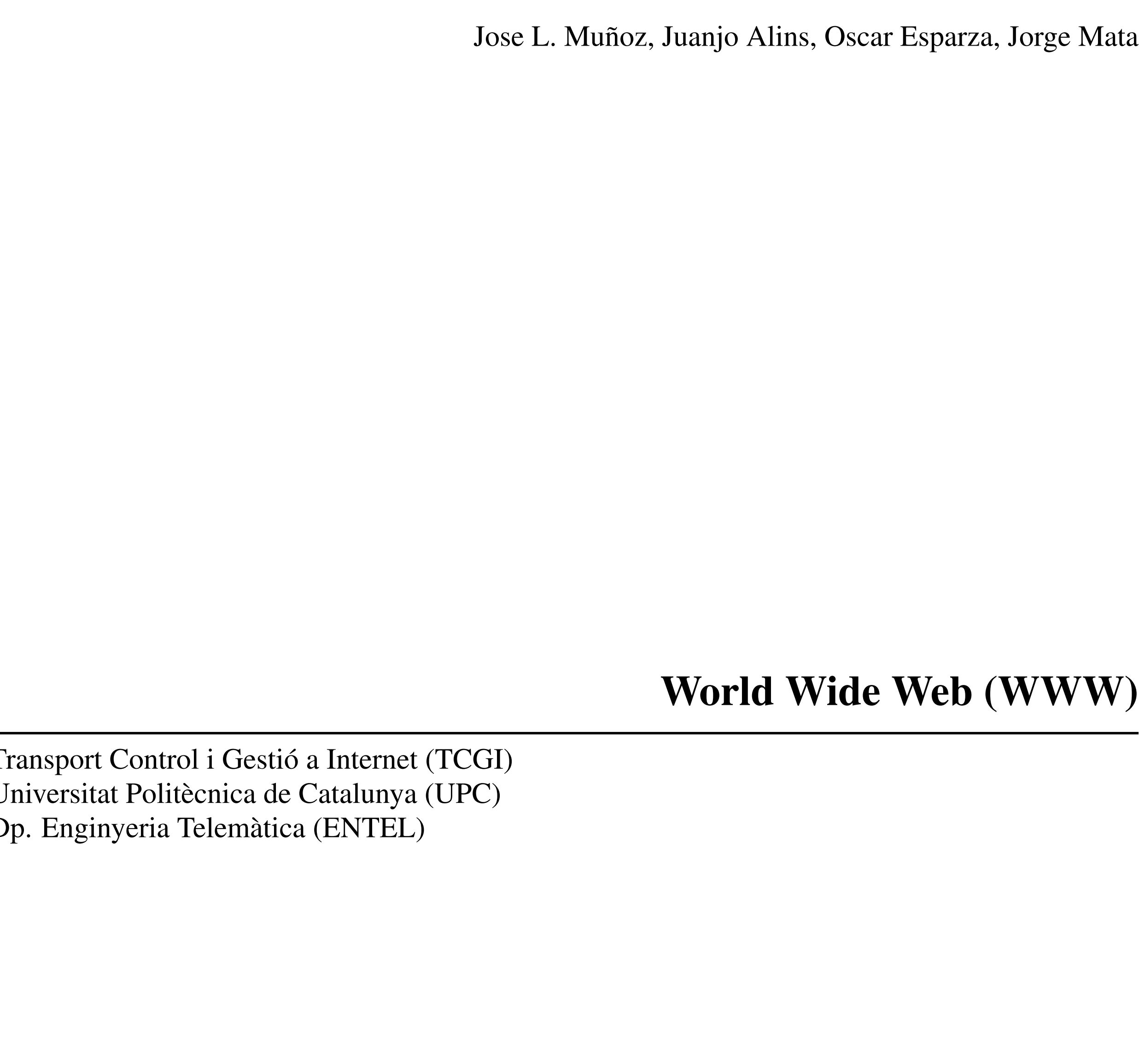
Jose L. Muñoz, Juanjo Alins, Oscar Esparza, Jorge Mata



World Wide Web (WWW)

Transport Control i Gestió a Internet (TCGI)
Universitat Politècnica de Catalunya (UPC)
Dp. Enginyeria Telemàtica (ENTEL)

Contents

1 WWW **5**
1.1 History 5
1.2 HTML Documents 5
1.3 HTTP Motivation 7
1.4 URL/URI 7
1.5 HTTP 1.0 8
1.5.1 HTTP Requests 8
1.5.2 Headers 8
1.5.3 HTTP Responses 9
1.6 Cookies 10
1.7 HTTP Proxies 11
1.8 Dynamic Web 11
1.8.1 Introduction 11
1.8.2 CGIs 11
1.8.3 Other Methods in Requests: HEAD and POST 12
1.8.4 HTML Forms 13
1.9 HTTP 1.1 14
1.9.1 Introduction 14
1.9.2 Host header 14
1.9.3 Chunked Data 15
1.9.4 Persistent Connections 15
1.9.5 Continue 16
1.9.6 Caching 16
1.9.7 Final Remarks 17
1.10 Related RFCs 17
1.11 Practical HTTP with `apache` 17
1.11.1 Introduction 17
1.11.2 Virtual Hosts (sites) 18
1.11.3 CGIs 19
1.11.4 Modules 20
1.12 Commands summary 20
1.13 Practices 20

Chapter 1

WWW

1.1 History

Tim Berners-Lee is credited with having created the initial World Wide Web (WWW) during 1985-1991, while he was a researcher at the European High-Energy Particle Physics lab at CERN (Centre Européen de Recherche Nucléaire). In this context, a multi-platform tool was needed to enable sharing documents between physicists and other researchers in the high energy physics community. Tim Berners-Lee wrote a proposal that was a solution for enabling such collaboration. Four basic technologies were part of his proposal:

- HTML (HyperText Markup Language): a language to write documents.
- HTTP (HyperText Transfer Protocol): a protocol to transmit resources (like HTML documents).
- A WEB server: a software that serves resources like HTML documents.
- A WEB browser: a software that acts as client to send requests and process responses for resources available on a WEB server (like HTML documents).

1.2 HTML Documents

HTML (HyperText Markup Language) as its name states is not a programming language like C or Java but a markup language. In plain English, this means that HTML is a language for describing how content (text, images, etc.) should be displayed. With the HTML language, we can create HTML documents to be displayed in a browser. HTML documents are just text files so you can edit them with any text editor. There are also available “HTML editors”, specially designed for writing HTML. Analyzing HTML documents is a good way of learning HTML. Let’s take a look at a simple HTML document (see Code 1.1).

```
<html>
  <head>
  <title> Hello World</title>
  <meta http−equiv="content−type" content="text/html; charset=UTF−8">
  </head>
  <body>
  Hello <b>World</b>!!!!!!!
  </body>
</html>
```

Code 1.1: Simple HTML document

As you observe, the HTML document is just text. However, some of the text is considered “hypertext”, which means that it has a special meaning in HTML. Text enclosed between the characters “<” and “>” is hypertext and those hypertexts are called “HTML tags”. HTML tags tell the browser to do something special. In our example,

"World" tells the browser to use the boldface font. As you see, some HTML tags have an opening tag and an ending tag. This is marked as <tag> ... </tag>, like in the case of the boldface tag. Other tags however, are just composed of a single tag. The HTML document is delimited by <html> and </html>. In addition, the HTML document is divided in two parts:

- **<head>**. This part is optional. When <head> exists, it can contain several labels like <title>, <meta> etc. For example, the <title> tag specifies the title that must be displayed in the browser's window. With the <meta> tag we can define the charset:

```
<meta http−equiv="content−type" content="text/html; charset=UTF−8">
```

- **<body>**. Inside the body is where the whole HTML document is specified. All text, images, etc. are contained between <body> and </body>.

On the other hand, we can also use tags to create hyperlinks to other resources (like other HTML documents). This is a fundamental feature in HTML. The hyperlink tag is <a>... . To see an example, look at Code 1.2:

```
<html>
  <head>
  <title> Hello World</title>
  <meta http−equiv="content−type" content="text/html; charset=UTF−8">
  </head>
  <body>
  Hello <b>World</b>!!!!!!!
  Go to <a href=docs/otherdoc.html> another
  document </a>
  </body>
</html>
```

Code 1.2: Simple HTML document with an hyperlink.

In the previous example, we link our HTML document with another HTML document that is located in a relative directory called "docs". Relative paths are described taking the location of the HTML document as reference. Notice that the hyperlink is specified as a parameter "href" inside the opening tag. The <img> tag is used to display an image. The src attribute provides the path to the image. Example:

```
<img src="pictures/image.gif">
```

On the other hand, blank spaces and new lines are called "whites". You can add as many "whites" as you like to make your HTML file easier to read but browsers display consecutive whites as a single space. If you need to create a paragraph, you have to use the labels <p> ... </p>. For paragraphs, the browser will adjust the text lines correctly based on the window width. If you really want to force a new line, you have to use the <br> tag. HTML has many tags but with a few of these tags, we can have an idea about how HTML works. Some more useful tags are:

- **<i> </i>** Sets text in italics.
- **<tt> </tt>** Sets text in teletype.
- **<h1> </h1>** Sets text in type "header 1". You can use numbers of headers in descending order of importance (size): <h2> </h2> ... <h6> </h6>
- **<hr>** Prints an horizontal line.
- **<center> </center>** Centers text and images.
- **<blockquote> </blockquote>** Indents text.
- **<pre> </pre>** Pre-formatted text, i.e. spaces and line breaks between these tags are maintained.
- **<!-- text comments... -->** Comments in the HTML file.

1.3 HTTP Motivation

Initially, HTTP (Hypertext Transfer Protocol) arised from the necessity of creating hyperlinks in HTML documents to resources that are not on the same host. HTTP is a text protocol and it is based on a client/server model that can be used over a TCP/IP network to deliver virtually any resource of the World Wide Web (WWW). For now, we will consider that a resource is just an HTML document. An HTTP server or WEB server is a network daemon that uses by default the well-known TCP port 80. HTTP clients, generically called WEB Browsers (e.g. `firefox` or `lynx`), send HTTP requests to the HTTP servers asking for a resource and the server responds with the requested resource (see Figure 1.1).

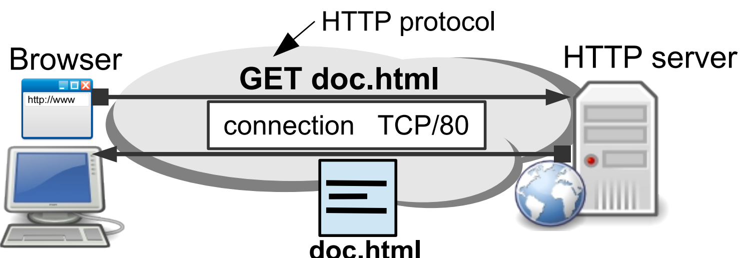


Figure 1.1: HTTP client/server.

1.4 URL/URI

The first issue to implement HTTP is to define how to identify resources. The identifiers used in HTTP were initially defined by Tim Berners in 1991. They were called URLs (Uniform Resource Locators) and they were first used to allow authors of HTML documents to establish hyperlinks in the WWW. An URL is just a text string with a standard format that allows you to name a resource based on its location on the WWW. In 1994, the URL concept was incorporated into a more general concept called URI (Uniform Resource Identifier). URI is the standard name for resource identifiers in the Internet, but the term URL is still widely used. The simplest URL/URI format is as follows:

```
protocol://hostname/directory/resource
```

But, other information can also be present in the URL:

```
protocol://username:password@hostname:port/directory/resource
```

The detailed specification for URL/URIs is in RFC 1738. Some examples are:

- http://www.example.com/pictures/upc.jpg
- http://www.example.com
- http://192.168.0.5
- http://www.example:8080/cgi-bin/time.sh
- http://user:hello1234@someserver.com/
- ftp://debian.org

If in the URL there is not any resource (filename) specified, it is assumed that the client is asking for a file called *index.html* or *index.htm*. As its name suggests, this file contains an HTML file with the Web site index.

On the other hand, we can use absolute or relative paths in HTTP hyperlinks. In an HTTP server, **absolute paths are related to a directory called DocumentRoot**. This parameter is defined in the configuration file of the HTTP server. For example, a typical DocumentRoot when using Linux is /var/www. In this case, the URL http://www.example.com/images/upc1.gif refers to a file called upc1.gif that is stored in the HTTP server in the directory /var/www/images. The following HTML file serves as an example of how to use absolute and relative paths:

```
<html>
  <head>
  <title> Hello World</title>
  </head>
  <body>
  <p>Hello <b>World</b>!!!!!!!</p>
  <p>Go to <a href=docs/otherdoc.html> another document </a></p>
  <p>You can visit the UPC home page at <a href="http://www.upc.edu">UPC home</a>. </p>
  <img src="/images/upc1.gif">
  <img src="/images/upc2.gif">
  <img src="http://www.example.com/images/upc1.gif">
  </body>
</html>
```

Code 1.3: Simple HTML Document with an Absolute Path and External Hyperlinks.

1.5 HTTP 1.0

HTTP is a text protocol that uses the client-server model like many other TCP/IP applications and 80 as its default port. Other TCP port can be used but the client must know this port and include it in the URL. Then, the HTTP client opens a TCP connection and sends an HTTP request to an HTTP server. If everything is correct, the server returns an HTTP response that contains the requested resource. After delivering the response, the HTTP server closes the TCP connection. HTTP is a stateless protocol, which means that HTTP does not maintain state information between different requests.

1.5.1 HTTP Requests

In an HTTP request, the first line is the only one mandatory and it contains the “request method”, the path to the resource and the HTTP version. Then, it follows a blank line (CR+LF). The minimal request in HTTP 1.0 is something like the following:

```
GET / HTTP/1.0
[blank line]
```

GET is the most commonly used request method and it means “give me this resource”. After the GET keyword we find a “/”. This means that the resource that we are requesting is the index file of the WEB server. Finally the line ends with a CR+LF ([blank line]). Another example is:

```
GET /images/upc1.gif HTTP/1.0
[blank line]
```

In this case, the client is requesting a file called upc1.gif that is stored in the HTTP server in the directory images (relative to the server’s DocumentRoot).

1.5.2 Headers

Requests (and also responses) can have header lines. Headers are text lines that provide additional information or functionality in requests/responses. The format is ”Header-Name: value1, value2”, ending with CR+LF. The header name is not case-sensitive. There can be any number of spaces or tabs between : and the value. The header lines starting with space or tab are actually part of the previous header line (used for readability). The following headers are equivalent:

```
Header1: some-long-value-1a, some-long-value-1b

Header1: some-long-value-1a
  some-long-value-1b
```

HTTP 1.0 defines 16 headers, though none is required. Typical headers included in the requests are:

- From: gives the email address of the user who makes the request.
- User-Agent: name of the browser and OS.

For example, a request with headers could be the following:

```
GET /path/file.html HTTP/1.0
From: user@example.net
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:24.0) Gecko/20100101 Firefox/24.0
[blank line]
```

The headers can help to solve the problems in web sites but they also reveal information about the user. Thus, notice that there is a trade-off between information provided for debugging and the user privacy.

1.5.3 HTTP Responses

HTTP responses are also composed of text lines. The first text line of an HTTP response is the status. Typical status lines are:

```
HTTP/1.0 200 OK
HTTP/1.0 404 Not Found
```

The first digit identifies the general category of the status:

- 1xx indicates an informational message only.
- 2xx indicates success of some kind.
- 3xx redirects the client to another URL.
- 4xx indicates an error in the client side.
- 5xx indicates an error in the server.

Examples:

- 301 Moved Permanently.
- 302 Moved Temporarily.
- 303 See Other (HTTP 1.1 only. Means that the resource has been moved to another URL given by the location header in the response).
- 500 Server Error.

On the other hand, a response can also have headers. The headers usually included in responses by servers are:

- Server: header is analogous to the User-Agent (it identifies the server software).
- Date: current date.
- Last-Modified: date of last modification of the resource being returned. This header is used for caching (explained later).

After the headers, if the resource was available in the server, we can find a CR+LF and then the response's body containing the requested resource. In general, if an HTTP message includes a body, there are at least two additional header lines to describe the body's content. These header lines are "Content-Type" and "Content-Length":

- Content-Type: MIME-type of the object.
- Content-Length: number of bytes of the object.

For example, to retrieve the file http://www.example.com/path/file.html using HTTP 1.0, the first step is to open a TCP connection with the server www.example.com using the HTTP default TCP port 80. Then, through this connection the client could send an HTTP 1.0 request like the following:

```
GET /path/file.html HTTP/1.0
From: user@example.net
[blank line]
```

Through the same socket (connection), the server could respond with something like the following:

```
HTTP/1.0 200 OK
Date: Mon, 21 Oct 2013 22:29:59 GMT
Content-Type: text/html
Content-Length: 50
[blank line]
<html>
<body>
<h1>It works!</h1>
</body>
</html>
```

After receiving the response, in the basic implementation of HTTP 1.0, the client closes the TCP socket.

1.6 Cookies

As previously mentioned, HTTP is a stateless protocol, which means that HTTP does not maintain state information between different requests. A **cookie** is a piece of information (UTF8 text) sent from an HTTP server and that is stored by the browser in the client's filesystem. Sometimes cookies are also called footprints. The browser returns cookies unchanged to the server. Cookies provide a state (memory of previous events) into otherwise stateless HTTP transactions. Without cookies, each retrieval of a web page or component of a web page is an isolated event, mostly unrelated to all other views of the pages of the same site. The most common uses of cookies are:

- User Control. For example, when a user enters his username and password, a cookie can store this information so there is no need to enter them again in a later visit to the web server.
- Getting information about user's browsing habits.

The HTTP server sends lines with the Set-Cookie header if the server wishes the browser to store these cookies. Set-Cookie is a directive for the browser to store the cookie and send it back in future requests to the server (subject to expiration time or other cookie attributes). For example, the browser requests the resource http://www.example.org/doc.html (see Figure 1.2).

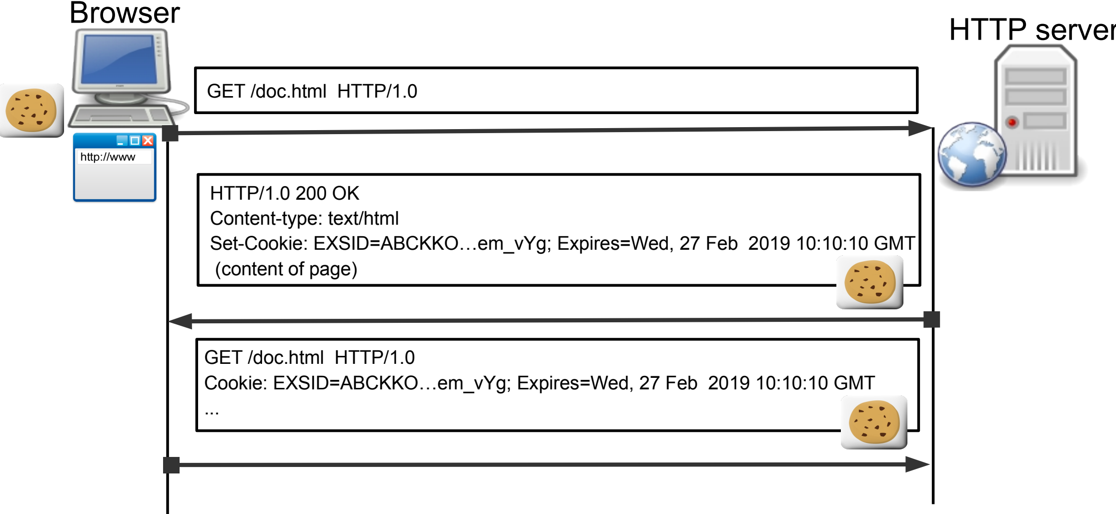
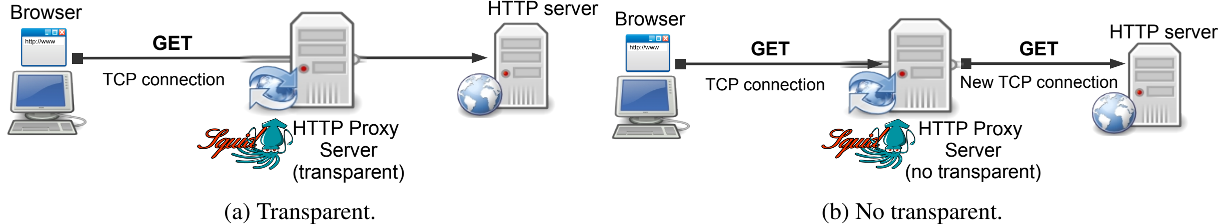


Figure 1.2: How HTTP cookies work.

The client sends a regular request, then the server asks the client to store the cookie. Then, the client sends the cookie in a subsequent request. It is worth to mention that there are more fields (like path and domain) in the cookie to help in deciding when to send it or not. Finally, as a you may imagine cookies can cause problems of privacy.

1.7 HTTP Proxies

An HTTP proxy is a program that acts as an intermediary between a browser and a Web server. HTTP Proxies are typically used for security (a single point of control) or efficiency (caching).



(a) Transparent. (b) No transparent.

Figure 1.3: HTTP Proxies.

From the point of view of users, there are two basic types of proxies:

- **Transparent** (Figure 1.3a). A transparent proxy intercepts normal communication at the network layer without requiring any special client configuration. Clients need not be aware of the existence of the proxy.

- **No transparent** (Figure 1.3b). A proxy that is not transparent receives requests from clients and sends requests to servers. The responses go the way back also using the proxy. Therefore, a proxy has functions of a client and a server. A non-transparent proxy can use another transparent or non-transparent proxy to reach the final server.

  Clients send their requests to the proxy instead of the real server specified in the URL (the proxy IP address and port is defined in the browser). HTTP requests using a non-transparent proxy must include the full URL of the resource (not only the relative path). In this way, the proxy knows to which server it must send the HTTP request. For example:

```
GET http://www.somehost.com/path/file.html HTTP/1.0
[blank line]
```

Finally, it is worth to mention that we have open source HTTP proxy implementations like Squid (which widely used).

1.8 Dynamic Web

1.8.1 Introduction

In today's Web, the content is not static but documents are generated on the fly by servers with information provided by clients. As a result, WWW is not just a huge database of documents or content but a platform to implement services and applications. Common applications of the dynamic web are searching engines, remote access to corporate applications and databases, etc.

1.8.2 CGIs

There are several ways of implementing the dynamic Web. In this document, we only deal with CGIs because they are easy to understand and they were the first method used for such purpose. **CGIs** or Common Gateway Interfaces are a standard procedure through which HTTP servers can use external applications to dynamically generate content (see Figure 1.4).

When we use a CGI, the URL identifies:

- An executable program (which is also called "the CGI").
- The parameters with which the CGI has to be executed.

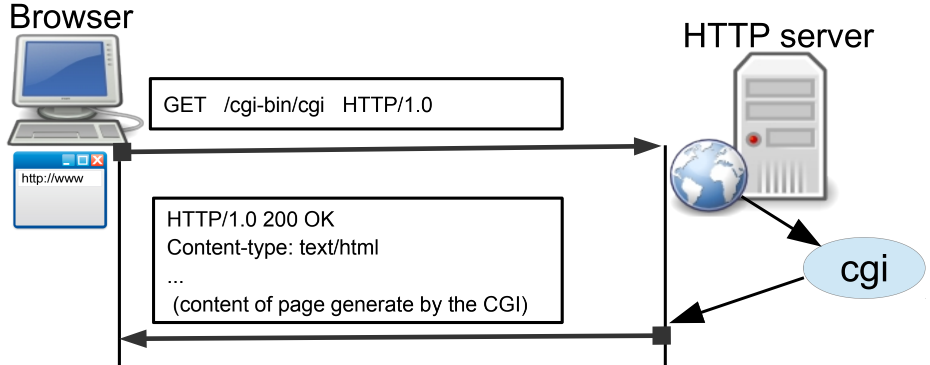


Figure 1.4: How CGIs work in HTTP.

The first issue to take into account is how a web server knows that it has to execute a program instead of sending a resource. An usual solution is to store all the CGIs in a special directory, typically called /cgi-bin/. In this way, if a client asks for www.example.com/cgi-bin/program the server knows that it must execute `program` instead of sending it. The second issue is how to send the parameters to the `program`. When using GET, the parameters are encoded in the URL. These parameters are added to the URL after a character "?" and separated by the character "&". Example:

```
http://www.example.com/cgi-bin/program?param1=value1&param2=value2...
```

Note. Spaces are translated using the character "+" and ASCII characters can also be sent in the format %NNN, where NNN is the ASCII code number.

Finally, before executing the CGI, the Web server establishes a special context for the `program` using environment variables. These variables are:

CONTENT_LENGTH, CONTENT_TYPE, REMOTE_HOST, REMOTE_USER, REQUEST_METHOD, SERVER_NAME, QUERY_STRING, GATEWAY_INTERFACE, HTTP_*

For GET requests, the QUERY_STRING variable takes the value of the parameters, as shown in the URL. In this manner, the CGI can get the parameters that the client has specified. Regarding the response, the CGI writes it to the standard output (STDOUT). Then, the server reads this answer and sends it to the client through the socket. Depending on the type of web server, the CGI application can act in two ways:

- NPH Server (No Parse Header). The CGI application must write the complete response including the HTTP headers.
- PH Server (Parse Headers). The CGI application must write a response without HTTP headers and it must pass information to the server on how to form the headers.

Typically, web servers are NPH.

Finally, we would like to remark that CGIs are not the most efficient solution because **a process is created per request**. Today we have other solutions more efficient or flexible to create dynamic Websites like Javascript, Phyton, PHP, JAVA servlets, etc.

1.8.3 Other Methods in Requests: HEAD and POST

On the other hand, there are other methods to request information from a Web server besides GET.

- The **HEAD** method is used when we want a response only with the status line and headers (without a body). HEAD is useful when the resources from the server are not actually needed. This can be the case in which we need to make some tests but we do not want to download a resource (which can be heavy).
- The **POST** request is used for dynamic Web. The difference between POST and GET is that POST requests use the body of the request to send parameters to programs instead of coding the parameters in the URL. CGIs with POST use the standard input (STDIN) to receive the request body (program's parameters) instead of using the variable QUERY_STRING.

1.8.4 HTML Forms

An HTML form allows a client to send parameters to a WEB server. The tag to declare a form is <FORM>. Different elements can be inserted into the form: text input elements, codes, images, files, checkboxes, etc. These elements are inserted in the form using the <INPUT> tag. All the items of the form have a “type” attribute and they might have a ”name” attribute. There are two special elements: RESET, which clears the form to its original state and SUBMIT, which presents a button to send the form. Example:

```
<html>
<head>
<title> Website title </title>
</head>
<body>
Form to select parametres to send to the server.
<form ACTION="/cgi-bin/process" METHOD="GET">

Enter a name: <INPUT NAME="a" TYPE="text"> <br>

Enter a password: <INPUT TYPE="password" NAME="b" MAXLENGHT="8"> <br>

Checkbox: <INPUT TYPE="checkbox" NAME="c"> <br>

<INPUT TYPE="reset"> <INPUT TYPE="submit">
<br>
</form>
</body>
</html>
```

Code 1.4: HTML document with a form.

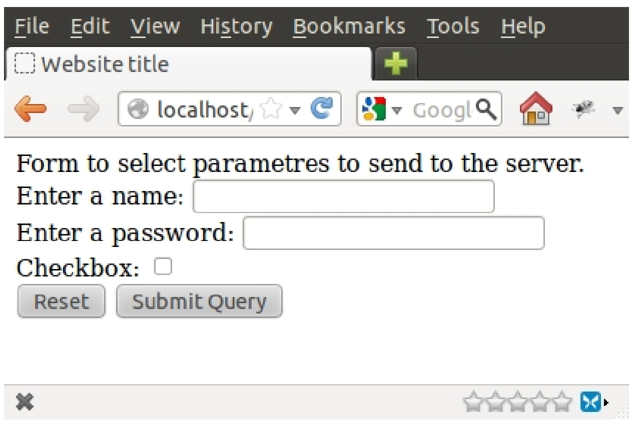


Figure 1.5: An HTML form viewed from a browser.

Figure 1.5 shows the form viewed in a WEB browser. When the form is sent (pressing the submit button), the client generates an HTTP request using the method (GET or POST) showed in the METHOD attribute to execute the script or application indicated in the ACTION attribute.

As already discussed, GET requests do not have a body but parameters for the execution of the application are encoded in the URL, while POST requests have a body with the parameters. Using the content-type header, the POST request defines how the parameters-values have been encoded:

- application/x-www-form-urlencoded. This is the default encoding type. It is similar to the encoding used by GET. You cannot send a body in the request.

- multipart/form-data. Separates parameters with a mark (boundary). You can also include a body (e.g. a binary file) in the request.

Example:

```
POST /cgi-bin/program HTTP/1.0
From: user@example.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
[blank line]
param1=value1&param2=value2
```

The question is: use GET or POST? Actually, each method has advantages and drawbacks. When using GET, the parameters for the server are encoded in the URL. This can be considered security vulnerability because these parameters can be read by anyone. Another drawback of GET is that it does not allow sending binary files in the body of the request. However, GET is useful to perform requests and store the results together with the associated URL (that contains all the parameters of the query). GET also allows to use the back button to go to the previous results.

On the other hand, with POST the parameters for the server are sent in the body of the request. With POST the parameters are not visible in the browser as a query string. In general, GET is useful for idempotent operations (which always give the same result). POST means ”carry out” an action with a ”side effect” or a change of state (non-idempotent operations).

1.9 HTTP 1.1

1.9.1 Introduction

HTTP 1.1 defines 46 headers, and one of them “Host” is mandatory in requests. HTTP 1.1 was defined to face up new needs and to overcome the shortcomings of HTTP 1.0. In general terms, HTTP 1.1 is a superset of HTTP 1.0. These improvements include:

- **Host header.** Provides efficient use of IP addresses. Now, multiple domains can be served from a single IP address.
- **Chunked encoding.** Allows a faster response for dynamically generated pages. Pages are divided and sent in chunks (fragments). In this way, a response can be sent before its total content or length is known.
- **Persistent connections.** A TCP connection is not opened/closed for each request. By allowing multiple HTTP transactions in one TCP connection we can reduce the total transmission delay.
- **Caching.** The protocol provides headers to implement caching. This allows a faster response and bandwidth savings.

HTTP 1.1 requires changes in both client and server. Next, we describe in more detail each of the previous features.

1.9.2 Host header

From HTTP 1.1, web servers can be multi-domain. For example, we can have the domain ”www.example.com“ and ”www.example.net” on the same server. Thus, the IP address of the server is not enough to figure out which is the domain to be served. An analogy is a situation in which several people share a phone, then, when we call, we have to ask who is speaking and possibly ask for the correct person. So, in HTTP 1.1, each request must specify the hostname (and optionally the port). A minimal HTTP request for version 1.1 could be the following:

```
GET / HTTP/1.1
Host: www.example.com:80
[blank line]
```

The host header contains the domain name or IP address of the WEB server. The port number (”:80”). In this case, specifying the port is not necessary because 80 is the default port for HTTP.

Regarding HTTP proxies and the Host header, the destination for a request can appear in the URL (as an absolute URI) as well as in the Host header. So, it is important for proxies to behave correctly when both appear. In short, the host and port in an absolute URI always override the Host header. For example:

```
GET http://example.net/foo HTTP/1.1
Host: www.example.com:8000
```

Here, the server that will be used is example.net and the port 80 (the default for HTTP).

1.9.3 Chunked Data

This mechanism allows a server to start sending a response before knowing the complete content, that is to say, before knowing the total length of the content. The idea is to divide the response in small pieces called "chunks" and send these chunks one after another. Responses divided in chunks are identified by the header "Transfer-Encoding: chunked". All HTTP 1.1 clients must be able to correctly process responses divided in chunks. The body of a message such as "chunked" contains: several fragments (chunks) followed by a line with "0" (zero). Optionally followed by the foot of the page (footers). Each "chunk" consists of two parts: (1) a line with the size of the chunk in hexadecimal + CR+LF and (2) Data + CR+LF. Example without chunks:

```
HTTP/1.1 200 OK
    Content-Type: text/plain
    Content-Length: 42
    [blank line]
    abcdefghijklmnopqrstuvwxyz1234567890abcdef
```

The same example with chunked data:

```
HTTP/1.1 200 OK
    Content-Type: text/plain
    Transfer-Encoding: chunked
    [blank line]
    1a
    abcdefghijklmnopqrstuvwxyz
    10
    1234567890abcdef
    0
    [blank line]
```

1.9.4 Persistent Connections

In HTTP 1.0, TCP connections are closed after each request/response by default. As we know, opening/closing TCP connections requires a substantial amount of CPU time, bandwidth, and memory. In practice, most web pages consist of several files (linked HTML documents, images, etc.) that are located on the same server. Consecutive requests (and their associated responses) can be more efficiently transmitted by allowing multiple requests/responses to be sent over a single connection. This is mechanism is called "persistent connections".

In HTTP 1.1, persistent connections are used by default. We do not need anything special to use persistent connections. Simply, the clients open a connection, send multiple requests one after another and then, read the corresponding responses in order. The client can include a header "Connection: close". Then, the server has to close the connection after the reply. This should only be used if the client is unable to process persistent connections or if it is known that the request will be the last.

On the other hand, if a response contains the header "Connection: close", then, the client cannot send more requests through that connection and it must close the connection after the response is received. A server may close the connection before sending all the answers. In this case, the client is responsible for tracking the answered requests and resend these unanswered requests if necessary. The HTTP 1.1 client can also send multiple requests through a single connection without having received any response (pipelining).

On its side, an HTTP 1.1 server must store queued requests while it can not process them, and it must send the responses in the same order as it received the requests. If a request includes the header "Connection: close", the server must interpret this as that the request is the latest and it must close after sending the corresponding response. The server also closes idle connections (after a period of time, typically 10 seconds). Some servers do not support

persistent connections to save resources (minimize the number of concurrent open sockets). If it is not wanted to use the persistent connection, then the server can include the header ”Connection: close” in each response.

Finally, it is worth to mention that typically, clients (browsers) open several simultaneous persistent TCP connections with each server. In the example of Figure 1.6, the browser uses 2 persistent connections with the HTTP server.

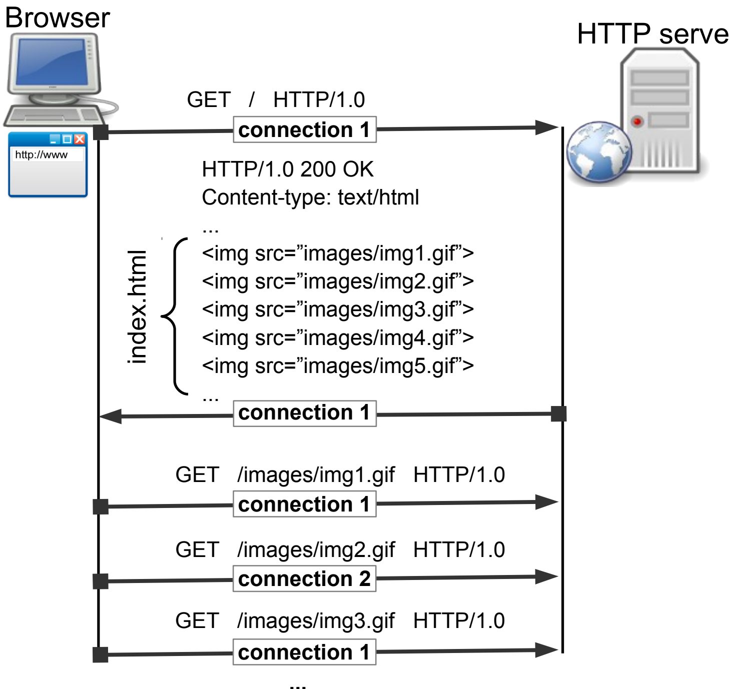


Figure 1.6: 2 Multiple Persistent Connections with an HTTP Server.

For example, in a real browser like `firefox` the default is having up to 6 persistent connections per server. In fact, this can be configured with the parameter `network.http.max-persistent-connections-per-server`. To configure this parameter, type `about:config` in the URL bar of `firefox`. Enabling multiple connections helps in increasing the performance since we can obtain more throughput from several connections than from just one connection. In addition, the client can send its requests in parallel through the different connections.

1.9.5 Continue

The “continue” mechanism allows to determine if the server is willing to accept a request based on the message headers. This is useful if a client has to send a request with a big body (e.g. big file). This mechanism prevents to waste time and resources if the server is going to reject the message (independently of its body). Clients include the header “Expect :100-continue“. Then, if the server is going to process the request must respond with 100 (Continue) status. A client should not send the Expect header if it is not going to send any body in its request.

1.9.6 Caching

All the responses (including errors) except the continue answers (status 100) should include the header ”date“. This header provides a time stamp which is used by HTTP 1.1 to implement caching. These time stamps use the Greenwich Mean Time (GMT). Unfortunately, due to earlier versions of HTTP, the value date can be in any of three possible formats:

If-Modified-Since: Mon, 27 Apr 2009 23:59:59 GMT
If-Modified-Since: Monday, 27-Apr-09 23:59:59 GMT

If-Modified-Since: Mon april 27 23:59:59 2009

Although servers can accept all three formats of date, HTTP 1.1 only generates the first type. There are two headers called ”If-Modified-since“ and ”If-Unmodified-Since“ that can be included in HTTP requests.

- The If-Modified-Since header means “send the response if it has changed since that date“.
- The If-Unmodified-Since header means ”send the response if it has not changed since that date“.

Clients are not required to use them but it is assumed that the HTTP 1.1 servers will consider these headers and proceed as follows:

- If we use If-Modified-Since in the request and the data of the response has not been changed, the server must send "304 Not Modified".
- If we use the header If-Unmodified-Since, and the data of the response has been modified, the server must send "412 Precondition Failed".

The most commonly used is the If-Modified-Since header. The If-Unmodified-Since has some not so common uses. As an example, it can be used in a situation in which you request a resource that needs other resources and that if someone changes the original resource in the meantime, this might lead to inconsistencies. In this case, we can use the if-unmodified-since header and the HTTP server will send us information if a record has been changed.

Finally, an **ETag** can be added. This is an identifier assigned by a web server to a specific version of a resource.

1.9.7 Final Remarks

As a final remark, say that HTTP 1.1 clients should:

- Include host header in each request.
- Accept responses with chunked data.
- Accept persistent connections or include the header ”Connection: close”.
- Manage the response ”100 Continue”.

HTTP 1.1 servers should:

- Require the host header in requests.
- Accept absolute URLs.
- Accept ”chunked” requests.
- Manage persistent connections (or use the header “Connection: close”)
- Properly use the status ”100 Continue”.
- Include the date in the header ”date” in each response (except Continue).
- Manage requests with headers “If-Modified-Since” or “If-Unmodified-Since”.
- At least, support the methods GET and HEAD.
- Support HTTP 1.0 requests.

1.10 Related RFCs

- RFC 1945. HTTP 1.0
- RFC 2616. Original specification of HTTP 1.1
- RFC 822. Structure of text messages including headers.
- RFC 2396. Definition of the URL/URI.
- RFC 1521. Definition of MIME types.

1.11 Practical HTTP with `apache`

1.11.1 Introduction

The Apache HTTP Server, commonly referred to as Apache, is a WEB server software notable for playing a key role in the initial growth of the WWW. Today it is also widely deployed in many sites. In our case, we are going to use its second version: the `apache2` daemon. One of the main advantages of `apache2` is its modular architecture. You can add or remove functionality as dictated by your requirements.

Debian-based distros store the Apache 2.0 configuration files in the directory /etc/apache2. Actually, this configuration file is used to load other configuration files. One of these other configuration files is ports.conf, which contains the Listen directives telling `apache2` what IP addresses and ports should listen to.

As usual, if you change the configuration of the daemon you have to stop and start it to apply the changes. As most of the network daemons, `apache2` can be started and stopped under Debian Linux using a script under the directory /etc/init.d. In particular, to stop `apache2` type:

```
# /etc/init.d/apache2 stop
```

To start the daemon type:

```
# /etc/init.d/apache2 start
```

1.11.2 Virtual Hosts (sites)

A virtual host is just a web site served by the HTTP server. Each virtual host or site has its own configuration file that contains all the directives that pertain only to that site (a sample configuration file is shown in Code 1.5).

```
<VirtualHost *:80>
        ServerAdmin webmaster@localhost
        DocumentRoot /var/www
       <Directory />
          Options FollowSymLinks
          AllowOverride None
     </Directory>
     <Directory /var/www/>
          Options Indexes FollowSymLinks MultiViews
          AllowOverride None
          Order allow,deny
          Allow from all
     </Directory>
          ScriptAlias /cgi-bin/ /var/www/cgi-bin/
     <Directory "/var/www/cgi-bin">
          AllowOverride None
          Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
          Order allow,deny
          Allow from all
     </Directory>
        ErrorLog ${APACHE_LOG_DIR}/error.log
        # Possible values include: debug, info, notice, warn, error, crit,
        # alert, emerg.
        LogLevel warn
        CustomLog ${APACHE_LOG_DIR}/access.log combined
        Alias /doc/ "/usr/share/doc/"
    <Directory "/usr/share/doc/">
        Options Indexes MultiViews FollowSymLinks
        AllowOverride None
        Order deny,allow
        Deny from all
        Allow from 127.0.0.0/255.0.0.0 ::1/128
    </Directory>
</VirtualHost>
```

Code 1.5: Sample Apache 2.0 configuration file for a virtualhost

In `apache2`, the configuration of virtual hosts are in the directory /etc/apache2/sites-available. To activate a site (virtual host), you can use the `a2ensite` command:

```
# a2ensite default
# /etc/init.d/apache2 restart
```

There is a respective `a2dissite` command for disabling a site:

```
# a2dissite default
# /etc/init.d/apache2 restart
```

Typically, if you only run one web site on your server, `apache2` uses the default virtual host. The configuration of the default site is in the file /etc/apache2/sites-available/default. After you enable this site, if you look at /etc/apache2/sites-enabled/, you will find that there is a symbolic link called 000-default. Looking at the configuration of the default site you can easily create other virtual hosts.

1.11.3 CGIs

Next, we discuss how CGIs work with `apache2`. A CGI defines a way for a web server to interact with external content-generating programs, which are often referred to as CGI programs or CGI scripts. This is one of the simplest ways of creating dynamic content on your web site. In the case of `apache2`, in order to get your CGI programs to work properly, you will need to have Apache configured to permit CGI execution. In Code 1.5 you can observe the following configuration line:

```
ScriptAlias /cgi-bin/ /var/www/cgi-bin/
```

This tells Apache that any request for a resource beginning with /cgi-bin/ should be served from the directory /var/www/cgi-bin/ and should be treated as a CGI program. For example, if the URL http://localhost/cgi-bin/datecgi.sh is requested, Apache will attempt to execute the file /var/www/cgi-bin/datecgi.sh and return the output. Of course, the file has to exist, be executable and return a correct output (e.g. an HTML file) or `apache2` will return an error message. You can use Code 1.6 for datecgi.sh.

```
#!/bin/sh
echo "Content-type: text/html"
echo
echo "<html> <body>"
echo -n "The current date is "
date
echo "</body> </html>"
```

Code 1.6: Simple CGI script with Bash

In Code 1.7 you have another example of a CGI (in C) that multiplies two numbers.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void){
char *data;
long x,y;
data = getenv("QUERY_STRING");
printf("Content-type: text/html\n\n");
printf("<html><body>\n");
printf("<h1>MULTIPLICATION</h1>\n<hr>\n");
if(data == NULL)
   printf("<P>ERROR: No query string received </P>");
else if(sscanf(data,"x=%ld&y=%ld",&x,&y)!=2)
   printf("<P>ERROR: Invalid Arguments </P>");
else
   printf("<P>The product of x=%ld and y=%ld is z=%ld </P>",x,y,x*y);
printf( "</body></html>\n");
return 0;
}
```

Code 1.7: Simple CGI in C

1.11.4 Modules

To manage modules, Debian-based distros use two directories: /etc/apache2/mods-enabled and /etc/apache2/mods-available. To activate a module, use the `a2enmod` command:

```
# a2enmod userdir
# /etc/init.d/apache2 restart
```

The previous `a2enmod` creates symbolic links in the mods-enabled directory. Likewise, to disable the "userdir" module you can type:

```
# a2dismod userdir
# /etc/init.d/apache2 restart
```

The "userdir" module is quite useful because it gives users a default place to setup their own WEB pages. As a system user, you just create a subdirectory called "public_html" in your home directory and place your files and HTML documents there. You can test this module locally with a browser using the following URL:

```
http://localhost/~username/
```

The "username" is your user and of course you can use an IP address instead of "localhost" to remotely connect to your personal site. If an "access denied" appears in your browser, this might be due to the fact that `apache2` runs using the system user *www-data* and your "public_html" directory is not readable by *www-data*. In general, all the resources on the server must be readable by the user *www-data*.

1.12 Commands summary

Table 1.1 summarizes the commands used within this section.

1.13 Practices

Exercise 1.1– In this exercise, we are going to practice with the WEB service. To do so, start the scenario *www* shown in Figure 1.7 on your physical host (**phyhost**) by typing the following command:

```
phyhost$ simctl www start
```

Table 1.1: *Commands for WWW.*

`firefox`	A WEB browser.
`apache2`	An HTTP server.
`a2enmod`	Enable an Apache 2.0 module.
`a2dismod`	Disable an Apache 2.0 module.
`service`	Start, stop, restart, etc. services (daemons).
`a2ensite`	Enable an Apache 2.0 WEB site (Virtual Host).
`a2dissite`	Disable an Apache 2.0 WEB site (Virtual Host).

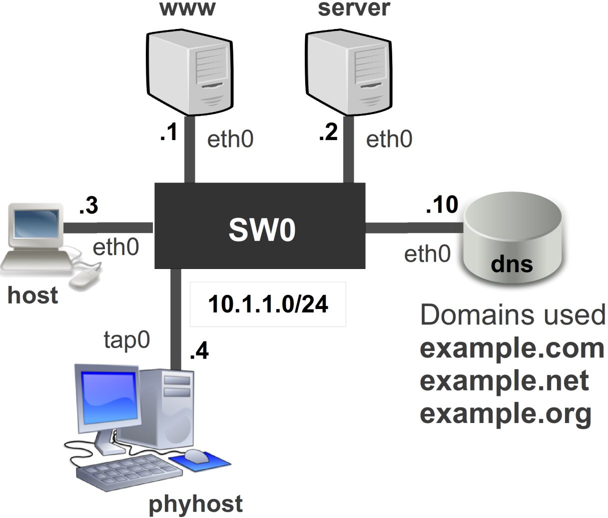


Figure 1.7: Scenario for testing the WEB service.

This scenario starts four virtual machines **host**, **www**, **server** and **dns**. Each virtual machine has also two consoles (0 and 1) enabled. Answer the following questions:

1. Capture the traffic on `tap0` and use a `lynx` browser in the **host** virtual machine to connect to *www.example.com* on port 8080.

   Which is the IP address associated to *www.example.com*?

   Why the browser is not able to establish a TCP connection with the server?

   Describe the DNS and TCP traffic captured.

2. Capture the traffic on `tap0` and repeat the previous experiment, but this time execute a `netcat` in the **www** machine listening on port 8080.

   Which version of HTTP is using the browser?

   Is the connection closed?

   Describe the DNS and HTTP traffic and kill the `netcat` to finish.

3. Capture the traffic on `tap0` and start the `apache2` Web server in the **www** machine. On the **host** machine, execute a `netcat` to connect to the `apache2` that you have just started.

   Over the connection established with `netcat` and using HTTP 1.0, send an HTTP GET request for the resource "/" and another GET request for the resource "/doc.html" .

   Which response do you obtain for each request?

   Is there a resource called doc.html in the **www** server?

   Describe the HTTP traffic captured for each GET request.

4. Configure the `tap0` interface of the physical host (**phyhost**) with the IP address 10.1.1.4/24. After that, ask for "/" and "/doc.html" from the **phyhost** using a `firefox` browser and the IP address 10.1.1.1.

   Describe the HTTP traffic captured.

   Can you use the name *www.example.com* from the **phyhost**? why?

Exercise 1.2– In this exercise we are going to practice with basic HTML content and hyperlinks.

1. Under the directory /tmp of the virtual machine **www** you will find files with images. In **www**, copy these images to a directory called "images" relative to the *DocumentRoot* of the Apache's default site.

   Note. You must create the "images" directory.

   Modify the HTML index of the server and create local hyperlinks to these images.

   Describe how you do it.

2. Start a capture on `tap0`. In the **phyhost** open a `firefox` browser and request for the index page of the **www** machine that you previously modified.

   Describe the HTTP traffic captured. In particular, discuss the GET requests that you observe and the number of connections. To do this analysis easier, in `wireshark`, you can use the option `statistics` ▷ `conversations`, go to the label for TCP and then, use the option `follow stream` for analyzing the data transmitted through each TCP connection.

   Now ask a second time for the index of **www**. Describe how this time HTTP caching works.

   Note. To reproduce the experiment you have to remove the cache of `firebox`. You can do this clicking in **"clear your recent history"** in the menu `Edit▷Preferences▷Privacy` of `firefox`.

3. Remove the cache of `firefox` and decrease its maximum number of persistent connections per server from 6 to 2. For this purpose, use the `about:config` string in the URL of `firefox` and then, search and modify the parameter `network.http.max-persistent-connections-per-server`.

   From `firefox` request for the index page of **www** that you previously modified. Describe the HTTP traffic captured. In particular, comment the number of persistent connections that you observe now and the GET requests through each connection.

   Note. When you finish the exercise, do not forget to set to 6 again the maximum number of persistent connections per server.

4. Execute a `netcat` in the **phyhost** to connect to 10.1.1.1 port 80 redirecting the output of this command to a file called `response.http`. Through the established connection type an HTTP request for the resource `upc1.gif` using HTTP 1.0.

   Explain how you do it.

   Edit appropriately the file `response.http` with `gedit` or `vi` to obtain the original image `upc1.gif`. Save the image with the name `image.gif` and test that it is correct using the command `display`:

   ```
   phyhost# display image.gif
   ```

   You should be able to see an UPC logo.

   After that, repeat the process using HTTP 1.1.

5. Repeat the process to obtain `upc1.gif` but this time use the command `wget`.

   Explain how you do it (consult the manual page of `wget` if necessary).

   Which version of HTTP is used by `wget`?

6. Start `apache2` on the **server** virtual machine. On this machine, modify its HTML index to include HTTP hyperlinks to the images upc1.gif and upc2.gif that are in **www**. Use domain names (not IP addresses) to create these hyperlinks.

   Explain how you do it.

7. Stop the `apache2` server in **www**. Start a capture on `tap0`. From **phyhost**, use a `firefox` browser to request for the index page of **server**. Now, from **host** (virtual machine), use a `lynx` browser to request for the index page of **server** using the short name (*server*) and the fully qualified name (*server.example.com*).

   Describe how you do it and explain the DNS, TCP and HTTP traffic captured.

8. Start `apache2` in the **www** machine. Start a capture on `tap0`. From **phyhost**, use a `firefox` browser to request for the index page of the **server** machine.

   Describe how you do it and explain the HTTP traffic captured.

Exercise 1.3– In this exercise we are going to practice with CGIs and HTML forms using the GET method.

1. Configure your apache server to use the CGI with Bash of Code 1.6.

   Describe the steps of your configuration.

2. Configure your apache server to use the CGI in C of Code 1.7 with an HTML form.

   Describe the steps of your configuration.

Exercise 1.4– In this exercise we are going to practice with the Web service using multiple domains and multiple IP addresses.

1. We are going add the name *www.example.net* in the **dns** server to be translated to the IP address 10.1.1.1. To do so, add an A register in the file */etc/bind/db.example.net* of the **dns** machine and restart `bind`:

   ```
   dns#/etc/init.d/bind9 restart
   ```

   Notice that after this configuration, *www.example.com* and *www.example.net* both are translated to the same IP address (10.1.1.1).

   Try that the configuration is correct with `ping`s from **host**.

2. For the previous domain names, we are going to create and activate two sites (or ”apache virtual hosts“) in `apache2`. To do so, type the following:

   ```
   www# cd /etc/apache2/sites-available/
   www# cp default www.example.com
   www# cp default www.example.net
   ```

   Edit with `vi` */etc/apache2/sites-available/www.example.com* and modify it as follows:

   ```
   DocumentRoot /var/www/com
   ServerName www.example.com

   # Other directives here
   ```

   Edit also with `vi` */etc/apache2/sites-available/www.example.net* and modify it as follows:

```
DocumentRoot /var/www/net
ServerName www.example.net

# Other directives here
```

Now, we need to enable these sites, to do so type:

```
www# a2ensite www.example.com
www# a2ensite www.example.net
```

And reload the configuration of `apache2`:

```
www# /etc/init.d/apache2 reload
```

Generate two different *index.html* files for each domain and put them on the right place. Capture on `tap0` and describe how you test this configuration. Try also to connect directly with the IP address 10.1.1.1.

Discuss the results.

3. Now we are going to test a configuration where the domain www.example.org is translated by the **dns** server to two different IP addresses: 10.1.1.1 and 10.1.1.2. The `bind9` server uses a round robin strategy for translating names with multiple IP addresses (a different IP address for each query in a cyclic way). So, add the A registers that you consider necessary in the file */etc/bind/db.example.org* in **dns** and reload the server. Then, activate this domain in **www** and **server** with the following configuration:

```
DocumentRoot /var/www/org
ServerName www.example.org

# Other directives here
```

Create the same `index.html` file in both web servers, test the configuration and discuss the results.

For what do you think that this configuration is useful?