Verteilte Systeme

HTTP 2 (Quelle: hauptsächlich https://daniel.haxx.se/http2)

by

Dr. Günter Kolousek

Überblick

- Nachfolger von HTTP/1.1
  - HTTP/1.1 ist mittlerweile ein umfangreicher Standard
    - keine Implementierung implementiert alles!
    - viele Optionen und Erweiterungsmöglichkeiten → Interoperabilitätsprobleme!
  - baut auf SPDY (von Google) auf
- RFC 7540

HTTP/1.x und Performance

- HTTP/1.0 … 1 Request ausständig zu einer Zeit je TCP Verbindung

HTTP/1.x und Performance

- HTTP/1.0 … 1 Request ausständig zu einer Zeit je TCP Verbindung
- HTTP/1.1 … Pipelining
  - aber kein verschränktes Senden und Empfangen
    - außer mehrere TCP Verbindungen (diese sind beschränkt: dzt. nicht mehr als 6-8 je Site durch Browser)
  - $\rightarrow$ HOL Blocking

HTTP/1.x und Performance

- HTTP/1.0 … 1 Request ausständig zu einer Zeit je TCP Verbindung
- HTTP/1.1 … Pipelining
  - aber kein verschränktes Senden und Empfangen
    - außer mehrere TCP Verbindungen (diese sind beschränkt: dzt. nicht mehr als 6-8 je Site durch Browser)
  - $\rightarrow$ HOL Blocking
- Allgemein
  - HTTP Header wiederholend und langatmig
    - $\rightarrow$ Netzwerkbelastung

HTTP/2 – Ziele

- kompatibel zu HTTP/1.1 sein
  - auf gewisser (hohen) Abstraktionsebene
- ”Performance” verbessern
  - d.h. Latenz reduzieren
- d.h. Anzahl der TCP Verbindungen reduzieren
  - nur eine Verbindung je Domain
- Sicherheit verbessern

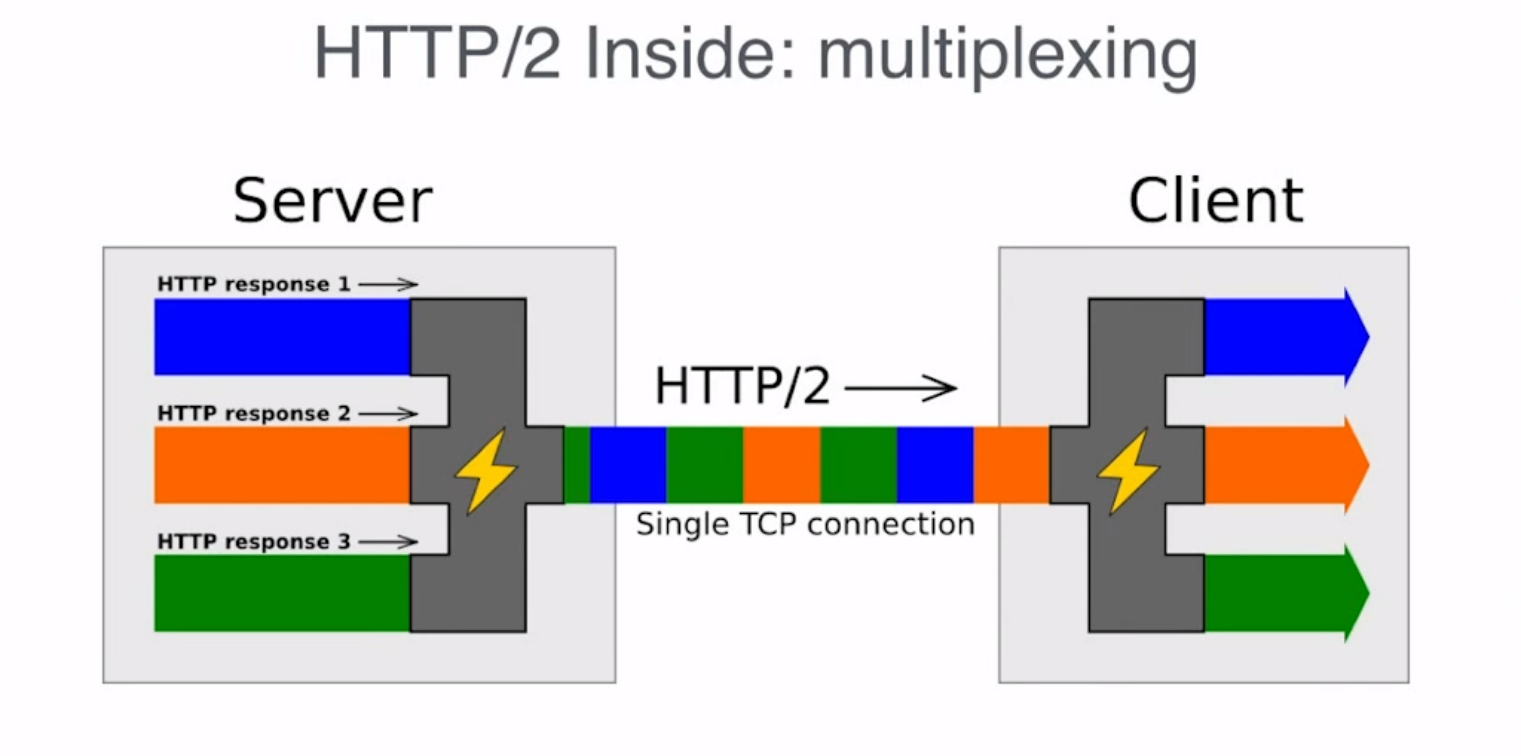
HTTP/2 – Features

- Multiplexen mehrerer Requests über eine TCP Verbindung
  - verschränktes Senden und Empfangen
- binäres Nachrichtenformat (!) & Komprimierung der Header
- Server Push
  - mehrere Antworten für einen Request
- Priorisierung der Requests
- Definition eines Profils für TLS
  - wenn HTTP/2 über TLS

HTTP/2 – Multiplexen

- mehrere Streams über *eine* TCP Verbindung
- Jeder Stream
  - besteht aus einer Folge von Frames
  - hat eine Stream-ID
  - hat eine Priorität (Änderung zur Laufzeit möglich)
  - kann vorzeitig beendet werden
  - hat eine Flusskontrolle (engl. flow control)
    - Schutz des Empfängers vor Überlastung
- Wirkungen
  - $\rightarrow$ Beheben des HOL
  - $\rightarrow$ eine Verbindung je Domain $\rightarrow$ ”Performance”

HTTP/2 – Multiplexen – 2



Quelle:
https://www.nginx.com/blog/http2-module-nginx/

Binär & Komprimierung der Header

- Binäres Format
  - $\rightarrow$ binäre Daten vs. Textdaten
  - weiters: Komprimierung der Header
- Wirkungen
  - effizienter zu parsen
  - geringere Datenmenge auf der Leitung
  - weniger Fehlerquellen
    - z.B. Behandlung von Whitespace, Groß/Kleinschreibung, Zeilenende, Leerzeilen,…
  - $\rightarrow$ allg. Verbesserung der ”Performance”

HTTP/2 – Server Push

- UA sendet Request für Ressource
- Server antwortet mit HTML *und* CSS, JS,…
- Wirkungen
  - $\rightarrow$ Reduzierung der Latenz

HTTP/2 – Priorisierung der Requests

- müssen Client und Server beherrschen
  - Client teilt Server Priorisierung mit
  - dzt. keine Möglichkeit für Frontend-Entwickler diese zu bestimmen
- z.B. HTML $\rightarrow$ CSS $\rightarrow$ JS $\rightarrow$ Bilder
- Wirkungen
  - $\rightarrow$ Darstellung einer Seite schneller

Funktionsweise

- (vorzugsweise) nur eine TCP Verbindung je Server
  - Empfehlung in RFC 7540: max. #Streams nicht unter 100 konfigurieren!
- Stream: Multiplexing einer TCP Verbindung
  - bidirektional
- Message: ein Stream überträgt Messages
  - Request (GET, POST,…), Response
- Frame: jede Message besteht aus einem oder mehreren Frames
  - $\rightarrow$ kleinste Kommunikationseinheit für binärkodierte Headerdaten und Nutzdaten

Funktionsweise – 2

- Frame
  - `Length`: 24 Bits
  - `Type`: 8 Bits
  - `Flags`: 8 Bits
  - R: reserviert, 1 Bit
  - `Stream Identifier`: 31 Bits ($\rightarrow$ Multiplexing)
  - `Frame Payload`

Funktionsweise – 3

- `Type`
  - `DATA`, `HEADERS`
    - Komprimierung der Header mittels neuem Algo *HPACK*
  - `CONTINUATION` … zum Senden von weiteren Headerblockfragmenten
  - `SETTINGS` … einer Verbindung
  - `PRIORITY` … Ändern der Priorität und Abhängigkeit zu anderen Stream (Elternstream) herstellen ($\rightarrow$ Baum)
    - Ressourcen nur an Kindstream, wenn Elternstream beendet oder kein Fortschritt beim Elternstream möglich
  - `GOAWAY` … beenden eines Streams
  - `RST_STREAM` … sofortiges Abbrechen eines Streams
    - so etwas geht in HTTP/1.x nicht!
  - `PUSH_PROMISE` … im Vorhinein mitteilen, dass Stream später angelegt wird
  - `PING` … messen der RTT
  - `WINDOW_UPDATE` … für Flusskontrolle

Webseiten optimieren

- kein Domain Sharding mehr!
  - Ursprüngliche Idee: Anzahl der gleichzeitigen Verbindungen durch Verwendung von Subdomainen ↑
  - z.B. Aufteilen der Bilder auf `img1.example.com` und `img2.example.com`
  - aber
    - jetzt werden je Subdomain eine neue TLS Verbindung aufgebaut!
    - TCP Slow Start → anfänglich geringere Bandbreite!
- kein Zusammenpacken von CSS und JS mehr!
  - Ursprüngliche Idee: Anzahl der zu ladenden Ressourcen reduzieren
  - aber
    - mehrere Dateien → Priorisierung möglich
    - kein einzelnes Caching möglich
      → Änderungen oder Zuteilung zu einzelnen Seiten

Webseiten optimieren – 2

- kein Inlining von CSS und JS in HTML mehr!
  - Ursprüngliche Idee: Seite schneller anzeigen können
  - aber
    - HTML Ressourcen deutlich größer
    - kein einzelnes Caching möglich
    - $\rightarrow$ HTTP/2 Server Push um Ressourcen vorweg zum Client zu schicken
- HTTP/2 Server Push
  - eigener Push Cache im Browser
  - Daten, die im Browser-Cache liegen bräuchten nicht gesendet werden
    - Browser hat Möglichkeit begonnen Push abzubrechen, aber…
    - Internet-Draft vorliegend (Cache Digests): Browser teilt Server mittels ”Cache Digests” mit, welche Ressourcen schon im Browser Cache

Kritik

- inkonsistent, unnötige Komplexität, verletzt das Schichten-Prinzip
- de facto Zwang zur Verschlüsselung (ursprünglich zwingend!) $\rightarrow$ Firefox, Chrome
  - oft nicht benötigt
  - Ressourcenbedarf $\rightarrow$ TLS (Handshake, Verschlüsselung)
  - Performance könnte sinken $\rightarrow$ kein Caching!
- verbessert nicht die Privatsphäre
  - z.B. Cookies bleiben bestehen
    - anstatt z.B. einer vom Client erzeugter Session-ID
  - Vermutung: Großfirmen (wie Google) $\rightarrow$ Geschäftsmodell
- verbessert Performance nur wenn CDN verfügbar
  - nicht bei einzelnem Server $\rightarrow$ erhöhter Aufwand!

hauptsächlich: `http://queue.acm.org/detail.cfm?id=2716278`

Zukunft

- Grundlegende Probleme mit HTTP/2
  - basiert auf TCP
  - ähnliches Problem wie HOL bleibt bestehen
    - Wenn TCP Segmente verloren gehen, dann werden die weiteren schon eingetroffenen Segmente erst bestätigt, wenn das verlorenen gegangene Segment nochmals gesendet und eingetroffen ist!
    - speziell bei unzuverlässigen Kommunikationskanälen ein Problem, wie z.B. bei mobilen Geräten
- deshalb: HTTP/3
  - HTTP über QUIC
  - wird von IETF standardisiert
  - wird schon verwendet von
    - Chrome (70% Marktanteil!)
    - der Facebook App

QUIC

- Quick UDP Internet Connections
- Transportprotokoll UDP!
- wird von IETF standardisiert (voraussichtlich 2021)
  - ursprünglich von Google entwickelt
- Vorteile
  - reduzierte Latenz bei Verbindungsaufbau
  - bessere Performance (auch bei Verlust von Datenpaketen)
  - Von Anfang an verschlüsselt
- Nachteile
  - Reifegrad nicht so hoch wie bei TCP
  - ”schwerer” für Router
    - sehen derzeit nur eine Folge von UDP Datagrams!
    - TCP hat im Gegensatz unverschlüsselte Header!

QUIC – Charakteristiken

- kein 3 Way Handshake beim Verbindungsaufbau
  - nur einfacher Handshake wie bei TLS, d.h.
    1. $\rightarrow$ `ClientHello`
    2. $\leftarrow$ `ServerHello`
    3. $\rightarrow$ `Finished`
- mehrere Streams über UDP (multiplexing)
  - Jeder Stream hat eigene Fehlerbehandlung
    - daher nicht das Problem wie bei TCP!
- IP Adressen können sich während Betrieb ändern
  - da UDP
  - z.B. Smartphone wechselt von mobilen Netzwerk ins WLAN
- TLS 1.3 integriert