openHPI Web Technologies 2015
Week 2

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

openHPI

URI – Uniform Resource Identifier

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Introduction: URI (1/3)

Basic components of the **WWW** – World Wide Web – are **hypermedia documents** with their links - the **hyperlinks**

- Hypermedia documents are distributed worldwide. In order to localize them and link them correctly, a **unique identification scheme** is required
- In daily life, two identification models have proven themselves:
  - Identification via **name** – valid for life but often not unique
  - Identification via **address** – uniquely determined but subject to occasional change. Hierarchical structure:
    - country, city, street, number, housing unit –

Introduction: URI (2/3)

Identification scheme in the WWW: **URI** - **Uniform Resource Identifier**

- There are two types of URIs:
  - **Uniform Resource Name** – **URN**:
    - Based on the name of resources
  - **Uniform Resource Locator** – **URL**:
    - Based on the address information of resources
- In practical application in the WWW, up to now only URLs have been established
  - **Advantage**: Unique identification of resources is easily possible
  - **Disadvantage**: Address changes cannot be carried out automatically
- Implementation of URN has failed until now due to lack of a universal name service

Introduction: URI (3/3)

Users can access a resource in the WWW via the browser by

- typing the URI (most of the time URL) of the resource
- "following" a hyperlink – by activating a special, designated element (text, image, video …) in the displayed WWW document, where an attached URL leads to the new hyperlinked document
- accessing the history directory of the browser, or
- accessing the bookmark directory of the browser
- …

URI – Uniform Resource Identifier (1/2)

Requirements for Uniform Resource Identifiers

- **Universality** – via the URI it should be possible to address every available resource in the Internet, regardless of its particular information service
- **Uniqueness** – every resource must be uniquely identifiable worldwide
- **Extensibility** – also new resources that have not been offered until now should be suitable to be equipped with an identifier
- **Fixability** – a URI should not only be interchangeable by electronic means, manual editing should also be possible, as well as printing

URI – Uniform Resource Identifier (2/2)

URI syntax

- URI are standardized by IETF and W3C in RFC 1630
- Syntax of a URI: **prefix : suffix**

- Besides complete URIs, RFC 1630 also specifies **relative URIs**
- According to RFC 1630, URI is either a
  - **URL** – specifies locality of a resource – or a
  - **URN** – specifies name of a resource

openHPI

URL and URN

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

URL – Uniform Resource Locator (1/3)

URL specifies the exact address of the place where a specific Web resource is located at a certain point in time

- URL syntax (RFC 1738, 1808) follows URI syntax: **prefix : suffix**
  - Prefix – **scheme**
  - Suffix – **scheme-specific** part and parameter

Possible schemes (1/2):

- **ftp**, e.g. ftp://ftp.hpi.uni-potsdam.de
- **http**, e.g. http://localhost:8080/Conference-Portal
- **https**, e.g. https://tele-lab.org
- **rtsp**, e.g. rtsp://webradio.com/stream (Real-Time Streaming Protocol)
- …

URL – Uniform Resource Locator (2/3)

More possible schemes (2/2):

- …
- **mailto** – Electronic mail address
- **news**
- **nntp** – Usenet news via NNTP access
- **telnet**
- **magnet** – Resources in peer-to-peer networks
- **file** – Host-specific file names

The growing number of services makes a permanent adaptability and expansion of the WWW browser necessary

- URI schemes are **registered with the IANA**, complete list at http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

URL – Uniform Resource Locator (3/3)

Scheme specific parts:

//[user[:password]@]host[:port]/path?query_string#fragment
(square brackets indicate optional entry)

- **User** – only useful in the case of access limitation to the resource
- **Password** – for authentication of a user
- **Host Name** – complete qualifying name or IP address
- **Port Number** – port for the connection to be established; is usually already determined for most services
- **Path Name** – specifies how to access resources at the specified host with the specified service
- **Query String** – parameters as key-value pairs for server-side applications, e.g. keywords for search (search.php?keyword=openhpi)
- **Fragment Identifier** – label for addressing inside a requested resource

URL Scheme-specific Parts – A Closer Look (1/5)

Many **RFCs** deal with the definition of URL schemes and partly contradict or deprecate prior versions (or even definitions in the same RFC)

Example 1 (1/4)**: user**-part of a URL

- RFC 1738, p.4 defines a common pattern for the scheme-specific part of URLs

```
     //<user>:<password>@<host>:<port>/<url-path>

Some or all of the parts "<user>:<password>@", ":<password>",
":<port>", and "/<url-path>" may be excluded.  The scheme specific
data start with a double slash "//" to indicate that it complies with
the common Internet scheme syntax. The different components obey the
following rules:
```

- In summary:
  - Username, password, port and path are optional components
  - URLs starting with "//" follow a "common Internet scheme syntax"
- …

URL Scheme-specific Parts – A Closer Look (2/5)

Example 1 (2/4)**: user**-part of a URL

- Empty password or no password are being denoted and handled differently

> Note that an empty user name or password is different than no user name or password; there is no way to specify a password without specifying a user name. E.g., <URL:ftp://@host.com/> has an empty user name and no password, <URL:ftp://host.com/> has no user name, while <URL:ftp://foo:@host.com/> has a user name of "foo" and an empty password.

RFC 1738, p.5

- In summary: **valid values** for the user-part are
  - //**user:password@**host (Username and password specified)
  - //**user:@**host (No password needed at all)
  - //**user@**host (User will be prompted for password later)
  - //**@**host (Empty username, will prompt for password)
- …

URL Scheme-specific Parts – A Closer Look (3/5)

Example 1 (3/4)**: user**-part of a URL

- However, while RFC 1738, p.6 repeats and confirms the user-part for the FTP protocol, p.8 of the same document declines it for HTTP

> An HTTP URL takes the form:
>
> http://<host>:<port>/<path>?<searchpart>
>
> where <host> and <port> are as described in Section 3.1. If :<port> is omitted, the port defaults to 80. No user name or password is allowed. <path> is an HTTP selector, and <searchpart> is a query

- This means: even though many browsers support username and password in HTTP URLs (or did this in prior versions), this pattern has never been part of the standard for HTTP URLs
  ➔ this is called a **de facto standard**
- …

URL Scheme-specific Parts – A Closer Look (4/5)

Example 1 (4/4)**: user**-part of a URL

- **Finally**: RFC 3986 (from 2004) deprecates the use of username *and* password in the user-part of URLs due to security concerns (p.17)

> Use of the format "user:password" in the userinfo field is deprecated. Applications should not render as clear text any data after the first colon (":") character found within a userinfo subcomponent unless the data after the colon is the empty string (indicating no password). Applications may choose to ignore or reject such data when it is received as part of a reference and should reject the storage of such data in unencrypted form. The passing of authentication information in clear text has proven to be a security risk in almost every case where it has been used.

- **Indeed**: HTTP of FTP traffic in general is unencrypted, passwords in the URL would be transferred as plain text
- Storing such URLs as a bookmark in the browser (also as plain text) would be even less secure than leaving it on a sticky note at the screen

URL Scheme-specific Parts – A Closer Look (5/5)

Example 2: host and **path** components

- RFC 1738: "/" between host and path is *mandatory*
  → https://open.hpi.de not valid (must be https://open.hpi.de/)
- RFC 3986: "/" belongs to path and is *optional for empty path*
  → both variants valid

Example 3: Allowed characters in FTP URLs

- RFC 1738: all characters after last "/" are interpreted as the *filename*
  → ftp://open.hpi.de/files/download.zip?q=20 would refer a file named "download.zip?q=20"
- RFC 3986, p.12 defines "?" and "=" as reserved characters which cannot be used within a path component

In summary: RFC definitions can be contradictory, the currently valid definition may be hard to find; implementation may differ from the standard

Relative URLs

Besides **absolute URLs** introduced so far, there are also **relative URLs**, e.g. */courses/webtech2015*

- Relative URLs lack a scheme and domain part
  ➔ only consist of path, query string and fragment identifier
- Relative URLs must be interpreted within the context of an embedding resource
  - Hyperlinks on a web page with relative URLs as target inherit the missing parts of the web page they belong to
  - For the above example:
    - the openHPI course list is found at https://open.hpi.de/courses
    - clicking the *Course Details* link for the Web Technologies course would resolve the relative URL /courses/webtech2015 to https://open.hpi.de/courses/webtech2015

URN – Uniform Resource Name

- The prefix tells whether a URI specifies a URL or URN
- Today, URNs are rarely supported
  - One of the few examples:
    German National Library → http://nbn-resolving.de/
  - Resource example: **urn:nbn:de:kobv:11-10028937**
    - *nbn:* National Bibliography Number
    - *de:* Germany, *kobv:* library of the Humboldt-University (Berlin)
    - access the actual document via:
      https://nbn-resolving.org/urn:nbn:de:kobv:11-10028937
- URN serves the worldwide unique and permanent identification of an information resource
- A list with URN name spaces can be found at the IANA page:
  http://www.iana.org/assignments/urn-namespaces

openHPI

Introduction to HTTP

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Introduction to HTTP

WWW – World Wide Web – is a giant hypermedia system with worldwide distributed resources

- WWW resources can be accessed by the HTTP protocol which provides procedures for the retrieval of **resources uniquely identified by URIs**
- **HTTP** – **Hypertext Transfer Protocol** – regulates communication between the information-requesting **WWW browser** (HTTP Client) and the information-providing **WWW server** (also: HTTP server)
- User does actually not get in contact with HTTP; based on intuitive user actions in their graphical interface, browsers determine the required sequence of HTTP commands and processes them accordingly

HTTP: Basic Operations (1/3)

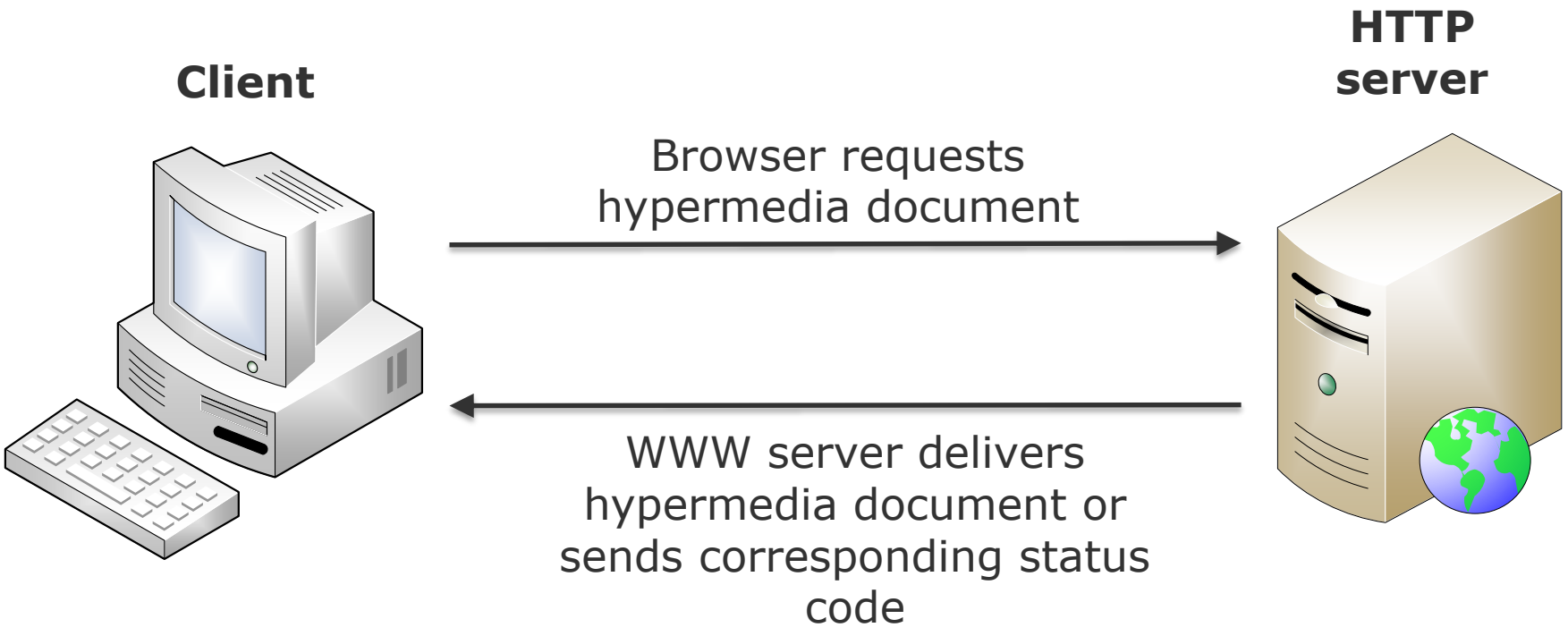
HTTP relies on the reliable, connection-based transport service **TCP** and is a **stateless protocol** that follows a simple **request/response** scheme

Request/Response Procedure:

- Browser as client initiates communication by requesting an information resource at a HTTP server
- WWW server accepts the request and processes it:
  - if the resource is available and browser can gain access to it, the server sends the resource together with a positive status code
  - if the resource is not available or access is prohibited, the server sends a negative status code
- HTTP requests can be triggered
  - by an explicit user action (clicking on a hyperlink, entering an URL)
  - or implicitly by the browser to load embedded resources (e.g. images)

HTTP: Basic Operations (2/3)

HTTP client-server architecture



HTTP: Basic Operations (3/3)

- HTTP has a number of **basic commands** and defined **message format**:
  - **GET** requests a web resource
  - **HEAD** requests a web resource, but expects only header information in response
  - **PUT** requests to write a web resource
  - **POST** requests to append information to a web resource
  - **DELETE** requests to remove a web resource
- Until the emergence of → REST-style web services, only GET and POST were used and implemented widely by browsers and HTTP servers
- HTTP Requests as well as HTTP Responses have a **header** and a **body section** in their message format
- HTTP Responses always have a **Status**, e.g. *200 OK* or *404 Resource not found*

HTTP: Intermediate Systems (1/3)

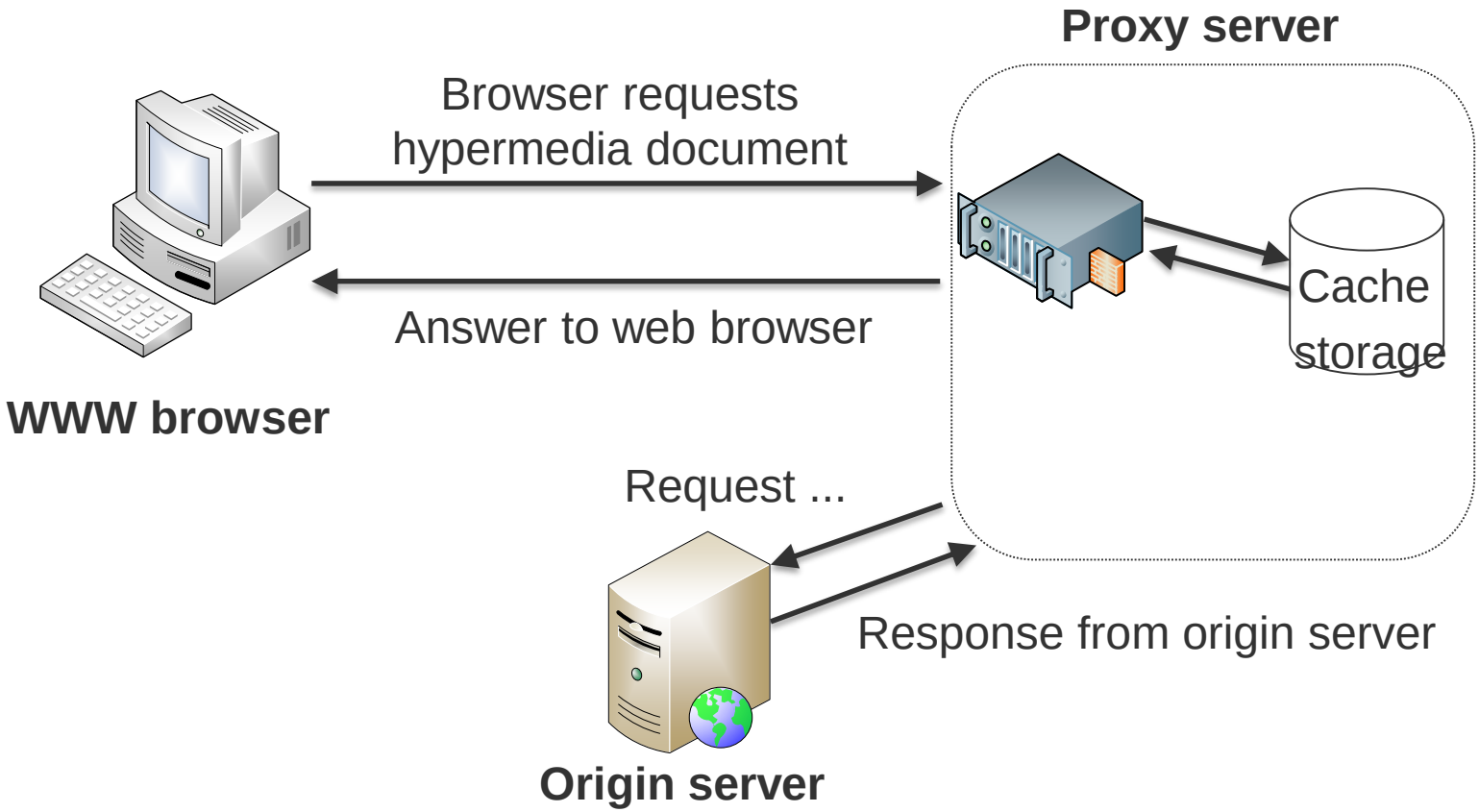
In practical application, the interaction between browser and server is more complicated as various intermediate systems – proxy servers and gateways – are integrated into the communication

Proxy server

- Hybrid role in the communication between client and server
  - works as a server to the client if it can carry out a request based on an earlier communication stored in its cache
  - works as a client to the server – **origin server** – in forwarding client's requests that it is unable to carry out
- All browser requests can be routed over a proxy server (adjustable)

HTTP: Intermediate Systems (2/3)

Operation of a proxy server



HTTP: Intermediate Systems (3/3)

Gateways

- … work like proxy servers, but without the client‘s knowledge
- … are located upstream WWW servers and ease their load or implement security-related access restrictions
- Well-known example of HTTP gateways are **load balancers**

openHPI

History of HTTP

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Historical Development (1/4)

Along with the URL addressing concept and the markup language HTML, **HTTP is a cornerstone of the World Wide Web**

- First HTTP version – **HTTP/0.9** – originated in 1989/90 and was developed at CERN together with URL
- First sophisticated version – **HTTP/1.0** – was introduced in 1992 (RFC 1945)
  - Developmental goal: simple and fast communication protocol
  - HTTP/1.0 offers the following methods:
    - GET – for the delivery of a requested document
    - POST – for information transfer from client to server

Historical Development (2/4)

HTTP/1.0 further offers:

- Adapted MIME concept for transmission of different media types
- Message format for bi-directional communication between client and server (Request / Response)
- Response format and status codes
- Simple user authentication mechanism

HTTP/1.0 has the following shortcomings:

- Limited to short request / response sequences
- Does not support non-IP based → virtual hosts
  → only one domain per IP address
- Very rudimentary caching strategies
- Overly simple and insecure authentication mechanism

Historical Development (3/4)

The current version **HTTP/1.1** was defined in 1997 in RFCs 2068 and 2616

Most important enhancements of HTTP/1.1 versus HTTP/1.0 (1/2):

- Persistent HTTP – **P-HTTP** – connections (via TCP) can be kept open between client and server even after end of a request / response sequence for further client queries to the server → minimizes TCP protocol overhead (less handshakes)
- Support of non-IP based → virtual hosts
- Additional methods: PUT, HEAD, DELETE, OPTIONS, TRACE, CONNECT
- Transfer of document fragments based on exact byte boundaries
- …

Historical Development (4/4)

Most important enhancements of HTTP/1.1 versus HTTP/1.0 (2/2):

- …
- Mechanism for "**content negotiation**"
  - support for client and server, e.g. to agree on language, display quality, coding, etc. of the delivered content
- Packaged coding for transmission of dynamically generated documents
- Improved caching strategies on the server side
- Improved authentication methods without plaintext transfer of name and password

The Future (1/2): HTTP/2

In **March 2012,** the IETF working group responsible for HTTP officially began work on HTTP/2 13 years (1) after adoption of HTTP/1.1 …

- First proposals were already presented in May 2012, among them:
  - Google's **SPDY 3.0** ("Speedy") and
  - Microsoft's **HTTP Speed+Mobility**
- August 2012:
  - IETF Working Group HTTPbis took **SPDY** as starting point for discussion on the draft for HTTP/2
- September 2014:
  - "Last Call for HTTP/2" (published by the Working Group HTTPbis)
- December 16, 2014:
  - Working Group submitted HTTP/2 as Proposed Standard to IESG
- …

The Future (2/2): HTTP/2

February 17th, 2015:

- Publication of HTTP/2 as RFC Internet Standard

Focus in standardization:

- Increase in the “speed perceived”
- Backwards compatibility to HTTP/1.1

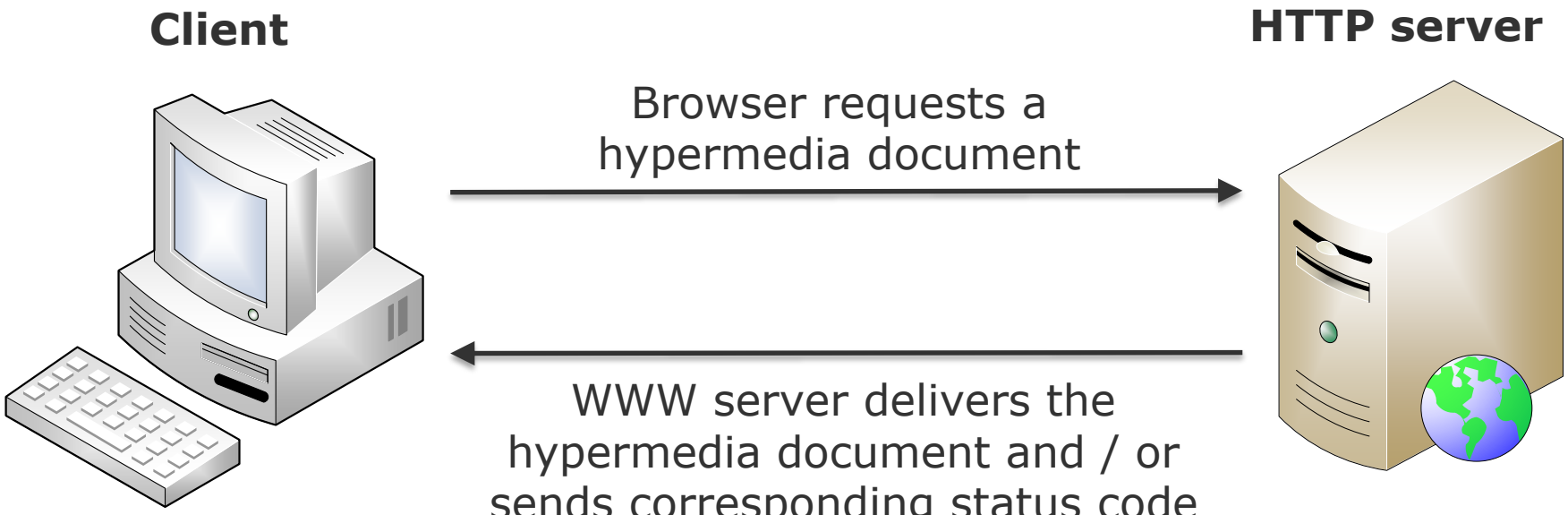
openHPI

HTTP Message Format and Status Codes

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

HTTP Request / Response Communication

Reminder: HTTP Client and Server communicate by means of **request** and **response messages**



HTTP: Generic Message Format

All HTTP messages fit a common structure, the **generic message format**

- **Start line** determines if the message is a request or response and contains either request- or response-specific information
- **Message headers** are simple pairs of names and values separated by colons, i.e.
  `<header-name>: <header-value>`
- **Message body** contains the actual information, e.g. the requested resource or a detailed error message
  - Message body is optional since not all HTTP messages need one

HTTP: Request Message Format (1/3)

A client starts a **HTTP session** by establishing a *TCP connection* to the server and sending a **request message**

Request Message Format:

- **Start line** is called **request line** and contains `<METHOD> <request-uri> <HTTP-VERSION>`, e.g. `GET /courses/webtech2015 HTTP/1.1`
  - URLs in the request line are usually relative
- **Message headers** section contains different types of headers
  - **General headers** are about the message itself and are not specific to request or response
  - **Request headers** contain details about the request itself
  - **Entity headers** describe the body content (if provided)

HTTP: Request Message Format (2/3)

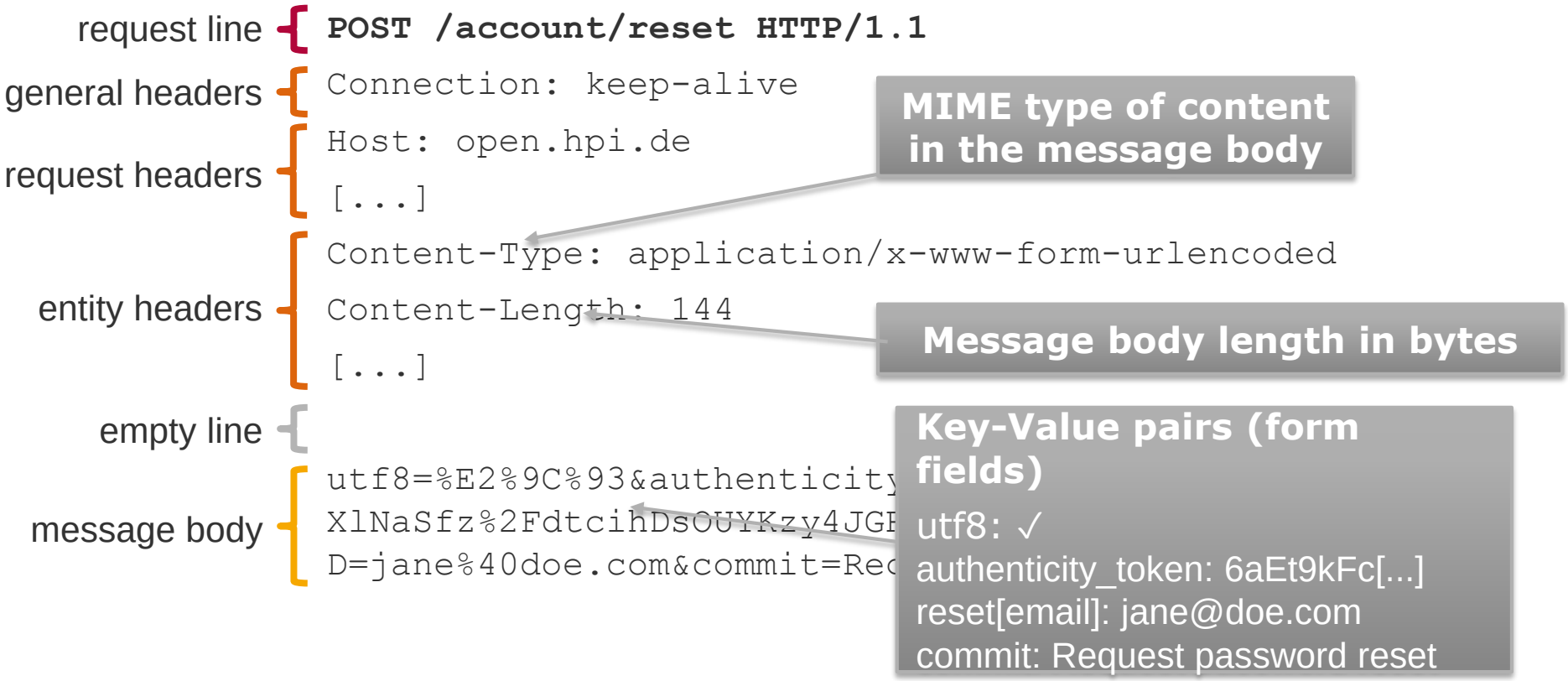
Example: GET request

Opening the openHPI courses overview at https://open.hpi.de/courses, the browser performs a GET request like this:

HTTP: Request Message Format (3/3)

Example: POST Request

The POST method is typically being used, when the browser should submit a form, e.g. when a new password is requested on openHPI

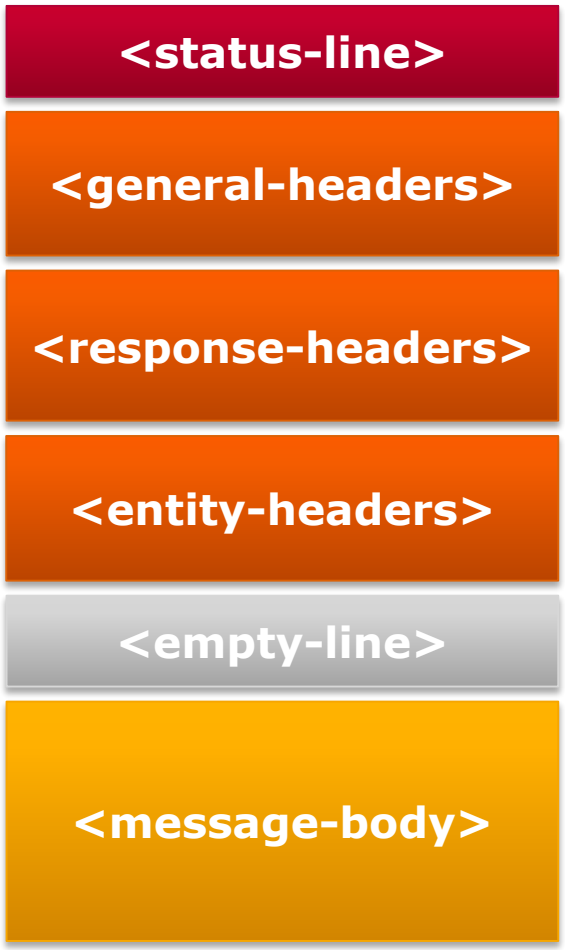


Response Message Format (1/4)

Server delivers requested resource or an error code as **response message**

Response Message Format:

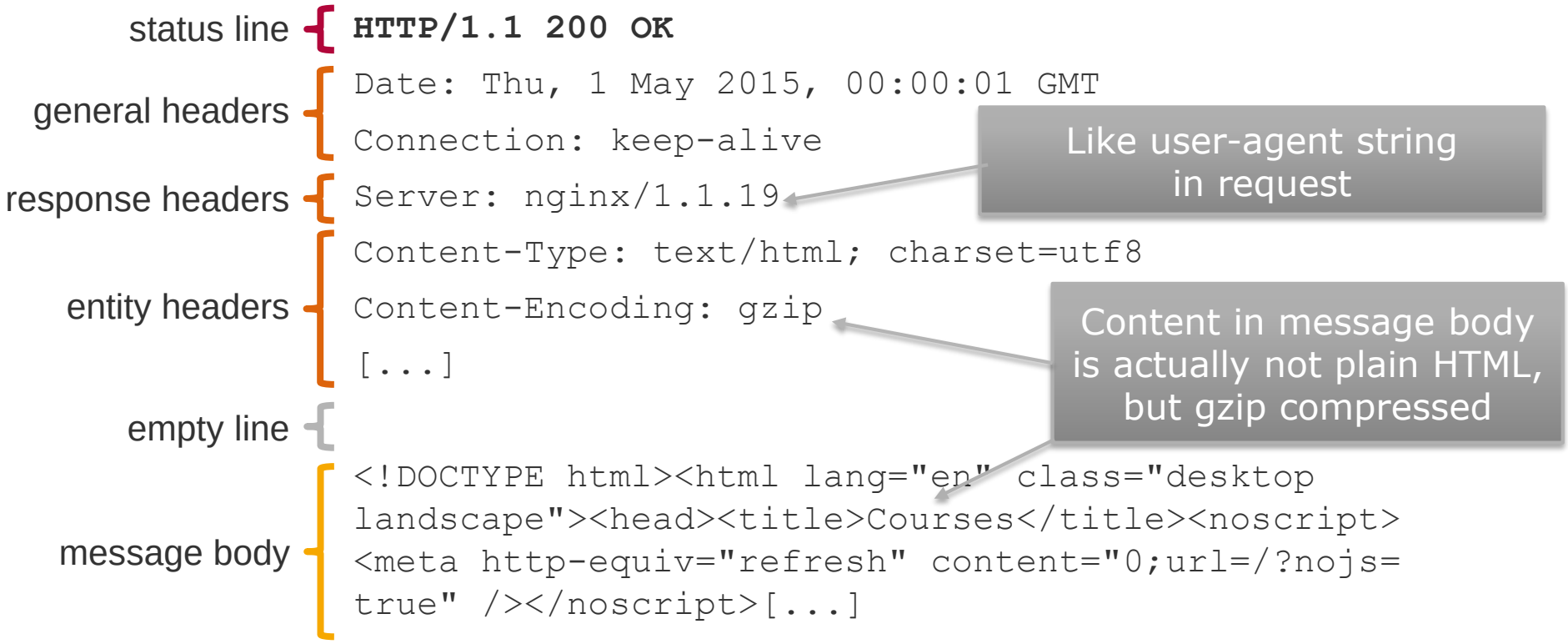
- **Start line** of a response is a **status line**:
  `HTTP-VERSION status-code reason-phrase`,
  e.g. `HTTP/1.1 404 Not found`
  - HTTP version may not be greater than the version from the corresponding request
- **General headers** and **entity headers** are the same as for request messages
- **Response headers** contain additional information for the client that varies depending on the actual status code of a message
- **Message body** either contains the delivered resource or information about an error



Response Message Format (2/4)

Example: Successful response

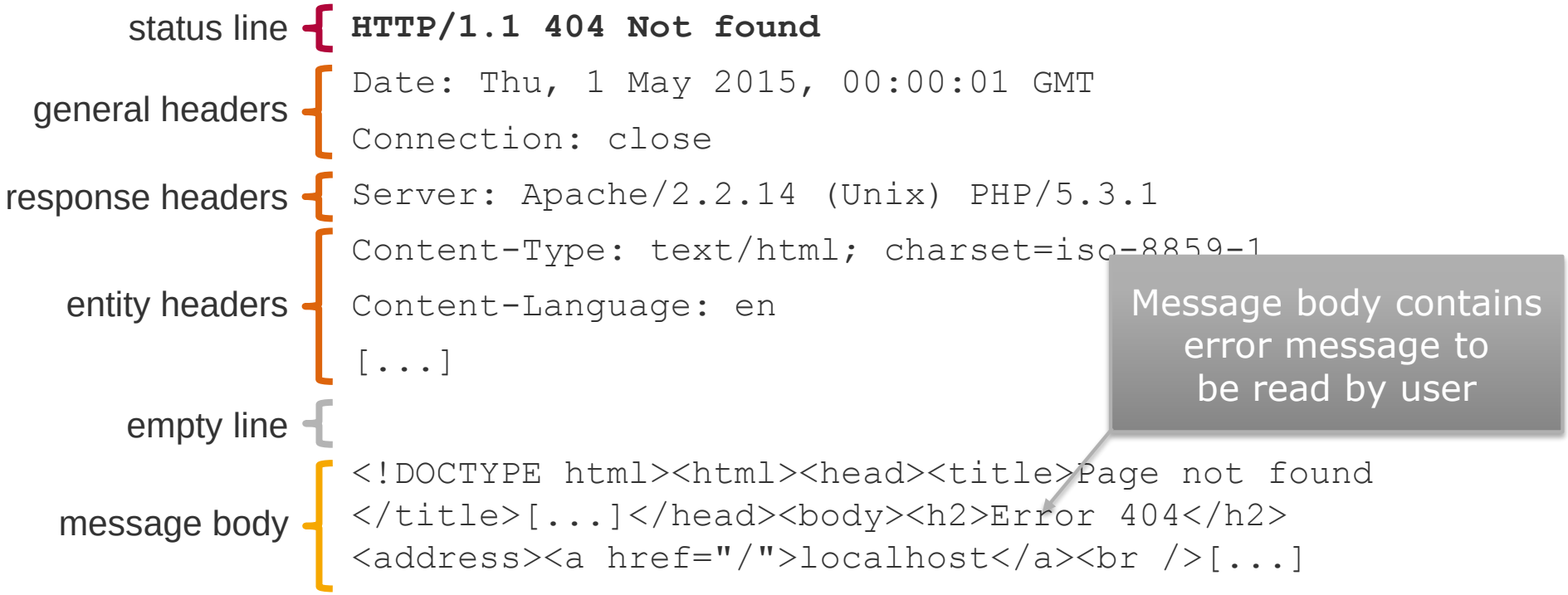
When a request can be answered successfully, the server delivers the resource with a **200 OK** status code



Response Message Format (3/4)

Example: Requested resource not found

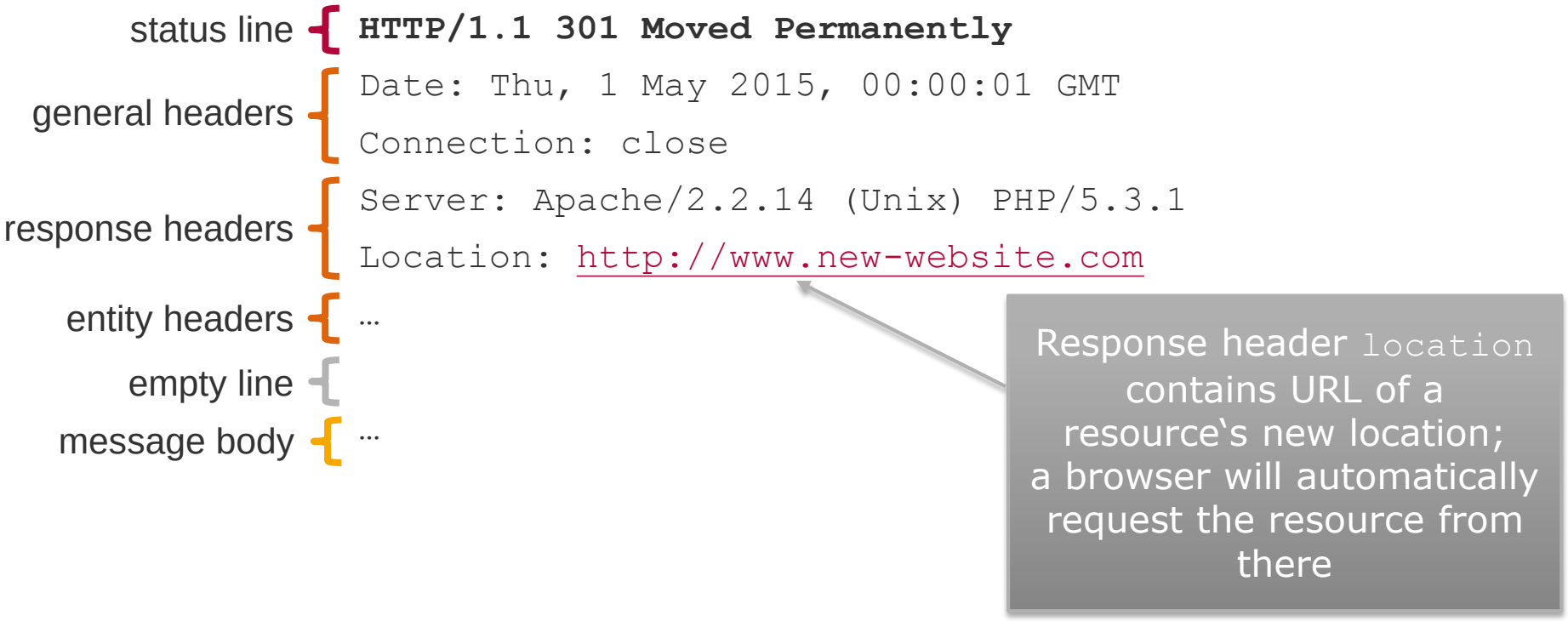
When the resource from the request cannot be found on the server, the answer will have the well known **404 Not found** status code



Response Message Format (4/4)

Example: Requested resource has been moved

When a site owner moves a resource, a **redirect** pointing to the new location can be implemented, e.g. with **301 Moved Permanently**



HTTP Status Codes (1/2)

HTTP defines different **classes of status codes**

- **1xx – Informational Messages**
  - Provisional response just for information
  - Example: **101 Switching Protocols**
    Client asks to switch protocol, server acknowledges to do so
- **2xx – Success**
  - Codes indicate that the server could receive and process the request successfully
  - Example: **200 OK**
- **3xx – Redirection**
  - Codes indicate that clients must perform an additional action (e.g. additional request) to receive the actual resource
  - Example: **301 Moved Permanently**

HTTP Status Codes (2/2)

HTTP defines different **classes of status codes**

- **4xx – Client Error**
  - Codes indicate that the client request is erroneous, i.e. malformed or requesting a non-existent or unauthorized resource
  - Example: **400 Bad Request**
    The client's request was syntactically malformed
- **5xx – Server Error**
  - The client's request was valid, but the server was not able to fulfill the request – and knows that the reason is server-side
  - Example: **503 Service Unavailable**
    Server could (temporarily) not answer request, e.g. due to overload

IANA maintains the official registry for HTTP status code:
http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

openHPI

HTTP Transfer Speed Optimization

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Plain HTTP is Inefficient

Reminder: Performing an HTTP request / response cycle includes establishing (and terminating) a TCP connection

- Today's websites include numerous **embedded resources**, such as images, style sheets or Javascript files, e.g.
  - opening the openHPI home page produces 43 HTTP request / response cycles
  - 41 of these requests point to the openHPI servers
- TCP handshakes pose a significant overhead without optimization
- Many requested resources do not change very often
- HTTP offers means for request reduction and speed enhancements
  - Persistent connections and HTTP Pipelining
  - Compression
  - ➔ Caching

Persistent Connections (1/2)

HTTP allows **multiple** request / response cycles within one HTTP session:

Persistent Connections

- A maximum of two persistent connections should be established between client and server (to avoid to TCP congestion)
- Persistent connections are established by setting the connection header field:
  - `Connection: keep-alive`
- If the server supports persistent connections, it also adds
  - `Connection: keep-alive` to the response
- Client terminates the session by setting the option "close" in the connection header field with the last HTTP request
  - If the client misbehaves, the connection will be kept open at the server (until timeout), leading to possible overload

Persistent Connections (2/2)

Persistent Connections are **default behavior** since HTTP/1.1

Advantages

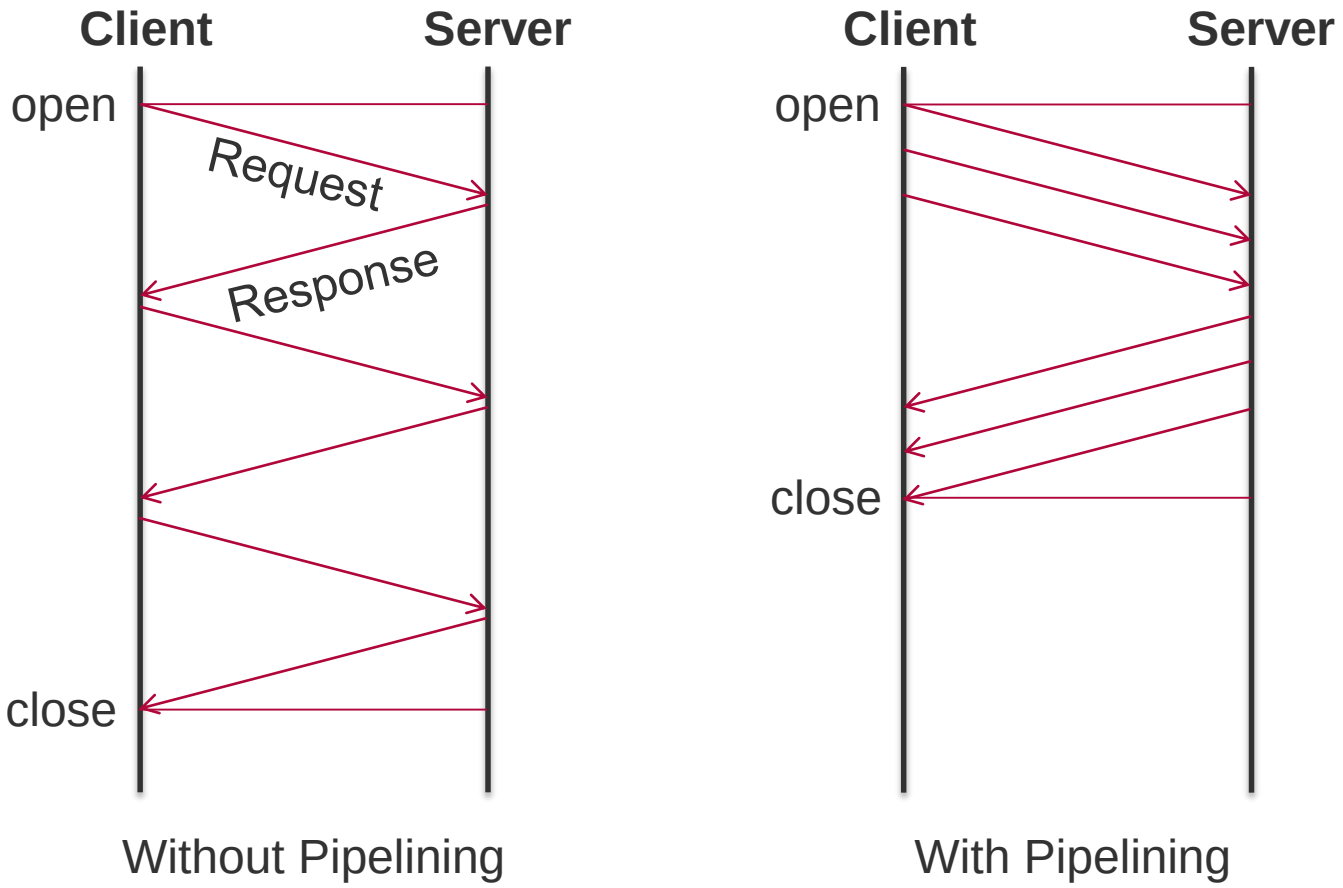
- More efficient use of operating system resources (cpu and memory) due to fewer simultaneous connections
- Better bandwidth utilization (fewer unnecessary TCP packets)
- Reduced latency for requests on embedded resources (no more TCP handshakes required)
- Persistent connections allow for ➔ **HTTP pipelining**

HTTP Pipelining (1/2)

HTTP/1.1 allows clients to send multiple requests through an open TCP connection without waiting for the response of the previous request: **HTTP Pipelining**

- Pipelining allows a great speedup especially for high latency connections
- Only sequences of requests where later ones do not rely on earlier requests should be pipelined, i.e.
  - sequences of GET- or HEAD-requests can always be pipelined
- Start of each next HTTP request must be featured explicitly:
  - Achieved by length specification in the `content-length` header field
- Most browsers support HTTP pipelining, however, most browsers disable the feature in default settings

HTTP Pipelining (2/2)



Compression

HTTP/1.1 allows servers to deliver data in the message body to be **compressed**

➔ higher transfer speed and bandwidth utilization

- Clients announce their compression capabilities in the request header:
  - `Accept-Encoding: gzip, deflate`
- Servers choose a matching compression method and indicate the chosen method in the entity header:
  - `Content-Encoding: gzip`
- Common encoding tokens:
  - **deflate**: deflate algorithm as in RFC 1951
  - **gzip**: GNU zip algorithm (RFC 1952), most broadly supported
  - **exi**: W3C efficient XML interchange
  - **identity**: no compression at all

openHPI
HTTP Caching

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Caching Principle

Reminder: Many requested resources do not change very often

- Efficiency of the WWW may be increased dramatically by avoiding repeated data transfer of the same information resources. Possible with the help of intelligent intermediate systems, so-called **caches**
- A **cache** acts as a third element within client-server interaction:
  - All browser requests to a WWW server are routed via an intermediate cache
  - The cache stores the respective answers for a limited time
  - In case the resource is re-requested, the intermediate cache does not route the request on to the server but can answer it directly
- Caching reduces communication traffic and server load

HTTP Cache Implementations

A cache can be placed in different ways between browser und server:

- **Client-side cache** – The cache is at the client and stores responses to client requests. When loading subsequent interlinked pages of the same websites, many (shared) resources (images, CSS, Javascript) are stored in separate files and included by every page
  → these shared resources can easily be cached
- **Independent cache** – A cache is logically placed independently between browser and server, e.g. at a gateway that connects an intranet with the Internet. Client must be configured accordingly so it can use the cache
  → Multiple clients using the same gateway can share the cache
- **Server-side cache** – A cache is at the server. The cache stores the responds sent by the server and delivers them independently upon subsequent requests

Cache Consistency

The cache must decide if the requested, already cached resource is still valid or not – invalid resources in the cache are called **stale**

- **Cache hit** – requested resource is cached and valid – cache can handle the request directly
- **Cache miss** – requested resource is either not cached or is cached but stale– cache must then forward request to origin server

Problem: **Cache consistency**

- Is the cached resource valid, i.e. does the original document match the variation stored in the cache?

  Solution:

  - Documents are delivered with timestamp and expiration date

HTTP Cache Headers (1/2)

The most important **header field for caching** is called `cache-control`

- **Example**: `cache-control: max-age: 3600`
- Important cache-control settings
  - **max-age:** defines the time in seconds after which a cached resource becomes stale and must be re-requested from the server
  - **no-cache:** setting the *cache-control* header to *no-cache* tells a cache to revalidate the resource on every request
  - **no-store** prevents the cache from storing a resource at all

An older HTTP cache header is the `expires` field

- This header takes a timestamp after which it becomes stale
- Setting the *max-age* in the cache-control header overrides the expires header

HTTP Cache Headers (2/2)

Content Revalidation

- Client must revalidate a cached resource if
  - *max-age* is reached
  - cache-control header is set *no-cache* or *must-revalidate*
- Client sends the timestamp when it last accessed the resource in the `if-modified-since` header
  - If the resource has been updated on the server since then, the server answers with a 200 OK and the updated resource
  - Otherwise, it sends a **304 Not Modified** status in the response and no resource in the message body (will be served from cache)

openHPI
Cookies

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Sessions

- Many applications require a “memory” of previous interaction between browser and WWW server, e.g.
  - use of a virtual shopping cart with online shopping,
  - a website with login, …
- HTTP is, however, a **stateless protocol**
- Strategies to overcome statelessness are based on **sessions**
  - i.e. the set of related subsequent user interactions causing a request-response cycle between the client and server
- Session is usually characterized with an ID; this ID can be transmitted via
  - **Cookies** – key-value pairs in the HTTP header
  - **“hidden fields”** in HTML forms
  - **URL rewriting**

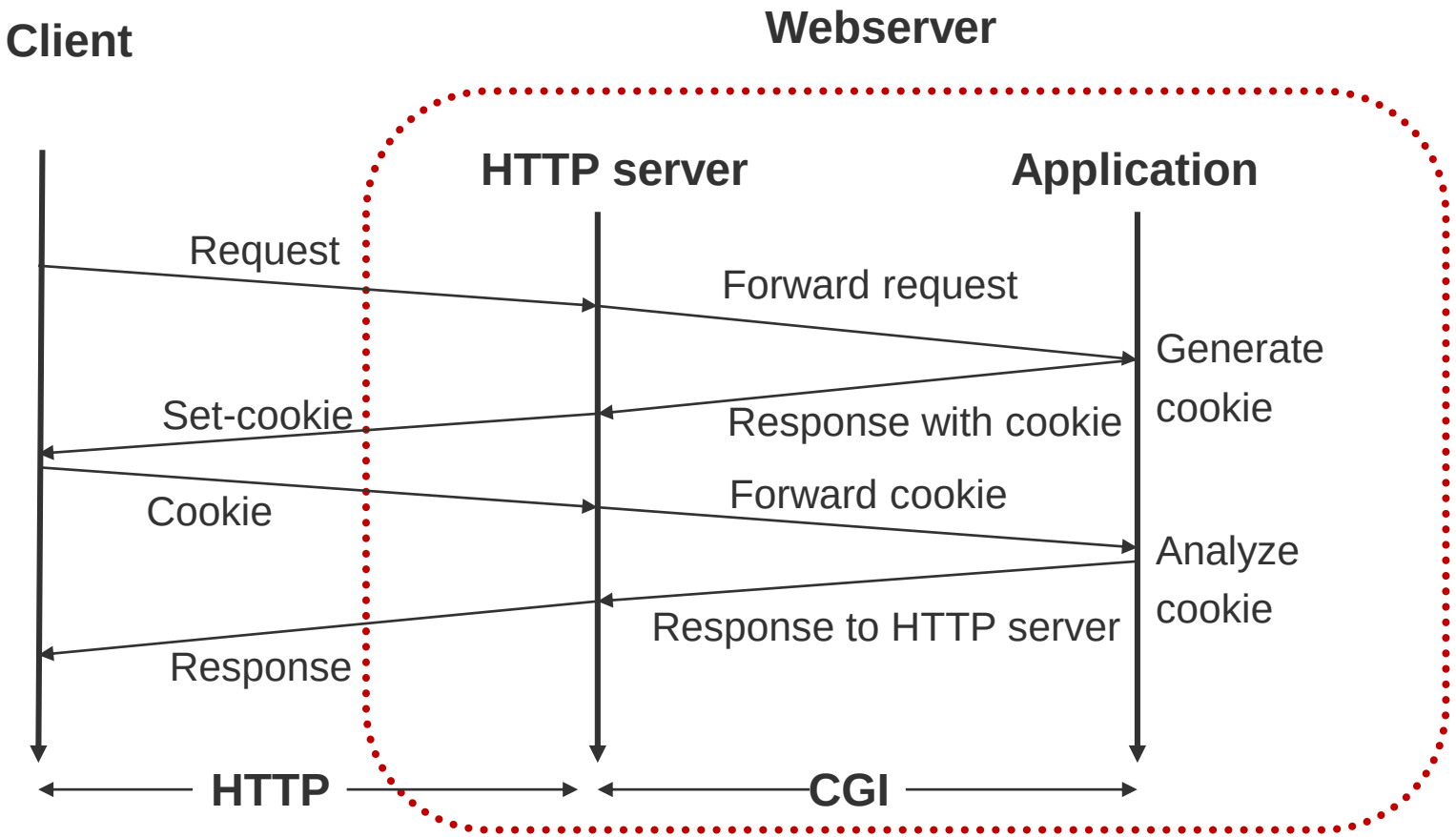
Cookies

Cookies – originally developed in the 1990s by Netscape

- Cookie mechanism allows to exchange status information for identifying earlier related request / response cycles without the need for persistent network connections
- **Procedure:**
  - HTTP server initiates a session by sending cookies to the browser in the response header field `set-cookie`
  - Browser stores the cookie in an internal database
  - On every request to the same server, the browser sends all stored cookies for that server in the `cookie` request header field
- Usually only IDs are stored in cookies and the actual information (e.g. virtual shopping cart) is stored on server-side

  → smaller message and therefore more efficient

HTTP Cookie Mechanism



openHPI
Content Negotiation

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

Resources are Available in Different Variants

- Servers can store information resources with the same URI in different variations, in respect to e.g.
  - Language – different language versions
  - Encoding – compressed with different compression methods
  - Charset – different character encodings

Content Negotiation:

- RFC 2068 specifies procedure for HTTP/1.1 that client and server can negotiate which variation of a resource is to be transferred

Content Negotiation Variants

There are **three variants** of content negotiation:

- **Server-driven negotiation**
  - Client specifies acceptable variants in the `accept` header fields
  - HTTP server takes over selection of suitable resource variant based on the client request
- **Agent-driven negotiation** (i.e. client-driven)
  - Server presents information about the available variations in a first response to the client with a **300 Multiple Choices** status
  - Client selects the most suitable among them and fires second explicit request for the selected resource variant
- **Transparent content negotiation**
  - Combination of server-driven and agent-driven negotiation by means of a proxy server

User Settings and Content Negotiation

User settings in the browser affect the **`accept`** fields in the HTTP header, for example the language of preference

Example:

- **Request**

```
GET / HTTP/1.1
Host: open.hpi.de
Connection: keep-alive
Accept: text/html,image/jpeg;q=0.4
Accept-Encoding: gzip,deflate
Accept-Language: de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
…
```

- **Response**

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Language: de
Connection: keep-alive
…
```

openHPI
HTTP Security

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

HTTP Security

In the following, we want to consider mechanisms for **HTTP security**

- **Authentication**
  - Allows to authenticate and authorize client (and server) in a HTTP communication
- **HTTPS – Transport Channel Security**
  - Provides means for encryption to defend the content of web documents when transported by HTTP over the open Internet from being spied out or even manipulated
  - Implements strong authentication to prevent interception of confidential data like passwords by unauthorized parties

Authentication Methods

- If security-relevant resources shall be accessed via HTTP, suitable authentication and authorization of the client is necessary
- HTTP is a stateless protocol, i.e. results from a previous communication cannot be "remembered", but is only valid in the time span between connection establishment by client request and delivery of the requested resources
- HTTP/1.1 knows two different methods for authentication:
  - **Basic Access Authentication**
  - **Digest Access Authentication**

HTTP Basic Authentication (1/3)

Procedure (1/2):

- Client sends request for protected resource
- Server checks availability of the resource and responds with status code **401 Unauthorized** together with `WWW-Authenticate` response header field
  - Header determines the authentication type, i.e. `WWW-Authenticate: Basic`
  - Header value also contains **realm name** – a name for the protected resources area, e.g.
    - `WWW-Authenticate: Basic realm="Restricted Area"`
- Client displays a login form and asks the user for credentials
- ...

HTTP Basic Authentication (2/3)

Procedure (2/2):

- Client concatenates username and password strings with a colon **(username:password)** and encodes this string with **Base64**, e.g.
  - *user:EMBn-4J&dtA → dXNlcjpFTUJuLTRKJmR0QQ==*
- Client sends new request for the protected resource with the `Authorization` request header field set
  - Header field contains the auth method and the base64 string, e.g.
    - **`Authorization: Basic dXNlcjpFTUJuLTRKJmR0QQ==`**
- Server checks the credentials and delivers the protected resource

HTTP Basic Authentication (3/3)

Potential pitfall:

- Base64 is a (reversible) *encoding*, not an *encryption*
- transmitting the encoded string is no more secure than plain text

→ Transport channel security must be ensured for Basic Auth

HTTP Digest Authentication

Digest Access Authentication is a **stronger authentication** method and meant to allow HTTP Authentication without the need for an encrypted transport channel

- **Digest Auth** applies a cryptographic hash function – MD5 – to the credentials before transmitting them
- A nonce (random value) sent by the server in the *WWW-Authenticate* header can be included in the credential-string before hashing to prevent replay attacks

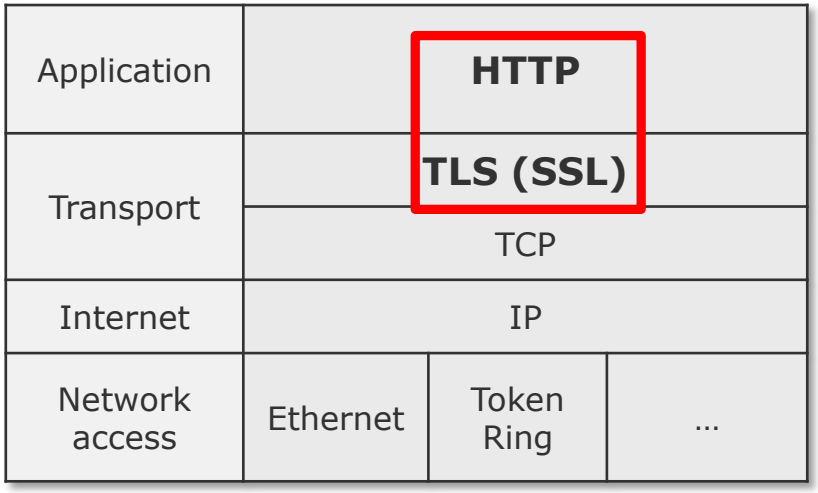
Limitations of digest authentication:

- Today MD5 is considered as broken and should not be used any longer in applications with high security standards
- Digest authentication is vulnerable to a Man-in-the-Middle attack:
  - The MitM can intercept the initial response and manipulate the *WWW-Authenticate* header to ask the client for *Basic Auth*

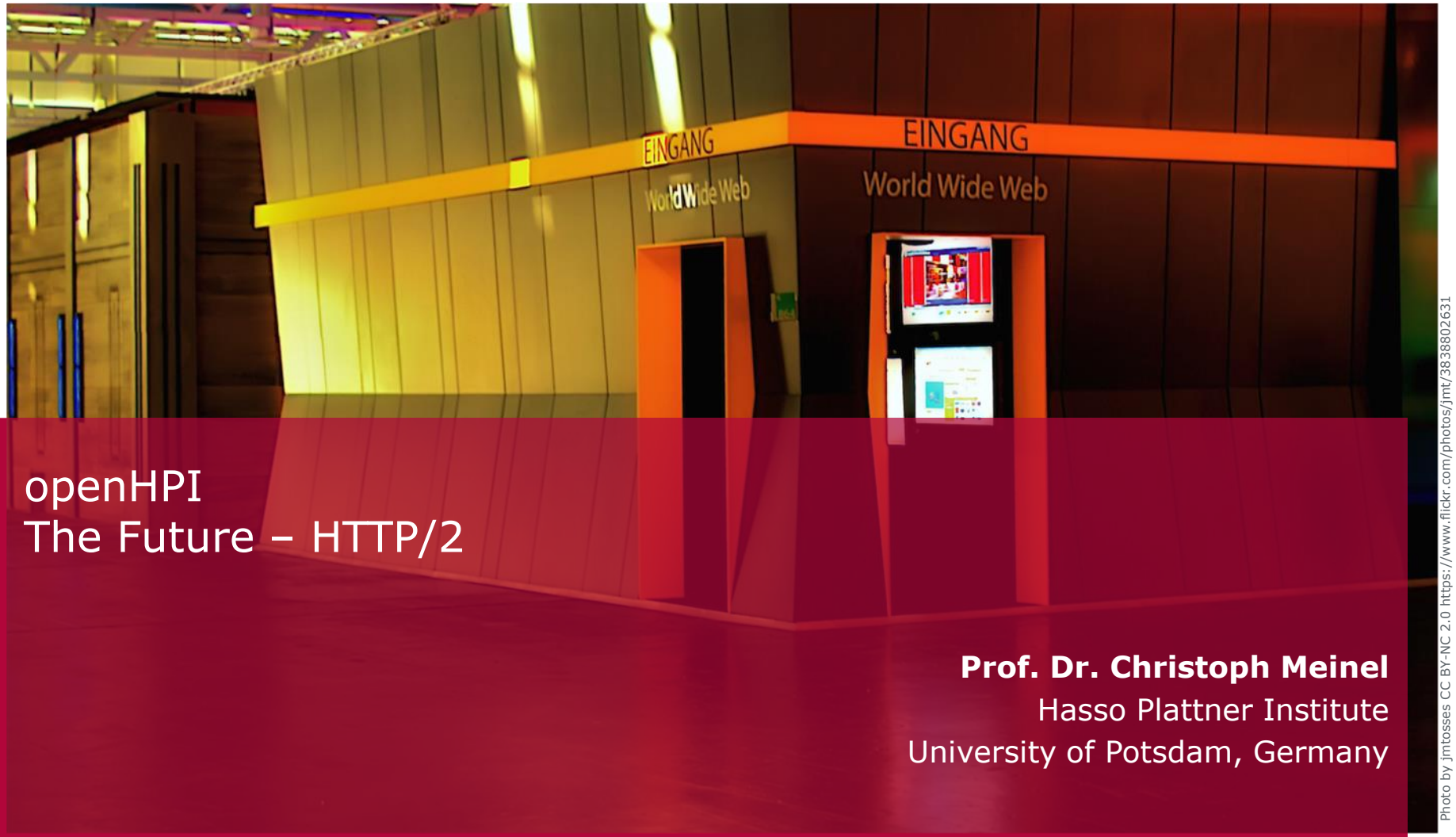
HTTPS – Transport Channel Security

Secure alternative to HTTP: HTTPS (RFC 2818)

- HTTPS – Hypertext Transfer Protocol *Secure*
- Encryption and strong authentication (optional) of communication between webserver and browser
- Basic idea: **HTTP via TLS** (SSL)
  - Realization via additional (sub)layer in the TCP/IP stack
  - TLS (Transport Layer Security), formerly known as SSL, provides encryption and certificate-based authentication



- With HTTPS, passwords and other confidential data can no longer be intercepted easily by unauthorized third parties



openHPI
The Future – HTTP/2

Prof. Dr. Christoph Meinel
Hasso Plattner Institute
University of Potsdam, Germany

HTTP/2 – Overview

In February 2015, the IESG approved HTTP/2 as a new Internet Standard

The HTTP/1.1 successor comes with many improvements, where the main focus is on an **improved perceived speed-up** for the end user

HTTP/2 …

is a binary protocol

introduces request multiplexing

can compress header information

specifies server push

is backward compatible to the HTTP/1.1 vocabulary
➔ methods and status codes from HTTP/1.1 will remain

may make the web more secure

HTTP/2 – Streams, Messages, Frames

HTTP/2 is a binary protocol – while all earlier versions have been plain-text based (and thus: human readable)

HTTP/2 introduces the concepts of streams, messages and frames for communication

Stream: a virtual channel within a connection, which carries bidirectional *messages*

Message: a logical HTTP message, such as a request, or response, which consists of one or more *frames*

Frame: the smallest unit of communication, which carries a specific type of data, e.g., HTTP headers, payload, etc.

Multiplexing

Known problem of HTTP/1.1: limited number of connections for a host
→ downloading many resources from a host only one after another

HTTP/1.1 best practise: **Domain Sharding**

Different resources of a website are published on subdomains (i.e. images, js, css) → different domains allow more concurrent TCP connections

HTTP/2 introduces multiplexing

allows to request and receive multiple resources one TCP/IP connection, intertwined

Requests are not blocking any longer
→ no need for multiple TCP connections on multiple domain names

HTTP/2 Compression

HTTP/2 introduces header compression

HTTP/1.1 headers had to be uncompressed; compression was only allowed for the message body

HTTP/2 allows for header compression, using a special compression method – HPACK

➔ especially interesting for sites with large cookies (1MB+ of cookie data are not uncommon nowadays)

HTTP/2 discourages the use of compression for the message body using gzip or deflate due to security concerns

- BREACH attack on HTTPS is based on a flaw caused by data compression
- Concerns against gzip was a reason to introduce HPACK for header compression

Server Side Push

In HTTP/1.1, resources can only be delivered to a client by means of a full request-/response cycle

Common situation: client performs first request, parses HTML document and must request many embedded resources

HTTP/2 introduces server push: server can send along a number of additional resources within upon the first HTTP request, saving unnecessary roundtrips

Unclear implementation: how can an HTTP server identify resources to be pushed?

- Idea 1: authors of web applications can initiate server push explicitly or via additional HTTP headers
- Idea 2: HTTP servers learn, which resources need to be pushed

HTTP/2 and Encryption

Google's SPDY was a planned as TLS-only protocol (enforced encryption), HTTP/2 will **not enforce TLS**

Specification allows plain HTTP/2 messages without encryption

However: major browser manufacurers will enforce the use of TLS in their implementation (Google Chrome, Firefox)

- *„Firefox will **only** be implementing **HTTP/2 over TLS** – and so far that means for https:// schemed URLs. It does enforce the protocol's >= TLS 1.2 requirement – if a server negotiates HTTP/2 with a lower TLS version it is treated as a protocol error.“*

HTTP/2 – Browser and Server Support

As of February 2015 – **Browsers:**

Internet Explorer 11 (Windows 10), only with TLS

Firefox 34+, enabled in defaults since 36, only with TLS

Chrome/Chromium (current versions), only with TLS

Servers:

Internet Information Server (Windows 10)

OpenLightSpeed

Lucid

H2O