Web-basierte Systeme

04: Hypertext Transfer Protocol

Wintersemester 2022

Rüdiger Kapitza

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

HTTP das Rückgrat des WWW

Vorlesungsplan

2. November	Einführung und Darstellung von Webseiten (HTML und CSS)
9. November	Hypertext Transfer Protocol
16. November	Browser Schnittstellen
23. November	Kommunikationsschnittstellen im Browser
30. November	WebAssembly
7. Dezember	**Architektur moderner Browser**
14. Dezember	Clientseitige Architekturmuster
21. Dezember	Vorbereitung Papieranalyse
11. Januar	Papieranalyse
18. Januar	Serverseitige Implementierung von Web-basierten Systemen
25. Januar	Caching bzw. Lastverteilung durch Zwischenspeicher
1. Februar	Thema noch offen – wahrscheinlich WebRTC
8. Februar	Zusammenfassung und Ausblick

Zielsetzung der Lerneinheit

- Verständnis wie mittels HTTP Daten ausgetauscht werden
- Verschlüsselte Übertragung unter Verwendung von HTTPS
- Verbesserungen von HTTP 0.9 bis HTTP/2

Hypertext Transfer Protocol

Ursprüngliche **Entwicklungsziele von HTTP** (HTTP 0.9)

- Einfache Anfrage/Antwort-basierte Kommunikation
- Eindeutige Identifikation von Ressourcen
  - Jede HTTP-Anfrage beinhaltet einen Uniform Resource Identifier (URI)
- Zustandsloses Protokoll
  - Informationen aus früheren Anforderungen gehen verloren
- Metadaten zur Verhandlung der Übertragung
  - Mittels Header-Informationen und Statuscodes können beliebige Daten übertragen werden

Weiterentwicklung

- Standardisierung erfolgt durch Arbeitsgruppen des W3C und die Internet Engineering Task Force (IETF)
- Meist verbreitete Version ist aktuell HTTP 1.1 - die Neueste: HTTP/2

Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP)

- Ursprüngliche Version 0.9 – bis Juni 2014 Version 1.1
- Textbasiertes Protokoll
  - Einfache Analysierbarkeit (ohne zusätzliche Werkzeuge)

Anfrage-Antwort-Interaktion

- Client sendet Anfragenachricht (`Request`)
- Server sendet Antwortnachricht (`Response`)

Hypertext Transfer Protocol

Hypertext-Übertragungsprotokoll – technische Einordnung

- HTTP bildet ein Protokoll der Anwendungsschicht
- HTTP benötigt ein zuverlässiges Transportprotokoll
  - D.h. (meist) das Transmission Control Protocol (TCP)

Uniform Resource Identifier

Uniform Resource Identifier (URI)

- eindeutige Referenzierbarkeit von Ressourcen
  - Dokumente, Mailadressen, Mailboxen, Webseiten, ...
- einheitliches Schema zur Referenzierung
  - erweiterbares, lesbares und maschinenlesbares Format

Heutige Sichtweise (Contemporary View)

- alles ist eine URI und verschiedene Schemata

Häufige, generische Aufteilung (z.B. bei http: und ftp:)

```
<schema> "://" [ <username> [":" <passwort>] "@" ] <hostname>
                 [":" <portnummer>] ["/" <url-pfad>
```

Literatur

- T. Berners-Lee, R. Fielding, L. Masinter: Uniform Resource Identifier (URI): Generic Syntax. IETF, RFC 3986, Jan. 2005.

Nachrichtenaufbau

Anfrage: Request-Methode

- Anfragetyp, Methodenauswahl
  - z.B. `GET /robots.txt HTTP/1.0`
- Request-Header
  - Zusatzinformationen zur Anfrage
  - z.B. `User-Agent: OperMoz` und `Host: www.fau.de`
- Leerzeile
- Optionaler Datenbereich (Body)
  - Inhalt beschrieben durch bestimmten Request-Header (Content-Type)
  - MIME-Type (Multi-Purpose Internet Mail Extension) [Body]

Nachrichtenaufbau

Beispiel für eine GET-Anfrage

```
GET / HTTP/1.1
Host: localhost:6789
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/604.5.6
        (KHTML, like Gecko) Version/11.0.3 Safari/604.5.6
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Methoden

Definierte Methoden

- `GET`: Anfrage nach Daten – identifiziert durch einen URI
  - Daten können vorliegen oder dynamisch erzeugt werden
- `POST`: Schicke Daten
  - Zum Beispiel: Inhalt eines Formulars oder Textfeldes
  - Information wird im Datenbereich der Nachricht abgelegt
  - Ziel der Daten wird durch die URI festgelegt

Typisch für Webserver: `GET` und `POST`

Methoden

Definierte Methoden (Fortsetzung)

- `HEAD:` Hole Response-Header ohne Datenanhang (sonst wie `GET`)
  - Verwendet zu Fehlersuche und Validitätsprüfung von URIs
- `PUT:` Speichere Daten
  - Führt zur Erzeugung eine Ressource falls diese noch nicht auf dem Server existiert
- `DELETE:` Löscht durch URI referenzierte Ressource
- `OPTIONS:` Hole Serverinformationen
  - z.B. ob PUT für eine Datei unterstützt wird

Nachrichtenaufbau – Antwort

Antwort: HTTP-Response

- Status der Antwort
  - z.B. `HTTP/1.0 200 OK`
- Response-Header
  - Zusatzinformationen zur Antwort
  - z.B. `Server: apache/2.0`
- Leerzeile
- optionaler Datenbereich (Body)

Daten

- Beliebig (orientiert an MIME-Types)
  - z.B. `Content Type: text/html`
- Mehrere Einzelteile durch Multipart-Typen
  - z.B. `multipart/mixed`

Nachrichtenaufbau

Beispiel für eine HTTP Antwort (gekürzt!):

```
HTTP/1.1 200 OK
Date: Mon, 09 Apr 2018 07:51:40 GMT
Server: Apache/2.4.18 (Ubuntu)
Strict-Transport-Security: max-age=31536000; includeSubDomains
Public-Key-Pins: pin-sha256="SdNy5BOgYHobIdEHk3CdsPUwgIiiMctIM5eVeDPN4qg="; ...
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Last-Modified: Wed, 22 Jun 2016 11:46:03 GMT
ETag: "1d6-535dc7b6b7f7e"
Accept-Ranges: bytes
Content-Length: 470
Vary: Accept-Encoding
Content-Type: text/plain

User-agent: *
Disallow: /projects/tclouds/download
Disallow: /projects/caprice
Disallow: /login
...
```

- Erzeugt via: `curl –verbose https://www.ibr.cs.tu-bs.de/robots.txt`

Nachrichtenaufbau – Antwort

HTTP Statuscodes

- **1xx** – Informational responses
  - Beispiel: `102 Processing` – Verhindert ein Timeout auf Browserseite
- **2xx** – Success
  - Beispiel: `200 OK`
- **3xx** – Redirection – „Location“-Header-Feld sagt wo die Daten zu finden sind
  - Beispiel: `301 Moved Permanently`
- **4xx** – Client errors
  - Beispiel: `400 Bad Request` – Anfrage-Nachricht war fehlerhaft aufgebaut
- **5xx** – Server errors
  - Beispiel: `500 Internal Server Error` – „Sammel-Statuscode“ für unerwartete Serverfehler.
  - Beispiel: `501 Not Implemented`

Abbildung von HTTP-Anfragen auf TCP/IP

In den Anfangstagen des WWW (HTTP 0.9-1.0):

- Eine **TCP/IP-Verbindung pro Anfrage/Antwort-Paar**
- Server sind völlig zustandslos

- Wiederkehrende Etablierung von neuen Verbindungen ist aufwändig
  - TCP Drei-Wege-Handschlag und slow-start
  - Bereitstellung von Systemressourcen

Literatur

- Hypertext Transfer Protocol – HTTP/1.0. IETF, RFC 1945, May 1996.

Aufbau aktueller Webseiten

- Top 1000 Websites zwischen 1. Juli 2017 – 1. März 2018 (`http://httparchive.org/`)

⇒ Heutige Webseiten setzen sich aus vielen Einzelteilen zusammen!

Abbildung von HTTP auf TCP/IP

Verbesserungen durch HTTP 1.0 und 1.1

- Wiederverwendung der Verbindung
  - Mehrere Anfragen über die gleiche Verbindung
  - Standard für HTTP 1.1 und konfigurierbar für HTTP 1.0 (`Connection: Keep-Alive`)
- Zusätzlich ist *Pipelining* möglich (HTTP 1.1)
  - Mehrere Anfragen hintereinander (über eine Verbindung)
  - Antworten kommen in gleicher Reihenfolge später
- **Problem:** Head-of-Line Blocking
  - Sollte eine Anfrage ins Stocken geraten werden alle weiteren gebremst!
  - Oft mehrere parallel TCP-Verbindungen (2-8) — hier gibt es aber auch Beschränkungen

Literatur

- Hypertext Transfer Protocol – HTTP/1.1. IETF RFC 2616, June 1999.

HTTP Secure

Sicherheitsmängel von HTTP

- Privatheit – Informationsaustausch erfolgt im Klartext
- Integrität – ausgetauschte Information kann manipuliert werden
- Authentifizierung – Zugangsdaten werden ungeschützt übertragen

HTTP Secure (HTTPS) als Lösung

- HTTPS realisiert HTTP unter Verwendung der Secure Socket Layer (SSL)
- Nachrichten werden vor Austausch verschlüsselt und am Zielort entschlüsselt

HTTP Secure

Technische Einbindung

- Eigener Standardport – 443 im Gegensatz zu Port 80
- URIs beginnen mit `https`
- Der Secure Socket Layer wird genutzt um Verschlüsselungsverfahren und Schlüssel auszuhandeln

HTTP Secure

1. Aktuelle Browser sind mit verschiedenen Root-Zertifikaten versehen. Diese beinhalten den öffentlichen Schlüssel der jeweiligen Zertifizierungsstelle.
2. Wird `https` angeboten so 'muss' in der Regel ein Zertifikat vorliegen welches durch eine Zertifizierungsstelle signiert ist.
3. Bei Verbindungsaufbau mit dem entfernten Webserver werden die unterstützten Verschlüsselungsverfahren mitgeteilt

HTTP Secure

4. Als Antwort auf die Vebindungsanfrage wird das durch die Zertifizierungsstelle signierte Zertifikat übertragen
5. Es wird überprüft ob das Zertifikat korrekt signiert ist und der Verbindungsaufbau abgeschlossen
6. Bei erfolgreichem Verbindungsaufbau kann nun der Datenaustausch erfolgen

HTTP Secure - Verbindungsaufbau

- Browser initiiert sichere Verbindung

  `https://www.ibr.cs.tu-bs.de`
- Bei Aufbau erfolgt ein SSL-Handschlag
- Als Ergebnis werden Verfahren und weitere Parameter etabliert
- (Sehr vereinfachte Darstellung!)

HTTP/2

Zielsetzung: Adressierung der Probleme von HTTP 1.0 und 1.1

- Reduktion der Empfindlichkeit gegenüber variablen RTTs
- Adressierung der Head-of-line Blocking Problematik
- Vermeiden von parallelen TCP-Verbindungen
- ...dennoch soll die Semantik von HTTP 1.1 erhalten bleiben!
  - Anwendungen müssen nicht umgeschrieben werden

Entstehungsgeschichte

- Entwicklung von HTTP/2 begann mit SPDY bei Google 2009
  - Zielsetzung: Halbierung der Ladezeit für typische Webseiten
- 2012 unterstützen druch Chrome, Firefox und Opera
- Als Konsequenz leitet die IETF HTTP Working Group einen Standardisierungsprozess ein
- Mai 2015 wurden der RFC 7540 (HTTP/2) und der RFC 7541 (HPACK) veröffentlicht

HTTP/2 kompakter Überblick

- **Eine TCP Verbindung** für die gesamte Kommunikation mit einem Webserver
- Anfrage $\rightarrow$ Stream
  - Multiplexen von Streams
  - Streams haben Prioritäten
- Binary-framing Schicht
  - Priorisierung
  - Flusskontrolle
  - Server push
- Header Kompression (HPACK)

HTTP/2 Binary Framing Layer

- Redefiniert wie HTTP Nachrichten ausgetauscht werden
- Framing Layer bleibt dabei für Anwendungs- und Socket-Schicht **transparent** – Browser und HTTP-Server müssen angepasst werden!
- HTTP Nachrichten werden in kleinere binärkodierte Nachrichten unterteilt
- Direkte Lesbarkeit geht verloren – Werkzeuge zur Protokollanalyse werden benötigt

Logischer Aufbau einer HTTP/2 Verbindung

Struktur einer HTTP/2 Verbindung

- Ein **Stream** bildet eine bidirektionale Verbindung über die Nachrichten (Messages) ausgetauscht werden
- **Message** eine Sequenz von Frames die zusammen eine (HTTP)-Anfrage- oder Antwortnachricht bilden
- **Frame** kleinste logische Einheit zur Datenübertragung
  - verfügt über einen Protokollkopf welche mindestens eine Zuordnung zu einem Stream enthält

HTTP/2 Multiplexing

- Bisher war es für HTTP/1.x nötig für Parallelität mehrere Verbindungen zu öffnen
- HTTP/2 ermöglicht es über eine Verbindung
  - mehrere Anfragen und Antworten parallel zu vermitteln
  - nur eine einzige TCP Verbindung zu nutzen
  - und macht bisherige HTTP/1.x Optimierung obsolet
    - z.B. *image sprites*

Priorisierung bei HTTP/2

- Zielsetzung ist es die übertragenen Daten so zu organisieren, dass sich eine Webseite möglichst schnell aufbaut
- Durch Priorisierung kann der Browser dem Webserver Hinweise geben welche Inhalte wann benötigt werden
- Entsprechende Hinweise können sich dynamisch ändern
  - z.B. wird das HTML einer Webseite primär benötigt, danach CCS und im folgenden Grafiken
  - Bei Grafiken kann es davon abhängen ob z.B. der Benutzer die Maus auf dem für die Grafik vorgesehene Ort platziert
- Webserver müssen diesen Hinweisen nicht folgen
  - Sind hoch priorisierte Daten aktuell nicht verfügbar – können andere Daten versendet werden

Priorisierung: Technische Umsetzung

Technische Umsetzung

- Jeder Stream erhält ein Gewicht zwischen 1 und 256
  - Je höher das Gewicht umso mehr Ressourcen werden angefordert
- Jeder Stream kann eine Abhängigkeit zu einem anderen Stream angeben
  - Abhängigkeiten haben Vorrang!
- Wird keine Abhängigkeit angegeben, so ist implizit eine Abhängigkeit zur Wurzel der Webseite gegeben

Priorisierung: Technische Umsetzung

Technische Umsetzung (Fortsetzung)

- Basierend auf der Gewichtung und den Abhängigkeiten entstehen Anforderungsbäume
  1. A und B teilen sich die Ressourcen 3/4 zu 1/4
     - Gewichte dieser Ebene durch das entsprechende Gewicht des Knotens
  2. Erst erhält D alle Ressourcen dann C

HTTP/2 Flusskontrolle

Wozu wird FlussKontrolle bei HTTP/2 benötigt?

- Beispiel: Browser hat ein Video mit hoher Bandbreite angefordert – der Benutzer pausiert aber den Stream
- Beispiel: Proxy Server hat mehr Systemressourcen als der Endbenutzer und will deshalb den Transfer zwischen sich und dem Webserver beschränken
- Greift hier nicht schon TCP?
  - ⇒ Nein, tut es nicht da TCP nur die gesamte Verbindung reguliert aber nicht individuelle Streams

HTTP/2 Flusskontrolle

Technische Umsetzung

- Jeder Empfänger kann pro Stream und Verbindung eine Fenstergröße festlegen
- Die Fenstergröße wird durch einen Empfänger beim Verbindung- und Streamaufbau vermittelt (SETTINGS Frame)
- Durch die Vermittlung von DATA Frames wird das Übertragungsfenster reduziert und durch WINDOW_UPDATE Frames wieder vergrößert
- Standardfenstergröße ist 65535 Bytes, wann genau das Übertragungsfenster verändert wird ist Aufgabe der Implementierung und nicht festgelegt.

HTTP/2 Server Push

Server Push

- Server kann auf eine Anfrage mehrere Antworten schicken
- Annahme entsprechende Informationen werde ohnehin benötigt!
  - Beispiel: Seite referenziert eine CSS und ein JavaScript Datei
- Zielsetzung ist es Informationen verfügbar zu machen bevor sie im Browser benötigt werden
  - In Bezug auf HTTP/1.x macht dies *Inlining* obsolet!
  - Vorzüge gegenüber Inlining – Daten können wie normale Ressourcen behandelt werden

HTTP/2 Server Push

Technische Umsetzung

- Proaktiv bereitgestellte Ressourcen via PUSH_PROMISE Frames
- Browseranfragen und PUSH_PROMISE Frames dürfen sich nicht überholen!
  - Lsg.: PUSH_PROMISE Frames und dann die Antwort versenden
- Browser kann PUSH_PROMISE Frames ablehnen (RST_STREAM) falls Daten schon lokal vorhanden
- Browser behält Kontrolle über die Nutzung
  - Bspw. kann er über SETTINGS Frame dies unterbinden

Wandel der Zeit: Removing HTTP/2 Server Push from Chrome

- `https://developer.chrome.com/blog/removing-push/`

HTTP/2 Header Komprimierung

- Header-Daten in HTTP/1.x werden immer als Text versendet und können zwischen 500-800 Byte groß sein oft auch größer

⇒ Speziell bei kleineren Ressourcen ist der Overhead für Metadaten recht hoch!

- HTTP/2 reduziert dies durch zwei Maßnahmen:
  1. Einzelne Felder eines Headers werden Huffman enkodiert
  2. Es wird ein gemeinsamer Kontext aufgebaut bei dem bereits versendete Inhalte nur noch durch einen Index übermittelt werden

HTTP/2 Header Komprimierung

- Als weitere Maßnahme wird eine statische und eine dynamische Tabelle eingeführt
  - Die statische Tabelle wird im Rahmen der Spezifikation definiert und enthält allgemeine Header-Felder
  - Die dynamische Tabelle wird pro Verbindung etabliert

Aufbau einer HTTP/2 Verbindung

- Die endgültige Transition zu HTTP/2 ist ein langlaufender Prozess
- Aktuell unterstützen alle etablierten Browser HTTP/2 aber nicht alle Webserver
- Es bestehen 3 Wege um eine HTTP/2 Verbindung zu etablieren:
  1. Aufbau zusammen mit TLS-Verbindung über ALPN (Standard)
  2. Upgrade einer normalen Verbindung (via HTTP upgrade)
  3. Pro-aktiver Aufbau einer HTTP/2 auf Grund von Vorwissen (z.B. DSN-Eintrag)
- Aktuelle Verfügbarkeit laut w3techs.com für die 10 Million beliebtesten Webseiten: 50,8%

Zusammenfassung

- Verständnis wie mittels HTTP Daten austauschen werden
- Verschlüsselte Übertragung unter Verwendung von HTTPS
- Verbesserungen von HTTP 0.9 bis HTTP/2
  - HTTP/2 eröffnet vielen neue Optionen und erübrigt eine Reihe von *Workarounds*
- HTTP/3 als nächster Schritt
  - Verwendung von QUIC welches UDP nutzt, zur Verbesserung der Widerstandsfähigkeit bei Paketverlusten

Referenzen

- High Performance Browser Networking `https://hpbn.co`