Lecture 4: The HyperText Transfer Protocol

Table of Contents

Objectives

Lecture

1. Introduction

Let's face it: there is not a day, probably not an hour, maybe not even a minute where you are not using the HTTP protocol in one way or another:

- You are certainly spending a lot of time **browsing and reading information** published

on the WWW.

- You are also using a wide variety of **applications** that have been built on top the WWW technical infrastructure. Think about social networks, e-commerce sites, online games, e-banking services: they all rely on HTTP to transfer information between the "brain" of the application (i.e. the server) and its "face" (i.e. the client). It does not matter if you are using the applications via your browser or via a **mobile** application: without HTTP you would not be able to poke your friends or to pay your bills.

- Even if you are not seeing any LCD display nearby, you will increasingly be exposed to HTTP without noticing it. The protocol is already used to connect all sorts of **machines** and **objects** to *The Cloud*. Just think about the coffee vending machine down the hall or about the baby scale that you are likely to get some day.

This shows that HTTP has become a generic protocol, on top of which many distributed systems are being built. Whether you are a **systems engineer**, a **software engineer**, an **embedded systems engineer** or a **security engineer**, you have to speak this protocol natively and must master its concepts and syntax. You have to be able to open a terminal, open a TCP connection with a remote server, type an HTTP request, hit return and interpret the response. *Resistance is futile*.

In this lecture, we will present the **key protocol concepts** and look at what a **typical HTTP conversation** looks like. We will also include **fragments of the 176-pages [RFC 2616]** (http://tools.ietf.org/html/rfc2616), which specify how these concepts are put in practice.

2. A First Look at an HTTP Conversation

Before looking at what is defined in the HTTP specification, let us observe what a typical HTTP exchange between a client and a server looks like. To capture the traffic, different options are available. Of course, it is possible to use a packet analyzer like **Wireshark**. But a lot of information is already available directly in your favorite browser. For instance, you may install the **Firebug** browser extension (and use the Network tab), or use the various developer-oriented features that are commonly available (every browser has its own menu, but the kind of information you get is always very similar).

In our test scenario, the user has typed `http://www.nodejs.org` in the browser navigation bar and hit return. At this point, you can observe that the following bytes are being **sent from the client to the server**:

```
GET / HTTP/1.1 CRLF
Host: www.nodejs.org CRLF
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:28.0) Gecko/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.8,fr;q=0.5,fr-fr;q=0.3 CRLF
Accept-Encoding: gzip, deflate CRLF
Cookie: __utma=212211339.431073283.1392993818.1395308748.1395311696.27;
__utmz=212211339.1395311696.27.19.utmcsr=stackoverflow.com|utmccn=(refer
|utmcct=/questions/7776452/retrieving-a-list-of-network-interfaces-in-no
Connection: keep-alive CRLF
CRLF
```

There are a couple of interesting things that we can observe. We can quite **easily read HTTP messages**. We see that we are facing a text-based protocol. We see that the structure of the **first line** is a bit different from the following ones, which all start with a name, followed by a colon (they look like **headers** or some kind of metadata)

Right after the previous bytes have been sent to the server, we see that **a response comes back from the server**:

```
HTTP/1.1 200 OK CRLF
Server: nginx CRLF
Date: Sat, 05 Apr 2014 11:45:48 GMT CRLF
Content-Type: text/html CRLF
Content-Length: 6368 CRLF
Last-Modified: Tue, 18 Mar 2014 02:18:40 GMT CRLF
Connection: keep-alive CRLF
Accept-Ranges: bytes CRLF
CRLF
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link type="image/x-icon" rel="icon" href="favicon.ico">
    <link type="image/x-icon" rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="pipe.css">
...
```

Again, the response is easy to interpret. Once again, we see that there is a **first line** with some information (we see a `OK`, so it looks like everything went well). Once again, we see a number of lines that look like **headers**. And this time, after an **empty line,** we see **HTML content** (note: only the beginning has been included in the previous transcript). So that must be the markup that should be rendered by the browser!

Now that we have a first sense of what HTTP messages look like, let us dig into the RFC and examine what are the most important concepts defined in the specification.

3. HTTP is a Stateless Request-Reply Transfer Protocol

HTTP is a client-server protocol, which is used to transfer representations of resources (more on this later) over TCP/IP networks. The server accepts connection requests on a TCP port (the default is port 80). Once connected, the clients send requests to either *ask for* (this is what happens when you click on a hyperlink) or *send* data (this is what happens when you press on the submit button in a HTML form). HTTP is used to transfer all sorts of **media types** (text, images, binary data, etc.), in all sorts of **formats** (plain text, HTML markup, XML markup) and using all sorts of **character encoding systems** (UTF-8, etc.).

HTTP is stateless because every request issued by a client is considered as independent from the others. Hence, **HTTP servers do not need to manage a session** and do not need to maintain a state for each client. Once they have sent back a response to a client, they can just forget about it. This design choice has a number of benefits:

- it makes the **implementation of clients and servers easier**;
- it **reduces the consumption of resources** (maintaining state for each client can require a lot of memory on the server side);
- it makes it easier to **deal with load and failure**, by using a cluster of equivalent HTTP servers (the client does not care which server will be fulfill its requests and each request might be fulfilled by a different server in the pool).

Having said that, there are two important points that you have to be aware and that we will discuss now. Firstly, while the HTTP protocol is stateless, many Web applications are not stateless. Secondly, the fact that HTTP is stateless does not mean that we have a different TCP connection for each request-reply exchange.

3.1. The HTTP Protocol is Stateless, But Many Web Applications are NOT

If you think about what you do with your web browser, this point about statelessness should be puzzling:

- When you are buying groceries on your favorite e-commerce site, you have a shopping cart and can add one product after the other. *So the server does maintain a state about the shopping session*.
- When you are checking your grades on the school information system, you first have to login, then can check your progress in several courses and finally can logout. *So the*

> *server does maintain a state about the conversation, knows who you are and remembers that you have provided valid credentials earlier in the conversation*.

Does that mean that the HTTP RFC is lying to us? Of course not. But that means that **it is the responsibility of the application built on top of HTTP to manage the conversation state**. There are a number of ways to do that:

- One way is to transport the entire conversation state in every request and every response exchanged in the context of an application session. One technique for doing that is to use hidden HTML form fields. The conversation between a client and a server would look like this (the client needs to **repeat** what he has said before each time he talks to the server):

```
C: Hello, I am new here. My name is Bob.
S: Welcome, let's have a chat [You told me that "My name is Bob"].
C: [My name is Bob]. What's the time?
S: Hi again Bob. It's 10:45 AM. [You told me that "My name is Bob". You
```

- Another way is to store the state on the server side, to assign a unique id to every application session and to pass this session id back and forth in every request and response. The conversation between a client and a server would look like:

```
C: Hello, I am new here. My name is Bob.
S: Welcome Bob, let's have a chat. Your session id is 42.
C: My session id is 42. What's the time?
S: -- checking my notes… hum… ok, I found what I remember about session
S: Hi again Bob. It's 10:45 AM.
C: My session id is 42. How do you do?
S: -- checking my notes… hum… ok, I found what I remember about session
S: I am fine, Bob, thank you.
C: My session id is 42. How do you do?
S: -- checking my notes… hum… ok, I found what I remember about session
S: I told you I am fine… are you stupid or what?
C: My session id is 42. If you take it like that, I am gone. Forever.
S: -- checking my notes… hum… ok, I found what I remember about session
S: -- putting 42 file into trash…
S: Bye Bob.
```

There are different ways to pass the session id back and forth. One way is to use a parameter in the query string (which has security implications). Another way is use **cookies**, as defined in the RFC 6265. We will talk about HTTP *headers* in a short while and this is how cookies are passed back and forth (there is a `Set-Cookie` header sent by the server and a `Cookie` header sent by the client). For now, let us focus on the principle of cookie-based state management:

- When the client sends the initial request to a server (starting a session), the server generates a session id, puts it into a cookie and sends the cookie back to the client with its response.
- Whenever the client sends a subsequent request to the server, it **has to** send the cookie(s) that it has previously received from this server.

Various features and subtleties are specified in RFC 6265 and implemented by browsers. We will come back to them at a later stage, when we look at HTTP from an infrastructure point of view (dealing with reverse proxies and load balancers).

3.2. HTTP is Stateless, But Several Requests Can Be Handled in One TCP Connection

In the previous version of the protocol, i.e. in HTTP 1.0, the interaction between a client and a server was based on the use of a different TCP connection for every request-reply exchange. The client would establish a connection, send its request. The server would send its response and close the connection. Very often, the client would immediately establish a new connection with the same server, send a new request and receive a response. Why immediately? Think about typical web pages: the main content may be the HTML markup, but various resources are referenced in HTML tags: inline images, CSS stylesheets, javascript scripts, etc. Often, the main markup and the referenced resources are hosted on the same server, so a typical interaction between the client and the server looks like:

```
C: GET /beautifulPage.html HTTP/1.0
S: HTTP/1.0 200 OK
C: -- parsing the html so that I can render it…
C: -- looking for <img>, <link> and similar tags…
C: GET /img/banana.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/apple.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/pear.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/unicorn.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/grape.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/ananas.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /css/happy.css HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /js/app.js HTTP/1.0
S: HTTP/1.0 200 OK
C: -- ok… now I am able to fully render the page…
```

Is that a problem? Yes, it is and the HTTP specification tells us why:

> Prior to persistent connections, a separate TCP connection was established to fetch each URL, increasing the load on HTTP servers and causing congestion on the Internet. The use of inline images and other associated data often require a client to make multiple requests of the same server in a short amount of time. Analysis of these performance problems and results from a prototype implementation are available [26] [30]. Implementation experience and measurements of actual HTTP/1.1 (RFC 2068) implementations show good results [39].
>
> […]
>
> A significant difference between HTTP/1.1 and earlier versions of HTTP is that persistent connections are the default behavior of any HTTP connection. That is, unless otherwise indicated, the client SHOULD assume that the server will maintain a persistent connection, even after error responses from the server.
>
> Persistent connections provide a mechanism by which a client and a server can signal the close of a TCP connection. This signaling takes place using the Connection header field (section 14.10). Once a close has been signaled, the client MUST NOT send any more requests on that connection.

Source: Section 8

One of the sources mentioned in the RFC is available here and explains why the slow start algorithm used by TCP for congestion control has a negative impact on HTTP 1.0 performance.

To address this issue, HTTP 1.1 has introduced **persistent connections**. This means that a client can now establish a TCP connection, send a first request (e.g. to retrieve one HTML page), receive the response, send a second request (e.g. to retrieve and embedded image), receive the response, and continue like this until the TCP connection is finally closed. Furthermore, HTTP 1.1 has also introduced the notion of **pipelining**, allowing the client to send a first request and to immediately send a second request before having received the first response (under certain conditions).

4. HTTP is Used to Transfer Representations of Resources

In the introduction, we have already mentioned that HTTP is not only used to transport static (HTML) documents, but also to implement distributed systems and applications. For this reason, you should not say that "HTTP is used to transport Web pages", because it is not precise enough and misleading.

> While this topic is out of scope for this course, you should also be aware that REST APIs, based on the REST architectural style have become a very popular, if not the de facto standard, way to expose and consume web services. You should also be aware that the concepts that we are about to discuss are at the core of the REST approach.

4.1. Resources vs Representation of Resources

The HTTP protocol specification defines three key concepts that you really have to understand:

- Firstly, the RFC defines the notion of **resource** as "a network data object or service that can be identified by a URI". The point is that **the notion of resource is very generic and can apply to very different things**. An article published in an online newspaper is a resource. A list of articles in a given category is a resource. A stock quote updated in realtime is a resource. A webcam is a resource.

- Secondly, the RFC defines the notion of **representation** and states that what is transported in the protocol messages is not the resources themselves (*would it make sense to transport an actual webcam?*), but **representations of the resources**. You have to remember that for one resource, there might be several ways to represent it. Think of the stock quote example: you might want a graphical representation (PNG data showing the evolution of the price over time), a human-readable representation (HMTL data) or a machine-understandable representation (XML or JSON data).

- Thirdly, the RFC explains that HTTP supports **content negotiation** between clients and server (and describes how special headers provide the actual mechanism). We will get back to this notion in the next paragraph.

Here is an excerpt from the RFC, in the Terminology Section:

```
message
   The basic unit of HTTP communication, consisting of a structured
   sequence of octets matching the syntax defined in section 4 and
   transmitted via the connection.

request
   An HTTP request message, as defined in section 5.

response
   An HTTP response message, as defined in section 6.

resource
   A network data object or service that can be identified by a URI,
   as defined in section 3.2. Resources may be available in multiple
   representations (e.g. multiple languages, data formats, size, and
   resolutions) or vary in other ways.

entity
   The information transferred as the payload of a request or
   response. An entity consists of metainformation in the form of
   entity-header fields and content in the form of an entity-body, as
   described in section 7.

representation
   An entity included with a response that is subject to content
   negotiation, as described in section 12. There may exist multiple
   representations associated with a particular response status.

content negotiation
   The mechanism for selecting the appropriate representation when
   servicing a request, as described in section 12. The
   representation of entities in any response can be negotiated
   (including error responses).
```

Source: Section 1.3

4.2. Content Negotiation

Content negotiation applies to resource representation and works on the following principle:

- When sending a request to get the representation of a given resource, **the client expresses its capabilities and preferences** (which partly depend on the capabilities and preferences of the end-users behind the client). For instance, it may say: my preference would be to receive a JSON representation of the resource, but I could also do something with an XML or plain text representation. Or it could say: my preference would be to receive a representation of this resource in french, but I could also deal with

a representation in english or in german.

- When the server processes the request, it will **try to do its best to satisfy the client**. However, not it will not be able to fulfill the first choice of the user. As the outcome is uncertain, the server will indicate what he has been able to do when sending back the response.

In both cases, **content negotiation is achieved thanks to special headers**, which we will look at shortly when describing the syntax of HTTP requests and responses.

5. HTTP Messages: Requests and Responses

Now that we have seen what a typical HTTP exchange looks like and that we have presented important concepts underlying the protocol design, let us look at some details. Namely, let us look at the syntax of requests and responses, as well at some of the methods, headers and status codes defined in the RFC.

5.1. Syntax

The syntax of HTTP messages is specified in BNF form. We will only describe some of its elements here, you will find all details in the RFC.

The high-level constructs are defined by the following rules:

```
HTTP-message   = Request | Response     ; HTTP/1.1 messages

Request        = Request-Line              ; Section 5.1
                 *(( general-header        ; Section 4.5
                  | request-header         ; Section 5.3
                  | entity-header ) CRLF)  ; Section 7.1
                 CRLF
                 [ message-body ]          ; Section 4.3


Request-Line   = Method SP Request-URI SP HTTP-Version CRLF

Response       = Status-Line               ; Section 6.1
                 *(( general-header        ; Section 4.5
                  | response-header        ; Section 6.2
                  | entity-header ) CRLF)  ; Section 7.1
                 CRLF
                 [ message-body ]          ; Section 7.2

Status-Line    = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Source: Section 4.1, Section 5 and Section 6.

Here are some comments about these definitions:

- The requests and responses have the same basic structure: there is a **first line**, followed by a variable number of **header lines**, followed by a **blank line**, followed by a **message body** (*note: the message body is sometimes empty*).
- The first line of the **request** includes a **method** (which indicates what the client wants to do with the target resource) and a **URI** (which identifies the target resource).
- The first line of the **response** includes a **status code** and a **reason phrase** that inform the client of the outcome (i.e. whether the server was able to fulfill the request).
- The **header lines** start with a **header name,** followed by a **colon**, followed by a **header value**.
- The sequence of `CR LF` (ASCII(13) ASCII(10) or `\r \n`) is used to **separate lines**.
- The message body may contain **binary data** (e.g. image), so it MUST NOT be read line by line.
- The **protocol version** is indicated both in the request and in the response, on the first line.

5.2. Some Interesting Methods

The first line in an HTTP request contains what is called a **method**. We have said that the method is used to indicate what the client intends to do with the target resource. The HTTP specification defines a number of methods, but also suggests that other protocols may extend this list with their own methods (which is an approach that has been taken by the WebDAV protocol for instance).

The default HTTP methods are listed in Section 5.1.1 and fully specified in Section 9:

```
Method         = "OPTIONS"                ; Section 9.2
               | "GET"                    ; Section 9.3
               | "HEAD"                   ; Section 9.4
               | "POST"                   ; Section 9.5
               | "PUT"                    ; Section 9.6
               | "DELETE"                 ; Section 9.7
               | "TRACE"                  ; Section 9.8
               | "CONNECT"                ; Section 9.9
               | extension-method
extension-method = token
```

Source: Section 5.1.1.

From this list, the most important ones are the following:

- `GET`, which is used by the client to request (i.e. read) the representation of the target resource. When you type a URL in the browser navigation bar or when you click on a hyperlink, this is what is sent to the server. The request body is empty.
- `POST`, which is used by the client to send data to the target resource. When you fill out an HTML form and hit the submit button, a POST request is issued and the content of the form is sent as the request body. (Note: we have mentioned REST APIs before; without digging into the details, POST requests are used to create data).
- `PUT`, which is used by the client to send the representation of a resource in order to replace the target resource. This is not something that you can typically do with a web browser, but which is used by other types of HTTP clients (in REST APIs, PUT requests are used to update data).
- `DELETE`, which is used by the client to request the deletion of the target resource (again, this is not something that you can usually do in a web browser but that that other types of clients can do).

To be complete, we should also mention that an additional method, PATCH, has been defined in RFC 5789. It is used to do partial updates of resources and its usage is growing (because of the popularity of REST APIs and of related needs).

5.3. Some Interesting Headers

The standard HTTP headers are described in Section 14. We will only describe some of them, in relation to the key protocol concepts that we have presented before.

- the `Accept` request header and the `Content-type` header are used for **one aspect of the content negotiation**. In the `Accept` header, the client indicates the media type that it is able to handle (e.g. `text/html`, `text/plain`,

`application/json`, `image/png`). In the `Content-type` header, the server indicates what it has been able to do and how the message body should be interpreted.

- the `Accept-Language` and `Content-Language` headers are used for **another aspect of the content negotiation**, allowing the client to indicate a preference for a representation in French and allowing the server to indicate that it is sending a representation in Schwyzerdütsch.

- the `Accept-Charset` and the `Content-type` headers are used for **yet another aspect of the content negotiation**, namely the preferred character encodings.

- the `Content-Length` header is used to indicate the **length of the message body**. This allows the client or the server receiving the message to know how many bytes must be read before reaching the next message. Note that this header may not always be provided, we will get back to this when looking at the ways to parse HTTP messages (see Section 4.4 in the RFC).

- the `Host` header, which is mandatory since HTTP 1.1 and which is extremely important to enable **virtual hosting**. We will get back to the notion of virtual hosting in an upcoming lecture.

- the `Authorization` header, which is used when accessing **protected resources**. We will get back to security aspects in an upcoming lecture.

- the `Transfer-Encoding` header, which is used to indicate that the message body has been transformed in order to transfer it in a particular way. For instance, it is possible for the message body to be compressed before being transferred. Also, in the case of dynamic content, the message body may be structured in multiple *chunks* (more on this later).

- the `If-Modified-Since`, `Last-modified`, `ETag`, `Expires` headers that are related to caching, a notion that we will look at in an upcoming lecture.

Note: a nice summary of HTTP headers, with references to the HTTP RFC is available here.

5.4. Some Interesting Status Codes

The status codes that are used by the server to indicate whether the request could be fulfilled or not are listed in Section 6.1.1 and documented in Section 10.

```
The first digit of the Status-Code defines the class of response. The
last two digits do not have any categorization role. There are 5
values for the first digit:

   - 1xx: Informational - Request received, continuing process
```

```
   - 2xx: Success - The action was successfully received,
     understood, and accepted

   - 3xx: Redirection - Further action must be taken in order to
     complete the request

   - 4xx: Client Error - The request contains bad syntax or cannot
     be fulfilled

   - 5xx: Server Error - The server failed to fulfill an apparently
     valid request

Status-Code    =
      "100"  ; Section 10.1.1: Continue
    | "101"  ; Section 10.1.2: Switching Protocols
    | "200"  ; Section 10.2.1: OK
    | "201"  ; Section 10.2.2: Created
    | "202"  ; Section 10.2.3: Accepted
    | "203"  ; Section 10.2.4: Non-Authoritative Information
    | "204"  ; Section 10.2.5: No Content
    | "205"  ; Section 10.2.6: Reset Content
    | "206"  ; Section 10.2.7: Partial Content
    | "300"  ; Section 10.3.1: Multiple Choices
    | "301"  ; Section 10.3.2: Moved Permanently
    | "302"  ; Section 10.3.3: Found
    | "303"  ; Section 10.3.4: See Other
    | "304"  ; Section 10.3.5: Not Modified
    | "305"  ; Section 10.3.6: Use Proxy
    | "307"  ; Section 10.3.8: Temporary Redirect
    | "400"  ; Section 10.4.1: Bad Request
    | "401"  ; Section 10.4.2: Unauthorized
    | "402"  ; Section 10.4.3: Payment Required
    | "403"  ; Section 10.4.4: Forbidden
    | "404"  ; Section 10.4.5: Not Found
    | "405"  ; Section 10.4.6: Method Not Allowed
    | "406"  ; Section 10.4.7: Not Acceptable
    | "407"  ; Section 10.4.8: Proxy Authentication Required
    | "408"  ; Section 10.4.9: Request Time-out
    | "409"  ; Section 10.4.10: Conflict
    | "410"  ; Section 10.4.11: Gone
    | "411"  ; Section 10.4.12: Length Required
    | "412"  ; Section 10.4.13: Precondition Failed
    | "413"  ; Section 10.4.14: Request Entity Too Large
    | "414"  ; Section 10.4.15: Request-URI Too Large
    | "415"  ; Section 10.4.16: Unsupported Media Type
    | "416"  ; Section 10.4.17: Requested range not satisfiable
    | "417"  ; Section 10.4.18: Expectation Failed
```

```
    | "500"  ; Section 10.5.1: Internal Server Error
    | "501"  ; Section 10.5.2: Not Implemented
    | "502"  ; Section 10.5.3: Bad Gateway
    | "503"  ; Section 10.5.4: Service Unavailable
    | "504"  ; Section 10.5.5: Gateway Time-out
    | "505"  ; Section 10.5.6: HTTP Version not supported
    | extension-code
```

Some comments can be made:

- remember that a status code starting with **4** indicates that the client responsible for the error, while a status code starting with **5** indicates that the server is responsible.
- If everything went well, you will get a **200**.
- **401** and **403** are related to security. **401** is used when the server does not know the identity of the client or user and asks for **authentication**. **403** is used when the server knows the identify of the client or user but decides that the access should not be **authorized**.
- **301** and **302** are used to indicate that the target resource has moved. The client should send a new request, using the URL provided in the `Location` header.
- **304** is used to indicate that the resource has not been modified since last request. Therefore, the server is not sending any representation and the client can reuse what it has previously received.

6. Parsing HTTP Messages

Whether you are implementing a HTTP server or a HTTP client, you will need to parse HTTP messages. In other words, you will receive a series of bytes from a socket and will need to convert it into either an HTTP request or an an HTTP response. The procedure is pretty straightforward, but there are a couple of possible variations:

- You should **read bytes, not characters** (because the payload may be binary data and you would not want this data to be interpreted as characters and modified).
- When **reading line by line** (at the beginning of the process), you should read bytes until you find the sequence of CR (ASCII 13, `\r`) and LF (ASCII 10, `\n`) bytes.
- You should start by **reading the first line** (request line or status line). You should extract the tokens (method, URI, protocol version, status code or reason phrase).
- You should then **read all of the headers**, line by line, until you find an empty line (in other words, until you find a sequence of CR LF CR LF bytes). You should keep these

headers in a data structure (e.g. a map).

- Depending on the situation, the next step will vary (the details are specified in Section 4.4 of the RFC 2616):
    - If you have found a `Content-length: n` header, then you have to read `n` bytes from the input stream.
    - If you have found a `Transfer-Encoding: chunked` header, then you have to start by reading the size of the first chunk or response. You will find it on the first line, in hexadecimal value. You will then read this amount of bytes, before finding a new line with the size of the next chunk. You will go through this process until you find a chunk size of 0. The RFC provides pseudo code for this algorithm in Section 19.4.6 (note that headers might appear after the last chunk, which we will not discuss here to keep things simple).
    - If you don't have any indication about the length of the message, then you have to assume that the process on the other side will close the TCP connection when it has sent all of the content (this is how HTTP 1.0 servers used to work). Therefore, you should read bytes until you detect the end of the stream.
    - The RFC also specifies a special case when the `multipart/byteranges` media type is used, but we will not discuss it here. The details are available in Section 19.2.

Resources

MUST read

- These Sections of the RFC 2616:
    - Section 1
    - Section 4 and in particular Section 4.4
    - Section 19.3
    - Section 5
    - Section 6
    - Section 8.1.1
    - Section 8.1.2
    - Section 19.3
    - Section 19.4.6

Additional resources

- The rest of the HTTP RFC

- History of the HTTP protocol specification
- History of the Web, going back to its roots

What Should I Know For The Test and The Exam?

Here is a **non-exhausive list of questions** that you can expect in the written tests and exams:

- Write a complete HTTP request and a complete HTTP response. Explain the structure and the role of the different lines and of the different elements on each line.
- Explain **how** and **why** TCP connections are handled differently in HTTP/1.0 and in HTTP/1.1.
- Explain the notion of **content negotiation** in HTTP and illustrate it with an example (include an HTTP request and an HTTP response in your explanation).
- What does it mean when we say that HTTP is a **stateless** protocol? Why is that sometimes a problem (give an example of an application where this is a problem) and what can be done about it?
- Why is the **chunked transfer encoding** useful (what would be the problem if we did not have it?)? Explain what a client needs to do in order to parse a response that uses the chunked transfer encoding.
- What are **cookies** in the context of the HTTP protocol? Explain why they are useful. Explain how they work at the protocol level (explain what the client and the server have to do, give examples of requests and responses).