Experiential Learning of Networking Technologies

Exploring Basics of the HTTP Protocol

Ram P. Rustagi and Viraj Kumar

Abstract

At the undergraduate level, most Computer Networking curricula aim to familiarize students with key networking technologies and standards, with an emphasis on breadth rather than depth of understanding. This is detrimental to students in two ways. First, students obtain a rudimentary or incomplete understanding of inherently complex yet fundamental concepts. Second, students are often unable to translate their limited understanding into diagnosis and action while troubleshooting networks. In this series of articles, we describe a small set of hands-on experiments (which have been iteratively refined over six years) that offer learners opportunities to reflect on their understanding. The feedback collected from our students confirms that this experiential learning helps students gain a lasting understanding of the workings of computer networks. All experiments can be conducted as 1-2 hour exercises in a networking lab, or even at home with minimal setup.

Introduction

The ACM/IEEE Computer Society Joint Task Force [1] has established guidelines for the Computer Science undergraduate curriculum (hereafter, CS2013). All learning outcomes (LOs) specified in CS2013 target one of three levels of mastery: *Familiarity*, *Usage* and *Assessment*. Almost all LOs associated with networking technologies and standards target the lowest-level (*Familiarity*) LO. This is reflective of how the introductory undergraduate course in Computer Networking is typically taught, with an emphasis on breadth rather than depth. We believe that this tradeoff is undesirable and unnecessary for many key Computer Networking concepts, which can be explored in greater depth using carefully designed experiments.

A lab-based environment allows students to learn *experientially* by exploring inherently complex concepts at the depth necessary to understand them, at a pace largely under their own control. This approach uses lab-hours more productively, without requiring additional lecture hours. These experiments though can be done alone, but in our experience, students who work in groups of 2 to 4 typically gain an even better understanding, provided the group discusses their experimental observations, and identifies those observations in line with expectation vs. those that represent “failures” of some kind (along with explanations for such failures).

This series of articles approaches the teaching of computer networks using a set of hands-on experiments, aimed at encouraging students to apply the underlying concepts to realistic situations, to enhance their ability to diagnose and troubleshoot networking problems as professionals.

Understanding the HTTP Protocol

In this first article, we present experiments that allow students to gain a deeper understanding of the basic HTTP protocol, which enables a web client to talk to web server and retrieve web pages. This operation forms the basis of almost every internet-based interaction, and most popular textbooks (e.g., Kurose and Ross [2], Peterson and Davie [3], Forouzan [4]) introduce the topic with illustrative HTTP requests and responses, such as shown below (Table-1 and Table-2). All these textbooks clearly specify the structure of both the request and response messages, but only [2] gently encourages students to dissect the workings of the protocol. We now describe experiments that help students to explore how HTTP headers affect the application behavior in significantly greater detail, to improve their grasp of the concepts. Students who fail to understand these concepts beyond the level of definitions struggle to make efficient use of the HTTP protocol when they develop network applications. The experiments described below require just two machines (laptops connected via a simple network). This could be two laptops connected via a Wi-Fi access point, or directly with an Ethernet cable.

Ram P. Rustagi and Viraj Kumar are with the Department of Computer Science and Engineering, PES University, Bangalore, India
{rprustagi, viraj.kumar}@pes.edu

Table-1: HTTP request message

```
GET /welcome.html HTTP/1.1
Host: myweb.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11;
    rv:48.0) Gecko/20100101 Firefox/48.0
Accept: text/html,application/xhtml+xml,
    application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.7,hi;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

Table-2: HTTP response message

```
HTTP/1.1 200 OK
Date: Sun, 04 Sep 2016 11:29:28 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sun, 04 Sep 2016 11:18:59 GMT
ETag: "1d1-53bacbae6b401-gzip"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 333
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Before defining the experiments and describing how they are conducted, let us briefly review the components of a web page and the HTTP protocol [5]. A web page consists of HTML content together with embedded objects, such as images, videos etc. The client communicates with a web server by sending a HTTP request message and the server replies with the response message. The request message contains the URL of the web page. The URL [6] consists of two parts: the hostname of the server where the web page resides, and the path of web page object on the server. For example, the URL http://myweb.com/welcome.html has myweb.com as the hostname and welcome.html as the path of the web page on the server.

An example of an HTTP request message is shown in Table 1. The first line (called as the request line) has three parts: HTTP method, URL, and the protocol version (`HTTP/1.1`). The request line is followed by number of request headers, each header on a separate line. There is no request data in this request message. An example of corresponding HTTP Repsonse message is shown in Table-2. The first line of the response (called as the status line) consists of 3 parts: Protocol version, the status code (`200`), and a description of the status code (`OK`). The status line is followed by response headers, which are followed by an empty line and then the response data (i.e. the contents of the web page). The status code provides information to the client on how to interpret the response, and belongs to one of the following five categories:

i. 1xx: Informational
ii. 2xx: Success
iii. 3xx: User action needed
iv. 4xx: Error on user part
v. 5xx: Error on server part

The status code consists of 3 digits, first digit identifies the category and remaining two digits (xx) correspond to specific response in the category. The category `1xx` indicates that request has been received and is being processed. The category `2xx` indicates that request is successfully processed and the required contents are part of

the response. The most common status value used is `200` implying success, i.e. the complete requested content has been sent. The category `3xx` indicates that client needs to take additional action for the request to be completed. This implies that the request cannot be served in its current form. The most common values in this category are 301 and 302, which informs the client that current URL has been changed to a new URL, and that the client needs to make a new request to using this new URL. The category `4xx` implies that client request has an error which cannot be served. Hence, the client needs to correct the error. The most common value in this category is 404, which corresponds to invalid URL (i.e., the content corresponding to the specified path does not exist). The last category `5xx` informs the client that the server cannot serve the request because something unusual happened at the server side while serving the request. The most common error in this category is 500 which corresponds to `Internal Server Error`. In this article, we will explain simple experiments where students trigger and observe status codes 200, 400, 403, and 404.

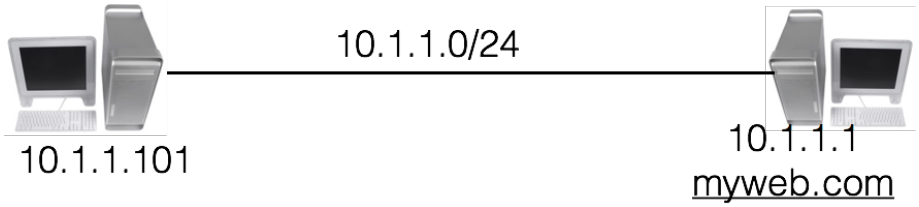

Figure 1 Basic setup for understanding the HTTP protocol

Currently, the most commonly used HTTP protocol version in use is 1.1, and thus we focus on the working of the HTTP protocol for version 1.1. In our experimental learning, we use the setup shown in **Error! Reference source not found.**, consisting of one client and one server. In our setup, both the web server runs Ubuntu OS with the Apache web server [7] software. The client is also an Ubuntu system, but Windows and Mac based systems can also be used for the client.

Suppose the IP address of our server is 10.1.1.1 and that of our client is 10.1.1.101. To access our server with the name `myweb.com` while avoiding DNS-related issues, we simply add the entry "`10.1.1.1 myweb.com`" in the `/etc/hosts` file of the client. We assume that the Apache web server's root directory (DocumentRoot, from where web pages are served) is defined as `var/www/html`. In this directory, create a simple HTML page named `welcome.html` (as shown in Table-3).

Table-3: A simple HTML web page

```
<html>
    <head>
          <title>Welcome Page</title>
    </head>
    <body>
          <h1>Welcome to HTTP Learning</h1>
              Welcome to experiential learning of HTTP protocol.
    </body>
</html>
```

Basic Working of the HTTP Protocol

As an example, let us choose Firefox browser although our experiment works equally well with Chrome. To study the working of the HTTP protocol, open the web console in Firefox browser (right click and select "Inspect element") to bring up the console inspection window in the browser (Figure 2). Select the Network tab to view the HTTP protocol communication that takes place when accessing a web page. Enter the URL http://myweb.com/welcome.html in the client browser to access the web page just created on the server. The content of the web page will be displayed in the browser window, and the Network tab will show the information

for this URL (status 200, HTTP method GET, and URL (File) welcome.html). Selecting this line will show the HTTP headers, and selecting the "Raw headers" tab will show the complete request and response HTTP headers (Figure 3). These headers provide a good amount of meta-data information about the page being displayed, such as:

i. Date and Time when request is served (`Date`)
ii. The date and time when web resource was last modified (`Last-Modified`)
iii. Identification of web server software, the OS on which it runs (`Server`)
iv. If web page content was compressed and compression algorithm (`Content-Encoding`)
v. Total size in bytes of the web page content
vi. The type of content to help browser in displaying it properly (`Content-Type`). The value `Text/html` tells the browser to format it as per HTML tags before rendering it in the browser window. `Text/plain` will cause the browser to display the content as it is.

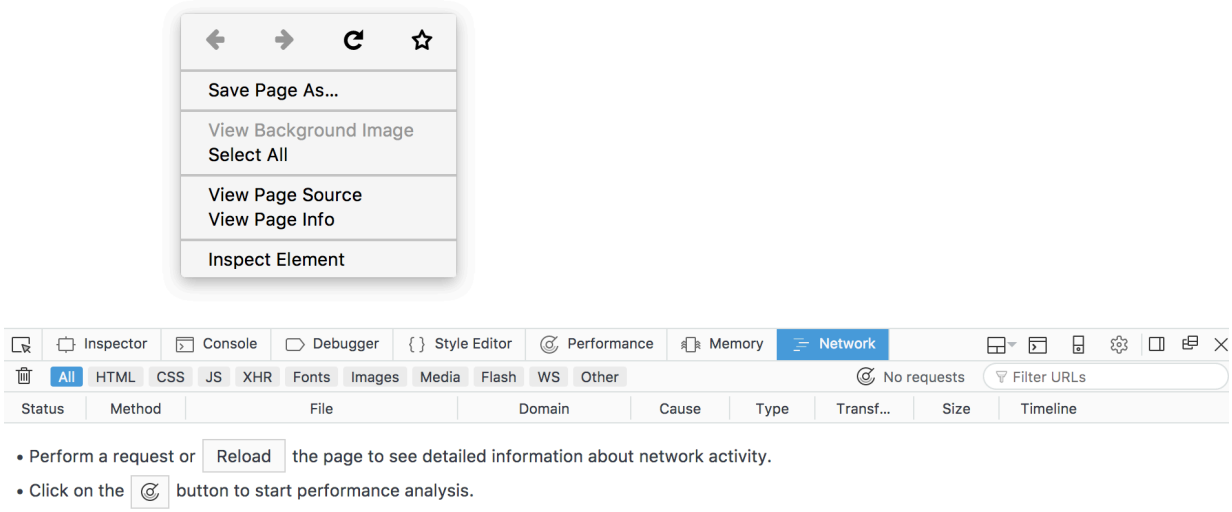

Figure 2: Network console in Firefox browser

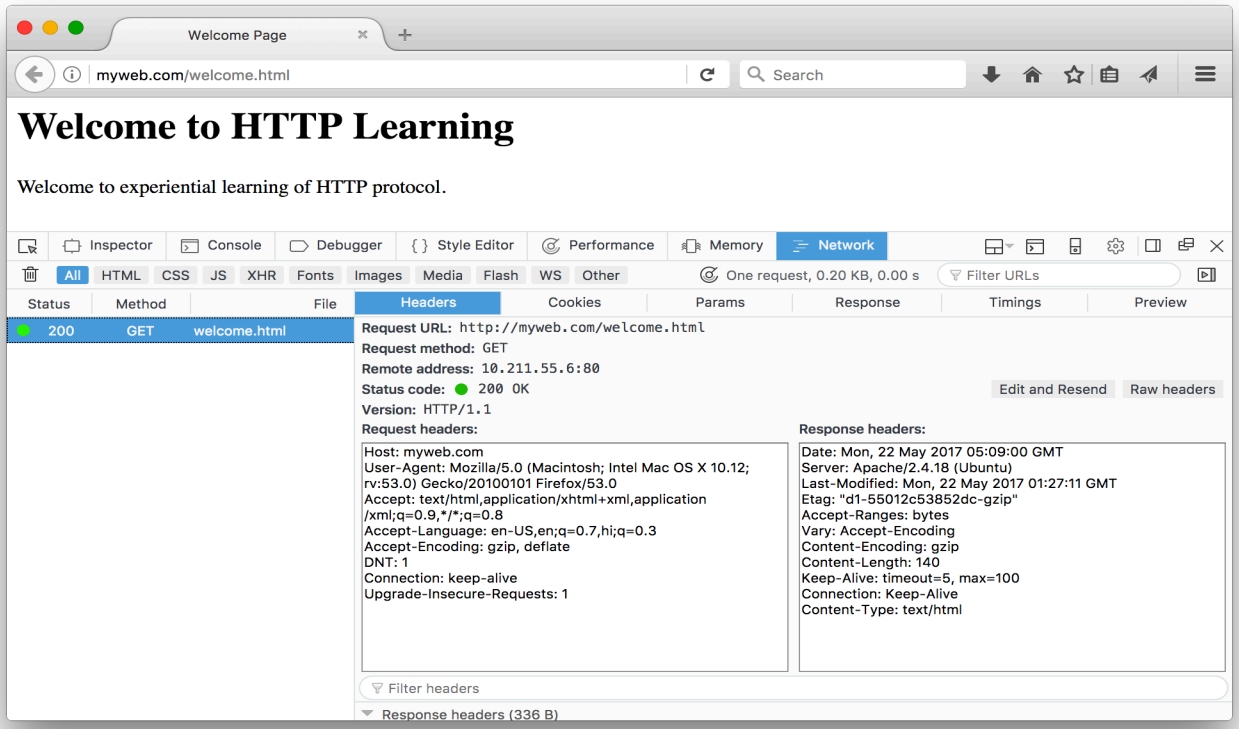

Figure 3 Request and Response HTTP headers

The content rendering by browsers depends upon the header `Content-Type`. To understand how this works, rename the file `welcome.html` to `welcome.txt` in the `DocumentRoot` directory (`/var/www/html`) on the web server. In the browser window, access the modified URL `http://myweb.com/welcome.txt`. Observe that the browser displays the content of the web page in textual form with all of its HTML tags. In the web console window, observe that the response header '`Content-Type`' has the value `Text/plain`. Thus, the browser displays the very same content differently even though both requests are served successfully with status code 200. To further study the impact of this header, place an image file (e.g., `picture.jpg`) in the web server's DocumentRoot directory, and create a copy of this file named `picture.txt`. Accessing the URL `http://myweb.com/picture.jpg` shows the picture correctly, whereas accessing the URL `http://myweb.com/picture.txt` displays gibberish in the browser window. The crucial lesson here is that the way the browser renders the content is mainly determined by the response header `Content-Type`. Thus, when accessing web, if content is being displayed incorrectly (e.g. gibberish) on the web browser, a possible cause is that the application running on the web server is setting this header incorrectly.

Understanding Basic Errors

Let us look at some other most common errors seen by a user when accessing the web. These pertain to the cases when the URL specifies a a) non-existent web resource (error code 404), b) forbidden or inaccessible content (error code 403), or c) invalid request parameters such as improper request headers, non-understandable requests (error code 400).

When a client makes a request for a non-existent web resource, the web server does not find the resource in its repository of contents and thus responds with a status code of 404 (`Not Found`). From the browser, for example,

type the URL `http://myweb.com/unknown.html` and the browser will indicate that the content is "**Not Found**". The HTTP response line will contain the status value 404, and the description will also be "`Not Found`".

Occasionally, when user accesses a web resource, the web server responds with a web page that says "**Forbidden"**. This error occurs when the web resource does exist, but the web server does not have the necessary permission to read the content of the requested resource. A simple case on Ubuntu occurs when the web server, running with the user id '`www-data`', does not have read permissions for a static resource. To demonstrate this, create a file `classified.html` in the web server `DocumentRoot` and remove the read permission for this file (e.g., using the command '`chmod 500 /var/www/html/classified.html`') using an account that is not www-data. A listing of files in the directory `/var/www/html` would look as follows (Table-4):

Table-4: List of files in DocumentRoot

```
rprustagi@ubuntu:/var/www/html$ ls -l
total 8
-rw------- 1 rprustagi rprustagi 152 May 23 10:51 classified.html
-rw-rw-r-- 1 rprustagi rprustagi 209 May 22 06:57 welcome.html
rprustagi@ubuntu:/var/www/html$
```

When the client browser tries to access http://myweb.com/classified.html, the web server is blocked from reading the contents of the file, and will therefore generate a Forbidden response with status code 403. The status code and corresponding headers can be verified from the Network tab of the web console. To fix this error, ensure that the file `classified.html` has read permissions for `www-data`. Reloading the URL will now result in a response with status 200, and the contents of classified.html will be visible. A more realistic scenario where this occurs is when the response comes from a web application (e.g., a business logic application written in Python, Perl, PHP, C/C++, Java, etc.) which determines that the requesting user is not authorized to access the resource.

Occasionally, when using a client application where web requests are made directly by the application and not by the browser itself (e.g., by an java applet or an Active-X component), there is a possibility that the generated request does not follow the syntax and semantics of the HTTP protocol precisely, thereby inadvertently creating an erroneous request. When such a request is received by the web server, it does not understand the request fully and therefore responds with an error with status 400 (`Bad Request`). Generating such requests from within the browser is difficult, because browsers are designed to always send properly formatted request. To experience such an error, we need to use more basic tools to generate erroneous requests. For this experiment, we will use the `telnet` command line tool, which is available by default on any Linux distribution (and Windows and Mac as well). From the command line terminal, type the command `telnet myweb.com 80` followed by a header line `Host myweb.com` (Table 5). This header has a deliberate syntax error (the colon between `Host` and `myweb.com` is missing). The web server encounters this error while parsing the request header, and thus responds with the error status 400 `Bad Request` as shown in Table 6.

Table-5 Invalid HTTP request header

```
telnet myweb.com 80
Trying 10.1.1.1...
Connected to myweb.com.
Escape character is '^]'.
Host myweb.com
```

Table 6 Web server response with status 400

```
HTTP/1.1 400 Bad Request
Date: Tue, 23 May 2017 07:15:12 GMT
Server: Apache/2.4.18 (Ubuntu)
Content-Length: 301
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.<br /></p>
<hr>
<address>Apache/2.4.18 (Ubuntu) Server at 127.0.1.1 Port 80</address>
</body></html>
```

To resolve such an error, the user needs to ensure that all the request headers and parameters follow the proper syntax. In this example, when user adds the missing colon (:) between the header name and value, the web server responds back with the proper response and status (200).

Summary

We have described an experimental setup where students configure an Apache webserver to study basic working of the HTTP protocol and experience both successful and erroneous responses that users encounter in real scenarios. We have also provided insights into how to fix such errors. We believe that this hands-on experience improves the quality of student understanding of the underlying concepts. In the next article, we will discuss how to make use of the most commonly used headers that deal with caching, which helps students design efficient web applications. This also helps students debug such applications more effectively, reducing the time for application development.

References

[1] ACM/IEEE Computer Society, "Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science", Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, ACM, New York, NY, 2013.

[2] Kurose, Ross, "Computer Networks: A Top Down Approach" 7th edition, Pearson Education Inc, 2016.

[3] Peterson, Davie, "Computer Networks: A Systems Approach" 5th edition, Morgan Kaufmann, 2012

[4] Forouzan, "Data Communications and Networking", 4th edition, The McGraw-Hill, 2011.

[5] RFC 2616, "Hyper Text Transfer Protocol – HTTP/1.1", Network Working Group, Request for Comments 2616. Fielding, Gettys et al. June 1999

[6] RFC 791, "Internet Protocol (IP), DARPA Internet Program Protocol Specifications, Request for Comments 791. ISI, Univ of Southern California, September 1981.

[7] Apache Web Server https://httpd.apache.org/docs/2.4