COMP 206 – Software Systems

Nov 14th, 2018

Lecture 19 – The World Wide Web

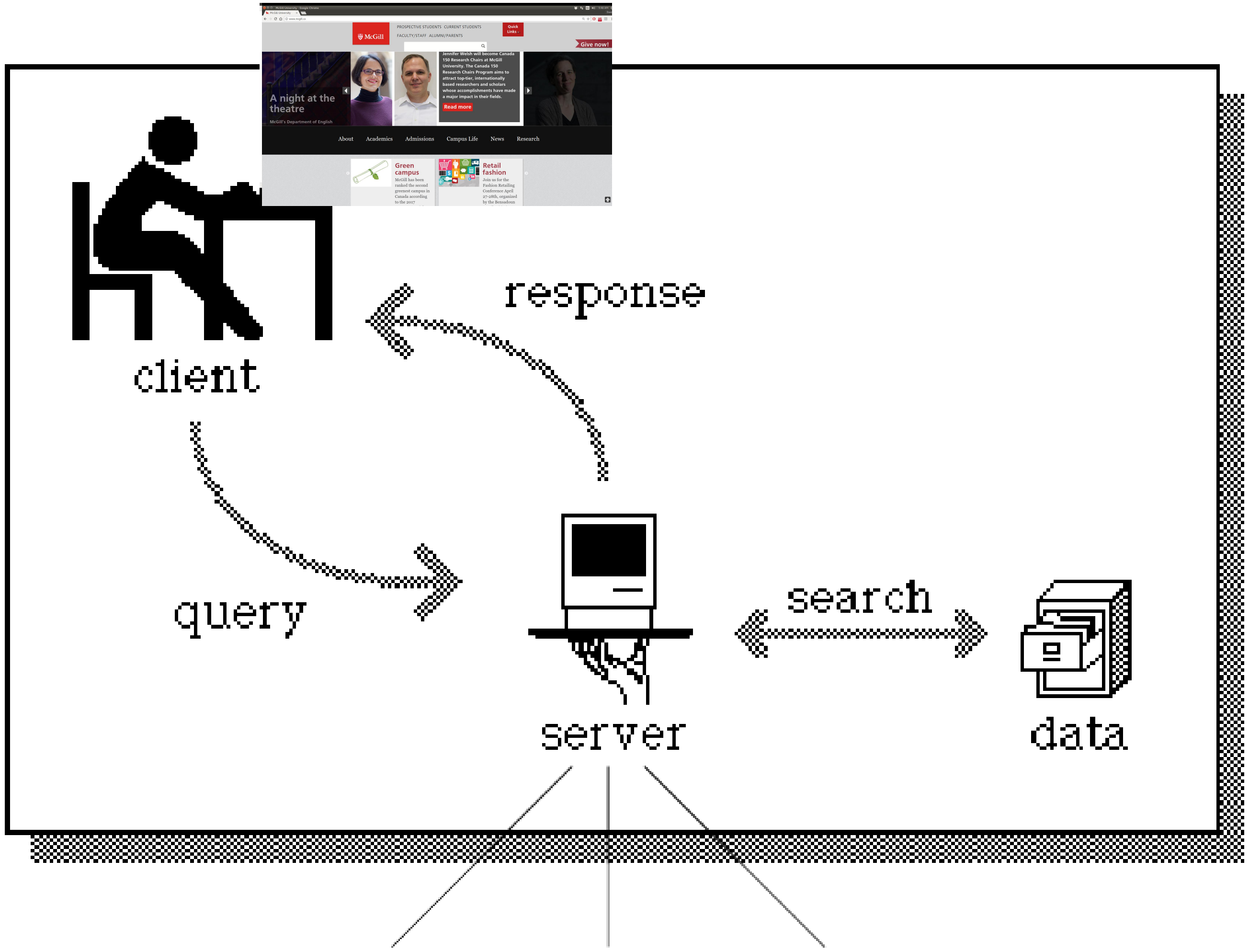
Plan today

- Proof that 206 allows you to be a web programmer!

- The Web is an application layer on the Internet
  - It uses what we know about sockets, IP, TCP to move data between servers and browsers
  - Explanation of the protocols, connection to concepts we know well
  - Simple C code examples of a web browser and web server

- To achieve its goals, the Web defines a few new protocols:
  - HTTP (today)
  - HTML (today)
  - CGI (next lecture)
  - Many more that we don’t cover here in 206

The Web as a Software System



McGill University - Google Chrome

McGill University

www.mcgill.ca

PROSPECTIVE STUDENTS CURRENT STUDENTS FACULTY/STAFF ALUMNI/PARENTS

Quick Links

Give now!

A night at the theatre

McGill's Department of English

Jennifer Welsh will become Canada 150 Research Chairs at McGill University. The Canada 150 Research Chairs Program aims to attract top-tier, internationally based researchers and scholars whose accomplishments have made a major impact in their fields.

Read more

About Academics Admissions Campus Life News Research

Green campus

McGill has been ranked the second greenest campus in Canada according to the 2017

Retail fashion

Join us for the Fashion Retailing Conference April 27-28th, organized by the Bensadoun

Read Query

Execute Query

Construct a Response

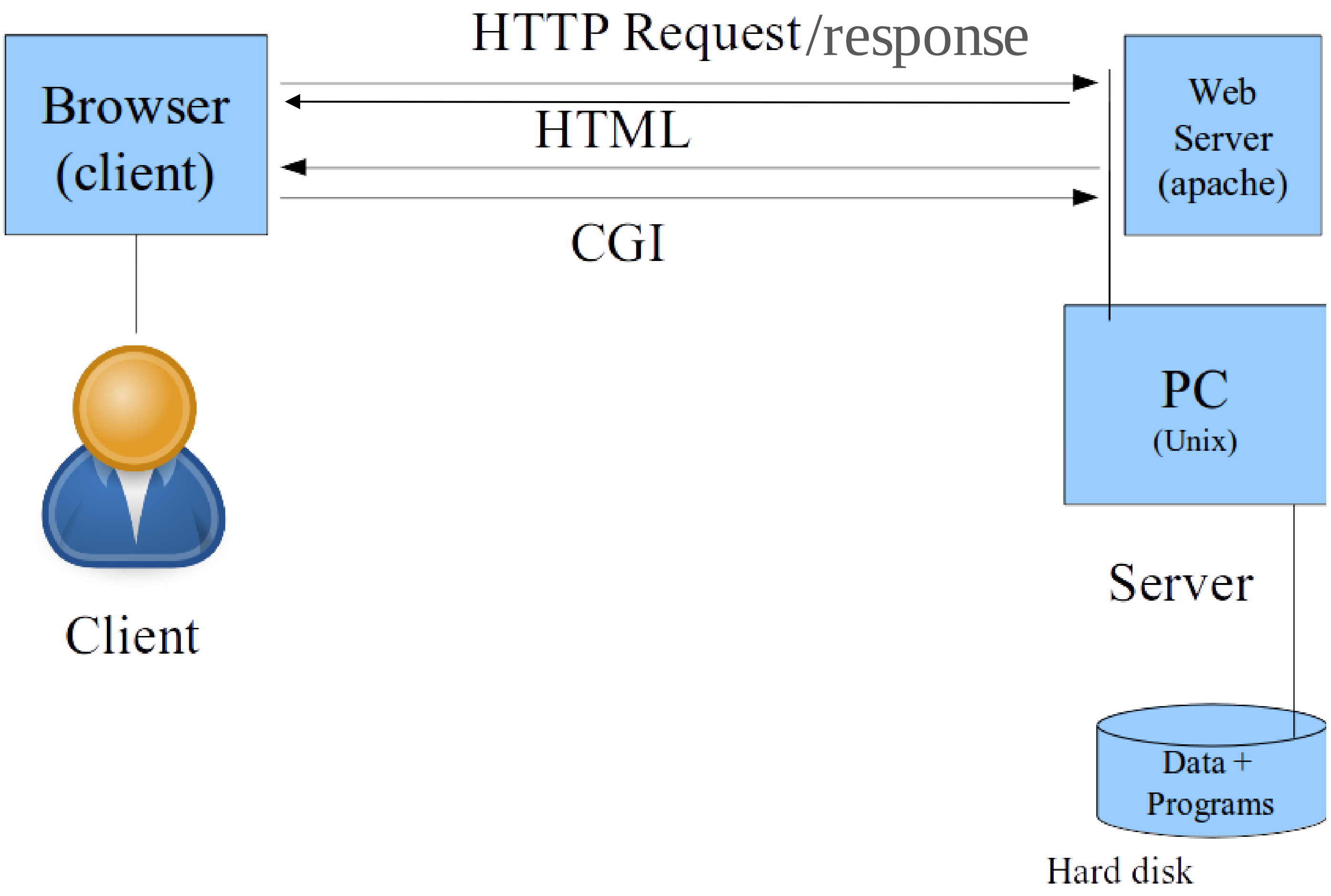
World Wide Web

- The web is one of the most pervasive application-level software systems on the internet

- It's formed out of a collection of quite simple protocols that everyone can easily implement, so that many people, companies, governments, etc can interoperate even without using the same software:
  - Many browsers: chrome, safari, edge, lynx

- Let's start to understand those protocols and write some code to work with them

Web Software Protocols



Overview of Web Protocols

- HyperText Transmission Protocol (HTTP) : the backbone for web inter-connections. Describes how to request and interpret basic web data
- Hyper Text Markup Language (HTML) : the basic building-block of web pages. Text, formatting, colors, images, extensible components for new ideas (3D graphics, UIs, etc)
- Common Gateway Interface (CGI) : A protocol to allow server-side programs to be run through web interactions, producing output that drives subsequent interactions
- Many more elements now exist on the client side: javascript, css, flash, mobile application eco-systems, and many more (not for 206)

HyperText Transfer Protocol (HTTP)

- TCP/IP allows end-points to transfer arbitrary binary data streams.
- This is like giving the computer a mouth and ears, but without a language, how can we write code that accomplishes anything?
- HTTP is the "language" of web connections. It lets browsers ask for web content and lets servers provide it. It is an open, evolving, community-defined standard that all web entities use to structure their transmissions so that each side can understand:
  - https://tools.ietf.org/html/rfc7230#section-2.1

HTTP and Networking

- HTTP's job is to allow clients to request the resources they desire and to allow servers to respond appropriately.
- It lives on top of lower network layers. The request and response are plain-text payloads with a specific format that must be transmitted somehow (of course we all use TCP/IP, through sockets!)
- A HTTP client establishes a connection over a predefined port :
  - 80 for normal HTTP
  - 443 for SSL HTTP (secure)
- An HTTP server sits waiting on the port, parses the client request and responds (perhaps on a different port, but let's ignore that for now)

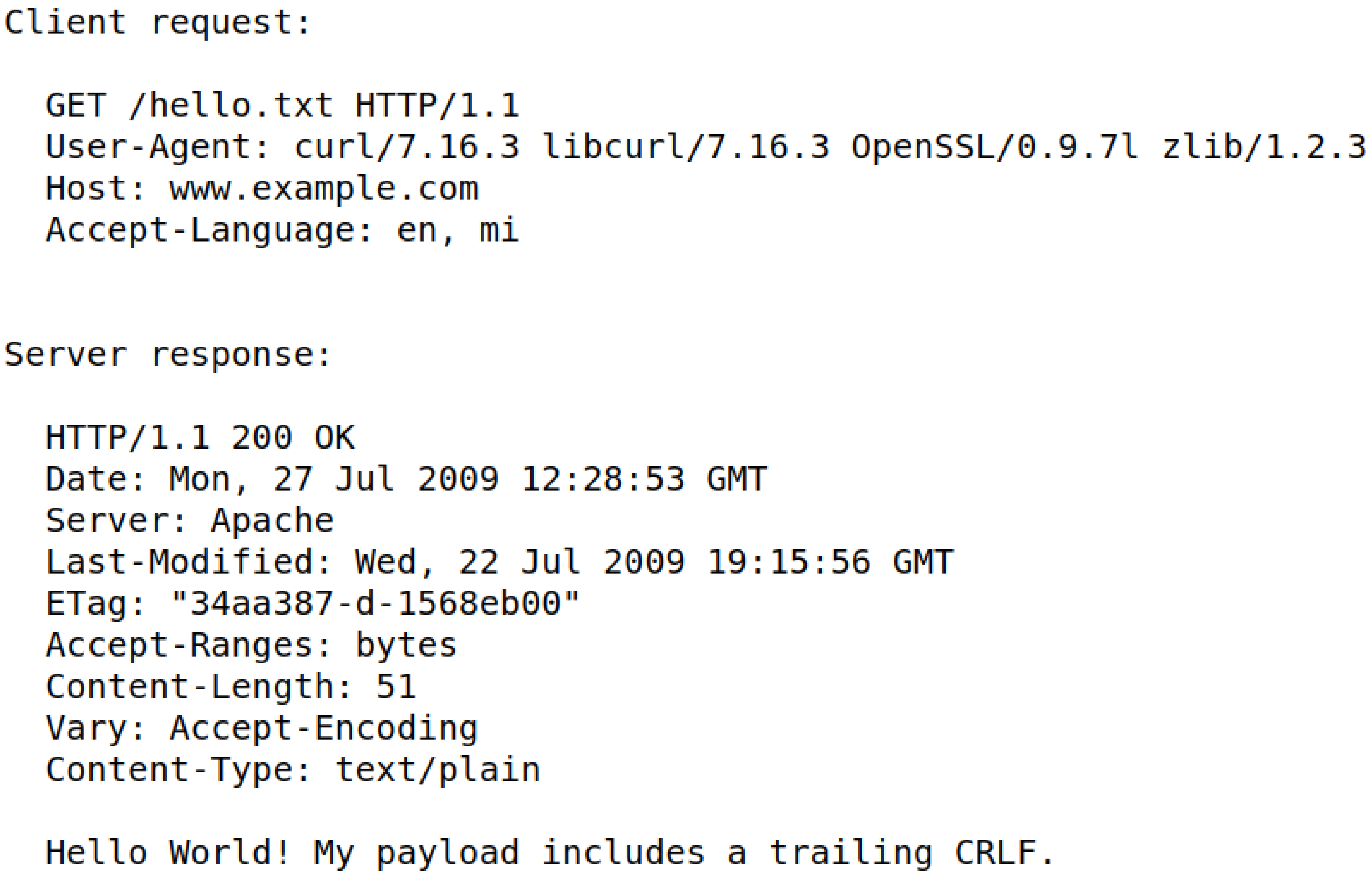
HTTP Requests and Responses

- Client starts the communication by sending an HTTP request that includes a ***METHOD, Uniform Resource Identifier (URI),*** protocol version, and header:
  - The most common method is GET, this means "Send me back the page"
- The server sends back a ***response code,*** response text, header, followed by the requested data, if any. Common response codes:
  - 200: OK
  - 401: Unauthorized
  - 403: Forbidden
  - 404: Not Found
  - 500: Internal Server Error

URI vs URL

- We are used to typing Uniform Resource Locators (URLs) into our web browser. They include:
  - the server name/address,
  - optionally, the port if we care to switch away from the default 80/443 (separate by colon, like mimi.cs.mcgill.ca:5000)
  - and then describes the path to a file on that server.
- The URI can be the portion only ***after the server address and port***, while the URL must include all of the above
  - It makes sense only to need to URI within an HTTP request, because we are sending this data to a server on a port (full URL re-dundant)
  - Unless the server is having an identity crisis (midterms are hard ok!)

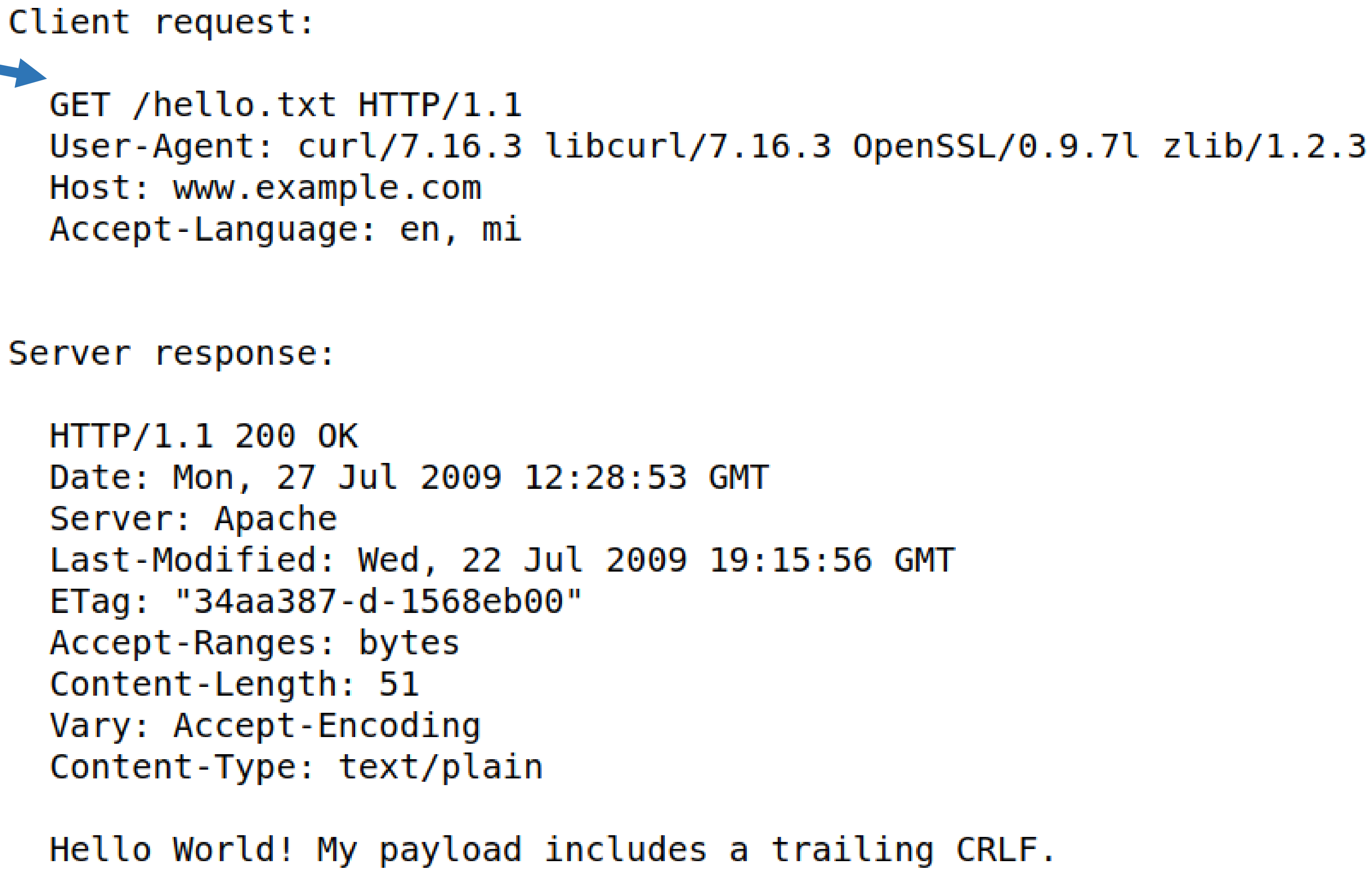
HTTP Session Example



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

HTTP Session Example

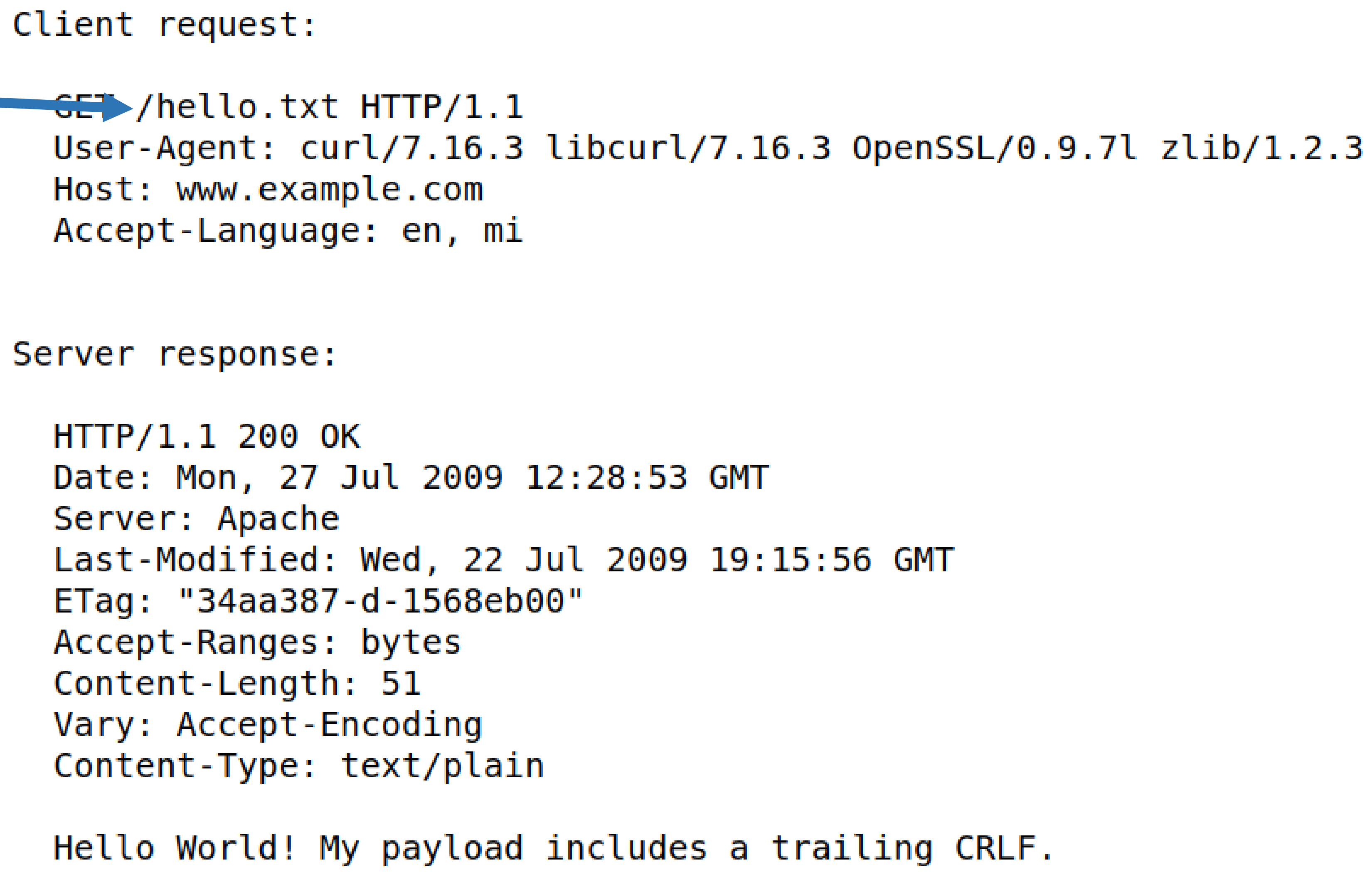
METHOD,
Uniform Resource Identifier (URI),
protocol version
header



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

HTTP Session Example

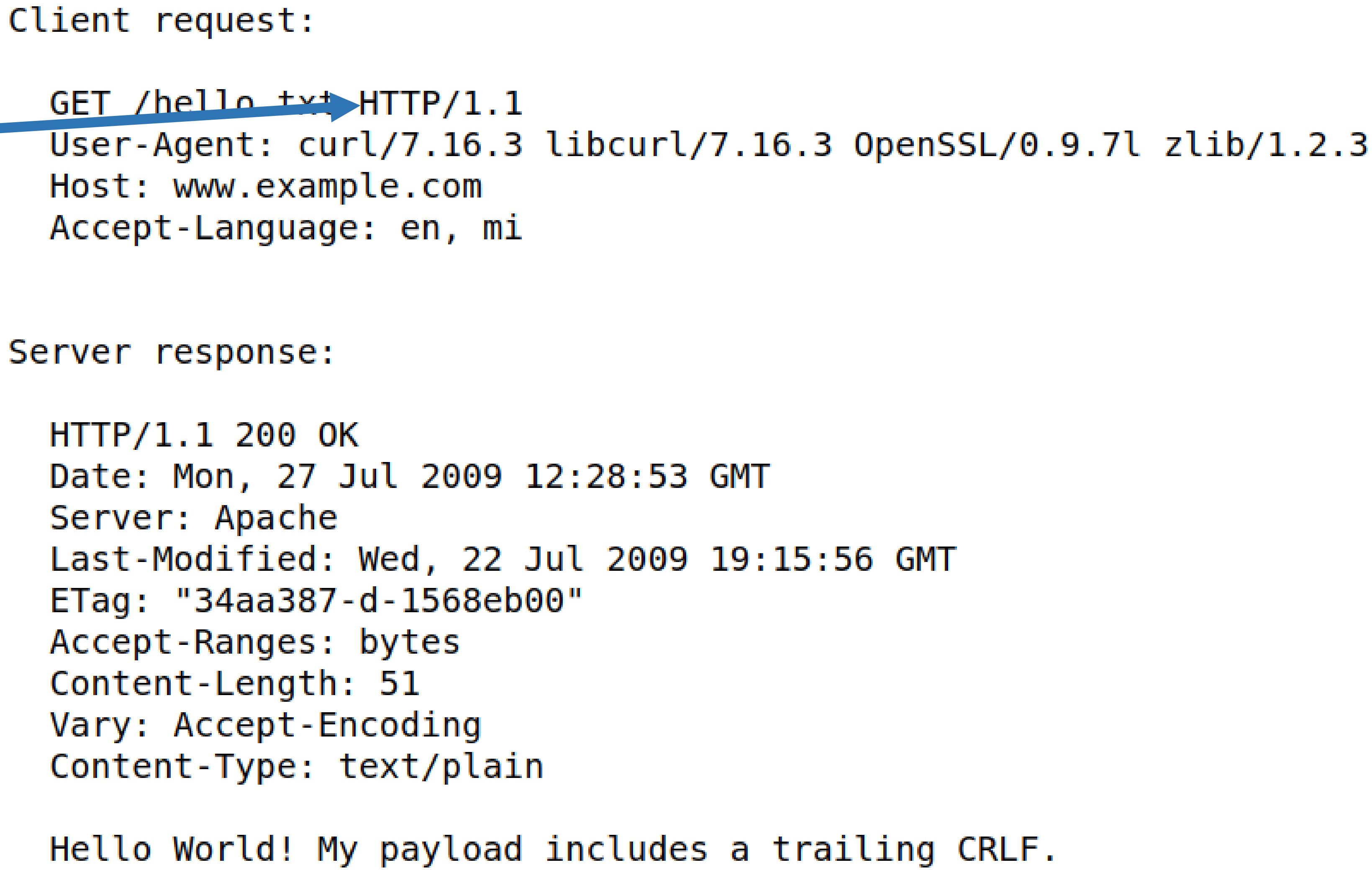
METHOD,
Uniform Resource Identifier (URI),
protocol version
header



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

HTTP Session Example

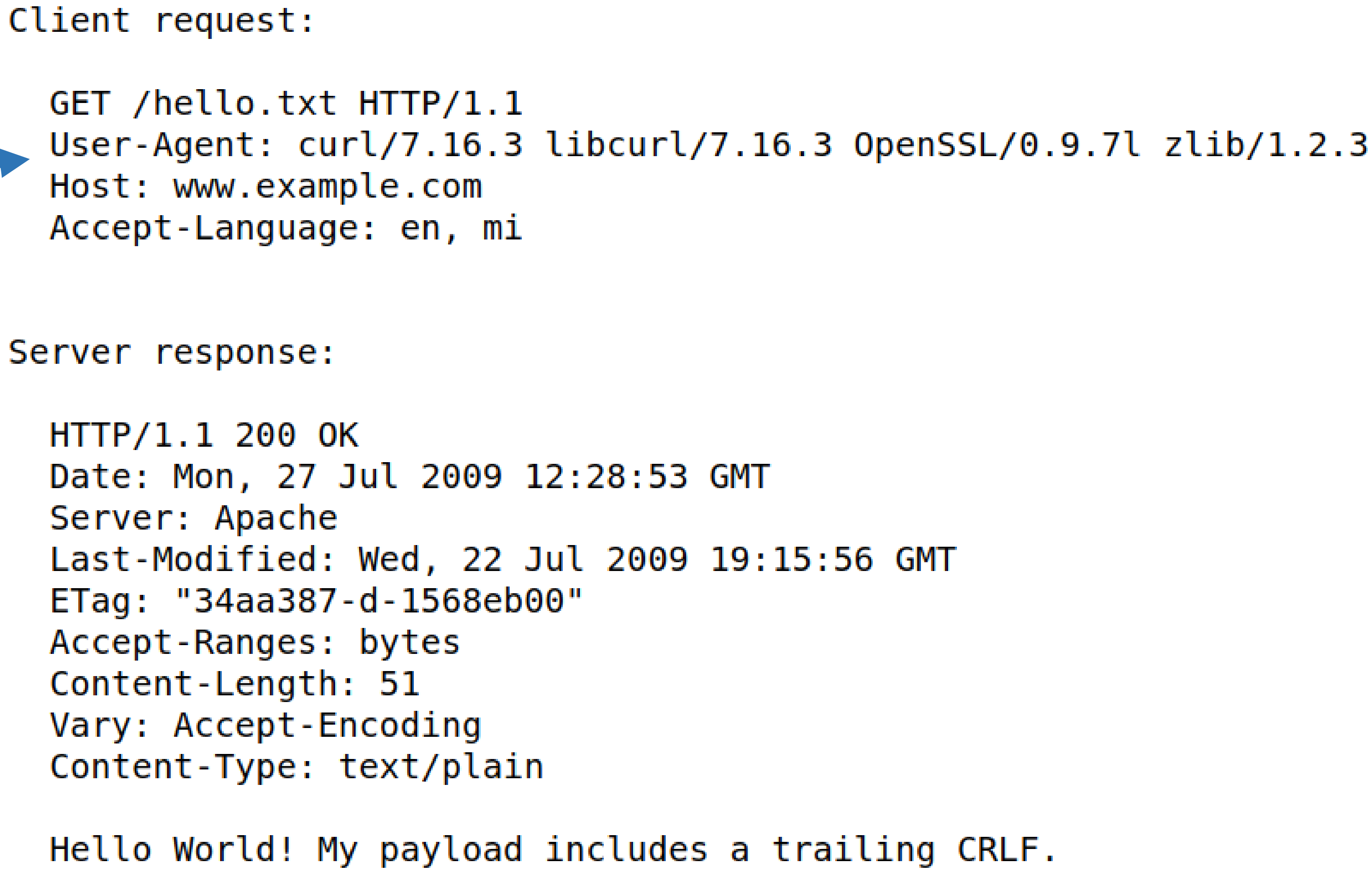
METHOD,
Uniform Resource Identifier (URI),
protocol version
header



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

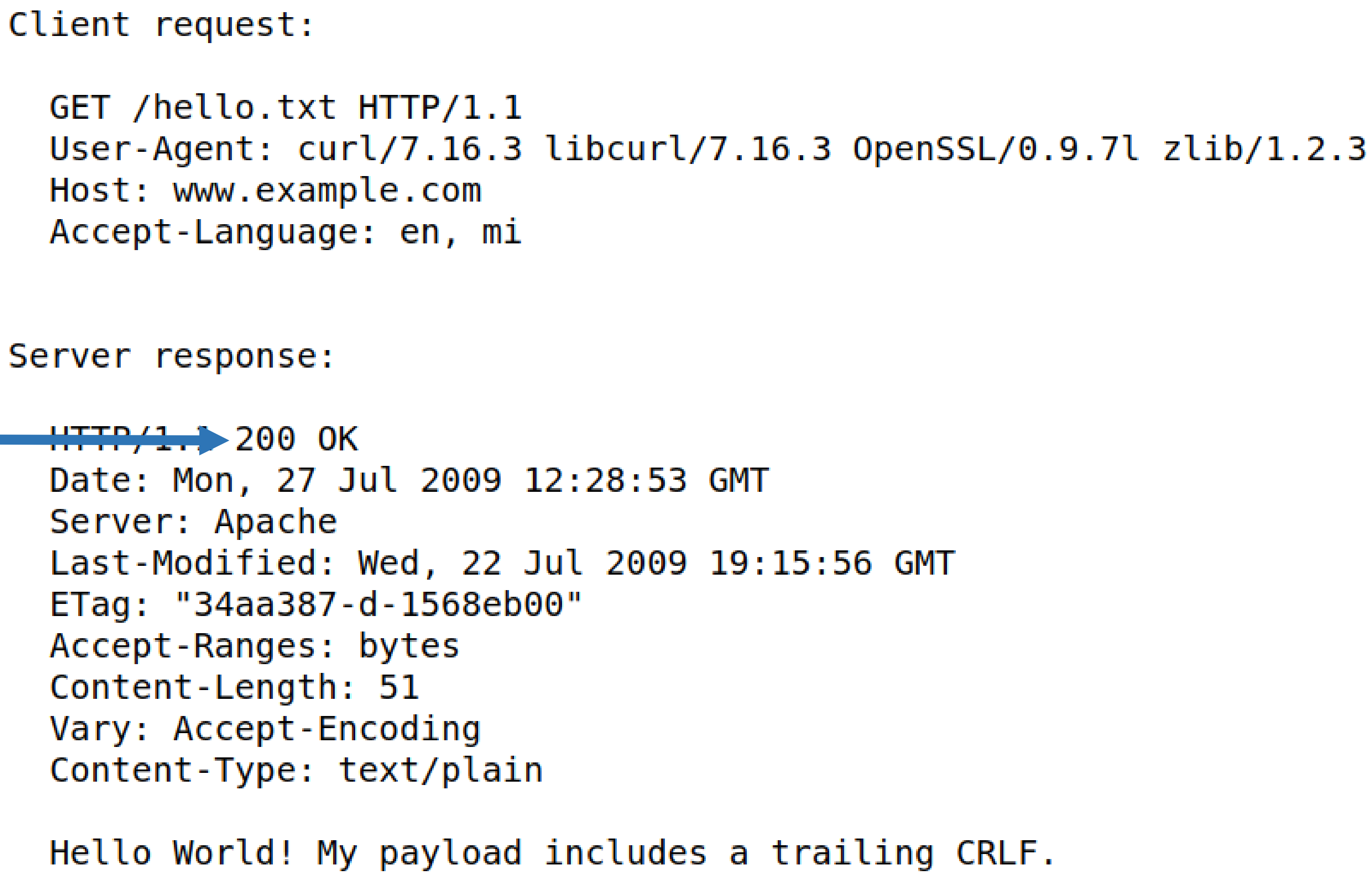
HTTP Session Example

METHOD,
Uniform Resource Identifier (URI),
protocol version
Headers (everything else)



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

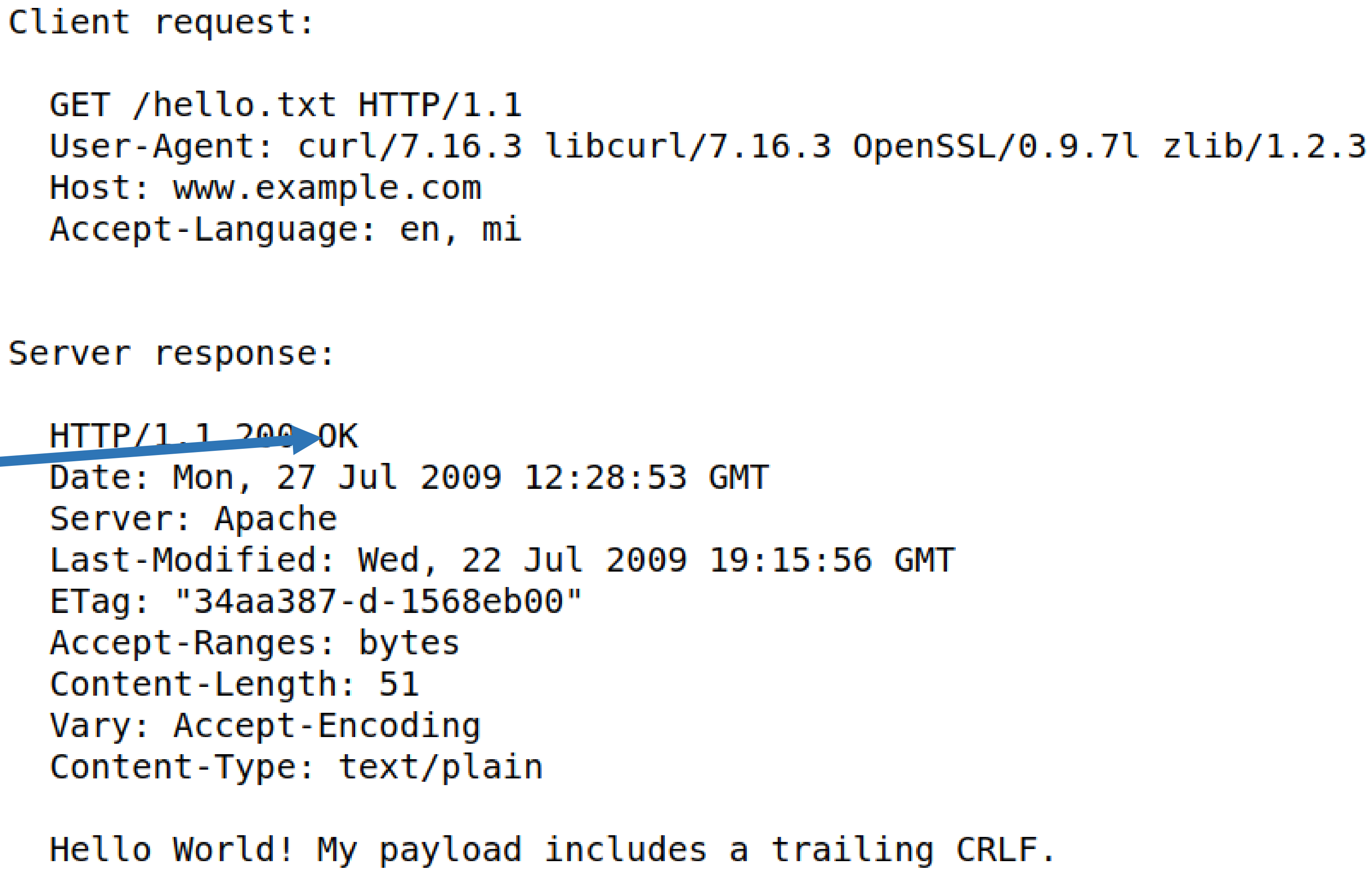
HTTP Session Example



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

response code
response text
header,
the requested data

HTTP Session Example

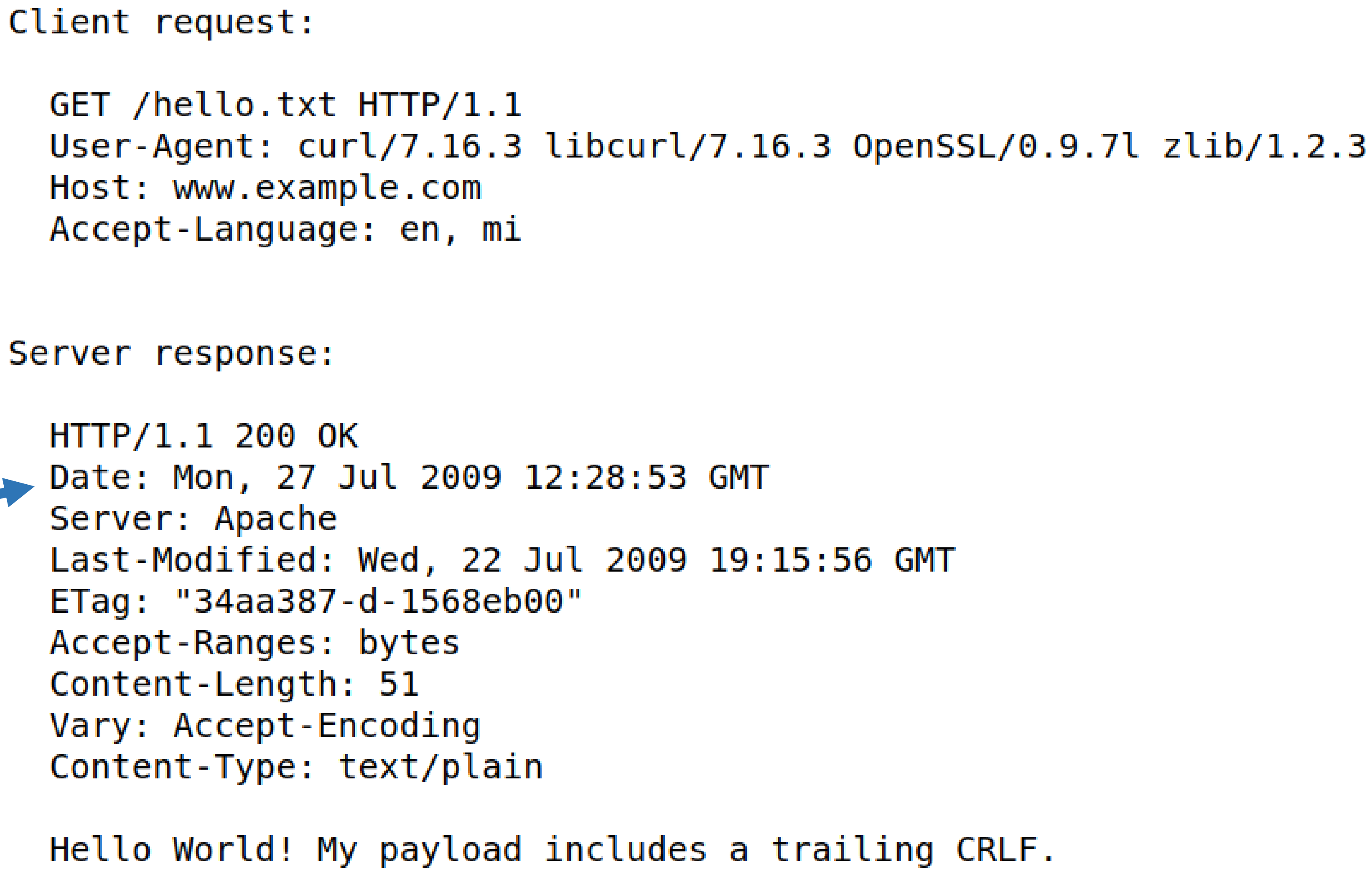


```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

response code
response text
header,
the requested data

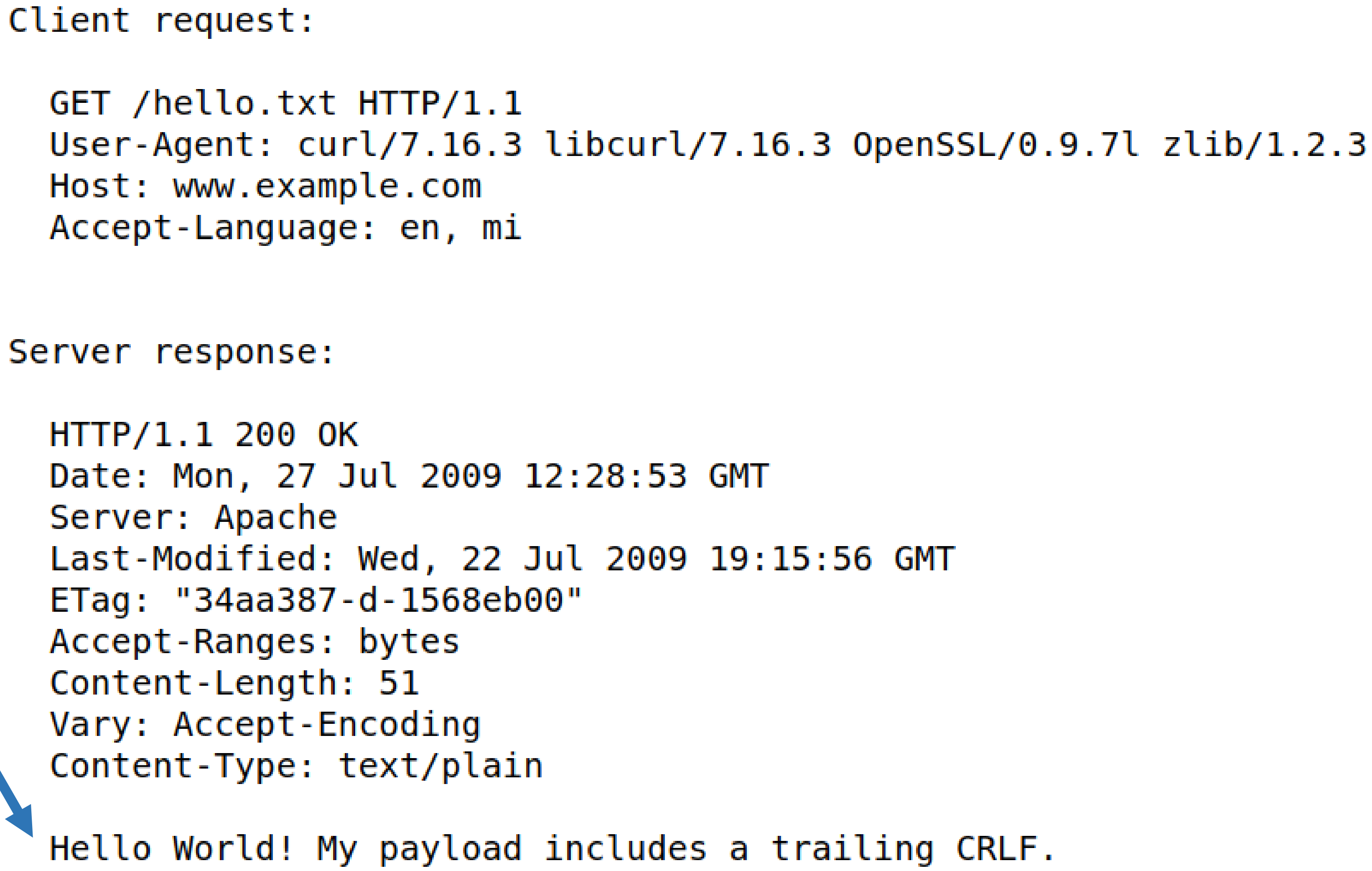
HTTP Session Example

response code
response text
header,
the requested data



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

HTTP Session Example



```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

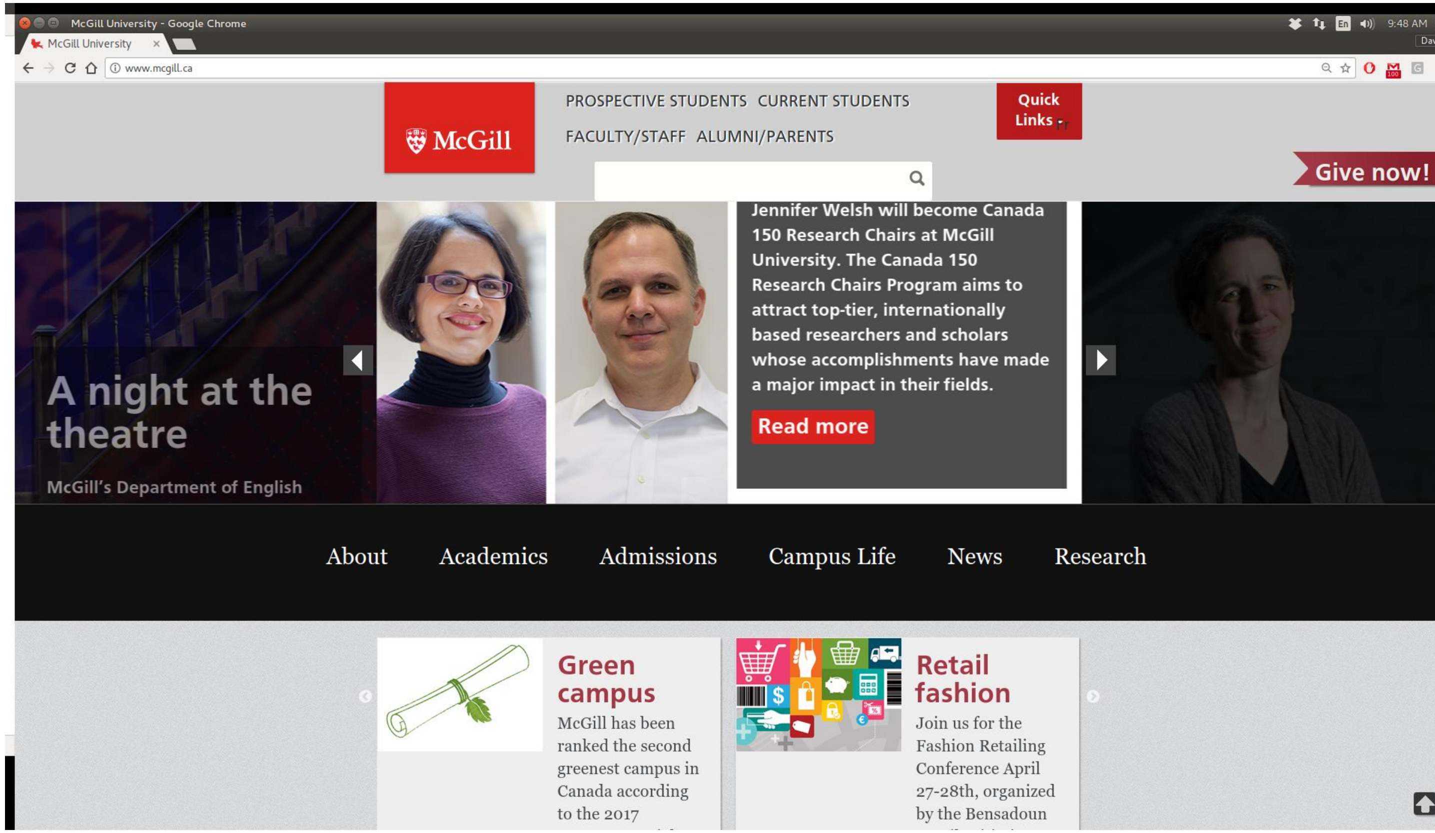
response code
response text
header,
the requested data

Hyper Text Markup Language (HTML)

- An example of formatted text (we know it well from A2 Q2)
- An HTML document is made up of tags with content. Examples:
  - <html></html> encloses the full document
  - <body></body> encloses the main content of the page
  - <a></a> encloses information about a link
- Each tag can also have arguments. Example:
  - <a href="http://cs.mcgill.ca>
- HTML is a convention that allows many web authors and many web browsers to see the web in a consistent way.
- It is not actually connected much with HTTP and the code that runs the web, so it's not a focus for 206 (although we will mess with it a bit when we write CGI functions)

Your Web Browser as a Software System

- You see rendered HTML including images, UI elements, color. How did this get there?



A night at the theatre

McGill's Department of English

Jennifer Welsh will become Canada 150 Research Chairs at McGill University. The Canada 150 Research Chairs Program aims to attract top-tier, internationally based researchers and scholars whose accomplishments have made a major impact in their fields.

Green campus

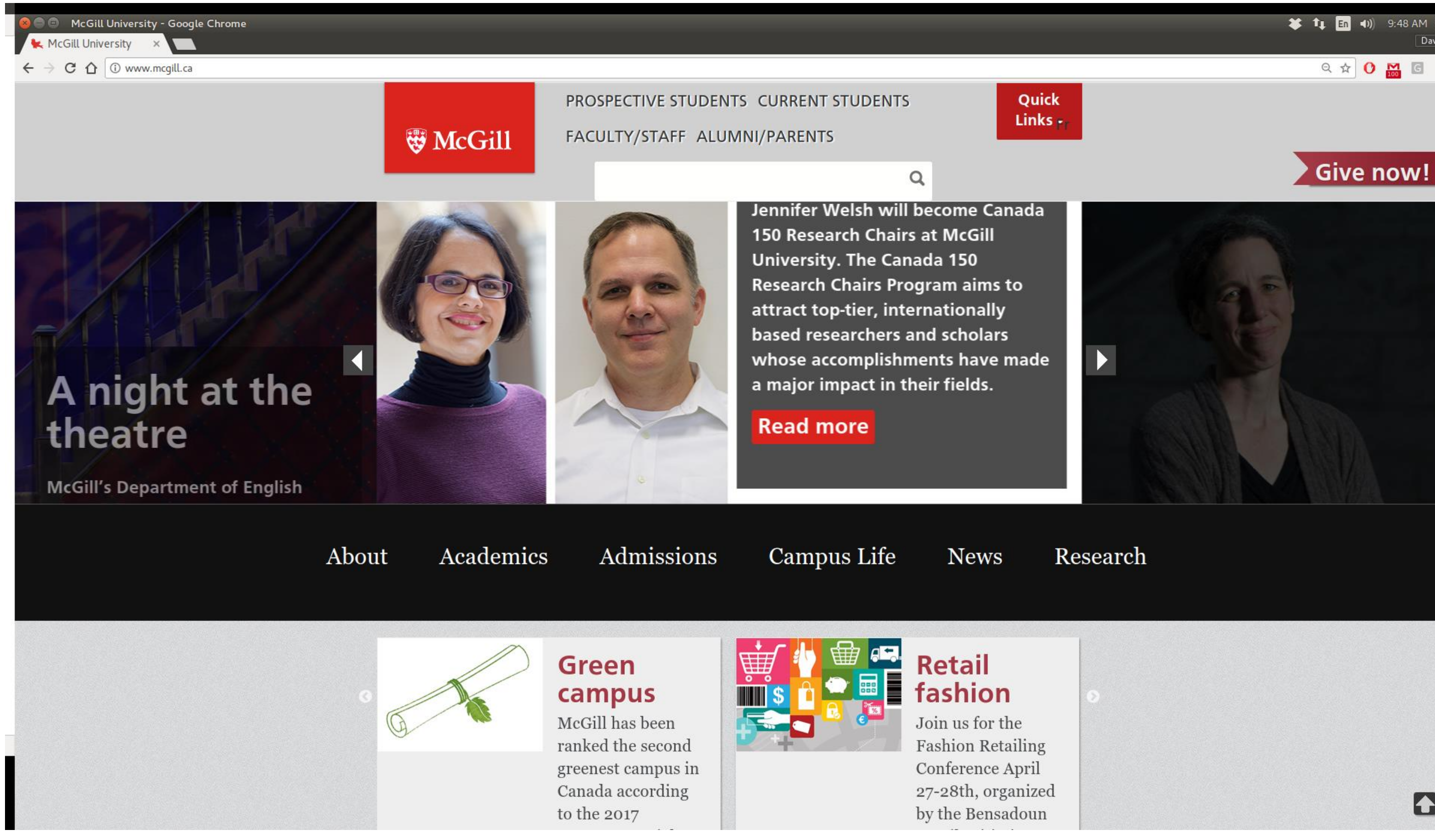
McGill has been ranked the second greenest campus in Canada according to the 2017

Retail fashion

Join us for the Fashion Retailing Conference April 27-28th, organized by the Bensadoun

Your Web Browser as a Software System

- You see rendered HTML including images, UI elements, color. How did this get there?
- Your browser is making HTTP Requests and reading the Responses:
  - Started each time you enter a URL or click a link
  - Parses the response (HTML page, image, etc) and displays what's appropriate



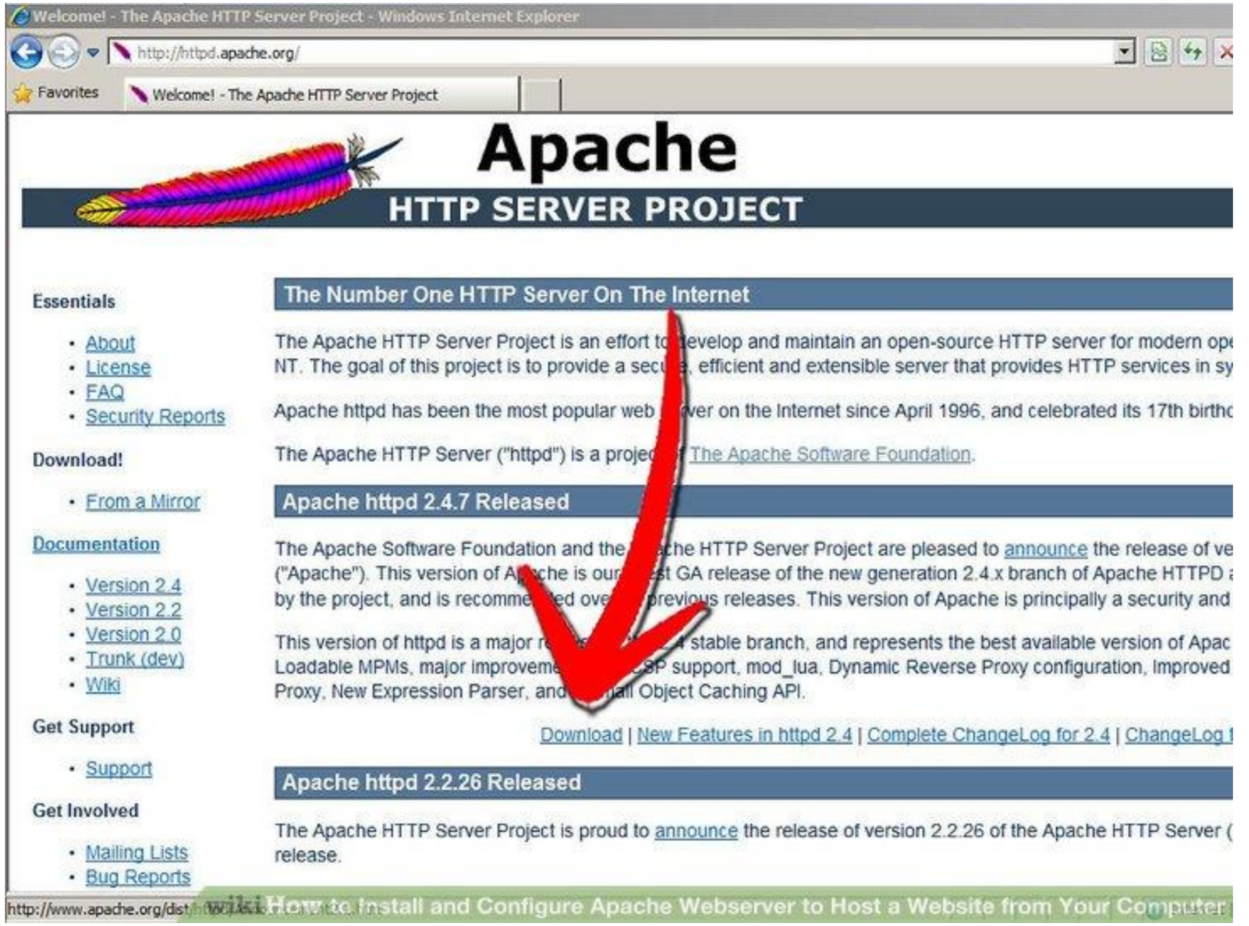
Jennifer Welsh will become Canada 150 Research Chairs at McGill University. The Canada 150 Research Chairs Program aims to attract top-tier, internationally based researchers and scholars whose accomplishments have made a major impact in their fields.

McGill has been ranked the second greenest campus in Canada according to the 2017

Join us for the Fashion Retailing Conference April 27-28th, organized by the Bensadoun

The other side: web server software

- A website is accessible because some computer is running persistent server code waiting for browsers to connect
- The server holds the data (HTML pages, images, programs, etc)
- Upon a connection, searches for the most appropriate response and returns the payload



Welcome! - The Apache HTTP Server Project - Windows Internet Explorer

http://httpd.apache.org/

Favorites | Welcome! - The Apache HTTP Server Project

Apache
HTTP SERVER PROJECT

Essentials

- About
- License
- FAQ
- Security Reports

Download!

- From a Mirror

Documentation

- Version 2.4
- Version 2.2
- Version 2.0
- Trunk (dev)
- Wiki

Get Support

- Support

Get Involved

- Mailing Lists
- Bug Reports

The Number One HTTP Server On The Internet

The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern ope
NT. The goal of this project is to provide a sec, efficient and extensible server that provides HTTP services in sy

Apache httpd has been the most popular web ver on the Internet since April 1996, and celebrated its 17th birthd

The Apache HTTP Server ("httpd") is a projec The Apache Software Foundation.

Apache httpd 2.4.7 Released

The Apache Software Foundation and the he HTTP Server Project are pleased to announce the release of ve
("Apache"). This version of Apche is our st GA release of the new generation 2.4.x branch of Apache HTTPD a
by the project, and is recommeed ove previous releases. This version of Apache is principally a security and

This version of httpd is a major r stable branch, and represents the best available version of Apac
Loadable MPMs, major improveme SP support, mod_lua, Dynamic Reverse Proxy configuration, Improved
Proxy, New Expression Parser, and all Object Caching API.

Download | New Features in httpd 2.4 | Complete ChangeLog for 2.4 | ChangeLog

Apache httpd 2.2.26 Released

The Apache HTTP Server Project is proud to announce the release of version 2.2.26 of the Apache HTTP Server (
release.

http://www.apache.org/dist

wikiHow to Install and Configure Apache Webserver to Host a Website from Your Computer

Managing HTTP in C

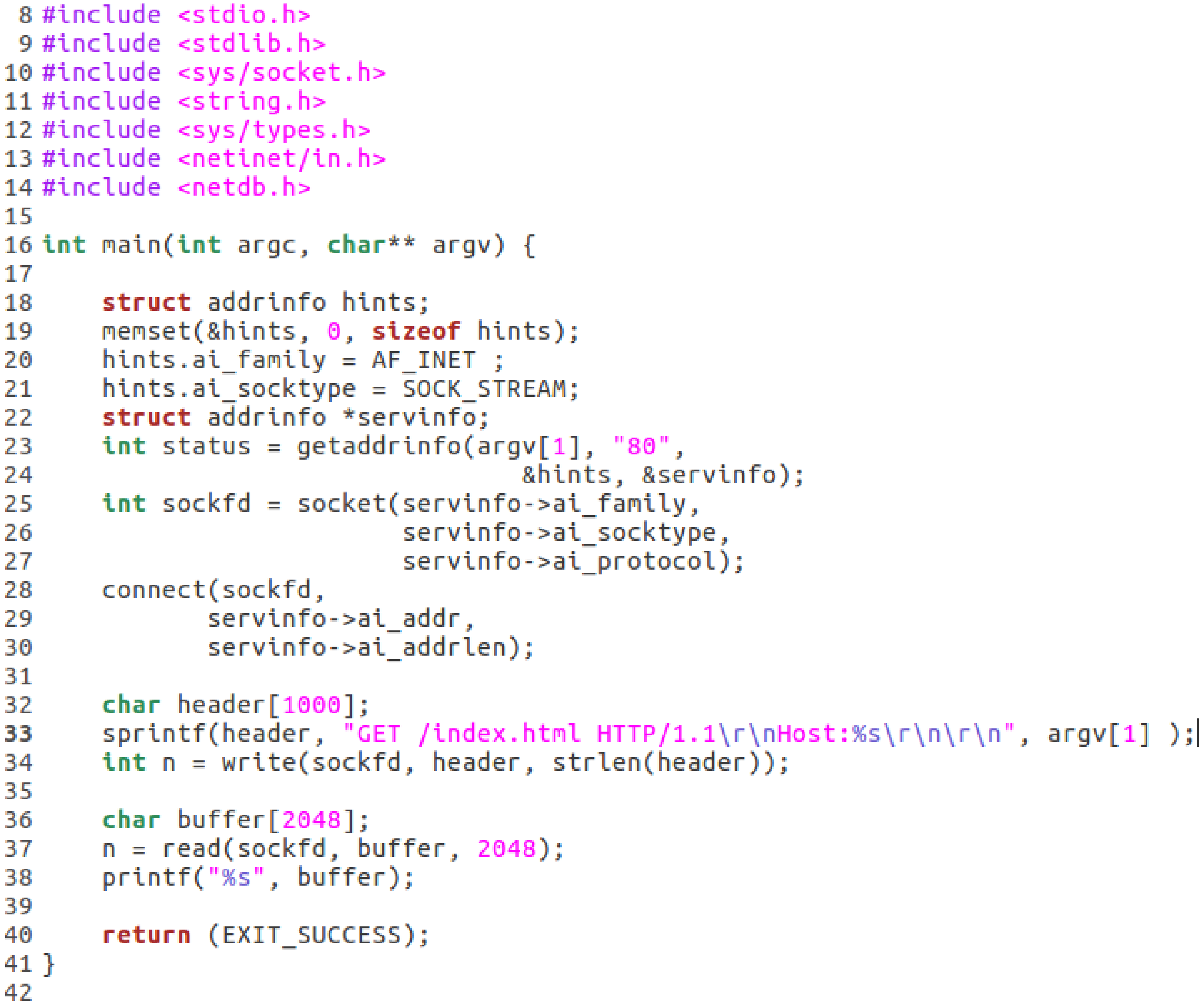
- Both the client (web browser) and server (web server) can be implemented in C quite easily with what we know now!
  - We will see a bit of this in Assignment 4

- Let's think for the next few slides about what the C code looks like and how we'd run it. Take a look in the ExampleCode folder on our Github to follow along

A Web Browser in C: pseudo-code

- Required to initiate the socket communications: socket() and connect()
  - Provide the server's address and use port number 80
- Required to form the HTTP Request. This is just a simple C string:
  - char *request = "GET / HTTP1.1"; (if we want to get the index.html)
- Send with write() onto the socket
- Wait for the response and gather it using read() from the socket
- Display information received if the code was OK, else tell the user the error

Web Client Example



```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>

int main(int argc, char** argv) {

    struct addrinfo hints;
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_INET ;
    hints.ai_socktype = SOCK_STREAM;
    struct addrinfo *servinfo;
    int status = getaddrinfo(argv[1], "80",
                                &hints, &servinfo);
    int sockfd = socket(servinfo->ai_family,
                        servinfo->ai_socktype,
                        servinfo->ai_protocol);
    connect(sockfd,
           servinfo->ai_addr,
           servinfo->ai_addrlen);

    char header[1000];
    sprintf(header, "GET /index.html HTTP/1.1\r\nHost:%s\r\n\r\n", argv[1] );
    int n = write(sockfd, header, strlen(header));

    char buffer[2048];
    n = read(sockfd, buffer, 2048);
    printf("%s", buffer);

    return (EXIT_SUCCESS);
}

```

Posted to Lecture19 folder
in ExampleCode

A Web Server in C: pseudo-code

- Loop forever awaiting connections, when one arrives:
  - Read the HTTP request, depending on the command and URI:
    - Perhaps look up a page on the local disk, read it into memory
    - Perhaps execute a CGI by forming the correct stdin, capture its stdout in memory (this to be covered in the next set of slides!)
    - Perhaps prepare an error response as a nicely formatted HTML page in memory
  - Write the HTTP response back to the client, forming the correct header and placing whatever we assembled in memory into the body

Web Server Example

- Ends up being too long to see on one slide reasonably. Check the file "httpd.c" in the Lecture19 folder in ExmapleCode

Exercises

- Run the browser and server yourself in a couple of places. You might get some problems with ports not being open, so be creative and use some trouble shooting (linux commands etc)

- Try to modify the provided browser and server example to change the HTTP requests in some way. Run and see what errors you get.

- Write an HTML page if you never have before. You can view it using your own browser using file://<path> or by placing it in the htdocs folder and running our sample server

- Writing a web client yourself from scratch. This will ensure you "get it" with sockets and HTTP request/responses