

HTTP

1

HTTP

- HyperText Transfer Protocol
- Created by Tim Berners-Lee at CERN
 - Defined 1989-1991
- Standardized and much expanded by the IETF
- Rides on top of TCP protocol
 - TCP provides: reliable, bi-directional, in-order byte stream
- Goal: transfer objects between systems
 - Do not confuse with other WWW concepts:
 - HTTP is not page layout language (that is HTML)
 - HTTP is not object naming scheme (that is URLs)
- Text-based protocol
 - Human readable

2

What's a protocol?

Human Protocols:

- "thank you ... you're welcome"
- "hello ... hi ... my name is ... pleased to meet you"
- Price haggling

... specific msgs sent
 ... specific actions taken when msgs received
 ... may be context or culture sensitive

Network protocols:

- drive device, rather than human, interaction
- all communication activity in Internet is governed by protocols

protocols define format & order of messages sent and received among network entities, and actions taken on message transmission, receipt

3

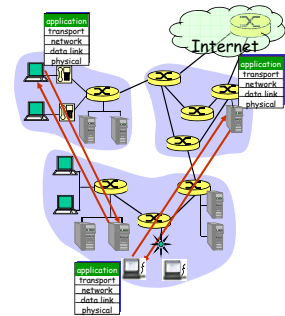
Applications and application-layer protocols

Application

- running in network hosts in "user space"
- exchange messages to implement app
- e.g., email, file transfer, the Web

Application-layer protocols

- one "piece" of an app
- define messages exchanged by apps and actions taken
- connectivity provided by lower layer protocols



4

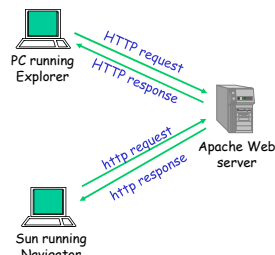
Client-Server Paradigm

Client:

- initiates contact with server ("speaks first")
- typically requests service from server,
- for Web, client is implemented in browser; for e-mail, in mail reader

Server:

- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail



HTTP in operation

Suppose user enters URL

www.toronto.edu/cs/index.html (containing text and references to 10 jpeg images)

- 1a. http client initiates TCP connection to http server (process) at www.toronto.edu. Port 80 is default for http server.
- 1b. http server at host www.toronto.edu waiting for TCP connection at port 80. "accepts" connection, notifying client
2. http client sends http request message (containing URL /cs/index.html) into TCP connection socket
3. http server receives request message, forms response message containing requested object (/cs/index.html), sends message into socket

time ↓

6

http in operation (cont.)

5. http client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

4. http server closes TCP connection.

6. Steps 1-5 repeated for each of 10 jpeg objects

7

HTTP request message: general format

```

HTTP method sp URL sp HTTP version cr lf request line
header field name : field value cr lf
...
header field name : field value cr lf
cr lf
Entity Body
  
```

8

HTTP request format

- Request line


```

HTTP method sp URL sp HTTP version cr lf
      
```

 - HTTP method
 - GET - return content of specified document
 - HEAD - return headers only of GET response
 - POST - execute specified doc with enclosed data
 - URL (only domain portion)
 - /host-identifier/path
 - e.g. /www.toronto.edu/headlines/
 - HTTP version
 - e.g. HTTP/1.0

9

HTTP request format

- Header fields


```

header field name : field value
      
```
- Examples:
 - Accept: text/html
 - Accept: image/jpg
 - Accept-language: en; en-gr; fr
 - If-modified-since: 17 May 2001
 - Content-Length: 2540

10

HTTP request example

```

GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
  
```

(extra carriage return, line feed)

request line (GET, POST, HEAD commands)

header lines

Carriage return, line feed indicates end of message

11

HTTP response message: general format

```

HTTP version sp status code sp status phrase cr lf response line
header field name : field value cr lf
...
header field name : field value cr lf
cr lf
Response Body
  
```

12

HTTP response format

Response line:

```
HTTP version sp status code sp status phrase cr lf
```

- o HTTP version
- o 3 digit response code
 - 1XX - informational
 - 2XX - success
 - 3XX - redirection
 - 4XX - client error
 - 5XX - server error
- o Brief text explanation of status code (e.g. OK)

13

Response status codes

A few commonly occurring sample codes:

200 OK

- o request succeeded, requested object later in this message

301 Moved Permanently

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

400 Bad Request

- o request message not understood by server

404 Not Found

- o requested document not found on this server

505 HTTP Version Not Supported

14

HTTP response format

Header fields

```
header field name : field value
```

Examples:

- o Content-Type: text/html
- o Content-Length: 4028
- o Language: en;
- o Last-modified: 17 May 2004

15

HTTP response example

```
HTTP/1.0 200 OK
Date: Thu, 25 Aug 2001 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Aug 2001 .....
Content-Length: 6821
Content-Type: text/html
```

```
data data data data data ...
data data data data data ...
data data data data data ...
```

status line:
(protocol
status code
status phrase)

header
lines

Carriage return,
line feed
indicates end
of message

data, e.g.,
requested
html file

16

HTTP 1.0 other features

POST

- o Client can send information to server
- o Forms, annotations

If-modified-since request header

- o Client tells server it has data and asks server whether it has fresher version or client is up to date

17

User-server interaction: conditional GET

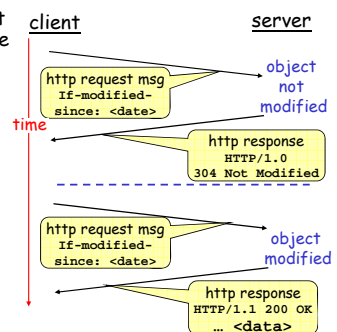
- o **Goal:** don't send object if client has up-to-date copy (cached)

- o client: specify date of cached copy in http request

```
If-modified-since: <date>
```

- o server: response contains no object if cached copy is up-to-date:

```
HTTP/1.0 304 Not Modified
```



18

HTTP is Stateless

- Server does not maintain status information across client requests
 - No way to correlate multiple request from same user

Protocols that maintain "state" are complex

- Past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

19

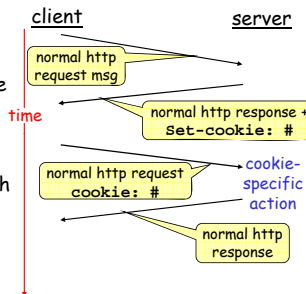
Cookies

- Store *cookie* on client side.
- Small amount of information (typically server-generated user id)
- Sent by client with each request
- Updated by server with response

20

User-server interaction: cookies

- server sends "cookie" to client in response msg
Set-cookie: 1678453
- client presents cookie in later requests
cookie: 1678453
- server matches presented-cookie with server-stored info
 - authentication
 - remembering user preferences, previous choices



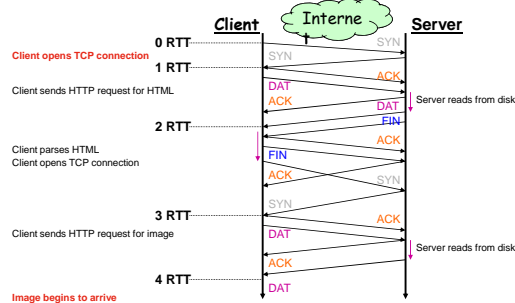
21

HTTP 1.0: Problems

- Each request opens new connection
 - Opening connection takes several packets (why?)
 - Starting up is slow (why?)

22

Web Page with Single Image

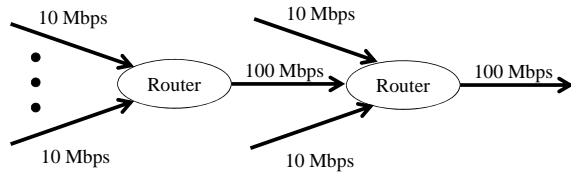


TCP

- HTTP rides on top of TCP transport service
- TCP provides: reliable, bi-directional, in-order byte stream
- Reliable
 - Prevent packet loss due to congestion
 - Overflow network queues
 - Send at rate at which network can forward packets
 - How to determine sending rate?
 - Dynamic
 - Depends on overall network condition

24

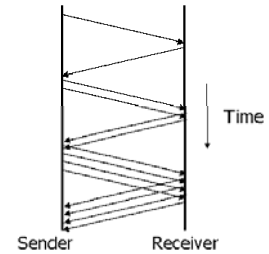
Network Congestion



25

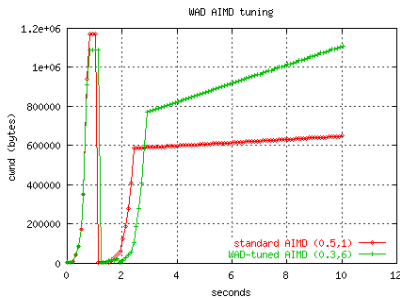
TCP Slow Start

- Determine sending rate by probing network
- Increase sending rate until a packet is dropped
- Double the number of unacknowledged packets (window size) for every new acknowledgement
- After a drop,
 - Reset window size to 1 packet
 - Cut maximum window size in half
 - Grow window with additive increase



26

TCP Slow Start



27

More Problems

- Short transfers are hard on TCP
 - Stuck in "slow start" phase of TCP connection
 - Loss recovery is poor when windows are small
- Lots of extra connections
 - Increases server state/processing

28

Netscape Solution

- Use multiple concurrent connections to improve response time
 - Different parts of Web page arrive independently
 - Can grab more of the network bandwidth than other users
- Doesn't necessarily improve response time
 - TCP loss recovery ends up being timeout dominated because windows are small

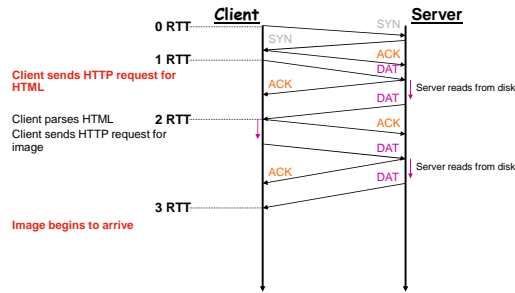
29

HTTP 1.1: Persistent Connections

- Keeps connection open for a time after server response so that multiple requests can ride on single connection -> reduced connection setup overhead.
 - GET index.html
 - Connection: keep-alive
 - ... multiple HTTP requests ...
 - Get banner.gif
 - Connection: close

30

Persistent Connections



Connection length

- When does the data end?
 - Without persistent connections, when connection closes.
 - With persistent connections, reply header includes content length.

32

Non-persistent and persistent connections

Non-persistent

- HTTP/1.0
- server parses request, responds, and closes TCP connection
- 2 RTTs to fetch each object
- Each object transfer suffers from slow start

But most 1.0 browsers use parallel TCP connections.

Persistent

- default for HTTP/1.1
- on same TCP connection: server, parses request, responds, parses new request,...
- Client sends requests for all referenced objects as soon as it receives base HTML.
- Fewer RTTs and less slow start.

33

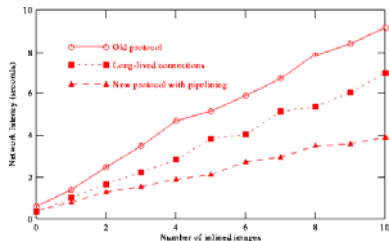
HTTP1.0 vs. HTTP1.1

- Venkata N. Padmanabhan and Jeffrey C. Mogul, "Improving HTTP Latency," in Proceedings of the The 2nd International WWW Conference, Chicago, IL, USA, Oct 1994
- Compared download latency for HTML documents with varying number of embedded images

34

HTTP1.0 vs. HTTP1.1

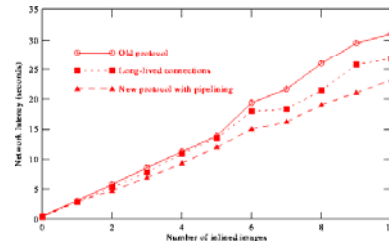
- Image size 2544 bytes



35

HTTP1.0 vs. HTTP1.1

- Image size 45566 bytes



36

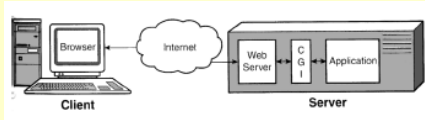
Persistent Connection Performance

- Benefits greatest for small objects.
- Serialized requests do not improve response time.
- Pipelining requests can result in large win.
- Server resource utilization reduced due to fewer connection establishments and fewer active connections.
- TCP behavior improved.
 - Longer connections help adaptation to available bandwidth.
 - Larger congestion window improves loss recovery.

37

Common Gateway Interface (CGI)

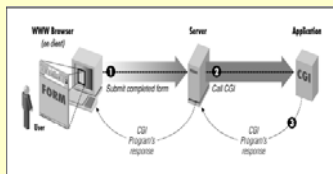
- Access to dynamic server-side content
 - Services
 - Data



CGI Model (Pieces)

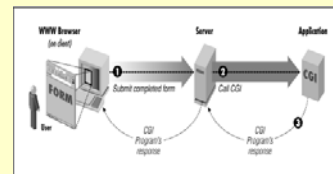
- Clients: web browser
- Server: web server mediates communication between browsers and handler programs
- Handler programs: any executable residing on the web server
- CGI Protocol: Specifies interaction between
 - Browser and handler programs
 - Function call syntax
 - Web server and handler programs

CGI Interaction



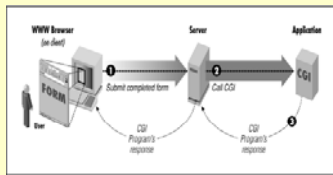
- Client sends request by specifying a URL+additional info.
- Web server receives the request.
- Web server identifies the request as a CGI request
- Web server locates the program to run

CGI Interaction



- Web server starts up the handling program (heavy weight process creation!!)
- Web server feeds request parameters to handler (through stdin or environment variables).

CGI Interaction



- Handler executes
- Output of the handler is sent via stdout back to the webserver for rerouting back to the requesting web browser.
- Output is typically a web page.
- Handler terminates.

Client-Handler Interaction

- Clients request the execution of a handler program by means of a:
 - A request method (e.g., GET, POST)
 - Universal Resource Identifier (URI)
 - Identifies handler
 - Extra arguments

URI Syntax

- Defined in section 2.1 of RFC 1808

<protocol>://<host><port>/<path-info><script>?"<query-string>

http://finance.yahoo.com/q?a=1&b=2

<protocol>	http
<host>	finance.yahoo.com
<port>	80
<path-info>	null
<script>	q
<query-string>	a=1, b=2

Query String Encoding

- RFC 2396
- Why encode?
 - Can think of a CGI script as a function, send arguments by specifying name/value pairs.
 - Way to pack and unpack multiple arguments into a single string

Encoding Rules

- **All arguments** will be concatenated into a single string of ampersand (&) separated name=value pairs, one pair for each form tag. Like this:
name_1=value_1&name_2=value_2&...
- **Spaces** in a name or value are replaced by a plus (+) sign. This is because url's cannot have spaces in them and under **METHOD=GET**, the form data is supplied in the query string in the url.
- **Other characters** (ie, =, &, +) are replaced by a percent sign (%) followed by the two-digit hexadecimal equivalent of the character in the ASCII character set.
 - Otherwise, it would be hard to distinguish these characters inside a variable from those between the variables in the first rule above.

Method

- GET
 - Arguments appear in the URL after the ?
 - Can be expressed as a URL
 - Limited amount of information can be passed this way
 - URL may have a length restriction on the server
 - Arguments appear on server log
- POST
 - Arguments are sent in the HTTP message body
 - Cannot be expressed as URL
 - Arbitrarily long form data can be communicated (some browsers may have limits (i.e. 7KB)).
 - Arguments usually does not appear in server logs.

Example

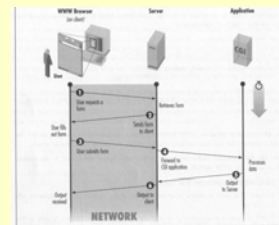
- In Web page

```
<a href="cgi-bin/hello.pl?a=1&n=2"> Click to run CGI </a>
```

- Request sent to web server

```
get /cgi-bin/hello.pl?a=1&b=2 http/1.0
```

Forms and CGI



- Specify request method in "method" attribute
- Automatically encodes all field values

Example

- In Web page

```
<form action="cgi-bin/login.pl" method="get">
  <input type="text" name="id" />
  <input type="text" name="password" />
</form>
```
 - Request sent to web server
 - Assuming user types 'myID' and 'secret' in fields
- get /cgi-bin/login.pl?id=myID&password=secret http/1.0

CGI Web Server-Handler Interaction

- Information about a request comes from
 - the request header
 - associated message-body (POST)
- Handler Input
 - Meta-variables
 - Stdin (message body)
- Handler Output
 - Stdout

Meta-variables

DOCUMENT_ROOT	root directory of your server
HTTP_COOKIE	visitor's cookie, if one is set
HTTP_HOST	hostname of the page
HTTP_REFERER	
HTTP_USER_AGENT	browser type of the visitor
HTTPS	"on" if called through a secure server
QUERY_STRING	query string
REMOTE_ADDR	IP address of the visitor
REMOTE_HOST	hostname of the visitor
REMOTE_USER	visitor's username
REQUEST_METHOD	GET or POST
REQUEST_URI	
SCRIPT_FILENAME	The full pathname of the current CGI
SERVER_SOFTWARE	server software (e.g. Apache 1.3)

Example – Print Environment Variables & Stdin

```
#!/local/bin/perl

print "Content-type: text/html\n\n";
print "<pre style='font: bolder 24pt'>\n";
print "Environment\n";
@keys = keys %ENV;          # variable names
@values = values %ENV;      # variable values
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}

print "STDIN\n";

while(<>){
    print;                  # request body
}
```

Example – Print Environment Variables & Stdin

```
<body>
<h3>GET</h3>
<form action="cgi-bin/stdin.pl" method="get">
  Student Number: <input type="text" name="studentNumber"> <br />
  Id: <input type="text" name="id"> <br />
  Password: <input type="password" name="password"> <br />
  Return Files: <input type="checkbox" name="returnFiles"> <br />
  <input type="submit" name="submit" />
</form>

<h3>POST</h3>
<form action="cgi-bin/stdin.pl" method="post">
  Student Number: <input type="text" name="studentNumber"> <br />
  Id: <input type="text" name="id"> <br />
  Password: <input type="password" name="password"> <br />
  Return Files: <input type="checkbox" name="returnFiles"> <br />
  <input type="submit" name="submit" />
</form>
</body>
```

Apache CGI

- Apache directories
 - cgi-bin .cgi files
 - htdocs .html, .gif, .jpeg, .css, .js
 - conf configuration files
 - httpd.conf
 - Main Apache configuration file
 - CGI are disabled by default
 - To enable CGI:
 - Add "ExecCGI" to the "Options" directive
- Options Indexes FollowSymLinks ExecCGI