Lab 1: Wireshark Warm-Up

Objective: Get Comfortable with the Lab Process.

Completion of this lab requires many of the skills you will use throughout this lab book. If you are a bit shaky on any answer, take time when reviewing the answers on page 9 to ensure you have mastered the necessary skill(s).

Trace File: wwb001-http.pcapng

Skills Covered in this Lab

In this lab, you will have a chance to work with many key functions in Wireshark. The answers to this lab demonstrate how to use functions, including, but not limited to:

- Perform address detection
- Measure DNS response time
- Add and sort columns to detect the highest/lowest field values
- Determine the trace file capture location
- Analyze conversation statistics
- Apply display filters on request and response packet values
- Detect HTTP host names
- Filter on TCP flags with `&&` (`and`) and == (`eq`)
- Use automatic display filter packet counts
- Compare TCP flag summaries in display filter results
- Detect HTTP request URI values with display filters
- Apply display filters to detect HTTP error responses
- Use the packet relationship indicator
- Evaluate port usage statistics in TCP conversations
- Identify TCP handshake option definitions
- Determine highest and lowest HTTP response times
- Change TCP preference settings
- Reassemble HTTP objects

- Identify TCP and UDP stream numbers and counts using display filtering
- Determine throughput
- Identify the capture application
- Analyze spurious retransmissions
- Use the right-click conversation filtering method
- Analyze the TCP Calculated Window Size value
- Display filter based on subnet/CIDR (Classless Interdomain Routing) values
- Identify the TCP conversation Initial Round-trip Time (iRTT) value
- Detect IP fragmentation
- Measure peak bandwidth utilization

Lab 1 - Q1. **What is the IP address of the DNS/HTTP client in this trace file?**

_____

Lab 1 - Q2. **What is the IP address of the DNS server?**

_____

Lab 1 - Q3. **What DNS response time is seen in this trace file?**

_____

Lab 1 - Q4. **Do you think this trace was taken closer to the HTTP client or closer to the HTTP servers?**

_____

Lab 1 - Q5. **What are the IP addresses of the HTTP servers to which the client successfully connected?**

_____

Lab 1 - Q6. **What are the HTTP host names of the target HTTP servers?**

_____

Lab 1 - Q7. **How many TCP SYN packets did the client send to the HTTP servers?**

_____

Lab 1 - Q8. **What Uniform Resource Identifier (URI) does the client request first in this trace file?**

_____

Lab 1 - Q9. **What HTTP error response(s) are seen in this trace file?**

_____

Lab 1 - Q10. What requested web object could not be found on the HTTP server?

_____

Lab 1 - Q11. What TCP port numbers did the client open to communicate with the HTTP server?

_____

Lab 1 - Q12. What TCP options are supported by the client?

_____

Lab 1 - Q13. What TCP options are supported by the HTTP servers?

_____

Lab 1 - Q14. What is the slowest HTTP response time seen in this trace file?

_____

Lab 1 - Q15. What is the fastest HTTP response time seen in this trace file?

_____

Lab 1 - Q16. What words are seen in the *featureb.jpg* image?

_____

Lab 1 - Q17. What is the average bits-per-second throughput rate in this trace file?

_____

Lab 1 - Q18. What application was used to capture this trace file?

_____

Lab 1 - Q19. What is the largest Calculated Window Size advertised by the client?

_____

Lab 1 - Q20. What is the fastest initial round-trip time seen in this trace file?

_____

Lab 1 - Q21. How many UDP streams are in this trace file?

_____

Lab 1 - Q22. How many TCP streams are in this trace file?

_____

Lab 1 - Q23. What is the purpose of TCP stream 7?

_____

Lab 1 - Q24. How many packets allow IP fragmentation in the IP header?

_____

Lab 1 - Q25. How many times does the packets-per-second rate reach over 125 in this trace file?

_____

[This page intentionally left blank.]

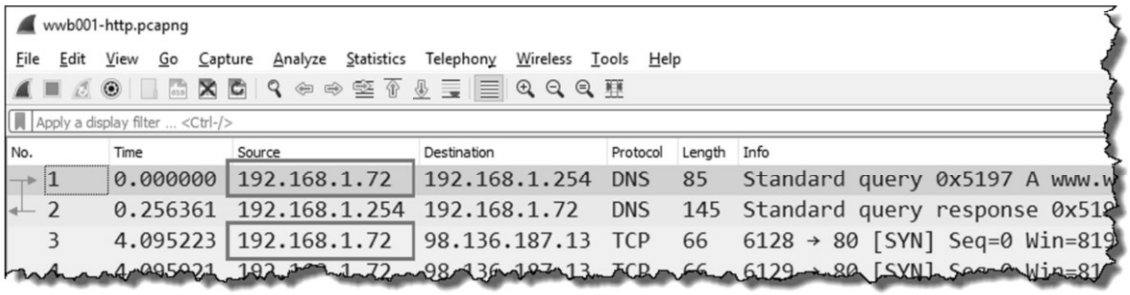
Lab 1 Solutions

Trace File: wwb001-http.pcapng

Lab 1 - A1. The IP address of the DNS/HTTP client is 192.168.1.72.

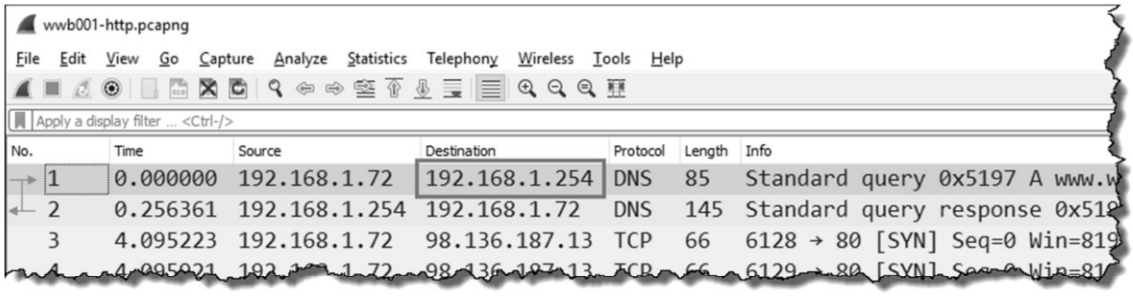
Frame 1 is a DNS request sent from 192.168.1.72. Frame 3 is a TCP SYN packet sent from 192.168.1.72 to an HTTP server at port 80. That is the IP address of the DNS/HTTP client in this trace file.

You could also look for GET requests to see the IP address of the HTTP client in this trace file.



Lab 1 - A2. The IP address of the DNS server is 192.168.1.254.

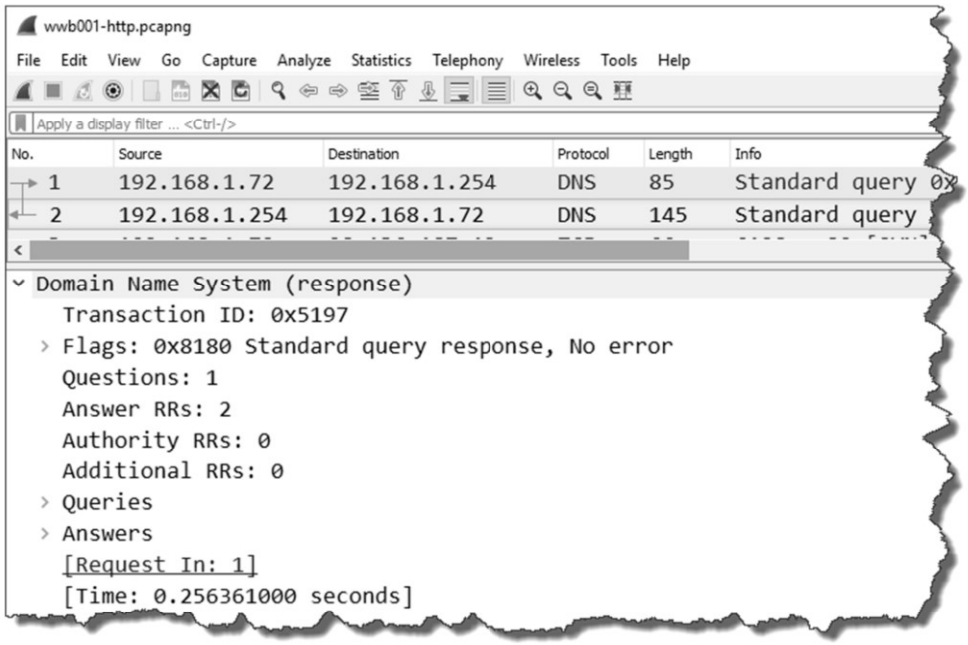
Frame 1 is a DNS packet addressed to 192.168.1.254. If the DNS traffic did not occur right at the top of the trace file, you could apply a display filter for `dns` to find the IP address of the DNS server.



Lab 1 - A3. The DNS response time seen in this trace file is 0.256361[3] seconds.

You can look at the *Time* column for this value since the DNS request and response packets are the first two packets of the trace file. The *Time* column indicates the DNS response arrived 0.256361 seconds (just over 256 ms) after the DNS request.

Alternately, you can expand the DNS section of the DNS response packet and look for the *[Time:]* value. This is Wireshark's measurement from the related DNS request to this DNS response in the trace file.



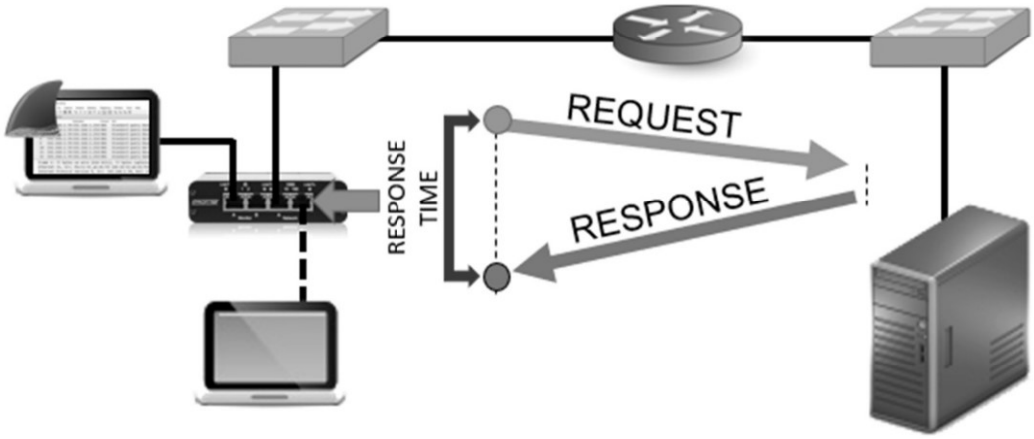
If you need to analyze a lot of DNS response times in a trace file, consider right-clicking on this Time *field in any DNS response packet and selecting* Apply as Column*. Once the field is added as a column in the Packet List pane, you can click twice on the column heading to sort DNS response times from high to low.*

Wireshark can measure the response times for several applications including DNS, HTTP, and SMB/SMB2. The response time measurement is based on when the related request is seen and is contained in **.time* fields. For example, the *dns.time* value of 0.256361 seconds is a measurement from the DNS request in frame 1 to the DNS response in frame 2.

[3] Throughout this book, we will only focus on time values to the millisecond-level, removing any trailing zeroes.

Where you capture the traffic does matter.

These response time values will be more accurate if you capture close to the client or even on the client system. If you capture closer to the server, your time does not include the time required to get the request to that point in the network or the time required to return the response to the client from that point in the network.



In the image above, I have placed a TAP[4] (Test Access Point) between the client and the upstream switch.

Although you might be tempted to use port spanning on switches, please consider placing a TAP on your network instead.

Too often I see Wireshark's Expert indicate "ACKed segment that wasn't captured" on trace files captured using a port spanning process. This means that the trace file contains an ACK that doesn't have a corresponding data packet. This is what you see when a switch drops data packets during the span process.

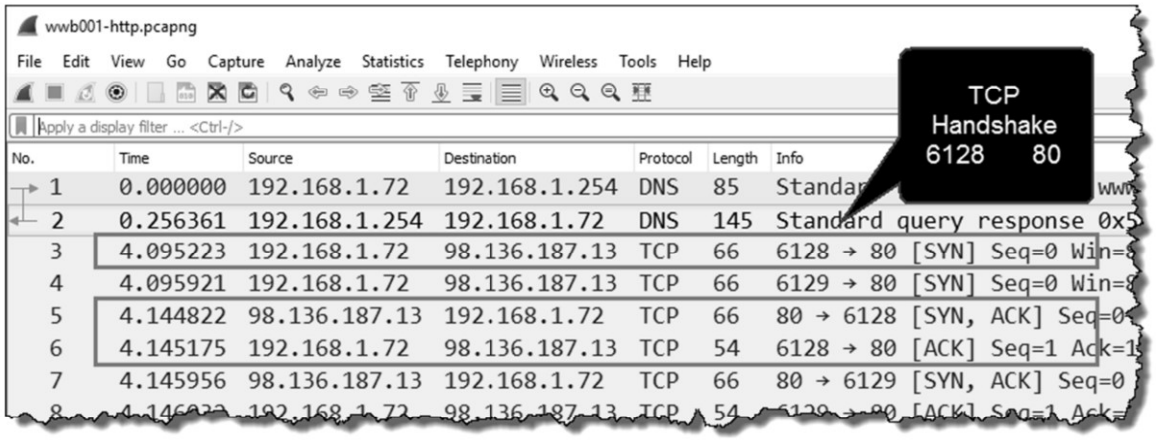
When I see these indications, I throw away that trace file. I cannot perform analysis on a "partial picture" of the communications. It can be very misleading and waste a LOT of time.

TAPs are well worth the investment – get them in place before you need them!

[4] The tap in this image is a Portable Gigabit Copper TAP from Profitap (*www.profitap.com*). Visit Profitap's website to check out some great white papers and case studies from many of my friends in the network analysis industry.

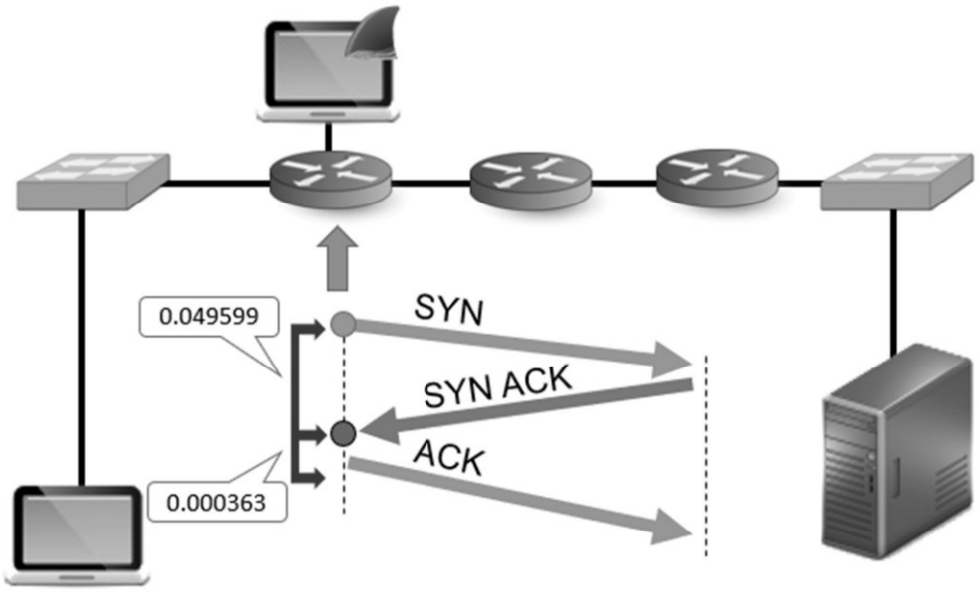
Lab 1 - A4. This trace file was taken close to (if not directly on) the client, 192.168.1.72.

We can use the TCP handshake to determine where the capture was taken. In the image below, notice that frames 3, 5, and 6 depict the TCP handshake between 192.168.1.72 and 98.136.187.13 using ports 6128 and 80.



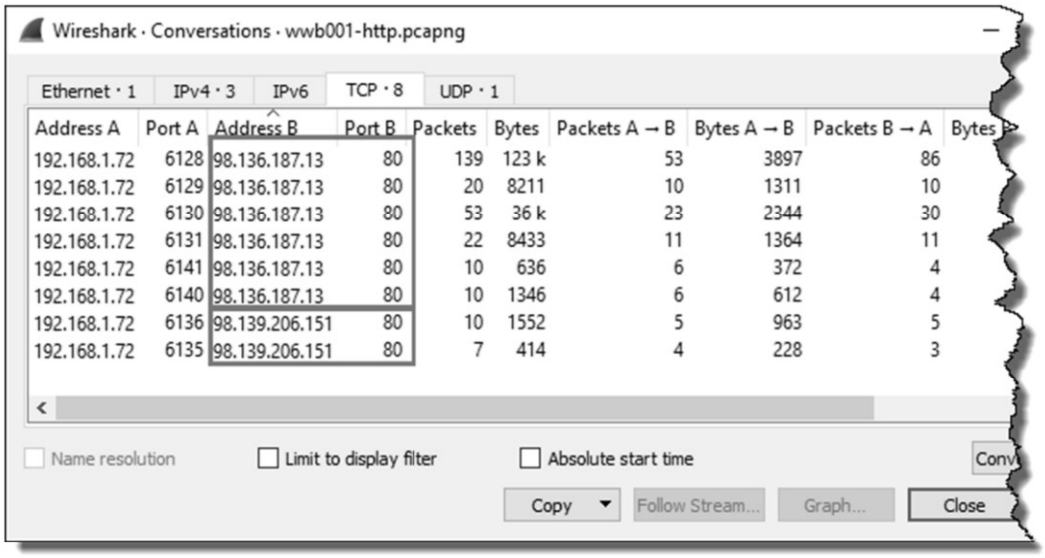
Frame #	Time Value	Packet Type	Traffic Direction	Delta
Frame 3	4.095223	SYN	Client to Server	N/A
Frame 5	4.144822	SYN/ACK	Server to Client	0.049599
Frame 6	4.145175	ACK	Client to Server	0.000353

When the delta time is remarkably smaller between the SYN/ACK and ACK (2nd and 3rd packets of the TCP handshake), the capture was likely performed close to the system that sent the SYN packet.

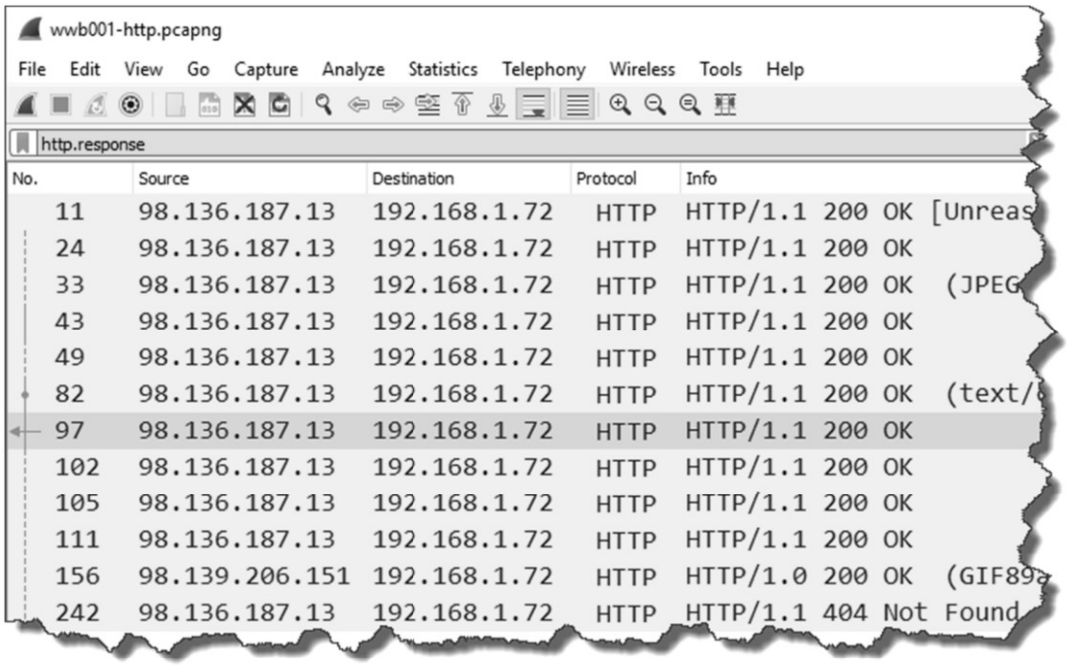


Lab 1 - A5. The IP addresses of the HTTP servers to which the client successfully connected are 98.136.187.13 and 98.139.206.151.

There are many ways to obtain this information. You can select *Statistics | Conversations* and look under the *TCP* tab. For clarity, sort the *Address B* column and you can clearly see the two HTTP servers listed with port 80.



Alternately, you can apply a display filter for `http.response` to view all HTTP responses and focus on the *Source* field, as shown below.



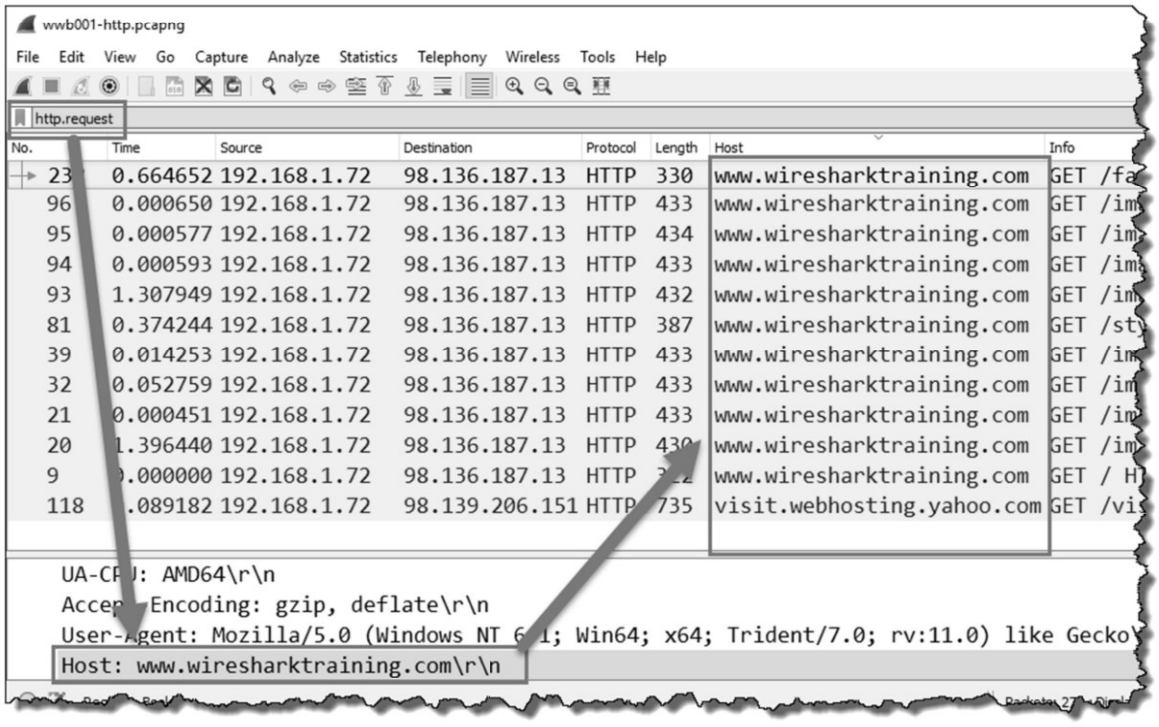
In the previous image, the TCP setting Allow subdissector to reassemble TCP streams *is off. If the setting is enabled, you will see the same number of packets, but the HTTP response code will be associated with the last packet of the object download, rather than the first packet of the object download.*

As of Wireshark v3, we have a No Reassembly *profile automatically installed. If you keep your TCP reassembly on in your other profiles, you can quickly switch to the* No Reassembly *profile when desired.*

Lab 1 - A6. The HTTP host names of the target HTTP servers are *www.wiresharktraining.com* and *visit.webhosting.yahoo.com*.

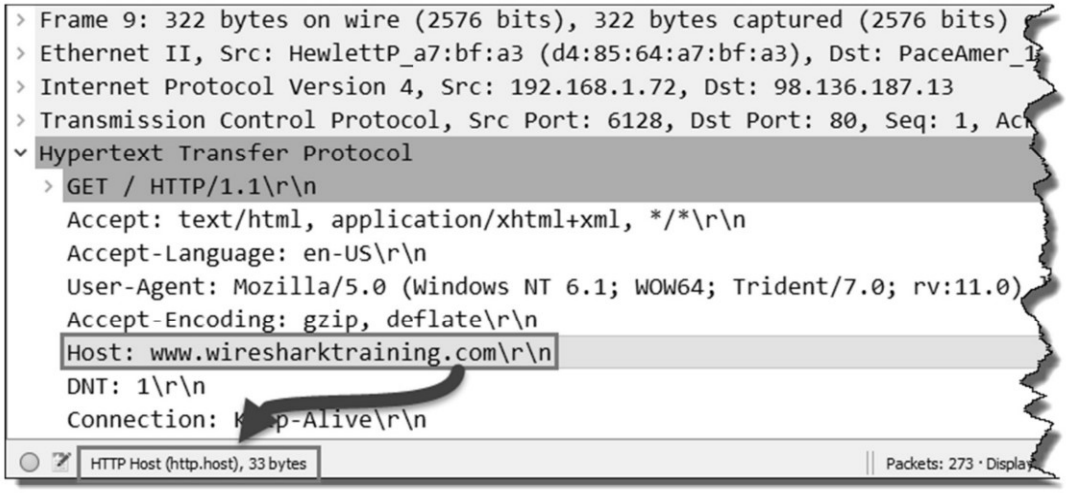
HTTP requests contain a *Host* field (`http.host`). Although you could scroll through the packets to look for this field separately in each HTTP request, there is almost always a better way to locate something than scrolling.

In this trace file, the first HTTP request is in frame 9. You could do a small bit of scrolling to locate that frame or you could apply a display filter for `http.request`.



Just 12 packets match this filter. Each of these packets has an `http.host` field in the HTTP section of the Packet Details pane.

Expand the HTTP section in the Packet Details pane, right-click on the *Host* field and select *Apply as Column*. Now you can clearly see the *Host* field values contained in the trace file. In the previous image, we've sorted on the HTTP *Host* column to make it easier to identify the various host names.



The only time I scroll through a trace file is when I want to understand how a protocol or application works.

In that case, I examine each packet of the process looking for the startup routine, request format, response format, timing, packet sizes, packet counts, closing process, etc.

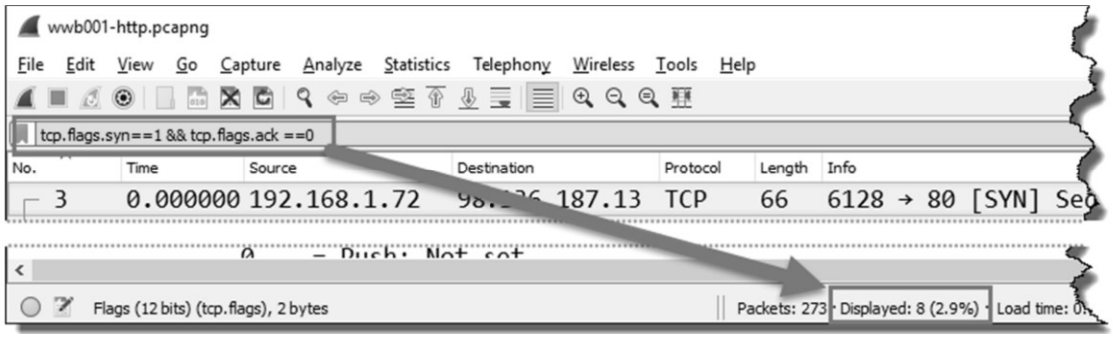
As you go through this workbook, you will see that I make a TON of display filters to find items of interest. When I am learning a protocol or application, I create filters and buttons for anything I would like to locate quickly later.

When troubleshooting that application or protocol, I know what I am looking for – whether it is an error response, timing issue, unexpected packet counts, or another anomaly. You need to know how things ***should*** *work before trying to find the cause of problems.*

Lab 1 - A7. The client sent eight TCP SYN packets to the HTTP servers.

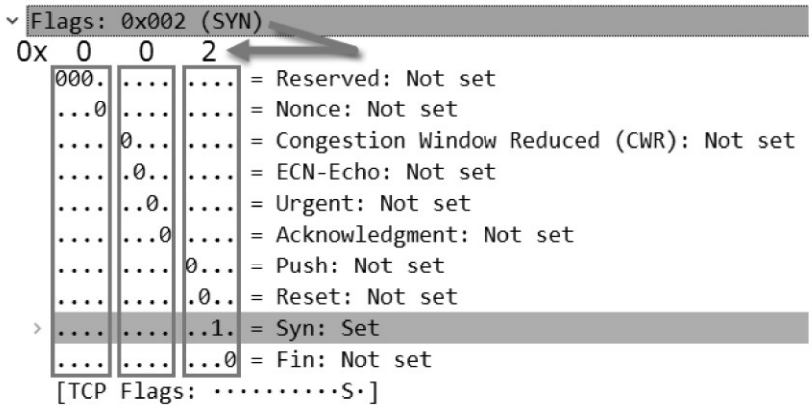
Note that we are not counting SYN/ACK packets, just SYN packets. So, let's consider creating a filter for what we really want to see; packets that only have the SYN bit set and not the ACK bit.

```
tcp.flags.syn==1 && tcp.flags.ack==0
```



You may have found the answer another way.

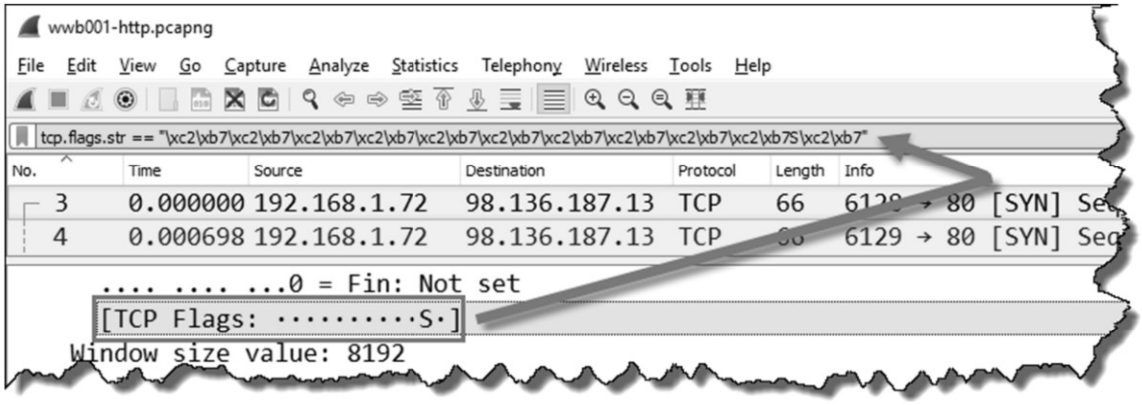
In classes, one or more of my students often suggests that we create and apply a filter based on the Flags summary line (`tcp.flags==0x002`), as shown below.



Using `tcp.flags==0x002`, you would still see 8 packets in this case, but the filter has a flaw. This *Flags* summary line includes the *Reserved, Nonce, Congestion Window Reduced (CWR),* and *ECN-Echo bits*, as well as the TCP *Urgent*, *Acknowledgment*, *Push*, *Reset*, and *Fin* bits. Your `tcp.flags==0x002` filter will not match TCP SYN packets where any bits in the *Flags* field are different from how they are set in this packet.

You could build a display filter based on the TCP Flags line below the FIN flag. Simply right-click on this TCP Flags summary line and select *Apply as Filter*. Wow! What an

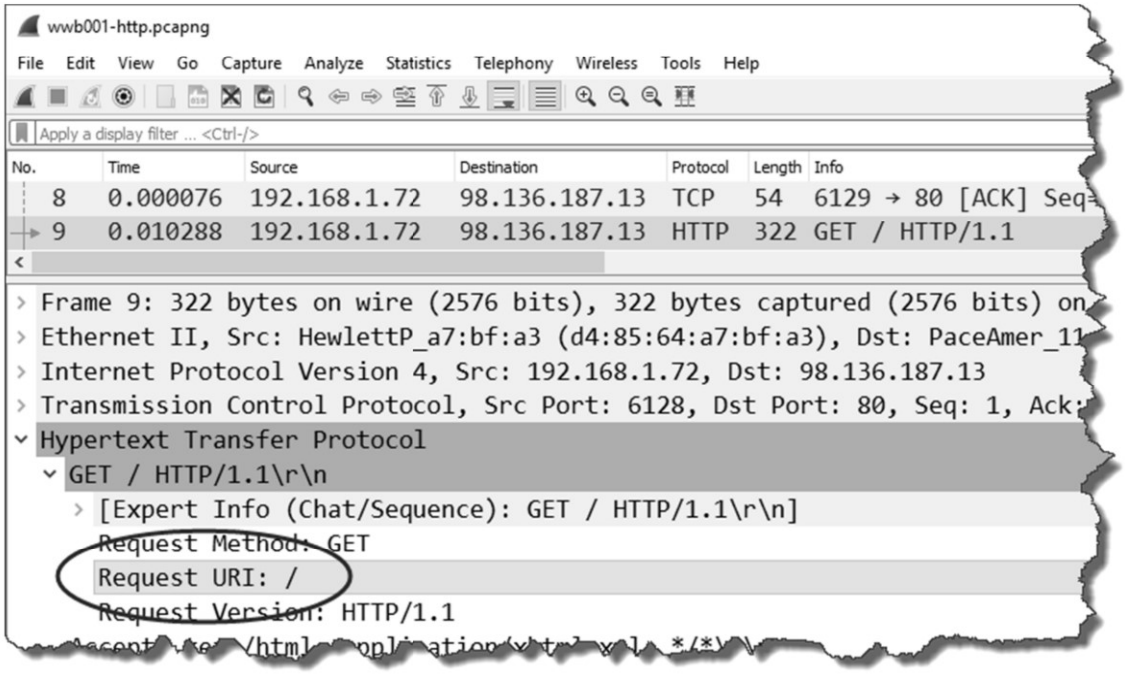
ugly filter! This filter is literally looking for UTF-8 middle dots in all positions except for an "S" in the next to last position.



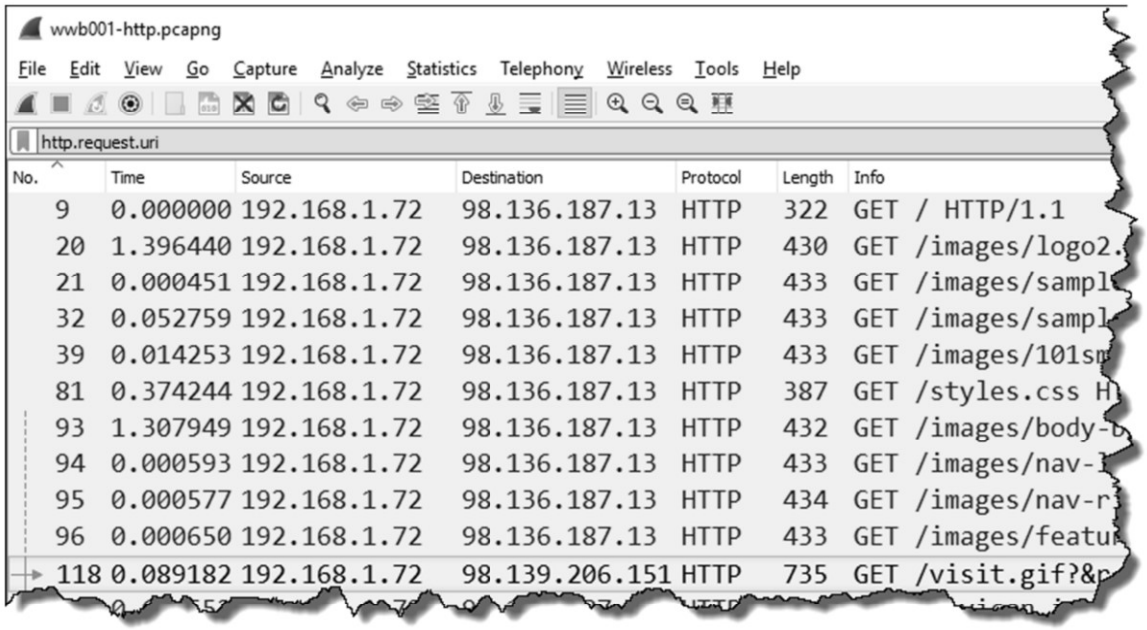
In Wireshark, there are often many ways to find specific packets. Sometimes, however, using summary lines from the Packet Details window will not yield the desired results.

Lab 1 - A8. The first Uniform Resource Identifier (URI) requested by the client is "/" (the default page).

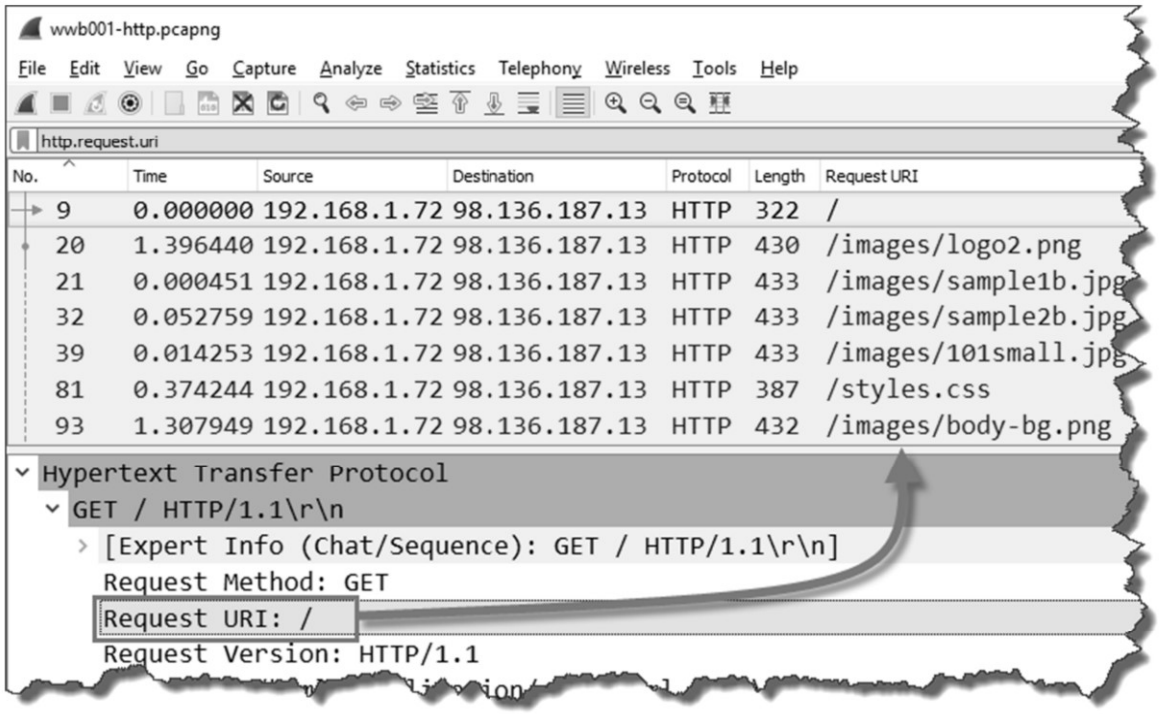
You could simply look at the Packet List pane and find this answer quickly by looking at the *Info* column of frame 9.



Alternately, you could use a `http.request.uri` display filter to view only packets that contain this field, as shown below.



Another option would be to add the `http.request.uri` field as a column in the Packet List pane, as shown below. Note that you will need to expand the request section of the HTTP packet to see the *Request URI* field.



You can probably already tell that adding columns to the Packet List pane is a powerful feature. If you don't have the screen space to handle all the columns you want to add, right-click on a column and deselect it to hide it temporarily. This powerful feature justifies the need for more than one monitor on your desk! You will be a much more efficient analyst if you have more space available to add columns and keep statistics windows open as needed.

Lab 1 - A9. There is a single 404 Not Found HTTP error response in frame 242.

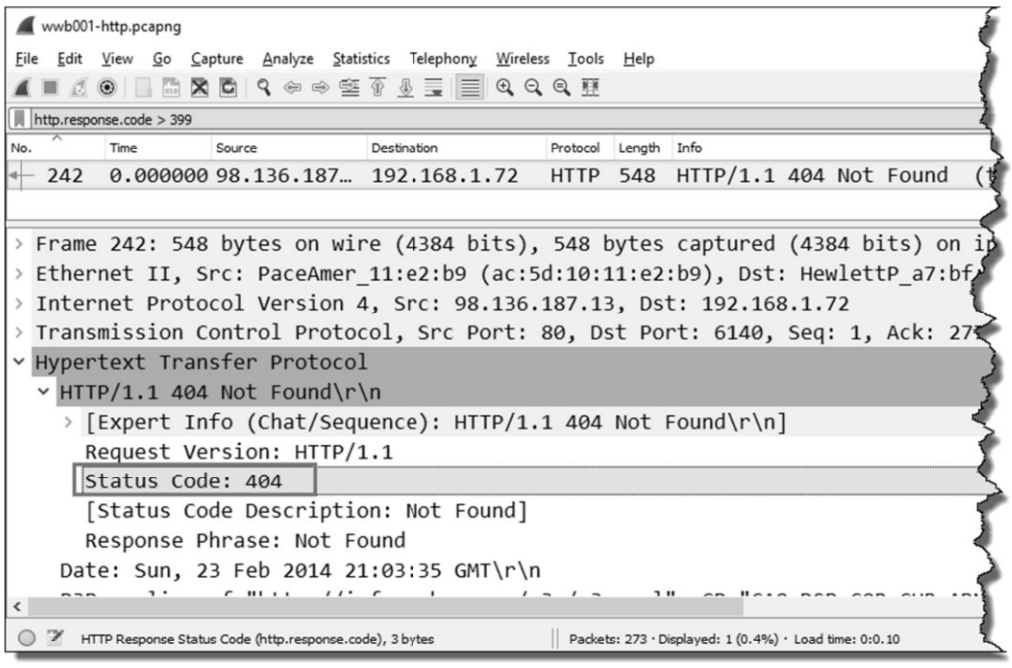
This is where a display filter will quickly help you locate application error responses. You need to know two things:

1. What is the response code field name?
2. What response codes constitute errors?

In HTTP, any response code greater than 399 indicates an error.

4xx	Client Errors
5xx	Server Errors

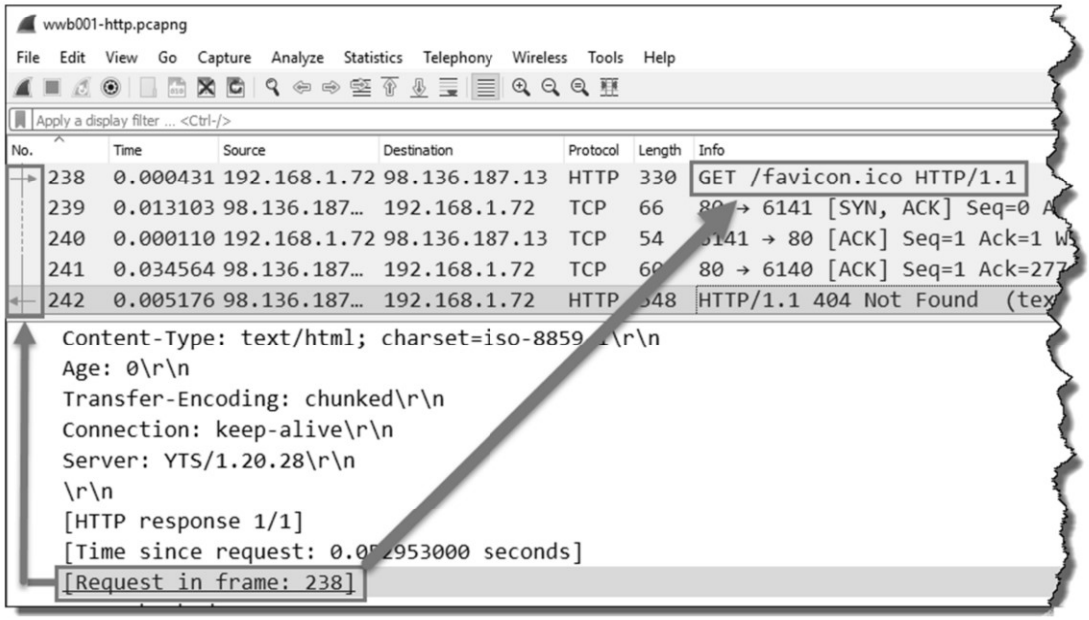
In the image below, we applied a display filter for `http.response.code > 399`. There's the 404 response in frame 242.



Lab 1 - A10. The web object that could not be found on the HTTP server is *favicon.ico*.

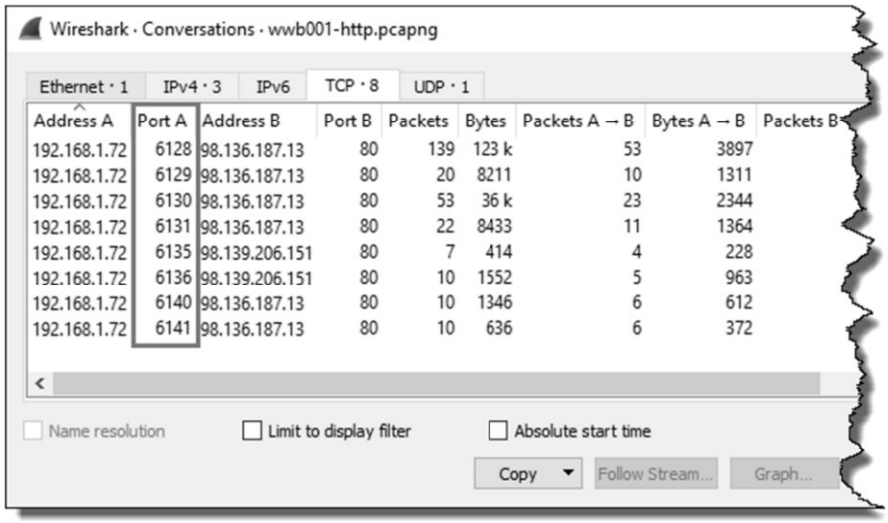
Now we want to find the URI value that triggered the 404 response in frame 242.

Wireshark detects related packets and denotes that information within the Packet Details pane and to the left of the frame number value in the *No.* column of the Packet List pane, as shown below.



Lab 1 - A11. The HTTP client opened TCP port numbers 6128, 6129, 6130, 6131, 6135, 6136, 6140, and 6141 to connect to the HTTP servers.

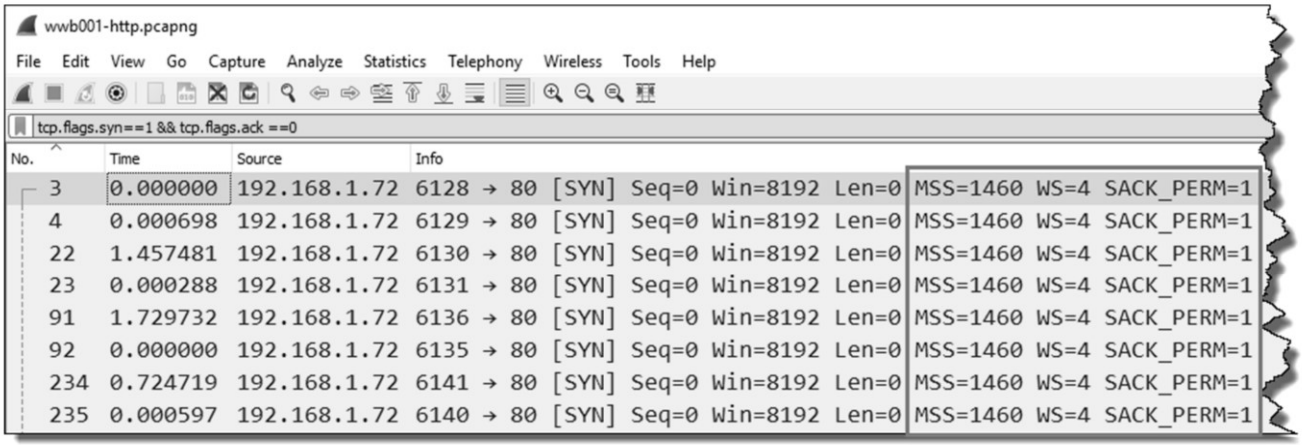
This information is readily available in the *Statistics | Conversations | TCP* view, as shown in the following image.



Lab 1 - A12. The client supports a Maximum Segment Size (MSS) of 1460 bytes, Window Scaling with a multiplier of 4, and Selective acknowledgment (SACK).

This information is contained in the client's SYN packets. You don't need to look inside those SYN packets in the Packet Details pane, however. Just look at the *Info* column of the Packet List pane. The *Info* column contains all the TCP options information, so you can quickly determine the capabilities of both sides of a TCP connection.

In the image below, we applied the display filter `tcp.flags.syn==1 && tcp.flags.ack==0` to view just the SYN packets in the trace file. (Note that we removed some columns from view to fit the desired information in the image.[5])



As much as I love to delve into packets and spend time looking at their contents, time is usually of the essence. It's important to find the issues as fast as possible so we can "point the finger" to the problem area on the network.

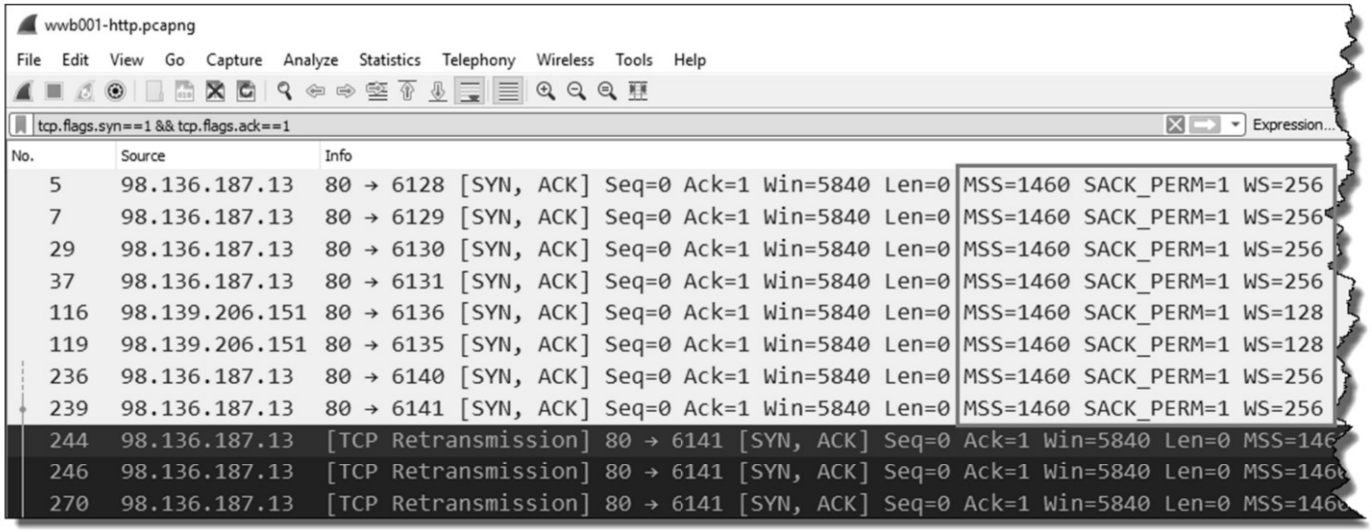
The Info *column offers so much information about the TCP handshake packets that I rarely need to look into the SYN, SYN/ACK, or ACK packets.*

[5] You should have at least two monitors on which to analyze traffic. As you add more and more columns to your profile, you will find that one monitor slows you down tremendously!

Lab 1 - A13. The TCP options supported by the HTTP servers are:

- 98.136.187.13: Maximum Segment Size (MSS) of 1460 bytes, Window Scaling with a multiplier of 256, and Selective Acknowledgment (SACK)
- 98.139.206.151: Maximum Segment Size (MSS) of 1460 bytes, Window Scaling with a multiplier of 128, and Selective Acknowledgment (SACK)

To obtain this information quickly, we can look at the second packet of the TCP handshake, the SYN/ACK packet. In the image below, we applied the display filter `tcp.flags.syn==1 && tcp.flags.ack==1` to view the server capabilities.



We will go over this later in this workbook, but now is a good time to mention that a server will not advertise certain options in its SYN/ACK packet if the client didn't mention those options in the SYN packet.

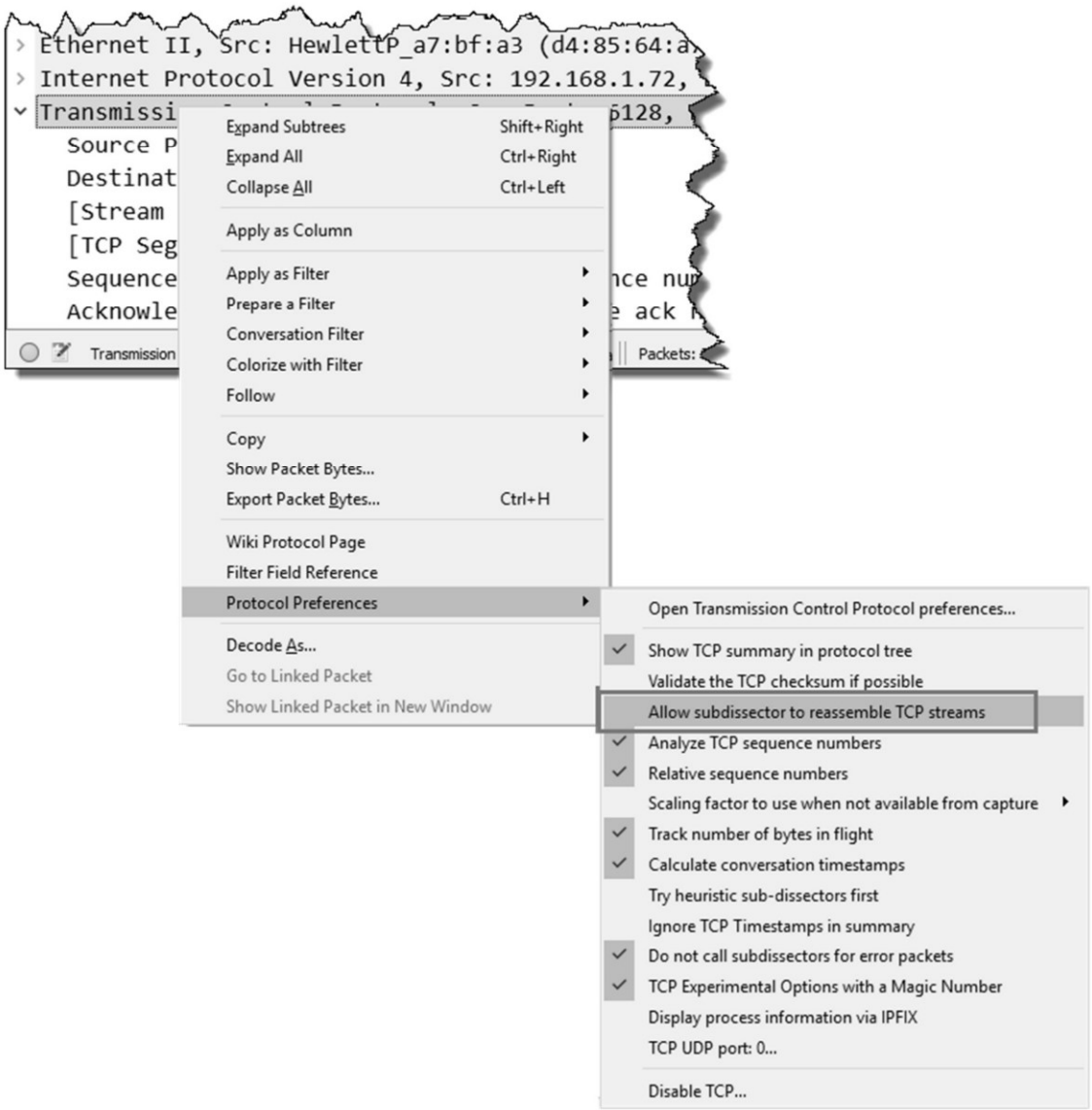
For example, if the client had not advertised that it supported SACK (SACK_PERM=1), the server would not have included that in it's SYN/ACK.

This is true for options such as SACK, Window Scaling, and TCP Timestamps. It is not true for the Maximum Segment Size (MSS) option. Each side of the TCP connection states its MSS value independently.

This means that if you want to know if a connection will support SACK, Window Scaling, and TCP Timestamps, you can just look at the SYN/ACK packet.

Lab 1 - A14. The slowest HTTP response time seen in this trace file is 0.109500 seconds (frame 156).

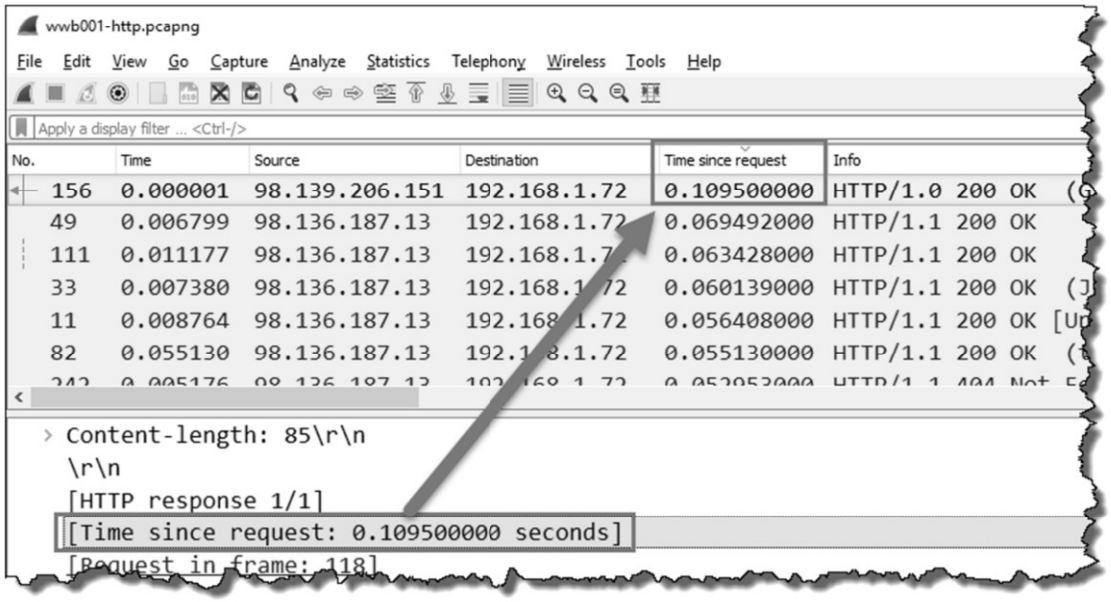
In order to measure the HTTP response time properly, we must disable the TCP preference setting *Allow subdissector to reassemble TCP streams*. Why? Well, with this setting enabled, Wireshark measures response time from the HTTP request to the **end** of the object download.



We want to measure the time from the HTTP request to the HTTP response (with the numerical code, such as *200 OK*). With this preference setting disabled, Wireshark does just that.

Right-click on any TCP header in the Packet Details pane and select *Protocol Preferences.* If the TCP reassembly preference is enabled, just click on it to disable it.

Any time you are looking for the largest or smallest number in a field (such as the *Time since request* field), consider adding that field as a column in the Packet List pane and then sorting the column.



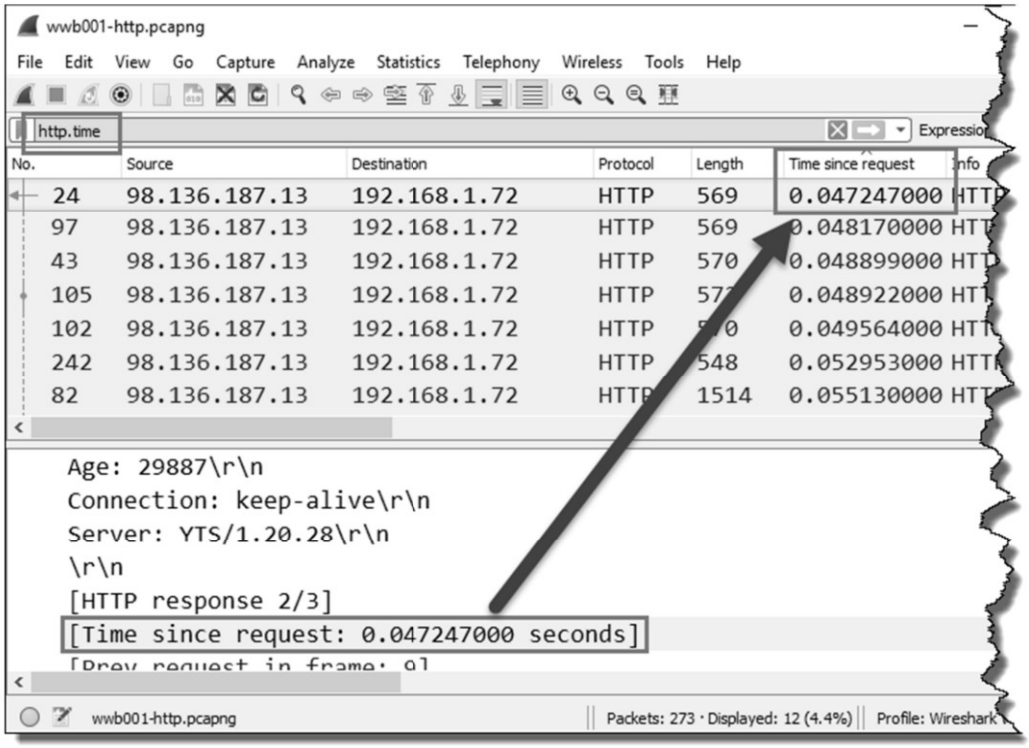
Although it might be tough to see it, there is a small sort order indicator at the top of the columns.

In the image above, you might be able to see the small down arrow above the Time since request *column heading.*

As you get older, these little itty bitty indications get tougher and tougher to see… sigh.

Lab 1 - A15. The fastest HTTP response time seen in this trace file is 0.047247 seconds.

Again, we sorted the `http.time` field (*Time since request*). This time, however, we added a filter to view only packets that contain the HTTP *Time since request* field. This filter only displays HTTP responses that have a corresponding HTTP request in the trace file.



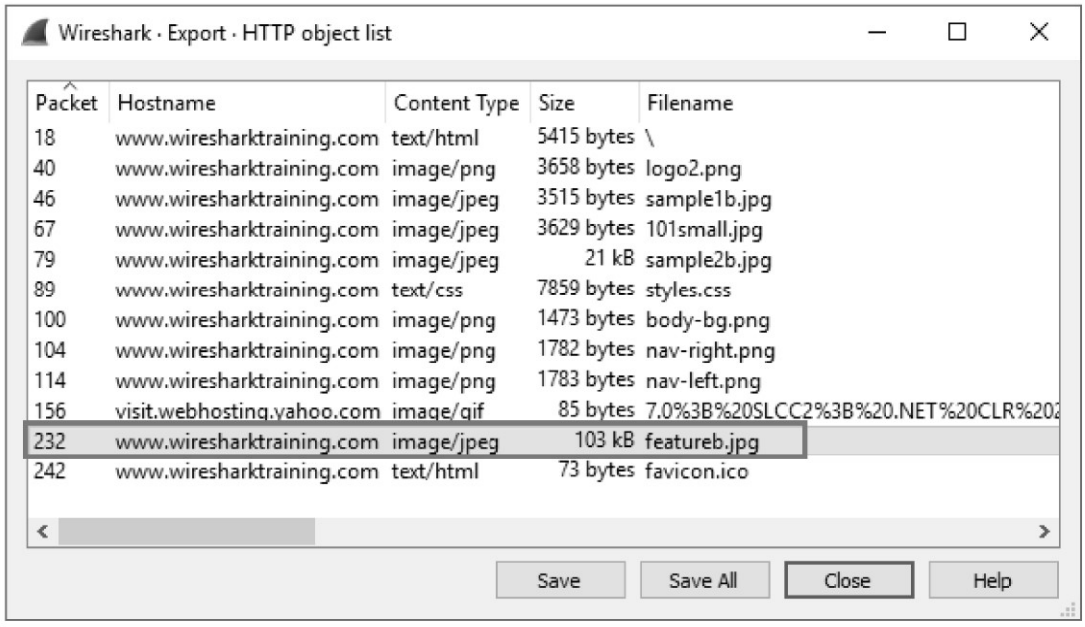
Lab 1 - A16. The words "Get Deep" are seen in the *featureb.jpg* image.

In this case, we need to perform two key steps. First, we need to enable the TCP reassembly preference so we can reassemble the *featureb.jpg* file. Next, we need to export the reassembled image (also called an HTTP "object") and look for the words.

Right-click on any TCP header in the Packet Details pane and enable *Allow subdissector to reassemble TCP streams*.

When should you enable or disable this TCP preference setting? Essentially, I keep this setting disabled by default so the `http.time` *value is the time from the request to the response, rather than the time from the request to the completion of the object download. I only enable it when I need to reassemble objects (as in this case) or work with TLS/SSL analysis.*

Now you can select *File | Export Objects | HTTP*. In the image below, we see the desired file near the bottom of the object list. Select the file, click *Save*, and select a directory for the file.

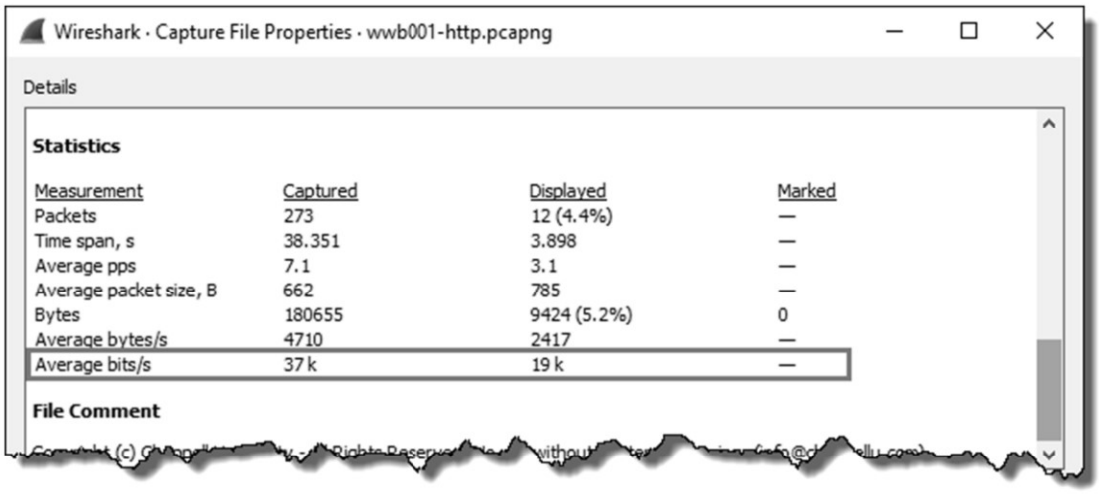


When you open the exported file, you will see the words “Get Deep” in the image.

Remember to turn Allow subdissector to reassemble TCP streams *off after finishing this reassembly.*

Lab 1 - A17. The average bits-per-second rate in this trace file is 37k.

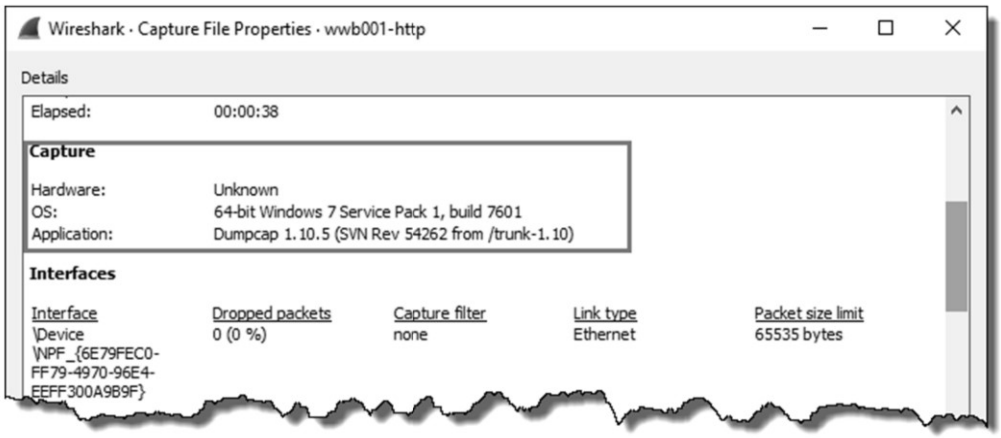
Select *Statistics | Capture File Properties* to view statistics and comments related to this trace file.



Alternately, click the *Capture File Properties* button on the Status Bar to open this window.

Lab 1 - A18. Dumpcap v1.10.5 was used to capture this trace file.

This information is also in the *Capture File Properties* window.

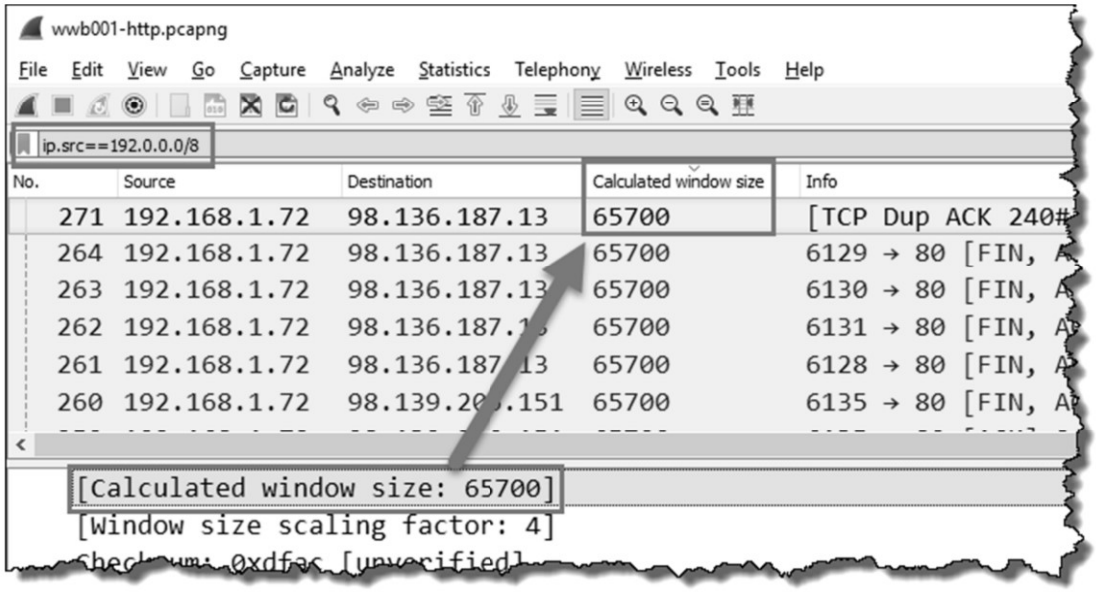


Lab 1 - A19. The largest Calculated Window Size advertised by the client is 65,700 bytes.

Earlier we learned that the client and HTTP servers in this trace file support Window Scaling.

Now, to determine the largest Calculated Window Scaling size of the client, we added a filter for traffic from the client first (`ip.src==192.0.0.0/8`).

Next, we added the *Calculated window size* field as a column and sorted the column.



The `ip.src==192.0.0.0/8` *filter is a pretty lazy filter.*

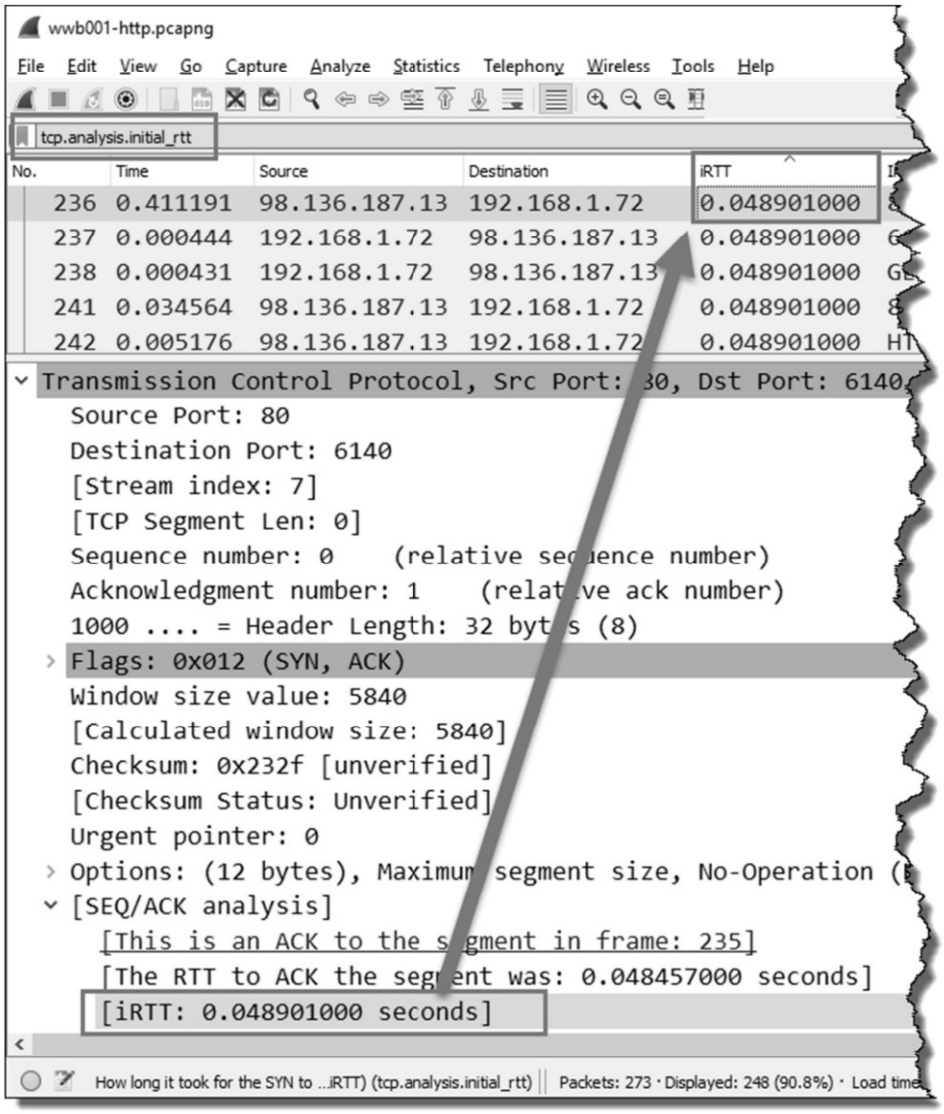
If this trace file had other hosts with IP addresses starting with 192, this wouldn't have worked. We would have had to be more explicit in the IP address of the source to look only at traffic from our client of interest.

Lab 1 - A20. The fastest initial round-trip time seen in this trace file is 0.048901 seconds.

Wireshark automatically calculates the Initial Round-trip Time (iRTT) for each TCP connection based on the time from the first to the third packet in the TCP handshake. This information is placed in all packets of a TCP conversation (except the SYN packet) under the *[SEQ/ACK analysis]* section, as shown below.

To find the fastest initial round-trip time, expand the *[SEQ/ACK analysis]* section of any TCP packet (except a SYN packet). Right-click on the *iRTT* field and select *Apply as Column*.

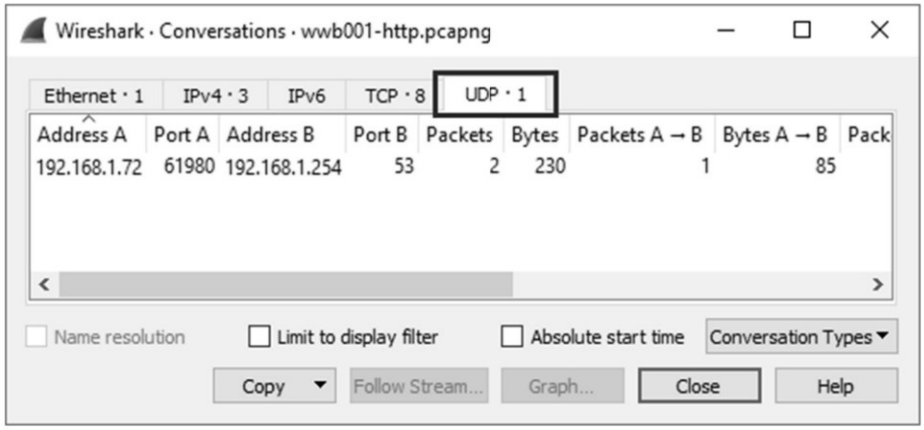
Now sort this column from low to high. Since many packets do not have this value in them (SYN packets and DNS packets), you will have some blank lines at the top of your sort list. Apply a filter for `tcp.analysis.initial_rtt` to only view packets that have this value.



The connection that has the fastest iRTT is between 98.136.187.13 and 192.168.1.72 on ports 80 and 6140.

Lab 1 - A21. There is one UDP stream in this trace file.

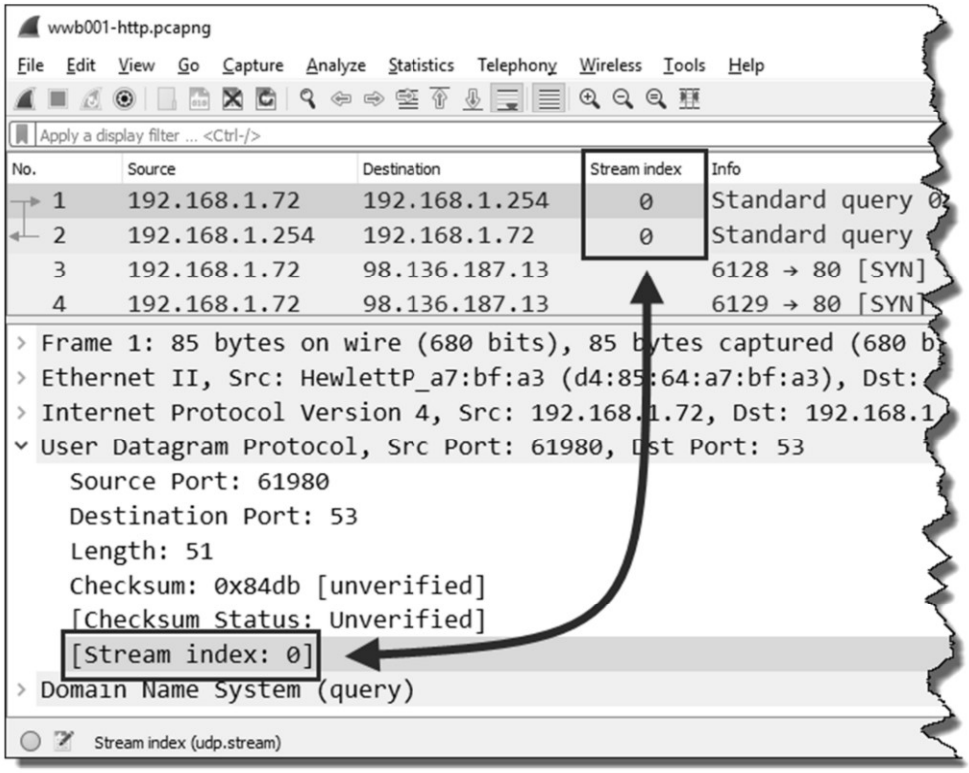
You can use the *Statistics | Conversations* window to identify the number of UDP and TCP streams in a trace file.



If you look inside TCP and UDP headers, you will see that every TCP and UDP conversation in the trace file is given a number by Wireshark (starting with 0). This is called the stream index number.

When Wireshark adds information and interpreted fields that are not actually inside a packet, it places square brackets around the field in the Packet Details pane. The stream index numbers are provided by Wireshark – they do not exist in TCP or UDP headers.

In the following image, we right-clicked on the UDP *Stream index* field in the UDP header and selected *Apply as Column*.



UDP stream index numbers and TCP stream index numbers are not related.

You can use these stream index numbers to filter on a conversation. For example, to apply a display filter for this conversation, the syntax would be `udp.stream==0`.

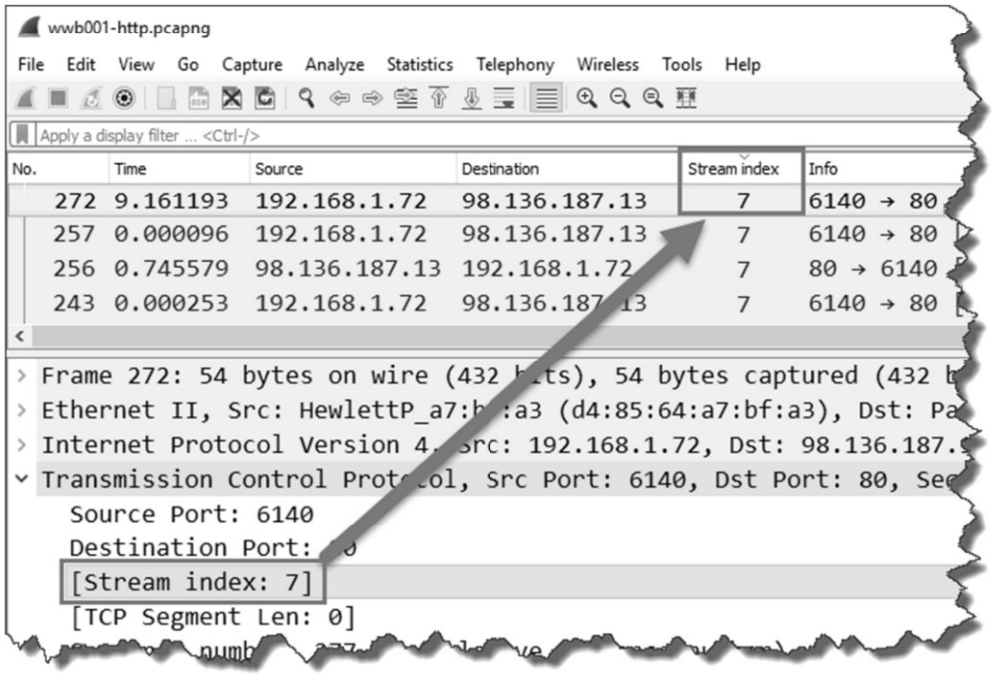
The stream indexing ability of Wireshark truly sped up the processing time for TCP and UDP conversation processing.

As you move ahead in this workbook, you will see how often I add a column for stream information.

Although you can get a conversation count by looking at the Statistics | Conversations window, you can also add the Stream index *column and sort from high to low to get the count.*

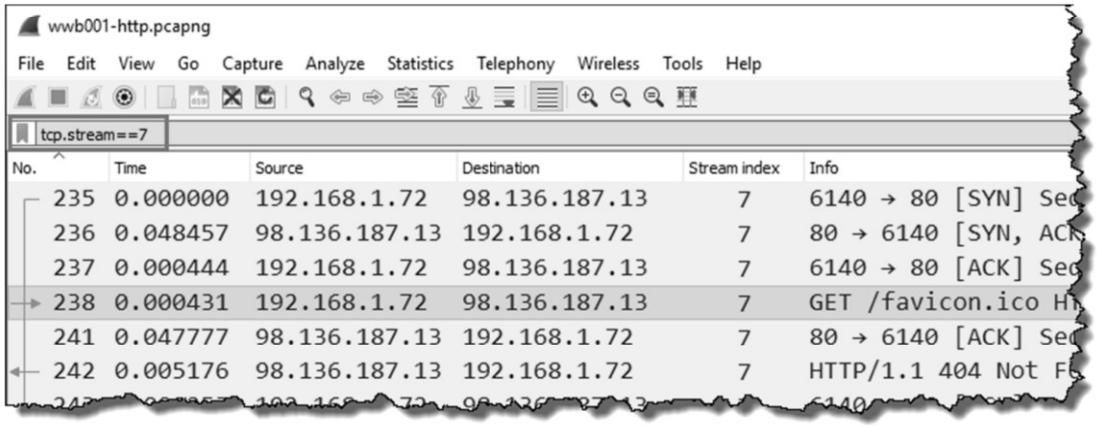
Lab 1 - A22. There are eight TCP streams in this trace file.

We added a column for the *TCP Stream index* field and then sorted the column from high to low. The highest TCP stream index value is 7, but remember that we start counting at 0. This indicates there are a total of 8 TCP streams in this trace file. You can also get this answer using *Statistics | Conversations | TCP*.



Lab 1 - A23. The purpose of TCP stream 7 is for the client to obtain the *favicon.ico* file from an HTTP server. This request is unsuccessful, however.

We used the display filter `tcp.stream==7` to look at this one TCP conversation. Since we'd sorted differently when answering the last question, we had to click the *No.* column heading to re-sort in the original order.

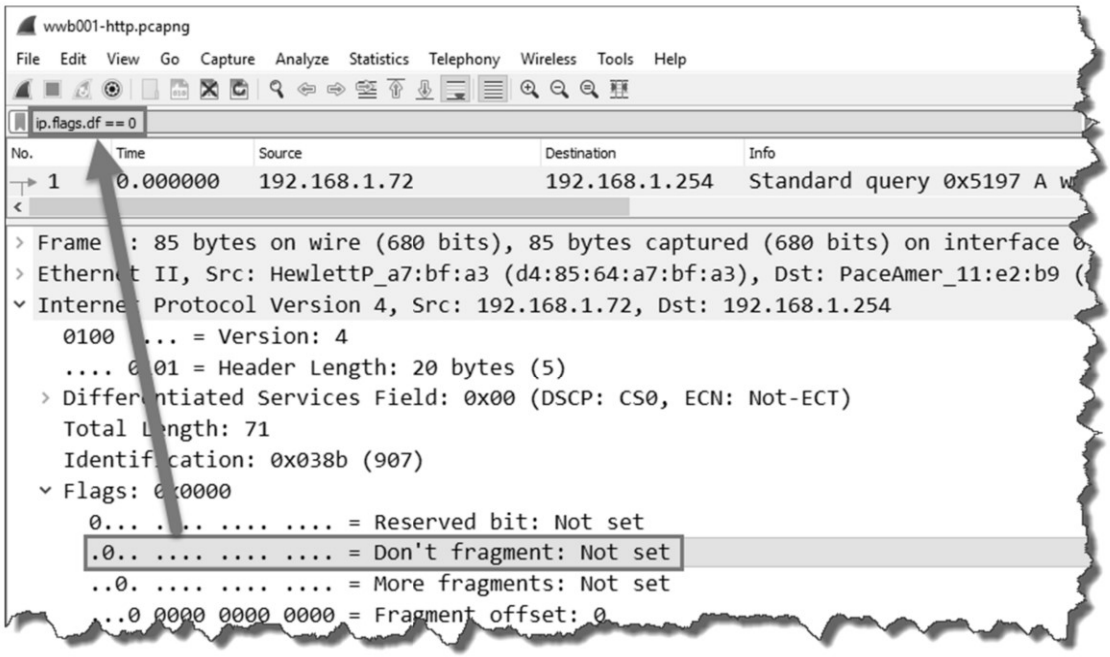


Lab 1 - A24. One packet allows IP fragmentation in the IP header in this trace file.

In this case, we are looking in the IP header at the *Don't fragment* field. When this field is set to 0, IP fragmentation is allowed. When this field is set to 1, IP fragmentation is not allowed.

You can select any packet in the trace file, right-click on the *Don't fragment* field and choose *Prepare a filter*. If you selected a packet that does not allow fragmentation, simply edit the filter before you apply it.

You should be filtering on `ip.flags.df==0`.



If you want to know if there are IP fragments in a trace file, consider the following filter:

```
ip.flags.mf==1 || ip.frag_offset > 0
```

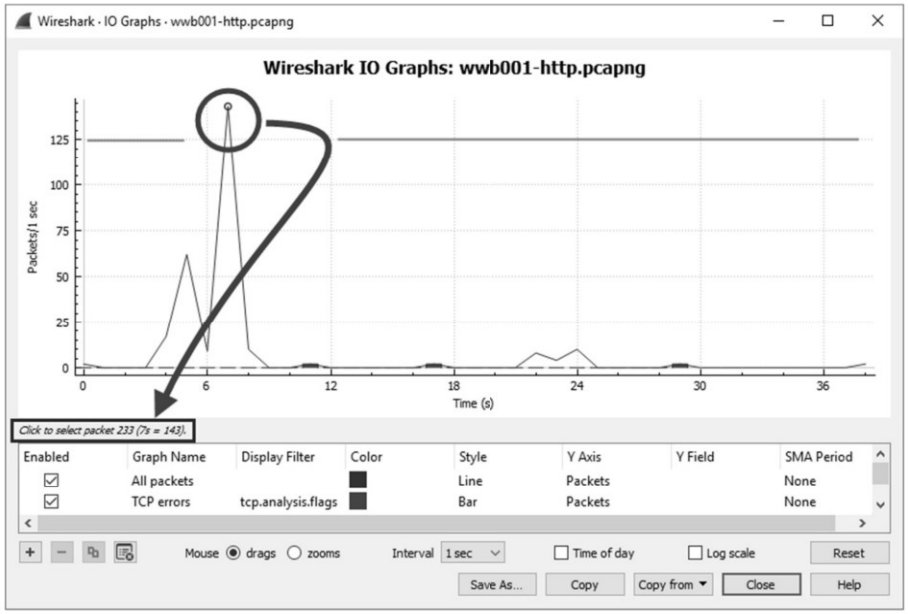
The first part, `ip.flags.mf==1`*, indicates that more fragments are to come in the fragment set. This will not catch the last fragment of a set, however.*

The second part, `ip.frag_offset > 0`*, means the fragment offset is greater than 0 – in other words, this is not the starting fragment. This will not catch the first fragment of a set, but it will catch the second through last fragment of the set.*

Lab 1 - A25. The packets-per-second rate reaches over 125 in this trace file just once.

Wireshark's IO Graph is the perfect tool to answer this question.

Select *Statistics | IO Graph*. By default, Wireshark provides the packets-per-second rate of all the traffic in the trace file and the TCP error count. In the next image shown, we can clearly see that the packets-per-second rate jumps above 125 just once in the trace file.



Hover over that high point in the graph and you will see that Wireshark indicates that the packets-per-second rate hit 143 at 7 seconds into the trace file.