Lecture 14 - HTTP (last updated: 10/26/22)
---

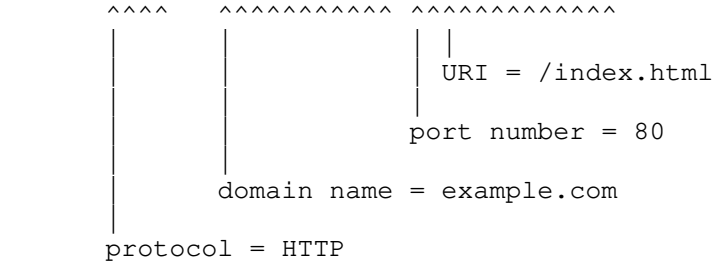
In the previous lecture, we saw how TCP connections help us establish a reliable, bidirectional channel of communication between two peers. In this lecture, we will see how web infrastructure builds upon those connections to allow web browsers to communicate with web servers.

In particular, we will learn about HTTP (HyperText Transfer Protocol). This application-level protocol allows HTTP clients (e.g., web browsers) to request resources (e.g., HTML files) from HTTP servers over a TCP connection.

We will focus our discussion on HTTP version 1.0 (usually written HTTP/1.0). Note that some of the technical details in this lecture note are outdated relative to the latest HTTP version (HTTP/3.0), and do not include discussion of secure HTTP (i.e., HTTPS). However, these modern protocols build off of many of the same principles found in HTTP/1.0. Besides, HTTP/1.1 (a direct extension of HTTP/1.0) is still widely in use today.

## Anatomy of URLs

When we visit a website, we must specify its URL (Uniform Resource Locator), which has the following format:




The browser resolves the domain name to an IP address, which it uses along with the port number to establish a TCP connection with the web server. You can also explicitly specify an IP address instead, to forego the DNS lookup:

```
http://93.184.216.34:80/index.html
```

The port number can be inferred from the protocol (e.g., the default port for HTTP is port 80), so it is usually omitted, i.e.:

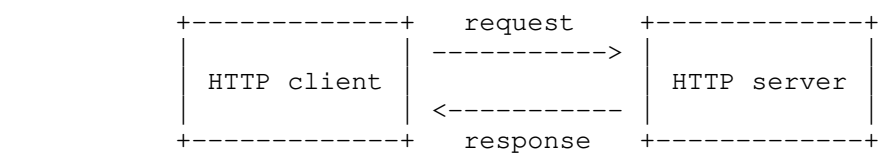
```
http://example.com/index.html
```

```
http://93.184.216.34/index.html
```

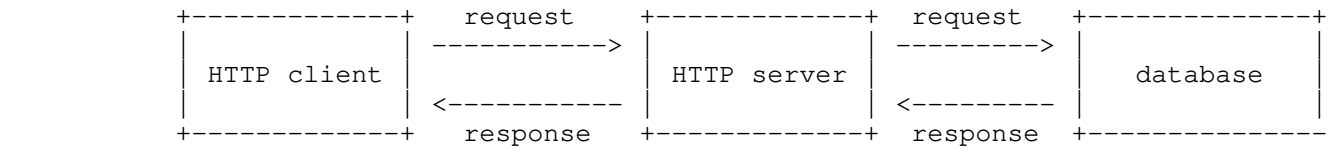
However, you will need to explicitly include the port number if you want to connect with a server on any other port.

The URI (Uniform Resource Identifier) tells the server what resource you want. It's important to keep in mind that although "/index.html" looks like an absolute file path, we're not really asking the server for a file named "index.html" in / (the root directory). It's up to the server to interpret the URI and decide what resource to serve the client (if any).

Servers will commonly append the URI to some pre-configured "web root" directory where website resources are stored as files. For example, when a client requests /index.html from a server whose web root is /home/www/web, the server will respond with the contents of the file located at /home/www/web/index.html. Note that these servers serve file content directly from the file system; we describe these kinds of servers as "static web servers."




On the other hand, "dynamic web servers" may generate resource content on the fly. For example, a server may perform a database query on behalf of the client (perhaps over another internal TCP connection), and generate a web page based on the results of that query:




This three-tiered architecture is useful for separating the concerns of the client, server, and database, allowing them to be developed and maintained more independently.

## Browsing hypertext

Textual web content is mostly written in HTML (HyperText Markup Language), which your browser knows how to render and display as a webpage. When you visit a website (e.g., http://example.com/index.html), your web browser retrieves the HTML file from the server according to the following steps:

- Your browser establishes a TCP connection with the server, using the domain name/IP address and port number inferred from the URL
- Your browser sends an HTTP request to the server over that TCP connection, asking for a resource named /index.html
- The server responds to the browser's request (over the same TCP connection), either fulfilling it (e.g., sending back the contents of /index.html), or denying it (e.g., because /index.html does not exist)

HTML files only contain text, but they may refer to other resources (such as images, stylesheets, scripts, or other HTML files). Your browser parses the HTML file, obtains the URLs for those other resources, and makes subsequent HTTP requests to retrieve those resources. It then renders the webpage according to those resources.

## Command-line HTTP clients

We normally use browsers to make HTTP requests, but we can also use command-line clients such as curl or wget:

```
$ curl http://example.com/index.html    # print page to stdout
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

... ((truncated for brevity)) ...


$ wget http://example.com/index.html    # saves page to disk

$ cat index.html
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

... ((truncated for brevity)) ...
```

You can pass the --http1.0 flag to curl to force it to use HTTP/1.0 (at this time of writing, my version of curl defaults to HTTP/1.1).

These tools are useful for performing simple HTTP requests without the complexity of a full-featured web browser, and are often used in shell scripts.

## GET requests over HTTP/1.0

Over a TCP connection, an HTTP client and server exchange bytes according to a syntax specified by HTTP. The protocol supports several "methods," including:

- GET: the client downloads a resource from the server
- POST: the client uploads a resource to the server

We will focus on GET requests, though the syntax for other methods follows a similar structure.

A GET request consists of a request line, followed by zero or more headers, concluding with a blank line. For example, a GET request might look like this (not including my annotations on the right, starting from the "#"):

```
GET /index.html HTTP/1.0            # request line
Host: clac.cs.columbia.edu:80       # header
User-Agent: curl/7.83.1             # header
Accept: */*                         # header
                                    # blank line
```

A few things to note about the request:

- The request line contains the method (GET), the URI (/index.html), and the HTTP request version (HTTP/1.0); all these values are space-separated.
- Each header consists of a field name, a colon, and the field value.

- Each line, including the blank line, ends with a carriage return followed by a newline, i.e., "\r\n" (written as a C string literal). Note that carriage returns are not visible on the terminal when followed by a newline.

The server responds after seeing the blank line concluding the GET request. A GET response consists of a status line, zero or more headers, a blank line, and the contents of the requested resource. For example, this is a response from CLAC's HTTP server (with my annotations; some headers omitted for brevity):

```
HTTP/1.1 200 OK                      # status line
Server: Apache/2.4.52 (Ubuntu)       # header
Content-Type: text/html              # header
                                     # blank line
<!DOCTYPE html>                      # resource content
                                     # resource content
<head>                               # resource content
    <title>Welcome to CLAC</title>   # resource content

... ((truncated for brevity)) ...
```

The server completes the session by closing the TCP connection.

A few things to note about the response:

- The status line contains an HTTP version (HTTP/1.1), a status code (200), and an optional reason phrase (OK); all these values are space-separated.
- The server does not have to respond with the exact same HTTP version as the HTTP client; this is ok since HTTP/1.1 is backwards-compatible.
- Response headers follow the same syntax as request headers.
- The status line, headers, and blank line all end with a carriage return followed by a newline, like in the request.
- However, the contents of the resource do NOT need to have lines terminated with a carriage return and a newline; the contents of the resource should be treated as arbitrary binary data, for the purposes of the HTTP session.

If you want curl to show you its HTTP session with the server, use the -v flag.

## HTTP status codes

Sometimes, clients may make requests that an HTTP server is not able to fulfill. For instance, a client may ask for a resource that doth not exist. Instead of 200 OK, the server may respond with other HTTP status codes to convey these issues to the client.

The first digit indicates the category of status code, while the latter digits convey a more specific scenario. Here are some common, standard status codes:

- 2xx: the request was successful
  - 200 OK: the standard response for indicating success
- 3xx: the client must take further action for the request to be fulfilled
  - 301 Moved Permanently: the request should be directed to another URI
- 4xx: there was an issue with the client's request
  - 400 Bad Request: there was a problem
  - 403 Forbidden: the client doth not have necessary permissions
  - 404 Not Found: the client requested something that doth not exist

- 5xx: there was an issue with the server
    - 500 Internal Server Error: an unspecified server-side error
    - 501 Not Implemented: the server does not implement the request method

## HTTP/1.1 and beyond (optional)

One shortcoming of HTTP/1.0 is that the client needs to make a new TCP connection for each HTTP request. For example, if it downloads an HTML page containing 200 images, then it will need to make 200 subsequent TCP connections to the server in order to retrieve those images. Time spent establishing each separate connection adds up to a lot of unnecessary overhead.

HTTP/1.1 solves this problem by allowing the client to reuse the same TCP connection to issue multiple requests to the server. Though this complicates the protocol, it vastly improves amortized request latency by eliminating the handshake that takes place prior to each TCP connection.

HTTP/2 further tackles latency issues by revising the request/response format. In particular, it cuts down on the number of bytes transmitted by allowing HTTP headers to be compressed.

HTTP/3, which was only finalized recently, foregoes TCP connections altogether and communicates over the faster but less reliable UDP (User Datagram Protocol).