



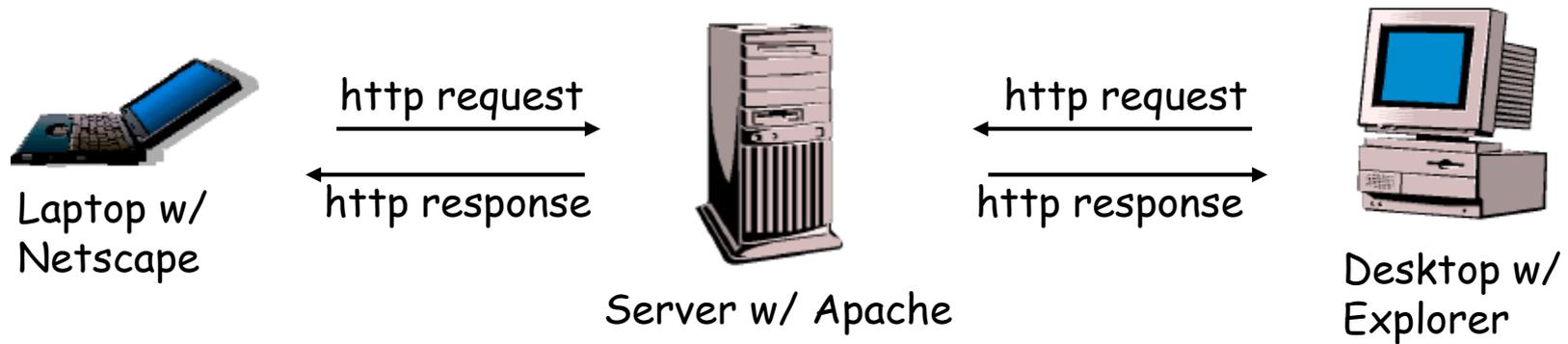
UNIVERSITY OF  
CALGARY

# HTTP Review

Carey Williamson

Department of Computer Science

University of Calgary



- HTTP: HyperText Transfer Protocol
  - Communication protocol between clients and servers
  - Application layer protocol for WWW
- Client/Server model:
  - Client: browser that requests, receives, displays object
  - Server: receives requests and responds to them
- Protocol consists of various operations
  - Few for HTTP 1.0 (RFC 1945, 1996)
  - Many more in HTTP 1.1 (RFC 2616, 1999)

- User clicks on something
- Uniform Resource Locator (URL):
  - `http://www.cnn.com`
  - `http://www.cpsc.ucalgary.ca`
  - `https://www.paymybills.com`
  - `ftp://ftp.kernel.org`
- Different URL schemes map to different services
- Hostname is converted from a name to a 32-bit IP address (DNS lookup, if needed)
- Connection is established to server (TCP)

- Client downloads HTML document
  - Sometimes called “container page”
  - Typically in text format (ASCII)
  - Contains instructions for rendering (e.g., background color, frames)
  - Links to other pages
- Many have embedded objects:
  - Images: GIF, JPG (logos, banner ads)
  - Usually automatically retrieved
    - I.e., without user involvement
    - can control sometimes (e.g. browser options, junkbusters)

```
<html>
<head>
<meta
name="Author"
content="Erich Nahum">
<title> Linux Web
Server Performance
</title>
</head>
<body text="#00000">


<h1>Hi There!</h1>
Here's lots of cool
linux stuff!
<a href="more.html">
Click here</a>
for more!
</body>
</html>
```

sample html file

- Respond to client requests, typically a browser
  - Can be a **proxy**, which aggregates client requests (e.g., AOL)
  - Could be search engine spider or robot (e.g., Keynote)
- May have work to do on client's behalf:
  - Is the client's cached copy still good?
  - Is client authorized to get this document?
- Hundreds or thousands of simultaneous clients
- Hard to predict how many will show up on some day (e.g., “flash crowds”, diurnal cycle, global presence)
- Many requests are in progress concurrently

```
GET /images/penguin.gif HTTP/1.0
User-Agent: Mozilla/0.9.4 (Linux 2.2.19)
Host: www.kernel.org
Accept: text/html, image/gif, image/jpeg
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: B=xh203jfsf; Y=3sdkfjej
<cr><lf>
```

- Messages are in ASCII (human-readable)
- Carriage-return and line-feed indicate end of headers
- Headers may communicate private information (browser, OS, cookie information, etc.)

## Called **Methods**:

- GET: retrieve a file (95% of requests)
- HEAD: just get meta-data (e.g., mod time)
- POST: submitting a form to a server
- PUT: store enclosed document as URI
- DELETE: removed named resource
- LINK/UNLINK: in 1.0, gone in 1.1
- TRACE: http “echo” for debugging (added in 1.1)
- CONNECT: used by proxies for tunneling (1.1)
- OPTIONS: request for server/proxy options (1.1)

- Similar format to requests (i.e., ASCII)

```
HTTP/1.0 200 OK
Server: Tux 2.0
Content-Type: image/gif
Content-Length: 43
Last-Modified: Fri, 15 Apr 1994 02:36:21 GMT
Expires: Wed, 20 Feb 2002 18:54:46 GMT
Date: Mon, 12 Nov 2001 14:29:48 GMT
Cache-Control: no-cache
Pragma: no-cache
Connection: close
Set-Cookie: PA=wefj2we0-jfjf
<cr><lf>
<data follows...>
```

- 1XX: Informational (def'd in 1.0, used in 1.1)  
100 Continue, 101 Switching Protocols
- 2XX: Success  
200 OK, 206 Partial Content
- 3XX: Redirection  
301 Moved Permanently, 304 Not Modified
- 4XX: Client error  
400 Bad Request, 403 Forbidden, 404 Not Found
- 5XX: Server error  
500 Internal Server Error, 503 Service Unavailable, 505 HTTP Version Not Supported

- This section describes the basics of servicing an HTTP GET request from user space
- Assume a single process running in user space, similar to Apache 1.3
- We'll mention relevant socket operations along the way

```
initialize;  
forever do {  
    get request;  
    process;  
    send response;  
    log request;  
}
```

server in  
a nutshell

```
s = socket(); /* allocate listen socket */
bind(s, 80); /* bind to TCP port 80 */
listen(s); /* indicate willingness to accept */
while (1) {
    newconn = accept(s); /* accept new connection */
}
```

- First thing a server does is notify the OS it is interested in WWW server requests; these are typically on TCP port 80. Other services use different ports (e.g., SSL is on 443)
- Allocate a socket and `bind()`'s it to the address (port 80)
- Server calls `listen()` on the socket to indicate willingness to receive requests
- Calls `accept()` to wait for a request to come in (and blocks)
- When the `accept()` returns, we have a new socket which represents a new connection to a client

```
remoteIP = getsockname(newconn) ;  
remoteHost = gethostbyname(remoteIP) ;  
gettimeofday(currentTime) ;  
read(newconn, reqBuffer, sizeof(reqBuffer)) ;  
reqInfo = serverParse(reqBuffer) ;
```

- **getsockname()** called to get the remote host name
  - for logging purposes (optional, but done by most)
- **gethostbyname()** called to get name of other end
  - again for logging purposes
- **gettimeofday()** is called to get time of request
  - both for Date header and for logging
- **read()** is called on new socket to retrieve request
- request is determined by parsing the data
  - Example: “GET /images/jul4/flag.gif”

```
fileName = parseOutFileName(requestBuffer);  
fileAttr = stat(fileName);  
serverCheckFileStuff(fileName, fileAttr);  
open(fileName);
```

- **stat()** called to test file path
  - to see if file exists/is accessible
  - may not be there, may only be available to certain people
  - "/microsoft/top-secret/plans-for-world-domination.html"
- **stat()** also used for file meta-data
  - e.g., size of file, last modified time
  - "Has file changed since last time I checked?"
- might have to **stat()** multiple files and directories
- assuming all is OK, **open()** called to open the file

```
read(fileName, fileBuffer);  
headerBuffer = serverFigureHeaders(fileName, reqInfo);  
write(newSock, headerBuffer);  
write(newSock, fileBuffer);  
close(newSock);  
close(fileName);  
write(logFile, requestInfo);
```

- `read()` called to read the file into user space
- `write()` is called to send HTTP headers on socket  
(early servers called `write()` for *each header!*)
- `write()` is called to write the file on the socket
- `close()` is called to close the socket
- `close()` is called to close the open file descriptor
- `write()` is called on the log file