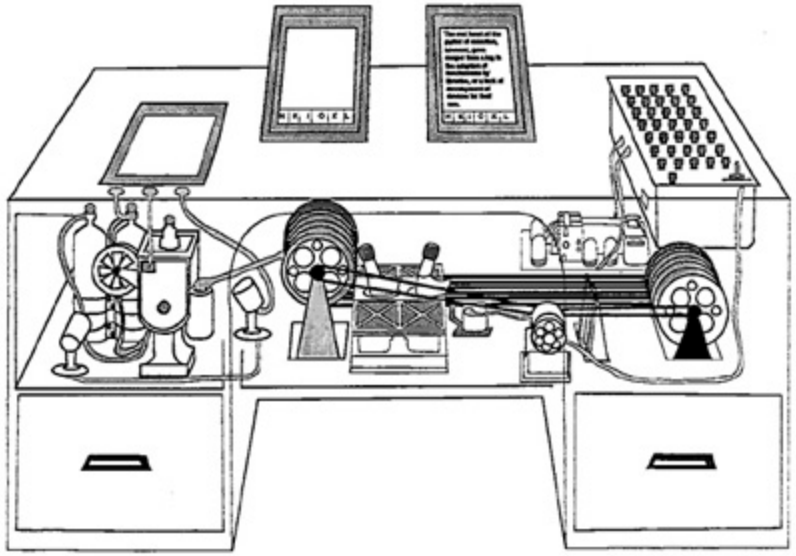
Web Services

15-213 / 18-213: Introduction to Computer Systems
22nd Lecture, April 8, 20134

Instructors:

Seth Copen Goldstein, Anthony Rowe, and Greg Kesden

Web History (seminal)



"Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and to coin one at random, "memex" will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory."

- **1945:**
  - Vannevar Bush, "As we may think", Atlantic Monthly, July, 1945
    - Describes the idea of a distributed hypertext system
    - A "memex" that mimics the "web of trails" in our minds

(Modern) 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)

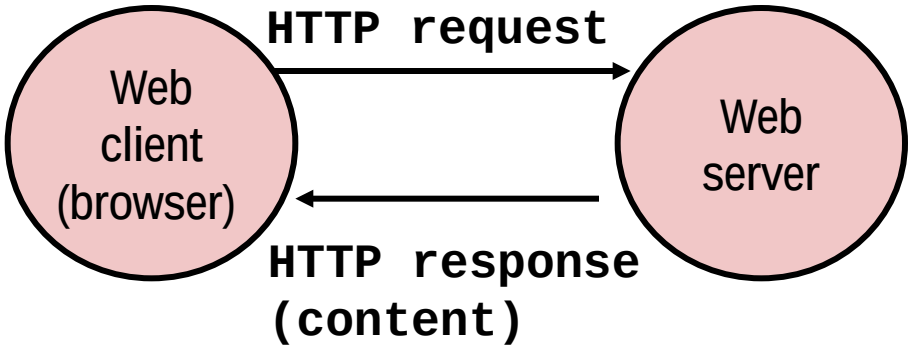
- **1992**
  - NCSA server released
  - 26 WWW servers worldwide
- **1993**
  - Marc Andreessen releases first version of NCSA Mosaic browser
  - Mosaic version released for (Windows, Mac, Unix)
  - Web (port 80) traffic at 1% of NSFNET backbone traffic
  - Over 200 WWW servers worldwide
- **1994**
  - Andreessen and colleagues leave NCSA to form “Mosaic Communications Corp” (predecessor to Netscape)
- **1996**
  - Cookies implemented in major browsers

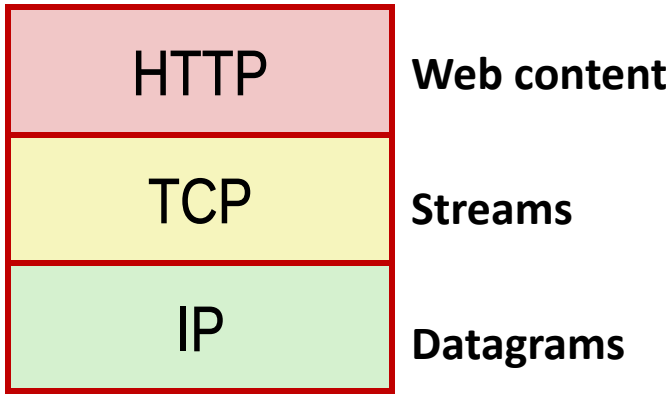
Web History (cont)

- **1999-2002**
  - Web 2.0 coined
  - Changes the web from a content delivery system to a framework for building interactive applications through the browser
- **2005**
  - AJAX coined
  - XMLHttpRequest specification to support truly asychronous web pages
- **2006**
  - jQuery released

Web Servers

- **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
`application/postscript`	Postcript document
`image/gif`	Binary image encoded in GIF format
`image/jpeg`	Binary image encoded in JPEG format
`application/json`	JSON object
`text/css`	CSS document

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
    - 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 which file containing executable code
- **Bottom line: *(some) Web content is associated with a file that is managed by the server***

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
    - Old 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`)

Example of an HTTP Transaction

```
unix> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.10.162...               Telnet prints 3 lines to the terminal
Connected to www.cmu.edu.
Escape character is '^]'.
GET / HTTP/1.1                       Client: request line
host: www.cmu.edu                    Client: required HTTP/1.1 HOST header
                                     Client: empty line terminates headers.
HTTP/1.1 301 Moved Permanently       Server: response line
Location: http://www.cmu.edu/index.shtml   Client should try again

Connection closed by foreign host.   Server: closes connection
unix>                                Client: closes connection and terminates
```

Example of an HTTP Transaction, Take 2

```
unix> telnet www.cmu.edu 80          Client: open connection to server
Trying 128.2.10.162...               Telnet prints 3 lines to the terminal
Connected to www.cmu.edu.
Escape character is '^]'.
GET /index.shtml HTTP/1.1            Client: request line
host: www.cmu.edu                    Client: required HTTP/1.1 HOST header
                                     Client: empty line terminates headers.
HTTP/1.1 200 OK                      Server: responds with web page
Date: Fri, 29 Oct 2010 19:41:08 GMT
Server: Apache/1.3.39 (Unix) mod_pubcookie/3.3.3 ...
Transfer-Encoding: chunked
Content-Type: text/html
...                                  Lots of stuff
Connection closed by foreign host.   Server: closes connection
unix>                                Client: closes connection and terminates
```

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`)

HTTP Requests (cont)

- **HTTP methods:**
  - GET: Retrieve static or dynamic content
    - Arguments for dynamic content are in URI
    - Workhorse method (99% of requests)
  - POST: Retrieve dynamic content
    - Arguments for dynamic content are in the request body
  - OPTIONS: Get server or file attributes
  - HEAD: Like GET but no data in response body
  - PUT: Write a file to the server!
  - DELETE: Delete a file on the server!
  - TRACE: Echo request in response body
    - Useful for debugging
- **Request headers: `<header name>: <header data>`**
  - Provide additional information to the server

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* (described later)
    - Transfer-Encoding: chunked
  - HTTP/1.1 adds additional support for caching

HTTP Responses

- **HTTP response is a *response line* followed by zero or more *response headers*, possibly followed by data**
- **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
    - 403 Forbidden Server lacks permission to access file
    - 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

GET Request to Apache Server From Firefox Browser

URI is just the suffix, not the entire URL

```
GET /~bryant/test.html HTTP/1.1
Host: www.cs.cmu.edu
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US;
rv:1.9.2.11) Gecko/20101012 Firefox/3.6.11
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
CRLF (\r\n)
```

GET Response From Apache Server

```
HTTP/1.1 200 OK
Date: Fri, 29 Oct 2010 19:48:32 GMT
Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.7m
mod_pubcookie/3.3.2b PHP/5.3.1
Accept-Ranges: bytes
Content-Length: 479
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
<html>
<head><title>Some Tests</title></head>

<body>
<h1>Some Tests</h1>
 . . .
</body>
</html>
```

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 and JPEG images
  - 226 lines of commented C code
  - Not as complete or robust as a real web server

Tiny Operation

- **Accept connection from client**
- **Read request from client (via connected socket)**
- **Split into method / uri / version**
  - If 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

From `tiny.c`

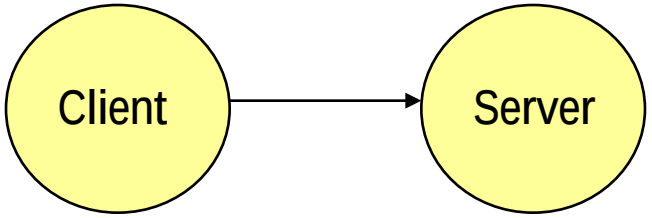
```
/* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n",
            buf,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

- Serve file specified by `filename`
- Use file metadata to compose header
- “Read” file via mmap
- Write to output

Serving Dynamic Content

- Client sends request to server

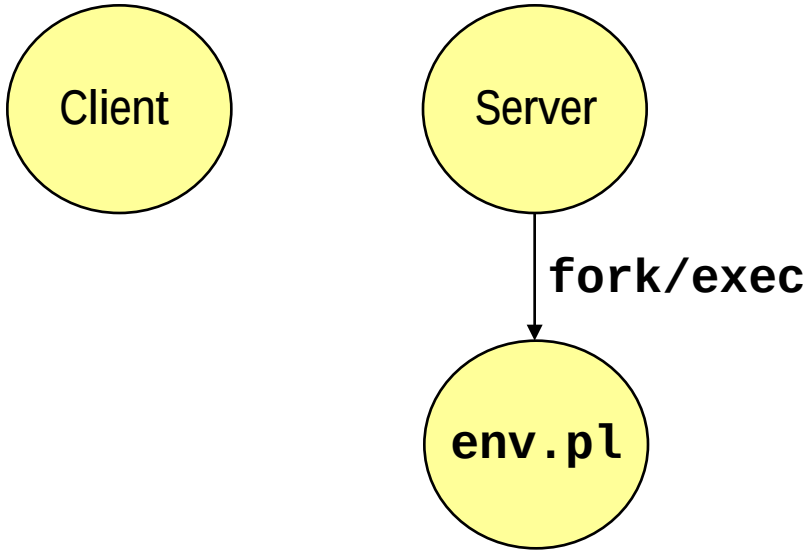
```
GET /cgi-bin/env.pl HTTP/1.1
```



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

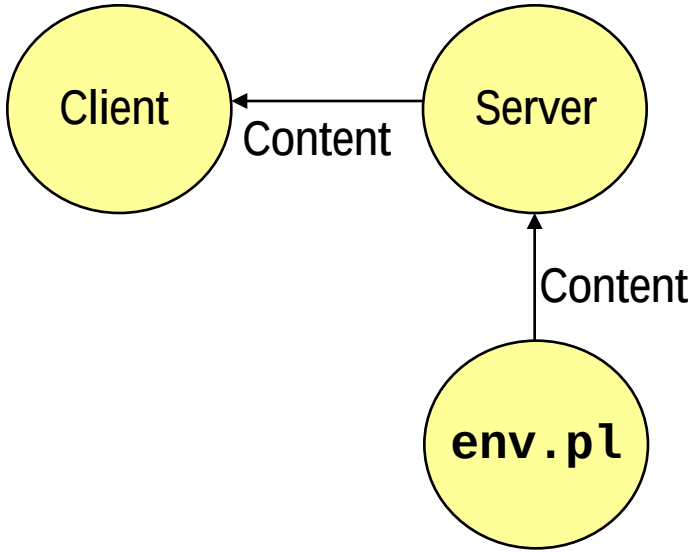
Serving Dynamic Content (cont)

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



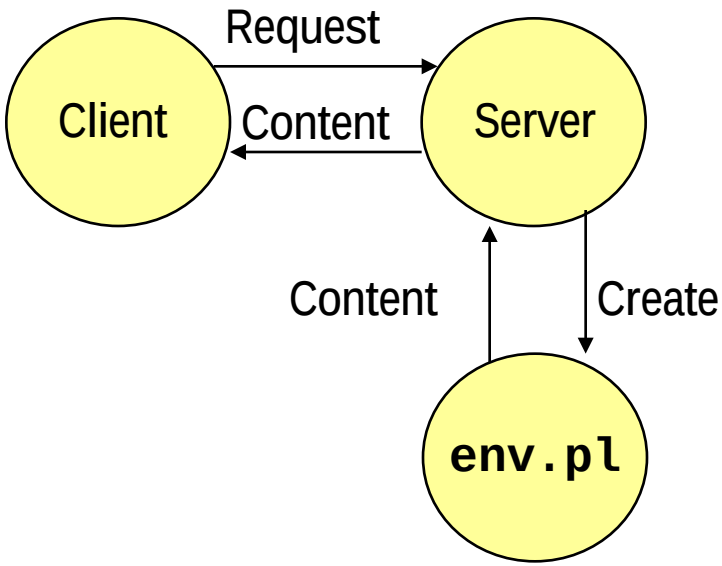
Serving Dynamic Content (cont)

- 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

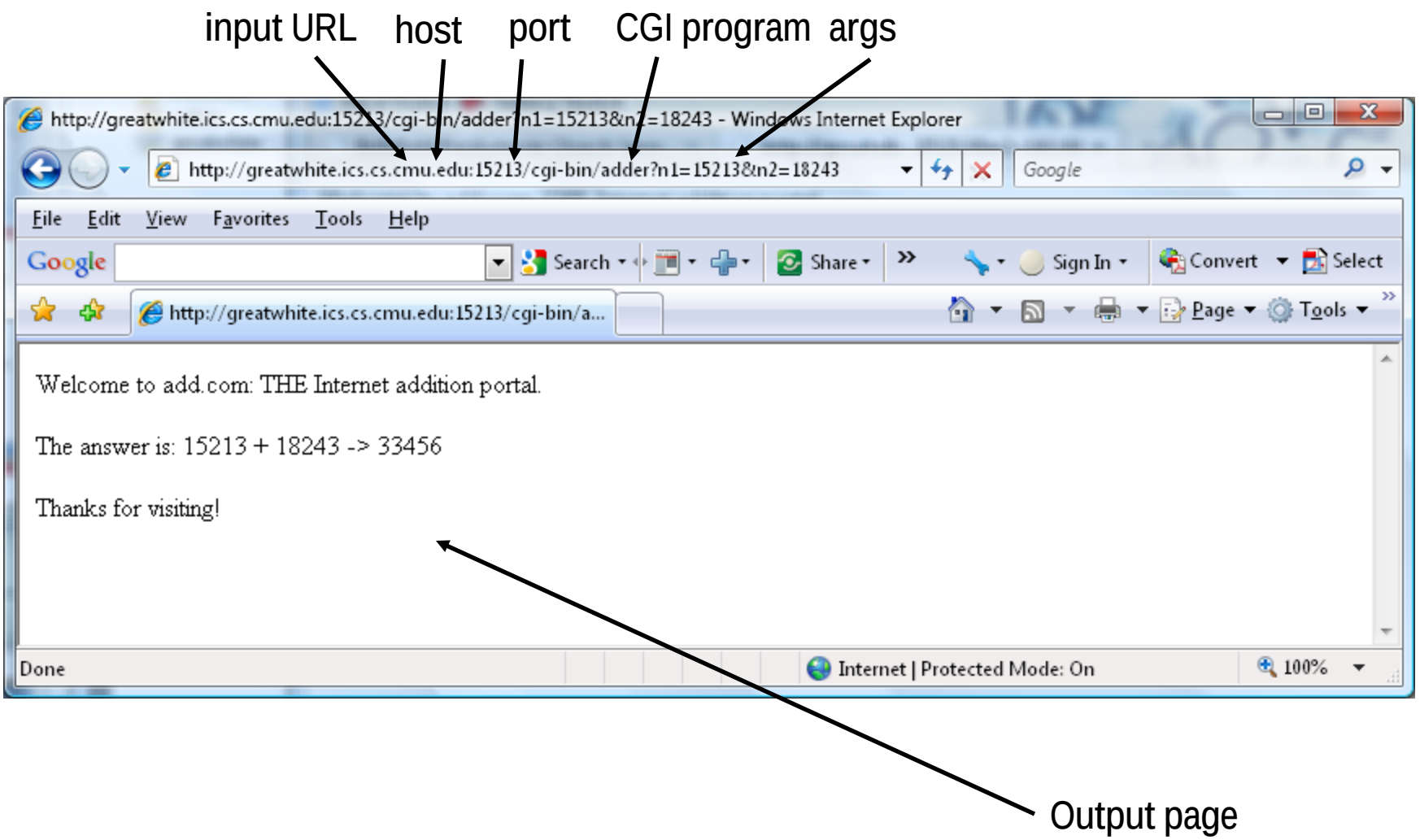
- How does the client pass program arguments to the server?
- How does the server pass these arguments to the child?
- How does the server pass other info relevant to the request to the child?
- How does the server capture the content produced by the child?
- These issues are addressed by the Common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



CGI

- Because the children are written according to the CGI spec, they are often called *CGI programs*.
- Because many CGI programs are written in Perl, they are often called *CGI scripts*.
- However, CGI really defines a simple standard for transferring information between the client (browser), the server, and the child process.

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?n1=15213&n2=18243`
  - `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:**
  - `cgi-bin/adder?n1=15213&n2=18243`

- **Result displayed on browser:**

Welcome to add.com: THE Internet addition portal. The answer is: 15213 + 18243 -> 33456
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 = "n1=15213&n2=18243"

From `adder.c`

```
if ((buf = getenv("QUERY_STRING")) != NULL) {
   if (sscanf(buf, "n1=%d&n2=%d\n", &n1, &n2) == 2)
     sprintf(msg, "%d + %d -> %d\n", n1, n2, n1+n2);
   else
     sprintf(msg, "Can't parse buffer '%s'\n", buf);
 }
```

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)

Even More CGI Environment Variables

- **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.
  - Notice that only the child knows the type and size of the content. Thus the child (not the server) must generate the corresponding headers.

From `adder.c`

```
/* 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: %s\r\n<p>",
            content, msg);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* Generate the HTTP response */
    printf("Content-length: %u\r\n", (unsigned) strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
```

Serving Dynamic Content With GET

```
linux> telnet greatwhite.ics.cs.cmu.edu 15213
Trying 128.2.220.10...
Connected to greatwhite.ics.cs.cmu.edu (128.2.220.10).
Escape character is '^]'.
GET /cgi-bin/adder?n1=5&n2=27 HTTP/1.1
host: greatwhite.ics.cs.cmu.edu
<CRLF>
HTTP/1.0 200 OK
Server: Tiny Web Server
Content-length: 109
Content-type: text/html

Welcome to add.com: THE Internet addition portal.
<p>The answer is: 5 + 27 -> 32

<p>Thanks for visiting!
Connection closed by foreign host.
```

HTTP request sent by client

HTTP response generated by the server

HTTP response generated by the CGI program

Tiny Serving Dynamic Content

From `tiny.c`

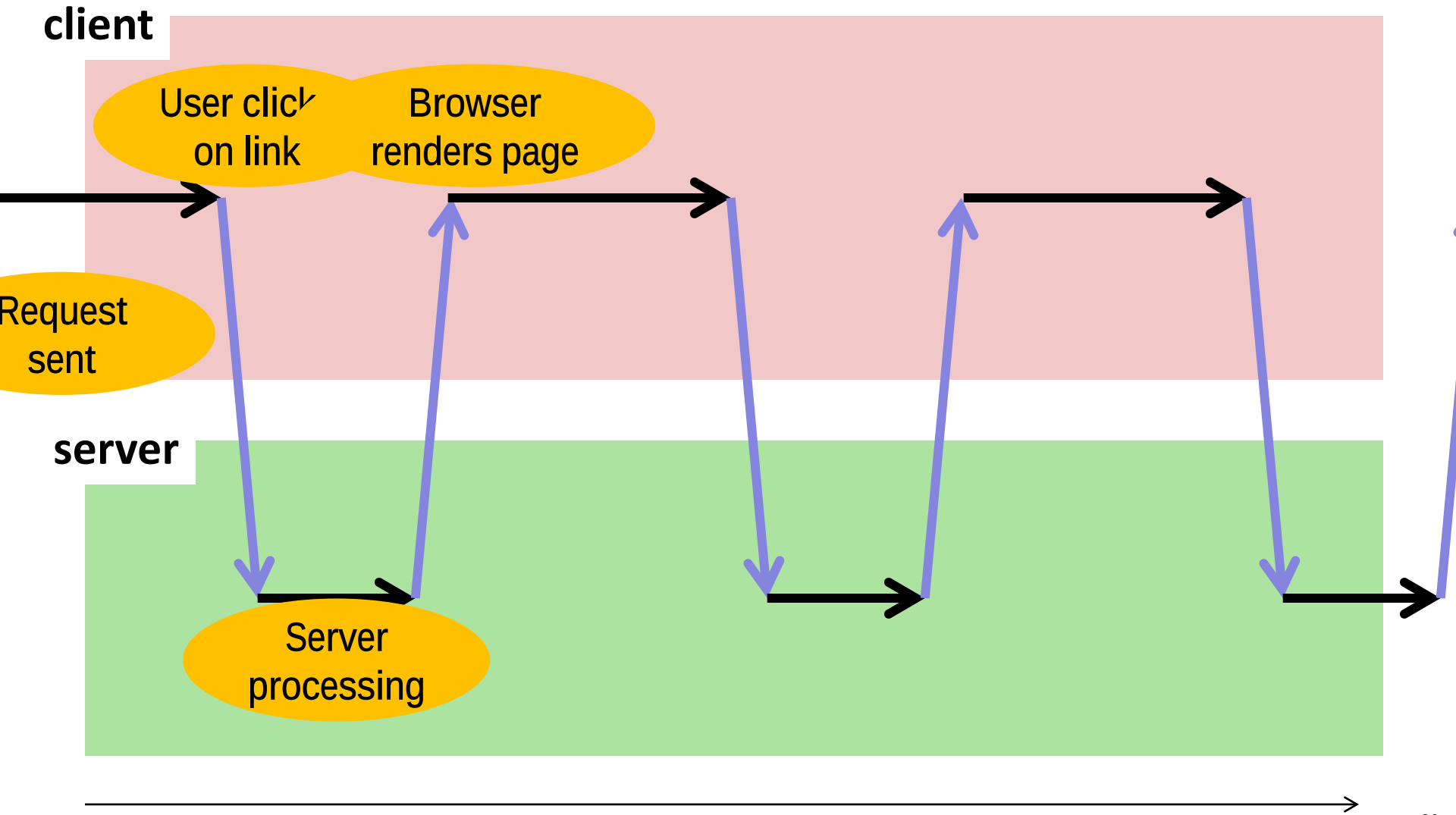
```
/* Return first part of HTTP response */
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Server: Tiny Web Server\r\n");
    Rio_writen(fd, buf, strlen(buf));

    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 prog  */
    }
    Wait(NULL); /* Parent waits for and reaps child */
```

- Fork child to execute CGI program
- Change stdout to be connection to client
- Execute CGI program with execve

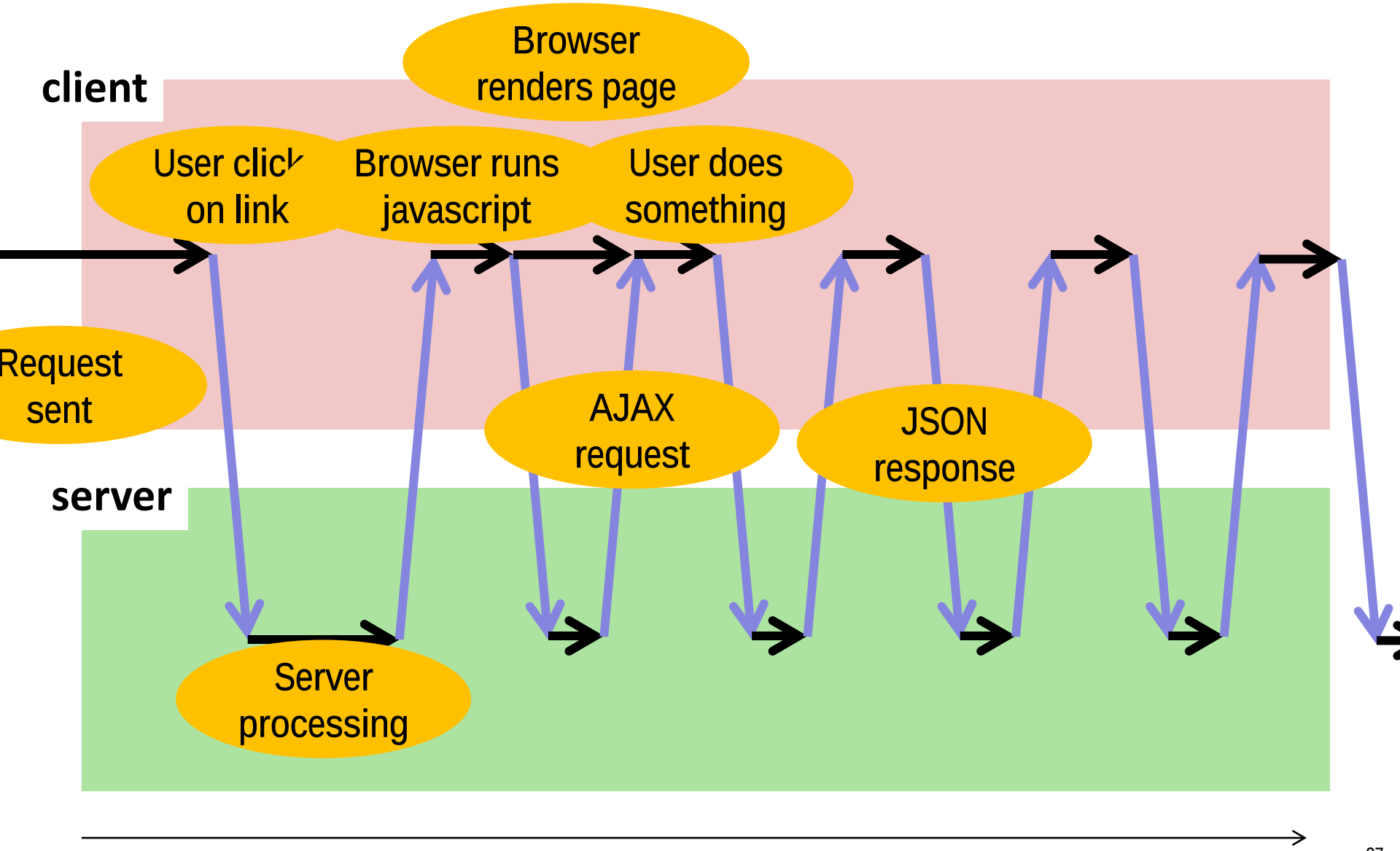
What really happens today?

- **Web page is a misnomer.**
  - www.facebook.com -> 114 requests!
    - Documents (2)
    - Images (79)
    - Scripts (18)
    - AJAX (8)
  - www.linkedin.com -> 123 requests!
- **Browser is a way to render the front-end of an application, "web" is an application framework**
  - client-side programming, e.g., javascript, AJAX
  - Server-side programming, e.g., ruby, php, nodejs
  - Database programming, e.g., sql, nosql

Web 1.0


client
User clicks on link
Browser renders page
Request sent
server
Server processing

Web 2.0


client
Browser renders page
User clicks on link
Browser runs javascript
User does something
Request sent
AJAX request
JSON response
server
Server processing

AJAX

- **Asynchronous JavaScript and XML (AJAX)**
  - A collection of client-side technologies that support interactive web applications
  - The key is are asynchronous requests to the server to get and store data without having to reload the page
- **Nothing more than a network request over http**
  - XMLHttpRequest object suppored in all major browsers
- **Made through javascript calls running in the client**
- **Typically returned data in either XML, or, even more likely JSON**

JSON: Javascript Object Notation

- **JavaScript-friendly notation**
  - Its main application is in Ajax Web application programming.
- **A method of serializing an object**
- **Represents a simple alternative to XML**
  - A text-based, human-readable format for representing simple data structures and associative arrays (called objects).
- **Used by a growing number of services**

JSON: Datatypes

- **Number** integer or floating point
- **String** double-quoted Unicode with backslash escaping
- **Boolean** true and false
- **Array** an ordered sequence of values, comma-separated and enclosed in square brackets
- **Object** collection of key:value pairs, comma-separated and enclosed in curly braces
- **null**

JSON: Example

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 19,
    "address": {
        "street": "5000 Forbes Ave",
        "city": "Pittsburgh",
        "state": "PA",
        "zip": "15213"
    },
    "phoneNumbers": [
        {   "type": "home",   "number": "412 555-1234" },
        {   "type": "cell",   "number": "412 555-4567" }
    ],
    "enrolled": false,
    "previous": null
}
```

Our simple adder for web 2.0

- Introduce form to gather operands of add
- When user hits submit, javascript sends back an ajax request

/a/add/15213/18243

- Server parses URI and executes add script and returns

{"status": 0, "result": 3346 }

- Client-side javascript processes result and renders result

Our initial web page

```
<HTML>
<HEAD>
  <script type="text/javascript" src="jquery.min.js"></scr
  <script type="text/javascript" src="adding.js"></script>

</HEAD>
<body>
<h1>Welcome to add.com</h1>
<p>THE Internet addition portal.</p>
n: <input type="input" length="5" id="n"></br>
m: <input type="input" length="5" id="m"></br>
<input type="button" id="doit" value="add"></br>
result is: <span id="result"></span>
</body>
</HTML>
```

Our client-side javascript

```
$(document).ready(function() {
 $('#doit').click(function() {
  var n=$('#n').val();
  var m=$('#m').val();
  $.ajax({ url: '/a/add/'+n+'/'+m,
           cache: false,
           dataType: 'json',
           success: function(reply) {
                  $('#result').empty();

      $('#result').append(reply.result);
                }
         });
 });
});
```

Our server loop

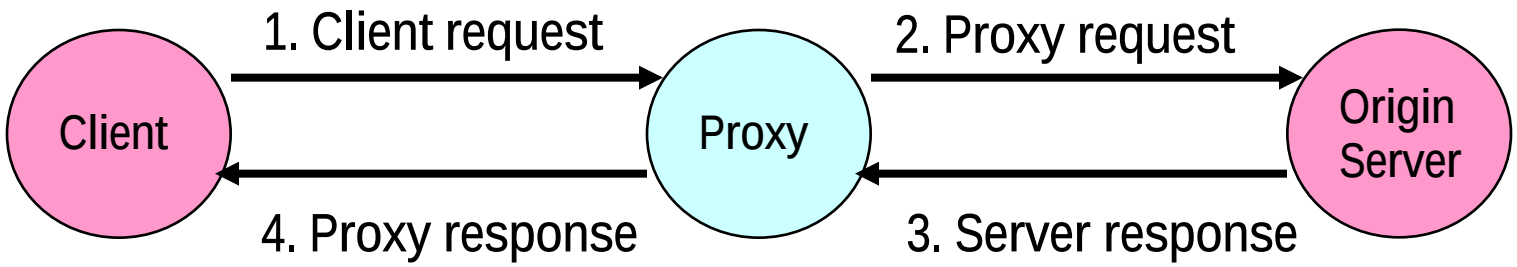
```
var http = require('http');
var url = require("url");
var fs = require('fs');

http.createServer(function (req, res) {
    var reqdata = url.parse(req.url, true);
    var args = reqdata.pathname.split('/');
    console.log('request: %j', args);
    processReq(reqdata.pathname, args, res);
}).listen(8896, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8896/');
```

Processing requests

```
function processReq(pathname, args, res) {
  if (args[1] == 'a') {
    res.writeHead(200, {"Content-Type": 'application/json'});
    if (args[2] == 'add') {
      var r=parseInt(args[3], 10)+parseInt(args[4], 10);
      res.write(JSON.stringify({status: 0, result:r}));
    } else {
      res.write(JSON.stringify({status: 1}));
    }
     res.end();
  } else {
    fs.readFile('../www'+pathname, "binary", function(err, file) {
        res.writeHead(200, { "Content-Type": 'text/html'});
        res.write(file);
       res.end();
    });
  }
}
```

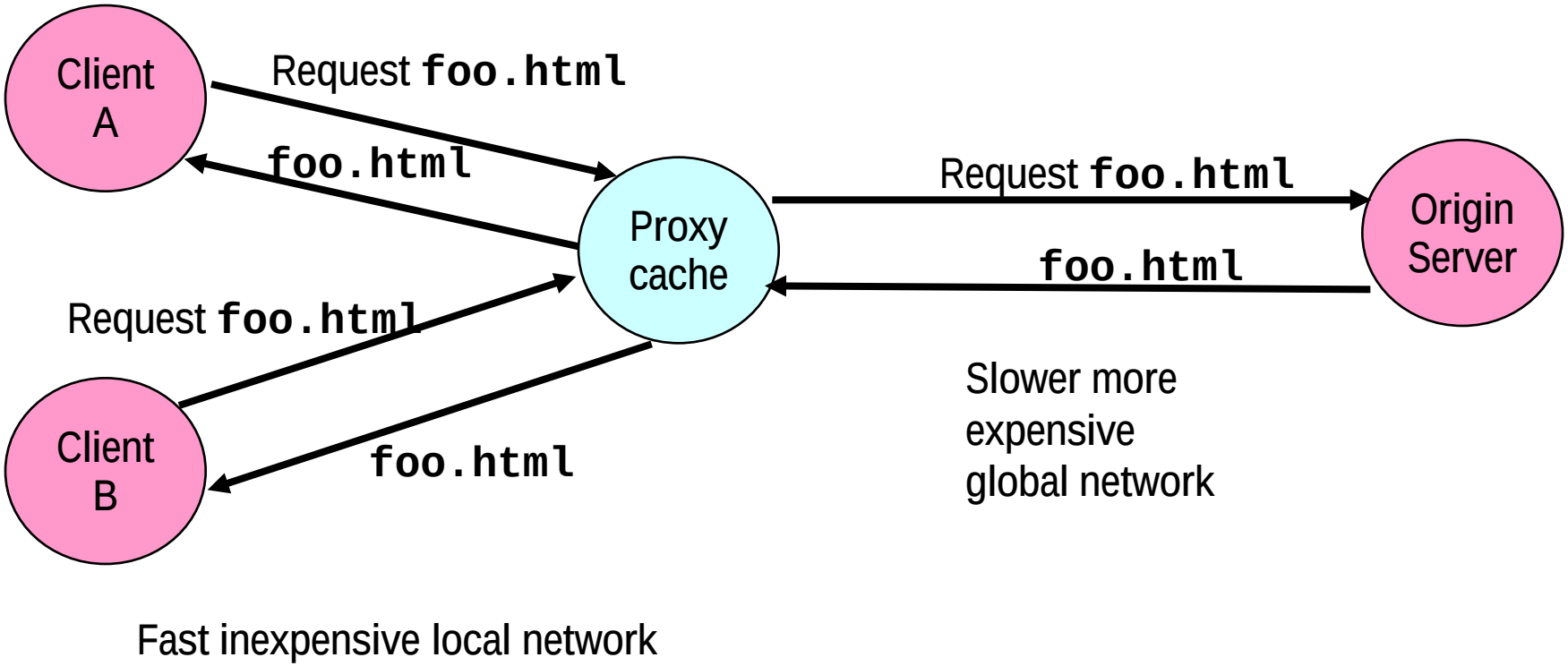
Proxies

- **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

For More Information

- **Study the Tiny Web server described in your text**
  - Tiny is a sequential Web server.
  - Serves static and dynamic content to real browsers.
    - text files, HTML files, GIF and JPEG images.
  - 220 lines of commented C code.
  - Also comes with an implementation of the CGI script for the add.com addition portal.

- **See the HTTP/1.1 standard:**
  - `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

Data Transfer Mechanisms

- **Standard**
  - Specify total length with content-length
  - Requires that program buffer entire message
- **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
```

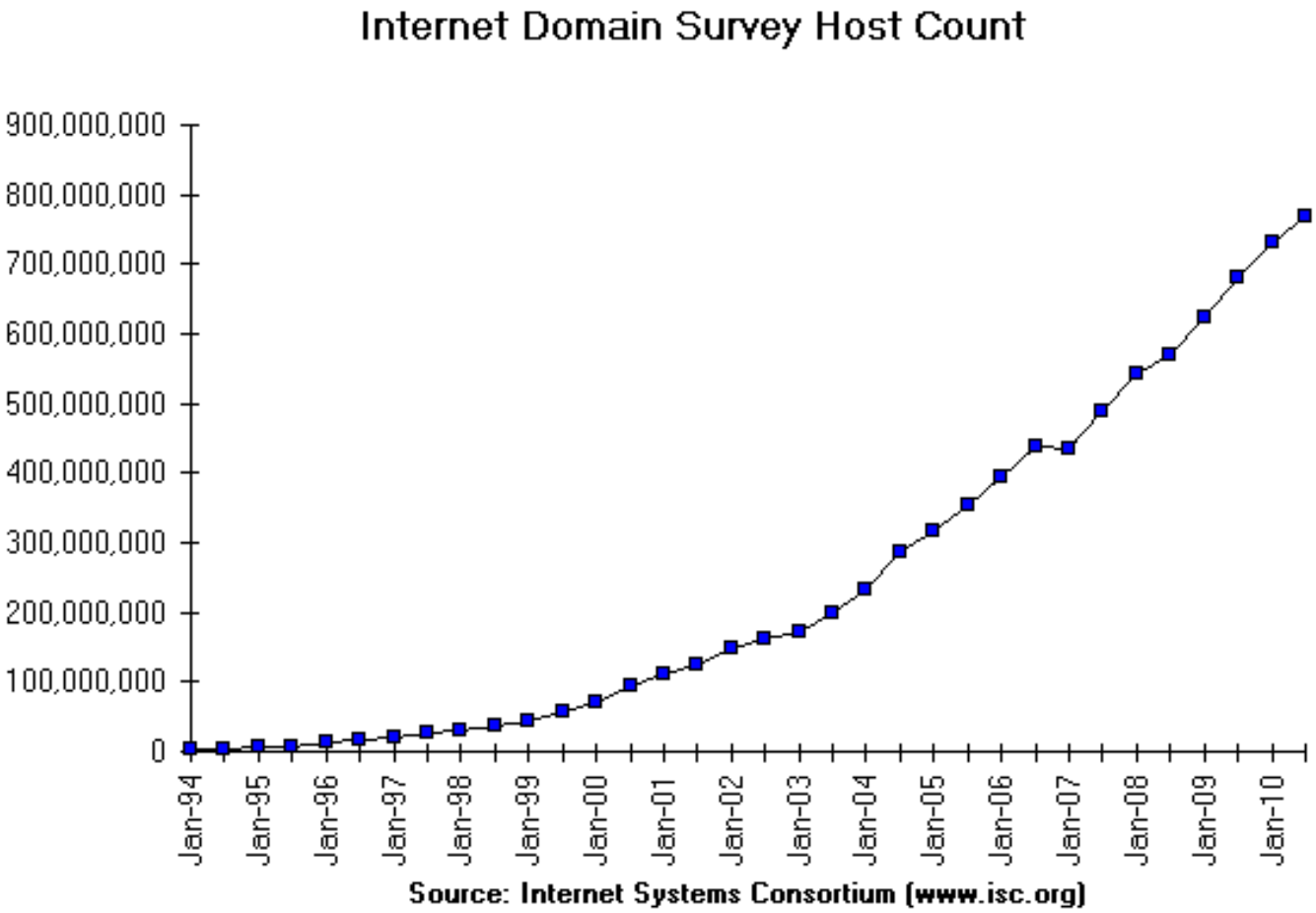
First Chunk: 0xd75 = 3445 bytes

Second Chunk: 0 bytes (indicates last chunk)

URLs

- **Each file managed by a server has a unique name called a URL (Universal Resource Locator)**
- **URLs for static content:**
  - `http://www.cs.cmu.edu:80/index.html`
  - `http://www.cs.cmu.edu/index.html`
  - `http://www.cs.cmu.edu`
    - 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`
- **Today distinction is really meaningless!**

Internet Hosts



- How many of the 2^{32} IP addresses have registered domain names?