Lab Exercise – Protocol Layers



Objective

To learn how protocols and layering are represented in packets. They are key concepts for structuring networks that are covered in §1.3 and §1.4 of your text. Review those sections before doing the lab.

Requirements

Wireshark: This lab uses the Wireshark software tool to capture and examine a packet trace. A packet trace is a record of traffic at a location on the network, as if a snapshot was taken of all the bits that passed across a particular wire. The packet trace records a timestamp for each packet, along with the bits that make up the packet, from the lower-layer headers to the higher-layer contents. Wireshark runs on most operating systems, including Windows, Mac and Linux. It provides a graphical UI that shows the sequence of packets and the meaning of the bits when interpreted as protocol headers and data. It color-codes packets by their type, and has various ways to filter and analyze packets to let you investigate the behavior of network protocols. Wireshark is widely used to troubleshoot networks. You can download it from <u>www.wireshark.org</u> if it is not already installed on your computer. We highly recommend that you watch the short, 5 minute video "Introduction to Wireshark" that is on the site.

wget / curl: This lab uses wget (Linux and Windows) and curl (Mac) to fetch web resources. wget and curl are command-line programs that let you fetch a URL. Unlike a web browser, which fetches and executes entire pages, wget and curl give you control over exactly which URLs you fetch and when you fetch them. Under Linux, wget can be installed via your package manager. Under Windows, wget is available as a binary; look for download information on <u>http://www.gnu.org/software/wget/</u>. Under Mac, curl comes installed with the OS. Both have many options (try "wget --help" or "curl --help" to see) but a URL can be fetched simply with "wget *URL*" or "curl *URL*".

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.

 Pick a URL and fetch it with wget or curl. For example, "wget <u>http://www.google.com</u>" or "curl <u>http://www.google.com</u>". 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.

djw@djw-fc13:~/temp					
File Edit View Terminal Help					
<pre>[djw@djw-fc13 temp]\$ [djw@djw-fc13 temp]\$ [djw@djw-fc13 temp]\$ [djw@djw-fc13 temp]\$ [djw@djw-fc13 temp]\$ wget http://www.google.com/ 2012-02-05 12:22:24 http://www.google.com/ Resolving www.google.com 74.125.127.104, 74.125.127.105, 74.125.127.106, Connecting to www.google.com[74.125.127.104]:80 connected. HTTP request sent, awaiting response 200 0K Length: unspecified [text/html]</pre>					
<pre>Saving to: "index.html" [<=>] 14,177 2012-02-05 12:22:24 (155 MB/s) - "index.html" saved [1 [diw@diw-fc13 temp]\$</pre>	K/s in 0s				

Figure 1: Using wget to fetch a URL

- 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 network 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. Your capture window should be similar to the one pictured below, other than our highlighting. Select the interface from which to capture as the main wired or wireless interface used by your computer to connect to the Internet. If unsure, guess and revisit this step later if your capture is not successful. Uncheck "capture packets in promiscuous mode". This mode is useful to overhear packets sent to/from other computers on broadcast networks. We only want to record packets sent to/from your computer. Leave other options at their default values. The capture filter, if present, is used to prevent the capture of other traffic your computer may send or receive. On Wireshark 1.8, the capture filter box is present directly on the options screen, but on Wireshark 1.9, you set a capture filter by double-clicking on the interface.

📶 Wireshark: C	apture C	ptions						_	
Capture Interface: Lo	cal	▼ Int	tel(R) 82567LM-3 Giga	bit N	etwork Con	nection:	\Device	\NPF_{5(
IP address: 128.208.2.151 Link-layer header type: Ethernet 💌						Wireles	s Setting	js	
Capture packets in promiscuous mode				Remote Settings					
Capture packets in pcap-ng format (experimental)				Buffer size:	1	*	megabyte(s)		
Capture Filte	r: tcp	port 80							•
Capture File(s) File: Use <u>m</u> ultip	ole files			Brow	se	Display C)ptions- ate list o	f packet	s in real time
✓ Next file every 1			megabyte(s)	-	Automatic scrolling in live capt			in live capture	
Next file every		1	_	v minute(s) v files		Hide capture info dialog			
Stop capture after 1		_	file(s)		Name Resolution				
Stop Capture .			_			Enab	le <u>M</u> AC	name re	esolution
🔲 after	1		↑ ▼ P	packet(s)		V Enab	le <u>n</u> etw	ork nam	e resolution
🔲 after	1		× v	megabyte(s)					
🔲 after	1	1		minute(s) 🔻		Enable transport name resolution			ne resolution
Help]						<u>S</u> tar	t	<u>C</u> ancel

Figure 2: Setting up the capture options

- 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!

📶 Intel(R) 82567LM-3 Gigab	oit Network Connection	(tcp port 80) - Wireshark		
<u>File Edit View Go Ca</u>	apture <u>A</u> nalyze <u>S</u> tati	stics Telephon <u>y T</u> ools <u>H</u> el	p	
	e 🖥 🗶 🛃 🗄	🔍 🔅 🔅 🎝 🚡 🕹		7 🎬 🗹 🍢 💥 💢
Filter:		•	Expression Clear Apply	
Number Time So	ource	Destination	Protocol Length Info	
1 0.000000 10	orikeet.cs.washi	npz-in-f147.1e100.ne	TCP 66 56836 >	http [SYN] Seq=0 Win=64240
2 0.008106 pz	z-in-f147.1e100.	n lorikeet.cs.washing	TCP 66 http >	56836 [SYN, ACK] Seq=0 Ack=1
	orikeet.cs.wasni orikeet cs washi	npz-1n-T14/.1e100.ne	ICP 34 30830 > HTTP 166 CET / H	TTP/1 0
5 0.016754 pz	z-in-f147.1e100.	n lorikeet.cs.washing	TCP 60 http >	56836 [ACK] Seg=1 Ack=113 Wi
6 0.024502 pz	z-in-f147.1e100.	nlorikeet.cs.washing	TCP 1484 [TCP se	gment of a reassembled PDU]
7 0.024593 pz	z-in-f147.1e100.	nlorikeet.cs.washing	TCP 1484 [TCP se	gment of a reassembled PDU]
8 0.024595 pz	z-in-f147.1e100.	n lorikeet.cs.washing	TCP 1283 [TCP se	gment of a reassembled PDU]
9 0.024606 10	orikeet.cs.washi	npz-in-f147.1e100.ne	TCP 54 56836 >	http [ACK] Seq=113 Ack=4090
10 0.0246/2 pz	z-1n-T14/.1e100. z-in-f147 1e100	n Torikeet.cs.washing	TCP 1484 [TCP Sec TCP 1484 [TCP Sec	gment of a reassembled PDU
12 0 024674 pz	z-in-f147.1e100.	nlorikeet cs.washing	TCP 1200 [TCP Se	gment of a reassembled PDU
13 0.024684 10	orikeet.cs.washi	npz-in-f147.1e100.ne	TCP 54 56836 >	http [ACK] Seg=113 Ack=8186
14 0.024889 pz	z-in-f147.1e100.	n lorikeet.cs.washing	TCP 1484 [TCP se	gment of a reassembled PDU]
15 0.025004 pz	z-in-f147.1e100.	nlorikeet.cs.washing	TCP 1484 [TCP se	gment of a reassembled PDU]
16 0.025005 pz	z-in-f147.1e100.	nlorikeet.cs.washing	TCP 1290 [TCP se	gment of a reassembled PDU]
17 0.025014 IC	orikeet.cs.washi	npz-in-f147.1e100.ne	TCP 54 56836 >	http [ACK] Seq=113 Ack=1228
18 0.025091 pz	z-1n-T14/.1e100.	n lorikeet.cs.washing	ICP 1484 [ICP Sec UTTD 1262 UTTD/1	gment of a reassembled PDUJ
20 0.025104 10	orikeet.cs.washi	npz-in-f147.1e100.ne	TCP 54 56836 $>$	http [ACK] Seg=113 Ack=1492
21 0.052841 10	orikeet.cs.washi	npz-in-f147.1e100.ne	TCP 54 56836 >	http [FIN, ACK] Seq=113 Ack
22 0.060792 pz	z-in-f147.1e100.	nlorikeet.cs.washing	TCP 60 http >	56836 [АСК] Seq=14921 Ack=11
•	III			Þ.
🗄 Frame 1: 66 bytes	s on wire (528 b	its), 66 bytes captur	ed (528 bits)	
🗄 Ethernet II, Src:	: Dell_d5:10:8b	(00:25:64:d5:10:8b),	Dst: IETF-VRRP-VRID_	01 (00:00:5e:00:01:01)
Internet Protoco	l, Src: lorikeet	.cs.washington.edu (1	28.208.2.151), Dst:	pz-in-f147.1e100.net (74.125
Transmission Cont	trol Protocol, S	rc Port: 56836 (56836	6), Dst Port: http (8	0), Seq: 0, Len: 0
				4
0000 00 00 5e 00 0	01 01 00 25 64 0	15 10 8b 08 00 45 00	^% dE.	*
0020 7f 93 de 04 0	0 50 63 8b 40 0	16 00 00 00 00 00 80 02	Jr	Ξ
0030 fa f0 4d 9e 0	00 00 02 04 05 1	04 01 03 03 00 01 01	M	
0040 04 02			••	
Ready to load or capture	2	Packets: 22 Displayed: 22 Marke	ed: 0 Dropped: 0	Profile: Default

Figure 3: Packet trace of wget traffic

Step 2: Inspect the Trace

Wireshark will let us select a packet (from the top panel) and view its protocol layers, in terms of both header fields (in the middle panel) and the bytes that make up the packet (in the bottom panel). In the figure above, the first packet is selected (shown in blue). Note that we are using "packet" as a general term here. Strictly speaking, a unit of information at the link layer is called a frame. At the network layer it is called a packet, at the transport layer a segment, and at the application layer a message. Wireshark is gathering frames and presenting us with the higher-layer packet, segment, and message structures it can recognize that are carried within the frames. We will often use "packet" for convenience, as each frame contains one packet and it is often the packet or higher-layer details that are of interest.

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.

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.



Figure 4: Protocol stack for a web fetch

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 capture 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 bottom 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 ...]". Details in your capture will vary, but this block is describing more than the packet itself. Most likely, the web response was sent across the network as a series of packets that were put together after they arrived at the computer. The packet labeled HTTP is the last packet in the web response, and the block lists packets that are joined together to obtain the complete web response. Each of these packets is shown as having protocol TCP even though the packets carry part of an HTTP response. Only the final packet is shown as having protocol HTTP when the complete HTTP message may be understood, and it lists the packets that are joined together to make the HTTP response.

• The second extra block says "Line-based text data ...". Details in your capture will vary, but this block is describing the contents of the web page that was fetched. In our case it is of type text/html, though it could easily have been text/xml, image/jpeg, or many other types. As with the Frame record, this is not a true protocol. Instead, it is a description of packet contents that Wireshark is producing to help us understand the network traffic.

trace-protocol-layers.pcap					
<u>File Edit View Go Capture Analyze Statistics Telephony Tools Internals Hel</u>	p				
	🕂 🔍 🍳 📅 👪 🔟 畅 % 💢				
Filter: Expression.	Clear Apply				
Numb Time Source Destination	Protocol Length Info				
6 0.025010 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1484 [TCP segment of a r				
7 0.025101 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1484 [TCP segment of a r				
9 0.025102 p2-in-f104.10100.net lorikeet.cs.washing	ton.edu TCP 1282 [TCP segment of a r				
10 0.025105 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1484 [TCP segment of a r				
11 0.025106 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1290 [TCP segment of a r				
12 0.025123 lorikeet.cs.washington.edu pz-in-f104.1e100.net	t TCP 54 56816 > http [ACK]				
13 0.025203 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1484 [TCP segment of a r				
14 0.025204 pz-1n-f104.10100.net TorTkeet.cs.washingt	TCP 54 56816 Shttp [ACK]				
16 0.025335 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1290 [TCP segment of a r				
17 0.025336 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu TCP 1484 [TCP segment of a r				
18 0.025337 pz-in-f104.1e100.net lorikeet.cs.washingt	ton.edu HTTP 1281 HTTP/1.0 200 ОК (t				
19 0.025349 lorikeet.cs.washington.edu pz-in-f104.1e100.net	t тср 54 56816 > http [АСК]				
20 0.028799 Torikeet.cs.washington.edu pz-in-f104.1e100.net	t TCP 54 56816 > http [FIN,]				
21 0.030850 p2-11-1104.10100.net Tor Reet. CS. washing	ton.edu TCP 60 HCCp > 56816 [ACK] +				
	, ,				
Frame 18: 1281 bytes on wire (10248 bits), 1281 bytes capture	20 (10248 DITS)				
	25.127.104). Dst: lorikeet.cs.washington.e				
Transmission Control Protocol, Src Port: http (80), Dst Port:	56816 (56816), Seq: 13711, Ack: 113, Len				
⊞ [11 Reassembled TCP Segments (14937 bytes): #6(1430), #7(1430)), #8(1228), #9(1430), #10(1430), #11(123				
🗄 Hypertext Transfer Protocol					
⊞ Line-based text data: text/html					
۲. III III III III III III III III III I	- F				
0000 00 25 64 d5 10 8b 00 18 74 15 44 80 08 00 45 00 .%d	t.DE.				
0010 04 f3 eb a2 00 00 2f 06 4e 16 4a 7d 7f 68 80 d0	/. N.J}.h				
0020 02 97 00 50 dd 10 24 10 19 10 49 dd 4e 70 50 19P 0030 00 5a 18 6e 00 00 2e 48 69 73 74 6f 72 <u>79 2e 69</u> .z.n	.H istory.i				
0040 6e 69 74 69 61 6c 69 7a 65 28 27 2f 27 29 7d 29 nitial	liz e('/')})				
0050 <u>30 69 66 28 67 67 67 67 67 66 52 26 6a 26 26 67 67</u> ; it (go	oog le.j&&g0				
Frame (trame), 1281 bytes Packets: 21 Displayed: 21 Marked: 0 Load tir	me: 0:00.002 Profile: Default				

Figure 5: Inspecting a HTTP "200 OK" response

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.

To work out sizes, observe that when you click on a protocol block in the middle panel (the block itself, not the "+" expander) then Wireshark will highlight the bytes it corresponds to in the packet in the lower panel and display the length at the bottom of the window. For instance, clicking on the IP version 4 header of a packet in our trace shows us that the length is 20 bytes. (Your trace will be different if it is IPv6, and may be different even with IPv4 depending on various options.) You may also use the overall packet size shown in the Length column or Frame detail block.

Turn-in: Hand in your packet drawing.

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 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 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.

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.

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"?
- 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"?

Turn-in: Hand in your answers to the above questions.

Explore on your own

We encourage you to explore protocols and layering once you have completed this lab. Some ideas:

- Look at a short TCP packet that carries no higher-layer data. To what entity is this packet destined? After all, if it carries no higher-layer data then it does not seem very useful to a higher layer protocol such as HTTP!
- In a classic layered model, one message from a higher layer has a header appended by the lower layer and becomes one new message. But this is not always the case. Above, we saw a trace in which the web response (one HTTP message comprised of an HTTP header and an HTTP pay-load) was converted into multiple lower layer messages (being multiple TCP packets). Imagine that you have drawn the packet structure (as in step 2) for the first and last TCP packet carrying the web response. How will the drawings differ?
- In the classic layered model described above, lower layers append headers to the messages passed down from higher layers. How will this model change if a lower layer adds encryption?
- In the classic layered model described above, lower layers append headers to the messages passed down from higher layers. How will this model change if a lower layer adds compression?

[END]