HTTP/2

Mirko Dziadzka

https://mirko.dziadzka.de
@MirkoDziadzka

OWASP Stammtisch München
2015-11-17

Wer bin ich

- "Old School Unix Geek, currently interested in IT-Security, Software Development, Unix, Python, Go and Distributed Computing"
- Mache IT/Unix/Internet beruflich seit '93. In Architektur, Entwicklung, Betrieb. Bei Schweizer Banken und Deutschen Startups.
- Seit 10 Jahren: "Brocade/Riverbed/Zeus/art of defence" in Regensburg. Web Application Firewall
- **Disclaimer**: Ich gebe hier meine Meinung wieder, das ist nicht unbedingt die Meinung meiner Firma

Worum gehts es heute

- Seit Anfang diesen Jahres ist HTTP/2 als Nachfolger von HTTP/1.1 verabschiedet
- Alle großen Browser haben es implementiert
- Server und Middleware können es auch
- Was bringt HTTP/2 an Neuigkeiten, warum will man es haben (oder auch nicht)

Worum es heute geht

"HTTP/2.0 is not a technical masterpiece. It has layering violations, inconsistencies, needless complexity, bad compromises, misses a lot of ripe opportunities, etc."

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

But see also: https://news.ycombinator.com/item?id=8850059

Worum es heute geht

- Von HTTP/0.9 bis HTTP/1.1
- Probleme von HTTP/1.1
- Was will HTTP/2 besser machen
- Einblicke in HTTP/2
- Probleme
- Tools im täglichen Einsatz

HTTP/0.9

```
GET /
<html><body><h1>It works!</h1></body></html>
```

53 bytes, 7 byte Request

Funktioniert heute noch so (zum Beispiel Apache)

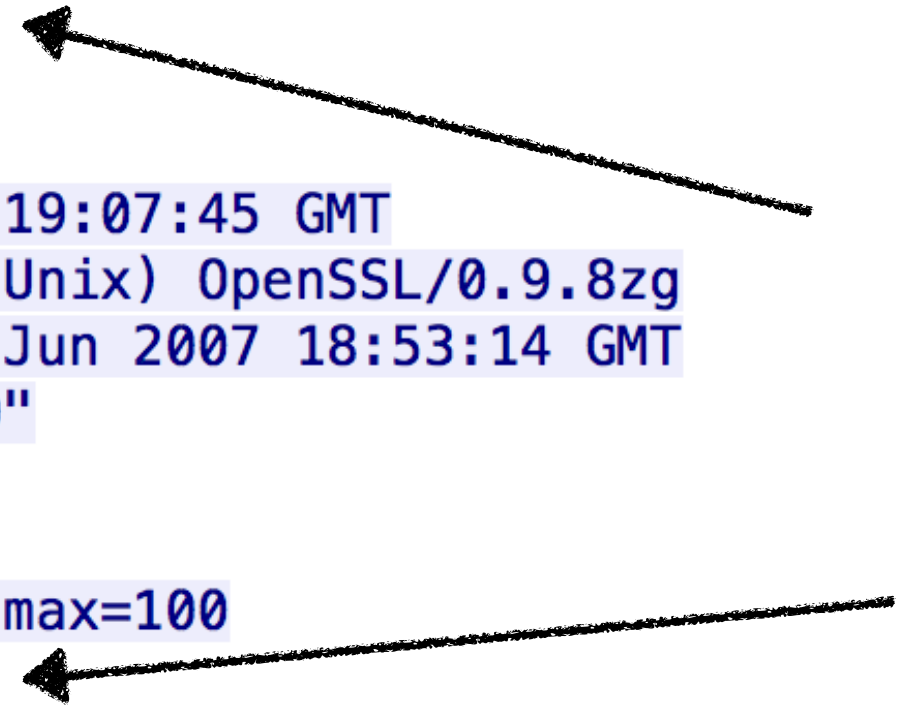
Wird gerne von Load-Balancern zum Health-Check eingesetzt (spart Bytes auf der Leitung)

HTTP/1.0

```
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 16 Nov 2015 19:03:51 GMT
Server: Apache/2.4.17 (Unix) OpenSSL/0.9.8zg
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"
Accept-Ranges: bytes
Content-Length: 45
Connection: close
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

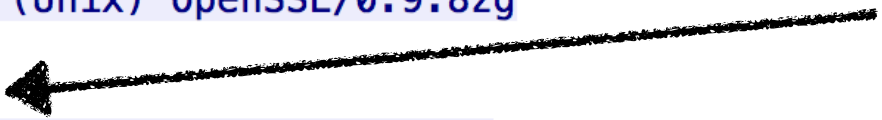
HTTP/1.0 w/ keep-alive

```
GET / HTTP/1.0
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Mon, 16 Nov 2015 19:07:45 GMT
Server: Apache/2.4.17 (Unix) OpenSSL/0.9.8zg
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"
Accept-Ranges: bytes
Content-Length: 45
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```



HTTP/1.1 - ups

```
GET / HTTP/1.1

HTTP/1.1 400 Bad Request
Date: Mon, 16 Nov 2015 19:34:56 GMT
Server: Apache/2.4.17 (Unix) OpenSSL/0.9.8zg
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
</body></html>
```



HTTP/1.1 - ok

```
GET / HTTP/1.1
Host: localhost

HTTP/1.1 200 OK
Date: Mon, 16 Nov 2015 19:36:31 GMT
Server: Apache/2.4.17 (Unix) OpenSSL/0.9.8zg
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"
Accept-Ranges: bytes
Content-Length: 45
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

HTTP/1.1 - vom Browser

```
GET / HTTP/1.1
Host: localhost:8080
Accept-Encoding: gzip, deflate
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_10_5) AppleWebKit/601.2.7 (KHTML, like Gecko)
Version/9.0.1 Safari/601.2.7
Accept-Language: en-us
DNT: 1
Connection: keep-alive

HTTP/1.1 200 OK
Date: Mon, 16 Nov 2015 12:30:09 GMT
Server: Apache/2.4.17 (Unix) OpenSSL/0.9.8zg
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"
Accept-Ranges: bytes
Content-Length: 45
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

HTTP - Wo ist das Problem

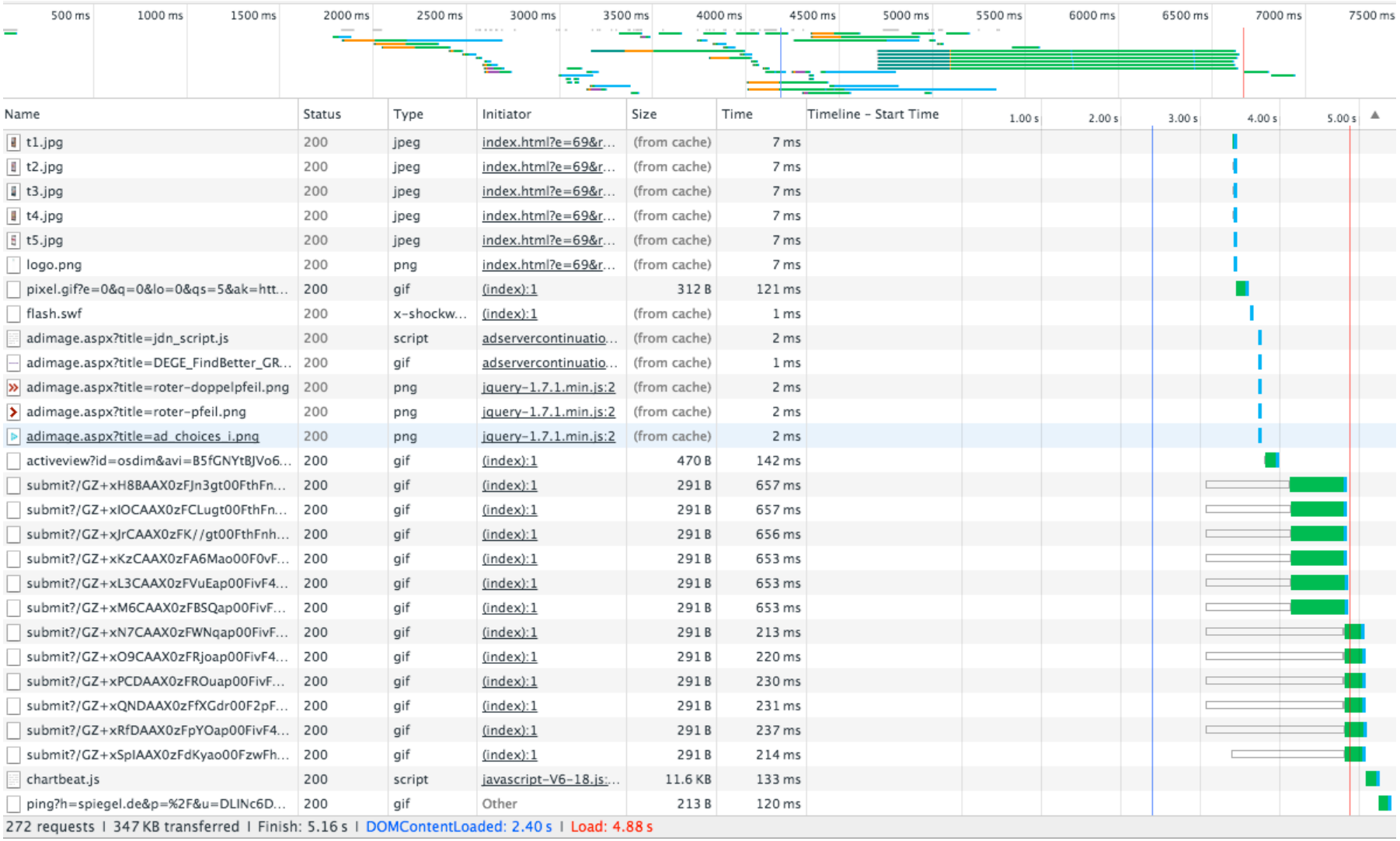
- Performance
- Privacy
- Problematisches State Handling
- Kaputte Infrastruktur (Proxy Server)

Performance

- Parallele Requests
- Durchsatz vs. Latenz
- Pipelining

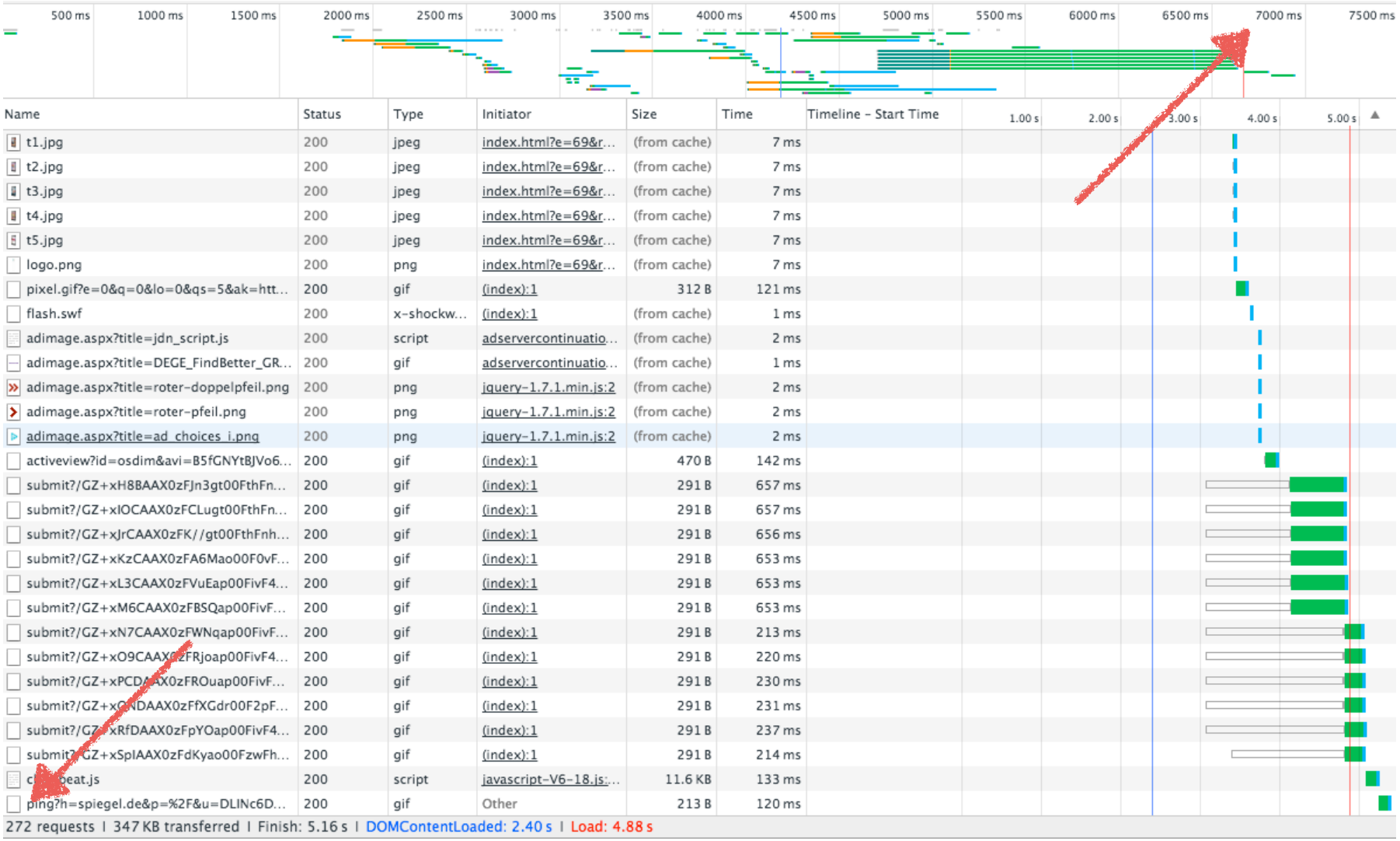
Performance

http://www.spiegel.de



Performance

http://www.spiegel.de



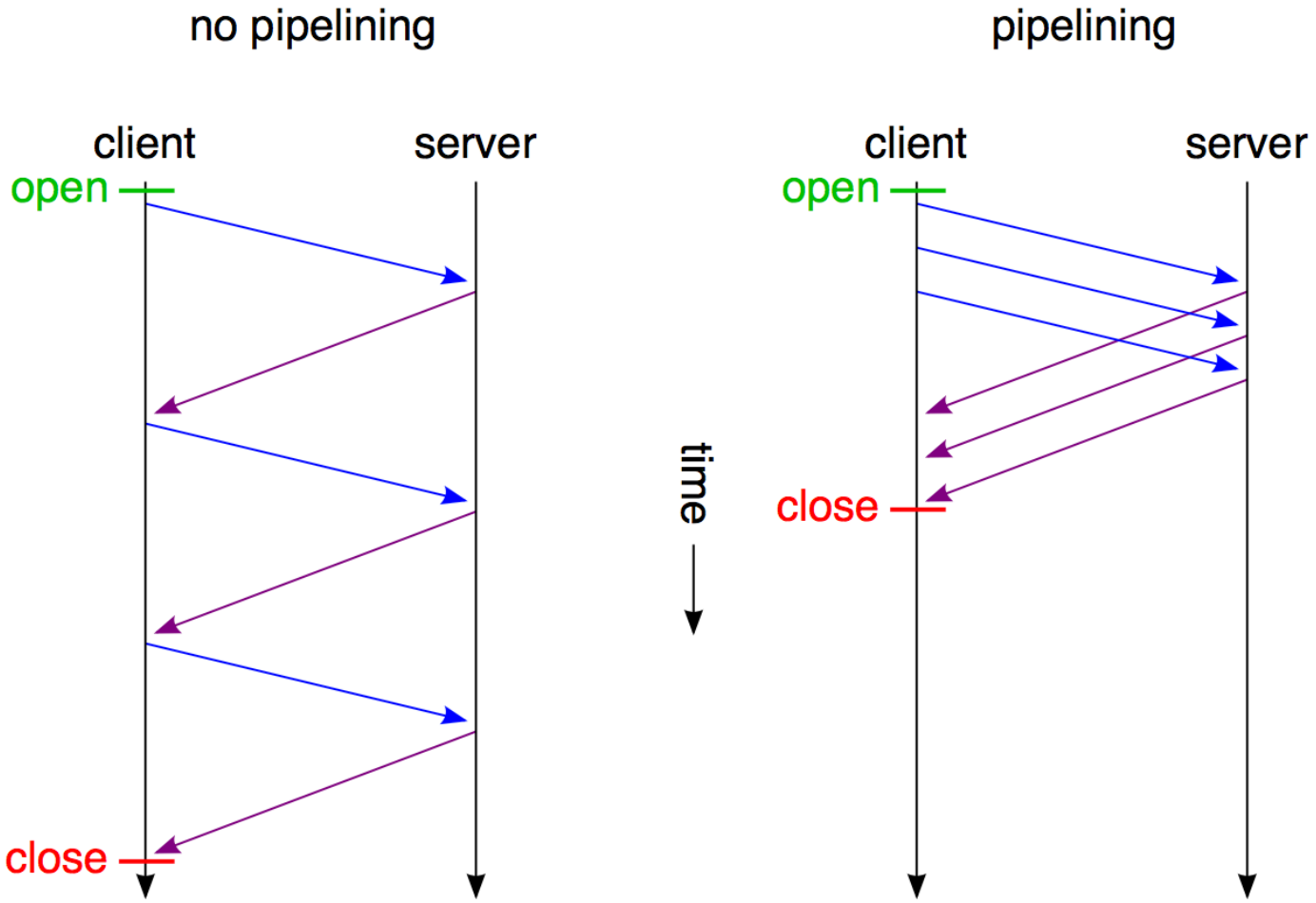
Performance: Latenz

- typischerweise haben wir im Internet eine Latenz von 30ms - 200ms
- Über Mobiles Netz oder Satelitenlink kann es auch mehr werden.
- Eigentlich ist das egal, solange wir Requests Parallel absetzen
- Ein Browser macht aber nur 6 (früher 2) TCP Connections zu einem Server auf.
- 272 Requests/Seite / 6 parallel * 200ms -> langsam

Performance: Pipelining

- Ein Client kann mehrere Requests über ein und dieselbe HTTP Connection schicken, ohne die Antworten des Servers abzuwarten.
- Da es sein kann, dass der Server kein Keep-Alive unterstützt, müssen der zweite und alle folgenden Requests idempotent sein, da sie eventuell verloren gehen können.
- Server unterstützen in der Regel Pipelining ohne Probleme

Performance: Pipelining



Performance: Pipelining

Implementation in web browsers [edit]

Out of all the major browsers, only Opera based on Presto layout engine had a fully working implementation that was enabled by default. In all other browsers HTTP pipelining is disabled or not implemented.[3]

- Internet Explorer 8 does not pipeline requests, due to concerns regarding buggy proxies and head-of-line blocking.[6]
- Mozilla browsers (such as Mozilla Firefox, SeaMonkey and Camino) support pipelining; however, it is disabled by default.[7][8] Pipelining is disabled by default to avoid issues with misbehaving servers.[9] When pipelining is enabled, Mozilla browsers use some heuristics, especially to turn pipelining off for older IIS servers.[10]
- Konqueror 2.0 supports pipelining, but it's disabled by default.[*citation needed*]
- Google Chrome previously supported pipelining, but it has been disabled due to bugs and problems with poorly behaving servers. [11]

https://en.wikipedia.org/wiki/HTTP_pipelining

Problematisches State Handling

- Keep Alive ist gut, aber es gibt eine race condition beim Schliessen der Verbindung durch den Server.
- Als Resultat, werden oftmals nur idempotente Requests über eine bereits offene Keep-Alive connection geschickt
- Jeder POST/PUT erzeugt eine neue Connections.
- Kein wirkliches Problem, aber auch nicht schön.

Kaputte Infrastruktur

- Proxy Server verhindern effektiv die Erweiterung von HTTP/1.1, da sich viele nicht an den HTTP Standard halten und sich deswegen Erweiterungen nicht durchsetzen lassen.
- Siehe Pipelining

HTTP/2

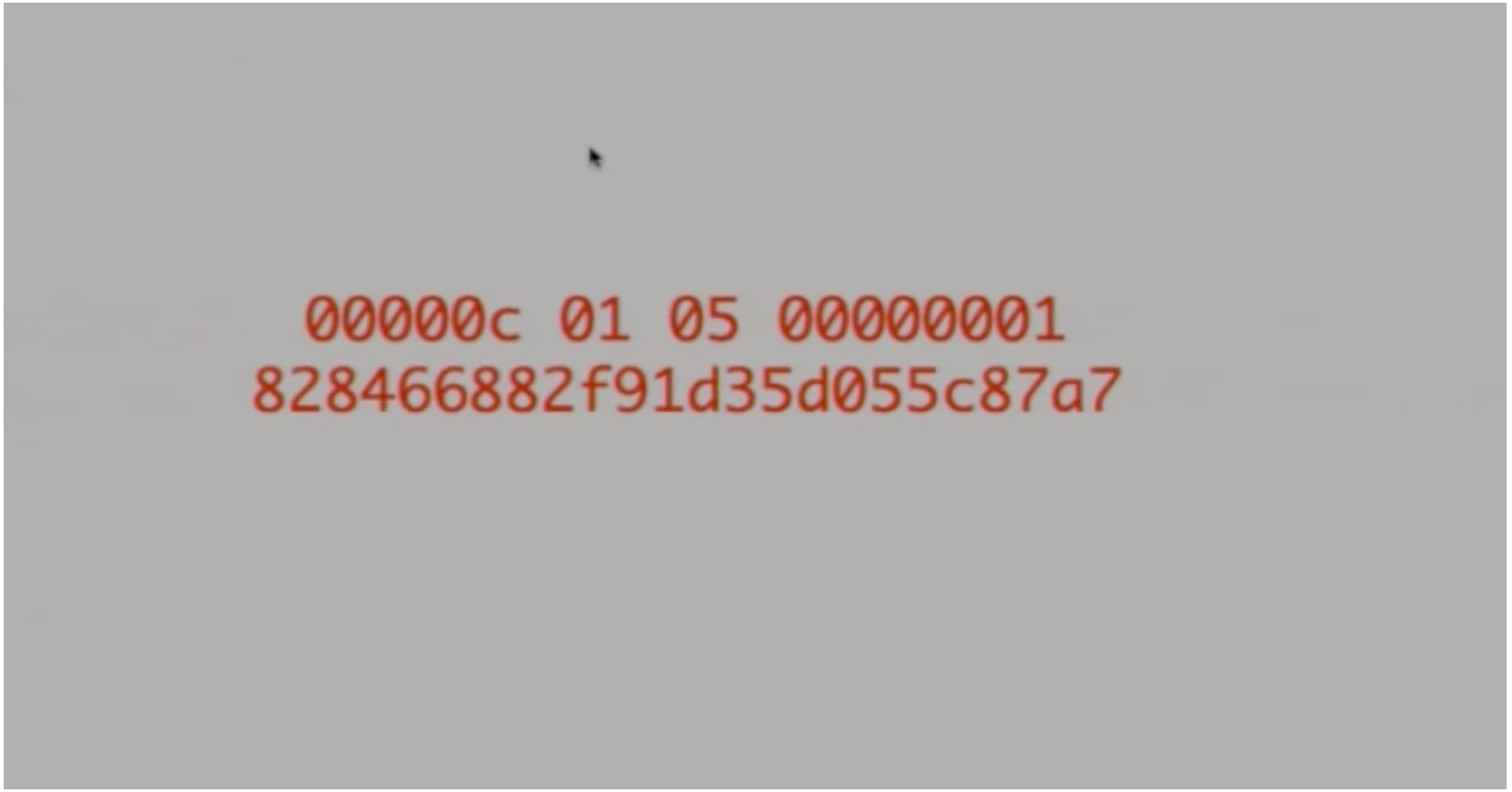
SPDY

Vorgeschichte: SPDY

- Google begann mit eigenen Experimenten, um die Performance Probleme von HTTP/1.1 zu lösen.
- Wenn Googles Chrome Browser mit einem Google Server redete, benutzten sie eventuell SPDY statt HTTP.
- Ausgehandelt über TLS Protokoll Erweiterung NPN, später ALPN
- Da TLS, konnte auch kein Proxy dazwischen Müll bauen.
- Spielwiese für Experimente
- Nachdem Google das Veröffentlichte, machte Facebook(?) den Vorstoss das als neuen RFC unter HTTPbis zu etablieren.

HTTP/2

HTTP/2



https://archive.fosdem.org/2015/schedule/event/http2_go/

HTTP/2

- Binäres Protokoll nach RFC 7540
- Paket(Frame)- und Stream-orientiert, mehrere Streams in einer TCP connection.
- Request und Response Header werden mit HPACK (RFC 7541) komprimiert
- Die Semantik eines einzelnen HTTP Request/Response Paares hat sich nicht geändert. Keine Anpassung der Applikation notwendig! Load-Balancer können HTTP/2 im Frontend nach HTTP/1.1 im Backend übersetzen.

HTTP/2 Frame

Eine HTTP/2 Connection besteht aus Frames

4.1. Frame Format

All frames begin with a fixed 9-octet header followed by a variable-length payload.

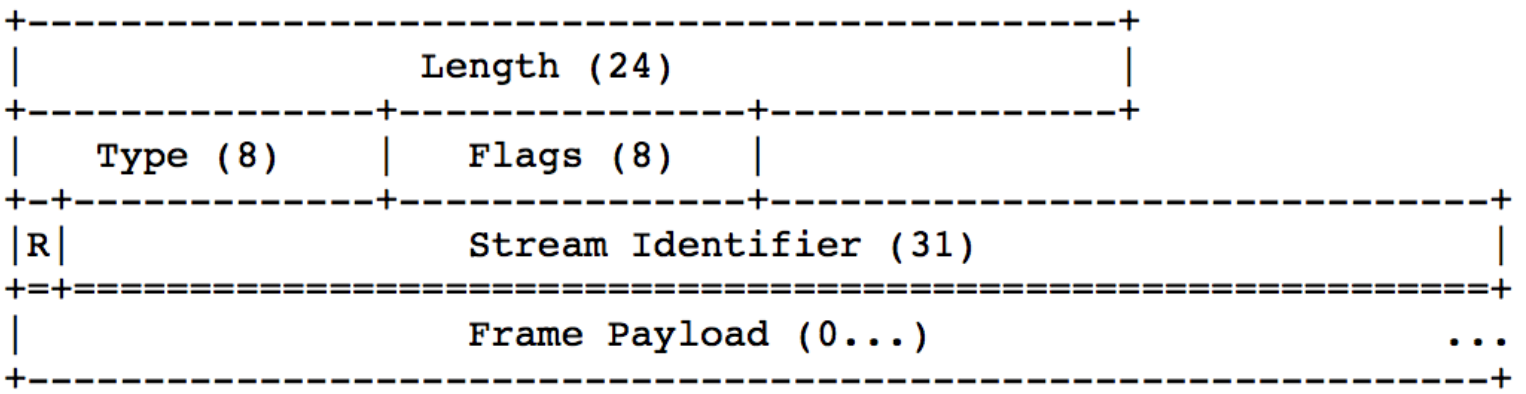


Figure 1: Frame Layout

https://tools.ietf.org/html/rfc7540

HTTP/2 Frame

- Length: 24bit - Maximalgröße eines Paketes (nicht eines Request), ohne weiteres dürfen aber nur 2^16 bytes gesendet werden.
- Type:
  - Header: HEADER, CONTINUATION
  - Body: DATA
  - Control: SETTINGS, RST_STREAM, GOAWAY
  - Control: PING, WINDOW_UPDATE, PRIORITY, PUSH_PROMISE
- Flags: END_HEADER, END_STREAM

HTTP/2 Frame

Eine HTTP/2 Connection besteht aus Frames

4.1. Frame Format

All frames begin with a fixed 9-octet header followed by a variable-length payload.

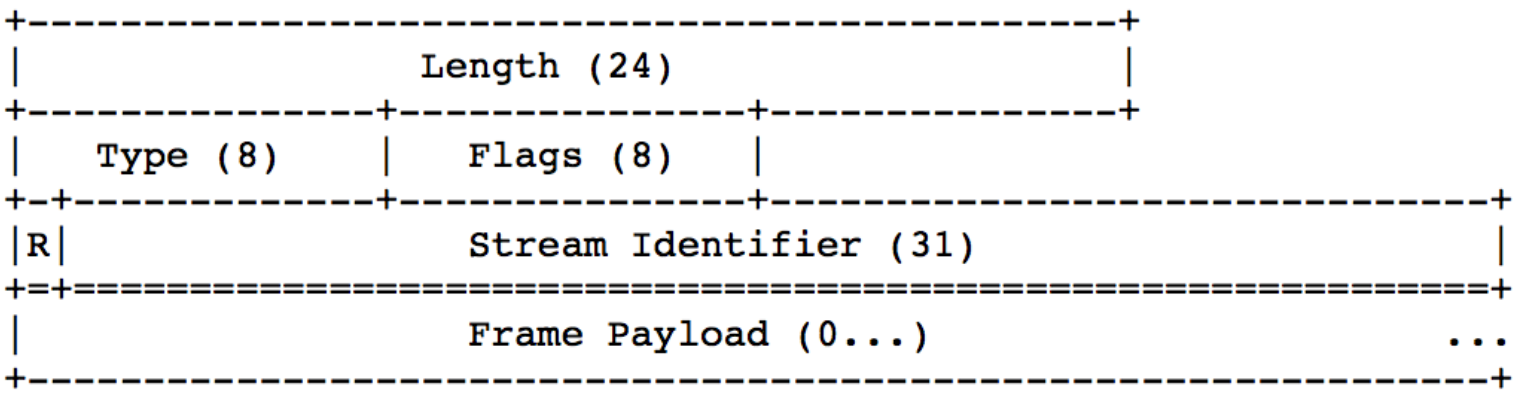


Figure 1: Frame Layout

https://tools.ietf.org/html/rfc7540

HTTP/2 Frame

- Stream-ID unterscheidet verschiedene sequentielle Datenströme. Stream-IDs für neue Stream sind monoton steigend. Mehrere Streams pro TCP connection können gleichzeitig aktiv sein.
- Auch der Server kann einen Stream initiieren (PUSH_PROMISE)

HTTP/2 Frame

- Binäres Format hat Vorteile:
  - Einfach zu parsen
  - Es ist klar, welches Daten zu welchem Request gehören und wann dieser zu Ende ist. Es ist klar, was Header und was Daten sind. Kein Response Splitting etc. auf dieser Ebene)
  - Quiz: Wer kennt alle Regeln, wann ein HTTP/1.1 Response zu Ende ist?

HTTP/2 Streams

- Ein Stream in HTTP/2 entspricht in etwa einer connection in HTTP/1.1
- Allerdings wird auf einem Stream nur ein Request mit seinem Reponse durchgeführt. Danach wird der Stream geschlossen.
- Ausnahmen wie zwei Responses zu einem Request ("100 Continue" + "200 OK" mal aussen vor)
- Ich habe das so nirgends explizit gelesen, aber alle Beispiel und das State-Diagramm aus dem RFC implizieren das.

HTTP/2 Header

- Ein Header besteht aus einer Liste von Key, Value Paaren.
- Header Name sind lowercase
- Es gibt spezielle Pseudo header, diese müssen vorhanden sein
  - :method ("GET")
  - :scheme ("https")
  - :path ("/index.cgi?foo=42")
  - :authority ("www.example.com") - anstelle HOST header, optional
  - :status (200 - response only)

HTTP/2 Header

- Der "cookie" header ist im RFC speziell behandelt und darf in mehrere Header aufgespalten werden (ist sogar empfohlen für compression)
- Andere Header können auch aufgespalten werden, sind dann aber equivalent zu einer Komma-separierten Liste der Werte (beim Cookie header Semikolon sepeariert)
- Ob das gut geht?

HTTP/2 Header

- Header werden komprimiert (RFC 7541, 55 Seiten)
- compression ist per connection und stateful.
- in SPDY gab es deflate, das hat aber Probleme (siehe CRIME attack)
- Beide Seiten können Größe der Kompressionstabellen aushandeln und auch auf 0 setzen -> keine Kompression.
- Komprimierung führt dazu, das Dinge wie der User-Agent oder ein Session Cookie in der Regel nur einmal pro TCP connection übertragen werden und ansonsten nur referenziert werden.

HTTP/2 Frame

- Length: 24bit - Maximalgröße eines Paketes (nicht eines Request), ohne weiteres dürfen aber nur 2^16 bytes gesendet werden.
- Type:
  - Header: HEADER, CONTINUATION
  - Body: DATA
  - Control: SETTINGS, RST_STREAM, GOAWAY
  - Control: PING, WINDOW_UPDATE, PRIORITY, PUSH_PROMISE
- Flags: END_HEADER, END_STREAM

HTTP/2 PUSH_PROMISE

- Der Server darf auch Streams Richtung Client initiieren.
- So ein Stream beginnt mit einem PUSH_PROMISE.
  - Das sagt in etwa: "Ich schicke dir gleich die Antwort auf folgenden Request, den du mir bestimmt gleich schicken willst"
  - Der Server kann dem Client also sagen: Ok, du willst diese Webseite, dann willst du bestimmt auch gleich diese JS/CSS/ Bilder
- Der Client kann das ignorieren, kann auch dem Server in einem SETTINGS frame sagen, dass er keine PUSH_PROMISES will.

HTTP/2 Frame

- Length: 24bit - Maximalgröße eines Paketes (nicht eines Request), ohne weiteres dürfen aber nur 2^16 bytes gesendet werden.
- Type:
  - Header: HEADER, CONTINUATION
  - Body: DATA
  - Control: SETTINGS, RST_STREAM, GOAWAY
  - Control: PING, WINDOW_UPDATE, PRIORITY, PUSH_PROMISE
- Flags: END_HEADER, END_STREAM

HTTP/2 Flow Control

- Jetzt wird es haarig ....
- Da HTTP/2 mehrere parallele Datenströme in einer TCP Connection implementiert, kann (muss) es sich auch um Flow Control kümmern
- Ströme können Prioritäten bekommen (JS und CSS ist für den Browser wichtiger als Bilder)
- Es wird im Prinzip TCP Flow Control nachgebildet (mit Puffergrößen und ACK Paketen)
- Das sind die layering violations und needless complexity aus

HTTP/2 und HTTP/1.x parallel

- Es gibt zwei theoretische Möglichkeiten, HTTP/2 auszuhandeln
  - via Update Header in HTTP/1.x
  - via TLS ALPN (Application Layer Protocol Negotiation) extension
- In der Praxis gibt es HTTP/2 nur via TLS.

Support - Browser

- Chrome 41
- Firefox 36 (35?)
- Safari 9
- iOS 9.1
- IE 11 auf Windows 10 / Edge

Support - Browser

https://http2.akamai.com/demo
Chrome 46, Web Developer Console

Support - Browser

https://http2.akamai.com/demo
Chrome 46, Web Developer Console

Support - Server

- Apache Web Server 2.4.17
  - braucht lib nghttp2 und configuration
- nginx web server (1.9.6)
  - braucht spezielle Übersetzung und configuration
- IIS in Windows 10 und Windows server 2016 (sagt Wikipedia)
- Akamai (HTTP/2 (h2) is incredibly exciting)

Support - Tools

- curl (libcurl 7.43)

```
$ curl -v  -o /dev/null --http2 https://http2.akamai.com/demo
```

- wireshark - upcoming wireshark 2 (bei mir Version 1.99.7) sollte HTTP/2 können. Nicht getestet.
- h2load (teil von nghttp2), h2spec, h2i (teil von Go http2)
- Programmiersprachen:
  - Go (ab voraussichtlich1.6 in der Standard Library)
  - Python (hyper 0.5 + other)
  - Perl, C, C++ wahrscheinlich via lib nghttp2 und bindings.

```
$ h2i http2.akamai.com
Connecting to http2.akamai.com:443 ...
Connected to 184.85.1.180:443
Negotiated protocol "h2"
...
h2i>
```

```
$ h2i http2.akamai.com
Connecting to http2.akamai.com:443 ...
Connected to 184.85.1.180:443
Negotiated protocol "h2"
...
h2i> headers
(as HTTP/1.1)> GET /demo HTTP/1.1
(as HTTP/1.1)> Host: http2.akamai.com
(as HTTP/1.1)>
Opening Stream-ID 1:
 :authority = http2.akamai.com
 :method = GET
 :path = /demo
 :scheme = https
```

```
$ h2i http2.akamai.com
Connecting to http2.akamai.com:443 ...
Connected to 184.85.1.180:443
Negotiated protocol "h2"
...
h2i> headers
(as HTTP/1.1)> GET /demo HTTP/1.1
(as HTTP/1.1)> Host: http2.akamai.com
(as HTTP/1.1)>
Opening Stream-ID 1:
 :authority = http2.akamai.com
 :method = GET
 :path = /demo
 :scheme = https
[FrameHeader HEADERS flags=END_HEADERS stream=1 len=573]
  :status = "200"
  server = "Apache" (SENSITIVE)
  content-type = "text/html" (SENSITIVE)
  etag = "\"74cc7d8c2111d6ce9e673c931de0db17:1435291619\"" (SENSITIVE)
  ...
[FrameHeader DATA stream=1 len=1279]
  "<html>\n<head lang=\"en\">\n
....
```

```
$ h2i http2.akamai.com
Connecting to http2.akamai.com:443 ...
Connected to 184.85.1.180:443
Negotiated protocol "h2"
...
h2i> headers
(as HTTP/1.1)> GET /demo HTTP/1.1
(as HTTP/1.1)> Host: http2.akamai.com
(as HTTP/1.1)>
Opening Stream-ID 1:
 :authority = http2.akamai.com
 :method = GET
 :path = /demo
 :scheme = https
[FrameHeader HEADERS flags=END_HEADERS stream=1 len=573]
  :status = "200"
  server = "Apache" (SENSITIVE)
  content-type = "text/html" (SENSITIVE)
  etag = "\"74cc7d8c2111d6ce9e673c931de0db17:1435291619\"" (SENSITIVE)
  ...
[FrameHeader DATA stream=1 len=1279]
  "<html>\n<head lang=\"en\">\n
....
[FrameHeader DATA flags=END_STREAM stream=1 len=0]
```

```
$ h2i http2.akamai.com
Connecting to http2.akamai.com:443 ...
Connected to 184.85.1.180:443
Negotiated protocol "h2"
...
h2i> headers
(as HTTP/1.1)> GET /demo HTTP/1.1
(as HTTP/1.1)> Host: http2.akamai.com
(as HTTP/1.1)>
Opening Stream-ID 1:
 :authority = http2.akamai.com
 :method = GET
 :path = /demo
 :scheme = https
[FrameHeader HEADERS flags=END_HEADERS stream=1 len=573]
  :status = "200"
  server = "Apache" (SENSITIVE)
  content-type = "text/html" (SENSITIVE)
  etag = "\"74cc7d8c2111d6ce9e673c931de0db17:1435291619\"" (SENSITIVE)
  ...
[FrameHeader DATA stream=1 len=1279]
  "<html>\n<head lang=\"en\">\n
....
[FrameHeader DATA flags=END_STREAM stream=1 len=0]
...
[FrameHeader GOAWAY len=8] Last-Stream-ID = 1; Error-Code = NO_ERROR (0)
```

Gotchas

- TLS 1.2 required, https://tools.ietf.org/html/rfc7540#section-9.2
- Update von HTTP/1.x funktioniert nicht mit jedem Server
- Es braucht tools, um HTTP/2 zu sprechen, telnet reicht nicht mehr aus

Probleme?

- "Intermediaries" (proxy und loadbalancer) brauchen eventuell mehr Ressourcen für das Header parsen. Das ist notwendig, wenn man Header modifizieren will und kostet ordentlich Speicher.

Probleme?

- Ideen für Pentest:
  - Was passiert bei Stream-ID 2^31. RFC hat eine Meinung dazu, aber Implementationen? Was passiert wen ich das reserved Bit vor der Stream-ID setze?
  - Der Standard sagt klar wo Pseudo-Header stehen dürfen und wo nicht und was zu tun ist. Halten sich alle Implementationen daran? Was passiert wenn ich einen zweite :path header am ende des Header Blocks angebe? Kann ich eine WAF umgehen?

Danke für's zuhören

Fragen?

Diskussion!