Web-basierte Systeme

06: Browser Kommunikationsschnittstellen

Wintersemester 2025

Rüdiger Kapitza

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Browser Kommunikationsschnittstellen

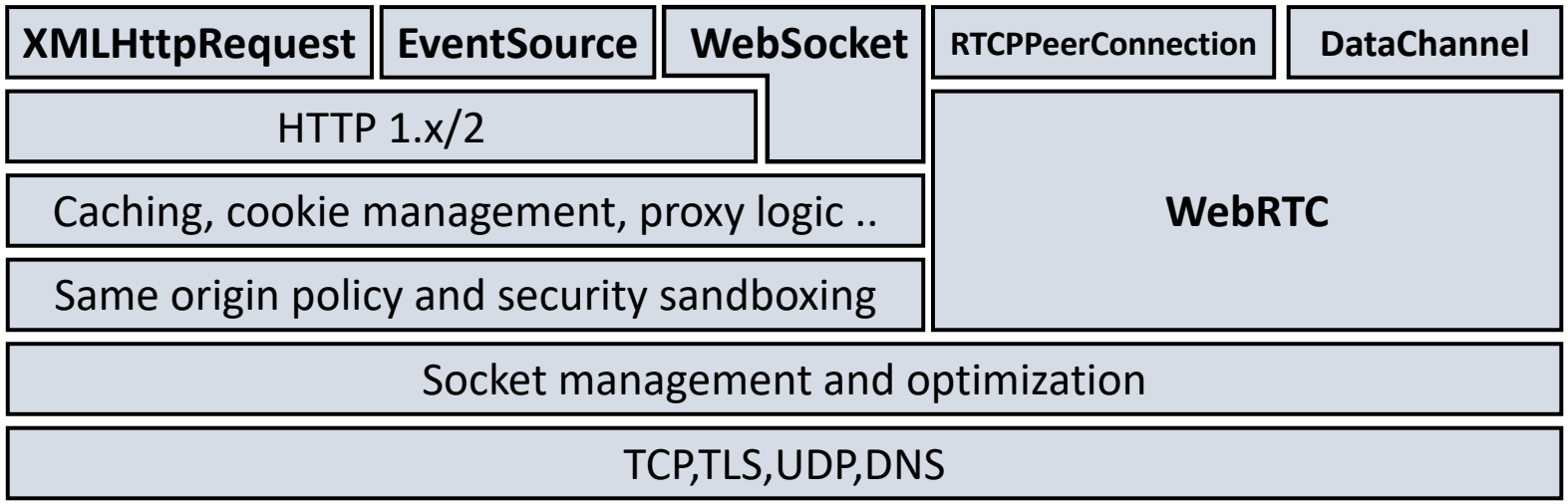
Vorläufiger Vorlesungsplan

16. Oktober	Einführung und Darstellung von Webseiten
23. Oktober	HTML und CSS
30. Oktober	Hypertext Transfer Protocol
6. November	Browser Schnittstellen
13. November	Kommunikationsschnittstellen im Browser
20. November	WebAssembly
27. November	Architektur moderner Browser
4. Dezember	Clientseitige Architekturmuster und serverseitige
11. Dezember	Implementierung von Web-basierten Systemen
	Vorbereitung Papieranalyse
8. Januar	Papieranalyse
15. Januar	**Lastverteilung durch Zwischenspeicher**
22. Januar	Aspekte von Web Sicherheit
29. Januar	Web3
5. Februar	Zusammenfassung und Ausblick

Browser Kommunikationsschnittstellen

Zielsetzung der Lerneinheit

- Verständnis der Kommunikationsschnittstellen zwischen Browser und Server sowie der Vor- und Nachteile der jeweiligen Mechanismen



XMLHttpRequest

XMLHttpRequest

Entstehung und Motivation von XMLHttpRequest (XHR)

- Ursprünglich erforderte die Aktualisierung einer Webanwendung das Laden einer neuen Seite
- Mit Hilfe von XHR war es nun möglich, direkt aus einem Skript heraus Anfragen an einen Server zu stellen, die im Hintergrund ablaufen
  - Basis für die *Asynchronous JavaScript and XML (AJAX)*-Revolution
- XHRs wurden mit dem Internet Explorer 5 eingeführt und quasi durch *Zufall* mit XML verknüpft
- Mozilla und weitere Browserhersteller zogen recht schnell nach
- 2006 erste W3C Working Draft Spezifikation
- 2008 gab es einen weiteren Entwurf mit *XMLHttpRequest Level 2*
- 2011 wurden beide Versionen zusammengeführt und es gibt seitdem nur noch eine XMLHttpRequest-Spezifikation
  - (`https://xhr.spec.whatwg.org`)

XMLHttpRequest

Einbettung von XHR in den Browser

- XHR ist eine recht komfortable Kommunikationsschnittstelle und regelt viele Details im Browser
  - Verwalten der Anfrage, caching, Authentifizierung, etc.
  - Sicherheitsmechanismen werden etabliert bzw. genutzt
- Zusätzliche HTTP-Header können einer Anfrage hinzugefügt werden (via `setRequestHeader()`), aber die Art der Header ist beschränkt:
  - Beispielsweise sind `Access-Control-*`, `Cookie`, `Origin` und `Refer` nicht möglich! (Der Browser verwirft sie.)
  - Wichtig, um sicherzustellen, dass der `Origin`-Header nicht verändert werden kann (*same-origin-policy*[1])
  - Zugriff auf Daten (cookies) von anderen Websites wäre möglich!

[1] Zwei URLs haben den gleichen *origin* falls Protokoll, Port und Host gleich sind.

XMLHttpRequest

XMLHttpRequest Beispiel

```
1  function reqListener () {
2    console.log(this.responseText);
3  }
4
5  var xhr = new XMLHttpRequest();
6  xhr.addEventListener("load", reqListener);
7  xhr.open("GET", "http://www.example.org/blub.txt");
8  xhr.send();
```

XMLHttpRequest

Unterstützte Datenformate

- `ArrayBuffer` - Binärdaten fester Länge
- `Blob` - Binärdaten (die im Browser nicht verändert werden)
- `Document` - HTML oder XML Daten
  - Bsp. für Nutzung: `elem.innerHTML = xhr.responseText;`
- `JSON`
  - Bsp. für Nutzung: `JSON.parse(xhr.responseText);`
- `Text`

Bestimmung des Datenformates

- Entweder über den HTTP content-type der Antwort
  oder explizit durch die Anwendung
  (via `xhr.responseType = 'blob';`)

XMLHttpRequest

Ereignisse

- `readystatechange`: wenn sich der Status geändert hat
- `loadstart`: Anfrage wurde gestartet.
- `progress`: Daten werden gesendet oder empfangen.
- `load`: Anfrage wurde erfolgreich abgeschlossen
- `loadend`: Anfrage wurde beendet (egal ob erfolgreich/gescheitert)
- `abort` und `error`

Mögliche Verarbeitungszustände via `readyState`

0. **UNSENT**: `open( )` ist noch nicht aufgerufen worden
1. **OPENED**: `send( )` aufgerufen
2. **HEADERS_RECEIVED**: send() aufgerufen und headers verfügbar
3. **LOADING**: Download läuft & Teilergebnis verfügbar
4. **DONE**: Operation ist abgeschlossen

XMLHttpRequest

XHR Beispiel für das Landen von Binärdaten

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('GET', '/images/some.jpg');
3  xhr.responseType = 'blob';
4
5  xhr.onload = function() {
6    if (this.status == 200) {
7      var img = document.createElement('img');
8      img.src = window.URL.createObjectURL(this.response);
9      img.onload = function() {
10         window.URL.revokeObjectURL(this.src);
11     }
12     document.body.appendChild(img);
13   }
14 };
15 xhr.send();
```

XMLHttpRequest

XHR Beispiel für Upload

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('POST','/upload');
3  xhr.onload = function() { ... };
4  xhr.send("text string");
```

- Up- und Download sind sehr ähnlich zueinander
- Eine stromorientierte Übertragung wird nicht unterstützt. Sie kann jedoch durch das Senden mehrerer Anfragen emuliert werden.

XMLHttpRequest

XHR Fortschrittsüberwachung

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('GET','/resource');
3  xhr.timeout = 5000;
4  ...
5  var onProgressHandler = function(event) {
6    if(event.lengthComputable) {
7      var progress = (event.loaded / event.total) * 100;
8      ...
9    }
10 }
11 xhr.addEventListener('progress', onProgressHandler);
12 xhr.send();
```

- Über den Fortschritt eines Uploads kann man sich informieren:

```
xhr.upload.addEventListener('progress', onProgressHandler);
```

XMLHttpRequest

Echtzeitbenachrichtigung

- Es sollen neue Meldungen angezeigt werden, sobald sie auf der Serverseite verfügbar sind
- Naive Lösung mit XHR durch *polling*:

```
1  function checkUpdates(url) {
2    var xhr = new XMLHttpRequest();
3    xhr.open('GET', url);
4    xhr.onload = function() { ... };
5    xhr.send();
6  }
7  setInterval(function() { checkUpdates('/up') }, 60000);
```

- Ist das gut? Geht das besser?

XMLHttpRequest

Naives Polling

- Naives Polling ist kritisch zu sehen, da ein blankes Anfrage/Antwort-Paar auch ohne neue Informationen 800 Byte Datentransfer verursacht!
- Dies ist ineffizient und nicht besonders energiefreundlich für mobile Endgeräte
- Außerdem kann sich die Zustellung neuer Nachrichten um bis zu einem Polling-Intervall verzögern!

Gibt es dazu eine Alternative?

XMLHttpRequest

Verzögertes Polling oder auch *Comet*

- Wenn serverseitig keine neue Nachricht vorliegt, wird die Antwort verzögert, bis dies der Fall ist.
- Client sendet nach Erhalt einer Antwort eine neue Anfrage

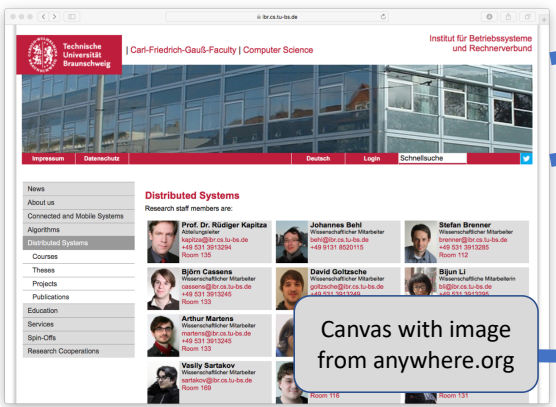
```
1  function checkUpdates(url) {
2    var xhr = new XMLHttpRequest();
3    xhr.open('GET', url);
4    xhr.onload = function() {
5      ...
6      checkUpdates('/updates');
7    };
8    xhr.send();
9  }
10 checkUpdates('/updates');
```

- Ist das jetzt in jedem Fall eine gute Lösung?

XMLHttpRequest

Cross-Origin Resource Sharing (CORS)

- Ein Skript soll Ressourcen von einem fremden Server abfragen
- Lösung bildet CORS



Canvas with image from anywhere.org

Web document
ibr.cs.tu-bs.de

Main request defines origin

GET /index.hml HTTP 1.1

GET /layout.css HTTP 1.1

Same origin request
(always allowed)

Webserver
ibr.cs.tu-bs.de

GET /me.png HTTP 1.1

Cross-origin request
(controlled by CORS)

Webserver
anywhere.org

XMLHttpRequest

Cross-Origin Resource Sharing (CORS)

- Browser stellt durch Header erweiterte Anfrage an den Ursprungsserver, der dann entscheiden kann

Anfrage mit fremden `Origin:` kontrolliert durch den Browser

```
1 GET /resources/public-data/ HTTP/1.1
2 Host: bar.other
3 ...
4 Origin: http://foo.example
```

Antwort: Uneingeschränkter Zugriff ist möglich (?!)

```
1 HTTP/1.1 200 OK
2 ...
3 Access-Control-Allow-Origin: *
4
5 [Data]
```

XMLHttpRequest

Cross-Origin Resource Sharing (CORS)

- Man unterscheidet zwischen einfachen bzw. *simple cross-origin requests* und erweiterten Anfragen
- Einfache Anfragen sind auf bestimmte HTTP-Methoden (GET, POST, HEAD) beschränkt, während erweiterte Anfragen für Cookies und eigene Header erforderlich sind
- In diesem Fall muss eine Voranfrage (*preflight request*) gestellt werden
- Konsequenz ist ein extra Nachrichtenaustausch – der aber zwischengespeichert werden kann

XMLHttpRequest

CORS Beispiel für Voranfrage

```
1 OPTIONS /resource.js HTTP/1.1
2 Host: thirdparty.com
3 Origin: http://example.com
4 Access-Control-Request-Method: POST
5 Access-Control-Request-Headers: My-Custom-Header
```

```
1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: http://example.com
3 Access-Control-Allow-Methods: GET, POST, PUT
4 Access-Control-Allow-Headers: My-Custom-Header
```

XMLHttpRequest

Zusammenfassung und Fazit

- XHR ermöglicht interaktive Webseiten
  ohne das Laden einer neuen Seite
- Sehr komfortable und einfach, da die Mechanismen des Browsers genutzt werden
- Austausch von Echtzeitnachrichten möglich, aber nicht optimal unterstützt
  - Besser Lösung bspw. Server-Sent Events
- Stromorientierter Nachrichtenaustausch ist nicht gut unterstützt!

Server-Sent Events

Server-Sent Events

Motivation und Eigenschaften

- Server-Sent Events (SSE) setzt sich aus zwei Komponenten zusammen:
  - der EventSource-Schnittstelle des Browsers, welche die server-initiierte Zustellung von *push notifications* als DOM-Ereignisse realisiert
  - und einem stromgewahren Datenformat

- SSE stellt eine stromorientierte Nachrichtenschnittstelle bereit

Server-Sent Events

Einfaches Beispiel

```
1  var source = new EventSource("news"); //relative URL
2
3  source.onopen = function () { ... };
4  source.onerror = function () { ... };
5
6  // Register handler for event of type "fun"
7  source.addEventListener("fun", function (event) {
8    processFun(event.data);
9  });
10
11 source.onmessage = function (event) {
12   log_message(event.id, event.data);
13   if (event.id == "CLOSE") {
14     source.close();
15   }
16 }
```

Server-Sent Events

Eigenschaften der zugrundeliegenden Infrastruktur

- Etablierung der Verbindung
- Empfangene Daten werden inkrementell aufbereitet (z.B. identifizieren von Nachrichtengrenzen)
- Auslösen entsprechender Ereignisse
- Zusätzlich wird die Wiederherstellung von abgebrochenen Verbindungen unterstützt
  - Unterstützung von zuverlässigen Übertragung durch Nachrichten-`IDs`

Server-Sent Events

Event Stream Protocol und dessen Abbildung auf HTTP

- Browser versendet reguläres HTTP GET mit Anfrage nach SSE Ereignissen (`text/event-stream`)

```
1 GET /stream HTTP/1.1
2 Host: example.com
3 Accept: text/event-stream
```

Server-Sent Events

Event Stream Protocol und dessen Abbildung auf HTTP

- Server antwortet mit einem Strom von Nachrichten (chunked)

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 15000

data: First message is a simple string.

data: {"message": "JSON payload"}

event: foo
data: Message of type "foo"

id: 42
event: bar
data: Multi-line message of
data: type "bar" and id "42"
```

Server-Sent Events

Nachrichtenformat

- Server schlägt ein Intervall vor, nachdem versucht werden soll eine abgebrochene Verbindung zu reetablieren
- Es können einfache Text- bzw. JSON-basierte Nachrichten versendet werden
- Optional können Nachrichten mit einem *Typ* versehen werden
- Nachrichten können *IDs* erhalten und mehrere Zeilen in Anspruch nehmen

Server-Sent Events

Erneute Versendung bei Verbindungsabbruch

- Durch das Hinzufügen von IDs kann sich der Browser merken, welches Ereignis er vor dem Verbindungsabbruch zuletzt geliefert hat
- Diese Information wird bei einem erneuten Verbindungsaufbau durch einen `Last-Event-ID` Header an den Server übermittelt.

- Verbindungsabbruch und nun erneuter Aufbau

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream
Last-Event-ID: 42
```

- Antwort

```
HTTP/1.1 200 OK
Content-Type: text/event-stream
Connection: keep-alive
Transfer-Encoding: chunked

id: 43
data: bar foo
```

Zusammenfassung

- SSE ermöglicht einfaches vermitteln von Echtzeitinformationen
- Basis ist die verbindungs- und stromorientierte Kommunikation
- Da SSE auf HTTP basiert, ist auch eine Komprimierung möglich
- Verbindungsabbrüche werden erkannt und behoben, verlorene Nachrichten werden wiederholt gesendet
- SSE hat dennoch zwei Nachteile
  - Es bietet keine Unterstützung für eine stromorientierte Übertragung vom **Browser zum Server**
  - Lediglich eine UTF-8 basierte Übertragung ist vorgesehen – dies erschwert die effiziente Übertragung von Binärdaten
    - Base64-Kodierung – was ineffizient ist
    - Alternativer Übertragungskanal und nur Signalisierung über SSE

WebSocket

WebSocket

Motivation und Eigenschaften

- Bidirektionale stormorientierte Verbindung zum Austausch von Text und Binärdaten
- Look and Feel wie Unix-Sockets, aber mit zusätzlichen Funktionen
  - Same-origin policy wird unterstützt
  - Kompatibel zum HTTP-Ökosystem
  - Effizientes Nachrichtenformat
  - Erweiterbar
    - Bspw. kann die Nutzung eines Unterprotokolls ausgehandelt werden
- WebSocket-Schnittstelle wird durch den W3C definiert
- Protokoll zum Datenaustausch durch den RFC 6455 festgelegt
- Erweiterbarkeit wird durch die HyBi Working Group (IETF) entwickelt

WebSocket

Einfaches Beispiel

```
1  var ws = new WebSocket('wss://example.com/socket');
2
3  ws.onerror = function (error) { ... }
4  ws.onclose = function () { ... }
5  ws.onopen  = function () { ws.send("Hello server!"); }
6
7  ws.onmessage = function(msg) {
8    if(msg.data instanceof Blob) {
9      processBlob(msg.data);
10   } else {
11     processText(msg.data);
12   }
13 }
```

- Nachrichten können als Text oder Binärdaten übertragen werden

WebSocket URL Schema

- WebSocket verfügt über zwei eigene Protokollkürzel
  - `ws` für Klartextkommunikation (bspw. `ws://example.com/socket`)
  - `wss` für verschlüsselte Kommunikation via TCP und TLS
- Warum gibt es ein eigenes Protokollkürzel?
  - De facto werden WebSockets hauptsächlich verwendet, um Verbindungen zwischen Browser und Server herzustellen, aber prinzipiell ist auch eine Verwendung außerhalb denkbar
  - Die HyBi Working Group hat deshalb ein eigenes URL-Schema eingeführt

Empfang und Verarbeitung von Daten

- Über WebSockets werden ganze Nachrichten verschickt/empfangen
  - Anwendungscode muss die Daten nicht zwischenspeichern, parsen oder Nachrichten wieder zusammenzusetzen
- Nachrichten können Text/Binärdaten sein, auch abwechselnd
  - Nachrichtenformat kodiert Länge und Typ

```
1  var ws = new WebSocket('wss://example.com/socket');
2  ws.binaryType = "arraybuffer";
3
4  ws.onmessage = function(msg) {
5    if(msg.data instanceof ArrayBuffer) {
6      processArrayBuffer(msg.data);
7    } else {
8      processText(msg.data);
9    }
10 }
```

- Binärdaten (`Blob`) können automatisch in einen `ArrayBuffer` umgewandelt werden

WebSocket

Empfang und Verarbeitung von Binärdaten

- `ArrayBuffer` wird sicherlich im Arbeitsspeicher gehalten
  - Einfaches Abbilden auf Datenstrukturen möglich
  - `var usernameView = new Uint8Array(buffer, 0, 16);`
- Blob wird eher als Datei abgelegt und es wird davon ausgegangen, dass die Daten unveränderlich sind

Versenden und Verarbeitung von Daten

```
1  var ws = new WebSocket('wss://example.com/socket');
2
3  ws.onopen = function () {
4    socket.send("Hello server!");
5    socket.send(JSON.stringify({'msg': 'payload'}));
6
7    var intview = new Uint32Array(buffer);
8    socket.send(intview);
9
10   var blob = new Blob([buffer]);
11   socket.send(blob);
12 }
```

- WebSockets übertragen entweder binär oder UTF-8 kodiert – andere Formate müssen in der Anwendung vereinbart werden

WebSocket

Versenden und Verarbeitung von Daten

- Da es sich um eine asynchrone Schnittstelle handelt, kommt ein Aufruf sofort zurück, unabhängig davon, ob bereits Daten gesendet wurden oder nicht
- Um zu ermitteln wie viele Daten noch auf Browser-Seite auf Versand warten kann man `bufferedAmount` abfragen
- Mit dieser Information kann man auch Priorisierung implementieren

```
1  var ws = new WebSocket('wss://example.com/socket');
2
3  ws.onopen = function () {
4    subscribeToApplicationUpdates(function(evt) {
5      if (ws.bufferedAmount == 0)
6        ws.send(evt.data);
7    });
8  };
```

Vereinbaren eines Protokolls

- Wie sollen sich Browser und Server über das ausgetauschte Nachrichtenformat einigen?
- Implizites Verständnis (beide Seiten kennen das Format bei Beginn der Kommunikation)
- Aushandeln eines Protokolls – wird von WebSockets unterstützt

```
1  var ws = new WebSocket('wss://example.com/socket',
2                         ['appProtocol', 'appProtocol-v2']);
3
4  ws.onopen = function () {
5    if (ws.protocol == 'appProtocol-v2') {
6      ...
7    } else {
8      ...
9    }
10 }
```

WebSocket

Etablierung einer Websocket-Verbindung

- WebSockets bieten eine Reihe von Möglichkeiten und dementsprechend muss beim Verbindungsaufbau ein entsprechender Informationsaustausch stattfinden
- HTTP und die etablierte Infrastruktur wird dazu genutzt
- Konkret wird HTTP Upgrade um einige spezifische Header ergänzt
  - `Sec-WebSocket-Version` – Version welche durch den Client unterstützt wird. Aktuell Version 13 (siehe RFC6455)
  - `Sec-WebSocket-Key` – Automatisch erzeugter Schlüssel als Mechanismus zum Verbindungsaufbau (nicht sicherheitsrelevant!)
  - `Sec-WebSocket-Accept` — Antwort des Servers bzgl. des Schlüssels
  - `Sec-WebSocket-Protocol` – Festlegung eines Subprotokolls, Client sendet Liste, Server antwortet mit ausgewähltem Protokoll
  - `Sec-WebSocket-Extensions` – Mechanismus zum aushandeln von Erweiterungen

Etablierung einer Websocket-Verbindung

- Anfrage

```
GET /socket HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Protocol: appProtocol, appProtocol-v2
Sec-WebSocket-Extensions: x-webkit-deflate-message, x-custom-extension
```

- Antwort

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Access-Control-Allow-Origin: http://example.com
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: appProtocol-v2
Sec-WebSocket-Extensions: x-custom-extension
```

- Cross-Origin Resource Sharing (CORS) wird ebenfalls behandelt

Browser Kommunikationsschnittstellen

Zusammenfassung und Vergleich

- Jeder der Technologien hat sein unmittelbares Anwendungsgebiet
- WebSockets bietet eine bidirektional stromorientierte Kommunikation – vergibt aber Mechanismen des Browsers

	XMLHTTPRequest	**Server-Sent Events**	**WebSocket**
Request streaming	no	no	yes
Response streaming	limited	yes	yes
Framing mechanism	HTTP	event stream	binary framing
Binary data transfers	yes	no (base64)	yes
Compression	yes	yes	limited
Protocol	HTTP	HTTP	WebSocket
Message Overhead	HTTP/1.x 500-800 byte HTTP/2 min. 8 byte	5 byte	2-14 byte per frame

Referenzen

- Vorlesung ist an Kapitel 15, 16 und 17 des Buchs angelehnt

Literatur

[1] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, Inc., 2013.