

# Hypertext Transport Protocol

---

James Brucker

# Hypertext Transport Protocol (HTTP)

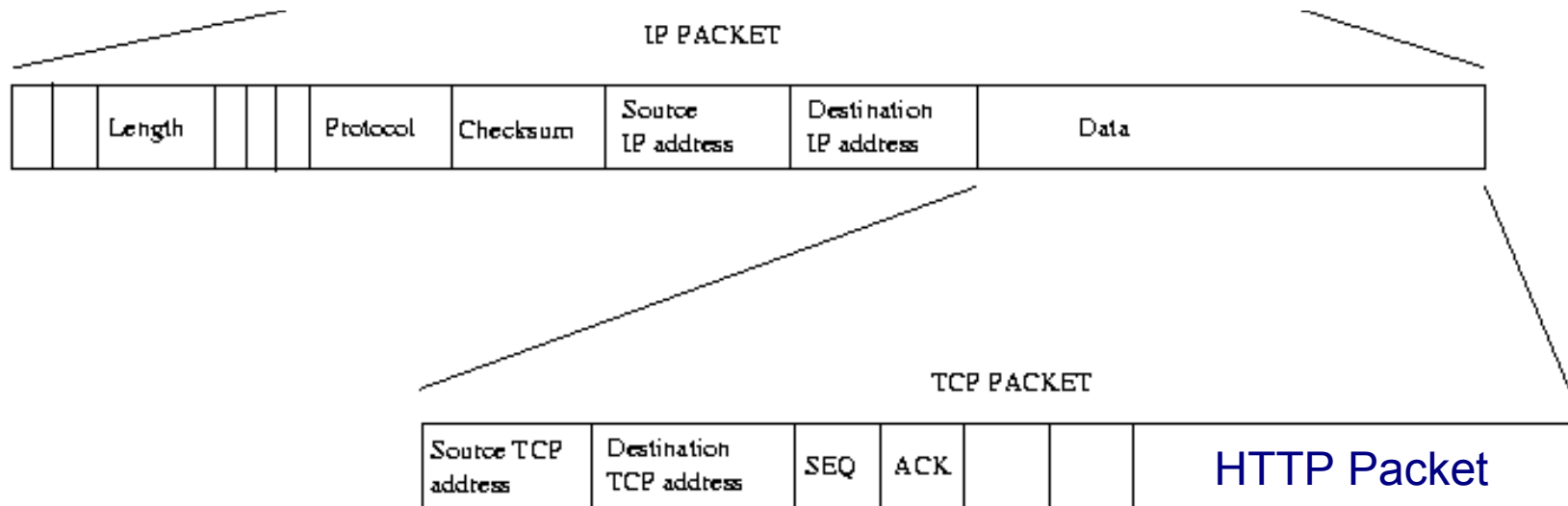
---

- Protocol used to access the Web
- Mostly widely used protocol on the Internet
- Platform independent
- Human readable

# HTTP uses TCP/IP

IP = **Internet Protocol**. Provides addressing and routing.

TCP = **Transmission Control Protocol**. Maintains a virtual connection between hosts, delivers data to apps in correct order, request resend of lost packets.



Source & dest.  
**port** numbers

# IP Addresses

---

IP version 4 - the most widely used:

4 bytes

158.108.216.5 - address of www.ku.ac.th

172.217.27.228 - www.google.com (has many addrs)

127.0.0.1 - "localhost". Address of your own host.

0.0.0.0 - address **pattern** meaning "anything"

IP version 6 - Newer IP with much larger address space

16 bytes

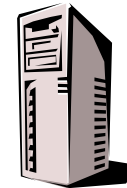
2406:3100:1010:100:0:0:0:5 - www.ku.ac.th

2406:3100:1010:100::5 - same thing, **0-bytes omitted**

2404:6800:4001:80e::2004 - www.google.com (has many)

# HTTP is Request / Response Protocol

- **Client** sends an HTTP request, server sends a response
- **Server** listens (waits) for incoming connections. It accepts client request and responds to it.
- Server is *stateless* - not required to remember any previous requests (but web apps may).

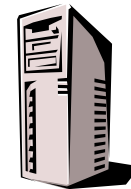


listen \*:80/TCP

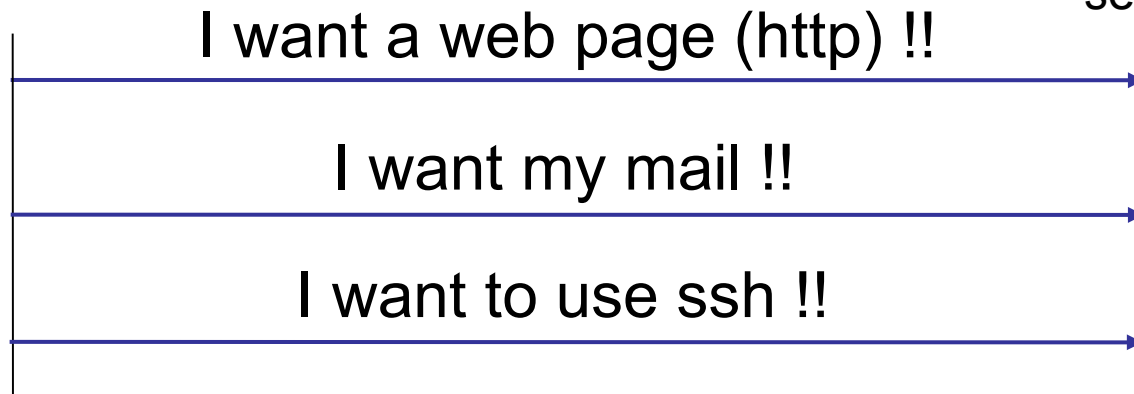


# What's a Port? Why do we need ports?

- A host may have many, many of internet connections at the same time!
- A server may offer many **services**: HTTP, mail, ssh, ...
- How does a host know **which packets** should go to **which application**???

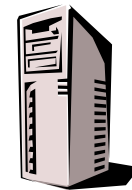


server

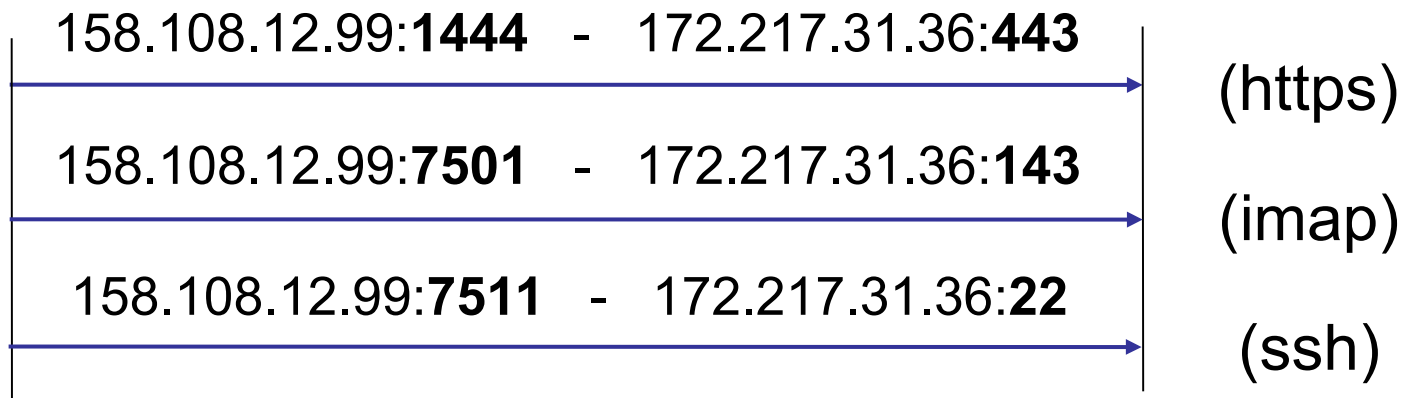


# Port is a number to identify connection

- A **connection** has a **port number** 1 - 65,535 for each end point.
- Servers ***listen*** for connections on **well-known port** nums.
- Each **ip\_address:port** pair identifies an endpoint.



server



# Port Numbers Identify *Services*

---

Standard ("well known") ports for common services.

<u>Service</u>	<u>TCP Port</u>
HTTP	80
HTTPS	443
Mail Transport (SMTP)	25, 465 (secure)
IMAP (client mail delivery)	143, 993 (secure)
SSH	22
MySQL server*	3306

*(\*) For security, you should not expose a database service to the Internet, and avoid the default port. See /etc/services, IANA, or Wikipedia for more ports.*



# A Service Can Use Any Port

---

Web servers usually use port 80 (http) and 443 (https).

But you can use **any port** for your web server.

Django development server listens on **port 8000** by default...

but you can tell it to use *any* port.

Ports 1-1023 are *privileged ports*. Only "root" or admin user can start a process on those ports.

# Exercise 1: View your connections

---

1. In a terminal window type:

```
Linux/MacOS> netstat -n --tcp
```

```
Windows> netstat -n -p tcp
```

2. In a web browser, visit a new web site.

3. Type "netstat" again ... are there new connections?

"-n" means show IP address instead of host name.

Omit -n to show host names, but it is slower.

# Exercise 2: Create Your Own Server

---

Use netcat (nc) or ncat for this:

1. Open a terminal window and start a server. `-l` means "listen", 4444 is port number. Any port > 1024 is ok.

```
cmd> netcat -v -l -p 4444
```

2. Open another terminal window and connect to "localhost" on port 4444. Type something...

```
cmd> netcat localhost 4444  
Hello? Is anyone there?
```

# HTTP uses TCP

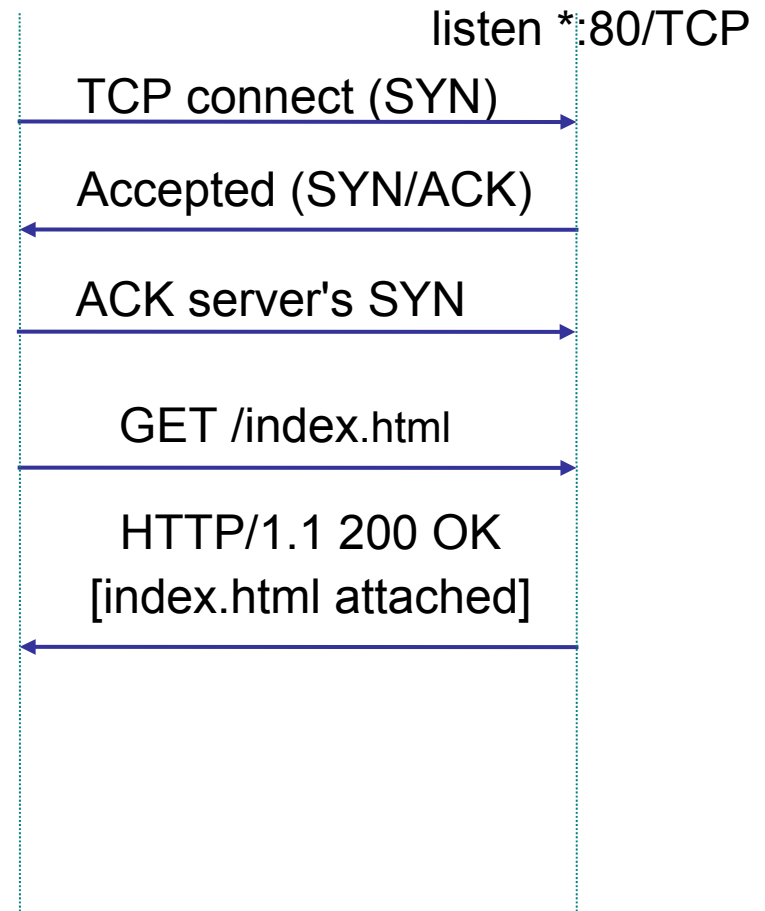
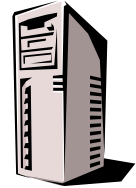
---

- HTTP uses TCP for data integrity and IP for transport
- TCP/IP connections are managed by the host OS.
- Creating a connection is **invisible** to web apps and web servers, but it affects *performance* so its good to know.

*Next slide illustrates...*

# Establish a TCP Connection

- TCP needs **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 compression for performance
- **HTTP/2.0** is much faster: header caching, multiplex overlapping requests, better compression



# HTTP Protocol Basics

---

1. HTTP Request format
2. HTTP Request methods
3. HTTP Response format
4. Header fields
5. Response codes (status codes)
6. URLs

# HTTP Request Example

In browser 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: server.host.name**

**Header1: xxxx**

**Header2: yyyy**

**Blank Line (CR/LF)**

← indicates end of headers

**REQUEST BODY (POST and PUT only)**

Only POST and PUT requests have a REQUEST BODY



# HTTP Request Methods

---

GET	get the <i>resource</i> 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

# Common Request Headers

**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 (X- = *custom headers*)

w3schools.net and httpwatch.com have a long list.

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

# 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

← Blank Line (CR) indicates  
end of headers

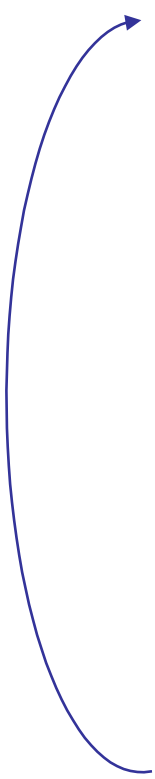
<html>

<head>blah blah</head>

<body>rest of the page</body>

</html>

# HTTP Response 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

First Line: **Protocol StatusCode 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](#), client needs to know how much data is in the server's response.

Example: server sends a JPEG file    How many bytes is it?

**Client uses the Content-Length header.**

# 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 uses the header "Connection: close".

After the response is sent, server 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 (a new resource was successfully created)

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](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)



# Why is "Host" header required?

---

HTTP Requests always include a "Host" header.

It is the name of the destination host.

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

WHY?

Surely, the server must **know** its own host name!

... or does it?

# Uniform Resource Locators (URL)

A Uniform Resource Locators (URL) locate resources on the Internet (not just the web).

Structure of a URL:

`http://www.cpe.ku.ac.th:8080/~jim/dictionary.txt`

Protocol:

http  
ftp  
jdbc  
file  
mysql

Hostname and port

or IP address

Port is optional

Path and resource name,

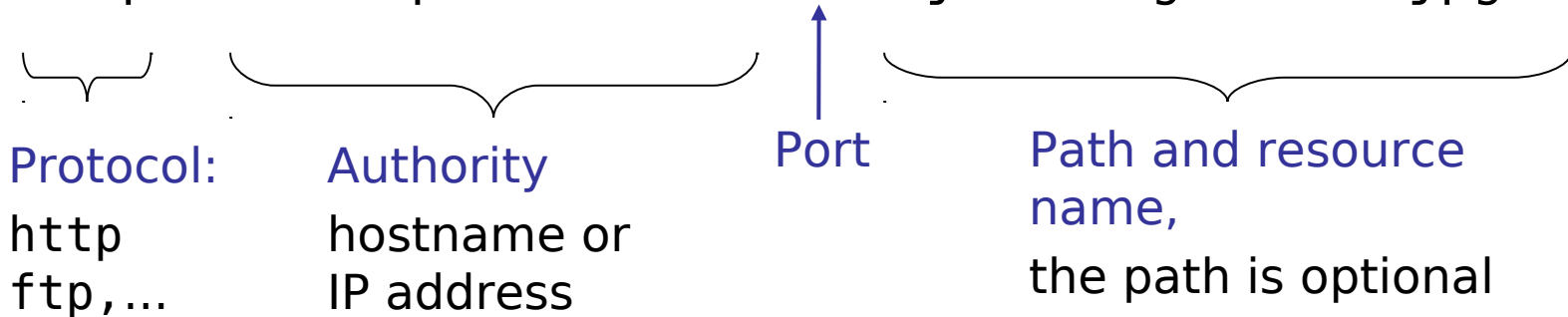
the path is optional

# Uniform Resource Locator

`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`



# URL Details

---

Encode special characters using %

http://host.com/web svc becomes:

`http://host.com/seb%20svc`

**Path Parameters** - extra info in path segment

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

**Query Parameters** - used for GET

`http://host.com/adduser.cgi? name=joe&age=23`

# URL for File, URL with user info

Use a web browser to open a **FILE** on your computer:

```
file:///home/me/workspace/unittesting/fraction.py
```

You can omit `"/"` since there is no host:

```
file:/home/me/workspace/unittesting/fraction.py
```

May include **user info** in a URL:

```
protocol://username:password@hostname/...
```

```
http://jim@cpe.ku.ac.th/something
```

**URL for database** (Django `django.db.backends` uses this):

```
mysql://myuser:mypassword@hostname/mydatabase
```

# Exercises

---

End of the HTTP basic slides.

Do the exercises described in class,  
or see the "*HTTP-in-Action*" slides.

# Optional Material

---

Stuff you aren't required to know.

Do the HTTP exercises first.

# "GET" in HTML Forms

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

GET puts 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"> 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 puts the form data in the *body* 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"> 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) a client?
- How can server remember what page you are on?

# How to Implement State

---

3 common ways:

## 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?*

# Exercise: 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?

# 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

# Tools for Viewing Http Traffic

---

**HttpFox or HttpRequester (free)** – monitor/inspect http requests (Firefox). Great for seeing what is happening.

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

**Rest Console** (Chrome)

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

**httpwatch** – Watches all traffic. Can perform security checks. Chrome & Firefox plugin (free and paid versions) [www.httpwatch.com](http://www.httpwatch.com)

These tools are great for testing web services.

# Command Line HTTP Tools

---

Sometimes you need to use command line

- **curl** - command line HTTP client (from Unix)
- **netcat (nc)** - send TCP or UDP, **listen** for TCP or UDP
- **telnet** - primitive way to access any TCP port



# 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.exploreocalifornia.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](http://maps.googleapis.com/maps/api/geocode/xml?address=Kasetsart%20University&sensor=false)

# 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
```

# Experiment with methods & headers

---

- Use netcat to get a web page from [iup.eng.ku.ac.th](http://iup.eng.ku.ac.th)
- Find the actual location of their 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

# Insecurity

---

There seems to be a bug in regis.ku.ac.th that allows unauthenticated download of pages, if you know the URL.

The 01219245 (450) class student list is here:

[https://regis.ku.ac.th/grade/download\\_file/class\\_01219245\\_611.txt](https://regis.ku.ac.th/grade/download_file/class_01219245_611.txt)

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

a) download it. (use wget)

b) can you download other course lists?

*You have to guess the last 3 digits, but so what?*

*Computers are good at repetitive tasks.*

c) can you **upload** a new class list (use 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](http://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*?

# 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? /  
>
```

# 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:

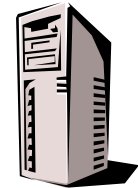
```
GET /path/image.jpeg HTTP/1.1
```

```
If-None-Match: "33101963682008"
```

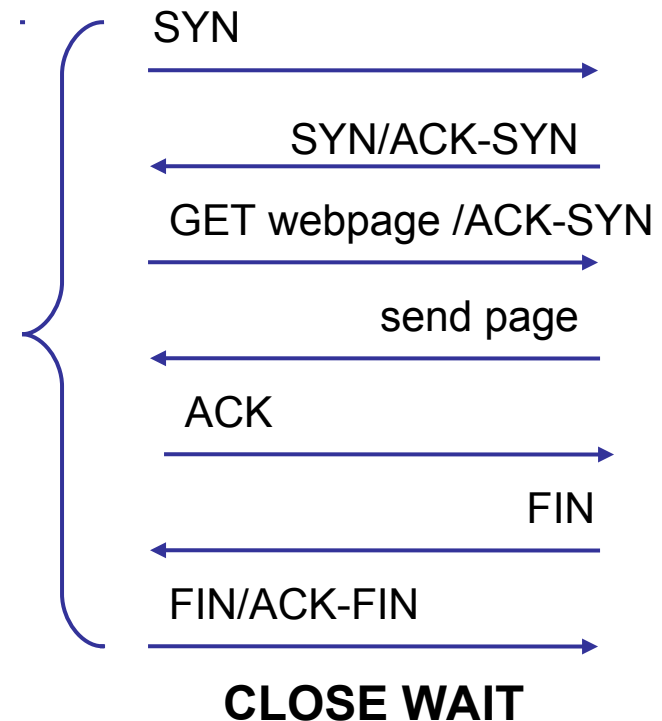
# Non-persistent Connection

- In HTTP 1.0, client must open a new connection for **each request**
- Lots of delay
- Lots of traffic and server overhead

Sequence repeated for  
every web request!



listen \*:80/TCP



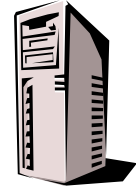
# Exercise: How many requests?

- 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">
```

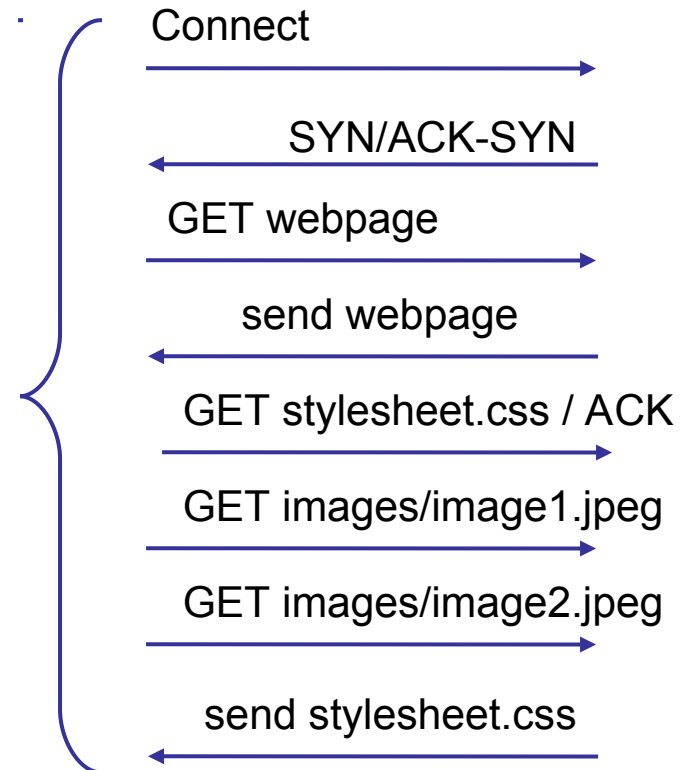
# Persistent Connection

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



listen \*:80/TCP

Multiple request/reply in  
one connection





# 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.