COMPUTER NETWORK NOTES

{snippets from PRACTICAL PACKET ANALYSIS by Chris Sanders}

Refer this playlist- https://youtube.com/playlist?list=PLowKtXNTBypH19whXTVoG3oKSuOcw_XeW

NETWORK BASICS

How Computers Communicate

To fully understand packet analysis, you must know exactly how computers communicate with each other. In this section, we'll examine the basics of network protocols, the Open Systems Interconnections (OSI) model, network data frames, and the hardware that supports it all.

Protocols

Modern networks are made up of a variety of systems running on many different platforms. To communicate between systems, we use a set of common languages called *protocols*. Common protocols include Transmission Control Protocol (TCP), Internet Protocol (IP), Address Resolution Protocol (ARP), and Dynamic Host Configuration Protocol (DHCP). A logical grouping of protocols that work together is called a *protocol stack*.

It might help to think of protocols as similar to the rules that govern human language. Every language has rules such as how to conjugate verbs, how to greet people, and even how to properly thank someone. Protocols work in much the same fashion, allowing us to define how packets should be routed, how to initiate a connection, and how to acknowledge the receipt of data.

A protocol can be extremely simple or highly complex, depending on its function. Although the various protocols can differ significantly, many protocols address the following issues:

Connection initiation Is it the client or server initiating the connection? What information must be exchanged prior to communication?

Negotiation of connection characteristics Is the communication of the protocol encrypted? How are encryption keys transmitted between communicating hosts?

Data formatting How is the data contained within the packet organized? In what order is the data processed by the devices receiving it?

Error detection and correction What happens in the event that a packet takes too long to reach its destination? How does a client recover if it cannot establish communication with a server for a short duration?

Connection termination How does one host signify to the other that communication has ended? What final information must be transmitted in order to gracefully terminate communication?

The Seven-Layer OSI Model

Protocols are separated according to their functions based on the industry-standard OSI reference model. This hierarchical model, with seven distinct layers, is very helpful for understanding network communications. In Figure 1-1, the layers of the OSI model are on the right, and the proper terminology for data at each of these layers is on the left. The application layer at the top represents the programs used to access network resources. The bottom layer is the physical layer, through which the network data travels. The protocols at each layer work together to ensure data is properly handled by the protocols at layers directly above and below.

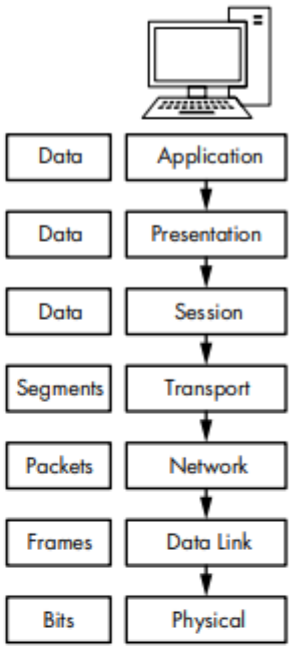


Figure 1-1: A hierarchical view of the seven layers of the OSI model

NOTE *The OSI model was originally published in 1983 by the International Organization for Standardization (ISO) as a document called ISO 7498. The OSI model is no more than an industry-recommended standard. Protocol developers are not required to follow it exactly. In fact, the OSI model is not the only networking model; for example, some people prefer the Department of Defense (DoD) model, also known as the TCP/IP model.*

Each OSI model layer has a specific function, as follows:

Application layer (layer 7) The topmost layer of the OSI model provides a means for users to access network resources. This is the only layer typically seen by end users, as it provides the interface that is the base for all of their network activities.

Presentation layer (layer 6) This layer transforms the data it receives into a format that can be read by the application layer. The data encoding and decoding done here depends on the application layer protocol that is sending or receiving the data. The presentation layer also handles several forms of encryption and decryption used to secure data.

Session layer (layer 5) This layer manages the *dialogue*, or session, between two computers. It establishes, manages, and terminates this connection among all communicating devices. The session layer is also responsible for establishing whether a connection is duplex (two-way) or half-duplex (one-way) and for gracefully closing a connection between hosts rather than dropping it abruptly.

Transport layer (layer 4) The primary purpose of the transport layer is to provide reliable data transport services to lower layers. Through flow control, segmentation/desegmentation, and error control, the transport layer makes sure data gets from point to point error-free.

Because ensuring reliable data transportation can be extremely cumbersome, the OSI model devotes an entire layer to it. The transport layer utilizes both connection-oriented and connectionless protocols. Certain firewalls and proxy servers operate at this layer.

Network layer (layer 3) This layer, one of the most complex of the OSI layers, is responsible for routing data between physical networks. It sees to the logical addressing of network hosts (for example, through an IP address). It also handles splitting data streams into smaller fragments and, in some cases, error detection. Routers operate at this layer.

Data link layer (layer 2) This layer provides a means of transporting data across a physical network. Its primary purpose is to provide an addressing scheme that can be used to identify physical devices (for example, MAC addresses). Bridges and switches are physical devices that operate at the data link layer.

Physical layer (layer 1) The layer at the bottom of the OSI model is the physical medium through which network data is transferred. This layer defines the physical and electrical nature of all hardware used, including voltages, hubs, network adapters, repeaters, and cabling specifications. The physical layer establishes and terminates connections, provides a means of sharing communication resources, and converts signals from digital to analog and vice versa.

NOTE *A common mnemonic device for remembering the layers of the OSI model is* Please Do Not Throw Sausage Pizza Away. *The first letter of each word refers to each layer of the OSI model, starting with the first layer.*

Table 1-1 lists some of the more common protocols used at each layer of the OSI model.

Table 1-1: Typical Protocols Used at Each Layer of the OSI Model

Layer	Protocols
Application	HTTP, SMTP, FTP, Telnet
Presentation	ASCII, MPEG, JPEG, MIDI
Session	NetBIOS, SAP, SDP, NWLink
Transport	TCP, UDP, SPX
Network	IP, IPX
Data link	Ethernet, Token Ring, FDDI, AppleTalk
Physical	wired, wireless

Although the OSI model is no more than a recommended standard, you should know it by heart as it provides a useful vocabulary for thinking about and describing network problems. As we progress through this book, you will find that router issues soon become "layer 3 problems" and software issues are readily recognized as "layer 7 problems."

NOTE *A colleague once told me about a user who complained that he could not access a network resource. The issue was the result of the user's entering an incorrect password. My colleague referred to this as a* layer 8 issue. *Layer 8 is the unofficial user layer. This term is commonly used among those who live at the packet level.*

Data Flow Through the OSI Model

The initial data transfer on a network begins at the application layer of the transmitting system. Data works its way down the seven layers of the OSI model until it reaches the physical layer, at which point the physical layer of the transmitting system sends the data to the receiving system. The receiving system picks up the data at its physical layer, and the data proceeds up the layers of the receiving system to the application layer at the top.

Each layer in the OSI model is capable of communicating only with the layers directly above and below it. For example, layer 2 can send and receive data only from layers 1 and 3.

None of the services provided by various protocols at any given level of the OSI is redundant. For example, if a protocol at one layer provides a particular service, then no other protocol at any other layer will provide this same service. Protocols at different levels may have features with similar goals, but they will function a bit differently.

Protocols at corresponding layers on the sending and receiving devices are complementary. So, for example, if a protocol at layer 7 of the sending device is responsible for formatting the data being transmitted, the corresponding protocol at layer 7 of the receiving device is expected to be responsible for reading that formatted data.

Figure 1-2 is a graphical representation of the OSI model as it relates to two communicating devices. You can see communication going from top to bottom on one device and then reversing when it reaches the second device.

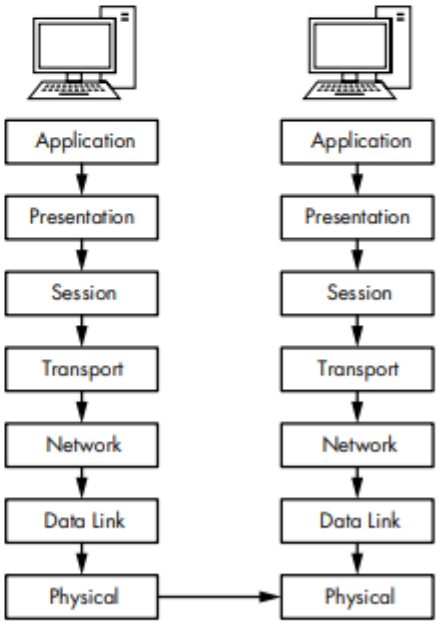


Figure 1-2: Protocols working at the same layer on both the sending and receiving systems

Data Encapsulation

The protocols at different layers of the OSI model pass data between each other with the aid of *data encapsulation*. Each layer in the stack is responsible for adding a

header or footer—extra bits of information that allow the layers to communicate—to the data being transferred. For example, when the transport layer receives data from the session layer, the transport layer adds its own header information to that data before passing it to the network layer.

The encapsulation process creates a protocol data unit (PDU), which includes the data being sent and all header or footer information added to it. As data moves down the OSI model and the various protocols add header and footer information, the PDU changes and grows. The PDU is in its final form when it reaches the physical layer, at which point it is sent to the destination device. The receiving device strips the protocol headers and footers from the PDU as the data climbs up the OSI layers in the reverse of the order they were added. Once the PDU reaches the top layer of the OSI model, only the original application layer data remains.

NOTE *The OSI model uses specific terms to describe packaged data at each layer. The physical layer contains bits, the data link layer contains frames, the network layer contains packets, and the transport layer contains segments. The top three layers simply use the term* data. *This nomenclature isn't used much in practice, so we'll generally just use the term* packet *to refer to a complete or partial PDU that includes header and footer information from a few or many layers of the OSI model.*

To illustrate how encapsulation of data works, we'll look at a simplified practical example of a packet being built, transmitted, and received in relation to the OSI model. Keep in mind that as analysts, we don't often talk about the session or presentation layers, so those will be absent in this example (and the rest of this book).

In this scenario, we are attempting to browse to *http://www.google.com/*. First, we must generate a request packet that is transmitted from our source client computer to the destination server computer. This scenario assumes that a TCP/IP communication session has already been initiated. Figure 1-3 illustrates the data encapsulation process in this example.

We begin on our client computer at the application layer. We are browsing to a website, so the application layer protocol being used is HTTP; the HTTP protocol will issue a command to download the file *index.html* from *google.com*.

NOTE *In practice, the browser will request the website document root first, signified by a forward slash (/). When the web server receives this request, it will redirect the browser to whatever file it is configured to serve upon receiving a document root request. This is usually something like* index.html *or* index.php. *We'll cover this more in Chapter 9 when we discuss HTTP.*

Once our application layer protocol has sent the command, our concern is with getting the packet to its destination. The data in our packet is passed down the OSI stack to the transport layer. HTTP is an application layer protocol that uses (or *sits on*) TCP, so TCP serves as the transport layer protocol used to ensure reliable delivery of the packet. A TCP header is generated

and added to the PDU, as shown in the transport layer of Figure 1-3. This TCP header includes sequence numbers and other data that are appended to the packet, ensuring that the packet is properly delivered.

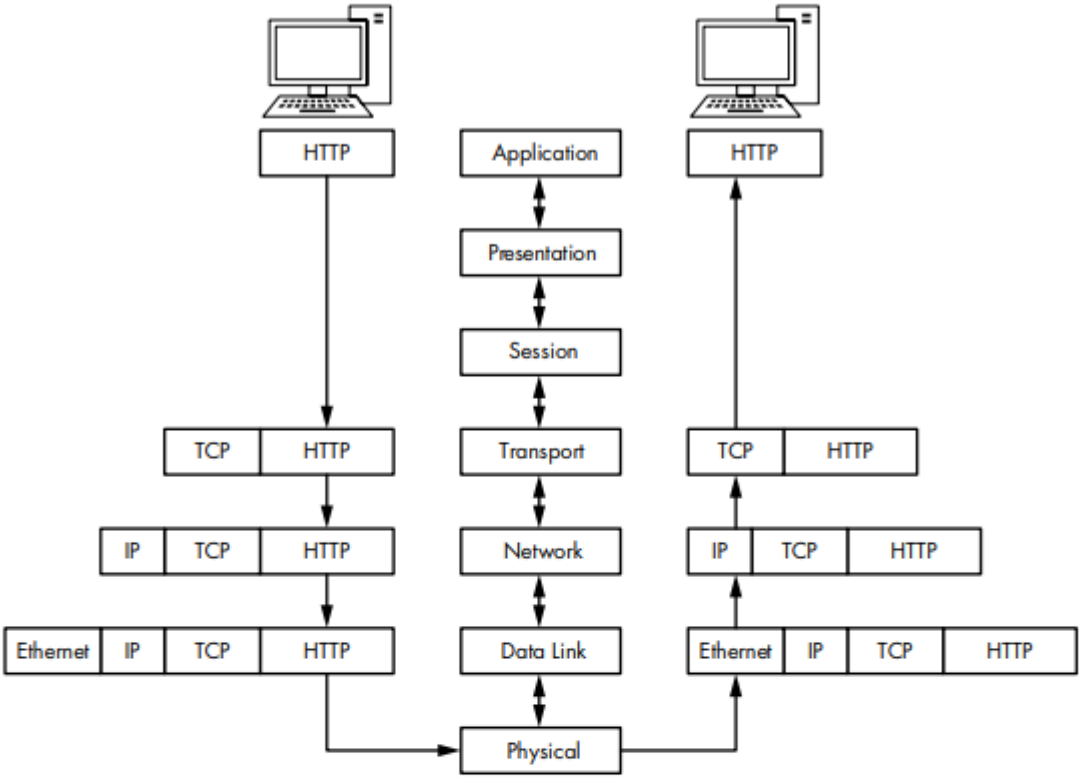


Figure 1-3: A graphical representation of encapsulation of data between client and server

NOTE *We often say that one protocol "sits on" or "rides on" another protocol because of the top-down design of the OSI model. An application protocol such as HTTP provides a particular service and relies on TCP to ensure reliable delivery of its service. Both of those services rely on the IP protocol at the network level to address and deliver their data. Therefore, HTTP sits on TCP, which sits on IP.*

Having done its job, TCP hands the packet off to IP, which is the layer 3 protocol responsible for the logical addressing of the packet. IP creates a header containing logical addressing information, adds it to the PDU, and passes the packet along to the Ethernet on the data link layer. Physical Ethernet addresses are stored in the Ethernet header. The packet is now fully assembled and passed to the physical layer, where it is transmitted as zeros and ones across the network.

The completed packet traverses the network cabling system, eventually reaching the Google web server. The web server begins by reading the packet from the bottom up, meaning that it first reads the data link layer, which contains the physical Ethernet addressing information that the

network card uses to determine that the packet is intended for a particular server. Once this information is processed, the layer 2 information is stripped away, and the layer 3 information is processed.

The layer 3 IP addressing information is read to ensure that the packet is properly addressed and is not fragmented. This data is also stripped away so that the next layer can be processed.

Layer 4 TCP information is now read to ensure that the packet has arrived in sequence. Then the layer 4 header information is stripped away to leave only the application layer data, which can be passed to the web server application hosting the website. In response to this packet from the client, the server should transmit a TCP acknowledgment packet so the client knows its request was received, followed by the *index.html* file.

All packets are built and processed as described in this example, regardless of which protocols are used. But at the same time, keep in mind that not every packet on a network is generated from an application layer protocol, so you will see packets that contain only information from layer 2, 3, or 4 protocols.

Network Hardware

Now it's time to look at network hardware, where the dirty work is done. We'll focus on just a few of the more common pieces of network hardware: hubs, switches, and routers.

Hubs

A *hub* is generally a box with multiple RJ-45 ports, like the NETGEAR hub shown in Figure 1-4. Hubs range from very small 4-port devices to larger 48-port devices designed for rack mounting in a corporate environment.

Figure 1-4: A typical 4-port Ethernet hub

Because hubs can generate a lot of unnecessary network traffic and are capable of operating only in *half-duplex mode* (they cannot send and receive data at the same time), you won't typically see them used in most modern or high-density networks; switches are used instead (discussed in the next section). However, you should know how hubs work, since they will be very important to packet analysis when using the "hubbing out" technique discussed in Chapter 2.

A hub is no more than a *repeating device* that operates on the physical layer of the OSI model. It takes packets sent from one port and transmits (repeats) them to every other port on the device, and it's up to the

receiving device to accept or reject each packet. For example, if a computer on port 1 of a 4-port hub needs to send data to a computer on port 2, the hub sends those packets to ports 2, 3, and 4. The clients connected to ports 3 and 4 examine the destination Media Access Control (MAC) address field in the Ethernet header of the packet and see that the packet is not for them, so they *drop* (discard) the packet. Figure 1-5 illustrates an example in which computer A is transmitting data to computer B. When computer A sends this data, all computers connected to the hub receive it. However, only computer B actually accepts the data; the other computers discard it.

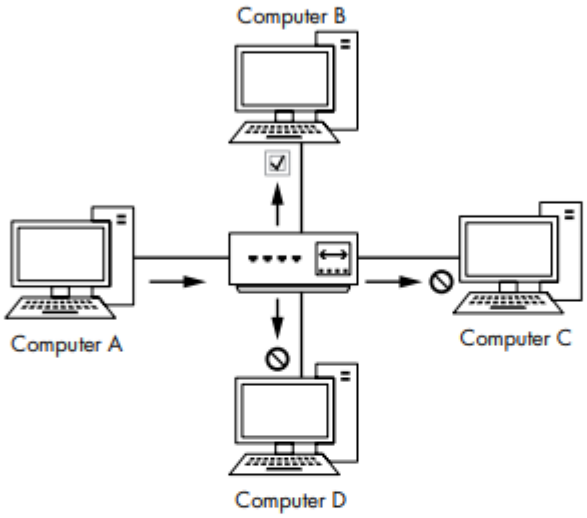


Figure 1-5: The flow of traffic when computer A transmits data to computer B through a hub

As an analogy, suppose that you sent an email with the subject line "Attention all marketing staff" to every employee in your company, rather than to only those people who work in the marketing department. The marketing department employees see the email is for them and open it. The other employees see it's not for them and discard it. You can see how this approach to communication would result in a lot of unnecessary traffic and wasted time, yet this is exactly how a hub functions.

The best alternatives to hubs in production and high-density networks are *switches*, which are *full-duplex devices* that can send and receive data synchronously.

Switches

Like a hub, a switch is designed to repeat packets. However, unlike a hub, rather than broadcasting data to every port, a switch sends data to only the computer for which the data is intended. Switches look just like hubs, as shown in Figure 1-6.

Figure 1-6: A rack-mountable 48-port Ethernet switch

Several larger switches on the market, such as Cisco-branded ones, are managed via specialized, vendor-specific software or web interfaces. These switches are commonly referred to as *managed switches*. Managed switches provide several features that can be useful in network management, including the ability to enable or disable specific ports, view port statistics, make configuration changes, and remotely reboot.

Switches also offer advanced functionality for handling transmitted packets. To be able to communicate directly with specific devices, switches must be able to uniquely identify devices based on their MAC addresses, which means that they must operate on the data link layer of the OSI model.

Switches store the layer 2 address of every connected device in a *CAM table*, which acts as a kind of traffic cop. When a packet is transmitted, the switch reads the layer 2 header information in the packet and, using the CAM table as reference, determines to which port(s) to send the packet. Switches send packets only to specific ports, thus greatly reducing network traffic.

Figure 1-7 illustrates traffic flow through a switch. In this figure, computer A is sending data to only the intended recipient: computer B. Multiple conversations can happen on the network at the same time, but information is communicated directly between the switch and intended recipient, not between the switch and all connected computers.

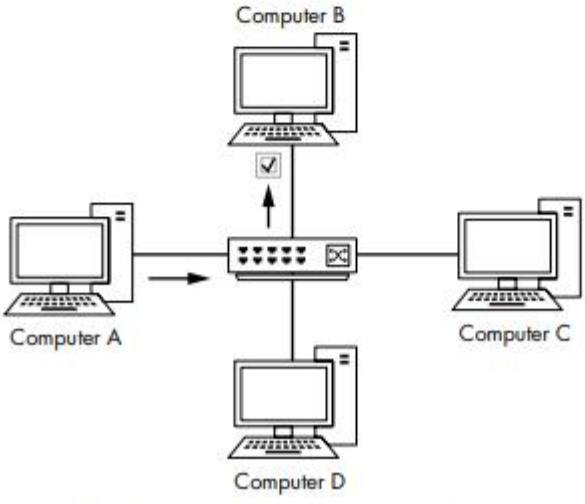


Figure 1-7: The flow of traffic when computer A transmits data to computer B through a switch

Routers

A *router* is an advanced network device with a much higher level of functionality than a switch or a hub. A router can take many shapes and forms, but most devices have several LED indicator lights on the front and a few network ports on the back, depending on the size of the network. Figure 1-8 shows an example of a small router.

Figure 1-8: A low-level Enterasys router suitable for use in a small to midsized network

Routers operate at layer 3 of the OSI model, where they are responsible for forwarding packets between two or more networks. The process used by routers to direct the flow of traffic among networks is called *routing*. Several types of routing protocols dictate how different types of packets are routed to other networks. Routers commonly use layer 3 addresses (such as IP addresses) to uniquely identify devices on a network.

A good way to illustrate the concept of routing is to use the analogy of a neighborhood with several streets. Think of the houses, with their addresses, as computers. Then think of each street as a network segment. Figure 1-9 illustrates this comparison. From your house, you can easily go visit your neighbors in the other houses on the same street by walking in a straight line from your front door to theirs. In the same way, a switch allows communication among all computers on a network segment.

However, communicating with a neighbor who lives on another street is like communicating with a computer that is not on the same segment. Referring to Figure 1-9, let's say that you're sitting at 502 Vine Street and need to get to 206 Dogwood Lane. In order to do this, you must first turn onto Oak Street and then turn onto Dogwood Lane. Think of this as crossing network segments. If the device at 192.168.0.3 needs to communicate with the device at 192.168.0.54, it must cross a router to get to the 10.100.1.x network and then cross the destination network segment's router before it can get to the destination network segment.

The size and number of routers on a network will typically depend on the network's size and function. Personal and home office networks may have only a small router located at the edge of the network. A large corporate network might have several routers spread throughout various departments, all connecting to one large central router or layer 3 switch (an advanced type of switch that also has built-in functionality to act as a router).

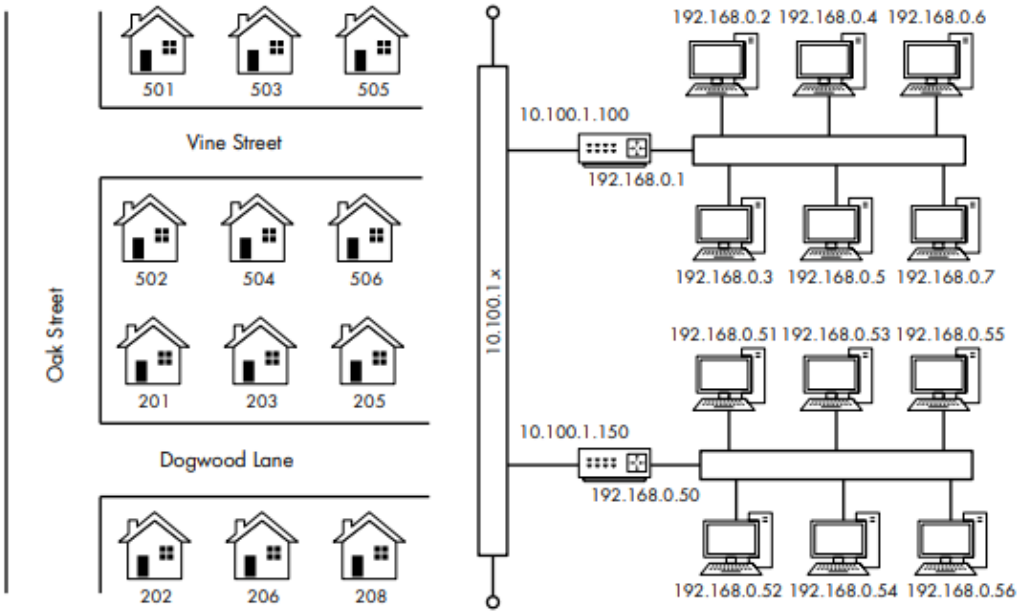


Figure 1-9: Comparison of a routed network to neighborhood streets

As you look at more and more network diagrams, you will come to understand how data flows through these various points. Figure 1-10 shows the layout of a very common form of routed network. In this example, two separate networks are connected via a single router. If a computer on network A wishes to communicate with a computer on network B, the transmitted data must go through the router.

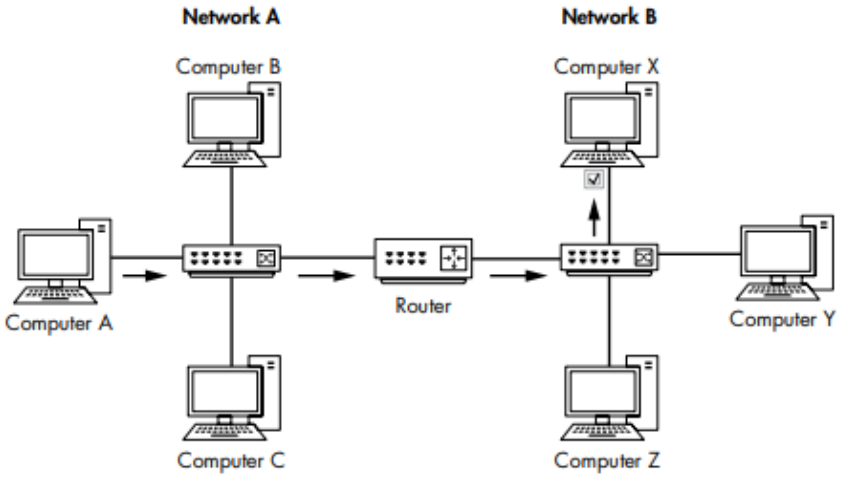


Figure 1-10: The flow of traffic when computer A on one network transmits data to computer X on another network through a router

Traffic Classifications

Network traffic can be classified as one of three types: broadcast, multicast, and unicast. Each classification has a distinct characteristic that determines how packets in that class are handled by networking hardware.

Broadcast Traffic

A *broadcast packet* is a packet that's sent to all ports on a network segment, regardless of whether a given port is a hub or switch.

There are layer 2 and layer 3 forms of broadcast traffic. On layer 2, the MAC address ff:ff:ff:ff:ff:ff is the reserved broadcast address, and any traffic sent to this address is broadcast to the entire network segment. Layer 3 also has a specific broadcast address, but it varies based on the network address range in use.

The highest possible IP address in an IP network range is reserved for use as the broadcast address. For example, if your computer has an address of 192.168.0.20 and a 255.255.255.0 subnet mask, then 192.168.0.255 is the broadcast address (more on IP addressing in Chapter 7).

The extent to which broadcast packets can travel is called the *broadcast domain*, which is the network segment where any computer can directly transmit to another computer without going through a router. In larger networks with multiple hubs or switches connected via different media, broadcast packets transmitted from one switch reach all the ports on all the other switches on the network, as the packets are repeated from switch to switch. Figure 1-11 shows an example of two broadcast domains on a small network. Because each broadcast domain extends until it reaches the router, broadcast packets circulate only within this specified broadcast domain.

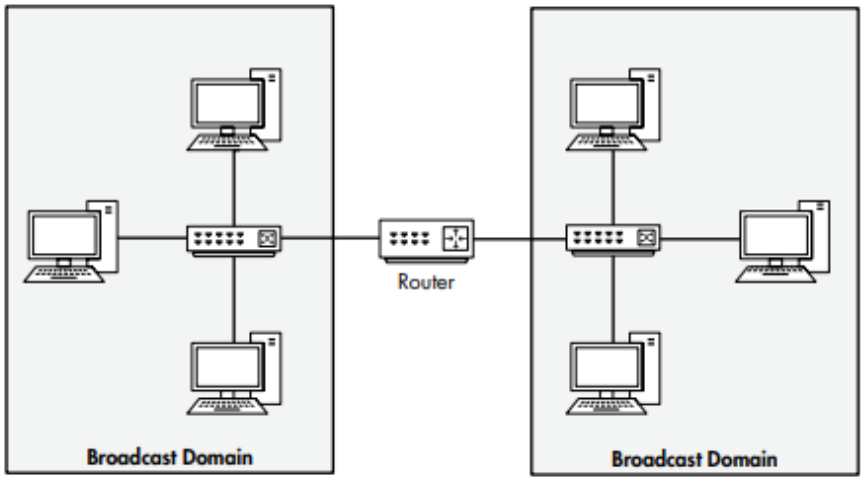


Figure 1-11: A broadcast domain extends to everything behind the current routed segment.

Our earlier neighborhood analogy provides good insight into how broadcast domains work, too. You can think of a broadcast domain as being like a neighborhood street where all your neighbors are sitting on their front porch. If you stand on your front porch and yell, the people on your street will be able to hear you. However, if you want to talk to someone on a different street, you need to find a way to speak to that person directly, rather than broadcasting (yelling) from your front porch.

Multicast Traffic

Multicast is a means of transmitting a packet from a single source to multiple destinations simultaneously. The goal of multicasting is to use as little bandwidth as possible. The optimization of this traffic lies in that a stream of data is replicated fewer times along its path to its destination. The exact handling of multicast traffic is highly dependent on its implementation in individual protocols.

The primary method of implementing multicast traffic is via an addressing scheme that joins the packet recipients to a multicast group. This is how IP multicast works. This addressing scheme ensures that the packets cannot be transmitted to computers to which the packets are not destined. In fact, IP devotes an entire range of addresses to multicast. If you see an IP address in the 224.0.0.0 to 239.255.255.255 range, it is most likely handling multicast traffic because these ranges are reserved for that purpose.

Unicast Traffic

A *unicast packet* is transmitted from one computer directly to another. The details of how unicast functions are dependent on the protocol using it. For example, consider a device that wishes to communicate with a web server. This is a one-to-one connection, so this communication process would begin with the client device transmitting a packet to only the web server.

Final Thoughts

This chapter covered the basics of networking that you need as a foundation for packet analysis. You *must* understand what is going on at this level of network communication before you can begin troubleshooting network issues. In Chapter 2, we will look at multiple techniques for capturing the packets you want to analyze.

NETWORK LAYER PROTOCOL

Address Resolution Protocol (ARP)

Both logical and physical addresses are used for communication on a network. Logical addresses allow for communication among multiple networks and indirectly connected devices. Physical addresses facilitate 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 in which 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 remote host's IP address (via DNS, covered in Chapter 9), 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 information 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 which port to send the traffic through. If the destination MAC address is unknown, the transmitting device will first check for the address in its cache; if the address isn't 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 7-1).

The transmitting computer sends out an ARP request that basically says, “Howdy, everybody. My IP address is 192.168.0.101, and my MAC address is f2:f2:f2:f2:f2:f2. I need to send something to whoever has the IP address 192.168.0.1, but I don’t know the hardware address. Will whoever has this IP address please respond with your MAC address?”

This packet is broadcast to every device on the network segment. Any device that doesn’t have this IP address simply discards the packet. The device that does have the address sends an ARP reply with an answer such as “Hey, transmitting device, I’m the one you’re looking for with the IP address 192.168.0.1. My MAC address is 02:f2:02:f2:02:f2.”

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

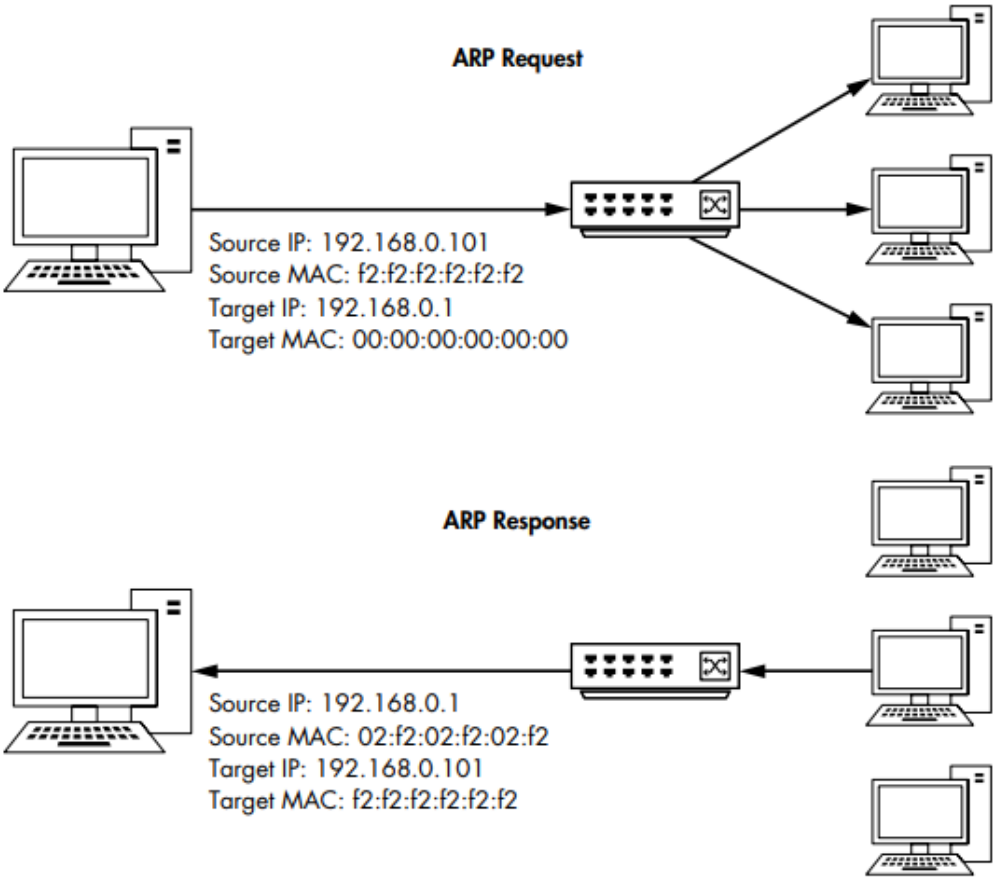


Figure 7-1: The ARP resolution process

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 understand how it works. But before we look at some examples, let's examine the ARP packet header.

ARP Packet Structure

As shown in Figure 7-2, 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

Address Resolution Protocol (ARP)					
Offsets	Octet	0	1	3	4
Octet	Bit	0–7	8–15	0–7	8–15
0	0	Hardware Type		Protocol Type	
4	32	Hardware Address Length	Protocol Address Length	Operation	
8	64	Sender Hardware Address			
12	96	Sender Hardware Address		Sender Protocol Address	
16	128	Sender Protocol Address		Target Hardware Address	
20	160	Target Hardware Address			
24+	192+	Target Protocol Address			

Figure 7-2: The ARP packet structure

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 (all zeroes in ARP requests)

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

Now open the file *arp_resolution.pcapng* 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.pcapng

The first packet is the ARP request, as shown in Figure 7-3. 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.

Given this structure, we can discern that this is indeed an ARP request on an Ethernet network using IPv4. 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) ❺. 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 ❹.

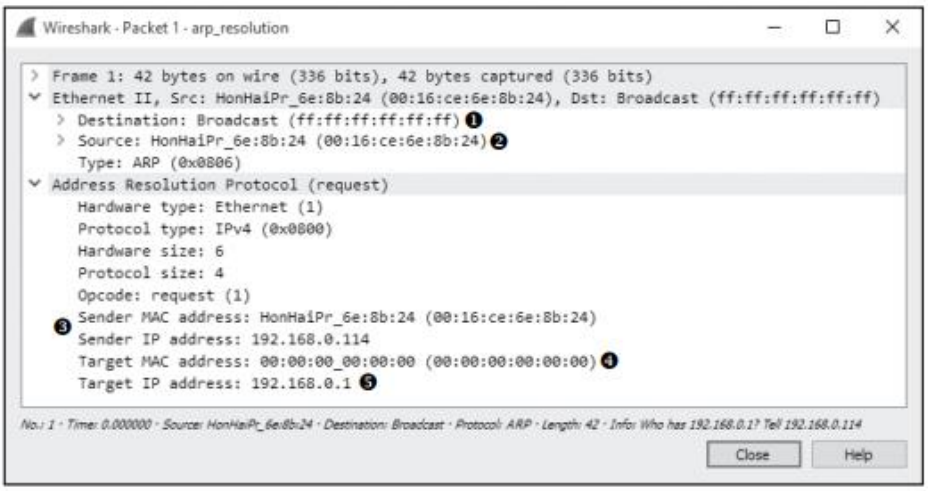


Figure 7-3: An ARP request packet

Packet 2: ARP Response

In the response to the initial request (see Figure 7-4), 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 ❸.
- Most important, all the information is present, meaning we now have the MAC address (00:13:46:0b:22:ba) ❷ of our host at 192.168.0.1.

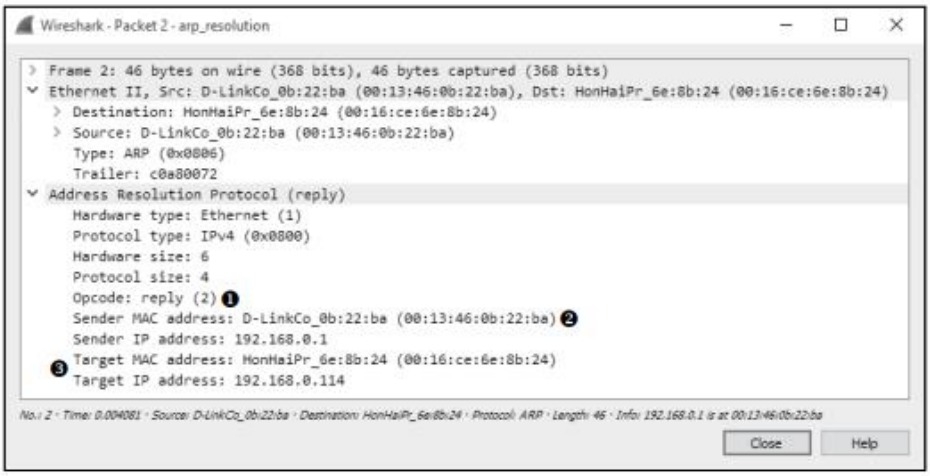


Figure 7-4: An ARP reply packet

Gratuitous ARP

arp_gratuitous.pcapng

Where I come from, when something is done "gratuitously," the word usually carries a negative connotation. A *gratuitous ARP*, however, is 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 7-5).

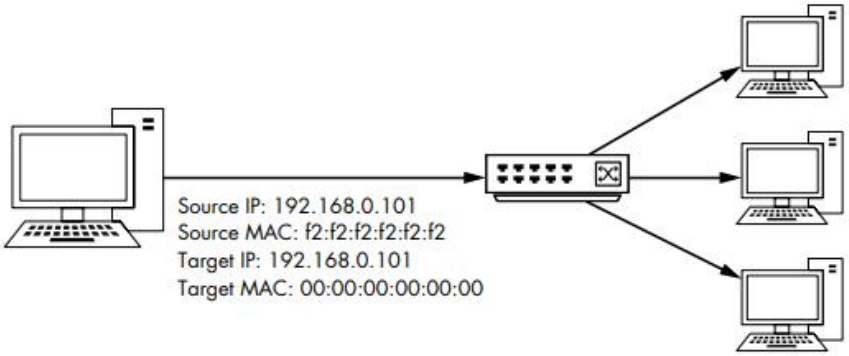


Figure 7-5: The gratuitous ARP process

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.pcapng*, and you'll see this in action. This file contains only a single packet (see Figure 7-6) because that's all that's involved in gratuitous ARP.

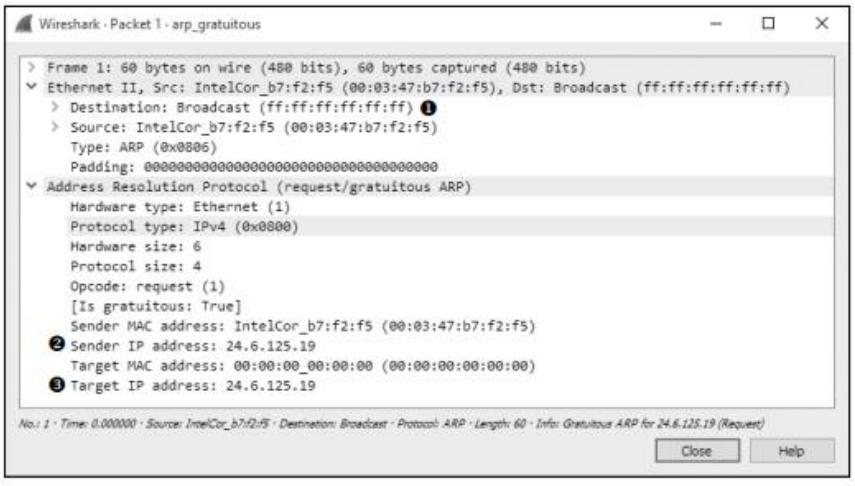


Figure 7-6: A gratuitous ARP packet

Internet Protocol (IP)

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 used in internetwork communication. A few protocols can do this, but the most common is the *Internet Protocol (IP)*, which currently has two versions in use—IP version 4 and IP version 6. We'll start by examining IP version 4 (IPv4), which is defined in RFC 791.

Internet Protocol Version 4 (IPv4)

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, 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.

IPv4 Addresses

IPv4 addresses are 32-bit assigned numbers used to uniquely identify devices connected to a network. It's 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* (or *dotted-decimal*) *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 7-7). 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.

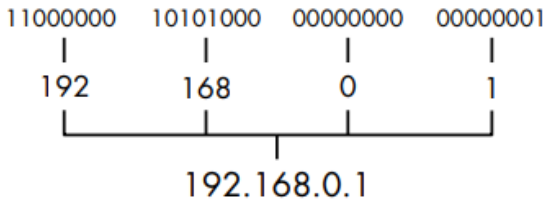


Figure 7-7: Dotted-quad IPv4 address notation

An IP address consists of two parts: a *network portion* and a *host portion*. The network portion identifies the LAN the device is connected to, and the host portion identifies the device itself on that network. The determination of which part of the IP address belongs to the network or host portion is not always the same. This information is communicated by another set of addressing information called the *network mask (netmask)* or sometimes referred to as a *subnet mask*.

NOTE *In this book, when we refer to an IP address, we will always be referring to an IPv4 address. Later in this chapter, we will look at IP version 6, which uses a different set of rules for addressing. Whenever we refer to an IPv6 address, it will be explicitly labeled as such.*

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

For example, consider the IP address 10.10.1.22, represented in binary as 00001010 00001010 00000001 00010110. 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 (10.10 or 00001010 00001010) is reserved for the network portion, and the last half of the IP address (.1.22 or 00000001 00010110) identifies the individual host on this network, as shown in Figure 7-8.

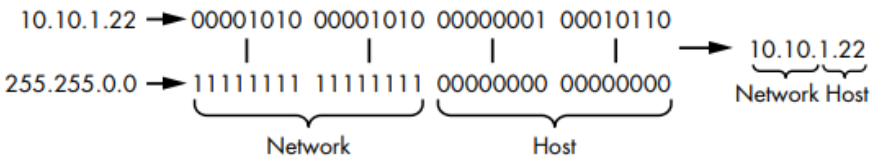


Figure 7-8: The netmask determines the allocation of the bits in an IP address.

As indicated in Figure 7-8, 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*. 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.

IPv4 Packet Structure

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'll find in a packet. The IP header is quite complex compared to the ARP packet we just examined; it includes a lot of extra functionality that helps IP do its job.

As shown in Figure 7-9, the IPv4 header has the following fields:

Version The version of IP being used (this will always be 4 for IPv4).

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 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 or seconds through routers.

Protocol Identifies the transport layer header that encapsulates the IPv4 header.

Header Checksum An error-detection mechanism used to verify that 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 Version 4 (IPv4)								
Offsets	Octet	0		1		2		3
Octet	Bit	0–3	4–7	8–15		16–18	19–23	24–31
0	0	Version	Header Length	Type of Service		Total Length		
4	32	Identification				Flags	Fragment Offset	
8	64	Time to Live		Protocol		Header Checksum		
12	96	Source IP Address						
16	128	Destination IP Address						
20	160	Options						
24+	192+	Data						

Figure 7-9: The IPv4 packet structure

Time to Live

ip_ttl_source.pcapng
ip_ttl_dest.pcapng

The *Time to Live (TTL)* value defines a period of time that can elapse or a maximum number of routers a packet can traverse before the packet is discarded for IPv4. 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 zero, and if the final destination of the packet is not on that network, the packet will be discarded (see Figure 7-10).

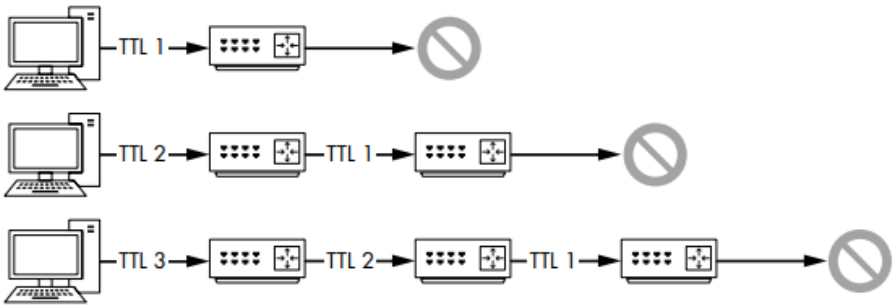


Figure 7-10: The TTL of a packet decreases every time it traverses a router.

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's being forwarded around a network in a never-ending loop.

An infinite loop can cause all sorts of issues, but it typically results in the crash of a program or an entire operating system. 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 denial of service condition occurred. To prevent this, TTL was created.

Let's look at an example of this in Wireshark. The file *ip_ttl_source.pcapng* contains two ICMP packets. ICMP (discussed later in this chapter) uses IP to deliver packets, as we can see by expanding the IP header section in the Packet Details pane (see Figure 7-11).

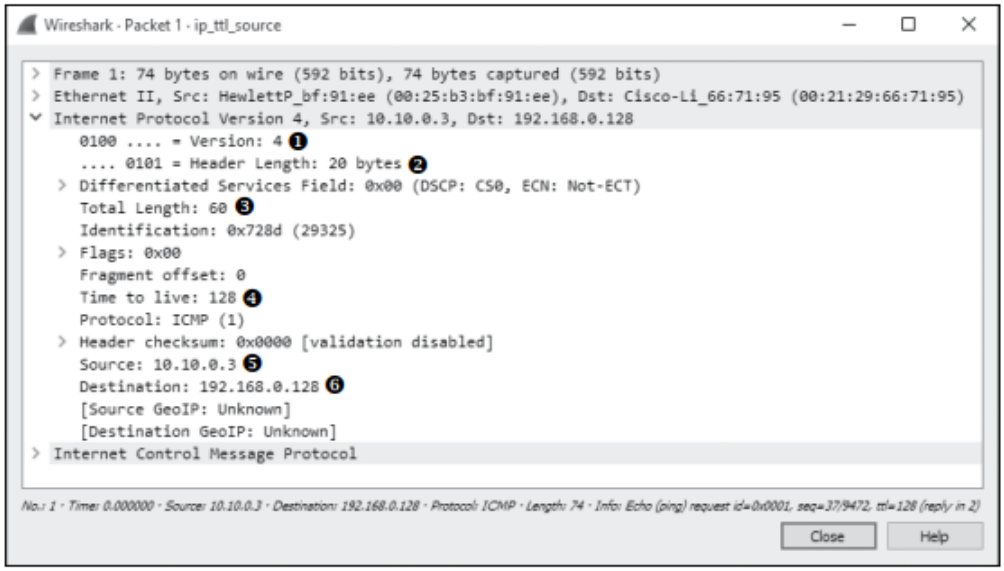


Figure 7-11: The IP header of the source packet

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.pcapng*. 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 7-12).

You should immediately notice that the TTL value is 127 ❶, 1 less than the original TTL of 128. Without even knowing the architecture of the network, we can conclude that one router separates these devices and thus the passage through that router reduced the TTL value by 1.

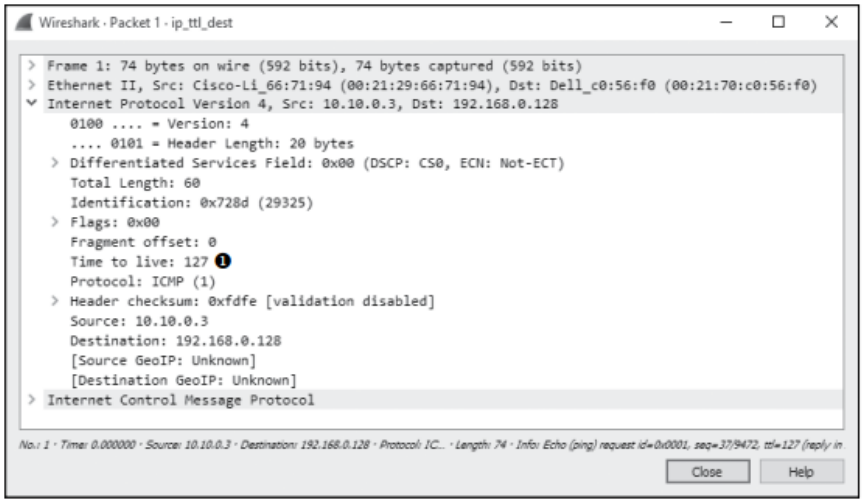


Figure 7-12: The IP header shows us that the TTL has been decremented by 1.

IP Fragmentation

ip_frag_source .pcapng

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 this layer 2 protocol. In most cases, the layer 2 data link protocol in use is Ethernet. Ethernet has a default MTU of 1,500, 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 packet 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 IP 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.pcapng* was taken from a computer with the address 10.10.0.3, transmitting a ping request to a device with the 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 7-13).

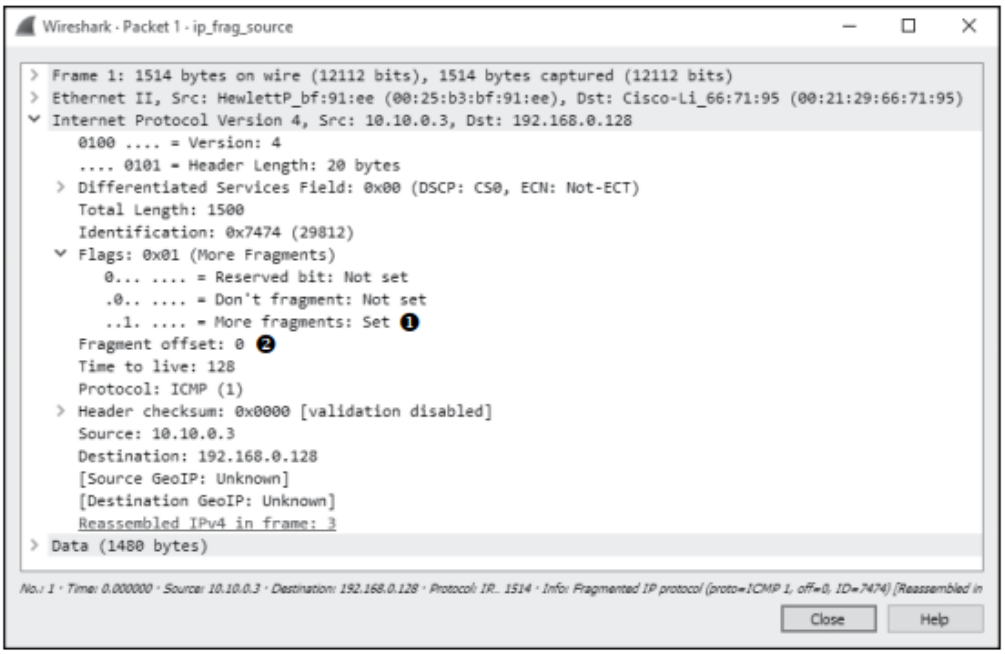


Figure 7-13: More fragments and Fragment offset values can indicate a fragmented packet.

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

The IP header of the second packet (see Figure 7-14) 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.

The third packet (see Figure 7-15) does not have the More fragments flag set ❷, which marks it as the last fragment in the data stream, and the Fragment offset is set to 2960 ❸, 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 ❶.

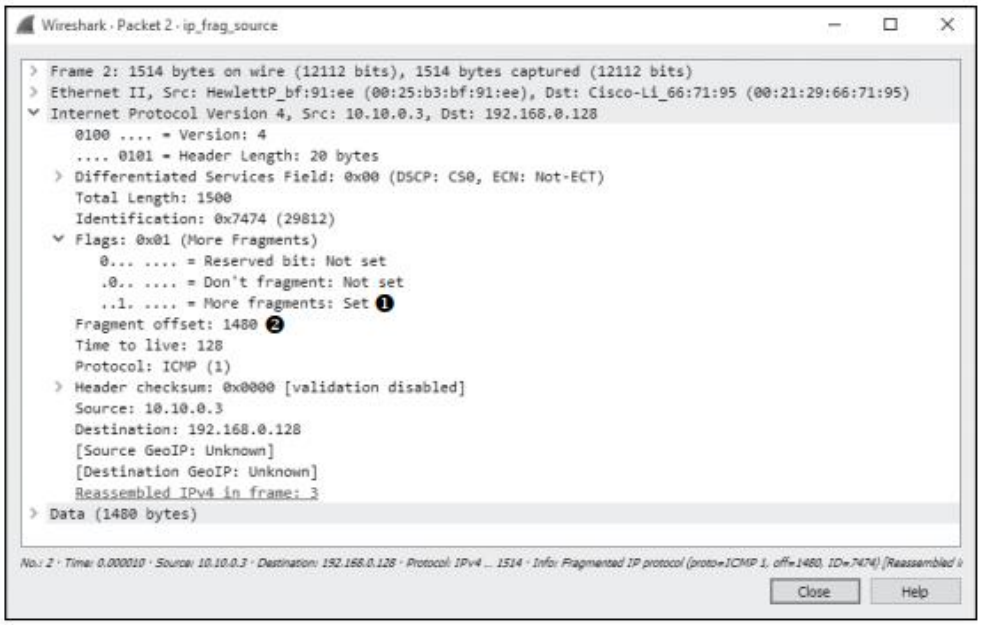


Figure 7-14: The Fragment offset value increases based on the size of the packets.

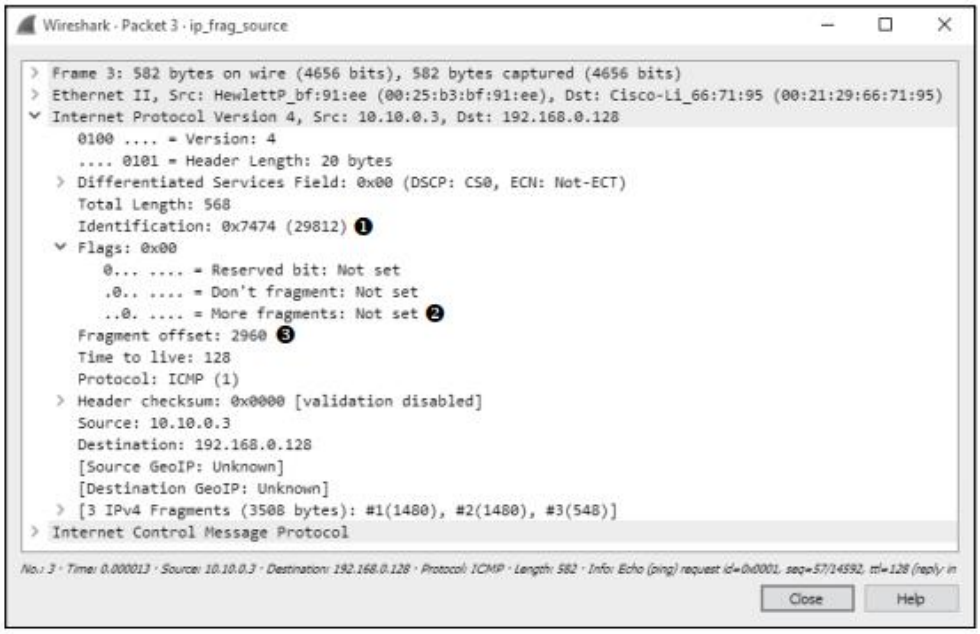


Figure 7-15: More fragments is not set, indicating that this fragment is the last.

While it isn't as common to see fragmented packets on a network as it used to be, understanding why packets are fragmented is useful so that when you do encounter them, you can diagnose issues or spot missing fragments.

Internet Protocol Version 6 (IPv6)

When the IPv4 specification was written, nobody had any idea that we would eventually have the number of internet-connected devices that exist today. The maximum IPv4 addressable space was limited to just south of 4.3 billion addresses. The actual amount of addressable space shrinks even further when you subtract ranges reserved for special uses such as testing, broadcast traffic, and RFC1918 internal addresses. While several efforts were made to delay the exhaustion of IPv4 addresses, ultimately the only way to address this limitation was to develop a new version of the IP specification.

Thus, the IPv6 specification was created, with its first version released in 1998 as RFC 2460. This version provided several performance enhancements, including a much larger address space. In this section, we'll look at the IPv6 packet structure and discuss how IPv6 communications differ from those of its predecessor.

IPv6 Addresses

IPv4 addresses were limited to 32 bits, a length that provided an addressable space measured in the billions. IPv6 addresses are 128 bit, providing an addressable space measured in undecillions (a trillion trillion trillion). That's quite an upgrade!

Since IPv6 addresses are 128 bits, they are unwieldy to manage in binary form. Most of the time, an IPv6 address is written in eight groups of 2 bytes in hexadecimal notation, with each group separated by a colon. For example, a very simple IPv6 address looks like this:

```
1111:aaaa:2222:bbbb:3333:cccc:4444:dddd
```

Your first thought is probably the same one many have who are used to remembering IPv4 addresses: IPv6 addresses are virtually impossible to memorize. That is an unfortunate trade-off for a much larger address space.

One feature of IPv6 address notation that will help in some cases is that some groups of zeroes can be collapsed. For example, consider the following IPv6 address:

```
1111:0000:2222:0000:3333:4444:5555:6666
```

You can collapse the grouping containing the zeroes completely so it isn't visible, like this:

```
1111::2222:0000:3333:4444:5555:6666
```

However, you can only collapse a single group of zeroes, so the following address would be invalid:

```
1111::2222::3333:4444:5555:6666
```

Another consideration is that leading zeroes can be dropped from IPv6 addresses. Consider this example in which there are zeroes in front of the fourth, fifth, and six groups:

```
1111:0000:2222:0333:0044:0005:ffff:ffff
```

You could represent the address more efficiently like this:

```
1111::2222:333:44:5:ffff:ffff
```

This isn't quite as easy to use as an IPv4 address, but it's a lot easier to deal with than the longer notation.

An IPv6 address has a network portion and a host portion, often called a *network prefix* and *interface identifier*, respectively. The distribution of these fields varies depending on the classification of the IPv6 communication. IPv6 traffic is broken down into three classifications: unicast, multicast, or anycast. In most cases, you'll probably be working with link-local unicast traffic, which is communication from one device to another inside a network. The format of a link-local unicast IPv6 address is shown in Figure 7-16.

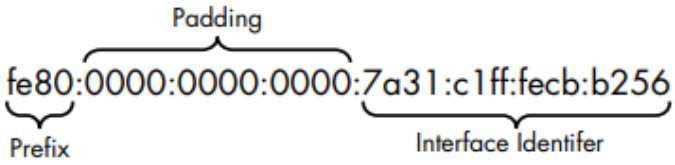


Figure 7-16: The parts of an IPv6 link-local unicast address

Link-local addresses are used when communication is intended for another device within the same network. A link-local address can be identified by having its most significant 10 bits set to 1111111010 and the next 54 bits set to all zeroes. Thus, you can spot a link-local address when the first half is fe80:0000:0000:0000.

The second half of a link-local IPv6 address is the interface ID portion, which uniquely identifies a network interface on an endpoint host. On Ethernet networks, this can be based on the MAC address of the interface. However, a MAC address is only 48 bits. To fill up the entire 64-bit space, the MAC address is cut in half, and the value 0xfffe is added between each half as padding to create a unique identifier. Lastly, the seventh bit of the first byte is inverted. That's a bit complex, but consider the interface ID in Figure 7-17. The original MAC address for the device represented by this ID was 78:31:c1:cb:b2:56. The bytes 0xfffe were added in the middle, and flipping the seventh bit of the first byte changed the *8* to an *a*.

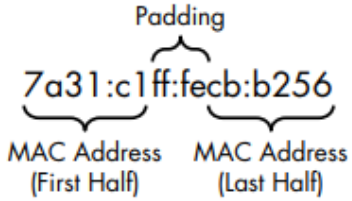


Figure 7-17: The interface ID utilizes an interface MAC address and padding.

IPv6 addresses can be represented with CIDR notation just like IPv4 addresses. In this example, 64 bits of addressable space are represented with a link-local address:

```
fe80:0000:0000:0000:/64
```

The composition of an IPv6 address changes when it is used with global unicast traffic that is routed over the public internet (see Figure 7-18). When used in this manner, a global unicast is identified by having its first 3 bits set to 001, followed by a 45-bit global routing prefix. The global routing prefix, which is assigned to organizations by the Internet Assigned Numbers Authority (IANA), is used to uniquely identify an organization's IP space. The next 16 bits are the subnet ID, which can be used for hierarchical addressing, similar to the netmask portion of an IPv4 address. The final 64 bits are used for the interface ID, just as with link-local unicast addresses. The routing prefix and subnet ID can vary in size.

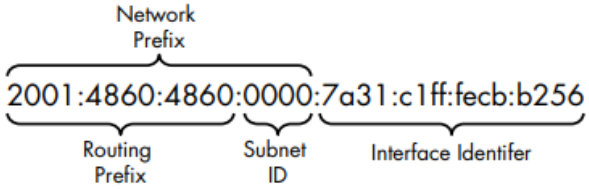


Figure 7-18: The parts of an IPv6 global unicast address

IPv6 provides a lot more efficiency than IPv4 in terms of routing packets to their destination and making effective use of address space. This efficiency is due to the larger range of addresses available and the use of link-local and global addressing along with unique host identifiers.

NOTE *It's easy for you to visually differentiate IPv6 and IPv4 addresses, but many programs cannot do so. If you need to specify an IPv6 address, some applications, such as browsers or command line utilities, require you to place square brackets around the address, like this: [1111::2222:333:44:5:ffff]. This requirement isn't always documented well and has been a source of frustration for many as they learn IPv6.*

IPv6 Packet Structure

http_ip4and6 .pcapng

The structure of the IPv6 header has grown to support more features, but it was also designed to be easier to parse. Instead of being variable in size with a header length field that needs to be checked to parse the header, headers are now a fixed 40 bytes. Additional options are provided via extension headers. The benefit is that most routers only need to process the 40-byte header to forward the packet along.

As shown in Figure 7-19, the IPv6 header has the following fields:

Version The version of IP being used (this is always 6 for IPv6).

Traffic Class Used to prioritize certain classes of traffic.

Internet Protocol Version 6 (IPv6)							
Offsets	Octet	0		1		2	3
Octet	Bit	0–3	4–7	8–11	12–15	16–23	24–31
0	0	Version	Traffic Class		Flow Label		
4	32	Payload Length				Next Header	Hop Limit
8	64	Source IP Address					
12	96						
16	128						
20	160						
24	192	Destination IP Address					
28	224						
32	256						
36	288						

Figure 7-19: The IPv6 packet structure

Flow Label Used by a source to label a set of packets belonging to the same flow. This field is typically used for quality of service (QoS) management and to ensure packets that are part of the same flow take the same path.

Payload Length The length of the data payload following the IPv6 header.

Next Header Identifies the layer 4 header that encapsulates the IPv6 header. This field replaces the Protocol field in IPv4.

Hop Limit Defines the lifetime of the packet, measured in hops through routers. This field replaces the TTL field in IPv4.

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

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

Let's compare an IPv4 and an IPv6 packet to examine a few of the differences by looking at *http_ip4and6.pcapng*. In this capture, a web server was configured to listen for both IPv4 and IPv6 connections on the same physical host. A single client configured with both IPv4 and IPv6 addresses browsed to a server using each of its addresses independently and downloaded the *index.php* page using HTTP via the curl application (Figure 7-20).

Upon opening the capture, you should readily see which packets belong to which conversation based on the addresses in the Source and Destination columns in the Packet List area. Packets 1 through 10 represent the IPv4 stream (stream 0), and packets 11 through 20 represent the IPv6 stream (stream 1). You can filter for each of these streams from the Conversations window or by entering `tcp.stream == 0` or `tcp.stream == 1` in the filter bar.

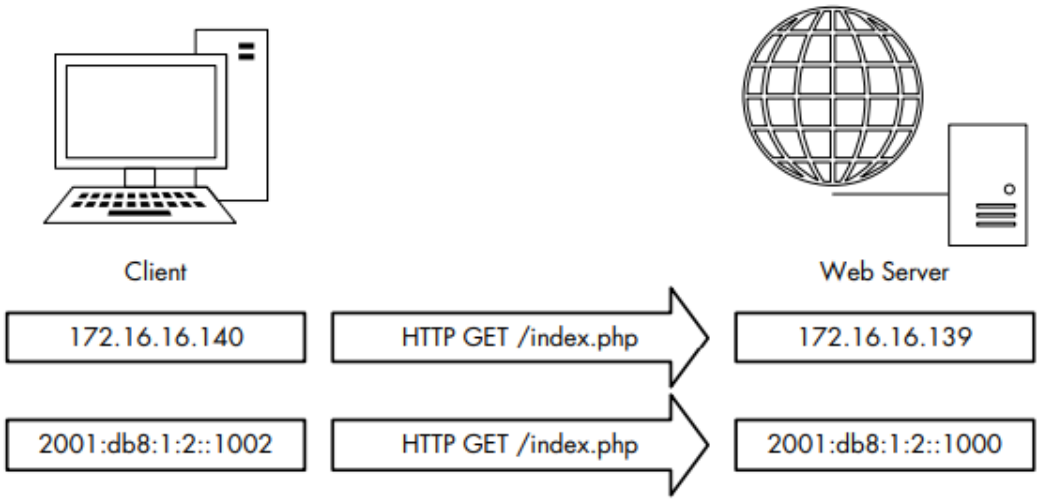


Figure 7-20: Connections between the same physical hosts using different IP versions

We'll cover HTTP, the protocol responsible for serving web pages on the internet, in depth in Chapter 8. In this example, just note that the business of serving web pages remains consistent regardless of which lower-layer network protocol is used. The same can be said of TCP, which also operates in a consistent manner. This is a prime example of encapsulation in action. Although IPv4 and IPv6 function differently, the protocols functioning at different layers are unaffected.

Figure 7-21 provides a side-by-side comparison of two packets with the same function—packets 1 and 11. Both packets are TCP SYN packets designed to initiate a connection from the client to the server. The Ethernet and TCP sections of these packets are nearly identical. However, the IP sections are completely different.

- The source and destination address formats are different ❻❿.
- The IPv4 packet is 74 bytes with a 60-byte total length ❶, which includes both the IPv4 header and payload and a 14-byte Ethernet header. The IPv6 packet is 96 bytes with a 40-byte IPv6 payload ❼ and a separate 40-byte IPv6 header along with the 14-byte Ethernet header. The IPv6 header is 40 bytes, double the IPv4 header's 20 bytes, to accommodate the larger address size.
- IPv4 identifies the protocol with the Protocol field ❹, whereas IPv6 identifies it with the Next header field (which can also be used to specify extension headers) ❽.
- IPv4 has a TTL field ❸, whereas IPv6 accomplishes the same functionality using the Hop limit field ❾.
- IPv4 includes a header checksum value ❺, while IPv6 does not.
- The IPv4 packet is not fragmented, but it still includes values for those options ❷. The IPv6 header doesn't contain this information because, if fragmentation were required, it would be implemented in an extension header.

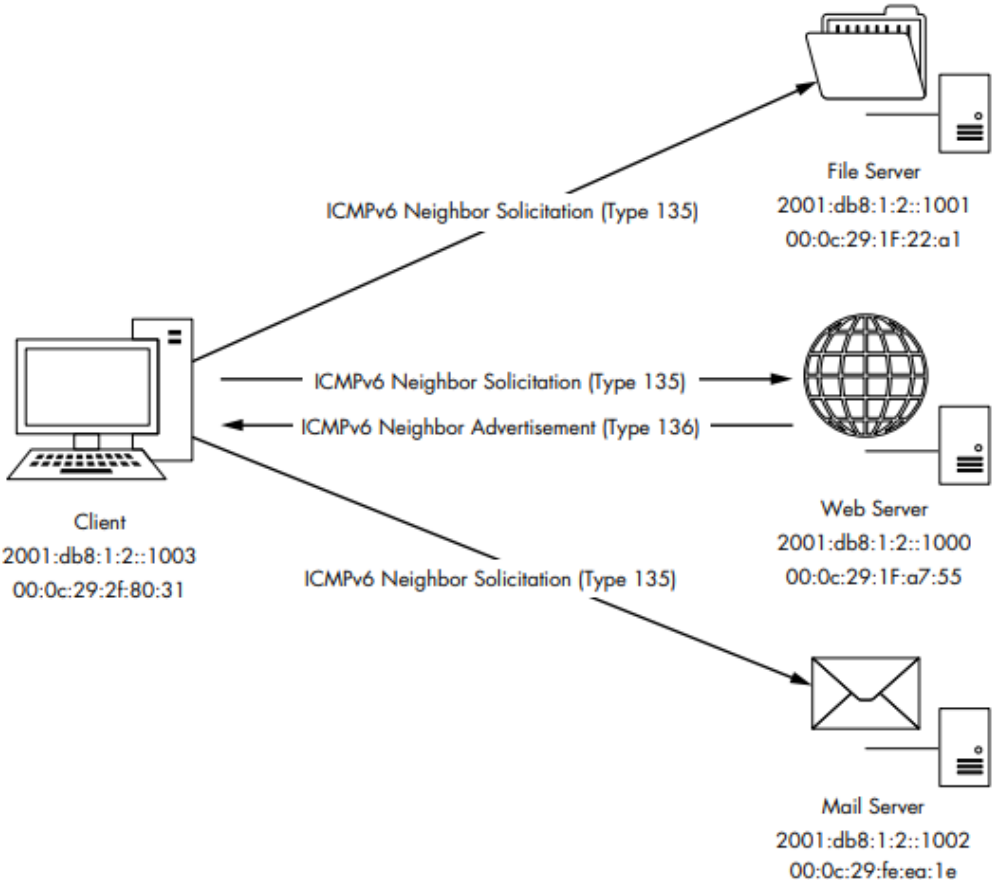
Neighbor Solicitation and ARP

icmpv6_neighbor_solicitation.pcapng

When we discussed the different classifications of traffic earlier, I listed unicast, multicast, and anycast but did not list broadcast traffic. IPv6 doesn't support broadcast traffic because broadcast is viewed as an inefficient mechanism for transmission. Because there is no broadcast, ARP can't be used for hosts to find each other on a network. So, how do IPv6 devices find each other?

The answer lies with a new feature called *neighbor solicitation*, a function of Neighbor Discovery Protocol (NDP), which utilizes ICMPv6 (discussed in the last section of this chapter) to do its legwork. To accomplish this task, ICMPv6 uses multicast, a type of communication in which only hosts that subscribe to a data stream will receive and process it. Multicast traffic can be identified quickly because it has its own reserved IP space (ff00::/8).

Although the address resolution process relies on a different protocol, it still uses a very simple request/response workflow. For example, let's consider a scenario in which a host with the IPv6 address 2001:db8:1:2::1003 wants to communicate with another host identified by the address 2001:db8:1:2::1000. Just as with IPv4, the source device must be able to determine the link-layer (MAC) address of the host it wants to communicate with, since this is intra-network communication. This process is described in Figure 7-22.



In this process, the host 2001:db8:1:2::1003 sends a Neighbor Solicitation (ICMPv6 type 135) packet to every device on the network via multicast, asking, "What is the MAC address for the device whose IP address is 2001:db8:1:2::1000? My MAC address is 00:0C:29:2f:80:31."

The device assigned that IPv6 address will receive this multicast transmission and respond to the originating host with a Neighbor Advertisement (ICMPv6 type 136) packet. This packet says, "Hi, my network address is 2001:db8:1:2::1000 and my MAC address is 00:0c:29:1f:a7:55." Once this message is received, communication can begin.

You can see this process in action in the capture file *icmpv6_neighbor_solicitation.pcapng*. This capture embodies the example we've just discussed in which 2001:db8:1:2::1003 wants to communicate with 2001:db8:1:2::1000. Look at the first packet and expand the ICMPv6 portion in the Packet Details window (Figure 7-23) to see that the packet is ICMP type 135 ❷ and was sent from 2001:db8:1:2::1003 to the multicast address ff02::1:ff00:1000 ❶. The source host provided the target IPv6 address it wanted to communicate with ❸, along with its own layer 2 MAC address ❹.

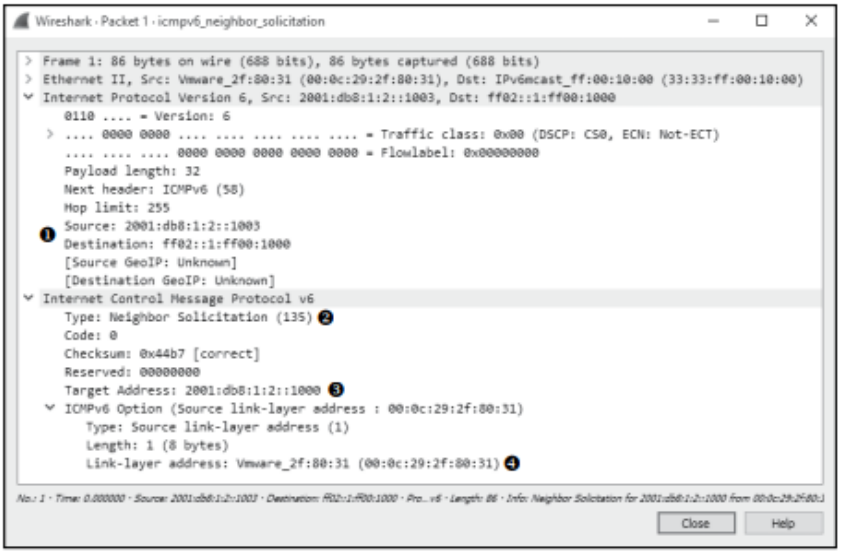


Figure 7-23: A neighbor solicitation packet

The response to the solicitation is found in the second packet in the capture file. Expanding the ICMPv6 portion of the Packet Details window (Figure 7-24) reveals this packet is ICMP type 136 ❷, was sent from 2001:db8:1:2::1000 back to 2001:db8:1:2::1003 ❶, and contains the MAC address 00:0c:29:1f:a7:55 associated with 2001:db8:1:2::1000 ❸.

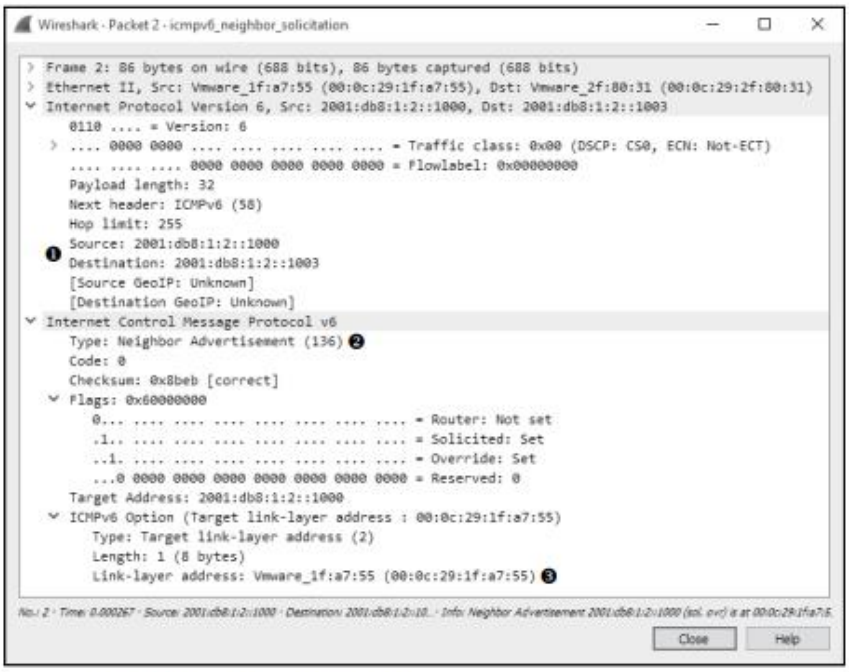


Figure 7-24: A neighbor advertisement packet

Upon completion of this process, 2001:db8:1:2::1003 and 2001:db8:1:2::1000 begin communicating normally with ICMPv6 echo request and reply packets, indicating the neighbor solicitation process and link-layer address resolution was successful.

IPv6 Fragmentation

ipv6_fragments.pcapng

Fragmentation support was built into the IPv4 header because it ensured packets could traverse all sorts of networks at a time when network MTUs varied wildly. In IPv6, fragmentation is used less, so the options supporting it are not included in the IPv6 header. A device transmitting IPv6 packets is expected to perform a process called *MTU discovery* to determine the maximum size of packets it can send before actually sending them. In the event that a router receives a packet that is too large for the MTU on the network it is forwarding to, it will drop the packet and return an ICMPv6 Packet Too Big (type 2) message to the originating host. Upon receipt, the originating host will attempt to resend the packet with a smaller MTU, if such action is supported by the upper-layer protocol. This process will repeat until a small enough MTU is reached or until the payload can be fragmented no more (Figure 7-25). A router will never be responsible for fragmenting packets on its own; the source device is responsible for determining an appropriate MTU for the transmission path and fragmenting appropriately.

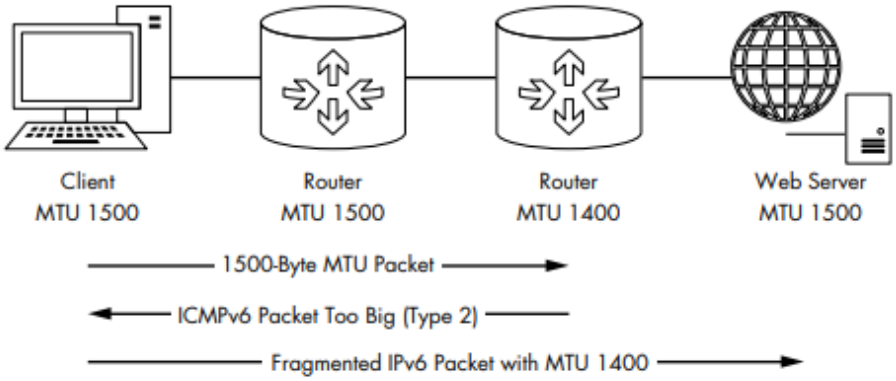


Figure 7-25: IPv6 MTU path discovery

If the upper-layer protocol being used in conjunction with IPv6 can't limit the size of the packet payload, then fragmentation must still be used. A fragmentation extension header can be added to the IPv6 packet to support this scenario. You will find a sample capture showing IPv6 fragmentation in the file named *ipv6_fragments.pcapng*.

Because the receiving device has a smaller MTU than the sending device, there are two fragmented packets to represent each ICMPv6 echo request and reply in the capture file. The fragmentation header from the first packet is shown in Figure 7-26.

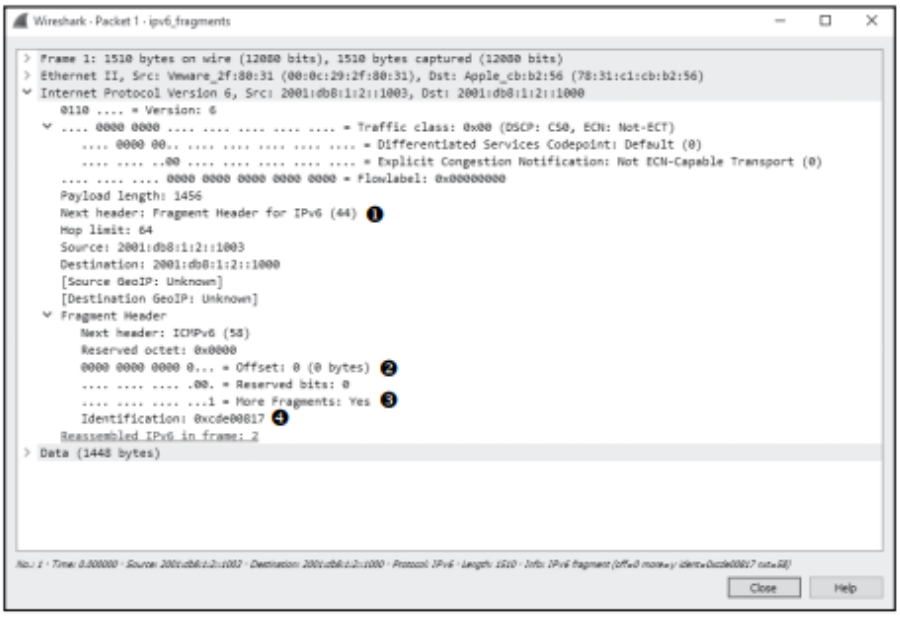


Figure 7-26: An IPv6 fragment header extension

The 8-byte extension header contains all the same fragmentation properties that are found in an IPv4 packet, such as a Fragment offset ❷, More Fragments flag ❸, and Identification field ❹. Instead of being present in every packet, it is only added to the end of packets requiring fragmentation. This more efficient process still allows the receiving system to reassemble the fragments appropriately. Additionally, if this extension header is present, the Next header field will point to the extension header rather than the encapsulating protocol ❶.

IPv6 Transitional Protocols

IPv6 addresses a very real problem, but its adoption has been slow because of the effort required to transition network infrastructure to it. To ease this transition, several protocols allow IPv6 communication to be tunneled across networks that support only IPv4 communication. In this respect, tunneling means that IPv6 communication is encapsulated inside of IPv4 communications just as other protocols may be encapsulated. Encapsulation is usually done in one of three ways:

Router to Router Uses a tunnel to encapsulate IPv6 traffic from the transmitting and receiving hosts on their networks over an IPv4 network. This method allows entire networks to communicate in IPv6 over intermediary IPv4 links.

Host to Router Uses encapsulation at the router level to transmit traffic from an IPv6 host over an IPv4 network. This method allows an individual host to communicate in IPv6 to an IPv6 network when the host resides on an IPv4-only network.

Host to Host Uses a tunnel between two endpoints to encapsulate IPv6 traffic between IPv4- or IPv6-capable hosts. This method allows IPv6 endpoints to communicate directly across an IPv4 network.

While this book won't cover transitional protocols in depth, it's helpful to be aware of their existence in case you ever need to investigate them while performing analysis at the packet level. The following are a few common protocols:

6to4 Also known as *IPv6 over IPv4*, this transitional protocol allows IPv6 packets to be transmitted across an IPv4 network. This protocol supports relays and routers to provide router-to-router, host-to-router, and host-to-host IPv6 communication.

Teredo This protocol, used for IPv6 unicast communications over an IPv4 network using NAT (network address translation), works by sending IPv6 packets over IPv4 encapsulated in the UDP transport protocol.

ISATAP This intrasite protocol allows communication between IPv4- and IPv6-only devices within a network in a host-to-host manner.

Internet Control Message Protocol (ICMP)

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.

ICMP Packet Structure

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 7-27, 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 varies depending on the Type and Code fields

Internet Control Message Protocol (ICMP)					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	Type	Code	Checksum	
4+	32+	Variable			

Figure 7-27: 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 the packet's classification and the Code field 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 also to 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.pcapng

ICMP's biggest claim to fame is the ping utility. *Ping* is used to test for connectivity to a device. While ping itself isn't a part of the ICMP spec, it utilizes ICMP to achieve its core functionality.

To use ping, enter `ping` *`ipaddress`* at the command prompt, replacing *`ipaddress`* 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 7-28 shows four successful replies that display their size; round trip time (or RTT), which is the time it takes for the packet to arrive and a response to be received; 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.

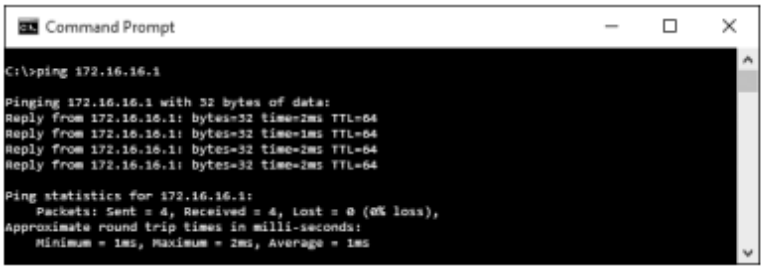


Figure 7-28: The `ping` command being used to test connectivity

Basically, the `ping` command sends one packet at a time to a device and listens for a reply to determine whether there is connectivity to that device, as shown in Figure 7-29.

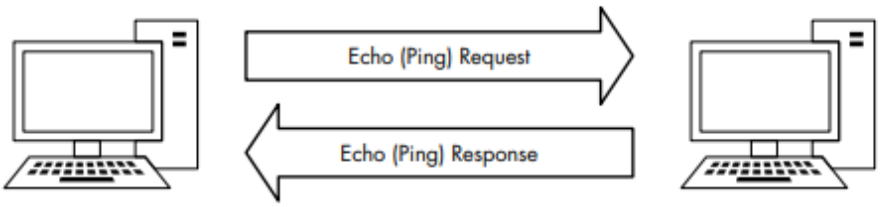


Figure 7-29: The `ping` command involves only two steps.

NOTE *Although ping has long been the bread and butter of IT, its results can be a bit deceiving when 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 whether a host is accessible might be deterred, but troubleshooting is also 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.pcapng* demonstrate what happens when you run ping.

The first packet (see Figure 7-30) 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 and the code is 0 ❸, indicating an echo request. (Wireshark should tell you what the displayed type/code 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 there is a random text string in the variable portion of the ICMP packet.

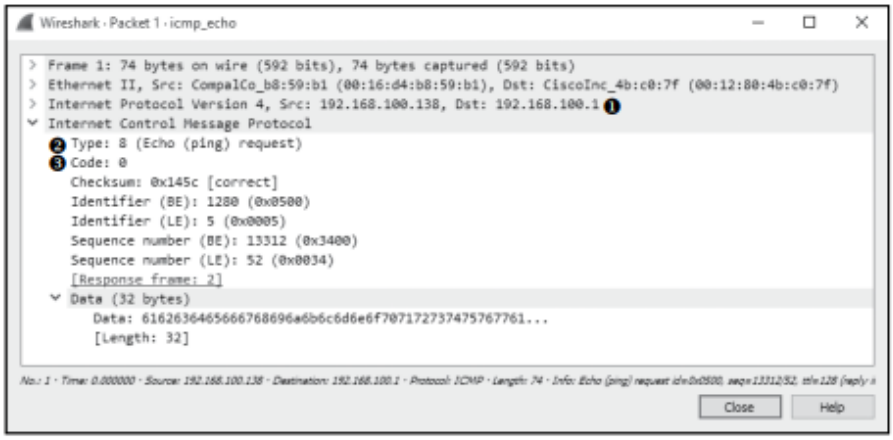


Figure 7-30: An ICMP echo request packet

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

The second packet in this sequence is the reply to our request (see Figure 7-31). The ICMP portion of the packet is type 0 ❶ and code 0 ❷, indicating that this is an echo reply. Because the sequence number and identifier in the second packet match those of the first ❸, we know that this echo reply matches the echo request in the previous packet. Wireshark displays the values of these fields in big-endian (BE) and little-endian (LE) format. In other words, it represents the data in a different order based on how a particular endpoint might process the data. 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.

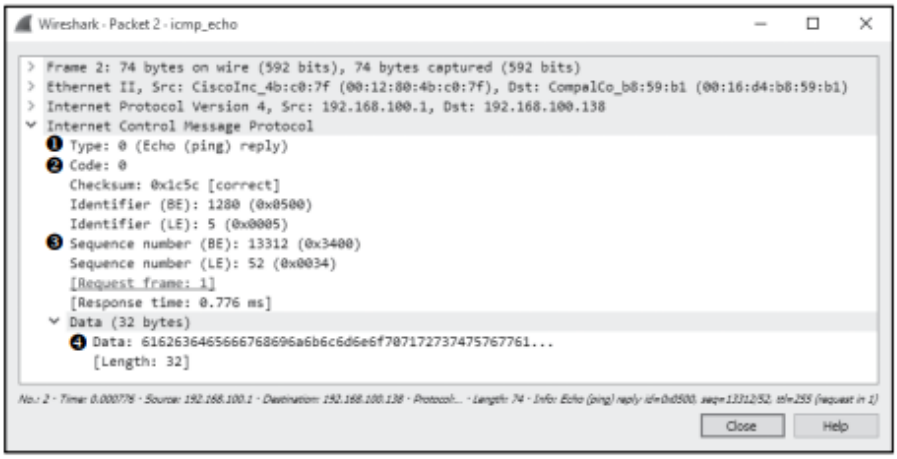


Figure 7-31: An ICMP echo reply packet

Note that you can use variations of the ping command to increase the size of the data padding in echo requests, which forces packets to be fragmented for various types of network troubleshooting. This may be necessary 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 .pcapng

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. Thus, it is 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.pcapng* is pretty similar to the echo request we looked at in the previous section (see Figure 7-32).

In this capture, the packets were generated by running the command `tracert 4.2.2.1`. To use traceroute on Windows, enter `tracert` *`ipaddress`* at the command prompt, replacing *`ipaddress`* with the actual IP address of a device whose path you want to discover. To use traceroute on Linux or Mac, use the command `traceroute` *`ipaddress`*.

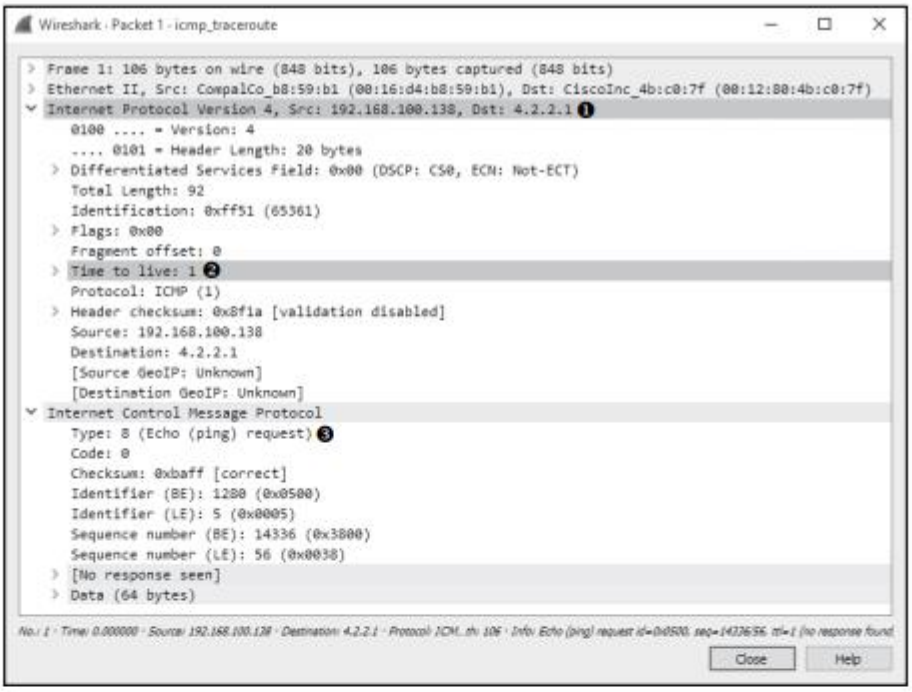


Figure 7-32: An ICMP echo request packet with a TTL value of 1

At first glance, this packet appears to be a simple echo request ❸ from 192.168.100.138 to 4.2.2.1 ❶, 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 something odd: the packet's TTL value is set to 1 ❷, meaning 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 7-33). 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 is type 11 ❶ and code 0 ❷, data that tells us that the destination was unreachable because the packet's TTL was exceeded during transit.

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 were sent in the original echo request. This information can prove very useful for troubleshooting.

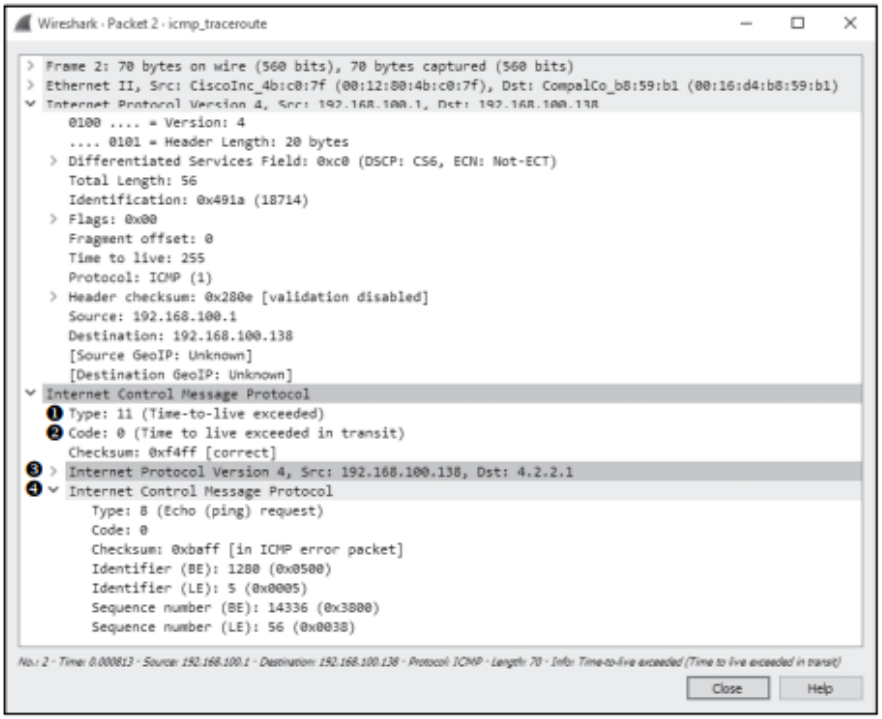


Figure 7-33: An ICMP response from the first router along the path

This process of sending packets with a TTL value of 1 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 1, until the destination 4.2.2.1 is reached. Right before that happens, however, you'll see in Figure 7-34 that the request on line 8 timed out. How can a request along the path time out and the process still complete successfully? Typically, this happens when a router is configured to not respond to ICMP requests. The router still receives the request and passes the data forward to the next router, which is why we're able to see the next hop on line 9 in Figure 7-34. It just didn't generate the ICMP time to live exceeded packet as the other hops did. With no response, tracert assumes the request has timed out and moves on to the next one.

To sum up, this traceroute process has communicated with each router along the path, building a map of the route to the destination. An example map is shown in Figure 7-34.

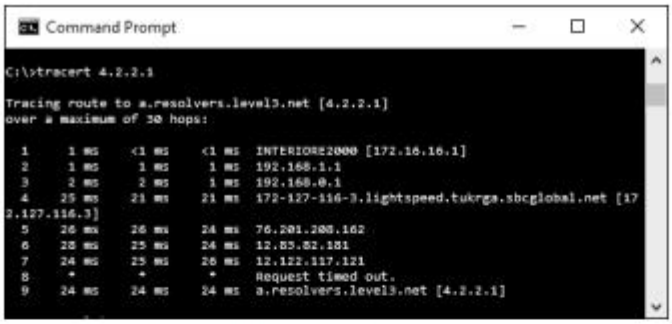


Figure 7-34: A sample output from the traceroute utility

NOTE *The discussion here on traceroute is generally Windows focused because this utility 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.*

ICMP Version 6 (ICMPv6)

The updated version of IP relies heavily on ICMP for functions such as neighbor solicitation and path discovery, as demonstrated in earlier examples. *ICMPv6* was established with RFC 4443 to support the feature set needed for IPv6, along with additional enhancements. We don't cover ICMPv6 separately in this book because it uses the same packet structure as do ICMP packets.

ICMPv6 packets are generally classified as either error messages or informational messages. You can find a full list of the available types and codes from IANA here: *http://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xhtml*.

This chapter has introduced you to a few of the most important protocols you will examine during the process of packet analysis. ARP, IP, and ICMP are at the foundation of all network communications, and they are critical to just about every daily task you will perform. In Chapter 8, we will look at common transport layer protocols, TCP and UDP.

TRANSPORT LAYER PROTOCOL

Transmission Control Protocol (TCP)

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, handles data sequencing and error recovery, and ultimately ensures that data gets where it's supposed to go. TCP is considered a *connection-oriented protocol* because it establishes a formal connection before transmitting data, tracks packet delivery, and usually attempts to formally close communication channels when transmission is complete. Many commonly used application-layer protocols rely on TCP and IP to deliver packets to their final destination.

TCP Packet Structure

TCP provides a great deal of functionality, as reflected in the complexity of its header. As shown in Figure 8-1, 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.

<table>
<tr><th colspan="7">Transmission Control Protocol (TCP)</th></tr>
<tr><td>Offsets</td><td>Octet</td><td colspan="2">0</td><td colspan="2">1</td><td>2</td><td>3</td></tr>
<tr><td>Octet</td><td>Bit</td><td>0–3</td><td>4–7</td><td colspan="2">8–15</td><td>16–23</td><td>24–31</td></tr>
<tr><td>0</td><td>0</td><td colspan="3">Source Port</td><td colspan="3">Destination Port</td></tr>
<tr><td>4</td><td>32</td><td colspan="6">Sequence Number</td></tr>
<tr><td>8</td><td>64</td><td colspan="6">Acknowledgment Number</td></tr>
<tr><td>12</td><td>96</td><td>Data Offset</td><td>Reserved</td><td colspan="2">Flags</td><td colspan="2">Window Size</td></tr>
<tr><td>16</td><td>128</td><td colspan="4">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td>20+</td><td>160+</td><td colspan="6">Options</td></tr>
</table>

Figure 8-1: The TCP header

TCP Ports

tcp_ports.pcapng

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 the other party 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.

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 isn't incredibly important and can be selected randomly. The remote server will simply determine the port to communicate with from the original packet it's sent (see Figure 8-2).

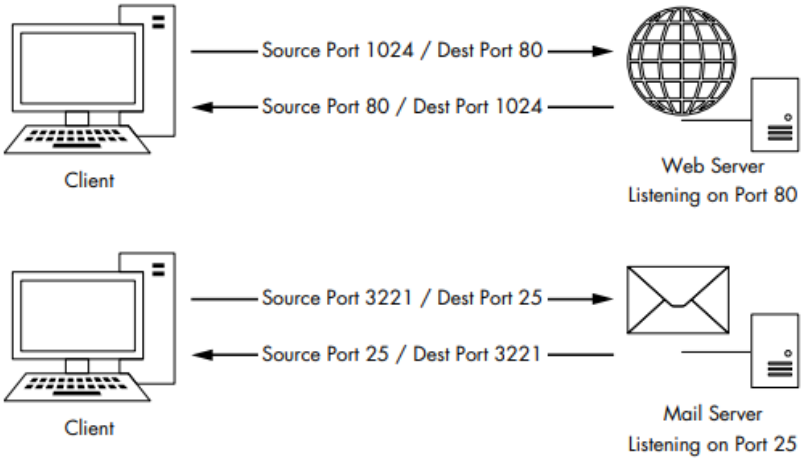


Figure 8-2: TCP uses ports to transmit data.

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

- The *system port group* (also known as the standard port or well-known port group) is from 1 through 1023 (ignoring 0 because it's reserved). Well-known, established services generally use ports that lie within the system 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 TCP packets and identify the port numbers they are using by opening the file *tcp_ports.pcapng*. In this file, we have the HTTP communication of a client browsing to two websites. As mentioned previously, HTTP uses TCP for communication, making it a great example of standard TCP traffic.

In the first packet in this file (see Figure 8-3), 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 ❶, 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 system port, port 80 ❷, the standard port used for web servers using HTTP.

Figure 8-3: 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 system ports are primarily the ones with labeled common uses, many ephemeral ports have commonly used services associated with them. The labeling of these ports can be confusing, so it's typically best to disable it by turning off transport name resolution. To do this, go to **Edit ▸ Preferences ▸ Name Resolution** and uncheck Enable Transport Name Resolution. If you wish to leave this option enabled but want to change how Wireshark identifies a certain port, you can do so by modifying the *services* file located in the Wireshark system directory. The contents of this file are based on the IANA common ports listing (see "Using a Custom hosts File" on page 86 for an example of how to edit a name resolution file).

The second packet is sent back from 212.58.226.142 to 172.16.16.128 (see Figure 8-4). As with the IP addresses, the source and destination ports are now also switched ❶.

In most cases, 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.

This sample capture file includes one more communication stream. See if you can locate the port numbers it uses for communication.

Figure 8-4: Switching the source and destination port numbers for reverse communication

NOTE *As we progress through this book, you'll learn more about the ports associated with common protocols and services. Eventually, you'll be able to profile services and devices by the ports they use. For a comprehensive list of common ports, look at the* services *file located in the Wireshark system directory.*

The TCP Three-Way Handshake

tcp_handshake .pcapng

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

- It allows the transmitting host to ensure that the recipient host is up and able to communicate.
- It lets the transmitting host check that the recipient is listening on the port the transmitting host is attempting to communicate on.
- 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 steps, as shown in Figure 8-5. 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.

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.pcapng*. 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 this, choose **Edit ▸ Preferences**, expand the **Protocols** heading, and choose **TCP**. In the window, uncheck the box next to **Relative Sequence Numbers** and click **OK**.

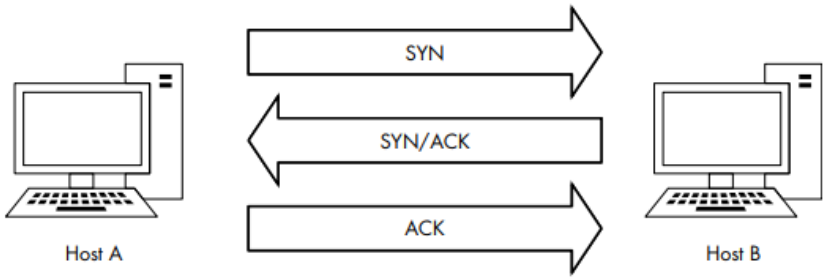


Figure 8-5: The TCP three-way handshake

The first packet in this capture represents our initial SYN packet ❷ (see Figure 8-6). 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.

The second packet in the handshake is the SYN/ACK response ❸ from 212.58.226.142 (see Figure 8-7). This packet also contains this host's initial sequence number (233779340) ❶ and an acknowledgment number (3691127925) ❷. The acknowledgment number shown here is 1 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.

The final packet is the ACK ❷ packet sent from 172.16.16.128 (see Figure 8-8). This packet, as expected, contains the sequence number 3691127925 ❶ as defined in the previous packet's Acknowledgment number field.

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

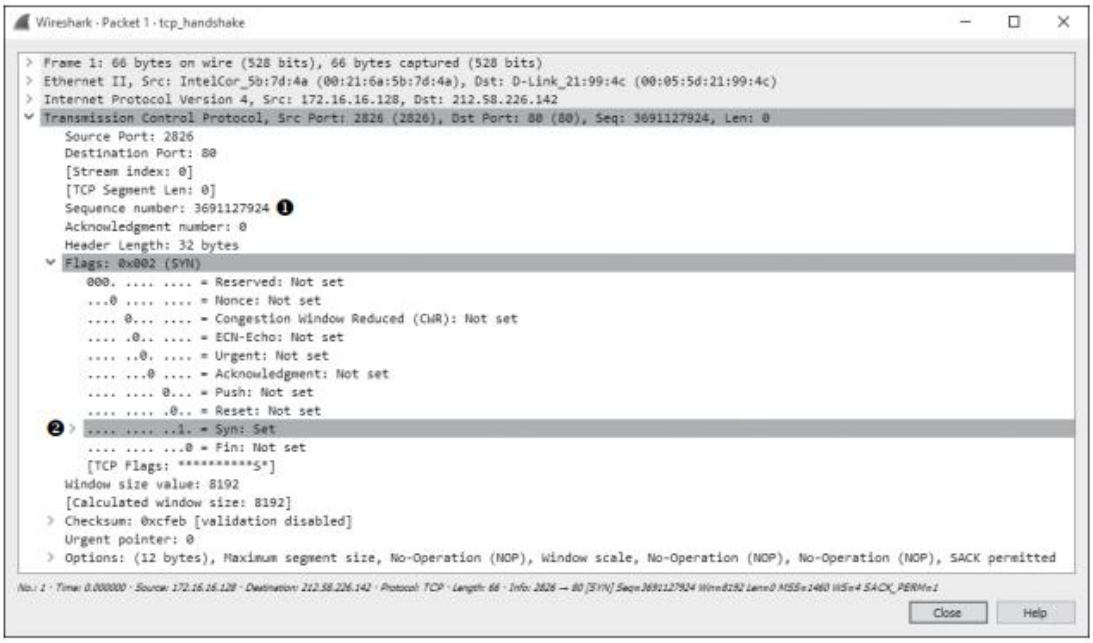


Figure 8-6: The initial SYN packet

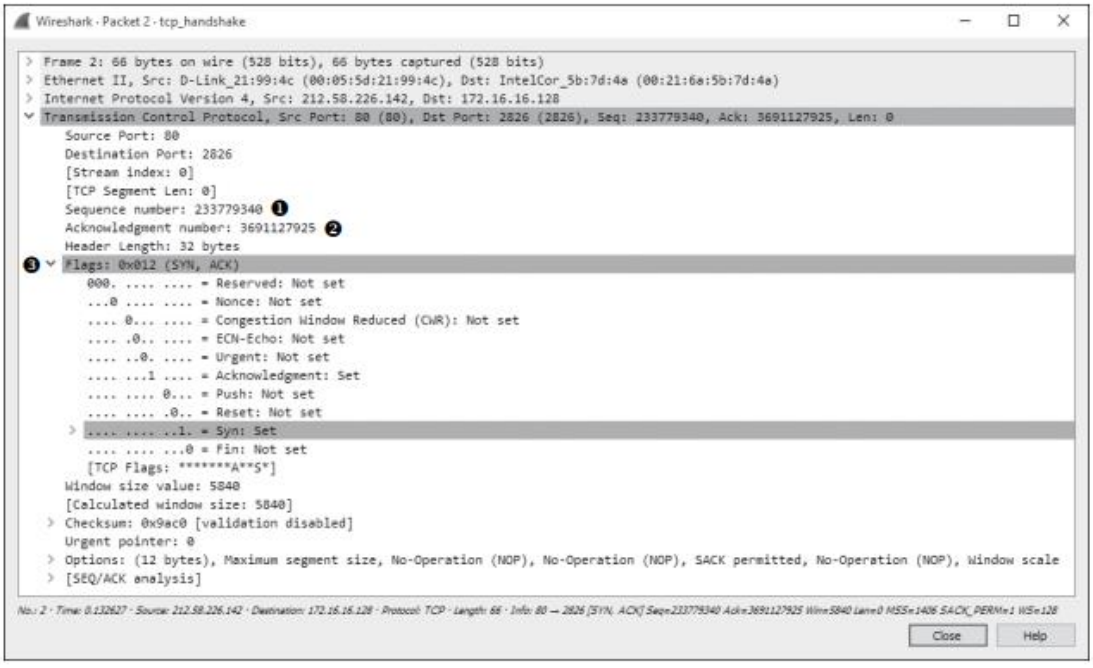


Figure 8-7: The SYN/ACK response

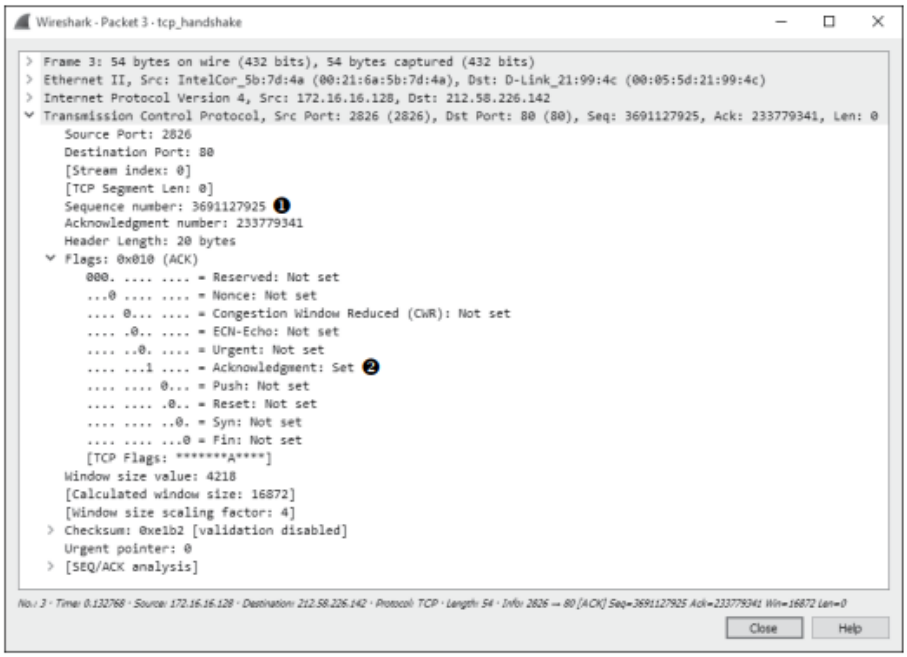


Figure 8-8: The final ACK

TCP Teardown

tcp_teardown.pcapng

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. This process is illustrated in Figure 8-9.

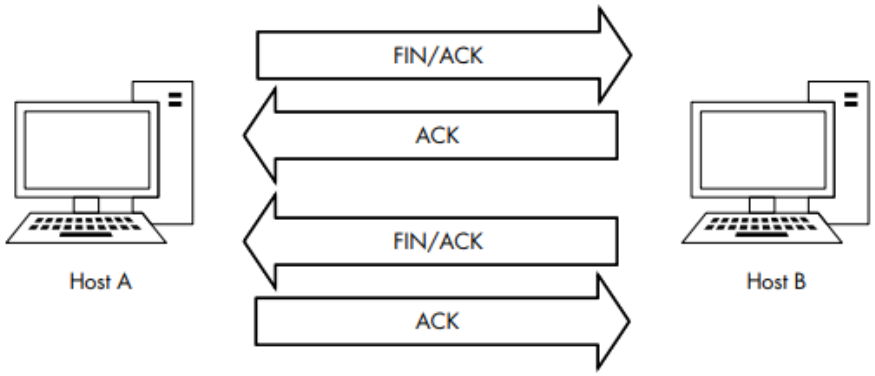


Figure 8-9: The TCP teardown process

To view this process in Wireshark, open the file *tcp_teardown.pcapng*. Beginning with the first packet in the sequence (see Figure 8-10), you can see that the device at 67.228.110.120 initiates teardown by sending a packet with the FIN and ACK flags set ❶.

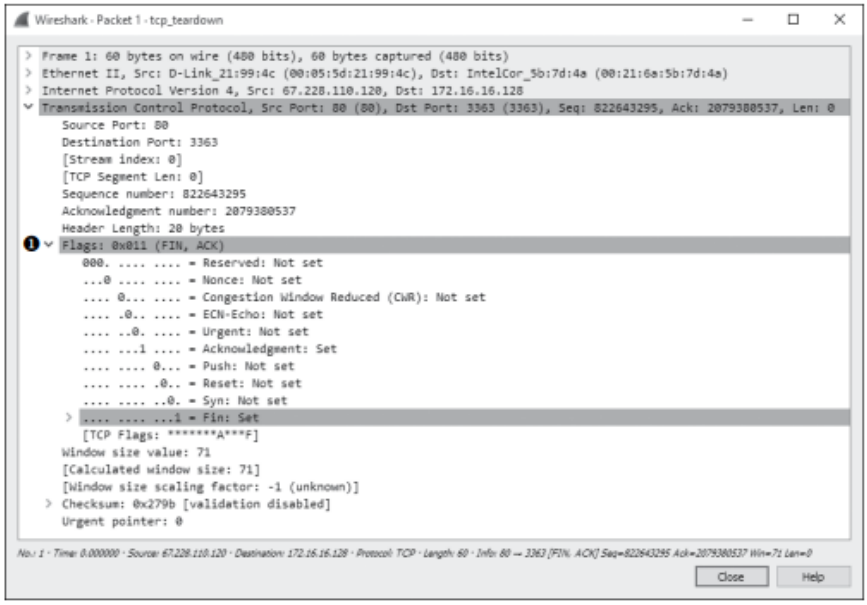


Figure 8-10: The FIN/ACK packet initiates the teardown process.

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. If they need to begin communicating again, they will have to complete a new TCP handshake.

TCP Resets

tcp_refuseconnection .pcapng

In an ideal world, every connection would end gracefully with a TCP teardown. In reality, connections often end abruptly. For example, a host may be misconfigured, or a potential attacker may perform a port scan. In these cases, when a packet is sent to a device that is not willing to accept it, a TCP packet with the RST flag set may be sent. The RST flag is used to indicate that a connection was closed abruptly or to refuse a connection attempt.

The file *tcp_refuseconnection.pcapng* displays an example of network traffic that includes an 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's a Cisco router with no web interface configured. There is no service configured to accept connections on that port. 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 8-11 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 ❶, and no further communication follows.

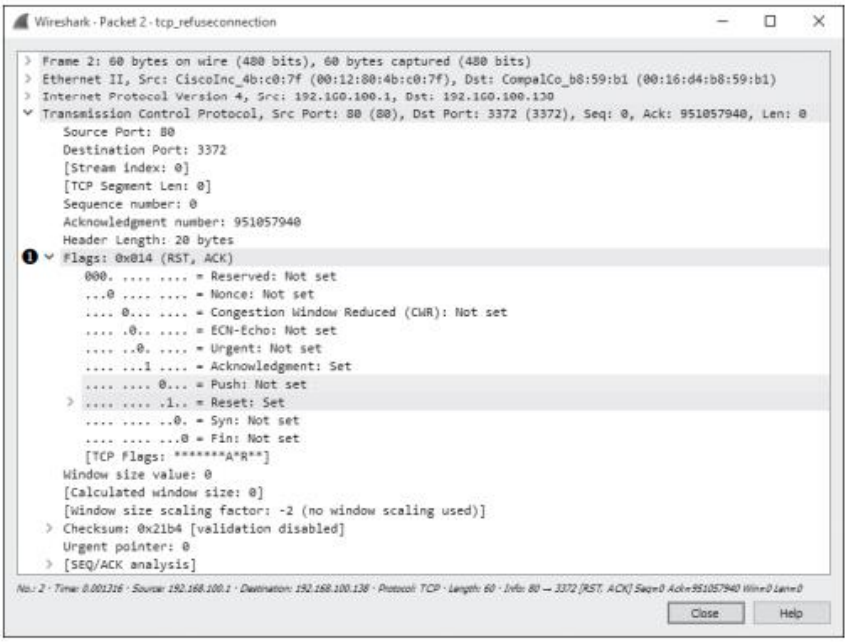


Figure 8-11: The RST and ACK flags signify the end of communication.

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.

User Datagram Protocol (UDP)

udp_dnsrequest.pcapng

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 doesn't 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.

UDP Packet Structure

udp_dnsrequest .pcapng

The UDP header is much smaller and simpler than the TCP header. As shown in Figure 8-12, 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 (UDP)					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	Source Port		Destination Port	
4	32	Packet Length		Checksum	

Figure 8-12: The UDP header

The file *udp_dnsrequest.pcapng* 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 8-13).

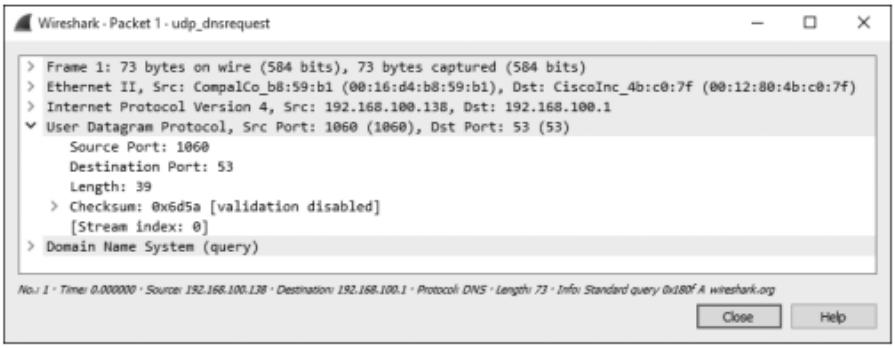


Figure 8-13: The contents of a UDP packet are very simple.

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. This is in contrast to TCP, which utilizes a formal connection setup and teardown, and has features in place to validate that packets were transmitted successfully.

This chapter has introduced you to the transport layer protocols TCP and UDP. Not unlike network protocols, TCP and UDP are at the core of most of your daily communication, and the ability to analyze them effectively is critical to becoming an effective packet analyst. In Chapter 9, we will look at common application-layer protocols.