pW:[illegible]OxOl, ex: r-xs=fun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.de	CRLF
Content-Encoding:	gzip	CRLF
Server:	GWS/2.1	CRLF
Content-Length:	1490	CRLF
Date:	Thu, 05 Jan 2006 13:50:00 GMT	CRLF

Content (encoded: 1.46 KiB / decoded: 3.34 KiB)

```
<html><head><meta http-equiv="content-type" content="text/html; charset=UTF-8"><title>Google</title><style><!--
body,td,a,p,.h{font-family:arial,sans-serif}
.h{font-size:20px}
.g{color:#77cc00}
// -->
</style>
<script>
<!--
function [illegible]
function setcl(e,et,as,pq){e.href="/url?sa="+as+"&ei="+pq+"&url="+escape(et.href).replace(/\
// -->
</script>
</head><body bgcolor=#[illegible] text=#[illegible] link=#[illegible] vlink=#[illegible] onload=sf(); topmargin=3 marginhe
<form action="/search" name="f"><table
function sf(){if (window.RegExp && window.encodeURIComponent){var ue=el.href+"&ie="+encodeURIComponent(document
// -->
</script><table border=0 cellspacing=0 cellpadding=4><tr><td nowrap><font size=-1><b>Web</b>    
```

Eine Seite mit noch mehr Einstellmöglichkeiten und vor allem mit sehr guten Erklärungen zu diesen Einstellungen ist **http://www.bolege.de/whoiam/**
Hier lohnt es sich auf jedem Fall einmal vorbeizuschauen!

9. Anhang

9.1 HTTP Status-Codes

Codebereich 1xx: Allgemeine Informationen

- **100 Continue:** Der Client hat einen Teil seiner Anforderung erfogreich gesendet und soll damit fortfahren.
  Header: keine
- **101 Switching protocols:** Der Client wünscht ein anderes und im upgrade-Header angegebenes Protokoll. Der Server ist einverstanden.
  Header: upgrade: Benennt das neue Protokoll.

Codebereich 2xx: Anfrage des Clients verstanden und erfüllt

- **200 OK:** Der Client hat eine erfolgreiche Anforderung gestellt. Der Server sendet die angeforderten Informationen mit header und body.
  Header: eine ganze Menge
- **201 Created:** Der Client hat mit seiner Anforderung den Server veranlaßt, ein neues Dokument (genauer: einen URL) zu erzeugen.
  Header: Location: Enthält die Adresse des neuen Dokuments (den URL)
- **202 Accepted:** Der Client hat eine Anforderung gesendet, die der Server akzeptiert, aber nicht verarbeitet.
  Header: Weitere Informationen sind ggf. im Body der Server-Antwort.
- **203 Non-authoritative information:** Der Client erhält vom Server Daten aus einer externen Quelle, die nicht - zumindest unmittelbaren - Einflussbereich des Servers liegen.
  Header: keine
- **204 No content:** Der Client hat eine erfolgreiche Anforderung gestellt. Der Server sendet aber nur header und keinen body zurück. Dies ist für CGI-Programme nützlich, die Informationen etwa aus einem Web-Formular auslesen, die bereits dargestellte Seite aber nicht verändern möchten. Web-Browser sollten die Darstellung nicht aktualisieren.
  Header: eine ganze Menge
- **206 Partial content:** Der Client hat in seiner Anfrage im *range-Header* angegeben, dass er nur einen Teil des geforderten Dokuments haben möchte. Daten können so in Teilen verarbeitet werden.
  Header: Content-Range: Enthält die Angabe des Bereichs der Daten als Byte-Positionen.

Codebereich 3xx: Anfrage des Clients verstanden, jedoch nicht erfüllbar

- **300 Multiple Choices:** Der Client hat eine Anforderung gestellt, für die mehrere Ressourcen in Frage kommen, z.B. Dokumente in mehreren Sprachen. Im Body kann der Server ggf. diese Quellen aufschlüsseln und dem Client resp. dem Benutzer damit eine Auswahl ermöglichen.
  Header: keine spezifischen
- **301 Moved permanently:** Der Client hat eine URL angefordert, die der Server nicht länger benutzt. Die neue Adresse der angeforderte Ressource ist aber bekannt und wird dem Client mitgeteilt, der auch in Zukunft nur noch diese nutzen soll.
  Header: Location: Enthält die neue Adresse
- **302 Found:** Veraltet. Siehe 307
- **303 See other:** Der Client hat eine Ressource angefordert, die an anderer Stelle zu finden ist.
  Header: Location: Enthält die richtige Adresse der Ressource
- **304 Not modified:** Der Client hat nachgefragt, ob eine ihm bereits bekannte Ressource zwischenzeitlich verändert ist. Der Server verneint dies, der Client kann seine lokale Kopie nutzen.
  Header: Client: If-Modified-Since oder If-None-Match
- **305 Use proxy:** Der Client soll die angeforderte Ressource nicht vom Server, sondern über den angegebenen Proxy beziehen
  Header: Location: Proxy

- **307 Moved temporarily:** Der Client hat einen URL angefordert, die der Server zur Zeit verschoben hat. Die neue Adresse der angeforderte Ressource wird dem Client mitgeteilt; sie ist jedoch nur temporär gültig, und der Client soll in Zukunft die ursprüngliche Adresse nutzen.
  Header: Location: Die temporäre Adresse.

Codebereich 4xx: Anfrage des Clients unvollständig oder fehlerhaft

- **400 Bad request:** Der Client hat gestellt eine Anforderung syntaktisch falsche.
  Header: Keine.
- **401 Unauthorized:** Der Client hat eine Anforderung auf eine geschützte Ressource gestellt. Dies ist keine Fehlermeldung im engeren Sinne. Der Server sendet mit diesem Status-Code Informationen über die Art der notwendigen Authentifizierung. Web-Browser verlangen daraufhin in der Regel vom Benutzer die Eingabe eines Benutzernamens und eines Passworts und stellen mit diesen Angaben die Anforderung erneut. Schlägt dies dann fehl, antwortet der Server mit Code 403.
  Auf Apache-Webservern wird diese Funktionen häufig mit htaccess-Dateien o.ä. realisiert.
  Header: WWW-Authenticate: Enthält Informationen über die Art der notwendigen Authentifizierung.
- **402 Payment required:** Der Client hat eine kostenpflichtige Ressource angefordert. Dieser Statuscode ist (noch) nicht HTTP-Bestandteil.
- **403 Forbidden:** Der Client hat eine Anforderung gestellt, die der Server ohne Angaben von Gründen verweigert.
- **404 Not found:** Der Client hat ein Dokument angefordert, das - zumindest an angegebener Stelle - nicht existiert.
- **405 Method not allowed:** Der Client hat eine Methode benutzt, die der Server an dieser Adresse nicht erlaubt (z.B. GET statt POST)
- **406 Not acceptable:** Der Client hat ein Dokument angefordert, dessen Format er nach eigenen Angaben im Accept-Header nicht unterstützt.
  Header: Client: Accept
- **407 Proxy authentification required:** Der Client hat ein Anfrage gestellt, die vom Proxy autorisiert werden muss, bevor er ihn weiterleitet kann.
  Header: Proxy-Authentificate
- **408 Request time-out:** Der Client hat in der vom Server vorgegebenen Zeit keine vollständige Anfrage stellen können. Der Server beendet die Verbindung.
- **409 Conflict:** Der Client hat eine Anfrage gestellt, die der Server wegen eines Konflikts (z.B. mit einem anderen Request) nicht wie gewünscht verarbeiten kann bzw. wird. Nähere Angaben sind ggf. im Body enthalten.
- **410 Gone:** Der Client hat einen URL angefordert, die auf dem Server nicht mehr existiert. Im Gegensatz zu: 301, 307 und 404,
- **411 Length required:** Der Client hat an den Server Daten übermittelt, den dieser nicht ohne die im Header Content-length angegebene Größe akzeptiert.
  Header: Content-length
- **412 Precondition failed:** Der Client hat eine oder mehrere If...-Header übermittelt und die darin formulierten Bedingungen konnten nicht erfüllt werden
  Header: If...

- **413 Request Entity Too Large:** Der Client hat eine Anforderung gestellt, die dem Server zu groß ist (z.B. beim Upload von Dateien). Der Server verweigert die Verarbeitung.
- **414 Request URL too long:** Der Client hat einen URL übermittelt, der dem Server zu groß ist (z.B. bei der Übergabe von Formularwerten mittels GET). Der Server verweigert die Verarbeitung.
- **415 Unsupported media type:** Der Client hat Daten (im Body) übermittelt, die der Server nicht unterstützt.
- **416 Request range not satisfiable:** Der Client hat in seiner Anfrage im *range-Header* einen Teilbereich des Dokuments angegeben, der nicht existiert. Siehe 206.
- **417 Expectation failed:** Der Client hat im Excpect-Header Wünsche geäußert, die der Server nicht erfüllen kann oder will.
- **424 Failed Dependency:** Die Anfrage konnte nicht durchgeführt werden, weil sie das Gelingen einer vorherigen Anfrage vorausgesetzt hätte.Codebereich 5xx: Fehler im Server

- **500 Internal Server Error:** Der Server hat bei sich einen Fehler entdeckt, z.B. durch eine fehlerhafte Konfiguration oder durch ein abgestürztes CGI-Programm. In den Log-Files des Servers sollten sich genauere Angaben finden.
- **501 Not implemented:** Der Server kann die Anforderung des Clients nicht ausführen, weil die nicht unterstützt wird.
- **502 Bad gateway:** Der Server oder eher der Proxy hat formal ungültige Antworten von einem anderen Server oder Proxy bekommen.
- **503 Service unavailible:** Der Server kann die Anforderung des Clients momentan nicht ausführen. Ggf. gibt der Server in einem Retry-After-Header zurück, wann der Dienst wieder zur Verfügung steht.
- **504 Gateway time-out:** Der Client hat selbst eine Anfrage an einen Gateway oder Proxy gestellt, die innerhalb der auf dem Server definierten Zeit nicht beantwortet wurde. Siehe auch: 408
- **505 HTTP version not supported:** Der Client unterstützt die in der Anforderung des Clients angegebene HTTP-Version nicht.

9.2 Request for Comments (RFC)

- RFC 768 (UDP: User Datagram Protocol)
- RFC 791 (IP: Internet Protokol)
- RFC 792 (ICMP: Internet Control Message Protocol)
- RFC 793 (TCP: Transmission Control Protocol)
- RFC 822 (Standard for the format of the ARPA Internet text messages)
- RFC 826 (ARP: Adress Resolution Protocol)
- RFC 862 (ECHO: Echo Protocol)
- RFC 864 (CHARGEN: Character Generator Protocol)
- RFC 867 (Daytime Protocol)
- RFC 868 (Time Protocol)
- RFC 951 (BOOTP: Internet Bootstrap Protokol; des Weiteren in 1532 und 1533)
- RFC 959 (FTP: File Transfer Protocol)
- RFC 1034 (DNS: Domain Name System; concepts and facilities)
- RFC 1035 (DNS: Domain Name System; implementation and specification)
- RFC 1094 (Network File System, Version 2)
- RFC 1123 (REquirements for Internet Hosts - Application and Support)
- RFC 1661 (Point-to-point protocol)
- RFC 1738 (URL: Uniform Resource Locator)
- RFC 1831 (Remote Procedure Call)
- RFC 1833 (Portmapper, auch RPCBind)
- RFC 1939 (POP-3: Post Office Protocol)
- RFC 1945 (HTTP/1.0)
- RFC 2065 (Domain Name System Security Extensions)
- RFC 2131 (DHCP, Dynamic Host Configuration Protocol)
- RFC 2440 (OpenPGP: Pretty Good Privacy)
- RFC 2616 (HTTP : Hypertext Transfer Protocol)
- RFC 2821 (SMTP: Simple Mail Transfer Protocol)
- RFC 2822 (E-Mail-Format)
- RFC 2965 (HTTP State Management Mechanism)
- RFC 3174 (SHA: Secure Hash Algorithmus)

10. Quellenangaben

www.wikipedia.de
www.rfc-editor.org
www.bolege.de
www.w3.org
www.informatik.uni-frankfurt.de
Wilde, Erik, World Wide Web, Technische Grundlagen
Lars Eilebrecht Vortragsfolien Apache-Content Negotiation, www.apache.org