# Kapitel WT:II

## II. Kommunikation und Protokolle für Web-Systeme

- ❑ Rechnernetze
- ❑ Prinzipien des Datenaustauschs
- ❑ Netzsoftware und Kommunikationsprotokolle
- ❑ Internetworking

- ❑ Client-Server-Interaktionsmodell
- ❑ Uniform Resource Locator

- ❑ Grundlagen HTTP-Protokoll
- ❑ Weitere HTTP-Konzepte
- ❑ Grundlagen TLS-Protokoll

- ❑ Zeichen und Codierung

# Grundlagen HTTP-Protokoll

Der HTTP-Standard sieht das Client-Server-Prinzip mit folgenden Funktionseinheiten vor [UML: Deployment] [RFC 2616] :

1. WWW-Client “bzw.” User-Agent
   Initiiert Verbindungen zu WWW-Servern; der WWW-Client ist in der Regel ein Web-Browser.

2. WWW-Server
   Wartet auf Verbindungswünsche von WWW-Clients und antwortet auf die gestellten Anfragen; liefert gewünschte Ressource oder Statusinformation.

3. Proxy-Server
   System zwischen WWW-Client und WWW-Server; arbeitet sowohl als WWW-Server (hat aufgrund früherer Kommunikation Antworten im Cache) als auch als WWW-Client gegenüber dem sogenannten *Origin-Server*.

4. Gateway
   Vergleichbar dem Proxy-Server mit dem Unterschied, dass der WWW-Client keine Kenntnis über die Existenz des Gateways besitzt.

# Grundlagen HTTP-Protokoll

Der HTTP-Standard sieht das Client-Server-Prinzip mit folgenden Funktionseinheiten vor [UML: Deployment] [RFC 2616] :

1. WWW-Client “bzw.” User-Agent
   Initiiert Verbindungen zu WWW-Servern; der WWW-Client ist in der Regel ein Web-Browser.

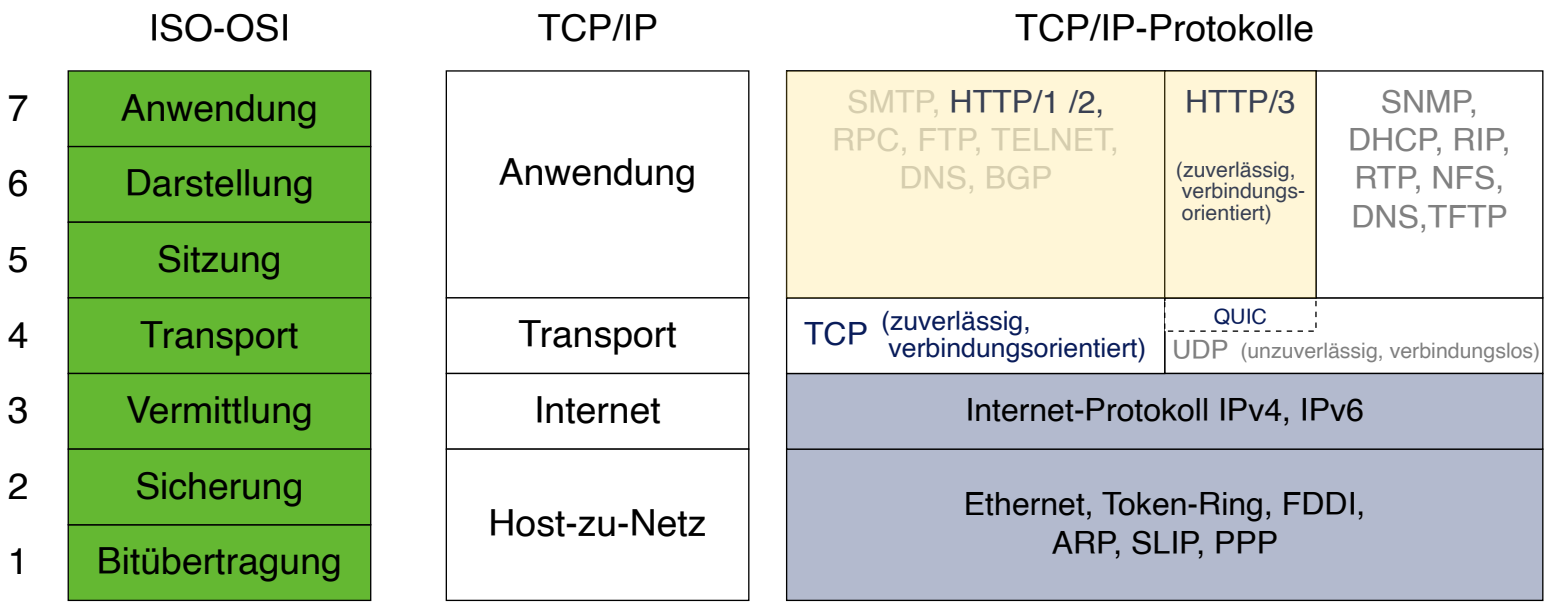
2. WWW-Server
   Wartet auf Verbindungswünsche von WWW-Clients und antwortet auf die gestellten Anfragen; liefert gewünschte Ressource oder Statusinformation.

3. Proxy-Server
   System zwischen WWW-Client und WWW-Server; arbeitet sowohl als WWW-Server (hat aufgrund früherer Kommunikation Antworten im Cache) als auch als WWW-Client gegenüber dem sogenannten *Origin-Server*.

4. Gateway
   Vergleichbar dem Proxy-Server mit dem Unterschied, dass der WWW-Client keine Kenntnis über die Existenz des Gateways besitzt.

# Grundlagen HTTP-Protokoll

## Einordnung im Protokollstapel [TLS]



# Grundlagen HTTP-Protokoll

## Historie

1992 HTTP/0.9 versteht nur die GET-Methode, keine Statusinformation und keine Information über Medientypen.

1996 HTTP/1.0 [RFC 1945]

1997-2014 HTTP/1.1 ermöglicht unter anderem Pipelining. [RFC 2068, RFC 7235] [W3C]

2015 HTTP/2 mit deutlich verbesserter Performanz, Multiplexing, Kompression, Priorisierungsstrategien, Server-initierten Anfragen. [RFC 7540] [HTTP/2 Home] [Wikipedia]

2020 HTTP/3 ersetzt TCP durch das Netzwerkprotokoll QUIC. Verwendet die gleiche Semantik (Anfragemethoden, Statuscodes, Nachrichtenfelder) wie die früheren HTTP-Versionen. [RFC 9000] [Wikipedia]

2022 Das IETF publiziert HTTP/3 als “Proposed Standard”. [RFC 9114] [Wikipedia]

Bemerkungen:

- Pipelining (HTTP/1.1) ist die Abwicklung von mehreren HTTP-Anfragen über *eine* TCP/IP-Verbindung.
- Multiplexing (HTTP/2) ist das Zusammenfassen von mehreren HTTP-Anfragen. Die Entwicklung des HTTP/2 Protokolls wurde von Google (SPDY) und Microsoft forciert.
- QUIC, die Basis von HTTP/3, setzt nicht mehr auf das verbindungsorientierte TCP, sondern realisiert eine verbindungsorientierte Kommunikation über das verbindungslose User Datagram Protocol (UDP).Weiterhin ist QUIC kompatibel mit dem Stream Control Transmission Protocol, SCTP.
- HTTP/3 ist generell verschlüsselt und verspricht deutliche Geschwindigkeitsvorteile, jedoch ist aus Gründen der Soft- und Hardware-Kompatibilität keine schnelle Verbreitung zu erwarten.

# Grundlagen HTTP-Protokoll

## Übersicht

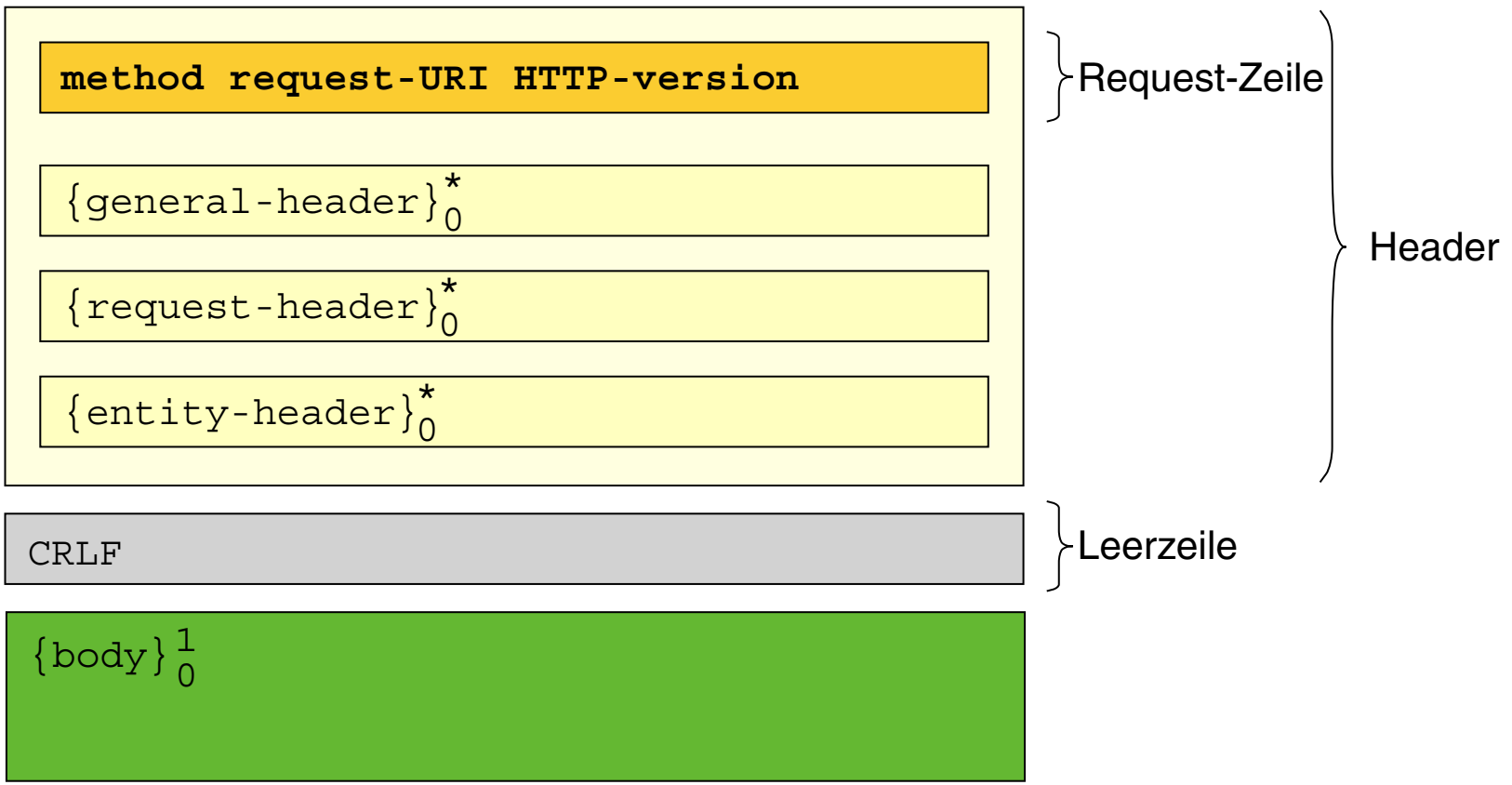
Das HTTP-Protokoll spezifiziert Nachrichtentypen, Datentransfer, Regeln zur Darstellung (Zeichensatz, Datenformate), Inhaltsabstimmung, Authentisierung, etc. zwischen den genannten Funktionseinheiten.

Kommunikationsablauf aus Client-Sicht:

1. Herstellen einer Verbindung zum WWW-Server.
   (a) Einrichtung einer zuverlässigen Verbindung via TCP. (bis HTTP/2)
   (b) Einrichtung einer zuverlässigen Verbindung mit QUIC auf UDP. (ab HTTP/3)
2. Request. Senden der Anforderung an WWW-Server.
3. Response. Empfangen der Antwort vom WWW-Server.
4. Schließen der Verbindung.

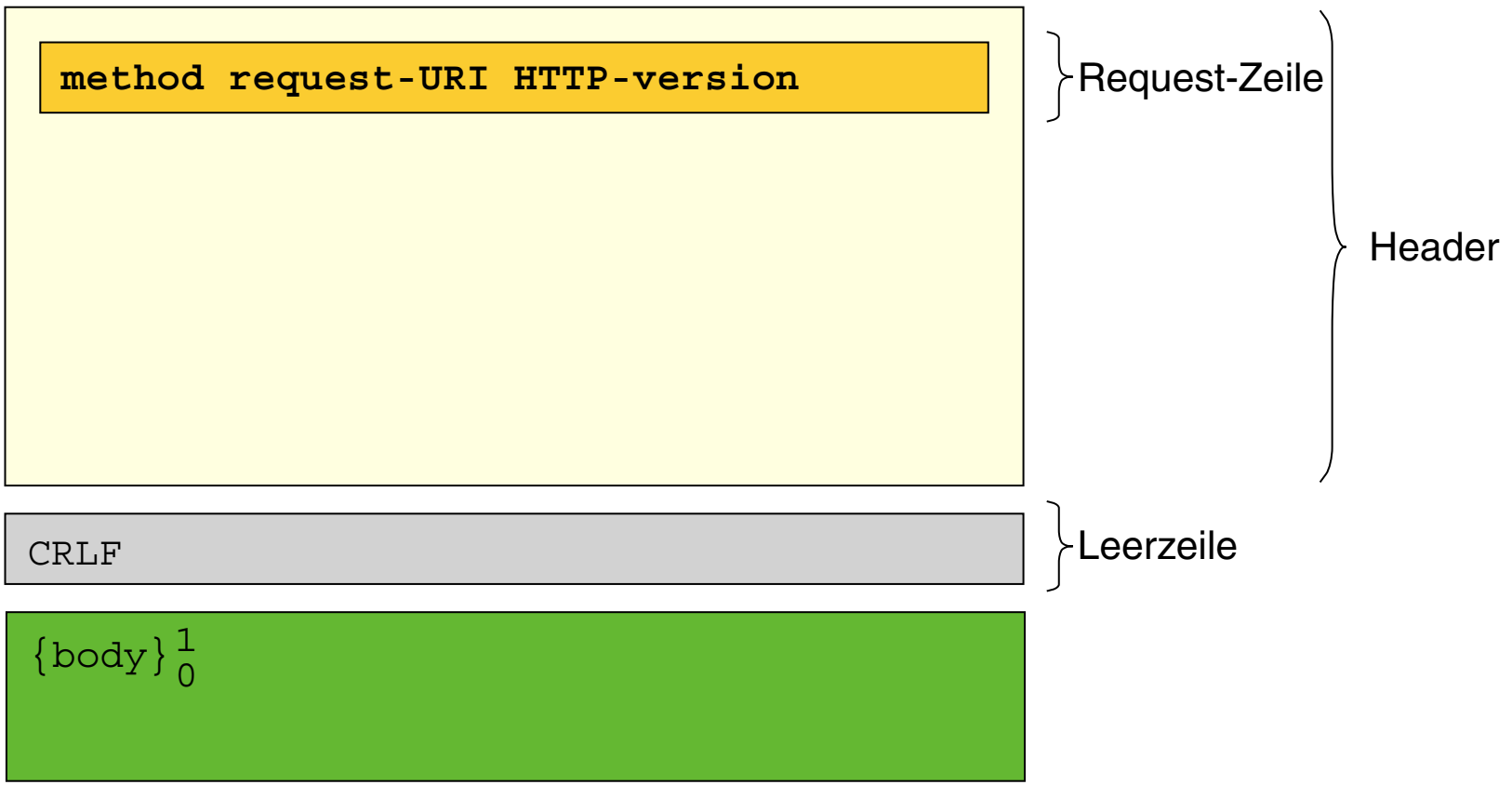
# Grundlagen HTTP-Protokoll

## HTTP-Request-Message [Message-Header]



# Grundlagen HTTP-Protokoll

## HTTP-Request-Message




Beispiel für Request-Zeile: `GET www.uni-weimar.de/ HTTP/1.1`

# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Methoden

| Methode | |
|---|---|
| GET | Anfrage der im Request-URI angegebenen Ressource. Client-Daten wie z.B. HTML-Feldwerte werden als Bestandteil der URI der Ressource übergeben. |

# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Methoden

| Methode | |
|---|---|
| GET | Anfrage der im Request-URI angegebenen Ressource. Client-Daten wie z.B. HTML-Feldwerte werden als Bestandteil der URI der Ressource übergeben. |
| POST | Wie GET, jedoch werden Client-Daten nicht an die URI angehängt, sondern im Message-Body untergebracht. |
| HEAD | Wie GET, jedoch darf der Server keinen Message-Body zurücksenden. Wird u.a. zur Cache-Validierung verwendet. |

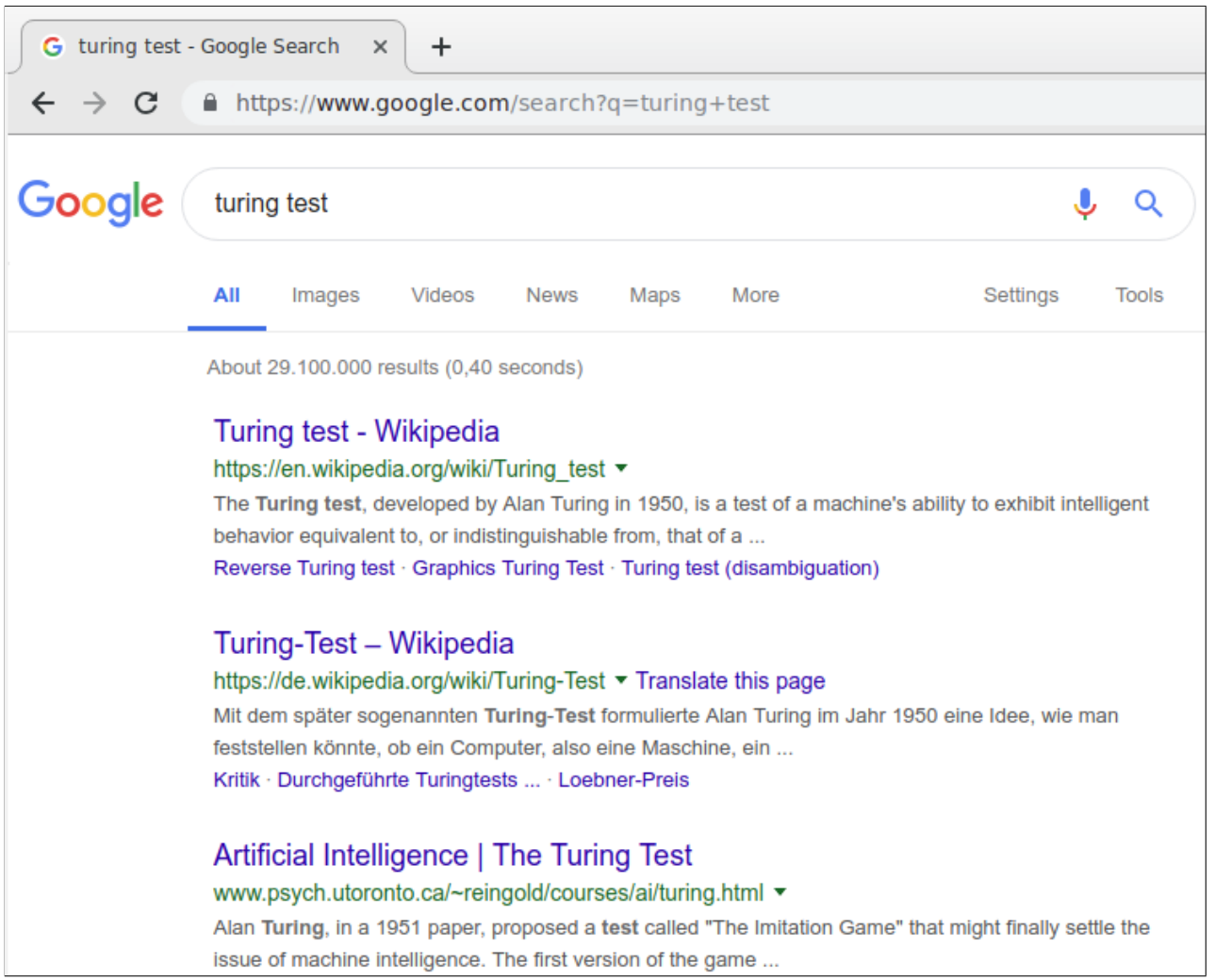
# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Methoden

| Methode | |
|---|---|
| GET | Anfrage der im Request-URI angegebenen Ressource. Client-Daten wie z.B. HTML-Feldwerte werden als Bestandteil der URI der Ressource übergeben. |
| POST | Wie GET, jedoch werden Client-Daten nicht an die URI angehängt, sondern im Message-Body untergebracht. |
| HEAD | Wie GET, jedoch darf der Server keinen Message-Body zurücksenden. Wird u.a. zur Cache-Validierung verwendet. |
| PUT | Client erzeugt mit Daten des Message-Body auf dem Server eine neue Ressource an der Stelle der angegebenen Request-URI. |
| DELETE | Löschen der im Request-URI angegebenen Ressource auf dem Server. |
| OPTIONS | Abfrage der vorhandenenen Kommunikationsmöglichkeiten entlang der Verbindungsstrecke zum Server. |
| TRACE | Verfolgen eines Requests auf dem Weg zum Server durch die Proxies. |
| CONNECT | Verbindungsherstellung zum Proxy-Server, um Tunnelbetrieb einzurichten. Anwendung: Einrichtung einer SSL-Verbindung (*Secure Socket Layer*) |

# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Beispielanfrage mittels GET-Methode (1)




[www.google.de]

# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Beispielanfrage mittels GET-Methode (2)

```
user@webis:~$ telnet webtec.webis.de 80
```

# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Beispielanfrage mittels GET-Methode (2)

```
user@webis:~$ telnet webtec.webis.de 80
Trying 141.54.132.157...
Connected to webtec.webis.de.
Escape character is '^]'.
```

# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Beispielanfrage mittels GET-Methode (2)

```
user@webis:~$ telnet webtec.webis.de 80
Trying 141.54.132.157...
Connected to webtec.webis.de.
Escape character is '^]'.
GET /helloworld.html HTTP/1.1
HOST: webtec.webis.de
```

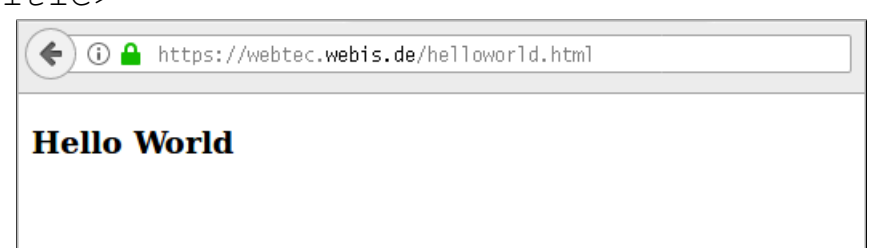
# Grundlagen HTTP-Protokoll

## HTTP-Request-Message: Beispielanfrage mittels GET-Methode (2)

```
user@webis:~$ telnet webtec.webis.de 80
Trying 141.54.132.157...
Connected to webtec.webis.de.
Escape character is '^]'.
GET /helloworld.html HTTP/1.1
HOST: webtec.webis.de

HTTP/1.1 200 OK
Server: nginx
Date: Tue, 22 Apr 2025 21:30:55 GMT
Content-Type: text/html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Helloworld</title>
</head>
<body>
  <h1>Hello World</h1>
</body>
</html>
```




[webtec.webis.de:80]

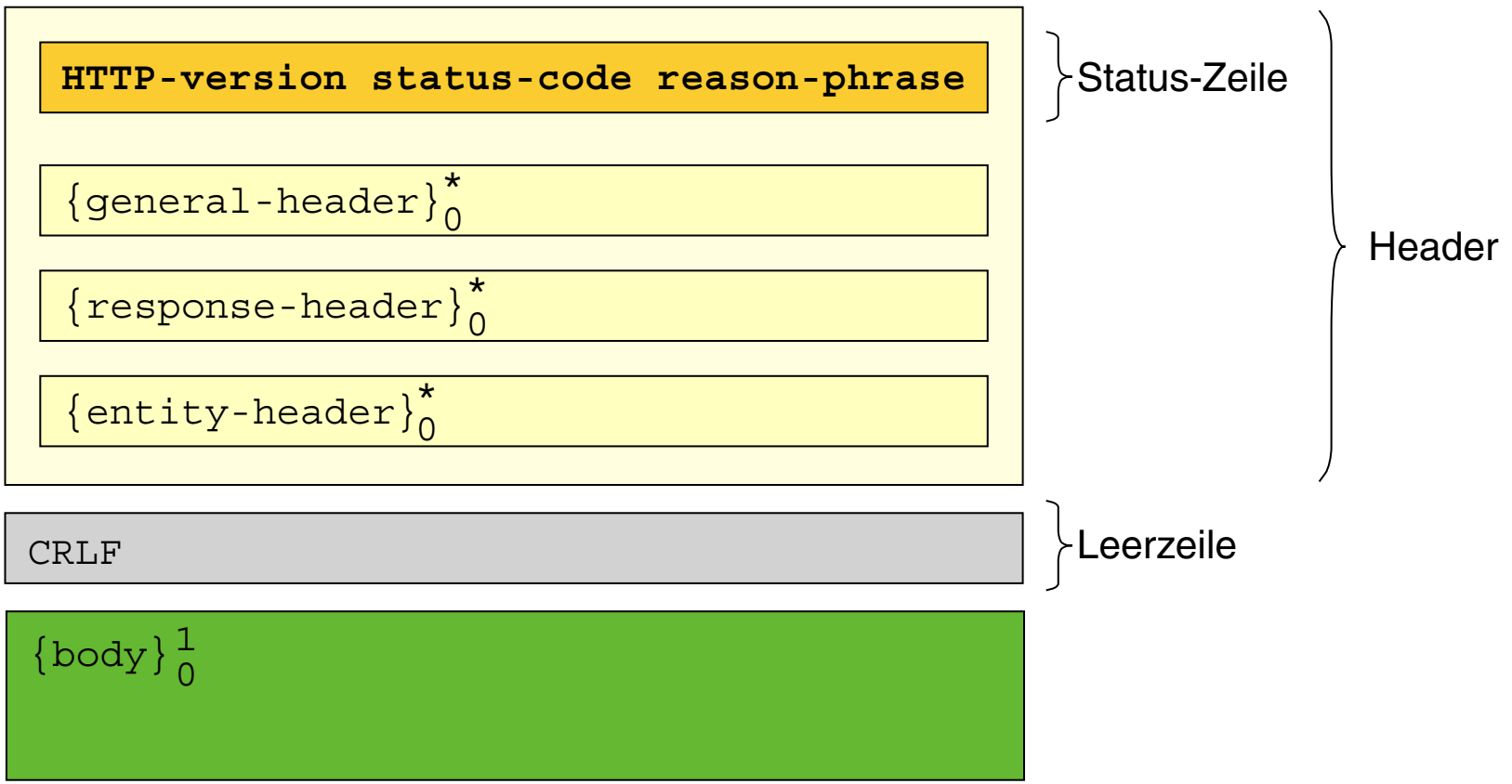
[webtec.webis.de:443]

Bemerkungen:

- Mittels “telnet” wird lokal der Telnet-Client gestartet, der sich mit dem entfernten Web-Server verbindet. Um vom Web-Server als Web-Client (bspw. Browser) akzeptiert zu werden, bildet man das HTTP-Protokoll durch die entsprechenden Eingaben nach.
- Telnet ist ein altes Client/Server-Protokoll und basiert auf einem zeichenorientierten Datenaustausch üblicherweise über eine TCP-Verbindung. [RFC 854] [Wikipedia]

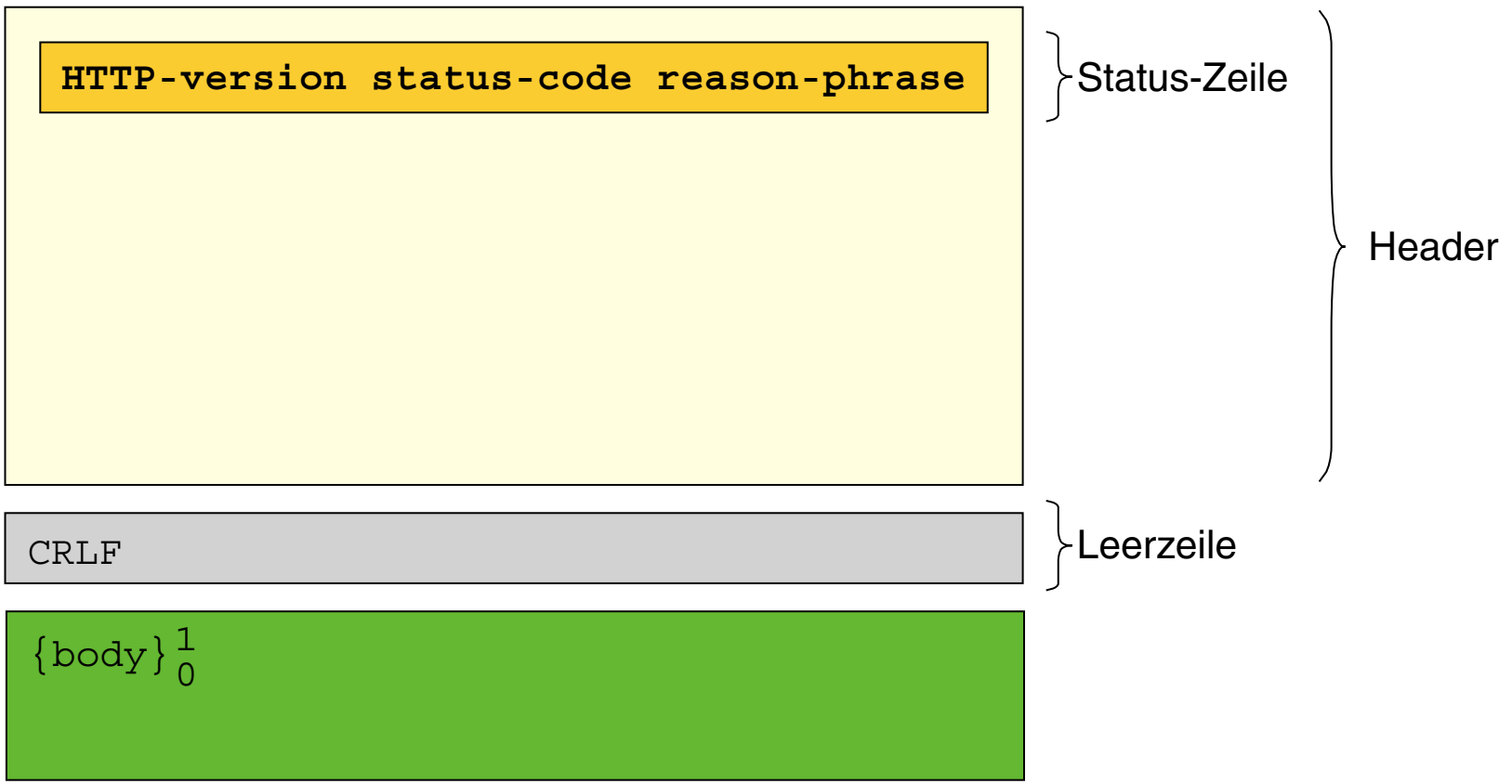
# Grundlagen HTTP-Protokoll

## HTTP-Response-Message [Message-Header]



# Grundlagen HTTP-Protokoll

## HTTP-Response-Message




Beispiel für Status-Zeile: `HTTP/1.1 200 OK`

# Grundlagen HTTP-Protokoll

## HTTP-Response-Message: Status-Codes

Der Status-Code besteht aus 3 Ziffern und gibt an, ob eine Anfrage erfüllt wurde bzw. welcher Fehler aufgetreten ist.

| Status-Code-Kategorie | | |
|---|---|---|
| 1xx | Informational | Anforderung bekommen (selten verwendet). |
| 2xx | Success | Anforderung bekommen, verstanden, akzeptiert und ausgeführt. |
| 3xx | Redirection | Anweisung an den Client, an welcher Stelle die Seite zu suchen ist. |
| 4xx | Client Error | Fehlerhafte Syntax oder unerfüllbar, da z.B. Seite nicht vorhanden. |
| 5xx | Server Error | Server kann Anforderung nicht ausführen aufgrund eines Fehlers in der Systemsoftware oder wegen Überlastung. |

# Grundlagen HTTP-Protokoll

## HTTP-Response-Message: Status-Codes (Fortsetzung)

100 Continue
101 Switching protocols

**200 OK**
201 Created
202 Accepted
203 Non-authoritative information
204 No content
205 Reset content
206 Partial content

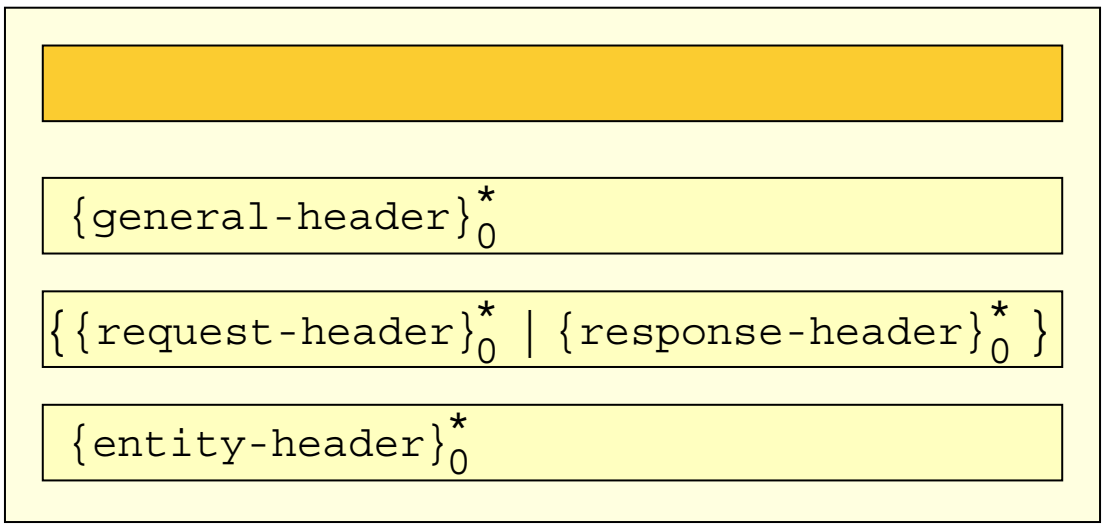
300 Multiple choices
**301 Moved permanently**
**302 (veraltet)** --> 303, 307
**303 See other**
**304 Not modified**
305 Use proxy
**307 Temporary redirect**

**400 Bad request**
**401 Unauthorized**
402 Payment required
**403 Forbidden**
**404 Not found**
405 Method not allowed
406 Not acceptable
407 Proxy authentication required
408 Request timeout
411 Length required
412 Precondition failed
413 Request entity too large
414 Request URI too large
415 Unsupported media type
416 Requested range not satisfiable
417 Expectation failed
**418 I'm a teapot**
**424 Site too ugly**

**500 Internal server error**
501 Not implemented
**502 Bad gateway**
**503 Service unavailable**
**504 Gateway time out**
505 HTTP version not supported

# Grundlagen HTTP-Protokoll

## HTTP-Message-Header [Request-Message] [Response-Message]




- General-Header
  Meta-Information zu Protokoll und Verbindung.
- Request-Header | Response-Header
  Informationen zur Anfrage oder zum Client | zur Antwort des Servers.
- Entity-Header
  Meta-Information über den Inhalt im Message-Body.
  Beispiele: Content-Encoding, Content-Language, Content-Type

Bemerkungen:

- BNF-Notation für den Aufbau von Header-Zeilen:

```
header   ::= keyword ':' value
keyword  ::= 'Date' | 'Server' | 'Last-Modified' | 'Set-Cookie' |
             'Content-Length' | 'Content-Type' | ...
```

- Analyse der Redirects einer URL mit HTTP-Status-Code-Checker. Beispiel:

```
http://uni-weimar.de
  →
https://www.uni-weimar.de/
  →
https://www.uni-weimar.de/de/
  →
https://www.uni-weimar.de/de/universitaet/start/
```

# Grundlagen HTTP-Protokoll

## HTTP-Response-Message: Beispielantwort [Java]

```
HTTP/1.1 200 OK
Date: Tue, 15 Apr 2014 19:17:35 GMT
Server: Apache/2.2.12 (Unix) DAV/1.0.3 PHP/4.3.10
    mod_ssl/2.8.16 OpenSSL/0.9.7c
Last-Modified: Sat, 22 Mar 2014 14:11:21 GMT
ETag: "205e812-1479-42402789"
Accept-Ranges: bytes
Content-Length: 5241
Connection: close
Content-Type: text/html; charset=utf-8
```

Status line
General header
Response header
Entity header
Response header
Entity header
General header
Entity header

```
<!DOCTYPE html>
<html lang="de-DE" xmlns="http://www.w3.org/1999/...
<head>
<base href="http://www.uni-weimar.de">
<title>Bauhaus-Universit&auml;t Weimar</title>
...
```

# Grundlagen HTTP-Protokoll

## Content Type / MIME Type / Media Type

Medientypen bestehen aus einem Top-Level-Typ und einem Untertyp und können durch Parameter weiter spezifiziert werden. [Wikipedia]

| Typ | Untertyp | Beschreibung |
|---|---|---|
| text | plain | unformatierter ASCII-Text |
| | enriched | ASCII-Text mit einfachen Formatierungen |
| image | gif | Standbild im GIF-Format |
| | jpeg | Standbild im JPEG-Format |
| audio | basic | Klangdaten |
| video | mp4 | MPEG-4 (ISO/IEC 14496) |
| application | octet-stream | nicht-interpretierte Byte-Folge |
| | postscript | druckbares Dokument im PostScript-Format |
| ... | ... | ... |

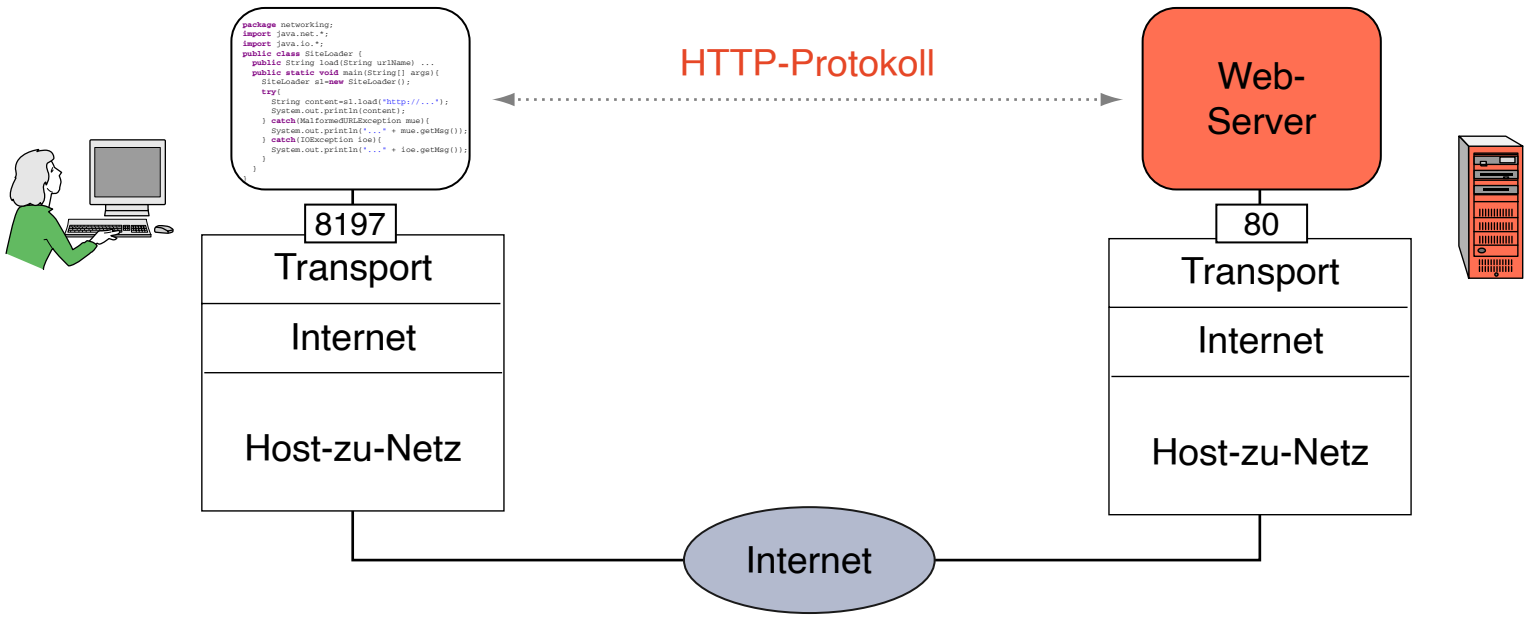
Eine Übersicht der erlaubten Medientypen findet sich bei der IANA.

Bemerkungen:

- Die MIME-Type-Spezifikation ist in RFC 2046 beschrieben.
- Die Abkürzung MIME steht für Multipurpose Internet Mail Extensions. Die zugehörigen Spezifikationen finden Anwendung bei der Deklaration von Inhalten in Internetprotokollen.

# Grundlagen HTTP-Protokoll

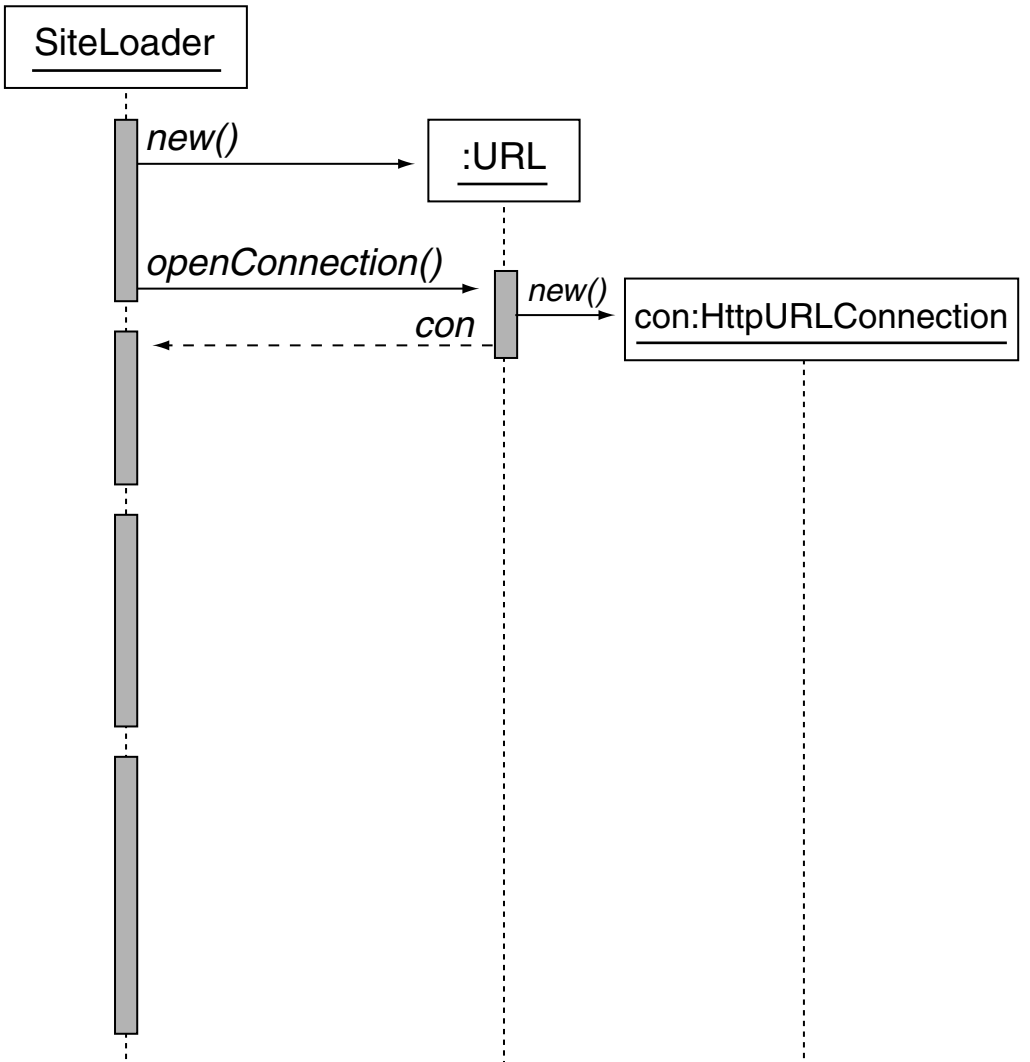
## HTTP-Kommunikation [Server-side] [ISO-OSI-Modell]




Auf der Server-Seite bildet ein WWW-Server das Gegenstück einer Verbindung zu dem Beispiel-Client. Der WWW-Server lässt sich auf diese Art anfragen, weil das HTTP-Protokoll vom Client korrekt abgewickelt wird. [RFC 2616]

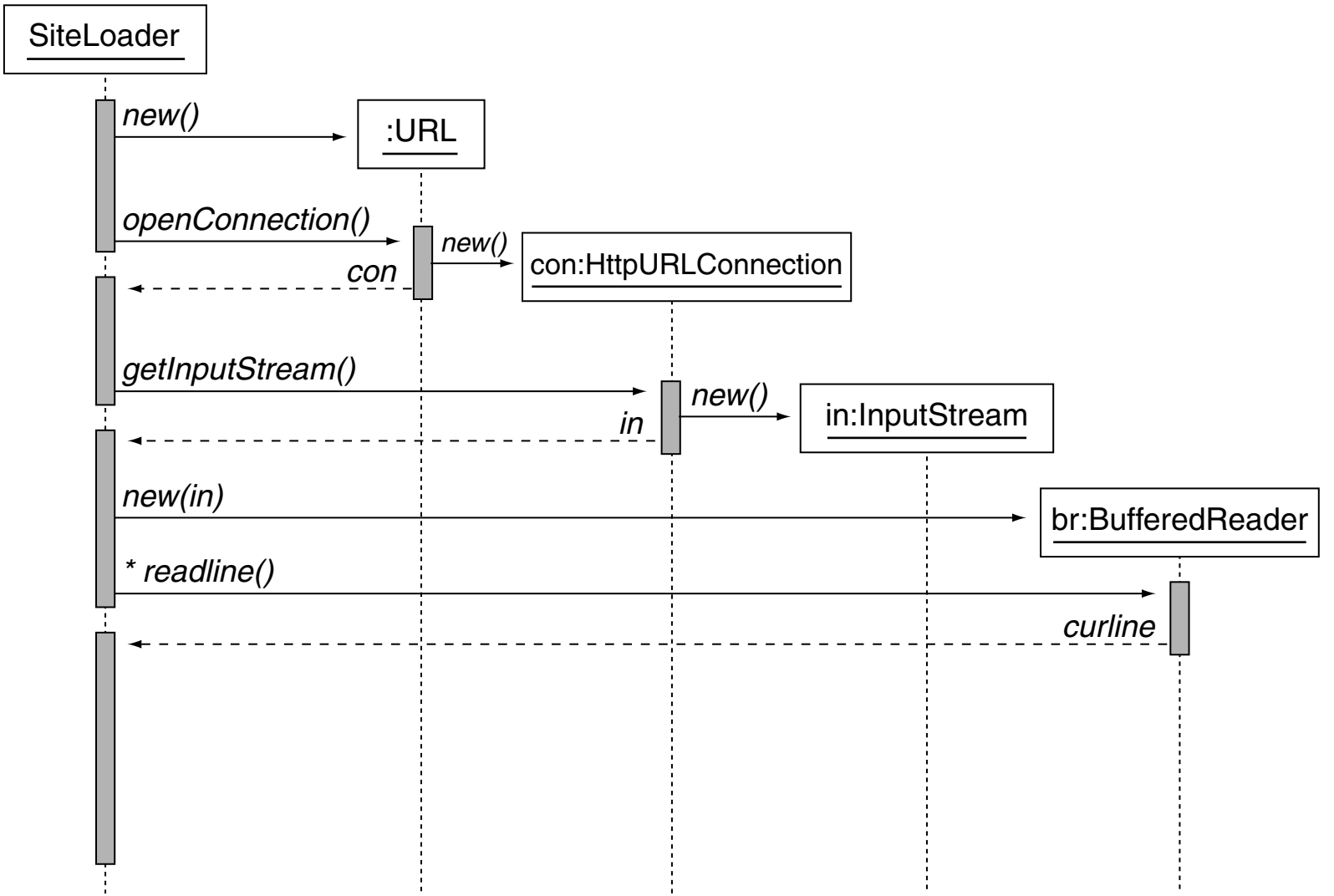
# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Java [Javadoc]



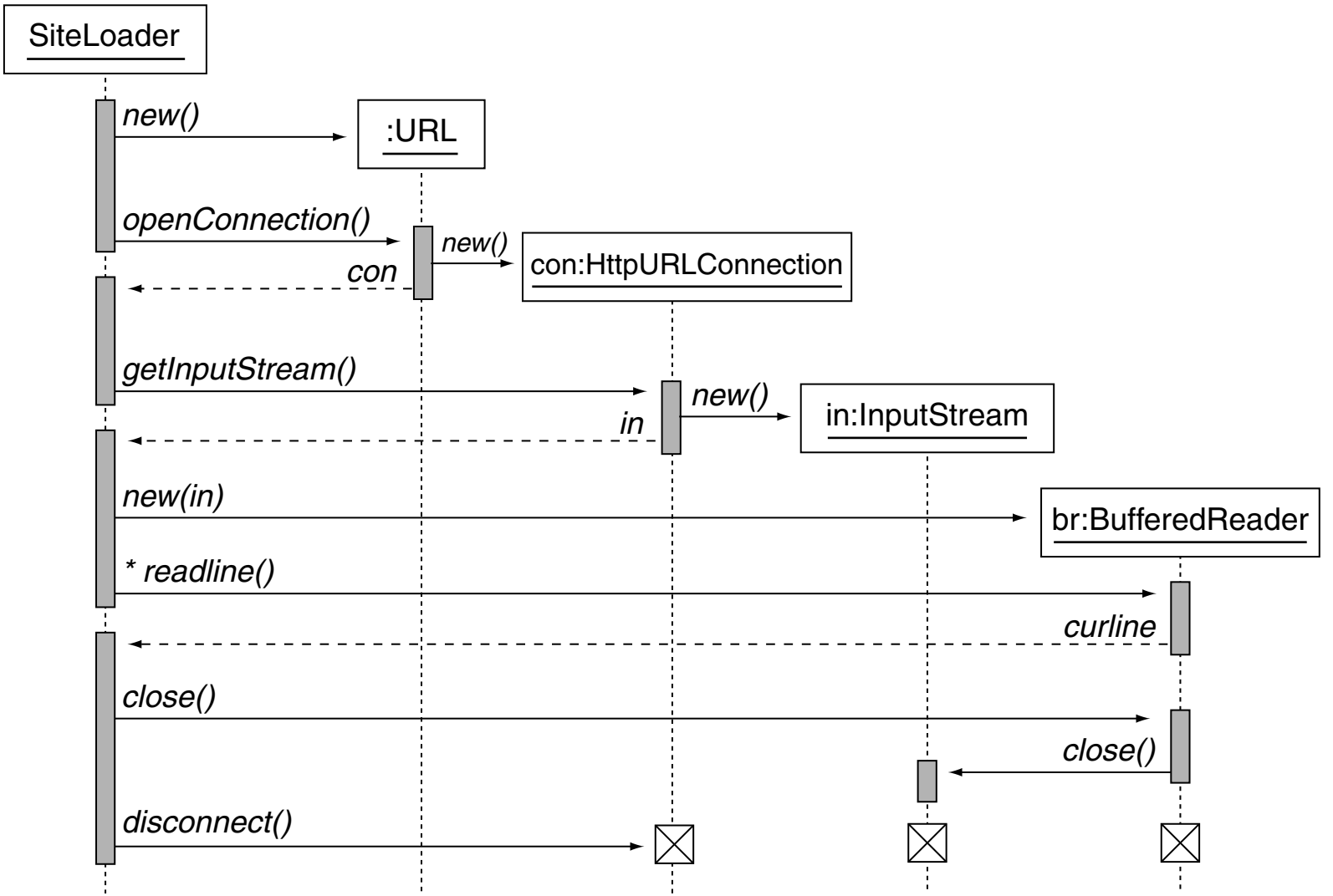
# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Java (Fortsetzung) [Javadoc]



# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Java (Fortsetzung) [Javadoc]



# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Java [Response-Message]

```
package networkprotocol;
import java.net.*;
import java.io.*;

public class SiteLoader {

    public static String load(String urlString) ...

    public static void main(String[] args) {
        try{
            String content = SiteLoader.load("https://www.heise.de");
            System.out.println(content);
        }
        catch(MalformedURLException e) {
            System.out.println("MalformedURLException:" + e.getMessage());
        }
        catch(IOException e) {
            System.out.println("IOException:" + e.getMessage());
        }
    }
}
```

# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Java (Fortsetzung) [Response-Message]

```
public static String load(String urlString) throws IOException {
    URL url = new URL(urlString);
    HttpURLConnection con = (HttpURLConnection) url.openConnection();

    System.out.println(con.getResponseCode() + " "
        + con.getResponseMessage()); // will print "200 OK"

    InputStream in = con.getInputStream();
    BufferedReader br = new BufferedReader(new InputStreamReader(in));
    String curline;
    StringBuilder content = new StringBuilder();

    while ((curline = br.readLine()) != null) {
        content.append(curline + '\n');
    }

    br.close();
    con.disconnect();
    return content.toString();
 }
```

# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Python [Response-Message]

```
from urllib import request, error

def load(url):
    with request.urlopen(url) as stream:
        print(stream.code, stream.reason)
        return stream.read().decode('UTF-8')

def main():
    try:
        content = load('https://heise.de')
        print(content)

    except error.HTTPError as e:
        print('HTTPError: ' + str(e))

    except error.URLError as e:
        print('URLError: ' + str(e))

if __name__ == '__main__':
    main()
```

# Grundlagen HTTP-Protokoll

## HTTP-Kommunikation mit Java / Python (Fortsetzung)

```
user@webis: bin$ java networkprotocol.SiteLoader | less

user@webis: python$ python3 site_loader.py | less

200 OK
<!DOCTYPE html>
<html lang="de">

<head>
    <title>heise online - IT-News, Nachrichten und Hintergrûnde
      </title>
        <meta name="description" content="News und Foren zu Computer, IT, ...
            <meta name="keywords" content="heise online, c't, iX, Technology ...
        <script type="text/javascript" src="/js/mobi/webtrekk_abtest.js"></script>

<meta charset="utf-8">
<meta name="publisher" content="Heise Medien" />
<meta name="viewport" content="width=1175" />
<link rel="alternate" media="only screen and...
<link rel="copyright" title="Copyright" href="/impressum.html" />

                <!--googleoff: all-->
    <meta property="fb:page_id"     content="333992367317" />
    <meta property="og:title"       content="IT-News, c&#39;t, iX, Technology ...
    <meta property="og:type"        content="website" />
    <meta property="og:locale"      content="de_DE" />
    <meta property="og:url"         content="https://www.heise.de/" />
    ...
```

# Kapitel WT:II

## II. Kommunikation und Protokolle für Web-Systeme

- Rechnernetze
- Prinzipien des Datenaustauschs
- Netzsoftware und Kommunikationsprotokolle
- Internetworking

- Client-Server-Interaktionsmodell
- Uniform Resource Locator

- Grundlagen HTTP-Protokoll
- Weitere HTTP-Konzepte
- Grundlagen TLS-Protokoll

- Zeichen und Codierung

# Weitere HTTP-Konzepte

## Session-Management

HTTP ist ein zustandsloses Protokoll = ein Protokoll „ohne Gedächtnis“.

Ein zustandsloses Protokoll berücksichtigt keine Information aus bereits stattgefundener Kommunikation.

Eine Session beschreibt einen Dialog, der sich über mehrere Anfrage/Antwort-Zyklen erstreckt. Innerhalb einer Session soll sich auf die vorangegangene Kommunikation bezogen werden können.

# Weitere HTTP-Konzepte

## Session-Management

Um eine Session aufrecht zu erhalten, muss der WWW-Client sich gegenüber dem Server bei jedem Request eindeutig identifizieren. Dies kann etwa über eine anfangs zufällig generierte Session-ID oder über eine Authentifizierung des Clients erfolgen (bspw. mittels Login-Daten oder Bearer-Token).

Techniken zur Codierung von Session-Information:

1. Cookies

   Session-Information wird beim Client gespeichert und im `Cookie`-Header mitgesendet.

2. Request-Authentisierung

   Client-Authentisierung wird bei jedem Request im `Authorization`-Header mitgesendet.

3. Hidden Fields

   Session-Information wird in unsichtbaren Formularfeldern untergebracht (oft unpraktikabel).

4. URL-Parameter

   Session-Information wird in URL codiert (gefährlich, da anfällig für Session-Hijacking).

# Weitere HTTP-Konzepte

## Session-Management

Um eine Session aufrecht zu erhalten, muss der WWW-Client sich gegenüber dem Server bei jedem Request eindeutig identifizieren. Dies kann etwa über eine anfangs zufällig generierte Session-ID oder über eine Authentifizierung des Clients erfolgen (bspw. mittels Login-Daten oder Bearer-Token).

Techniken zur Codierung von Session-Information:

1. Cookies

   Session-Information wird beim Client gespeichert und im `Cookie`-Header mitgesendet.

2. Request-Authentisierung

   Client-Authentisierung wird bei jedem Request im `Authorization`-Header mitgesendet.

3. Hidden Fields

   Session-Information wird in unsichtbaren Formularfeldern untergebracht (oft unpraktikabel).

4. URL-Parameter

   Session-Information wird in URL codiert (gefährlich, da anfällig für Session-Hijacking).

# Weitere HTTP-Konzepte

## Session-Management: Cookies

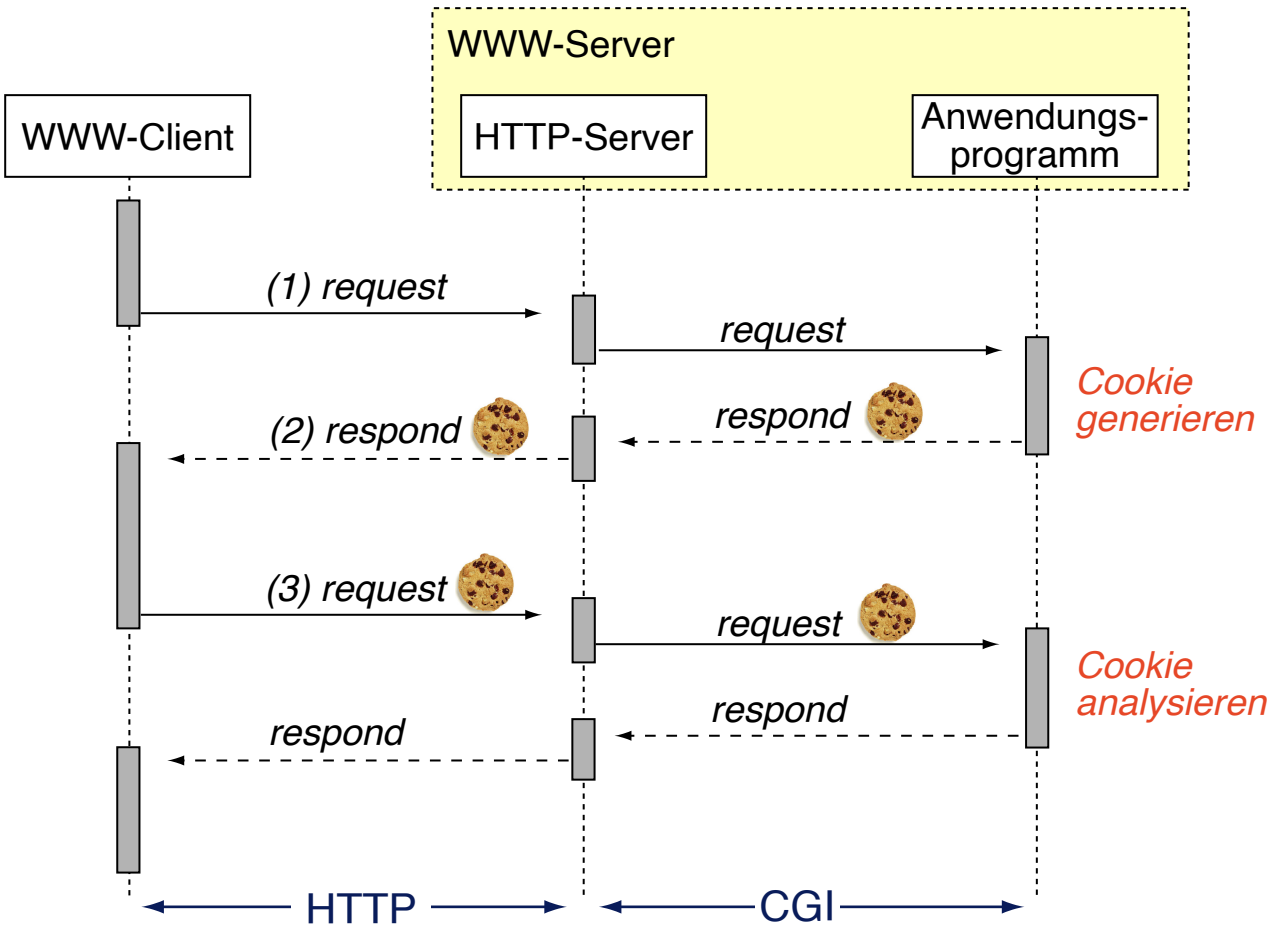
Ein Cookie (Keks) ist eine Zeichenkette (< 4KB), die zwischen WWW-Client und WWW-Server ausgetauscht wird. Standardisiert in RFC 6265.

Verwendungsmöglichkeiten von Cookies:

- Session-Management
- Benutzer-Authentisierung
- Seitenabfolgesteuerung
- Erstellung von Nutzerprofilen
- Generierung nutzerspezifischer Seiten
- Informationsaustausch ergänzend zum HTTP-Protokoll

# Weitere HTTP-Konzepte

## Session-Management: Cookies (Fortsetzung)



# Weitere HTTP-Konzepte

## Session-Management: Cookies (Fortsetzung)

1. WWW-Client fragt Google-Startseite an:

```
Ethernet II, Src: 00:0c:f1:e8:fe:be, Dst: 00:00:0c:07:ac:01
Internet Protocol, Src Addr: 141.54.178.123, Dst Addr: 66.249.85.99
Transmission Control Protocol, Src Port: 1577, Dst Port: http (80), ...
Hypertext Transfer Protocol
    GET / HTTP/2.0
    Accept: text/html,application/xhtml+xml,application/xml,q=0.9,*/*;...
    Accept-Language: de
    Accept-Encoding: gzip, deflate, br
    User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:66.0) ...
    Host: www.google.de
    Connection: Keep-Alive
```

# Weitere HTTP-Konzepte

## Session-Management: Cookies (Fortsetzung)

1. WWW-Client fragt Google-Startseite an:

```
Ethernet II, Src: 00:0c:f1:e8:fe:be, Dst: 00:00:0c:07:ac:01
Internet Protocol, Src Addr: 141.54.178.123, Dst Addr: 66.249.85.99
Transmission Control Protocol, Src Port: 1577, Dst Port: http (80), ...
Hypertext Transfer Protocol
    GET / HTTP/2.0
    Accept: text/html,application/xhtml+xml,application/xml,q=0.9,*/*;...
    Accept-Language: de
    Accept-Encoding: gzip, deflate, br
    User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:66.0) ...
    Host: www.google.de
    Connection: Keep-Alive
```

| | | |
|---|---|---|
| 5–7 | Anwendung: | HTTP-Message |
| 4 | Transport: | Port |
| 3 | Internet: | IP-Adresse |
| 1+2 | Host-zu-Netz: | Mac-Adresse |

# Weitere HTTP-Konzepte

## Session-Management: Cookies (Fortsetzung)

2. WWW-Server antwortet:

```
Ethernet II, Src: 00:09:e9:a6:0b:fc, Dst: 00:0c:f1:e8:fe:be
Internet Protocol, Src Addr: 66.249.85.99, Dst Addr: 141.54.178.123
Transmission Control Protocol, Src Port: http (80), Dst Port: 1577, ...
Hypertext Transfer Protocol
    HTTP/2.0 200 OK
    Cache-Control: private, max-age=0
    Content-Type: text/html; charset=UTF-8
    Set-Cookie: NID=181=qUoo...9Ya8; domain=.google.de; HttpOnly
    Content-Encoding: gzip
    Content-Length: 57433
    Date: Tue, 16 Apr 2019 10:23:32 GMT
    ...

    Content-encoded entity body (gzip)
    Line-based text data: text/html
    <html><head><meta http-equiv=content-typecontent=text/html; ...
```

| | | |
|---|---|---|
| 5–7 | Anwendung: | HTTP-Message |
| 4 | Transport: | Port |
| 3 | Internet: | IP-Adresse |
| 1+2 | Host-zu-Netz: | Mac-Adresse |

# Weitere HTTP-Konzepte

## Session-Management: Cookies (Fortsetzung)

3. WWW-Client sendet Google-Query „test“:

```
Ethernet II, Src: 00:0c:f1:e8:fe:be, Dst: 00:00:0c:07:ac:01
Internet Protocol, Src Addr: 141.54.178.123, Dst Addr: 66.249.85.99
Transmission Control Protocol, Src Port: 1577, Dst Port: http (80), ...
```

```
Hypertext Transfer Protocol
    GET /search?...&q=test&...
    Referer: https://www.google.de/
    Accept: text/html,application/xhtml+xml,application/xml,q=0.9,*/*;...
    Accept-Language: de
    Accept-Encoding: gzip, deflate, br
    User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:66.0) ...
    Host: www.google.de
    Connection: Keep-Alive
    Cookie: NID=181=qUoo...9Ya8; ...
```

# Weitere HTTP-Konzepte

## Session-Management: Cookies (Fortsetzung)

3. WWW-Client sendet Google-Query „test“:

```
Ethernet II, Src: 00:0c:f1:e8:fe:be, Dst: 00:00:0c:07:ac:01
Internet Protocol, Src Addr: 141.54.178.123, Dst Addr: 66.249.85.99
Transmission Control Protocol, Src Port: 1577, Dst Port: http (80), ...
```

```
Hypertext Transfer Protocol
    GET /search?...&q=test&...
    Referer: https://www.google.de/
    Accept: text/html,application/xhtml+xml,application/xml,q=0.9,*/*;...
    Accept-Language: de
    Accept-Encoding: gzip, deflate, br
    User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:66.0) ...
    Host: www.google.de
    Connection: Keep-Alive
    Cookie: NID=181=qUoo...9Ya8; ...
```

Auf Client-Seite gespeicherter Cookie:

```
NID
181=qUoo...9Ya8
```

Bemerkungen:

- Nicht der Web-Server, sondern ein Anwendungsprogramm ist am Setzen von Cookies interessiert.
- Der Client kann Cookies mit einer Lebensdauer versehen und spezifisch für Web-Seiten erlauben oder sperren.
- Der Client wählt das passende Cookie anhand der URL der Informationsquelle aus und schickt es mit der Anfrage zum Web-Server. Dabei darf der Client nur Cookies an denjenigen Web-Server senden, von dem diese Cookies stammen.
- Der Client bringt Cookie-Information im Request-Header `Cookie` unter.
- Der Server bringt Cookie-Information im Reponse-Header `Set-Cookie` unter.
- Browser speichern Cookies im Home-Bereich des Nutzers; für Windows siehe hier.
- Analyse des Protokollstapels mit dem Programm wireshark.

# Weitere HTTP-Konzepte

## Content-Negotiation

Ressourcen auf dem WWW können in sprachspezifischen, qualitätspezifischen oder codierungsspezifischen Varianten vorliegen, besitzen jedoch dieselbe URI.

Seit HTTP/1.1 können WWW-Client und WWW-Server aushandeln, welche der angebotenen Varianten einer Informationsressource geliefert werden soll.

Arten der Content-Negotiation:

1. Server-driven. WWW-Server verantwortlich für Auswahl.
   Kriterien: Header des HTTP-Requests, Informationen über die Ressourcen.

2. Agent-driven. WWW-Client verantwortlich für Auswahl.
   Client erfragt Liste verfügbarer Varianten, trifft dann Auswahlentscheidung.

3. Transparent. Proxy-Server verhandelt in der Agenten-Rolle.
   Vorteile: Lastverteilung, WWW-Client stellt nur eine Anfrage.

# Weitere HTTP-Konzepte

## Content-Negotiation

Ressourcen auf dem WWW können in sprachspezifischen, qualitätspezifischen oder codierungsspezifischen Varianten vorliegen, besitzen jedoch dieselbe URI.

Seit HTTP/1.1 können WWW-Client und WWW-Server aushandeln, welche der angebotenen Varianten einer Informationsressource geliefert werden soll.

Arten der Content-Negotiation:

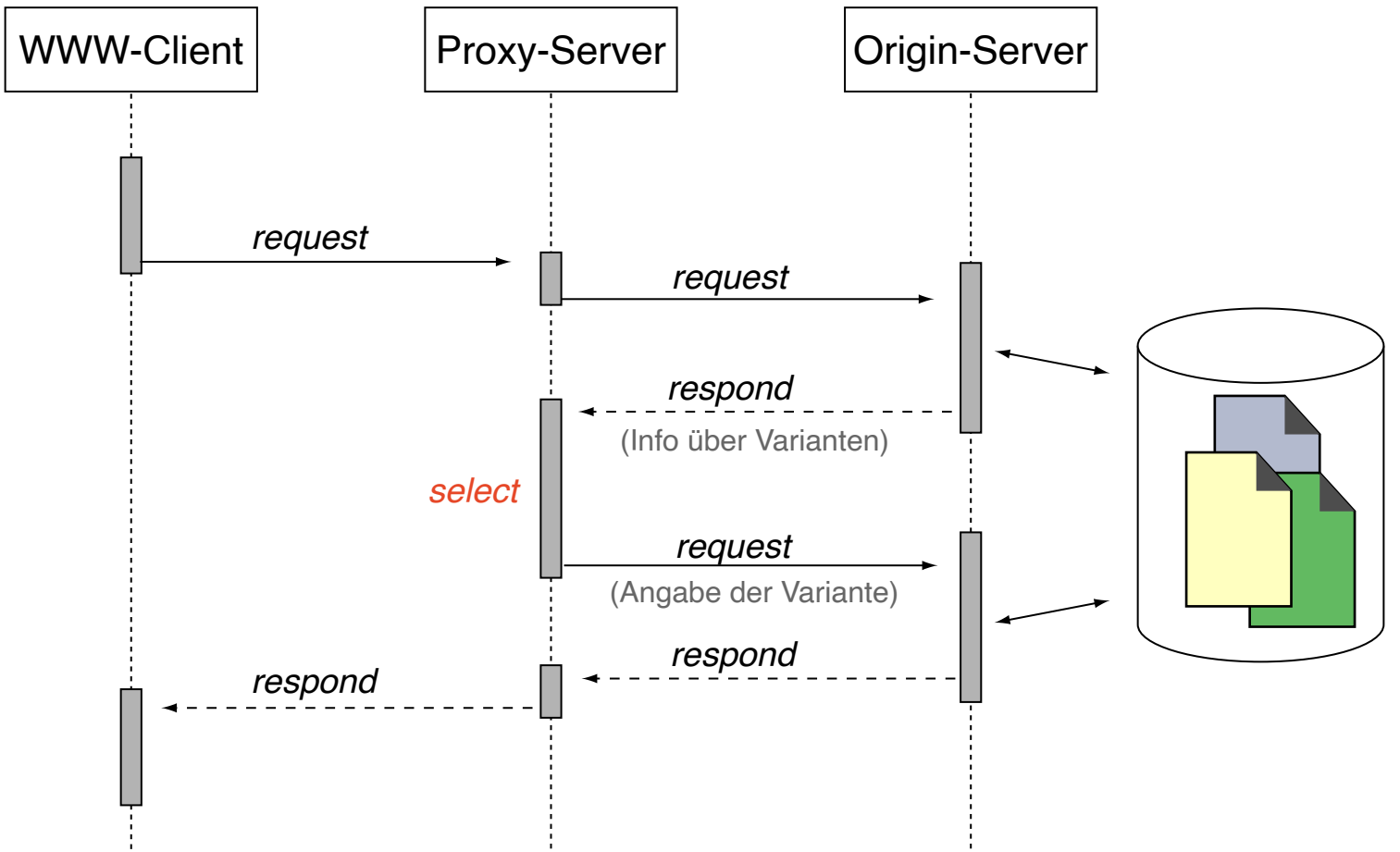
1. Server-driven. WWW-Server verantwortlich für Auswahl.
   Kriterien: Header des HTTP-Requests, Informationen über die Ressourcen.

2. Agent-driven. WWW-Client verantwortlich für Auswahl.
   Client erfragt Liste verfügbarer Varianten, trifft dann Auswahlentscheidung.

3. Transparent. Proxy-Server verhandelt in der Agenten-Rolle.
   Vorteile: Lastverteilung, WWW-Client stellt nur eine Anfrage.

# Weitere HTTP-Konzepte

## Content-Negotiation: transparent



Bemerkungen:

- Die drei Arten der Content-Negotiation unterscheiden sich dahingehend, wer die Auswahloperation ausführt.
- Vorteil der Server-driven Content-Negotiation: der Server ist nah an den Ressourcen und hat den besten Überblick.
  Nachteil: der Server ist auf Information aus den Request-Headern des Clients angewiesen (primär aus den standardisierten `Accept-*`-Headern).
- Vorteil der Agent-driven Content-Negotiation: der Client weiß genau, was er will.
  Nachteil: Overhead an Kommunikation; erfordert einen zusätzlichen Roundtrip, da der Client erst die verfügbaren Varianten erfragen muss.
- In der Praxis ist Server-driven Content-Negotiation die einfachste und mit Abstand üblichste Form. Für Agent-driven Content-Negotiation wurden zwar die HTTP-Statuscodes 300 (Multiple Choices) und 406 (Not Acceptable) reserviert, ein konkretes Format zur Auflistung der Varianten und ihrer jeweiligen URIs wurde hingegen nie spezifiziert.

# Weitere HTTP-Konzepte

## Optimierte Ausnutzung der Verbindung

HTTP/1.0 öffnet für jede Anfrage eine Verbindung, die nach jeder Antwort unmittelbar geschlossen wird:

- \+ Protokoll sehr einfach, leicht zu implementieren
- \+ Verbindungsabbruch signalisiert auch Abschluss der HTTP-Antwort
- – Verbindungsaufbau zeit- und ressourcenintensiv

HTTP/1.1 hat persistente Verbindungen, die Client-seitig mit `Connection: close` (General-Header) oder nach einem Timeout geschlossen werden:

- \+ effizientere Nutzung von Betriebssystemressourcen, weniger Pakete
- \+ Pipelining: Versenden einer weiteren Anfrage ohne auf Antwort zu warten
- – aufwändigeres Protokoll, da “Chunked Encoding” notwendig

HTTP/2 und HTTP/3 besitzen Multiplexing und andere Verbesserungen.

[HTTP/2: Home, key differences] [HTTP/3: key differences]

# Weitere HTTP-Konzepte

## Optimierte Ausnutzung der Verbindung

HTTP/1.0 öffnet für jede Anfrage eine Verbindung, die nach jeder Antwort unmittelbar geschlossen wird:

- \+ Protokoll sehr einfach, leicht zu implementieren
- \+ Verbindungsabbruch signalisiert auch Abschluss der HTTP-Antwort
- – Verbindungsaufbau zeit- und ressourcenintensiv

HTTP/1.1 hat persistente Verbindungen, die Client-seitig mit `Connection: close` (General-Header) oder nach einem Timeout geschlossen werden:

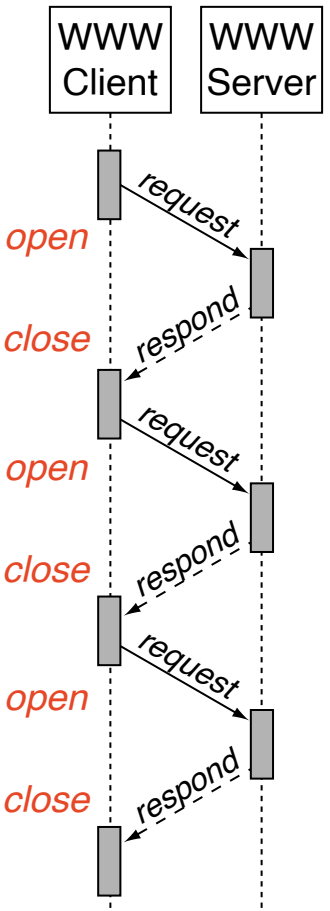
- \+ effizientere Nutzung von Betriebssystemressourcen, weniger Pakete
- \+ Pipelining: Versenden einer weiteren Anfrage ohne auf Antwort zu warten
- – aufwändigeres Protokoll, da “Chunked Encoding” notwendig

HTTP/2 und HTTP/3 besitzen Multiplexing und andere Verbesserungen.

[HTTP/2: Home, key differences] [HTTP/3: key differences]

# Weitere HTTP-Konzepte

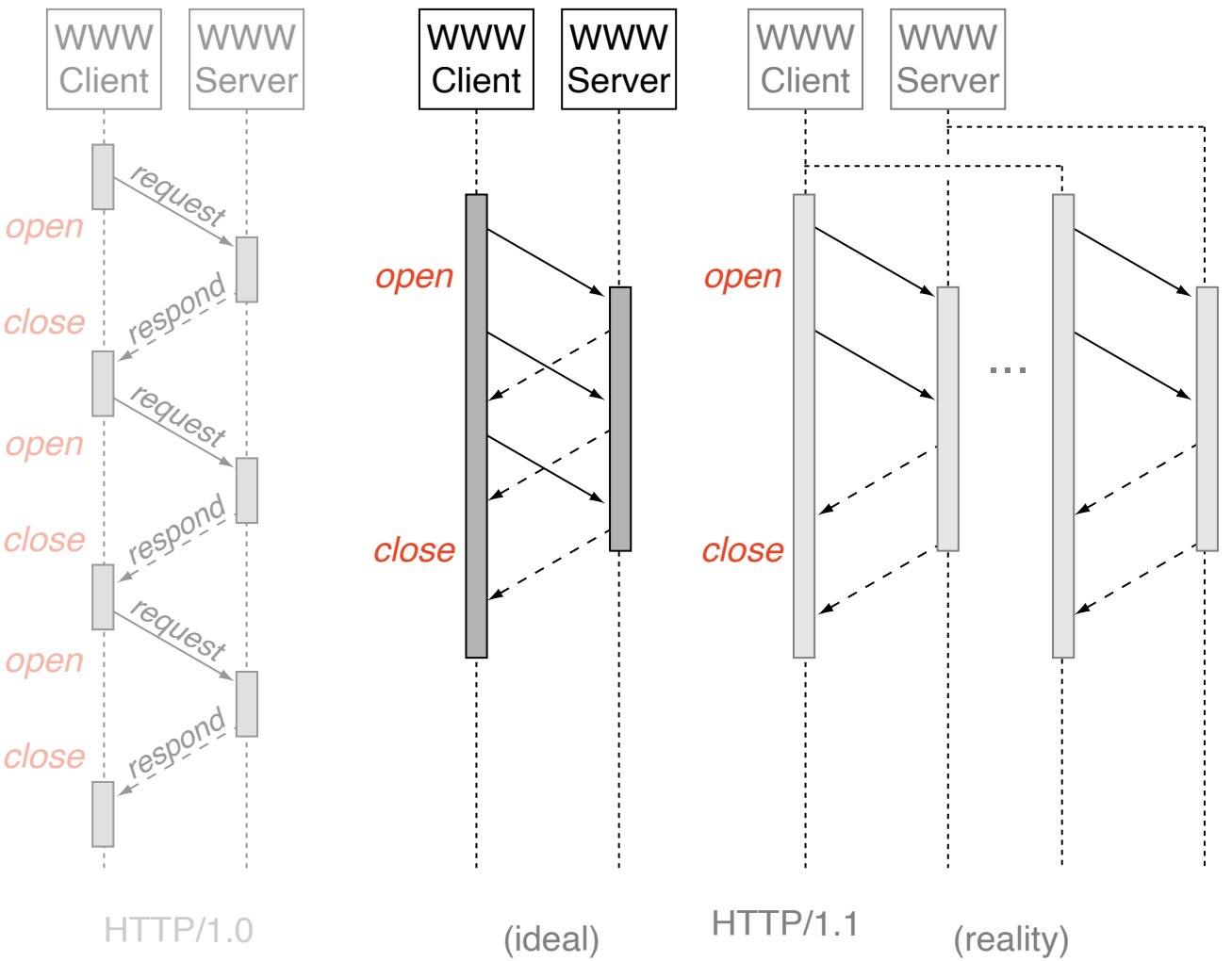
## Optimierte Ausnutzung der Verbindung: Illustration




HTTP/1.0

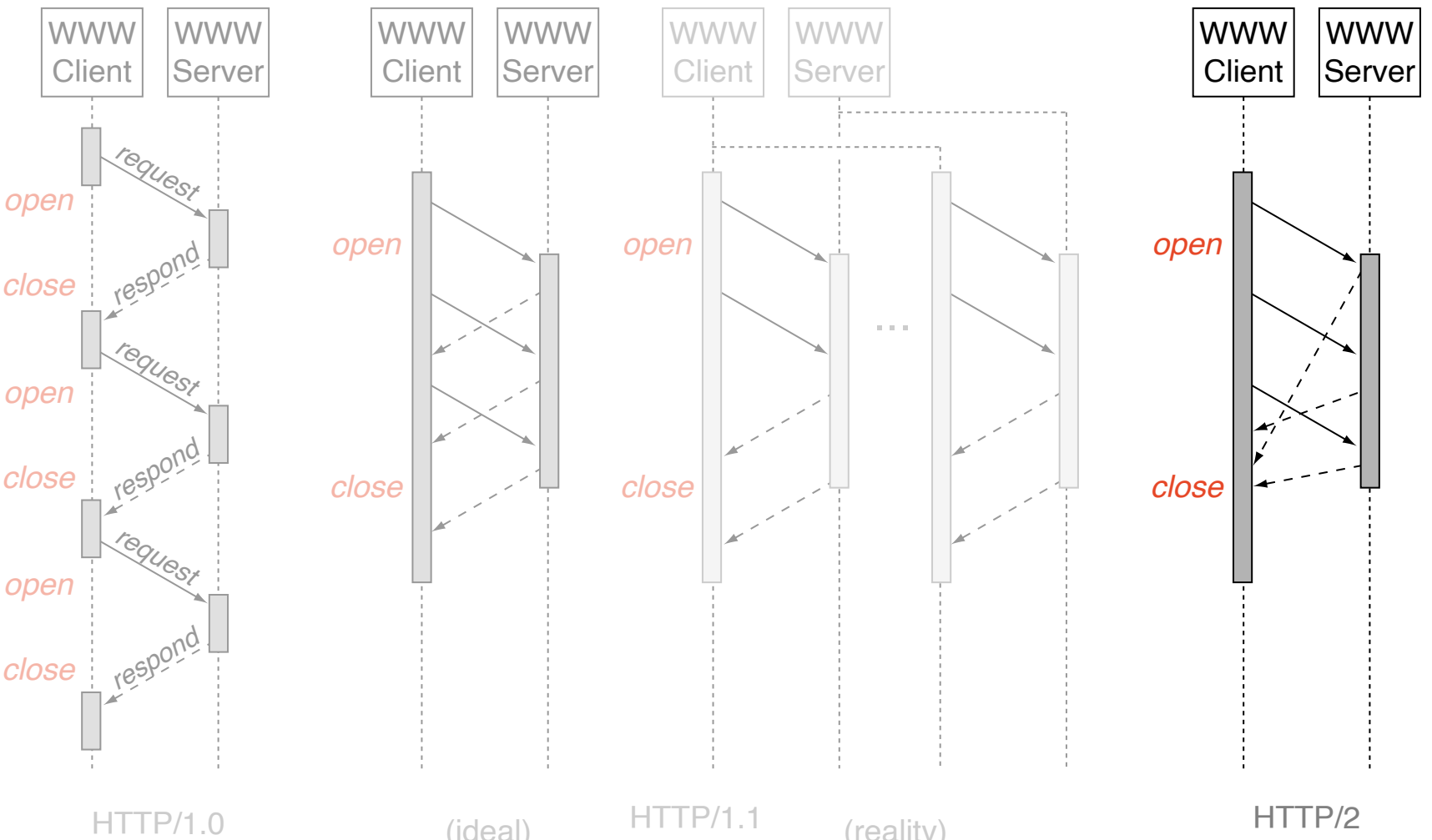
# Weitere HTTP-Konzepte

## Optimierte Ausnutzung der Verbindung: Illustration



# Weitere HTTP-Konzepte

## Optimierte Ausnutzung der Verbindung: Illustration



# Kapitel WT:II

## II. Kommunikation und Protokolle für Web-Systeme

- Rechnernetze
- Prinzipien des Datenaustauschs
- Netzsoftware und Kommunikationsprotokolle
- Internetworking

- Client-Server-Interaktionsmodell
- Uniform Resource Locator

- Grundlagen HTTP-Protokoll
- Weitere HTTP-Konzepte
- Grundlagen TLS-Protokoll

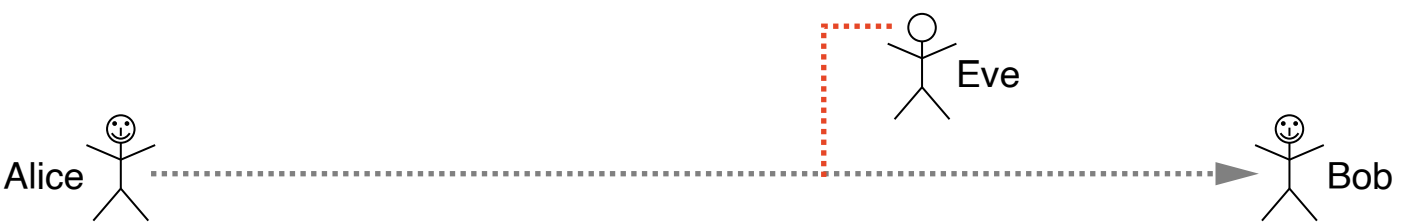
- Zeichen und Codierung

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie

Bei der Übertragung einer Nachricht kann „viel passieren“; sie kann abgehört, abgefangen, umgeleitet oder ausgetauscht werden.

Standardszenario der Kryptografie:




Wichtige Aspekte in diesem Zusammenhang:

- Vertraulichkeit: Geheimhaltung des Inhalts
- Integrität: Aufdeckung von Inhaltsveränderungen
- Authentizität: Garantie der Urheberschaft

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Sei $P$ die Menge aller Texte (*plain texts*), $K$ die Menge aller Schlüssel (*keys*), $C$ die Menge aller verschlüsselten Texte (*cipher texts*) und $e_k, d_k$ zwei Funktionen:

$$e_k : P \to C$$
$$d_k : C \to P$$

mit $\quad d_k(e_k(x)) = x, \quad x \in P,\ k \in K$

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Sei $P$ die Menge aller Texte (*plain texts*), $K$ die Menge aller Schlüssel (*keys*), $C$ die Menge aller verschlüsselten Texte (*cipher texts*) und $e_k, d_k$ zwei Funktionen:

$$e_k : P \to C$$
$$d_k : C \to P$$

mit $\quad d_k(e_k(x)) = x, \quad x \in P,\ k \in K$

Schritte bei einer symmetrischen Verschlüsselung:

1. Alice und Bob wählen einen gemeinsamen Schlüssel $k \in K$.
2. Alice versendet Nachricht $x$ als $y = e_k(x)$ an Bob.
3. Bob entschlüsselt $y$ und erhält $x = d_k(y)$.

„Einziges“ Problem: die Übermittlung des Schlüssels $k$.

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Sei $P$ die Menge aller Texte (*plain texts*), $K$ die Menge aller Schlüssel (*keys*), $C$ die Menge aller verschlüsselten Texte (*cipher texts*) und $e_k, d_k$ zwei Funktionen:

$$e_k : P \to C$$
$$d_k : C \to P$$

mit $\quad d_k(e_k(x)) = x, \quad x \in P,\ k \in K$

Idee der asymmetrischen Public-Key-Kryptografie: Alice und Bob haben je zwei Schlüssel $k_{\text{pub}}$ (öffentlich) und $k_{\text{priv}}$ (privat) mit $d_{k_{\text{pub}}}(e_{k_{\text{priv}}}(x)) = d_{k_{\text{priv}}}(e_{k_{\text{pub}}}(x)) = x$.

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Sei $P$ die Menge aller Texte (*plain texts*), $K$ die Menge aller Schlüssel (*keys*), $C$ die Menge aller verschlüsselten Texte (*cipher texts*) und $e_k, d_k$ zwei Funktionen:

$$e_k : P \to C \qquad d_k : C \to P \qquad \text{mit} \quad d_k(e_k(x)) = x, \quad x \in P,\ k \in K$$

Idee der asymmetrischen Public-Key-Kryptografie: Alice und Bob haben je zwei Schlüssel $k_{\text{pub}}$ (öffentlich) und $k_{\text{priv}}$ (privat) mit $d_{k_{\text{pub}}}(e_{k_{\text{priv}}}(x)) = d_{k_{\text{priv}}}(e_{k_{\text{pub}}}(x)) = x$.

Schritte bei einer asymmetrischen Verschlüsselung:

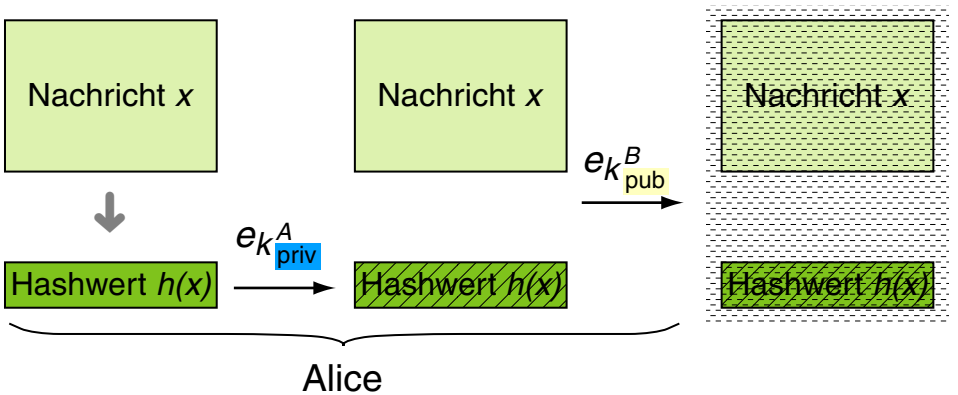
1. Alice und Bob wählen jeweils für sich die Schlüssel $k^A_{\text{pub}}, k^A_{\text{priv}}$ und $k^B_{\text{pub}}, k^B_{\text{priv}}$.
2. Beide veröffentlichen ihren Schlüssel $k_{\text{pub}}$.
3. Alice versendet Nachricht $x$ als $y = e_{k^B_{\text{pub}}}(x)$ an Bob.
4. Bob entschlüsselt $y$ und erhält $x = d_{k^B_{\text{priv}}}(y)$.

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Q1: Woher weiß Bob, dass Alice und nicht Eve der Autor von Nachricht $x$ ist?

Sei $h : P \to N$ eine Hashfunktion, die von einer Nachricht $x$ eine eindeutige Charakterisierung $h(x)$ fester Länge (*message digest*) berechnet.

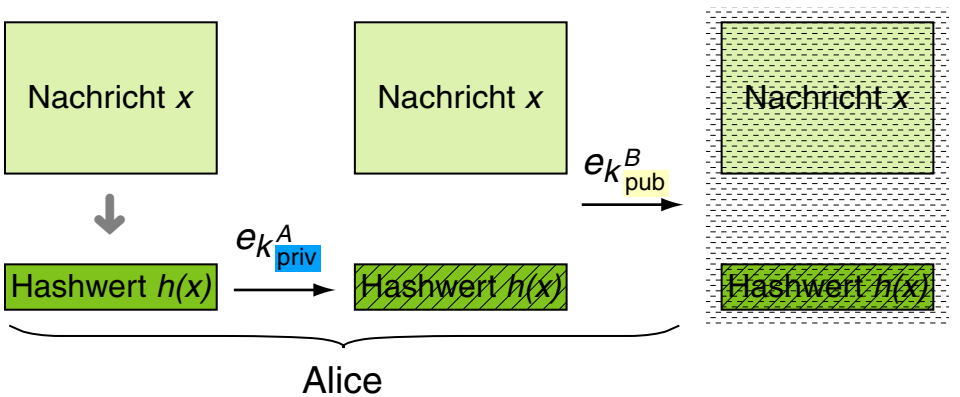


# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Q1: Woher weiß Bob, dass Alice und nicht Eve der Autor von Nachricht $x$ ist?

Sei $h : P \to N$ eine Hashfunktion, die von einer Nachricht $x$ eine eindeutige Charakterisierung $h(x)$ fester Länge (*message digest*) berechnet.




Schritte beim digitalen Signieren:

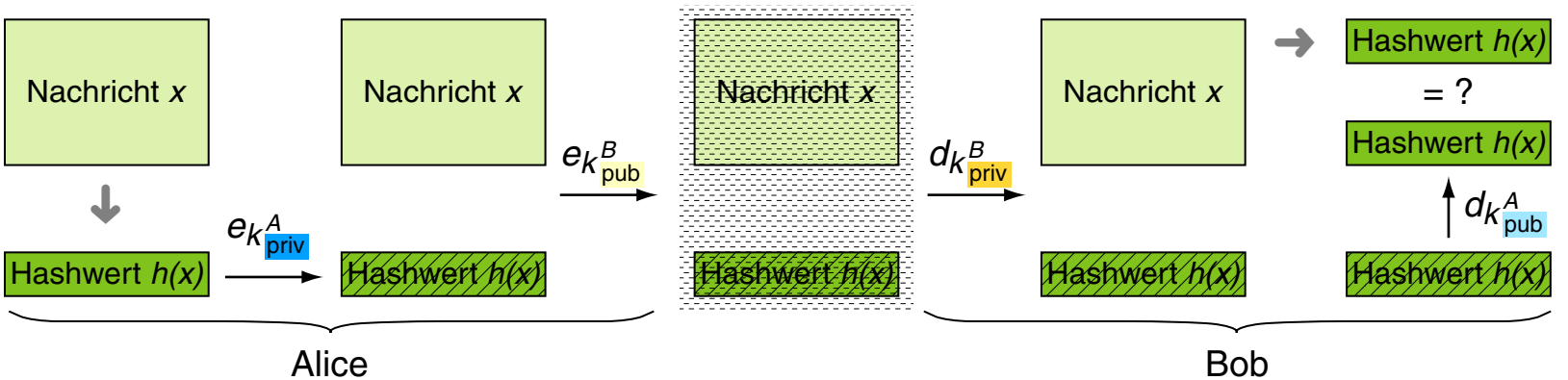
1. Alice berechnet für Nachricht $x$ den Hashwert $h(x)$.
2. Alice verschlüsselt $h(x)$ als $y_h = e_{k^A_{priv}}(h(x))$.
3. Alice versendet Nachricht $x + y_h$ als $e_{k^B_{pub}}(x + y_h)$ an Bob.

# Grundlagen TLS-Protokoll

## Konzepte der Kryptografie (Fortsetzung)

Q1: Woher weiß Bob, dass Alice und nicht Eve der Autor von Nachricht $x$ ist?

Sei $h : P \to N$ eine Hashfunktion, die von einer Nachricht $x$ eine eindeutige Charakterisierung $h(x)$ fester Länge (*message digest*) berechnet.




Schritte beim digitalen Verifizieren:

1. Bob entschlüsselt mittels $d_{k^B_{priv}}$ die Nachricht und erhält $x + y_h$.
2. Bob berechnet für Nachricht $x$ den Hashwert $h(x)$.
3. Bob berechnet $d_{k^A_{pub}}(y_h)$ und vergleicht den Wert mit $h(x)$.

Bemerkungen:

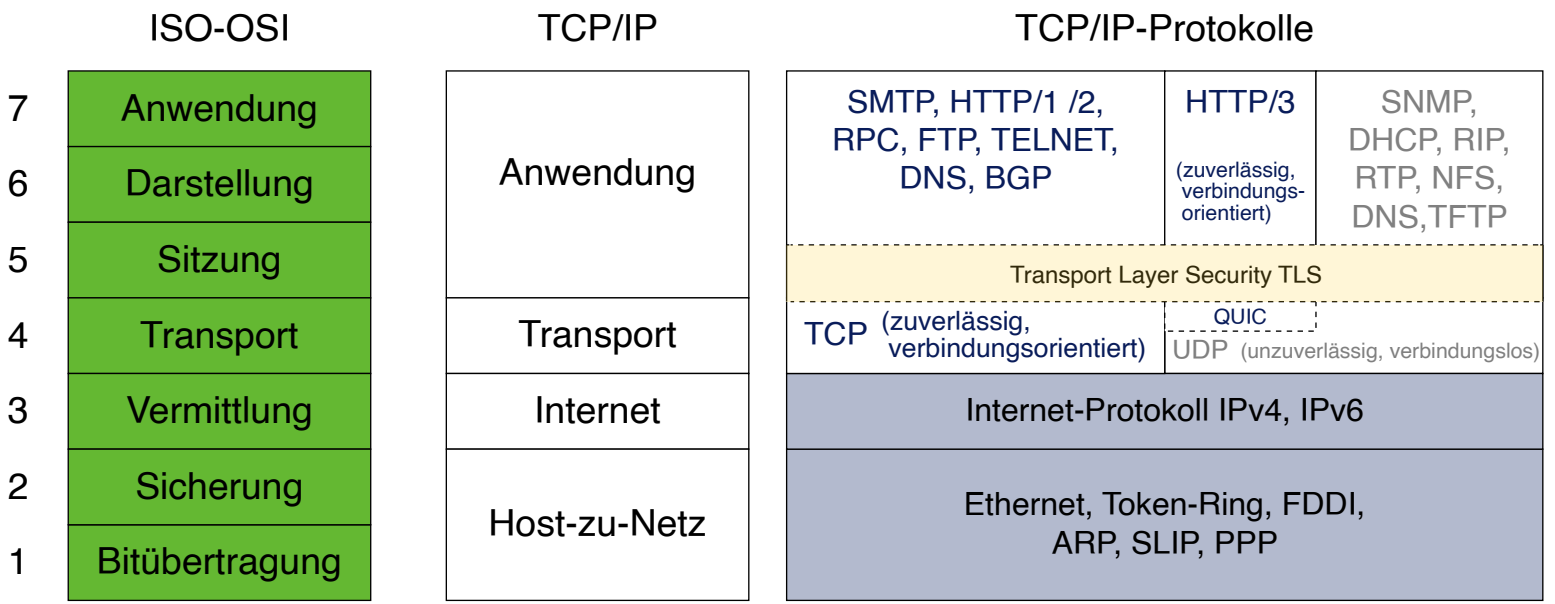
- Der Vergleich von digitalen Signaturen mit realen Unterschriften ist gerechtfertigt, da Bob durch Abgleich der Signatur von Alice ihre Identität zu verifizieren sucht. Ein entscheidender Punkt bei digitalen Signaturen ist die Verhinderung der Trennung von Nachricht und Signatur mittels der Funktion $h()$.
- Q2: Woher weiß Alice, welches der öffentliche Schlüssel von Bob ist?

  Falls Alice nicht den öffentlichen Schlüssel von Bob kennt (Normalfall), muss Alice sicherstellen, dass es sich bei $k^B_{\text{pub}}$ tatsächlich um den öffentlichen Schlüssel von Bob handelt. Erst dann weiß sie, dass $k^B_{\text{pub}}(x)$ nur durch Bob entschlüsselt werden kann, weil nur er in Besitz des passenden privaten Schlüssels $k^B_{\text{priv}}$ ist.

  Dieses Problem wird mittels Zertifikaten und einer Public-Key-Infrastruktur (PKI) addressiert.

# Grundlagen TLS-Protokoll

## Einordnung im Protokollstapel [HTTP]




Grundsätzlich liegt TLS zwischen der Transport- (4) und der Anwendungsebene (5); die Abgrenzung zu diesen Ebenen verschwimmt jedoch teilweise. Insbesondere seit TLS 1.3 gibt es aus Effizienzgründen eine stärkere Vermischung mit Aufgaben des TCP-Protokolls.

# Grundlagen TLS-Protokoll

## Historie

1994 SSL 1.0 wegen schwerer Sicherheitsmängel nicht veröffentlicht.

1995-1996 SSL 2.0 und SSL 3.0 offenbaren im Laufe der Zeit verschiedene Angriffsmöglichkeiten.
Seit März 2012 bzw. Juni 2015 deprecated. [Wikipedia]

1999-2006 TLS 1.0 wird Nachfolger von SSL 3.0. TLS 1.1 mit verbessertem Schutz gegen Cipher-Block-Chaining-Angriffe.
Seit März 2021 deprecated. [BEAST] [Wikipedia]

2008 TLS 1.2 ersetzt MD5 / SHA-1 durch SHA-256 und bringt weitere Verbesserungen. [RFC 5246] [Wikipedia]

2018 TLS 1.3 mit effizienterem Handshake, simplifizierter Cipher-Suite-Aushandlung, sowie Entfernung alter, unsicherer und Einführung neuer, stärkerer Algorithmen. [RFC 8446] [Wikipedia]

# Grundlagen TLS-Protokoll

## Übersicht

Das TLS-Protokoll dient zur Verschlüsselung und sicheren Datenübertragung im Internet und wird vor allem mit HTTPS eingesetzt. Eine TLS-Session besteht aus:

1. Einem Handshake, der für den Schlüsselaustausch und zur Authentisierung der Kommunikationspartner (aber insbesondere des Servers) dient.
2. Einer zustandsbehafteten Verbindung, die mithilfe des im Handshake ausgehandelten symmetrischen Sitzungsschlüssels abgesichert ist.

# Grundlagen TLS-Protokoll

## Übersicht (Fortsetzung)

Das TLS-Protokoll dient zur Verschlüsselung und sicheren Datenübertragung im Internet und wird vor allem mit HTTPS eingesetzt. Eine TLS-Session besteht aus:

1. Einem Handshake, der für den Schlüsselaustausch und zur Authentisierung der Kommunikationspartner (aber insbesondere des Servers) dient.
2. Einer zustandsbehafteten Verbindung, die mithilfe des im Handshake ausgehandelten symmetrischen Sitzungsschlüssels abgesichert ist.

Phasen des TLS-Handshake-Protokolls:

(a) Abstimmung bzgl. TLS-Version und Cipher-Suite.

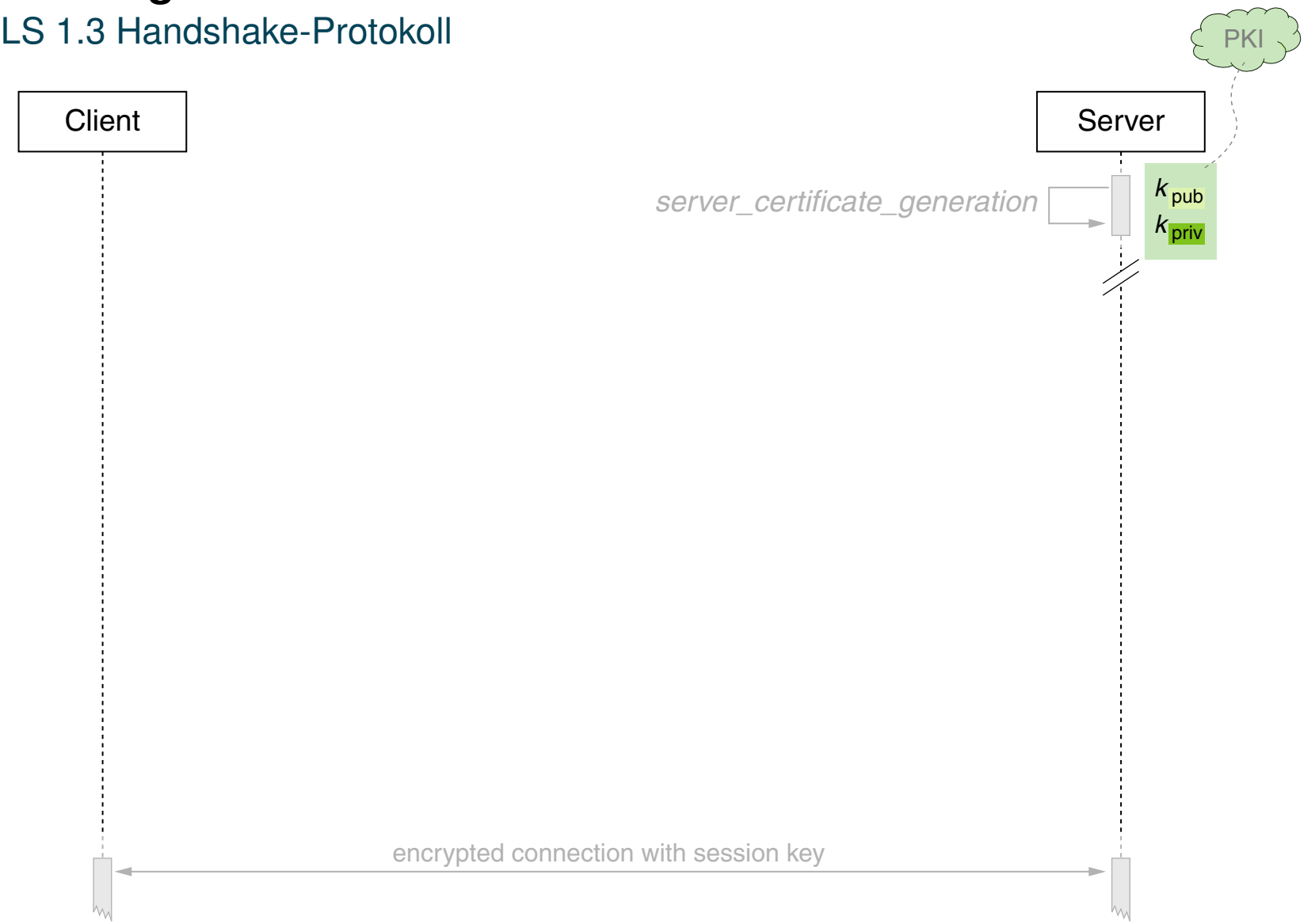
(b) Server identifiziert sich gegenüber Client mittels X.509-Zertifikat und zeigt, dass er einen Private-Key passend zum Public-Key des Zertifikats besitzt.

(c) Berechnung des Pre-Master-Secrets via Diffie-Hellman-Schlüsselaustausch (alternativ: Client generiert Zufallsnachricht und sendet sie via RSA-Schlüsselaustausch).

(d) Generierung der Sitzungsschlüssel aus dem Pre-Master-Secret.

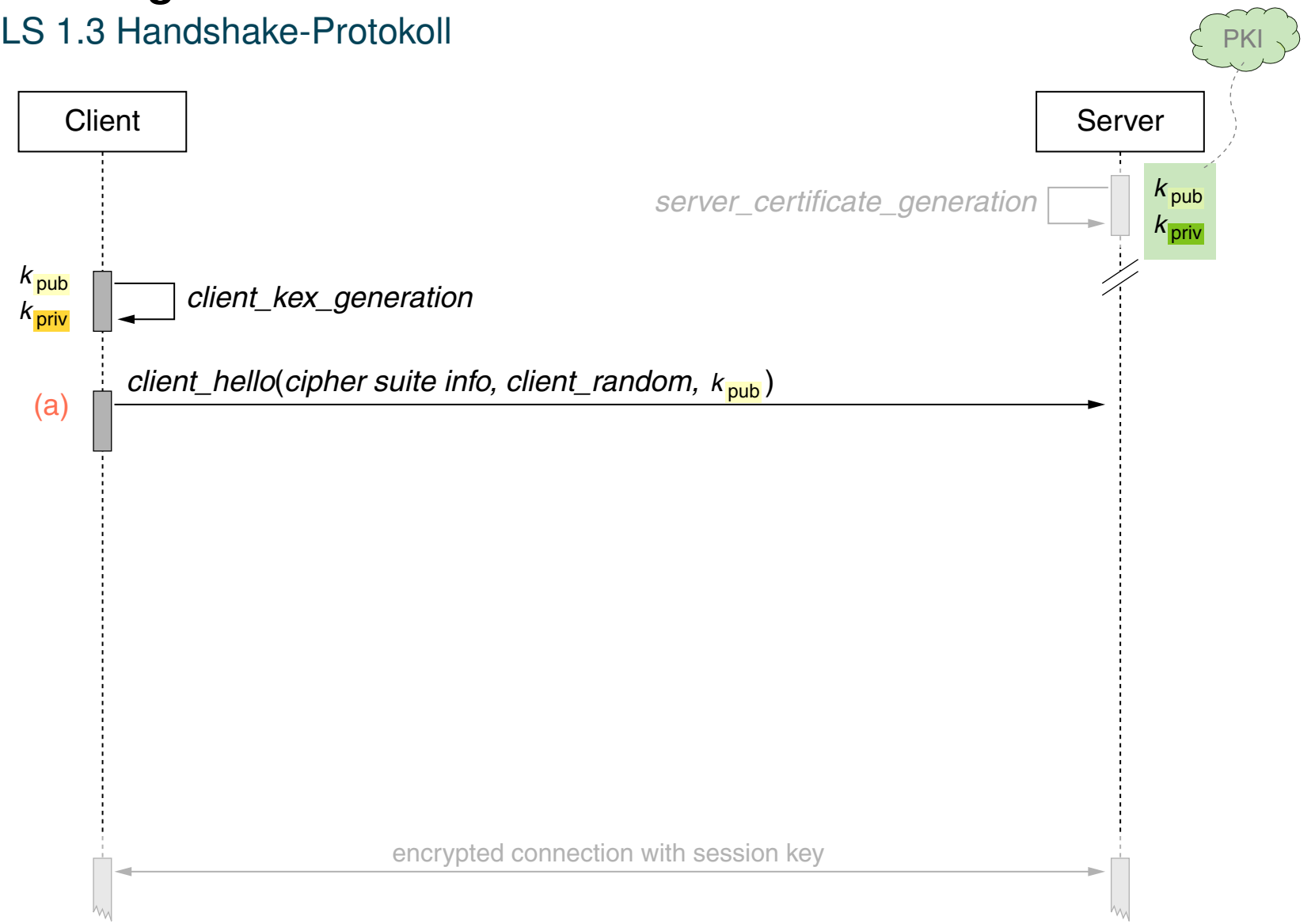
# Grundlagen TLS-Protokoll

## TLS 1.3 Handshake-Protokoll



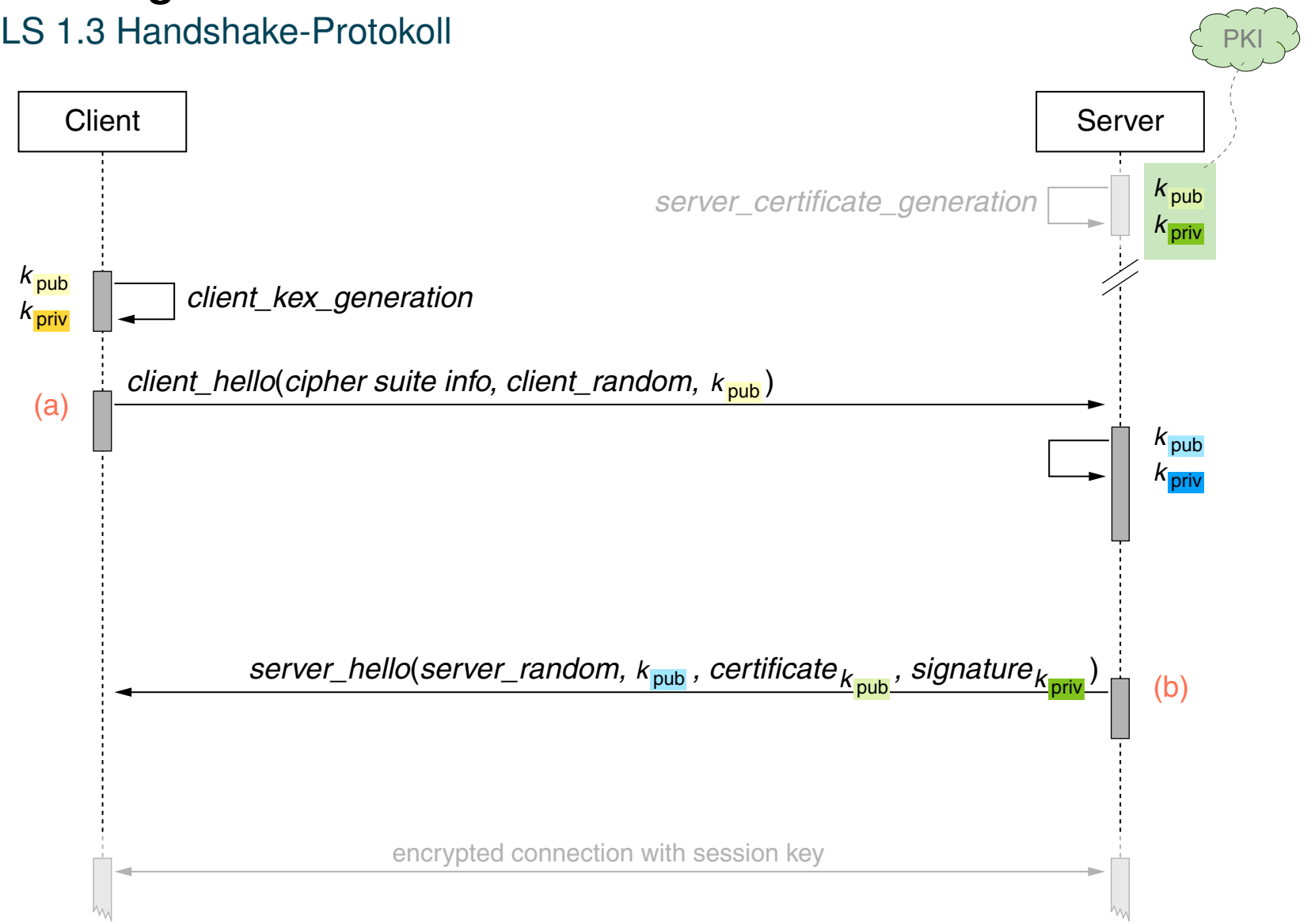
# Grundlagen TLS-Protokoll

## TLS 1.3 Handshake-Protokoll



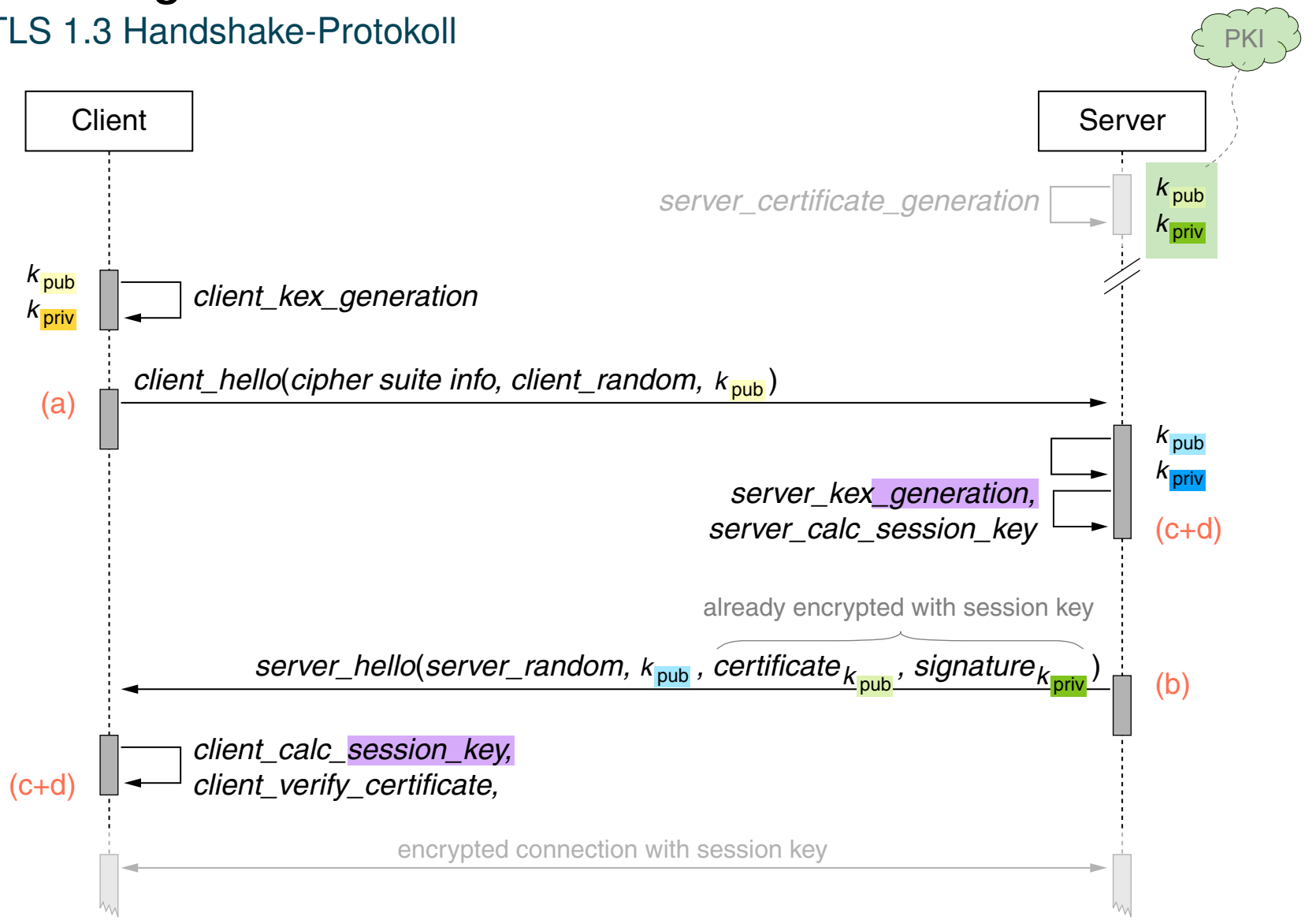
# Grundlagen TLS-Protokoll

## TLS 1.3 Handshake-Protokoll



# Grundlagen TLS-Protokoll

## TLS 1.3 Handshake-Protokoll



Bemerkungen:

- kex = key exchange
- Der Schlüsselaustausch (kex) verwendet statt des zum Serverzertifikat gehörigen (statischen) privaten Schlüssels temporäre (ephemeral) Schlüssel. Der statische private Schlüssel dient nur zur Signatur des Austauschs. Hierdurch wird im Falle eines kompromittierten Serverschlüssels die Vertraulichkeit vergangener Kommunikation gewährleistet. [perfect forward secrecy]
- Zusammenfassung des Handshakes und Unterschiede zwischen TLS 1.2 und 1.3. [Cloudflare 1, Cloudflare 2]

Bemerkungen: (Fortsetzung)

- Im TLS-Protokoll definiert die Cipher-Suite eine Kombination aus bis zu vier Algorithmen, die zum Aufbau der gesicherten Datenverbindung verwendet werden sollen:
  1. Schlüsselaustausch (RSA, ECDHE, DHE, PSK)
  2. Authentisierung / Signatur (RSA, ECDSA, DSA)
  3. Block- oder Stream-Chiffre (AES, ChaCha20, 3DES, DES, RC4, IDEA)
  4. Hash zur Message Authentication (MAC) (SHA-2, Poly1305, SHA-1, MD5)
- Jede Cipher-Suite hat einen Namen, der die enthaltenen Algorithmen angibt, etwa `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`. Seit TLS 1.3 fällt die Angabe für den Schlüsselaustausch weg: `TLS_AES_128_GCM_SHA256`.
- Die Angabe der Hashfunktion bezieht sich auf den HMAC, der (wenn nicht durch den Cipher-Mode selbst bereitgestellt) zur Authentisierung des Ciphertextes und (seit TLS 1.2) zur Ableitung des Master-Secrets dient. Der Message-Digest in der Zertifikatssignatur ist davon unabhängig.