Experiential Learning of Networking Technologies

HTTP Protocol Mechanisms for High Performance Applications

Ram P. Rustagi and Viraj Kumar

Abstract

HTTP is the most widely used protocol today, and is supported by almost every device that connects to a network. As web pages continue to grow in size and complexity, web browsers and the HTTP protocol itself have evolved to ensure that end users can meaningfully engage with this rich content. This article describes how HTTP has evolved from a simple request-response paradigm to include mechanisms for high performance applications.

Introduction

The original HTTP protocol (commonly referred to as version 0.9) was designed around simple request and response communications running on top of TCP/IP, and was intended to support the transfer of simple HTML documents. Each request consists of a one-line ASCII character string (GET followed by the path of the resource being requested) terminated by CR/LF, and the response is also an ASCII character stream. There are no headers containing meta-information about requests and responses. A new TCP connection is opened with each request, and closed after the response is received.

This protocol soon evolved to support additional features, including: serving non-HTML documents (such as images), enabling clients and servers to negotiate the type of content to be exchanged, its size, its encoding, etc. These features are formally captured in the HTTP/1.0 Informational RFC 1945 [1], which describes the common usage of the HTTP protocol between clients and servers. It clarifies that HTTP is defined as "*an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems*". The first formal standard for HTTP/1.1 was defined in RFC 2068 [2], which was later superseded by RFC 2616 [3]. This is the standard most commonly used today, even though the protocol has evolved to the new HTTP/2 standard defined in RFC 7540 [4].

The growing complexity of web pages is a key driving force behind the evolution of the HTTP standard. Loading a typical web page today accesses more than 100 subsidiary URLs (including 20+ JavaScript files, 7+ CSS files and 50+ images) totaling nearly 3MB of data [9]. Thus, it is interesting to ask the following questions: How does the server manage this significant per-client load without exhausting its resources? Since some of this data is repetitive, how does the client avoid redundant downloads (thereby reducing load times and network usage, as well as reducing server load)?

To address these questions, this article will show how the HTTP protocol has evolved to include multiple mechanisms to improve the application performance delivery. For web application developers, factors such as network performance and hardware configuration are not controllable. Hence, developers must consider the following aspects while optimizing web network performance:

i. Make use of HTTP persistent connections (using the Keep-Alive header). This avoids one round trip delay, thereby improving performance.
ii. Make efficient use of caching as supported by the HTTP protocol.
iii. Make use of chunk-transfer-encoding to display content as it is received, rather than waiting to receive the entire content before displaying it.

We will focus on these three techniques, although there are several other ways to improve performance including compression, minimizing the number of HTTP redirects (more than 80% of websites use

Ram P. Rustagi (rprustagi@cavisson.com) is working with Cavission Systems Inc, and Viraj Kumar (viraj.kumar@pes.edu) is with the Department of Computer Science and Engineering, PES University, Bangalore

redirects [9]), reducing the number of DNS resolutions, and making use of Content Delivery Networks (CDNs) to locate large-sized content (typically, multimedia) closer to users, thereby reducing end-to-end delay.

Experimental Setup

Following the experiential approach taken in first article of this series (June 2017 issue of ACC.digital [11]), we describe the setup for experiments that will illustrate these concepts. The basic setup is shown in Figure 1, consisting of a web server machine (running Apache web server [7]) named myweb.com with IP address 10.1.1.1, and a client machine with IP address 10.1.1.101 running a web browser. The two machines (desktops or laptops) are connected via a simple network (e.g., a Wi-Fi access point or directly with an Ethernet cable)[1].

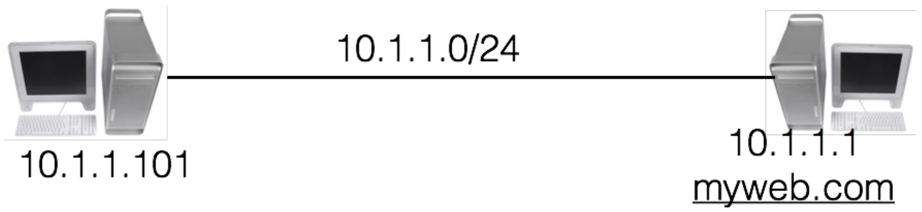


Figure 1: Simple setup

For our experiments, we need to observe several details of the TCP connections established and terminated between these two machines. To do so, we will make use of the most widely used network diagnostic tool: Wireshark [5]. This utility allows us to capture and analyze each byte of every packet that traverses the network. To gain a detailed understanding of Wireshark, we refer readers to the user guide [5].

Non-persistent and Persistent Connections

The default approach in both HTTP v0.9 and HTTP v1.0 protocols is to establish separate TCP connection for each URL access. Such connections are called *non-persistent*. Although non-persistent connections are simple, they have three main disadvantages:

i. A new TCP connection must be established for each URL access. This results in an additional round trip delay for setting up a TCP connection.
ii. For each new TCP connection, the web server must allocate resources (buffers, TCP state variables, etc.) which results in significant overhead for the web server.
iii. Most web page today consists of multiple objects, so the web server needs to concurrently serve several TCP connections for a single client web browser.

In HTTP v1.1, the default[2] is to use *persistent* connections. Here, a single TCP connection is reused to serve multiple HTTP requests, thereby overcoming the disadvantages listed above. This change has led to such a significant performance improvement that it has been back ported to HTTP 1.0. The HTTP protocol defines two headers (`Connection:` and `Keep-Alive:`) that identify whether a web request is using persistent or non-persistent connections. The header line `Connection: Close` means that the TCP connection should be closed immediately after the web request is served, implying

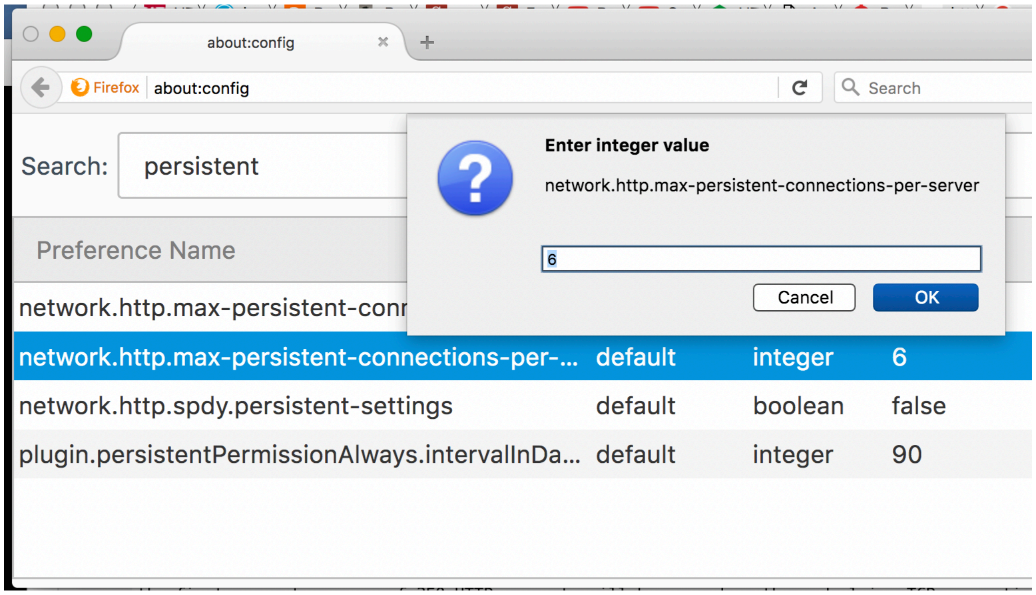
[1] these IP addresses in your experimental setup are likely to be different and should be used appropriately.

[2] All modern browsers use persistent connections by default.

non-persistence. In contrast, the header line `Connection: Keep-Alive` means that the underlying TCP connection should remain open for serving web requests in the future, implying persistence. To establish a persistent connection, both the client and the web server should support such connections. If either one does not support persistence, then the connection is non-persistent.

A persistent TCP connection should not remain open forever. For instance, when the client navigates to a different web site, keeping the underlying TCP connection open will lead to unnecessary resource consumption on the original web server, causing its performance to degrade. Further, the web server will quickly reach the limit on the maximum number of open TCP connections it can support, after which it will no longer be able to cater to new requests. Thus, there is a need to define a timeout after which a persistent TCP connection needs to be closed. Further, it is prudent to impose a limit on the total number of web requests that can be served on a given persistent TCP connection. To understand why, suppose a web server has a bug that leads to resource inconsistency (e.g., a memory leak) while serving a web request. The greater the number of web requests served on a TCP connection, the greater the resource leak. Hence, permitting several web requests on one TCP connection could exhaust web server resources. The header `Keep-Alive` has two parameters (*`timeout`* and *`max`*) to handle these two concerns. For example, the header line `Keep-Alive: timeout=100, max=250` states that the underlying TCP connection can be kept idle for up to 100 seconds. If no new HTTP requests are received for 100 seconds after serving the last request, the underlying TCP connection will be closed. Further, the parameter `max=250` states that after the first request, a maximum of 250 additional HTTP requests can be served on the underlying TCP connection. If another (251st) request is sent, the web server sends the HTTP header `Connection: Close` to indicate that the TCP connection is now closed. To service the next (252nd) request from the same web client, a new TCP connection will be established before web content can be exchanged. Further details about persistent and non-persistent connections can be found in the textbook by Kurose and Ross [6].

Although RFC 2616 [3] suggests that only two concurrent persistent connections should be established between a client and a web server, modern browsers allow a greater number of parallel connections. The default values for Firefox and Google Chrome browsers is 6, whereas IE10 sets the default value at 8 and Edge (Windows 10) sets the default number of concurrent connections to a website even higher. Among these browsers, only Firefox allows users to easily change the number of concurrent persistent connections. Enter *`about:config`* in the URL bar search (this triggers a warning), find the field *`max-persistent-connections-per-server`,* and double-click on it to change the value as shown in Figure 2.



Enter integer value

network.http.max-persistent-connections-per-server

6

```
network.http.max-persistent-conn
network.http.max-persistent-connections-per-...   default   integer   6
network.http.spdy.persistent-settings             default   boolean   false
plugin.persistentPermissionAlways.intervalInDa... default   integer   90
```

Figure 2: Configuring the maximum number of persistent connections per server

Experimenting with Non-persistent and Persistent Connections

To gain hands-on experience with persistent connections, we will configure the Apache web server to understand how web servers support such connections. The Apache web server's configuration directives (which specify how the web server will process web requests) are typically defined in the file `/etc/apache2/apache2.conf`[3] on an Ubuntu Linux installation. These directives have the format "*Name-of-directive Value*". The directive `KeepAlive On`, which is the default configuration at the time of installation, tells the web server to allow persistent connections. To configure Apache to use non-persistent connections, edit the configuration file by changing this directive to `KeepAlive Off`. The two directives discussed below are applicable only for persistent connections (i.e., when directive `KeepAlive On` is set). The directive `KeepAliveTimeout` N specifies the maximum number of seconds N that the web server can wait on a persistent connection for next web request to arrive before closing the TCP connection. The default value of N is 5. Similarly, the directive `MaxKeepAliveRequests` M specifies that the web server can serve a maximum of M HTTP requests on a single TCP connection. On the first request served by the server, its response header will contain the value `Keep-Alive: max=`*M*`, timeout=`*N*, and in subsequent requests the value of `max` will keep decreasing by one. In the last request, the web server will set the response header to `Connection: Close`.

For our first experiment, create a web page *pictures.html* with several embedded URLs such as images. The source code for this page is shown in Table 1. Deploy this page on the web server myweb.com (i.e. 10.1.1.1) in its document root (typically in the directory */var/www/html*). Choose ten image files *img-01.jpg, img-02.jpg,…, img-10.jpg* which are embedded in the web page *pictures.html* and place these within the document root in a folder called *img*.

Table 1: A simple HTML web page with embedded images

```
        <!DOCTYPE html>
        <html> <body>
```

[3] The location of this configuration file may vary depending upon the installation setup.

```
        <h1>Image Gallery </h1>
        <img src="img/img-01.jpg" />
        <img src="img/img-02.jpg" />
        <img src="img/img-03.jpg" />
        <img src="img/img-04.jpg" />
        <img src="img/img-05.jpg" />
        <img src="img/img-06.jpg" />
        <img src="img/img-07.jpg" />
        <img src="img/img-08.jpg" />
        <img src="img/img-09.jpg" />
        <img src="img/img-10.jpg" />
        </body></html>
```

On the web server, edit the apache configuration file */etc/apache2/apache2.conf*, and set the directive as `KeepAlive Off`. Start the Wireshark program using sudo (i.e.root privileges), either on the client machine or on the web server. Select the appropriate network interface (e.g. *eth0* or *enp0s5*), define the capture filter to capture only TCP SYN packets from the client browser to web server (e.g. `tcp[tcpflags] & (tcp-syn) != 0 and dst myweb.com`[4] `and port 80`) as we are interested in observing how many TCP connections are established when the web page *picture.html* is served. (Note: use the IP Address of web server as per your setup being used). In the browser access the web page 'http:/myweb.com/pictures.html' and study the packets captured in the wireshark window. (Note: to ensure that hostname *myweb.com* is resolved to your web server IP address, make an entry in */etc/hosts* files in your browser and map it to your web server. Once the web page is displayed in the browser (Figure 3), stop the capture in wireshark window. The packet captures will be like as shown in Figure 4. The web page *pictures.html* has 10 embedded images and thus to display the page, a total of 11 URLs will be accessed (1 for main url *pictures.html* itself and 10 for images). For non-persistent connections, the browser should establish a total of 11 TCP Connections. The Figure 4 shows 11 SYN packets (indicating starting of a TCP connection) which is on expected lines.

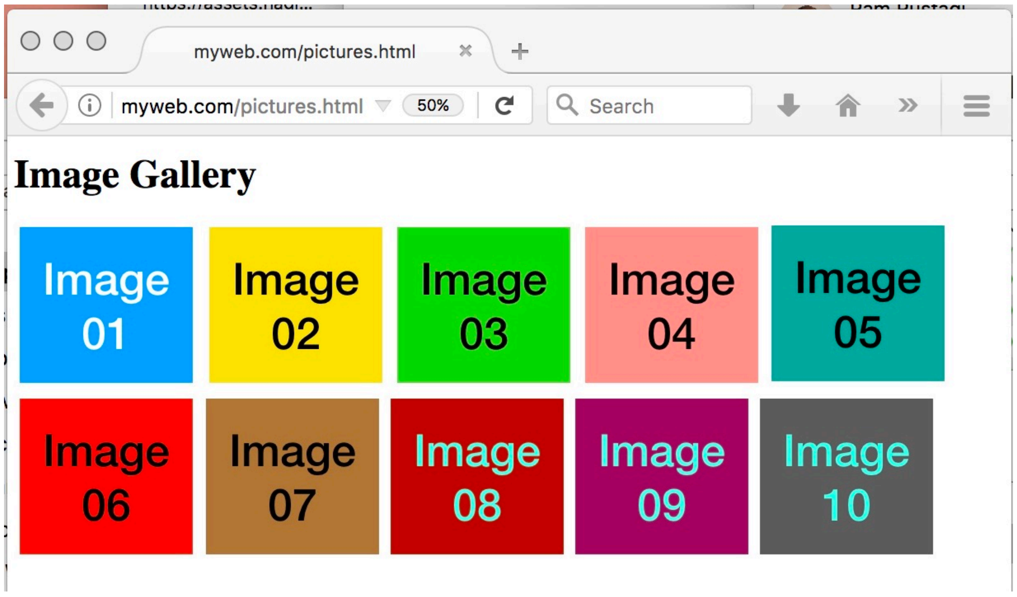
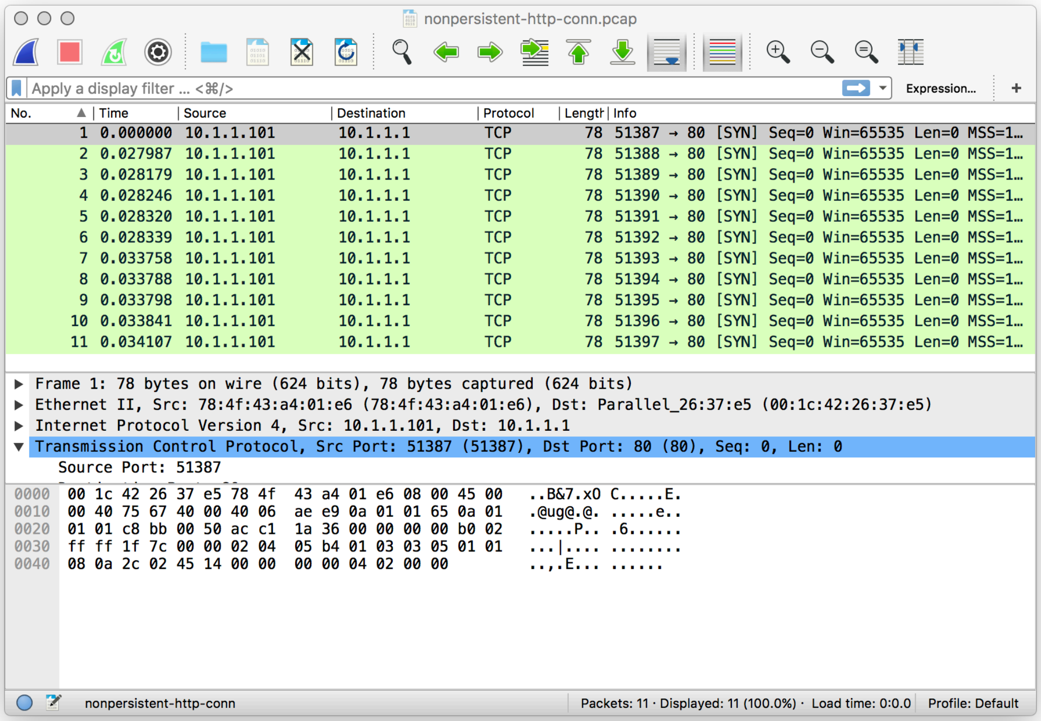


Figure 3: Web page with 10 embedded images

Can specify the IP Address 10.1.1.1 as well in place of myweb.com



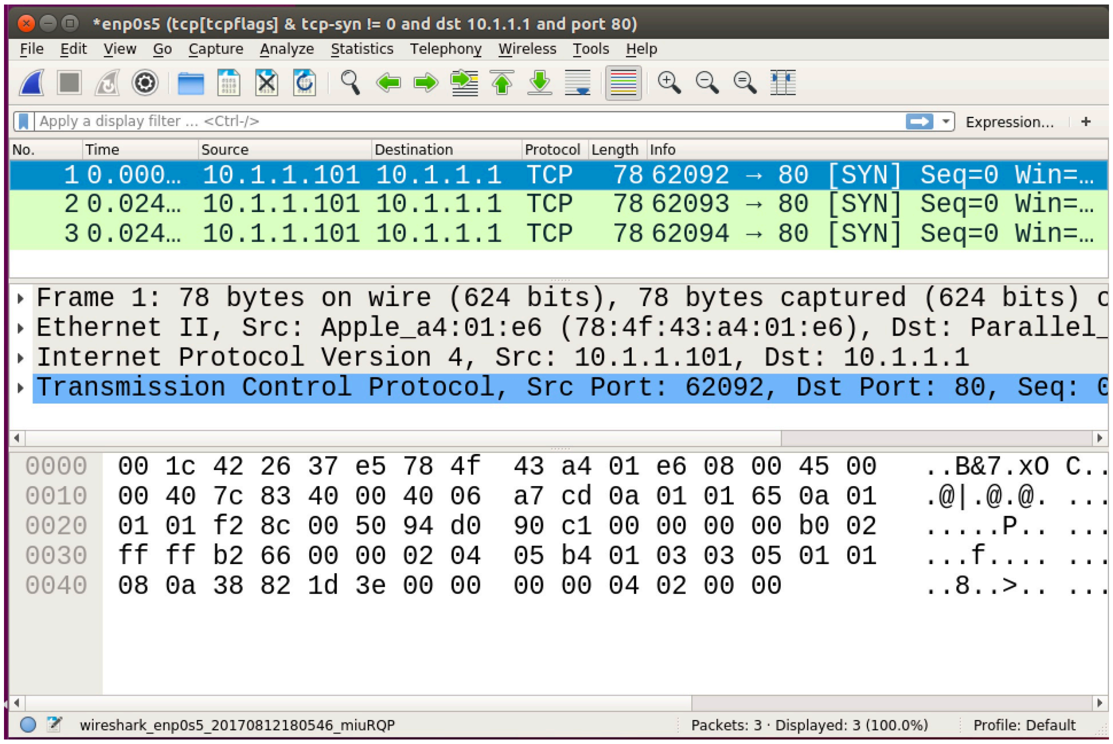
```
nonpersistent-http-conn.pcap

Apply a display filter ... <⌘/>

No.   Time      Source       Destination  Protocol Length Info
    1 0.000000  10.1.1.101   10.1.1.1     TCP      78 51387 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    2 0.027987  10.1.1.101   10.1.1.1     TCP      78 51388 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    3 0.028179  10.1.1.101   10.1.1.1     TCP      78 51389 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    4 0.028246  10.1.1.101   10.1.1.1     TCP      78 51390 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    5 0.028320  10.1.1.101   10.1.1.1     TCP      78 51391 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    6 0.028339  10.1.1.101   10.1.1.1     TCP      78 51392 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    7 0.033758  10.1.1.101   10.1.1.1     TCP      78 51393 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    8 0.033788  10.1.1.101   10.1.1.1     TCP      78 51394 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
    9 0.033798  10.1.1.101   10.1.1.1     TCP      78 51395 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
   10 0.033841  10.1.1.101   10.1.1.1     TCP      78 51396 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…
   11 0.034107  10.1.1.101   10.1.1.1     TCP      78 51397 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1…

▶ Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
▶ Ethernet II, Src: 78:4f:43:a4:01:e6 (78:4f:43:a4:01:e6), Dst: Parallel_26:37:e5 (00:1c:42:26:37:e5)
▶ Internet Protocol Version 4, Src: 10.1.1.101, Dst: 10.1.1.1
▼ Transmission Control Protocol, Src Port: 51387 (51387), Dst Port: 80 (80), Seq: 0, Len: 0
    Source Port: 51387

0000  00 1c 42 26 37 e5 78 4f  43 a4 01 e6 08 00 45 00   ..B&7.xO C.....E.
0010  00 40 75 67 40 00 40 06  ae e9 0a 01 01 65 0a 01   .@ug@.@. .....e..
0020  01 01 c8 bb 00 50 ac c1  1a 36 00 00 00 00 b0 02   .....P.. .6......
0030  ff ff 1f 7c 00 00 02 04  05 b4 01 03 03 05 01 01   ...|.... ........
0040  08 0a 2c 02 45 14 00 00  00 00 04 02 00 00         ..,.E... ......

nonpersistent-http-conn     Packets: 11 · Displayed: 11 (100.0%) · Load time: 0:0.0     Profile: Default
```

Figure 4: Capture of TCP SYN requests for page with embedded images

Experiencing Persistent connections with browser default configuration

In this experiment, we will study use of persistent connections, which is the default behaviour of any browser. On the web server configuration file (*/etc/apache2/apache2.conf*), restore the directive `Keep-Alive: on`, and define the other two related directives as `MaxKeepAliveRequests 100`, (initially configured default value is 100) and `KeepAliveTimeout 5`. Restart the web server (`sudo service apache2 restart`), and start the new capture with the capture filter corresponding to TCP SYN (`'tcp[tcpflags] & tcp-syn != 0 and dst myweb.com and port 80'`) on the wireshark. This capture filter wil will only show those SYN packets that are sent by browser client to the web server. (Note: these can also be seen on terminal window with *tcpdump* utility program (e.g. `sudo tcpdump -n -i eth0 -ttttt 'tcp[tcpflags] & tcp-syn != 0 and dst myweb.com and port 80'`).

Using the firefox browser, set the number of concurrent persistent connections to 3 (set the value as shown in Figure 2). Restart the capture in wireshark, and in the Firefox browser, reload the url *http://myweb.com/pictures.html*. The wireshark will show only 3 SYN packets (Figure 5) being sent from client browser to the web server. There are only 3 SYN packets sent by the client browser even though all the actual 11 URLs are accessed. (Note: To analyze the complete packet exchange between browser and web server, use the capture filter as `'host myweb.com and port 80')`. Out of these 3 TCP Connections serving 11 web requests, two TCP connection would have served 4 web request each and one TCP connection would have served 3 web requests. It is possible that these 11 request may not be fairly distributed among these 3 TCP connections. To study the impact of the directive `Timeout 5`, refresh the browser page or reload the page within 4 seconds after the initial rendering of the web page. In the wireshark capture window capturing only TCP connection requests (SYN packet), no new packets will be captured. Keep refreshing the browser page few times, but within less than 4 seconds and there will be no new TCP connection requests and thus no new packets will be shown on wireshark capture window. This is because when the connection keep alive timeout is kept to 5 seconds, and page is refreshed in less than 4 seconds, the connection is still alive and new web page access happens on the existing connection and no new connection is setup. To see the actual impact of this timeout, refresh the browser windows after 6 seconds. This will ensure that no activity has taken place in the TCP connections serving HTTP Request and timeout would be applicable and thus on this refresh after 6 seconds, all the 3 TCP Connections will be re-established again and same can be verified in wireshark capture.

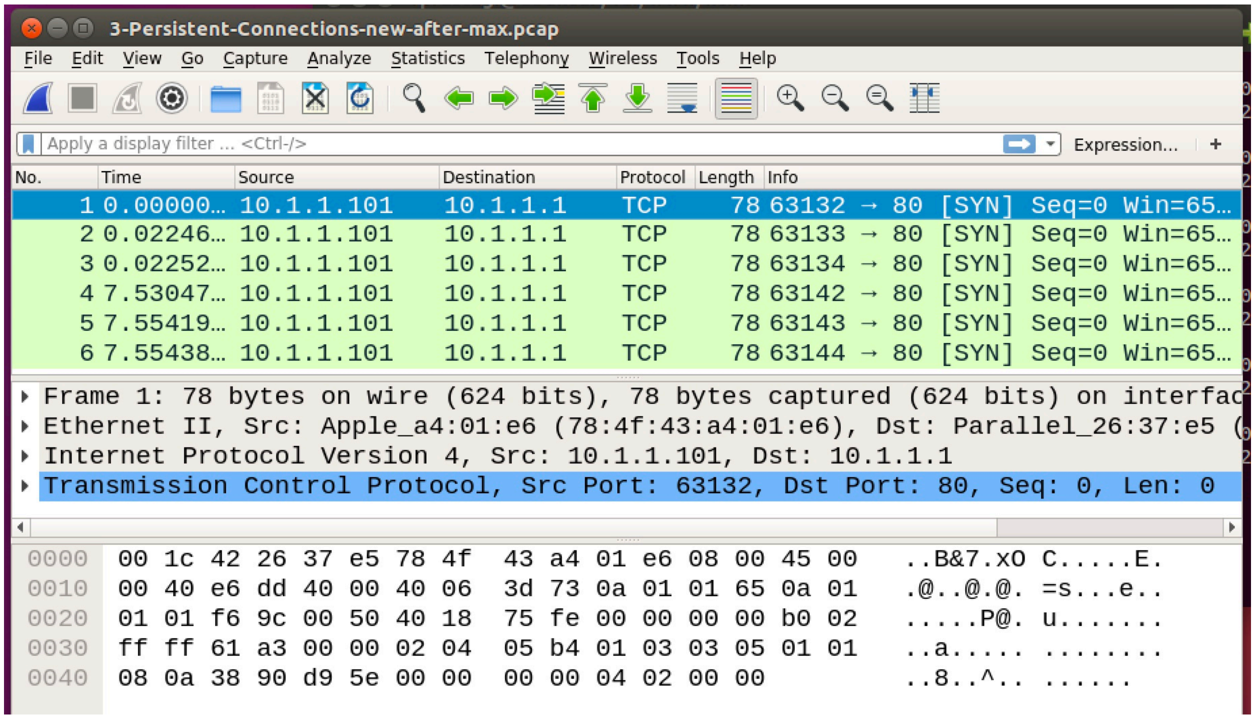

*enp0s5 (tcp[tcpflags] & tcp-syn != 0 and dst 10.1.1.1 and port 80)
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
wireshark_enp0s5_20170812180546_miuRQP
Packets: 3 · Displayed: 3 (100.0%)
Profile: Default

```
No. Time     Source     Destination Protocol Length Info
  1 0.000…   10.1.1.101 10.1.1.1    TCP      78 62092 → 80 [SYN] Seq=0 Win=…
  2 0.024…   10.1.1.101 10.1.1.1    TCP      78 62093 → 80 [SYN] Seq=0 Win=…
  3 0.024…   10.1.1.101 10.1.1.1    TCP      78 62094 → 80 [SYN] Seq=0 Win=…

▸ Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) c
▸ Ethernet II, Src: Apple_a4:01:e6 (78:4f:43:a4:01:e6), Dst: Parallel_
▸ Internet Protocol Version 4, Src: 10.1.1.101, Dst: 10.1.1.1
▸ Transmission Control Protocol, Src Port: 62092, Dst Port: 80, Seq: 0

0000  00 1c 42 26 37 e5 78 4f  43 a4 01 e6 08 00 45 00   ..B&7.xO C..
0010  00 40 7c 83 40 00 40 06  a7 cd 0a 01 01 65 0a 01   .@|.@.@. ...
0020  01 01 f2 8c 00 50 94 d0  90 c1 00 00 00 00 b0 02   .....P.. ...
0030  ff ff b2 66 00 00 02 04  05 b4 01 03 03 05 01 01   ...f.... ...
0040  08 0a 38 82 1d 3e 00 00  00 00 04 02 00 00         ..8..>.. ...
```

Figure 5: Packet capture when persistent connections=3

To analyze the maximum number of HTTP requests that can be served on one TCP Connection, edit the apache configuration file and define the directive value as `MaxKeepAliveRequests 11`. Restart the web server (`sudo service apache2 restart`). Restart the wireshark capture and reload/refresh the web page in the browser. The firebox browser is still configured to use 3 persistent connections. Reload the webpage http://myweb.com/pictures.html in the browser and wireshark capture will show output of 3 TCP connections setup. Refresh this web page every 1 or 2 seconds. On the 4th refresh of the web page, wireshark will show setup of another 3 new TCP Connections as shown in Figure 6. In this capture, first 3 connections are started at time `00:00:00.000000s`, `00:00:00.021872`, and `00:00:00.022212` corresponding to first 3 connections. On 4th successive refresh, the 3 new connections are started after about 7 seconds and their connection start times are `00:00:07.53047s`, `00:00:07.55419s`, and `00:00:07.55438s`. To understand this behavior, note that web server is configured to serve max of 11 HTTP requests (from the directive `MaxKeepAliveRequests`

`11)`. Each web page results in 11 URL access. Thus, after serving 11 new web requests on a TCP connection, the connection will be closed and a new TCP Connection will be established. Since initially firefox browser starts with 3 TCP Connections for first 3 web access, after serving other 33 web requests (a total of 1+11 requests on each TCP connection), the connection will be closed and new connection will be established. Note: It is possible that distribution of 33 web requests on 3 TCP connections may not be uniform and in such cases new 3 connections may not be established around same and may have noticeable time lag.


3-Persistent-Connections-new-after-max.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No. Time        Source      Destination Protocol Length Info
  1 0.00000...  10.1.1.101  10.1.1.1    TCP      78 63132 → 80 [SYN] Seq=0 Win=65...
  2 0.02246...  10.1.1.101  10.1.1.1    TCP      78 63133 → 80 [SYN] Seq=0 Win=65...
  3 0.02252...  10.1.1.101  10.1.1.1    TCP      78 63134 → 80 [SYN] Seq=0 Win=65...
  4 7.53047...  10.1.1.101  10.1.1.1    TCP      78 63142 → 80 [SYN] Seq=0 Win=65...
  5 7.55419...  10.1.1.101  10.1.1.1    TCP      78 63143 → 80 [SYN] Seq=0 Win=65...
  6 7.55438...  10.1.1.101  10.1.1.1    TCP      78 63144 → 80 [SYN] Seq=0 Win=65...

Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interfac
Ethernet II, Src: Apple_a4:01:e6 (78:4f:43:a4:01:e6), Dst: Parallel_26:37:e5 (
Internet Protocol Version 4, Src: 10.1.1.101, Dst: 10.1.1.1
Transmission Control Protocol, Src Port: 63132, Dst Port: 80, Seq: 0, Len: 0

0000  00 1c 42 26 37 e5 78 4f  43 a4 01 e6 08 00 45 00   ..B&7.xO C.....E.
0010  00 40 e6 dd 40 00 40 06  3d 73 0a 01 01 65 0a 01   .@..@.@. =s...e..
0020  01 01 f6 9c 00 50 40 18  75 fe 00 00 00 00 b0 02   .....P@. u.......
0030  ff ff 61 a3 00 00 02 04  05 b4 01 03 03 05 01 01   ..a..... ........
0040  08 0a 38 90 d9 5e 00 00  00 00 04 02 00 00         ..8..^.. ......
```

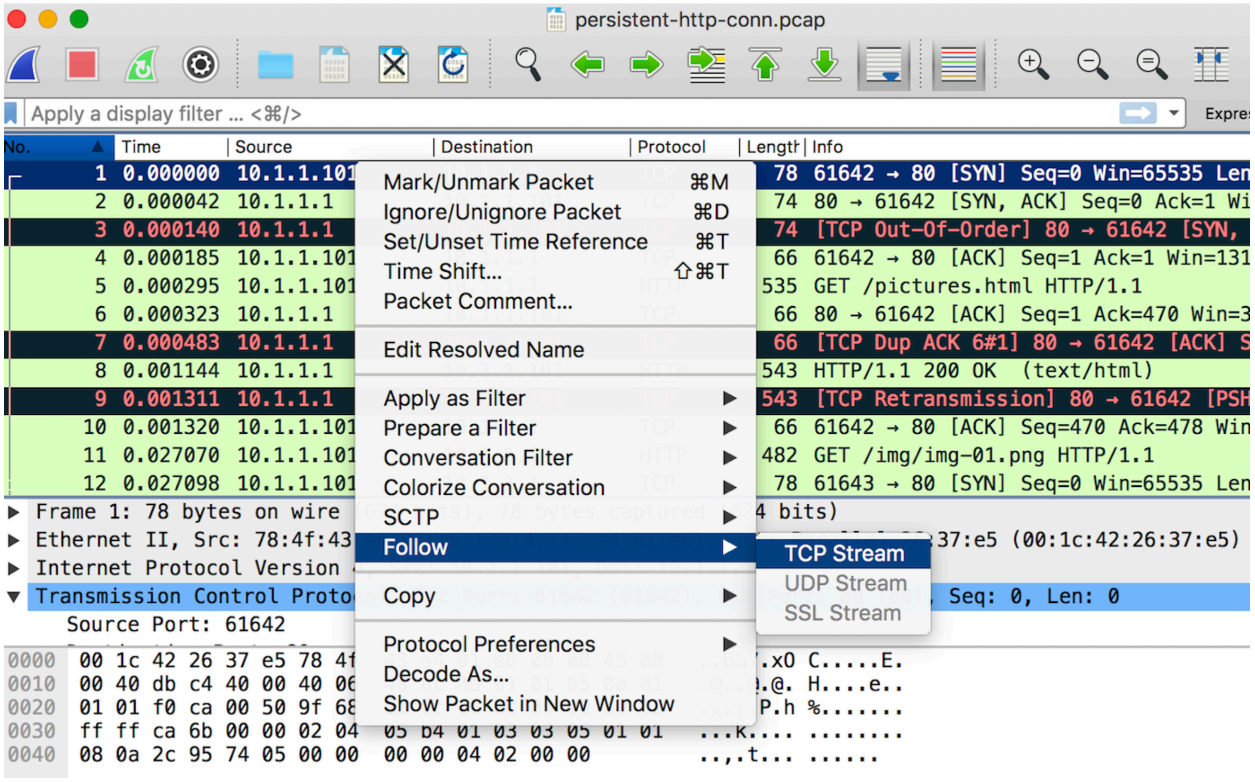
Figure 6: Capture of new connection when MaxKeepAlive reached

To analyse which URLs are served on which web page, restart the wireshark capture with a simple capture filter of '*host 10.1.1.101 and host myweb.com and port 80'* i.e. capture all the packets that are exchanged between 10.1.1.101 and myweb.com (i.e. 10.1.1.1) on port 80. Select any TCP packet, right click on it, select `Follow->TCP Stream` as shown in Figure 7. This will open up a window which will show all the HTTP requests served on this TCP Connection. To further understand the behaviour and how the web server keeps track of number of web requests served on a TCP connection, analyze the HTTP Header '*Keep-Alive*' in subsequent requests. The first response will have value '`Keep-Alive: timeout=5, max=11`', the next reponse will have value `Keep-Alive: timeout=5, max=10`', and so on. The field value max continues to decrease by one with each response, and last request will have value 'Connection: close' and thus the TCP connection is then closed.

To develop a better understanding of persistent connections, repeat the experiment with following setup

i. Create a simple web page with 20 embedded URLs.
ii. Configure Firefox browser with number of persistent connections to 6.
iii. Configure the web server with `MaxKeepAliveRequests` to 10.

iv. Keep the Keep alive time out as 10 seconds.
v. In wireshark, capture the packets with host myweb.com(or 10.1.1.1) and port 80.



```
No.   Time      Source      Destination  Protocol  Length Info
  1 0.000000 10.1.1.101                           78 61642 → 80 [SYN] Seq=0 Win=65535 Len
  2 0.000042 10.1.1.1                             74 80 → 61642 [SYN, ACK] Seq=0 Ack=1 Wi
  3 0.000140 10.1.1.1                             74 [TCP Out-Of-Order] 80 → 61642 [SYN,
  4 0.000185 10.1.1.101                           66 61642 → 80 [ACK] Seq=1 Ack=1 Win=131
  5 0.000295 10.1.1.101                          535 GET /pictures.html HTTP/1.1
  6 0.000323 10.1.1.1                             66 80 → 61642 [ACK] Seq=1 Ack=470 Win=3
  7 0.000483 10.1.1.1                             66 [TCP Dup ACK 6#1] 80 → 61642 [ACK] S
  8 0.001144 10.1.1.1                            543 HTTP/1.1 200 OK  (text/html)
  9 0.001311 10.1.1.1                            543 [TCP Retransmission] 80 → 61642 [PSH
 10 0.001320 10.1.1.101                           66 61642 → 80 [ACK] Seq=470 Ack=478 Win
 11 0.027070 10.1.1.101                          482 GET /img/img-01.png HTTP/1.1
 12 0.027098 10.1.1.101                           78 61643 → 80 [SYN] Seq=0 Win=65535 Len
▶ Frame 1: 78 bytes on wire                        4 bits)
▶ Ethernet II, Src: 78:4f:43                       37:e5 (00:1c:42:26:37:e5)
▶ Internet Protocol Version
▼ Transmission Control Proto                       Seq: 0, Len: 0
    Source Port: 61642
0000  00 1c 42 26 37 e5 78 4f                      .xO C.....E.
0010  00 40 db c4 40 00 40 06                      .@. H....e..
0020  01 01 f0 ca 00 50 9f 68                      P.h %.......
0030  ff ff ca 6b 00 00 02 04   05 b4 01 03 03 05 01 01   ...k.... ........
0040  08 0a 2c 95 74 05 00 00   00 00 04 02 00 00         ..,.t... ......
```

Figure 7 : Identify all HTTP Request on a TCP Connection

Refresh the web page 5 times at the interval of about 5 seconds and analyze the results. It should work as follows. Each web page with 20 embedded images results in 21 web accesses. First 6 TCP connections will server 6×(1+10) = 66 web accesses which will cater to first 3 web refreshes. 4th refresh should result in establishing another set of 6 new TCP connections. Similarly, refresh the web page after about 11 seconds. This should result all the 6 TCP connections getting established again. Repeat the exercise with other browsers such Chrome or any other browser and analyze the results.

Cache Control and Content Caching

The webpage *pictures.html* defined above has 10 images embedded in it. A typical web page in the internet contain 50+ images [9]. On a website, a large number of these images are common across multiple web pages. Thus, when a different webpage is accessed on the same website, browser accesses these images again via the same URL. In general, size of these images adds up to few MBs and if each of these images is to be downloaded every time fully, it not only wastes the network bandwidth and hence download time, compute resources but also delays the display of web page. Thus, to improve the display performance of such web pages, these images are typically cached by the browser. However, before a browser can use the image from the cached content, it must ensure that these images have not changed on the web server, and if there is a change in the image then it should get the new updated images and discard the image in the cache. When the content has not changed and, the browser displays the content from its cache rather than fetching it again from specified web server, significantly improves the web page display performance.

HTTP protocol provides extensive support on content caching and their use while rendering the content on browser web page. In the above mentioned 3rd experiment, HTTP request with its accompanying headers when fetching an image looks as shown in Table 2 and corresponding HTTP response with headers from web server is shown in Table 3.

Table 2 : A typical HTTP Request header for a cached image

```
GET /img/img-09.jpg HTTP/1.1
Host: 10.1.1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:54.0) Gecko/20100101 Firefox/54.0
Accept: */*
Accept-Language: en-US,en;q=0.7,hi;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://10.1.1.1/pictures.html
DNT: 1
Connection: keep-alive
If-Modified-Since: Sat, 05 Aug 2017 16:44:12 GMT
If-None-Match: "5009-5560452ce2725"
Cache-Control: max-age=0
```

Table 3: The HTTP response from the web server for the cached image requested by the browser

```
HTTP/1.1 304 Not Modified
Date: Sun, 06 Aug 2017 10:12:36 GMT
Server: Apache/2.4.18 (Ubuntu)
Connection: Keep-Alive
Keep-Alive: timeout=5, max=7
ETag: "5009-5560452ce2725"
```

At times, it is seen that even when a device is not connected to internet, and if user open the new browser window (or a new tab), it displays the content of the default homepage with embedded images. This display comes from the cache maintained by the browser. Quite often, when a user enters the URL in the browser, it connects to the web server and requests it to send the contents only if these contents have changed from what browser has received on its last time access and already stored in its cache. Thus, the web server checks that if the content has not changed from the last access, then it simply responds with HTTP status *304 Not Modified* indicating the browser should use the content from its cache. To enable browser to use this content storage in browser cache, HTTP protocol provides the required support to enable such caching. The HTTP headers corresponding to dealing with Cache are If-Modified-Since, If-None-Match, Cache-Control, ETag: and Last-Modified: etc. This header should not be confused with another header *Date:* in the response. The latter only provides information about date and time of when this response is sent by web server and has no bearing when the content was created or updated.

When a web server responds to a web request, it generally provides the response header *Last-Modified:* with the date time value of when the accessed resource was last updated/edited. The HTTP header *Cache-Control* is the key header that controls how a cached content should be used by browser and when this content expires and need to be refreshed again. This header provides additional information about caching ability of this data. Below we discuss how this HTTP header is used.

The header *Cache*-Control includes the following fields.

i. *max-age*
ii. *public (applicable to proxies as well)*
iii. *private*
iv. *no-cache*
v. *no-store*
vi. *must-revalidate*
vii. *s-proxy* (to be used by a proxy server)
viii. *proxy-revalidate* (to be used by a proxy server)

The field `max-age`*:* defines the maximum cache validity duration in seconds. The value is relative to the time when the request is made. Once these many seconds are elapsed, the cache becomes invalid and needs to be refreshed. The field `public` implies that any entity, such as proxy server, in the path between web client and web server, can cache this content and use. The field value `private` indicates that this content can be cached only by the end user client and can't be cached by any proxy server in the path. The field `no-cache` is somewhat a misnomer. This indicates to the client, the web client can cache this content, but before using it in the display, this content must be revalidated with the web server. The field `no-store` implies that the content can not be cached at all. A related HTTP header is `Expires`*:* which has same meaning as in the field `max-age` in the header `Cache-control`*.* However, when both `Cache-control` and `Expires` headers are present, the former takes precedence and latter is ignored. The use of fields in `Cache-control` is depicted in Figure 8 and covered in great detail in ([6][8][10]

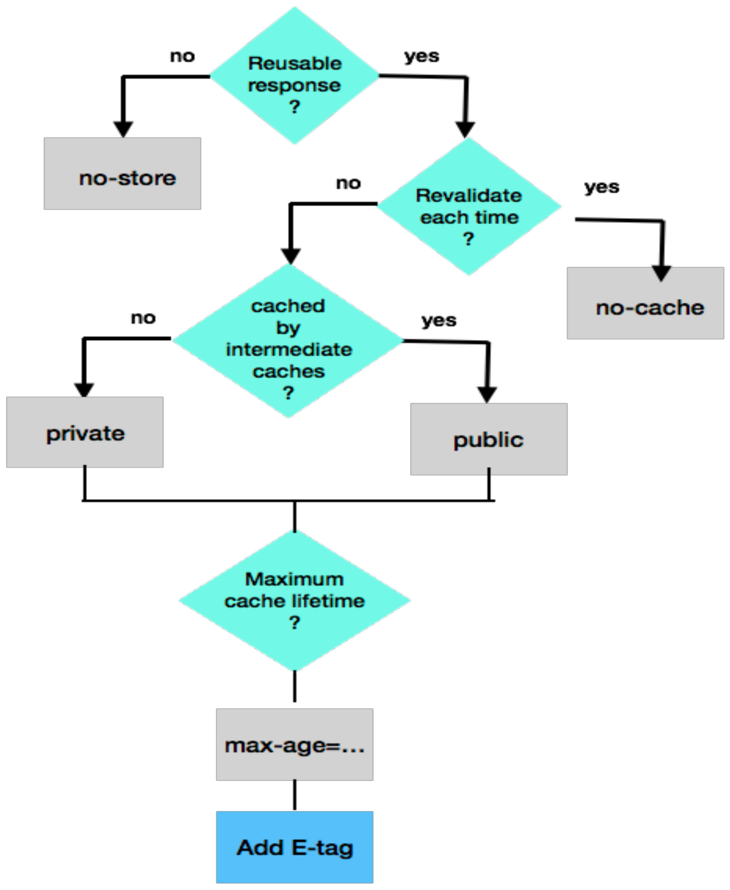
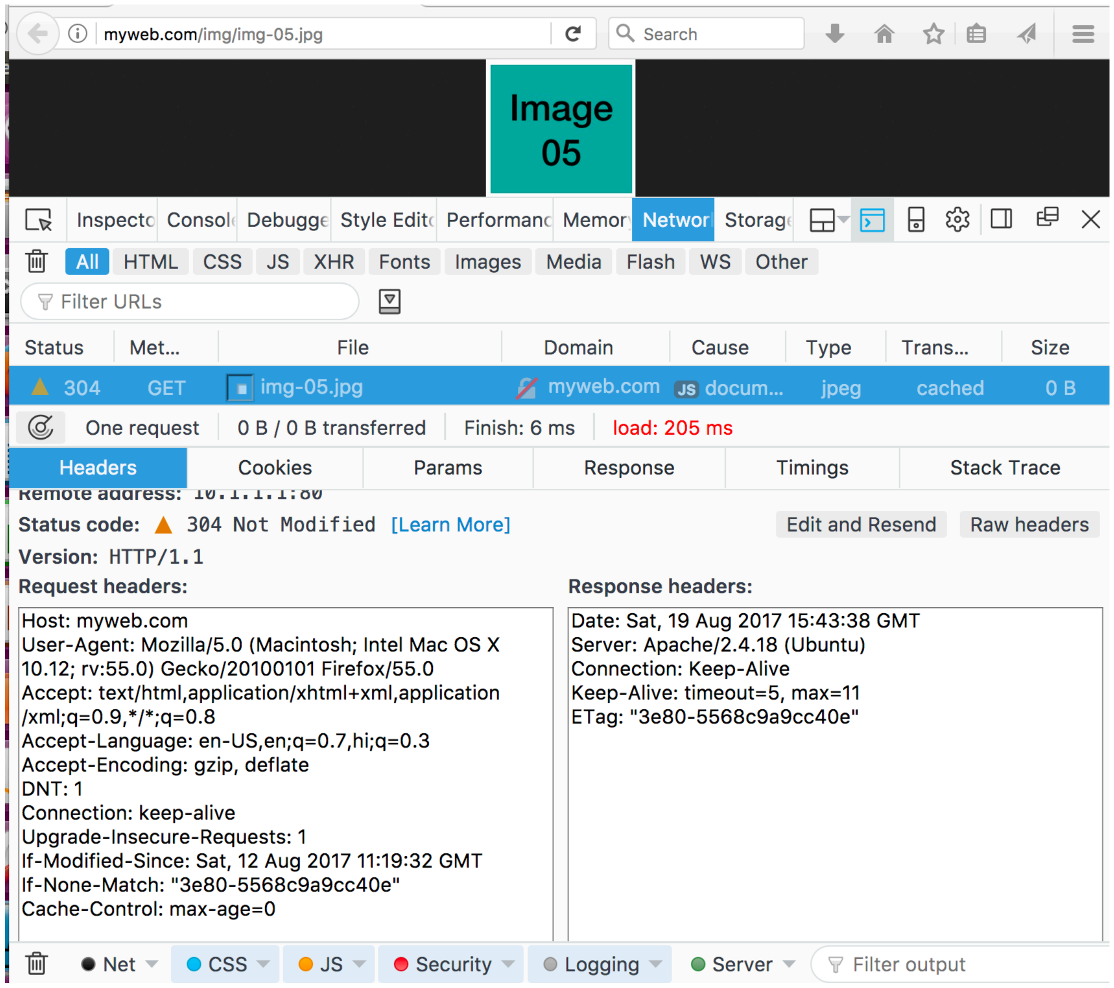


Figure 8: Processing of Cache-Control fields

Now we will discuss the use of other HTTP headers used for managing cache contents in addition to *Cache*-control. Each web response contains the header *Last-Modified*-Date, which corresponds to the date and time of the update/modification made on the resource. The client makes use of this received response header in the following way. When client caches the contents, it stores this last modified date along with it. When web client needs to check for freshness of the cached content, it makes a HTTP request for the cached resource, and sends the stored value of *Last-Modified*-Date in its request header *If-Modified-Since:*. Web server compares the date as received in the request header *If-Modified-Since:* with the last modification date of requested resource, and if the last modification date of this resource is prior to the value received in *If-Modified-Since*, then the web server does not send the content of the requested resource but instead sends the HTTP Response with status code value `304 (Not Modified)`. This indicates to the web client that it should use the resource from its cache to render it in the browser window. To experience the use of this header, in the firefox browser window, open the web developer tools (*Tools -> Web Developer -> Network*). Enter the URL corresponding to one of the image e.g. http://myweb.com/img/img-05.jpg, and in developer tools, under the *Header:* columnm, select the *Raw headers:*. This will show both the request and response headers corresponding to the request and response (see Figure 9).


myweb.com/img/img-05.jpg
Search
Image 05
Inspector Console Debugger Style Editor Performance Memory Network Storage
All HTML CSS JS XHR Fonts Images Media Flash WS Other
Filter URLs
Status Met... File Domain Cause Type Trans... Size
304 GET img-05.jpg myweb.com docum... jpeg cached 0 B
One request 0 B / 0 B transferred Finish: 6 ms load: 205 ms
Headers Cookies Params Response Timings Stack Trace
Edit and Resend Raw headers
Net CSS JS Security Logging Server Filter output

```
Remote address: 10.1.1.1:80
Status code: 304 Not Modified [Learn More]
Version: HTTP/1.1
```

Request headers:

```
Host: myweb.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10.12; rv:55.0) Gecko/20100101 Firefox/55.0
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
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
If-Modified-Since: Sat, 12 Aug 2017 11:19:32 GMT
If-None-Match: "3e80-5568c9a9cc40e"
Cache-Control: max-age=0
```

Response headers:

```
Date: Sat, 19 Aug 2017 15:43:38 GMT
Server: Apache/2.4.18 (Ubuntu)
Connection: Keep-Alive
Keep-Alive: timeout=5, max=11
ETag: "3e80-5568c9a9cc40e"
```

Figure 9: Request and Response headers when cached content is accessed

The request contains the header *If-Modified-Since: Sat,12 Aug 2017 11:19:32 GMT*. The web server has this value in its first response to this image URL and since this image has not changed since this time, it responds with HTTP Status code of 304 and browser displays this image from the cache. Now just to see the behaviour of this header when cached content has become outdated, change the modified date of this image. On the web server, just update the date (e.g. `sudo touch /var/www/html/img/img-05.jpg`). Now refresh the browser. This time since the document modified date is later than the value received in the request header *If-Modified-Since:*, web server responds with the full content with status code `200 OK`. However, sometimes when content is changed but date still remains earlier than requested, then this methodology will not work. For example, consider an earlier version of this image which has been restored. The previous versions would have their modified date value prior to modified date of the last version and thus if web server were to use only the modified date criteria, then it will incorrectly serve the request with status code 304 even though content has changed. To deal with such situations, HTTP protocol provides another header *Etag:*. The web server computes unique signature of the resource and sends this signature via this response header *Etag:*. For example, in the Figure 9, this header is Etag: "3e80-5568c9a9cc40e". The browser will also store this Etag value along with the cached content, and send this value in the request header as *If-None-Match:* "3e80-5568c9a9cc40e ". This tells the web server that if the signature of the resource has changed even though date is not updated, then web server would serve the content in full rather than responding with status code `304`. The server can see that if such an entity tag is still valid and if so, it will not send the response content and indicates the same with response status code {`304 Not Modified`}. If the entity tag has changed, it will send the response with updated document and status code will correspond to {200} or another code in the status category {*2xx*}.

Mechanisms for performance with Partial Content Delivery

In general, with each web response, the web server provides the total repsonse size in the header *Content-Length:* and browser renders the content only after it receives the entire content. When a web request results in a large size response, network latency plays a signifcant role and it takes a while for web client to receive the full content. During the time period when content is being received, browser just waits for the full content and does not render any partial content in the browser window. This results in user getting a feeling of non-responsive website or slow internet even though both browser and server are actively communicating. Example of such large response size could be a large picture, a big table display as a result of *sql* query response in the database, etc. However to keep the user engaged, and for a better user exerience, it would be desirable that if the content can be rendered as and when it is received rather than waiting for full content. Example of such cases include keep displaying the rows of a table having large number of rows, keep rendering the image from top to bottom etc. and thus keep the user engaged. To support such continuous rendering of contents, HTTP 1.1 supports the concept of *chunk* based transfer. It provides for request and response header *Transfer-Encoding: chunked*. This header is infrequently used in HTTP request but finds a good usage in HTTP response. For a table with large number of rows, this header enables sending of few rows data at a time as a chunk, and when data corresponding to all the rows is sent, the last chunk is sent with size 0 indicating end of chunk transfer. Similarly, for a picture or multimedia contents, the web sever can start sending rows of pixels in separate chunks and browser will display the picture as and when pixels are received in the chunk. Even for large text content such as big news article, each section of such content can be sent as one chunk each.

The web server when using *Transfer-Encoding: chunked* in the response header, the output contains size of chunk in hexadecimal format on a line by itself followed by the chunk data followed by empty line, and then size of next chunk, data of the next chunk, empty line and so on. The last chunk size of 0 implies entire data is sent. A sample example of chunked encoded response is given below in Table 4. In this first chunk is of size 19 (hex 13) bytes, 2nd chunk is of size 10 (hex A) bytes, third chunk is of size 17 (hex 11) bytes and last chunk of size 0 indicating end of chunked transfer.

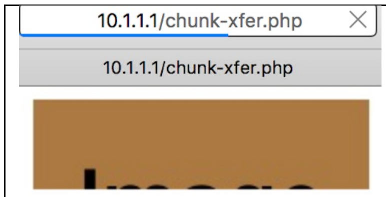
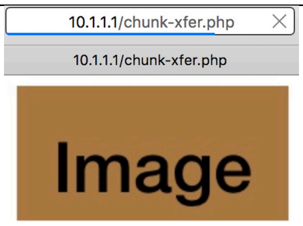
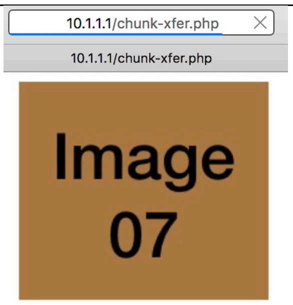
Table 4: An example of a chunk based response

```
HTTP/1.1 200 OK
:
Transfer-Encoding: chunked
:

13
```

```
    This is an example

    A
    of Chunked

    11
    Transfer encoding

    0
```

When the sender also includes *Content-Length* header along with the *Transfer-Encoding: chunked*, the former must be ignored and only the latter that should be catered. Such a response is typically generated by web application program invoked by the web server to generate the dynamic content. The php program in appendix A provides an example of such a server side URL. This URL http://myweb.com/chunk-xfer.php will display the incrementally and slowly each with a chunk size of 1KBytes. The Table 5 shows three snapshots of such image displays with chunk transfer encoding with each chunk size of 1000 (hex 3E8) bytes. To get a better understanding your own chunk transfer, create your own web page and analyze how various chunk affect the user experience when web page is displayed. Analyze the entire response with wireshark capture.

Table 5: An image being displayed with chunk transfer encoding

		 10.1.1.1/chunk-xfer.php 10.1.1.1/chunk-xfer.php Image 07

HTTP 1.1 protocol also provides another mechanism to support partial delivery of contents. This is generally used during downloads of large content. A client achieves this by specifying byte ranges in the request header and server would serve only that much data which has been asked for. For example request `Range: bytes=0-199` imply that a web server would send only first 200 bytes; Similarly, a request header `Range: bytes=200-299` would result in web server sending 100 bytes data of fetched resource starting from the offset 200. This way a client makes multiple requests for a resource with different ranges identifying smaller chunk sizes and a web server will respond for those chunks. An example of partial response is given below in Table 6. Since it is partial content, the status response code will be `206 Partial Content` instead of `200 OK`. The response header `Content-Range: bytes 200-299/873` indicates that this response contains 100 bytes of data starting from offset 200 out of a total size of 873 bytes. This partial response is particularly very useful when downloading a large file and connection breaks. This enables client to resume the download from the point of failure rather than start from the beginning.

Table 6: The response header for partial contents

```
HTTP/1.1 206 Partial Content
    Server: Apache/2.4.7 (Ubuntu)
    :
```

```
    Accept-Ranges: bytes
    Content-Length: 200
    Vary: Accept-Encoding
    Content-Range: bytes 200-299/873
    Content-Type: text/html
```

To experience a good understanding of partial content, do the experiment with following setup. As we need to make a HTTP request with request header “*Range:*”, use the *curl* tool to make the request for partial chunks.

i. Create a simple web page (e.g. mypage.html) with some contents (it does not need to have any embedded links).
ii. Deploy this web page on the web serer. Note down the size N of this web page. For simplicity, let us assume N=500.
iii. Access this web page in 3 chunks as chunk 1 (0-199), chunk 2 (200 to 399) and chunk 3 (400 to 500). Compute your own chunk sizes.
iv. Using *curl*, download the 3 chunks as follows. *(*If *curl* is not available, it can be installed on Ubuntu as `sudo apt-get install curl`)
    a. `curl –H “Range: bytes=0-199” –o chunk1.html http://myweb.com/mypage.html`
    b. `curl –H “Range: bytes=200-399” –o chunk2.html http://myweb.com/mypage.html`
    c. `curl –H “Range: bytes=400-” –o chunk3.html http://myweb.com/mypage.html`
v. Combine the 3 chunks into single document e.g. `cat chunk1.html chunk2.html chunk3.html >response.html`
vi. Verify that content of response.html is same as that of the web page mypage.html created in the first step.

Summary

We have discussed 3 different performance mechanisms that are supported by HTTP protocol to enable a better user experience. The mechanism of multiple persistent connections is used by default in all the commonly used web browsers. However, HTTP/2 protocol makes use of only 1 persistent connections and still provides better performance than HTTP/1.1. This we will discuss in subsequent articles. The mechanism of caching the contents helps on saving the bandwidth and network latency especially when web page consists of multi-media contents. The mechanism of transfer chunk encoding is used to provide a better user experience and engage the user even before the full response is received. The mechanism of partial content is useful when downloading a large URL and when downloading aborts and has to be resumed from the point of last failure. As all these performance mechanisms are used to deal with the issues of poor network latency, in the next article we will discuss network latency and how to measure it.

Appendix A

The PHP code to demonstrate the use of *Transfer-Encoding: chunked*.

```
<?php
$file = 'img/img-07.jpg';
if (is_file($file)) {
    header('Content-Type: image/jpeg');
    header('Transfer-Encoding: chunked');
```

```
        $chunkSize = 1000;
        $handle = fopen($file, 'rb');
        while (!feof($handle)) {
            $buffer = fread($handle, $chunkSize);
            # send chunk size in hex, chunk content, new line
            echo sprintf("%x\r\n", $chunkSize);
            echo $buffer; echo "\r\n";
            ob_flush(); flush();
            usleep(500000); # emulate network latency
        }
        fclose($handle);
        exit;
    } else {
        header('Content-Type: text/html');
        echo "\r\nNo picture available for $file\r\n";
    }
    >
```

References

[1] RFC 1945, "Hyper Text Transfer Protocol – HTTP/1.0", Network Working Group, Informational, Berners-Lee(MIT), Fielding (UC Irvine), Frystyk(MIT), May 1996.

[2] RFC 2068, "Hypertext Transfer Protocol – HTTP/1.1", Proposed Standard, Fielding (UC Irvine), Gettys, Mogul(DEC), FRystyk, Berners-LEE (MIT), Jan 1997.

[3] RFC 2616, "Hyper Text Transfer Protocol – HTTP/1.1", Network Working Group, Request for Comments 2616. Fielding (UC Irvine), Gettys (Compaq), Mogul (Compaq), Frystyk(MIT), Masinter (Xerox), Leach (Microsot), Berners-Lee(MIT), June 1999

[4] RFC 7540., "Hypertext Transfer Protocol Version 2 (HTTP/2)", IETF, Proposed Standard, Belshe (Bitgo), Peon (Google), Thmoson (Mozilla), May 2015

[5] Wiireshark – The Network Protocol Analyzer, https://www.wireshark.org/#learnWS, Accessed Aug 2017.

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

[7] Apache Web Server https://httpd.apache.org/docs/2.4, accessed July 2017

[8] https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en, Accessed August 2017

[9] http://httparchive.org/trends.php. Accessed August 2017

[10] https://www.mnot.net/cache_docs/, Accessed August 2017

[11] http://acc.digital/experiential-learning/, The first article on experiential learning of Networking Technologies, Jun 30, 2017, ACCS India.