Lab 1 - Protocol Layers

Step 1: Capture a Trace

Proceed as follows to capture a trace of network traffic; alternatively, you may use a supplied trace. We want this trace to look at the protocol structure of packets. A simple Web fetch of a URL from a server of your choice to your computer, which is the client, will serve as traffic.

1. Pick a URL and fetch it with wget or curl. For example, “wget http://www.google.com” or “curl http://www.google.com”. This will fetch the resource and either write it to a file (wget) or to the screen (curl). You are checking to see that the fetch works and retrieves some content. A successful example is shown below (with added highlighting) for wget. You want a single response with status code “200 OK”. If the fetch does not work then try a different URL; if no URLs seem to work then debug your use of wget/curl or your Internet connectivity.

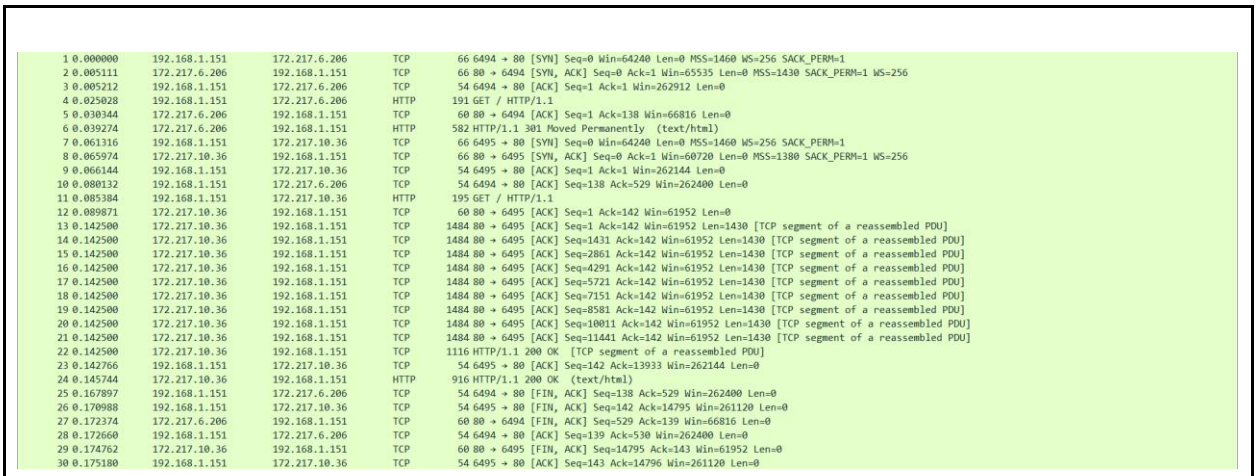
```
wsl@DESKTOP-OLHVFBO:/mnt/c/Users/esam5/Desktop/School/Fall 2020/CSCI-415/Labs/Lab-1/Data-Communications-and-Networking$ wget "google.com"
Will not apply HSTS. The HSTS database must be a regular and non-world-writable file.
ERROR: could not open HSTS store at '/home/wsl/.wget-hsts'. HSTS will be disabled.
--2020-09-30 16:06:57--  http://google.com/
Resolving google.com (google.com)... 172.217.10.46, 2607:f8b0:4006:814::200e
Connecting to google.com (google.com)|172.217.10.46|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://www.google.com/ [following]
--2020-09-30 16:06:57--  http://www.google.com/
Resolving www.google.com (www.google.com)... 172.217.9.228, 2607:f8b0:4006:813::2004
Connecting to www.google.com (www.google.com)|172.217.9.228|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: ‘index.html’

index.html                          [ <=>                                                ]  13.71
K  --.-KB/s    in 0s

2020-09-30 16:06:57 (45.4 MB/s) - ‘index.html’ saved [14034]
```

2. *Close unnecessary browser tabs and windows*. By minimizing browser activity you will stop your computer from fetching unnecessary web content, and avoid incidental traffic in the trace.

3. *Launch Wireshark and start a capture with a filter of* “tcp port 80” and check “enable net-work name resolution”. This filter will record only standard web traffic and not other kinds of packets that your computer may send. The checking will translate the addresses of the computers sending and receiving packets into names, which should help you to recognize whether the packets are going to or from your computer.

4. *When the capture is started, repeat the web fetch using* wget/curl *above.* This time, the packets will be recorded by Wireshark as the content is transferred.

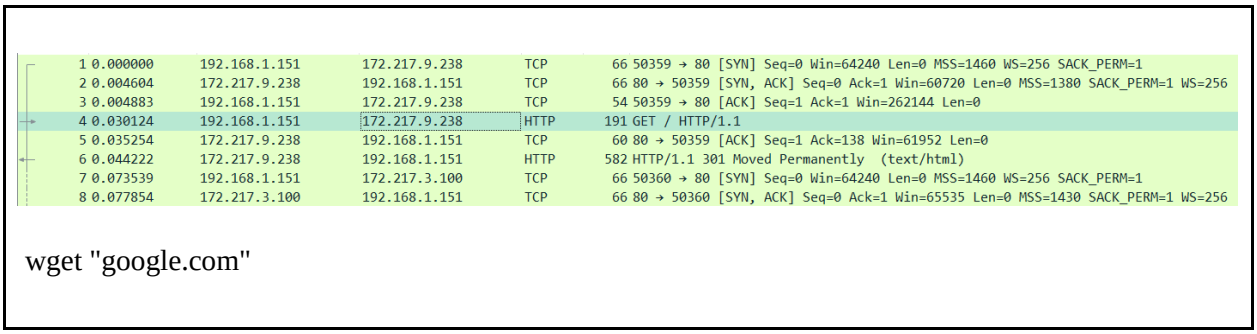
5. *After the fetch is successful, return to Wireshark and use the menus or buttons to stop the trace.* If you have succeeded, the upper Wireshark window will show multiple packets, and most likely it will be full. How many packets are captured will depend on the size of the web page, but there should be at least 8 packets in the trace, and typically 20-100, and many of these packets will be colored green. An example is shown below. Congratulations, you have captured a trace!



1	0.000000	192.168.1.151	172.217.6.206	TCP	66	6494 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.005111	172.217.6.206	192.168.1.151	TCP	66	80 → 6494 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1430 SACK_PERM=1 WS=256
3	0.005212	192.168.1.151	172.217.6.206	TCP	54	6494 → 80 [ACK] Seq=1 Ack=1 Win=262912 Len=0
4	0.025028	192.168.1.151	172.217.6.206	HTTP	191	GET / HTTP/1.1
5	0.030344	172.217.6.206	192.168.1.151	TCP	60	80 → 6494 [ACK] Seq=1 Ack=138 Win=66816 Len=0
6	0.039274	172.217.6.206	192.168.1.151	HTTP	582	HTTP/1.1 301 Moved Permanently (text/html)
7	0.061316	192.168.1.151	172.217.10.36	TCP	66	6495 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
8	0.065974	172.217.10.36	192.168.1.151	TCP	66	80 → 6495 [SYN, ACK] Seq=0 Ack=1 Win=60720 Len=0 MSS=1380 SACK_PERM=1 WS=256
9	0.066144	192.168.1.151	172.217.10.36	TCP	54	6495 → 80 [ACK] Seq=1 Ack=1 Win=262144 Len=0
10	0.080132	192.168.1.151	172.217.6.206	TCP	54	6494 → 80 [ACK] Seq=138 Ack=529 Win=262400 Len=0
11	0.085384	192.168.1.151	172.217.10.36	HTTP	195	GET / HTTP/1.1
12	0.089871	172.217.10.36	192.168.1.151	TCP	60	80 → 6495 [ACK] Seq=1 Ack=142 Win=61952 Len=0
13	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=1 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
14	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=1431 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
15	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=2861 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
16	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=4291 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
17	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=5721 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
18	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=7151 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
19	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=8581 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
20	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=10011 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
21	0.142500	172.217.10.36	192.168.1.151	TCP	1484	80 → 6495 [ACK] Seq=11441 Ack=142 Win=61952 Len=1430 [TCP segment of a reassembled PDU]
22	0.142500	172.217.10.36	192.168.1.151	TCP	1116	HTTP/1.1 200 OK [TCP segment of a reassembled PDU]
23	0.142766	192.168.1.151	172.217.10.36	TCP	54	6495 → 80 [ACK] Seq=142 Ack=13933 Win=262144 Len=0
24	0.145744	172.217.10.36	192.168.1.151	HTTP	916	HTTP/1.1 200 OK (text/html)
25	0.167897	192.168.1.151	172.217.6.206	TCP	54	6494 → 80 [FIN, ACK] Seq=138 Ack=529 Win=262400 Len=0
26	0.170988	192.168.1.151	172.217.10.36	TCP	54	6495 → 80 [FIN, ACK] Seq=142 Ack=14795 Win=261120 Len=0
27	0.172374	172.217.6.206	192.168.1.151	TCP	60	80 → 6494 [FIN, ACK] Seq=529 Ack=139 Win=66816 Len=0
28	0.172660	192.168.1.151	172.217.6.206	TCP	54	6494 → 80 [ACK] Seq=139 Ack=530 Win=262400 Len=0
29	0.174762	172.217.10.36	192.168.1.151	TCP	60	80 → 6495 [FIN, ACK] Seq=14795 Ack=143 Win=61952 Len=0
30	0.175180	192.168.1.151	172.217.10.36	TCP	54	6495 → 80 [ACK] Seq=143 Ack=14796 Win=261120 Len=0

Step 2: Inspect the Trace

Select a packet for which the Protocol column is "HTTP" and the Info column says it is a GET. It is the packet that carries the web (HTTP) request sent from your computer to the server. (You can click the column headings to sort by that value, though it should not be difficult to find an HTTP packet by inspection.) Let's have a closer look to see how the packet structure reflects the protocols that are in use.



1	0.000000	192.168.1.151	172.217.9.238	TCP	66	50359 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.004604	172.217.9.238	192.168.1.151	TCP	66	80 → 50359 [SYN, ACK] Seq=0 Ack=1 Win=60720 Len=0 MSS=1380 SACK_PERM=1 WS=256
3	0.004883	192.168.1.151	172.217.9.238	TCP	54	50359 → 80 [ACK] Seq=1 Ack=1 Win=262144 Len=0
4	0.030124	192.168.1.151	172.217.9.238	HTTP	191	GET / HTTP/1.1
5	0.035254	172.217.9.238	192.168.1.151	TCP	60	80 → 50359 [ACK] Seq=1 Ack=138 Win=61952 Len=0
6	0.044222	172.217.9.238	192.168.1.151	HTTP	582	HTTP/1.1 301 Moved Permanently (text/html)
7	0.073539	192.168.1.151	172.217.3.100	TCP	66	50360 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
8	0.077854	172.217.3.100	192.168.1.151	TCP	66	80 → 50360 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1430 SACK_PERM=1 WS=256

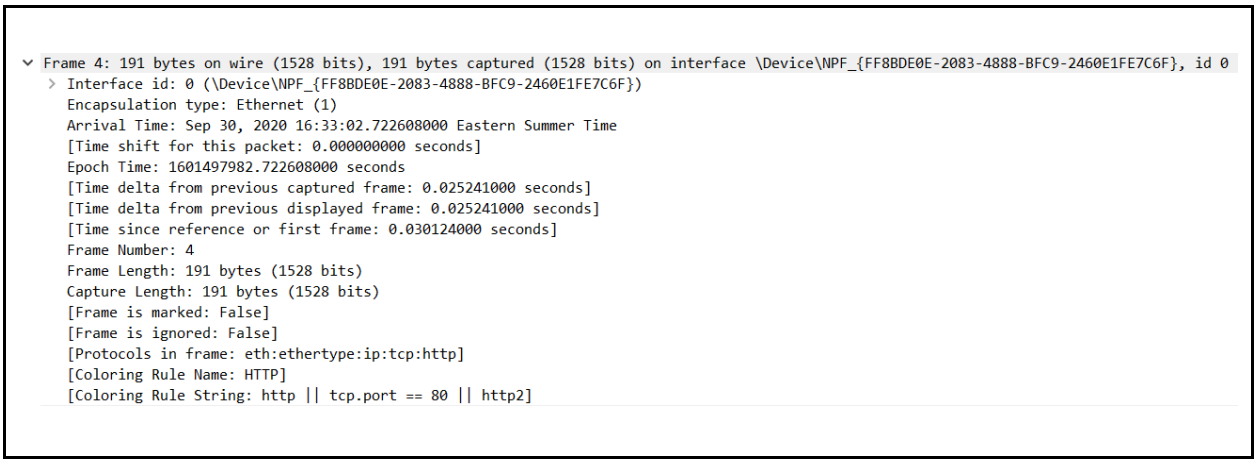
wget "google.com"

Since we are fetching a web page, we know that the protocol layers being used are as shown below. That is, HTTP is the application layer web protocol used to fetch URLs. Like many Internet applications, it runs on top of the TCP/IP transport and network layer protocols. The link and physical layer protocols

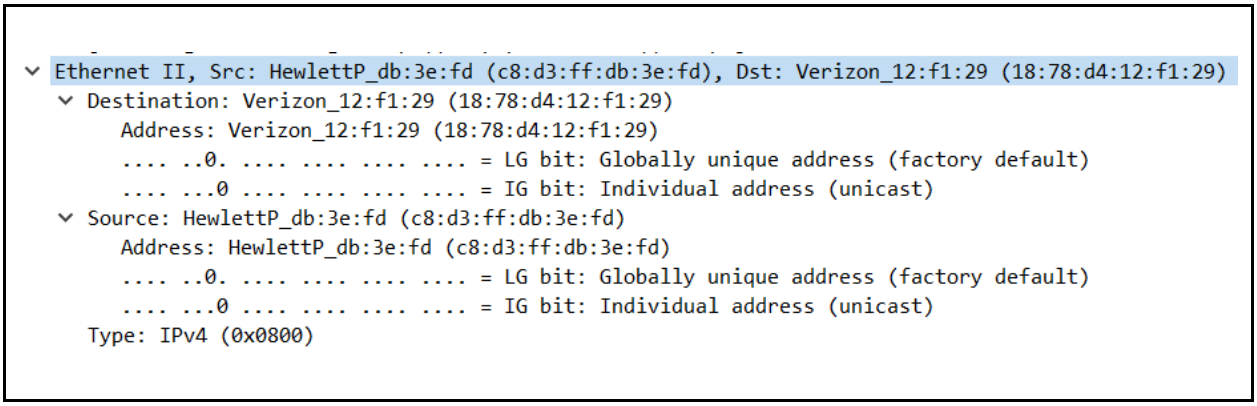
depend on your network, but are typically combined in the form of Ethernet (shown) if your computer is wired, or 802.11 (not shown) if your computer is wireless.

With the HTTP GET packet selected, look closely to see the similarities and differences between it and our protocol stack as described next. The protocol blocks are listed in the middle panel. You can expand each block (by clicking on the "+" expander or icon) to see its details.

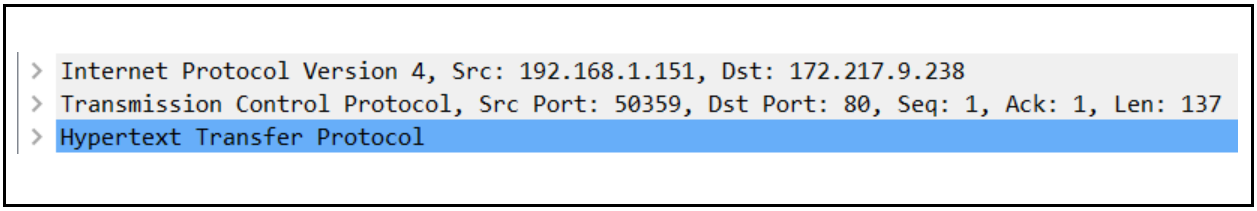
• The first Wireshark block is "Frame". This is not a protocol, it is a record that describes overall information about the packet, including when it was captured and how many bits long it is.



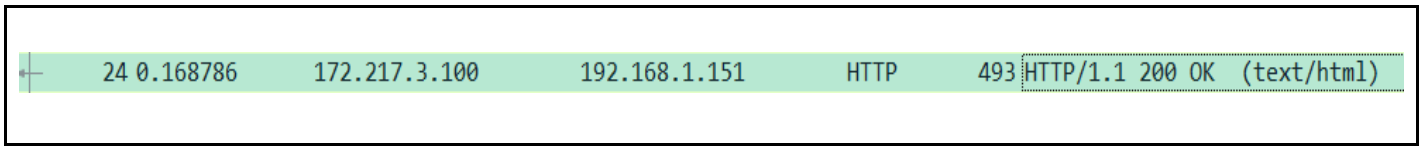
• The second block is "Ethernet". This matches our diagram! Note that you may have taken a trace on a computer using 802.11 yet still see an Ethernet block instead of an 802.11 block. Why? It happens because we asked Wireshark to capture traffic in Ethernet format on the cap-ture options, so it converted the real 802.11 header into a pseudo-Ethernet header.



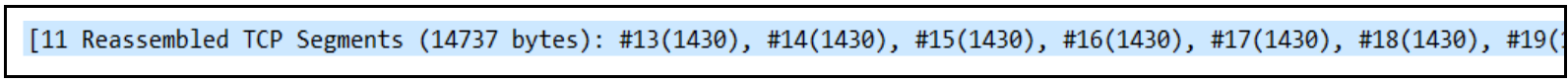
• Then come IP, TCP, and HTTP, which are just as we wanted. Note that the order is from the bot-tom of the protocol stack upwards. This is because as packets are passed down the stack, the header information of the lower layer protocol is added to the front of the information from the higher layer protocol, as in Fig. 1-15 of your text. That is, the lower layer protocols come first in the packet “on the wire”.



Now find another HTTP packet, the response from the server to your computer, and look at the structure of this packet for the differences compared to the HTTP GET packet. This packet should have “200 OK” in the Info field, denoting a successful fetch. In our trace, there are two extra blocks in the detail panel as seen in the next figure.



• The first extra block says “[11 reassembled TCP segments …]”.



• The second extra block says “Line-based text data …”.

Step 3: Packet Structure

To show your understanding of packet structure, draw a figure of an HTTP GET packet that shows the position and size in bytes of the TCP, IP and Ethernet protocol headers. Your figure can simply show the overall packet as a long, thin rectangle. Leftmost elements are the first sent on the wire. On this drawing, show the range of the Ethernet header and the Ethernet payload that IP passed to Ethernet to send over the network. To show the nesting structure of protocol layers, note the range of the IP header and the IP

payload. You may have questions about the fields in each protocol as you look at them. We will explore these protocols and fields in detail in future labs.

Turn-in: Hand in your packet drawing.

HTTP 191 bytes	TCP 20 bytes	IP 20 bytes	Ethernet 14 bytes	Get Packet

Step 4: Protocol Overhead

Estimate the download protocol overhead, or percentage of the download bytes taken up by protocol overhead. To do this, consider HTTP data (headers and message) to be useful data for the network to carry, and lower layer headers (TCP, IP, and Ethernet) to be the overhead. We would like this overhead to be small, so that most bits are used to carry content that applications care about. To work this out, first look at only the packets in the download direction for a single web fetch. You might sort on the Destination column to find them. The packets should start with a short TCP packet described as a SYN ACK, which is the beginning of a connection. They will be followed by mostly longer packets in the middle (of roughly 1 to 1.5KB), of which the last one is an HTTP packet. This is the main portion of the download. And they will likely end with a short TCP packet that is part of ending the connection. For each packet, you can inspect how much overhead it has in the form of Ethernet / IP / TCP headers, and how much useful HTTP data it carries in the TCP payload. You may also look at the HTTP packet in Wireshark to learn how much data is in the TCP payloads over all download packets.

Turn-in: Your estimate of download protocol overhead as defined above. Tell us whether you find this overhead to be significant.

Estimation:

```
 1 0.000000    192.168.1.151   172.217.9.238   TCP      66 50359 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
 2 0.004604    172.217.9.238   192.168.1.151   TCP      66 80 → 50359 [SYN, ACK] Seq=0 Ack=1 Win=60720 Len=0 MSS=1380 SACK_PERM=1 WS=256
 3 0.004883    192.168.1.151   172.217.9.238   TCP      54 50359 → 80 [ACK] Seq=1 Ack=1 Win=262144 Len=0
 4 0.030124    192.168.1.151   172.217.9.238   HTTP    191 GET / HTTP/1.1
 5 0.035254    172.217.9.238   192.168.1.151   TCP      60 80 → 50359 [ACK] Seq=1 Ack=138 Win=61952 Len=0
 6 0.044222    172.217.9.238   192.168.1.151   HTTP    582 HTTP/1.1 301 Moved Permanently  (text/html)
 7 0.073539    192.168.1.151   172.217.3.100   TCP      66 50360 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
 8 0.077854    172.217.3.100   192.168.1.151   TCP      66 80 → 50360 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1430 SACK_PERM=1 WS=256
 9 0.077916    192.168.1.151   172.217.3.100   TCP      54 50360 → 80 [ACK] Seq=1 Ack=1 Win=262912 Len=0
10 0.099184    192.168.1.151   172.217.9.238   TCP      54 50359 → 80 [ACK] Seq=138 Ack=529 Win=261632 Len=0
11 0.101689    192.168.1.151   172.217.3.100   HTTP    195 GET / HTTP/1.1
12 0.106188    172.217.3.100   192.168.1.151   TCP      60 80 → 50360 [ACK] Seq=1 Ack=142 Win=66816 Len=0
13 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=1 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
14 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=1431 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
15 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=2861 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
16 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=4291 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
17 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=5721 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
18 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=7151 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
19 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=8581 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
20 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=10011 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
21 0.163518    172.217.3.100   192.168.1.151   TCP    1484 80 → 50360 [ACK] Seq=11441 Ack=142 Win=66816 Len=1430 [TCP segment of a reassembled PDU]
22 0.163518    172.217.3.100   192.168.1.151   TCP    1482 80 → 50360 [PSH, ACK] Seq=12871 Ack=142 Win=66816 Len=1428 [TCP segment of a reassembled PDU]
23 0.163972    192.168.1.151   172.217.3.100   TCP      54 50360 → 80 [ACK] Seq=142 Ack=14299 Win=262912 Len=0
24 0.168786    172.217.3.100   192.168.1.151   HTTP    493 HTTP/1.1 200 OK  (text/html)
25 0.208890    192.168.1.151   172.217.3.100   TCP      54 50360 → 80 [ACK] Seq=142 Ack=14738 Win=262656 Len=0
26 0.221061    192.168.1.151   172.217.9.238   TCP      54 50359 → 80 [FIN, ACK] Seq=138 Ack=529 Win=261632 Len=0
27 0.225233    172.217.9.238   192.168.1.151   TCP      60 80 → 50359 [FIN, ACK] Seq=529 Ack=139 Win=61952 Len=0
28 0.225236    192.168.1.151   172.217.3.100   TCP      54 50360 → 80 [FIN, ACK] Seq=142 Ack=14738 Win=262656 Len=0
29 0.225537    192.168.1.151   172.217.9.238   TCP      54 50359 → 80 [ACK] Seq=139 Ack=530 Win=261632 Len=0
30 0.230040    172.217.3.100   192.168.1.151   TCP      60 80 → 50360 [FIN, ACK] Seq=14738 Ack=143 Win=66816 Len=0
31 0.230396    192.168.1.151   172.217.3.100   TCP      54 50360 → 80 [ACK] Seq=143 Ack=14739 Win=262656 Len=0
```

Frame #:	TCP | IP | Ethernet (Overhead) bytes	HTTP Data (Useful) bytes
2	66	0
3	54	0
4	40	191
5	60	0
8	66	0
9	54	0
10	54	0
11	40	195
12	60	0
13	1484	0
14	1484	0
15	1484	0
16	1484	0
17	1484	0
18	1484	0
19	1484	0

20	**1484**	**0**
21	**1484**	**0**
22	**1482**	**0**
23	**54**	**0**
24	**40**	**493**
25	**54**	**0**

Total Overhead: 15480 bytes

Useful data: 879 bytes

15480 bytes + 879 bytes = 16719 bytes

15480/16719 *100 = 94.74%

The overhead is significant as ~95% of the byte data consists of overhead. 5% is relevant (http) data.

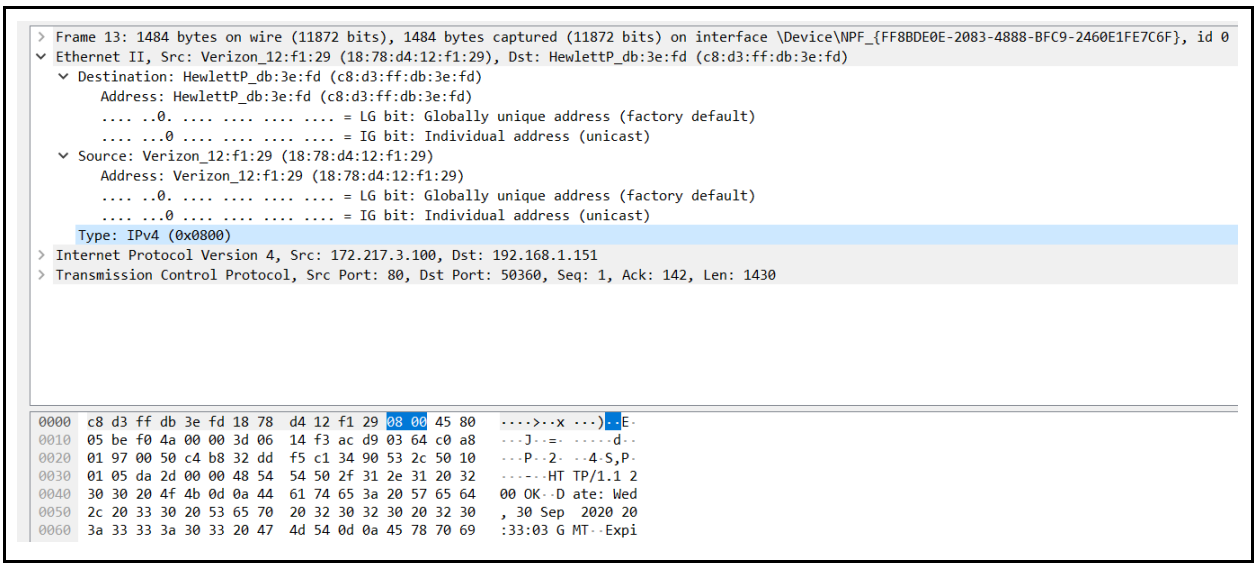
Step 5: Demultiplexing Keys

When an Ethernet frame arrives at a computer, the Ethernet layer must hand the packet that it contains to the next higher layer to be processed. The act of finding the right higher layer to process received packets is called demultiplexing. We know that in our case the higher layer is IP. But how does the Ethernet protocol know this? After all, the higher-layer could have been another protocol entirely (such as ARP). We have the same issue at the IP layer – IP must be able to determine that the contents of IP message is a TCP packet so that it can hand it to the TCP protocol to process. The answer is that protocols use information in their header known as a "demultiplexing key" to determine the higher layer. CN5E Labs (1.0) © 2012 D. Wetherall 8

Turn-in: Hand in your answers to the [below] questions.

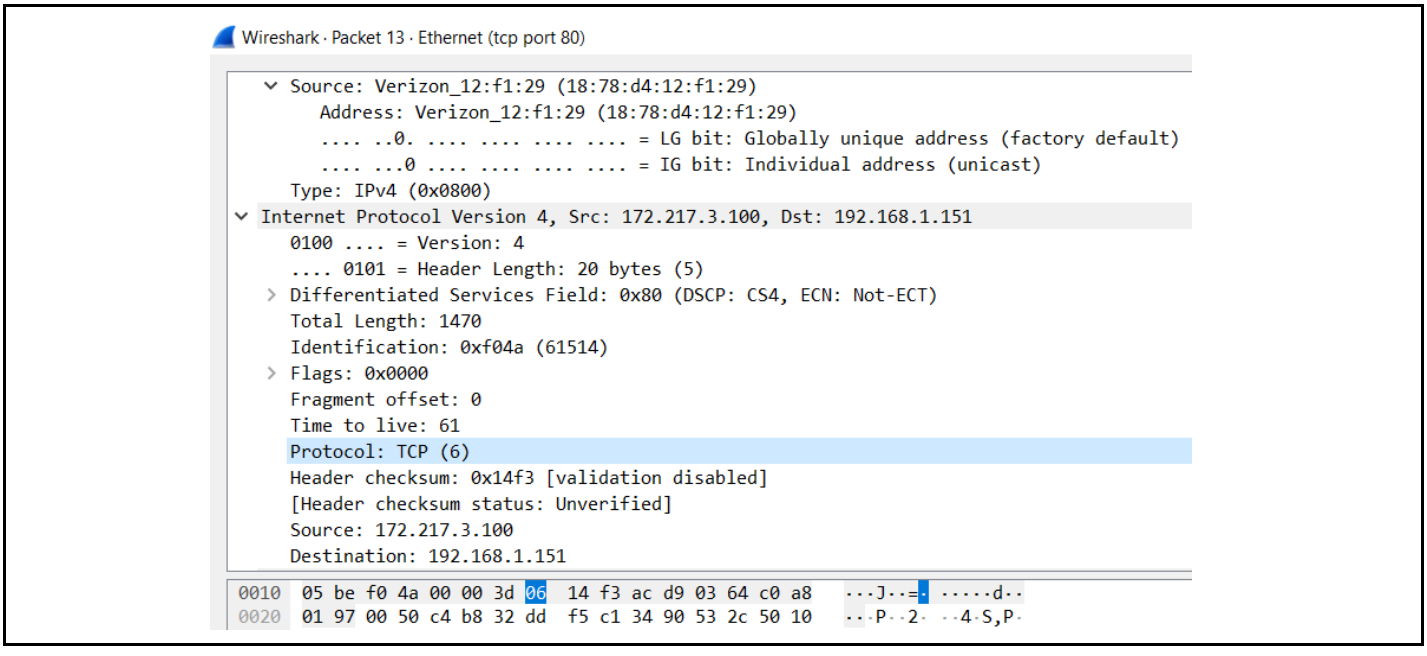
Look at the Ethernet and IP headers of a download packet in detail to answer the following questions:

1. Which Ethernet header field is the demultiplexing key that tells it the next higher layer is IP? What value is used in this field to indicate "IP"?



The protocol field is the demultiplexing key that tells it the next higher layer is IP. [08 00] is the value used to indicate IP.

2. Which IP header field is the demultiplexing key that tells it the next higher layer is TCP? What value is used in this field to indicate "TCP"?



The protocol field is the demultiplexing key that tells it the next higher layer is IP. [06] is the value used to indicate IP.