Protocolo HTTP

SISTEMAS INFORMÁTICOS I

SLIDES 12

Web and HTTP

- Web page consists of objects
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of base HTML-file which includes several referenced objects
- Each object is addressable by a URL
- Example URL:

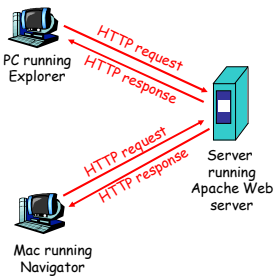
`http://www.dei.uc.pt/aulas_sd/pic.gif`

host name | path name

HTTP overview

HTTP protocol

- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068



HTTP overview

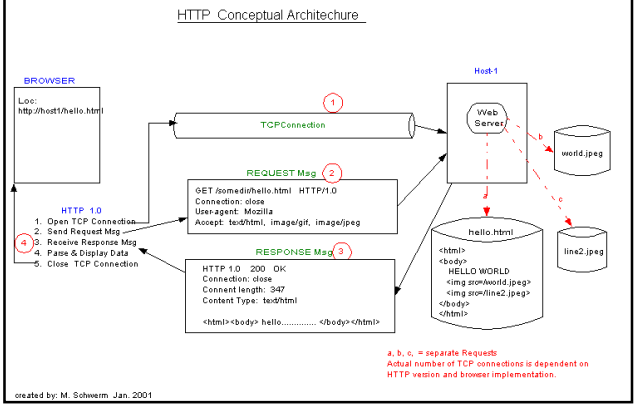
Uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- HTTP specifies the messages sent between the browser (HTTP client) and the Web server (HTTP server).
- Body field of message responses sent to browser are represented in HTML.

HTTP is "stateless"

- server maintains no information about past client requests

HTTP Protocol: Architecture



HTTP connections

Non-persistent HTTP

- At most one object is sent over a TCP connection.
- HTTP/1.0 uses non-persistent HTTP

Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.
- HTTP/1.1 uses persistent connections in default mode

Nonpersistent HTTP

Suppose user enters URL
`www.dei.uc.pt/sd/home.index`

(contains text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.dei.uc.pt on port 80

1b. HTTP server at host www.dei.uc.pt waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `sd/home.index`

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Nonpersistent HTTP (cont.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing **html** file and then displays html. Parsing the html file, it finds 10 referenced jpeg objects

6. Repeat steps 1-5 for each of 10 jpeg objects

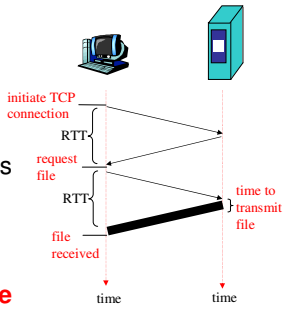
time

RTT for a HTML Page Request

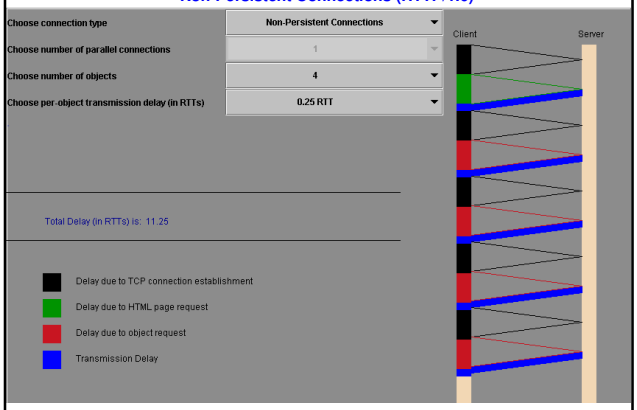
Response time:

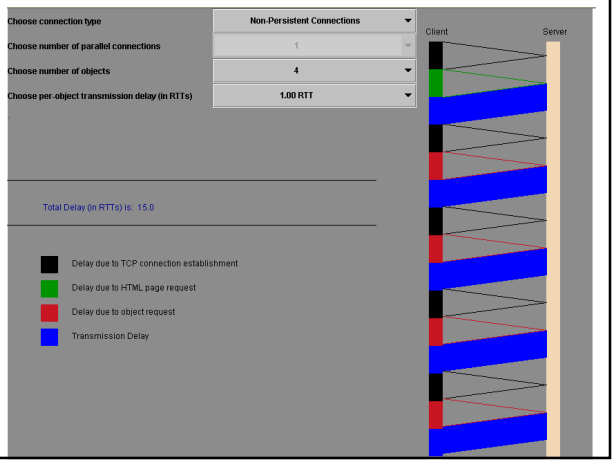
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time

total = 2RTT+transmit time

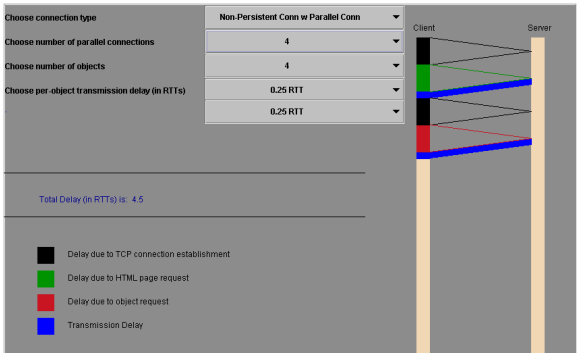


Non-Persistent Connections (HTTP/1.0)





Non-Persistent with Parallel Sessions



Persistent HTTP

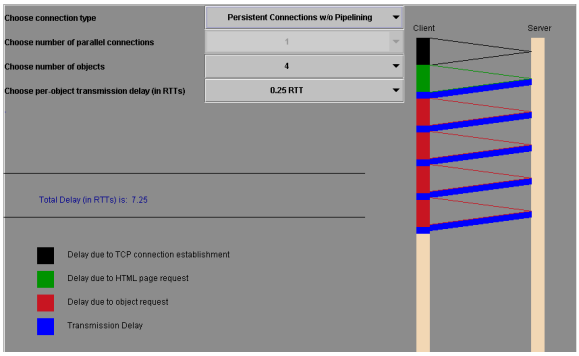
Non-persistent HTTP:

- requires 2 RTTs per object
- OS must allocate host resources for each TCP connection
- but browsers often open parallel TCP connections to fetch referenced objects.

Persistent HTTP

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server are sent over the same connection.
- HTTP server closes the connection with it is not used for a certain time.

Persistent HTTP (HTTP/1.1)



Overhead of HTTP/1.0

- 1 RTT overhead for each start (each request / response)

- if 10 objects:

TTT = [10 * 1 tcp/rtt] + [10 * 1 req/resp rtt] = 20 rtt

Note:
- this does not account for server processing
- RTT: Round-trip-time (client to server and back to client)
- TTT: Total transmission time

HTTP/1.1

- HTTP/1.1 [1998, rfc 2068]; persistent connections -- very helpful with multi object requests
  - by default, server keeps TCP connection open
  - only one slow start per server connection
  - if 10 objects

    TTT = [1 * 1 tcp/rtt] + [10 * 1 req/resp rtt] = 11 rtt

  - this implies sequential request / response
  - non- pipelining: next request not sent until response received for previous request

Persistent HTTP: Pipelining

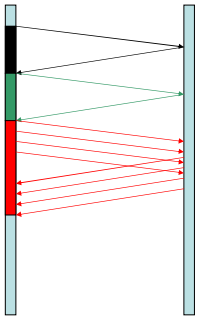
Persistent without pipelining:

- client issues new request only when previous response has been received
- one RTT for each referenced object.

Persistent with pipelining:

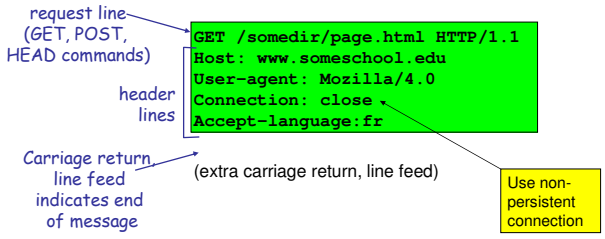
- default in HTTP/1.1
- client sends requests as soon as it encounters a referenced object.
- as little as one RTT for all the referenced objects.

Persistent with Pipelining



HTTP request message

- Two types of HTTP messages: *request*, *response*
- HTTP request message:



Method types

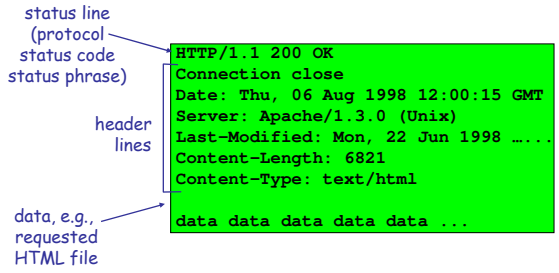
HTTP/1.0

- **GET**
- **POST**
- **HEAD**
  - asks server to leave requested object out of response

HTTP/1.1

- **GET, POST, HEAD**
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field
- TRACE: http "echo" for debugging (added in 1.1)
- CONNECT: used by proxies for tunneling (1.1)
- OPTIONS: request for server/proxy options (1.1)

HTTP response message



HTTP Request Message

```
GET /somedir/page.html HTTP/1.1
User-agent: Mozilla (compatible; MSIE 5.01, Windows NT)
Accept: text/html, image/gif, image/jpeg
Accept-language: en-us
/* a blank line */
```

HTTP Response Message

```
HTTP/1.1 200 OK
Date: Thu, 06 Aug 1999 12:00:36 GMT
Server: Apache/1.3.0 (Unix)br> Last -Modified: Mon,
22 Jun 1999 09:23:24 GMT
Content-Length: 6821
Content-Type: text/html

...
```

HTTP Response: Status Codes

- The status code is a three-digit integer, and the first digit identifies the general category of response:

- **1xx** indicates an informational message only
- **2xx** indicates success of some kind
- **3xx** redirects the client to another URL
- **4xx** indicates an error on the client's part
- **5xx** indicates an error on the server's part

HTTP response status codes

200 OK
- request succeeded, requested object later in this message

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

304 Not Modified

400 Bad Request
- request message not understood by server

403 Forbidden

404 Not Found
- requested document not found on this server

500 Internal Server Error

503 Service Unavailable

505 HTTP Version Not Supported

Try out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet eden.dei.uc.pt 80
```

Opens TCP connection to port 80 at eden.dei.uc.pt

2. Type in a GET HTTP request:

```
GET /~sdp/sumt.htm HTTP/1.0
```

By typing this in (hit carriage return twice), you send this minimal GET request to HTTP server

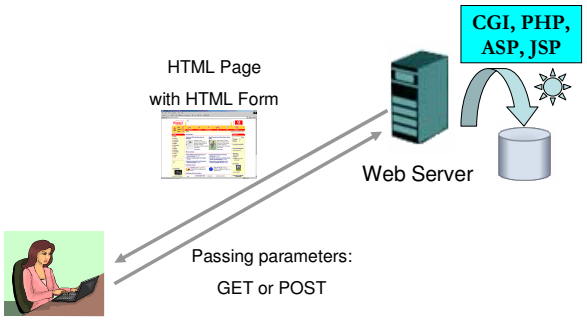
3. Look at response message sent by HTTP server!

HEAD Request

- A HEAD request is just like a GET request, except it asks the server to return the status line and response headers only, and not the actual resource (i.e. no message body).
- This is useful to check characteristics of a resource without actually downloading it, thus saving bandwidth. Use HEAD when you don't actually need a file's contents.

Server-Side Programs: GETs and POSTs

Server-side Programming



POST Request

- A POST request is used to send data to the server to be processed by a CGI script, a ASP, a JSP/Java Servlet or a PHP program.
- A POST request is different from a GET request in the following ways:

(1) There's a block of data sent with the request, in the message body.
(2) There are usually extra headers to describe this message body, like **Content-Type:** and **Content-Length:.**
(3) The *request URI* is not a resource to retrieve; it's usually a program to handle the data you're sending.
(4) The HTTP response is normally program output, not a static file.

Passing Parameters with a POST

The CGI script/JSP/JavaServlet receives the message body and decodes it. Here's a typical form submission, using POST:

```
POST /path/script.cgi HTTP/1.0
User-Agent: HTTPTool/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

username=joe&password=xpto
```

Just make sure the sender and the receiving program agree on the format.

Web Page Example

```
<html>
<head>

<title> Create New Account </title>

</head>
<body>

<form action = "cgi-bin/create.p1" method = "post" [get] >
    Enter user name:      <input name = "user">
    Password:        <input name = "pass1" Type = "password">
    Re-enter Password :      <input name = "pass2" Type = "password">
    <br>
    <input type = "submit" value = "create account" >
    <input type = "reset" value = "start over" >
    </form>

</body> </html>
```

The user fills in the form and hits the SUBMIT button

Request Message for POST

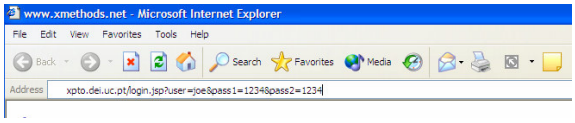
```
POST   cgi-bin/create.p1    HTTP 1.1
Host:   xpto.dei.uc.pt
Accept:   image/gif, image/x-xbit, image-jpeg, image/pjpeg, */
Content-type:   application/x-www-form-urlencoded
Content-length:   37

user=joe&pass1=1234&pass2=1234
```

Data is sent to the server inside the HTML message.

Request Message for GET

```
GET  login.jsp?user=joe&pass1=1234&pass2=1234   HTTP 1.1
Host: xpto.dei.uc.pt
Accept: image/gif, image/x-xbit, image-jpeg, image/pjpeg, */*
```



Beware that some systems put a limit of 256 chars in the URL...
If the resulting URL is bigger than 256 chars is better to use POST.
The other problem is related with security...

Advanced Topics:

- *Cookies*
- *Proxies*
- *Web caching*
- *Conditional Get*
- *Content Distribution Networks*

Host Header in HTTP1.1

- Starting with HTTP 1.1, one server at one IP address can be *multi-homed*, i.e. the home of several Web domains.
- For example, "`www.host1.com`" and "`www.host2.com`" can live on the same server.
- Thus, every HTTP request must specify which host name) the request is intended for, with the **Host:** header. A complete HTTP 1.1 request might be:

```
GET /path/file.html HTTP/1.1
Host: www.host1.com
[blank line here]
```

How to solve the problem of a stateless HTTP?

- A problem of the HTTP protocol is that every request is completely unrelated to any other previous request.
- The protocol at the HTTP Server is stateless.
- Solution: use **cookies**.

- The server returns a "**Set-cookie**" header that gives a cookie name, expiry time and some more info.

- **Cookies** are stored as plain text files in the local disk.
- When the user returns to the same URL the browser returns the cookie if it hasn't expired.

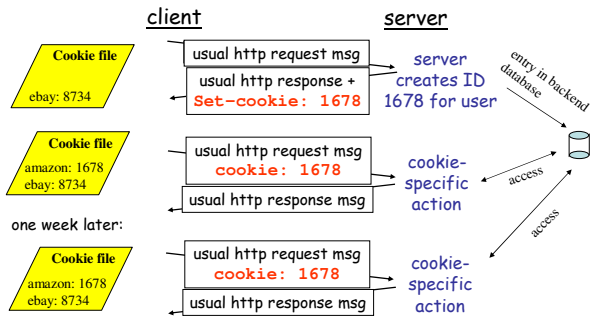
Cookies: keeping "state"

Many major Web sites use cookies

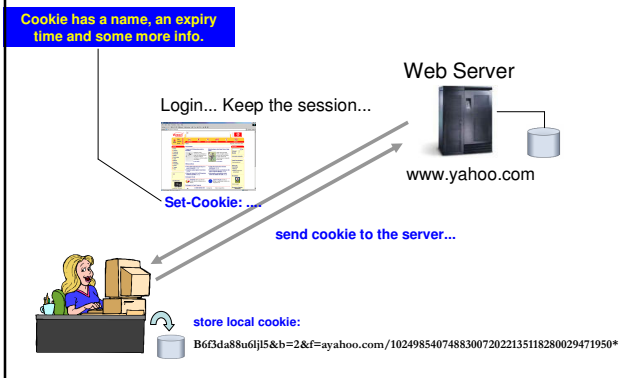
Four components:

1) cookie header line in the HTTP Response message
2) cookie header line in HTTP Request message
3) cookie file kept on user's host and managed by user's browser
4) back-end database at Web site

Cookies: keeping "state" (cont.)



Cookies



Cookies (continued)

What cookies can bring:

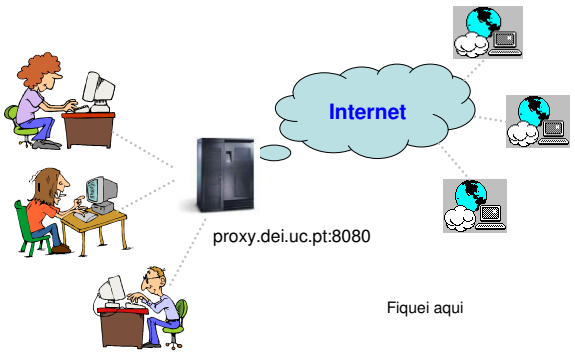
- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

Cookies and privacy:

- cookies permit sites to learn a lot about you
- search engines use redirection & cookies to learn yet more

Local Caches + Proxy Caches



Fiquei aqui

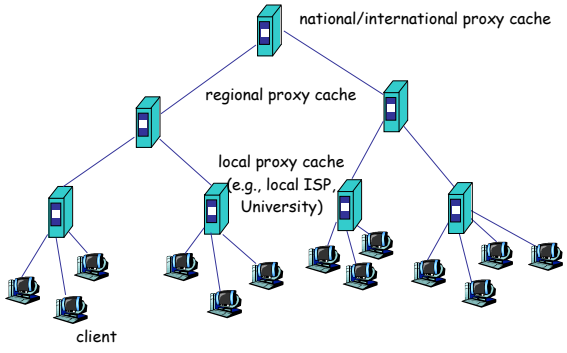
HTTP Proxies

- An *HTTP proxy* is a program that acts as an intermediary between a client and a server.
- It receives requests from clients, and forwards those requests to the intended servers. The responses pass back in the same way.
- Proxies are commonly used in firewalls, for LAN-wide caches, or in other situations.

- When a client uses a proxy, it typically sends all requests to that proxy, instead of to the servers in the URLs.
- Requests to a proxy differ from normal requests in one way: in the first line, they use the complete URL of the resource being requested, instead of just the path. For example,
  GET http://www.somehost.com/path/file.html HTTP/1.0

- That way, the proxy knows which server to forward the request to.

Web Caching Hierarchy



Why Caching?

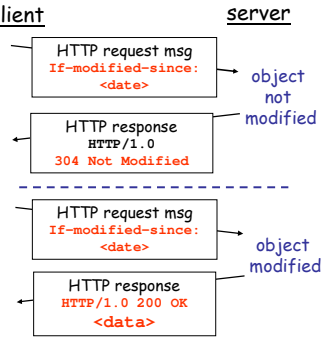
- Reduce response time for client request.
- Reduce traffic on an institution's access link.

Caching...

- Not all objects can't be cached
  - E.g., dynamic objects
- Cache consistency
  - strong
  - weak
- Cache Replacement Policies
  - Variable size objects
  - Varying cost of not finding an object (a "miss") in the cache
- Prefetch?
  - A large fraction of the requests are single requests..

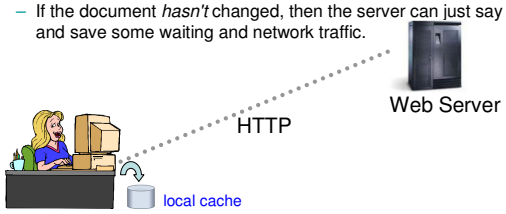
Conditional GET: client-side caching

- **Goal: don't send object if client has up-to-date cached version**
- **client: specify date of cached copy in HTTP request**
  `If-modified-since: <date>`
- **server: response contains no object if cached copy is up-to-date:**
  `HTTP/1.0 304 Not Modified`

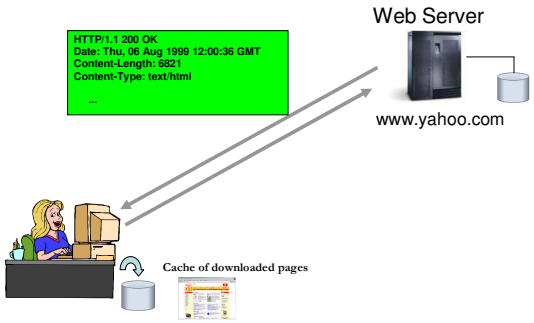


Caching of HTML Documents

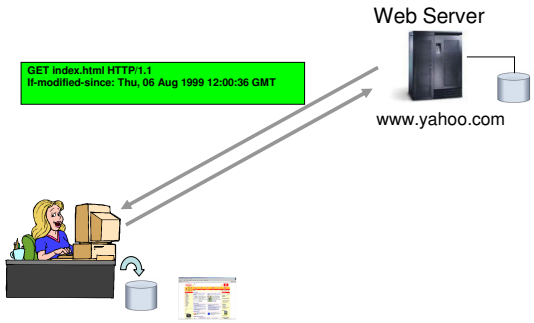
- If a browser already has a version of the document in its **cache** it can include the field **If-Modified-Since** set it to the time it retrieved that version.
- The server can then check if the document has been modified since the browser last downloaded it and send it again if necessary.
- If the document *hasn't* changed, then the server can just say so and save some waiting and network traffic.



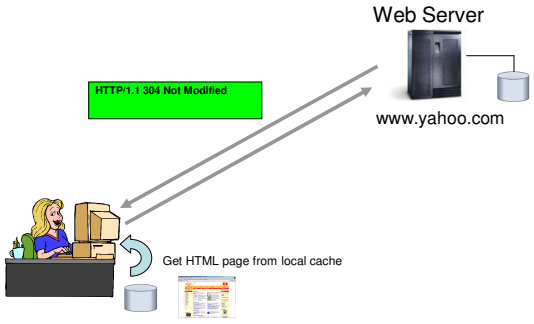
Caching: first response



Caching: next request

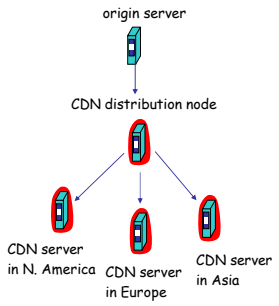


Caching: second request



Content distribution networks (CDNs)

- **Content replication**
- **CDN company installs hundreds of CDN servers throughout Internet**
  - **in lower-tier ISPs, close to users**
- **CDN replicates its customers' content in CDN servers. When provider updates content, CDN updates servers**



Persistent Connections

- In HTTP 1.0, TCP connections are closed after each request and response, so each resource to be retrieved requires its own connection.
- Opening and closing TCP connections takes a substantial amount of CPU time, bandwidth, and memory.
- Most Web pages consist of several files on the same server, so much can be saved by allowing several requests and responses to be sent through a single *persistent connection*.
- Persistent connections are the default in HTTP 1.1.
- The browser just opens a connection and send several requests in series (called *pipelining*), and read the responses in the same order as the requests were sent.

"Connection: close" Header

- If a client includes the "**Connection: close**" header in the request, then the connection will be closed after the corresponding response.
- **Use this if you don't support persistent connections**, or if you know a request will be the last on its connection.
- Similarly, if a response contains this header, then the server will close the connection following that response, and the client shouldn't send any more requests through that connection.

- **Connection: close**
- **Connection: keep-alive**

The Date: Header

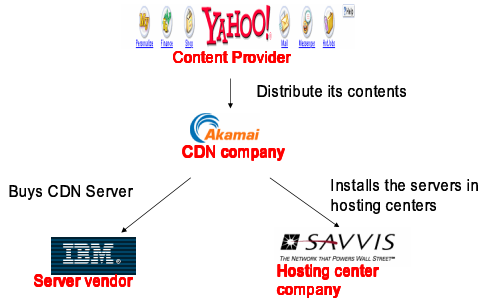
- Caching is an important improvement in HTTP 1.1, and can't work without timestamped responses.
- So, servers must timestamp every response with a **Date:** header containing the current time, in the form:

Date: Fri, 31 Dec 1999 23:59:59 GMT

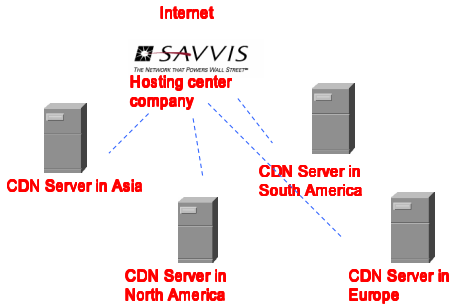
- All responses except those with 100-level status must include the **Date:** header.
- All time values in HTTP use Greenwich Mean Time.

Content Distribution Networks

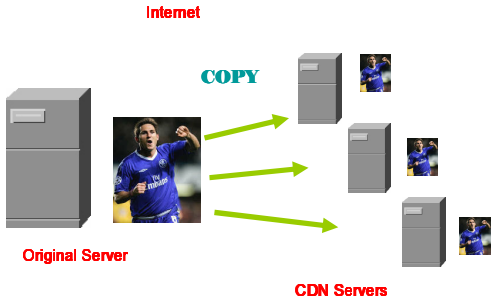
4 independent companies



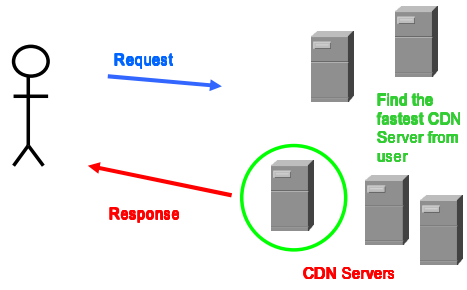
How to provide CDNs services



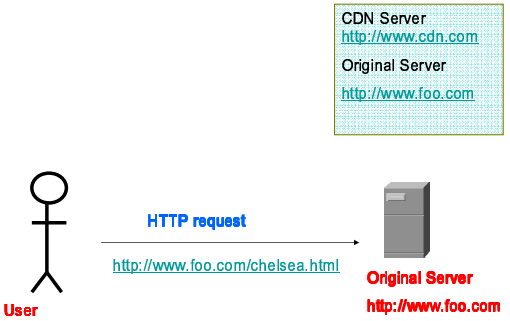
How to provide CDNs services



How to provide CDNs services



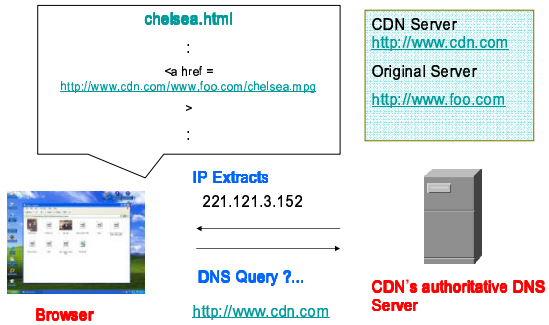
Usage Scenario



Páginas HTML são distribuídas pelo content-provider.

Ficheiros de vídeo são distribuídas pela empresa que presta o serviço de CDN.

Usage Scenario



Usage Scenario