HTTP

HyperText Transfer Protocol

The World-Wide Web

Transfer of resources is using HTTP
Browser-HTTPD Interaction

Browser

Web Server

Files

host
www.google.com

user requests
http://www.google.com

index.html

The Browser

- Gets an IP Address
- Establishes a TCP Connection
- Sends an HTTP Request
- Receives an HTTP Response
- Presents a Page

Web Server

How?

To which port?

Can it present the page now?
Universal Resource Location

protocol://host:port/path#anchor?parameters

http://www.cs.technion.ac.il/~cs236607/index.html

http://www.google.com/search?hl=en&q=blabla

• Are URLs good identifiers?
• Can they be used as keys of resources?
URL, URN and URI

- **URL** is Universal Resource Location
- **URN** is Universal Resource Name
  - Independent of a specific location, e.g.,
    - urn:ietf:rfc:3187
- **URI** is either a URN or a URL
  - There are many possible formats to URI’s
    - mailto:<account@site>
    - news:<newsgroup-name>
    - http://www.cs.technion.ac.il/~cs236607#key123456

Terminology

- **Web Server** is an implementation of an HTTP Daemon (either HTTP/1.0 or HTTP/1.1)
- **User Agent** (UA) is a client (e.g., browser)
- **Origin Server** is the server that has the resource that is requested by a client
- **Proxy** acts on behalf of a client
- **Reverse Proxy** acts on behalf of a server
Proxy Servers

- Sometimes, a browser sends its request via a proxy?
  - The goals:
    - Improve Web traffic
    - Add anonymity
  - How does the proxy affects HTTP message exchange?
    - How does it change messages?
    - Can the browser affect the behavior of the proxy?
    - Can the Web server affect the behavior of the proxy?

The proxy can serve the resource from its own cache, if it is there, without sending the request to the origin server.
Proxy Caches reduce latency for a given user agent if they can serve the request from their cache. As a result, they also save bandwidth and reduce the load on the origin server.

Therefore, they reduce latency also for requests that must be sent to the origin server.
Main Features of HTTP

- Stateless
- Persistent connection (in HTTP/1.1)
- Pipelining (in HTTP/1.1)
- Caching (improved in HTTP/1.1)
- Compression negotiation (improved in 1.1)
- Content negotiation (improved in 1.1)
- Interoperability of HTTP/1.0 and HTTP/1.1

Requests and Responses

- A UA sends a request and gets back a response
- Requests and responses have headers
- HTTP 1.0 defines 16 headers
  - None is required
- HTTP 1.1 defines 46 headers
  - The Host header is required in all requests
Hop-by-Hop vs. End-to-End

- HTTP requests and responses may travel between the UA and the origin server through a series of proxies
- Thus, in an HTTP connection there is a distinction between
  - Hop-by-Hop, and
  - End-to-End
- Some headers are hop-by-hop and some are end-to-end (in HTTP/1.1)

Each hop is a separate TCP connection

How is the Chain of Proxies Discovered?

- A browser sends requests to the proxy that is specified in the browser settings
- Alternatively, Web proxies can be automatically discovered, for example
  - the router redirects all HTTP requests to the proxy ("transparent caching")
- Each proxy knows the address of the next proxy along the way to the origin server
Interoperability

- Even if the UA and the origin server comply with HTTP/1.1, some proxies along the way may only comply with HTTP/1.0
- The design of HTTP/1.1 had to take it into account
- We will point out features of HTTP/1.1 that were introduced to ensure interoperability with HTTP/1.0

How can HTTP support both backward (to the past) and forward (to the future) interoperability?
Note

- HTTP (both 1.0 and 1.1) has always specified that an implementation should ignore a header that it does not understand
  - The header should not be deleted – just ignored!
- This rule allows extensions by means of new headers, without any changes in existing specifications
The Format of a Request

Method | URI | Version
---|---|---
**header** : value | **header** : value
*header lines*

Entity
(Message Body)

The URI is specified without the host name, unless the request is sent to a proxy.

An Example of a Request

**Method**

GET/index.html HTTP/1.1

**Request URI**

Accept: image/gif, image/jpeg
User-Agent: Mozilla/4.0
Host: www.cs.technion.ac.il:80
Connection: Keep-Alive

[blank line here]
Common Request Methods

- **GET** returns the content of a resource
- **HEAD** only returns the headers
- **POST** sends data to the given URI

- **OPTIONS** requests information about the communication options available for the given URI, such as supported content types
  - * instead of a URI requests information that applies to the given Web server in general

*OPTIONS is not fully specified*
Additional Request Methods

- **PUT** replaces the content of the given URI or generates a new resource at the given URI if none exists
- **DELETE** deletes the resource at the given URI
- **TRACE** invokes a remote loop-back of the request
  - The final recipient *should* reflect the message back to the client
- **CONNECT** switches the proxy to become a tunnel

Do servers really support PUT or DELETE?

---

Range and Conditional Requests (Usually GET)

- **Range requests** are requests with the **Range** header (only in HTTP/1.1)
- **Conditional requests** are related to caching and they use the following headers (some only in HTTP/1.1)

  - If-Unmodified-Since
  - If-Match
  - If-None-Match
  - If-Modified-Since
  - If-Range
Where Do Request Headers Come From?

- The **UA** sends headers with each request
  - The **user** may determine some of these headers through the browser configuration
- **Proxies** along the way may add their own headers and delete existing (hop-by-hop) headers

The Host Header in Requests
(It is Required in HTTP/1.1 but not in HTTP/1.0)
In HTTP/1.0

- If the URL is http://www.example.com/home.html, then the HTTP/1.0 syntax is GET /home.html HTTP/1.0

and the TCP connection is to port 80 at the IP address corresponding to www.example.com

Why is the Host Header Required in HTTP/1.1?

Why is the Host Header Required in HTTP/1.1?

- In HTTP/1.0, there can be at most one HTTP server per IP address
  - This wastes IP addresses, since companies like to use many “vanity URLs” (that is, URLs that only consist of hostnames)
- In HTTP/1.1, requests to different HTTP servers can be sent to port 80 at the same IP address, since each request contains the host name in the Host header

Why is the Hostname not in the URL?
Why is the Hostname not in the URL?

- To ensure interoperability with HTTP/1.0
  - An HTTP/1.0 server will incorrectly process a request that has an absolute URL (i.e., a URL that includes the hostname)
  - An HTTP/1.1 must reject any HTTP/1.1 (but not HTTP/1.0) request that does not have the Host header
The Format of a Response

```
version | status code | phrase
--------|-------------|--------
header   | : | value
        | cr | lf
```

Entity
(Message Body)

An Example of a Response

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
  <body>
    <h1>Hello World</h1>
    (more file contents) . . .
  </body>
</html>
```
Status Codes in Responses

- The status code is a three-digit integer, and the first digit identifies the general category of response:
  - 1xx indicates an informational message
  - 2xx indicates success of some kind
  - 3xx redirects the client to another URL
  - 4xx indicates an error on the client's part
    - Yes, the system blames it on the client if a resource is not found (i.e., 404)
  - 5xx indicates an error on the server's part
Where Do Response Headers Come From?

- The **Web server**, based on its settings, determines some headers
- **Applications** that create dynamic pages may add additional headers
- **Proxies** along the way may add their own headers and delete existing (hop-by-hop) headers

Where Do Status Codes Come From?

- Web servers and applications creating dynamic pages determine status codes
- It is important to configure Web servers and write applications creating dynamic pages so that
  - **they will return correct, meaningful and useful status codes and headers**
Apache HTTP Server

- Apache lets each user put an `.htaccess` file in her www directory
  - The `.htaccess` file applies to all subdirectories as well, unless it is overridden by `.htaccess` files in those subdirectories
  - The `.htaccess` file may contain commands that add headers to responses (as well as commands that do other things)

Tomcat

- Tomcat is a simple web server that we will use in this course
- In Tomcat, configuration of HTTP response headers is in the `server.xml` file
Setting HTTP Headers for Dynamically Generated Content

- Headers can be set by using appropriate methods, e.g.,
  - `myServlet.setContentType(...)`
  - `myServlet.setContentLength(...)`

META HTTP-EQUIV Tags

- The browser interprets these tags as if they were headers in the HTTP response
- For example
  `<META HTTP-EQUIV="Refresh"
    CONTENT="5; URL=http://host/path/">`
- If the value is 0 (instead of 5) and there is no URL parameter, the same page is continuously refreshed, causing the Back button to stop working
META HTTP-EQUIV Tags are Only Read by Browsers

- META HTTP-EQUIV tags are interpreted by browsers
- Proxies usually don’t read the HTML documents – they only read the headers of the HTTP requests and responses
- Therefore, cache-control headers in META HTTP-EQUIV tags actually apply only to the browser’s cache

Examples
3/20/2013

[kanza@csa ~]$ telnet www.cs.technion.ac.il 80
Trying 132.68.32.15...
Connected to csn.cs.technion.ac.il (132.68.32.15).
Escape character is '^]'.
GET /~kanza/test.html HTTP/1.0

HTTP/1.1 200 OK
Date: Wed, 16 Jan 2008 00:10:20 GMT
Server: Apache/2.0.54 (Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g PHP/5.0.4 DAV/2 mod_perl/1.999.21 Perl/v5.8.6
Last-Modified: Wed, 16 Jan 2008 00:07:33 GMT
ETag: "9a42e-79-53ebbc40"
Accept-Ranges: bytes
Content-Length: 121
Connection: close
Content-Type: text/html

<html>
<head>
<title>Test for cs236607</title>
</head>
<body>
This page is being used for testing HTTP.
</body>
</html>

Connection closed by foreign host.

[kanza@csa ~]$ telnet www.cs.technion.ac.il 80
Trying 132.68.32.15...
Connected to csn.cs.technion.ac.il (132.68.32.15).
Escape character is '^]'.
GET /~kanza/test.html HTTP/1.1
Host: www.cs.technion.ac.il

HTTP/1.1 200 OK
Date: Wed, 16 Jan 2008 00:28:48 GMT
Server: Apache/2.0.54 (Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g PHP/5.0.4 DAV/2 mod_perl/1.999.21 Perl/v5.8.6
Last-Modified: Wed, 16 Jan 2008 00:07:33 GMT
ETag: "9a42e-79-53ebbc40"
Accept-Ranges: bytes
Content-Length: 121
Content-Type: text/html

<html>
<head>
<title>Test for cs236607</title>
</head>
<body>
This page is being used for testing HTTP.
</body>
</html>

Connection closed by foreign host.

[kanza@csa ~]$
[kanza@csa ~]$ telnet www.cs.technion.ac.il 80
Trying 132.68.32.15...
Connected to cs.technion.ac.il (132.68.32.15).
Escape character is '^]'.
GET /~kanza/test.html HTTP/1.1
HTTP/1.1 400 Bad Request
Date: Wed, 16 Jan 2008 00:31:20 GMT
Server: Apache/2.0.54 (Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g PHP/5.0.4 DAV/2
mod_perl/1.999.21 Perl/v5.8.6
Content-Length: 387
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.</p>
<hr>
<address>Apache/2.0.54 (Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g PHP/5.0.4 DAV/2
mod_perl/1.999.21 Perl/v5.8.6 Server at www.cs.technion.ac.il Port 80</address>
</body></html>
Connection closed by foreign host.
[kanza@csa ~]$

Persistent Connections and Pipelining

HTTP/1.1 Supports Both
What we see on the browser can be a combination of several resources

What is wrong with a naïve retrieval of the resources?

How can we improve the efficiency of presenting a page?

The faculty's homepage requires seven HTTP requests

HttpWatch
The Problem

- Typically, each resource consists of several files, rather than just one
  - Each file requires a separate HTTP request
- HTTP/1.0 requires opening a new TCP connection for each request
- TCP has a slow start and therefore, opening a series of new connections is inefficient

Persistent Connections are the Default in HTTP/1.1

- In HTTP/1.1, several requests can be sent on the same TCP connection
  - The slow-start overhead is incurred only once per resource
- A connection is closed if it remains idle for a certain amount of time
- Alternatively, the server may decide to close it after sending the response
  - If so, the response should include the header Connection: close
Pipelining

- When the connection is persistent, the next request can be sent before receiving the response to the previous request
- Actually, a client can send many requests before receiving the first response
- Performance can be greatly improved
  - No need to wait for network round-trips

Best-Possible Use of TCP

- A Client sends requests in some given order
- TCP guarantees that the requests are received in the order that they were sent
- The server sends responses in the order that it received the corresponding requests
- TCP guarantees that responses are received in the order that they were sent
- Thus, the client knows how to associate the responses with its requests
But a TCP Connection is Just a Byte Stream

- So, how does the client know where one response ends and another begins?
  - Parsing is inefficient and anyhow will not work (why?)
- The server must add the **Content-Length** header to the response
  - or else it must close the connection after sending the response

Will it work for dynamic pages?

Sending Dynamic Pages

- A server has to **buffer a whole dynamic page** to know its length (and only then the server can send the page)
  - The latency is increased
- Alternatively, the server can **break an entity into chunks** of arbitrary length and send these chunks in a series of responses
  - Only one chunk at-a-time has to be buffered
Chunked Transfer Encoding

- Each chunk is sent in a separate message that includes the header
  \textit{Transfer-Encoding: Chunked}
  and also includes the length of the chunk in the \textit{Content-Length} header
- A zero-length chunk marks the end of the message

Trailers

- If an entity is sent in chunks, some header values can be computed only after the whole entity has been sent
- The first chunk includes a \textit{Trailer} header that lists all the headers that are deferred until the trailer
- A server cannot send a trailer unless the information is purely optional, or the client has sent the header \textit{TE: trailers}
The Content-Length Header in Requests

- The **Content-Length** header is also applicable to POST and PUT requests.

More on the Connection Header

- The **Connection** header may contain *connection tokens*, e.g., **close** (discussed earlier).
- This header also lists all the hop-by-hop headers, thereby telling the recipient that all these headers must be removed before forwarding the message.
Interoperability Rule in HTTP/1.1

- If a **Connection** header is received in an HTTP/1.0 message, it means that it was incorrectly forwarded by an HTTP/1.0 proxy
  - Therefore, all the headers it lists were incorrectly forwarded and must be ignored

Caching in HTTP
Type of Web Caches

- **Browser Caches**
  - A portion of the hard disk is used to store representations of resources that have already been displayed
  - If a resource is requested again (for example, by hitting the “back” button), the request is served from the browser cache

- **Proxy Caches**
  - These are *shared caches* – they serve many users

---

**Proxy Caches**

```
GET /fruit/apple.gif
```

```
GET /fruit/apple.gif
```

```
GET /fruit/apple.gif
```
Benefit of Caching

24%-32% hit rate is possible, since many users share the cache and, therefore, there is a large number of shared hits.

Reasons for Using Web Caches

- Web caches reduce latency
  - Since the cache is closer to the client, it takes less time for the client to get the resource and display it
- Web caches save bandwidth
  - Since a resource has to be brought from the server just once, clients that need this resource consume less bandwidth
More Reasons for Using Web Caches

- Web caches reduce the load on servers (for the same reason that they save bandwidth)
- Since bandwidth is saved and server load is reduced, the latency is reduced for everyone
- Web caches give some measure of redundancy

For example, how much traffic is saved if the Google icon is not sent back with each search result?
Points to Consider When Designing a Web Site

- Caches can help the Web site to load faster
- Caches may “hide” the users of the Web site, making it difficult to see who is using the site
- Caches may serve content that is out of date, or *stale*

Terminology

- **Representations** are copies of resources that are stored in caches
  - actually, caches store complete responses, including headers
  - If a request is served from a cache, then it should be *semantically transparent* that is, it should be the same as a request that is served from the origin server
  - A representation is *fresh* if it is identical to the resource that is available at the origin server
  - If it is not identical, then it is *stale*
The Risk in Caching and How to Avoid It

- Responses might not be semantically transparent
- The cache should determine that the representation is fresh before sending it to the client
- If it is not fresh, the cache should forward the request to the origin server or to another cache

Caching Improves Latency and Saves Bandwidth in Two Ways

- In some cases, caching eliminates the need to send \textit{requests} to the origin server by using an \textit{expiration mechanism}
- In other cases, caching eliminates the need to return \textit{full responses} from the origin server by using a \textit{validation mechanism}
An Example of Using a Validation Mechanism

• Client: `GET /fruit/apple.gif`
• Server responds with `Last-Modified-Date: ...`
• Client caches object and last-modified-date
• Client sends `GET /fruit/apple.gif ...`
  ```
  If-Modified-Since: ...
  ```
• Server returns either
  304 Not Modified
  or resource

Validating an Object

• If the object is stale (i.e., not fresh), the cache will ask the origin server to validate the object
• In response, the origin server will either
  • tell the cache that the object has not changed, or
  • send a new copy of the object to the cache
Validation Mechanisms

- If-modified-since last-modified date
  - Cannot be used with dynamic pages
- ETags can be used for dynamic pages and also when a site cycles through several possible responses

Are there Limitations on what to Store in Cache?

- Should a proxy store in the cache all the responses it ever received?
The Following Resources are not Cached

- The headers of a response tell the cache not to keep the resource
- The response has no validator (i.e., an Expires value, a Max-Age value, a Last-Modified value or an ETag)
- The resource is authenticated or secured
- Furthermore, it is difficult to cache dynamic pages and pages with cookies

Fresh Objects Are Served From the Cache

- An object is fresh in the following cases:
  - The object has an expiry time or other age-controlling directive, and is still within the fresh period
  - The browser cache has already seen the object, and has been set to check for newer versions once a session
  - A proxy cache has received the object recently, and the object was modified relatively long ago (this is a heuristic – see later)
The Expires HTTP Header

- A response may include an **Expires** header:
  
  **Expires: Fri, 31 Oct 2008 14:19:41 GMT**

- If an expiry time is not specified, the cache can heuristically estimate the expiry time

Expiration Model

**Section 13.2 of RFC 2616**

- The Expires header cannot be used correctly if there is a clock skew and the resource is fresh for only a short time

- The header **Cache-Control: Max-Age** is used to calculate the freshness lifetime:
  
  \[
  \text{freshness\_lifetime} = \text{max\_age\_value}
  \]

- If there is no max-age directive, then
  
  \[
  \text{freshness\_lifetime} = \text{expires\_value} - \text{date\_value}
  \]

- All the information comes from the origin server; hence, not vulnerable to clock skew
Age Calculations (Sec. 13.2.3)

- When a proxy sends a response that is obtained from its cache, it must calculate (an upper bound on) the age and include it in the *Age* response header
  - The calculation uses values specified in the headers of the cached message and the proxy's own clock
  - The calculation adds the *resident time* + an upper bound on the *transmission time* to the an upper bound on the *received age*
  - Is it always a reliable (correct) calculation?
  - What happens if some proxy along the way runs HTTP/1.0?

The freshness lifetime (from the previous slide) is compared with the age to determine if the response is still fresh (and, hence, can be sent)
A Possible Heuristic

- If the cache received the object 10 hours after it was last modified, then it can heuristically determine that the expiry time is 1 hour after it has received it.
- In general, add 10% (or some other value) of the interval between the last-modification time (given by the `Last-Modified` header) and the time it was received.

The Cache-Control Header (Introduced in HTTP 1.1)

- The following are possible values for the `Cache-Control` header in responses:
  - `max-age=<seconds>`
    - Specifies the maximum amount of time that an object will be considered fresh (similar to, but overrides the Expires header).
  - `s-maxage=<seconds>`
    - Similar to max-age, except that it only applies to proxy (shared) caches.
More Possible Values for the Cache-Control Header

- **public**
  - Document is cacheable even if normal rules say that it shouldn’t be (e.g., authenticated document)

- **private**
  - The document is for a single user and can only be stored in private (non-shared) caches

- **no-store** (may also appear in requests)
  - The response should never be cached and should not even be stored in a temporary location on a disk (this value is intended to prevent inadvertent copies of sensitive information)

More Possible Values for the Cache-Control Header

- **must-revalidate**
  - Tell caches that they must obey any freshness information provided with the object (HTTP allows caches to take liberties with the freshness of objects)

- **proxy-revalidate**
  - Similar to must-revalidate, except that it only applies to proxy (shared) caches
No-Cache

- Some values of the Cache-Control header are meaningful in either responses or requests
- **no-cache**
  - In a response, it means not to use the response again without revalidation (this value can apply to cache directive headers; see Sec. 14.9 of RFC2616)
  - In a request, it means to bring a copy from the origin server (i.e., not to use a cache)

More Possible Values for the Cache-Control Header in Requests

- **max-age=<seconds>**
  - The response should not be older than the given value
- **max-stale=<seconds>**
  - The response could exceed its expiration time by the specified amount
- **min-fresh=<seconds>**
  - The response should remain fresh for at least the specified amount of time
- See Sec. 14.9 of RFC2616 for more details
The Pragma Header

- In a request, the header `Pragma: no-cache` is the same as `Cache-Control: no-cache`
- Don’t use `Pragma` – its meaning is specified only for requests and it is used just for compatibility with HTTP/1.0
- For interoperability, it is safer to set both the `Pragma` and the `Cache-Control` response headers to the value `no-cache`

The Reload (Refresh) Button

- Hitting the reload button in the browser brings a copy from a shared cache, but not necessarily from the origin server
  - There is no 100% guarantee that this is a fresh copy
- Hitting `Shift+Reload` brings a 100%-guaranteed fresh copy (i.e., from the origin server)
How Can a Client Force a Fresh Copy?

- A fresh copy is obtained from the origin server if the request includes the following header
  - Cache-Control: no-cache
- The proxy must revalidate its copy with the origin server if the following header is included in the request
  - Cache-Control: max-age=0

Who Adds Cache-Control Headers?

- The server
  - The configuration of the server determines which cache-control headers are added to responses
  - The author of the page can add headers by means of the .htaccess file (only in the Apache server)
- The application that generates dynamic pages, e.g., servlets, ASP, PHP
Cache-Control in HTTP-EQUIV

- The author of the page can add, to the document itself, a `cache-control` header by means of the META HTTP-EQUIV tag
  ```html
  <meta http-equiv="cache-control" content="no cache">
  ```
- But usually only the browser interprets this tag
- Proxies along the way don’t read it, since they don’t read the document

Validators

- A validator is any mechanism that may help in determining whether a copy is fresh or stale
  - A strong validator is, for example, a counter that is incremented whenever the resource is changed
  - A weak validator is, for example, a counter that is incremented only when a significant change is made

  For example, a weak validator may not change if the only change in the site is the number of visitors …
Last-Modified Header

- The most common validator is the time when the document was last changed, the last-modified time
  - It is given by the Last-Modified header
  - In principle, this header should be included in every response; however, there is no last-modified time for dynamic pages
  - It is a weak validator if an object can change more than once within a one-second interval

ETag (Entity Tag)

- ETag is a strong validator (i.e., a unique identifier) generated by the server
  - It is part of the HTTP/1.1 specification (not available in HTTP/1.0)
  - The specification does not say how to generate it
- The preferred behavior for an HTTP/1.1 origin server is to send both an ETag header and a Last-Modified header
Conditional Requests

- The conditional headers are
  - `If-Modified-Since`
  - `If-Unmodified-Since`
  - `If-Match`
  - `If-None-Match`
  - `If-Range`

- These headers are used to validate an object (i.e., check with the origin server whether the object has changed)

If-Modified-Since Header

- The `If-Modified-Since` header is used with a GET request
- If the requested resource has been modified since the given date, the server returns the resource as it normally would (i.e., the header is ignored)
- Otherwise, the server returns a 
  **304 Not Modified** response, including the `Date` header, but with no message body

```
HTTP/1.1 304 Not Modified
Date: Fri, 31 Dec 1999 23:59:59 GMT
```
If-None-Match Header

- A cache may store several responses for the same URI, each having a different ETag
  - A server may cycle through a set of possible responses
  - The cache sends a request with a list of ETags in the header **If-none-match**
  - If no ETag on the list matches the resource’s current ETag, the server returns a normal response
  - Otherwise, the server returns a response with **304 (Not Modified)** and an **ETag** header that indicates which cache entry is currently valid

If-Unmodified-Since Header

- The **If-Unmodified-Since** header can be used with any method
  - If the resource has *not* been modified since the given date, the server returns the same response as it normally would
  - Otherwise, the server returns a **412 Precondition Failed** response
More on Conditional Requests

- The following conditional headers are useful in requests that are more complex than just a simple GET request; for example, in range requests
  - If-Unmodified-Since
  - If-Match
  - If-Range

The Vary Header

- A response may depend on some header fields of the request
  - For example, the `Accept-Language` and the `Accept-Charset` headers determine the specific response
  - The `Vary` header in a response lists all the relevant `selecting` header fields of the request
Finding Relevant Cache Entries

- A cache stores responses using the URI as a key
- A cache can return a stored response if
  - The URI of the new request matches the URI of stored response
  - The selecting headers of the new request match the selecting header fields in the \texttt{vary} header of the stored response

No Transform

- Sometimes proxies transform responses (for example, to reduce image size before transmitting over a slow link)
- Some responses cannot be blindly transformed without losing information
- The \texttt{no-transform} directive in the \texttt{Cache-Control} header is used to prevent transformations (it applies to both requests and responses)
Restrict Access

- Some applications should restrict access to authorized users only
  - IP-address-based
    - Access is permitted only to certain IP addresses
  - Form-based
    - The first page shown to the user is a form that requests for a password
  - HTTP Basic

Does it also allow the user application authenticate the server?
HTTP Basic

- The user tries to access the page
- The server response is
  - HTTP/1.1 401 Unauthorized
  - WWW-Authenticate: Basic realm="Description of the restricted site"
- The browser pops up a prompt window asking for a user name and password
- The user input is encoded and sent to the server
  - Authorization: Basic emFjaGFyawFzOMFwcGxcGlCg==
- If authorization succeeds, resources are sent to the browser

Sessions
HTTP is Stateless

- Theoretically, each request-response is an independent interaction
- How can we implement an online store
  - Payment and shipment are according to the state of some virtual shopping cart
- Does persistent connection provide a solution?

Sessions

- A session is a sequence of related interactions between a client and a server
- A session allows responses to be according to a state
  - A shared state can be shared by several users
  - A session state is a state of a single user
  - A transient state refers to a single interaction
Implementing Sessions

- URL Rewriting
- Hidden Form Fields
- Cookies

Advanced Topics
Bandwidth Optimization

- Range requests
- Expect and 100 (Continue)
- Compression

Range Requests

- A range request uses the Range header for specifying the requested portions of a resource
- A range response is returned with the Content-Range header that specifies the offset and length of the returned range
- The multipart/byteranges MIME type allows the transmission of multiple ranges in one response
When to Use Range Requests

- To read the initial part of an object
  - For example, if the object is an image, reading the initial part provides the information for doing the layout
- To complete a response transfer that was interrupted (either by the user or by network failure)
- To read the tail of a growing object

Range Requests and Caching

- A range response is returned with the status code 206 (Partial Content)
  - This prevents HTTP/1.0 proxies from accidentally treating the response as a full one, and using it later as a cached response
Conditional Range Requests

- To request conditionally the prefix of a resource, the If-None-Match header can be used
  - This happens when the client has a response containing the prefix in its cache, and the client wants to validate that response

The If-Range Header

- Sometimes the client’s cache may have the object, but without the requested range
  - Hence, the client sends a range request
- The server should return the requested range if the object has not changed
- Otherwise, the server should send back a full response
The Clients Wants the Range only if the Object has not Changed

- The client sends a range request with the If-Match header
  - The server returns the the range (i.e., normal) response if the object has not changed
  - Otherwise, the server returns 412 (Precondition Failed) and the client should send a new request for the full object
- Two requests might be needed
  - The If-Range header does the above interaction in one request

Expect and 100 (Continue)

- A request (e.g., POST) may contain a large object
- Sometimes there is no need to send the object to find out that the request fails
  - For example, if the client lacks authorization, or the server is too busy
- In HTTP/1.1, the client can send just the headers and wait for the server’s indications that it can also send the object
The Expect Header

- The client must include the new header Expect: 100 with the rest of the headers that it initially sends (why?)
  - The server should respond with the status code 100 (Continue), or with the usual status code if it cannot handle the request
  - HTTP/1.1 has some rules for avoiding infinite waits by clients or wasted bandwidth

Compression

- HTTP/1.1 makes a clear distinction between end-to-end encoding (the Content-Encoding response header) and hop-by-hop encodings (the Transfer-Encoding response header)
- A client uses the Accept-Encoding for specifying the content encodings that it can handle and the ones it prefers
- The client uses the TE header similarly for transfer encodings
Content Negotiations

- Server-driven content negotiation
  - The client sends its preferences using the headers Accept-Language, Accept-Charset, etc.
  - The server chooses the representation that best matches the client’s preferences
- The headers controlling content negotiations may include wildcards and *quality values (qvalues)* between 0.0 and 1.0

```
Accept-Language: en, fr;q=0.5, da;q=0.1
```

Agent-Driven Content Negotiation

- When the client requests a varying resource, the server replies with a 300 (Multiple Choices) response and it lists
  - The available representations and their properties (e.g., language, charset, etc.)
  - The Alternate header has been reserved for this purpose, but its specification has not been completed
- Hence, server-driven negotiation is the only usable form
The Vary Headerb

- Content negotiation and caching can interact in subtle ways
  - Hence, the \texttt{vary} header (that was mentioned earlier)

Warnings (New in HTTP/1.1)

- The Warning header has codes indicating some potential problems with the response, even if the status code is 200 (OK)
  - For example, when returning a stale response because it could not be validated
- Warnings are divided into two types based on the first digit (out of three) digit
  - Warning of one type should be deleted after a successful revalidation and those of the second type should be retained
  - Hence, this mechanism is extensible to future warning codes
New Status Codes in HTTP/1.1

- 24 new status codes in HTTP/1.1
  - 100 (Continue)
  - 206 (Partial Content)
  - 300 (Multiple Choices)
  - 409 (Conflict) is used when a request conflicts with the current state of the resource (e.g., a PUT request might violate a versioning policy)
  - 410 (Gone) is used when a resource has been removed permanently
    - It indicates that links to the resource should be deleted

Links

- Request for Comments 2616 (rfc2616)
- A caching tutorial at http://www.mnot.net/cache_docs/