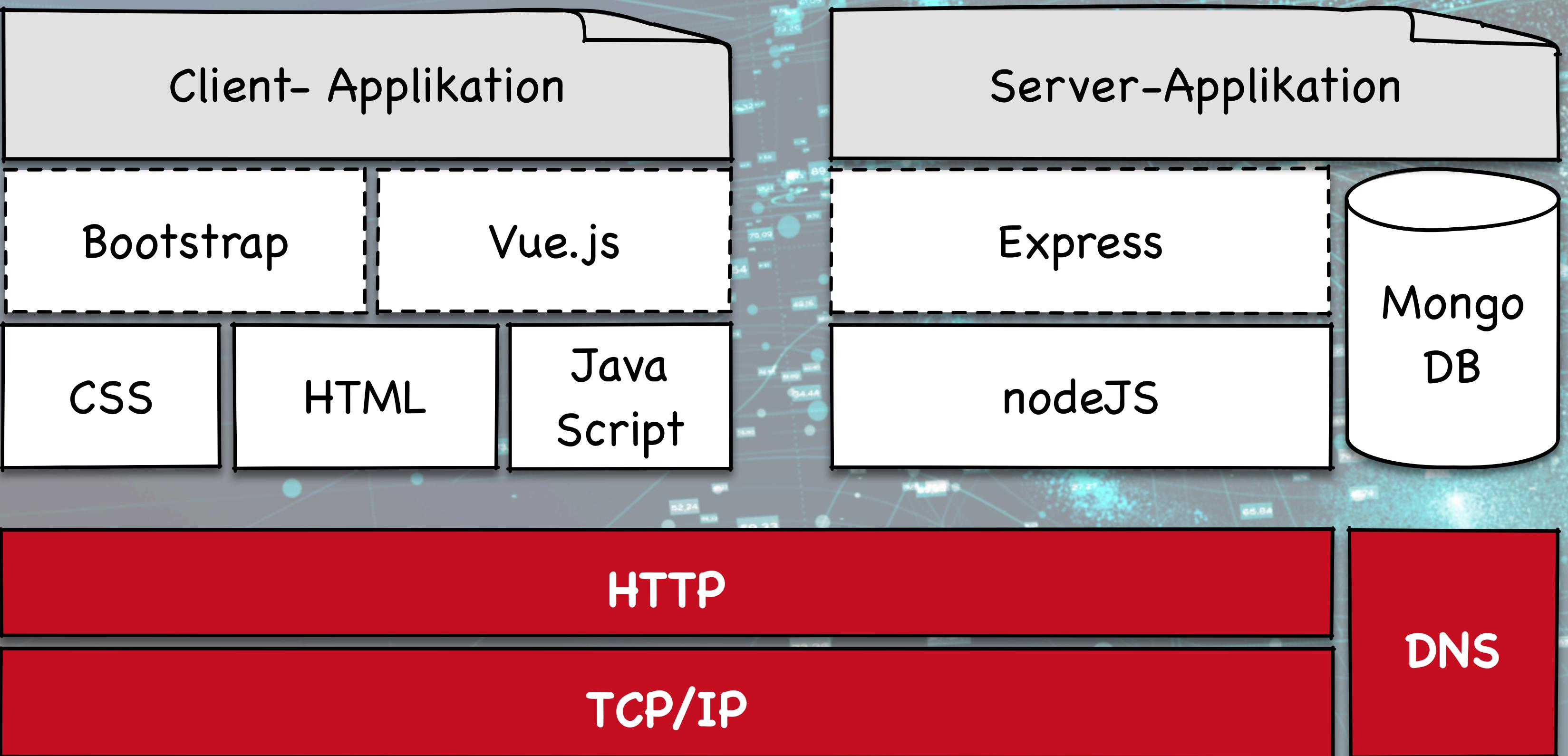


WEB TECHNOLOGIEN 2022

KAPITEL 6: KOMMUNIKATION IM WEB

PROF. DR. AXEL KÜPPER
FACHGEBIET SERVICE-CENTRIC NETWORKING I TU BERLIN & T-LABS

WEBTECHNOLOGIEN ÜBERBLICK



6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

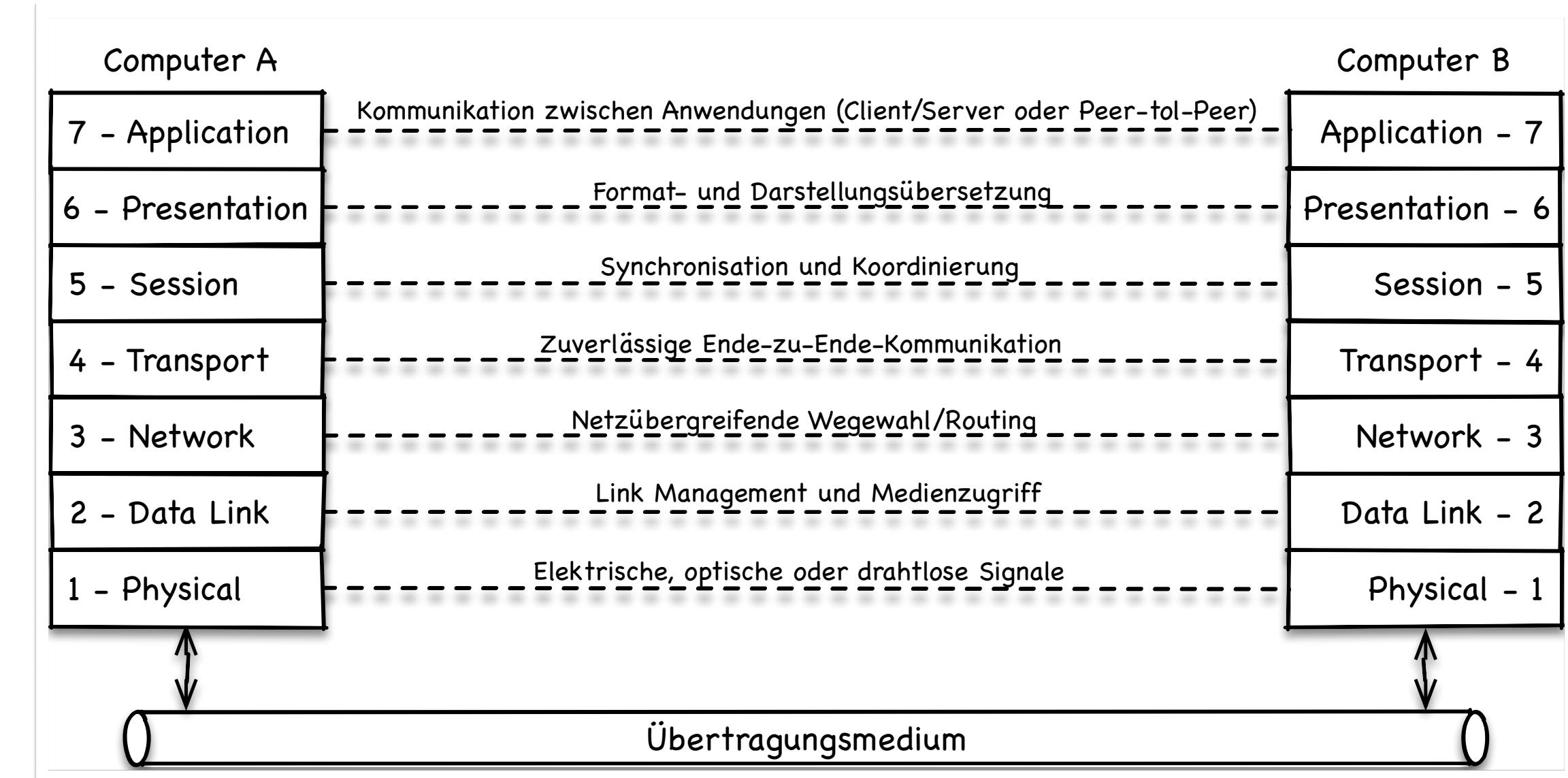
VIELE AUFGABEN - HOHE KOMPLEXITÄT



→ DATENKOMMUNIKATION IST EINE SEHR KOMPLEXE ANGELEGENHEIT MIT VIELEN TEILAUFGABEN

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

PROTOKOLLSTAPEL - OSI-REFERENZMODELL



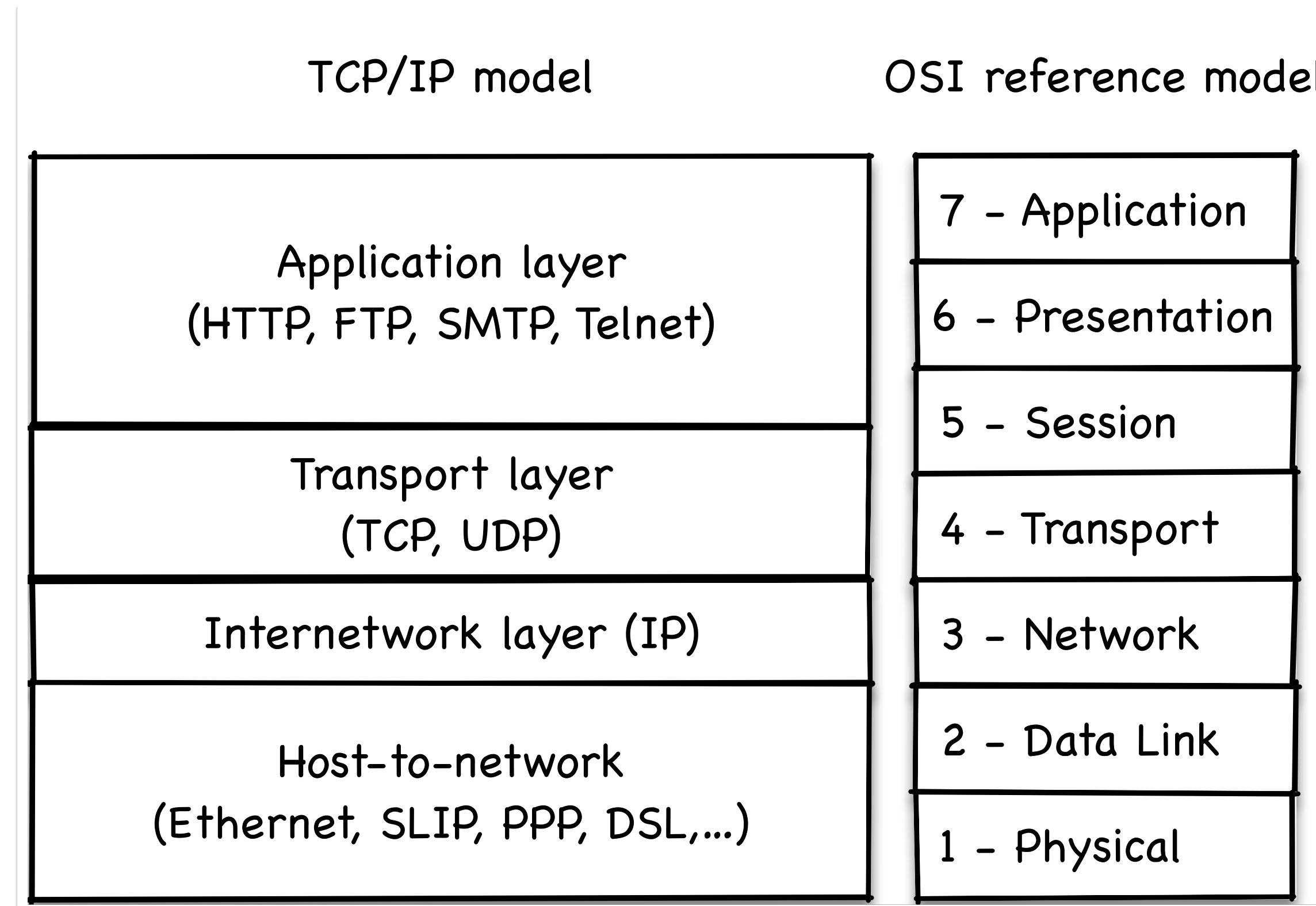
PRINZIPIEN GESCHICHTETER PROTOKOLLE

- (Theoretisches) Modell aus sieben Protokollsichten (Protocol Layers) mit jeweils begrenzten Aufgaben
- Definiert von der International Organization for Standardization der Gruppe Open Systems Interconnection (OSI), erstmals 1977
- Ziel: Beherrschung der Komplexität durch Protokolle auf verschiedenen Schichten
- Prinzip: Schicht n nutzt Dienste der Schicht $n-1$ und stellt der Schicht $n+1$ Dienste bereit
- Instanzen einer Schicht kommunizieren mit Instanzen der gleichen Schicht auf der Gegenseite

Protokolle: Sind Anfolgen, auf die man sich geeignet hat.

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

OSI-REFERENZMODELL VERSUS TCP/IP-MODELL (I)



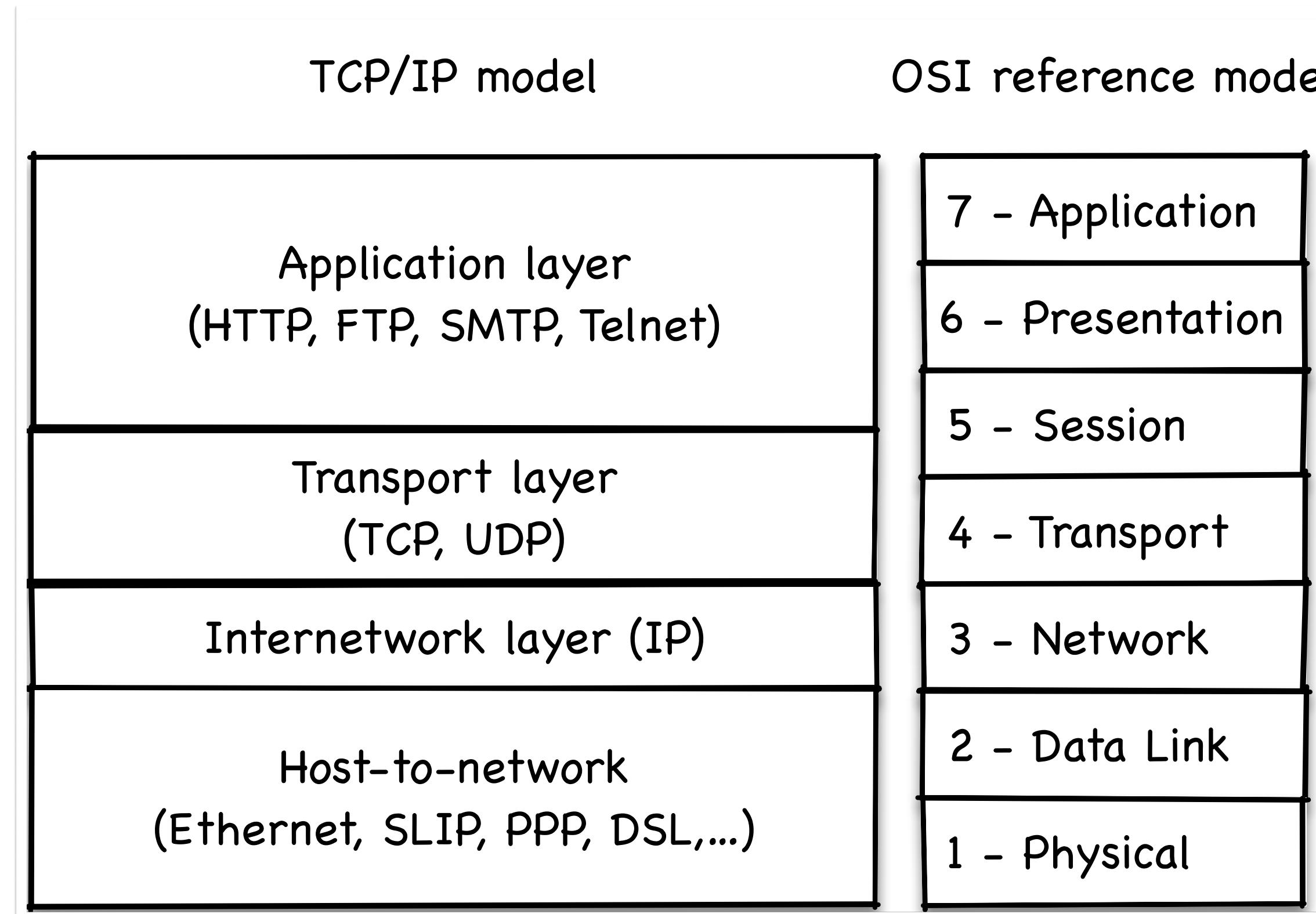
- ISO/OSI-Referenzmodell dient lediglich als Modell für konzeptionelle Betrachtungen der Datenkommunikation hat kaum praktische Relevanz
- TCP/IP-Modell: in der Praxis als De-Facto-Standard etabliertes und akzeptiertes Modell

HOST-TO-NETWORK LAYER (NETZZUGRIFF)

- Bitübertragungsschicht: Übertragung von Bits über ein Kommunikationsmedium (lokales Netz) durch Manipulation von Signalen in Abhängigkeit der zu übertragenden Bits (Modulation beim Sender) bzw. Erzeugung von Bits abhängig von den empfangenen Signalen (Demodulation beim Empfänger)
- Sicherungsschicht: Aufteilung des Datenstroms in Rahmen (Frames) und Steuerung des Medienzugriffs sowie Fehlererkennung und -korrektur
- Beispiele: IEEE 802.3 (Ethernet) , IEEE 802.11 (WLAN), Digital Subscriber Line (DSL), LTE Air Interface

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

OSI-REFERENZMODELL VERSUS TCP/IP-MODELL (II)



INTERNETWORK LAYER (VERMITTLUNGSSCHICHT)

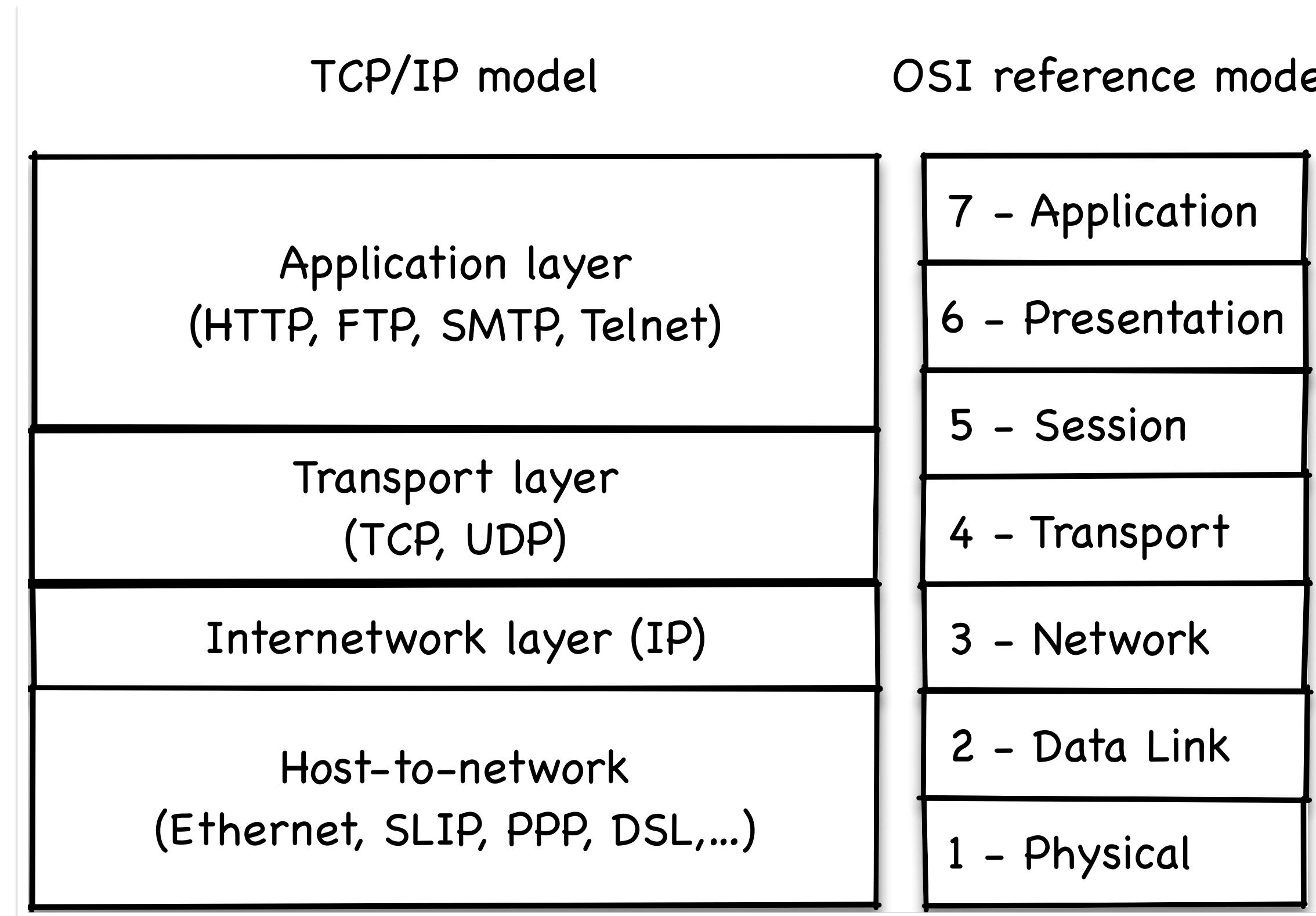
- Kommunikation zwischen Rechnern die an verschiedenen lokalen Netzen angeschlossen sind
- Wegewahl (Routing) und Adressierung
- Paket- statt Leitungsvermittlung: Daten werden in mehreren Paketen übertragen, die über unterschiedliche Wege zum Empfänger gelangen

TRANSPORT LAYER (TRANSPORTSCHICHT)

- Erweiterung der Basisfunktionen der Vermittlungsschicht
- Erstes Schicht die ausschließlich Ende-zu-Ende arbeitet, d.h. es gib in der Regel keine Protokollinstanzen im Netz
- Transmission Control Protocol (TCP): zuverlässiges, verbindungsorientiertes Protokoll zur fehlerfreien Ende-zu-Ende-Übertragung eines Datenstroms *bekannter Fehler der IP-Adresse ausgleichen*
- User Datagram Protocol (UDP): unzuverlässiges, verbindungsloses Protokoll zur schnellen, aber unzuverlässigen Ende-zu-Ende-Kommunikation

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

OSI-REFERENZMODELL VERSUS TCP/IP-MODELL (III)



APPLICATION LAYER (ANWENDUNGSSCHICHT)

- Protokollschicht für bestimmte Anwendungen
- Ermöglicht Aufbau, Aufrechterhalten und Abbau von Sitzungen
- Verhandlung über Syntax und Semantik der ausgetauschten Daten
- *Sitzung*: zeitlich begrenztes Verhältnis mit definierterem Zustand zwischen verteilten Anwendungen, inklusive Wiedererkennung des Kommunikationspartners und Synchronisation
- Beispiele: HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Electronic Mail (SMTP), Domain Name System (DNS)

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

DATENKAPSELUNG



DATENKAPSELUNG

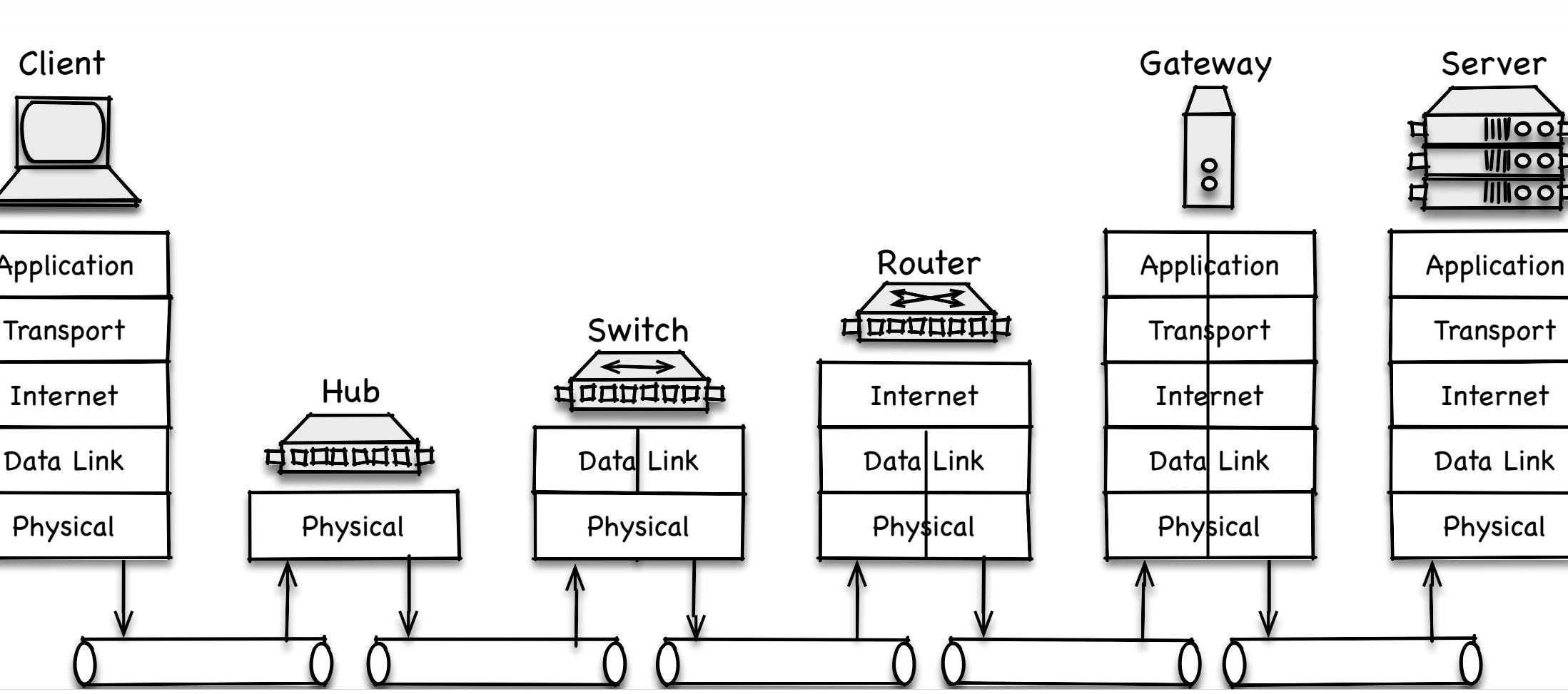
- Protokolle auf jeder Schicht übertragen Daten in Blöcken, sogenannten Protocol Data Units (PDUs)
- Schichtung manifestiert sich in der Kapselung von PDUs: PDUs der Schicht n werden in PDUs der Schicht $n-1$ eingepackt
- PDUs auf jeder Schicht bestehen aus einem Header, der Kontroll- und Steuerinformationen enthält, und einem Body, der die Nutzdaten enthält

BEISPIELE

- HTTP-Nachrichten werden in TCP-Segmente gekapselt
- TCP-Segmente werden in IP-Pakete gekapselt
- IP-Pakete werden in Rahmen (Frames) des Host-to-Network-Layer gekapselt (nicht in der Abbildung)
- Nachrichten, Segmente, Pakete und Rahmen sind spezifische, schichtenabhängige Bezeichnungen für PDUs

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

NETZGERÄTE (I)



HUB

- Verfügt über mehrere Anschlüsse und verbindet mehrere Computer mit einem lokalen Netz
- Prinzip Mehrfachsteckdose: Signale die an einem Anschluss ankommen, werden über alle anderen Anschlüsse weitergeleitet, unabhängig davon, wo sich der Empfänger befindet
- Hubs arbeiten auf OSI-Schicht 1
- Alternative Bezeichnungen: Repeater oder Repeating-Hub

SWITCH

- Verfügt über mehrere Anschlüsse ähnlich einem Hub und verbindet verschiedene Netzsegments miteinander
- Leitet Frames der OSI-Schicht 2 zielgerichtet anhand von Informationen (MAC-Adresse) im Header der Frames auf einem bestimmten Anschluss weiter

6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

NETZGERÄTE (II)

ROUTER

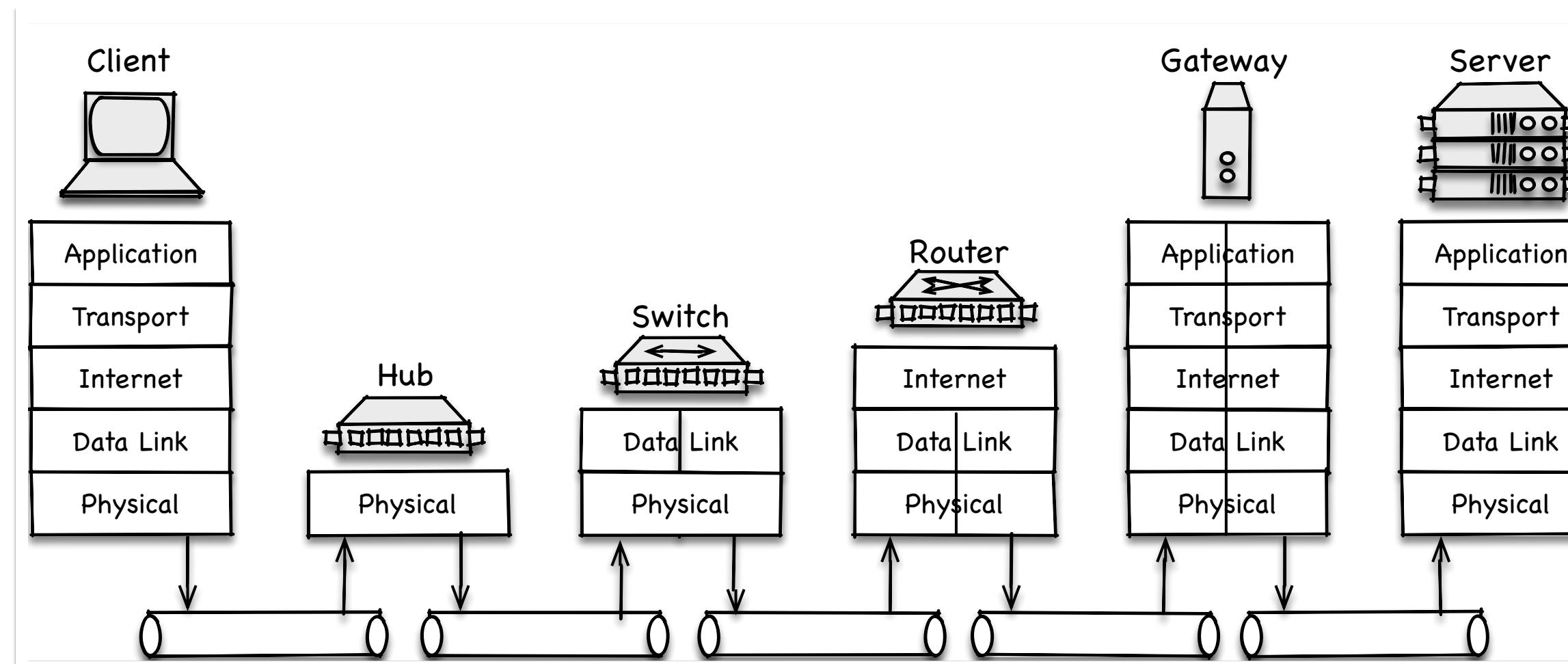
- Verbindet lokale Netze auf Schicht 3
- Wegewahl prüft IP-Adresse im Header und wählt einen Ausgang basierend auf IP-Adresse und Routing-Tabelle

GATEWAY

- Implementieren den gesamten Protokollstapel, um verschiedene Teilnetze mit unterschiedlichen Transport- oder Anwendungsprotokollen zu verbinden
- Übernehmen Managementaufgaben wie Load Balancing, Firewall, etc.

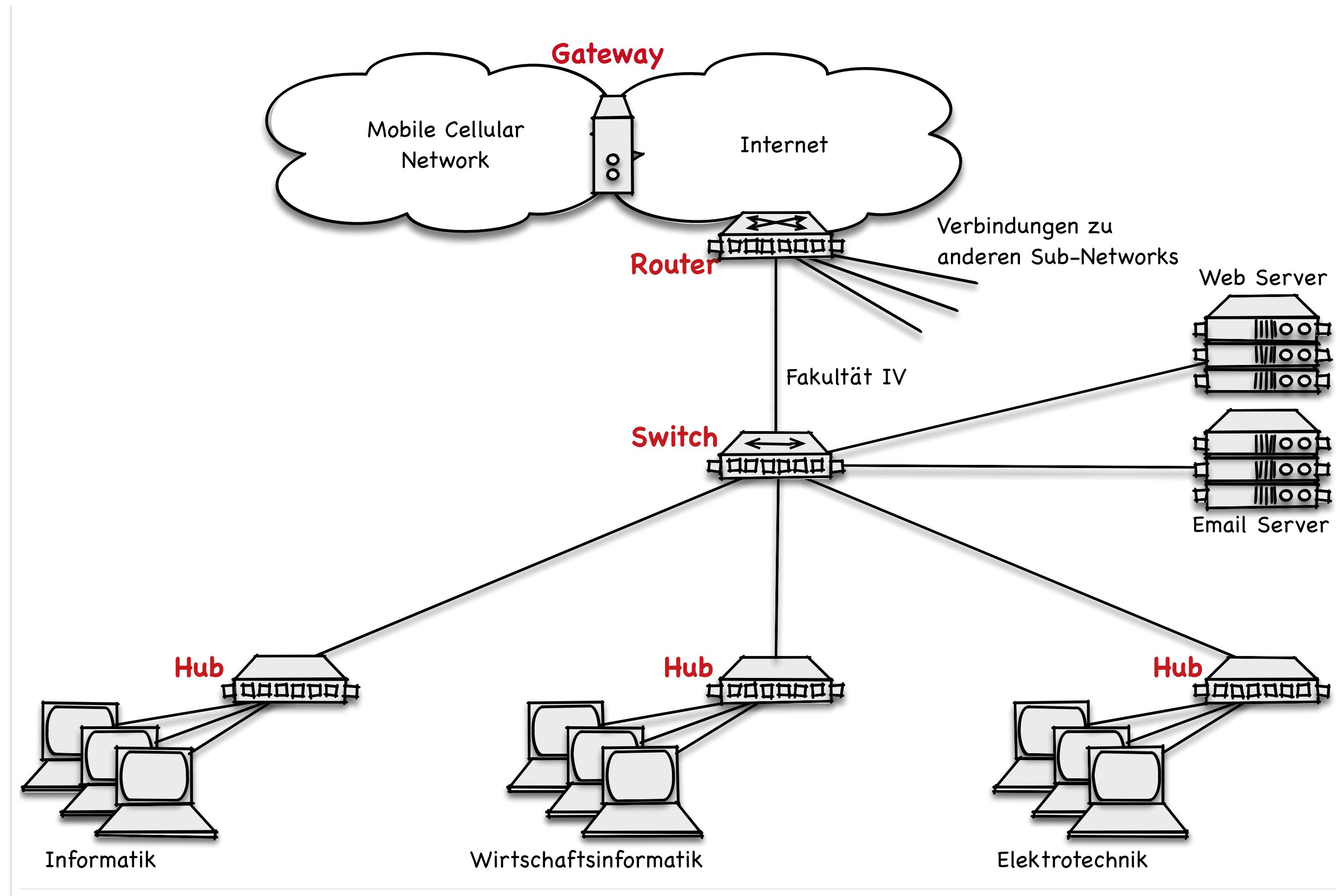
CLIENT, SERVER (ENDSYSTEME)

- Implementieren den gesamten Protokollstapel
- Protokoll der Anwendungsschicht bildet Schnittstelle zwischen Anwendung (z.B. Webserver oder -browser) und den übrigen Protokollen



6.1 GRUNDLAGEN DER DATENKOMMUNIKATION

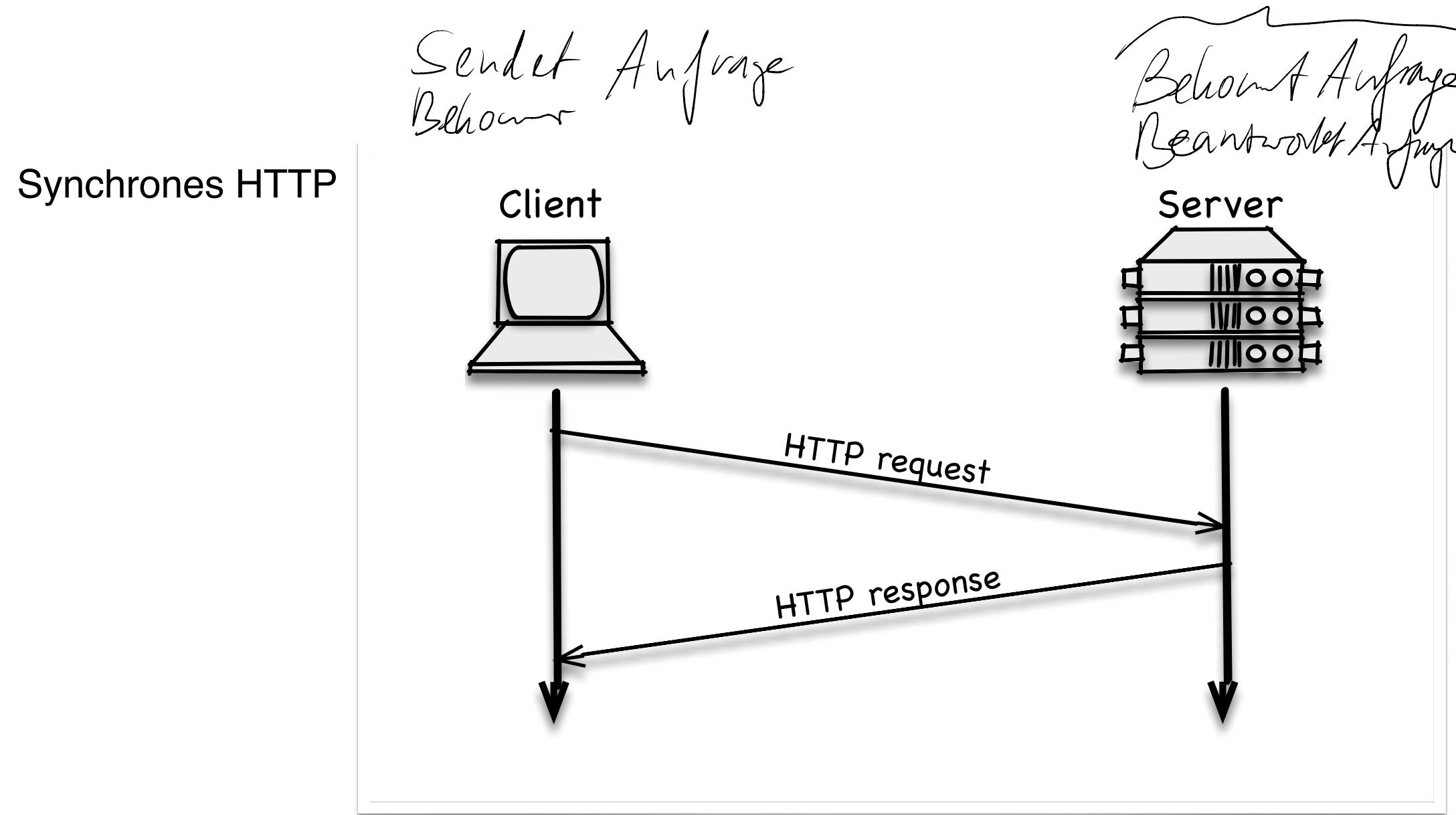
NETZGERÄTE (III)



6.2 HTTP OVER TCP

HTTP-EIGENSCHAFTEN UND METHODEN

Stellt Ressourcen zur Verfügung



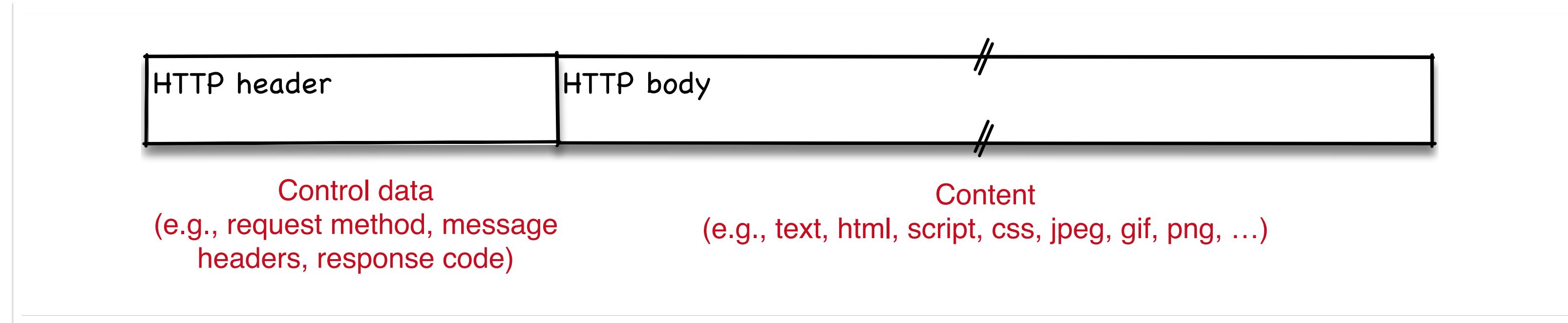
| Methoden in HTTP | Request methods | Description |
|------------------|-----------------|-----------------------------------------------|
| | GET | Request to read a web page |
| | HEAD | Request to read a web page's header |
| | PUT | Request to store a web page |
| | POST | Append to a named resource (e.g., a web page) |
| | DELETE | Remove the web page |
| | TRACE | Echo the incoming request |
| | CONNECT | Reserved for future use |
| | OPTIONS | Query certain options |

- Synchrones Protokoll der Anwendungsschicht für die Client/Server-Kommunikation
- HTTP Request wird immer durch Client, HTTP Response wird immer durch Server gesendet
- Synchrone Ablauf: strikte Einhaltung des Request/Response-Zyklus, d.h. Client sendet Request und wartet (d.h. ist blockiert bis) bis Antwort eintrifft
- HTTP 1.0: erste offizieller Standard des Protokolls, veröffentlicht 1996
- HTTP 1.1: gegenwärtige Version, optimiert um Keepalive und Caching Mechanismen, veröffentlicht im Januar 1997
- HTTP/2: Verbesserungen bzgl. Ladegeschwindigkeit (SPDY) und Sicherheit, Verabschiedung des Standards in Vorbereitung
- SPDY: "Zwischenstandard" von Google mit Mechanismen zur Komprimierung, Multiplexing, Priorisierung

6.2 HTTP OVER TCP

HTTP-STRUKTUR UND STATUS CODES

STRUKTUR EINER HTTP-NACHRICHT



GRUPPEN DER HTTP STATUS CODES

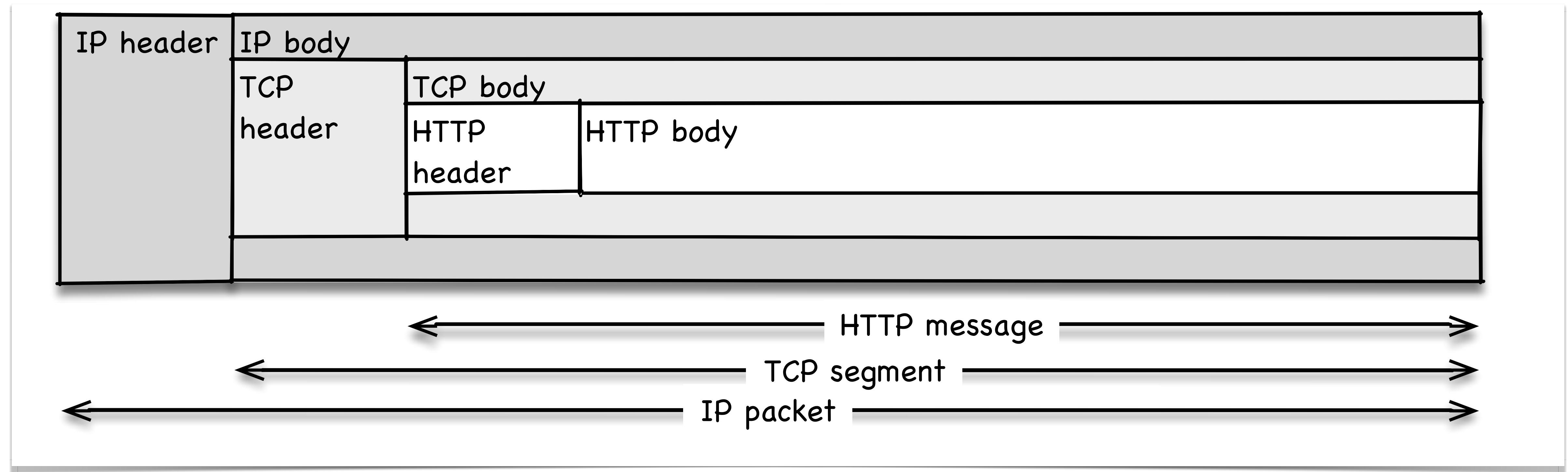
| Code | Meaning | Examples |
|------|--------------|------------------------------------------------|
| 1xx | Information | 100=server agrees to handle client's request |
| 2xx | Success | 200=request succeeded; 204=no content present |
| 3xx | Redirection | 301=page moved; 304=cached page still valid |
| 4xx | Client error | 403=forbidden page; 404=page not found |
| 5xx | Server error | 500=internal server error; 503=try again later |

6.2 HTTP OVER TCP

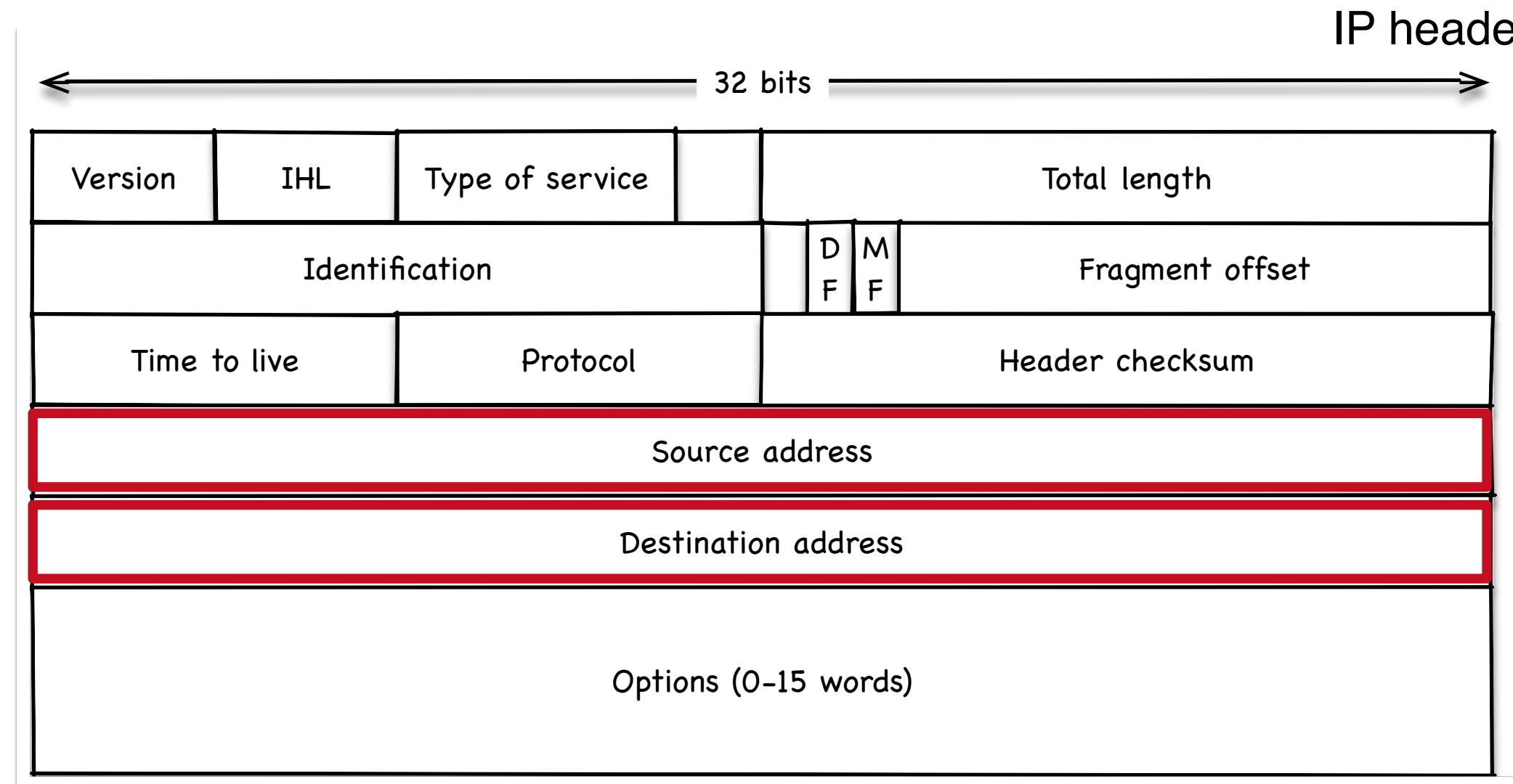
HTTP HEADER

| Header | Type | Description |
|------------------|----------|-----------------------------------------------------------------------|
| User-Agent | Request | Information about the browser and its platform |
| Accept | Request | Type of pages the client can handle |
| Accept-Charset | Request | Character sets that are acceptable to the client |
| Accept-Encoding | Request | Page encoding the client can handle |
| Accept-Language | Request | Natural languages the client can handle |
| Host | Request | The server's DNS name |
| Authorization | Request | List of the client's credentials |
| Connection | Request | What kind of connection the user-agent would prefer (e.g., keepalive) |
| Cookie | Request | Sends a previously set cookie back to the server |
| Date | Both | Date and time the message was sent |
| Upgrade | Both | Protocol the server wants to switch to |
| Server | Response | Information about the server |
| Content-Encoding | Response | How the content is encoded (e.g., gzip) |
| Content-Language | Response | Natural language used in the page |
| Content-Length | Response | Page length in bytes |
| Content-Type | Response | Page's MIME type |
| Last-modified | Response | Time and date the page was last changed |
| Location | Response | A command to the client to send its request elsewhere |
| Accept-Ranges | Response | Indicated that server will accept byte range requests |
| Set-Cookie | Response | Server wants the client to save a cookie |

6.2 HTTP OVER TCP DATENKAPSELUNG



6.2 HTTP OVER TCP IP-ADRESSEN



Netzmaske: Berechnung von Präfix und Postfix

| | Dezimal | Binär | Berechnung | |
|--------------|-----------------|-------------------------------------|------------------------------|--|
| IP-Adresse | 203.000.113.195 | 11001011 00000000 01110001 11000011 | <i>ip-adresse</i> | |
| Netzmaske | 255.255.255.224 | 11111111 11111111 11111111 11100000 | AND <i>netzmaske</i> | |
| Netzwerkadr. | 203.000.113.192 | 11001011 00000000 01110001 11000000 | = <i>netzwerkteil</i> | |
| | | | | |
| IP-Adresse | 203.000.113.195 | 11001011 00000000 01110001 11000011 | <i>ip-adresse</i> | |
| Netzmaske | 255.255.255.224 | 11111111 11111111 11111111 11100000 | AND (<i>NOT netzmaske</i>) | |
| Geräteteil | | 00000000 00000000 00000000 00011111 | = <i>geräteteil</i> | |

- Adresse in Computernetzen die auf dem Internetprotokoll basieren
- Ermöglicht die Adressierbarkeit von Geräten und die Zustellung von Paketen
- Jedes Paket enthält die IP-Adressen des Empfängers und Absenders
- Bei der Weiterleitung von Paketen entscheiden Router auf welchem Ausgang ein Paket weitergeleitet werden muss, um den Empfänger zu erreichen
- Unterteilt sich in einen Präfix für die Adresse des Netzes und einen Postfix für den Rechner innerhalb des Netzes
- Länge des Präfix wird durch die Netzmaske angegeben
- IP-Version 4: 32-Bit-Adressen, d.h. 4.294.967.296 verschiedene Adressen
- IP-Version 6: 128-Bit-Adressen, d.h. 340.282.366.920.938.463.463.374.607.431.768.456 Adressen (oder $6,65 \cdot 10^{17}$ Adressen pro Quadratmillimeter der Erdoberfläche)

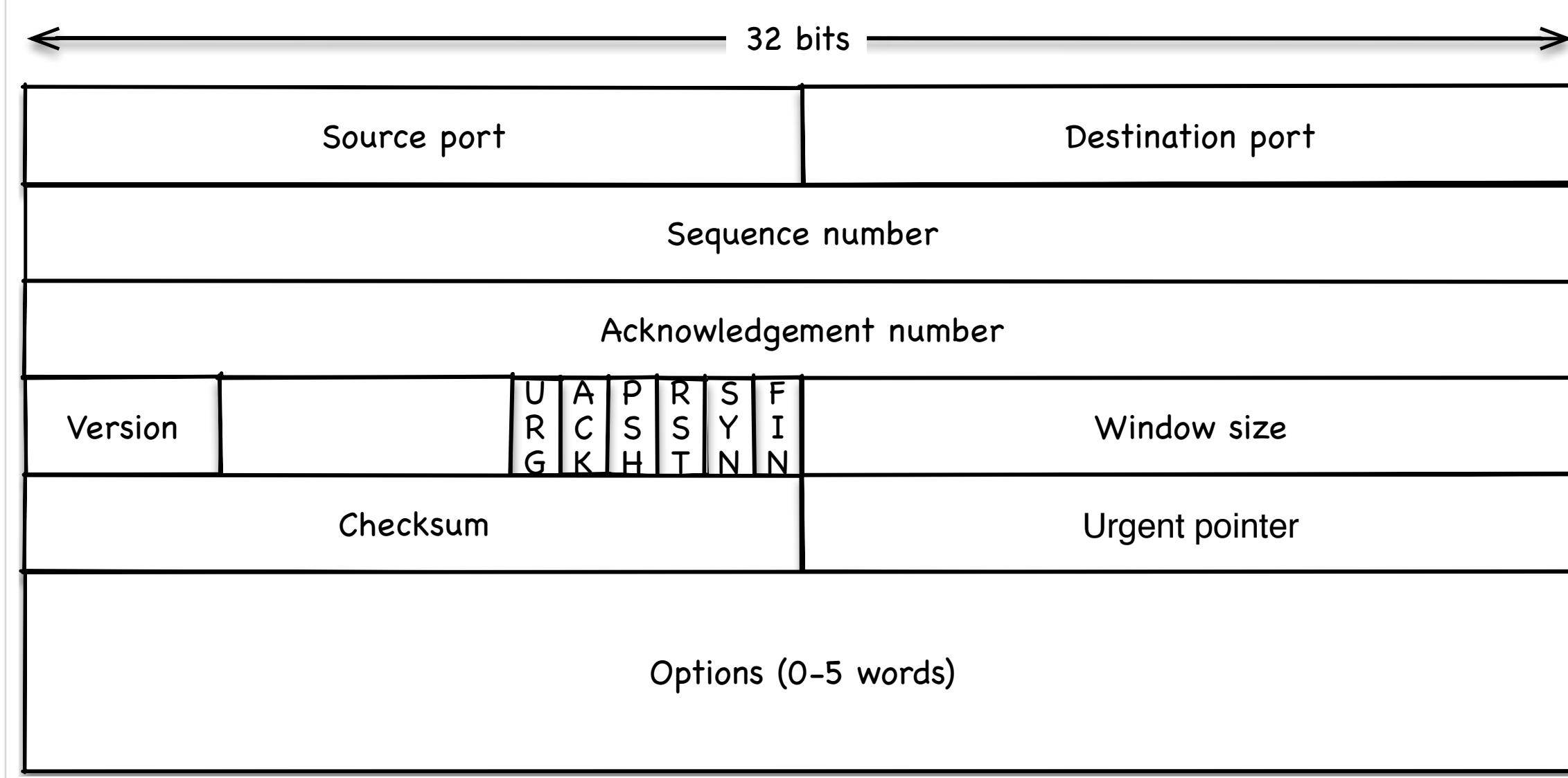
6.2 HTTP OVER TCP

TCP-ÜBERBLICK

Wichtige Systemports

- 21: File Transfer Protocol (FTP)
- 22: Secure Shell (SSH)
- 23: Telnet remote login service
- 25: Simple Mail Transfer Protocol (SMTP)
- 53: Domain Name System (DNS) service
- 80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web
- 110: Post Office Protocol (POP3)
- 119: Network News Transfer Protocol (NNTP)
- 143: Internet Message Access Protocol (IMAP)
- 161: Simple Network Management Protocol (SNMP)
- 194: Internet Relay Chat (IRC)
- 443: HTTP Secure (HTTPS)
- 465: SMTP Secure (SMTPS)
- 8443: Router remote access

TCP Header



- Ende-zu-Ende-Transportprotokoll im Internet
- Zuverlässig, verbindungsorientiert, paketvermittelt
- Bidirektional, d.h. Übertragung in beide Richtungen
- Flusskontrolle vermeidet Überlastung des Empfängers
- Staukontrolle drosselt die Übertragung bei (angeblicher) Überlastung des Netzes
- Fehlerkontrolle ermöglicht Wiederübertragung bei fehlerhaften oder verlorenen Segmenten
- Sender: Unterteilt den eingehenden Datenstrom auf mehrere Segmente und übergibt sie der Internetschicht
- Empfänger: Setzt die eingehenden Segmente in der ursprünglichen Reihenfolge zu einem Ausgabestrom zusammen und übergibt ihn der Anwendungsschicht

6.2 HTTP OVER TCP PORTS

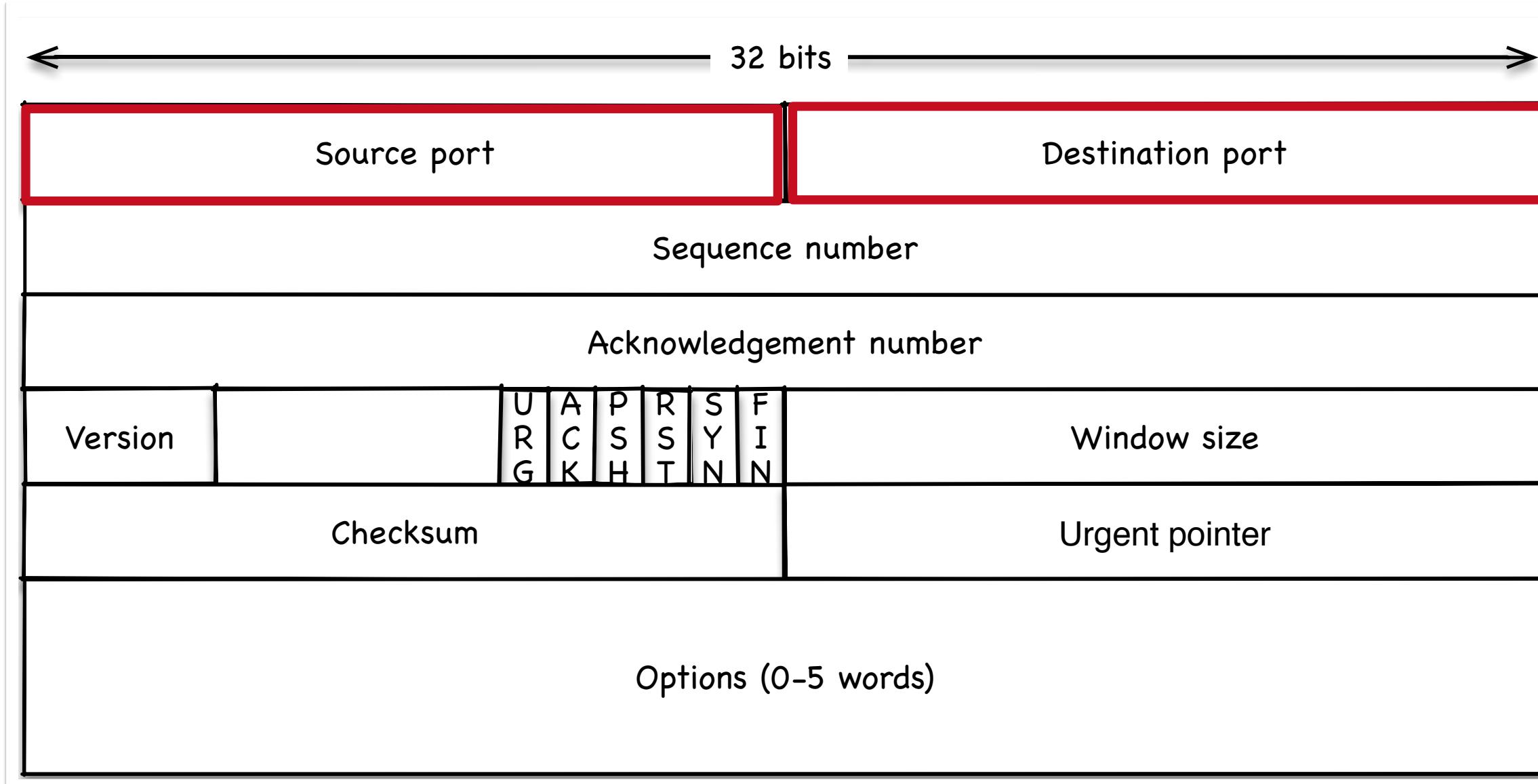
WICHTIGE SYSTEMPORTS

- 21: File Transfer Protocol (FTP)
- 22: Secure Shell (SSH)
- 23: Telnet remote login service
- 25: Simple Mail Transfer Protocol (SMTP)
- 53: Domain Name System (DNS) service
- 80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web
- 110: Post Office Protocol (POP3)
- 119: Network News Transfer Protocol (NNTP)
- 143: Internet Message Access Protocol (IMAP)
- 161: Simple Network Management Protocol (SNMP)
- 194: Internet Relay Chat (IRC)
- 443: HTTP Secure (HTTPS)
- 465: SMTP Secure (SMTPS)
- 8443: Router remote access

- Lokale Hausnummer innerhalb eines Rechners zur Adressierung einer bestimmten Anwendung
- Gültige Portnummern: 0-65535
- Systemports: 0-1023, reserviert für bestimmte Dienste, Standardisierung durch die IETF
- Registrierte Ports: 1024-49151, reserviert für registrierte Dienste, Zuweisung ohne IETF (veraltet)
- Dynamische Ports: 49152-65536, Vergabe durch das Betriebssystem

6.2 HTTP OVER TCP HEADER

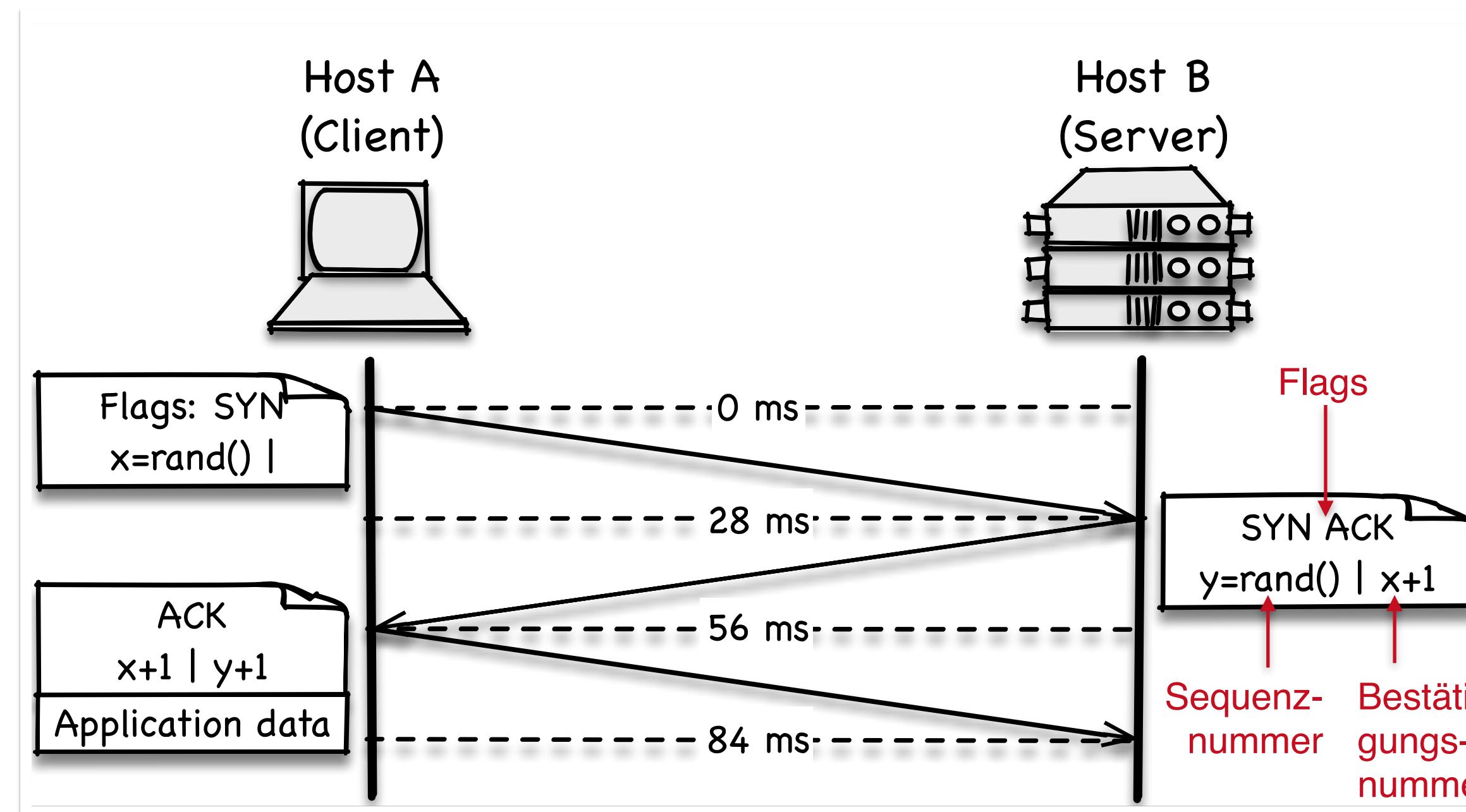
TCP HEADER



- TCP-Verbindung wird durch zwei Endpunkte identifiziert
- Endpunkt wird durch Internetadresse und Port repräsentiert
- Socket: Softwareschnittstelle des Endpunktes Sockets werden beim Verbindungsauftbau auf beiden Seiten initialisiert
- Jede TCP-Verbindung wird repräsentiert durch (IP-Adresse lokal, Port lokal, IP-Adresse entfernt, Port entfernt)

6.2 HTTP OVER TCP

TCP HANDSHAKE (I)



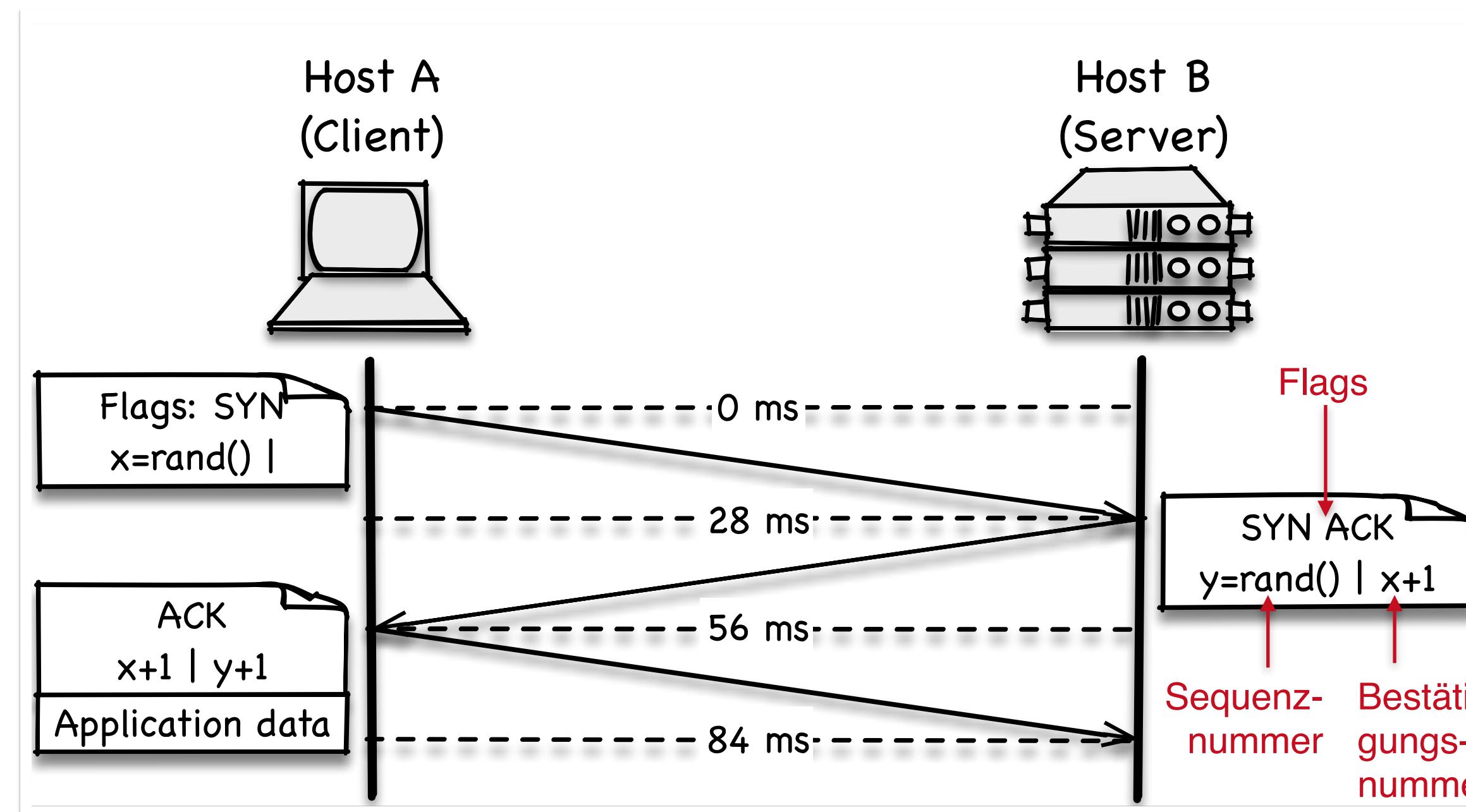
- Alle TCP-Verbindungen beginnen mit einem TCP-Handshake zum Verbindungsaufbau
- Nutzdaten können erst nach erfolgreichem Verbindungsaufbau, d.h. nach Handshake, gesendet werden

FLAGS

- SYN: kennzeichnet einen Verbindungsaufbau
- ACK: Segment bestätigt dem Empfänger, dass seiner vorherige Übertragung in Rückrichtung erfolgreich war
- FIN: Sender wünscht einen Verbindungsabbau

6.2 HTTP OVER TCP

TCP HANDSHAKE (II)



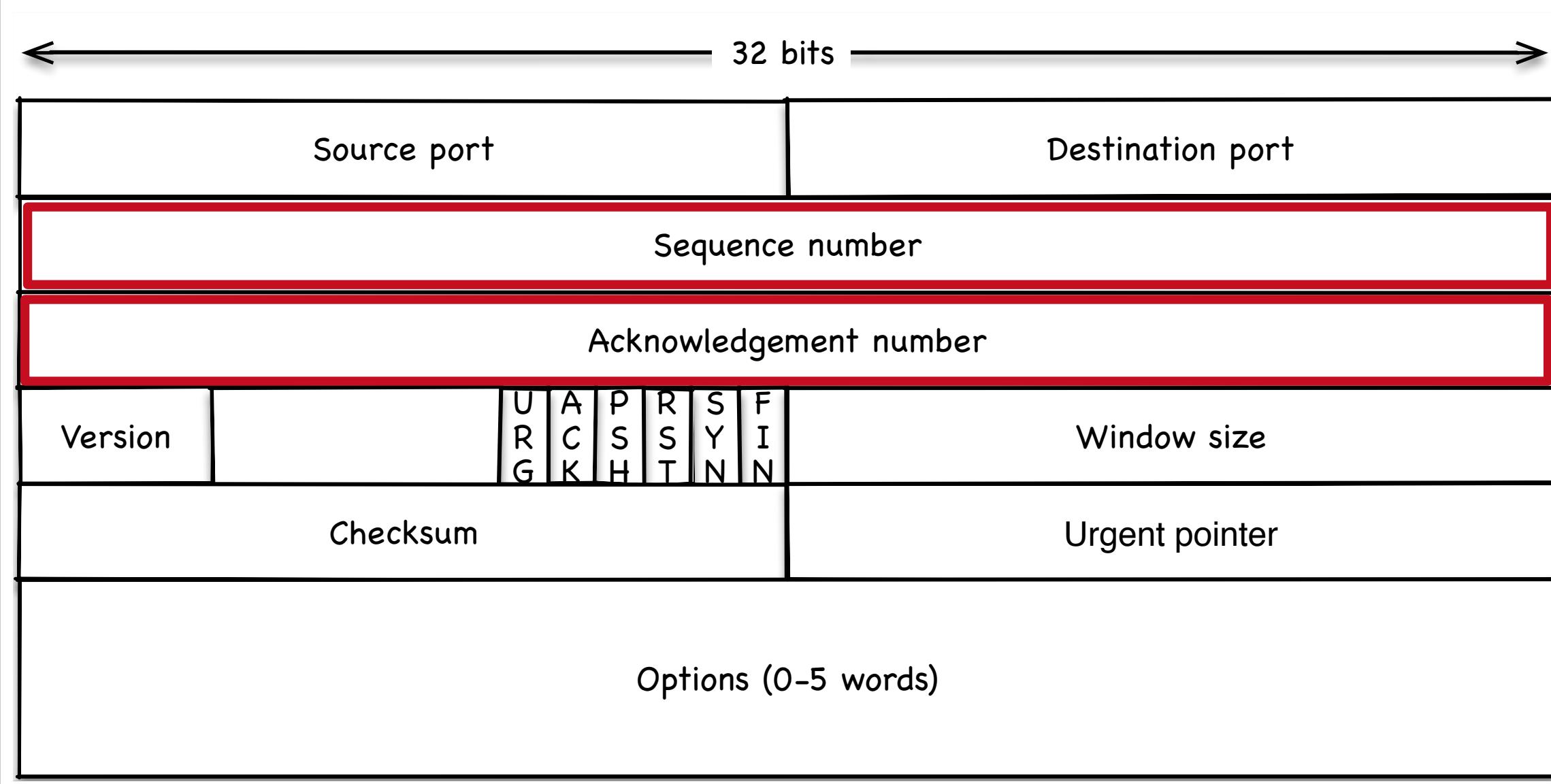
SEQUENZNUMMER (SEQUENCE NUMBER)

- Bezeichnet die Byte-Position des Segments im Datenstrom
- Wertebereich: 2^{32}
- Wird um die Anzahl der gesendeten Bytes erhöht
- Ausnahme: Segmente mit gesetztem SYN und FIN-Flag erhöhen die Sequenznummer um 1 (diese Segmente enthalten aber keine Daten)
- Beim Verbindungsauftbau wird die Startsequenznummer gewürfelt

6.2 HTTP OVER TCP

TCP HANDSHAKE (II)

TCP HEADER



BESTÄTIGUNGNUMMER (ACKNOWLEDGEMENT NUMBER)

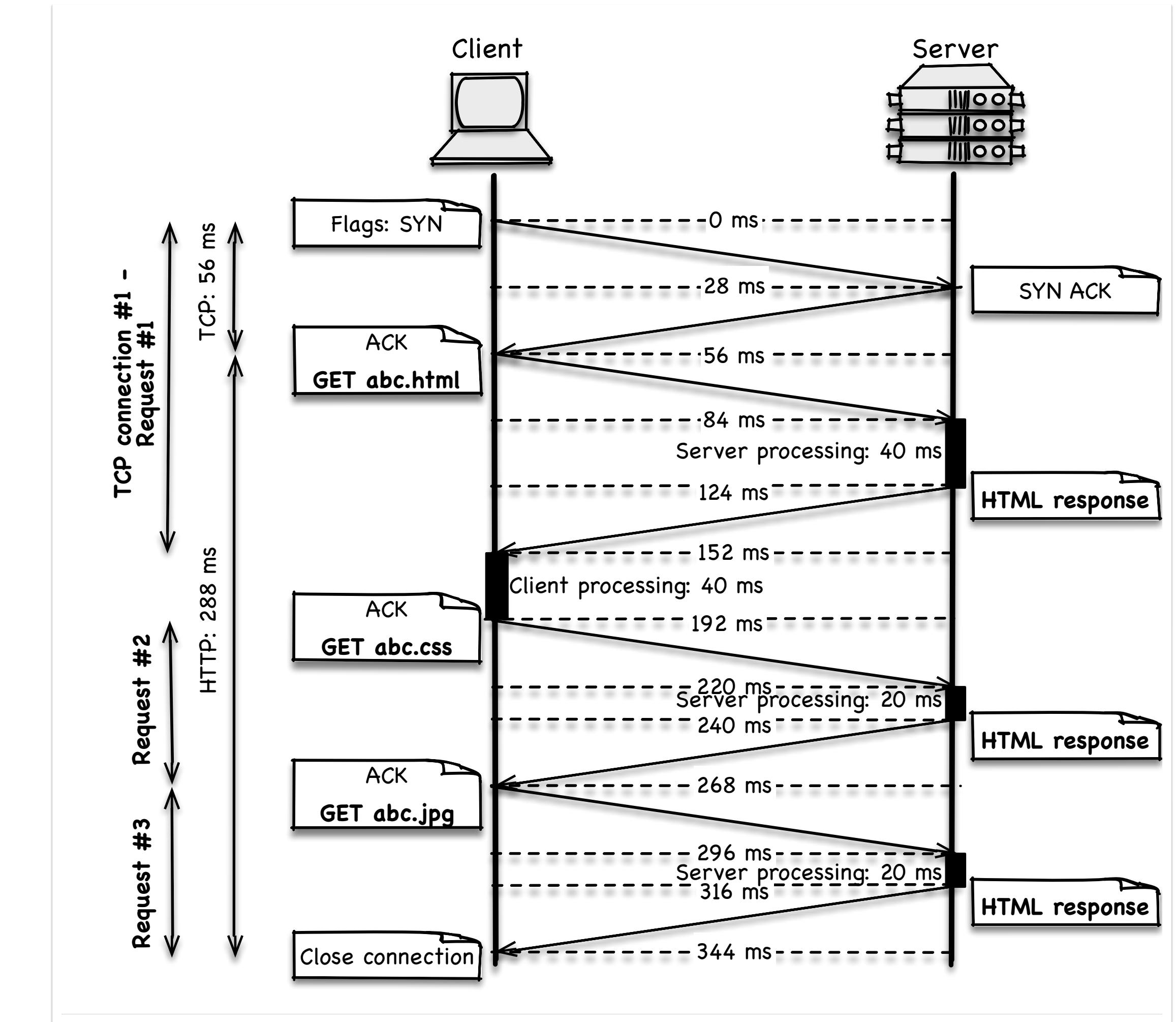
- Bestätigen dem ursprünglichen Absender den erfolgreichen Empfang seiner Daten
- Bestätigt wird das zuletzt erfolgreich empfangene Byte
- Aber: Bestätigungsnummer gibt das als nächste erwartete Byte an

6.2 HTTP OVER TCP

CLIENT/SERVER-KOMMUNIKATION MIT HTTP (II)

KEEPALIVE

- Alle HTTP-Anfragen gehen über dieselbe TCP-Verbindung, d.h. TCP-Verbindung wird gehalten bis alle Inhalte geladen sind
- HTTP 1.0: Keepalive wird durch den Connection-Parameter zwischen Client und Server ausgehandelt
- HTTP 1.1: TCP-Verbindung werden bis zu einem Timeout (abhängig von Webserver und -Browser) offen gehalten

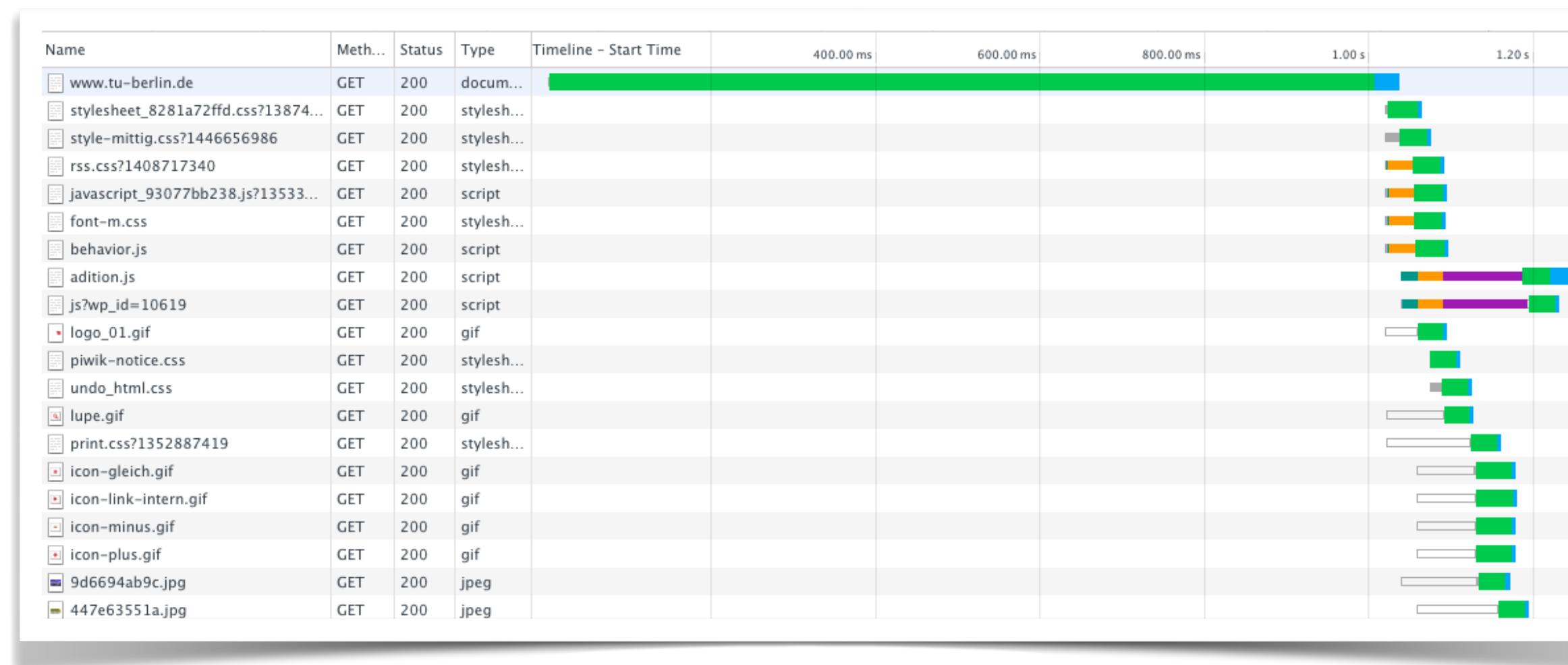


6.2 HTTP OVER TCP

MULTIPLE TCP CONNECTION UND DOMAIN SHARDING (I)

VERWENDUNG MEHRERER TCP-VERBINDUNGEN

- Aufgrund unzureichender Unterstützung von Multiplexing und Pipelining in HTTP 1.x öffnen Browser mehrere TCP-Verbindungen zu einem Webserver gleichzeitig
- Client und Server unterhalten i.d.R. bis zu sechs TCP-Verbindungen gleichzeitig
- Maximale Anzahl von sechs TCP-Verbindung ist ein Kompromiss zwischen Optimierung der Ladezeit und erhöhtem CPU- und Speicheraufwand für Einrichtung und Unterhaltung von Sockets auf beiden Seiten



DOMAIN SHARDING

- Sechs parallele TCP-Verbindungen sind oft nicht schnell genug
- Sharding: anstelle einer Domain (www.example.de) werden Anfragen auf mehrere Domains verteilt (shard1.example.de, shard2.example.de, shard3.example.de)
- Unterhaltung von sechs TCP-Verbindungen zu jeder Shard Domain

6.2 HTTP OVER TCP

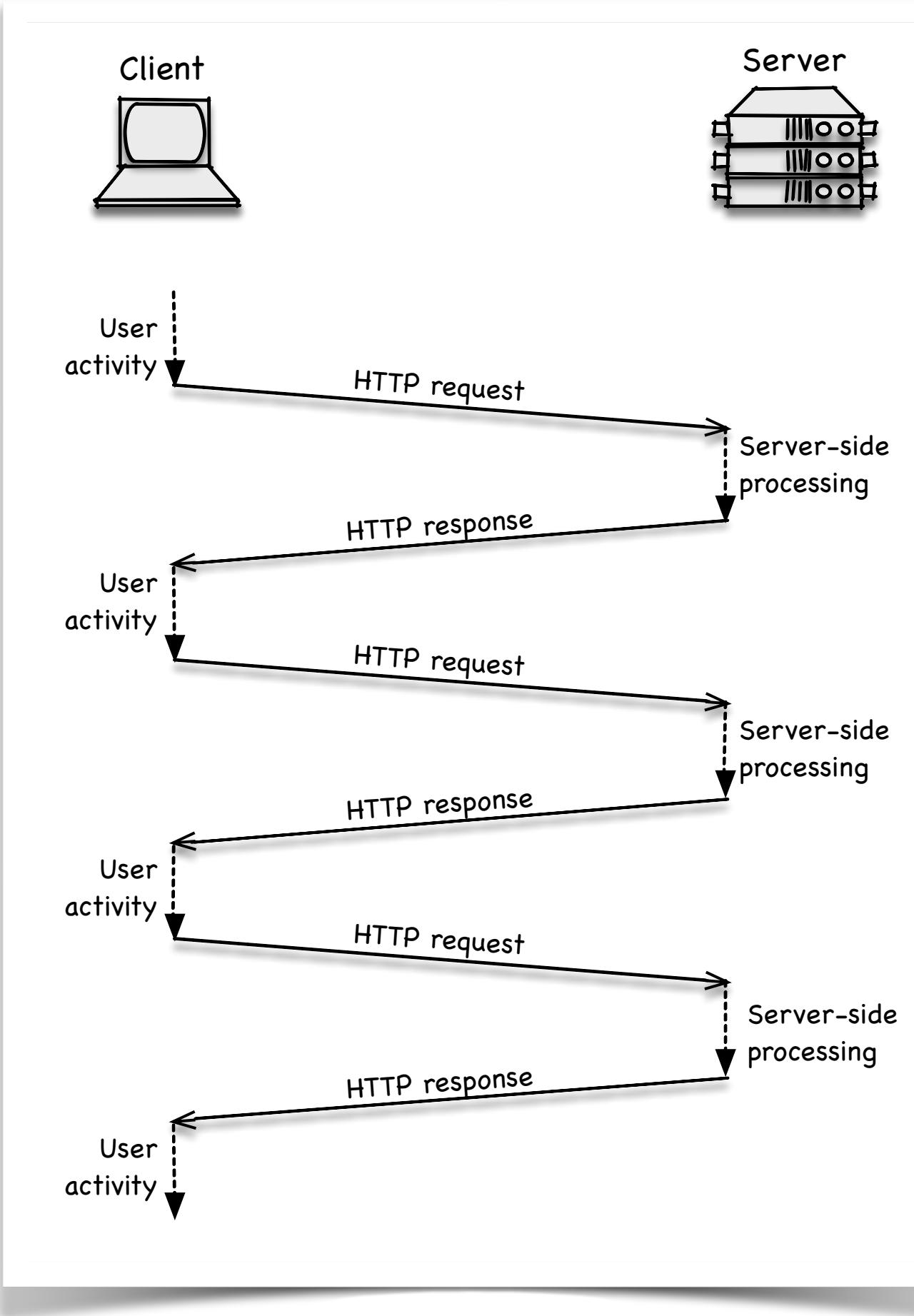
MULTIPLE TCP CONNECTION UND DOMAIN SHARDING (II)



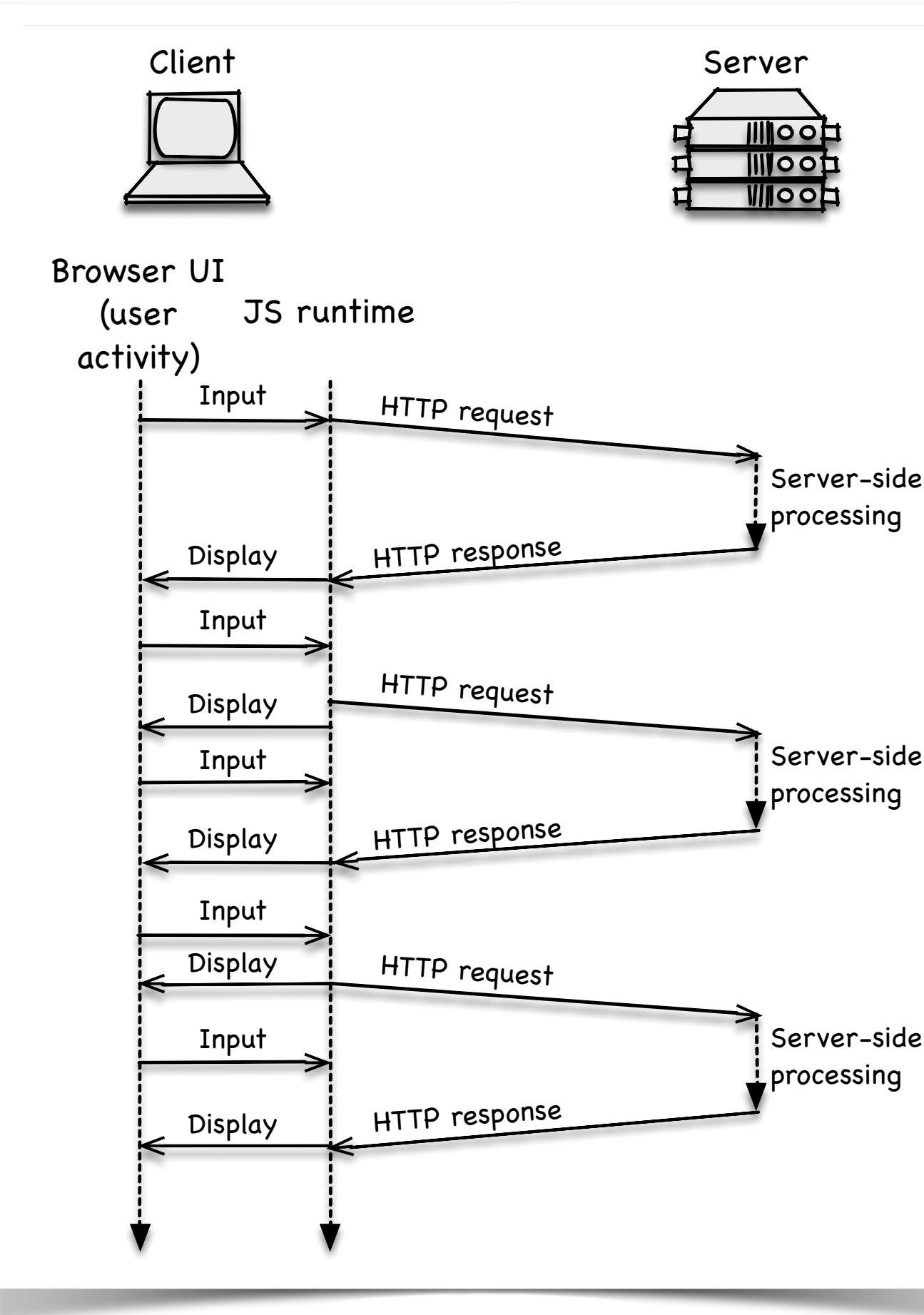
6.3 ASYNCHRONE KOMMUNIKATION

KLASSISCHES MODELL DER WEBKOMMUNIKATION VERSUS AJAX

SYNCHRONE KOMMUNIKATION



ASYNCHRONE KOMMUNIKATION



ASYNCHRONOUS JAVASCRIPT AND XML

- Konzept für die asynchrone Datenübertragung zwischen Browser und Server
 - Durchführung von HTTP-Anfragen während der Anzeige einer HTML-Seite und Veränderung dieser Seite im Hintergrund, d.h. ohne Nutzerinteraktion
 - Begriff bezeichnet nicht eine einzelne, sondern eine Gruppe von Technologien
 - HTML
 - DOM
 - JavaScript
 - XMLHttpRequest-Object: API, um Daten auf asynchroner Basis mit dem Webbrowser austauschen zu können
 - XML oder JSON als Datenaustauschformate

6.3 ASYNCHRONE KOMMUNIKATION

METHODEN UND EIGENSCHAFTEN VON XMLHTTPREQUEST

| Wichtige Eigenschaften | Beschreibung |
|------------------------|--------------------------------------------------------------------------------------------|
| timeout | Anzahl Millisekunden, die eine Anfrage dauern darf, bevor sie automatisch abgebrochen wird |
| status und statusText | Status-Code und -Text der HTTP-Antwort |
| readyState | ready-der HTTP-Anfrage |
| response | Antwort der Anfrage als ArrayBuffer, string, Blob, Document oder JavaScript Objekt (json) |
| responseText | Antwort der Anfrage als Text |
| responseType | Datentyp der Antwort |

| Wichtige Ereignisse | Beschreibung |
|---------------------|--------------------------------------------------|
| readystatechange | Wenn sich der Status geändert hat |
| loadstart | Anfrage wurde gestartet |
| progress | Daten werden gesendet oder empfangen |
| abort | Anfrage wurde abgebrochen |
| error | Anfrage ist auf Netzwerkebene gescheitert |
| load | Anfrage wurde erfolgreich abgeschlossen |
| loadend | Anfrage wurde erfolgreich oder erfolglos beendet |

| Wichtige Methoden | Beschreibung |
|-------------------------|---------------------------------------------------------|
| abort() | Bricht die Anfrage ab, falls sie bereits gesendet wurde |
| getAllResponseHeaders() | Liefert alle Header der HTTP-Antwort |
| getResponseHeader() | Liefert den angegebenen Header der HTTP-Antwort |
| open() | Initialisiert eine Anfrage |
| send() | Sendet eine Anfrage |
| setRequestHeader() | Setzt den Wert eines Headers der HTTP-Anfrage |

XMLHttpRequest

- JavaScript-Objekt zu asynchronen Kommunikation zwischen Browser und Server
- Entwickelt von Microsoft, später von Google, Apple und Mozilla übernommen
- Standardisierung durch W3C
- Eignet sich für den Austausch jeder Art von Daten (trotz seines Namens)



<https://developer.mozilla.org/de/docs/Web/API/XMLHttpRequest>

6.3 ASYNCHRONE KOMMUNIKATION

ABLAUF EINER VERBINDUNG

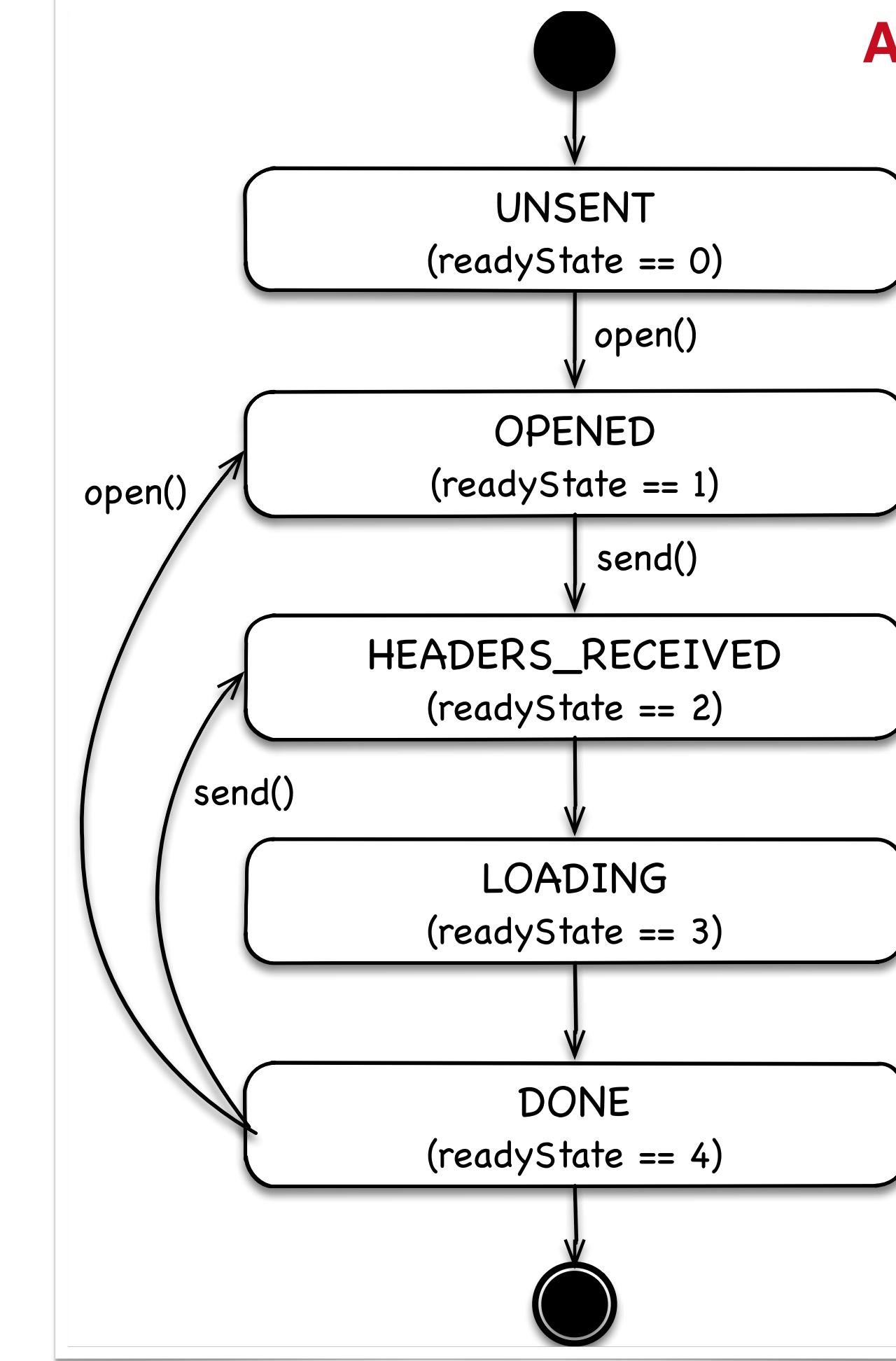
SYNCHRONE ANFRAGE

```
> var xhr=new XMLHttpRequest();
  xhr.open("GET", "https://dummyjson.com/products", false);
  xhr.send();
  console.log(JSON.parse(xhr.response));
  ▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

ASYNCHRONE ANFRAGE

```
> var xhr=new XMLHttpRequest();
  xhr.open("GET", "https://dummyjson.com/products", true);
  xhr.onreadystatechange=function(){
    if (xhr.readyState==4 && xhr.status==200) {
      console.log(JSON.parse(xhr.response));
    };
  };
  xhr.send();
< undefined
  ▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

ABLAUF EINER VERBINDUNG



- Die open()-Methode wurde erfolgreich ausgeführt
- Mit setRequestHeader() können Request-Header gesetzt werden
- send() kann nun aufgerufen werden
- HTTP-Header der Antwort wurden empfangen
- Nutzdaten der Antwort werden geladen
- Wenn kein Fehler aufgetreten ist, ist der Datentransfer abgeschlossen



<https://developer.mozilla.org/de/docs/Web/API/XMLHttpRequest>

6.3 ASYNCHRONE KOMMUNIKATION

DIE NEUEN EREIGNISSE load UND error

readystatechange

```
> var xhr=new XMLHttpRequest();
  xhr.open("GET", "https://dummyjson.com/products", true);
  xhr.onreadystatechange=function(){
    if (xhr.readyState==4 && xhr.status==200) {
      console.log(JSON.parse(xhr.response));
    };
  };
  xhr.send();
< undefined
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

load UND error

```
> var xhr=new XMLHttpRequest();
  xhr.open("GET", "https://dummyjson.com/products", true);
  xhr.onload=function() {
    console.log(JSON.parse(xhr.response));
  };
  xhr.onerror=function() {
    console.error("Netzwerkfehler");
  };
  xhr.send();
< undefined
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

- Anstelle von `readystatechange` können Event Handler auch an die Ereignisse `load` und `error` gebunden werden
- `load` wird ausgelöst wenn eine HTTP Antwort vollständig geladen wird
- `error` wird bei einem Fehler auf Netzwerkebene ausgelöst, d.h. wenn keine Anfrage gesendet werden konnte oder keine Anfrage kommt

6.3 ASYNCHRONE KOMMUNIKATION

AUTOCOMPLETE

Übernommen von: https://www.w3schools.com/xml/tryit.asp?filename=try_dom_xmlhttprequest_suggest

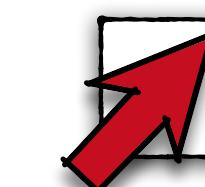
```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <script>
5          function showHint(str) {
6              if (str.length==0) {
7                  document.getElementById("txtHint").innerHTML="";
8                  return;
9              } else {
10                  var xmlhttp=new XMLHttpRequest();
11                  xmlhttp.onreadystatechange=function() {
12                      if (xmlhttp.readyState==4 && xmlhttp.status==200) {
13                          document.getElementById("txtHint").innerHTML=xmlhttp.responseText;
14                      }
15                  }
16                  xmlhttp.open("GET","gethint.php?q="+str,true);
17                  xmlhttp.send();
18              }
19      }
20  </script>
21 </head>
22 <body>
23     <p><b>Start typing a name in the input field below:</b></p>
24     <form action="">
25         First name: <input type="text" id="txt1" onkeyup="showHint(this.value)">
26     </form>
27     <p>Suggestions: <span id="txtHint"></span></p>
28 </body>
29 </html>
```

Start typing a name in the input field below:

First name:

Suggestions: Anna, Amanda

- Wenn der Nutzer mit der Eingabe von Text beginnt, werden automatisch Vorschläge für Namen vom Server geladen, deren Präfix mit der bisherigen Eingabe übereinstimmt



<https://developer.mozilla.org/de/docs/Web/API/XMLHttpRequest>

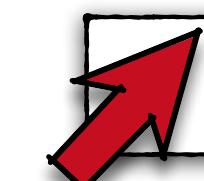
6.3 ASYNCHRONE KOMMUNIKATION

DIE fetch API (I)

```
> var xhr=new XMLHttpRequest();
  xhr.open("GET", "https://dummyjson.com/products", true);
  xhr.onload=function() {
    console.log(JSON.parse(xhr.response));
  };
  xhr.onerror=function() {
    console.error("Netzwerkfehler");
  };
  xhr.send();
< undefined
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

```
> fetch("https://dummyjson.com/products")
  .then(response=>{
    if (response.status==200)
      console.log(response.json());
  })
  .catch(()=>console.log("Netzwerkfehler"));
< ▶ Promise {<pending>}
  ▶ Promise {<pending>}
```

- fetch API als moderne Alternative zu XMLHttpRequest
- Basiert auf Promises
- fetch-Methode wird aufgerufen mit
 - einer URL oder
 - einer URL und einem Objekt, welches HTTP-Methode, headers, body der Anfrage und weitere Parameter spezifiziert oder
 - einem Request-Objekt
- HTTP-Antwort wird dann von der an then übergebenen Funktion abgefangen und ausgewertet
- Netzwerkfehler werden den an catch übergebenen Funktion behandelt



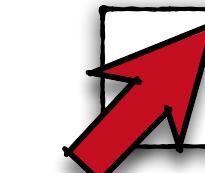
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

6.3 ASYNCHRONE KOMMUNIKATION

DIE fetch API (II)

```
> fetch("https://dummyjson.com/products")
  .then(response=>response.json())
  .then(data=>console.log(data))
  .catch(()=>console.log("Netzwerkfehler"));
<- ▶ Promise {<pending>}
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

- Mit Chained Promises kann eine HTTP-Antwort in mehreren asynchronen Schritten bearbeitet werden
- Umwandlung der im Body der HTTP-Antwort enthaltenen Daten im JSON-Format mittels der Methode `json()` auf `response` (asynchron)
- Hinweis: im Gegensatz zu `JSON.parse()` arbeitet `response.json()` asynchron
- Danach Ausgabe des JSON-Objektes



https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

6.3 ASYNCHRONE KOMMUNIKATION

DIE fetch API (III)

```
> fetch('https://dummyjson.com/products/add', {  
    method: 'POST',  
    headers: { 'Content-Type': 'application/json' },  
    body: JSON.stringify({  
        title: 'LG 32-Zoll-Monitor'  
    })  
}).then(res => res.json())  
.then(console.log);  
< ► Promise {<pending>}  
► {id: 101, title: 'LG 32-Zoll-Monitor'}
```

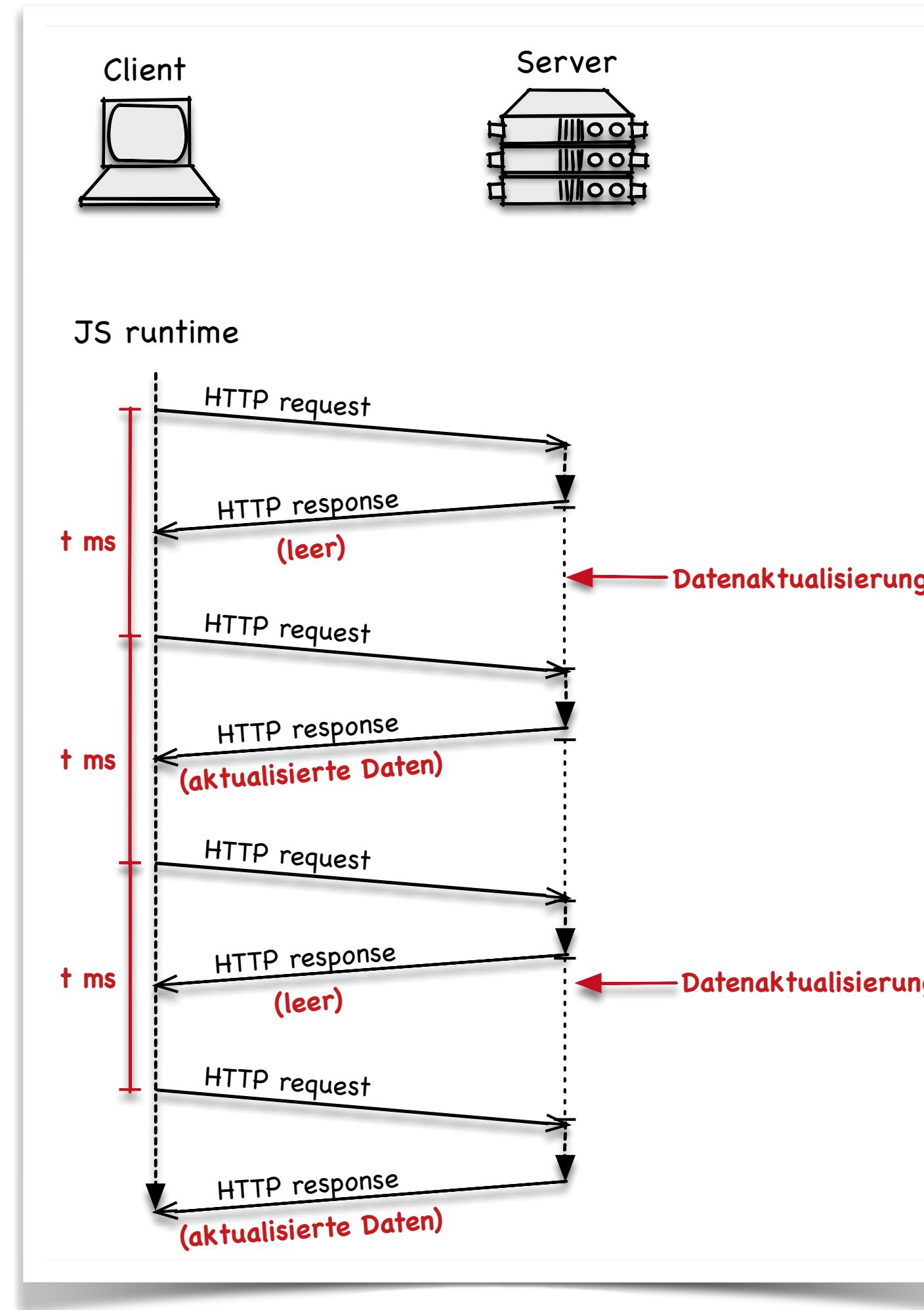
```
> var req=new Request('https://dummyjson.com/products/add', {  
    method: 'POST',  
    headers: {'Content-Type': 'application/json'},  
    body: JSON.stringify({  
        title: 'LG 32-Zoll-Monitor'  
    })  
});  
< undefined  
> fetch(req)  
.then(res=>res.json())  
.then(console.log);  
< ► Promise {<pending>}  
► {id: 101, title: 'LG 32-Zoll-Monitor'}
```

- `fetch` kann mit einem zweiten Argument, neben der URL, angepasst werden
 - Spezifikation der HTTP-Methode
 - Definition von HTTP-Headers (Achtung: nicht alle Headers sind aus Sicherheitsgründen erlaubt)
 - Spezifikation des HTTP-Body
 - Weitere Parameter, siehe `fetch` API
- Alternativ können `fetch`-Argumente mit einem `Request`-Konstruktor spezifiziert werden



https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

6.3 ASYNCHRONE KOMMUNIKATION REGULAR ODER SHORT POLLING (I)

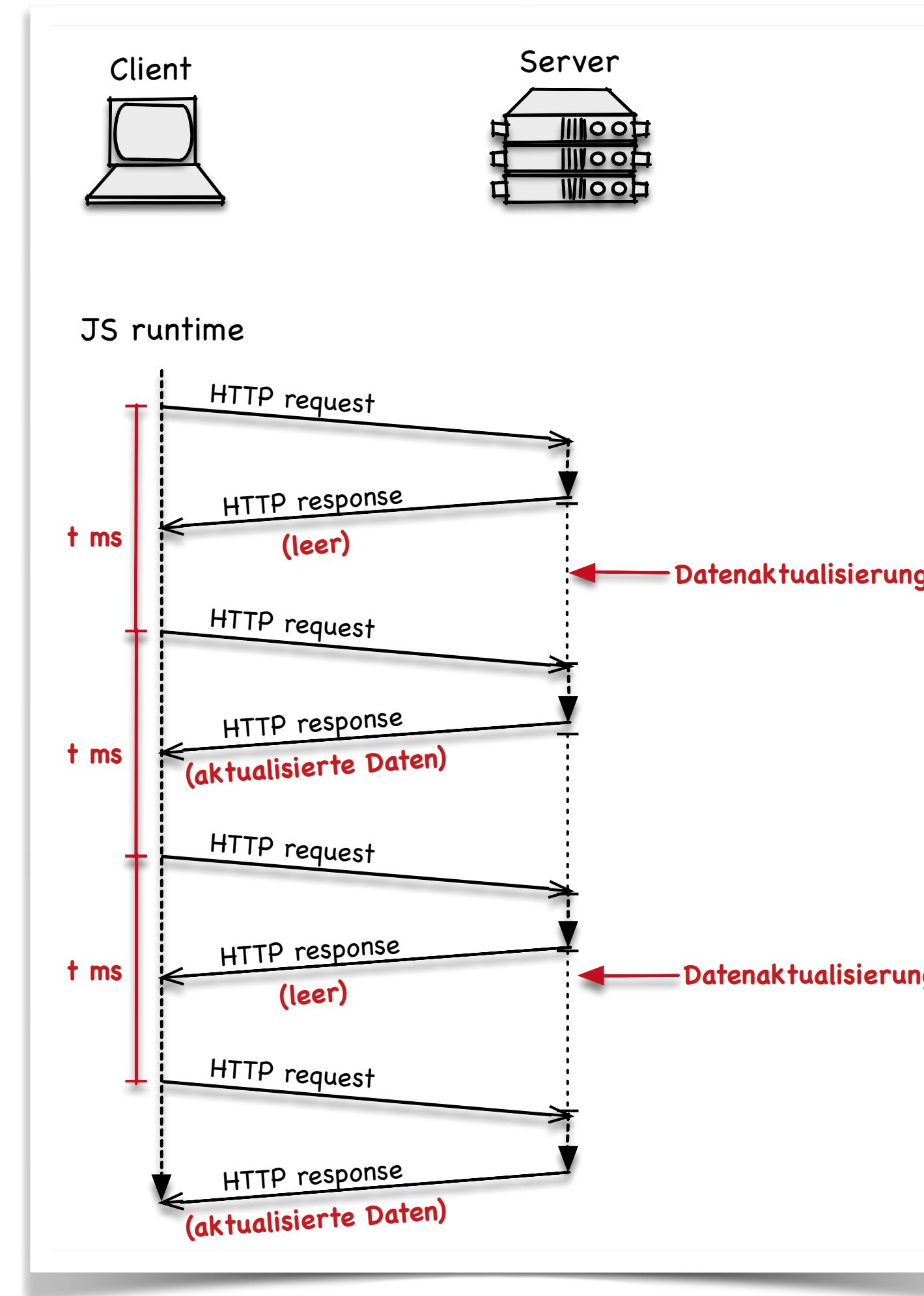


- Hoher Bedarf an Webanwendungen mit Push-Funktionalität: Chat, Alarme, Collaborative Office-Umgebungen, etc.
- Problem: Server kann keine Verbindungen zum Client aufbauen

POLLING

- Wiederholtes Aufrufen der `send()`-Methode in regelmäßigen Abständen von $t \text{ ms}$
- Nach Erhalt jeder Anfrage prüft der Server ob seit der letzten Anfrage eine Datenaktualisierung aufgetreten ist
- Wenn ja, sendet Server die Daten zum Client
- Wenn nein, wird leere Antwort gesendet
- Steuerung durch `setInterval()`-Funktion

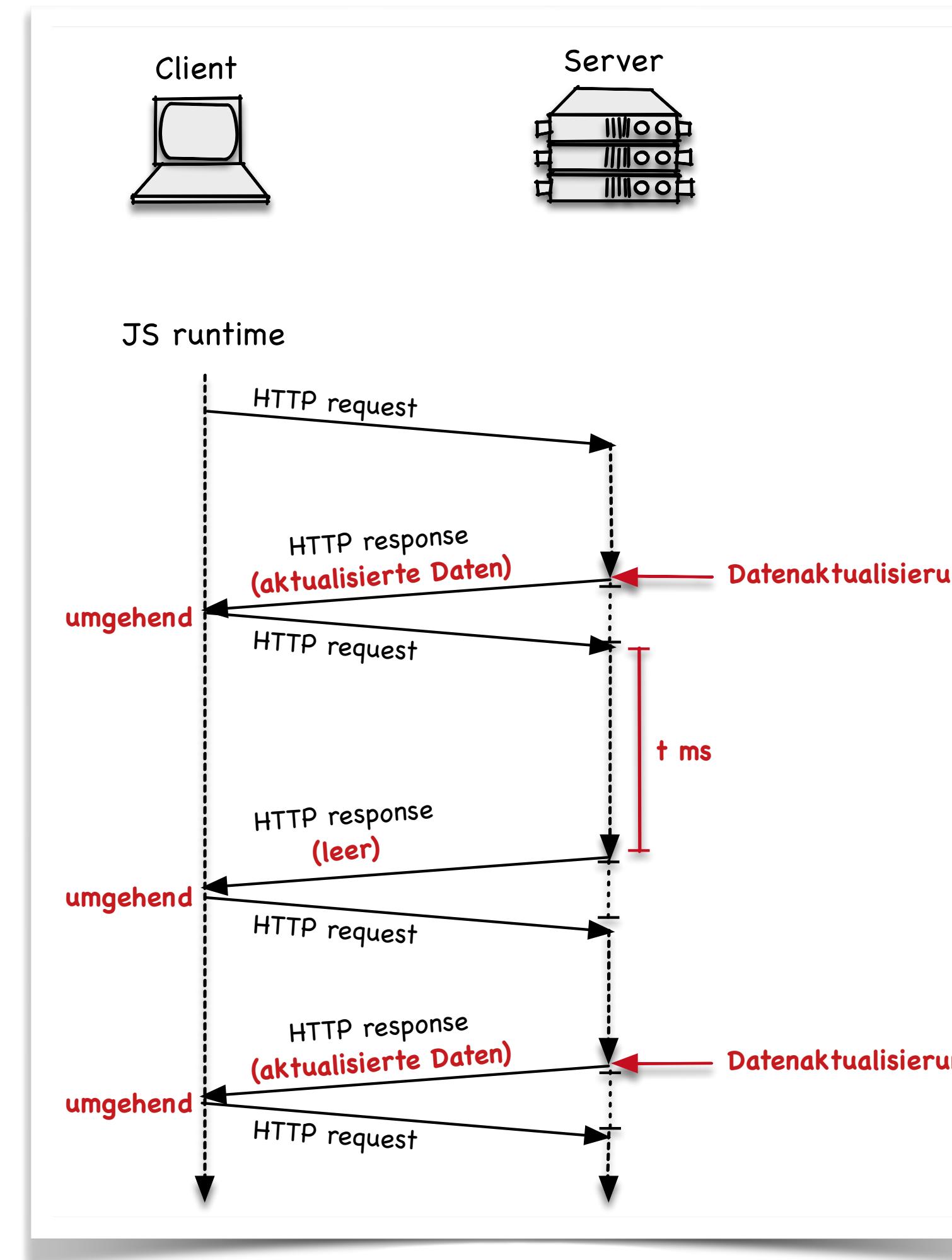
6.3 ASYNCHRONE KOMMUNIKATION REGULAR ODER SHORT POLLING (II)



```
> setInterval(()=>{
    fetch("https://dummyjson.com/products")
      .then((response) => {console.log(response.json())})
  },1000)
< 47
▶ Promise {<pending>}
```

6.3 ASYNCHRONE KOMMUNIKATION

LONG POLLING (I)

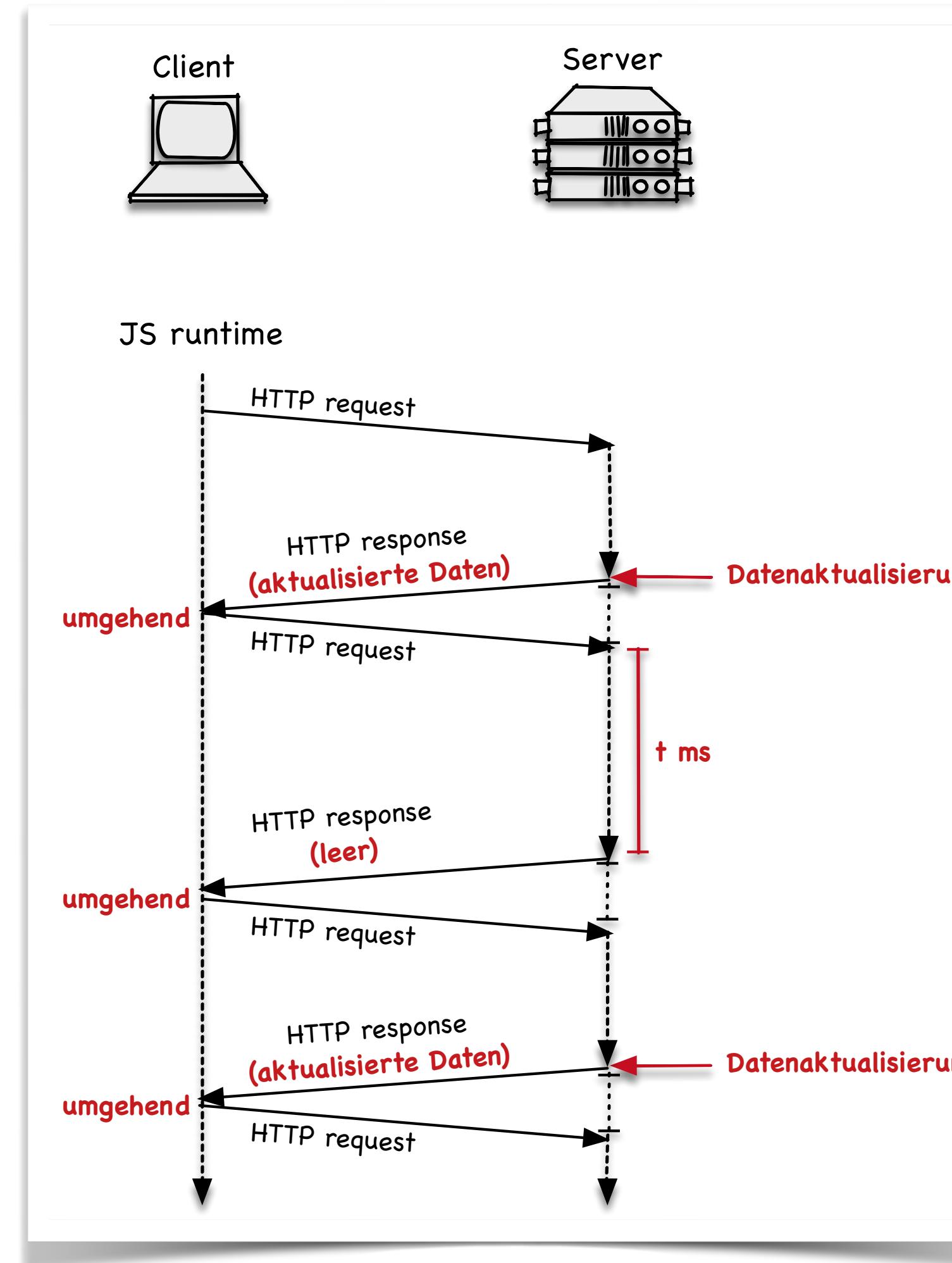


- Problem von Polling: Tradeoff zwischen dem Aufwand für den Austausch von HTTP-Anfragen und Antworten und der zeitnahen Zustellung von Datenaktualisierungen
- Reduzierung der Aufwands durch Vergrößerung des Anfrageintervalls führt zu verzögerter Zustellung von Daten
- Geringe Verzögerung führt zu häufigen Anfrage/Antwort-Zyklen

LONG POLLING

- Serverseitige Implementierung des Timeouts
- Server beantwortet eine HTTP-Anfrage nicht bis eine Datenaktualisierung oder ein Timeout eintritt
- Bei Datenaktualisierung wird Antwort mit aktualisierten Daten gesendet
- Nach Eintreffen der Antwort initiiert der Client sofort eine neue HTTP-Anfrage
- Client und Server verfügen über eine quasi permanent andauernde Verbindung

6.3 ASYNCHRONE KOMMUNIKATION LONG POLLING (II)



```
> function longPolling() {
  fetch('https://dummyjson.com/products')
    .then(res=>res.json())
    .then(data=>console.log(data))
    .then(()=>longPolling())
};

< undefined
> longPolling()
< undefined

▶ {products: Array(30), total: 100, skip: 0, limit: 30}
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
▶ {products: Array(30), total: 100, skip: 0, limit: 30}
```

FORTSETZUNG FOLGT



VORLESUNG

Prof. Dr. Axel Küpper

TU Berlin | T-Labs | Fachgebiet *Service-centric Networking*
Ernst-Reuter-Platz 7 | 10587 Berlin | Germany

 axel.kuepper@tu-berlin.de

 <https://twitter.com/kuepp>

 <https://www.linkedin.com/in/axelkuepper/>

 <http://www.snet.tu-berlin.de/kuepper>

ÜBUNGSLEITER

- Thomas Cory
- Sanjeet Raj Pandey
- Christian René Sechting

TUTOREN

- Nastassia Lukyanovich
- Maximilian Oliver Fisch
- Leonhardt Frederik Hollatz
- Adrian Siebing