

概要

11/06/27 10:46:54

文書間に違いがあります。

新文書：

[2nd_04_07](#)

86 ページ (4.39 MB)

11/06/27 10:45:18

旧文書：

[1st_04_07](#)

60 ページ (3.07 MB)


11/06/27 10:45:12

結果の表示に使用します。


[最初の変更箇所が 1 ページ目にあります。](#)


削除されたページはありません

このレポートの読み方

 は、変更箇所を示します。

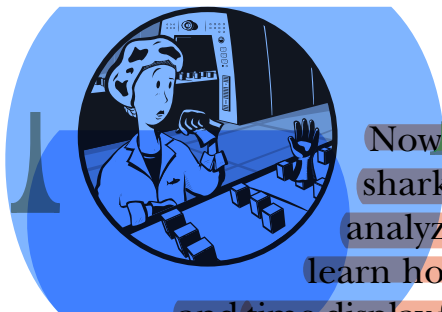
 は、削除された内容を示します。

 は、ページが変更されたことを示します。

 は、ページが移動されたことを示します。

4

WORKING WITH CAPTURED PACKETS



Now that you've been introduced to Wireshark, you're ready to start capturing and analyzing packets. In this chapter, you'll learn how to work with capture files, packets, and time-display formats. We'll also cover more advanced options for capturing packets and dive into the world of filters.

Working with Capture Files

As you perform packet analysis, you will find that a good portion of the analysis you do will happen after your capture. Usually, you will perform several captures at various times, save them, and analyze them all at once. Therefore, Wireshark allows you to save your capture files to be analyzed later. You can also merge multiple capture files.

Saving and Exporting Capture Files

To save a packet capture, select **File ▶ Save As**. You should see the Save File As dialog, as shown in Figure 4-1. You're asked for a location to save your packet capture and for the file format you wish to use. If you do not specify a file format, Wireshark will use the default *.pcap* file format.

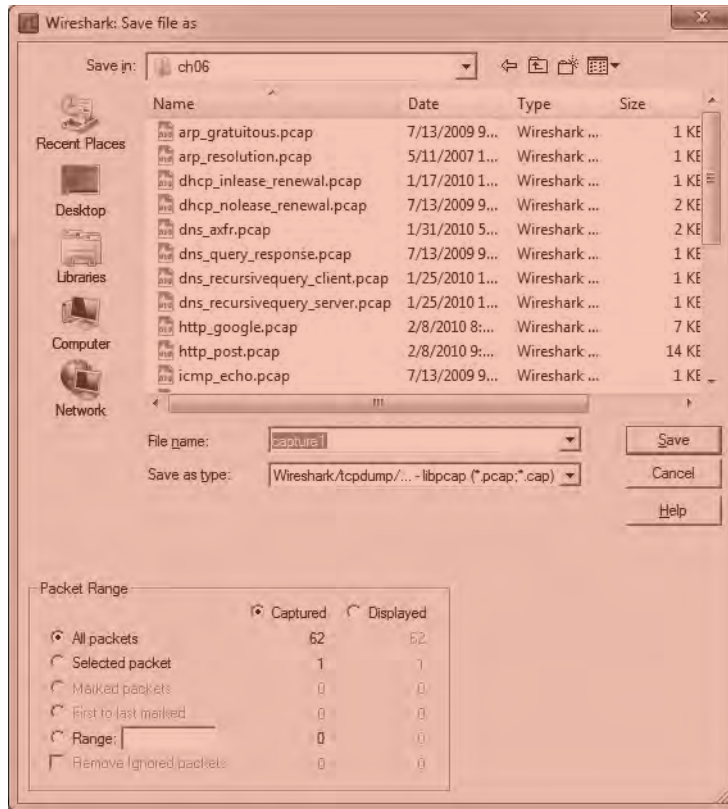


Figure 4-1: The Save File As dialog allows you to save your packet captures.

One of the more powerful features of the Save File As dialog is the ability to save a specific packet range. This is a great way to thin bloated packet capture files. You can choose to save only packets in a specific number range, marked packets, or packets visible as the result of a display filter (marked packets and filters are discussed later in this chapter).

You can export your Wireshark capture data into several different formats for viewing in other media or for importing into other packet-analysis tools. Formats include plaintext, PostScript, comma-separated values (CSV), and XML. To export your packet capture, choose **File ▶ Export**, and then select the format for the exported file. You will see a Save As dialog containing options related to that specific format.

Merging Capture Files

Certain types of analysis require the ability to merge multiple capture files. This is a common practice when comparing two data streams or combining streams of the same traffic that were captured separately.

To merge capture files, open one of the capture files you want to merge and choose **File ▶ Merge** to bring up the Merge with Capture File dialog, shown in Figure 4-2. Select the new file you wish to merge into the already open file, and then select the method to use for merging the files. You can prepend the selected file to the currently open one, append it, or merge the files chronologically based on their timestamps.

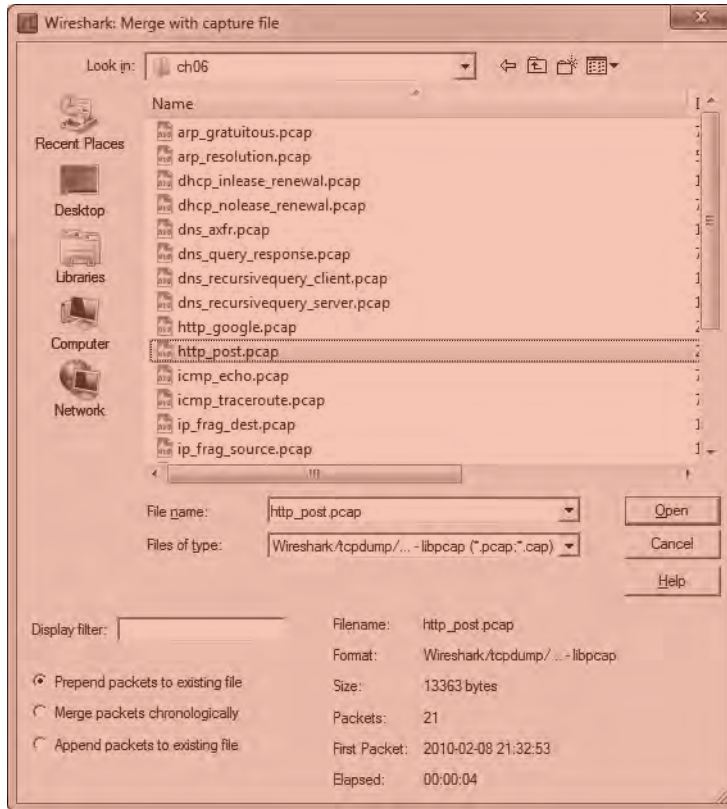


Figure 4-2: The Merge with Capture File dialog allows you to merge two capture files.

Working with Packets

You will eventually encounter situations involving a very large number of packets. As the number of these packets grows into the thousands and even millions, you will need to be able to navigate through packets more efficiently. For this purpose, Wireshark allows you to find and mark packets that match certain criteria. You can also print packets for easy reference.

Finding Packets

To find packets that match particular criteria, open the Find Packet dialog, shown in Figure 4-3, by pressing CTRL-F.

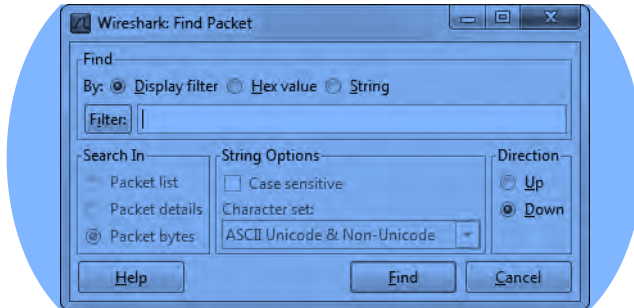


Figure 4-3: Finding packets in Wireshark based on specified criteria

This dialog offers three options for finding packets:

- The Display filter option allows you to enter an expression-based filter that will find only those packets that satisfy that expression.
- The Hex value option searches for packets with a hexadecimal (with bytes separated by colons) value you specify.
- The String option searches for packets with a text string you specify.

Table 4-1 shows examples of these search types.

Table 4-1: Search Types for Finding Packets

Search Type	Examples
Display filter	<code>not ip</code> <code>ip addr==192.168.0.1</code> <code>arp</code>
Hex value	<code>00:ff</code> <code>ff:ff</code> <code>00:AB:B1:f0</code>
String	<code>Workstation1</code> <code>UserB</code> <code>domain</code>

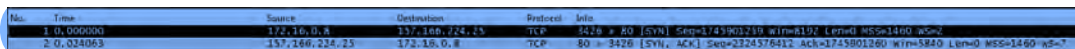
Other options include the ability to select the window in which you want to search, the character set to use, and the search direction. You can extend the capability of your string searches by specifying the pane the search is performed in, setting the character set used, and making the search case sensitive.

Once you've made your selections, enter your search criteria in the text box, and click **Find** to find the first packet that meets your criteria. To find the next matching packet, press CTRL-N; find the previous matching packet by pressing CTRL-B.

Marking Packets

After you have found the packets that match your criteria, you can mark those of particular interest. For example, you may want to mark packets to be able to save those packets separately or to find them quickly based on the coloration. Marked packets stand out with a black background and white text, as shown in Figure 4-4. (You can also sort out only marked packets when saving packet captures.)

To mark a packet, right-click it in the Packet List pane and choose **Mark Packet** from the pop-up or click a packet in the Packet List pane and press CTRL-M. To unmark a packet, toggle this setting off using CTRL-M again. You can mark as many packets as you wish in a capture. To jump forward and backward between marked packets, press SHIFT-CTRL-N and SHIFT-CTRL-B, respectively.



No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.0.8	157.140.224.25	TCP	5428 → 80 [SYN] Seq=1745903219 win=0 Len=0 MSS=1460 wScale=0
2	0.034063	157.140.224.25	172.16.0.8	TCP	80 → 5428 [SYN, ACK] Seq=2324576412 Ack=1745901260 win=5840 Len=0 MSS=1460 wScale=7

Figure 4-4: A marked packet is highlighted on your screen. In this example, packet 1 is marked and appears darker.

Printing Packets

Although most analysis will take place on the computer screen, you may need to print captured data. I often print out packets and tape them to my desk so that I can quickly reference their contents while doing other analysis. Being able to print packets to a PDF file is also very convenient, especially when preparing reports.

To print captured packets, open the Print dialog by choosing **File ▶ Print** from the main menu. You will see the Print dialog, as shown in Figure 4-5.

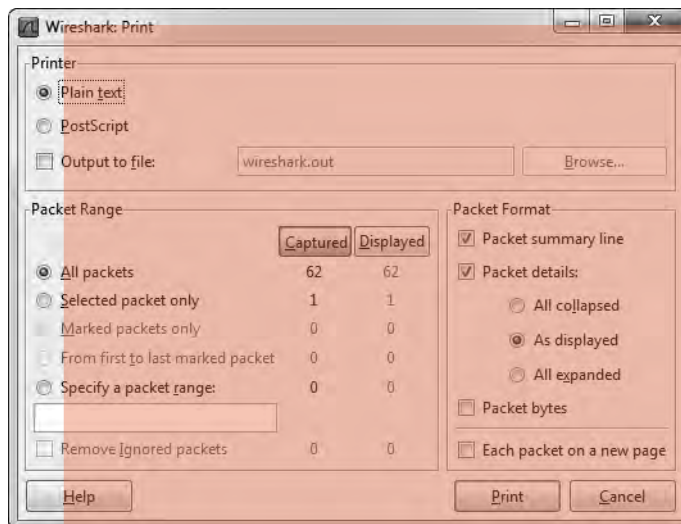


Figure 4-5: The Print dialog allows you to print the packets you specify.

You can print the selected data as plaintext or PostScript, or to an output file. As with the Save File As dialog, you can print a specific packet range, marked packets only, or packets displayed as the result of a filter. You can also select which of Wireshark's three main panes to print for each packet. Once you have selected the options, click **Print**.

Setting Time Display Formats and References

Time is of the essence—especially in packet analysis. Everything that happens on a network is time sensitive, and you will need to examine trends and network latency in nearly every capture file. Wireshark recognizes the importance of time and supplies several configurable options relating to it. In this section, we'll look at time display formats and references.

Time Display Formats

Each packet that Wireshark captures is given a timestamp, which is applied to the packet by the operating system. Wireshark can show the absolute timestamp indicating the exact moment when the packet was captured, as well as the time in relation to the last captured packet and the beginning and end of the capture.

The options related to the time display are found under the View heading on the main menu. The Time Display Format section, shown in Figure 4-6, lets you configure the presentation format as well as the precision of the time display. The presentation format option lets you choose various options for time display. The precision options allow you to set the time display precision to automatic or to a manual setting, such as seconds, milliseconds, microseconds, and so on. We will be changing these options later in the book, so you should familiarize yourself with them now.

Packet Time Referencing

Packet time referencing allows you to configure a certain packet so that all subsequent time calculations are done in relation to that specific packet. This feature is particularly handy when you are examining a series of sequential events that are triggered somewhere other than the start of the capture file.

To set a time reference to a certain packet, select the reference packet in the Packet List pane, and then choose **Edit ▶ Set Time Reference** from the main menu. To remove a time reference from a certain packet, select the packet and toggle off the **Edit ▶ Set Time Reference** setting.

When you enable a time reference on a particular packet, the Time column in the Packet List pane will display *REF*, as shown in Figure 4-7.

Setting a packet time reference is useful only when the time display format of a capture is set to display the time in relation to the beginning of the capture. Any other setting will produce no usable results and will create a set of times that can be very confusing.

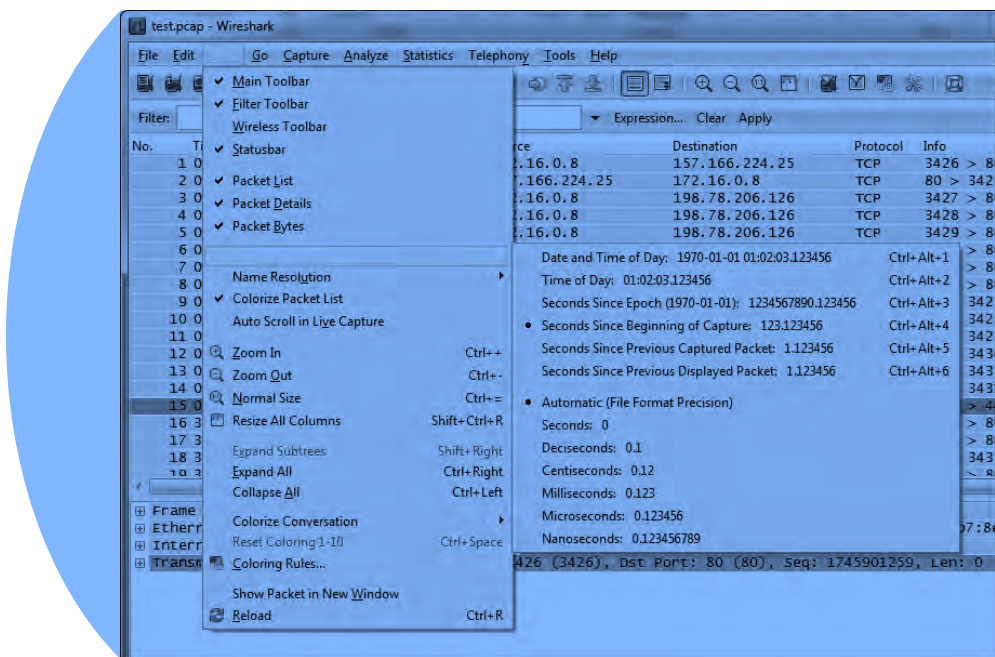


Figure 4-6: Several time display formats are available.

No.	Time	Source	Destination
4	0.118129	172.16.0.8	198.78.206.126
5	*REF*	172.16.0.8	198.78.206.126
6	0.000077	172.16.0.8	198.78.206.126
7	0.000153	172.16.0.8	198.78.206.126

Figure 4-7: A packet with the packet time reference toggle enabled

Setting Capture Options

We walked through a very basic packet capture in Chapter 3. Wireshark offers quite a few more capture options in the Capture Options dialog, shown in Figure 4-8. To open this dialog, choose **Capture ► Interfaces** and click the **Options** button next to the interface on which you want to capture packets.

The Capture Options dialog has more bells and whistles than you can shake a stick at, all designed to give you more flexibility while capturing packets. It's divided into Capture, Capture Files, Stop Capture, Display Options, and Name Resolution sections, which we'll examine separately.

Capture Settings

The Interface drop-down list in the Capture section is where you can select the network interface to configure. The left drop-down list allows you to specify whether the interface is local or remote, and the right drop-down list shows all available capture interfaces. The IP address of the interface you have selected is displayed directly below this drop-down list.

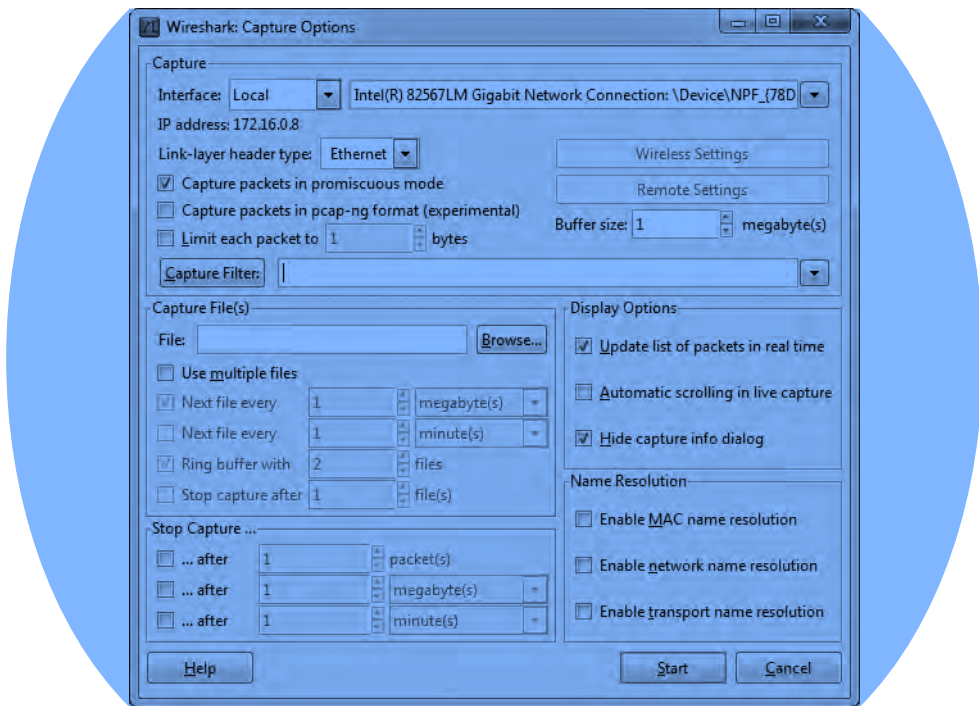


Figure 4-8: The Capture Options dialog

The three checkboxes on the left side of the dialog box allow you to enable or disable promiscuous mode (always enabled by default), capture packets in the currently experimental pcap-ng format, and limit the size of each capture packet by bytes.

The buttons on the right side of the Capture section let you access wireless and remote settings (as applicable). Beneath those is the buffer size option, which is available only on systems running Microsoft Windows. You can specify the amount of capture packet data that is stored in the kernel buffer before it is written to disk. (This is a value you won't normally modify unless you begin noticing that you are dropping a lot of packets.) The Capture Filter option lets you specify a capture filter.

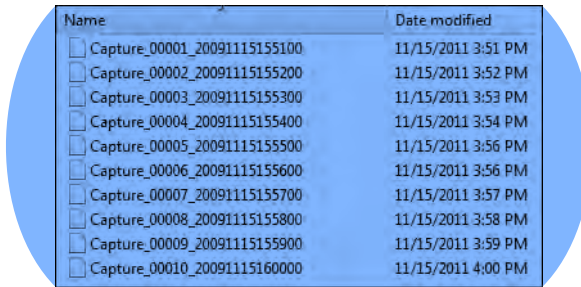
Capture File(s) Settings

The Capture File(s) section allows you to automatically store capture packets in a file, rather than capturing them first and then saving the file. Doing so offers you a great deal more flexibility in managing how packets are saved. You can choose to save them as a single file or a file set, or even use a ring buffer to manage the number of files created. To enable this option, enter a complete file path and name in the File text box.

When capturing a large amount of traffic or performing long-term captures, file sets can prove particularly useful. A file set is a grouping of multiple

files separated by a particular condition. To save to a file set, check the Use Multiple Files option here.

Wireshark uses various triggers to manage saving to file sets based upon a file size or time condition. To enable these options, place a check mark next to the Next File Every option (the top one for file-size triggers and the one beneath that for time-based triggers), and then specify the value and unit on which to trigger. For instance, you can create a trigger that creates a new file after every 1MB of traffic captured, or after every minute of traffic captured, as shown in Figure 4-9.



Name	Date modified
<input type="checkbox"/> Capture_00001_20091115155100	11/15/2011 3:51 PM
<input type="checkbox"/> Capture_00002_20091115155200	11/15/2011 3:52 PM
<input type="checkbox"/> Capture_00003_20091115155300	11/15/2011 3:53 PM
<input type="checkbox"/> Capture_00004_20091115155400	11/15/2011 3:54 PM
<input type="checkbox"/> Capture_00005_20091115155500	11/15/2011 3:56 PM
<input type="checkbox"/> Capture_00006_20091115155600	11/15/2011 3:56 PM
<input type="checkbox"/> Capture_00007_20091115155700	11/15/2011 3:57 PM
<input type="checkbox"/> Capture_00008_20091115155800	11/15/2011 3:58 PM
<input type="checkbox"/> Capture_00009_20091115155900	11/15/2011 3:59 PM
<input type="checkbox"/> Capture_00010_20091115160000	11/15/2011 4:00 PM

Figure 4-9: A file set created by Wireshark at one-minute intervals

These options can also be used in combination. For example, if you specify both triggers, a new file will be created when 1MB of data is captured *or* when a minute has elapsed—whichever comes first.

The Ring Buffer With option lets you use a ring buffer when creating a file set. This is used by Wireshark as a first in, first out (FIFO) method of writing multiple files. Although the term *ring buffer* has multiple meanings throughout information technology, for our purposes here, it is essentially a file set that specifies that upon completion of writing the last file, the first file is overwritten when more data must be saved to disk. You can check this option and specify the maximum number of files you wish to cycle through. For example, say you choose to use multiple files for your capture with a new file created every hour, and you set your ring buffer to 6. Once the sixth file has been created, the ring buffer will cycle back around and overwrite the first file rather than create a seventh file. This ensures that no more than six files (or in this case, hours) of data will remain on your hard drive, while still allowing new data to be written.

The Stop Capture After option allows you to stop the current capture once a certain number of files have been created.

Stop Capture Settings

The Stop Capture section lets you stop the running capture after certain triggers are met. As with multiple file sets, you can trigger based on file size and time interval, as well as number of packets. These options can be used with the multiple file options previously discussed.

Display Options

The Display Options section controls how packets are shown as they are being captured. The Update List of Packets in Real Time option is self-explanatory and can be paired with the Automatic Scrolling in Live Capture option. When both of these options are enabled, all captured packets are displayed on the screen, with the most recently captured ones shown instantly.

WARNING When paired, the Update List of Packets in Real Time and Automatic Scrolling in Live Capture options can be quite processor intensive, even when capturing a reasonable amount of data. Unless you have a specific need to see the packets in real time, it's best to deselect both options.

The Hide Capture Info Dialog option lets you suppress the display of a small window that shows the number and percentage of packets that have been captured, by protocol.

Name Resolution Settings

The Name Resolution section options allow you to enable automatic MAC (layer 2), network (layer 3), and transport (layer 4) name resolution for your capture. We'll discuss name resolution in Wireshark more in depth, including its drawbacks, in Chapter 5.

Using Filters

Filters allow you to specify exactly which packets you have available for analysis. Simply stated, a filter is an expression that defines criteria for the inclusion or exclusion of packets. If there are packets you don't want to see, you can write a filter that gets rid of them. If there are packets you want to see exclusively, you can write a filter that shows only those packets.

Wireshark offers two main types of filters:

- Capture filters are specified when packets are being captured and will capture only those packets that are specified for inclusion/exclusion in the given expression.
- Display filters are applied to an existing set of captured packets in order to hide unwanted packets or show desired packets based on the specified expression.

Let's look at capture filters first.

Capture Filters

Capture filters are used during the actual packet-capturing process. One primary reason for using a capture filter is performance. If you know that you do not need to analyze a particular form of traffic, you can simply filter it out with a capture filter and save the processing power that would typically be used in capturing those packets.

The ability to create custom capture filters comes in handy when dealing with large amounts of data. The analysis process can be sped up by ensuring that you are looking at only the packet relevant to the issue at hand.

A simple example of when you might use a capture filter is when capturing traffic on a server with multiple roles. Suppose you are troubleshooting an issue with a service running on port 262. If the server you are analyzing runs several different services on a variety of ports, finding and analyzing only the traffic on port 262 can be quite a job in itself. To capture only the port 262 traffic, you can use a capture filter. To do so, you can use the Capture Options dialog, discussed earlier in this chapter, as follows:

1. Choose **Capture ▶ Interfaces** and click the **Options** button next to the interface on which you want to capture packets to open the Capture Options dialog.
2. Select the interface you wish to capture packets on, and choose a capture filter.
3. You can apply the capture filter by entering an expression next to the Capture Filter button. We want our filter to show only traffic inbound and outbound to port 262, so we enter **port 262**, as shown in Figure 4-10. (We'll discuss expressions in more detail in the next section.)
4. Once you have set your filter, click **Start** to begin the capture.

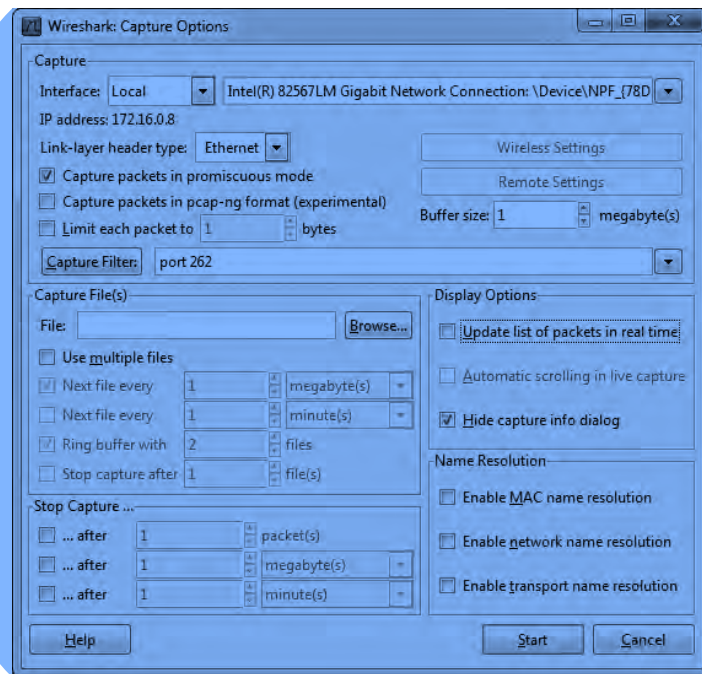


Figure 4-10: Creating a capture filter in the Capture Options dialog

After collecting an adequate sample, you should now see only the port 262 traffic and be able to more efficiently analyze this particular data.

Capture/BPF Syntax

Capture filters are applied by WinPcap and use the Berkeley Packet Filter (BPF) syntax. This syntax is common in several packet-sniffing applications, mostly because most packet-sniffing applications rely on the libpcap/WinPcap libraries, which allow for the use of BPFs. A knowledge of BPF syntax is crucial as you dig deeper into networks at the packet level.

A filter created using the BPF syntax is called an *expression*, and each expression consists of one or more *primitives*. Primitives consist of one or more *qualifiers* (as listed in Table 4-2) followed by an ID name or number, as shown in Figure 4-11.

Table 4-2: The BPF Qualifiers

Qualifier	Description	Examples
Type	Identifies what the ID name or number refers to	host, net, port
Dir	Specifies a transfer direction to or from the ID name or number	src, dst
Proto	Restricts the match to a particular protocol	ether, ip, tcp, udp, http, ftp

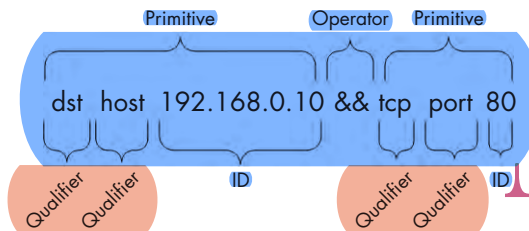


Figure 4-11: A sample capture filter

Given the components of an expression, a qualifier of `src` and an ID of `192.168.0.10` would combine to form a primitive. This primitive alone is an expression that would capture traffic only with a source IP address of `192.168.0.10`.

You can use logical operators to combine primitives to create more advanced expressions. Three logical operators are available:

- Concatenation operator AND (`&&`)
- Alternation operator OR (`||`)
- Negation operator NOT (`!`)

For example, the following expression will capture only traffic with a source IP address of `192.168.0.10` and a source or destination port of `80`:

```
src 192.168.0.10 && port 80
```

Hostname and Addressing Filters

Most filters you create will center on a particular network device or grouping of devices. Depending on the circumstances, filtering can be based on a device's MAC address, IPv4 address, IPv6 address, or its DNS hostname.

For example, say you're curious about the traffic of a particular host that is interacting with a server on your network. From the server, you can create a filter using the host qualifier that captures all traffic associated with that host's IPv4 address:

```
host 172.16.16.149
```

If you are on an IPv6 network, you would filter based on an IPv6 address using the host qualifier as shown here:

```
host 2001:db8:85a3::8a2e:370:7334
```

You can also filter based on a device's hostname with the host qualifier, like so:

```
host testserver2
```

Or, if you're concerned that the IP address for a host might change, you can filter based on its MAC address as well by adding the ether protocol qualifier:

```
ether host 00-1a-a0-52-e2-a0
```

The transfer direction qualifiers are often used in conjunction with filters like the ones in the previous examples to capture traffic based on whether it's going to or coming from a host. For example, to capture only traffic coming from a particular host, add the src qualifier

```
src host 172.16.16.149
```

To capture only data leaving server 172.16.16.149 that is destined for a questionable host, use the dst qualifier:

```
dst host 172.16.16.149
```

When you don't use a type qualifier (host, net, or port) with a primitive, the host qualifier is assumed. Therefore, the equivalent of the preceding example could exclude that qualifier:

```
dst 172.16.16.149
```

Port and Protocol Filters

In addition to filtering on hosts, you can filter based on the ports used in each packet. Port filtering can be used to filter based on services and applications that use known service ports. For example, here's a simple filter to capture traffic only on port 8080:

```
port 8080
```

To capture all traffic except that on port 8080, this will work:

```
!port 8080
```

The port filters can be combined with transfer direction qualifiers. For example, to capture only traffic going to the web server listening on the standard HTTP port 80, use the `dst` qualifier:

```
dst port 80
```

Protocol Filters

Protocol filters let you filter packets based on certain protocols. They are used to match non-application-layer protocols that can't simply be defined by the use of a certain port. Thus, if you want to see only ICMP traffic, you could use this filter:

```
icmp
```

To see everything but IPv6 traffic, this will do the trick:

```
!ip6
```

Protocol Field Filters

One of the real powers of the BPF syntax is the ability that it gives us to examine every byte of a protocol header in order to create very specific filters based on that data. The advanced filters that we'll discuss in this section will allow you to retrieve a specific number of bytes from a packet beginning at a particular location.

For example, suppose that we want to filter based on the type field of an ICMP header. The type field is located at the very beginning of a packet, which puts it at offset 0. To identify the location to examine within a packet, specify the byte offset in square brackets next to the protocol qualifier—`icmp[0]` in this example. This specification will return a 1-byte integer value that we can compare against. For instance, to get only ICMP packets that

represent destination unreachable (type 3) messages, we use the equal to operator in our filter expression, as follows:

```
icmp[0] == 3
```

To examine only ICMP packets that represent an echo request (type 8) or echo reply (type 0), use two primitives with the OR operator:

```
icmp[0] == 8 || icmp[0] == 0
```

These filters work great, but they filter based on only 1 byte of information within a packet header. Luckily, you can also specify the length of the data to be returned in your filter expression by appending the byte length after the offset number within the square brackets, separated by a colon.

For example, say we want to create a filter that captures all ICMP destination-unreachable, host-unreachable packets, identified by type 3, code 1. These are 1-byte fields, located next to each other at offset 0 of the packet header. To do this, we create a filter that checks 2 bytes of data beginning at offset 0 of the packet header, and compare that against the hex value 0301 (type 3, code 1), like this:

```
icmp[0:2] == 0x0301
```

A common scenario is to capture only TCP packets with the RST flag set. We will cover TCP extensively in Chapter 6. For now, you just need to know that the flags of a TCP packet are located at offset 13. This is an interesting field because it is collectively 1 byte in size as the flags field, but each particular flag is identified by a single bit within this byte. Multiple flags can be set simultaneously in a TCP packet, so we can't efficiently filter by a single tcp[13] value because several may represent the RST bit being set. Therefore, we must specify the location within the byte that we wish to examine by appending that location to the current primitive with a single ampersand (&). The RST flag is at the bit representing the number 4 within this byte, and the fact that this bit is set to 4 tells us that the flag is set. The filter looks like this:

```
tcp[13] & 4 == 4
```

To see all packets with the PSH flag set, which is identified by the bit location representing the number 8, our filter would use that location instead:

```
tcp[13] & 8 == 8
```

Sample Capture Filter Expressions

You will often find that the success or failure of your analysis depends on your ability to create filters appropriate for your current situation. Table 4-3 shows a few of the capture filters that I use most frequently.

Table 4-3: Commonly Used Capture Filters

Filter	Description
<code>tcp[13] & 32 == 32</code>	TCP packets with the URG flag set
<code>tcp[13] & 16 == 16</code>	TCP packets with the ACK flag set
<code>tcp[13] & 8 == 8</code>	TCP packets with the PSH flag set
<code>tcp[13] & 4 == 4</code>	TCP packets with the RST flag set
<code>tcp[13] & 2 == 2</code>	TCP packets with the SYN flag set
<code>tcp[13] & 1 == 1</code>	TCP packets with the FIN flag set
<code>tcp[13] == 18</code>	TCP SYN-ACK packets
<code>ether host 00:00:00:00:00:00</code> (replace with your MAC)	Traffic to or from your MAC address
<code>!ether host 00:00:00:00:00:00</code> (replace with your MAC)	Traffic not to or from your MAC address
<code>broadcast</code>	Broadcast traffic only
<code>icmp</code>	ICMP traffic
<code>icmp[0:2] == 0x0301</code>	ICMP destination unreachable, host unreachable
<code>ip</code>	IPv4 traffic only
<code>ip6</code>	IPv6 traffic only
<code>udp</code>	UDP traffic only

Display Filters

A *display filter* is one that, when applied to a capture file, tells Wireshark to display only packets that match that filter. You can enter a display filter in the Filter text box above the Packet List pane.

Display filters are used more often than capture filters because they allow you to filter packet data without actually omitting the rest of the data in the capture file. That way, if you need to revert back to the original capture, you can simply clear the filter expression.

You might use a display filter to clear irrelevant broadcast traffic from a capture file; for instance, to clear ARP broadcasts from the Packet List pane when these packets don't relate to the current problem being analyzed. However, because those ARP broadcast packets may be useful later, it's better to filter them temporarily than it is to delete them.

To filter out all ARP packets in the capture window, simply place your cursor in the Filter text box at the top of the Packet List pane and enter `!arp` to remove all ARP packets from the Packet List pane, as shown in Figure 4-12. To remove the filter, click the **Clear** button.



Figure 4-12: Creating a display filter using the Filter text box above the Packet List pane

The Filter Expression Dialog (the Easy Way)

The Filter Expression dialog, shown in Figure 4-13, makes it easy for novice Wireshark users to create capture and display filters. To access this dialog, click the **Capture Filter** button in the Capture Options dialog, and then click the **Expression** button.

The left side of the dialog lists all possible protocol fields. These fields specify all possible filter criteria. To create a filter, follow these steps:

1. To view the specific criteria fields associated with a protocol, expand that protocol by clicking the plus (+) symbol next to it. Once you find the criterion you want to base your filter on, click to select it.
2. Choose the way that your selected field will relate to the criterion value you supply. This relation is specified as equal to, greater than, less than, and so on.
3. Create your filter expression by specifying a criterion value that will relate to your selected field. You can define this value or select it from predefined ones programmed into Wireshark.
4. When you've finished, click **OK** to view the completed text-only version of your filter.

The Filter Expression dialog is great for novice users, but once you get the hang of things, you will find that manually entering filter expressions greatly increases their efficiency. The display filter expression syntax structure is simple, yet extremely powerful.

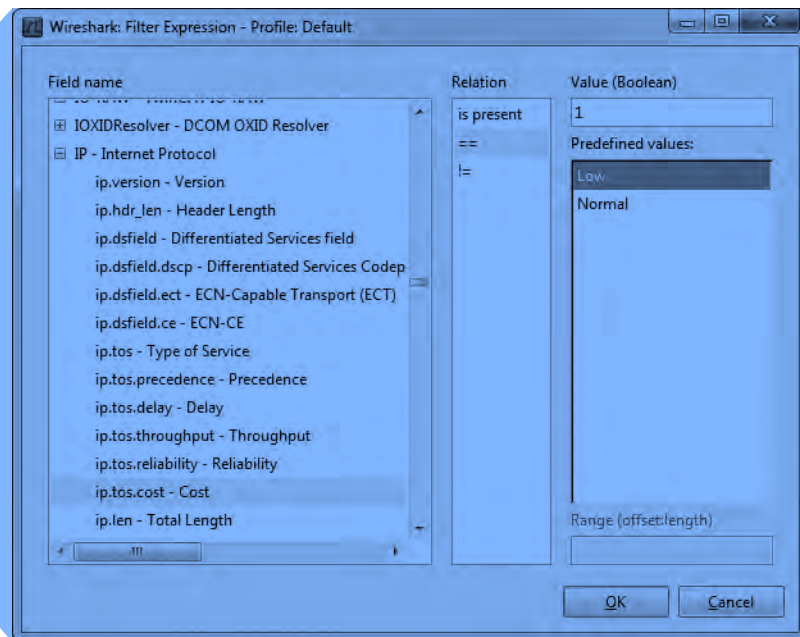


Figure 4-13: The Filter Expression dialog allows for easy creation of filters in Wireshark.

The Filter Expression Syntax Structure (the Hard Way)

You will most often use a capture or display filter to filter based on a specific protocol. For example, say you are troubleshooting a TCP problem and you want to see only TCP traffic in a capture file. If so, a simple tcp filter will do the job.

Now let's look at things from the other side of the fence. Imagine that in the course of troubleshooting your TCP problem, you have used the ping utility quite a bit, thereby generating a lot of ICMP traffic. You could remove this ICMP traffic from your capture file with the filter expression !icmp.

Comparison operators allow you to compare values. For example, when troubleshooting TCP/IP networks, you will often need to view all packets that reference a particular IP address. The equal-to comparison operator (==) will allow you to create a filter showing all packets with an IP address of 192.168.0.1:

```
ip.addr==192.168.0.1
```

Now suppose that you need to view only packets that are less than 128 bytes in length. You can use the "less than or equal to" operator (<=) to accomplish this goal in a filter expression like this:

```
frame.len <= 128
```

Table 4-4 shows Wireshark's comparison operators.

Table 4-4: Wireshark Filter Expression Comparison Operators

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators allow you to combine multiple filter expressions into one statement, dramatically increasing the effectiveness of our filters. For example, say that we're interested in displaying only packets to two IP addresses. We can use the or operator to create one expression that will display packets containing either IP address, like this:

```
ip.addr==192.168.0.1 or ip.addr==192.168.0.2
```

Table 4-5 lists Wireshark's logical operators.

Table 4-5: Wireshark Filter Expression Logical Operators

Operator	Description
and	Both conditions must be true.
or	Either one of the conditions must be true.
xor	One and only one condition must be true.
not	Neither one of the conditions is true.

Sample Display Filter Expressions

Although the concepts related to creating filter expressions are fairly simple, you will need to use several specific keywords and operators when creating new filters for various problems. Table 4-6 shows some of the display filters that I use most often. For a complete list, see the Wireshark display filter reference at <http://www.wireshark.org/docs/dfref/>.

Table 4-6: Commonly Used Display Filters

Filter	Description
<code>!tcp.port==3389</code>	Clear RDP traffic
<code>tcp.flags.syn==1</code>	TCP packets with the SYN flag set
<code>tcp.flags.rst==1</code>	TCP packets with the RST flag set
<code>!arp</code>	Clear ARP traffic
<code>http</code>	All HTTP traffic
<code>tcp.port==23 tcp.port 21</code>	Cleartext admin traffic (Telnet or FTP)
<code>smtp pop imap</code>	Cleartext email traffic (SMTP, POP, or IMAP)

Saving Filters

Once you begin creating a lot of capture and display filters, you will find that you use certain ones frequently. Fortunately, you don't need to type these in each time you want to use them, because Wireshark lets you save your filters for later use. To save a custom capture filter, follow these steps:

1. Select **Capture ► Capture Filters** to open the Capture Filter dialog.
2. Create a new filter by clicking the **New** button on the left side of the dialog.
3. Enter a name for your filter in the Filter Name box.
4. Enter the actual filter expression in the Filter String box.
5. Click the **Save** button to save your filter expression in the list.

To save a custom display filter, follow these steps:

1. Select **Analyze ► Display Filters**, or click the **Filter** button above the Packet List pane to open the Display Filter dialog, shown in Figure 4-14.

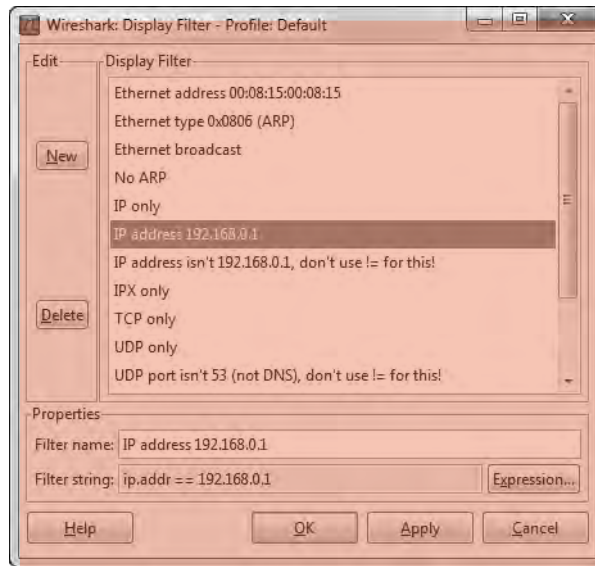


Figure 4-14: The Display Filter dialog allows you to save filter expressions.

2. Create a new filter by clicking the **New** button on the left side of the dialog.
3. Enter a name for your filter in the Filter Name box.
4. Enter the actual filter expression in the Filter String box.
5. Click the **Save** button to save your filter expression in the list.

Wireshark includes several built-in filters that are great examples of what a filter should look like. You will want to use them (together with the Wireshark help pages) when creating your own filters. We will use filters in examples throughout this book.

5

ADVANCED WIRESHARK FEATURES



Once you master the basics of Wireshark, the next step is to delve into its analysis and graphing capabilities. In this chapter, we'll look at some of these powerful features, including the Endpoints and Conversations windows, the finer points of name resolution, protocol dissection, stream following, IO graphing, and more.

Network Endpoints and Conversations

In order for network communication to take place, you must have data flowing between at least two devices. An *endpoint* is a device that sends or receives data on the network. For instance, there are two endpoints in TCP/IP communication: the IP addresses of the systems sending and receiving data, such as 192.168.1.25 and 192.168.1.30.

For example, on layer 2, the communication takes place between two physical NICs and their MAC addresses. If the NICs sending and receiving data have addresses of 00:ff:ac:ce:0b:de and 00:ff:ac:e0:dc:0f, those addresses are the endpoints of communication, as you can see in Figure 5-1.

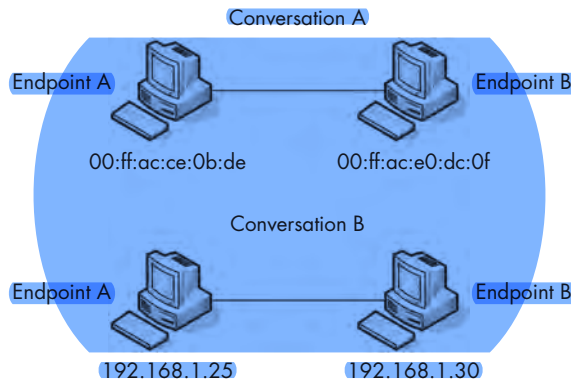


Figure 5-1: Endpoints on a network

A *conversation* on a network, like a conversation between two people, describes the communication that takes place between two hosts (endpoints). For example, Jim and Sally’s conversation might consist of, “Hey, how are you?” “I’m great! Yourself?” and “Couldn’t be better!” A conversation between 192.168.1.5 and 192.168.0.8 might look like “SYN,” “SYN/ACK,” and “ACK.” (We’ll look at the TCP/IP communication process in more detail in Chapter 6.)

Viewing Endpoints

When analyzing traffic, you may find that you can pinpoint a problem to a specific endpoint on a network. Wireshark’s Endpoints window (**Statistics ▸ Endpoints**) shows several helpful statistics for each endpoint (see Figure 5-2), including the addresses and the number of packets and bytes transmitted and received by each.

The tabs at the top of the window show all supported and recognized endpoints in the current capture file. To narrow the list of endpoints to specific protocols, click a tab. Check the Name resolution checkbox to use name resolution within the Endpoints window.

You can use the Endpoints window to filter out specific packets for display in the Packet List pane. Right-click a specific endpoint to see several options, including the ability to create a filter to display only traffic related to this endpoint or all traffic excluding the selected endpoint. You can also export the endpoint directly into a colorization rule (coloring rules are discussed in Chapter 3).

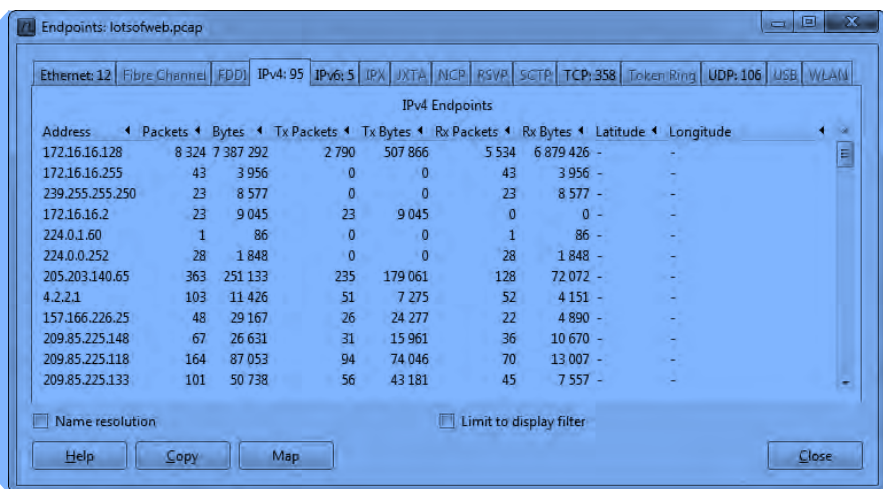


Figure 5-2: The Endpoints window lets you view each of the endpoints in a capture file.

Viewing Network Conversations

The Wireshark Conversations window (**Statistics ▶ Conversations**), shown in Figure 5-3, displays the addresses of the endpoints involved in the conversation listed as *Address A* and *Address B*, and the packets and bytes transmitted to and from each device.

The conversations listed in this window are divided by the protocol they use, which can be selected via the tabs at the top of the window. Right-clicking a specific conversation allows you to create filters that may be useful, such as displaying all traffic transmitted from device A, all traffic received by device B, or all traffic communicated between devices A and B.

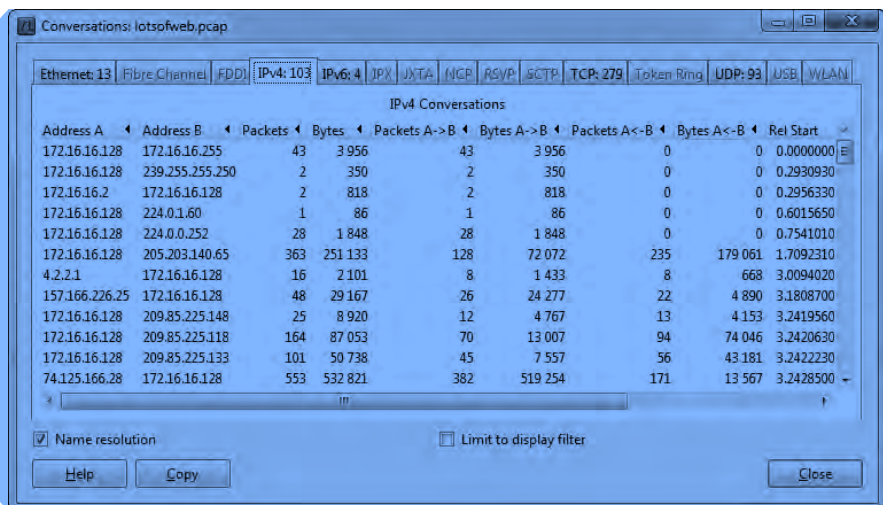


Figure 5-3: The Conversations window lets you interact with each conversation in a capture file.

Troubleshooting with the Endpoints and Conversations Windows

lotsofweb.pcap

The Endpoints and Conversations windows are crucial in network troubleshooting, especially when you're trying to locate the source of a significant amount of traffic on the network or determine which one of your servers is talking the most.

For example, when you open the file *lotsofweb.pcap*, you will see a lot of HTTP traffic representing multiple clients browsing the Internet. If you start by viewing the Endpoints window, you can immediately draw some conclusions about the traffic you are viewing.

Looking at the IPv4 tab (see Figure 5-4), you see that your first address when sorting by bytes is the local 172.16.16.128 address, meaning this device on your network is the top talker (host responsible for the most communication) among your data set. The second address of 74.125.103.163 is a non-local address, so at this point, you can assume that you have one client talking to this IP address a lot, or that multiple clients are talking to it a moderate amount. A quick WHOIS (<http://whois.arin.net/ui/>) tells you that this IP address belongs to Google, and perusing the packets will identify this as YouTube traffic.

NOTE IP address assignments are managed by different entities, depending on their geographic location. In our example here, we used the American Registry for Internet Numbers (ARIN), which is responsible for the IP address assignments of the United States (and some surrounding areas). Generally, you would perform a WHOIS for an IP at the website of the organization responsible for that IP. If you don't know the geographic region and perform the search at the wrong registry site, you will be pointed toward the right location. Some other such address registries include AfriNIC (Africa), RIPE (Europe), and APNIC (Asia/Pacific).

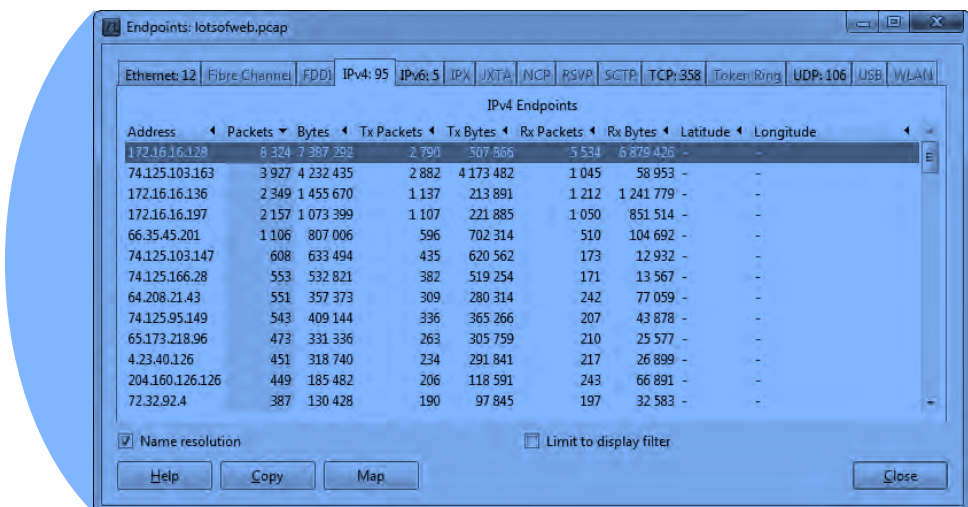


Figure 5-4: The Endpoints window shows which hosts are talking the most.

Given this information, would it be safe to assume that your top communicating endpoints comprise your largest conversation? If you now open the Conversations window and go to the IPv4 tab, you can indeed verify this by sorting the list by bytes. In this view, you can see that the traffic is consistent with a video download, because the number of bytes transmitted from Address A (74.125.103.163) greatly outnumbers the number of bytes transmitted from Address B (172.16.16.128) (see Figure 5-5).

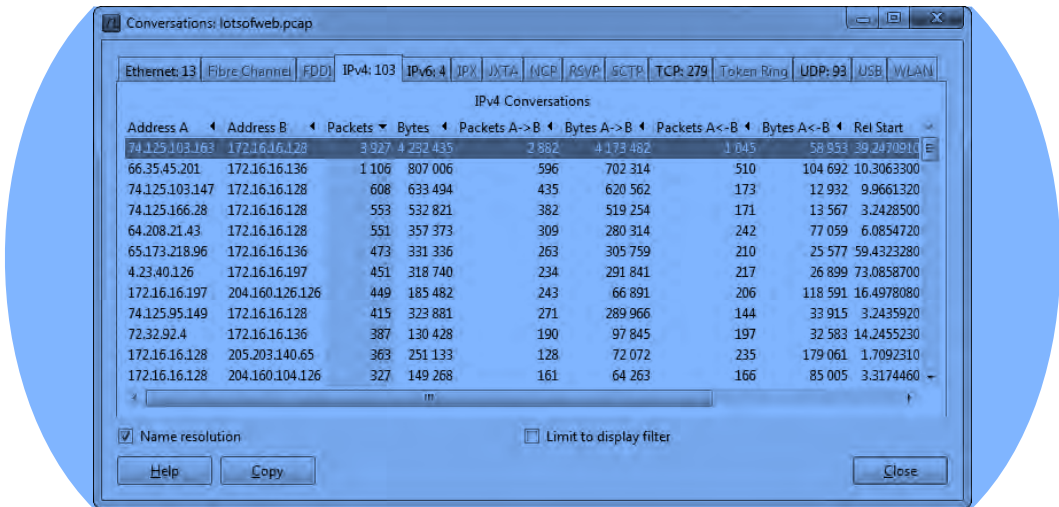


Figure 5-5: The Conversations window confirms that the two top talkers are communicating with each other.

You will see how to use the Endpoints and Conversations windows in practical scenarios later in this book.

Protocol Hierarchy Statistics

lotsofweb.pcap

When dealing with extremely large capture files, you sometimes need to determine the distribution of protocols in the file—that is, what percentage of a capture is TCP, IP, DHCP, and so on. Rather than counting each packet and totaling the results, you can use Wireshark's Protocol Hierarchy Statistics window, which is a great way to benchmark your network. For instance, if you know that 10 percent of your network traffic is usually made up of ARP traffic, and one day you take a capture that is 50 percent ARP traffic, then you know something might be wrong.

With the *lotsofweb.pcap* file still open, open the Protocol Hierarchy Statistics window (shown in Figure 5-6) by choosing **Statistics ► Protocol Hierarchy**. Notice that not all totals add up to exactly 100 percent. Because many of the packets contain multiple protocols from various layers, the count of each protocol as compared to each packet may be off. Nevertheless, you will still get an accurate view of the distribution of protocols in the capture file.

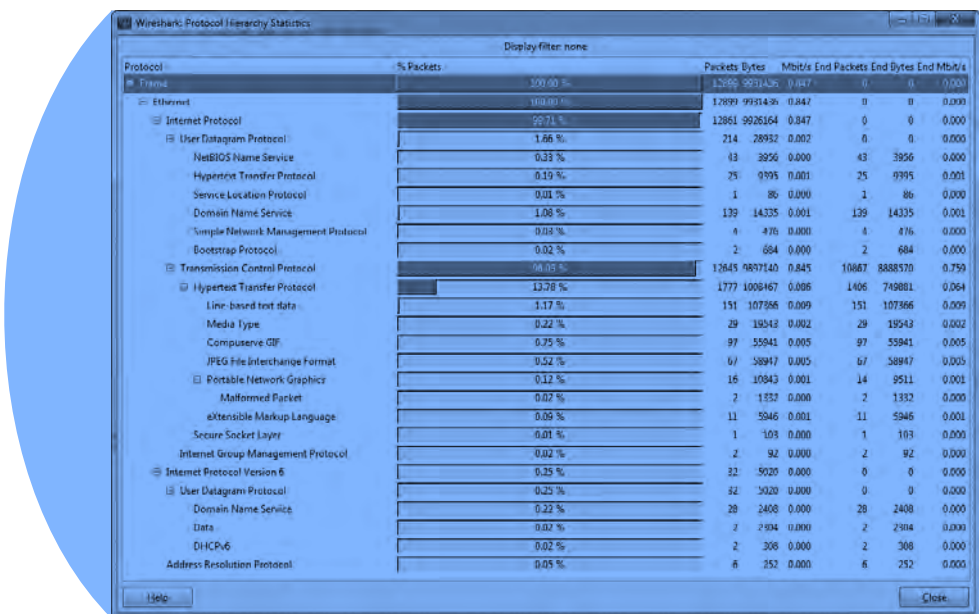


Figure 5-6: The Protocol Hierarchy Statistics window shows the distribution of various protocols.

The Protocol Hierarchy Statistics window is often one of the first windows you look at when examining traffic. It really gives you a good snapshot of the type of activity occurring on a network. As you begin to look at more traffic, you will eventually be able to profile the users and devices on a network just by looking at the distribution of protocols in use. I've found that simply by looking at traffic from a network segment, I can often immediately identify the network segment as belonging to the IT department due to the presence of administrative protocols such as ICMP or SNMP, or to the order-fulfillment department due to the high volume of SMTP traffic, or even to that pesky new intern in the corner with his *World of Warcraft* traffic!

Name Resolution

Network data is transported via various alphanumeric addressing systems that are often too long or complicated to remember, such as the physical hardware address 00:16:CE:6E:8B:24. *Name resolution* (also called *name lookup*) is the process a protocol uses to convert one identifying address into another. For example, while a computer might have the physical MAC address 00:16:CE:6E:8B:24, the DNS and ARP protocols allow us to see its name as *Marketing-2.domain.com*. By associating easy-to-read names with these cryptic addresses, we make them easier to remember and identify.

Enabling Name Resolution

To enable name resolution, open the Capture Options dialog by choosing **Capture ▶ Options**. As shown in Figure 5-7, three types of name resolution are available in Wireshark:

MAC name resolution This type of name resolution uses the ARP protocol to attempt to convert layer 2 MAC addresses, such as 00:09:5B:01:02:03, into layer 3 addresses, such as 10.100.12.1. If attempts at these conversions fail, Wireshark will use the *ethers* file in its program directory to attempt conversion. Wireshark's last resort is to convert the first 3 bytes of the MAC address into the device's IEEE-specified manufacturer name, such as *Netgear_01:02:03*.

Network name resolution This type of name resolution attempts to convert a layer 3 address, such as the IP address 192.168.1.50, into an easy-to-read DNS name such as *MarketingPC1.domain.com*.

Transport name resolution This type of name resolution attempts to convert a port number into a name associated with it. An example of this would be to display port 80 as *http*.

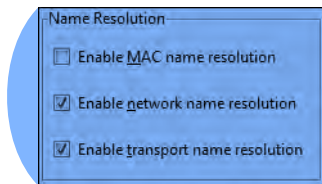


Figure 5-7: Enabling name resolution in the Capture Options dialog

You can leverage the various name resolution tools to make your capture files more readable and to save a lot of time in certain situations. For example, you can use DNS name resolution to help readily identify the name of a computer you are trying to pinpoint as the source of a particular packet.

Potential Drawbacks to Name Resolution

Given its benefits, using name resolution may seem like a no-brainer, but there are some potential drawbacks, including the following:

- Name resolution can fail, typically because the name is unknown by the name server the query was sent to.
- Name resolution must take place every time you open a specific capture file because this information is not saved in the file. This means that if the servers that a file's name resolution depends on are not available, name resolution will fail.
- The dependence on DNS may cause additional packets to be generated. The resulting traffic to resolve all DNS-based addresses will cloud your capture file. It's typically a rule of thumb that you don't want to see your own traffic on the wire when analyzing another issue.

- Name resolution requires additional processing overhead. If you are dealing with a very large capture file and are running low on memory, you may want to forgo the name resolution feature in order to conserve system resources.

Protocol Dissection

A *protocol dissector* allows Wireshark to break down a protocol into various sections so that it can be analyzed. For example, the ICMP protocol dissector allows Wireshark to take the raw data off the wire and format it as an ICMP packet.

You can think of a dissector as the translator between the raw data flowing across the wire and the Wireshark program. In order for a protocol to be supported by Wireshark, it must have a dissector built into it (or you can write your own in C or Python).

Wireshark uses several dissectors in unison to interpret each packet. It determines which dissectors to use by using its programmed logic and making a well-educated guess.

Changing the Dissector

wrongdissector.pcap

Unfortunately, Wireshark does not always make the right choices when selecting the dissector to use on a packet. This is especially true when it is using a protocol on the network in a nonstandard configuration, such as a nondefault port (which is often configured by network administrators as a security precaution or by employees trying to circumvent access controls). Luckily, we can change the way Wireshark implements certain dissectors.

For example, open the trace file *wrongdissector.pcap*. Notice that this file contains a bunch of SSL communication between two computers. SSL is the Secure Socket Layer protocol, which is used for secure encrypted communication between hosts. Under most normal circumstances, viewing SSL traffic in Wireshark won't yield much usable information due to its encrypted nature. However, there is something definitely wrong here. If you peruse the contents of several of these packets by clicking them and examining the Packet Bytes pane, you will quickly find plaintext traffic. In fact, if you look at packet 4, you will find mention of the FileZilla FTP server application. The next few packets clearly display a request and response for both a username and a password.

If this were actually SSL traffic, you wouldn't be able to read any of the data contained in the packets, and you certainly wouldn't see all usernames and passwords transmitted in the clear (see Figure 5-8). Given the information that is shown here, it is safe to assume that this is probably FTP traffic, rather than SSL traffic. This is most likely because this FTP traffic is using port 443, which is the standard port used for HTTPS (HTTP over SSL).

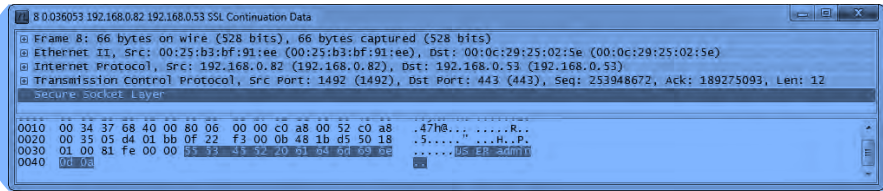


Figure 5-8: Plaintext usernames and passwords? This looks more like FTP than SSL!

To fix this problem, you can force Wireshark to use the FTP protocol dissector on these packets, a process referred to as a *forced decode*. To perform this process:

1. Right-click one of the **SSL** packets and select **Decode As**. This will bring up a dialog in which you can select the dissector you wish to use.
2. Tell Wireshark to decode all TCP source port **443** traffic using the **FTP** dissector by selecting **destination (443)** from the drop-down menu, and then selecting **FTP** under the Transport tab (see Figure 5-9).

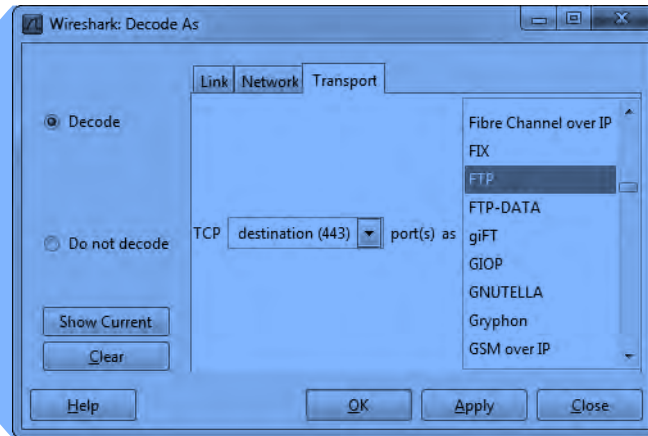


Figure 5-9: The Decode As dialog allows you to create forced decodes.

3. Once you have made your selections, click **OK** to see the changes immediately applied to the capture file.

You should see the data nicely decoded so that you can analyze it from the Packet List pane without needing to dig deep into its individual bytes.

WARNING The changes you make when creating a forced decode are not saved when you save the capture file and close Wireshark. You must re-create your forced decodes every time you open the capture file.

You can use the forced decode feature multiple times within the same capture file. Because it can be hard to keep track of the forced decodes you have applied when you use more than one in a capture file, Wireshark does this for you. From the Decode As dialog, you can click the Show Current button to display all of the forced decodes you have created so far (see Figure 5-10). You can also clear them by clicking the Clear button.

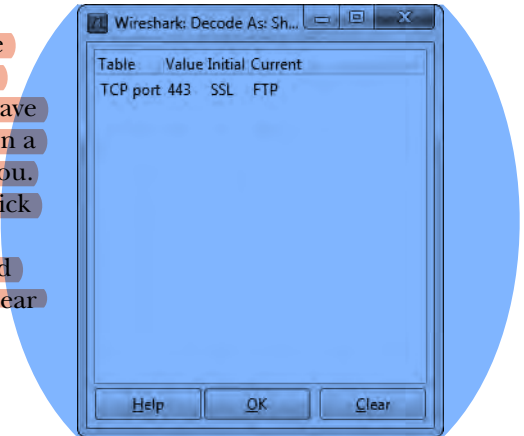


Figure 5-10: Clicking the Show Current button shows all of the forced decodes you have created for a capture file.

Viewing Dissector Source Code

The beauty of working with an open source application is that if you are confused as to why something is occurring, you can look at the source code and find out the exact reason. This really comes in handy when trying to determine why a particular protocol has been interpreted incorrectly.

Examining the source code of protocol dissectors can be done directly from the Wireshark website by hovering over the Develop link and clicking Browse the Code. This link will send you to the Wireshark subversion repository, where you can view the current release code for Wireshark as well as the code for previous releases. Clicking the releases folder will present you with all of the official Wireshark (and even Ethereal) releases, with the newest at the bottom of the list. Once you select the release you want to examine, the protocol dissectors can be found in the *epan/dissectors* folder. Each dissector is labeled with *packets-protocolname.c*.

These files can be rather complex, but you will find they all follow a standard template and tend to be commented very well. You don't need to be an expert C programmer to understand the basic function of each dissector. If you want to get a truly deep understanding of what you are seeing in Wireshark, I recommend at least taking a look at dissectors for some of the simpler protocols.

Following TCP Streams

[http_google.pcap](#)

One of Wireshark's most satisfying analysis features is its ability to reassemble TCP streams into an easily readable format. Rather than viewing data being sent from client to server in a bunch of small chunks, the Follow TCP Stream feature sorts the data to make it easier to view. This comes in handy when viewing plaintext application layer protocols such as HTTP, FTP, and so on. (We'll take a closer look at how these common protocols work in the next chapter.)

For example, let's consider a simple HTTP transaction. Open the file `http_google.pcap`. Click any of the TCP or HTTP packets in the file, right-click the file, and choose **Follow TCP Stream**. This will bring up the TCP stream in a separate window (see Figure 5-11).

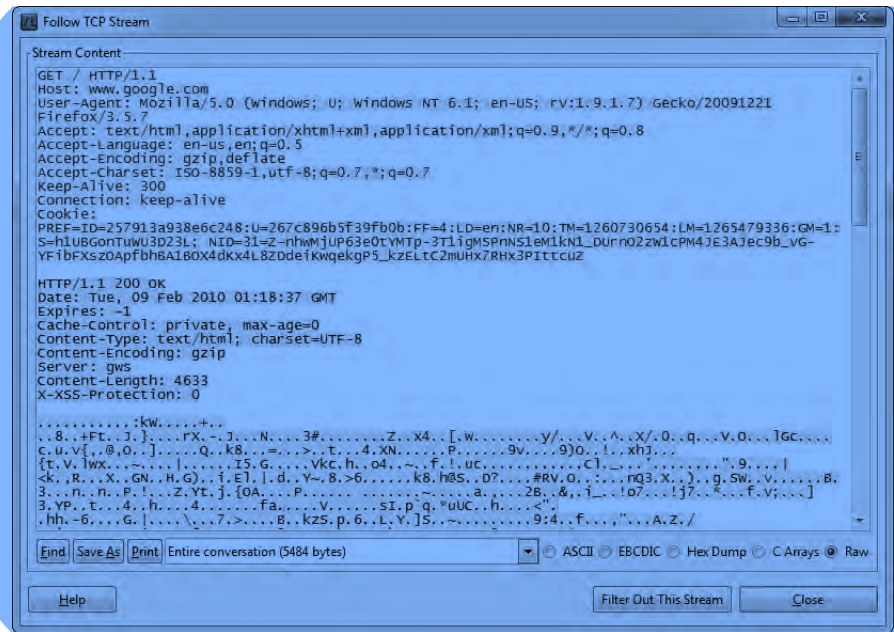


Figure 5-11: The Follow TCP Stream window reassembles the communication in an easily readable format.

Notice that the text displayed in this window is in two colors. The red text is used to signify traffic from the source to the destination, and the blue text is used to identify traffic in the opposite direction, from the destination to the source. The color relates to which side initiated the communication. For instance, in our example, the client initiated the connection to the web server, so it is displayed in red.

Given this TCP stream, you can clearly see a great majority of the communication between these two hosts. This communication begins with an initial GET request for the web root director (/) and a response from the server that the request was successful in the form of an HTTP/1.1 200 OK. A similar pattern is repeated throughout the stream as individual files are requested by the client and the server responds with them. You are seeing a user browsing to the Google home page. You're actually seeing what the end user is seeing, but from the inside out.

In addition to viewing the raw data in this window, you can also search within the text, save it as a file, print it, or choose to view the data in ASCII, EBCDIC, hex, or C array format. These options can be found at the bottom of the Follow TCP Stream window.

Following TCP streams will become your best friend when dealing with certain protocols.

Packet Lengths

`download-slow.pcap`

The size of a single packet or group of packets can tell you a lot about a situation. Under normal circumstances, the maximum size of a frame on an Ethernet network is 1,518 bytes. When you subtract the Ethernet, IP, and TCP headers from this number, that leaves you with 1,460 bytes that can be used for the transmission of a layer 7 protocol header or data. With that knowledge, you can begin to use the distribution of packet lengths in a capture to make some educated guesses about the traffic.

Opening the file `download-slow.pcap` will provide a great example of this. Once the file is opened, select **Statistics ▸ Packet Lengths** and click **Create Stat**. The result is the window shown in Figure 5-12.

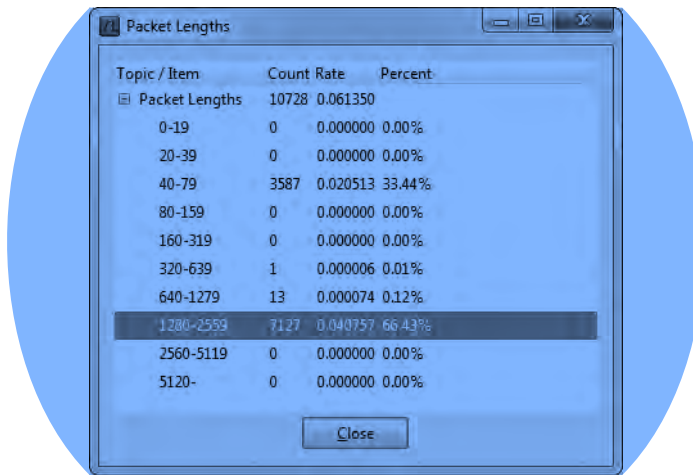


Figure 5-12: The Packet Lengths window helps you make educated guesses about the traffic in the capture file.

I've highlighted the section showing statistics for packets ranging from 1,280 to 2,559 bytes in size. Larger packets such as these typically indicate the transfer of data, whereas smaller packets indicate protocol control sequences. In this case, we have a fairly large percentage of large packets (66.43 percent). Without even seeing the packets in the file, we can conclude that the capture file contains one or multiple transfers of data. This could be in the form of an HTTP download, an FTP upload, or any other type of network communication where data is transferred between hosts.

Most of the remaining packets (33.44 percent) are in the 40 to 79 bytes range. Packets in this range are usually TCP control packets that don't carry data. Let's consider the typical size of protocol headers. The Ethernet header is 14 bytes (plus a 4-byte CRC), the IP header is a minimum of 20 bytes, and a TCP packet with no data or options is also 20 bytes. This means that standard TCP control packets—such as SYN, ACK, RST, and FIN packets—will be around 54 bytes in size and fall in this range. Of course, the addition of IP or TCP options will increase this size.

Examining packet lengths is a great way to get a bird's-eye view of a capture. If there are a lot of large packets, it may be safe to assume that data is being transferred. If the majority of packets are small, you may assume that the capture consists of protocol control commands, without a great deal of data being passed. These are not hard-and-fast rules, but making such assumptions is sometimes safe before taking on deeper analysis.

Graphing

Graphs are the bread and butter of analysis, and one of the best ways to get an overview of a data set. Wireshark includes a few different graphing features to assist in understanding capture data, the first of which is its IO graphing capabilities.

Viewing IO Graphs

download-fast
.pcap
download-slow
.pcap

Wireshark's IO Graphs window allows you to graph the throughput of data on a network. You can use such graphs to find spikes and lulls in data throughput, discover performance lags in individual protocols, and to compare simultaneous data streams.

To view an example of the IO graph of a computer as it downloads a file from the Internet, open *download-fast.pcap*. Click any TCP packet to highlight it, and then select **Statistics ▸ IO Graphs**.

The IO Graphs window shows a graphical view of the flow of data over the course of the capture file. In the example in Figure 5-13, you can see that the download that this graph represents averages around 500 packets per tick and stays somewhat consistent throughout its course, tapering off at the end.

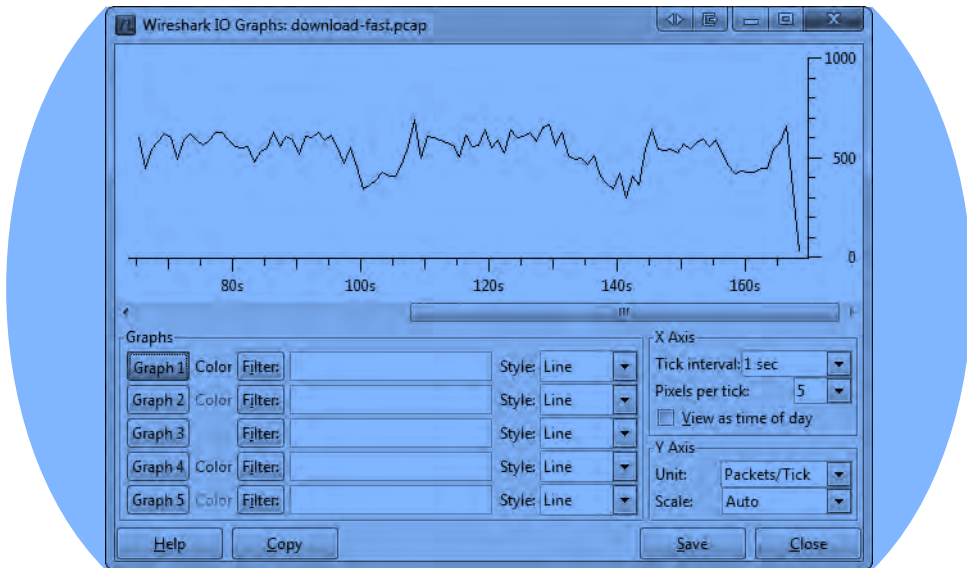


Figure 5-13: The IO graph of the fast download is mostly consistent.

Let's compare this to an example of a slower download. Leave the current file open, open another instance of Wireshark, and open *download-slow.pcap*. Bring up the IO graph of this download, and you will see a much different story, as shown in Figure 5-14.

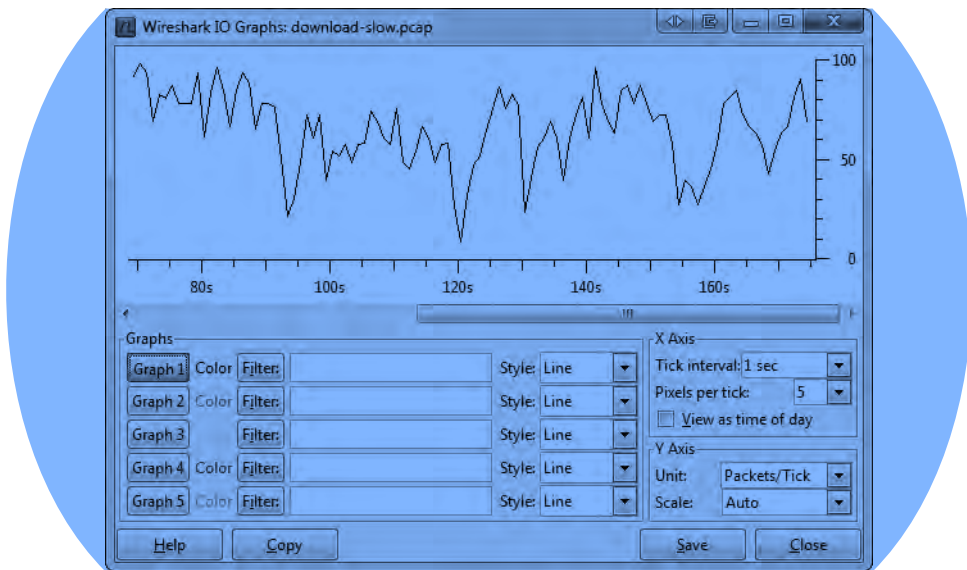


Figure 5-14: The IO graph of the slow download is not consistent at all.

This download has a transfer rate of between 0 and 100 packets per second, and is far from consistent, sometimes even momentarily nearing 0 packets per second. You can see these inconsistencies more clearly if you place the IO graphs of the two files next to each other (see Figure 5-15).

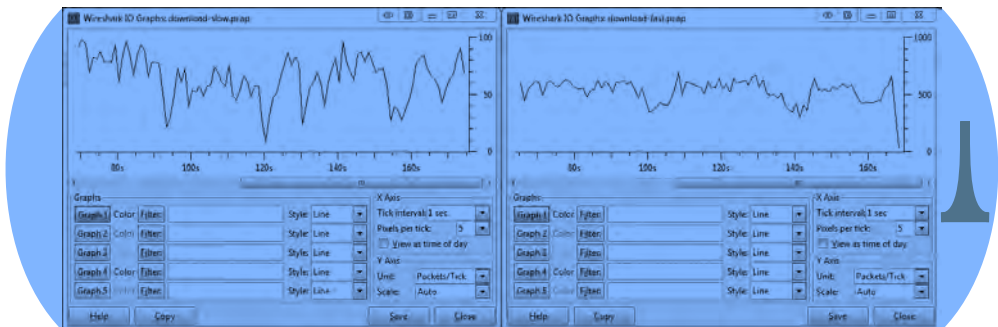


Figure 5-15: Viewing multiple IO graphs side by side can be helpful in spotting variance.

Notice the configurable options at the bottom of this window. You can create up to five unique filters (using the same syntax as a display or capture filter, as discussed in Chapters 6 and 7) and specify display colors for those filters. For instance, you could create filters to show ARP and DHCP traffic, and display the lines on the graph in red and blue so that you can more easily differentiate the throughput trends between these two protocol types.

download-fast
.pcap

Round-Trip Time Graphing

Another graphing feature of Wireshark is the ability to view a plot of round-trip times for a given capture file. The *round-trip time (RTT)* is the time it takes for an acknowledgment to be received for a packet. Effectively, this is the time it took your packet to get to its destination and for the acknowledgment of that packet to be sent back to you. Analysis of RTTs is often done to find slow points or bottlenecks in communication and to determine if there is any latency.

Let's try out this feature. Open the file *download-fast*. View the RTT graph of this file by selecting a TCP packet, and then choosing **Statistics ▸ TCP Stream Graph ▸ Round Trip Time Graph**. The RTT graph for *download-fast.pcap* is shown in Figure 5-16.

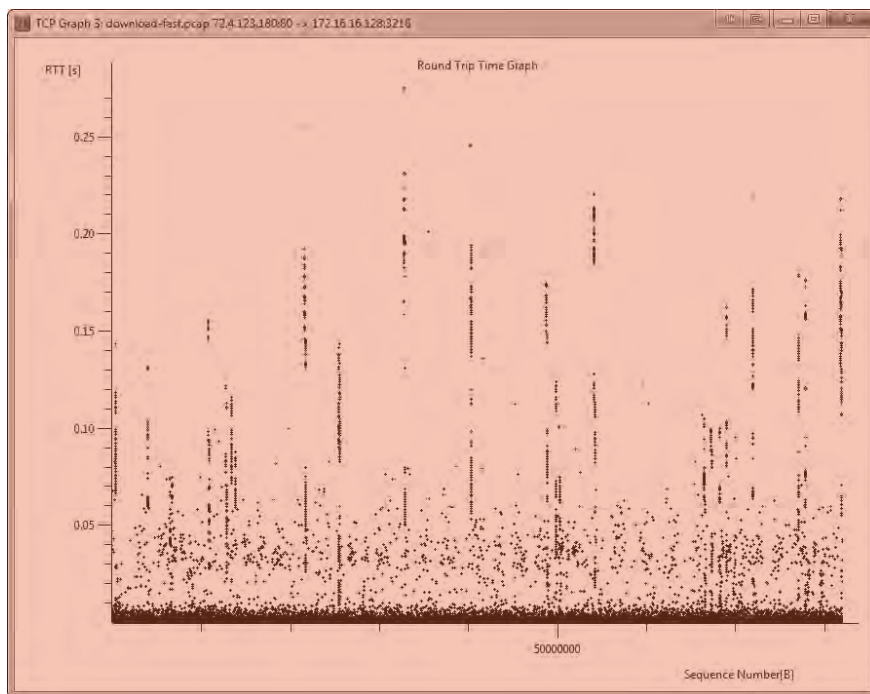


Figure 5-16: The RTT graph of this download appears mostly consistent, with only a few stray values.

Each point in the graph represents the RTT of a packet. The default view shows these values sorted by sequence number. You can click a plotted point within the graph to be taken directly to that packet in the Packet List pane.

It appears as though the RTT graph for the fast download has RTT values mostly under 0.05 seconds, with a few slower points between 0.10 and 0.25 seconds. Although there are quite a few values above acceptable limits, the majority of the RTT values are okay, so this would be considered an acceptable RTT for a file download.

Flow Graphing

`http_google.pcap`

The flow graphing feature is very useful for visualizing connections and showing the flow of data over time. Basically, a flow graph contains a column-based view of a connection between hosts and organizes the traffic so you can interpret it visually.

To create a flow graph, open the file `http_google.pcap` and select **Statistics ▶ Flow Graph**. You'll see a small dialog that gives a few simple options regarding the packets to process and the flow type. Just accepting the default values will be fine for this example, so click **OK** to generate the flow graph (see Figure 5-17).



Figure 5-17: The TCP flow graph allows us to visualize the connection much better.

Expert Information

`download-slow.pcap`

The dissectors for each protocol in Wireshark define *expert info* that can be used to alert you about particular states within a packet using that protocol. These states are separated into four categories:

Chat Basic information about the communication

Note Unusual packets that may be part of normal communication

Warning Unusual packets that are most likely not a part of normal communication

Error An error in a packet or the dissector interpreting it

For example, open the file *download-slow.pcap*. Then click **Analyze**, and select **Expert Info Composite** to bring up the Expert Infos window for this capture file (see Figure 5-18).

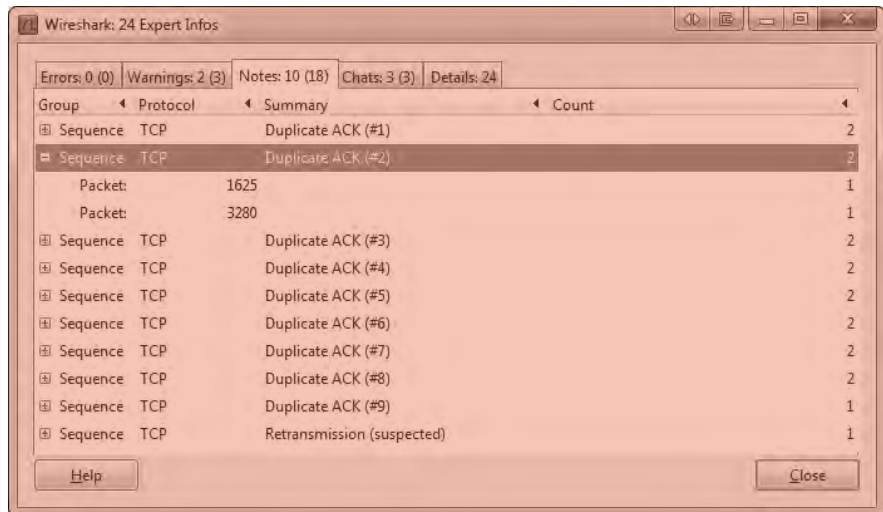


Figure 5-18: The Expert Infos window shows information from the expert system programmed within the protocol dissectors.

Notice that the window has tabs for each classification of information, and that there are no errors, 3 warnings, 18 notes, and 3 chats. On the tabs, the number not inside parentheses indicates the amount of unique messages, and the number inside parentheses is the total of occurrences for that category.

All of the messages within this capture file are TCP-related, simply because the expert information system hasn't really been implemented for any other protocols as of this writing. At this time, there are 14 expert info messages configured for TCP, and they are quite useful when troubleshooting capture files. These messages will flag an individual packet when it meets certain criteria, as listed here:

Chat messages

Window Update Sent by a receiver to notify a sender that the size of the TCP receive window has changed.

Note messages

TCP Retransmission Result of packet loss. Occurs when a duplicate ACK is received or the retransmission timer of a packet expires.

Duplicate ACK When a host doesn't receive the next sequence number it is expecting, it generates a duplicate ACK of the last data it received.

Zero Window Probe Used to monitor the status of the TCP receive window after a zero window packet has been transmitted (covered in Chapter 9).

Keep Alive ACK Sent in response to keep-alive packets.

Zero Window Probe ACK Sent in response to zero-window-probe packets.

Window is Full Used to notify a transmitting host that the receiver's TCP receive window is full.

Warning messages

Previous Segment Lost Indicates packet loss. Occurs when an expected sequence number in a data stream is skipped.

ACKed Lost Packet Occurs when an ACK packet is seen but the packet it is acknowledging is not.

Keep Alive Triggered when a connection keep-alive packet is seen.

Zero Window Seen when the size of the TCP receive window is reached and a zero window notice is sent out, requesting the sender to stop sending data.

Out-of-Order Utilizes sequence numbers to detect when packets are received out of sequence.

Fast Retransmission A retransmission that occurs within 20 milliseconds of a duplicate ACK.

Error messages

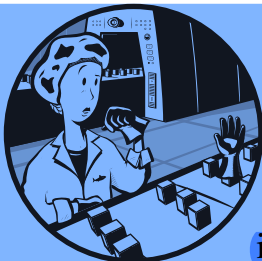
No Error Messages

The meaning of these messages will become clearer as we study TCP in Chapter 6 and troubleshooting slow networks in Chapter 9.

Although some of the features discussed in this chapter may seem as if they would be used in only obscure situations, you will probably find yourself using them more than you might expect. It is important that you familiarize yourself with these windows and options; I will be referencing them a lot in the next few chapters.

6

COMMON LOWER-LAYER PROTOCOLS



Whether troubleshooting latency issues, identifying malfunctioning applications, or zeroing in on security threats in order to be able to spot abnormal traffic, you must first understand normal traffic. In the next couple of chapters, you'll learn how normal network traffic works at the packet level. We'll look at the most common protocols, including the workhorses TCP, UDP, and IP, and more commonly used application-layer protocols such as HTTP, DHCP, and DNS. Each protocol section has at least one associated capture file, which you can download and work with directly. This chapter will specifically focus on the lower-layer protocols found in reference to layers 1 through 4 of the OSI model.

These are arguably the most important chapters in this book. Skipping the discussion would be like cooking Sunday supper without cornbread. Even if you already have a good grasp of how each protocol functions, give these chapters at least a quick read in order to review the packet structure of each.

Address Resolution Protocol

Both logical and physical addresses are used for communication on a network. The use of logical addresses allows for communication between multiple networks and indirectly connected devices. The use of physical addresses facilitates communication on a single network segment for devices that are directly connected to each other with a switch. In most cases, these two types of addressing must work together in order for communication to occur.

Consider a scenario where you wish to communicate with a device on your network. This device may be a server of some sort or just another workstation you need to share files with. The application you are using to initiate the communication is already aware of the IP address of the remote host (via DNS, covered in Chapter 7), meaning the system should have all it needs to build the layer 3 through 7 information of the packet it wants to transmit. The only piece of information it needs at this point is the layer 2 data link data containing the MAC address of the target host.

MAC addresses are needed because a switch that interconnects devices on a network uses a *Content Addressable Memory (CAM) table*, which lists the MAC addresses of all devices plugged into each of its ports. When the switch receives traffic destined for a particular MAC address, it uses this table to know through which port to send the traffic. If the destination MAC address is unknown, the transmitting device will first check for the address in its cache; if it is not there, then it must be resolved through additional communication on the network.

The resolution process that TCP/IP networking (with IPv4) uses to resolve an IP address to a MAC address is called the *Address Resolution Protocol (ARP)*, which is defined in RFC 826. The ARP resolution process uses only two packets: an ARP request and an ARP response (see Figure 6-1).

NOTE *An RFC, or Request for Comments, is the official document that defines the implementation standards for protocols. You can search for RFC documentation at the RFC Editor home page, <http://www.rfc-editor.org/>.*

The transmitting computer sends out an ARP request that basically asks, “Howdy everybody, my IP address is XX.XX.XX.XX, and my MAC address is XX:XX:XX:XX:XX:XX. I need to send something to whoever has the IP address XX.XX.XX.XX, but I don’t know its hardware address. Will whoever has this IP address please respond back with your MAC address?”

This packet is broadcast to every device on the network segment. Any device that does not have this IP address simply discards the packet. The device that does have this IP address sends an ARP reply with an answer like, “Hey, transmitting device, I’m who you are looking for with the IP address of XX.XX.XX.XX. My MAC address is XX:XX:XX:XX:XX:XX.”

Once this resolution process is completed, the transmitting device updates its cache with the MAC-to-IP address association of this device, and it can begin sending data.

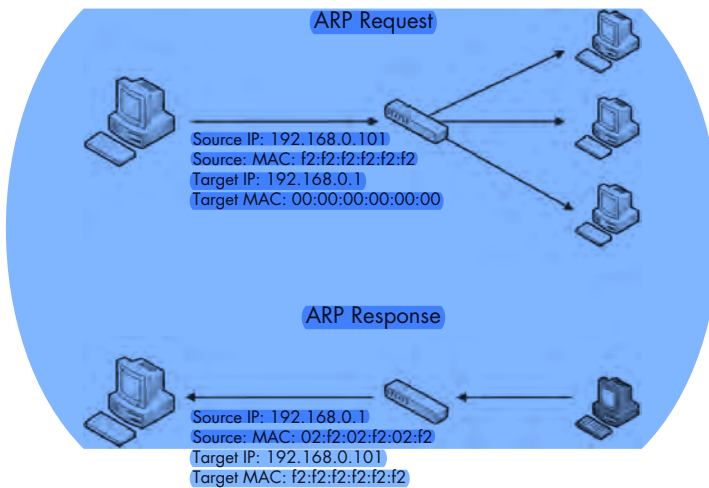


Figure 6-1: The ARP resolution process

NOTE You can view the ARP table of a Windows host by typing `arp -a` from a command prompt.

Seeing this process in action will help you to understand how it works. But before we look at some examples, let's examine the ARP packet header.

The ARP Header

As shown in Figure 6-2, the ARP header includes the following fields:

Hardware Type The layer 2 type used. In most cases, this is Ethernet (type 1).

Protocol Type The higher-layer protocol for which the ARP request is being used.

Hardware Address Length The length (in octets/bytes) of the hardware address in use (6 for Ethernet).

Protocol Address Length The length (in octets/bytes) of the logical address of the specified protocol type.

Operation The function of the ARP packet: either 1 for a request or 2 for a reply.

Sender Hardware Address The hardware address of the sender.

Sender Protocol Address The sender's upper-layer protocol address.

Target Hardware Address The intended receiver's hardware address (zeroed in ARP requests).

Target Protocol Address The intended receiver's upper-layer protocol address.

Address Resolution Protocol		
Bit Offset	0–7	8–15
0	Hardware Type	
16	Protocol Type	
32	Hardware Address Length	Protocol Address Length
48	Operation	
64	Sender Hardware Address (1st 16 Bits)	
80	Sender Hardware Address (2nd 16 Bits)	
96	Sender Hardware Address (3rd 16 Bits)	
112	Sender Protocol Address (1st 16 Bits)	
128	Sender Protocol Address (2nd 16 Bits)	
144	Target Hardware Address (1st 16 Bits)	
160	Target Hardware Address (2nd 16 Bits)	
176	Target Hardware Address (3rd 16 Bits)	
192	Target Protocol Address (1st 16 Bits)	
208	Target Protocol Address (2nd 16 Bits)	

Figure 6-2: The ARP packet structure

Now open the file `arp_resolution.pcap` to see this resolution process in action. We'll focus on each packet individually as we walk through this process.

Packet 1: ARP Request

`arp_resolution.pcap`

The first packet is the ARP request, as shown in Figure 6-3. We can confirm that this packet is a true broadcast packet by examining the Ethernet header in Wireshark's Packet Details pane. The packet's destination address is `ff:ff:ff:ff:ff:ff` ❶. This is the Ethernet broadcast address, and anything sent to it will be broadcast to all devices on the current network segment. The source address of this packet in the Ethernet header is listed as our MAC address ❷.

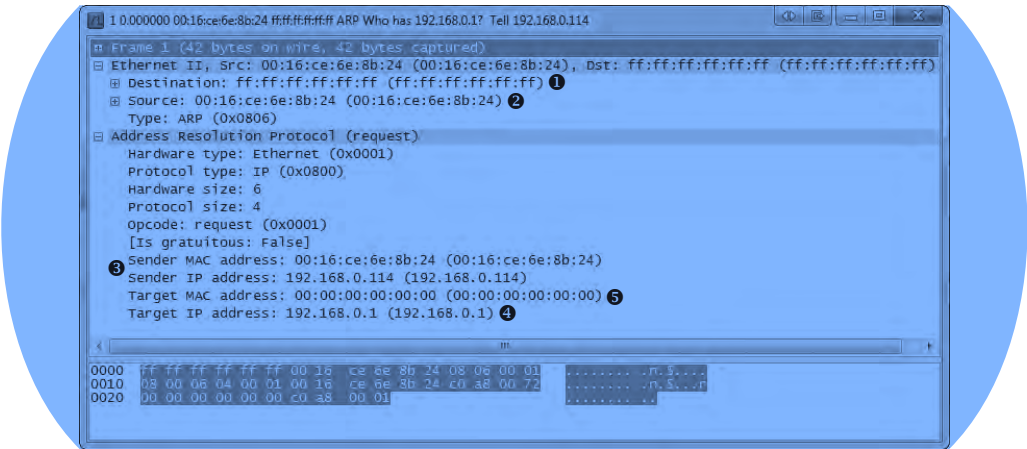


Figure 6-3: An ARP request packet

Given this structure, we can discern that this is indeed an ARP request on an Ethernet network using IP. The sender's IP address (192.168.0.114) and MAC address (00:16:ce:6e:8b:24) are listed ❸, as is the IP address of the target (192.168.0.1) ❹. The MAC address of the target—the information we are trying to get—is unknown, so the target MAC is listed as 00:00:00:00:00:00 ❺.

Packet 2: ARP Response

In our response to the initial request (see Figure 6-4), the Ethernet header now has a destination address of the source MAC address from the first packet. The ARP header looks similar to that of the ARP request, with a few changes:

- The packet's operation code (opcode) is now 0x0002 ❶, indicating a reply rather than a request.
- The addressing information is reversed—the sender MAC address and IP address are now the target MAC address and IP address ❷.
- Most important, all of the information is present, meaning we now have the MAC address (00:13:46:0b:22:ba) ❸ of our host at 192.168.0.1.

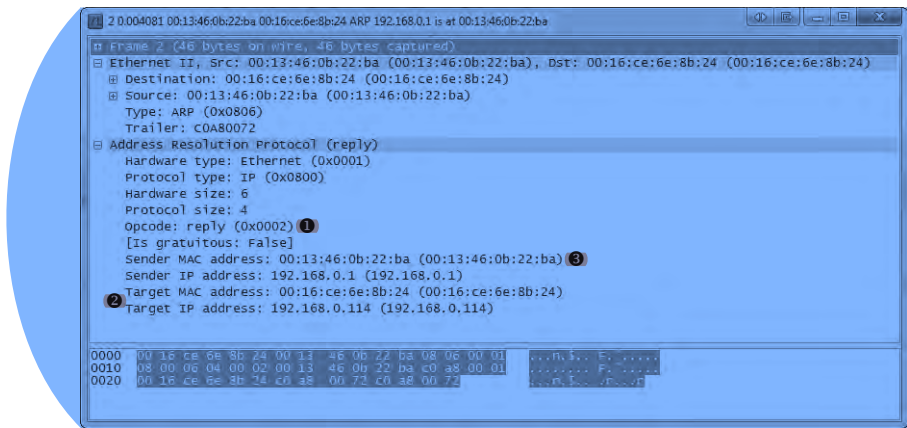


Figure 6-4: An ARP reply packet

Gratuitous ARP

arp_gratuitous
.pcap

Where I come from, when something is done “gratuitously,” that usually carries a negative connotation. A *gratuitous ARP*, however, is actually a good thing.

In many cases, a device's IP address can change. When this happens, the IP-to-MAC address mappings that hosts on the network have in their caches will be invalid. To prevent this from causing communication errors, a gratuitous ARP packet is transmitted on the network to force any device that receives it to update its cache with the new IP-to-MAC address mapping (see Figure 6-5).

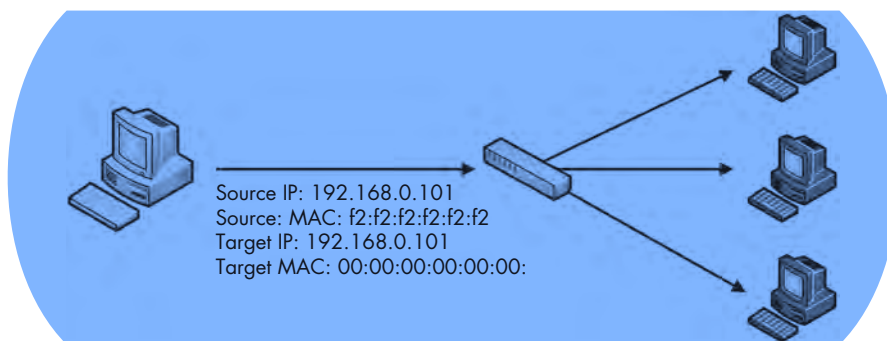


Figure 6-5: The gratuitous ARP process

A few different scenarios can spawn a gratuitous ARP packet. One of the most common is the changing of an IP address. Open the capture file *arp_gratuitous.pcap*, and you'll see this in action. This file contains only a single packet (see Figure 6-6) because that's all that's involved in gratuitous ARP.

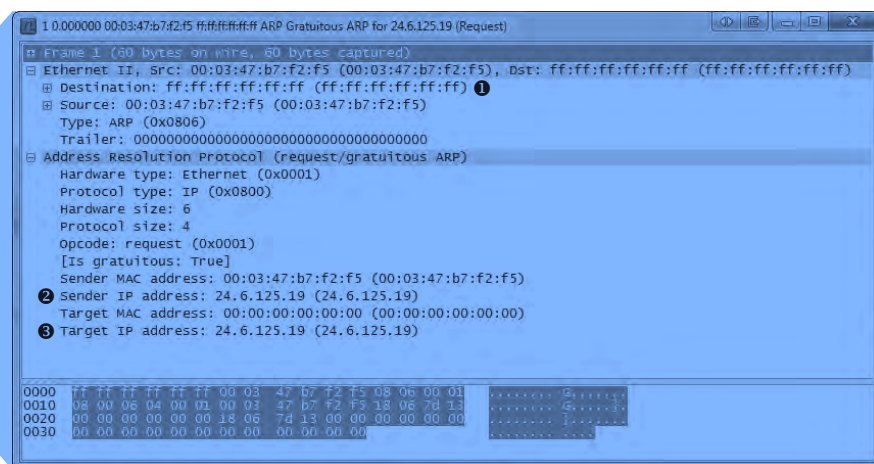


Figure 6-6: A gratuitous ARP packet

Examining the Ethernet header, you can see that this packet is sent as a broadcast so that all hosts on the network receive it ❶. The ARP header looks like an ARP request, except that the sender IP address ❷ and the target IP address ❸ are the same. When received by other hosts on the network, this packet will cause them to update their ARP tables with the new IP-to-MAC address association. Because this ARP packet is unsolicited but results in a client updating its ARP cache, the packet is considered gratuitous.

You will notice gratuitous ARP packets in a few different situations. As mentioned, changing a device's IP address will generate one. Also, some operating systems will perform a gratuitous ARP on startup. Additionally, you may notice gratuitous ARP packets on systems that use them for load-balancing of incoming traffic.

Internet Protocol

The primary purpose of protocols at layer 3 of the OSI model is to allow for communication between networks. As you just saw, MAC addresses are used for communication on a single network at layer 2. In much the same fashion, layer 3 is responsible for addresses for internetwork communication. A few protocols can do this, but the most common is the *Internet Protocol (IP)*. Here, we'll examine IP version 4 (IPv4), which is defined in RFC 791.

In order to understand the functionality of IPv4, you need to know how traffic flows between networks. IPv4 is the workhorse of the communication process and is ultimately responsible for carrying data between devices, regardless of where the communication endpoints are located.

A simple network in which all devices are connected via hubs or switches is called a *local area network (LAN)*. When you want to connect two LANs together, you can do so with a router. Complex networks can consist of thousands of LANs connected through thousands of routers worldwide. The Internet itself is a collection of millions of LANs and routers.

IP Addresses

IP addresses are 32-bit addresses used to uniquely identify devices connected to a network. It is a bit much to expect someone to remember a sequence of ones and zeros that is 32 characters long, so IP addresses are written in *dotted-quad notation*.

In dotted-quad notation, each of the four sets of ones and zeros that make up an IP address is converted to base 10 and represented as a number between 0 and 255 in the format *A.B.C.D* (see Figure 6-7). For example, consider the IP address 11000000 10101000 00000000 00000001. This value is obviously a bit much to remember or notate. Fortunately, using dotted-quad notation, we can represent it as 192.168.0.1.

IP addresses are divided into four distinct parts for a reason. An IP address consists of two parts: a *network address* and a *host address*. The network address identifies the LAN the device is connected to, and the host address identifies the device itself on that network. The determination of which part of the IP address belongs to the network or host address is not always the same. This is actually determined by another set of addressing information called the *network mask (netmask)*, sometimes also referred to as a *subnet mask*.

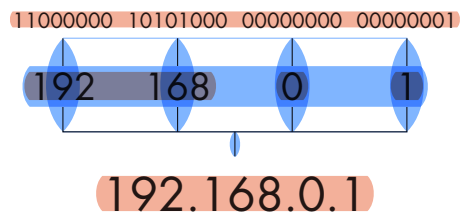


Figure 6-7: Dotted-quad IPv4 address notation

The netmask identifies which portion of the IP address belongs to the network address and which part belongs to the host address. The netmask number is also 32 bits long, and every bit that is set to a 1 identifies the portion of the IP address that is reserved for the network address. The remaining bits set to 0 identify the host address.

For example, consider the IP address 10.10.1.22, represented in binary as 00001010 00001010 00000001 00010110. In order to determine the allocation of each section of the IP address, we can apply our netmask. In this case, our netmask is 11111111 11111111 00000000 00000000. This means that the first half of the IP address is reserved for the network address (10.10 or 00001010 00001010) and the last half of the IP address identifies the individual host on this network (.1.22 or 00000001 00010110), as shown in Figure 6-8.

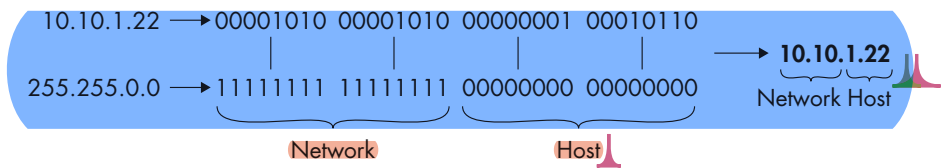


Figure 6-8: The netmask determines the allocation of the bits in an IP address.

Netmasks can also be written in dotted-quad notation. For example, the netmask 11111111 11111111 00000000 00000000 is written as 255.255.0.0.

IP addresses and netmasks are commonly written in *Classless Inter-Domain Routing (CIDR) notation* for shorthand. In this form, an IP address is written in full, followed by a forward slash (/) and the number of bits that represent the network portion of the IP address. For example, an IP address of 10.10.1.22 and a netmask of 255.255.0.0 would be written in CIDR notation as 10.10.1.22/16.

The IPv4 Header

The source and destination IP addresses are the crucial components of the IPv4 packet header, but that's not all of the IP information you will find within a packet. The IP header is actually quite complex compared with the ARP packet we just examined. It includes a lot of extra functionality that helps IP do its job.

As shown in Figure 6-9, the IPv4 header has the following fields:

Version The version of IP being used

Header Length The length of the IP header

Type of Service A precedence flag and type of service flag, which are used by routers to prioritize traffic

Total Length The length of the IP header and the data included in the packet

Identification A unique identification number used to identify a packet or sequence of fragmented packets

Flags Used to identify whether or not a packet is part of a sequence of fragmented packets

Fragment Offset If a packet is a fragment, the value of this field is used to reassemble the packets in the correct order.

Time to Live Defines the lifetime of the packet, measured in hops/seconds through routers

Protocol Used to identify the type of packet coming next in the sequence of packets

Header Checksum An error-detection mechanism used to verify the contents of the IP header are not damaged or corrupted

Source IP Address The IP address of the host that sent the packet

Destination IP Address The IP address of the packet's destination

Options Reserved for additional IP options. It includes options for source routing and timestamps.

Data The actual data being transmitted with IP

Internet Protocol					
Bit Offset	0–3	4–7	8–15	16–18	19–31
0	Version	HDR Length	Type of Service	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol	Header Checksum	
96	Source IP Address				
128	Destination IP Address				
160	Options				
160 or 192+	Data				

Figure 6-9: The IPv4 packet structure

Time to Live

ip_ttl_source.pcap
ip_ttl_dest.pcap

The *Time to Live (TTL)* value defines a period of time that can be elapsed or a maximum number of routers a packet can traverse before the packet is discarded. A TTL is defined when a packet is created, and generally is decremented by 1 every time the packet is forwarded by a router. For example, if a packet has a TTL of 2, the first router it reaches will decrement the TTL to 1 and forward it to the second router. This router will then decrement the TTL to 0, and if the final destination of the packet is not on that network, the packet will be discarded (see Figure 6-10). Since the TTL value is technically time-based, a very busy router could decrement the TTL value by more than 1, but generally, it's safe to assume that one routing device will decrement a TTL by only 1 most of the time.

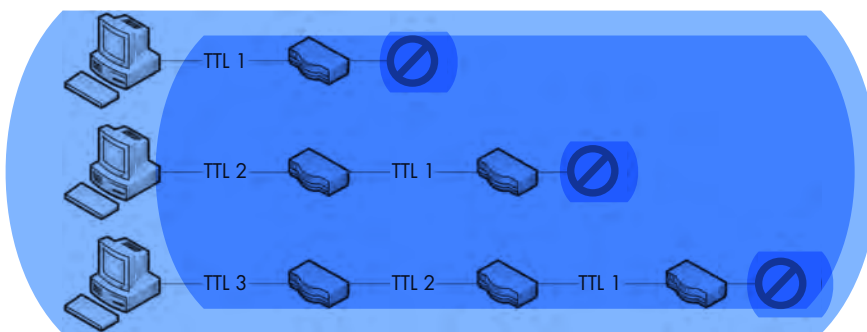


Figure 6-10: The TTL of a packet decreases every time it traverses a router.

Why is the TTL value important? Typically, we are concerned about the lifetime of a packet only in terms of the time that it takes to travel from its source to its destination. However, consider a packet that must travel to a host across the Internet while traversing dozens of routers. At some point in that packet's path, it could encounter a misconfigured router and lose the path to its final destination. In such a case, the router could do a number of things, one of which could result in the packet being forwarded around a network in a never-ending loop.

If you have any programming background at all, you know that a loop that never ends can cause all sorts of issues, typically resulting in a program or an entire operating system crashing. Theoretically, the same thing could occur with packets on a network. The packets would keep looping between routers. As the number of looping packets increased, the available bandwidth on the network would deplete until a DoS condition occurred. To prevent this potential problem, the TTL field of the IP header was created.

Let's look at an example of this in Wireshark. The file `ip_ttl_source.pcap` contains two ICMP packets. ICMP (discussed later in this chapter) utilizes IP to deliver packets, as we can see by expanding the IP header section in the Packet Details pane (see Figure 6-11).

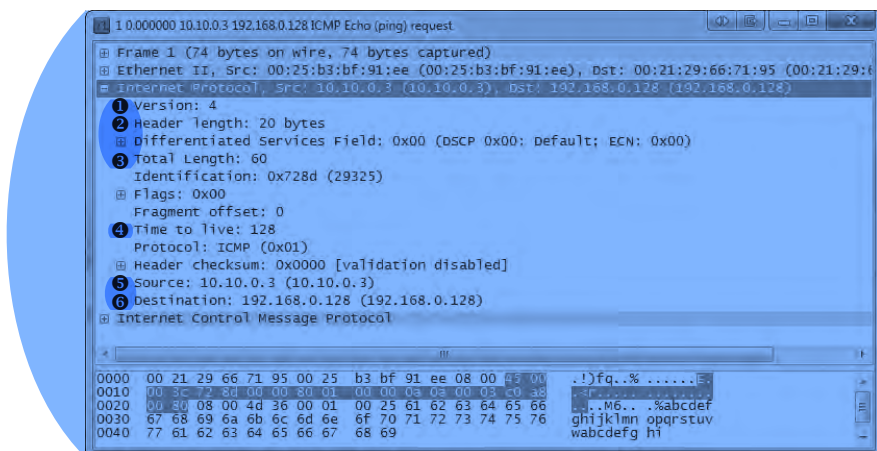


Figure 6-11: The IP header of the source packet

You can see that the version of IP being used is version 4 ❶, the IP header length is 20 bytes ❷, the total length of the header and payload is 60 bytes ❸, and the value of the TTL field is 128 ❹.

The primary purpose of an ICMP ping is to test communication between devices. Data is sent from one host to another as a request, and the receiving host should send that data back as a reply. In this file, we have one device with the address of 10.10.0.3 ❺ sending an ICMP request to a device with the address 192.168.0.128 ❻. This initial capture file was created at the source host, 10.10.0.3.

Now open the file `ip_ttl_dest.pcap`. In this file, the data was captured at the destination host, 192.168.0.128. Expand the IP header of the first packet in this capture to examine its TTL value (see Figure 6-12).

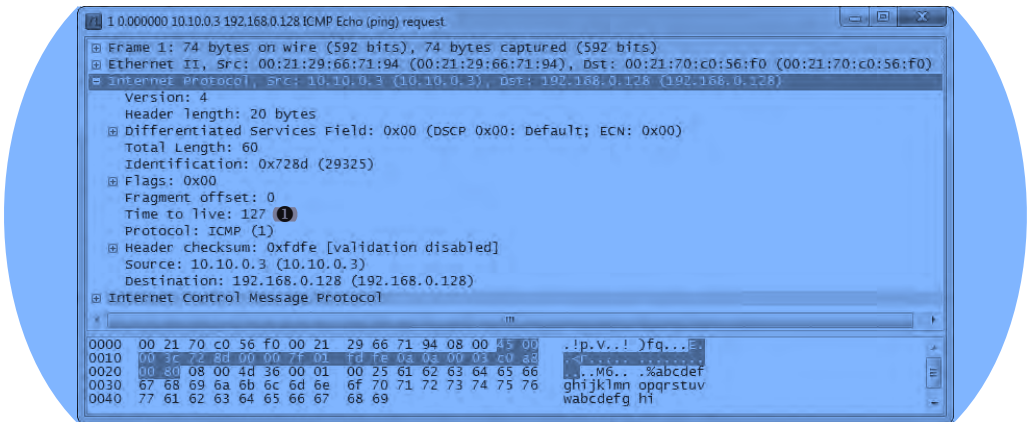


Figure 6-12: The IP header tells us that the TTL has been lowered by 1.

You should immediately notice that the TTL value is 127 ❶, one less than the original TTL of 128. Without even knowing the architecture of the network, we can conclude that these two devices are separated by one router and that the passage through that router reduced the TTL value by one.

IP Fragmentation

`ip_frag_source.pcap`

Packet fragmentation is a feature of IP that permits reliable delivery of data across varying types of networks by splitting a data stream into smaller fragments.

The fragmentation of a packet is based on the maximum transmission unit (MTU) size of the layer 2 data link protocol in use and the configuration of the devices using these layer 2 protocols. In most cases, the layer 2 data link protocol in use is Ethernet. Ethernet has a default MTU of 1500, which means that the maximum packet size that can be transmitted over an Ethernet network is 1,500 bytes (not including the 14-byte Ethernet header itself).

NOTE

Although there are standard MTU settings, the MTU of a device can be reconfigured manually in most cases. An MTU setting is assigned on a per-interface basis and can be modified on Windows and Linux systems, as well as on the interfaces of managed routers.

When a device prepares to transmit an IP packet, it determines whether it must fragment the packets by comparing the packet's data size to the MTU of the network interface from which the packet will be transmitted. If the data size is greater than the MTU, the packet will be fragmented. Fragmenting a packet involves the following steps:

1. The device splits the data into the number of packets required for successful data transmission.
2. The Total Length field of each header is set to the segment size of each fragment.
3. The More Fragments flag is set to 1 on all packets in the data stream, except for the last one.
4. The Fragment Offset field is set in the IP header of the fragments.
5. The packets are transmitted.

The file *ip_frag_source.pcap* was taken from a computer with the address 10.10.0.3, transmitting a ping request to a device with address 192.168.0.128. Notice that the Info column of the Packet List pane lists two fragmented IP packets, followed by the ICMP (ping) request.

Begin by examining the IP header of packet 1 (see Figure 6-13).

You can see that this packet is part of a fragment based on the More Fragments and Fragment Offset fields. Packets that are fragments either will have a positive Fragment Offset value or will have the More Fragments flag set. In the first packet, the More Fragments flag is set ①, indicating that the receiving device should expect to receive another packet in this sequence. The Fragment Offset is set to 0 ②, indicating this packet is the first in a series of fragments.

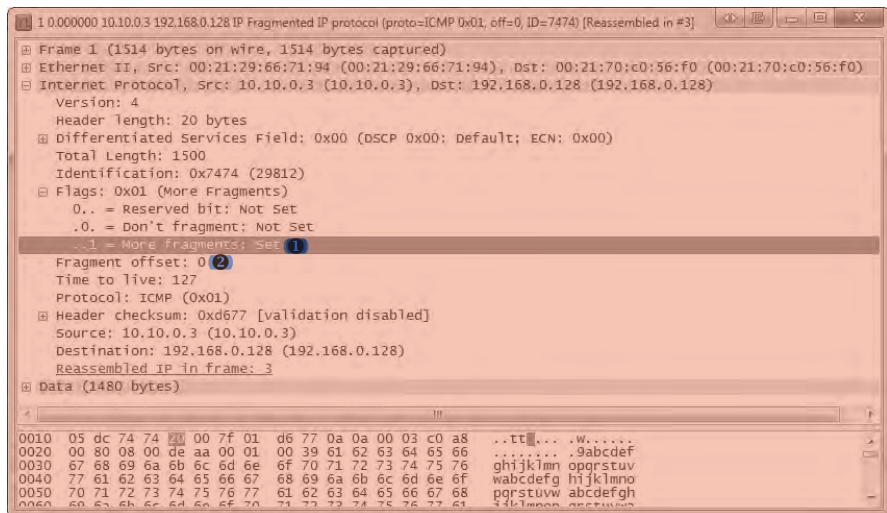


Figure 6-13: More fragments and fragment offset values can indicate a fragmented packet.

The IP header of the second packet (see Figure 6-14) also has the More Fragments flag set ❶, but in this case, the fragment offset value is 1480 ❷. This is indicative of the 1,500-byte MTU, minus 20 bytes for the IP header.

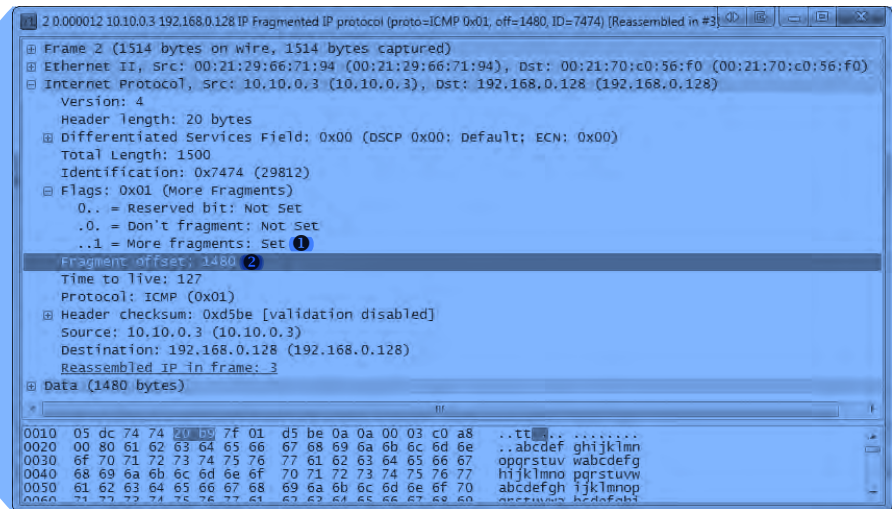


Figure 6-14: The Fragment Offset value increases based on the size of the packets.

The third packet (see Figure 6-15) does not have the More Fragments flag set ❶, which marks it as the last fragment in the data stream, and the Fragment Offset is set to 2960 ❷, the result of 1480 + (1500 – 20). These fragments can all be identified as part of the same series of data because they have the same values in the Identification field of the IP header ❸.

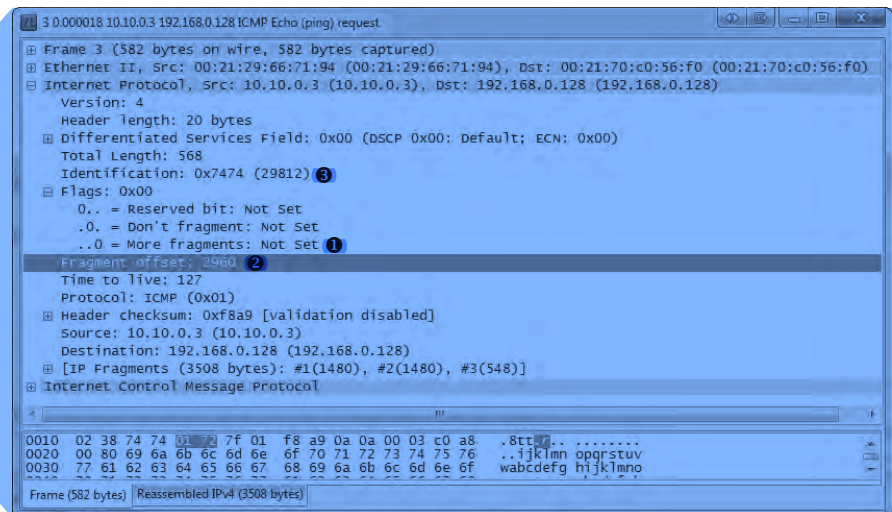


Figure 6-15: More Fragments is not set, indicating the last fragment.

Transmission Control Protocol

The ultimate goal of the *Transmission Control Protocol (TCP)* is to provide end-to-end reliability for the delivery of data. TCP, which is defined in RFC 793, operates at layer 4 of the OSI model. It handles data sequencing and error recovery, and ultimately ensures that data gets where it is supposed to go. A lot of commonly used application-layer protocols rely on TCP and IP to deliver packets to their final destination.

The TCP Header

TCP provides a great deal of functionality, as reflected in the complexity of its header. As shown in Figure 6-16, the following are the TCP header fields:

Source Port The port used to transmit the packet.

Destination Port The port to which the packet will be transmitted.

Sequence Number The number used to identify a TCP segment. This field is used to ensure that parts of a data stream are not missing.

Acknowledgment Number The sequence number that is to be expected in the next packet from the other device taking part in the communication.

Flags The URG, ACK, PSH, RST, SYN, and FIN flags for identifying the type of TCP packet being transmitted.

Window Size The size of the TCP receiver buffer in bytes.

Checksum Used to ensure the contents of the TCP header and data are intact upon arrival.

Urgent Pointer If the URG flag is set, this field is examined for additional instructions for where the CPU should begin reading the data within the packet.

Options Various optional fields that can be specified in a TCP packet.

Transmission Control Protocol				
Bit Offset	0-3	4-7	8-15	16-31
0	Source Port			Destination Port
32	Sequence Number			
64	Acknowledgment Number			
96	Data Offset	Reserved	Flags	Window Size
128	Checksum			Urgent Pointer
160	Options			

Figure 6-16: The TCP header

TCP Ports

`tcp_ports.pcap`

All TCP communication takes place using source and destination *ports*, which can be found in every TCP header. A port is like the jack on an old telephone switchboard. A switchboard operator would monitor a board of lights and plugs. When a light lit up, he would connect with the caller, ask who she wanted to talk to, and then connect her to her destination by plugging in a cable. Every call needed to have a source port (the caller) and a destination port (the recipient). TCP ports work in much the same fashion.

In order to transmit data to a particular application on a remote server or device, a TCP packet must know the port the remote service is listening on. If you try to access an application on a port other than the one configured for use, the communication will fail.

The source port in this sequence is not incredibly important and can be selected randomly. The remote server will simply determine the port to communicate with from the original packet it is sent (see Figure 6-17).

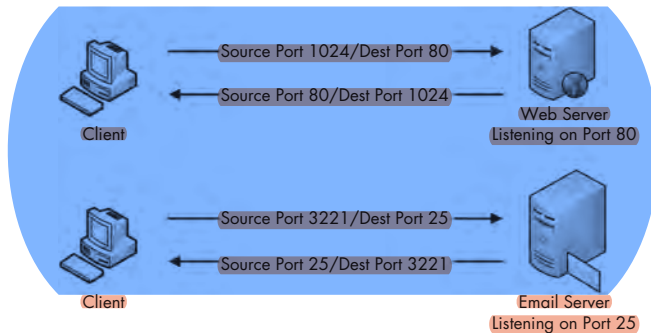


Figure 6-17: TCP uses ports to transmit data.

There are 65,535 ports available for use when communicating with TCP. We typically divide these into two groups:

- The *standard port group* is from 1 through 1023 (ignoring 0 because it is reserved). Particular services use standard ports, which generally lie within the standard port grouping.
- The *ephemeral port group* is from 1024 through 65535 (although some operating systems have different definitions for this). Only one service can communicate on a port at any given time, so modern operating systems select source ports randomly in an effort to make communications unique. These source ports are typically located in the ephemeral range.

Let's examine a couple of different TCP packets and identify the port numbers they are using by opening the file `tcp_ports.pcap`. In this file, we have the HTTP communication of a client browsing to two websites. As mentioned previously, HTTP uses TCP for communication, which makes it a great example of standard TCP traffic.

In the first packet in this file (see Figure 6-18), the first two values represent the packet's source port and destination port. This packet is being sent from 172.16.16.128 to 212.58.226.142. The source port is 2826, an ephemeral

port. (Remember that source ports are chosen at random by the operating system, although they can increment from that random selection.) The destination port is a standard port, port 80 ❷, the standard port used for web servers using HTTP.

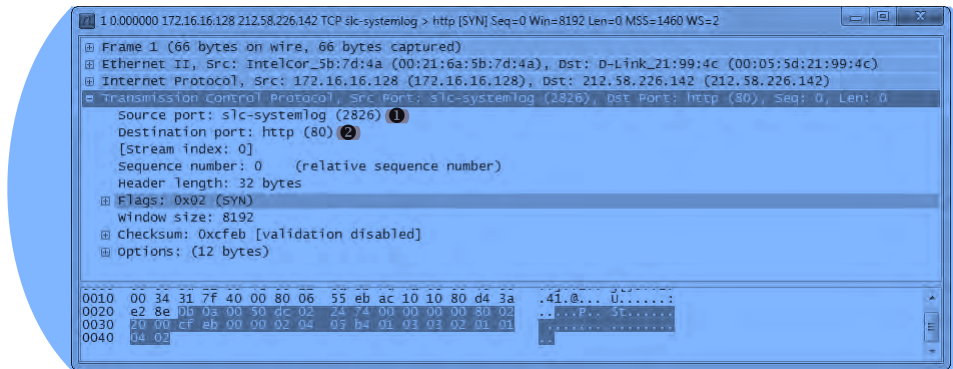


Figure 6-18: The source and destination ports can be found in the TCP header.

Notice that Wireshark labels these ports as slc-systemlog (2826) and http (80). Wireshark maintains a list of ports and their most common uses. Although these are primarily standard ports, many ephemeral ports have commonly used services associated with them. The labeling of these ports can be quite confusing, so it's typically best to disable it by turning off transport name resolution. To do so, choose **Edit ▶ Preferences ▶ Name Resolution**, and then remove the check mark next to Enable Transport Name Resolution. If you wish to leave this functionality enabled but want to change how Wireshark identifies a certain port, you can do so by modifying the *Services* file located in the Wireshark program directory, which is based on the Internet Assigned Numbers Authority (IANA) common ports listing.

The second packet is being sent back from 212.58.226.142 to 172.16.16.128 (see Figure 6-19). As with the IP addresses, the source and destination ports are now also switched ❶.

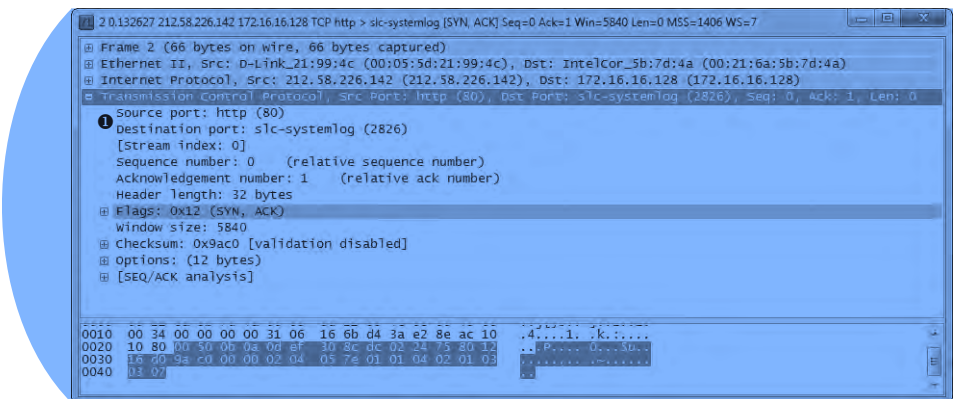


Figure 6-19: The source and destination port numbers are switched for reverse communication.

All TCP-based communication works the same way: a random source port is chosen to communicate to a known destination port. Once this initial packet is sent, the remote device communicates with the source device using the established ports.

There is one more communication stream included in this sample capture file. See if you can locate the port numbers it uses for communication.

NOTE As we progress through this book, you will learn more about the ports associated with common protocols and services. Eventually, you will be able to profile services and devices by the ports they use. For a thorough list of common ports, see <http://www.iana.org/assignments/port-numbers/>.

The TCP Three-Way Handshake

tcp_handshake
:pcap

All TCP-based communication must begin with a *handshake* between two hosts. This handshake process serves a few different purposes:

- It allows the transmitting host to ensure that the destination host is up and able to communicate.
- It lets the transmitting host check that it is listening on the port on which the source is attempting to communicate.
- It allows the transmitting host to send its starting sequence number to the recipient so that both hosts can keep the stream of packets in proper sequence.

The TCP handshake occurs in three separate steps, as shown in Figure 6-20. In the first step, the device that wants to communicate (host A) sends a TCP packet to its target (host B). This initial packet contains no data other than the lower-layer protocol headers. The TCP header in this packet has the SYN flag set and includes the initial sequence number and maximum segment size (MSS) that will be used for the communication process. Host B responds to this packet by sending a similar packet with the SYN and ACK flags set, along with its initial sequence number. Finally, host A sends one last packet to host B with only the ACK flag set. Once this process is completed, both devices should have all of the information they need to begin communicating properly.

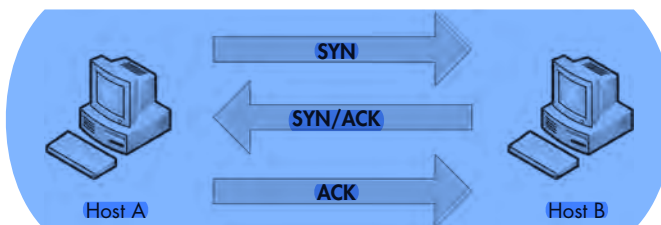


Figure 6-20: The TCP three-way handshake

NOTE TCP packets are often referred to by the flags they have set. For example, rather than refer to a packet as a TCP packet with the SYN flag set, we call that packet a SYN packet. As such, the packets used in the TCP handshake process are referred to as SYN, SYN/ACK, and ACK.

To see this process in action, open *tcp_handshake.pcap*. Wireshark includes a feature that replaces the sequence numbers of TCP packets with relative numbers for easier analysis. For our purposes, we'll disable this feature in order to see the actual sequence numbers. To disable it, choose **Edit ▸ Preferences**, expand the **Protocols** heading, and choose **TCP**. In the window, uncheck the box next to **Relative Sequence Numbers and Window Scaling**, and then click **OK**.

The first packet in this capture represents our initial SYN packet (see Figure 6-21). The packet is transmitted from 172.16.16.128 on port 2826 to 212.58.226.142 on port 80. We can see here that the sequence number transmitted is 3691127924 ❶.



Figure 6-21: The initial SYN packet

The second packet in the handshake is the SYN/ACK response from 212.58.226.142 (see Figure 6-22). This packet also contains this host's initial sequence number (233779340) ❶ and an acknowledgment number (3691127925) ❷. The acknowledgment number shown here is one more than the sequence number included in the previous packet, because this field is used to specify the next sequence number the host expects to receive.


```

2 0132627 212.58.226.142 172.16.16.128 TCP http -> sllc-systemlog [SYN, ACK] Seq=233779340 Ack=3691127925 Win=5040 Len=0 MSS=1406 WS=7
+ Frame 2 (66 bytes on wire, 66 bytes captured)
+ Ethernet II, Src: D-Link_21:99:4c (00:03:5d:21:99:4c), Dst: IntelCor_sb7d4a (00:21:6a:3b:7d:4a)
+ Internet Protocol, Src: 212.58.226.142 (212.58.226.142), Dst: 172.16.16.128 (172.16.16.128)
+ Transmission Control Protocol, Src Port: 80 (80), Dst Port: sllc-systemlog (2326), Seq: 233779340, Ack: 3691127925, Len: 0
  Source port: http (80)
  Destination port: sllc-systemlog (2326)
  [Stream index: 0]
  Sequence number: 233779340 ❶
  Acknowledgement number: 3691127925 ❷
  Header length: 32 bytes
  Flags: 0x12 (SYN, ACK)
    0... .. = Congestion window reduced (CWR): Not set
    .0.. .... = ECN-Echo: Not set
    ..0. .... = Urgent: Not set
    ...1... = Acknowledgment: Set
    ....0... = Push: Not set
    ....0... = Reset: Not set
    10.....1 = SYN: Set
    .....0 = FIN: Not set
  Window size: 5040
  Checksum: 0x240 (validation disabled)
  Options: (12 bytes)
    Maximum segment size: 1406 bytes
    NOP
    NOP
    SACK permitted
    NOP
    window scale: 7 (multiply by 128)
  [Seq/Ack analysis]
0010 00 34 00 00 00 00 31 06 16 0b d4 7a e2 8e ac 10 4...1..K2...
0020 10 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0030 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...

```

Figure 6-22: The SYN/ACK response

The final packet is the **ACK** packet sent from 172.16.16.128 (see Figure 6-23). This packet, as expected, contains the sequence number 3691127925 ❶ as defined in the previous packet's Acknowledgment Number field.

```

1 0132768 172.16.16.128 212.58.226.142 TCP sllc-systemlog -> http [ACK] Seq=3691127925 Ack=233779341 Win=4218 Len=0
+ Frame 3 (54 bytes on wire, 54 bytes captured)
+ Ethernet II, Src: IntelCor_sb7d4a (00:21:6a:3b:7d:4a), Dst: D-Link_21:99:4c (00:03:5d:21:99:4c)
+ Internet Protocol, Src: 172.16.16.128 (172.16.16.128), Dst: 212.58.226.142 (212.58.226.142)
+ Transmission Control Protocol, Src Port: sllc-systemlog (2326), Dst Port: http (80), Seq: 3691127925, Ack: 233779341, Len: 0
  Source port: sllc-systemlog (2326)
  Destination port: http (80)
  [Stream index: 0]
  Sequence number: 3691127925 ❶
  Acknowledgement number: 233779341
  Header length: 20 bytes
  Flags: 0x10 (ACK)
    0... .. = Congestion window reduced (CWR): Not set
    .0.. .... = ECN-Echo: Not set
    ..0. .... = Urgent: Not set
    ...1... = Acknowledgment: Set
    ....0... = Push: Not set
    ....0... = Reset: Not set
    ....0... = SYN: Not set
    ....0... = FIN: Not set
  Window size: 4218
  Checksum: 0x2b2 (validation disabled)
  [Seq/Ack analysis]
0000 00 01 5d 21 99 4c 00 21 6a 3b 7d 4a 08 00 45 00 ...1...1...K...
0010 00 28 31 80 40 00 80 08 55 f6 ac 10 10 80 d4 5a ...10...0.....
0020 e2 8e 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...

```

Figure 6-23: The final ACK

A handshake occurs before every TCP communication sequence. When sorting through a busy capture file in search of the beginning of a communication sequence, the sequence of SYN-SYN/ACK-ACK is a great marker.

TCP Teardown

tcp_teardown
.pcap

Most greetings eventually have a good-bye, and in the case of TCP, every handshake has a teardown. The *TCP teardown* is used to gracefully end a connection between two devices after they have finished communicating. This process involves four packets, and it utilizes the **FIN** flag to signify the end of a connection.

In a teardown sequence, host A tells host B that it is finished communicating by sending a TCP packet with the FIN and ACK flags set. Host B responds with an ACK packet, and transmits its own FIN/ACK packet. Host A responds with an ACK packet, ending the communication process. This process is illustrated in Figure 6-24.

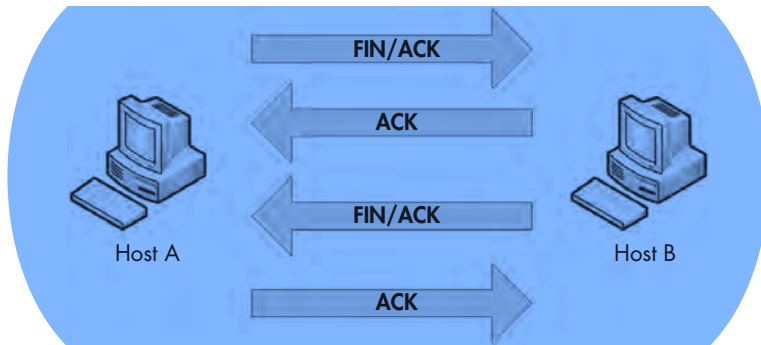


Figure 6-24: The TCP teardown process

To view this process in Wireshark, open the file `tcp_teardown.pcap`. Beginning with the first packet in the sequence, (see Figure 6-25), you can see that the device at 67.228.110.120 initiates the teardown sequence by sending a packet with the FIN and ACK flags set ❶.

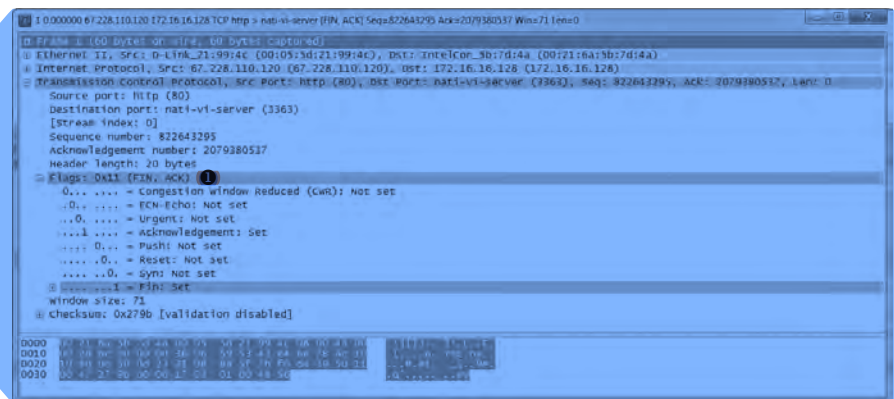


Figure 6-25: The FIN/ACK initiates the teardown process.

Once this packet is sent, 172.16.16.128 responds with an ACK packet to acknowledge receipt of the first packet, and it sends a FIN/ACK packet. The process is complete when 67.228.110.120 sends a final ACK. At this point, the communication between the two devices ends, and they must complete a new TCP handshake in order to begin communicating again.

TCP Resets

tcp_
refuseconnection
.pcap

In an ideal world, every connection would end gracefully with a TCP tear-down. In reality, connections often end abruptly. For example, this may occur due to a potential attacker performing a port scan or simply a misconfigured host. In these cases, a TCP packet with the RST flag set is used. The RST flag is used to indicate a connection was closed abruptly or to refuse a connection attempt.

The file `tcp_refuseconnection.pcap` displays an example of network traffic that includes a RST packet. The first packet in this file is from the host at 192.168.100.138, which is attempting to communicate with 192.168.100.1 on port 80. What this host doesn't know is that 192.168.100.1 isn't listening on port 80 because it is a Cisco router, with no web interface configured, meaning that no service is listening for connections on port 80. In response to this attempted communication, 192.168.100.1 sends a packet to 192.168.100.138, telling it that communication won't be possible over port 80. Figure 6-26 shows the abrupt end to this attempted communication in the TCP header of the second packet. The RST packet contains nothing other than RST and ACK flags, and no further communication follows.

An RST packet ends communication whether it arrives at the beginning of an attempted communication sequence, as in this example, or is sent in the middle of the communication between hosts.

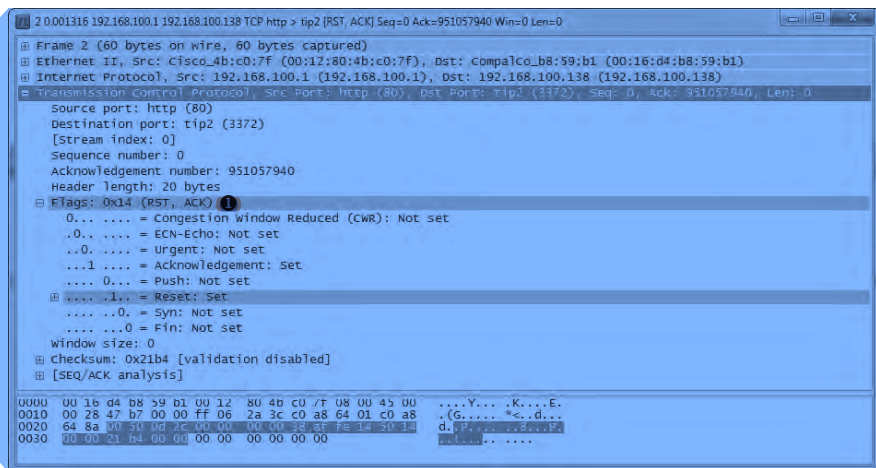


Figure 6-26: The RST and ACK flags signify the end of communication.

User Datagram Protocol

The *User Datagram Protocol* (UDP) is the other layer 4 protocol commonly used on modern networks. While TCP is designed for reliable data delivery with built-in error checking, UDP aims to provide speedy transmission. For this reason, UDP is a best-effort service, commonly referred to as a *connectionless*

protocol. A connectionless protocol does not formally establish and terminate a connection between hosts, unlike TCP with its handshake and teardown processes.

With a connectionless protocol, which doesn't provide reliable services, it would seem that UDP traffic would be flaky at best. That would be true, except that the protocols that rely on UDP typically have their own built-in reliability services, or use certain features of ICMP to make the connection somewhat more reliable. For example, the application-layer protocols DNS and DHCP, which are highly dependent on the speed of packet transmission across a network, use UDP as their transport layer protocol, but they handle error checking and retransmission timers themselves.

The UDP Header

udp_dnsrequest.pcap

The UDP header is much smaller and simpler than the TCP header. As shown in Figure 6-27, the following are the UDP header fields:

- Source Port** The port used to transmit the packet
- Destination Port** The port to which the packet will be transmitted
- Packet Length** The length of the packet in bytes
- Checksum** Used to ensure that the contents of the UDP header and data are intact upon arrival

User Datagram Protocol		
Bit Offset	0–15	16–31
0	Source Port	Destination Port
32	Packet Length	Checksum

Figure 6-27: The UDP header

The file `udp_dnsrequest.pcap` contains one packet. This packet represents a DNS request, which uses UDP. When you expand the packet's UDP header, you'll see four fields (see Figure 6-28).

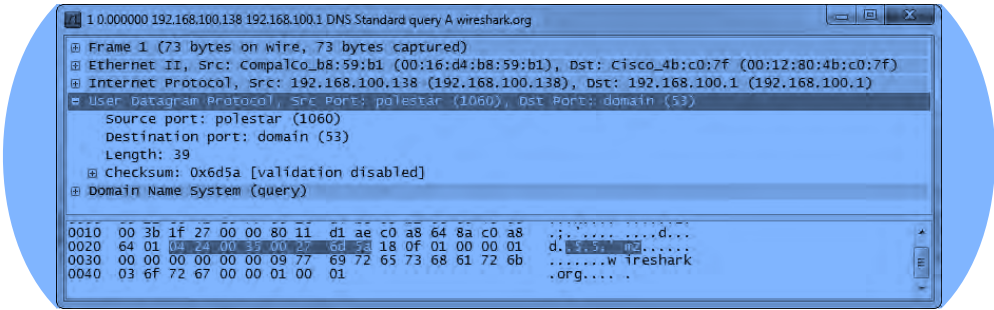


Figure 6-28: The contents of a UDP packet are very simple.

The key point to remember is that UDP does not care about reliable delivery. Therefore, any application that uses UDP must take special steps to ensure reliable delivery, if it is necessary.

Internet Control Message Protocol

Internet Control Message Protocol (ICMP) is the utility protocol of TCP/IP, responsible for providing information regarding the availability of devices, services, or routes on a TCP/IP network. Most network troubleshooting techniques and tools center around common ICMP message types. ICMP is defined in RFC 792.

The ICMP Header

ICMP is part of IP, and it relies on IP to transmit its messages. ICMP contains a relatively small header that changes depending on its purpose. As shown in Figure 6-29, the ICMP header contains the following fields:

Type The type or classification of the ICMP message, based on the RFC specification

Code The subclassification of the ICMP message, based on the RFC specification

Checksum Used to ensure that the contents of the ICMP header and data are intact upon arrival

Variable A portion that depends on the Type and Code fields

Internet Control Message Protocol			
Bit Offset	0-15		16-31
0	Type	Code	Checksum
32	Variable		

Figure 6-29: The ICMP header

ICMP Types and Messages

As noted, the structure of an ICMP packet depends on its purpose, as defined by the values in the *Type* and *Code* fields.

You might consider the ICMP Type field as the packet's classification and the Code field as its subclass. For example, a Type field value of 3 indicates "Destination Unreachable." While this information alone might not be enough to troubleshoot a problem, if that packet were to also specify a Code field value of 3, indicating "Port Unreachable," you could conclude that there is an issue with the port with which you are attempting to communicate.

NOTE For a full list of available ICMP types and codes, see <http://www.iana.org/assignments/icmp-parameters>.

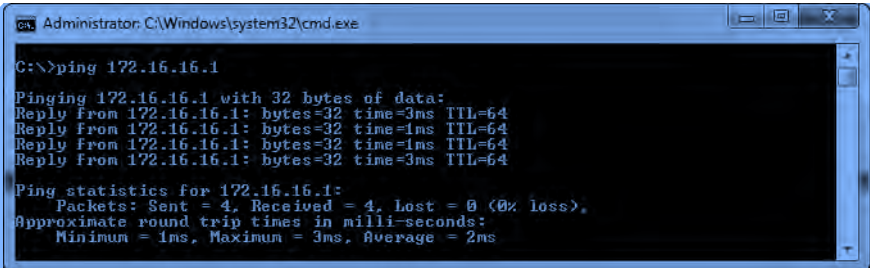
Echo Requests and Responses

icmp_echo.pcap

ICMP's biggest claim to fame is thanks to the ping utility. Ping is used to test for connectivity to a device. Most information technology (IT) professionals are familiar with ping.

To use ping, enter `ping <ip address>` at the command prompt, replacing `<ip address>` with the actual IP address of a device on your network. If the target device is turned on, your computer has a communication route to it, and there is no firewall blocking that communication, you should see replies to your ping command.

The example in Figure 6-30 shows four successful replies that display their size, RTT, and TTL used. The Windows utility also provides a summary detailing how many packets were sent, received, and lost. If communication fails, you should see a message telling you why.



```
Administrator: C:\Windows\system32\cmd.exe

C:\>ping 172.16.16.1

Pinging 172.16.16.1 with 32 bytes of data:
Reply from 172.16.16.1: bytes=32 time=3ms TTL=64
Reply from 172.16.16.1: bytes=32 time=1ms TTL=64
Reply from 172.16.16.1: bytes=32 time=1ms TTL=64
Reply from 172.16.16.1: bytes=32 time=3ms TTL=64

Ping statistics for 172.16.16.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 3ms, Average = 2ms
```

Figure 6-30: The ping command being used to test connectivity

Basically, the ping command sends one packet at a time to a device and listens for a reply to determine if there is connectivity to that device, as shown in Figure 6-31.

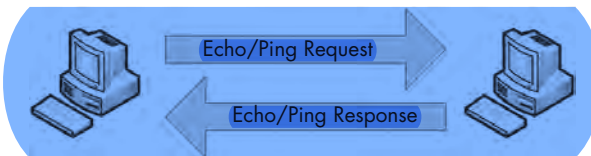


Figure 6-31: The ping command involves only two steps.

NOTE Although ping has long been the bread and butter of IT, its results can be a bit deceiving as host-based firewalls are deployed. Many of today's firewalls limit the ability of a device to respond to ICMP packets. This is great for security, because potential attackers using ping to determine if a host is accessible might be deterred, but troubleshooting is also made more difficult—it can be frustrating to ping a device to test for connectivity and not receive a reply when you know you can communicate with that device.

The ping utility in action is a great example of simple ICMP communication. The packets in the file *icmp_echo.pcap* demonstrate what happens when you run ping.

The first packet (see Figure 6-32) shows that host 192.168.100.138 is sending a packet to 192.168.100.1 ❶. When you expand the ICMP portion of this packet, you can determine the ICMP packet type by looking at the Type and Code fields. In this case, the packet is type 8 ❷, code 0 ❸, indicating an echo request. (Wireshark should tell you what the type/code being displayed actually is.) This echo (ping) request is the first half of the equation. It is a simple ICMP packet, sent using IP, that contains a small amount of data. Along with the type and code designations and the checksum, we also have a sequence number that is used to pair requests with replies, and a random text string in the variable portion of the ICMP packet.

NOTE The terms echo and ping are often used interchangeably, but just remember that ping is actually the name of a tool. The ping tool is used to send ICMP echo request packets.

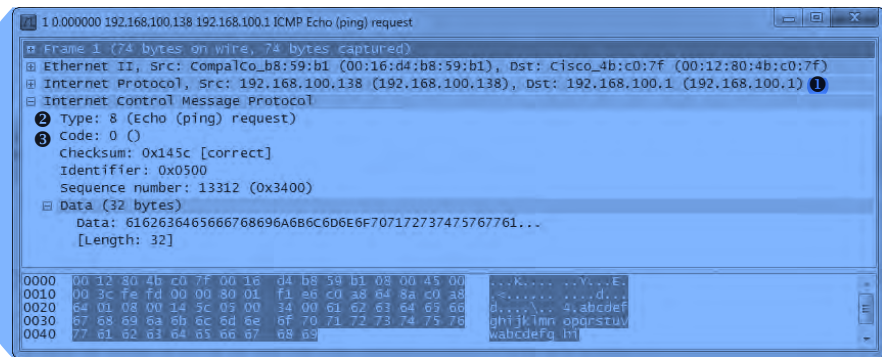


Figure 6-32: The ICMP echo request packet

The second packet in this sequence is the reply to our request (see Figure 6-33). The ICMP portion of the packet is type 0 ❶, code 0 ❷, indicating that this is an echo reply. Because the sequence number in the second packet matches that of the first ❸, we know that this echo reply matches the echo request in the previous packet. This reply packet also contains the same 32-byte string of data that was transmitted with the initial request ❹. Once this second packet has been received by 192.168.100.138, ping will report success (see Figure 6-30, shown earlier).

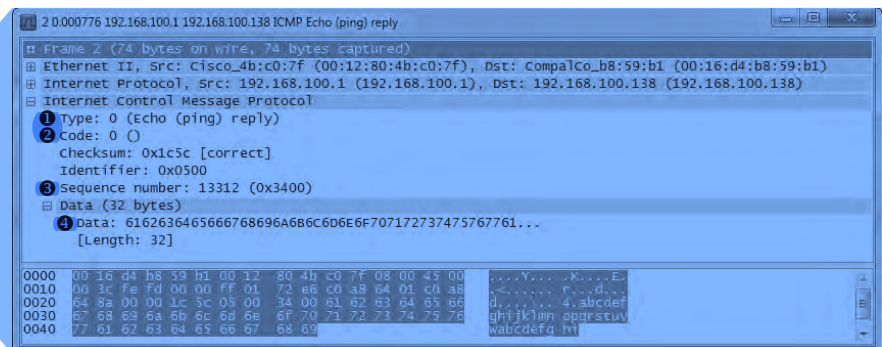


Figure 6-33: The ICMP echo reply packet

Note that you can use variations of ping to increase the size of the data padding, which forces packets to be fragmented for various types of network troubleshooting. This may be required when you're troubleshooting networks that require a smaller fragment size.

NOTE The random text used in an ICMP echo request can be of great interest to a potential attacker. Attackers can use the information in this padding to profile the operating system used on a device. Additionally, attackers can place small bits of data in this field as a method of covert communication.

Traceroute

icmp_traceroute.pcap

The traceroute utility is used to identify the path from one device to another. On a simple network, a path may go through only a single router or no router at all. On a complex network, however, a packet may need to go through dozens of routers to reach its final destination, which is why it's crucial to be able to trace the exact path a packet takes from one destination to another in order to troubleshoot communication.

By using ICMP (with a little help from IP), traceroute can map the path packets take. For example, the first packet in the file *icmp_traceroute.pcap* is pretty similar to the echo request we looked at in the previous section (see Figure 6-34).

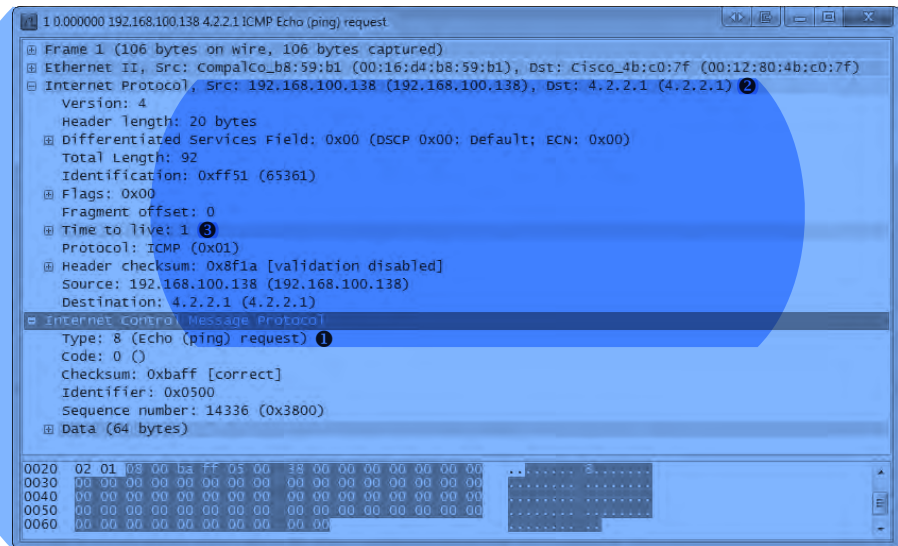


Figure 6-34: An ICMP echo request packet with a TTL value of 1

At first glance, this packet appears to be a simple echo request ❶ from 192.168.100.138 to 4.2.2.1 ❷, and everything in the ICMP portion of the packet is identical to the formatting of an echo request packet. However, when you expand the IP header of this packet, you'll notice one odd value: The packet's TTL value is set to 1 ❸, which means that the packet will be dropped at the first router that it hits. Because the destination 4.2.2.1

address is an Internet address, we know that there must be at least one router between our source and destination devices, so there is no way this packet will reach its destination. That's good for us, because traceroute relies on the fact that this packet will make it to only the first router it traverses.

The second packet is, as expected, a reply from the first router we reached along the path to our destination (see Figure 6-35). This packet reached this device at 192.168.100.1, its TTL was decremented to 0, and the packet could not be transmitted further, so the router replied with an ICMP response. This packet's type 11 ❶, code 0 ❷ tells us that the destination was unreachable because the packet's TTL was exceeded during transit.

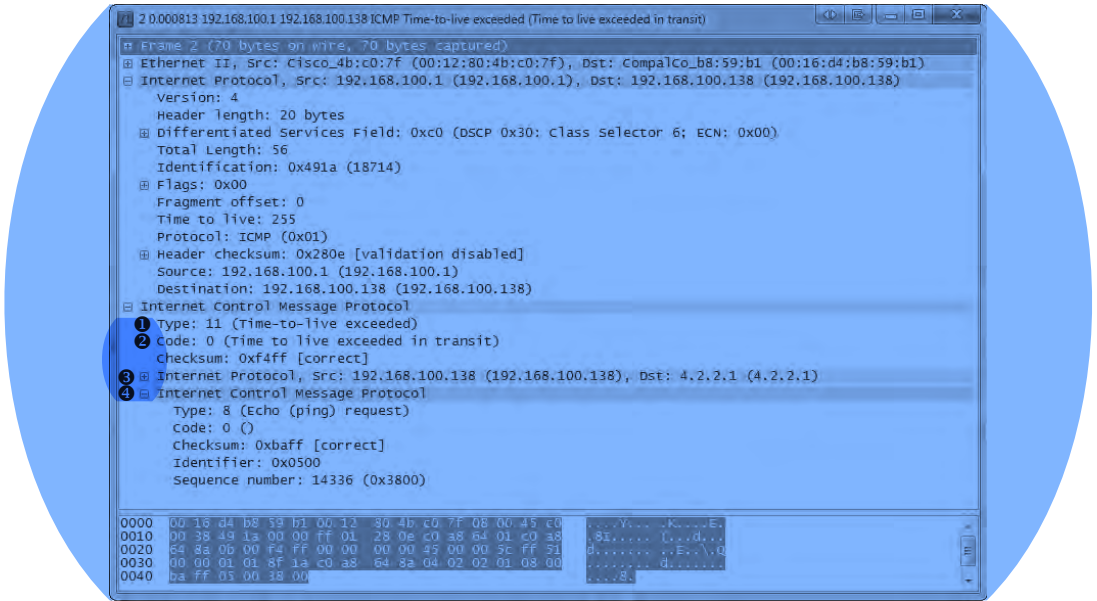


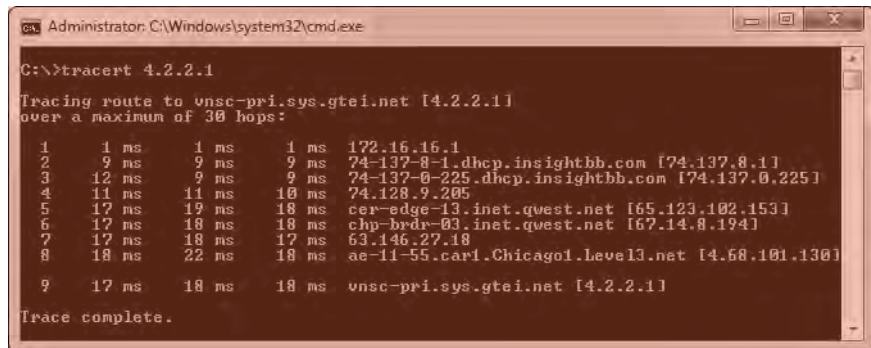
Figure 6-35: An ICMP response from the first router along the path

This ICMP packet is sometimes called a *double-headed packet*, because the tail end of its ICMP portion contains a copy of the IP header ❸ and ICMP data ❹ that was sent in the original echo request. This information can prove to be very useful for troubleshooting.

This process of sending packets with incremented TTL values occurs two more times before we get to packet 7. Here, you see the same thing you saw in the first packet, except that this time, the TTL value in the IP header is set to 2, which ensures the packet will make it to the second hop router before it is dropped. As expected, we receive a reply from the next hop router, 12.180.241.1, with the same ICMP destination unreachable and TTL exceeded messages. This process continues with the TTL value increasing by one until the destination 4.2.2.1 is reached.

To sum up, this traceroute process has communicated with each router along the path, building a map of the route to the destination. This map is shown in Figure 6-36.

NOTE *The discussion here on traceroute is generally Windows-focused because it uses ICMP exclusively. The traceroute utility on Linux is a bit more versatile and can utilize other protocols in order to perform route path tracing.*



```
Administrator: C:\Windows\system32\cmd.exe

C:\>tracert 4.2.2.1

Tracing route to vncs-pri.sys.gtei.net [4.2.2.1]
over a maximum of 30 hops:

  0  0 ms  0 ms  0 ms  172.16.16.1
  1  9 ms  9 ms  9 ms  74-137-8-1.dhcp.insightbb.com [74.137.8.1]
  2 12 ms  9 ms  9 ms  74-137-0-225.dhcp.insightbb.com [74.137.0.225]
  3 11 ms 11 ms 10 ms 74.128.9.205
  4 17 ms 19 ms 18 ms cer-edge-13.inet.qwest.net [65.123.102.153]
  5 17 ms 18 ms 18 ms chp-brdr-03.inet.qwest.net [67.14.8.194]
  6 17 ms 18 ms 17 ms 63.146.27.18
  7 18 ms 22 ms 18 ms ae-11-55.car1.Chicago1.Level3.net [4.68.101.130]
  8 17 ms 18 ms 18 ms vncs-pri.sys.gtei.net [4.2.2.1]

Trace complete.
```

Figure 6-36: A sample output from the traceroute utility

As you'll see throughout this book, ICMP has many different functions. We'll use ICMP frequently as we analyze more scenarios.

This chapter has introduced you to a few of the most important protocols you will examine in the process of packet analysis. IP, TCP, UDP, and ICMP are at the foundation of all network communications, and they are critical to just about every daily task you perform. In the next chapter, we will look at a grouping of common application-layer protocols.

7

COMMON UPPER-LAYER PROTOCOLS



In this chapter, we'll continue to examine the functions of individual protocols, as well as what they look like when viewed with Wireshark. We'll discuss three of the most common upper-layer (layer 7) protocols: DHCP, DNS, and HTTP.

Dynamic Host Configuration Protocol

In the early days of networking, when a device wanted to communicate over a network, it needed to be assigned an address by hand. As networks grew, this manual process quickly became cumbersome. To solve this problem, Bootstrap Protocol (BOOTP) was created to automatically assign addresses to network-connected devices. BOOTP was later replaced with the more sophisticated Dynamic Host Configuration Protocol (DHCP).

DHCP is an application layer protocol responsible for allowing a device to automatically obtain an IP address (and addresses of other important network assets, such as DNS servers and routers). Most DHCP servers today also provide other parameters to clients, such as the addresses of the default gateway and DNS servers in use on the network.

The DHCP Packet Structure

DHCP packets can carry quite a lot of information to a client. As shown in Figure 7-1, the following fields are present within a DHCP packet:

OpCode Indicates whether the packet is a DHCP request or a DHCP reply

Hardware Type The type of hardware address (10MB Ethernet, IEEE 802, ATM, and so on)

Hardware Length The length of the hardware address

Hops Used by relay agents to assist in finding a DHCP server

Transaction ID A random number used to pair requests with responses

Seconds Elapsed Seconds since the client first requested an address from the DHCP server

Flags The types of traffic the DHCP client can accept (unicast, broadcast, and so on)

Client IP Address The client's IP address (derived from the Your IP Address field)

Your IP Address The IP address offered by the DHCP server (ultimately becomes the Client IP Address field value)

Server IP Address The DHCP server's IP address

Gateway IP Address The IP address of the network's default gateway

Client Hardware Address The client's MAC address

Server Host Name The server's host name (optional)

Boot File A boot file for use by DHCP (optional)

Options Used to expand the structure of the DHCP packet to give it more features

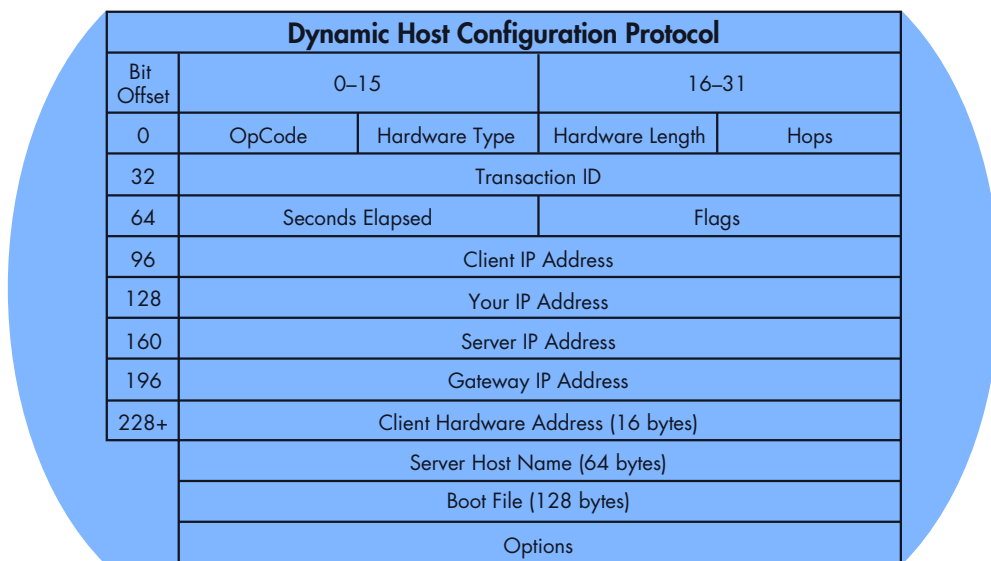


Figure 7-1: The DHCP packet structure

The DHCP Renewal Process

`dhcp_nolease_renewal.pcap`

The primary goal of DHCP is to assign addresses to clients during the renewal process. The renewal process takes place between a single client and a DHCP server, as shown in the file `dhcp_nolease_renewal.pcap`. The DHCP renewal process is often referred to as the DORA process because it uses four types of DHCP packets: discover, offer, request, and acknowledgment, as shown in Figure 7-2. Here, we'll take a look at each type of DORA packet.

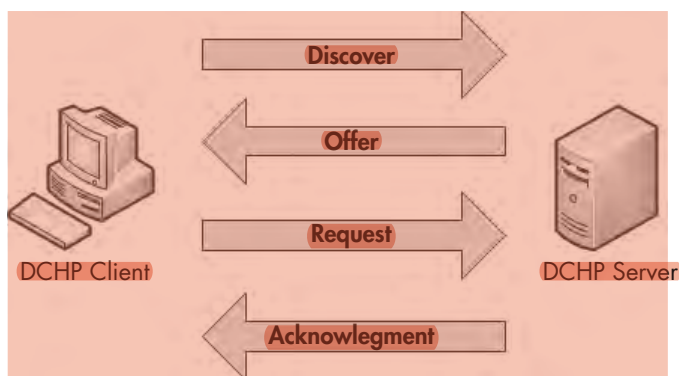


Figure 7-2: The DHCP DORA process

The Discover Packet

As you can see in the referenced capture file, the first packet is sent from 0.0.0.0 on port 68 to 255.255.255.255 on port 67. The client uses 0.0.0.0 because it does not yet have an IP address. The packet is sent to 255.255.255.255 because this is the network-independent broadcast address, thus ensuring that this packet will be sent out to every device on the network. Because the device doesn't know the address of a DHCP server, this first packet is sent in an attempt to find a DHCP server that will listen.

Examining the Packet Details pane, the first thing we notice is that DHCP relies on UDP as its transport layer protocol. DHCP is very concerned with the speed at which a client receives the information it's requesting. DHCP has its own built-in reliability measures, which means UDP is a perfect fit. You can see the details of the discovery process by examining the first packet's DHCP portion in the Packet Details pane, as shown Figure 7-3.

NOTE Because Wireshark still references BOOTP when dealing with DHCP, you'll see a Bootstrap Protocol section in the Packet Detail pane, rather than a DHCP section. Nevertheless, I'll refer to this as the packet's DHCP portion throughout this book.

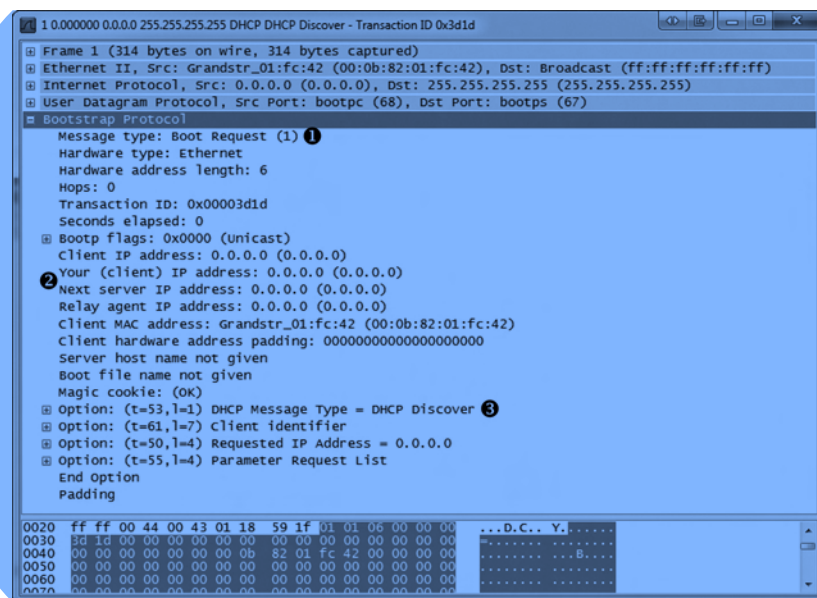


Figure 7-3: The DHCP discover packet

This packet is a request, identified by the (1) in the Message Type field ❶. Most of the fields in this discovery packet are either blank (as you can see in the IP Address fields ❷) or pretty self-explanatory, based on the listing of DHCP fields in the previous section. The meat of this packet is in its four option fields:

DHCP Message Type This is option type 53 (t=53), with length 1 and a value of 1 ❸. These values indicate that this is a DHCP discover packet.

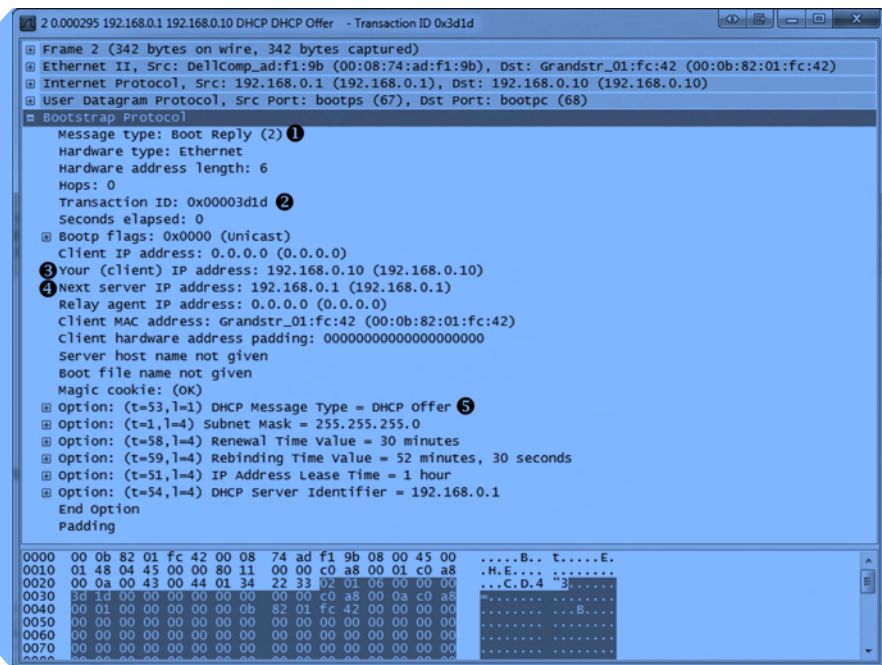
Client Identifier This provides additional information about the client requesting an IP address.

Requested IP Address This supplies the IP address the client would like to receive (typically its previously used IP address).

Parameter Request List This lists the different configuration items (IP addresses of other important network devices) the client would like to receive from the DHCP server.

The Offer Packet

The second packet in this file lists valid IP addresses in its IP header, showing a packet traveling from 192.168.0.1 to 192.168.0.10, as shown in Figure 7-4. The client does not actually have the 192.168.0.10 address yet, so the server will first attempt to communicate with the client using its hardware address, as provided by ARP. If communication is not possible, it will simply broadcast the offer to communicate.



The image shows a Wireshark packet capture window titled "2 0.000295 192.168.0.1 192.168.0.10 DHCP DHCP Offer - Transaction ID 0x3d1d". The packet list on the left shows the following details:

- Frame 2 (342 bytes on wire, 342 bytes captured)
- Ethernet II, Src: DellComp_ad:f1:9b (00:08:74:ad:f1:9b), Dst: Grandstr_01:fc:42 (00:0b:82:01:fc:42)
- Internet Protocol, Src: 192.168.0.1 (192.168.0.1), Dst: 192.168.0.10 (192.168.0.10)
- User Datagram Protocol, Src Port: bootps (67), Dst Port: bootpc (68)

The packet details pane shows the following information:

- Message type: Boot Reply (2) ①
- Hardware type: Ethernet
- Hardware address length: 6
- Hops: 0
- Transaction ID: 0x00003d1d ②
- Seconds elapsed: 0
- Bootp flags: 0x0000 (Unicast)
- Client IP address: 0.0.0.0 (0.0.0.0)
- ③ Your (client) IP address: 192.168.0.10 (192.168.0.10)
- ④ Next server IP address: 192.168.0.1 (192.168.0.1)
- Relay agent IP address: 0.0.0.0 (0.0.0.0)
- Client MAC address: Grandstr_01:fc:42 (00:0b:82:01:fc:42)
- Client hardware address padding: 00000000000000000000
- Server host name not given
- Boot file name not given
- Magic cookie: (OK)
- Option: (t=53,l=1) DHCP Message Type = DHCP offer ⑤
- Option: (t=1,l=4) Subnet Mask = 255.255.255.0
- Option: (t=58,l=4) Renewal Time value = 30 minutes
- Option: (t=59,l=4) Rebinding Time value = 52 minutes, 30 seconds
- Option: (t=51,l=4) IP Address Lease Time = 1 hour
- Option: (t=54,l=4) DHCP Server Identifier = 192.168.0.1
- End option padding

The packet bytes pane shows the raw data in hexadecimal and ASCII format.

Figure 7-4: The DHCP offer packet

The DHCP portion of this second packet, called the *offer packet*, indicates that the message type is a reply ①. This packet contains the same transaction ID as the previous packet ②, which tells us that this reply is indeed to respond to our original request.

The offer packet is sent by the DHCP server in order to offer its services to the client. It does so by supplying information about itself and the addressing it wants to provide the client. In Figure 7-4, the IP address 192.168.0.10 in

the Your (Client) IP Address field is being offered to the client ③. The value 192.168.0.1 in the Next Server IP Address field ④ indicates that our DHCP server and default gateway share the same IP address.

The first option listed identifies the packet as a DHCP Offer ⑤. The options that follow are supplied by the server and indicate the additional information it can offer, along with the client's IP address. You can see that it offers the following:

- A subnet mask of 255.255.255.0
- A renewal time of 30 minutes
- A rebinding time value of 52 minutes and 30 seconds
- An IP address lease time of 1 hour
- A DHCP server identifier of 192.168.0.1

The Request Packet

Once the client receives an offer from the DHCP server, it should accept it with a DHCP request packet, as shown in Figure 7-5.

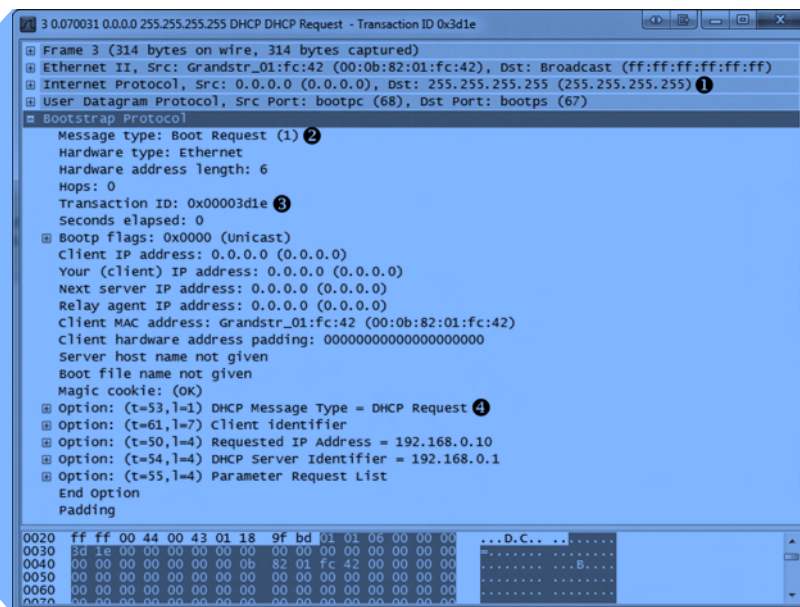


Figure 7-5: The DHCP request packet

The third packet in this capture still comes from IP address 0.0.0.0, because we have not yet completed the process of obtaining an IP address ①. The packet now knows the DHCP server it is communicating with.

The Message Type field shows that this packet is a request ②. Although every packet in this capture file is part of the same renewal process, it has a new transaction ID, since this is a new request/reply transaction ③. This packet is similar to the discover packet, in that all of its IP addressing information is blank.

Finally, in the options fields ④, we see that this is a DHCP Request. Notice that the requested IP address is no longer blank, and that the DHCP Server Identifier field also contains an address.

The Acknowledgment Packet

In the final step in this process, the DHCP server sends the requested IP addresses to the client in an acknowledgment packet and records that information in its database, as shown in Figure 7-6.

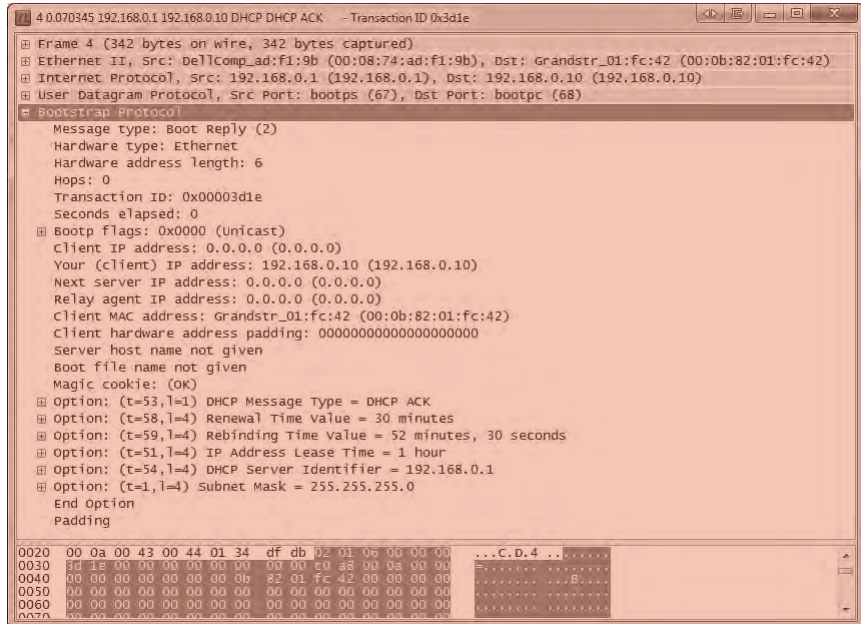


Figure 7-6: The DHCP acknowledgment packet

The client now has an IP address and can use it to begin communicating on the network.

DHCP In-Lease Renewal

`dhcp_inlease_`
`renewal.pcap`

When a DHCP server assigns an IP address to a device, it *leases* it to the client. This means that the client is allowed to use the IP address for only a limited amount of time before it must renew the lease. The DORA process just discussed occurs the first time a client gets an IP address or when its lease time has expired. In either case, the device is considered to be *out of lease*.

When a client with an IP address in-lease reboots, it must perform a truncated version of the DORA process in order to reclaim its IP address. This process is called an *in-lease renewal*.

In the case of a lease renewal, the Discovery and Offer packets are unnecessary. Think of it as the same DORA process used in an out-of-lease renewal, but the in-lease renewal doesn't need to *do* as much, leaving only the request and acknowledgment steps. You'll find a sample capture of an in-lease renewal in the file *dhcp_inlease_renewal.pcap*.

DHCP Options and Message Types

DHCP's real flexibility lies in its available options. As you've seen, the packet's DHCP options can vary in size and content. The packet's overall size depends on the combination of options used. You can view a full list of the many DHCP options at <http://www.iana.org/assignments/bootp-dhcp-parameters/>.

The only option required in all DHCP packets is the Message Type option (option 53). This option identifies how the DHCP client or server will process the information contained within the packet. There are eight message types, as defined in Table 7-1.

Table 7-1: DHCP Message Types

Type Number	Message Type	Description
1	Discover	Used by the client to locate available DHCP servers
2	Offer	Sent by the server to the client in response to a discover packet
3	Request	Sent by the client to request the offered parameters from the server
4	Decline	Sent by the client to the server to indicate invalid parameters within a packet
5	ACK	Sent by the server to the client with the configuration parameters requested
6	NAK	Sent by the client to the server to refuse a request for configuration parameters
7	Release	Sent by the client to the server to cancel a lease by releasing its configuration parameters
8	Inform	Sent by the client to the server to ask for configuration parameters when the client already has an IP address

Domain Name System

The Domain Name System (DNS) is one of the most crucial Internet protocols because it is the proverbial molasses that holds the bread together. DNS ties names, such as *www.google.com*, to IP addresses, such as 74.125.159.99. When we want to communicate with a networked device and we don't know its IP address, we access that device via its DNS name.

DNS servers store a database of *resource records* of IP address-to-DNS name mappings, which they share with clients and other DNS servers.

NOTE Because the architecture of DNS servers is complicated, we will just look at some common types of DNS traffic. You can review the various DNS-related RFCs at <http://www.isc.org/community/reference/RFCs/DNS>.

The DNS Packet Structure

As you can see in Figure 7-7, the DNS packet structure is somewhat different from the packet types we've discussed previously. The following fields can be present within a DNS packet:

DNS ID Number Used to associate DNS queries with DNS responses.

Query/Response (QR) Denotes whether the packet is a DNS query or response.

OpCode Defines the type of query contained in the message.

Authoritative Answers (AA) If this value is set in a response packet, it indicates that the response is from a name server with authority over the domain.

Truncation (TC) Indicates that the response was truncated because it was too large to fit within the packet.

Recursion Desired (RD) When set in a query, this value indicates that the DNS client requests a recursive query if the target name server does not contain the information requested.

Recursion Available (RA) If this value is set in a response, it indicates that the name server supports recursive queries.

Reserved (Z) Defined by RFC 1035 to be set as all zeros; however, sometimes it's used as an extension of the RCode field.

Response Code (RCode) Used in DNS responses to indicate the presence of any errors.

Question Count The number of entries in the Questions section.

Answer Count The number of entries in the Answers section.

Name Server Count The number of name server resource records in the Authority section.

Additional Records Count The number of other resource records in the Additional Information section.

Questions section Variable-sized section that contains one or more queries for information to be sent to the DNS server.

Answers section Variable-sized section that carries one or more resource records that answer queries.

Authority section Variable-sized section that contains resource records that point to authoritative name servers that can be used to continue the resolution process.

Additional Information section Variable-sized section that contains resource records that hold additional information related to the query that is not absolutely necessary to answer the query.

Domain Name System						
Bit Offset	0–15	16–31				
0	DNS ID Number	QR	OpCode	AA	TC	RD
32	Question Count	Answer Count				
64	Name Server Count	Additional Records Count				
96	Questions Section	Answers Section				
128	Authority Section	Additional Information Section				

Figure 7-7: The DNS packet structure

A Simple DNS Query

dns_query_response.pcap

DNS functions in a query/response format. A client wishing to resolve a DNS name to an IP address sends a *query* to a DNS server, and the server sends the requested information in its *response*. In its simplest form, this process takes two packets, as can be seen in the capture file *dns_query_response.pcap*.

The first packet, shown in Figure 7-8, is a DNS query sent from the client 192.168.0.114 to the server 205.152.37.23 on port 53, which is the standard port used by DNS.

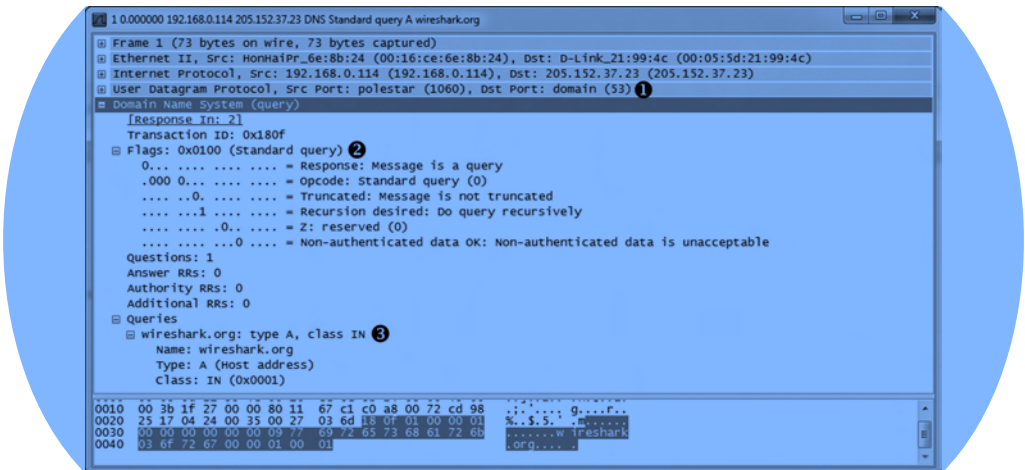


Figure 7-8: The DNS query packet

When you begin examining the headers in this packet, you will see that DNS also relies on UDP ❶.

In the DNS portion of the packet, you can see that smaller fields near the beginning of the packet are condensed by Wireshark into a single Flags section. Expand this section, and you'll see that the message is indeed a standard query ❷, that it is not truncated, and that recursion is desired (we will cover recursion shortly). Only a single question is identified, which can be found by expanding the Queries section. There, you can see the query is for the name *wireshark.org* for a host (type A) Internet (IN) address ❸. This packet is basically asking, "Which IP address is associated with the *wireshark.org* domain?"

The response to this request is in packet 2, as shown in Figure 7-9. Because this packet has an identical identification number ❶, we know that it contains the correct response to the original query.

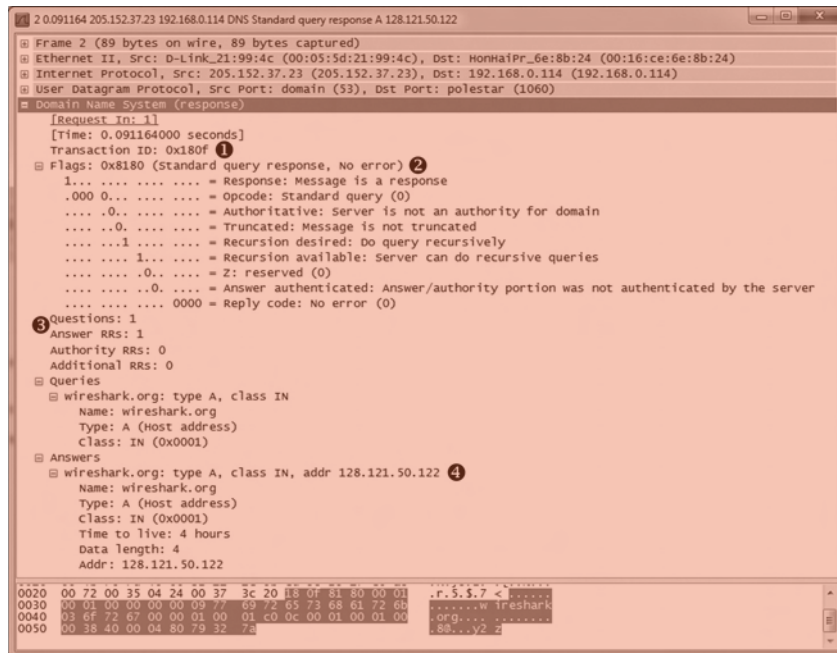


Figure 7-9: The DNS response packet

The Flags section confirms that this is a response and that recursion is available if necessary ❷. This packet contains only one question and one resource record ❸, because it includes the original question in conjunction with its answer. Expanding the Answers section gives us the response to the query: the IP address of *wireshark.org* is 128.121.50.122 ❹. With this information, the client can now construct IP packets and begin communicating with *wireshark.org*.

DNS Question Types

The Type fields used in DNS queries and responses indicate the resource record type that the query or response is for. Some of the more common message/resource record types are listed in Table 7-2. You will be seeing these types throughout normal traffic and this book.

Table 7-2: Common DNS Resource Record Types

Value	Type	Description
1	A	IPv4 host address
2	NS	Authoritative name server
5	CNAME	Canonical name for an alias
15	MX	Mail exchange
16	TXT	Text string
28	AAAA	IPv6 host address
251	IXFR	Incremental zone transfer
252	AXFR	Full zone transfer

The list in Table 7-2 is brief and by no means exhaustive. To review all DNS resource record types, visit <http://www.iana.org/assignments/dns-parameters/>.

DNS Recursion

`dns_`
`recursivequery_`
`client.pcap,`
`dns_`
`recursivequery_`
`server.pcap`

Due to the hierarchical nature of the Internet's DNS structure, DNS servers must be able to communicate with each other in order to answer the queries submitted by clients. While we expect our internal DNS server to know the name-to-IP address mapping of our local intranet server, we can't expect it to know the IP address associated with Google or Dell.

When a DNS server needs to find an IP address, it queries another DNS server on behalf of the client making the request. In effect, the DNS server acts like a client, and this process is called *recursion*.

To view the recursion process from both the DNS client and server perspectives, open the file `dns_recursivequery_client.pcap`. This file contains a capture of a client's DNS traffic file in two packets. The first packet is the initial query sent from the DNS client 172.16.0.8 to its DNS server 172.16.0.102, as shown in Figure 7-10.

When you expand the DNS portion of this packet, you'll see that this is a standard query for an A type record for the DNS name `www.nostarch.com` ❶. To learn more about this packet, expand the Flags section, and you'll see that recursion is desired ❷.

The second packet is what we would expect to see in response to the initial query, as shown in Figure 7-11.

This packet's transaction ID matches that of our query ❶, no errors are listed, and we receive the A type resource record associated with `www.nostarch.com` ❷.

1 0.000000 172.16.0.8 172.16.0.102 DNS Standard query A www.nostarch.com

Frame 1 (76 bytes on wire, 76 bytes captured)

Ethernet II, Src: 00:25:b3:bf:91:ee (00:25:b3:bf:91:ee), Dst: 00:0c:29:92:94:9f (00:0c:29:92:94:9f)

Internet Protocol, Src: 172.16.0.8 (172.16.0.8), Dst: 172.16.0.102 (172.16.0.102)

User Datagram Protocol, Src Port: 56125 (56125), Dst Port: 53 (53)

Domain Name System (query)

[Request In: 2]

Transaction ID: 0x8b34

Flags: 0x0100 (Standard query)

0... .. = Response: Message is a query
 .000 0... .. = Opcode: Standard query (0)
 0... .. = Truncated: Message is not truncated
 1... .. = Recursion desired: Do query recursively 2
 0... .. = Z: reserved (0)
 0... .. = Non-authenticated data OK: Non-authenticated data is unacceptable

Questions: 1
 Answer RRs: 0
 Authority RRs: 0
 Additional RRs: 0

Queries

www.nostarch.com: type A, class IN

1 Name: www.nostarch.com
 Type: A (Host address)
 Class: IN (0x0001)

0010 00 3e 66 47 00 00 80 11 00 00 ac 10 00 08 ac 10 .>fG... ..
 0020 00 66 db 3d 00 35 00 2a 58 ca 8d 34 01 00 00 01 .f.=.5.* X..4....
 0030 00 00 00 00 00 00 03 77 77 77 08 6e 6f 73 74 61W ww.nosta
 0040 72 63 68 03 63 6f 6d 00 00 01 00 01rch.com.

Figure 7-10: The DNS query with the recursion desired bit set

2 0.183134 172.16.0.102 172.16.0.8 DNS Standard query response A 72.32.92.4

Frame 2 (92 bytes on wire, 92 bytes captured)

Ethernet II, Src: 00:0c:29:92:94:9f (00:0c:29:92:94:9f), Dst: 00:25:b3:bf:91:ee (00:25:b3:bf:91:ee)

Internet Protocol, Src: 172.16.0.102 (172.16.0.102), Dst: 172.16.0.8 (172.16.0.8)

User Datagram Protocol, Src Port: 53 (53), Dst Port: 56125 (56125)

Domain Name System (response)

[Request In: 1]

[Time: 0.183134000 seconds]

Transaction ID: 0x8b34 1

Flags: 0x8180 (Standard query response, No error)

Questions: 1
 Answer RRs: 1
 Authority RRs: 0
 Additional RRs: 0

Queries

www.nostarch.com: type A, class IN

Name: www.nostarch.com
 Type: A (Host address)
 Class: IN (0x0001)

Answers

www.nostarch.com: type A, class IN, addr 72.32.92.4

Name: www.nostarch.com
 Type: A (Host address) 2
 Class: IN (0x0001)
 Time to live: 1 hour
 Data length: 4
 Addr: 72.32.92.4

0020 00 08 00 35 db 3d 00 3a 3a 70 8b 34 81 80 00 01 ...S.=.: .p.4....
 0030 00 01 00 00 00 00 03 77 77 77 08 6e 6f 73 74 61W ww.nosta
 0040 72 63 68 03 63 6f 6d 00 00 01 00 01 c0 0c 00 01rch.com.
 0050 00 01 00 00 0e 10 00 04 48 20 5c 04H ..

Figure 7-11: The DNS query response

The only way that we can see that this query was answered by recursion is by listening to the DNS server's traffic when the recursion is taking place, as demonstrated in the file *dns_recursivequery_server.pcap*. This file shows a capture of the traffic on the local DNS server when the query was initiated. The first packet is the same initial query we saw in the previous capture file. At this point, the DNS server has received the query, checked its local database, and realized it does not know the answer to the question of which IP address goes with the

DNS name (*nostarch.com*). Because the packet was sent with the recursion desired bit set, the DNS server can ask another DNS server this question in an attempt to locate the answer, as you can see in the second packet.

In the second packet, the DNS server at 172.16.0.102 transmits a new query to 4.2.2.1, which is the server to which it is configured to forward upstream requests, as shown in Figure 7-12. This query mirrors the original one, effectively turning the DNS server into a client.



Figure 7-12: The recursive DNS query

We can tell that this is a new query because the transaction ID number differs from the transaction ID number in the previous capture file. Once this packet is received by server 4.2.2.1, the local DNS server receives the response shown in Figure 7-13.

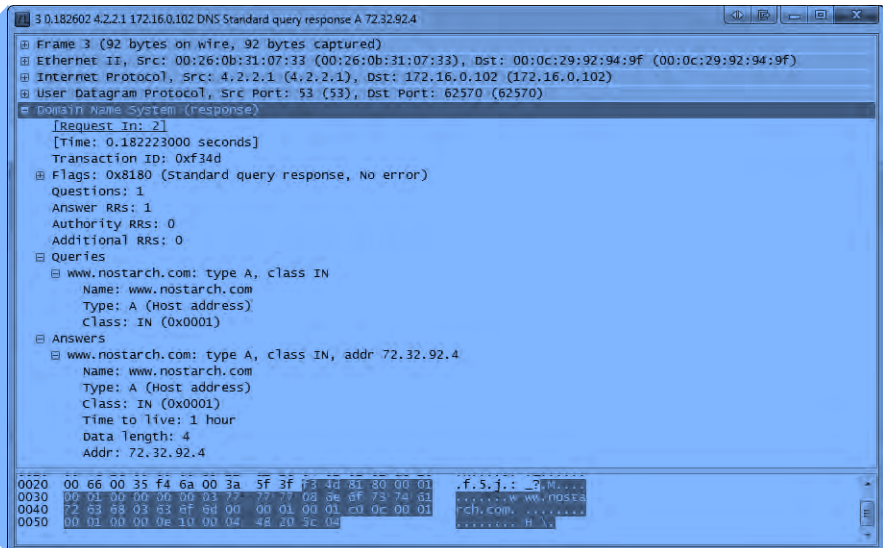


Figure 7-13: Response to the recursive DNS query

Having received this response, the local DNS server can transmit the fourth and final packet to the DNS client with the information requested.

Although this example showed only one layer of recursion, recursion can occur many times for a single DNS request. Here, we received an answer from the DNS server at 4.2.2.1, but that server could have retransmitted the query recursively to another server in order to find the answer. A simple query can travel all over the world before it finally gets a correct response. Figure 7-14 illustrates the recursive DNS query process.



Figure 7-14: A recursive DNS query

DNS Zone Transfers

`dns_axfr.pcap`

A *DNS zone* is the namespace (or group of DNS names) that a DNS server has been delegated to manage. For instance, Emma's Diner might have one DNS server responsible for *emmasdiner.com*. In that case, devices both inside and outside Emma's Diner wishing to resolve *emmasdiner.com* to an IP address would need to contact that DNS server as the authority for that zone. If Emma's Diner were to grow, it could add a second DNS server to handle the email portion of its DNS namespace only, say *mail.emmasdiner.com*, and that server would be the authority for that mail subdomain. Additional DNS servers might be added for subdomains as necessary, as shown in Figure 7-15.

A *zone transfer* occurs when zone data is transferred between two devices, typically out of desire for redundancy. For example, in organizations with multiple DNS servers, administrators commonly configure a secondary DNS server to maintain a copy of the primary server's DNS zone information in case the primary DNS server becomes unavailable. There are two types of zone transfers:

Full zone transfer (AXFR) These types of transfers send an entire zone between devices.

Incremental zone transfer (IXFR) These types of transfers send only a portion of the zone information.

The file *dns_axfr.pcap* contains an example of a full zone transfer between the hosts 172.16.16.164 and 172.16.16.139.

When you first look at this file, you may wonder whether you've opened the right file, because rather than UDP packets, you see TCP packets. Although DNS relies on UDP, it uses TCP for certain tasks, such as zone transfers, because TCP is more reliable for the amount of data being transferred. The first three packets in this capture file are the TCP three-way handshake.

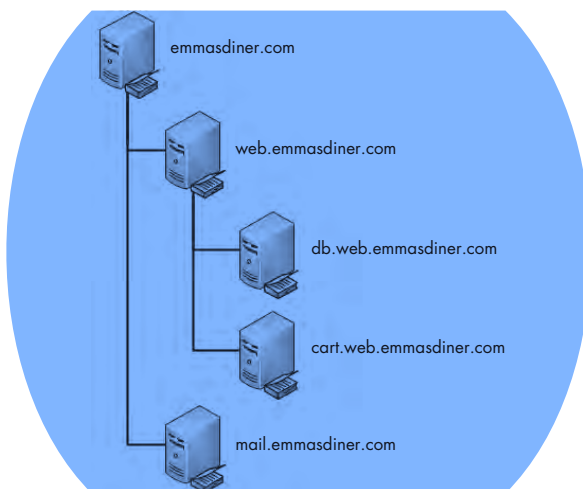


Figure 7-15: DNS zones divide responsibility for namespaces.

The fourth packet begins the actual zone transfer request between 172.16.16.164 and 172.16.16.139. This packet doesn't contain any DNS information. It is marked as a "TCP segment of a reassembled PDU" because the data sent in the zone transfer request packet was sent in multiple packets. Packets 4 and 6 contain the packet's data. Packet 5 is the acknowledgment that packet 4 was received. These packets are displayed in this manner because of the way in which Wireshark parses and displays TCP packets for easier readability. For our purposes, we can reference packet 6 as the complete DNS zone transfer request, as shown in Figure 7-16.

The zone transfer request is a standard query **1**, but instead of requesting a single record type, it requests the AXFR type **2**, meaning that it wishes to receive the entire DNS zone from the server. The server responds with the zone records in packet 7, as shown in Figure 7-17. As you can see, the zone transfer contains quite a bit of data, and this is one of the simpler examples! With the zone transfer complete, the capture file ends with the TCP connection teardown process.

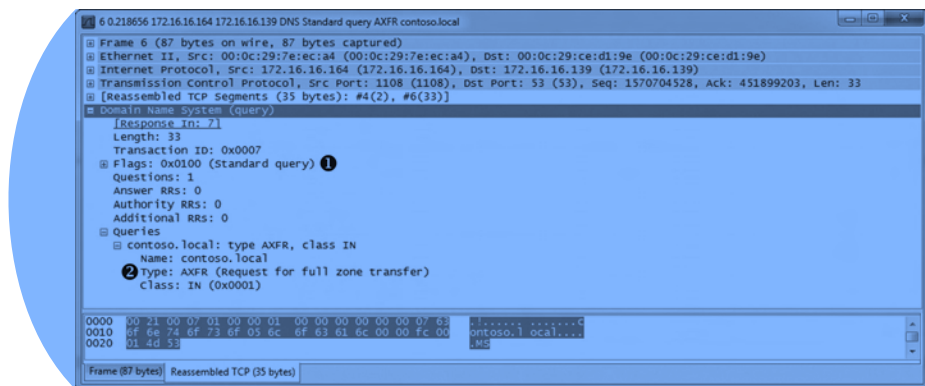


Figure 7-16: DNS full zone transfer request

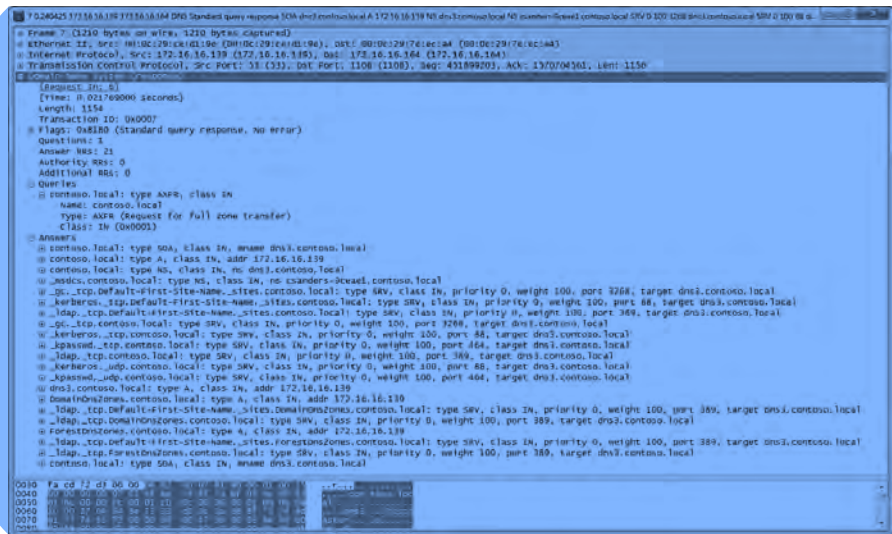


Figure 7-17: The DNS full zone transfer occurring

WARNING *The data contained in a zone transfer can be very dangerous in the wrong hands. For example, by enumerating a single DNS server, you can map a network's entire infrastructure.*

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is the delivery mechanism of the World Wide Web, allowing web browsers to connect to web servers to view web pages. In most organizations, HTTP represents, by far, the highest percentage of traffic seen going across the wire. Every time you do a Google search, connect to Twitter to send a tweet, or check University of Kentucky basketball scores on ESPN.com, you're using HTTP.

We won't look at the packet structures for an HTTP transfer. Because the contents of those packets vary widely depending on their purpose, that exercise is left to you. Here, we'll look at some practical applications of HTTP.

Browsing with HTTP

http_google.pcap

HTTP is most commonly used to browse web pages on a web server using a web browser. The capture file *http_google.pcap* shows such an HTTP transfer, using TCP as the transport layer protocol. Communication begins with a three-way handshake between the client 172.16.16.128 and the Google web server 74.125.95.104.

Once communication is established, the first packet is marked as an HTTP packet from the client to the server, as shown in Figure 7-18.

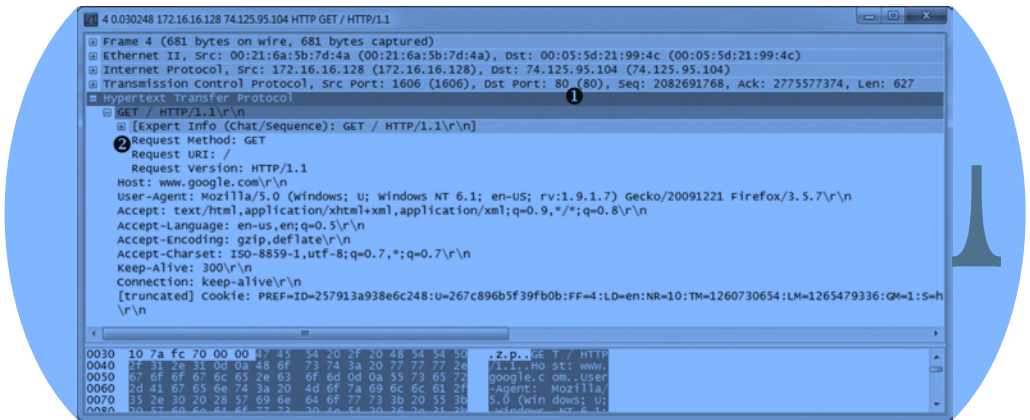


Figure 7-18: The initial HTTP GET request packet

The HTTP packet is delivered over TCP to the server's port 80 ❶, the standard port for HTTP communication (8080 is also commonly used).

HTTP packets are identified by one of eight different request methods (defined in HTTP specification version 1.1), which indicate the action the packet's transmitter will perform on the receiver. As shown in Figure 7-18, this packet identifies its method as GET, its request Uniform Resource Indicator (URI) as /, and the request version as HTTP/1.1 ❷. This information tells us that the client is sending a request to download (GET) the root web directory (/) of the web server using version 1.1 of HTTP.

Next, the host sends information about itself to the web server. This information includes things such as the user agent (browser) being used, languages accepted by the browser (Accept-Languages), and cookie information (at the bottom of the capture). The server can use this information to determine which data to return to the client in order to ensure compatibility.

When the server receives the HTTP GET request in packet 4, it responds with a TCP ACK, acknowledging the packet, and begins transmitting the requested data from packets 6 to 11. HTTP is used only to issue application layer commands between the client and server. When it's time to transfer data, application layer control is not seen, except for at the beginning and end of the data stream.

Data is sent from the server in packets 6 and 7, an acknowledgment from the client in packet 8, two more data packets in packets 9 and 10, and another acknowledgment in packet 11, as shown in Figure 7-19. All of these packets are shown in Wireshark as TCP segments, rather than HTTP packets, although HTTP is still responsible for their transmission.

No.	Time	Source	Destination	Protocol	Info
6	0.101202	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]
7	0.101465	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]
8	0.101495	172.16.16.128	74.125.95.104	TCP	1606 > 80 [ACK] Seq=2082692395 Ack=2775580186 Win=4218 Len=0
9	0.102282	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]
10	0.102350	74.125.95.104	172.16.16.128	TCP	[TCP segment of a reassembled PDU]
11	0.102364	172.16.16.128	74.125.95.104	TCP	1606 > 80 [ACK] Seq=2082692395 Ack=2775581694 Win=4218 Len=0

Figure 7-19: TCP transmitting data between the client browser and web server

Once the data is transferred, a reassembled stream of the data is sent, as shown in Figure 7-20.

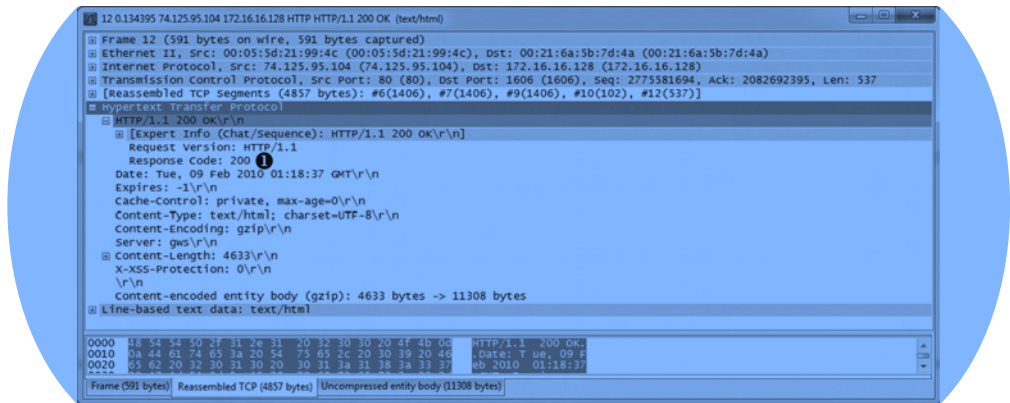


Figure 7-20: Final HTTP packet with response code 200

HTTP uses a number of predefined response codes to indicate the results of a request method. In this example, we see a packet with response code 200, which indicates a successful request method. The packet also includes a timestamp and some additional information about the encoding of the content and configuration parameters of the web server. When the client receives this packet, the transaction is complete.

Posting Data with HTTP

http_post.pcap

Now that we have looked at the process of downloading data from a web server, let's turn our attention to uploading data. The file *http_post.pcap* contains a very simple example of an upload: a user posting a comment to a website. After the initial three-way handshake, the client (172.16.16.128) sends an HTTP packet to the web server (69.163.176.56), as shown in Figure 7-21.

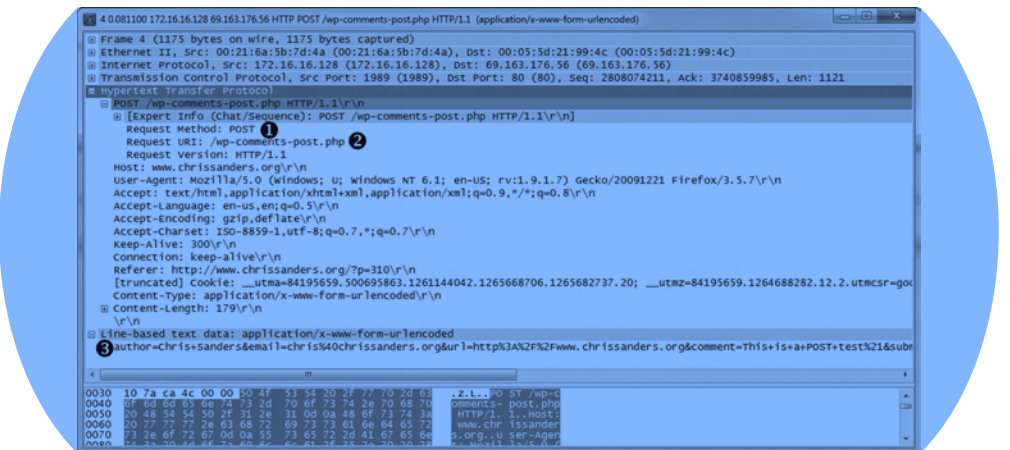


Figure 7-21: The HTTP POST packet

This packet uses the POST method ❶ to upload data to a web server for processing. The POST method used here specifies the URI /wp-comments-post.php ❷, and the HTTP 1.1 Request version. To see the contents of the data posted, expand the Line-based Text Data portion of the packet ❸.

Once the data is transmitted in this POST, an ACK packet is sent. As shown in Figure 7-22, the server responds with packet 6, transmitting the response code 302 ❶, which means “found.”

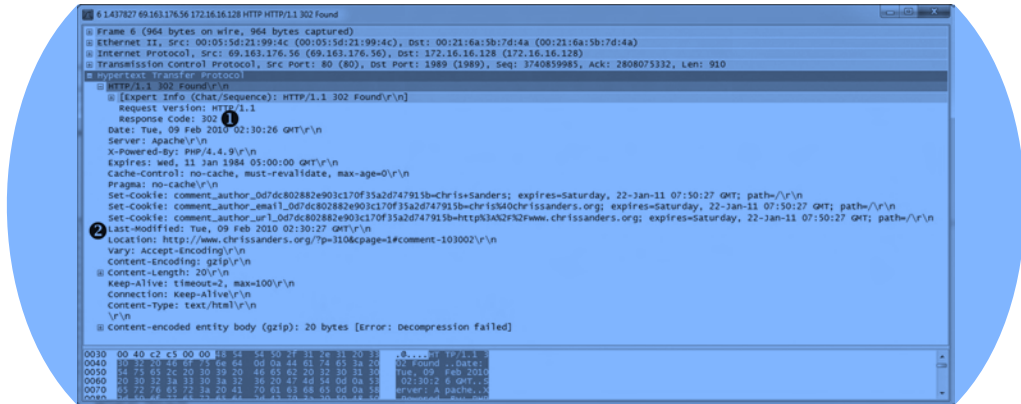


Figure 7-22: HTTP response 302 is used to redirect.

The 302 response code is a common means of redirection in the HTTP world. The Location field in this packet specifies where the client is to be directed ❷. In this case, that’s to the place on the originating web page where the comment was posted. Finally, the server transmits status code 200, and the page’s content is sent over the next several packets to complete the transmission.

Final Thoughts

The chapter has introduced the most common protocols you will encounter when examining traffic at the application layer. In the following chapters, we’ll examine new protocols and additional features of the protocols we’ve covered here, as we explore a wide range of real scenarios.

To learn more about individual protocols, read their associated RFC or have a look at *The TCP/IP Guide* by Charles Kozierok (No Starch Press, 2005). Also, see the list of resources in the appendix.