Hypertext Transport Protocol

James Brucker

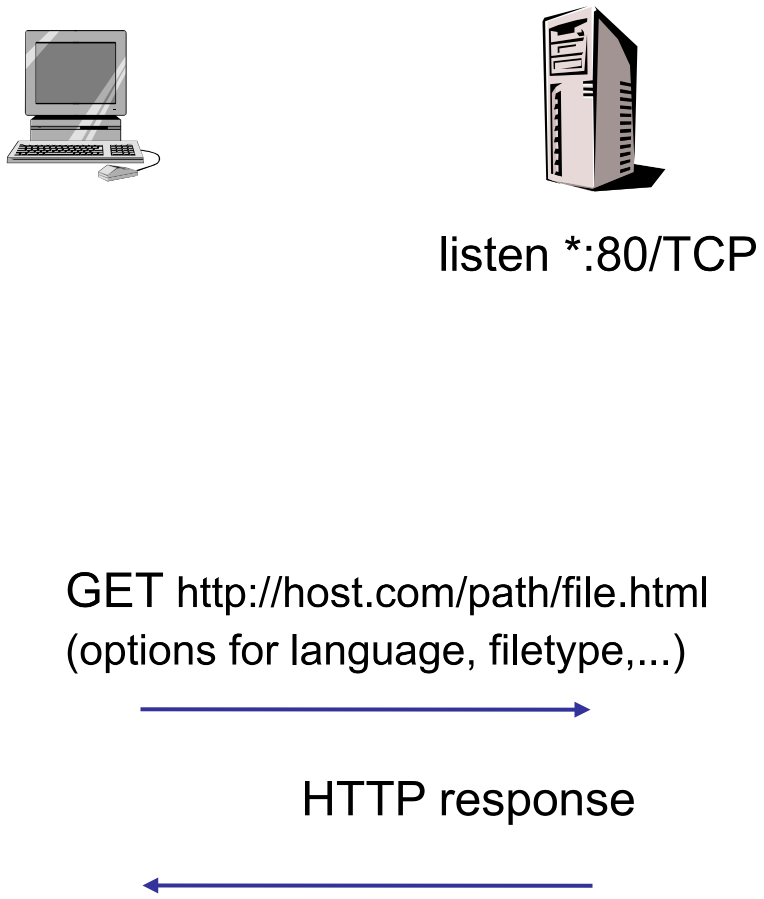
Slides from Web Services course.

Hypertext Transport Protocol (HTTP)

- Used to access Web *resources*
- Mostly widely used protocol on the Internet
- Platform independent
- Human readable

HTTP Request / Response

- "Request - response" protocol
- Server is always listening for requests
- Client sends an HTTP request
- Server processes it and sends response
- Server is *stateless* - not required to remember any previous requests.



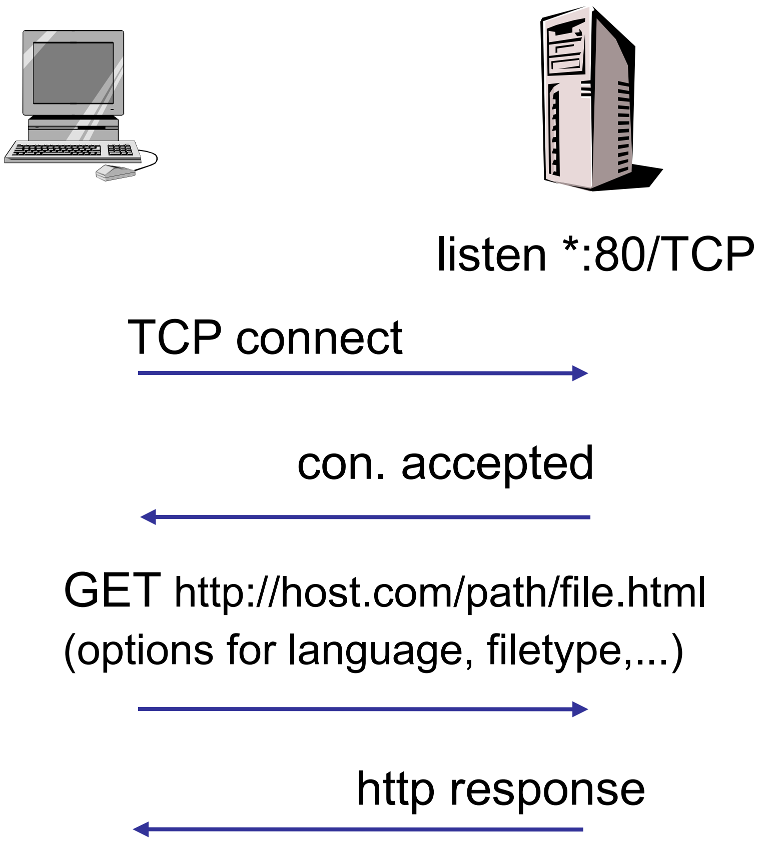
listen *:80/TCP

GET http://host.com/path/file.html
(options for language, filetype,...)

HTTP response

HTTP Runs on TCP

- Server listens on TCP port 80
- Each HTTP requests may need 3 packets just to establish a connection
- HTTP 1.0: one request/reply per connection. Connection closed immediately.
- HTTP 1.1 allows *persistent* connections (many requests and reply) to improve performance



listen *:80/TCP

TCP connect

con. accepted

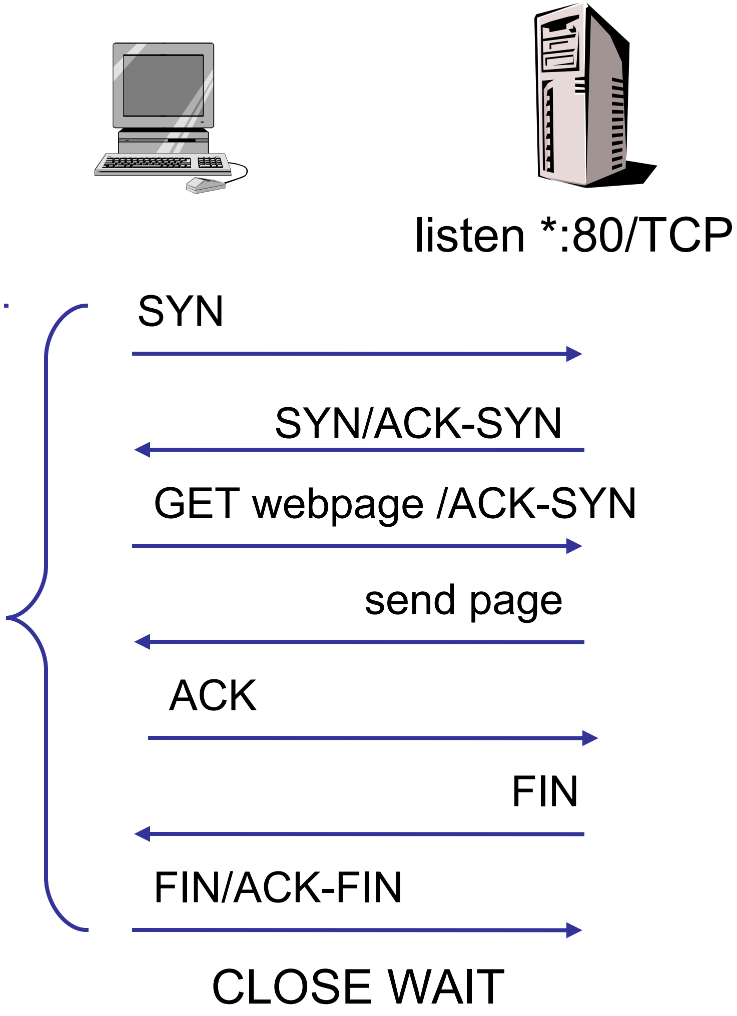
GET http://host.com/path/file.html
(options for language, filetype,...)

http response

Connection details - nonpersistent

- Used in HTTP 1.0
- Client establishes a new connection for each item included in Web page
- Delay for open + slow start
- lots of traffic and server overhead

Sequence repeated for *<u>every</u> web request!*



listen *:80/TCP

SYN

SYN/ACK-SYN

GET webpage /ACK-SYN

send page

ACK

FIN

FIN/ACK-FIN

CLOSE WAIT

Exercise: How many requests?

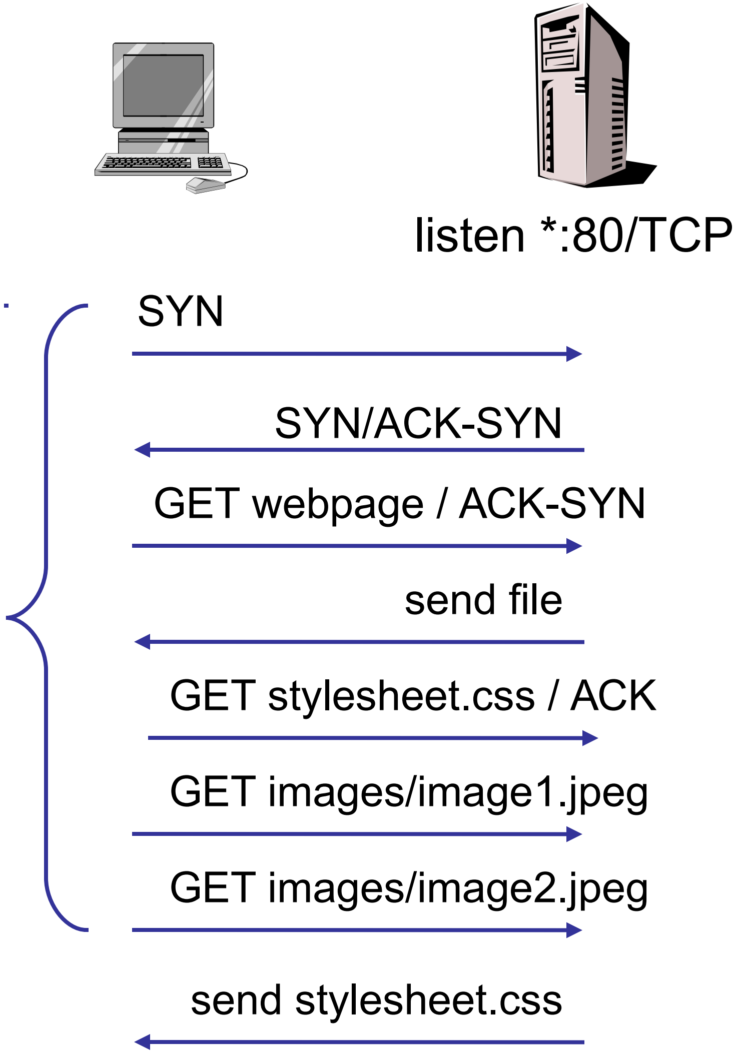
- To download and display this web page, how many requests does client have to send to server?
- For HTTP/1.0 how many connections to server are needed?

```
<HTML>
<link rel="stylesheet" href="stylesheet.css">
<BODY>
<h1>My vacation</h1>
<p>
For vacation we went to <a
  href=”http://www.unseen.com/bangkok”>Bangkok</a>.
Here's a photo of <em>Wat Phra Kaeo</em> <br>
<IMG SRC="images/watprakaew.jpeg">
```

Persistent Connection

- Default in HTTP 1.1
- Client can request using keep-alive
- server keeps connection open *briefly*
- client can pipeline requests
- client needs to know length of data

Multiple request/reply in
one connection



listen *:80/TCP

SYN

SYN/ACK-SYN

GET webpage / ACK-SYN

send file

GET stylesheet.css / ACK

GET images/image1.jpeg

GET images/image2.jpeg

send stylesheet.css

```
<HTML>
<link rel="stylesheet"
  href="stylesheet.css">
<BODY>
Some text and now an image
<IMG SRC="images/image1.jpeg">
```

HTTP Request Example

In browser you enter: http://www.cpe.ku.ac.th/index.html

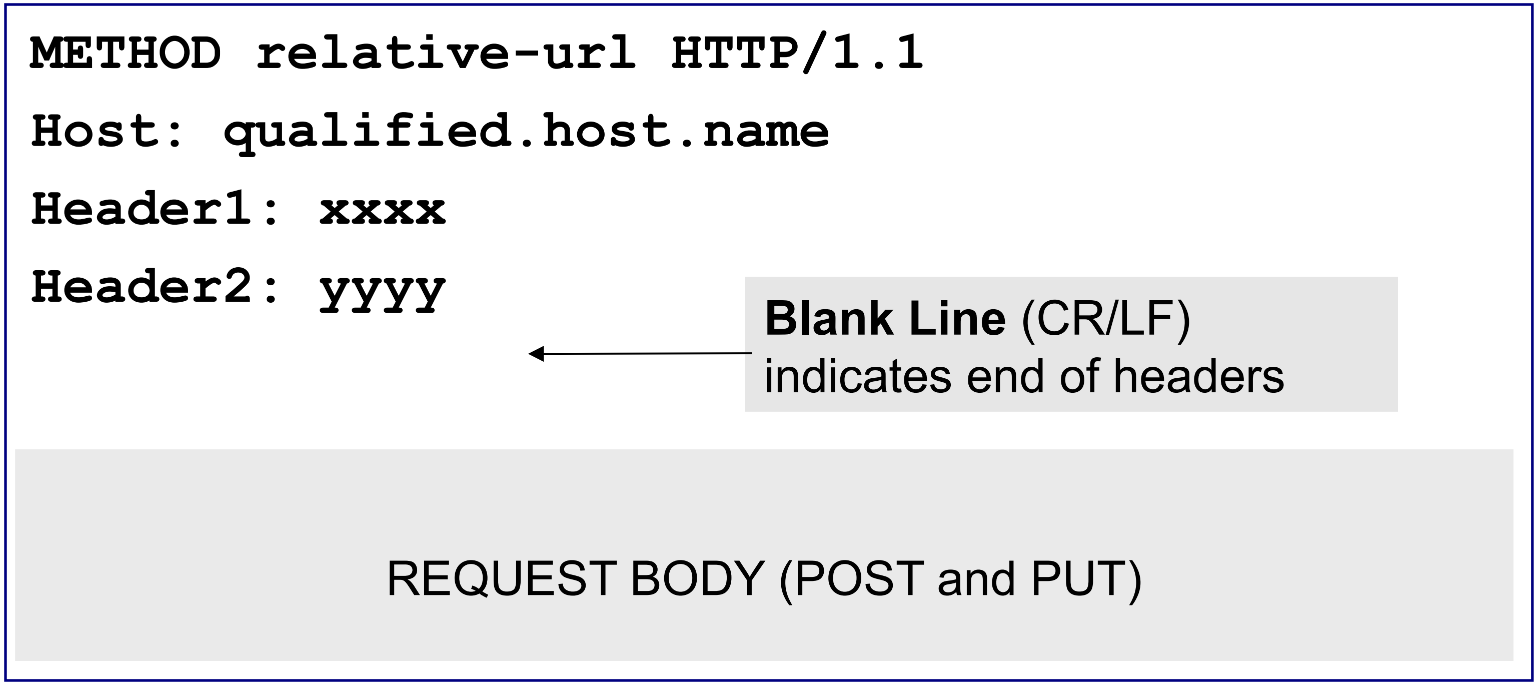
```
GET /index.html HTTP/1.1
Host: www.cpe.ku.ac.th
User-Agent: Mozilla/5.0
Accept: text/html, text/plain, image/gif,
  image/jpeg
Accept-Language: en, th;q=0.5
Accept-Charset: ascii, ISO8859-1, ISO8859-13
Accept-Encoding: gzip,deflate

```

Two CR/LF (one empty line) indicates end of headers

Accept: includes "**text/plain**" or "*/*" as a last resort.

HTTP Request Format



```
METHOD relative-url HTTP/1.1
Host: qualified.host.name
Header1: xxxx
Header2: yyyy
```

Blank Line (CR/LF) indicates end of headers

REQUEST BODY (POST and PUT)

Only POST and PUT methods have a REQUEST BODY

HTTP Request Methods

GET	get the *resource* specified by URL
POST	send information to server using body may have side effects; not repeatable
PUT	save or update resource at the given URL used to create or update resource at URL
DELETE	delete specified URL
OPTIONS	request info about available options
HEAD	retrieve meta-information about URL (used by search engines & web crawlers)
TRACE	trace request through network
CONNECT	connect to another server; used by proxies

Some Request Headers

w3schools.net and httpwatch.com have long list.

RFC2616: http://www.w3.org/Protocols/rfc2616/rfc2616.html

```
Accept: text/html,application/xhtml+xml,text/plain
Accept-Language: en-US,en-GB;q=0.5
Accept-Encoding: gzip, deflate
Host: www.google.com
User-Agent: Mozilla/5.0
Connection: Keep-Alive
Content-Length: 2048 (for POST and PUT)
X-Powered-By: Godzilla
```

Tools for Viewing Http Traffic

HttpFox (free) – monitor/inspect http requests (Firefox)

Great for monitoring what is happening.

Chrome "Developer Tools" – use Network tab to watch network traffic.

Curl – command line tool.

httpwatch – Watches all traffic. Can perform security checks. Chrome & Firefox plugin (free and paid versions) www.httpwatch.com

How many requests?

1) Install Firefox HttpFox add-on.

Use Ctrl-Shift-F2 to open (or add it to toolbar)

2) Get http:// www.cpe.ku.ac.th

How many requests did the browser send?

Why so many?

3) Repeat using Chrome Developer Tools → Network

Note: Look at the *timeline* of requests. Does the browser wait for a reply before sending next request?

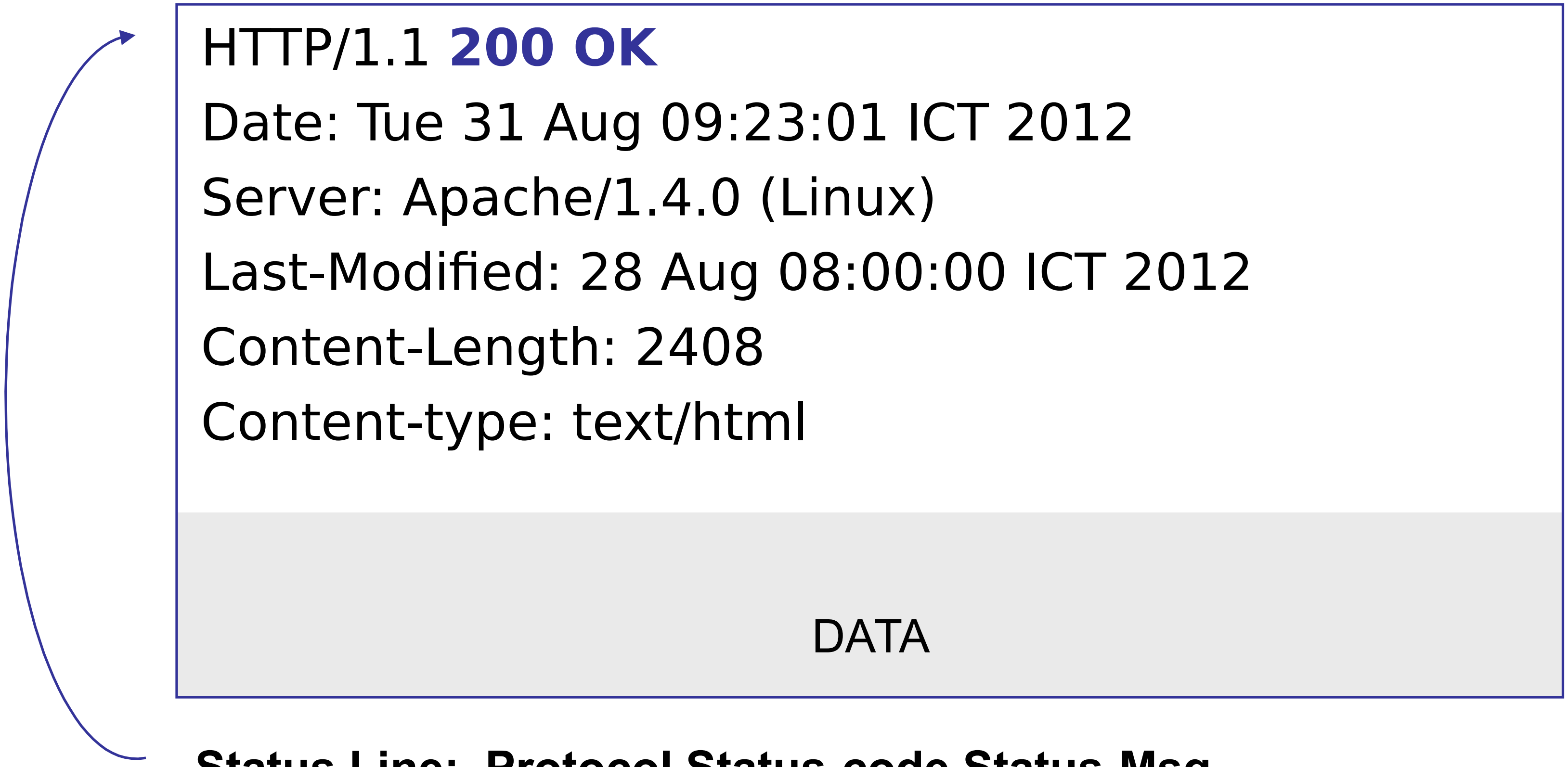
HTTP Response Example

```
HTTP/1.1 200 OK
Date: Mon, 28 Jul 2014
Server: Apache/2.2.24
Keep-Alive: timeout=5,max=100
Content-Type: text/html
Content-Length: 240

<html>
<head>
blah blah
<body>rest of the page</body></html>
```

Blank Line (CR) indicates end of headers

HTTP Response Message Format

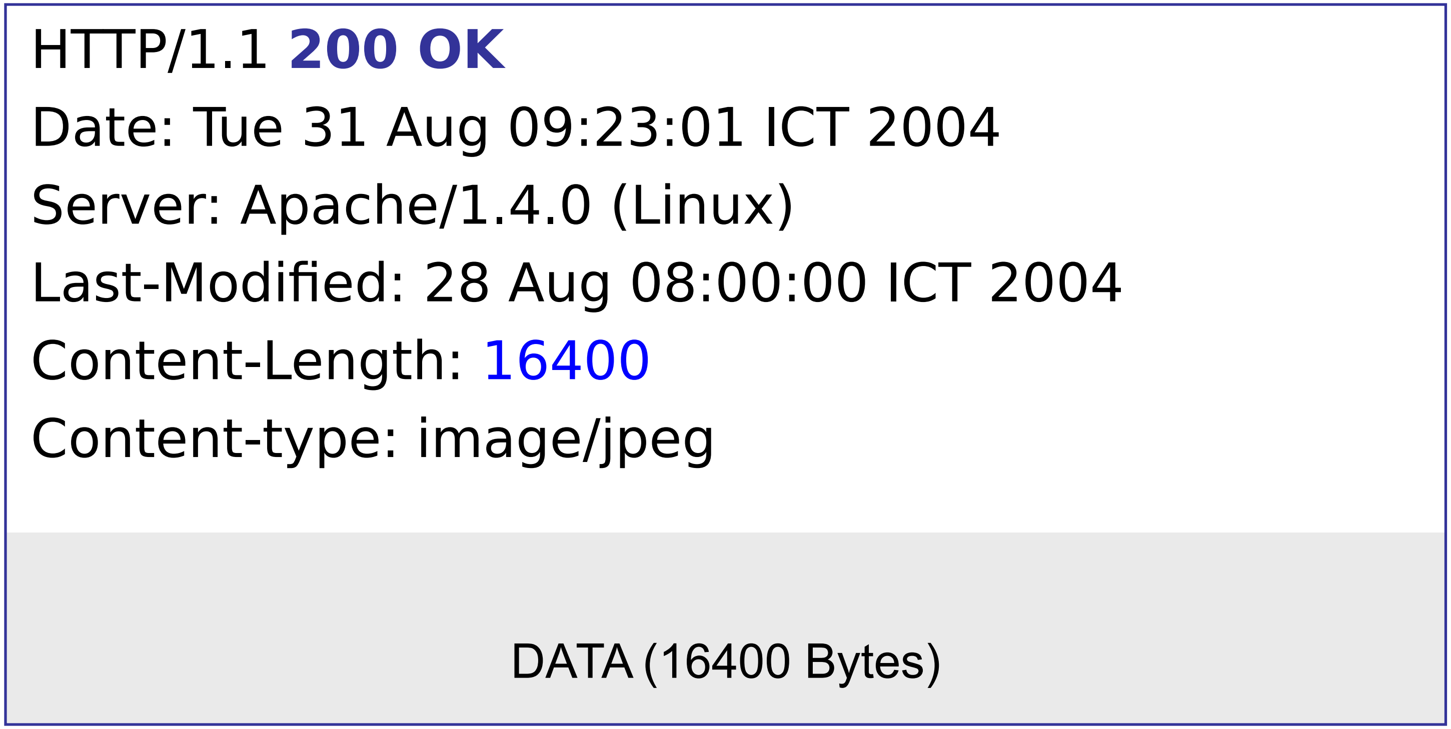


```
HTTP/1.1 200 OK
Date: Tue 31 Aug 09:23:01 ICT 2012
Server: Apache/1.4.0 (Linux)
Last-Modified: 28 Aug 08:00:00 ICT 2012
Content-Length: 2408
Content-type: text/html
```

DATA

Status Line: Protocol Status-code Status-Msg

Response Content-Length



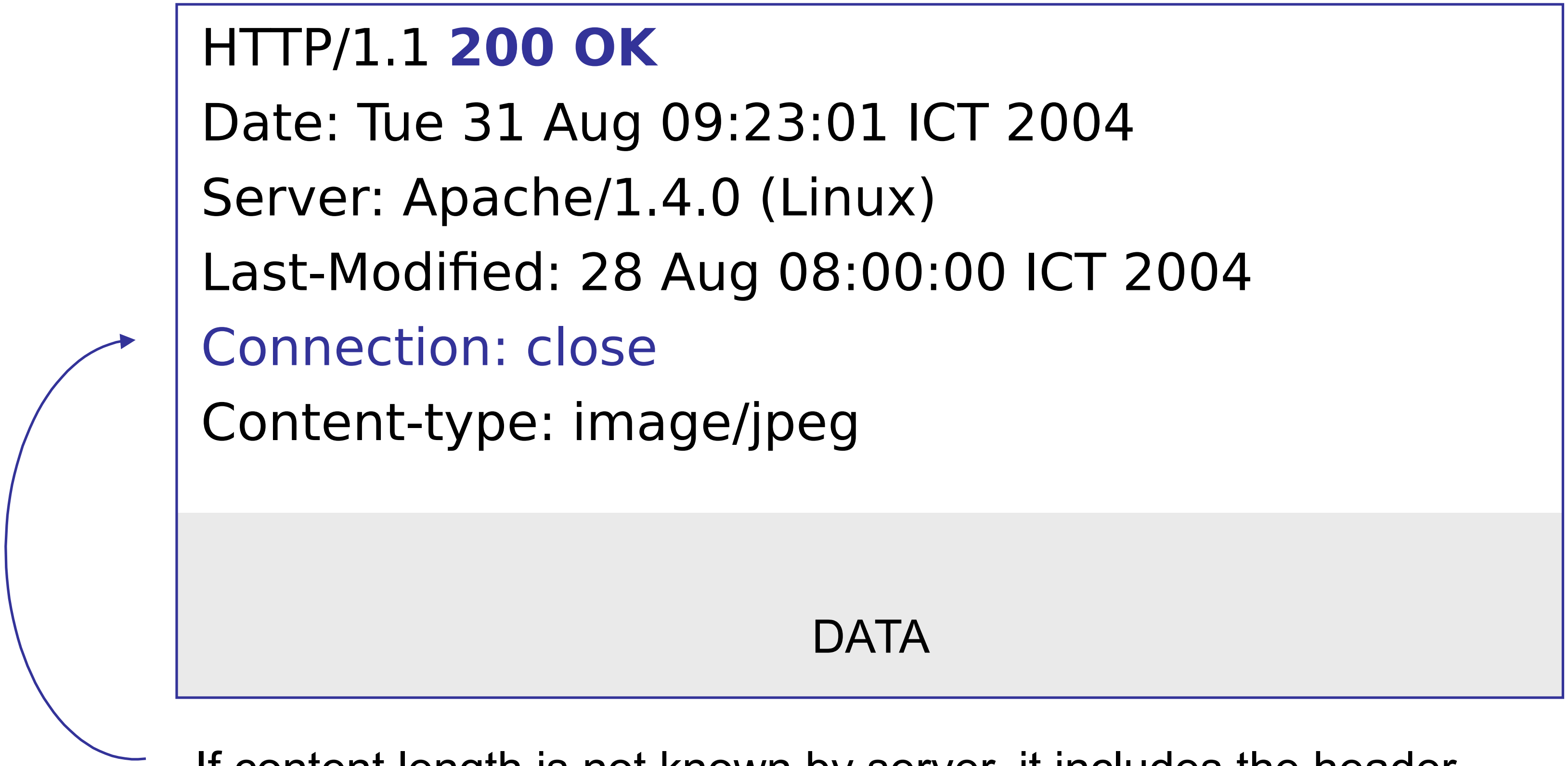
```
HTTP/1.1 200 OK
Date: Tue 31 Aug 09:23:01 ICT 2004
Server: Apache/1.4.0 (Linux)
Last-Modified: 28 Aug 08:00:00 ICT 2004
Content-Length: 16400
Content-type: image/jpeg
```

DATA (16400 Bytes)

For persistent connections, how does the client know amount of data to read for each object requested? For example, download of a JPEG file?

Client uses the Content-Length field.

Unknown Content Length



```
HTTP/1.1 200 OK
Date: Tue 31 Aug 09:23:01 ICT 2004
Server: Apache/1.4.0 (Linux)
Last-Modified: 28 Aug 08:00:00 ICT 2004
Connection: close
Content-type: image/jpeg

DATA
```

If content length is not known by server, it includes the header "Connection: close".

After the response is sent, it closes the connection.

This way, the client can read data until end-of-input.

Response Codes

```
HTTP/1.1 200 OK
```

Response Codes:

1xx Information

100 Continue

2xx Success

200 OK

201 Created (no response body)

202 Accepted (I'll process your request later)

3xx Redirection

301 Moved Permanently. New URL in `Location` header.

302 Moved Temporarily. New URL in `Location` header.

303 Redirect and change POST to GET method

304 Not Modified (*Look in your cache, stupid*)

Error Response Codes

4xx Client Error

400 Bad Request

401 Not Authorized (client not authorized to do this)

404 Not Found

5xx Server Error

500 Internal Server Error (application error, config prob.)

503 Service Unavailable

List of all HTTP status codes:

http://stat.us

http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Why is "Host:" header required?

HTTP requires each request include the "Host:" header. Why?

```
GET /index.html HTTP/1.1
Host: www.cpe.ku.ac.th
```

Surely, the receiver must know its own host name!

… or does it?

Tools for a Single Request

Sometimes we want to...

- *manually create* & send an HTTP request (for testing)
- control what headers are sent
- *inspect* details of the HTTP request and response

These tools are great for testing your web service:

HttpRequester (Firefox)

Rest Console (Chrome)

Dev HTTP Client *aka* "Rest HTTP API Client" (Chrome)

Get KU's Home Page

Use Chrome DHC extension.

1) send a GET request to: http: //www.ku.ac.th

What is the response?

2) send a GET request to the refresh url in the response.

What is the new response?

Where does it tell you to go? What is different?

3) send a GET to the new location.

Keep going...

How would you make KU's web site more efficient?

Get KU's Home Page in *English*

After you successfully get KU's home page,
try adding some request headers (one at a time):

Accept-Language: en
Accept: text/plain
Accept: image/*

Do they work?

What *methods* does this URL allow? Do they work?

Example Web Services

Explore California

http://services.explorecalifornia.org/pox/tours.php

(pox = Plain Old XML, or "rss" or "json")

Google Maps API

http://maps.googleapis.com/maps/api/geocode/xml?

address=Kasetsart%20University&sensor=false

Command Line HTTP Tools

Sometimes you need to use command line

- curl - command line HTTP client (from Unix)
- telnet - really primitive way to access any TCP port
- Windows: MinGW+Msys provide curl and a better telnet

curl Examples

- Get a resource (web page, image, anything):

```
curl -v  http://somehost.com/favicon.jpg
```

- Send a POST request with username=hacker

```
curl http://somehost.com/login.jpg
                     --data username=hacker
```

- Specify a header option in request

```
curl -H "Accept: text/plain" http://somehost.com/path
```

- Get help

```
curl --help
```

Many options have 2 forms: -d *or* --data

curl Exercise

Get KU's home page *in English.*

```
cmd> curl -H "Accept-language: en"
  http://www.ku.ac.th/web2012/index.php
```

Telnet: "Do it yourself HTTP"

Telnet is a command-line tool for connection to another host. Use telnet to manually create requests in any text protocol (http, smtp).

MinGW telnet is easier to use than Windows telnet.

port

```
cmd> telnet www.ku.ac.th 80
GET /web2012/index.php HTTP/1.1
Host: www.ku.ac.th
Accept-language: en
```

Telnet (2)

Response contains link to new home: http://ku.ac.th/

```
cmd> telnet ku.ac.th 80
GET / HTTP/1.1
Host: ku.ac.th
Accept-language: en
```

Exercise

- Use telnet to get a web page from iup.eng.ku.ac.th
- Find the actual location of default home page
- What METHODS does it accept?
  - GET POST PUT HEAD OPTIONS DELETE ?
- Send some invalid requests and note the responses
  - send to invalid URL
  - send unsupported method: DELETE, PUT, POST
  - try to DELETE something!
  - send header that server can't handle, e.g:
    Accept: text/plain or application/xml
    Accept-language: jp

More mischief

The Web Service class student list is here:

https://www.regis.ku.ac.th/grade/download_file/class_01219451_571.txt

(You can download it w/o logging in.)

a) download it.

b) can you download other course lists?

c) can you upload a new file (using PUT or POST)?

Exercise

- Find a web page containing a FORM using POST

```
<form method=”POST” action=”some_url”>
   <input type=”text” name=”username” .../>
```

1. examine the page source
2. note the FORM URL and what fields it sends
3. send the form (with data) using Curl or Dev HTTP

```
POST /some/url HTTP/1.1
    Host: www.example.com
    Content-length: 26

name=jim&birthday=1/1/1900
```

Compression

Accept-Encoding: gzip, deflate

Allow server to compress response body.

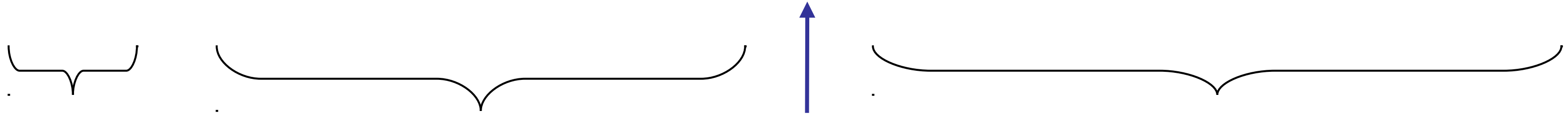
Q? Can HTTP transmit data in *binary form?*

Uniform Resource Locator

GET http://www.cpe.ku.ac.th/forms/junk.html?
name=jim%40.cpe.ku.ac.th&msgid=0x4412858798

General Form of a Uniform Resource Location (URL)

`http://www.cpe.ku.ac.th:80/~jim/images/cat.jpg`



Protocol:
`http`
`ftp,...`

Hostname
or IP address

Port

Path and resource name,
the path is optional

Path Parameters and Query Parameters (data)

http://www.ku.ac.th/somepath/file.cgi;*params*?*query*

URL Details

Encode special characters using %

//books.com/Web Svc becomes:

```
http://books.com/Web%20Svc
```

Path Parameters - extra info in path segment

```
http://finger.com/person;name=joe/telephone;co=th
```

Query Parameters (data) - used for GET

```
http://cpe.ku.ac.th/cgi-bin/file.cgi?name=joe&age=23
```

Surreptitious User Tracking

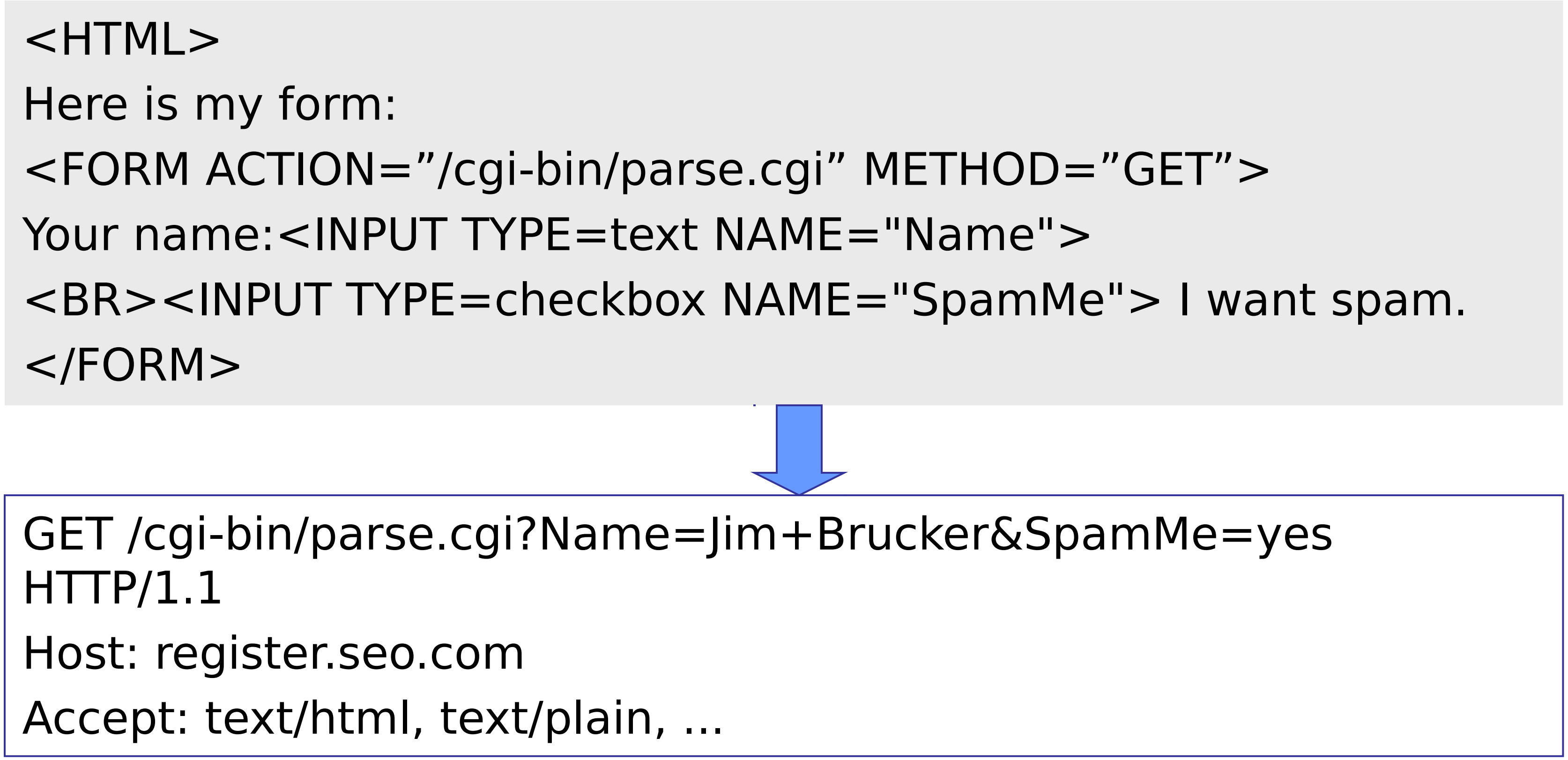
If you *open an E-mail message*, does the sender know you looked at it?

```
<HTML>
<BODY>
Hello, victim.  So you think just opening e-mail is safe?
Well, think again.  You'll be getting more SPAM from us soon!
<img src=http://www.spammer.com/images/barf.gif?id=428683927566 />
<!-- this is better, no query params -->
<img src=http://www.spammer.com/images/428683927566.gif? />
```

"GET" in HTML Forms

Two methods of sending data from HTML forms to Web server: GET and POST.

GET includes all form data in the URL.



```
<HTML>
Here is my form:
<FORM ACTION=”/cgi-bin/parse.cgi” METHOD=”GET”>
Your name:<INPUT TYPE=text NAME="Name">
<BR><INPUT TYPE=checkbox NAME="SpamMe"> I want spam.
</FORM>
```

```
GET /cgi-bin/parse.cgi?Name=Jim+Brucker&SpamMe=yes
HTTP/1.1
Host: register.seo.com
Accept: text/html, text/plain, ...
```

"POST" in HTML Forms

POST places the form data in the *body section* of the HTTP request. POST can transfer more data than GET.

```
<HTML>
Here is my form:
<FORM ACTION=”/cgi-bin/parse.cgi” METHOD=”POST”>
Your name:<INPUT TYPE=text NAME="Name">
<BR><INPUT TYPE=checkbox NAME="SpamMe"> I want spam.
</FORM>
```

```
POST /cgi-bin/parse.cgi  HTTP/1.1
Host: register.seo.com

Name=Jim+Brucker
SpamMe=yes
```

Implementing State

- HTTP is *stateless*
- So, how can web server remember (identify) client?
- How can server remember what page you are on?

How to Implement State

1. Hidden fields

```
<form method=”GET”>
<input type=”hidden” name=”id” value=”123456789”>
```

2. path parameters or custom URL

3. Cookies. In HTTP response, server adds header:

`Set-cookie:` some_string_or_random_number

Exercise: View your Cookies

- Look at some cookies in your browser cache.
- What information is included in a cookie?

Firefox: Preferences → Privacy → Remove Individual Cookies

Chrome: Settings → Show Advanced → [Content Settings] button → [All Cookies and Site Data]

Why does Chrome make cookies so hard to find?

Conditional GET

- A Client can request a resource only if it has been modified since a given date.
- Used for efficiency & caching.
- Use "If-modified-since: " or "Etag:" headers.

```
GET /path/index.html HTTP/1.1
If-modified-since:  1 Aug 18:32:00 ICT 2014
...etc...
```

If page has not been modified, the server responds:

```
HTTP/1.1 304 Not Modified
```

Conditional GET: server response

- If page **has** been modified, server responds:

```
HTTP/1.1 200 OK
Content-type: blah

DATA
```

- If page has **not** been modified, server responds:

```
HTTP/1.1 304 Not Modified
```

Conditional GET using Etag

- A server can include an "Etag" as page identifier. It is usually an MD5 hash but can be anything:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Etag:  "33101963682008"

Image data
```

- Next time the client needs the image (but its still in his cache) he sends:

```
GET /path/image.jpeg HTTP/1.1
If-None-Match: "33101963682008"
```

Web Caching

- Caching is critical to performance of the web
- Multiple levels of caching:
  - client (web browser cache)
  - server (manually configured cache)
  - gateway (transparent cache engine)
  - network (CDN, cooperating caches)

Cache Engines

- Harvest (free)
- Squid (free)
- Cisco Cache Engine (based on Linux and Harvest)

Why Web Caching?

- Decrease use of network bandwidth
- Faster response time
- Decrease server load
- Security and web access controls (auth, blocking)

Content Delivery Networks

- Akamai, DigitalIsland, etc.
- Has its own network of servers that replicates content of the content provider (e.g. cnn.com), e.g. all images
  - in the index.html file all references of:

    www.cnn.com/images/sports.gif
  - is re-mapped to www.akamai.com/www.cnn.com/images/sports.gif
- Akamai servers cache images and index files for cnn.com
- Server domain name: www.akamai.com
- Index file changed to: www.akamai.com/.../images/sports.gif

Content Delivery Networks (2)

- When client downloads http://www.cnn.com/index.html he gets a cached (modified) file from cache server, containing

<img src=http://www.akamai.com/www.cnn.com/images/sports.gif>

- Next, client tries to resolve "www.akamai.com"
- DNS server of Akamai will...
  - identify client's location based on client's IP address (database)
  - chooses one of Akamai's cache servers which is "closest" to the client's location
  - returns IP address for "www.akamai.com" closest to client.