Web Services

Introduction to Computer Systems
24th Lecture, Dec. 24, 2018

Instructors:

Xiangqun Chen , Junlin Lu

Guangyu Sun , Xuetao Guan

Outline

- **Web history**
- Web and HTTP overview
- Tiny web server
- More examples

Web History

- **1989:**
  - Tim Berners-Lee (CERN) writes internal proposal to develop a distributed hypertext system
    - Connects “a web of notes with links”
    - Intended to help CERN physicists in large projects share and manage information
- **1990:**
  - Tim BL writes a graphical browser for Next machines

Web History (cont)

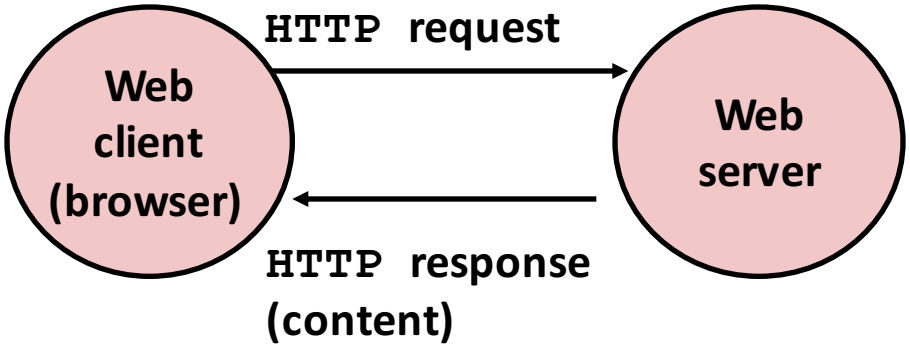
- **1992**
  - NCSA server released
  - 26 WWW servers worldwide
- **1993**
  - Marc Andreessen releases first version of NCSA Mosaic browser
  - Mosaic version released for (Windows, Mac, Unix)
  - Web (port 80) traffic at 1% of NSFNET backbone traffic
  - Over 200 WWW servers worldwide
- **1994**
  - Andreessen and colleagues leave NCSA to form “Mosaic Communications Corp” (predecessor to Netscape)

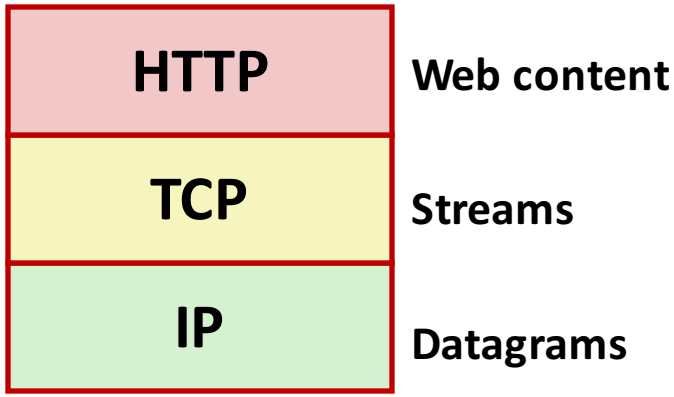
Outline

- Web history
- **Web and HTTP overview**
- Tiny web server
- More examples

Web Server Basics

- **Clients and servers communicate using the HyperText Transfer Protocol (HTTP)**
  - Client and server establish TCP connection
  - Client requests content
  - Server responds with requested content
  - Client and server close connection (eventually)
- **Current version is HTTP/1.1**
  - RFC 2616, June, 1999.





`http://www.w3.org/Protocols/rfc2616/rfc2616.html`

Web Content

- **Web servers return *content* to clients**
  - *content:* a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

- **Example MIME types**

`text/html`	HTML document
`text/plain`	Unformatted text
`image/gif`	Binary image encoded in GIF format
`image/png`	Binar image encoded in PNG format
`image/jpeg`	Binary image encoded in JPEG format

You can find the complete list of MIME types at:
`http://www.iana.org/assignments/media-types/media-types.xhtml`

Static and Dynamic Content

- **The content returned in HTTP responses can be either *static* or *dynamic***
  - *Static content*: content stored in files and retrieved in response to an HTTP request
    - Examples: HTML files, images, audio clips, Javascript programs
    - Request identifies which content file
  - *Dynamic content*: content produced on-the-fly in response to an HTTP request
    - Example: content produced by a program executed by the server on behalf of the client
    - Request identifies file containing executable code
- **Bottom line: *Web content is associated with a file that is managed by the server***

URLs

- **Web page consists of objects**
  - Object can be HTML file, JPEG image, Java applet, audio file,…
  - Web page consists of base HTML-file which includes several **referenced objects**
  - Each object is addressable by a URL (Universal Resource Locator)
- **URLs for static content:**
  - `http://www.cs.cmu.edu:80/index.html`
  - `http://www.cs.cmu.edu/index.html`
    - Identifies a file called `index.html`, managed by a Web server at `www.cs.cmu.edu` that is listening on port 80
- **URLs for dynamic content:**
  - `http://www.cs.cmu.edu:8000/cgi-bin/proc?15000&213`
    - **Identifies an executable file called `proc`**, managed by a Web server at `www.cs.cmu.edu` that is listening on port 8000, that should be called with two argument strings: `15000` and `213`

URLs and how clients and servers use them

- **Unique name for a file: URL (Universal Resource Locator)**
- **Example URL: `http://www.cmu.edu:80/index.html`**
- **Clients use *prefix* (`http://www.cmu.edu:80`) to infer:**
  - What kind (protocol) of server to contact (HTTP)
  - Where the server is (`www.cmu.edu`)
  - What port it is listening on (80)
- **Servers use *suffix* (`/index.html`) to:**
  - Determine if request is for static or dynamic content.
    - No hard and fast rules for this
    - One convention: executables reside in `cgi-bin` directory
  - Find file on file system
    - Initial "`/`" in suffix denotes home directory for requested content.
    - Minimal suffix is "`/`", which server expands to configured default filename (usually, `index.html`)

HTML (Hypertext Markup Language)

```
<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>

<h1>http://info.cern.ch - home of the first website</h1>
<p>From here you can:</p>
<ul>
<li><a href="http://info.cern.ch/hypertext/WWW/TheProject.html">Browse the
first website</a></li>
<li><a href="http://line-mode.cern.ch/www/hypertext/WWW/TheProject.html">Browse
the first website using the line-mode browser simulator</a></li>
<li><a href="http://home.web.cern.ch/topics/birth-web">Learn about the birth of
the web</a></li>
<li><a href="http://home.web.cern.ch/about">Learn about CERN, the physics
laboratory where the web was born</a></li>
</ul>
</body></html>
```

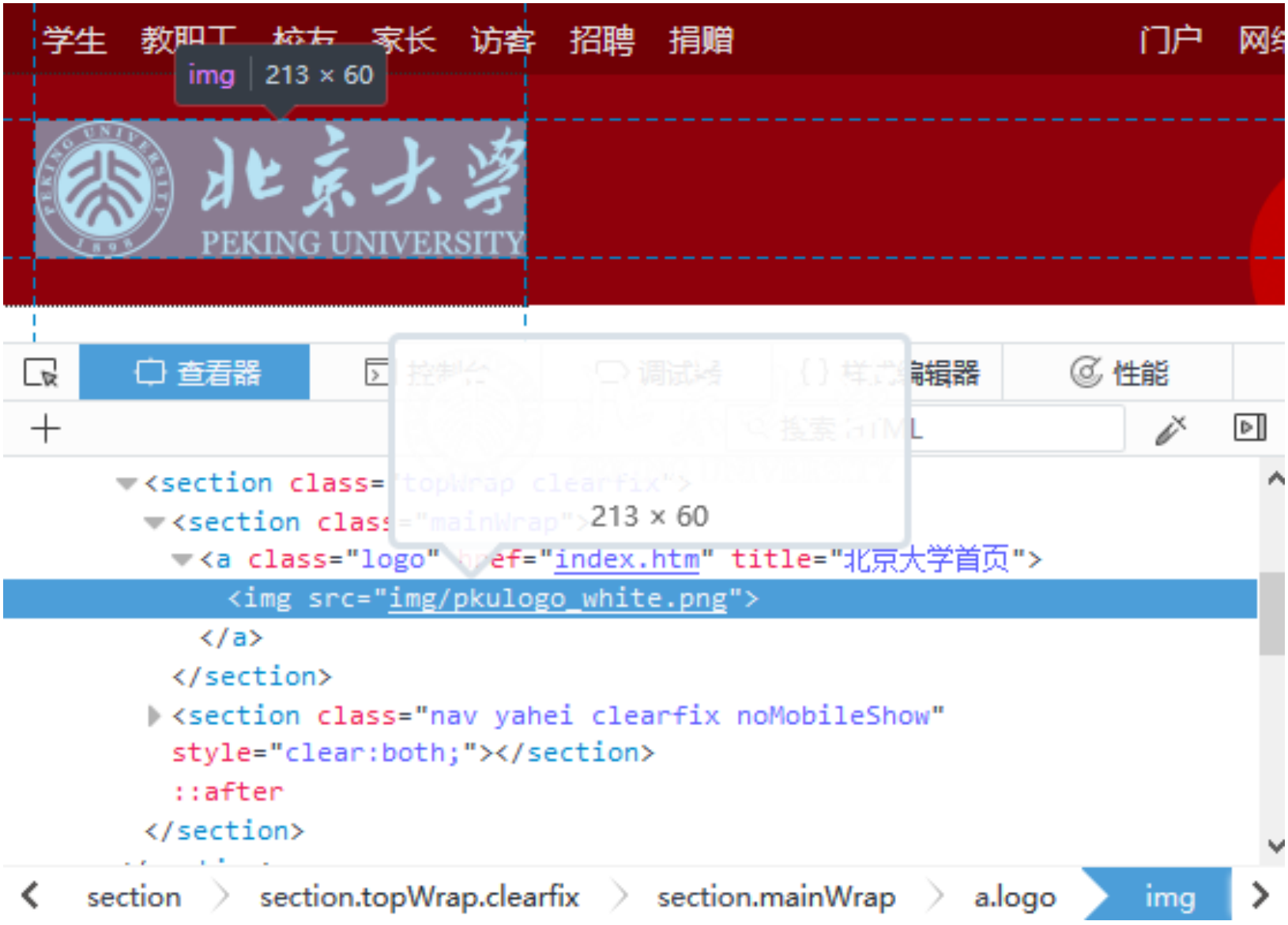
```
<head>……</head>
<title>……</title>
<body>……</body>
<p>……</p>
<a href="……">……</a>
<ul>……</ul>
<li>……</li>
```

http://info.cern.ch - home of the first website

From here you can:

- Browse the first website
- Browse the first website using the line-mode browser simulator
- Learn about the birth of the web
- Learn about CERN, the physics laboratory where the web was born

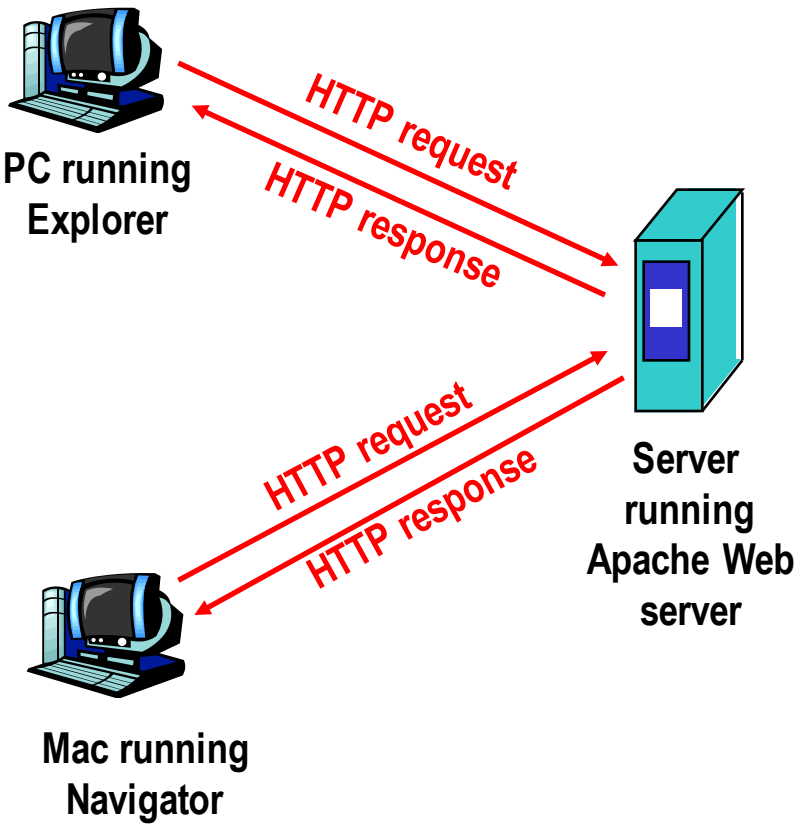
HTML source (cont.)



HTTP overview

HTTP: hypertext transfer protocol

- **Web's application layer protocol**
- **client/server model**
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests



HTTP Requests

- **HTTP request is a *request line*, followed by zero or more *request headers***

- **Request line: `<method> <uri> <version>`**
  - `<method>` is one of `GET`, `POST`, `OPTIONS`, `HEAD`, `PUT`, `DELETE`, or `TRACE`
  - `<uri>` is typically URL for proxies, URL suffix for servers
    - A URL is a type of URI (Uniform Resource Identifier)
    - See http://www.ietf.org/rfc/rfc2396.txt
  - `<version>` is HTTP version of request (`HTTP/1.0` or `HTTP/1.1`)

- **Request headers: `<header name>: <header data>`**
  - Provide additional information to the server

HTTP Responses

- **HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by *content*, with blank line ("\r\n") separating headers from content.**

- **Response line:**

  <version> <status code> <status msg>

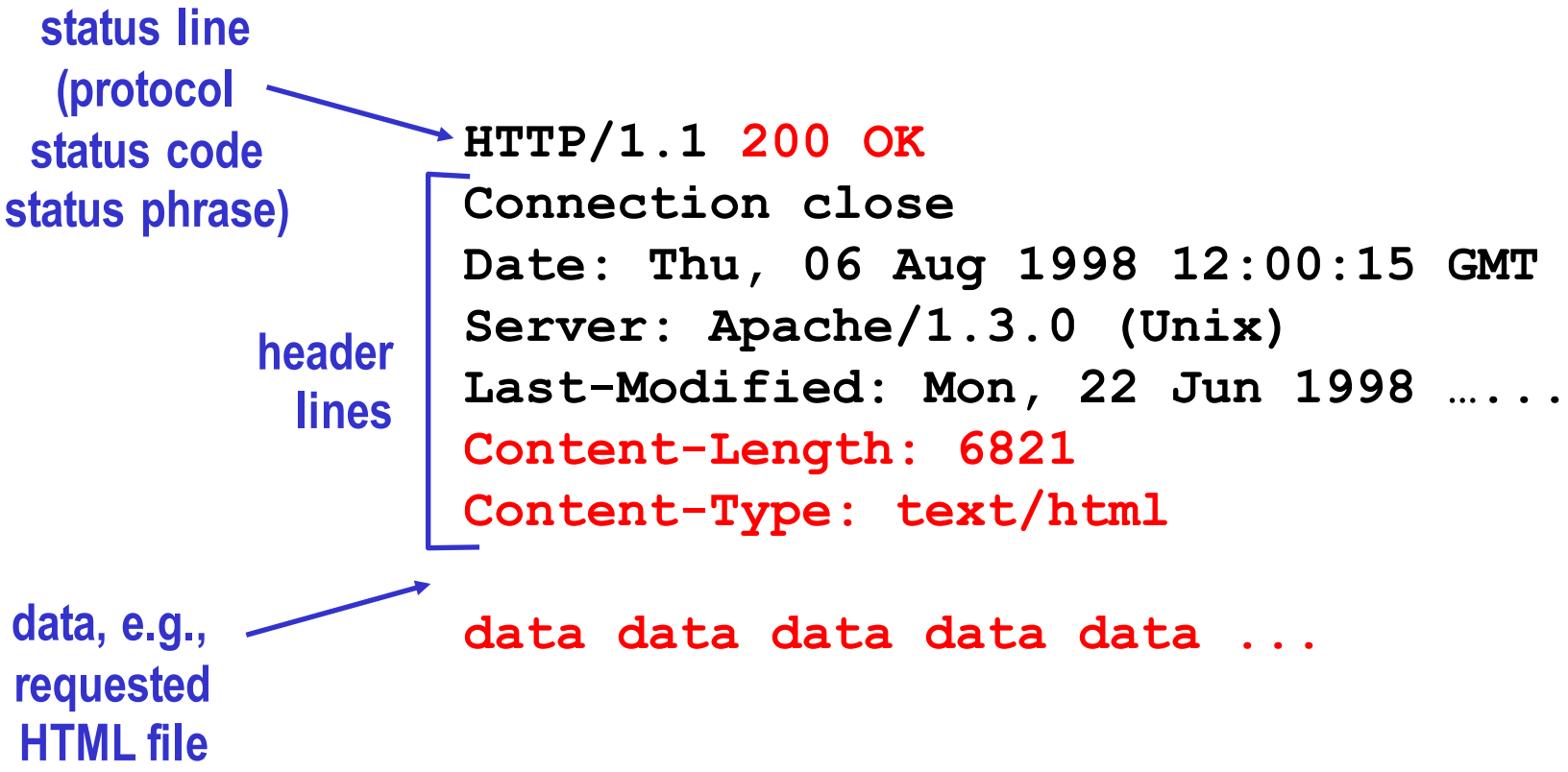
  - <version> is HTTP version of the response
  - <status code> is numeric status
  - <status msg> is corresponding English text
    - 200 OK Request was handled without error
    - 301 Moved Provide alternate URL
    - 404 Not found Server couldn't find the file

- **Response headers: <header name>: <header data>**
  - Provide additional information about response
  - `Content-Type:` MIME type of content in response body
  - `Content-Length:` Length of content in response body

HTTP Versions

- **Major differences between HTTP/1.1 and HTTP/1.0**
  - HTTP/1.0 uses a new connection for each transaction
  - HTTP/1.1 also supports *persistent connections*
    - multiple transactions over the same connection
    - `Connection: Keep-Alive`
  - HTTP/1.1 requires `HOST` header
    - `Host: www.cmu.edu`
    - Makes it possible to host multiple websites at single Internet host
  - HTTP/1.1 supports *chunked encoding*
    - `Transfer-Encoding: chunked`
  - HTTP/1.1 adds additional support for caching

HTTP response message



Example HTTP Transaction

```
whaleshark> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.42.52...                      Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET / HTTP/1.1                             Client: request line
Host: www.cmu.edu                          Client: required HTTP/1.1 header
                                           Client: empty line terminates headers
HTTP/1.1 301 Moved Permanently             Server: response line
Date: Wed, 05 Nov 2014 17:05:11 GMT        Server: followed by 5 response headers
Server: Apache/1.3.42 (Unix)               Server: this is an Apache server
Location: http://www.cmu.edu/index.shtml   Server: page has moved here
Transfer-Encoding: chunked                 Server: response body will be chunked
Content-Type: text/html; charset=...       Server: expect HTML in response body
                                           Server: empty line terminates headers
15c                                        Server: first line in response body
<HTML><HEAD>                               Server: start of HTML content
...
</BODY></HTML>                             Server: end of HTML content
0                                          Server: last line in response body
Connection closed by foreign host.         Server: closes connection
```

- HTTP standard requires that each text line end with "`\r\n`"
- Blank line ("`\r\n`") terminates request and response headers

Example HTTP Transaction, Take 2

```
whaleshark> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.42.52...                      Telnet prints 3 lines to terminal
Connected to WWW-CMU-PROD-VIP.ANDREW.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1                  Client: request line
Host: www.cmu.edu                          Client: required HTTP/1.1 header
                                           Client: empty line terminates headers
HTTP/1.1 200 OK                            Server: response line
Date: Wed, 05 Nov 2014 17:37:26 GMT        Server: followed by 4 response headers
Server: Apache/1.3.42 (Unix)
Transfer-Encoding: chunked
Content-Type: text/html; charset=...
                                           Server: empty line terminates headers
1000                                       Server: begin response body
<html ..>                                  Server: first line of HTML content
…
</html>
0                                          Server: end response body
Connection closed by foreign host.         Server: close connection
```

Data Transfer Mechanisms

- **`Transfer-Encoding: standard`**
  - Specify total length with content-length
  - Requires that program buffer entire message
- **`Transfer-Encoding: chunked`**
  - Break into blocks
  - Prefix each block with number of bytes (Hex coded)

Chunked Encoding Example

```
HTTP/1.1 200 OK\n
Date: Sun, 31 Oct 2010 20:47:48 GMT\n
Server: Apache/1.3.41 (Unix)\n
Keep-Alive: timeout=15, max=100\n
Connection: Keep-Alive\n
Transfer-Encoding: chunked\n
Content-Type: text/html\n
\r\n
d75\r\n
<html>
<head>
.<link href="http://www.cs.cmu.edu/style/calendar.css" rel="stylesheet"
type="text/css">
</head>
<body id="calendar_body">

<div id='calendar'><table width='100%'  border='0' cellpadding='0'
cellspacing='1' id='cal'>

 . . .
</body>
</html>
\r\n
0\r\n
\r\n
```

First Chunk: 0xd75 = 3445 bytes

Second Chunk: 0 bytes (indicates last chunk)

Outline

- Web history
- Web and HTTP overview
- **Tiny web server**
- More examples

Tiny Web Server

- **Tiny Web server described in text**
  - Tiny is a sequential Web server
  - Serves static and dynamic content to real browsers
    - text files, HTML files, GIF, PNG, and JPEG images
  - 239 lines of commented C code
  - Not as complete or robust as a real Web server
    - You can break it with poorly-formed HTTP requests (e.g., terminate lines with “\n” instead of “\r\n”)

Tiny Operation

- **Accept connection from client**
- **Read request from client (via connected socket)**
- **Split into <method> <uri> <version>**
  - If method not GET, then return error
- **If URI contains "`cgi-bin`" then serve dynamic content**
  - (Would do wrong thing if had file "`abcgi-bingo.html`")
  - Fork process to execute program
- **Otherwise serve static content**
  - Copy file to output

Tiny Serving Static Content

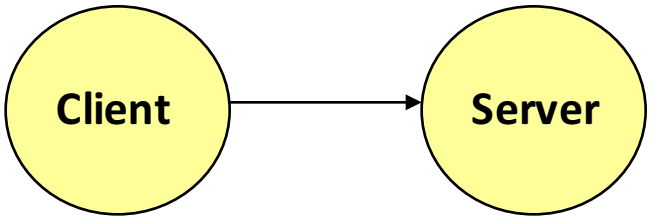
```
void serve_static(int fd, char *filename, int filesize)
{
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    /* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    Rio_writen(fd, buf, strlen(buf));

    /* Send response body to client */
    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}
```

tiny.c

Serving Dynamic Content

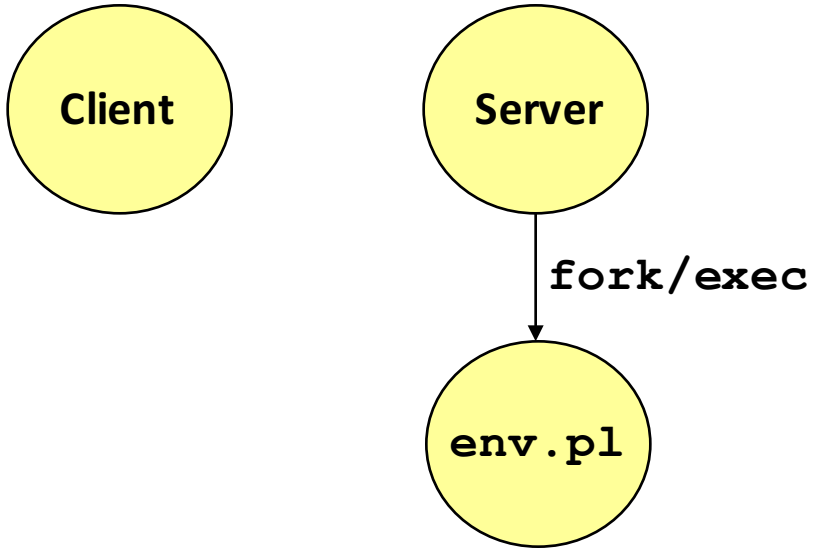
- Client sends request to server
- If request URI contains the string "`/cgi-bin`", the Tiny server assumes that the request is for dynamic content

```
GET /cgi-bin/env.pl HTTP/1.1
```



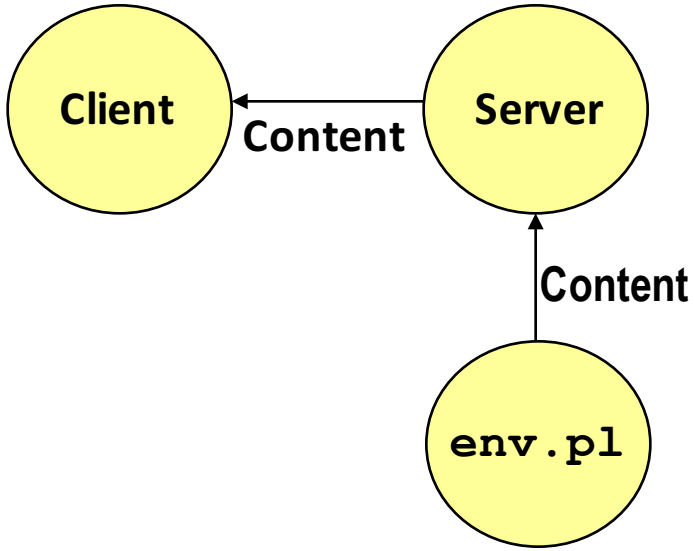
Serving Dynamic Content (cont)

- The server creates a child process and runs the program identified by the URI in that process



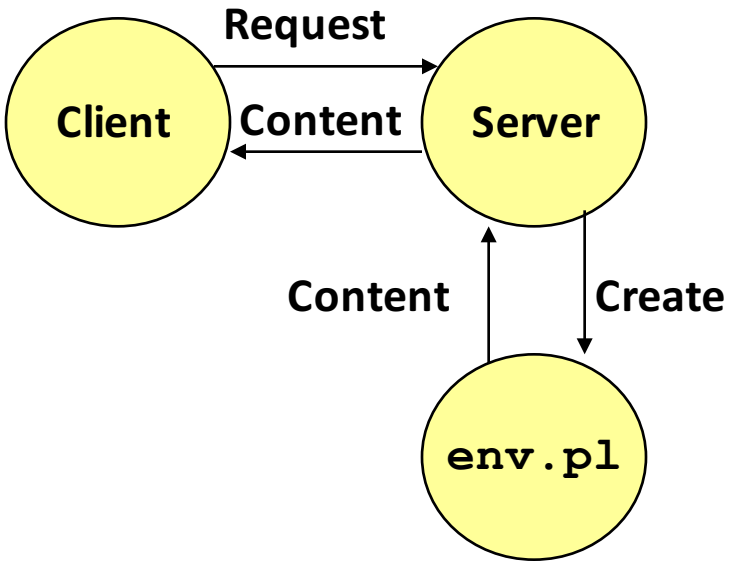
Serving Dynamic Content (cont)

- The child runs and generates the dynamic content
- The server captures the content of the child and forwards it without modification to the client



Issues in Serving Dynamic Content

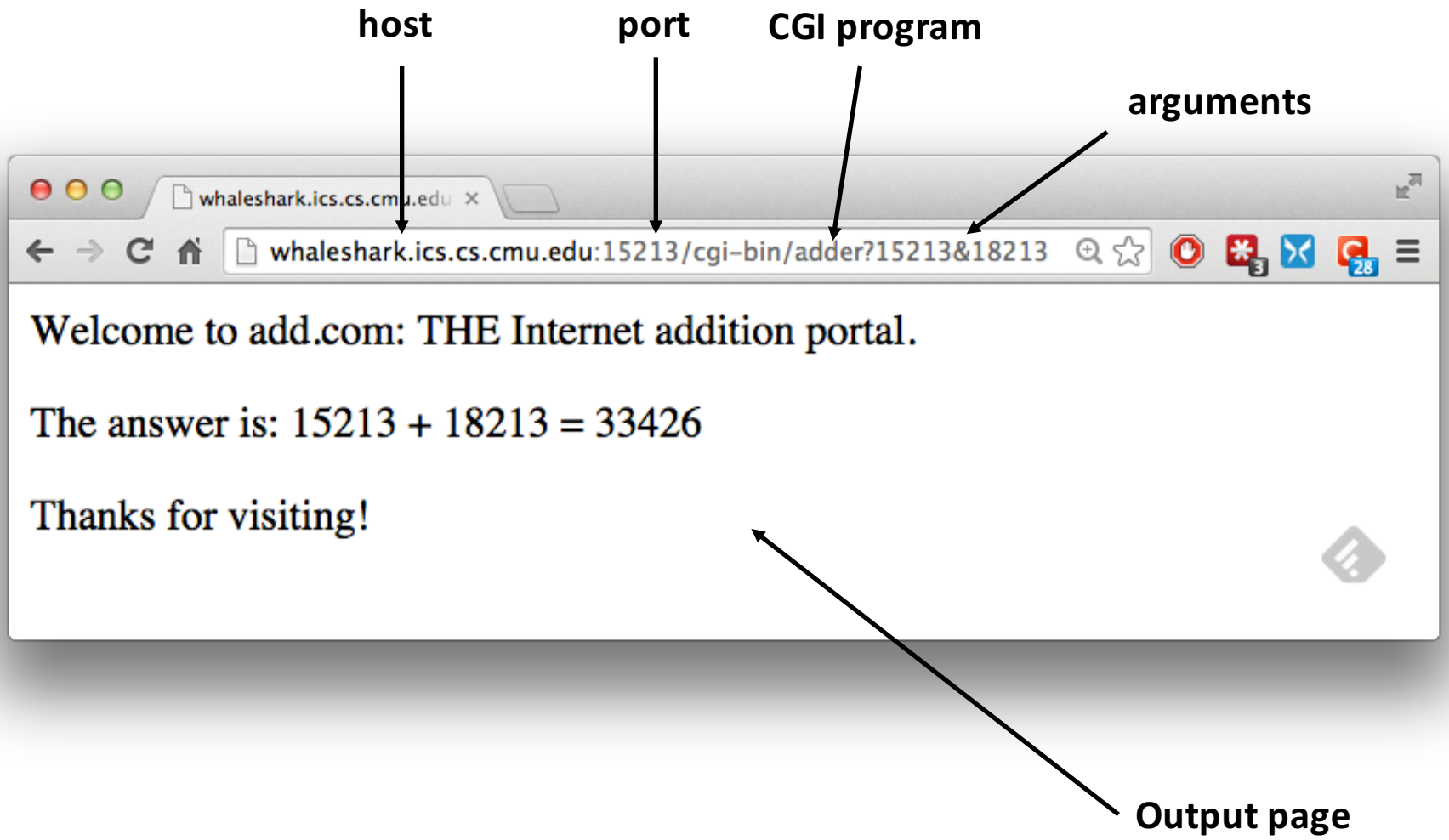
- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the Common Gateway Interface (CGI) specification.



CGI

- **Because the children are written according to the CGI spec, they are often called *CGI programs.***
- **However, CGI really defines a simple standard for transferring information between the client (browser), the server, and the child process.**
- **CGI is the original standard for generating dynamic content. Has been largely replaced by other, faster techniques:**
  - E.g., fastCGI, Apache modules, Java servlets, Rails controllers
  - Avoid having to create process on the fly (expensive and slow).

The add.com Experience



Serving Dynamic Content With GET

- **Question: How does the client pass arguments to the server?**
- **Answer: The arguments are appended to the URI**

- **Can be encoded directly in a URL typed to a browser or a URL in an HTML link**
  - **`http://add.com/cgi-bin/adder?15213&18213`**
  - **`adder`** is the CGI program on the server that will do the addition.
  - argument list starts with "**`?`**"
  - arguments separated by "**`&`**"
  - spaces represented by "**`+`**" or "**`%20`**"

Serving Dynamic Content With GET

- **URL suffix:**
  - `cgi-bin/adder?15213&18213`

- **Result displayed on browser:**

```
Welcome to add.com: THE Internet addition portal.

The answer is: 15213 + 18213 = 33426

Thanks for visiting!
```

Serving Dynamic Content With GET

- **Question: How does the server pass these arguments to the child?**
- **Answer: In environment variable QUERY_STRING**
  - A single string containing everything after the "?"
  - For add: `QUERY_STRING` = "`15213&18213`"

```
/* Extract the two arguments */
if ((buf = getenv("QUERY_STRING")) != NULL) {
    p = strchr(buf, '&');
    *p = '\0';
    strcpy(arg1, buf);
    strcpy(arg2, p+1);
    n1 = atoi(arg1);
    n2 = atoi(arg2);
}
```

adder.c

Additional CGI Environment Variables

- **General**
  - `SERVER_SOFTWARE`
  - `SERVER_NAME`
  - `GATEWAY_INTERFACE` (CGI version)
- **Request-specific**
  - `SERVER_PORT`
  - `REQUEST_METHOD` (`GET`, `POST`, etc)
  - **`QUERY_STRING` (contains `GET` args)**
  - `REMOTE_HOST` (domain name of client)
  - `REMOTE_ADDR` (IP address of client)
  - `CONTENT_TYPE` (for `POST`, type of data in message body, e.g., `text/html`)
  - `CONTENT_LENGTH` (length in bytes)
- **In addition, the value of each header of type *type* received from the client is placed in environment variable `HTTP_`*type***
  - Examples (any "-" is changed to "_") :
    - `HTTP_ACCEPT`
    - `HTTP_HOST`
    - `HTTP_USER_AGENT`

Serving Dynamic Content with GET

- Question: How does the server capture the content produced by the child?
- Answer: The child generates its output on `stdout`. Server uses `dup2` to redirect `stdout` to its connected socket.

```
void serve_dynamic(int fd, char *filename, char *cgiargs)
{
    char buf[MAXLINE], *emptylist[] = { NULL };

    /* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    if (Fork() == 0) { /* Child */
        /* Real server would set all CGI vars here */
        setenv("QUERY_STRING", cgiargs, 1);
        Dup2(fd, STDOUT_FILENO);          /* Redirect stdout to client */
        Execve(filename, emptylist, environ); /* Run CGI program */
    }
    Wait(NULL); /* Parent waits for and reaps child */
}
```

tiny.c

Serving Dynamic Content with GET

- Notice that only the CGI child process knows the content type and length, so it must generate those headers.

```
/* Make the response body */
sprintf(content, "Welcome to add.com: ");
sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
        content, n1, n2, n1 + n2);
sprintf(content, "%sThanks for visiting!\r\n", content);

/* Generate the HTTP response */
printf("Content-length: %d\r\n", (int)strlen(content));
printf("Content-type: text/html\r\n\r\n");
printf("%s", content);
fflush(stdout);

exit(0);
```

adder.c

Serving Dynamic Content With GET

```
bash:makoshark> telnet whaleshark.ics.cs.cmu.edu 15213
Trying 128.2.210.175...
Connected to whaleshark.ics.cs.cmu.edu (128.2.210.175).
Escape character is '^]'.
GET /cgi-bin/adder?15213&18213 HTTP/1.0

HTTP/1.0 200 OK
Server: Tiny Web Server
Connection: close
Content-length: 117
Content-type: text/html

Welcome to add.com: THE Internet addition portal.
<p>The answer is: 15213 + 18213 = 33426
<p>Thanks for visiting!
Connection closed by foreign host.
bash:makoshark>
```

HTTP request sent by client

HTTP response generated by the server

HTTP response generated by the CGI program

For More Information

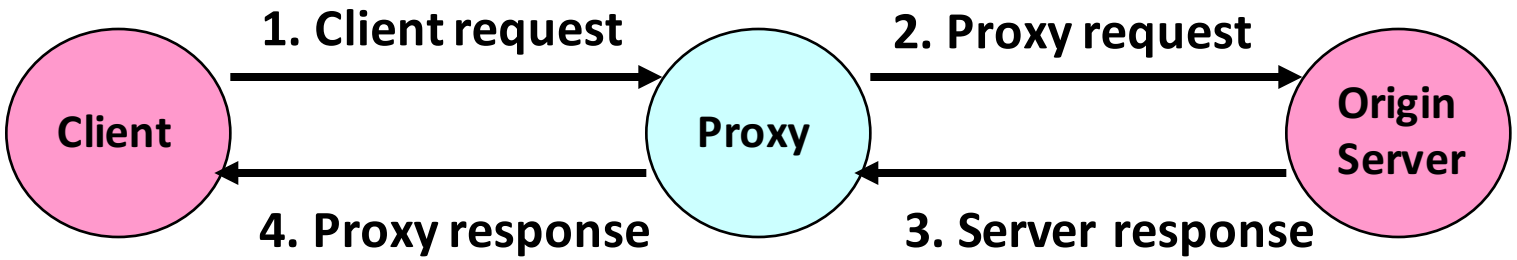
- **W. Richard Stevens et. al. "Unix Network Programming: The Sockets Networking API", Volume 1, Third Edition, Prentice Hall, 2003**
  - THE network programming bible.
- **Michael Kerrisk, "The Linux Programming Interface", No Starch Press, 2010**
  - THE Linux programming bible.
- **Complete versions of all code in this lecture is available from the 213 schedule page.**
  - `http://www.cs.cmu.edu/~213/schedule.html`
  - csapp.{.c,h}, hostinfo.c, echoclient.c, echoserveri.c, tiny.c, adder.c

Outline

- Web history
- Web and HTTP overview
- Tiny web server
- **More examples**

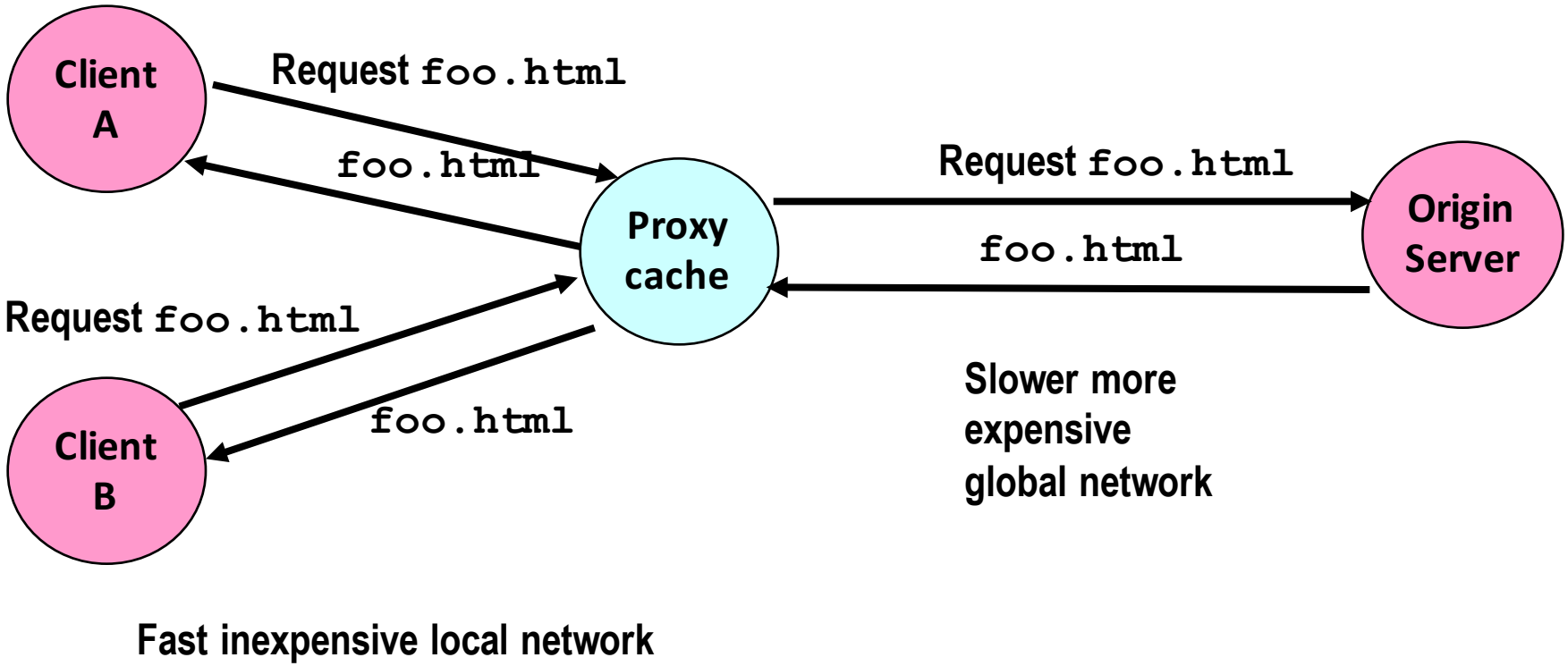
Proxies

- A *proxy* is an intermediary between a client and an *origin server*
  - To the client, the proxy acts like a server
  - To the server, the proxy acts like a client



Why Proxies?

- **Can perform useful functions as requests and responses pass by**
  - Examples: Caching, logging, anonymization, filtering, transcoding



Two types of web proxy

- **Explicit (browser-known) proxies**
  - Used by configuring browser to send requests to proxy
  - Each request specifies entire URL
    - allowing proxy to know target server

- **Transparent proxies**
  - Browser/client behaves as though there is no proxy
  - Proxy runs on network component in route between client and server
    - intercepting and interposing on web requests

Get Google PageRank score

- Send a HTTP GET request to a Google server (www.google.com) with a query command:

  /search?client=navclient-auto
- appended with parameters \&ch=61658376380\&features=Rank\&q=info:http://www.yahoo.com.
- The string “61658376380" is transformed from http://www.yahoo.com by a transformation function that accepts a URL as input.
- The returned results of the GET request contains the score

Get Google PageRank score

HTTP response status codes

In first line in server->client response message.
A few sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

- requested object moved, new location specified later in this message (Location:)

400 Bad Request

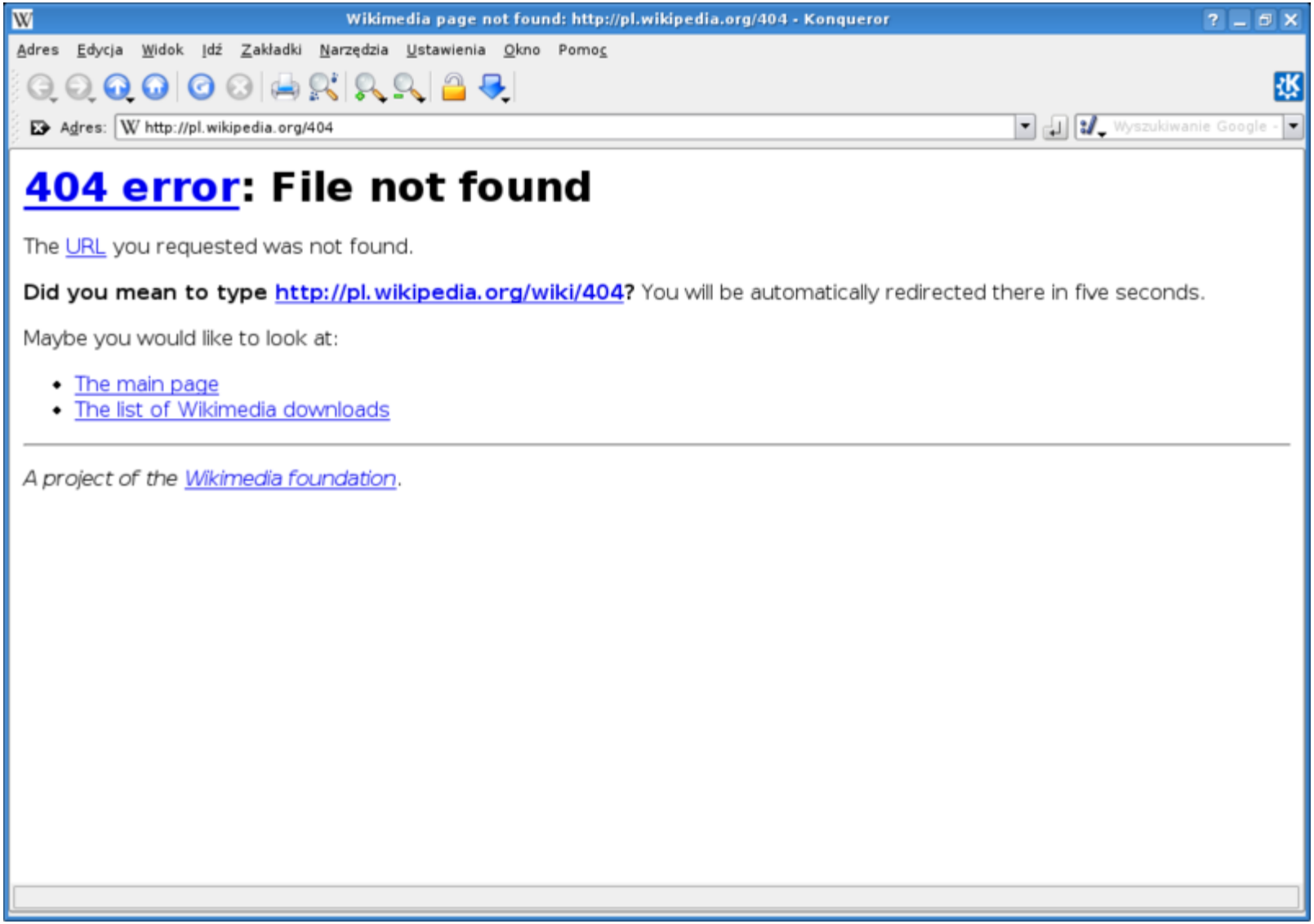
- request message not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

HTTP response status codes – 404 error

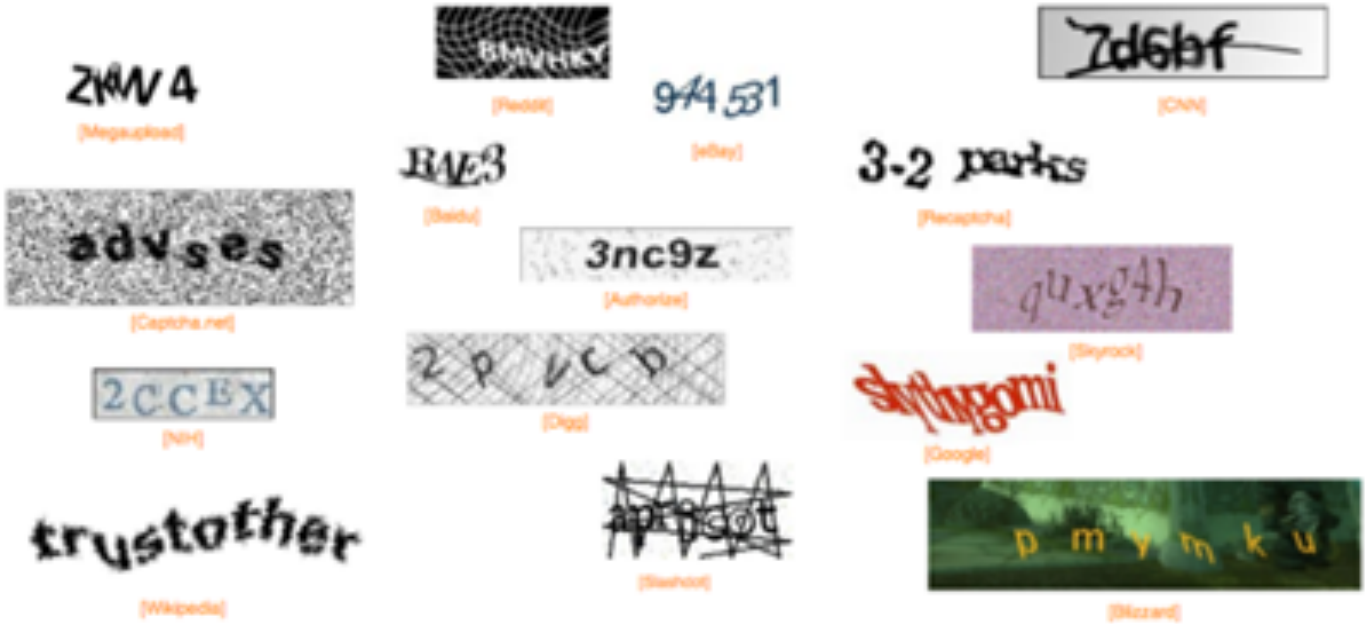


HTTP 404 公益

NotFound Project公益项目：**利用闲置网络资源**
发挥公益的力量让更多人帮忙寻找失踪儿童。

CAPTCHA

- **CAPTCHA： Completely Automated Public Turing Test to Tell Computers and Humans Apart**
  - 全自动区分计算机和人类的图灵测试



reCAPTCHA

- reCAPTCHA是利用CAPTCHA的原理，借助于人类大脑对难以识别的字符的辨别能力，进行对古旧书籍中难以被OCR识别的字符进行辨别的技术。