Goals of Today’s Lecture

- Main ingredients of the Web
  - URIs, HTML, HTTP
- Key properties of HTTP
  - Request-response, stateless, and resource meta-data
- Performance of HTTP
  - Parallel connections, persistent connections, pipelining
- Web components
  - Clients, proxies, and servers
  - Caching vs. replication
The Web – History (I)

1945: Vannevar Bush, Memex:
"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"

(See http://www.iath.virginia.edu/elab/hfl0051.html)

The Web – History (II)

1967, Ted Nelson, Xanadu:
- A world-wide publishing network that would allow information to be stored not as separate files but as connected literature
- Owners of documents would be automatically paid via electronic means for the virtual copying of their documents
- Coined the term “Hypertext”
The Web – History (III)

- World Wide Web (WWW): a distributed database of “pages” linked through Hypertext Transport Protocol (HTTP)
  - First HTTP implementation - 1990
    - Tim Berners-Lee at CERN
  - HTTP/0.9 – 1991
    - Simple GET command for the Web
  - HTTP/1.0 –1992
    - Client/Server information, simple caching
  - HTTP/1.1 - 1996

Web Components

- Content
  - Objects
- Clients
  - Send requests / Receive responses
- Servers
  - Receive requests / Send responses
  - Store or generate the responses
- Proxies
  - Placed between clients and servers
    - Act as a server for the client, and a client to the server
    - Provide extra functions
      - Caching, anonymization, logging, transcoding, filtering access
      - Explicit or transparent (“interception”)
HTML

- A Web page has several components
  - Base HTML file
  - Referenced objects (e.g., images)
- HyperText Markup Language (HTML)
  - Representation of hypertext documents in ASCII format
  - Web browsers interpret HTML when rendering a page
  - Several functions:
    - Format text, reference images, embed hyperlinks (HREF)
- Straight-forward to learn
  - Syntax easy to understand
  - Authoring programs can auto-generate HTML
  - Source almost always available

URI

Uniform Resource Identifier (URI)

- Uniform Resource Locator (URL)
  - Provides a means to get the resource
    - scheme | authority | path | query | fragment
    - /foo://example.com:80%2f/over/there?name=ferret#nose
  - urn:example:animal:ferret:nose
- Uniform Resource Name (URN)
  - Names a resource independent of how to get it
  - urn:ietf:rfc:3986 is a standard URN for RFC 3986
**URL Syntax**

Content: How?

**protocol** : //**hostname**[ :**port**]/**directorypath**/**resource**

<table>
<thead>
<tr>
<th>protocol</th>
<th>http, ftp, https, smtp, rtsp, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hostname</strong></td>
<td>Fully Qualified Domain Name (FQDN), IP address</td>
</tr>
<tr>
<td><strong>port</strong></td>
<td>Defaults to protocol’s standard port</td>
</tr>
<tr>
<td>e.g.</td>
<td>http: 80/tcp  https: 443/tcp</td>
</tr>
<tr>
<td><strong>directory path</strong></td>
<td>Hierarchical, often reflecting file system</td>
</tr>
<tr>
<td><strong>resource</strong></td>
<td>Identifies the desired resource</td>
</tr>
<tr>
<td>Can also extend to program executions:</td>
<td></td>
</tr>
<tr>
<td><a href="http://us.f413.mail.yahoo.com/ym/ShowLetter?box=440840Bulk&amp;MsgId=2604.1744106_29699.1123.1261.0.269">http://us.f413.mail.yahoo.com/ym/ShowLetter?box=440840Bulk&amp;MsgId=2604.1744106_29699.1123.1261.0.269</a> 17.3552.1289957100&amp;Search=NS&amp;Nhead=3&amp;YY=31454&amp;order=down&amp;sort=date&amp;pos=0&amp;view=a&amp;head=b</td>
<td></td>
</tr>
</tbody>
</table>

---

**HTTP**

Client-Server: How?

- HyperText Transfer Protocol (HTTP)
  - Client-server protocol for transferring resources
- Important properties:
  - Request-response protocol
  - Resource _metadata_
  - _Stateless_
  - ASCII format

% telnet www.cs.berkeley.edu 80
GET /istoica/ HTTP/1.0
<blank line, *i.e.*, CRLF>
HTTP Big Picture

Client-to-Server Communication

- HTTP Request Message
  - Request line: method, resource, and protocol version
  - Request headers: provide information or modify request
  - Body: optional data (e.g., to “POST” data to the server)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
```
Client-to-Server Communication

- **HTTP Request Message**
  - Request line: *method*, resource, and protocol version
  - Request *headers*: provide information or modify request
  - Body: optional data (e.g., to “POST” data to the server)

  - Request *methods* include:
    - **GET**: Return current value of resource, run program, …
    - **HEAD**: Return the meta-data associated with a resource
    - **POST**: Update resource, provide input to a program, …

- *Headers* include:
  - Useful info for the server
    - *e.g.* desired language

Server-to-Client Communication

- **HTTP Response Message**
  - Status line: protocol version, status code, status phrase
  - Response *headers*: provide information
  - Body: optional data

  - Status line
    - **HTTP/1.1 200 OK**
    - *Connection close*
    - *Date: Thu, 06 Aug 2006 12:00:15 GMT*
    - *Server: Apache/1.3.0 (Unix)*
    - *Last-Modified: Mon, 22 Jun 2006* ...
    - *Content-Length: 6821*
    - *Content-Type: text/html*
    - (blank line)

  - Body
    - e.g., requested HTML file
Server-to-Client Communication

- HTTP Response Message
  - Status line: protocol version, **status code**, status phrase
  - Response headers: provide information
  - Body: optional data

- Response code classes
  - Similar to other ASCII app. protocols like SMTP

<table>
<thead>
<tr>
<th>Code</th>
<th>Class</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xx</td>
<td>Informational</td>
<td>100 Continue</td>
</tr>
<tr>
<td>2xx</td>
<td>Success</td>
<td>200 OK</td>
</tr>
<tr>
<td>3xx</td>
<td>Redirection</td>
<td>304 Not Modified</td>
</tr>
<tr>
<td>4xx</td>
<td>Client error</td>
<td>404 Not Found</td>
</tr>
<tr>
<td>5xx</td>
<td>Server error</td>
<td>503 Service Unavailable</td>
</tr>
</tbody>
</table>

Web Server: Generating a Response

- Return a file
  - URL matches a file (*e.g.*, /www/index.html)
  - Server returns file as the response
  - Server generates appropriate response header

- Generate response dynamically
  - URL triggers a program on the server
  - Server runs program and sends output to client

- Return meta-data with no body
HTTP Resource Meta-Data

- **Meta-data**
  - Info about a resource
  - A separate entity
- **Examples:**
  - Size of a resource
  - Last modification time
  - Type of the content
    - Data format classification
      - e.g., Content-Type: text/html
    - Enables browser to automatically launch an appropriate viewer
    - Borrowed from e-mail’s Multipurpose Internet Mail Extensions (MIME)
- **Usage example:** Conditional GET Request
  - Client requests object “If-modified-since”
  - If object hasn’t changed, server returns “HTTP/1.1 304 Not Modified”
  - No body in the server’s response, only a header

HTTP is Stateless

**Client-Server: How?**

- **Stateless** protocol
  - Each request-response exchange treated independently
  - Servers not required to retain state
- **This is good**
  - Improves scalability on the server-side
    - Don’t have to retain info across requests
    - Can handle higher rate of requests
    - Order of requests doesn’t matter
- **This is bad**
  - Some applications need persistent state
    - Need to uniquely identify user or store temporary info
    - e.g., Shopping cart, user preferences and profiles, usage tracking, …
State in a Stateless Protocol: Cookies

- Client-side state maintenance
  - Client stores small state on behalf of server
  - Client sends state in future requests to the server
- Can provide authentication

Putting All Together  Client Server: How?

- Client-Server
  - Request-Response
    - HTTP
    - Stateless
      - Get state with cookies
  - Content
    - URI/URL
    - HTML
    - Meta-data
Web Browser
Client-Server: In Practice

- Is the client
- Generates HTTP requests
  - User types URL, clicks a hyperlink or bookmark, clicks “reload” or “submit”
  - Automatically downloads embedded images
- Submits the requests (fetches content)
  - Via one or more HTTP connections
- Presents the response
  - Parses HTML and renders the Web page
  - Invokes helper applications (e.g., Acrobat, RealPlayer)
- Maintains cache
  - Stores recently-viewed objects and ensures freshness

Web Server
Client-Server: In Practice

- Handle client request:
  1. Accept a TCP connection
  2. Read and parse the HTTP request message
  3. Translate the URI to a resource
  4. Determine whether the request is authorized
  5. Generate and transmit the response
- Web site vs. Web server
  - **Web site**: one or more Web pages and objects united to provide the user an experience of a coherent collection
  - **Web server**: program that satisfies client requests for Web resources
5 Minute Break

Questions Before We Proceed?

HTTP Performance

Most Web pages have multiple objects ("items")
- e.g., HTML file and a bunch of embedded images

How do you retrieve those objects?
⇒ One item at a time

What transport behavior does this remind you of?
HTTP Performance

Most Web pages have multiple objects ("items")
- *e.g.*, HTML file and a bunch of embedded images

How do you retrieve those objects?
⇒ *One item at a time*
  - What are the performance implications?
  - Where does the analogy possibly break down?
  - What else can we do?
Improving HTTP Performance: Concurrent Requests & Responses

- Use multiple connections in parallel
- Does not necessarily maintain order of responses
  - Client = ☺ Why?
  - Server = ☺ Why?
  - Network = 😞 Why?
- Is this fair?
  - $N$ parallel connections use bandwidth $N$ times more aggressively than just one
  - What’s a reasonable/fair limit as traffic competes with that of other users?

Improving HTTP Performance: Pipelined Requests & Responses

- Batch requests and responses
  - Reduce connection overhead
  - Multiple requests sent in a single batch
  - Small items (common) can also share segments
  - Maintains order of responses
  - Item 1 always arrives before item 2
- How is this different from concurrent requests/responses?
Improving HTTP Performance:

Persistent Connections

- Enables multiple transfers per connection
  - Maintain TCP connection across multiple requests
  - Including transfers subsequent to current page
  - Client or server can tear down connection

- Performance advantages:
  - Avoid overhead of connection set-up and tear-down
  - Allow TCP to learn more accurate RTT estimate
  - Allow TCP congestion window to increase
    - *i.e.*, leverage previously discovered bandwidth
  - Default in HTTP/1.1
  - Can use this to *batch* requests on a *single* connection

Example:

5 objects, RTT=50ms

---

Improving HTTP Performance

- Many clients transfer *same information*
  - Generates redundant server and network load
  - Clients experience unnecessary latency
Improving HTTP Performance: Caching

How?
- Modifier to GET requests:
  - If-modified-since — returns “not modified” if resource not modified since specified time
- Response header:
  - Expires — how long it’s safe to cache the resource
  - No-cache — ignore all caches; always get resource directly from server

Improving HTTP Performance: Caching on the Client

Example: Conditional GET Request
- Return resource only if it has changed at the server
  - Save server resources!

Request from client to server:

```
GET /~ee122/fa08/ HTTP/1.1
Host: inst.eecs.berkeley.edu
User-Agent: Mozilla/4.03
If-Modified-Since: Sun, 27 Aug 2006 22:25:50 GMT
<CRLF>
```

How?
- Client specifies “if-modified-since” time in request
- Server compares this against “last modified” time of desired resource
- Server returns “304 Not Modified” if resource has not changed
- .... or a “200 OK” with the latest version otherwise
Improving HTTP Performance:
Caching with Reverse Proxies

Cache documents close to **server**
→ decrease server load
- Typically done by content providers
- Only works for **static content**

![Diagram of reverse proxies](image)

---

Improving HTTP Performance:
Caching with Forward Proxies

Cache documents close to **clients**
→ reduce network traffic and decrease latency
- Typically done by ISPs or corporate LANs

![Diagram of forward proxies](image)
Improving HTTP Performance: Caching w/ Content Distribution Networks

- Integrate forward and reverse caching functionality
  - One overlay network (usually) administered by one entity
  - e.g., Akamai
- Provide document caching
  - Pull: Direct result of clients’ requests
  - Push: Expectation of high access rate
- Also do some processing
  - Handle dynamic web pages
  - Transcoding

Improving HTTP Performance: Caching with CDNs (cont.)
Example: Akamai

- Akamai creates new domain names for each client content provider.
  - e.g., a128.g.akamai.net
- The CDN’s DNS servers are authoritative for the new domains
- The client content provider modifies its content so that embedded URLs reference the new domains.
  - “Akamaize” content, e.g.: http://www.cnn.com/image-of-the-day.gif becomes http://a128.g.akamai.net/image-of-the-day.gif

Example: Akamaimmmmm

“Akamaized” response object has inline URLs for secondary content at a128.g.akamai.net and other Akamai-managed DNS names.
Improving HTTP Performance: Caching vs. Replication

- Why move content closer to users?
  - Reduce latency for the user
  - Reduce load on the network and the server

- How?
  - Caching
    - Replicate content "on demand" after a request
    - Store the response message locally for future use
    - Challenges:
      - May need to verify if the response has changed
      - ... and some responses are not cacheable
  - Replication
    - Planned replication of content in multiple locations
    - Update of resources handled outside of HTTP
    - Can replicate scripts that create dynamic responses

Conclusions

- Key ideas underlying the Web
  - Uniform Resource Identifier (URI), Locator (URL)
  - HyperText Markup Language (HTML)
  - HyperText Transfer Protocol (HTTP)
  - Browser helper applications based on content type

- Performance implications
  - Concurrent connections, pipelining, persistent conns.

- Main Web components
  - Clients, servers, proxies, CDNs

- Next lecture: drilling down to the link layer
  - K & R 5.1-5.3, 5.4.1