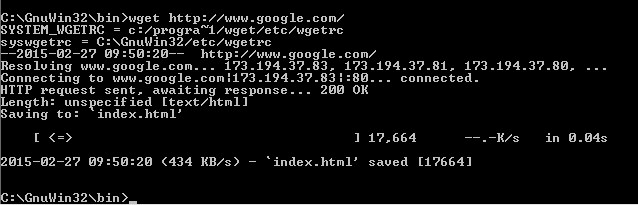
Lab Exercise – Protocol Layers

**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.

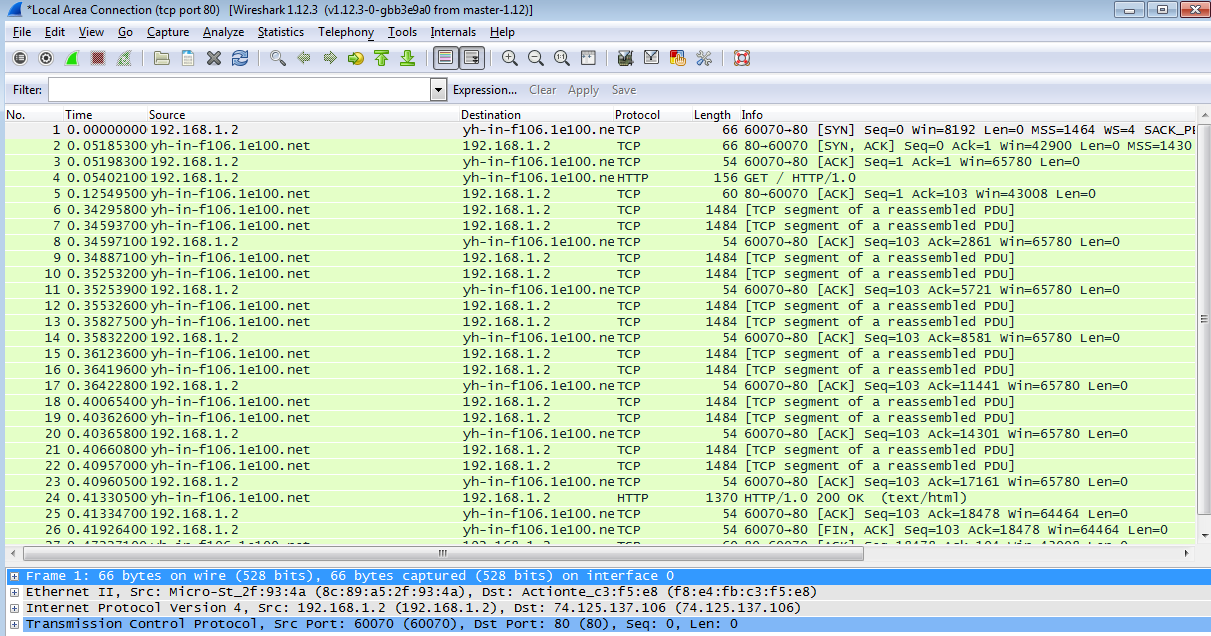


2. *Close unnecessary browser tabs and windows*. By minimizing browser activity you will stop your computer from fetching unnecessary web content, and avoid incidental traffic in the trace.

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!

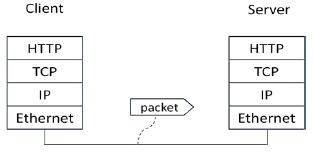


**Step 2: Inspect the Trace**

*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 inspec-tion.) Let’s have a closer look to see how the packet structure reflects the protocols that are in use.

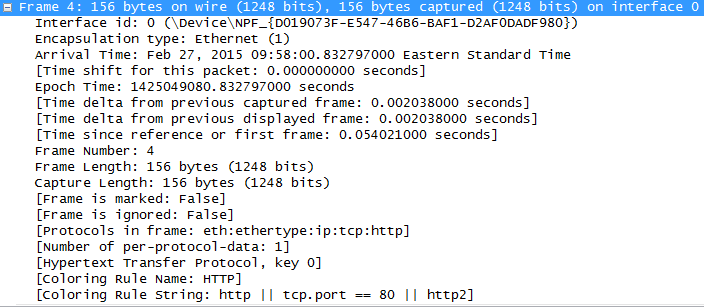
C:\Users\TheGrimAngel\Desktop\CSCI 415\Lab1\4.png

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.

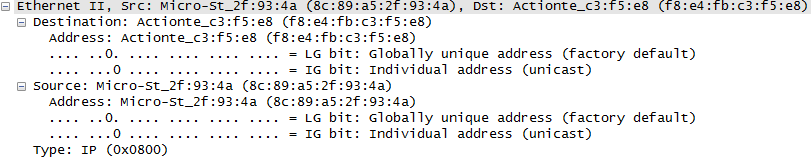


*With the HTTP GET packet selected, look closely to see the similarities and differences between it and our protocol stack as described next.* The protocol blocks are listed in the middle panel. You can expand each block (by clicking on the “+” expander or icon) to see its details.

• 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”.

C:\Users\TheGrimAngel\Desktop\CSCI 415\Lab1\6.png

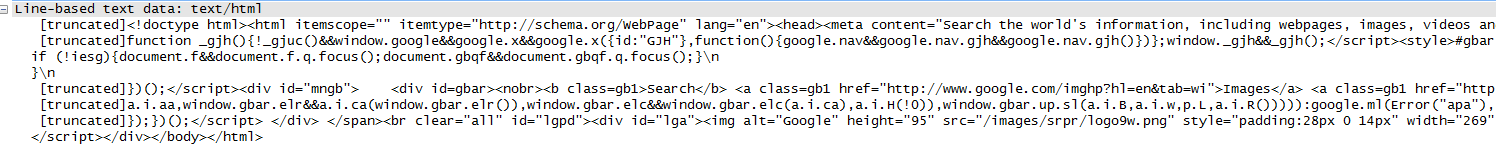
*Now find another HTTP packet, the response from the server to your computer, and look at the structure of this packet for the differences compared to the HTTP GET packet.* This packet should have “200 OK” in the Info field, denoting a successful fetch. In our trace, there are two extra blocks in the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

C:\Users\TheGrimAngel\Desktop\CSCI 415\Lab1\7.png

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

C:\Users\TheGrimAngel\Desktop\CSCI 415\Lab1\8.png

• 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** 102 bytes

**TCP**20  
bytes

**IP**20bytes

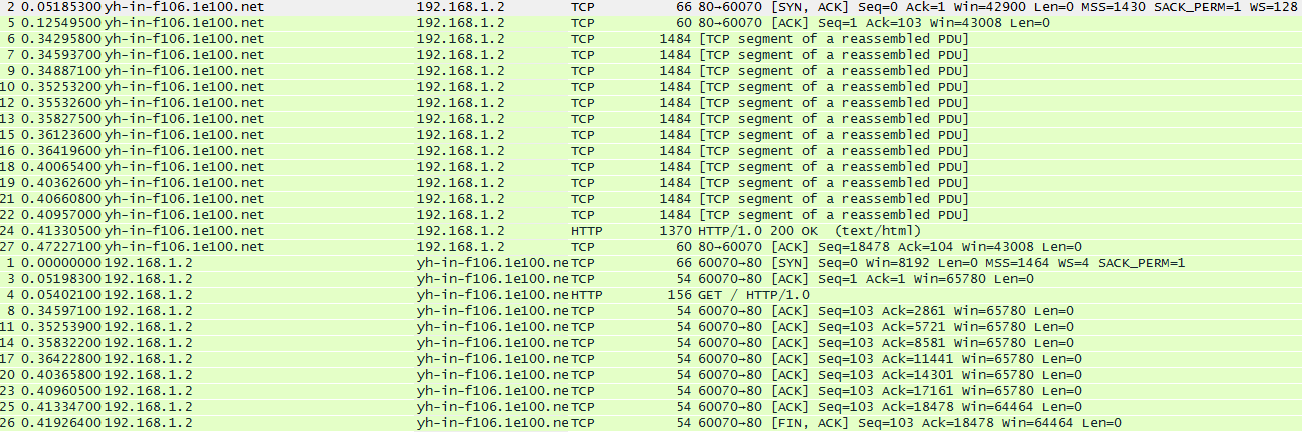
**Ethernet  
14   
bytes**

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.



|  |  |  |
| --- | --- | --- |
| **Frame** | **Ethernet/IP/TCP Overhead** | **Useful HTTP data** |
| **2** | 66 bytes | 0 bytes |
| **5** | 60 bytes | 0 bytes |
| **6** | 1484 bytes | 0 bytes |
| **7** | 1484 bytes | 0 bytes |
| **9** | 1484 bytes | 0 bytes |
| **10** | 1484 bytes | 0 bytes |
| **12** | 1484 bytes | 0 bytes |
| **13** | 1484 bytes | 0 bytes |
| **15** | 1484 bytes | 0 bytes |
| **16** | 1484 bytes | 0 bytes |
| **18** | 1484 bytes | 0 bytes |
| **19** | 1484 bytes | 0 bytes |
| **21** | 1484 bytes | 0 bytes |
| **22** | 1484 bytes | 0 bytes |
| **24** | 54 bytes | 816 bytes |
| **27** | 60 bytes | 0 bytes |
| **TOTAL** | **18048 bytes** | **816 bytes** |

18048 ÷ 18864 = .956743002 = **95.67%** of the bytes are overhead.

It would appear that for the mount of relevant data (HTTP bytes), there is just way too much overhead. However I understand that they are needed due to protocol to ensure a connection.

**Step 5: Demultiplexing Keys**

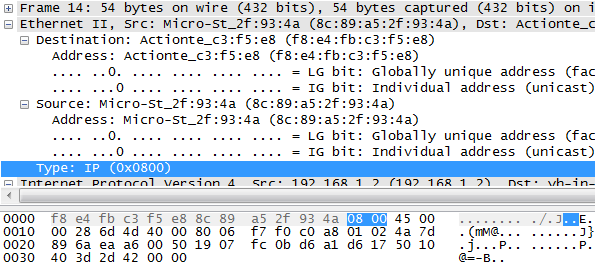
When an Ethernet frame arrives at a computer, the Ethernet layer must hand the packet that it contains to the next higher layer to be processed. The act of finding the right higher layer to process received packets is called demultiplexing. We know that in our case the higher layer is IP. But how does 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:*

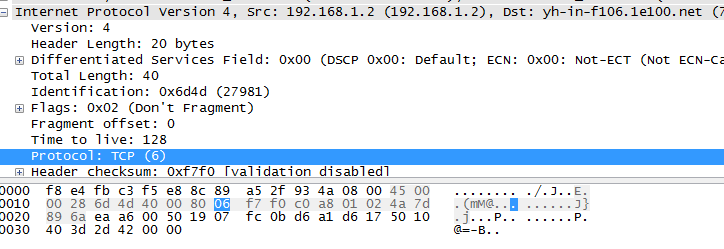
*C:\Users\TheGrimAngel\Desktop\CSCI 415\Lab1\11.png*

*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”?*



0008 (Assuming little endian? )

*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”?*

****

06