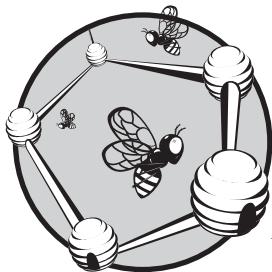


# 26

## IPV6 DATAGRAM ENCAPSULATION AND FORMATTING



Delivery of data over Internet Protocol version 6 (IPv6) internetworks is accomplished by encapsulating higher-layer data into IPv6 datagrams. These serve the same general purpose for IPv6 as IPv4 datagrams do in the older version of the protocol. However, they have been redesigned as part of the overall changes represented by IPv6. IPv6 datagrams have a flexible structure, and their format better matches the needs of current IP networks.

In this chapter, I take a look at the format used for IPv6 datagrams. I begin with an overview of the general structure of IPv6 datagrams, describe the major changes, and show how main and extension headers are arranged in the datagram. I then describe the format of the main header, and define and describe the various extension header types. I conclude with a brief explanation of IPv6 options and how they are implemented.

**BACKGROUND INFORMATION** *This chapter assumes basic understanding of IPv6 addressing concepts (see the previous chapter) and general familiarity with the IPv4 datagram format (described in Chapter 21).*

## IPv6 Datagram Overview and General Structure

The method by which IPv6 encapsulates data received from higher-layer protocols for transmission across the internetwork is basically the same as the one used by IPv4. The data received from the transport or higher layers is made the payload of an IPv6 datagram, which has one or more headers that control the delivery of the message. These headers provide information to routers in order to enable them to move the datagram across the network. They also provide information to hosts so they can tell which datagrams they are intended to receive.

While the basic use of datagrams hasn't changed since IPv4, many modifications were made to their structure and format when IPv6 was created. This was done partly out of necessity: IPv6 addresses are different from IPv4 addresses, and IP addresses go in the datagram header. The increase in the size of IP addresses from 32 bits to 128 bits adds a whopping extra 192 bits, or 24 bytes, of information to the header. This led to an effort to remove fields that weren't strictly necessary in order to compensate for the necessary increase in size. However, changes were also made to IPv6 datagrams to add features to them and to make them better suit the needs of modern internetworking.

The following is a list of the most significant overall changes to datagrams in IPv6:

**Multiple-Header Structure** Rather than a single header that contains all fields for the datagram (possibly including options), the IPv6 datagram supports a main header and then extension headers for additional information when needed.

**Streamlined Header Format** Several fields have been removed from the main header to reduce its size and increase efficiency. Only the fields that are truly required for pretty much *all* datagrams remain in the main header; others are put into extension headers and used as needed. Some were removed because they were no longer needed, such as the Internet Header Length field. The IPv6 header is of fixed length. I'll examine this more thoroughly in a moment.

**Renamed Fields** Some fields have been renamed to better reflect their actual use in modern networks.

**Greater Flexibility** The extension headers allow for a great deal of extra information that will accompany datagrams when needed. Options are also supported in IPv6.

**Elimination of Checksum Calculation** In IPv6, a checksum is no longer computed on the header. This saves both the calculation time spent by every device that packages IP datagrams (hosts and routers) and the space the checksum field took up in the IPv4 header.

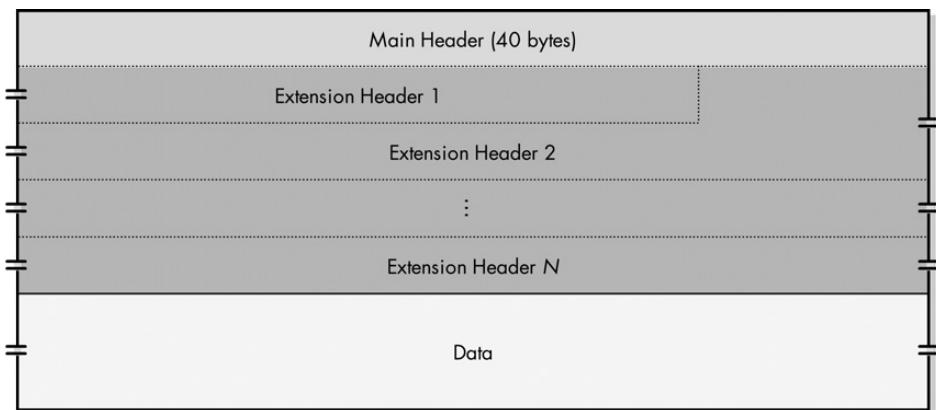
**Improved Quality of Service Support** A new field, the Flow Label, is defined to help support the prioritization of traffic.

**KEY CONCEPT** IPv6 datagrams use a general structure that begins with a mandatory main header that's 40 bytes in length, followed by optional extension headers, and then a variable-length Data area. This structure was created to allow the main header to be streamlined, while allowing devices to add extra information to datagrams when needed.

As I mentioned previously, IPv6 datagrams now include a main header format (which has no official name in the standards; it's just "the header") and zero or more extension headers. The overall structure of an IPv6 datagram is shown in Table 26-1 and illustrated in Figure 26-1.

**Table 26-1:** IPv6 General Datagram Structure

Component	Number of Components per Datagram	Size (Bytes)	Description
Main Header	1	40	Contains the source and destination addresses, and important information that's required for every datagram.
Extension Headers	0 or more	Variable	Each contains one type of extra information that supports various features, including fragmentation, source routing, security, and options.
Data	1	Variable	The payload from the upper layer that will be transmitted in the datagram.



**Figure 26-1:** IPv6 general datagram structure

Note that as with IPv4, large payloads may be fragmented prior to encapsulation in order to ensure that the total size of the datagram doesn't exceed the maximum size permitted on an underlying network. However, the details of fragmentation in IPv6 are different than in IPv4, as explained in Chapter 27.

## IPv6 Datagram Main Header Format

IPv6 datagrams use a structure that includes a regular header and, optionally, one or more extension headers. This regular header is like the header of IPv4 datagrams, though it has a different format, as you will see shortly. The standards don't give this header a name; it is just "*the IPv6 header.*" To differentiate it from IPv6 extension headers, I call it the *main header*.

The IPv6 main header is required for every datagram. It contains addressing and control information that are used to manage the processing and routing of the datagram. The main header format of IPv6 datagrams is described in Table 26-2 and illustrated in Figure 26-2.

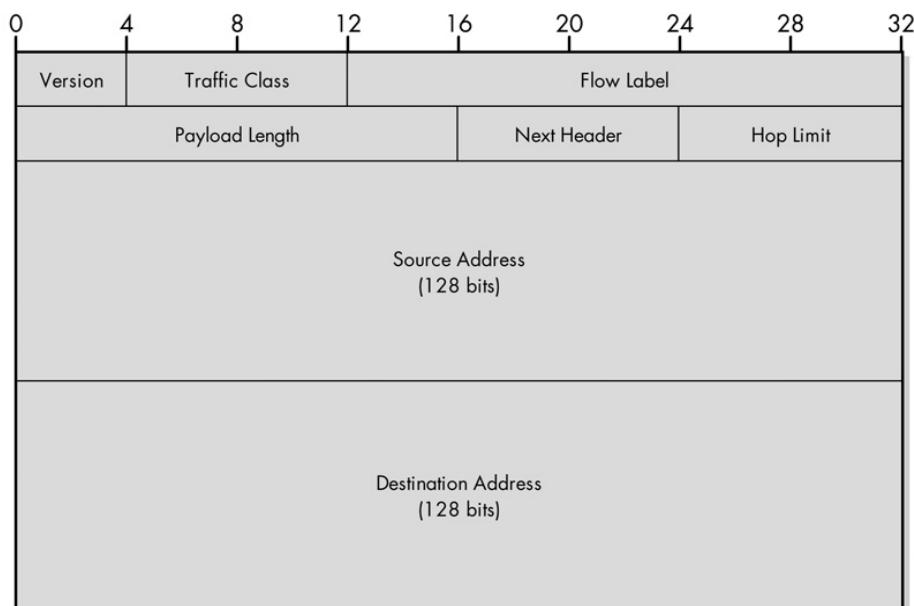
**Table 26-2:** IPv6 Main Header Format

Field Name	Size (Bytes)	Description
Version	1/2 (4 bits)	This identifies the version of IP that's used to generate the datagram. This field is used the same way as in IPv4, except that it carries the value 6 (0110 binary).
Traffic Class	1	This field replaces the Type of Service (TOS) field in the IPv4 header. It is used not in the original way that the TOS field was defined (with Precedence, D, T, and R bits), but rather, using the new <i>Differentiated Services (DS)</i> method defined in RFC 2474. That RFC actually specifies quality-of-service (QoS) techniques for both IPv4 and IPv6; see the IPv4 format description (Chapter 21) for a bit more information.
Flow Label	2 1/2 (20 bits)	This large field was created to provide additional support for real-time datagram delivery and QoS features. The concept of a <i>flow</i> is defined in RFC 2460 as a sequence of datagrams sent from a source device to one or more destination devices. A unique flow label is used to identify all the datagrams in a particular flow, so that routers between the source and destination all handle them the same way. This helps to ensure uniformity in how the datagrams in the flow are delivered. For example, if a video stream is being sent across an IP internetwork, the datagrams containing the stream could be identified with a flow label to ensure that they are delivered with minimal latency. Not all devices and routers may support flow label handling, and the use of the field by a source device is entirely optional. Also, the field is still somewhat experimental and may be refined over time.
Payload Length	2	This field replaces the Total Length field from the IPv4 header, but it is used differently. Rather than measuring the length of the whole datagram, it contains only the number of bytes of the payload. However, if extension headers are included, their length is counted here as well. In simpler terms, this field measures the length of the datagram less the 40 bytes of the main header itself.
Next Header	1	This field replaces the Protocol field and has two uses. When a datagram has extension headers, this field specifies the identity of the first extension header, which is the next header in the datagram. When a datagram has just this "main" header and no extension headers, it serves the same purpose as the old IPv4 <i>Protocol</i> field and has the same values, though new numbers are used for the IPv6 versions of common protocols. In this case the "next header" is the header of the upper layer message the IPv6 datagram is carrying. I'll discuss this in more detail a bit later in this chapter.
Hop Limit	1	This replaces the Time to Live (TTL) field in the IPv4 header; its name better reflects the way that TTL is used in modern networks (because TTL is really used to count hops, not time).

(continued)

**Table 26-2:** IPv6 Main Header Format (continued)

Field Name	Size (Bytes)	Description
Source Address	16	The 128-bit IP address of the originator of the datagram. As with IPv4, this is always the device that originally sent the datagram.
Destination Address	16	The 128-bit IP address of the intended recipient of the datagram: unicast, anycast, or multicast. Again, even though devices such as routers may be the intermediate targets of the datagram, this field is always for the ultimate destination.



**Figure 26-2:** IPv6 main header format

### IPv6 Next Header Field

The Next Header field is one of the most important additions to the IPv6 datagram format. When an IPv6 datagram uses extension headers, this field contains an identifier for the first extension header, which, in turn, uses its own Next Header field to point to the next header, and so on. The last extension header then references the encapsulated higher-layer protocol. Because the higher-layer protocol's header appears at the start of the IPv6 Data field, it is like the “next header” to the device receiving the datagram. For some folks, this is a bit tough to see conceptually; you can find more detail on how the field works (including a useful illustration, Figure 26-3) in the “IPv6 Header Chaining Using the Next Header Field” section later in this chapter.

Some of the most common values for the Next Header field in IPv6 are shown in Table 26-3.

**Table 26-3:** Common IPv6 Next Header Values

Value (Hexadecimal)	Value (Decimal)	Protocol/Extension Header
00	0	Hop-By-Hop Options Extension Header (Note that this value was "Reserved" in IPv4)
01	1	Internet Control Message Protocol version 4 (ICMPv4)
02	2	Internet Group Management Protocol version 4 (IGMPv4)
04	4	IP-in-IP Encapsulation
06	6	Transmission Control Protocol (TCP)
08	8	Exterior Gateway Protocol (EGP)
11	17	User Datagram Protocol (UDP)
29	41	IPv6
2B	43	Routing Extension Header
2C	44	Fragmentation Extension Header
2E	46	Resource Reservation Protocol (RSVP)
32	50	Encrypted Security Payload (ESP) Extension Header
33	51	Authentication Header (AH) Extension Header
3A	58	ICMPv6
3B	59	No Next Header
3C	60	Destination Options Extension Header

The total length of the main IPv6 header format is 40 bytes. This is double the size of the IPv4 header without options, largely because of the extra 24 bytes needed for the monstrous IPv6 addresses. There are only 8 bytes of nonaddress header fields in the IPv6 main header, compared to 12 in the IPv4 header.

### ***Key Changes to the Main Header Between IPv4 and IPv6***

To summarize, the IPv6 main header compares to the IPv4 header as follows:

**Unchanged Fields** Three fields are used the same way, and they retain the same name (though they have different content and/or size): Version, Source Address, and Destination Address.

**Renamed Fields** Two fields are used the same way, but they are renamed: Traffic Class and Hop Limit.

**Modified Fields** Two fields are used in a way similar way to their IPv4 predecessors, but they are slightly different in meaning and also renamed: Payload Length and Next Header.

**Added Field** There is one new field: Flow Label.

**Removed Fields** To cut down on header length and unnecessary work, five IPv4 header fields are removed from the IPv6 header:

- The *Internet Header Length* field is no longer needed, because the main IPv6 header is fixed in length at 40 bytes.
- The *Identification*, *Flags*, and *Fragment Offset* fields are used for fragmentation, which is done less in IPv6 than IPv4, so these fields are now found only when needed in the Fragmentation extension header.
- The *Header Checksum* field is no longer needed, because the decision was made to eliminate header checksum calculations in IPv6. It was viewed as redundant with higher-layer error-checking and data link layer CRC calculations. This saves processing time for routers and 2 bytes in the datagram header.

In addition, while options were formerly considered part of the main header in IPv4, they are separate in IPv6.

## IPv6 Datagram Extension Headers

After the mandatory main header in an IPv6 datagram, one or more extension headers may appear before the encapsulated payload. These headers were created in an attempt to provide both flexibility and efficiency in the creation of IPv6 datagrams. All the fields that are needed for only special purposes are put into extension headers and placed in the datagram when needed. This allows the size of the main datagram header to be made small and streamlined, containing only those fields that really must be present all the time.

There is often confusion regarding the role of extension headers, especially when compared to datagram options. The IPv4 datagram had only one header, but it included a provision for options, and IPv6 also has options, so why bother with extension headers?

It would have been possible to do everything using options. However, it was deemed a better design to employ extension headers for certain sets of information that are needed for common functions such as fragmenting. Options are indeed still supported in IPv6; they are used to supply even more flexibility by providing variable-length fields that can be used for any purpose. They are themselves defined using extension headers, as you will see shortly.

When extension headers are included in an IPv6 datagram, they appear one after the other following the main header. Each extension header type has its own internal structure of fields.

### IPv6 Header Chaining Using the Next Header Field

The only field common to all extension header types is the Next Header field, which actually appears at the end of one header type, the ESP header. The 8-bit Next Header field is used to logically link all the headers in an IPv6 datagram, as follows:

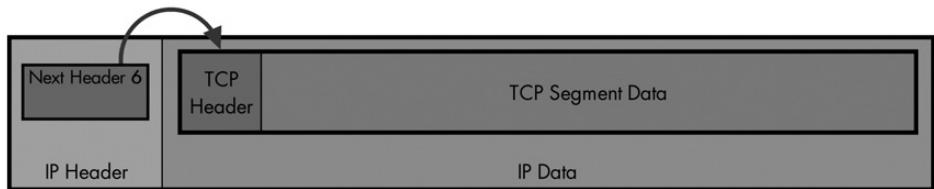
- The Next Header field in the main header contains a reference number for the first extension header type.

- The Next Header field in the first extension header contains the number of the second extension header type, if there is a second one. If there's a third, the second header's Next Header points to it, and so on.
- The Next Header field of the last extension header contains the protocol number of the encapsulated higher-layer protocol. In essence, this field points to the “next header” within the payload itself.

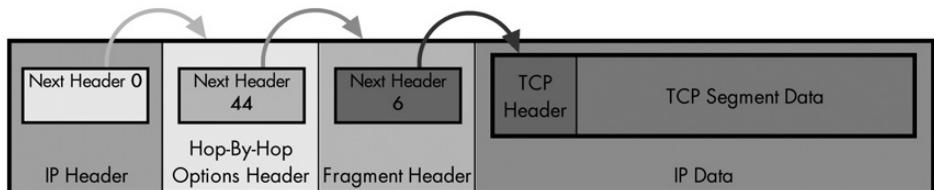
For example, suppose a datagram that encapsulates TCP has a Hop-By-Hop Options extension header and a Fragment extension header. Then, the Next Header fields of these headers would contain the following values:

- The main header would have a Next Header value of 0, indicating the Hop-By-Hop Options header.
- The Hop-By-Hop Options header would have a Next Header value of 44 (decimal), which is the value for the Fragment extension header.
- The Fragment header would have a Next Header value of 6.

This is illustrated in Figure 26-3.



**IPv6 Datagram with No Extension Headers Carrying TCP Segment**



**IPv6 Datagram with Two Extension Headers Carrying TCP Segment**

**Figure 26-3: IPv6 extension header linking using the Next Header field** The Next Header field allows a device to more easily process the headers in a received IPv6 datagram. When a datagram has no extension headers, the “next header” is actually the header at the start of the IP Data field, which, in this case, is a TCP header with a value of 6. This is the same way the Protocol field is used in IPv4. When extension headers do appear, the Next Header value of each header contains a number indicating the type of the following header in the datagram, so they logically chain together the headers.

**KEY CONCEPT** The IPv6 Next Header field is used to chain together the headers in an IPv6 datagram. The Next Header field in the main header contains the number of the first extension header; its Next Header contains the number of the second, and so forth. The last header in the datagram contains the number of the encapsulated protocol that begins the Data field.

## **Summary of IPv6 Extension Headers**

Table 26-4 lists the different extension headers, showing each one's Next Header value, length, defining RFC, and a brief description of how it is used.

**Table 26-4:** IPv6 Extension Headers

<b>Next Header Value (Decimal)</b>	<b>Extension Header Name</b>	<b>Length (Bytes)</b>	<b>Description</b>	<b>Defining RFC</b>
0	Hop-By-Hop Options	Variable	Defines an arbitrary set of options that are intended to be examined by all devices on the path from the source to destination device(s). This is one of two extension headers used to define variable-format options.	2460
43	Routing	Variable	Defines a method for allowing a source device to specify the route for a datagram. This header type actually allows the definition of multiple routing types. The IPv6 standard defines the Type 0 Routing extension header, which is equivalent to the "loose" source routing option in IPv4. It's used in a similar way. See the "IPv6 Routing Extension Header" section in this chapter for the format of this extension header.	2460
44	Fragment	8	When a datagram contains only a fragment of the original message, contains the Fragment Offset, Identification, and More Fragment fields that were removed from the main header. See the "IPv6 Fragment Extension Header" section in this chapter for the format of this extension header, and the topic on fragmentation and reassembly (Chapter 27) for details on how the fields are used.	2460
50	Encapsulating Security Payload (ESP)	Variable	Carries encrypted data for secure communications. This header is described in detail in Chapter 29, which covers IPsec.	2406
51	Authentication Header (AH)	Variable	Contains information used to verify the authenticity of encrypted data. This header is described in detail in Chapter 29.	2402
60	Destination Options	Variable	Defines an arbitrary set of options that are intended to be examined only by the destination(s) of the datagram. This is one of two extension headers used to define variable-format options.	2460

Note that the Next Header value of the IPv6 main header is 41; that of an IPv4 header is 4 (its protocol number). There is also a "dummy" extension header called No Next Header that has a value of 59. This is a placeholder that, when found in the Next Header field, indicates that there is nothing after that extension header.

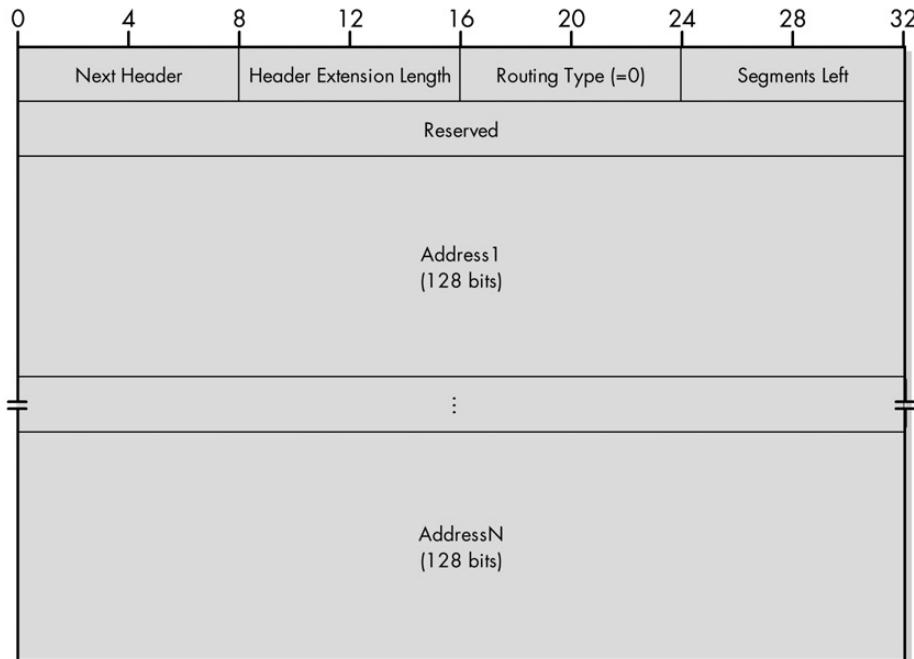
As mentioned in Table 26-4, the formats for several of the headers are provided in other areas of this book. I will describe two of them here, however: the Routing extension header and the Fragment extension header.

## IPv6 Routing Extension Header

The Routing extension header is used to perform source routing in IPv6. It is described in Table 26-5 and illustrated in Figure 26-4.

**Table 26-5:** IPv6 Routing Extension Header Format

Field Name	Size (Bytes)	Description
Next Header	1	Contains the protocol number of the next header after the Routing header. Used to link headers together, as described earlier in this chapter.
Hdr Ext Len	1	For Header Extension Length, specifies the length of the Routing header in 8-byte units, not including the first 8 bytes of the header. For a Routing Type field of 0, this value is thus two times the number addresses embedded in the header.
Routing Type	1	Allows multiple routing types to be defined; at present, the only value used is 0.
Segments Left	1	Specifies the number of explicitly named nodes remaining in the route until the destination.
Reserved	4	Not used; set to zeros.
Address1 . . . AddressN	Variable (Multiple of 16)	A set of IPv6 addresses that specify the route to be used.



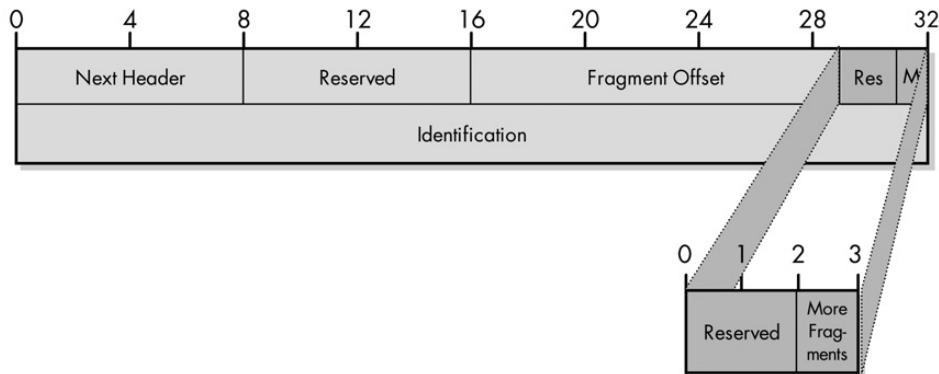
**Figure 26-4:** IPv6 Routing extension header format

## IPv6 Fragment Extension Header

The Fragment extension header is included in fragmented datagrams to provide the information that's necessary to allow the fragments to be reassembled. It is described in Table 26-6 and illustrated in Figure 26-5.

**Table 26-6:** IPv6 Fragment Extension Header Format

Field Name	Size (Bytes)	Description
Next Header	1	Contains the protocol number of the next header after the Fragment header. Used to link headers together, as described earlier in this chapter.
Reserved	1	Not used; set to zeros.
Fragment Offset	13/8 (13 bits)	Specifies the offset, or position, in the overall message where the data in this fragment goes. It is specified in units of 8 bytes (64 bits) and used in a manner very similar to the field of the same name in the IPv4 header.
Res	1/4 (2 bits)	Not used; set to zeros.
M Flag	1/8 (1 bit)	For More Fragments Flag, same as the flag of the same name in the IPv4 header. When set to 0, indicates the last fragment in a message; when set to 1, indicates that more fragments are yet to come in the fragmented message.
Identification	4	Same as the field of the same name in the IPv4 header, but expanded to 32 bits. It contains a specific value that is common to each of the fragments belonging to a particular message. This ensures that pieces from different fragmented messages are not mixed together.



**Figure 26-5:** IPv6 Fragment extension header format

## IPv6 Extension Header Order

Each extension header appears only once in any datagram (with one exception, as you'll see shortly). Also, only the final recipients of the datagram examine extension headers, not intermediate devices (again with one exception, which you will see momentarily).

RFC 2460 specifies that when multiple headers appear, they should be in the following order, after the main header and before the higher-layer encapsulated header in the IPv6 datagram payload:

1. Hop-By-Hop Options
2. Destination Options (for options to be processed by the destination as well as devices specified in a Routing header)
3. Routing
4. Fragmentation
5. Authentication Header
6. Encapsulating Security Payload
7. Destination Options (for options processed only by the final destination)

Now let's look at those exceptions. The only header that can appear twice is Destination Options. Normally, it appears as the last header. However, the datagram may also have a Destination Options header that contains options that must be examined by a list of devices specified in a source route, in addition to the destination. In this case, the Destination Options header for these options is placed before the Routing header. A second such header containing options for only the final destination may also appear.

**KEY CONCEPT** Each extension header may appear only once in an IPv6 datagram, and each one must appear in a fixed order. The exception is the Destination Options header, which may appear twice: near the start of the datagram for options to be processed by devices en route to the destination and at the end of the extension headers for options intended for only the final destination.

The only header normally examined by all intermediate devices is the Hop-By-Hop Options extension header. It is used specifically to convey management information to all routers in a route. The Hop-By-Hop Options extension header must appear as the first extension header if present. Since it is the only one that every router must read (and this represents a performance drain on routers), it is given top billing to make it easier and faster to find and process.

Finally, note that all extension headers must be a multiple of eight bytes in length for alignment purposes. Also, remember that the Next Header value for a particular extension header appears in the Next Header field of the preceding header, not the header itself.

## IPv6 Datagram Options

In IPv4, all extra information required for various purposes is placed into the datagram in the form of options that appear in the IPv4 header. In IPv6, the new concept of extension headers is introduced, as you just saw. These headers take the place of many of the predefined IPv4 options. However, the concept of options is still maintained in IPv6 for a slightly different purpose.

Options allow the IPv6 datagram to be supplemented with arbitrary sets of information that aren't defined in the regular extension headers. They provide maximum flexibility, thereby allowing the basic IPv6 protocol to be extended in ways the designers never anticipated, with the goal of reducing the chance of the protocol becoming obsolete in the future.

I said that IPv6 options supplement extension headers; in fact, they are actually implemented as extension headers. There are two different ones used to encode options. These two headers differ only in terms of how devices will process the options they contain; otherwise, they are formatted the same and used in the same way.

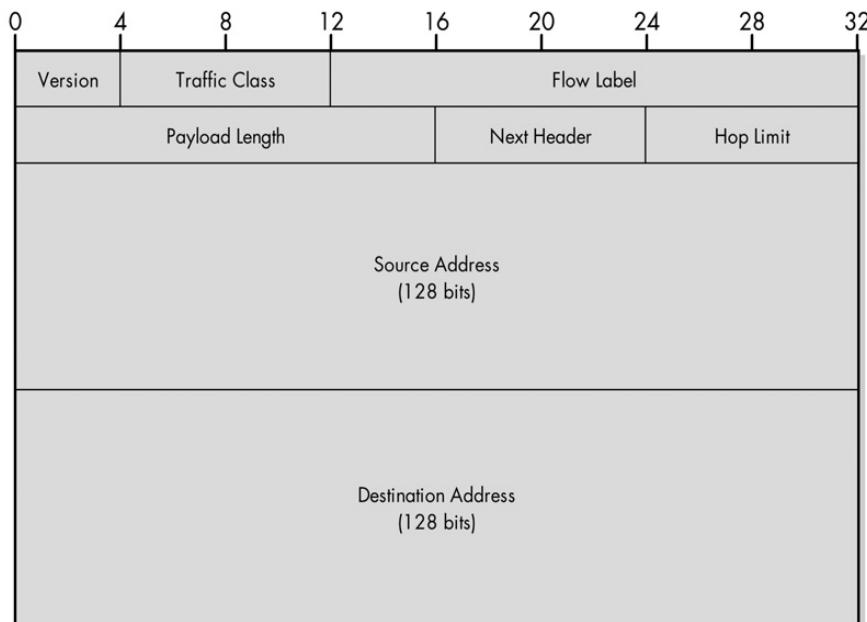
The two extension header types are as follows:

**Destination Options** Contains options that are intended only for the ultimate destination of the datagram (and perhaps a set of routers in a Routing header, if present).

**Hop-By-Hop Options** Contains options that carry information for every device (router) between the source and destination.

Each of these header types has a one-byte Next Header field, and a one-byte Header Extension Length field that indicates the header's overall length. The rest of the header has one or more option fields.

Figure 26-6 illustrates the overall format of these two headers. The format of each option is similar to that of IPv4 options, as shown in Tables 26-7 and 26-8.



**Figure 26-6: IPv6 Hop-By-Hop Options and Destination Options header formats** Each of these extension headers begins with two fixed fields, Next Header and Header Extension Length. The rest of the header consists of a sequence of variable-length options. Each option has a structure that consists of a type/length/value triplet, shown in Table 26-7.

**Table 26-7:** IPv6 Option Format

Subfield Name	Size (Bytes)	Description
Option Type	1	This field indicates the type of option. The bits are interpreted according to the sub-subfield "structure, described in Table 26-8.
Opt Data Len	1	Specifies the length of the Option Data subfield. Note that this is a change in semantics from IPv4, where the Length field indicated the size of the entire option; in IPv6 the length of the Option Type and Option Data Length fields are not included.
Option Data	Variable	The data to be sent as part of the option, which is specific to the option type. Also sometimes referred to as the Option Value.

**Table 26-8:** IPv6 Option Type Subfields

Sub-Subfield Name	Size (Bytes)	Description
Unrecognized Option Action	2/8 (2 bits)	The first two bits specify what action should be taken if the device processing the option doesn't recognize the Option Type. The four values are as follows: 00: Skip option; process rest of header. 01: Discard datagram; do nothing else. 10: Discard datagram and send an ICMP Parameter Problem message with code 2 back to the datagram source. 11: Discard datagram and send the ICMP message as for value 10, only if destination was not a multicast address.
Option Change Allowed Flag	1/8 (1 bit)	Set to 1 if the Option Data can change while the datagram is en route, or left at 0 if it cannot.
Remainder of Option Type	5/8 (5 bits)	Five remaining bits that allow the specification of 32 different combinations for each combination of the three preceding bits.

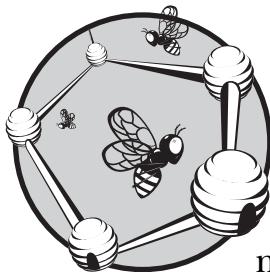
**NOTE** *The Option Type subfield is a bit strange in terms of how it is interpreted. Even though it has a substructure with three sub-subfields (as shown in Table 26-8, that structure is informal—the eight bits of this field are taken as a single entity. Despite the special meaning of the three highest-order bits, the entire field is called the Option Type, not just the last five bits, and the whole is used as a single value from 0 to 255. In fact, the sub-subfield names aren't even specified in the standard; I made them up.*

Since each option has a subfield for type, length, and value (data), the options are sometimes said to be TLV-encoded. If there are multiple options, they are placed one after each other in the header. At the end of all the options, in a Hop-By-Hop Options or Destination Options extension header, a device may place padding to ensure that the header is a multiple of eight bytes in length.

**KEY CONCEPT** Two IPv6 extension header types, Hop-By-Hop Options and Destination Options, are used to carry arbitrary optional information in IPv6 datagrams. Each consists of a set of variable-length options that are defined using three subfields that indicate the option's type, length, and value.

# 27

## **IPV6 DATAGRAM SIZE, FRAGMENTATION, REASSEMBLY, AND ROUTING**



Internet Protocol version 6 (IPv6) changes many of the operating details of IP, but most of the basics are the same. In particular, devices still need to deliver datagrams over an internetwork that may use different underlying network technologies. This means that we must be concerned here, as we were in IPv4, with the mechanics of datagram sizing, handling fragmentation and reassembly, and dealing with issues related to routing.

In this chapter, I complete the discussion of IPv6 by examining these matters, with an eye toward contrasting how they work in IPv6. This includes a look at IPv6 datagram sizing, changes to the maximum transmission unit (MTU), and fragmentation and reassembly. I also briefly discuss areas where IPv6 routing is performed in the same way as in IPv4, as well as where routing has changed.

## Overview of IPv6 Datagram Sizing and Fragmentation

The job of IP is to convey messages across an internetwork of connected networks. When datagrams are sent between hosts on distant networks, they are carried along their journey by routers, one hop at a time, over many physical network links. On each step of this journey, the datagram is encoded in a data link layer frame for transmission.

In order for a datagram to be successfully carried along a route, its size must be small enough to fit within the lower-layer frame at each step of the way. The term *maximum transmission unit (MTU)* describes the size limit for any given physical network. If a datagram is too large for the MTU of a network, it must be broken into pieces—a process called *fragmentation*—and then the pieces are *reassembled* at the destination device. This has been a requirement since IPv4, and I explain the concepts and issues related to datagram size, MTUs, fragmentation, and reassembly in detail in the associated IPv4 discussion, in Chapter 22.

All of these issues apply to sending datagrams in IPv6 as much as they did in IPv4. However, as in other areas of the protocol, some important details of how fragmentation and reassembly are done have changed. These changes were made to improve the efficiency of the routing process and to reflect the realities of current networking technologies: Most can handle average IP datagrams without needing fragmentation.

The most important differences between IPv4 and IPv6 with respect to datagram size, MTU, and fragmentation and reassembly are as follows:

**Increased Default MTU** In IPv4, the minimum MTU that routers and physical links were required to handle was 576 bytes. In IPv6, all links must handle a datagram size of at least 1280 bytes. This more than doubling in size improves efficiency by increasing the ratio of maximum payload to header length and reduces the frequency with which fragmentation is required.

**Elimination of en Route Fragmentation** In IPv4, datagrams may be fragmented by either the source device or by routers during delivery. In IPv6, only the source node can fragment; routers do not. The source must fragment to the size of the smallest MTU on the route before transmission. This has both advantages and disadvantages, as you will see. Reassembly is still done only by the destination, as in IPv4.

**MTU Size Error Feedback** Since routers cannot fragment datagrams, they must drop them if they are forced to try to send a too-large datagram over a physical link. Using the Internet Control Message Protocol version 6 (ICMPv6; see Chapter 31), a feedback process has been defined that allows routers to tell source devices that they are using datagrams that are too large for the route.

**Path MTU Discovery** Since source devices must decide on the correct size of fragments, it is helpful if they have a mechanism for determining what this should be. This capability is provided through a special technique called *Path MTU Discovery*, which was originally defined for IPv4 but has been refined for IPv6.

**Movement of Fragmentation Header Fields** To reflect the decreased importance of fragmentation in IPv6, the permanent fields related to the process that were in the IPv4 header have been farmed out to a Fragment extension header and are included only when needed.

## Implications of IPv6's Source-Only Fragmentation Rule

I find the changes in the fragmentation and reassembly process interesting. While many other changes in IPv6 represent a shift in responsibility for functions from host devices to routers, this one is the opposite. In IPv4, a source node can send a datagram of any size that its local link can handle, and let the routers take care of fragmenting it as needed. This seems like a sensible model; nodes communicate on a large, virtual network, and the details of splitting messages as needed for physical links are handled invisibly.

The problem with this is that it represents a performance drag on routing. It is much faster for a router to forward a datagram intact than to spend time fragmenting it. In some cases, fragmentation would need to occur multiple times during the transmission of a datagram, and remember that this must happen for every datagram on a route. It is a lot more efficient for the source to just send datagrams that are the right size in the first place.

Of course, there's a problem here: How does the source know what size to use? The source has no understanding of the physical networks used by the route datagrams will take to a destination; in fact, it doesn't even know what the routes are! Thus, it has no idea of what MTU would be best. It has two choices:

**Use the Default MTU** The first option is simply to use the default MTU of 1280 bytes, which all physical networks must be able to handle. This is a good choice, especially for short communications or for sending small amounts of data.

**Use Path MTU Discovery** The alternative is to make use of the Path MTU Discovery feature, as described later in the chapter. This feature, defined in RFC 1981, defines a method whereby a node sends messages over a route to determine what the overall minimum MTU for the path is. It's a technique that's very similar to the way it is done in IPv4, as discussed in Chapter 22.

Since routers can't fragment in IPv6, if a datagram is sent by a source that is too large for a router, it must drop the datagram. It will then send back to the source feedback about this occurrence, in the form of an ICMPv6 Packet Too Big message. This tells the source that its datagram was dropped and that it must fragment (or reduce the size of its fragments).

This feedback mechanism is also used in discovering path MTUs. The source node sends a datagram that has the MTU of its local physical link, since that represents an upper bound on the MTU of the path. If this goes through without any errors, it knows it can use that value for future datagrams to that destination. If it gets back any Packet Too Big messages, it tries again using a smaller datagram size. The advantage of this over the 1280 default is that it may allow a large communication to proceed with a higher MTU, which improves performance.

**KEY CONCEPT** In IPv6, fragmentation is performed only by the device that's sending a datagram, not by routers. If a router encounters a datagram too large to send over a physical network with a small MTU, the router sends an ICMPv6 *Packet Too Big* message back to the source of the datagram. This can be used as part of a process called *Path MTU Discovery* to determine the minimum MTU of an entire route.

One drawback of the decision to only fragment at the source is that it introduces the potential for problems if there is more than one route between devices or if routes change. In IPv4, fragmentation is dynamic and automatic; it happens on its own and adjusts as routes change. Path MTU Discovery is a good feature, but it is static. It requires that hosts keep track of MTUs for different routes and update them regularly. IPv6 does this by redoing Path MTU Discovery if a node receives a Packet Too Big message on a route for which it has previously performed Path MTU Discovery. However, this takes time.

## The IPv6 Fragmentation Process

The actual mechanics of fragmentation in IPv6 are similar to those in IPv4, with the added complication that extension headers must be handled carefully. For purposes of fragmentation, IPv6 datagrams are broken into the following two pieces:

**Unfragmentable Part** This includes the main header of the original datagram, as well as any extension headers that need to be present in each fragment. This means the main header, and any of the following headers, if present: Hop-By-Hop Options, Destination Options (for those options to be processed by devices along a route), and Routing.

**Fragmentable Part** This includes the data portion of the datagram, along with the other extension headers, if present—Authentication Header, Encapsulating Security Payload, and/or Destination Options (for options to be processed only by the final destination).

The Unfragmentable Part must be present in each fragment, while the Fragmentable Part is split up among the fragments. So to fragment a datagram, a device creates a set of fragment datagrams, each of which contains the following, in order:

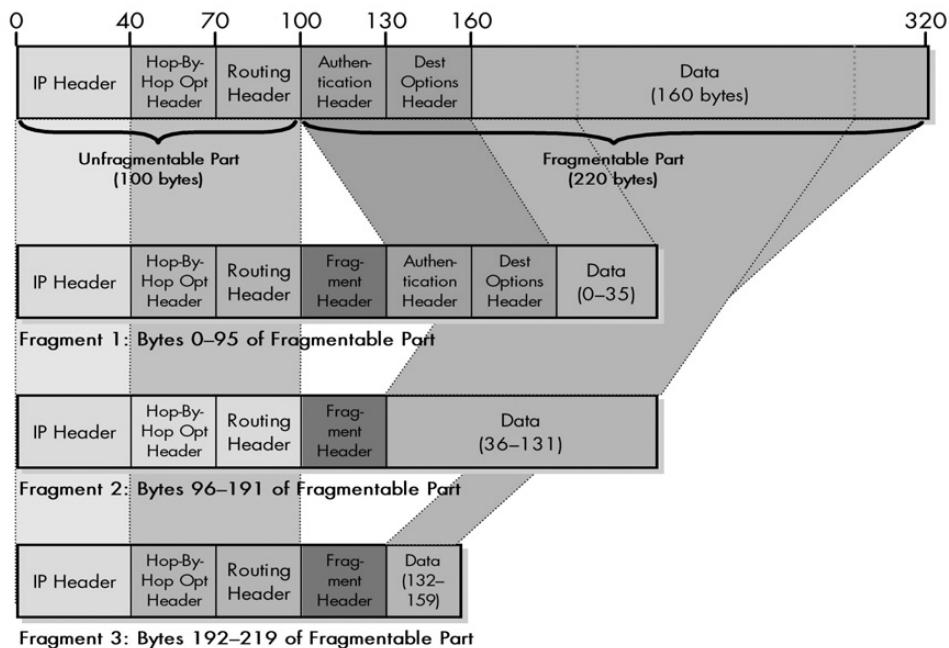
1. **Unfragmentable Part** The full Unfragmentable Part of the original datagram, with its Payload Length changed to the length of the fragment datagram.
2. **Fragment Header** A Fragment header with the Fragment Offset, Identification, and M flags set in the same way they are used in IPv4.
3. **Fragment** A fragment of the Fragmentable Part of the original datagram. Note that each fragment must have a length that is a multiple of 8 bytes, because the value in the Fragment Offset field is specified in multiples of 8 bytes.

**KEY CONCEPT** Fragmentation is done in IPv6 in a manner similar to that of IPv4, except that extension headers must be handled specially. Certain extension headers are considered *unfragmentable* and appear in each fragment; others are fragmented along with the data.

Let's use an example to illustrate how IPv6 fragmentation works. Suppose you have an IPv6 datagram exactly 370 bytes wide, consisting of a 40-byte IP header, four 30-byte extension headers, and 210 bytes of data. Two of the extension headers are unfragmentable, while two are fragmentable. (In practice you would never

need to fragment such a small datagram, but I am trying to keep the numbers simple.) Suppose you need to send this over a link with an MTU of only 230 bytes. You would actually require three fragments, not the two you might expect, because of the need to put the two 30-byte unfragmentable extension headers in each fragment, and the requirement that each fragment be a length that is a multiple of 8. Here is how the fragments would be structured (see Figure 27-1):

- First Fragment** The first fragment would consist of the 100-byte Unfragmentable Part, followed by an 8-byte Fragment header and the first 120 bytes of the Fragmentable Part of the original datagram. This would contain the two fragmentable extension headers and the first 60 bytes of data. This leaves 150 bytes of data to send.
- Second Fragment** This would also contain the 100-byte Unfragmentable Part, followed by a Fragment header, and 120 bytes of data (bytes 60 to 179). This would leave 30 bytes of data remaining.
- Third Fragment** The last fragment would contain the 100-byte Unfragmentable Part, a Fragment header, and the final 30 bytes of data.



**Figure 27-1: IPv6 datagram fragmentation** In this illustration, a 370-byte IPv6 datagram, containing four 30-byte extension headers, is broken into three fragments. The sizes of the fields are shown to scale. The Unfragmentable Part, shown in lighter shading on the left, begins each fragment, followed by the Fragment header (abbreviated as FH in the figure and shown in dark gray). Then, portions of the Fragmentable Part are placed into each fragment in sequence. The Authentication and Destination Options extension headers are part of the Fragmentable Part, so that they appear as part of the first fragment.

The M (More Fragments) flag would be set to 1 in the first two fragments and 0 in the third, and the Fragment Offset values would be set appropriately. See Chapter 22, which covers IPv4 fragmentation, for more on how these fields are used.

The receiving device reassembles by taking the Unfragmentable Part from the first fragment and then assembling the Fragment data from each fragment in sequence.

## IPv6 Datagram Delivery and Routing

IP functions such as addressing, datagram encapsulation, and, if necessary, fragmentation and reassembly, all lead up to the ultimate objective of the protocol: the actual delivery of datagrams from a source device to one or more destination devices. Most of the concepts related to how datagram delivery is accomplished in IPv6 are the same as in IPv4:

- Datagrams are delivered directly when the source and destination nodes are on the same network. When they are on different networks, delivery is indirect, using routing to the destination's network, and then direct to the destination.
- Routers look at IP addresses and determine which portion is the network identifier (network ID) and which is the host identifier (host ID). IPv6 does this in the same basic way as in classless IPv4, despite the fact that IPv6 unicast addresses are assigned using a special hierarchical format.
- Routing is still done on a next-hop basis, with sources generally not knowing how datagrams get from point A to point B.
- Routing is performed by devices called *routers*, which maintain tables of routes that tell them where to forward datagrams to reach different destination networks.
- Routing protocols are used to allow routers to exchange information about routes and networks.

Most of the changes in routing in IPv6 are directly related to changes in other areas of the protocol, as discussed in the previous chapters. Some of the main issues of note related to routing and routers in IPv6 include the following:

**Hierarchical Routing and Aggregation** One of the goals of the structure used for organizing unicast addresses was to improve routing. The unicast addressing format is designed to provide a better match between addresses and Internet topology and to facilitate route aggregation. Classless addressing using CIDR in IPv4 was an improvement but lacked any formal mechanism for creating a scalable hierarchy.

**Scoped Local Addresses** Local-use addresses, including site-local and link-local addresses, are defined in IPv6, and routers must be able to recognize them. They must route them or *not* route them when appropriate. Multicast addresses also have various levels of scope.

**Multicast and Anycast Routing** Multicast is standard in IPv6, not optional as in IPv4, so routers must support it. Anycast addressing is a new type of addressing in IPv6.

**More Support Functions** Capabilities must be added to routers to support new features in IPv6. For example, routers play a key role in implementing autoconfiguration without the help of a server and Path MTU Discovery in the new IPv6 fragmentation scheme.

**New Routing Protocols** Routing protocols such as RIP must be updated to support IPv6.

**Transition Issues** Last, but certainly not least, routers play a major role in supporting the transition from IPv4 to IPv6. They will be responsible for connecting together IPv6 “islands” and performing translation to allow IPv4 and IPv6 devices to communicate with each other during the multiyear migration to the new protocol.



# PART II-5

## IP-RELATED FEATURE PROTOCOLS

The previous two parts thoroughly explored versions 4 and 6 of the Internet Protocol (IP). IP is a very capable protocol that provides the functionality necessary to address, package, and deliver information on TCP/IP internetworks. However, IP was intentionally designed to be simple, without a lot of bells and whistles. To deal with special needs, a number of other protocols have been created to enhance or expand on IP's capabilities. I call these *IP-related feature protocols*.

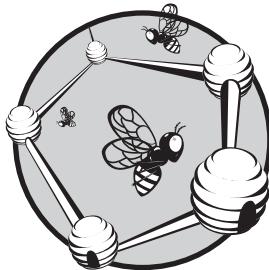
This part contains three chapters that provide complete explanations of three of the more important IP-related feature protocols. The first chapter describes *IP Network Address Translation (IP NAT or NAT)*, which allows private networks to be accessed on the Internet and IP addresses to be shared. The second chapter explores *IP Security (IPsec)*, a set of subprotocols that allows IP datagrams to be authenticated and/or encrypted. The third chapter covers the *Mobile IP* protocol, which corrects some of the problems associated with using TCP/IP with mobile hosts.

This part assumes that you have a good understanding of the operation of IP, discussed in Parts II-3 and II-4.



# 28

## IP NETWORK ADDRESS TRANSLATION (NAT) PROTOCOL



To help extend the life of the Internet Protocol version 4 (IPv4) addressing scheme while the newer IPv6 protocol is developed and deployed, other technologies have been developed. One of the most important of these is *IP Network Address Translation (NAT)*. This technology allows a small number of public IP addresses to be shared by a large number of hosts using private addresses. This essential work-around allows the global Internet to actually have far more hosts on it than its address space would normally support. At the same time, it provides some security benefits by making hosts more difficult to address directly by foreign machines on the public Internet.

In this chapter, I provide a description of the concepts behind IP NAT and an explanation of operation of IP NAT types. I begin with an overview of the protocol and discussion of its advantages and disadvantages. I describe the address terminology that you need to know in order to understand how NAT functions and the differences between various translation techniques. I explain the way that address mappings are performed and the difference between static and dynamic address mapping.

I then explain the operation of the four main types of NAT: unidirectional, bidirectional, port-based, and overlapping. I conclude with a bit more information on compatibility issues associated with NAT.

NAT was developed in large part to deal with the address shortage problem in IPv4, so it is associated and used with IPv4. It is possible to implement an IPv6-compatible version of NAT, but address translation isn't nearly as important in IPv6, which was designed to give every TCP/IP device its own unique address. For this reason, in this chapter, I focus in on the use of NAT with IPv4.

**NOTE** *Incidentally, most people just call this technology Network Address Translation without the IP. However, this sounds to me rather generic, and since the version I'm discussing here is specific to IP, I prefer to make it clear that this is an IP feature. That said, for simplicity I often just say "NAT," too, since that's shorter. I should also point out that there are quite a few people who don't consider NAT to be a protocol in the strictest sense of the word.*

## IP NAT Overview

The decision to make IP addresses only 32 bits long as part of the original design of IP led to a serious problem when the Internet exploded in popularity beyond anyone's expectations: the exhaustion of the address space. Classless addressing helped make better use of the address space, and IPv6 was created to ensure that we will never run out of addresses again. However, classless addressing has only slowed the consumption of the IPv4 address space, and IPv6 has taken years to develop and will require years more to deploy.

The shortage of IP addresses promised to grow critical by the end of the 1990s unless some sort of solution was implemented until the transition to IPv6 was completed. Creative engineers on the Internet Engineering Task Force (IETF) were up to the challenge. They created a technique that would not only forestall the depletion of the address space, but could also be used to address the following two other growing issues in the mid- to late 1990s:

**Increasing Cost of IP Addresses** As any resource grows scarce, it becomes more expensive. Even when IP addresses were available, it cost more to get a larger number from a service provider than a smaller number. It was desirable to conserve them not only for the sake of the Internet as a whole, but to save money.

**Growing Concerns over Security** As Internet use increased in the 1990s, more bad guys started using the network also. The more machines a company had directly connected to the Internet, the greater their potential exposure to security risks.

One solution to these problems was to set up a system whereby a company's network was not connected directly to the Internet, but rather *indirectly*. Setting up a network this way is possible due to the following important characteristics of how most organizations use the Internet:

**Most Hosts Are Client Devices** The Internet is client/server based, and the majority of hosts are clients. Client devices generally don't need to be made publicly accessible. For example, when using your local PC to access the World Wide Web,

you issue requests to servers and they respond back, but servers don't have any reason to try to initiate contact with you. Clients, not servers, begin most correspondence, by definition.

**Few Hosts Access the Internet Simultaneously** When you have a large number of hosts that are connected to the Internet on the same network, usually only a small number of those hosts are trying to access the Internet at any given time. It isn't necessary to assume they will all need to access servers at once. Even while you actively browse the Web, you pause for a number of seconds to read information from time to time; you are only accessing the web server for the time it takes to perform a transaction.

**Internet Communications Are Routed** Communications between an organization's network and the Internet go through a router, which acts as a control point for traffic flows.

The best way to explain why these attributes matter is to draw an analogy to how telephones are used in an organization, because many of the same attributes apply there. Most of the telephones in a typical organization are used to let employees make calls out. Usually there is no need to have any way to call employees directly; instead, one system or person can handle all incoming calls. Only a few employees are ever making a call to the outside world at any given time. And all calls are routed through a central point that manages the telephone system.

For these reasons, to save money, organizations don't run separate public telephone lines to every employee's desk. Instead, they set up a telephone system whereby each employee gets an *extension*, which is basically a local telephone number valid only within the organization. A small number of outside lines is made available in a pool for employees to share, and the telephone system matches the inside extensions to the outside lines as needed. A voice mail system and human receptionist handle the routing of calls into the organization. (Yes, of course some companies have a direct mapping between extension numbers and real telephone numbers.)

A very similar technique can be used for connecting an organization's computers to the Internet. In TCP/IP networks, this technology was first formalized in RFC 1631, "The IP Network Address Translator (NAT)," which was adopted in May 1994. The word *translator* refers to the device (router) that implements NAT. More commonly, the technology as a whole is called *IP Network Address Translation (IP NAT or NAT)*.

**NOTE** *The document status of RFC 1631 is informational. This means that, technically, IP NAT is not an official Internet standard.*

A basic implementation of NAT involves setting up an organization's internal network using one of the private addressing ranges set aside for local IP networks. One or more public (Internet) addresses are also assigned to the organization as well, and one or more NAT-capable routers are installed between the local network and the public Internet. The public IP addresses are like outside lines in the telephone system, and the private addresses are like internal extensions.

The NAT router plays the role of telephone system computer and receptionist. It maps internal extensions to outside lines as needed, and also handles “incoming calls” when required. It does this by not just routing IP datagrams, but also by modifying them as needed, thereby translating addresses in datagrams from the private network into public addresses for transmission on the Internet, and back again.

**KEY CONCEPT** *IP Network Address Translation (IP NAT or NAT)* is a technique that allows an organization to set up a network using private addresses, while still allowing for communication on the public Internet. A NAT-capable router translates private to public addresses and vice versa as needed. This allows a small number of public IP addresses to be shared among a large number of devices and provides other benefits as well, but it also has some drawbacks.

Over time, newer versions of NAT have also been created. They solve other problems or provide additional capabilities. *Port-Based NAT* allows for the sharing of even more hosts on a limited number of IP addresses by letting two or more devices share one IP address at a time. So-called *twice NAT* helps with the implementation of virtual private networks (VPNs) by translating both source and destination addresses in both incoming and outgoing datagrams.

### ***Advantages of IP NAT***

NAT is one of those technologies that has a long list of advantages and disadvantages. This means it can be extremely useful in a variety of scenarios, but also problematic in others. The main advantages are as follows:

**Public IP Address Sharing** A large number of hosts can share a small number of public IP addresses. This saves money and also conserves IP address space.

**Easier Expansion** Since local network devices are privately addressed and a public IP address isn’t needed for each one, it is easy to add new clients to the local network.

**Greater Local Control** Administrators get all the benefits of control that come with a private network, but can still connect to the Internet.

**Greater Flexibility in Internet Service Provider (ISP) Service** Changing the organization’s ISP is easier, because only the public addresses change. It isn’t necessary to renumber all the client machines on the network.

**Increased Security** The NAT translation represents a level of indirection. Thus, it automatically creates a type of firewall between the organization’s network and the public Internet. It is more difficult for any client devices to be accessed directly by someone malicious because the clients don’t have publicly known IP addresses.

**(Mostly) Transparent** NAT implementation is mostly transparent, because the changes take place in one or perhaps a few routers. The dozens or hundreds of hosts themselves don’t need to be changed.

## ***Disadvantages of IP NAT***

The previously listed advantages are all good reasons to use NAT, but there are drawbacks to the technique as well:

**Complexity** NAT represents one more complexity in terms of setting up and managing the network. It also makes troubleshooting more confusing due to address substitutions.

**Problems Due to Lack of Public Addresses** Certain functions won't work properly due to lack of a real IP address in the client host machines.

**Compatibility Problems with Certain Applications** I said earlier that NAT was only mostly transparent. There are, in fact, compatibility issues with certain applications that arise because NAT tinkers with the IP header fields in datagrams but not in the application data. This means tools like the File Transfer Protocol (FTP; see Chapter 72), which pass IP addresses and port numbers in commands, must be specially handled, and some applications may not work.

**Problems with Security Protocols** Protocols like IPsec are designed to detect modifications to headers and commonly balk at the changes that NAT makes, since they cannot differentiate those changes from malicious datagram hacking. It is still possible to combine NAT and IPsec, but this becomes more complicated.

**Poor Support for Client Access** The lack of a public IP address for each client is a double-edged sword; it protects against hackers trying to access a host, but it also makes it difficult for legitimate access to clients on the local network. Peer-to-peer applications are harder to set up, and something like an organizational website (accessed from the Internet as a whole) usually needs to be set up without NAT.

**Performance Reduction** Each time a datagram transitions between the private network and the Internet, an address translation is required. In addition, other work must be done as well, such as recalculating header checksums. Each individual translation takes little effort, but when you add it up, you are giving up some performance.

Some of these cancel out some of the benefits of certain items in the previous list. However, many organizations feel that the advantages outweigh the disadvantages, especially if they use the Internet in primarily a client/server fashion, as most do. For this reason, NAT has become quite popular. However, bear in mind that the main problem that led to NAT is lack of address space. IPv6 fixes this problem, while NAT merely finds a clever work-around for it. For this reason, many people consider NAT a kludge. Once IPv6 is deployed, it will no longer be needed, and some folks don't like it even for IPv4. On the other hand, some feel its other benefits make it worthy of consideration even in IPv6.

**NOTE** A kludge (or kluge) is something that is used to address a problem in an inelegant way, like hammering a nail using the side of an adjustable wrench.

## IP NAT Address Terminology

As its name clearly indicates, IP NAT is all about the *translation* of IP addresses. When datagrams pass between the private network of an organization and the public Internet, the NAT router changes one or more of the addresses in these datagrams. This translation means that every transaction in a NAT environment involves not just a source address and a destination address, but also potentially multiple addresses for each of the source and destination.

In order to make clearer the explanation of how NAT operates, several special designations have been developed to refer to the different types of addresses that can be found in an IP datagram when NAT is used. Unfortunately, the terminology used for addressing in NAT can be confusing, because it's hard to visualize what the differences are between the (often similar-sounding) names. However, without knowing what these addresses mean, a proper understanding of NAT operation is impossible.

The first way that addresses are differentiated is based on where the device is in the network that the address is referring to, as follows:

**Inside Address** Any device on the organization's private network that is using NAT is said to be on the inside network. Thus, any address that refers to a device on the local network in any form is called an *inside address*.

**Outside Address** The public Internet—that is, everything outside the local network—is considered the outside network. Any address that refers to a public Internet device is an *outside address*.

**KEY CONCEPT** In NAT, the terms *inside* and *outside* are used to identify the location of devices. *Inside addresses* refer to devices on the organization's private network. *Outside addresses* refer to devices on the public Internet.

An inside device always has an inside address; an outside device always has an outside address. However, there are two different ways of addressing either an inside or an outside device, depending on the part of the network in which the address appears in a datagram:

**Local Address** This term describes an address that appears in a datagram on the inside network, *whether it refers to an inside or outside address*.

**Global Address** This term describes an address that appears in a datagram on the outside network, again *whether it refers to an inside or outside address*.

**KEY CONCEPT** In NAT, the terms *local* and *global* are used to indicate in what network a particular address appears. *Local addresses* are used on the organization's private network (whether to refer to an inside device or an outside device). *Global addresses* are used on the public Internet (again, whether referring to an inside or outside device).

This is a bit confusing, so I will try to explain further. The NAT translating router has the job of interfacing the inside network to the outside network (the Internet). Inside devices need to be able to talk to outside devices and vice versa, but inside devices can use only addressing consistent with the local network-addressing scheme. Similarly, outside devices cannot use local addressing. Thus, both inside and outside devices can be referred to with local or global address versions. This yields four different specific address types:

**Inside Local Address** An address of a device on the local network, expressed using its normal local device representation. So, for example, if you had a client on a network using the 10.0.0.0 private address block and assigned it address 10.0.0.207, this would be its *inside local* address.

**Inside Global Address** This is a global, publicly routable IP address that's used to represent an inside device to the outside world. In a NAT configuration, *inside global* addresses are those real IP addresses assigned to an organization for use by the NAT router. Let's say that device 10.0.0.207 wants to send an HTTP request to an Internet server located at address 204.51.16.12. It forms the datagram using 10.0.0.207 as the source address. However, if this datagram is sent out to the Internet as is, the server cannot reply back because 10.0.0.207 is not a publicly routable IP address. So the NAT router will translate 10.0.0.207 in the datagram into one of the organization's registered IP addresses, let's say, 194.54.21.10. This is the *inside global* address that corresponds to 10.0.0.207. It will be used as the destination when the server sends its HTTP response. Note that, in some situations, the inside local address and outside local address may be the same.

**Outside Global Address** An address of an external (public Internet) device as it is referred to on the global Internet. This is basically a regular, publicly registered address of a device on the Internet. In the previous example, 204.51.16.12 is an *outside global* address of a public server.

**Outside Local Address** An address of an external device as it is referred to by devices on the local network. In some situations, this may be identical to the *outside global* address of that outside device.

Phew, it's still confusing, isn't it? Let's try another way of looking at this. Of these four addresses, two types are the addresses as they are known natively by either an inside or outside device, while the other two are translated addresses. Here is a summary:

**Inside Device Designations** For an inside device, the *inside local* address is its normal, or native, address. The *inside global* address is a translated address used to represent the inside device on the outside network, when necessary.

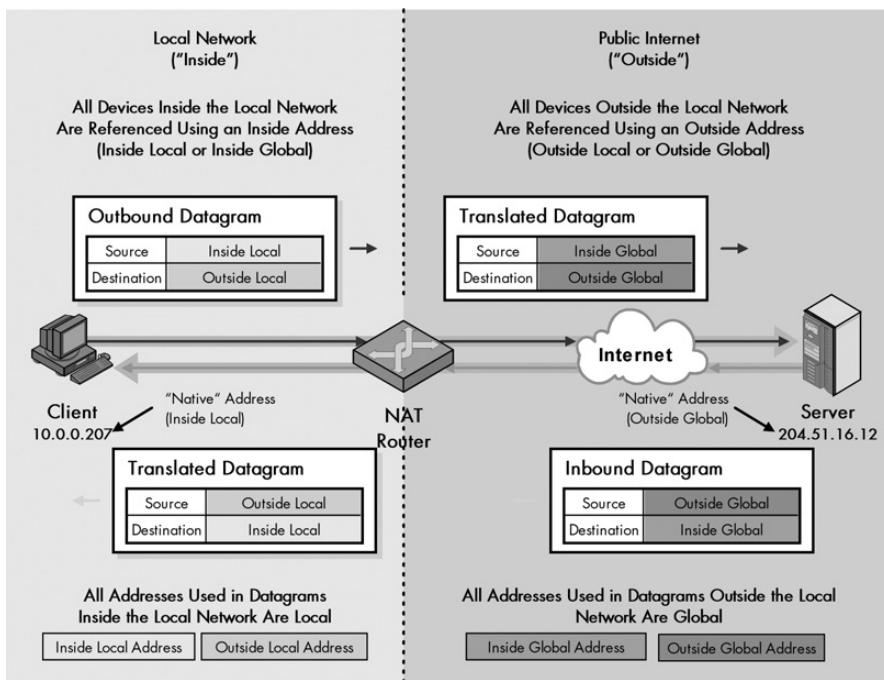
**Outside Device Designations** For an outside device, the *outside global* address is its normal, or native, address. The *outside local* address is a translated address used to represent the outside device on the inside network, when necessary.

So what NAT does then is translate the identity of either inside or outside devices from local representations to global representations and vice versa. Which addresses are changed and how will depend on the specific type of NAT employed. For example, in traditional NAT, inside devices refer to outside devices using their proper (global) representation, so the outside global and outside local addresses of these outside devices are the same.

**KEY CONCEPT** A NAT router translates *local* addresses to *global* ones and vice versa. Thus, an *inside local* address is translated to an *inside global* address (and vice versa) and an *outside local* address is translated to an *outside global* address (and vice versa).

And after all that, it's still confusing! One of the big problems is that the words *inside* and *local* are somewhat synonymous, as are *outside* and *global*, yet they mean different things in NAT. And the typical paradox in trying to explain networking concepts rears its ugly head here again: I wanted to define these addresses to make describing NAT operation easier, but find myself wanting to use an example of NAT operation to clarify how the addresses are used.

Even after writing this material I find these terms confusing, so I created Figure 28-1, which shows this same terminology in graphical form and may be of some help. That diagram is also used as a template for the illustrations of each of the different types of NAT in the rest of this chapter. As you read about NAT operation, look back here if you want to double-check the meaning of address types. Don't get discouraged if it takes a couple of times to get the addresses straight.



**Figure 28-1: IP Network Address Translation (NAT) terminology** Hopefully this diagram will help you to better understand the whole "inside/outside/local/global" thing.

## IP NAT Static and Dynamic Address Mappings

NAT allows you to connect a private (inside) network to a public (outside) network such as the Internet by using an address translation algorithm implemented in a router that connects the two. Each time a NAT router encounters an IP datagram that crosses the boundary between the two networks, it must translate addresses as appropriate. But how does it know what to translate and what to use for the translated address?

The NAT software in the router must maintain a *translation table* to tell it how to operate. The translation table contains information that maps the *inside local* addresses of internal devices (their regular addresses) to *inside global* address representations (the special public addresses used for external communication). It may also contain mappings between *outside global* addresses and *outside local* addresses for inbound transactions, if appropriate.

There are two basic ways that entries can be added to the NAT translation table: statically or dynamically.

### Static Mappings

A static mapping represents a permanent, fixed relationship defined between a *global* and a *local* representation of the address of either an *inside* or an *outside* device. For example, you can use a static translation if you want the internal device with an *inside local* address of 10.0.0.207 to *always* use the *inside global* address of 194.54.21.10. Whenever 10.0.0.207 initiates a transaction with the Internet, the NAT router will replace that address with 194.54.21.10.

### Dynamic Mappings

With dynamic mapping, *global* and *local* address representations are generated automatically by the NAT router, which is used as needed and then discarded. The most common way that this is employed is in allowing a *pool* of *inside global* addresses to be shared by a large number of *inside* devices.

For example, say you were using dynamic mapping with a pool of *inside global* addresses available from 194.54.21.1 through 194.54.21.20. When 10.0.0.207 sent a request to the Internet, it would not automatically have its source address replaced by 194.54.21.10. One of the 20 addresses in the pool would be chosen by the NAT router. The router would then watch for replies back using that address and translate them back to 10.0.0.207. When the session was completed, it would discard the entry to return the *inside global* address to the pool.

### Choosing Between Static and Dynamic Mapping

The trade-offs between static and dynamic NAT mappings are pretty much the same as they always are when the choice is between static and dynamic. For example, the same issues arises in Address Resolution Protocol (ARP) caching; see Chapter 13.

Static mappings are permanent and therefore ideal for devices that need to be always represented with the same public address on the outside network. They may also be used to allow inbound traffic to a particular device; that is, they can be used

for transactions initiated on the public network that send to a special server on the inside network. However, they require manual setup and maintenance, and they don't allow IP sharing on the internal network.

Dynamic mapping is normally used for regular clients in order to facilitate public IP address sharing—a prime goal of most NAT implementations. It is more complicated than static mapping, but once you set it up, it's automatic.

It is possible to mix dynamic and static mapping on the same system, of course. You can designate certain devices that are statically mapped and let the rest use dynamic mapping. You just have to make sure that the static mappings don't overlap with the pool used for dynamic assignment.

Incidentally, another way you can perform dynamic mapping of global and local addressing is through domain name resolution using the Domain Name System (DNS; see Chapter 52). This is particularly common when external devices access internal hosts using bidirectional NAT (inbound transactions). Since hosts on the public Internet know nothing about the organization's private network, they issue a request for the DNS name of the device they want to access. This causes the generation of a NAT translation entry that maps the inside local public address of the host to an inside global address for use by those outside the network. See the description of bidirectional NAT later in this chapter for more details on how this works.

## IP NAT Unidirectional (Traditional/Outbound) Operation

Now it's time to get down to the nitty gritty of how it works. There are many different flavors of NAT, and four common ones are covered in this chapter. It makes sense to start by looking at the original variety of NAT described in RFC 1631. This is the simplest NAT method, and therefore the easiest one to explain.

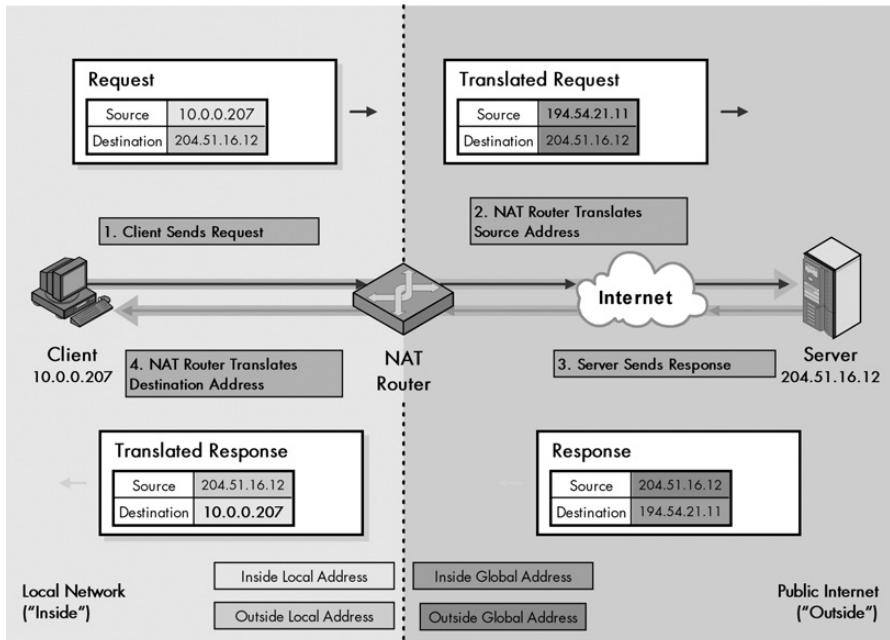
NAT was designed to allow hosts on a private network to share public IP addresses in accessing an Internet. Since most hosts are clients that initiate transactions, NAT was designed under the assumption that a client/server request/response communication would begin with a datagram sent from the *inside* network to the *outside*. For this reason, this first type of NAT is sometimes called *unidirectional* or *outbound* NAT. Since it is the oldest flavor, it is also now called *traditional* NAT to differentiate it from newer varieties.

To show how unidirectional NAT works, I will use an example. Let's assume the inside network has 250 hosts that use private (inside local) addresses from the 10.0.0.0/8 address range (which I selected because it has small numbers!). These hosts use dynamic NAT sharing a pool of 20 inside global addresses from 194.54.21.1 through 194.54.21.20.

In this example, device 10.0.0.207 wants to access the World Wide Web server at public address 204.51.16.12. Table 28-1 shows the four basic steps that are involved in this (simplified) transaction. I did this in table form so I could show you explicitly what happens to the addresses in both the request datagram (in steps 1 and 2) and the response datagram (steps 3 and 4). I have also highlighted the translated address values for clarity, and provided Figure 28-2, which shows the process graphically.

**Table 28-1:** Operation of Unidirectional (Traditional/Outbound) NAT

Step #	Description	Datagram Type	Datagram Source Address	Datagram Destination Address
1	<b>Inside Client Generates Request and Sends to NAT Router:</b> Device 10.0.0.207 generates an HTTP request that is eventually passed down to IP and encapsulated in an IP datagram. The source address is itself, 10.0.0.207, and the destination is 204.51.16.12. The datagram is sent to the NAT-capable router that connects the organization's internal network to the Internet.	Request (from inside client to outside server)	10.0.0.207 (inside local)	204.51.16.12 (outside local)
2	<b>NAT Router Translates Source Address and Sends to Outside Server:</b> The NAT router realizes that 10.0.0.207 is an <i>inside local</i> address and knows it must substitute an <i>inside global</i> address in order to let the public Internet destination respond. It consults its pool of addresses and sees the next available one is 194.54.21.11. It changes the source address in the datagram from 10.0.0.207 to 194.54.21.11. The destination address is not translated in traditional NAT. In other words, the <i>outside local</i> address and <i>outside global</i> address are the same. The NAT router puts the mapping from 10.0.0.207 to 194.54.21.11 into its translation table. It sends out the modified datagram, which is eventually routed to the server at 204.51.16.12.		194.54.21.11 (inside global)	204.51.16.12 (outside global)
3	<b>Outside Server Generates Response and Sends Back to NAT Router:</b> The server at 204.51.16.12 generates an HTTP response. It has no idea that NAT was involved; it sees 194.54.21.11 in the request sent to it, so that's where it sends back the response. It is then routed back to the original client's NAT router.	Response (from outside server to inside client)	204.51.16.12 (outside global)	194.54.21.11 (inside global)
4	<b>NAT Router Translates Destination Address and Delivers Datagram to Inside Client:</b> The NAT router sees 194.54.21.11 in the response that arrived from the Internet. It consults its translation table and knows this datagram is intended for 10.0.0.207. This time, the destination address is changed but not the source. It then delivers the datagram back to the originating client.		204.51.16.12 (outside local)	10.0.0.207 (inside local)



**Figure 28-2: Operation of unidirectional (traditional/outbound) NAT** You can see the four steps in this process by following the steps in clockwise order. Translated addresses are shown in bold. Refer to Table 28-1 and Figure 28-1 for an explanation of the four address types.

As you can see, this really isn't rocket science, and it's fairly easy to understand what is going on as soon as you get used to the terminology and concepts. In unidirectional NAT, the source address is translated on outgoing datagrams and the destination address is translated on incoming ones. Traditional NAT supports only this sort of outbound transaction, which is started by a device on the inside network. It cannot handle a device that sends a request to a private address on the public Internet.

**KEY CONCEPT** In unidirectional (traditional) NAT, the NAT router translates the source address of an outgoing request from inside local to inside global form. It then transforms the destination address of the response from inside global to inside local. The outside local and outside global addresses are the same in both request and reply.

Also note that even though I am focusing on the changes that the NAT router makes to addresses, it also must make other changes to the datagram. Changing any field in the IP header means that the IP Header Checksum field will need to be recalculated. User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) checksums need to be recalculated, and depending on the nature of the data in the datagram, other changes may also be required. I discuss these issues in the section on NAT compatibility issues, at the end of this chapter.

Incidentally, this simplified example assumes the existence of just one router between the private and public networks. It is possible to have more than one router between these networks. If this configuration is used, however, it is essential

that they both use the same translation table. Otherwise, if Router R1 processes the request, but Router R2 receives the response, Router R2 won't know how to translate back the destination address on the incoming datagram. This makes dynamic mapping extremely difficult: Routers would have to coordinate their address mappings.

## IP NAT Bidirectional (Two-Way/Inbound) Operation

Traditional NAT is designed to handle only outbound transactions; clients on the local network initiate requests and devices on the Internet and send back responses. However, in some circumstances, we may want to go in the opposite direction. That is, we may want to have a device on the outside network initiate a transaction with one on the inside. To permit this, we need a more capable type of NAT than the traditional version. This enhancement goes by various names, most commonly Bidirectional NAT, Two-Way NAT, and Inbound NAT. All of these convey the concept that this kind of NAT allows both the type of transaction you saw in the previous topic and also transactions initiated from the outside network.

Performing NAT on inbound transactions is more difficult than conventional outbound NAT. To understand why, remember that the network configuration when using NAT is inherently *asymmetric*: The inside network generally knows the IP addresses of outside devices, since they are public, but the outside network doesn't know the private addresses of the inside network. Even if they did know them, they could never be specified as the target of an IP datagram initiated from outside since they are not routable—there would be no way to get them to the private network's local router.

Why does this matter? Well, consider the case of outbound NAT from Device A on the inside network to Device B on the outside. The local client, A, always starts the transaction, so Device A's NAT router is able to create a mapping between Device A's inside local and inside global address during the request. Device B is the recipient of the already-translated datagram, so the fact that Device A is using NAT is hidden. Device B responds back, and the NAT router does the reverse translation without Device B ever even knowing NAT was used for Device A.

Now let's look at the inbound case. Here, Device B is trying to send to Device A, which is using NAT. Device B can't send to Device A's private (inside local) address. It needs Device A's inside global address in order to start the ball rolling. However, Device A's NAT router isn't proximate to Device B. In fact, Device B probably doesn't even know the identity of Device A's NAT router!

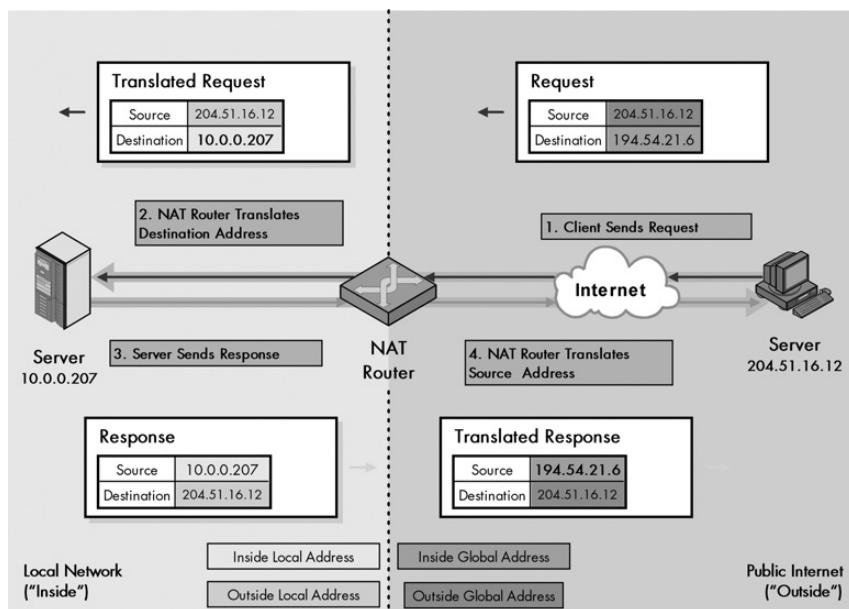
This leaves only two methods. One is to use static mapping for devices like servers on the inside network that need to be accessed from the outside. When static mapping is employed, the global address of the device that is using the static mapping will be publicly known, which solves the "where do I send my request to" problem.

The other solution is to make use of DNS. As explained in detail in the section on DNS (see Part III-1), this protocol allows requests to be sent as names instead of IP addresses. The DNS server translates these names to their corresponding addresses. It is possible to integrate DNS and NAT so they work together. This process is described in RFC 2694, "DNS Extensions to Network Address Translators (DNS\_ALG)."

In this technique, an outside device can make use of dynamic mapping. The basic process (highly simplified) is as follows:

1. The outside device sends a DNS request using the name of the device on the inside network it wishes to reach. For example, it might be www.ilikenat.com.
2. The DNS server for the internal network resolves the www.ilikenat.com name into an *inside local* address for the device that corresponds to this DNS entry.
3. The *inside local* address is passed to NAT and used to create a dynamic mapping between the *inside local* address of the server being accessed from the outside, and an *inside global* address. This mapping is put into the NAT router's translation table.
4. When the DNS server sends back the name resolution, it tells the outside device the *inside global* (public) address mapped in the previous step, not the *inside local* (private) address of the server being sought.

Once the inside global address of the device on the inside network is known by the outside device, the transaction can begin. Let's use the same example as in the previous section, but let's reverse it so that the outside device 204.51.16.12 is initiating a request (and is thus now the *client*) to inside device 10.0.0.207 (which is the *server*). Let's say that either static mapping or DNS has been used so that the outside device knows the inside global address of 10.0.0.207 is actually 194.54.21.6. Table 28-2 describes the transaction in detail, and it is illustrated Figure 28-3.



**Figure 28-3: Operation of bidirectional (two-way/inbound) NAT** This figure is very similar to Figure 28-2, except that the transaction is in reverse, so start at the upper right and go counterclockwise. Translated addresses are shown in bold. Table 28-2 contains a complete explanation of the four steps. Refer to Figure 28-1 for an explanation of address types.

**Table 28-2:** Operation of Bidirectional (Two-Way/Inbound) NAT

Step #	Description	Datagram Type	Datagram Source Address	Datagram Destination Address
1	<b>Outside Client Generates Request and Sends to NAT Router:</b> Device 204.51.16.12 generates a request to the inside server. It uses the <i>inside global</i> address 194.54.21.6 as the destination. The datagram will be routed to the address's local router, which is the NAT router that services the inside network where the server is located.	Request (from outside client to inside server)	204.51.16.12 (outside global)	194.54.21.6 (inside global)
2	<b>NAT Router Translates Destination Address and Sends to Inside Server:</b> The NAT router already has a mapping from the <i>inside global</i> address to the <i>inside local</i> address of the server. It replaces the 194.54.21.6 destination address with 10.0.0.207, and performs checksum recalculations and other work as necessary. The source address is not translated. The router then delivers the modified datagram to the inside server at 10.0.0.207.		204.51.16.12 (outside local)	10.0.0.207 (inside local)
3	<b>Inside Server Generates Response and Sends Back to NAT Router:</b> The server at 10.0.0.207 generates a response, which it addresses to 204.51.16.12 since that was the source of the request to it. This is then routed to the server's NAT router.	Response (from inside server to outside client)	10.0.0.207 (inside local)	204.51.16.12 (outside local)
4	<b>NAT Router Translates Source Address and Routes Datagram to Outside Client:</b> The NAT router sees the private address 10.0.0.207 in the response and replaces it with 194.54.21.6. It then routes this back to the original client on the outside network.		194.54.21.6 (inside global)	204.51.16.12 (outside global)

As you can see, once the outside device knows the inside device's *inside global* address, you'll find that inbound NAT is very similar to outbound NAT. It just does the exact opposite translation. Instead of modifying the source address on the outbound request and the destination on the inbound response, the router changes the destination on the inbound request and the source on the outbound reply.

**KEY CONCEPT** In traditional NAT, a transaction must begin with a request from a client on the local network, but in *bidirectional* (two-way/inbound) NAT, it is possible for a device on the public Internet to access a local network server. This requires the use of either static mapping or DNS to provide to the outside client the address of the server on the inside network. Then the NAT transaction is pretty much the same as in the unidirectional case, except in reverse: The incoming request has its destination address changed from *inside global* to *inside local*; the response has its source changed from *inside local* to *inside global*.

## IP NAT Port-Based (Overloaded) Operation

Both traditional NAT and bidirectional NAT work by swapping inside network and outside network addresses as needed in order to allow a private network to access a public one. For each transaction, there is a one-to-one mapping between the *inside local* address of a device on the private network and the *inside global* address that

represents it on the public network. We can use dynamic address assignment to allow a large number of private hosts to share a small number of registered public addresses.

However, there is a potential snag here. Consider the earlier NAT example, where 250 hosts share 20 inside global (public) addresses. If 20 hosts already have transactions in progress, what happens when the 21st tries to access the Internet? There aren't any *inside global* addresses available for it to use, so it won't be able to.

Fortunately, there is a mechanism already built into TCP/IP that can help us alleviate this situation. The two TCP/IP transport layer protocols, TCP and UDP, both make use of additional addressing components called *ports*. The port number in a TCP or UDP message helps identify individual connections between two addresses. It is used to allow many different applications on a TCP/IP client and server to talk to each simultaneously, without interference. For example, you use this capability when you open multiple browser windows to access more than one web page on the same site at the same time. This sharing of IP addresses among many connections is called *multiplexing*. Chapter 43, which describes TCP and UDP ports, covers all of this in much more detail.

Now let's come back to NAT. We are already translating IP addresses as we send datagrams between the public and private portions of the internetwork. What if we could also translate port numbers? Well, we can! The combination of an address and port uniquely identifies a connection. As a datagram passes from the private network to the public one, we can change not just the IP address, but also the port number in the TCP or UDP header. The datagram will be sent out with a different source address and port. The response will come back to this same address and port combination (called a *socket*) and can be translated back again.

This method goes by various names. Since it is a technique that can have multiple inside local addresses share a single inside global address, it is called *overloading* of an *inside global* address, or alternatively, just *overloaded NAT*. More elegant names that better indicate how the technique works include *Port-Based NAT*, *Network Address Port Translation (NAPT)*, and *Port Address Translation (PAT)*.

**KEY CONCEPT** *Port-Based* or overloaded NAT is an enhancement of regular NAT that allows a large number of devices on a private network to simultaneously share a single inside global address by changing the port numbers used in TCP and UDP messages.

Whatever its name, the use of ports in translation has tremendous advantages. It can allow all 250 hosts on the private network to use only 20 IP addresses—and potentially even fewer than that. In theory, you could even have all 250 share a single public IP address at once! You don't want to share so many local hosts that you run out of port numbers, but there are thousands of port numbers to choose from.

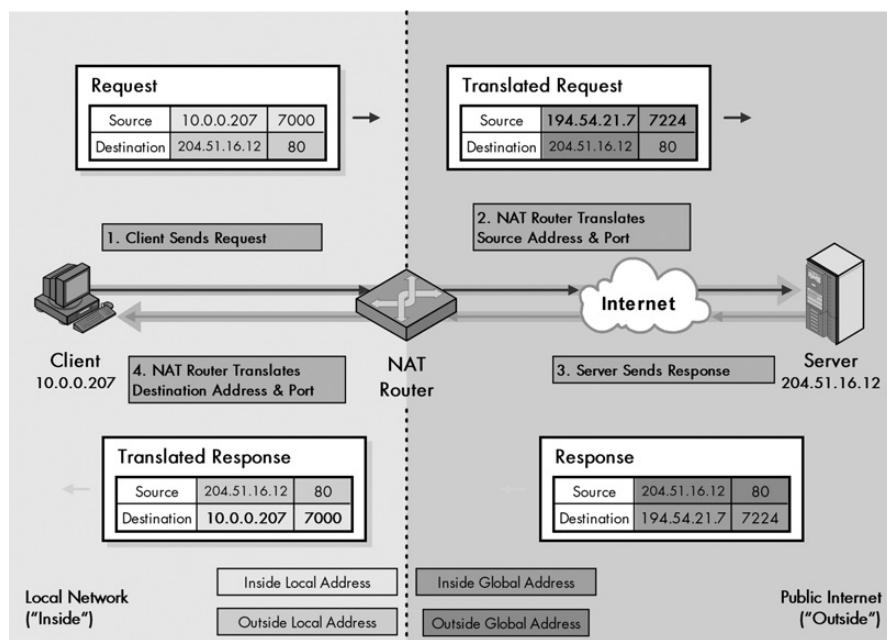
Port-Based NAT requires a router that is programmed to make the appropriate address and port mappings for datagrams as it transfers them between networks. The disadvantages of the method include this greater complexity, and also the potential for more compatibility issues (such as with applications like FTP), since you must now watch for port numbers at higher layers and not just IP addresses.

The operation of NAPT/PAT is very similar to the way regular NAT works, except that port numbers are also translated. For a traditional outbound transaction, the source port number is changed on the request at the same time that the source address is modified; the destination port number is modified on the response with the destination address.

Let's consider again the example you looked at in the topic on traditional NAT, but this time in a PAT environment. Device 10.0.0.207 was one of 250 hosts on a private network accessing the World Wide Web server at 204.51.16.12. Let's say that because PAT is being used, in order to save money, all 250 hosts are sharing a single *inside global* address, 194.54.21.7, instead of a pool of 20. The transaction would proceed as described in Table 28-3 and illustrated in Figure 28-4.

**KEY CONCEPT** In Port-Based NAT, the NAT router translates the source address and port of an outgoing request from inside local to inside global form. It then transforms the destination address and port of the response from inside global to inside local. The outside local and outside global addresses are the same in both request and reply.

One other issue related to NAPT/PAT is worth mentioning: It assumes that all traffic uses either UDP or TCP at the transport layer. Although this is generally the case, it may not always be true. If there is no port number, port translation cannot be done and the method will not work.



**Figure 28-4: Operation of Port-Based (overloaded) NAT** This figure is very similar to Figure 28-4, except that the source and destination port numbers have been shown, since they are used in this type of NAT. Translated addresses and ports are in bold. Table 28-3 contains a complete explanation of the four steps in Port-Based NAT. Refer to Figure 28-1 for an explanation of address types.

**Table 28-3: Operation of Port-Based (Overloaded) NAT**

Step #	Description	Datagram Type	Datagram Source Address:Port	Datagram Destination Address:Port
1	<b>Inside Client Generates Request and Sends to NAT Router:</b> Device 10.0.0.207 generates an HTTP request to the server at 204.51.16.12. The standard server port for WWW is 80, so the destination port of the request is 80; let's say the source port on the client is 7000. The datagram is sent to the NAT-capable router that connects the organization's internal network to the Internet.	Request (from inside client to outside server)	10.0.0.207:7000 (inside local)	204.51.16.12:80 (outside local)
2	<b>NAT Router Translates Source Address and Port and Sends to Outside Server:</b> The NAT router realizes that 10.0.0.207 is an <i>inside local</i> address and knows it must substitute an <i>inside global</i> address. Here though, there are multiple hosts sharing the single <i>inside global</i> address 194.54.21.7. Let's say that port 7000 is already in use for that address by another private host connection. The router substitutes the <i>inside global</i> address and also chooses a new source port number, say 7224, for this request. The destination address and port are not changed.  The NAT router puts the address and port mapping into its translation table. It sends the modified datagram out, which arrives at the server at 204.51.16.12.		194.54.21.7:7224 (inside global)	204.51.16.12 (outside global)
3	<b>Outside Server Generates Response and Sends Back to NAT Router:</b> The server at 204.51.16.12 generates an HTTP response. It has no idea that NAT was involved; it sees an address of 194.54.21.7 and port of 7224 in the request sent to it, so it sends back to that address and port.	Response (from outside server to inside client)	204.51.16.12:80 (outside global)	194.54.21.7:7224 (inside global)
4	<b>NAT Router Translates Destination Address and Port and Delivers Datagram to Inside Client:</b> The NAT router sees the address 194.54.21.7 and port 7224 in the response that arrived from the Internet. It consults its translation table and knows this datagram is intended for 10.0.0.207, port 7000. This time, the destination address and port are changed but not the source. The router then delivers the datagram back to the originating client.		204.51.16.12:80 (outside local)	10.0.0.207:7000 (inside local)

## IP NAT Overlapping/Twice NAT Operation

All three of the versions of NAT discussed so far—traditional, bidirectional, and Port-Based—are normally used to connect a network using private, nonroutable addresses to the public Internet, which uses unique, registered, routable addresses. With these kinds of NAT, there will normally be no overlap between the address

spaces of the inside and outside network, since the former are private and the latter are public. This enables the NAT router to be able to immediately distinguish inside addresses from outside addresses just by looking at them.

In the examples you've seen so far, the inside addresses were all from the RFC 1918 block 10.0.0.0. These can't be public Internet addresses, so the NAT router knew any address referenced by a request from the inside network within this range was a local reference within the inside network. Similarly, any addresses outside this range are easy to identify as belonging to the outside world.

There are circumstances, however, in which there may indeed be an overlap between the addresses used for the inside network, and the addresses used for part of the outside network. Consider the following cases:

**Private Network-to-Private Network Connections** The example network using 10.0.0.0 block addresses might want to connect to another network using the same method. This situation might occur if two corporations merged and happened to be using the same addressing scheme (and there aren't that many private IP blocks, so this isn't that uncommon).

**Invalid Assignment of Public Address Space to Private Network** Some networks might have been set up, not by using a designated private address block, but rather by using a block containing valid Internet addresses. For example, suppose an administrator decided that the network he was setting up would never be connected to the Internet (ha!), and numbered the whole thing using 18.0.0.0 addresses, which belong to the Massachusetts Institute of Technology (MIT). Then later, this administrator's shortsightedness would backfire when the network did indeed need to be connected to the Internet.

**Stale Public Address Assignment** Company A might have been using a particular address block for years that was reassigned or reallocated for whatever reason to Company B. Company A might not want to go through the hassle of renumbering its network, and would then keep its addresses, even while Company B started using them on the Internet.

What these situations all have in common is that the inside addresses used in the private network *overlap* with addresses on the public network. When a datagram is sent from within the local network, the NAT router can't tell if the intended destination is within the inside network or the outside network. For example, if you want to connect host 10.0.0.207 in the private network to host 10.0.0.199 in a different network, and you put 10.0.0.199 in the destination of the datagram and send it, how does the router know if you mean 10.0.0.199 on your own local network or the remote one? For that matter, you might need to send a request to 10.0.0.207 in the other private network, your own address! Take the network that was numbered with MIT's address block. How does the router know when a datagram is actually being sent to MIT as opposed to another device on the private network?

The solution to this dilemma is to use a more sophisticated form of NAT. The other versions you have seen so far always translate either the source address *or* the destination address as a datagram passes from the inside network to the outside network or vice versa. To cope with overlapping addresses, we must translate both

the source address *and* the destination address on each transition from the inside to the outside or the other direction. This technique is called *overlapping NAT* in reference to the problem it solves, or *Twice NAT* due to how it solves it. (Incidentally, despite the latter name, regular NAT is *not* called Once NAT.)

Twice NAT functions by creating a set of mappings not only for the private network the NAT router serves, but also for the overlapping network (or networks) that conflict with the inside network's address space. In order for this to function, Twice NAT relies on the use of DNS, just as does bidirectional NAT. This lets the inside network send requests to the overlapping network in a way that can be uniquely identified. Otherwise, the router can't tell what overlapping network our inside network is trying to contact.

Let's try a new example. Suppose the network has been improperly numbered so that it is not in the 10.0.0.0 private block but in the 18.0.0.0 block used by MIT. A client on our private network, 18.0.0.18, wants to send a request to the server www.twicenat.mit.edu, which has the address 18.1.2.3 at MIT. The client can't just make a datagram with 18.1.2.3 as the destination and send out, as the router will think it's on the local network and not route it. Instead, 18.0.0.18 uses a combination of DNS and NAT to get the outside device address, as follows:

1. The client on the local network (18.0.0.18) sends a DNS request to get the address of www.twicenat.mit.edu.
2. The (Twice-NAT compatible) NAT router serving the local network intercepts this DNS request. It then consults its tables to find a special mapping for this outside device. Let's say that it is programmed to translate www.twicenat.mit.edu into the address 172.16.44.55. This is a private, nonroutable RFC 1918 address.
3. The NAT router returns this value, 172.16.44.55, to the source client, which uses it for the destination.

Once the client has the translated address, it initiates a transaction just as before. NAT will now perform translation of the inside devices and the outside devices as well. The outside device address must be translated because the inside device is using 172.16.44.55, which isn't a valid address for the server it is trying to reach. The inside device address must still be translated as in regular NAT because 18.0.0.18 is not a valid public address for you. It may refer to a real machine in MIT and you aren't supposed to be using it on the Internet!

Let's say that you are still using the pool of 20 inside global addresses from 194.54.21.1 through 194.54.21.20 for inside addresses, and let's further suppose that the NAT router chooses 194.54.21.12 for this particular exchange. The transaction sequence would be roughly as described in Table 28-4 and illustrated in Figure 28-5.

Overlapping NAT is used in situations where both the source and destination addresses in a datagram are private addresses or otherwise cannot be used regularly on the public Internet. In this case, unlike with the other types of NAT, the NAT router translates both the source and destination addresses of incoming and outgoing datagrams. On outgoing messages, *inside local* addresses are changed to *inside global* and *outside local* to *outside global*; on incoming messages, *inside global* addresses are changed to *inside local* and *outside global* addresses are changed to *outside local*.

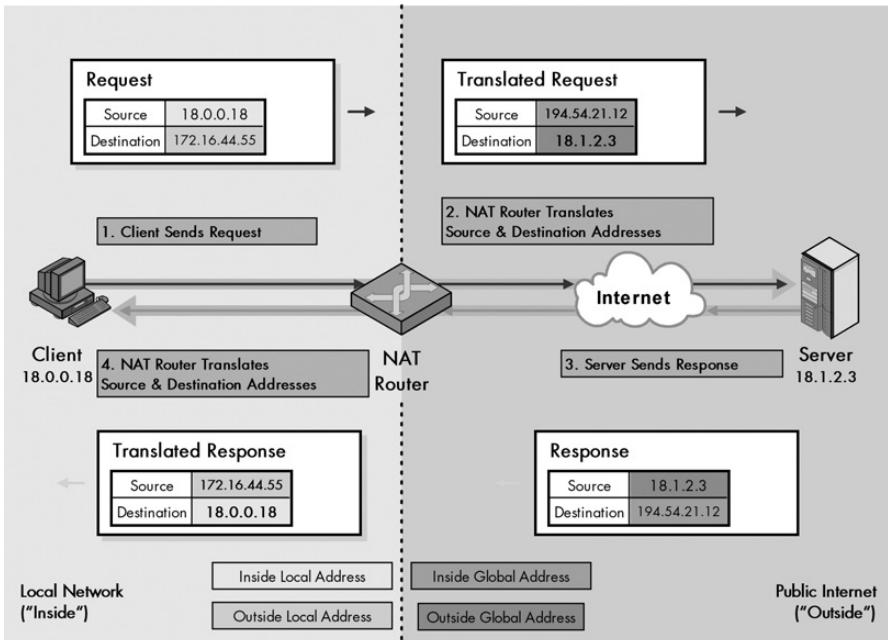
**Table 28-4:** Operation of Overlapping NAT/Twice NAT

<b>Step #</b>	<b>Description</b>	<b>Datagram Type</b>	<b>Datagram Source Address</b>	<b>Datagram Destination Address</b>
1	<b>Inside Client Generates Request and Sends to NAT Router:</b> Device 18.0.0.18 generates a request using the destination 172.16.44.55, which it got from the (NAT-intercepted) DNS query for www.twicenat.mit.edu. The datagram is sent to the NAT router for the local network.	Request (from inside client to outside server)	18.0.0.18 (inside local)	172.16.44.55 (outside local)
2	<b>NAT Router Translates Source Address and Destination Address and Sends to Outside Server:</b> The NAT router makes two translations. First, it substitutes the 18.0.0.18 address with a publicly registered address, which is 194.54.21.12 for this example. It then translates the bogus 172.16.44.55 back to the real MIT address for www.twicenat.mit.edu. It routes the datagram to the outside server.		194.54.21.12 (inside global)	18.1.2.3 (outside global)
3	<b>Outside Server Generates Response and Sends Back to NAT Router:</b> The MIT server at 18.1.2.3 generates a response and sends it back to 194.54.21.12, which causes it to arrive back at the NAT router.	Response (from outside server to inside client)	18.1.2.3 (outside global)	194.54.21.12 (inside global)
4	<b>NAT Router Translates Source Address and Destination Address and Delivers Datagram to the Inside Client:</b> The NAT router translates back the destination address to the actual address that's being used for the inside client, as in regular NAT. It also substitutes back in the 172.16.44.55 value it is using as a substitute for the real address of www.twicenat.mit.edu.		172.16.44.55 (outside local)	18.0.0.18 (inside local)

As you can see, in this example, the *outside local* and *outside global* addresses are different, unlike in the preceding NAT examples. Twice NAT can also handle an inbound transaction by watching for datagrams coming in from the Internet that overlap with the addresses used on the local network and doing double substitutions as required.

## IP NAT Compatibility Issues and Special Handling Requirements

In a perfect world NAT could be made transparent to the devices using it. We would like to be able to have a NAT router change IP addresses in request datagrams as they leave the network and change them back in responses that come back, and have none of the hosts be any wiser. Unfortunately, this isn't a perfect world.



**Figure 28-5: Operation of Overlapping NAT/Twice NAT** This figure is very similar to Figure 28-2, except that as you can see, the NAT router translates both source and destination addresses each time (shown in bold). Table 28-4 contains a complete explanation of the four steps in overlapping NAT. Refer to Figure 28-1 for an explanation of address types.

It is not possible for NAT to be completely transparent to the devices that use it. There are potential compatibility problems that arise if NAT doesn't perform certain functions. These functions go beyond simply swapping IP addresses and possibly port numbers in the IP header. The main problem is that even though IP addresses are supposedly the domain of IP, they are really used by other protocols as well, both at the network layer and in higher layers. When NAT changes the IP address in an IP datagram, it must often also change addresses in other places to make sure that the addresses in various headers and payloads still match.

These compatibility issues require that even though NAT should theoretically work only at the level of IP at the network layer, in practical terms, NAT routers must be aware of many more protocols and perform special operations as required. Some are required for all datagrams that are translated; others only apply to certain datagrams and not others. And even when these techniques are added to NAT routers, some things still may not work properly in a NAT environment.

Let's take a look at some of the main issues and requirements:

**TCP and UDP Checksum Recalculations** Changing the IP addresses in the IP header means that the IP header checksum must be calculated. Since both UDP and TCP also have checksums, and these checksums are computed over a pseudo header that contains the IP source and destination address as well, they too, must be recalculated each time a translation is made.

**ICMP Manipulations** Since NAT works so intimately with IP headers, and since IP is closely related to its “assistant” protocol the Internet Control Management Protocol (ICMP; see Chapter 31), NAT must also look for certain ICMP messages and make changes to addresses contained within them. Many ICMP messages, such as Destination Unreachable and Parameter Problem, contain the original IP header of the datagram that lead to the ICMP message as data. Since NAT is translating addresses in IP headers, it must watch for these messages and translate addresses in included headers as required.

**Applications That Embed IP Addresses** A number of TCP/IP applications embed IP addresses within the actual application data payload. The most notorious example of this is FTP, which actually sends address and port assignments as text information in datagrams between devices during a connection. In order for NAT to support FTP, it must be specifically programmed with algorithms to look for this information and make changes as needed. The level of complication can go even beyond this. Consider what happens when an FTP message containing these text addresses or port numbers is *fragmented*. Part of the address that will be translated may be in two different IP datagrams and may be hard to recognize!

**Additional Issues with Port Translation** When Port-Based NAT (PAT) is used, the previous issues that apply to addresses now apply to ports as well, making even more work for the router to perform.

**Cascading Impact of Changes to Address or Port Numbers** Take the example of an FTP datagram encoding an IP address that NAT must change. The address being substituted might require more characters than the original; in the first example, 10.0.0.207 (10 ASCII characters) is replaced by 194.54.21.11 (12 ASCII characters). Making this substitution changes the size of the payload! This means that TCP sequence numbers (see Chapter 46) also must be modified. In these situations, NAT itself is supposed to take care of any additional work that may be required.

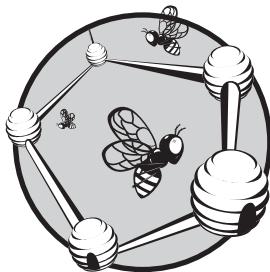
**Problems with IPsec** When IPsec is used in transport mode, both the Authentication Header (AH) and Encapsulating Security Payload (ESP) protocols use an integrity check that is based on the value of the entire payload. When NAT tries to update the TCP or UDP checksum in the IP datagram, this changes the value of data that the receiving device uses in performing the AH or ESP integrity check. The check will fail. Thus, NAT can’t be used in IPsec transport mode. It may still work in tunnel mode, but there can be complications here as well.

Most NAT implementations do take at least some of the previous issues into account. Certainly, common applications like FTP are widely supported by NAT routers, or no one would want to use NAT. That said, there might be some applications that will not work over NAT. The fact that NAT really isn’t transparent and must do these extra sorts of “hacks” to other protocol headers and even payloads is a big part of the reason why many people refer to NAT as a kludge; elegant solutions don’t have so many special cases that need special handling.



# 29

## IP SECURITY (IPSEC) PROTOCOLS



One of the weaknesses of the original Internet Protocol (IP) is that it lacks any sort of general-purpose mechanism for ensuring the authenticity and privacy of data as it is passed over the internetwork. Since IP datagrams must usually be routed between two devices over unknown networks, any information in them is subject to being intercepted and even possibly changed. With the increased use of the Internet for critical applications, security enhancements were needed for IP. To this end, a set of protocols called *IP Security* or *IPsec* was developed.

In this chapter, I provide a brief description of IPsec concepts and protocols. I begin with an overview of IPsec, including a discussion of the history of the technology and a definition of the standards. I describe the main components and protocols of the IPsec suite and its different architectures and methods for implementation. I then move to actually discussing how IPsec works, beginning with a description of the two IPsec modes (transport and tunnel) and how they differ. I describe security associations and related

constructs such as the Security Parameter Index (SPI). The last three topics cover the three main IPsec protocols: IPsec Authentication Header (AH), IPsec Encapsulating Security Payload (ESP), and the IPsec Internet Key Exchange (IKE).

**NOTE** *IPsec was initially developed with IPv6 in mind, but has been engineered to provide security for both IPv4 and IPv6 networks, and operation in both versions is similar. There are some differences in the datagram formats used for AH and ESP. These differences depend on whether you use IPsec in IPv4 or IPv6, because the two versions have different datagram formats and addressing. I highlight these differences where appropriate.*

## IPsec Overview, History, and Standards

The big problem with the original IP version (IPv4) is the pending exhaustion of its address space. This situation arose due to the rapid expansion of the Internet beyond anyone's expectations when IPv4 was developed. This same mismatch between how the Internet was when IPv4 was created and how it is now has led to another major problem with IP: the lack of a definitive means of ensuring security on IP internetworks.

The security problem arose because 25 years ago, the Internet was tiny and relatively private. Today it is enormous and truly public. As the Internet has grown, the need for security has grown with it. Consider that TCP/IP and the early Internet precursors were developed as very small networks used by government researchers at the United States Defense Advanced Research Projects Agency (*DARPA* or *ARPA*). People who were well known and would generally have had security clearance controlled all the hardware. In such a network, you don't need to build security in to the protocols—you build it into the building! It's easier to use locks and guards to ensure security than fancy encryption. The easiest way to keep someone from snooping or tampering with data on the network is simply to deny them access to the hosts that connect to the network.

This worked fine at first when there were only a few dozen machines on the Internet. And even when the Internet first started to grow, it was used pretty much only to connect together researchers and other networking professionals. New sites were added to the network slowly at first, and at least someone knew the identity of each new site added to the growing internetwork. However, as the Internet continued to increase in size and was eventually opened to the public, maintaining security of the network as a whole became impossible. Today, the “great unwashed masses” are on the Internet. Many routers—owned by “who knows” and administered by “who knows”—stand between you and most other devices you want to connect with. You cannot assume that the data you send or receive is secure.

A number of methods have evolved over the years to address the need for security. Most of these are focused at the higher layers of the OSI protocol stack in order to compensate for IP's lack of security. These solutions are valuable for certain situations, but they can't be generalized easily because they are particular to various applications. For example, we can use Secure Sockets Layer (SSL) for certain applications like World Wide Web access or File Transfer Protocol (FTP), but there are dozens of applications that this type of security was never intended to work with.

What was really needed was a solution to allow security at the IP level so all higher-layer protocols in TCP/IP could take advantage of it. When the decision was made to develop a new version of IP (IPv6), this was the golden opportunity to

resolve not just the addressing problems in the older IPv4, but the lack of security as well. New security technology was developed with IPv6 in mind, but since IPv6 has taken years to develop and roll out, and the need for security is now, the solution was designed to be usable for both IPv4 and IPv6.

The technology that brings secure communications to the IP is called *IP Security*, commonly abbreviated *IPsec*. The capitalization of this abbreviation is variable, so you'll see IPSec and IPSEC.

### **Overview of IPsec Services and Functions**

IPsec is not a single protocol, but rather a set of services and protocols that provide a complete security solution for an IP network. These services and protocols combine to provide various types of protection. Since IPsec works at the IP layer, it can provide these protections for any higher-layer TCP/IP application or protocol without the need for additional security methods, which is a major strength. Some of the kinds of protection services offered by IPsec include the following:

- Encryption of user data for privacy
- Authentication of the integrity of a message to ensure that it is not changed en route
- Protection against certain types of security attacks, such as replay attacks
- The ability for devices to negotiate the security algorithms and keys required to meet their security needs
- Two security modes, tunnel and transport, to meet different network needs

**KEY CONCEPT** *IPsec* is a contraction of *IP Security*, and it consists of a set of services and protocols that provide security to IP networks. It is defined by a sequence of several Internet standards.

### **IPsec Standards**

Since IPsec is actually a collection of techniques and protocols, it is not defined in a single Internet standard. Instead, a collection of RFCs defines the architecture, services, and specific protocols used in IPsec. Some of the most important of these are shown in Table 29-1, all of which were published in November 1998.

**Table 29-1: Important IP Security (IPsec) Standards**

RFC Number	Name	Description
2401	Security Architecture for the Internet Protocol	The main IPsec document, describing the architecture and general operation of the technology, and showing how the different components fit together.
2402	IP Authentication Header	Defines the IPsec Authentication Header (AH) protocol, which is used for ensuring data integrity and origin verification.
2403	The Use of HMAC-MD5-96 within ESP and AH	Describes a particular encryption algorithm for use by the AH and Encapsulation Security Payload (ESP) protocols called Message Digest 5 (MD5), HMAC variant.

*(continued)*

**Table 29-1:** Important IP Security (IPsec) Standards (continued)

RFC Number	Name	Description
2404	The Use of HMAC-SHA-1-96 within ESP and AH	Describes a particular encryption algorithm for use by AH and ESP called Secure Hash Algorithm 1 (SHA-1), HMAC variant.
2406	IP Encapsulating Security Payload (ESP)	Describes the IPsec ESP protocol, which provides data encryption for confidentiality.
2408	Internet Security Association and Key Management Protocol (ISAKMP)	Defines methods for exchanging keys and negotiating security associations.
2409	The Internet Key Exchange (IKE)	Describes the IKE protocol that's used to negotiate security associations and exchange keys between devices for secure communications. Based on ISAKMP and OAKLEY.
2412	The OAKLEY Key Determination Protocol	Describes a generic protocol for key exchange.

Deployment of IPsec has only really started to take off in the last few years. A major use of the technology is in implementing virtual private networks (VPNs). It appears that the future is bright for IPsec, as more and more individuals and companies decide that they need to take advantage of the power of the Internet, while also protecting the security of the data they transport over it.

## IPsec General Operation, Components, and Protocols

IPsec isn't the only difficult topic in this book, but it is definitely a subject that baffles many. Most discussions of it jump straight to describing the mechanisms and protocols, without providing a general description of what it does and how the pieces fit together. Well, I recognized that IPsec is important, and I don't shy away from a challenge. Thus, here's my attempt to provide a framework for understanding IPsec's various bits and pieces.

So what exactly does IPsec do, and how does it do it? In general terms, it provides security services at the IP layer for other TCP/IP protocols and applications to use. What this means is that IPsec provides the tools that devices on a TCP/IP network need in order to communicate securely. When two devices (either end-user hosts or intermediate devices such as routers or firewalls) want to engage in secure communications, they set up a *secure path* between themselves that may traverse across many insecure intermediate systems. To accomplish this, they must perform (at least) the following tasks:

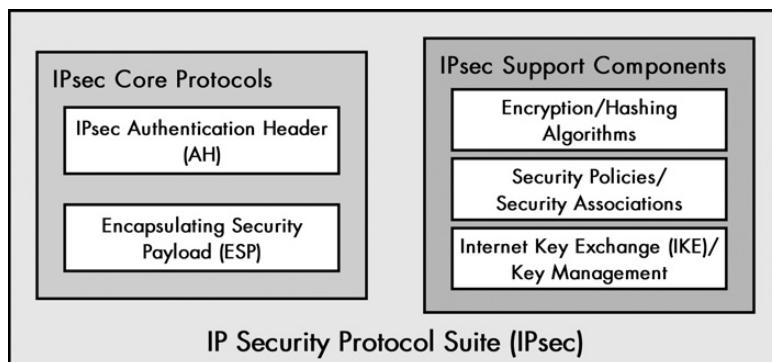
- They must agree on a set of security protocols to use so that each one sends data in a format the other can understand.
- They must decide on a specific encryption algorithm to use in encoding data.
- They must exchange keys that are used to “unlock” data that has been cryptographically encoded.
- Once this background work is completed, each device must use the protocols, methods, and keys previously agreed upon to encode data and send it across the network.

## IPsec Core Protocols

To support these activities, a number of different components make up the total package known as IPsec, as shown in Figure 29-1. The two main pieces are a pair of technologies sometimes called the *core protocols* of IPsec, which actually do the work of encoding information to ensure security:

**IPsec Authentication Header (AH)** This protocol provides authentication services for IPsec. It allows the recipient of a message to verify that the supposed originator of a message was actually fact the one that sent it. It also allows the recipient to verify that intermediate devices en route haven't changed any of the data in the datagram. It also provides protection against so-called *replay attacks*, whereby a message is captured by an unauthorized user and resent.

**Encapsulating Security Payload (ESP)** AH ensures the integrity of the data in datagram, but not its privacy. When the information in a datagram is “for your eyes only,” it can be further protected using ESP, which encrypts the payload of the IP datagram.



**Figure 29-1: Overview of IPsec protocols and components** IPsec consists of two core protocols, AH and ESP, and three supporting components.

## IPsec Support Components

AH and ESP are commonly called *protocols*, though this is another case where the use of this term is debatable. They are not really distinct protocols but are implemented as headers that are inserted into IP datagrams, as you will see. They thus do the “grunt work” of IPsec, and can be used together to provide both authentication and privacy. However, they cannot operate on their own. To function properly, they need the support of several other protocols and services (see Figure 29-1). The most important of these include the following:

**Encryption/Hashing Algorithms** AH and ESP are generic and do not specify the exact mechanism used for encryption. This gives them the flexibility to work with a variety of such algorithms and to negotiate which one to use as needed. Two common ones used with IPsec are *Message Digest 5 (MD5)* and *Secure Hash Algorithm 1 (SHA-1)*. These are also called *hashing* algorithms because they work by computing a formula called a *hash* based on input data and a key.

**Security Policies, Security Associations, and Management Methods** Since IPsec provides flexibility in letting different devices decide how they want to implement security, they require some means to keep track of the security relationships between themselves. This is done in IPsec using constructs called *security policies* and *security associations*, and by providing ways to exchange security association information.

**Key Exchange Framework and Mechanism** For two devices to exchange encrypted information, they need to be able to share keys for unlocking the encryption. They also need a way to exchange security association information. In IPsec, a protocol called the *Internet Key Exchange (IKE)* provides these capabilities.

**KEY CONCEPT** IPsec consists of a number of different components that work together to provide security services. The two main ones are protocols called the *Authentication Header (AH)* and *Encapsulating Security Payload (ESP)*, which provide authenticity and privacy to IP data in the form of special headers added to IP datagrams.

Well, that's at least a start at providing a framework for understanding what IPsec is all about and how the pieces fit together. You'll examine these components and protocols in more detail as you proceed through this chapter.

## IPsec Architectures and Implementation Methods

The main reason that IPsec is so powerful is that it provides security to IP, which is the basis for all other TCP/IP protocols. In protecting IP, you are protecting pretty much everything else in TCP/IP as well. An important issue, then, is how exactly do you get IPsec into IP? There are several implementation methods for deploying IPsec. These represent different ways that IPsec may modify the overall layer architecture of TCP/IP.

Three different implementation architectures are defined for IPsec in RFC 2401. The one you use depends on various factors including the version of IP used (IPv4 or IPv6), the requirements of the application, and other factors. These, in turn, rest on a primary implementation decision: Should IPsec be programmed into all hosts on a network, or just into certain routers or other intermediate devices? This is a design decision that must be based on the requirements of the network:

**End-Host Implementation** Putting IPsec into all host devices provides the most flexibility and security. It enables end-to-end security between any two devices on the network. However, there are many hosts on a typical network, so this means far more work than just implementing IPsec in routers.

**Router Implementation** This option is much less work because it means you make changes to only a few routers instead of hundreds or thousands of clients. It provides protection only between pairs of routers that implement IPsec, but this may be sufficient for certain applications such as VPNs. The routers can be used to provide protection for just the portion of the route that datagrams take outside the organization, thereby leaving connections between routers and local hosts unsecured (or possibly, secured by other means).

Three different architectures are defined that describe methods for how to get IPsec into the TCP/IP protocol stack: integrated, bump in the stack, and bump in the wire.

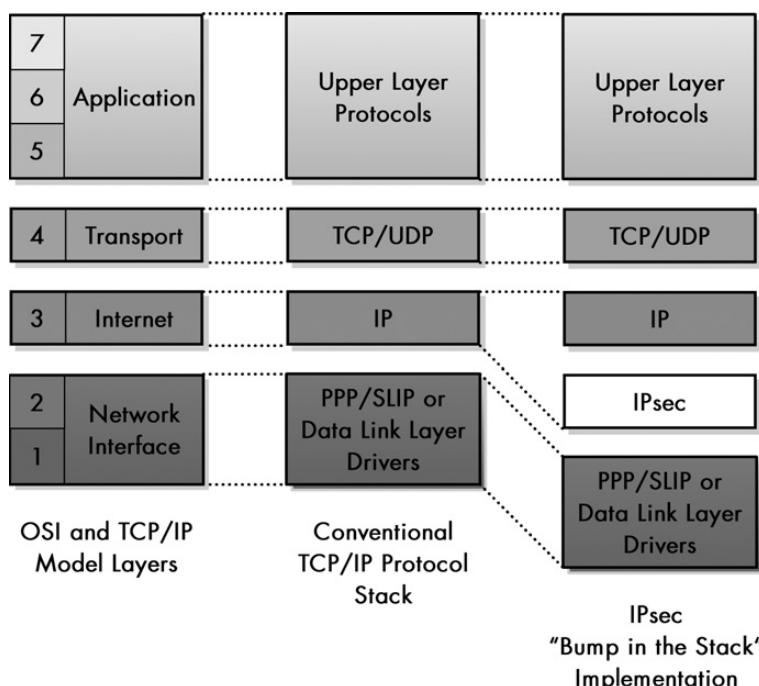
### **Integrated Architecture**

Under ideal circumstances, we would integrate IPsec's protocols and capabilities directly into IP itself. This is the most elegant solution, because it allows all IPsec security modes and capabilities to be provided just as easily as regular IP. No extra hardware or architectural layers are needed.

IPv6 was designed to support IPsec. Thus, it's a viable option for hosts or routers. With IPv4, integration would require making changes to the IP implementation on each device, which is often impractical (to say the least!).

### **Bump in the Stack (BITS) Architecture**

In the bump in the stack (BITS) technique, IPsec is made a separate architectural layer between IP and the data link layer. The cute name refers to the fact that IPsec is an extra element in the networking protocol stack, as you can see in Figure 29-2. IPsec intercepts IP datagrams as they are passed down the protocol stack, provides security, and passes them to the data link layer.

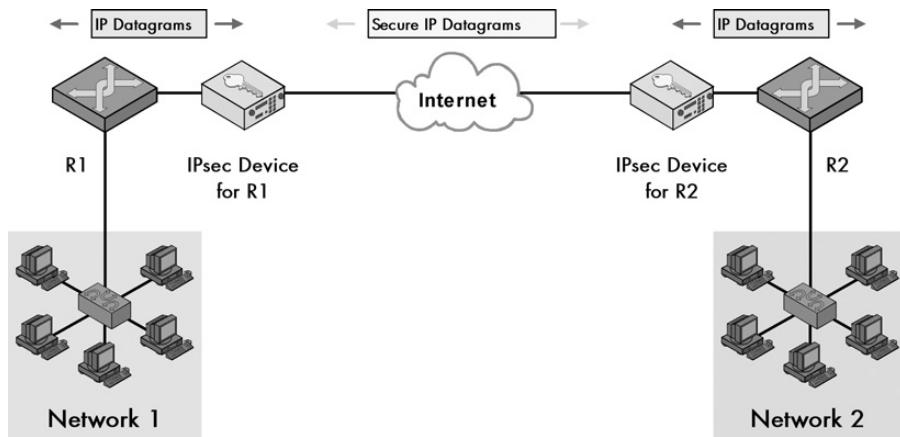


**Figure 29-2: IPsec bump in the stack (BITS) architecture** In this type of IPsec implementation, IPsec becomes a separate layer in the TCP/IP stack. It is implemented as software that sits below IP and adds security protection to datagrams created by the IP layer.

The advantage of this technique is that IPsec can be retrofitted to any IP device, since the IPsec functionality is separate from IP. The disadvantage is that there is a duplication of effort compared to the integrated architecture. BITS is generally used for IPv4 hosts.

### Bump in the Wire (BITW) Architecture

In the bump in the wire (BITW) method, we add a hardware device that provides IPsec services. For example, suppose we have a company with two sites. Each has a network that connects to the Internet using a router that is not capable of IPsec functions. We can interpose a special IPsec device between the router and the Internet at both sites, as shown in Figure 29-3. These devices will then intercept outgoing datagrams, add IPsec protection to them, and strip it off incoming datagrams.



**Figure 29-3: IPsec bump in the wire (BITW) architecture** In this IPsec architecture, IPsec is actually implemented in separate devices that sit between the devices that wish to communicate securely. These repackage insecure IP datagrams for transport over the public Internet.

Just as BITS lets you add IPsec to legacy hosts, BITW can retrofit non-IPsec routers to provide security benefits. The disadvantages are complexity and cost.

**KEY CONCEPT** Three different architectures or implementation models are defined for IPsec. The best is integrated architecture, in which IPsec is built into the IP layer of devices directly. The other two are *bump in the stack* (BITS) and *bump in the wire* (BITW), which are ways of layering IPsec underneath regular IP, using software and hardware solutions, respectively.

As you will see in the next section, the choice of architecture has an important impact on which of the two IPsec modes can be used. Incidentally, even though BITS and BITW seem quite different, they are actually do the same thing. In the case of BITS, we have an extra software layer that adds security to existing IP datagrams; in BITW, distinct hardware devices do this same job. In both cases, the result is the same, and the implications on the choice of IPsec mode is likewise the same.

## IPsec Modes: Transport and Tunnel

You just saw that three different basic implementation architectures could be used to provide IPsec facilities to TCP/IP networks. The choice of which implementation you use, as well as whether you implement in end hosts or routers, impacts the specific way that IPsec functions. Two specific modes of operation that are related to these architectures are defined for IPsec. They are called *transport mode* and *tunnel mode*.

IPsec modes are closely related to the function of the two core protocols, AH and ESP. Both of these protocols provide protection by adding a header (and possibly other fields) containing security information to a datagram. The choice of mode does not affect the method by which each generates its header, but rather, changes what specific parts of the IP datagram are protected and how the headers are arranged to accomplish this. In essence, the mode really describes, not prescribes, how AH or ESP do their thing. It is used as the basis for defining other constructs, such as security associations (SAs).

### Transport Mode

As its name suggests, in transport mode, the protocol protects the message passed down to IP from the transport layer. The message is processed by AH and/or ESP, and the appropriate header(s) are added in front of the transport (UDP or TCP) header. The IP header is then added in front of that by IP.

Another way of looking at this is as follows: Normally, the transport layer packages data for transmission and sends it to IP. From IP's perspective, this transport layer message is the payload of the IP datagram. When IPsec is used in transport mode, the IPsec header is applied only over this IP payload, not the IP header. The AH and ESP headers appear between the original, single IP header and the IP payload. This is illustrated in Figure 29-4.

### Tunnel Mode

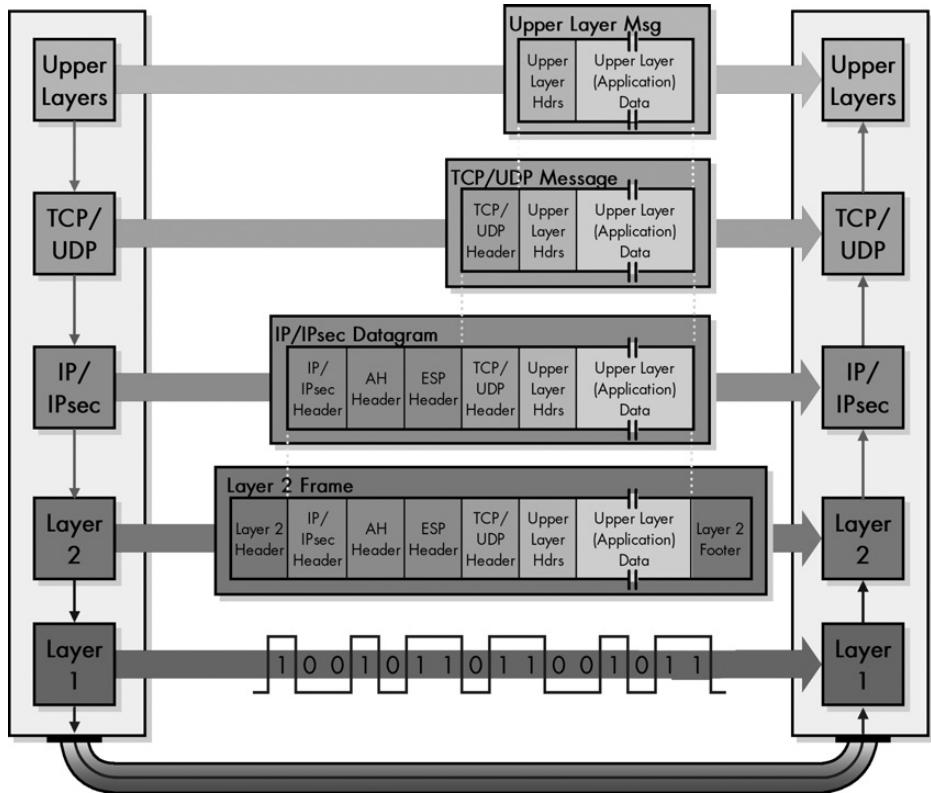
In tunnel mode, IPsec is used to protect a completely encapsulated IP datagram after the IP header has already been applied to it. The IPsec headers appear in front of the original IP header, and then a new IP header is added in front of the IPsec header. That is to say, the entire original IP datagram is secured and then encapsulated within another IP datagram. This is shown in Figure 29-5.

### Comparing Transport and Tunnel Modes

The bottom line in understanding the difference between the two IPsec modes is this: Tunnel mode protects the original IP datagram as a whole, header and all, while transport mode does not. Thus, in general terms, the order of the headers is as follows:

**Transport Mode** IP header, IPsec headers (AH and/or ESP), IP payload (including transport header)

**Tunnel Mode** New IP header, IPsec headers (AH and/or ESP), old IP header, IP payload

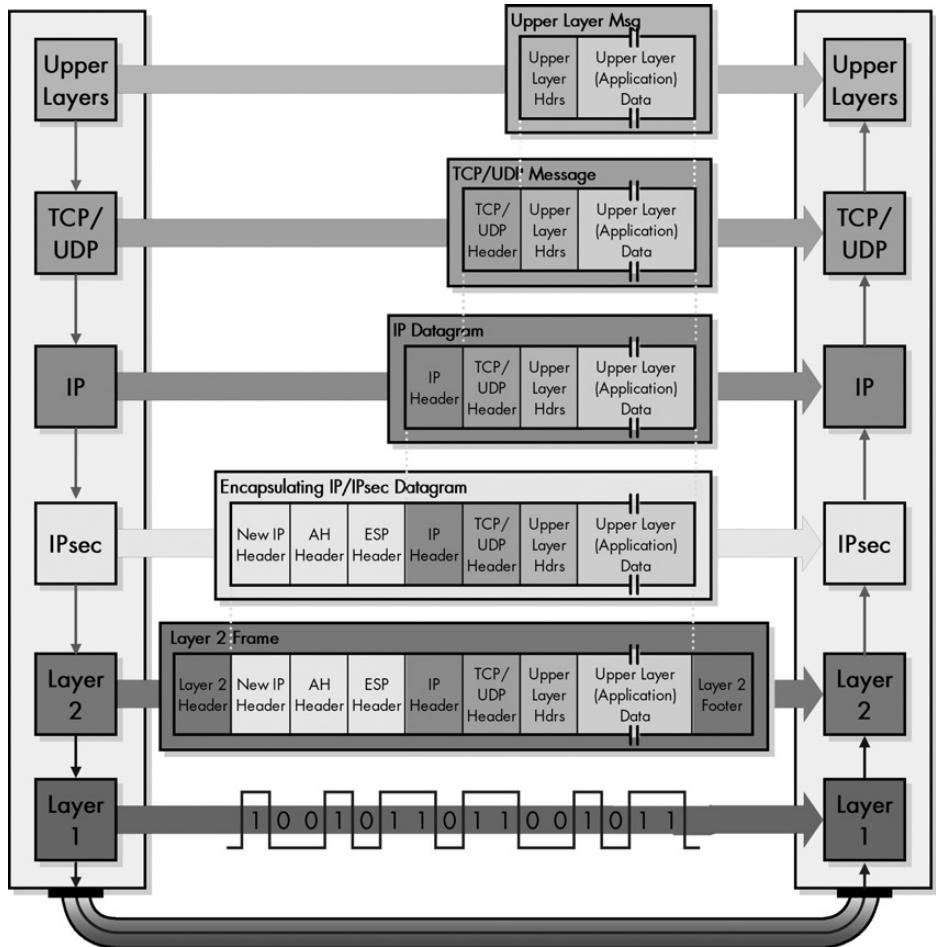


**Figure 29-4: IPsec transport mode operation** When IPsec operates in transport mode, it is integrated with IP and used to transport the upper layer (TCP/UDP) message directly. After processing, the datagram has just one IP header that contains the AH and ESP IPsec headers. Contrast this to tunnel mode, shown in Figure 29-5.

Again, this is a simplified view of how IPsec datagrams are constructed; the reality is significantly more complex. The exact way that the headers are arranged in an IPsec datagram in both transport and tunnel modes depends on which version of IP is being used. IPv6 uses extension headers that must be arranged in a particular way when IPsec is used. The header placement also depends on which IPsec protocol is being used, AH or ESP. Note that it is also possible to apply both AH and ESP to the same datagram; if so, the AH header always appears before the ESP header.

There are thus three variables and eight basic combinations of mode (tunnel or transport), IP version (IPv4 or IPv6) and protocol (AH or ESP). The coming discussions of AH and ESP describe the four format combinations of transport/tunnel mode and IPv4/IPv6 applicable to each protocol. Note that ESP also includes an ESP trailer that goes after the data protected.

You could probably tell by reading these descriptions how the two modes relate to the choice of IPsec architecture you looked at earlier. Transport mode requires that IPsec be integrated into IP, because AH/ESP must be applied as the original IP packaging is performed on the transport layer message. This is often the choice for implementations requiring end-to-end security with hosts that run IPsec directly.



**Figure 29-5: IPsec tunnel mode operation** IPsec tunnel mode is so named because it represents an encapsulation of a complete IP datagram, thereby forming a virtual tunnel between IPsec-capable devices. The IP datagram is passed to IPsec, where a new IP header is created with the AH and ESP IPsec headers added. Contrast this to transport mode, shown in Figure 29-4.

Tunnel mode represents an encapsulation of IP within the combination of IP plus IPsec. Thus, it corresponds with the BITS and BITW implementations, where IPsec is applied after IP has processed higher-layer messages and has already added its header. Tunnel mode is a common choice for VPN implementations, which are based on the tunneling of IP datagrams through an unsecured network such as the Internet.

**KEY CONCEPT** IPsec has two basic modes of operation. In *transport mode*, IPsec AH and ESP headers are added as the original IP datagram is created. Transport mode is associated with integrated IPsec architectures. In *tunnel mode*, the original IP datagram is created normally, and then the entire datagram is encapsulated into a new IP datagram containing the AH/ESP IPsec headers. Tunnel mode is most commonly used with *bump in the stack* (BITS) and *bump in the wire* (BITW) implementations.

## IPsec Security Constructs

Important IPsec security constructs include security associations, the security association database, security policies, the security policy database, selectors, and the security parameter index. These items are all closely related and essential to understand before you begin looking at the core IPsec protocols. These constructs are used to guide the operation of IPsec in a general way and particularly to guide exchanges between devices. The constructs control how IPsec works and ensure that each datagram coming into or leaving an IPsec-capable device is treated properly.

### Security Policies, Security Associations, and Associated Databases

Let's begin by considering the problem of how to apply security in a device that may be handling many different exchanges of datagrams with others. There is overhead involved in providing security, so you do not want to do it for every message that comes in or out. Some types of messages may need more security; others may need less. Also, exchanges with certain devices may require different processing than others.

To manage all of this complexity, IPsec is equipped with a flexible, powerful way of specifying how different types of datagrams should be handled. To understand how this works, you must first define the following two important logical concepts:

**Security Policies and the Security Policy Database (SPD)** A *security policy* is a rule that is programmed into the IPsec implementation. It tells the implementation how to process different datagrams received by the device. For example, security policies decide if a particular packet needs to be processed by IPsec or not. AH and ESP entirely bypass those that do not need processing. If security is required, the security policy provides general guidelines for how it should be provided, and if necessary, links to more specific detail. Security policies for a device are stored in the device's *security policy database (SPD)*.

**Security Associations (SAs) and the Security Association Database (SAD)** A security association (SA) is a set of security information that describes a particular kind of secure connection between one device and another. You can consider it a contract, if you will, that specifies the particular security mechanisms that are used for secure communications between the two. A device's security associations are contained in its *security association database (SAD)*.

It's often hard to distinguish between the SPD and the SAD, because they are similar in concept. The main difference between them is that security policies are general, while security associations are more specific. To determine what to do with a particular datagram, a device first checks the SPD. The security policies in the SPD may reference a particular SA in the SAD. If so, the device will look up that SA and use it for processing the datagram.

## **Selectors**

One issue I haven't covered yet is how a device determines what security policies or SAs to use for a specific datagram. Again here, IPsec defines a very flexible system that lets each security association define a set of rules for choosing datagrams that the SA applies to. Each of these rule sets is called a *selector*. For example, you might define a selector that says that a particular range of values in the Source Address of a datagram, combined with another value in the Destination Address, means that a specific SA must be used for the datagram.

## **Security Association Triples and Security Parameter Index (SPI)**

Each secure communication that a device makes to another requires that an SA be established. SAs are unidirectional, so each one only handles either inbound or outbound traffic for a particular device. This allows the level of security for a flow from Device A to Device B to be different than the level for traffic coming from Device B to Device A. In a bidirectional communication of this sort, both Device A and Device B would have two SAs; Device A would have SAs that you could call SAdeviceBin and SAdeviceBout. Device B would have SAs SAdeviceAin and SAdeviceAout.

SAs don't actually have names, however. They are instead defined by a set of three parameters, called a *tuple*:

**Security Parameter Index (SPI)** A 32-bit number that is chosen to uniquely identify a particular SA for any connected device. The SPI is placed in AH or ESP datagrams and thus links each secure datagram to the security association. It is used by the recipient of a transmission so it knows what SA governs the datagram.

**IP Destination Address** The address of the device for which the SA is established.

**Security Protocol Identifier** Specifies whether this association is for AH or ESP. If both are in use with this device, they have separate SAs.

As you can see, the two security protocols AH and ESP are dependent on SAs, security policies, and the various databases that control the operation of those SAs and policies. Management of these databases is important, but it's another complex subject entirely. Generally, SAs can either be set up manually (which is of course extra work) or you can deploy an automated system using a protocol like IKE (discussed near the end of this chapter).

Confused? I don't blame you, despite my best efforts, and remember that this is all highly simplified. Welcome to the wonderful world of networking security. If you are ever besieged by insomnia, I highly recommend RFC 2401!

## **IPsec Authentication Header (AH)**

As I mentioned earlier in this chapter, AH is one of the two core security protocols in IPsec. This is another protocol whose name has been well chosen. It provides *authentication* of either all or part of the contents of a datagram through the addition

of a *header* that is calculated based on the values in the datagram. The parts of the datagram that are used for the calculation, and the placement of the header, depend on the mode (tunnel or transport) and the version of IP (IPv4 or IPv6).

The operation of AH is surprisingly simple, especially for any protocol that has anything to do with network security. The simplicity is analogous to the algorithms used to calculate checksums or perform cyclic redundancy (CRC) checks for error detection. In those cases, the sender uses a standard algorithm to compute a checksum or CRC code based on the contents of a message. This computed result is transmitted along with the original data to the destination, which repeats the calculation and discards the message if any discrepancy is found between its calculation and the one done by the source.

This is the same idea behind AH, except that instead of using a simple algorithm known to everyone, it uses a special hashing algorithm and a specific key known only to the source and the destination. An SA between two devices specifies these particulars, so that the source and destination know how to perform the computation but nobody else can. On the source device, AH performs the computation and puts the result (called the *integrity check value*, or *ICV*) into a special header with other fields for transmission. The destination device does the same calculation using the key that the two devices share. This enables the device to see immediately if any of the fields in the original datagram were modified (due to either error or malice).

Just as a checksum doesn't change the original data, neither does the ICV calculation change the original data. The presence of the AH header allows us to verify the integrity of the message, but doesn't encrypt it. Thus, AH provides *authentication* but not *privacy* (that's what ESP is for).

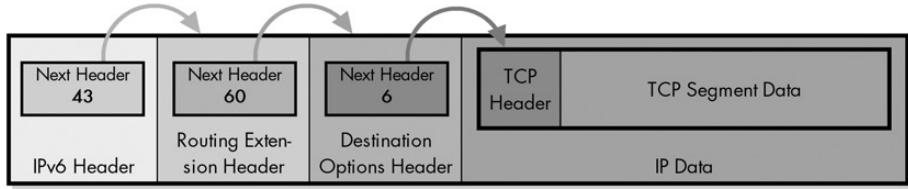
## AH Datagram Placement and Linking

The calculation of AH is similar for both IPv4 and IPv6. One difference is in the exact mechanism used for placing the header into the datagram and for linking the headers together. I'll describe IPv6 first because it is simpler, and because AH was really designed to fit into its mechanism for this.

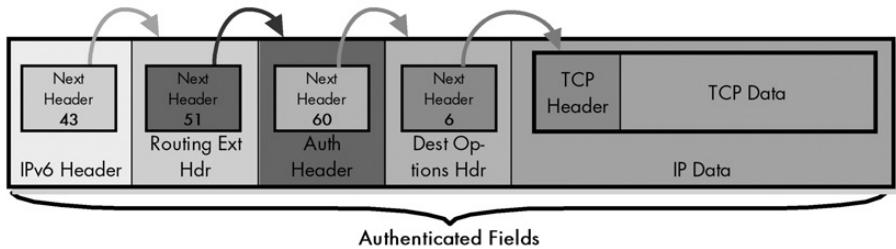
### IPv6 AH Placement and Linking

In IPv6, the AH is inserted into the IP datagram as an extension header, following the normal IPv6 rules for extension header linking. It is linked by the previous header (extension or main), which puts the assigned value for the AH header (51) into its Next Header field. The AH header then links to the next extension header or the transport layer header using its Next Header field.

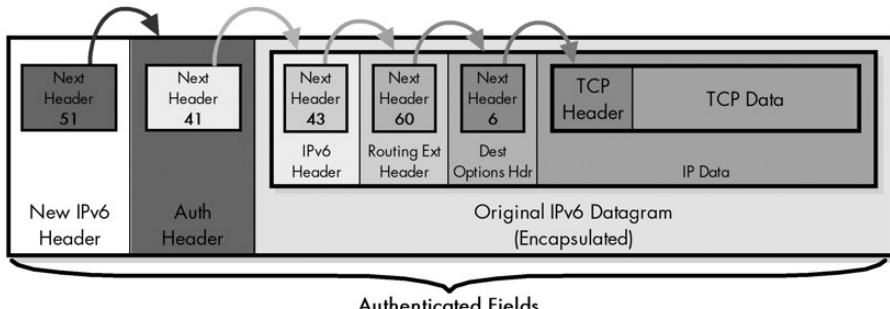
In transport mode, the AH is placed into the main IP header and appears before any Destination Options header that contains options intended for the final destination, and before an ESP header if present, but after any other extension headers. In tunnel mode, it appears as an extension header of the new IP datagram that encapsulates the original one being tunneled. This is shown graphically in Figure 29-6.



**Original IPv6 Datagram Format (Including Routing Extension Header and Destination-Specific Destination Options Extension Header)**



**IPv6 AH Datagram Format - IPsec Transport Mode**



**IPv6 AH Datagram Format - IPsec Tunnel Mode**

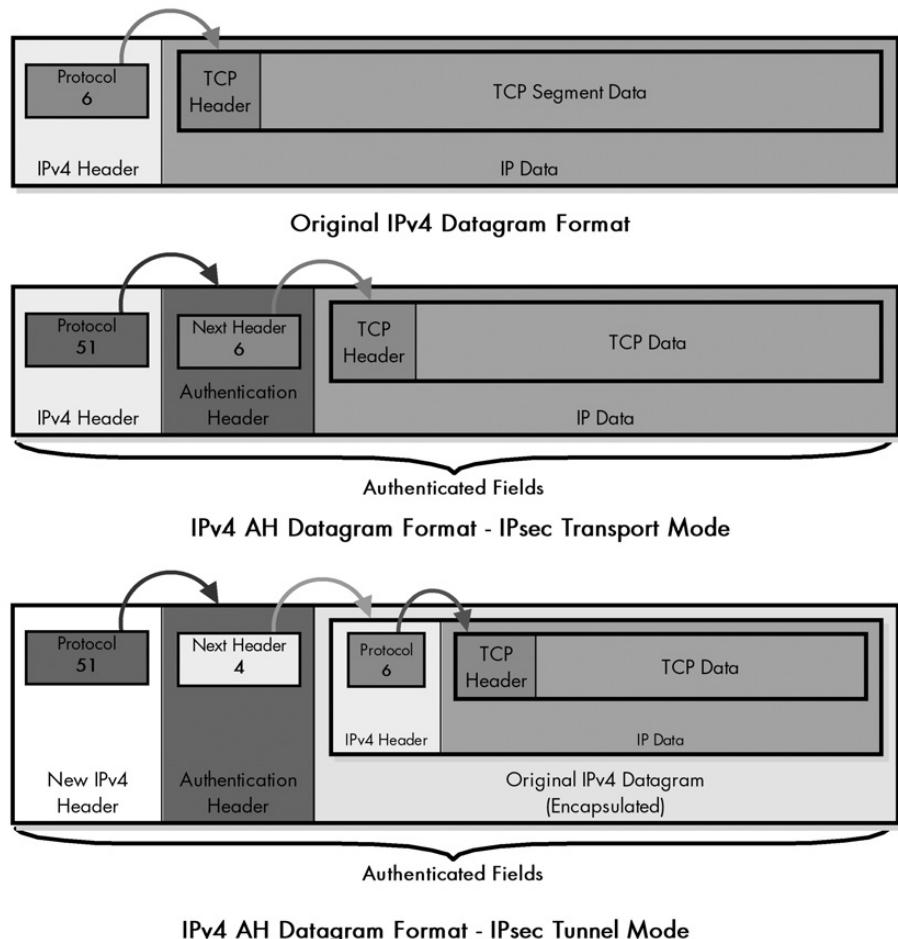
**Figure 29-6: IPv6 datagram format with IPsec Authentication Header (AH)** This is an example of an IPv6 datagram with two extension headers that are linked using the standard IPv6 mechanism (see Figure 26-3 in Chapter 26). When AH is applied in transport mode, it is simply added as a new extension header (as shown in dark shading) that goes between the Routing extension header and the Destination Options header. In tunnel mode, the entire original datagram is encapsulated into a new IPv6 datagram that contains the AH header. In both cases, the Next Header fields are used to link each header one to the next. Note the use of Next Header value 41 in tunnel mode, which is the value for the encapsulated IPv6 datagram.

### IPv4 AH Placement and Linking

In IPv4, a method that is similar to the IPv6 header-linking technique is employed. In an IPv4 datagram, the Protocol field indicates the identity of the higher-layer protocol (typically TCP or UDP) that's carried in the datagram. As such, this field points to the next header, which is at the front of the IP payload. AH takes this value and

puts it into its Next Header field, and then places the protocol value for AH itself (51 in dotted decimal) into the IP Protocol field. This makes the IP header point to the AH, which then points to whatever the IP datagram pointed to before.

Again, in transport mode, the AH header is added after the main IP header of the original datagram; in tunnel mode it is added after the new IP header that encapsulates the original datagram that's being tunneled. This is shown in Figure 29-7.



**Figure 29-7: IPv4 datagram format with IPsec AH** Here is an example of an IPv4 datagram; it may or may not contain IPv4 options (which are not distinct entities as they are in IPv6). In transport mode, the AH header is added between the IP header and the IP data; the Protocol field of the IP header points to it, while its Next Header field contains the IP header's prior protocol value (in this case 6, for TCP). In tunnel mode, the IPv4 datagram is encapsulated into a new IPv4 datagram that includes the AH header. Note that in tunnel mode, the AH header uses the value 4 (which means IPv4) in its Next Header field.

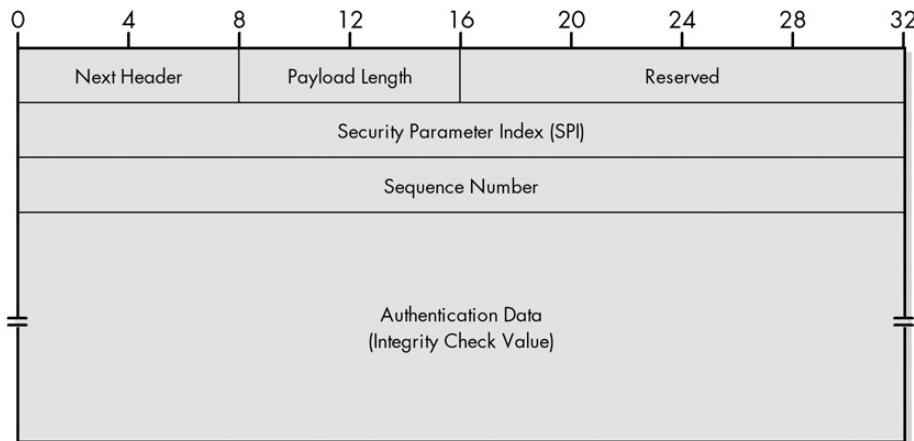
**KEY CONCEPT** The IPsec Authentication Header (AH) protocol allows the recipient of a datagram to verify its authenticity. It is implemented as a header that's added to an IP datagram that contains an *integrity check value (ICV)*, which is computed based on the values of the fields in the datagram. The recipient can use this value to ensure that the data has not been changed in transit. AH does not encrypt data and thus does not ensure the privacy of transmissions.

## AH Format

The format of AH is described in Table 29-2 and illustrated in Figure 29-8.

**Table 29-2:** IPsec Authentication Header (AH) Format

Field Name	Size (Bytes)	Description
Next Header	1	Contains the protocol number of the next header after the AH. Used to link headers together.
Payload Len	1	Despite its name, this field measures the length of the authentication header itself, not the payload. (I wonder what the history is behind that!) It is measured in 32-bit units, with 2 subtracted for consistency with how header lengths are normally calculated in IPv6.
Reserved	2	Not used; set to zeros.
SPI	4	A 32-bit value that, when combined with the destination address and security protocol type (which is obviously the one for AH here), identifies the security association (SA) that will be used for this datagram. (SAs are discussed earlier in this chapter.)
Sequence Number	4	A counter field that is initialized to zero when an SA is formed between two devices, and then incremented for each datagram sent using that SA. This uniquely identifies each datagram on an SA and is used to provide protection against replay attacks by preventing the retransmission of captured datagrams.
Authentication Data	Variable	Contains the result of the hashing algorithm, called the integrity check value (ICV), performed by the AH protocol.



**Figure 29-8:** IPsec Authentication Header (AH) format

The size of the Authentication Data field is variable to support different datagram lengths and hashing algorithms. Its total length must be a multiple of 32 bits. Also, the entire header must be a multiple of either 32 bits (for IPv4) or 64 bits (for IPv6), so additional padding may be added to the Authentication Data field if necessary.

You may also notice that no IP addresses appear in the header, which is a prerequisite for it being the same for both IPv4 and IPv6.

## IPsec Encapsulating Security Payload (ESP)

The IPsec AH provides integrity authentication services to IPsec-capable devices so that they can verify that messages are received intact from other devices. For many applications, however, this is only one piece of the puzzle. We want to not only protect against intermediate devices changing the datagrams, but also to protect against them examining their contents as well. For this level of private communication, AH is not enough; we need to use the ESP protocol.

The main job of ESP is to provide the privacy we seek for IP datagrams by encrypting them. An encryption algorithm combines the data in the datagram with a key to transform it into an encrypted form. This is then repackaged using a special format that you will see shortly, and then transmitted to the destination, which decrypts it using the same algorithm. ESP also sports its own authentication scheme like the one used in AH, or it can be used in conjunction with AH.

### **ESP Fields**

ESP has several fields that are the same as those used in AH, but it packages its fields in a very different way. Instead of having just a header, it divides its fields into three components:

**ESP Header** This contains two fields, SPI and Sequence Number, and comes before the encrypted data. Its placement depends on whether ESP is used in transport mode or tunnel mode, as explained earlier in this chapter.

**ESP Trailer** This section is placed after the encrypted data. It contains padding that is used to align the encrypted data through a Padding and Pad Length field. Interestingly, it also contains the Next Header field for ESP.

**ESP Authentication Data** This field contains an ICV that's computed in a manner that's similar to how the AH protocol works. The field is used when ESP's optional authentication feature is employed.

There are two reasons why these fields are broken into pieces like this. The first is that some encryption algorithms require the data to be encrypted to have a certain block size, and so padding must appear after the data and not before it. That's why padding appears in the ESP Trailer field. The second is that the ESP Authentication Data appears separately because it is used to authenticate the rest of the encrypted datagram after encryption. This means that it cannot appear in the ESP Header or ESP Trailer.

## **ESP Operations and Field Use**

This is still a bit boggling so I'm going to try to explain this procedurally by considering three basic steps performed by ESP: calculating the header, then the trailer, and then the Authentication field.

### **Header Calculation and Placement**

The first thing to consider is how the ESP header is placed. This is similar to how AH works and depends on the IP version, as follows:

**IPv6** The ESP Header field is inserted into the IP datagram as an extension header, following the normal IPv6 rules for extension-header linking. In transport mode, it appears before a Destination Options header that contains options intended for the final destination, but after any other extension headers, if present. In tunnel mode, it appears as an extension header of the new IP datagram that encapsulates the original one being tunneled. This is shown in Figure 29-9.

**IPv4** As with AH, the ESP Header field is placed after the normal IPv4 header. In transport mode, it appears after the IP header of the original datagram; in tunnel mode, it appears after the IP header of the new IP datagram that's encapsulating the original one. You can see this in Figure 29-10.

### **Trailer Calculation and Placement**

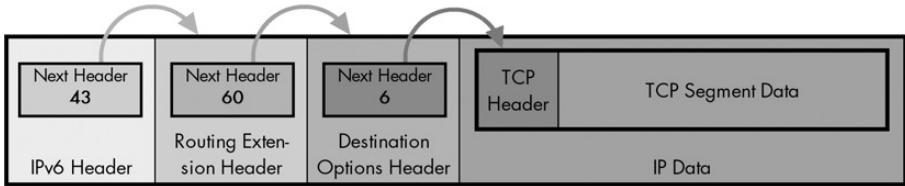
The ESP Trailer field is appended to the data that will be encrypted. ESP then performs the encryption. The payload (TCP/UDP message or encapsulated IP datagram) and the ESP trailer are both encrypted, but the ESP header is not. Note again that any other IP headers that appear between the ESP header and the payload are also encrypted. In IPv6, this can include a Destination Options extension header.

Normally, the Next Header field would appear in the ESP Header and would be used to link the ESP Header to the header that comes after it. However, the Next Header field in ESP appears in the trailer and not the header, which makes the linking seem a bit strange in ESP. The method is basically the same as what's used in AH and in IPv6 in general, with the Next Header and Protocol fields being used to tie everything together. However, in ESP the Next Header field appears *after* the encrypted data, and so it points back to one of the following: a Destination Options extension header (if present), a TCP/UDP header (in transport mode), or an IPv4/IPv6 header (in tunnel mode). This is also shown in Figures 29-9 and 29-10.

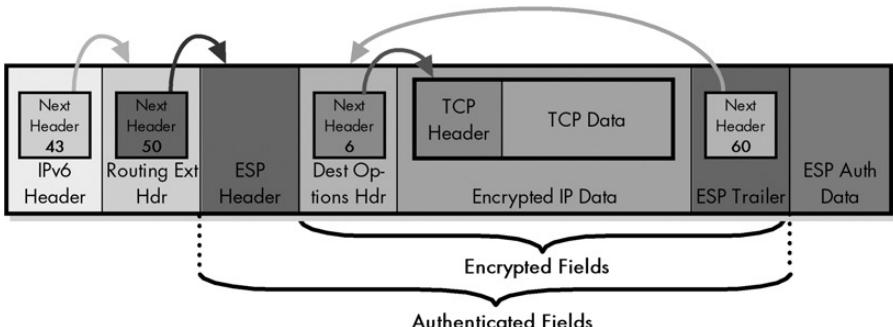
### **ESP Authentication Field Calculation and Placement**

If the optional ESP authentication feature is being used, it is computed over the entire ESP datagram (except the Authentication Data field itself, of course). This includes the ESP header, payload, and trailer.

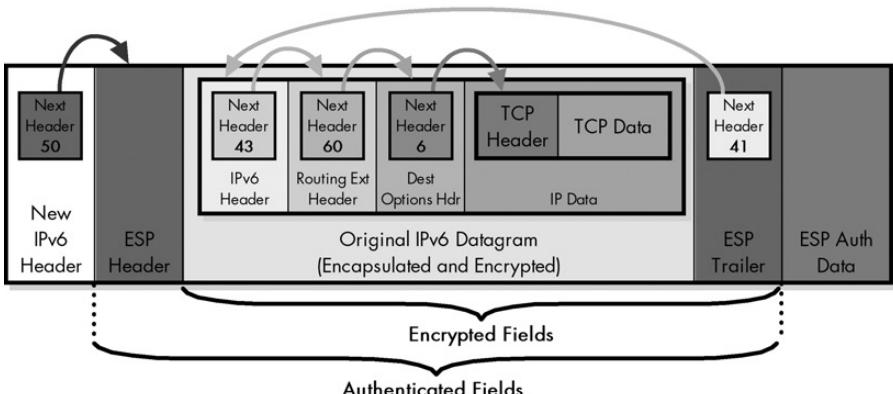
**KEY CONCEPT** The IPsec ESP protocol allows the contents of a datagram to be encrypted, which ensures that only the intended recipient is able to see the data. ESP is implemented using three components: an *ESP Header* that's added to the front of a protected datagram, an *ESP Trailer* that follows the protected data, and an optional *ESP Authentication Data* field that provides authentication services similar to those provided by AH.



**Original IPv6 Datagram Format (Including Routing Extension Header and Destination-Specific Destination Options Extension Header)**

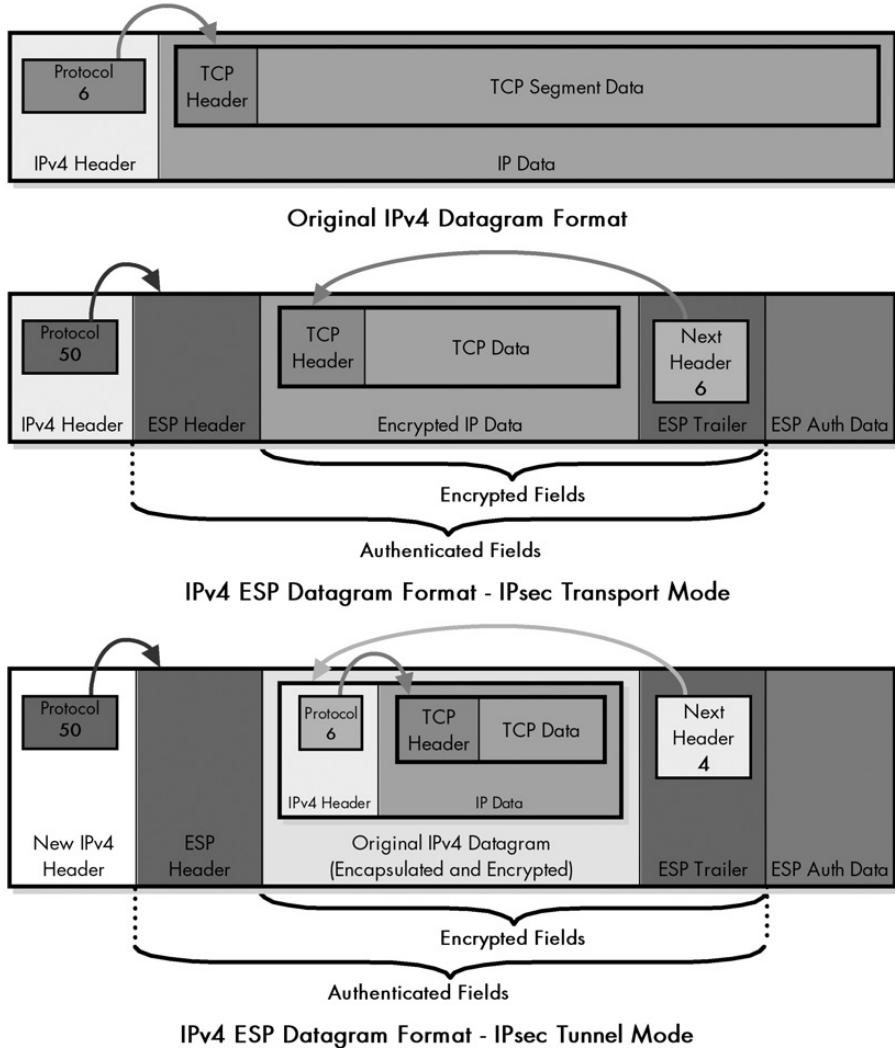


**IPv6 ESP Datagram Format - IPsec Transport Mode**



**IPv6 ESP Datagram Format - IPsec Tunnel Mode**

**Figure 29-9: IPv6 datagram format with IPsec ESP** Here is the same example of an IPv6 datagram with two extension headers that you saw in Figure 29-6. When ESP is applied in transport mode, the ESP Header field is added to the existing datagram as in AH, and the ESP Trailer and ESP Authentication Data fields are placed at the end. In tunnel mode, the ESP Header and Trailer fields bracket the entire encapsulated IPv6 datagram. Note the encryption and authentication coverage in each case, and also how the Next Header field points back into the datagram since it appears in the ESP Trailer.



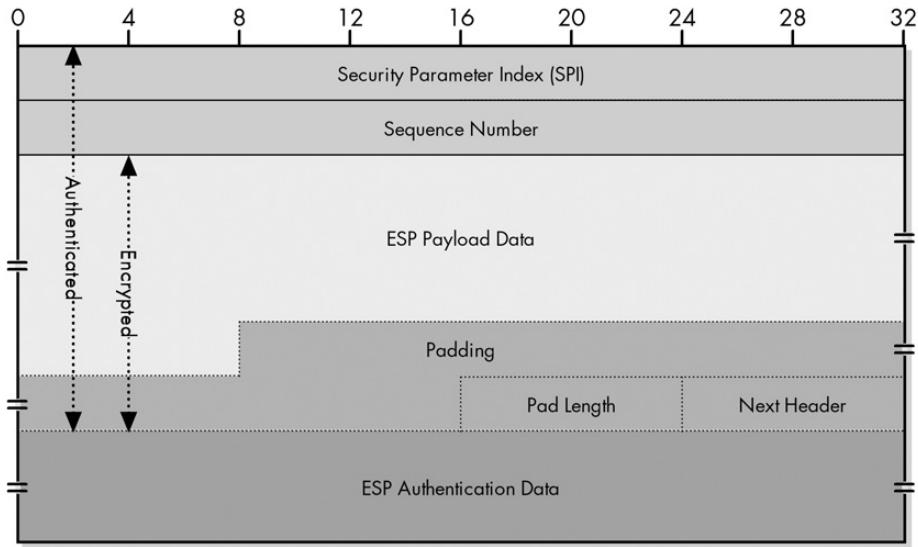
**Figure 29-10: IPv4 datagram format with IPsec ESP** Here is the same sample IPv4 datagram that you saw in Figure 29-7. When ESP processes this datagram in transport mode, the ESP Header field is placed between the IPv4 header and data, with the ESP Trailer and ESP Authentication Data fields following. In tunnel mode, the entire original IPv4 datagram is surrounded by these ESP components, rather than just the IPv4 data. Again, as in Figure 29-9, note the encryption and authentication coverage, and how the Next Header field points back to specify the identity of the encrypted data or datagram.

## ESP Format

The format of the ESP sections and fields is described in Table 29-3 and illustrated in Figure 29-11. In both the figure and the table, I have shown the encryption and authentication coverage of the fields explicitly, to clarify how it all works.

**Table 29-3:** IPsec Encapsulating Security Payload (ESP) Format

Section	Field Name	Size (Bytes)	Description	Encryption Coverage	Authentication Coverage
ESP Header	SPI	4	A 32-bit value that is combined with the destination address and security protocol type to identify the SA that will be used for this datagram. (SAs are discussed earlier in this chapter.)		
	Sequence Number	4	A counter field initialized to zero when an SA is formed between two devices, and then incremented for each datagram that's sent using that SA. This is used to provide protection against replay attacks.		
Payload	Payload Data	Variable	The encrypted payload data, which consists of a higher-layer message or encapsulated IP datagram. It may also include support information such as an initialization vector that's required by certain encryption methods.		
ESP Trailer	Padding	Variable (0 to 255)	Additional padding bytes are included as needed for encryption or for alignment.		
	Pad Length	1	The number of bytes in the preceding Padding field.		
	Next Header	1	Contains the protocol number of the next header in the datagram. Used to chain together headers.		
ESP Authentication Data	Variable		Contains the ICV resulting from the application of the optional ESP authentication algorithm.		



**Figure 29-11: IPsec ESP format** Note that most of the fields and sections in this format are variable length. The exceptions are the SPI and Sequence Number fields, which are four bytes long, and the Pad Length and Next Header fields, which are one byte each.

The Padding field is used when encryption algorithms require it. Padding is also used to make sure that the ESP Trailer field ends on a 32-bit boundary. That is, the size of the ESP Header field plus the Payload field, plus the ESP Trailer field must be a multiple of 32 bits. The ESP Authentication Data field must also be a multiple of 32 bits.

## IPsec Internet Key Exchange (IKE)

IPsec, like many secure networking protocol sets, is based on the concept of a shared secret. Two devices that want to send information securely encode and decode it using a piece of information that only the devices know. Anyone who isn't in on the secret is able to intercept the information but is prevented either from reading it (if ESP is used to encrypt the payload) or from tampering with it undetected (if AH is used). Before either AH or ESP can be used, however, it is necessary for the two devices to exchange the secret that the security protocols themselves will use. The primary support protocol used for this purpose in IPsec is called *Internet Key Exchange (IKE)*.

IKE is defined in RFC 2409, and it is one of the more complicated of the IPsec protocols to comprehend. In fact, it is simply impossible to truly understand more than a real simplification of its operation without significant background in cryptography. I don't have a background in cryptography, and I must assume that you, my reader, do not either. So rather than fill this topic with baffling acronyms and unexplained concepts, I will just provide a brief outline of IKE and how it is used.

## IKE Overview

The purpose of IKE is to allow devices to exchange information that's required for secure communication. As the title suggests, this includes cryptographic keys that are used for encoding authentication information and performing payload encryption. IKE works by allowing IPsec-capable devices to exchange SAs, which populate their SADs. These SADs are then used for the actual exchange of secured datagrams with the AH and ESP protocols.

IKE is considered a hybrid protocol because it combines (and supplements) the functions of three other protocols. The first of these is the *Internet Security Association and Key Management Protocol (ISAKMP)*. This protocol provides a framework for exchanging encryption keys and security association information. It operates by allowing security associations to be negotiated through a series of phases.

ISAKMP is a generic protocol that supports many different key exchange methods. In IKE, the ISAKMP framework is used as the basis for a specific key exchange method that combines features from two key exchange protocols:

**OAKLEY** Describes a specific mechanism for exchanging keys through the definition of various key exchange modes. Most of the IKE key exchange process is based on OAKLEY.

**SKEME** Describes a different key exchange mechanism than OAKLEY. IKE uses some features from SKEME, including its method of public key encryption and its fast rekeying feature.

## IKE Operation

IKE doesn't strictly implement either OAKLEY or SKEME but takes bits of each to form its own method of using ISAKMP. Clear as mud, I know. Because IKE functions within the framework of ISAKMP, its operation is based on the ISAKMP phased-negotiation process. There are two phases, as follows:

**ISAKMP Phase 1** The first phase is a setup stage where two devices agree on how to exchange further information securely. This negotiation between the two units creates an SA for ISAKMP itself: an *ISAKMP SA*. This security association is then used for securely exchanging more detailed information in Phase 2.

**ISAKMP Phase 2** In this phase, the ISAKMP SA established in Phase 1 is used to create SAs for other security protocols. Normally, this is where the parameters for the “real” SAs for the AH and ESP protocols would be negotiated.

An obvious question is why IKE bothers with this two-phased approach. Why not just negotiate the SA for AH or ESP in the first place? Well, even though the extra phase adds overhead, multiple Phase 2 negotiations can be conducted after one Phase 1, which amortizes the extra cost of the two-phase approach. It is also possible to use a simpler exchange method for Phase 2 once the ISAKMP SA has been established in Phase 1.

The ISAKMP SA negotiated during Phase 1 includes the negotiation of the following attributes used for subsequent negotiations:

- An encryption algorithm, such as the *Data Encryption Standard (DES)*
- A hash algorithm (MD5 or SHA, as used by AH or ESP)
- An authentication method, such as authentication using previously shared keys
- A *Diffie-Hellman* group

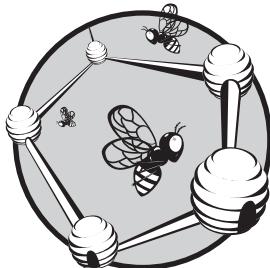
**NOTE** *Diffie and Hellman were two pioneers in the industry who invented public-key cryptography. In this method, instead of encrypting and decrypting with the same key, data is encrypted using a public key that anyone can know, and decrypted using a private key that is kept secret. A Diffie-Hellman group defines the attributes of how to perform this type of cryptography. Four predefined groups derived from OAKLEY are specified in IKE, and provision is allowed for defining new groups as well.*

Note that even though SAs in general are unidirectional, the ISAKMP SA is established bidirectionally. Once Phase 1 is complete, either device can set up a subsequent SA for AH or ESP using the ISAKMP SA.



# 30

## **INTERNET PROTOCOL MOBILITY SUPPORT (MOBILE IP)**



The Internet Protocol (IP) is the most successful network layer protocol in computing due to its many strengths, but it also has some weaknesses, most of which have become more important as networks have evolved over time. Technologies like classless addressing and Network Address Translation (NAT) combat the exhaustion of the IP version 4 (IPv4) address space, while IPsec provides it with the secure communications it lacks. Another weakness of IP is that it was not designed with mobile computers in mind.

While mobile devices can certainly use IP, the way that devices are addressed and datagrams routed causes a problem when they are moved from one network to another. At the time IP was developed, computers were large and rarely moved. Today, we have millions of notebook computers and smaller devices, some of which even use wireless networking to connect to

the wired network. The importance of providing full IP capabilities for these mobile devices has grown dramatically. To support IP in a mobile environment, a new protocol called *IP Mobility Support*, or more simply, *Mobile IP*, was developed.

In this chapter, I describe the special protocol that was developed to overcome the problems with mobile computers attaching to IP internetworks. I begin with an overview of Mobile IP and a more detailed description of why it was created. I discuss important concepts that define Mobile IP and its general mode of operation. I then move on to some of the specifics of how Mobile IP works. This includes a description of the special Mobile IP addressing scheme, an explanation of how agents are discovered by mobile devices, a discussion of the process of registration with the device's home agent, and finally, an explanation of how data is encapsulated and routed. I discuss the impact that Mobile IP has on the operation of the TCP/IP Address Resolution Protocol (ARP). I end the chapter by examining some of the efficiency and security issues that come into play when Mobile IP is used.

**NOTE** *This section specifically describes how IP mobility support is provided for IPv4 networks. It does not deal with the more specific details for how mobility is implemented in IPv6.*

**BACKGROUND INFORMATION** *If you are not familiar with the basics of IP addressing and routing, I strongly suggest reading at least Chapters 16 and 23 before reading about Mobile IP.*

## Mobile IP Overview, History, and Motivation

Mobile computing has greatly increased in popularity over the past several years, largely due to advances in miniaturization. Today, we can get the power that once required a hulking behemoth of a machine in a notebook PC or even a handheld computer. We also have wireless LAN (WLAN) technologies that easily let a device move from place to place and retain networking connectivity at the data link layer. Unfortunately, IP was developed back in the era of the behemoths, and it isn't designed to deal gracefully with computers that move around. To understand why IP doesn't work well in a mobile environment, you must take a look back at how IP addressing and routing work.

### The Problem with Mobile Nodes in TCP/IP

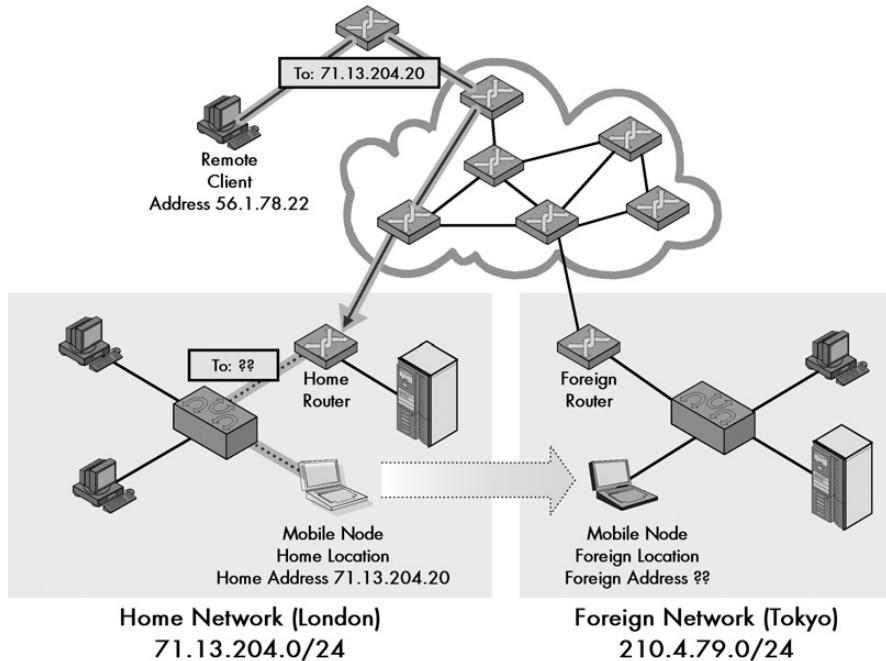
If you've read any of the materials in this book on IP addressing—and I certainly hope that you have—you know that IP addresses are fundamentally divided into two portions: a network identifier (network ID) and a host identifier (host ID). The network ID specifies which network a host is on, and the host ID uniquely specifies hosts within a network. This structure is fundamental to datagram routing, because devices use the network ID portion of the destination address of a datagram to determine if the recipient is on a local network or a remote one, and routers use it to determine how to route the datagram.

This is a great system, but it has one critical flaw: The IP address is tied tightly to the network where the device is located. Most devices never (or at least rarely) change their attachment point to the network, so this is not a problem for them, but it is certainly an issue for a mobile device. When the mobile device travels away from its home location, the system of routing based on IP address breaks. This is illustrated in Figure 30-1.

The tight binding of network ID and host IP address means that there are only two real options under conventional IP when a mobile device moves from one network to another:

**Change IP Address** We can change the IP address of the host to a new address that includes the network ID of the network to which it is moving.

**Decouple IP Routing from Address** We can change the way routing is done for the device, so that instead of routers sending datagrams to a device based on its network ID, they route based on its entire address.



**Figure 30-1: The main problem with mobile devices on IP internetworks** In this example, a mobile device (the notebook PC) has been moved from its home network in London to another network in Tokyo. A remote client (upper left) decides to send a datagram to the mobile device. However, it has no idea the device has moved. Since it sends by using the mobile node's home address, 71.13.204.20, its request is routed to the router responsible for that network, which is in London. The mobile device isn't there, so the router can't deliver it. Mobile IP solves this problem by giving mobile devices and routers the capability to forward datagrams from one location to another.

These both seem like viable options at first glance, and if only a few devices tried them, they might work. Unfortunately, they are both inefficient, often impractical, and neither is scalable (practical when thousands or millions of devices try them) for these reasons:

- Changing the IP address each time a device moves is time-consuming and normally requires manual intervention. In addition, the entire TCP/IP stack would need to be restarted, thereby breaking any existing connections.

- If we change the mobile device's IP address, how do we communicate the change of address to other devices on the Internet? These devices will only have the mobile node's original home address, which means they won't be able to find it, even if we give it a new address matching its new location.
- Routing based on the entire address of a host would mean the entire Internet would be flooded with routing information for each and every mobile computer. Considering how much trouble has gone into developing technologies like classless addressing to reduce routing table entries, it's obvious this is a Pandora's box no one wants to touch.

**KEY CONCEPT** The basic problem with supporting mobile devices in IP internetworks is that routing is performed using the IP address. This means the IP address of a device is tied to the network where that the device is located. If a device changes networks, data sent to its old address cannot be delivered by conventional means. Traditional work-arounds, such as routing by the full IP address or changing IP addresses manually, often create more problems.

### The Solution: Mobile IP

The solution to these difficulties was to define a new protocol especially to support mobile devices, which adds to the original IP. This protocol, called *IP Mobility Support for IPv4*, was first defined in RFC 2002, was updated in RFC 3220, and is now described in RFC 3344. The formal name given in that document title is rather long; the technology is more commonly called *Mobile IP*, both in the RFC itself and by networking professionals.

To ensure its success, Mobile IP's designers had to meet a number of important goals. The resulting protocol has these key attributes and features:

**Seamless Device Mobility Using Existing Device Address** Mobile devices can change their physical network attachment method and location while continuing to use their existing IP address.

**No New Addressing or Routing Requirements** The overall scheme for addressing and routing as in regular IP is maintained. IP addresses are still assigned in the conventional way by the owner of each device. No new routing requirements are placed on the internetwork, such as host-specific routes.

**Interoperability** Mobile IP devices can still send to and receive from existing IP devices that do not know how Mobile IP works, and vice versa.

**Layer Transparency** The changes made by Mobile IP are confined to the network layer. Transport layer and higher-layer protocols and applications are able to function as in regular IPv4, and existing connections can even be maintained across a move.

**Limited Hardware Changes** Changes are required to the mobile device's software as well as to routers used directly by the mobile device. Other devices, however, do not need changes, including routers between the ones on the home and visited networks.

**Scalability** Mobile IP allows a device to change from any network to any other, and supports this for an arbitrary number of devices. The scope of the connection change can be global; you could detach a notebook from an office in London and move it to Australia or Brazil, for example, and it will work the same as if you took it to the office next door.

**Security** Mobile IP works by redirecting messages, and includes authentication procedures to prevent an unauthorized device from causing problems.

Mobile IP accomplishes these goals by implementing a *forwarding system* for mobile devices. When a mobile unit is on its home network, it functions normally. When it moves to a different network, datagrams are sent from its home network to its new location. This allows normal hosts and routers that don't know about Mobile IP to continue to operate as if the mobile device had not moved. Special support services are required to implement Mobile IP; these services allow activities such as letting a mobile device determine where it is, telling the home network where to forward messages, and more.

**KEY CONCEPT** Mobile IP solves the problems associated with devices that change network locations by setting up a system whereby datagrams sent to the mobile node's home location are forwarded to it wherever it may be located. It is particularly useful for wireless devices, but can be used for any device that moves between networks periodically.

Mobile IP is often associated with wireless networks, since devices using WLAN technology can move so easily from one network to another. However, it wasn't designed specifically for wireless. It can be equally useful for moving from an Ethernet network in one building to a network in another building, city, or country. Mobile IP can be of great benefit in numerous applications for traveling salespeople, consultants who visit client sites, administrators who walk around a campus troubleshooting problems, and many more.

### ***Limitations of Mobile IP***

It's important to realize that Mobile IP has certain limitations in its usefulness in a wireless environment. It was designed to handle the mobility of devices, but only relatively infrequent mobility. This is due to the work involved with each change. This overhead isn't a big deal when you move a computer once a week, once a day, or even once an hour. It can be an issue for "real-time" mobility, such as roaming in a wireless network, where handoff functions operating at the data link layer may be more suitable. Mobile IP was designed under the specific assumption that the attachment point would not change more than once per second.

Mobile IP is intended to be used with devices that maintain a static IP configuration. Since the device needs to be able to always know the identity of its home network and normal IP address, it is much more difficult to use it with a device that obtains an IP address dynamically, using something like the Dynamic Host Configuration Protocol (DHCP).

## Mobile IP Concepts and General Operation

I like analogies because they provide a way of explaining often dry technical concepts in terms that you can relate to. The problem of mobile devices in an IP internetwork can easily be compared to a real-life mobility and information transmission problem: mail delivery for those who travel.

Suppose you are a consultant working for a large corporation with many offices. Your home office is in London, England, and you spend about half your time there. The rest of the time is split between other offices in, say, Rome, Tokyo, New York City, and Toronto. You also occasionally visit client sites that can be just about anywhere in the world. You may be at these remote locations for weeks at a time.

The problem is how do you arrange things so that you can receive your mail regardless of your location? You have the same problem that regular IP has with a mobile device, and without taking special steps, you have the same two unsatisfactory options for resolving it: address changing or decoupling routing from your address. You can't change your address each time you move because you would be modifying it constantly; by the time you told everyone about your new address, it would change again. And you certainly can't "decouple" the routing of mail from your address, unless you want to set up your own postal system!

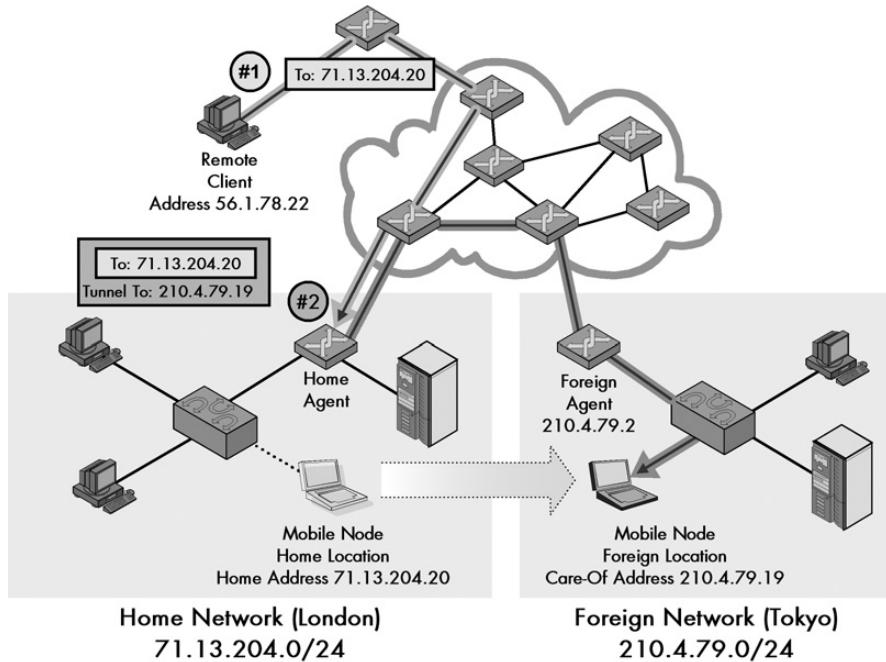
The solution to this dilemma is *mail forwarding*. Let's say that you leave London for Tokyo for a couple of months. You tell the London post office (PO) that you will be in Tokyo. They intercept mail headed for your normal London address, relabel it, and forward it to Tokyo. Depending on where you are staying, this mail might be redirected either straight to a new address in Tokyo or to a Tokyo PO where you can pick it up. If you leave Tokyo to go to another city, you just call the London PO and tell them your new location. When you come home, you cancel the forwarding and get your mail as always. (Yes, I'm assuming London and Tokyo each have only one PO.)

The advantages of this system are many. It is relatively simple to understand and implement. It is also transparent to everyone who sends you mail; they still send to you in London and it gets wherever it needs to go. And handling of the forwarding mechanism is done only by the London PO and possibly the PO where you are presently located; the rest of the postal system doesn't even know anything out of the ordinary is going on.

There are some disadvantages, too. The London PO may allow occasional forwarding for free, but would probably charge you if you did this on a regular basis. You might also need a special arrangement in the city you travel to. You need to keep communicating with your home PO each time you move. And every piece of mail must be sent through the system twice—first to London and then to wherever you are located—which is inefficient.

Mobile IP works in a manner very similar to the mail-forwarding system I just described. The traveling consultant is the device that goes from network to network. Each network can be considered like a different city, and the internetwork of routers is like the postal system. The router that connects any network to the Internet is like that network's post office, from an IP perspective.

The mobile node is normally resident on its home network, which is the one that is indicated by the network ID in its IP address. Devices on the internetwork always route using this address, so the pieces of “mail” (datagrams) always arrive at a router at the device’s “home.” When the device travels to another network, the home router (“post office”) intercepts these datagrams and forwards them to the device’s current address. It may send them straight to the device using a new, temporary address, or it may send them to a router on the device’s current network (the “other post office” or Tokyo in our analogy) for final delivery. You can see an overview of Mobile IP operation in Figure 30-2.



**Figure 30-2: General operation of Mobile IP** This diagram is similar to Figure 30-1, except that it shows Mobile IP implemented. The mobile node’s home router serves as home agent, and the router in Tokyo serves as the foreign agent. The mobile has been assigned a temporary “care-of” address to use while in Tokyo (which in this case is a co-located care-of address, meaning that it is assigned directly to the mobile node. Figure 30-3 shows the same example using the other type of care-of address). In step 1, the remote client sends a datagram to the mobile using its home address, as in normal TCP/IP. It arrives in London as usual. In step 2, the home agent encapsulates that datagram in a new one and sends it to the mobile node in Tokyo.

## Mobile IP Device Roles

As you can see, just as mail forwarding requires support from one or more POs, Mobile IP requires the help of two routers. The following special names are given to the three main players that implement the protocol (also shown in Figure 30-2):

**Mobile Node** This is the mobile device, the one moving around the internetwork.

**Home Agent** This is a router on the home network that is responsible for catching datagrams intended for the mobile node and forwarding them to it when it is traveling. It also implements other support functions that are necessary to run the protocol.

**Foreign Agent** This is a router on the network to which the mobile node is currently attached. It serves as a “home away from home” for the mobile node, and normally acts as its default router and implements Mobile IP functions. Depending on the mode of operation, it may receive forwarded datagrams from the home agent and forward them to the mobile node. It also supports the sharing of mobility information to make Mobile IP operate. The foreign agent may not be required in some Mobile IP implementations but is usually considered part of how the protocol operates.

**KEY CONCEPT** Mobile IP operates by setting up the TCP/IP equivalent of a mail-forwarding system. A router on a *mobile node*'s home network serves as the mobile device's *home agent*, and one on its current network acts as the *foreign agent*. The home agent receives datagrams destined for the mobile's normal IP address and forwards them to the mobile node's current location, either directly or by sending the datagrams to the foreign agent. The home agent and foreign agent are also responsible for various communication and setup activities that are required for Mobile IP to work.

## Mobile IP Functions

An important difference between Mobile IP and this mail-forwarding example is one that represents the classic distinction between people and computers: People are smart, and computers are not. When the consultant is traveling in Tokyo, he always knows he's in Tokyo and that his mail is being forwarded. He knows that he must deal with the Tokyo PO to get his mail. The PO in London knows what forwarding is all about and how to do it. The traveler and the POs can communicate easily using the telephone.

In contrast, in the computer world, when a device travels using Mobile IP, things are more complicated. Let's suppose the consultant flies to Tokyo, turns on his notebook, and plugs it in to the network. When the notebook is first turned on, it has no clue what is going on. The notebook has to figure out that it is in Tokyo. It needs to find a foreign agent in Tokyo. It needs to know what address to use while in Tokyo. It needs to communicate with its home agent back in London to tell it that it is in Tokyo and that the agent should start forwarding datagrams. Furthermore, it must accomplish its communication without any telephone.

To this end, Mobile IP includes a host of special functions that are used to set up and manage datagram forwarding. To see how these support functions work, let's look at the general operation of Mobile IP as a simplified series of steps:

1. **Agent Communication** The mobile node finds an agent on its local network by engaging in the *Agent Discovery* process. It listens for Agent Advertisement messages that are sent out by agents, and from this it can determine where it is located. If it doesn't hear these messages it can ask for one using an Agent Solicitation message.

2. **Network Location Determination** The mobile node determines whether it is on its home network or on a foreign one by looking at the information in the Agent Advertisement message.

If it is on its home network, it functions using regular IP. To show how the rest of the process works, let's say the device sees that it just moved to a foreign network. The remaining steps are as follows:

1. **Care-Of Address Acquisition** The device obtains a temporary address called a *care-of address*. This either comes from the Agent Advertisement message from the foreign agent or through some other means. This address is used only as the destination point for forwarding datagrams, and for no other purpose.
2. **Agent Registration** The mobile node informs the home agent on its home network of its presence on the foreign network and enables datagram forwarding by *registering* with the home agent. This may be done either directly between the node and the home agent or indirectly using the foreign agent as a conduit.
3. **Datagram Forwarding** The home agent captures datagrams intended for the mobile node and forwards them. It may send them either directly to the node or indirectly to the foreign agent for delivery, depending on the type of care-of address in use.

Datagram forwarding continues until the current agent registration expires. The device can then renew it. If it moves again, it repeats the process to get a new care-of address and then registers its new location with the home agent. When the mobile node returns to its home network, it *deregisters* to cancel datagram forwarding and resumes normal IP operation.

The following sections look in more detail at the functions summarized in each of the previous steps.

## Mobile IP Addressing: Home and Care-Of Addresses

Just as most of us have only a single address used for mail, most IP devices have only a single address. Our traveling consultant, however, needs to have two addresses; a normal one and one that is used while he is away. Continuing the earlier analogy, the Mobile IP-equipped notebook the consultant carries needs to have two addresses:

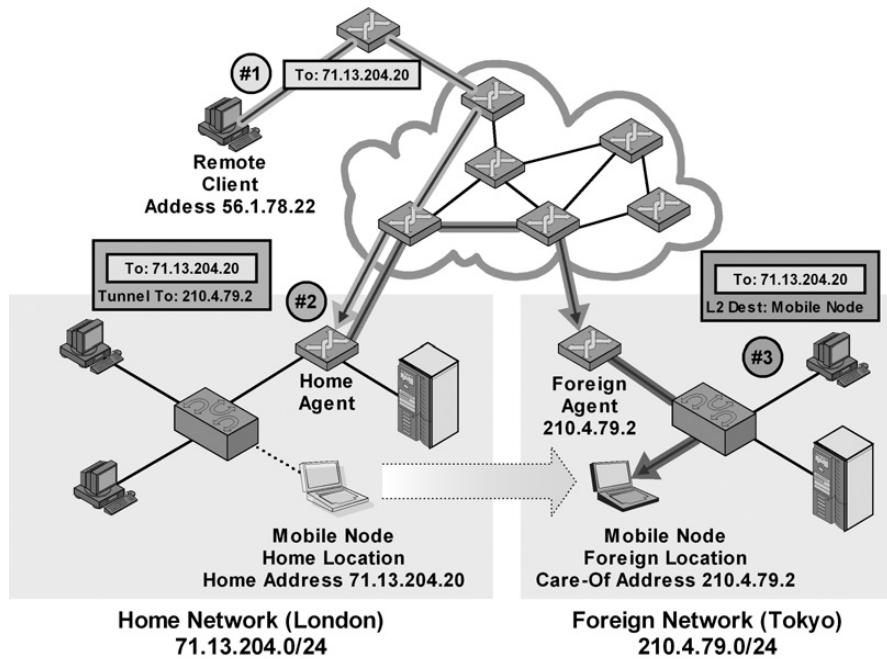
**Home Address** The normal, permanent IP address assigned to the mobile node. This is the address used by the device on its home network, and the one to which datagrams intended for the mobile node are always sent.

**Care-Of Address** A secondary, temporary address used by a mobile node while it is traveling away from its home network. It is a normal 32-bit IP address in most respects, but is used only by Mobile IP for forwarding IP datagrams and for administrative functions. Higher layers never use it, nor do regular IP devices when creating datagrams.

The care-of address is a slightly tricky concept. There are two different types, which correspond to two distinctly different methods of forwarding datagrams from the home agent router.

### **Foreign Agent Care-Of Address**

The care-of address is provided by a foreign agent in its *Agent Advertisement* message. It is, in fact, the IP address of the foreign agent itself. When this type of care-of address is used, all datagrams captured by the home agent are not relayed directly to the mobile node, but indirectly to the foreign agent, which is responsible for final delivery. In this arrangement, the mobile node has no distinct IP address valid on the foreign network, so this is typically done using a layer 2 technology. This arrangement is illustrated in Figure 30-3.



**Figure 30-3: Mobile IP operation with a foreign agent care-of address** This diagram is similar to Figure 30-2, except that instead of the mobile node having a co-located (distinct) IP address, here the mobile node is using a foreign agent care-of address. This means that the node's care-of address is actually that of the foreign agent itself. Step 1 is the same as in Figure 30-2, but in step 2, the home agent forwards not to the mobile node directly, but to the foreign agent (since that router is the one whose IP address the mobile is using). In step 3, the foreign agent strips off the home agent's packaging and delivers the original datagram to the mobile node. This is typically done using whatever layer 2 (LAN or WLAN) technology connects the mobile node and foreign agent.

In the consultant analogy, this type of care-of address is like forwarding from the London PO to the Tokyo PO. The London personnel would take a letter for John Smith sent to his London address, and repackage it for delivery to John Smith, care of the Tokyo post office. The Tokyo PO (or John Smith himself) would need to worry about the last leg of the delivery.

## **Co-Located Care-Of Address**

The co-located care-of address is assigned directly to the mobile node using some means that is external to Mobile IP. For example, it may be assigned on the foreign network manually, or it may be assigned automatically using DHCP. In this situation, the care-of address is used to forward traffic from the home agent directly to the mobile node. This was the type of address shown earlier in Figure 30-2.

In the consultant analogy, this is like John Smith obtaining a temporary address for his use while in Tokyo. The London PO would forward directly to his Tokyo address. They would not specifically send it to the Tokyo PO (although that PO would handle the mail at some point).

**KEY CONCEPT** In Mobile IP, each mobile device uses a temporary care-of address while on a foreign network. A co-located care-of address is one that is assigned directly to the mobile node and enables direct delivery of datagrams to the node. The alternative is to use a foreign agent care-of address. In this situation, the mobile node actually uses the IP address of the foreign agent. Datagrams are sent to the foreign agent, which delivers them to the mobile node.

## **Advantages and Disadvantages of the Care-Of Address Types**

The foreign agent care-of address is considered the type used in classic Mobile IP, where there is both a home agent and a foreign agent. While it seems less efficient than the co-located address method, it offers some important advantages, a key one being that the same foreign agent care-of address can be used for all mobile nodes visiting that network. Datagrams for all mobile nodes on that network are sent to the foreign agent, which completes the delivery to the individual nodes. Since the mobile nodes use the foreign agent's address, no extra addresses or extra work is required for each mobile node.

The co-located care-of address has the advantage that traffic can be forwarded directly from the home agent to the mobile node. In this type of arrangement, it is possible for a Mobile IP device to travel to a foreign network where there is no Mobile IP-aware router to act as a foreign agent. This does mean, however, that the Mobile IP implementation must include all the functions of communicating with the home agent that the foreign agent normally performs.

When co-located care-of addresses are used, an issue is how the temporary address is obtained. In many foreign networks, automatic assignment of an IP address using something like DHCP may be possible, but if not, a temporary IP address would need to be assigned. Either way, some of the foreign network's limited IP address space would need to be set aside for mobile nodes, each of which would use an address while present on the network. In some cases, this could lead to an address depletion issue.

Foreign agent care-of addressing is usually preferred due to its more automatic nature, when a foreign agent is present on the visited network. Considering that all datagrams will need to go through some router on the foreign network to reach the mobile node anyway, we might as well save the extra IP addresses. Co-located care-of addresses would be used when there is no foreign agent, or might be practical for long-term connections even when a foreign agent is present.

**KEY CONCEPT** In Mobile IP, *co-located care-of addresses* have the advantage of flexibility, but require each device to have a unique IP address on the remote network. Foreign agent care-of addresses have the chief advantage of allowing many mobile devices on a foreign network without each requiring a distinct IP address.

Remember that the care-of address represents only the destination to which mobile node datagrams are forwarded. Foreign agents provide services other than forwarding, so it is possible for a mobile node to use a co-located care-of address even when a foreign agent is present, while continuing to take advantage of the other foreign agent services.

For more information on how datagrams are forwarded between the home agent and the mobile node's care-of address, see the section on Mobile IP encapsulation and tunneling, later in this chapter.

## Mobile IP Agent Discovery

When a mobile node is first turned on, it cannot assume that it is still at home, the way normal IP devices do. It must first determine where it is, and if it is not at home, begin the process of setting up datagram forwarding from its home network. This process is accomplished by communicating with a local router that's serving as an agent through the process called *Agent Discovery*.

### Agent Discovery Process

Agent discovery encompasses the first three steps in the simplified five-step Mobile IP operational summary I gave earlier in discussing general Mobile IP operation. The main goals of Agent Discovery include the following:

**Agent/Node Communication** Agent Discovery is the method by which a mobile node first establishes contact with an agent on the local network to which it is attached. Messages containing important information about the agent are sent from the agent to the node. A message can also be sent from the node to the agent asking for this information to be sent.

**Orientation** The node uses the Agent Discovery process to determine where it is. Specifically, it learns whether it is on its home network or a foreign network by identifying the agent that sends it messages.

**Care-Of Address Assignment** The Agent Discovery process is the method used to tell a mobile node the care-of address it should use, when foreign agent care-of addressing is used.

Mobile IP agents are routers that have been given additional programming to make them Mobile IP-aware. The communication between a mobile node and the agent on its local network is basically the same as the normal communication required between a device on an IP network and its local router, except more information needs to be sent when the router is an agent.

## **Agent Advertisement and Agent Solicitation Messages**

Provision already exists for exchanges of data between a router and a node in the form of Internet Control Message Protocol (ICMP) messages that are used for the regular IP *Router Discovery* process. Two messages are used for this purpose: Router Advertisement messages that let routers tell local nodes that they exist and describe their capabilities, and Router Solicitation messages that let a node prompt a router to send an advertisement. These are described in Chapter 33.

Given the similarity to normal Router Discovery, it made sense to implement Agent Discovery as a modification to the existing process rather than set up a whole new system. The messages used in the Agent Discovery process are as follows:

**Agent Advertisement** This is a message transmitted regularly by a router acting as a Mobile IP agent. It consists of a regular Router Advertisement message that has one or more *extensions* added that contain Mobile IP-specific information for mobile nodes.

**Agent Solicitation** This message can be sent by a Mobile IP device to nudge a local agent to send an Agent Advertisement message.

The use of these messages is described in the Mobile IP standard in detail, and unsurprisingly, is very similar to how regular Router Advertisement and Router Solicitation messages are employed. Agents are normally configured to send out Agent Advertisements on a regular basis, with the rate set to ensure reasonably fast contact with mobile nodes without consuming excessive network bandwidth. They are required to respond to any Agent Solicitation messages they receive by sending an Advertisement. It is possible that some agents may be configured to send Advertisements only upon receipt of a Solicitation.

Mobile nodes are required to accept and process Agent Advertisements. They distinguish these from regular Router Advertisements by looking at the size of the message. They then parse the extension(s) to learn the capabilities of the local agent. They determine whether they are on their home network or a foreign network, and in the case of a foreign agent, how the agent should be used. Mobile nodes are required to use Agent Advertisements to detect when they have moved, using one of two algorithms defined in the standard. Mobile nodes are also required to detect when they have returned to their home network after they have been traveling. Finally, they are also required to be able to send Agent Solicitation messages if they don't receive an Agent Advertisement after a certain period of time. They are restricted to sending these only infrequently, however, in order to keep traffic manageable.

Now let's look at the formats of the two message types.

### **Agent Solicitation Message Format**

The Agent Solicitation message is simple. In fact, there is no new message format defined for this at all; it is identical to the format of a Router Solicitation message (see Chapter 33).

The reason no new message type is required here is that a solicitation is an extremely simple message: “Hey, if there are any routers out there, please tell me who you are and what you can do.” No extra Mobile IP information needs to be sent. When a regular IP router receives a Router Solicitation, it will send a Router Advertisement, but a Mobile IP router automatically sends the longer Agent Advertisement instead when prompted by any solicitation, whether it comes from a Mobile IP node or a regular IP device.

### **Agent Advertisement Message Format**

The Agent Advertisement begins with the normal fields of an ICMP Router Advertisement message (see Chapter 33). The destination of the message is either the “all devices” multicast address (224.0.0.1) if multicast is supported on the local network, or the broadcast address (255.255.255.255) otherwise. The *Router Address* fields are filled in with the address(es) of the agent.

**NOTE** *It is possible that a device may wish to advertise its ability to handle Mobile IP messages, but not act as a regular router. In this case it changes the normal Code field in the header of the Router Advertisement message from 0 to 16.*

Following the regular fields, one or more extensions are added:

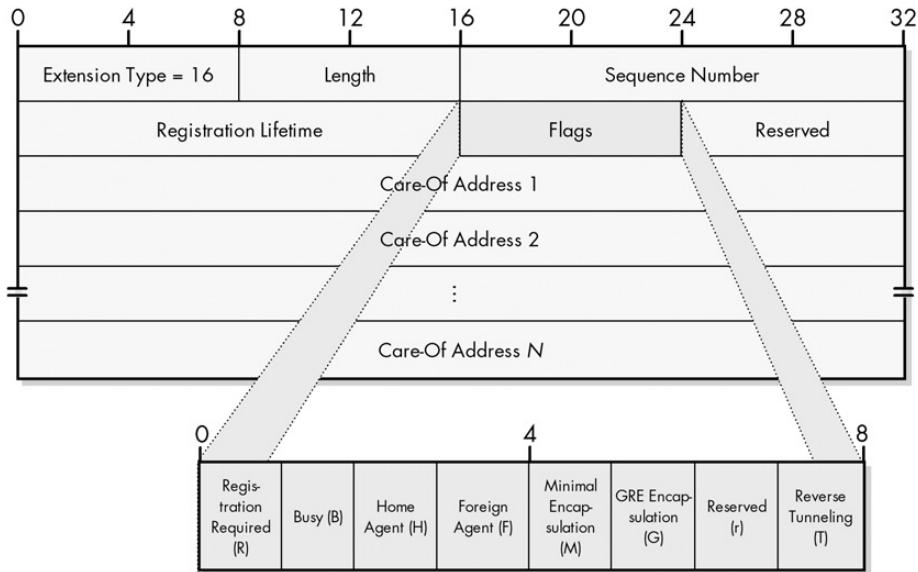
**Mobility Agent Advertisement Extension** This is the main extension used to convey Mobile IP capabilities of the agent to mobile nodes on the local network. This field is described in Tables 30-1 and 30-2 and illustrated in Figure 30-4.

**Prefix-Lengths Extension** This is an optional extension that tells a mobile node the prefix length(s) of the router address(es) contained in the regular portion of the Agent Advertisement message; that is, the Router Address field in the regular Router Advertisement part of the message. The prefix length is another term for the number of bits of a network ID in an address, so this specifies the network ID in the router addresses. This field is described in Table 30-3 and illustrated in Figure 30-5.

**One-Byte Padding Extension** Some implementations require ICMP messages to be an even number of bytes, so a byte of padding is needed. This field is just a single byte of all zeros.

**KEY CONCEPT** Mobile IP Agent Discovery is the process by which a mobile node determines where it is located and establishes contact with a home or foreign agent. To indicate their capabilities, routers that can function as agents regularly send *Agent Advertisement* messages, which are modified versions of regular *Router Advertisements*. To request the sending of an *Advertisement*, a mobile node can also send an *Agent Solicitation*, which is the same as a regular *Router Solicitation*.

I should point out that Mobile IP does not include any provisions for the authentication of *Agent Advertisement* and *Agent Solicitation* messages. They may be authenticated using IPsec, if that has been implemented.



**Figure 30-4: Mobile IP Mobility Agent Advertisement Extension format** This extension appears after the normal fields of a Router Advertisement message, as shown in Chapter 33.

**Table 30-1: Mobile IP Mobility Agent Advertisement Extension Format**

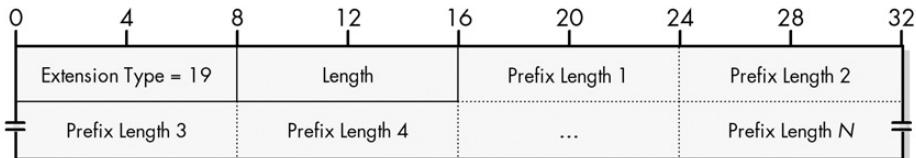
Field Name	Size (Bytes)	Description
Type	1	Identifies the Agent Advertisement extension type. For the Mobility Agent Advertisement Extension, it is set to 16.
Length	1	Length of the extension in bytes, excluding the Type and Length fields. Thus, it is equal to 6 plus 4 for each care-of address in the message.
Sequence Number	2	A sequential counter is set to zero when the router initializes and then incremented for each advertisement sent out.
Registration Lifetime	2	The maximum length of time, in seconds, that the agent is willing to accept for registration requests. A value of 65,535 (all ones) means infinite. Note that this field is for registration only and has no relation to the regular Lifetime field in the regular Router Advertisement part of the message.
Flags	1	A one-byte field containing several informational flags that convey specific information about the agent's capabilities and status. There are seven one-bit flags, which, when set, convey the meanings shown in Table 30-2.
Reserved	1	Sent as zero and ignored by recipient.
Care-Of Addresses	Variable (4 per address)	Zero or more addresses provided by a foreign agent for a mobile node to use as a foreign agent care-of address. A foreign agent must always provide at least one address in its advertisement. A router that cannot act as a foreign agent will typically omit this field.

**Table 30-2:** Mobile IP Mobility Agent Advertisement Extension Flags

Subfield Name	Size (Bytes)	Description
R	1/8 (1 bit)	Registration Required: The mobile node must register through the foreign agent, even when using a co-located care-of address.
B	1/8 (1 bit)	Busy: The agent is currently too busy to accept further registrations from mobile nodes.
H	1/8 (1 bit)	Home Agent: The agent is willing to function as a home agent on this link (it will forward datagrams, and so on). Note that a device can offer services as both a home agent and a foreign agent.
F	1/8 (1 bit)	Foreign Agent: The agent is willing to function as a foreign agent. Again, a device can act as both a home agent and a foreign agent simultaneously.
M	1/8 (1 bit)	Minimal Encapsulation: The agent can receive tunneled datagrams using minimal encapsulation.
G	1/8 (1 bit)	GRE Encapsulation: The agent can receive tunneled datagrams using GRE encapsulation.
r	1/8 (1 bit)	Reserved: Not used; sent as zero.
T	1/8 (1 bit)	Reverse Tunneling: The agent supports reverse tunneling.

**Table 30-3:** Mobile IP Prefix-Lengths Extension Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the Agent Advertisement extension type. For the Prefix-Lengths Extension, it is set to 19.
Length	1	Length of the extension in bytes, excluding the Type and Length fields. Thus, it is equal to the number of prefix lengths (since each takes 1 byte).
Prefix Lengths	Variable (1 per length)	One prefix length number for each router address in the regular, Router Advertisement portion of the Agent Advertisement.

**Figure 30-5: Mobile IP Prefix-Lengths Extension format** This extension appears after the normal fields of a Router Advertisement message, as shown in Chapter 33.

See the section on Mobile IP encapsulation later in this chapter for details on minimal and Generic Routing Encapsulation (GRE) encapsulation and reverse tunneling.

# Mobile IP Home Agent Registration and Registration Messages

Once a mobile node has completed Agent Discovery, it knows whether it is on its home network or a foreign network. If it's on its home network, it communicates as a regular IP device, but if it's on a foreign network, it must activate Mobile IP. This requires that it communicate with its home agent so that information and instructions can be exchanged between the two. This process is called *home agent registration*, or more simply, just *registration*.

The main purpose of registration is to actually start Mobile IP working. The mobile node must contact the home agent and tell it that it is on a foreign network and request that datagram forwarding be turned on. It also must let the home agent know its care-of address so the home agent knows where to send the forwarded datagrams. The home agent needs to communicate various types of information back to the mobile node when registration is performed. Note that the foreign agent is not really involved in registration, except perhaps to relay messages.

## Mobile Node Registration Events

Successful registration establishes what is called in the standard a *mobility binding* between a home agent and a mobile node. For the duration of the registration, the mobile node's regular home address is tied to its current care-of address, and the home agent will encapsulate and forward datagrams addressed to the home address over to the care-of address. The mobile node is supposed to manage its registration and handle various events using the following actions:

**Registration** The mobile node initiates a *registration* when it first detects it has moved from its home network to a foreign network.

**Deregistration** When the mobile node returns home, it should tell the home agent to cancel forwarding—a process called *deregistration*.

**Reregistration** If the mobile node moves from one foreign network to another, or if its care-of address changes, it must update its registration with the home agent. It also must do so if its current registration is about to expire, even if it remains stationary on one foreign network.

Each registration is established only for a specific length of time, which is why regular reregistration is required whether or not the device moves. Registrations are time-limited to ensure that they do not become stale. If, for example, a node forgets to deregister when it returns home, the datagram forwarding will eventually stop when the registration expires.

## Registration Request and Registration Reply Messages

To perform registration, two new message types have been defined in Mobile IP: the *Registration Request* and the *Registration Reply*. Each of these does what you would expect from its name. Interestingly, these are not ICMP messages like the ones used in Agent Discovery; they are User Datagram Protocol (UDP) messages. Thus,

technically speaking, registration is performed at a higher layer than the rest of Mobile IP communication. Agents listen for Registration Requests on well-known UDP port 434, and respond back to mobile nodes using whatever ephemeral port the node used to send the message.

## ***Registration Process***

There are two different procedures defined for registration, depending on the type of care-of address used by the mobile node, and other specifics that I will get into shortly. The first is the direct registration method, which has only two steps:

1. The mobile node sends a Registration Request to the home agent.
2. The home agent sends a Registration Reply back to the mobile node.

In some cases, however, a slightly more complex process is required, whereby the foreign agent conveys messages between the home agent and the mobile node. In this situation, the process has four steps:

1. The mobile node sends a Registration Request to the foreign agent.
2. The foreign agent processes a Registration Request and forwards it to the home agent.
3. The home agent sends a Registration Reply to a foreign agent.
4. The foreign agent processes a Registration Reply and sends back to the mobile node.

The first, simpler method is normally used when a mobile node is using a co-located care-of address. In that situation, the node can easily communicate directly with the home agent, and the mobile node is also set up to directly receive information and datagrams from the home agent. When there is no foreign agent, this is obviously the only method available. It is also obviously the only method when a mobile node is deregistering with its home agent after it arrives back on the home network.

The second method is required when a mobile node is using a foreign care-of address. You'll recall that in this situation, the mobile node doesn't have its own unique IP address at all; it is using a shared address that was given to it by the foreign agent, which precludes direct communication between the node and the home agent. Also, if a mobile node receives an Agent Advertisement with the R flag set, it also should go through the foreign agent, even if it has a co-located care-of address.

Note that the foreign agent really is just a middleman; the exchange is still really between the home agent and the mobile node. However, the foreign agent can deny registration if the request violates whatever rules are in place for using the foreign network. It is for this reason that some foreign agents may require that they be the conduits for registrations even if the mobile node has a co-located care-of address. Of course, if the foreign agent can't contact the home agent the registration will not be able to proceed.

**KEY CONCEPT** Mobile IP *home agent registration* is the process by which a *mobility binding* is created between a home agent and a traveling mobile node to enable datagram forwarding to be performed. The mobile node that sends a Registration Request message performs registration, and the home agent returns a Registration Reply. The foreign agent may be required to act as a middleman in order to facilitate the transaction, but is otherwise not involved.

The previous description is really a highly simplified explanation of the basics of registration. The Mobile IP standard specifies many more details on exactly how agents and nodes perform registration, including particulars on when requests and replies are sent, how to handle various special conditions such as invalid requests, rules for how home agents maintain a table of mobility bindings, and much more. The standard covers the definition of extensions to the regular registration messages that support authentication, which is required for secure communications (see the section on security issues later in this chapter for more details). It also includes the ability to have a mobile node that maintains more than one concurrent binding, when needed.

### **Registration Request Message Format**

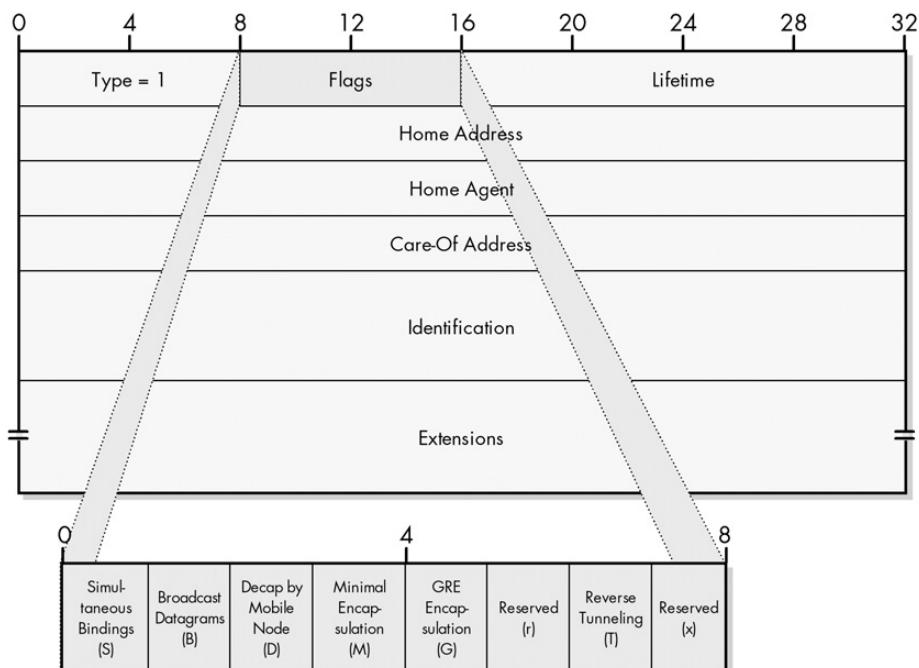
Registration Request messages have the format shown in Tables 30-4 and Table 30-5 and illustrated in Figure 30-6. See the section on Mobile IP encapsulation later in this chapter for details on minimal and GRE encapsulation and reverse tunneling.

**Table 30-4:** Mobile IP Registration Request Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the registration message type. For a request, this field is 1.
Flags	1	A one-byte field containing several informational flags that convey specific requests that are being made by the mobile node to the home agent. When set, the flags conveys the meanings shown in Table 30-5.
Lifetime	2	Length of time, in seconds, that the mobile node requests from the home agent for this registration.
Home Address	4	The home (normal) IP address of the mobile node when on its home network. Uniquely identifies the device regardless of how the request is conveyed to the home agent.
Home Agent	4	The IP address of the device acting as the mobile node's home agent.
Care-Of Address	4	The IP address being used by the mobile node as its care-of address.
Identification	8	A 64-bit number that uniquely identifies the <i>Registration Request</i> and is used to match requests to replies. It also provides protection against replay attacks; see the section on Mobile IP security issues later in this chapter for more information.
Extensions	Variable	Extension fields are included here for authentication of the request. Other extensions may also be included.

**Table 30-5: Registration Request Flags**

Subfield Name	Size (Bytes)	Description
S	1/8 (1 bit)	Simultaneous Bindings: Mobile node requests that prior mobility bindings be retained in addition to the one in the current request.
B	1/8 (1 bit)	Broadcast Datagrams: Mobile node requests that broadcasts on the home network be forwarded to it.
D	1/8 (1 bit)	Decapsulation by Mobile Node: Mobile node is telling the home agent that it will itself decapsulate encapsulated datagrams, as opposed to a foreign agent. In other words, when this is one, the mobile node is using a co-located care-of address; when zero, it is using a foreign agent care-of address.
M	1/8 (1 bit)	Minimal Encapsulation: Mobile node requests that home agent use minimal encapsulation for forwarded datagrams.
G	1/8 (1 bit)	GRE Encapsulation: Mobile node requests that home agent use GRE encapsulation for forwarded datagrams.
r	1/8 (1 bit)	Reserved: Not used; sent as zero.
T	1/8 (1 bit)	Reverse Tunneling: Mobile node requests that reverse tunneling be used by the home agent.
x	1/8 (1 bit)	Reserved: Not used; sent as zero.



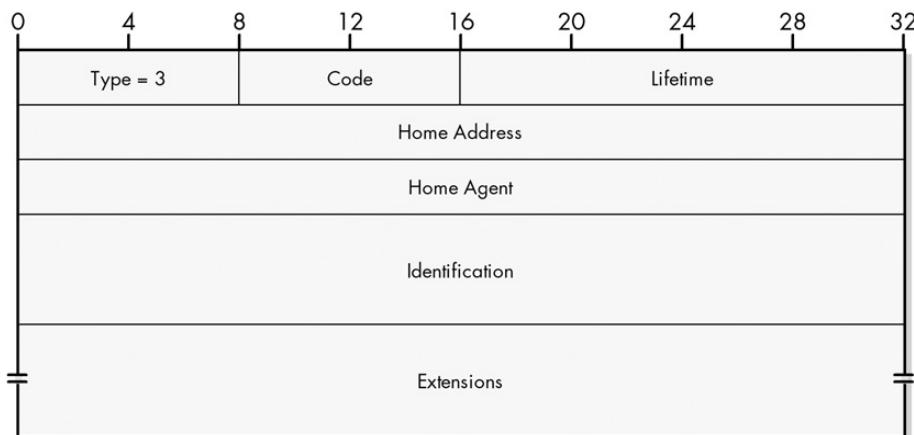
**Figure 30-6: Mobile IP Registration Request message format** This message is carried in the payload of a User Datagram Protocol (UDP) message, the headers of which are not shown.

## Registration Reply Message Format

Registration Reply messages are formatted as shown in Table 30-6 and illustrated in Figure 30-7.

**Table 30-6:** Mobile IP Registration Reply Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the registration message type. For a reply, this field is 3.
Code	1	Indicates the result of the registration request. This field is set to 0 if the registration was accepted, 1 if it was accepted but simultaneous bindings were requested and are not supported. If the registration was denied, a different reason code is provided that indicates the reason for the rejection, as well as whether it was the home agent or foreign agent that denied it.
Lifetime	2	If the registration was accepted, this represents the length of time in seconds until the registration expires. This may be a different value than the mobile node requested.
Home Address	4	The home (normal) IP address of the mobile node when it's on its home network. Uniquely identifies the device regardless of how the request is conveyed to the home agent so that the message can be delivered to it if the same foreign agent serves multiple mobile nodes.
Home Agent	4	The IP address of the device acting as the mobile node's home agent.
Identification	8	A 64-bit number that uniquely identifies the <i>Registration Reply</i> and is matched to the Identification field of the request that precipitated it.
Extensions	Variable	Extension fields are included here for the authentication of the reply. Other extensions may also be included.



**Figure 30-7: Mobile IP Registration Reply Message format** This message is carried in the payload of a UDP message, the headers of which are not shown.

## Mobile IP Data Encapsulation and Tunneling

Once a mobile node on a foreign network has completed a successful registration with its home agent, the Mobile IP datagram forwarding process described earlier in this chapter will be fully “activated.” The home agent will intercept datagrams

intended for the mobile node as they are routed to its home network, and forward them to the mobile node. Encapsulating the datagrams, and then sending them to the node's care-of address, does this.

Encapsulation is required because each datagram that you intercept and forward needs to be resent over the network to the device's care-of address. In theory, the designers might conceivably have done this by just having the home agent change the destination address and stick it back out on the network, but there are various complications that make this unwise. It makes more sense to take the entire datagram and wrap it in a new set of headers before retransmitting. In my mail-forwarding analogy, this is comparable to taking a letter received for the traveling consultant and putting it into a fresh envelope for forwarding, as opposed to just crossing off the original address and putting a new one on.

The default encapsulation process used in Mobile IP is called *IP Encapsulation within IP*, which is as it's defined in RFC 2003. It's commonly abbreviated *IP-in-IP*. It is a relatively simple method that describes how to take an IP datagram and make it the payload of another IP datagram. In Mobile IP, the new headers specify how to send the encapsulated datagram to the mobile node's care-of address.

In addition to IP-in-IP, the following two encapsulation methods may be optionally used: *Minimal Encapsulation within IP*, which is defined in RFC 2004, and *Generic Routing Encapsulation (GRE)*, which is defined in RFC 1701. To use either of these, the mobile node must request the appropriate method in its Registration Request, and the home agent must agree to use it. If foreign agent care-of addressing is used, the foreign agent also must support the method desired.

## **Mobile IP Conventional Tunneling**

The encapsulation process creates a logical construct called a *tunnel* between the device that encapsulates and the one that decapsulates. This is the same idea of a tunnel used in discussions of virtual private networks (VPNs), IPsec tunnel mode, or the various other tunneling protocols used for security. The tunnel represents a conduit over which datagrams are forwarded across an arbitrary internetwork, with the details of the encapsulated datagram (meaning the original IP headers) temporarily hidden.

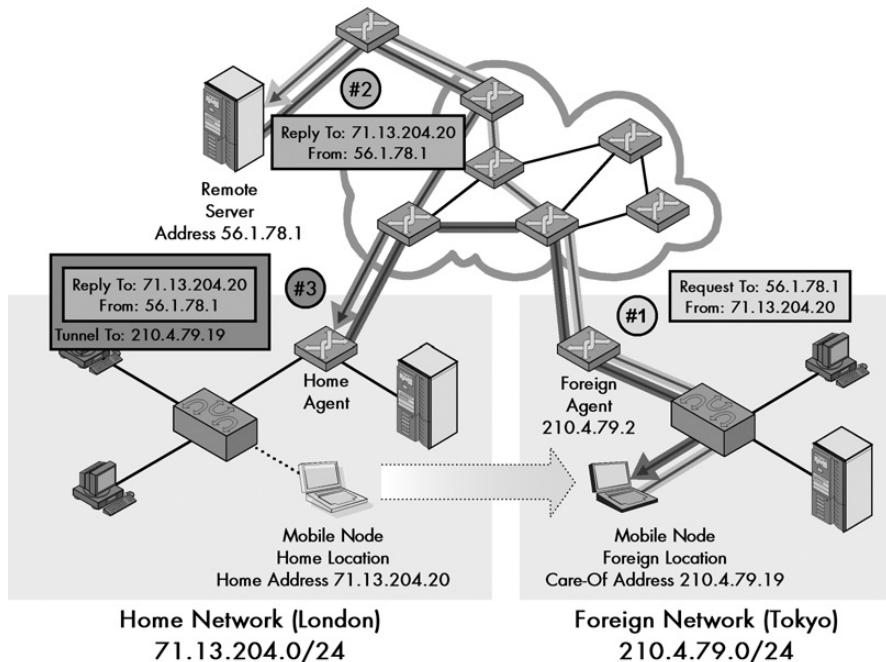
In Mobile IP, the start of the tunnel is the home agent, which does the encapsulation. The end of the tunnel depends on which of the two types of care-of address is being used:

**Foreign Agent Care-Of Address** The foreign agent is the end of the tunnel. It receives encapsulated messages from the home agent, strips off the outer IP header, and then delivers the datagram to the mobile node. This is generally done using layer 2, because the mobile node and foreign agent are on the same local network, and the mobile node does not have its own IP address on that network, because it is using the foreign agent's address.

**Co-Located Care-Of Address** The mobile node itself is the end of the tunnel and strips off the outer header.

Normally, the tunnel described previously is used only for datagrams that have been sent to the mobile node and captured by the home agent. When the mobile node wants to send a datagram, it doesn't tunnel it back to the home agent; this would be needlessly inefficient. Instead, it just sends out the datagram directly using whatever router it can find on its current network, which may or may not be a foreign agent. When it does this, it uses its own home address as the source address for any requests it sends. As a result, any response to those requests will go back to the home network. This sets up a triangle of three transmissions for these kinds of transactions (illustrated in Figure 30-8):

1. The mobile node sends a request from the foreign network to some third-party device somewhere on the internetwork.
2. The third-party device responds back to the mobile node. However, this sends the reply back to the mobile node's home address on its home network.
3. The home agent intercepts the response on the home network and tunnels it back to the mobile node.



**Figure 30-8: Mobile IP encapsulation and tunneling** This example illustrates how a typical request/reply message exchange in Mobile IP results in a triangle of communication. In step 1, the mobile node sends a request to a remote server somewhere on the Internet. It uses its own home address as the source for this request, so in step 2, the reply goes back to the home agent. Step 3 consists of the home agent tunneling the reply back to the mobile node.

The reverse transaction would be pretty much the same, except in the reverse order. In that case, the third-party (Internet) device would send a request to mobile node, which would be received and forwarded by the home agent. The mobile node would reply back directly to the Internet host.

**KEY CONCEPT** Once Mobile IP is set up and operational, it works by having the home agent *encapsulate* and *tunnel* received datagrams to the mobile node. The mobile device normally sends datagrams directly to Internet hosts, which respond back to the mobile's home agent, which forwards those datagrams to the mobile node. This means a request/reply communication takes three transmissions.

## Mobile IP Reverse Tunneling

There may be situations where it is not feasible or desired to have the mobile node send datagrams directly to the internetwork using a router on the foreign network, as you just saw. In this case, an optional feature called *reverse tunneling* may be deployed if it is supported by the mobile node, the home agent, and, if relevant, the foreign agent. When this is done, a reverse tunnel that complements the normal one is set up between the mobile node and the home agent, or between the foreign agent and the home agent, depending on the care-of address type. All transmissions from the mobile node are tunneled back to the home network where the home agent transmits them over the internetwork, thereby resulting in a more symmetric operation as opposed to the triangle just described. This is basically what I described earlier as being needlessly inefficient, because it means each communication requires four steps. Thus, it is used only when necessary.

One situation for which reverse tunneling may be required is if the network where the mobile node is located has implemented certain security measures that prohibit the node from sending datagrams using its normal IP address. In particular, a network may be set up to disallow outgoing datagrams with a source address that doesn't match its network prefix. This is often done to prevent *spoofing* (impersonating another's IP address).

**KEY CONCEPT** An optional feature called *reverse tunneling* may be used in certain cases, such as when a network does not allow outgoing datagrams with a foreign source IP address. When enabled, rather than sending datagrams directly, the mobile node tunnels all transmissions back to the home agent, which sends them on the Internet.

Note that everything I've just discussed is applicable to normal—meaning unicast—datagrams that are sent to and from the mobile node. Broadcast datagrams on the home network, which would normally be intended for the mobile node if it were at home, are not forwarded unless the node specifically asks for this service during registration. Multicast operation on the foreign network is also supported, but extra work is required by the mobile node to set it up.

## Mobile IP and TCP/IP Address Resolution Protocol (ARP) Operation

Mobile IP is a protocol that does a good job of implementing a difficult function: It transparently allows an IP device to travel to a different network. Unfortunately, a problem with any protocol that tries to change how IP works is dealing with special cases. Having a home agent intercept datagrams and tunnel them to the mobile

node works well in general terms, but there are some instances in which extra work is required. One of these is the use of ARP, which breaks under Mobile IP unless we take special steps.

**BACKGROUND INFORMATION** *Some understanding of how ARP works in general terms is assumed in this topic. This includes ARP proxying, which is described in Chapter 13.*

To understand what the problem is with ARP, consider a mobile node that is on a foreign network and has successfully registered with its home agent. The home agent will intercept all datagrams that come onto the home network, particularly the ones intended for the mobile node, and then encapsulate and forward them. For this to happen, though, the home agent (home router) must see the datagram. This normally occurs only when a datagram comes onto the home network from the outside and is processed by the router.

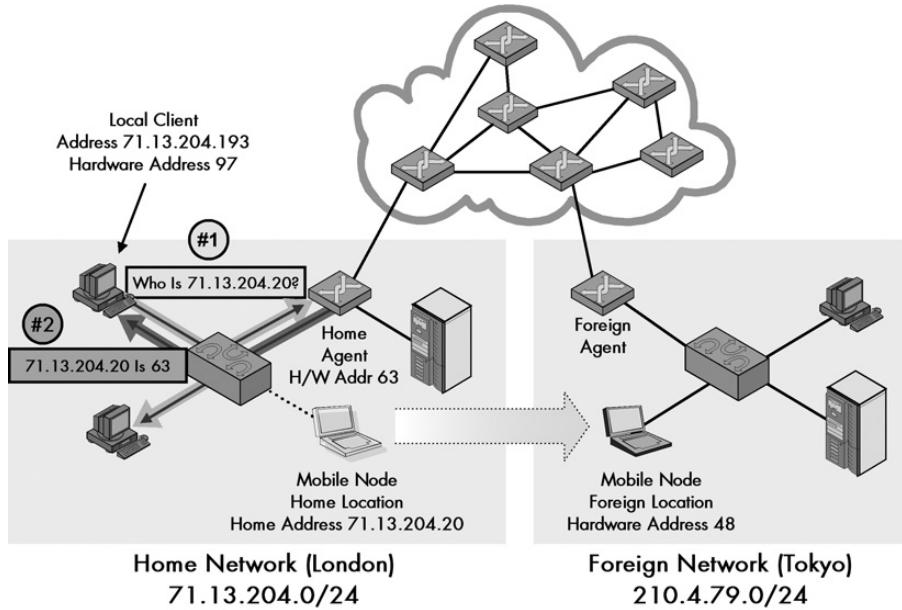
What happens when a local device on the home network itself wants to transmit to a mobile node that has traveled elsewhere? Remember that this device may not be mobile itself and probably knows nothing about Mobile IP. It will follow the standard process for deciding what to do with a datagram that it needs to send, as explained in Chapter 23. It will compare its network ID to that of the mobile node and realize that it doesn't need to route its datagram; it can send it directly to the mobile node.

The local host will attempt to use ARP to find the data link layer address of the mobile node so that it can send the datagram to it directly. The host will start by looking in its ARP cache, and if it finds the node's data link layer address there, it will use it to send at layer 2. The mobile node is no longer on the local network segment, so the message will never be received. If there is no ARP cache entry, the host on the home network will attempt to send an ARP Request to the mobile node to determine its layer 2 address. Again, the mobile node has traveled away, so this request will go unanswered.

Solving this problem requires the intervention of, you guessed it, the home agent. It must perform two tasks to enable local hosts to send to the mobile node:

**ARP Proxying** The home agent must listen for any ARP Requests that are sent by nodes on the same network as any of the mobile nodes that are currently registered to it. When it hears one, it replies in the mobile node's stead, and specifies its own data link layer address as the binding for the mobile node's IP address. This will cause hosts on the home network to send any datagrams that are intended for the mobile node to the home agent where they can be forwarded. This process is illustrated in Figure 30-9.

**Gratuitous ARP** Proxying helps with ARP Requests, but what about devices that already have cache entries for the mobile node? As soon as the mobile node leaves the network, these become automatically stale. To correct them, the home agent sends what is called a *gratuitous* ARP message, which tells devices on the local network to associate the mobile node's IP address with the home agent's data link layer address. The term *gratuitous* refers to the fact that the device isn't sending the message in order to perform an actual address resolution, but merely to cause caches to be updated. It may be sent more than once to ensure that every device gets the message.



**Figure 30-9: ARP proxying by Mobile IP home agent** The home agent must take special steps to deal with transmissions from devices on the local network to the mobile node. In this example (using short hardware addresses for simplicity), the hardware address of the mobile node is 48 and the home agent is 63. A local client on the home network with hardware address 97 sends an ARP Request to find out the hardware address of the mobile node. The home agent responds on the mobile's behalf, specifying not hardware address 48 but rather its own address: 63. The client will thus send to the home agent, which can then forward the data to the mobile node on the foreign network.

**KEY CONCEPT** In theory, problems can occur with hosts on the mobile node's home network that are trying to send datagrams to the host at layer 2. To address these issues, the home agent is required to use proxy ARP to direct such devices to send to the home agent so they can be forwarded. It must also use gratuitous ARP to update any existing ARP caches to that effect.

Once these steps are taken, ARP should function normally on the home link. When the mobile device returns back to the home network, the process must be reversed. Upon deregistration with the home agent, the mobile device will stop proxying for the mobile node. Both the mobile node and the home agent will also send gratuitous ARP broadcasts that update local device caches. These will again associate the mobile node's IP address with its own layer 2 address, instead of the layer 2 address of the home agent.

## Mobile IP Efficiency Issues

Having the home agent forward all datagrams to the mobile node wherever it may be is a convenient solution to mobility, but it's also a rather inefficient one. Since the device must send every datagram first to the home network and then forward it

to the mobile node, the datagrams are going to travel over some part of the internetwork twice. The degree of inefficiency represented by forwarding can be significant and may lead to problems with certain applications.

To see what the problem is, let's consider a traveling mobile Node M and a regular device that wants to send to it, Device A. The degree of the inefficiency of Mobile IP is a function of the internetwork distance between Device A and Node M's home network, compared to the internetwork distance between Device A and Node M's current network. By *distance*, I mean the term as it is used in determining routes on an internetwork. Two devices are closer when it takes less time and fewer hops to communicate between them, and they are farther when more hops are required. (I use geography in the following examples to represent this notion of distance, but remember that geographical distance is only one factor in internetwork distance.)

Let's consider the case in which mobile Node M is on a foreign network that's quite far from home, and Device A wants to send a datagram using Node M's home IP address. Suppose the home network is in London and the device is again in Tokyo. Let's look at the inefficiency factor of Mobile IP, compared to the alternative of having the mobile node just get a new temporary IP address on the foreign network and not use Mobile IP. The following examples are arranged in order of increasing inefficiency:

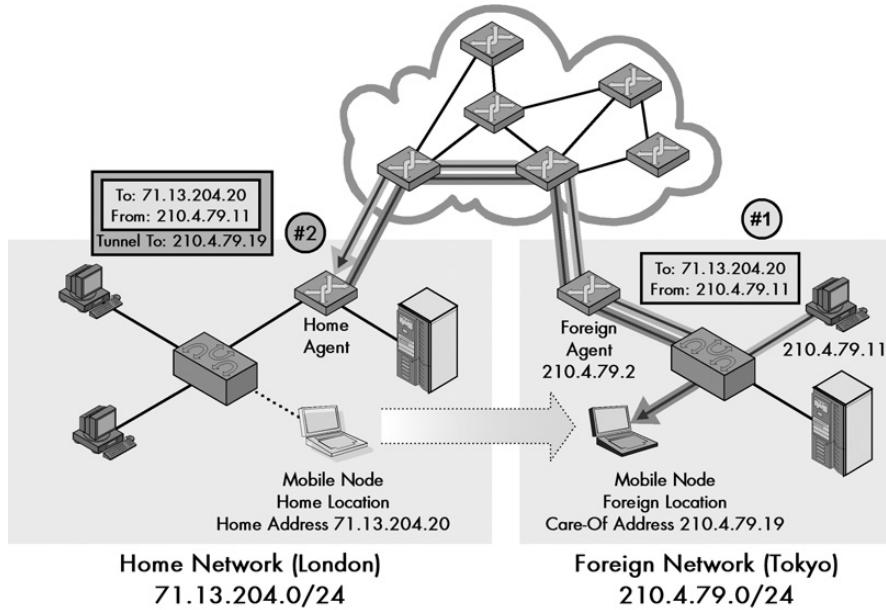
**Sending Device on Home Network** In this situation, Device A will send a datagram that is immediately intercepted by the home agent on the home network and forwarded to the mobile node. There is really no inefficiency here at all (except for overhead for encapsulation and such), because even if Device A did send the datagram directly to the mobile node with a new foreign address, the datagram would probably be routed through the home agent router anyway.

**Sending Device on Network Close to Home Network** Let's say a device in Paris wants to send to the mobile node. The datagram goes from Paris to London and then to Tokyo. That's not too bad.

**Sending Device on Network Close to Foreign Network** Now suppose the sending device is in Taipei, Taiwan. In this situation, Mobile IP becomes quite inefficient. The datagram must be sent from Taipei all the way to London, and then all the way back to Tokyo.

**Sending Device on Foreign Network** The greatest inefficiency occurs when the sending device is actually on the foreign network that the mobile node is visiting. If Device A is on the mobile node's current network in Tokyo, it must send all the way to London, and then have the result forwarded all the way back again to Tokyo. Without Mobile IP, all you would need to do is use ARP and then deliver directly at layer 2 without needing routing at all! This scenario is illustrated in Figure 30-10.

Unfortunately, the worst-case scenario of the sending device on a foreign network is one that occurs quite often. It's common for a mobile device to connect with a foreign network in order for it to communicate specifically with the hosts on that network.



**Figure 30-10: A Mobile IP inefficiency worst-case scenario** This diagram shows the worst possible case of Mobile IP inefficiency. When a device on the foreign network where the mobile is located tries to send data to the mobile device. The sender here, 210.4.79.11, uses the mobile node's home address so that the transmission must be routed all the way back to London, and then forwarded back to Tokyo, even though the two devices might be sitting on the same desk!

To make matters worse, consider what happens if reverse tunneling is used! Here, tunneling is done not just for datagrams sent to the mobile node, but for datagrams sent from the device as well. In the worst-case example, a request/reply pair from the mobile node to another device on the foreign network requires *two* complete round-trips from Tokyo to London and back. Clearly, this is far from ideal.

**KEY CONCEPT** Since datagrams are sent to a Mobile IP at its home address, each datagram sent to the mobile device must first go back to its home network and then be forwarded to its current location. The level of inefficiency that results depends on how far the sender is from the mobile's home network. The worst case actually occurs if the sender and mobile are on the same foreign network, in which case each transmission must make a round-trip to the mobile's home network and then back again.

There really isn't any solution to this problem within Mobile IP itself; it's just a natural consequence of how the protocol works. The only way to really improve things is to "hack in" a solution that ultimately boils down to one of the two options we always have in IP without mobility support: Either give the mobile device a temporary real IP address on the foreign network, or use a host-specific route for the mobile device while it's on the foreign network.

You've already seen that these both have problems, which is why Mobile IP was created in the first place. There may be situations, however, in which efficiency is more important than the transparent portability that Mobile IP provides. For a

long-term deployment on a foreign network far from the home network, or for applications where efficiency is paramount, it may make sense to employ one of these techniques. For example, a corporation that has a small number of offices in different cities that are connected using the Internet might set up special routing. This would let mobile devices visiting from other cities talk directly to nodes that are local to the foreign part of the network without being routed across the Internet.

## Mobile IP Security Considerations

Security is always a concern in any internetworking environment these days, but is especially important with Mobile IP. There are a number of reasons for this. The reasons are related to both how the protocol is used and the specific mechanisms by which it is implemented.

In terms of use, security was kept in mind during Mobile IP's development because mobile devices often use wireless networking technologies. Wireless communication is inherently less secure than wired communication, because transmissions are sent out in the open, where they can be intercepted. It's also easier for malicious users to disrupt the operation of wireless devices.

In terms of operation, Mobile IP has a number of risks due to the fact that it uses a registration system and then forwards datagrams across an unsecured internetwork. A malicious device could interfere with registration process, thereby causing the datagrams intended for a mobile device to be diverted. A bad guy might also interfere with the data forwarding process itself by encapsulating a bogus datagram to trick a mobile node into thinking it was sent something that it never was.

For these reasons, the Mobile IP standard includes a limited number of explicit provisions to safeguard against various security risks. One security measure was considered sufficiently important that it was built into the Mobile IP standard directly: the authentication of Registration Request and Registration Reply messages. This authentication process is accomplished in a manner that's somewhat similar to how the IPsec Authentication Header (AH) operates, as described in Chapter 29. Its goal is to prevent unauthorized devices from intercepting traffic by tricking an agent into setting up, renewing, or canceling a registration improperly.

All Mobile IP devices are required to support authentication. Nodes must use it for requests, and agents must use it for replies. Keys must be assigned manually because there is no automated system for secure key distribution. The default authentication method uses *HMAC-MD5* (specified in RFC 2403), which is one of two hashing algorithms used by IPsec.

Another concern is a security problem called a *replay attack*. In this type of attack, a third party intercepts a datagram, holds on to it, and then resends it later on. This seems fairly harmless, but consider the importance of timing. Imagine a mobile node that registers with its home agent, and then later returns home and deregisters. If a malicious device captures a copy of the original Registration Request and resends it, the home agent might be fooled into thinking the node has traveled away from home when it has not. It could then intercept the forwarded datagrams.

The Identification field used in Registration Request and Registration Reply messages is designed to prevent replay attacks. Since each request has a different Identification number, nodes and agents can match up requests with replies and reject any datagrams they receive that are repeats of ones they have seen already. The Mobile IP standard also specifies alternative methods for protecting against replays.

While Mobile IP includes authentication measures for registration messages, it does not for other types of messages. It also doesn't specify authentication of encapsulated datagrams being forwarded from the home agent to the mobile node. Encryption is also not provided to safeguard the privacy of either control messages or forwarded datagrams. The obvious solution when stronger assurances of privacy or authenticity are required is to make use of the IPsec AH and/or Encapsulating Security Payload (ESP) protocols (described in Chapter 29).

# PART II-6

## IP SUPPORT PROTOCOLS

The Internet Protocol (IP) is the key network layer protocol that implements the TCP/IP protocol suite. Since IP is the protocol that provides the mechanism for delivering datagrams between devices, it is designed to be relatively basic. For example, it lacks provisions for some way to allow errors to be reported back to a transmitting device, and for tests and special tasks to be accomplished. These auxiliary capabilities are necessary for the operation of an internetwork, however, so TCP/IP includes *support protocols* that help IP perform these tasks. This part examines the two main IP support protocols: the *Internet Control Message Protocol (ICMP)* and the *Neighbor Discovery (ND)* protocol.

The bulk of this part thoroughly describes ICMP, which was initially developed to be a companion to the original IP version 4 (IPv4). With the creation of IP version 6 (IPv6), a new version of ICMP, called ICMP version 6 (ICMPv6), was created as well. The original ICMP is now sometimes called *ICMPv4* to differentiate it, just as the original IP is now often called IPv4.

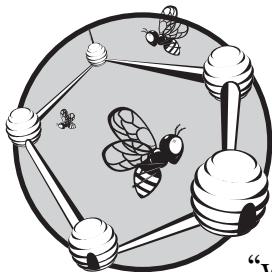
The two versions of ICMP have some differences in their specifics, but they are very similar in overall operation. For this reason, I have integrated the general operation description of both versions of ICMP in the first chapter of this part. The area where ICMPv4 and ICMPv6 most differ is in specific message types and formats, so these have been described separately in the second through fifth chapters. These chapters describe the error messages and informational messages in each version.

The final chapter describes ND, which was created specifically to assist in the operation of IPv6 and is closely related to ICMPv6.

Due to the close relationship between ICMP and IP, this part assumes that you are familiar with basic IP concepts, including IP addressing, the general format of IP datagrams, and how they are routed (covered in Part II-3). To better understand ICMPv6 details, you may also want to reference the IPv6 addressing and datagram encapsulation information (covered in Part II-4).

# 31

## **ICMP CONCEPTS AND GENERAL OPERATION**



The Internet Control Message Protocol (ICMP) is one of the underappreciated “worker bees” of the networking world. Everyone knows how important key protocols such as the Internet Protocol (IP) are to TCP/IP, but few realize that the suite as a whole relies on many functions that ICMP provides. Originally created to allow the reporting of a small set of error conditions, ICMP messages are now used to implement a wide range of error-reporting, feedback, and testing capabilities. While each message type is unique, they are all implemented using a common message format, sent, and then received based on relatively simple protocol rules. This makes ICMP one of the easiest TCP/IP protocols to understand. (Yes, I actually said something in this book was easy!)

In this chapter, I provide a general description of ICMP. I begin with an overview of ICMP, discussing its purpose, history, and the versions and standards that define it. I describe the general method by which ICMP operates and discuss the rules that govern how and when ICMP messages are created and processed. I then outline the common format used for ICMP messages in

versions 4 and 6 of the protocol (ICMPv4 and ICMPv6), and how data is encapsulated in them in general terms. I conclude with a discussion of ICMP message classifications and a summary of different message types and codes for both ICMPv4 and ICMPv6.

## ICMP Overview, History, Versions, and Standards

IP is the foundation of the TCP/IP protocol suite, because it is the mechanism responsible for delivering datagrams. Three of the main characteristics that describe IP's datagram delivery method are *connectionless*, *unreliable*, and *unacknowledged*. This means that datagrams are just sent over the internetwork with no prior connection established, no assurance they will show up, and no acknowledgment sent back to the sender that they arrived. On the surface, this seems like it would result in a protocol that is difficult to use and impossible to rely on, and thus would be a poor choice for designing a protocol suite. However, even though IP makes no guarantees, it works very well because most of the time, IP internetworks are sufficiently robust that messages get where they need to go.

Even the best-designed system still encounters problems, of course. Incorrect packets are occasionally sent, hardware devices have problems, routes are found to be invalid, and so forth. IP devices also often need to share specific information in order to guide them in their operation, and they need to perform tests and diagnostics. However, IP itself includes no provision that allows devices to exchange low-level control messages. Instead, these features are provided in the form of a companion protocol to IP called the *Internet Control Message Protocol (ICMP)*.

A good analogy for the relationship between IP and ICMP is to consider the one between a high-powered executive and her experienced administrative assistant. The executive is busy and her time is very expensive. She is paid to do a specific job and to do it well, and not to spend time on administrative tasks. However, without someone doing those tasks, the executive could not do her job properly. The administrative assistant does the important support jobs that make it possible for the executive to focus on her work. The working relationship between them is very important; a good pair will work together like a cohesive team, even anticipating each other's needs.

In TCP/IP, IP is the executive, and ICMP is its administrative assistant. IP focuses on its core activities, such as addressing, datagram packaging, and routing. ICMP provides critical support to IP in the form of *ICMP messages* that allow different types of communication to occur between IP devices. These messages use a common general format and are encapsulated in IP datagrams for transmission. They are divided into different categories, and each type has a specific use and internal field format.

Just as an administrative assistant often has a special location in an organization chart, and usually connects with a dotted line directly to the executive she assists, ICMP occupies a unique place in the TCP/IP protocol architecture (see Chapter 8). Technically, you might consider ICMP to belong to layer 4, because it creates messages that are encapsulated in IP datagrams and sent using IP at layer 3. However, in the standard that first defined it, ICMP is specifically declared to be not only part of the network layer, but also, as stated in RFC 792, is “actually an integral part of IP, [that] must be implemented by every IP module.” This was the initial defining

standard for ICMP, titled simply “Internet Control Message Protocol.” It was published at the same time as the standard for IP, which was RFC 791. This is further indication that IP and ICMP really are a team of sorts.

Due to the close relationship between the two, when the new version 6 of the Internet Protocol (IPv6) was developed in the mid-1990s, it was necessary to define a new version of ICMP as well. This was of course called the “Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification.” It was first published as RFC 1885 in 1995, and revised in RFC 2463 in 1998. Just as the original IP is now often called IPv4 to differentiate it from IPv6, the original ICMP is now also called *ICMPv4*.

**KEY CONCEPT** In TCP/IP, diagnostic, test, and error-reporting functions at the internetwork layer are performed by the *Internet Control Message Protocol (ICMP)*, which is like IP’s “administrative assistant.” The original version, now called ICMPv4, is used with IPv4, and the newer ICMPv6 is used with IPv6.

These two RFCs, 792 and 2463, define the basic operation of ICMPv4 and ICMPv6, respectively, and also describe some of the ICMP message types supported by each version of the protocol. ICMPv4 and ICMPv6 are very similar in most respects, although they have some differences, most of which are a direct result of the changes made to IP itself. Another document, RFC 1122, “Requirements for Internet Hosts—Communication Layers,” contains rules for how ICMPv4 is used, as you will see soon in the section on ICMP message creation and processing conventions later in this chapter. RFC 1812, “Requirements for IP Version 4 Routers,” is also relevant.

Both versions of the protocol define a general messaging system that was designed to be expandable. This means that in addition to the messages defined in the ICMP standards themselves, other protocols may also define message types used in ICMP. Some of the more important of these are shown in Table 31-1.

**Table 31-1:** Non-ICMP Internet Standards That Define ICMP Messages

<b>ICMP Version of Message Types Defined</b>	<b>RFC Number</b>	<b>Name</b>	<b>ICMP Message Types Defined</b>
ICMPv4	950	Internet Standard Subnetting Procedure	Address Mask Request, Address Mask Reply
	1256	ICMP Router Discovery Messages	Router Advertisement, Router Solicitation
	1393	Traceroute Using an IP Option	Traceroute
	1812	Requirements for IP Version 4 Routers	Defines three new codes (subtypes) for the Destination Unreachable message.
ICMPv6	2461	Neighbor Discovery for IP Version 6 (IPv6)	Router Advertisement, Router Solicitation, Neighbor Advertisement, Neighbor Solicitation, Redirect
	2894	Router Renumbering for IPv6	Router Renumbering

This chapter includes a full list of the ICMPv4 and ICMPv6 message types covered in this book and the standards that define each one.

## ICMP General Operation

ICMP is one of the simplest protocols in the TCP/IP protocol suite. Most protocols implement a particular type of functionality to either facilitate basic operation of a part of the network stack or an application. To this end, they include many specific algorithms and tasks that define the protocol, which is where most of the complexity lies. ICMP, in contrast, is exactly what its name suggests: a protocol that defines control messages. As such, pretty much all of what ICMP is about is providing a mechanism for any IP device to send control messages to another device.

### ***The ICMP Message-Passing Service***

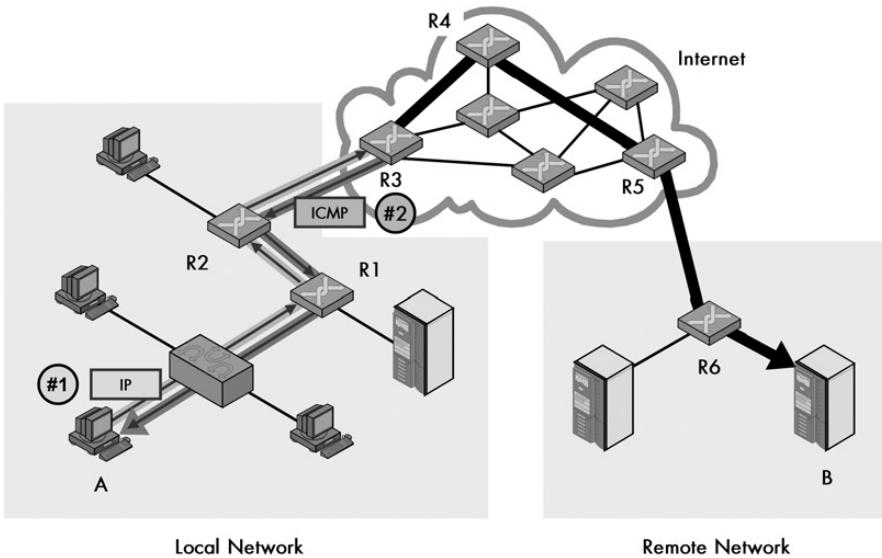
Various message types are defined in ICMP that allow different types of information to be exchanged. These are usually either generated for the purpose of reporting errors or for exchanging important information of different sorts that is needed to keep IP operating smoothly. ICMP itself doesn't define how all the different ICMP messages are used; this is done by the protocols that use the messages. In this manner, ICMP describes a simple message-passing service to other protocols.

**KEY CONCEPT** ICMP is not like most other TCP/IP protocols in that it does not perform a specific task. It defines a mechanism by which various control messages can be transmitted and received to implement a variety of functions.

As mentioned in the preceding section, ICMP is considered an integral part of IP, even though it uses IP to send its messages. Typically, the operation of ICMP involves some portion of the TCP/IP protocol software on a machine detecting a condition that causes it to generate an ICMP message. This is often the IP layer itself, though it may be some other part of the software. The message is then encapsulated and transmitted like any other TCP/IP message, and is given no special treatment compared to other IP datagrams. The message is sent over the internetwork to the IP layer at the receiving device, as shown in Figure 31-1.

Again, since many of the ICMP messages are actually intended to convey information to a device's IP software, the IP layer itself may be the ultimate destination of an ICMP message once a recipient gets it. In other cases, the ultimate destination may be some other part of the TCP/IP protocol software, which is determined by the type of message received. ICMP does not use ports like the User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) to direct its messages to different applications on a host. The software recognizes the message type and directs it accordingly within the software.

ICMP was originally designed with the idea that most messages would be sent by routers, but they can be sent by both routers and by regular hosts as well, depending on the message type. Some are obviously sent only by routers, such as Redirect messages; others may be sent by either routers or hosts. Many of the ICMP messages are used in matched pairs, especially in various kinds of Request and Reply messages, and Advertisement and Solicitation messages.



**Figure 31-1: ICMP general operation** A typical use of ICMP is to provide a feedback mechanism when an IP message is sent. In this example, Device A is trying to send an IP datagram to Device B. However, when it gets to Router R3, a problem of some sort is detected that causes the datagram to be dropped. Router R3 sends an ICMP message back to Device A to tell it that something happened, hopefully with enough information to let Device A correct the problem, if possible. Router R3 can only send the ICMP message back to Device A, not to Router R2 or R1.

### ICMP Error Reporting Limited to the Datagram Source

One interesting general characteristic of ICMP's operation is that when errors are detected, they can be reported using ICMP, but only back to the original source of a datagram. This is actually a big drawback in how ICMP works. Refer back to Figure 31-1 and consider again client Host A sending a message to server Host B, with a problem detected in the datagram by Router R3. Even if Router R3 suspects that the problem was caused by one of the preceding routers that handled the message, such as Router R2, it *cannot* send a problem report to Router R2. It can send an ICMP message only back to Host A.

This limitation is an artifact of how IP works. You may recall from looking at the IP datagram format that the only address fields are for the original source and ultimate destination of the datagram. (The only exception is if the IP Record Route option is used, but devices cannot count on this.) When Router R3 receives a datagram from Router R2 that Router R2 in turn received from Router R1 (and prior to that, from Device A), it is only Device A's address in the datagram. Thus, Router R3 *must* send a problem report back to Device A, and Device A must decide what to do with it. Device A may decide to change the route it uses or to generate an error report that an administrator can use to troubleshoot Router R2.

In addition to this basic limitation, several special rules and conventions have been put in place to govern the circumstances under which ICMP messages are generated, sent, and processed. I'll discuss these later in the chapter.

**KEY CONCEPT** ICMP error-reporting messages sent in response to a problem seen in an IP datagram can be sent back only to the originating device. Intermediate devices cannot be the recipients of an ICMP message because their addresses are normally not carried in the IP datagram's header.

## ICMP Message Classes, Types, and Codes

ICMP messages are used to allow the communication of different types of information between IP devices on an internetwork. The messages themselves are used for a wide variety of purposes, and they are organized into general categories as well as numerous specific types and subtypes.

### ICMP Message Classes

At the highest level, ICMP messages are divided into two classes:

**Error Messages** These messages are used to provide feedback to a source device about an error that has occurred. They are typically generated specifically in response to some sort of action, usually the transmission of a datagram, as shown in the example in Figure 31-1. Errors are usually related to the structure or content of a datagram or to problem situations on the internetwork encountered during datagram routing.

**Informational (or Query) Messages** These are messages that are used to let devices exchange information, implement certain IP-related features, and perform testing. They do not indicate errors and are typically not sent in response to a regular datagram transmission. They are generated either when directed by an application or on a regular basis to provide information to other devices. An informational ICMP message may also be sent in reply to another informational ICMP message, since they often occur in request/reply or solicitation/advertisement functional pairs.

**KEY CONCEPT** ICMP messages are divided into two general categories: *error messages* that are used to report problem conditions, and *informational messages* that are used for diagnostics, testing, and other purposes.

### ICMP Message Types

Each individual kind of message in ICMP is given its own unique Type value, which is put into the field of that name in the ICMP common message format. This field is 8 bits wide, so a theoretical maximum of 256 message types can be defined. A separate set of Type values is maintained for each of ICMPv4 and ICMPv6.

In ICMPv4, Type values were assigned sequentially to both error and informational messages on a first-come, first-served basis (sort of), so we cannot tell just by the Type value what type of message each is. One minor improvement made in ICMPv6 was that the message types were separated. In IPv6, error messages have Type values from 0 to 127, and informational messages have values from 128 to 255. Only some of the Type values are currently defined.

**KEY CONCEPT** A total of 256 different possible message types can be defined for each of ICMPv4 and ICMPv6. The Type field that appears in the header of each message specifies the kind of ICMP message. In ICMPv4, there is no relationship between Type value and message type. In ICMPv6, error messages have a Type value of 0 to 127, and informational messages have a Type value of 128 to 255.

## ICMP Message Codes

The message type indicates the general purpose of each kind of ICMP message. ICMP also provides an additional level of detail within each message type in the form of a Code field, which is also 8 bits. You can consider this field as a message subtype. Thus, each message type can have up to 256 subtypes that are more detailed subdivisions of the message's overall functionality. A good example is the Destination Unreachable message, which is generated when a datagram cannot be delivered. In this message type, the Code value provides more information on exactly why the delivery was not possible.

## ICMP Message Class and Type Summary

The next four chapters of the book describe all of the major ICMP message types for both ICMPv4 and ICMPv6. For convenience, I have summarized all these message types in Table 31-2, which shows each of the Type values for the messages covered in this book, the name of each message, a very brief summary of its purpose, and the RFC that defines it. (To keep the table from being egregiously large, I have not shown each of the Code values for each Type value; these can be found in the individual message type descriptions.) The table is organized into sections that correspond to the four chapters that describe ICMP message types, except this table is sorted by ascending Type value within each category for easier reference.

**Table 31-2:** ICMP Message Classes, Types, and Codes

Message Class	Type Value	Message Name	Summary Description of Message Type	Defining RFC Number
ICMPv4 Error Messages	3	Destination Unreachable	Indicates that a datagram could not be delivered to its destination. The Code value provides more information on the nature of the error.	792
	4	Source Quench	Lets a congested IP device tell a device that is sending it datagrams to slow down the rate at which it is sending them.	792
	5	Redirect	Allows a router to inform a host of a better route to use for sending datagrams.	792
	11	Time Exceeded	Sent when a datagram has been discarded prior to delivery due to expiration of its Time to Live field.	792
	12	Parameter Problem	Indicates a miscellaneous problem (specified by the Code value) in delivering a datagram.	792

(continued)

**Table 31-2: ICMP Message Classes, Types, and Codes (continued)**

<b>Message Class</b>	<b>Type Value</b>	<b>Message Name</b>	<b>Summary Description of Message Type</b>	<b>Defining RFC Number</b>
ICMPv4 Informational Messages (part 1 of 2)	0	Echo Reply	Sent in reply to an Echo (Request) message; used for testing connectivity.	792
	8	Echo (Request)	Sent by a device to test connectivity to another device on the internetwork. The word <i>Request</i> sometimes appears in the message name.	792
	9	Router Advertisement	Used by routers to tell hosts of their existence and capabilities.	1256
	10	Router Solicitation	Used by hosts to prompt any listening routers to send a Router Advertisement.	1256
	13	Timestamp (Request)	Sent by a device to request that another send it a timestamp value for propagation time calculation and clock synchronization. The word <i>Request</i> sometimes appears in the message name.	792
	14	Timestamp Reply	Sent in response to a Timestamp (Request) to provide time calculation and clock synchronization information.	792
	15	Information Request	Originally used to request configuration information from another device. Now obsolete.	792
ICMPv4 Informational Messages (part 2 of 2)	16	Information Reply	Originally used to provide configuration information in response to an Information Request message. Now obsolete.	792
	17	Address Mask Request	Used to request that a device send a subnet mask.	950
	18	Address Mask Reply	Contains a subnet mask sent in reply to an Address Mask Request.	950
	30	Traceroute	Used to implement the experimental enhanced traceroute utility.	1393
ICMPv6 Error Messages	1	Destination Unreachable	Indicates that a datagram could not be delivered to its destination. Code value provides more information on the nature of the error.	2463
	2	Packet Too Big	Sent when a datagram cannot be forwarded because it is too big for the maximum transmission unit (MTU) of the next hop in the route. This message is needed in IPv6 and not IPv4 because in IPv4, routers can fragment oversized messages, but in IPv6 they cannot.	2463
	3	Time Exceeded	Sent when a datagram has been discarded prior to delivery due to the Hop Limit field being reduced to zero.	2463
	4	Parameter Problem	Indicates a miscellaneous problem (specified by the Code value) in delivering a datagram.	2463

*(continued)*

**Table 31-2: ICMP Message Classes, Types, and Codes (continued)**

<b>Message Class</b>	<b>Type Value</b>	<b>Message Name</b>	<b>Summary Description of Message Type</b>	<b>Defining RFC Number</b>
ICMPv6 Informational Messages	128	Echo Request	Sent by a device to test connectivity to another device on the internetwork.	2463
	129	Echo Reply	Sent in reply to an Echo (Request) message; used for testing connectivity.	2463
	133	Router Solicitation	Prompts a router to send a Router Advertisement.	2461
	134	Router Advertisement	Sent by routers to tell hosts on the local network that the router exists. It also describes its capabilities.	2461
	135	Neighbor Solicitation	Sent by a device to request the layer 2 address of another device while providing its own as well.	2461
	136	Neighbor Advertisement	Provides information about a host to other devices on the network.	2461
	137	Redirect	Redirects transmissions from a host to either an immediate neighbor on the network or a router.	2461
	138	Router Renumbering	Conveys renumbering information for router renumbering.	2894

You can see that several of the message types are quite similar in ICMPv4 and ICMPv6, but there are some slight differences. An obvious one is that Redirect is considered an error message in ICMPv4, but it's an informational message in ICMPv6. Messages are often used differently as well. In IPv6, the use of many of the ICMP informational messages is described in the Neighbor Discovery (ND) protocol, which is new to IPv6 (see Chapter 36).

Note that the Information Request and Information Reply messages were originally created to allow devices to determine an IP address and possibly other configuration information. This function was later implemented using host configuration protocols such as the Reverse Address Resolution Protocol (RARP; see Chapter 14), Boot Protocol (BOOTP; see Chapter 60), and Dynamic Host Configuration Protocol (DHCP, discussed in Chapters 61 through 64). These message types are now obsolete.

## **ICMP Message Creation and Processing Conventions and Rules**

In the overview of ICMP earlier in this chapter, I compared the relationship between IP and ICMP to that between an executive and an administrative assistant. One of the characteristics that many executives value in a good assistant is that the assistant does his work independently, without causing unnecessary disruption. A good assistant should save the executive time, not cost her time.

As the assistant to IP, ICMP must similarly help IP function without taking up too much of its resources. Here, the resource being conserved is not so much time as bandwidth. ICMP messages are important, but must be considered part of the overhead of running a network. They carry no user data, so each one represents a small loss of overall end-user bandwidth on the network. For this reason, we want to send them only when necessary, and to carefully control the circumstances under which they are generated.

Administrative assistants have some serious advantages over networking protocols: common sense and experience. They usually know where the line is drawn between help and hindrance; computers don't. To partially compensate, ICMP's operation is guided by a set of *conventions* or *rules* for how messages are created and processed. For ICMPv4, these conventions are described in part in the defining RFC 792, but much more in RFC 1122, "Requirements for Internet Hosts—Communication Layers," which provides specific information on implementing TCP/IP in host devices. In ICMPv6, the information related to ICMP implementation that appears in RFC 1122 has been largely incorporated into the main document that defines ICMPv6, RFC 2463.

Most of the issues related to message generation have to do with error messages, not informational messages. The latter class of messages usually doesn't cause problems because they are generated based on specific rules already established in the protocols that use them. For example, routers send Router Advertisement messages on a regular basis, and the routers make sure this is infrequent. They are also sent in response to Router Solicitation messages sent on occasion by hosts, and as long as a host doesn't go haywire and start sending tons of Solicitations, there won't be a problem. Even then, you can give a router enough smarts not to send Router Advertisements too often.

### ***Limitations on ICMP Message Responses***

The problem comes up with error messages specifically because they are sent *in response* to so many situations. Potentially, they may even be sent in response to each other. Without special care, loops or cascading message generation might occur. For example, consider a situation in which Device A encounters an error and sends an error report to Device B. Device B finds an error in Device A's message and sends an error report back to Device A. This could result in billions of messages being sent back and forth, thereby clogging the network, until a human figures out what is wrong and fixes it.

To prevent such problems, an ICMP error message *must not* be generated in response to any of the following:

**An ICMP Error Message** This prevents loops of the type just mentioned. Note, however, that an ICMP error message *can* be generated in response to an ICMP informational message.

**A Broadcast or Multicast Datagram** What would happen if a datagram were broadcast to 5,000 hosts, and each of them found an error in it and tried to send a report back to the source? Something unpleasant!

**IP Datagram Fragments Except the First** In many cases, the same situation that might cause a device to generate an error for one fragment would also apply to each successive one, causing unnecessary ICMP traffic. For this reason, when a datagram is fragmented, a device may send an error message only in response to a problem in the first fragment.

**Datagrams with Non-Unicast Source Address** If a datagram's source address doesn't define a unique, unicast device address, an error message cannot be sent back to that source. This prevents ICMP messages from being broadcast, unicast, or sent to nonroutable special addresses such as the loopback address.

**KEY CONCEPT** In order to prevent excessive numbers of ICMP messages from being sent on a network, a special set of rules governs when and how they may be created. Most of these are designed to eliminate situations in which very large numbers of ICMP error messages would be generated in response to certain occurrences.

These rules apply to both ICMPv4 and ICMPv6, but in ICMPv6 there are a couple of special cases. In certain circumstances, an ICMPv6 Packet Too Big message may be sent to a multicast address, as this is required for Path MTU Discovery (described in Chapter 27) to work. Certain Parameter Problem messages may also be sent to multicast or broadcast addresses. Finally, in addition to the rules just mentioned, IPv6 implementations are specifically directed to limit the rate at which they send ICMPv6 messages overall.

### ***ICMP Message Processing Conventions***

Message processing generally takes place as described earlier in the section on ICMP general operation, with the ICMP message delivered either to the IP software or other protocol software implementation as required. What is done with the message usually depends on its type. Some messages are destined for only the IP software itself, but many are intended for the higher-layer protocol that generated the datagram that led to the error. In the next section, you will see that ICMP error messages include information that allows the upper-layer protocol to be extracted for the purpose of passing the message to the appropriate software layer.

In IPv6, the class of message (error or informational) can be determined from the Type value. This knowledge can then be used to guide processing of ICMP messages with unknown Type values. The rule is that ICMP error messages with unknown Type values must be passed to the appropriate upper-layer protocol. Informational messages with unknown Type values are discarded without taking action.

In addition to these general rules, there are specific rules put into place to guide the processing of some of the message types. I describe some of these conventions in the chapters that discuss individual ICMP messages.

An important final point is that ICMP messages, especially error messages, are not considered binding on the device that processes them. To stick with the office analogy, they have the equivalent status in an office of only of an FYI memo, not an assignment. It is often the case that a device *should* take action upon processing an

ICMP message, but the device is not required to. The exception, again, is when informational messages are used for specific purposes. For example, most of the messages that come in pairs are designed so that a Request results in the matching Reply and a Solicitation yields an Advertisement.

**KEY CONCEPT** A device receiving an ICMP message is not required to take action unless a protocol using a message type dictates a specific response to a particular message type. In particular, devices are not mandated to perform any specific task when receiving an ICMP error message.

## ICMP Common Message Format and Data Encapsulation

As you have seen so far in this chapter, ICMP is not so much a protocol that performs a specific function as a framework for the exchange of error reports and information. Since each of the message types is used for a different purpose, they differ in the types of information they contain. This means each ICMP message has a slightly different format. At the same time, however, ICMP message types also have a degree of commonality—a portion of each message is common between message types.

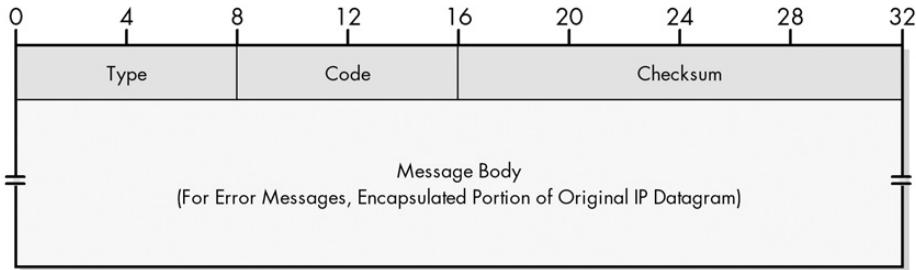
### ICMP Common Message Format

You can think of the structure of an ICMP message as having a *common part* and a *unique part*. The common part consists of three fields that have the same size and same meaning in all ICMP messages (although the values in the fields aren't the same for each ICMP message type). The unique part contains fields that are specific to each type of message.

Interestingly, the common message format is basically the same for ICMPv4 and ICMPv6. It is described in Table 31-3 and illustrated in Figure 31-2.

**Table 31-3:** ICMP Common Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type. For ICMPv6, values from 0 to 127 are error messages, and values 128 to 255 are informational messages. Common values for this field are given in Table 31-2.
Code	1	Identifies the subtype of message within each ICMP message Type value. Thus, up to 256 subtypes can be defined for each message type. Values for this field are shown in the following chapters on individual ICMP message types.
Checksum	2	A 16-bit checksum field that is calculated in a manner similar to the IP header checksum in IPv4. It provides error-detection coverage for the entire ICMP message. Note that in ICMPv6, a pseudo header of IPv6 header fields is prepended for checksum calculation; this is similar to the way this is done in TCP.
Message Body/Data	Variable	Contains the specific fields used to implement each message type. This is the unique part of the message.



**Figure 31-2: ICMP common message format** This overall, generic message format is used for both ICMPv4 and ICMPv6 message types.

### Original Datagram Inclusion in ICMP Error Messages

The message body typically contains one or several fields that carry information of relevance to each specific type of ICMP message. All ICMP error messages include a portion of the original IP datagram that led to the ICMP error message. This aids in diagnosing the problem that caused the ICMP message to be generated, by allowing the error to be communicated to higher layers.

The inclusion of original IP datagram information is done differently for the two ICMP versions:

**ICMPv4 Error Messages** Each error message includes the full IP header and the first 8 bytes of the payload. Since the beginning of the payload will contain the encapsulated higher-layer header, the ICMP message also carries either the full UDP header or the first 8 bytes of the TCP header. In both cases, the source and destination port numbers are part of what is included. If the original header was a standard IP header with no options, the Message Body will therefore have a length of 28 bytes; if options are present, it will be larger.

**ICMPv6 Error Messages** Each error message includes as much of the IPv6 datagram as will fit without causing the size of the ICMPv6 error message (including its IP header encapsulation) to exceed the minimum IPv6 maximum transmission unit size, which is 1280 bytes. This provides additional information for diagnostic purposes when compared to ICMPv4, while ensuring that no ICMPv6 error messages will be too large for any physical network segment. The larger size of the included data allows the IPv6 extension headers to be included in the error message, since the error could be in one of those extension headers.

**NOTE** Remember that in IPv6, routers cannot fragment IP datagrams; any datagram that is oversized for an underlying physical network is dropped. ICMPv6 is thus designed to ensure that this does not happen by not creating ICMPv6 datagrams over the universal IPv6 MTU size of 1280.

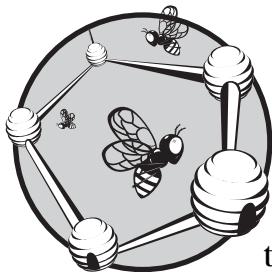
**KEY CONCEPT** Each kind of ICMP message contains data unique to that message type, but all messages are structured according to a common ICMP message format. ICMP error messages always include in their message body field some portion of the original IP datagram that resulted in the error being generated.

## ***ICMP Data Encapsulation***

After an ICMP message is formatted, it is encapsulated in an IP datagram like any other IP message. This is why some people believe ICMP is architecturally a higher layer than IP, though as I discussed earlier, it is really more of a special case. You can also see that when an ICMP error message is generated, we end up with the original IP header and part or all of the payload, encapsulated in the ICMP message, which in turn is encapsulated within a new IP header that will be sent back as an error report, usually to the device that sent the original IP message.

# 32

## **ICMPV4 ERROR MESSAGE TYPES AND FORMATS**



Routers and hosts use Internet Control Message Protocol (ICMP) error messages to tell a device that sent a datagram about problems that were encountered during delivery. The original ICMP version 4 (ICMPv4) defined five different error messages, which are all described in the original ICMP standard, RFC 792. These are some of the most important ICMP messages. They provide critical feedback about error conditions and may help a transmitting device take corrective action to ensure reliable and efficient datagram delivery.

In this first of four chapters on specific ICMP types, I look at the ICMPv4 error messages. I begin with Destination Unreachable messages, which are sent due to datagram delivery failures, and Source Quench messages, which are used to tell a device to slow down the rate at which it sends datagrams. Next, I describe Time Exceeded messages, which are sent when a datagram has been traveling the network too long or takes too long to be reassembled.

from fragments, and Redirect messages, which let a router provide feedback about better routes to a host. Finally, I discuss Parameter Problem messages, which are generic messages used for problems not covered by other ICMP error messages.

## ICMPv4 Destination Unreachable Messages

Since the Internet Protocol (IP) is an unreliable protocol, there are no guarantees that a datagram sent by one device to another will ever actually get there. The internetwork of hosts and routers will make a best effort to deliver the datagram, but it may not get where it needs to for any number of reasons. Devices on an IP network understand that and are designed accordingly. IP software never assumes its datagrams will always be received, and higher-layer protocols like the Transmission Control Protocol (TCP) take care of providing reliability and acknowledgments of received data for applications that need these features.

This setup, with higher layers handling failed deliveries, is sufficient in some cases. For example, suppose Device A tries to send to Device B, but a router near Device B is overloaded, so it drops the datagram. In this case, the problem is likely intermittent, so Device A can retransmit and eventually reach Device B. But what about a situation where a device is trying to send to an IP address that doesn't exist, or a problem with routing that isn't easily corrected? Having the source just continually retry in this case would be inefficient, to say the least.

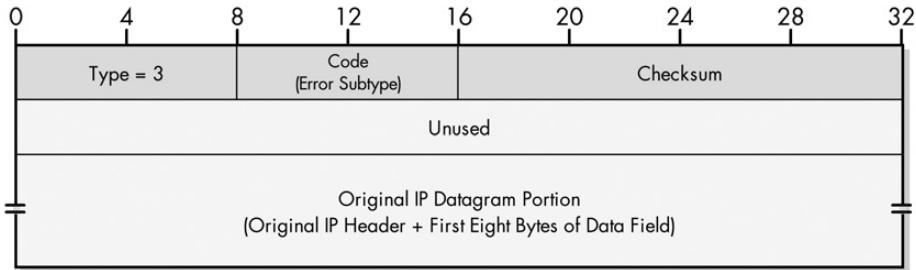
IP is designed to allow IP datagram deliveries to fail, and we should take any such failures seriously. What we really need is a feedback mechanism that can tell a source device that something improper is happening and why. In IP version 4 (IPv4), this service is provided through the transmission of *Destination Unreachable* ICMP messages. When a source node receives one of these messages, it knows there was a problem sending a datagram, and can then decide what action, if any, it wants to take. Like all ICMP error messages, Destination Unreachable messages include a portion of the datagram that could not be delivered, which helps the recipient of the error figure out what the problem is.

### ICMPv4 Destination Unreachable Message Format

Table 32-1 and Figure 32-1 show the specific format for ICMPv4 Destination Unreachable messages.

**Table 32-1: ICMPv4 Destination Unreachable Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type; for Destination Unreachable messages, this is set to 3.
Code	1	Identifies the subtype of unreachable error being communicated. See Table 32-2 for a full list of codes and what they mean.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Unused	4	The 4 bytes that are left blank and not used.
Original Datagram Portion	Variable	The full IP header and the first 8 bytes of the payload of the datagram that prompted this error message to be sent.



**Figure 32-1:** ICMPv4 Destination Unreachable message format

### **ICMPv4 Destination Unreachable Message Subtypes**

There are many different reasons why it may not be possible for a datagram to reach its destination. Some of these may be due to erroneous parameters (like the invalid IP address example mentioned earlier). A router might have a problem reaching a particular network for whatever reason. There can also be other more esoteric reasons related to why a datagram cannot be delivered.

For this reason, the ICMPv4 Destination Unreachable message type can be considered as a class of related error messages. The receipt of a Destination Unreachable message tells a device that the datagram it sent couldn't be delivered, and the Code field in the ICMP header indicates the reason for the nondelivery. Table 32-2 shows the different Code values, corresponding message subtypes, and a brief explanation of each.

**Table 32-2:** ICMPv4 Destination Unreachable Message Subtypes

<b>Code Value</b>	<b>Message Subtype</b>	<b>Description</b>
0	Network Unreachable	The datagram could not be delivered to the network specified in the network ID portion of the IP address. This usually means a problem with routing but could also be caused by a bad address.
1	Host Unreachable	The datagram was delivered to the network specified in the network ID portion of the IP address but could not be sent to the specific host indicated in the address. Again, this usually implies a routing issue.
2	Protocol Unreachable	The protocol specified in the Protocol field was invalid for the host to which the datagram was delivered.
3	Port Unreachable	The destination port specified in the UDP or TCP header was invalid.
4	Fragmentation Needed and DF Set	This is one of those esoteric codes. Normally, an IPv4 router will automatically fragment a datagram that it receives if it is too large for the maximum transmission unit (MTU) of the next physical network link the datagram needs to traverse. However, if the DF (Don't Fragment) flag is set in the IP header, this means the sender of the datagram does not want the datagram ever to be fragmented. This puts the router between the proverbial rock and a hard place, and it will be forced to drop the datagram and send an error message with this code. This message type is most often used in a clever way by intentionally sending messages of increasing size to discover the MTU size that a link can handle. This process is called Path MTU Discovery (described in Chapter 27).

(continued)

**Table 32-2:** ICMPv4 Destination Unreachable Message Subtypes (continued)

<b>Code Value</b>	<b>Message Subtype</b>	<b>Description</b>
5	Source Route Failed	Generated if a source route was specified for the datagram in an option but a router could not forward the datagram to the next step in the route.
6	Destination Network Unknown	Not used; code 0 is used instead.
7	Destination Host Unknown	The host specified is not known. This is usually generated by a router local to the destination host and usually means a bad address.
8	Source Host Isolated	Obsolete, no longer used.
9	Communication with Destination Network Is Administratively Prohibited	The source device is not allowed to send to the network where the destination device is located.
10	Communication with Destination Host Is Administratively Prohibited	The source device is allowed to send to the network where the destination device is located, but not that particular device.
11	Destination Network Unreachable for Type of Service	The network specified in the IP address cannot be reached due to the inability to provide service specified in the Type of Service field of the datagram header (see Chapter 31).
12	Destination Host Unreachable for Type of Service	The destination host specified in the IP address cannot be reached due to the inability to provide service specified in the datagram's Type of Service field.
13	Communication Administratively Prohibited	The datagram could not be forwarded due to filtering that blocks the message based on its contents.
14	Host Precedence Violation	Sent by a first-hop router (the first router to handle a sent datagram) when the Precedence value in the Type of Service field is not permitted.
15	Precedence Cutoff in Effect	Sent by a router when receiving a datagram whose Precedence value (priority) is lower than the minimum allowed for the network at that time.

As you can see in Table 32-2, not all of these codes are actively used at this time. For example, code 8 is obsolete and code 0 is used instead of 6. Also, some of the higher numbers related to the Type of Service field aren't actively used because Type of Service isn't actively used.

**KEY CONCEPT** ICMPv4 *Destination Unreachable* messages are used to inform a sending device of a failure to deliver an IP datagram. The message's Code field provides information about the nature of the delivery problem.

### ***Interpretation of Destination Unreachable Messages***

It's important to remember that just as IP is a best effort, the reporting of unreachable destinations using ICMP is also a best effort. Realize that these ICMP messages are themselves carried in IP datagrams. More than that, however, remember that there may be problems that prevent a router from detecting failure of delivery of an

ICMP message, such as a low-level hardware problem. A router could, theoretically, also be precluded from sending an ICMP message even when failure of delivery *is* detected for whatever reason.

For this reason, the sending of Destination Unreachable messages should be considered supplemental. There is no guarantee that every problem sending a datagram will result in a corresponding ICMP message. No device should count on receiving an ICMP Destination Unreachable for a failed delivery any more than it counts on the delivery in the first place. This is why the higher-layer mechanisms mentioned at the start of this discussion are still important.

## ICMPv4 Source Quench Messages

When a source device sends out a datagram, it will travel across the internetwork and eventually arrive at its intended destination (at least, that's what we hope will happen). At that point, it is up to the destination device to process the datagram by examining it and determining which higher-layer software process to hand the datagram.

If a destination device is receiving datagrams at a relatively slow rate, it may be able to process each datagram on the fly as it is received. However, datagram receipt in a typical internetwork can tend to be uneven or bursty, with alternating higher and lower rates of traffic. To allow for times when datagrams are arriving faster than they can be processed, each device has a *buffer* where it can temporarily hold datagrams it has received until it has a chance to deal with them.

However, this buffer is itself limited in size. Assuming the device has been properly designed, the buffer may be sufficient to smooth out high-traffic and low-traffic periods most of the time. Certain situations can still arise in which traffic is received so rapidly that the buffer fills up entirely. Some examples of scenarios in which this might happen include the following:

- A single destination is overwhelmed by datagrams from many sources, such as a popular website being swamped by HTTP requests.
- Device A and Device B are exchanging information, but Device A is a much faster computer than Device B, and can generate outgoing and process incoming datagrams much faster than Device B can.
- A router receives a large number of datagrams over a high-speed link that it needs to forward over a low-speed link; they start to pile up while waiting to be sent over the slow link.
- A hardware failure or other situation causes datagrams to sit at a device unprocessed.

A device that continues to receive datagrams when it has no more buffer space is forced to discard them and is said to be *congested*. A source that has its datagram discarded due to congestion won't have any way of knowing this, since IP itself is unreliable and unacknowledged. Therefore, while it is possible to simply allow higher-layer protocols to detect the dropped datagrams and generate replacements, it makes a lot more sense to have the congested device provide feedback to the sources by telling them that it is overloaded.

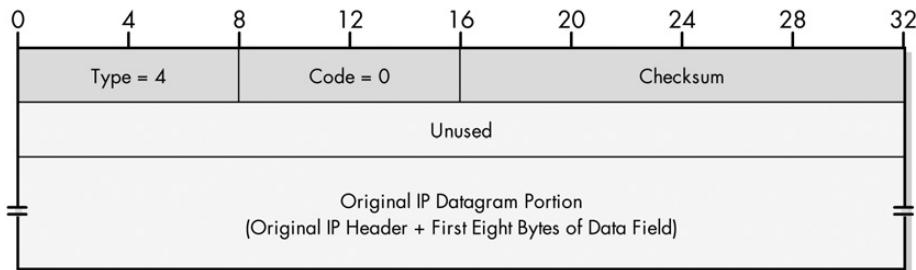
In IPv4, a device that is forced to drop datagrams due to congestion provides feedback to the sources that overwhelmed it by sending them ICMPv4 *Source Quench* messages. Just as you use water to quench a fire, a Source Quench message is a signal that attempts to quench a source device that is sending too fast. In other words, it's a polite way for one IP device to tell another, "Slow down!" When a device receives one of these messages, it knows it needs to reduce the speed at which it is sending datagrams to the device that sent it.

### **ICMPv4 Source Quench Message Format**

Table 32-3 and Figure 32-2 show the specific format for ICMPv4 Source Quench messages.

**Table 32-3:** ICMPv4 Source Quench Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type; for Source Quench messages, this is set to 4.
Code	1	Identifies the subtype of error being communicated. For Source Quench messages, this is not used, and the field is set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Unused	4	The 4 bytes that are left blank and not used.
Original Datagram Portion	Variable	The full IP header and the first 8 bytes of the payload of the datagram that was dropped due to congestion.



**Figure 32-2:** ICMPv4 Source Quench message format

### **Problems with Source Quench Messages**

What's interesting about the Source Quench format is that it is basically a null message. It tells the source that the destination is congested but provides no specific information about that situation, nor does it specify what exactly the destination wants the source to do other than cut back on its transmission rate in some way. There is also no method for the destination to signal a source that it is no longer congested, and that the source should resume its prior sending rate. This means the response to a Source Quench message is left up to the device that receives it. Usually, a device will cut back its transmission rate until it no longer receives the messages, and then it may try to slowly increase the rate again.

In a similar manner, there are no rules about when and how a device generates Source Quench messages in the first place. A common convention is that one message is generated for each dropped datagram. However, more intelligent algorithms may be employed, especially on higher-end routers, to predict when the device's buffer will be filled and preemptively quench certain sources that are sending too quickly. Devices may also decide whether to quench all sources when they become busy, or only certain ones. As with other ICMP error messages, a device cannot count on a Source Quench message being sent when a busy device discards one of its datagrams.

The lack of information communicated in Source Quench messages makes them a rather crude tool for managing congestion. In general terms, the process of regulating the sending of messages between two devices is called *flow control*, and this is usually a function of the transport layer. TCP actually has a flow control mechanism (discussed in Chapter 49) that is far superior to the use of ICMP Source Quench messages.

Another issue with Source Quench messages is that they can be abused. Transmission of these messages by a malicious user can cause a host to be slowed down when there is no valid reason. This security issue, combined with the superiority of the TCP method for flow control, has caused the use of Source Quench messages to largely fall out of favor.

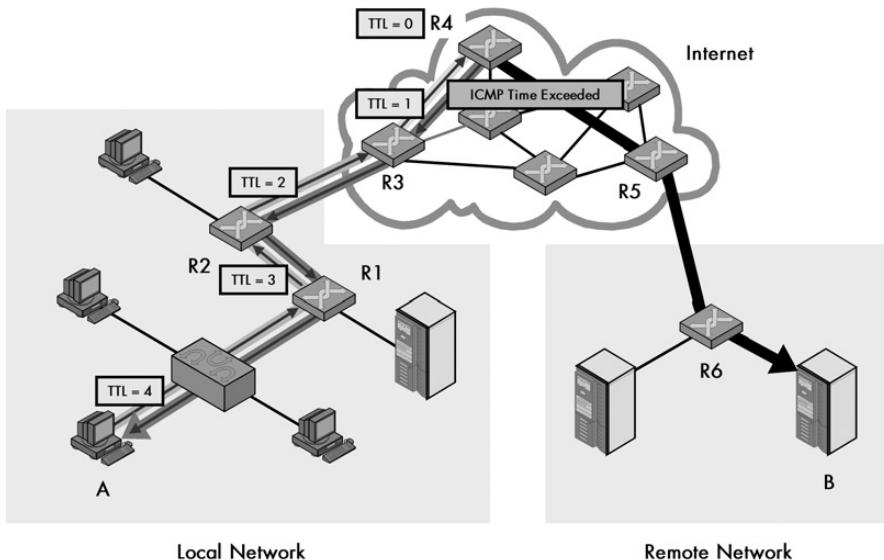
**KEY CONCEPT** ICMPv4 *Source Quench* messages are sent by a device to request that another reduce the rate at which it is sending datagrams. The messages are a rather crude method of flow control compared to more capable mechanisms such as those provided by TCP.

## ICMPv4 Time Exceeded Messages

Large IP internetworks can have thousands of interconnected routers that pass datagrams between devices on various networks. In large internetworks, the topology of connections between routes can become complex, which makes routing more difficult. Routing protocols will normally allow routers to find the best routes between networks, but in some situations, an inefficient route might be selected for a datagram. In the worst case, a *router loop* may occur. An example of this situation is where Router A thinks datagrams intended for Network X should next go to Router B, which thinks they should go to Router C, which thinks they need to go to Router A. (See the ICMPv6 Time Exceeded Message description in Chapter 34 for an illustration of a router loop.)

If a loop like this occurred, datagrams for Network X that were entering this part of the internetwork would circle forever, chewing up bandwidth and eventually leading to the network being unusable. As insurance against this occurrence, each IP datagram includes in its header a Time to Live (TTL) field. This field was originally intended to limit the maximum time (in seconds) that a datagram could be on the internetwork, but now limits the life of a datagram by limiting the number of times the datagram can be passed from one device to the next. The TTL is set to a value by the source that represents the maximum number of hops it wants for the datagram. Each router decrements the value; if it ever reaches zero, the datagram is said to have *expired* and is discarded.

When a datagram is dropped due to expiration of the TTL field, the device that dropped the datagram will inform the source of this occurrence by sending it an ICMPv4 *Time Exceeded* message, as shown in Figure 32-3. Receipt of this message indicates to the original sending device that there is a routing problem when sending to that particular destination, or that it set the TTL field value too low in the first place. As with all ICMP messages, the device receiving it must decide whether and how to respond to receipt of the message. For example, it may first try to resend the datagram with a higher TTL value.



**Figure 32-3: Expiration of an IP datagram and Time Exceeded message generation** In this example, Device A sends an IP datagram to Device B, which has a Time to Live (TTL) field value of only 4 (perhaps not realizing that Device B is seven hops away). On the fourth, hop the datagram reaches Router R4, which decrements its TTL field to 0 and then drops it as it expires. Router R4 then sends an ICMP Time Exceeded message back to Device A.

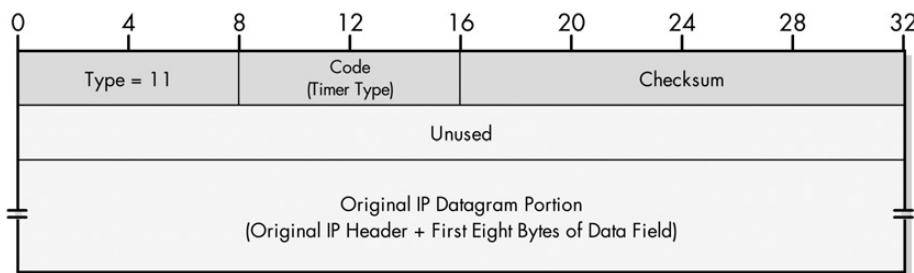
There is another time expiration situation where ICMP Time Exceeded messages are used. When an IP message is broken into fragments, the destination device is charged with reassembling them into the original message. One or more fragments may not make it to the destination, so to prevent the device from waiting forever, it sets a timer when the first fragment arrives. If this timer expires before the others are received, the device gives up on this message. The fragments are discarded, and a Time Exceeded message is generated.

### ICMPv4 Time Exceeded Message Format

Table 32-4 and Figure 32-4 show the specific format for ICMPv4 Time Exceeded messages.

**Table 32-4: ICMPv4 Time Exceeded Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type; for Time Exceeded messages, this is set to 11.
Code	1	Identifies the subtype of error being communicated. A value of 0 indicates expiration of the IP TTL field; a value of 1 indicates that the fragment reassembly time has been exceeded.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Unused	4	The 4 bytes that are left blank and not used.
Original Datagram Portion	Variable	The full IP header and the first 8 bytes of the payload of the datagram that was dropped due to expiration of the TTL field or reassembly timer.



**Figure 32-4: ICMPv4 Time Exceeded message format**

### **Applications of Time Exceeded Messages**

ICMP Time Exceeded messages are usually sent in response to the two conditions described in Table 32-4: TTL or reassembly timer expiration. Generally, routers generate TTL expiration messages as they try to route a datagram, while end hosts indicate reassembly violations. However, there is actually a very clever application of these messages that has nothing to do with reporting errors at all.

The TCP/IP *traceroute* (or *tracert*) utility is used to show the sequence of devices over which a datagram is passed on a particular route between a source and destination. The traceroute utility also shows the amount of time it takes for a datagram to reach each hop in that route. This utility was originally implemented using Time Exceeded messages by sending datagrams with successively higher TTL values.

First, a dummy datagram is sent with a TTL value of 1, causing the first hop in the route to discard the datagram and send back an ICMP Time Exceeded message; the time elapsed for this could then be measured. Then, a second datagram is sent with a TTL value of 2. This causes the second device in the route to report back a Time Exceeded message, and so on. By continuing to increase the TTL value you can get reports back from each hop in the route. See Chapter 88 for more details on traceroute's operation.

**KEY CONCEPT** ICMPv4 *Time Exceeded* messages are sent in two different time-related circumstances. The first is if a datagram's Time to Live (TTL) field is reduced to zero, causing it to expire and the datagram to be dropped. The second is when all the pieces of a fragmented message are not received before the expiration of the recipient's reassembly timer.

## ICMPv4 Redirect Messages

Every device on an internetwork needs to be able to send to every other device. If hosts were responsible for determining the routes to each possible destination, each host would need to maintain an extensive set of routing information. Since there are so many hosts on an internetwork, this would be a very time-consuming and maintenance-intensive situation.

Instead, IP internetworks are designed around a fundamental design decision: Routers are responsible for determining routes and maintaining routing information. Hosts determine only when they need a datagram routed, and then hand the datagram off to a local router to be sent where it needs to go. I discuss this in more detail in my overview of IP routing concepts (see Chapter 23).

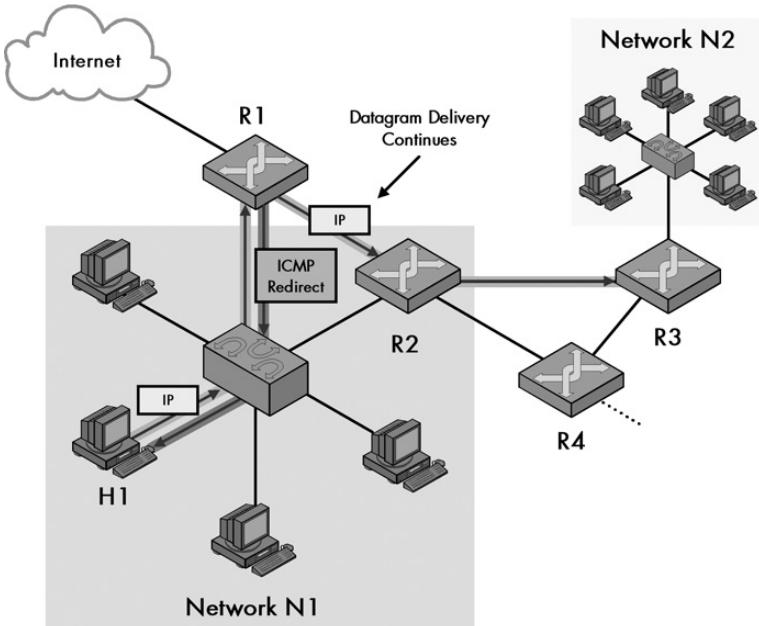
Since most hosts do not maintain routing information, they must rely on routers to know about routes and where to send datagrams intended for different destinations. Typically, a host on an IP network will start out with a routing table that basically tells it to send everything not on the local network to a single *default router*, which will then figure out what to do with it. Obviously, if there is only one router on the network, the host will use that as the default router for all nonlocal traffic. However, if there are two or more routers, sending all datagrams to just one router may not make sense. It is possible that a host could be manually configured to know which router to use for which destinations, but another mechanism in IP can allow a host to learn this automatically.

Consider a Network N1 that contains a number of hosts (H1, H2, and so on) and two routers, R1 and R2. Host H1 has been configured to send all datagrams to Router R1, as its default router. Suppose it wants to send a datagram to a device on Network N2. However, Network N2 is most directly connected to Network N1 using Router R2, not R1. The datagram will first be sent to Router R1, which will look in its routing table and see that datagrams for Network N2 need to be sent through Router R2. “But wait,” R1 says. “R2 is on the local network, and H1 is on the local network—so why am I needed as a middleman? H1 should just send datagrams for N2 directly to R2 and leave me out of it.”

In this situation, Router R1 will send an ICMPv4 *Redirect* message back to Host H1, telling it that in the future, it should send this type of datagram directly to Router R2. This situation is shown in Figure 32-5. Router R1 will also forward the datagram to Router R2 for delivery, since there is no reason to drop the datagram. Thus, despite usually being grouped along with true ICMP error messages, Redirect messages are really arguably not error messages at all. They represent a situation in which only inefficiency exists, not outright error. (In fact, in ICMPv6, they have been reclassified.)

### ICMPv4 Redirect Message Format

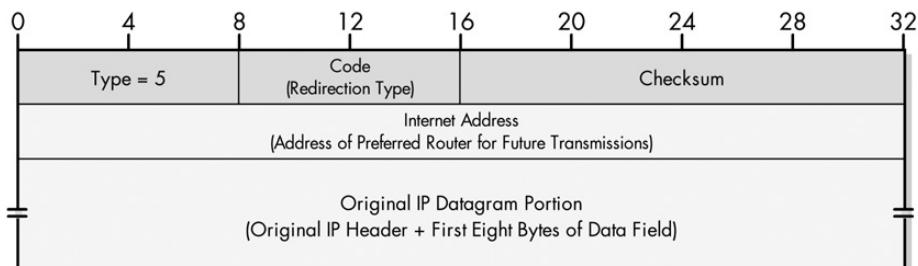
Table 32-5 and Figure 32-6 show the specific format for ICMPv4 Redirect messages.



**Figure 32-5: Host redirection using an ICMP Redirect message** In this example, Host H1 sends to Router R1 a datagram destined for Network N2. However, Router R1 notices that Router R2 is on the same network and is a more direct route to Network N2. It forwards the datagram on to Router R2, but also sends an ICMP Redirect message back to Host H1 to tell it to use Router R2 next time.

**Table 32-5: ICMPv4 Redirect Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type; for Redirect messages, this value is 5.
Code	1	Identifies the meaning or scope of the Redirect message. See Table 32-6 for an explanation of how this field is used in Redirect messages.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Internet Address	4	The address of the router to which future datagrams sent to the original destination should be sent.
Original Datagram Portion	Variable	The full IP header and the first 8 bytes of the payload of the datagram that led to the creation of the Redirect.



**Figure 32-6: ICMPv4 Redirect message format**

## **Redirect Message Interpretation Codes**

When a Redirect message is received back by a device, it inspects the included portion of the original datagram. Since this contains the original destination address of the redirected target device, this tells the original sender which addresses should be redirected in the future. The Internet Address field tells it which router it should use for subsequent datagrams. The Code field tells the sender how broadly to interpret the redirection. There are four different Code values, as shown in Table 32-6.

**Table 32-6:** ICMP Redirect Message Interpretation Codes

<b>Code Value</b>	<b>Message Subtype</b>	<b>Meaning</b>
0	Redirect Datagrams for the Network (or Subnet)	Redirect all future datagrams sent not only to the device whose address caused this Redirect, but also to all other devices on the network (or subnet) where that device is located. (This code is now obsolete; see the note that follows this table.)
1	Redirect Datagrams for the Host	Redirect all future datagrams only for the address of the specific device to which the original datagram was sent.
2	Redirect Datagrams for the Type of Service (TOS) and Network (or Subnet)	Same as for Code value 0, but only for future datagrams that have the same TOS value as the original datagram. (This code is now obsolete; see the note that follows this table.)
3	Redirect Datagrams for the TOS and Host	As for Code value 1, but only for future datagrams that have the same TOS value as the original datagram.

**NOTE** One problem with Redirects for whole networks is that the network specification may be ambiguous in an environment where subnetting or classless addressing is used. For this reason, the use of Code values 0 and 2 was prohibited by RFC 1812; the values are considered obsolete on the modern Internet.

Obviously, routers usually generate Redirect messages and send them to hosts; hosts do not normally create them. The specific rules for when Redirect messages are created can be fairly complex, as a number of conditions may exist that preclude these messages from being sent. In particular, special rules exist for when a router may redirect an entire network (or subnet) instead of just a single host. Also, remember that the TOS field is optional and often not used, so Redirects with Code values of 2 or 3 are less common than those with values of 0 and 1.

## **Limitations of Redirect Messages**

Keep in mind that ICMP Redirect messages are *not* a mechanism by which the general routing process in IP is implemented; they are only a support function. They are a convenient way for hosts to be given information about routes by local routers, but are not used to communicate route information between routers.

This means that a Redirect message can tell a host to use a more efficient first-hop router, but cannot tell a router to use a more efficient second-hop router. In the previous example (illustrated in Figure 32-5), suppose that in addition to the connections mentioned, Router R2 is connected to Router R3 and Router R4. Router R2 sends the datagram in question to Router R3, which realizes it needs to

send to Router R4, a router already directly connected to Router R2. Router R3 *cannot* send a Redirect message to Router R2 telling it to use Router R4 next time. The messages are simply not designed for this purpose—remember that ICMP messages always go back to the source of the original datagram, which would not be Router R2 in this case. Such inefficiencies must be resolved using routing protocols.

**KEY CONCEPT** A router uses ICMPv4 *Redirect* messages to inform a host of a preferred router that will be used for future datagrams that are sent to a particular host or network. They are not used to alter routes between routers.

## ICMPv4 Parameter Problem Messages

The previous sections in this chapter describe four specific ICMPv4 message types that allow a device to report various error conditions to the original sender of a datagram. However, other error situations may arise that don't correspond to any of these four specific message types. Typically, the problem results when a device attempts to process the header fields of an IP datagram and finds something in it that doesn't make sense.

If a device finds a problem with any of the parameters in an IP datagram header that is serious enough that it cannot complete processing the header, it must discard the datagram. As in other cases where a datagram must be tossed out, this is serious enough to warrant communication of the problem back to the device that sent the original datagram. This is accomplished in ICMPv4 using the *Parameter Problem* message type.

This is a catchall type of message that can be used to indicate an error in any header field of an IP datagram. The message type does not contain any specific fields or coding to indicate what the problem is. This was done intentionally to keep the Parameter Problem message generic and ensure that it could indicate any sort of error. Instead of special error codes, most Parameter Problem messages tell the original source which parameter caused the problem by including a special pointer that indicates which field in the original datagram header caused the problem. Both hosts and routers can generate Parameter Problem messages.

### ICMPv4 Parameter Problem Message Format

Table 32-7 and Figure 32-7 show the specific format for ICMPv4 Parameter Problem messages.

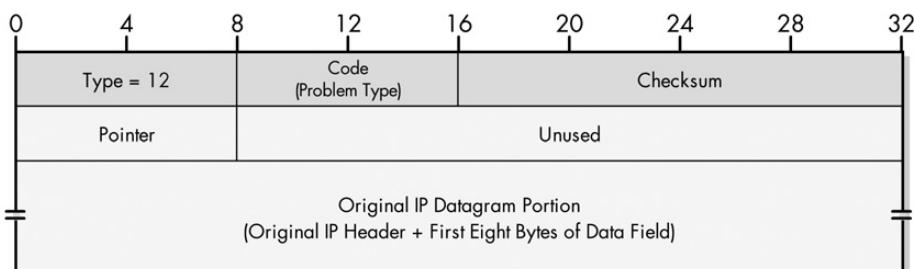


Figure 32-7: ICMPv4 Parameter Problem message format

**Table 32-7:** ICMPv4 Parameter Problem Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type; for Parameter Problem messages, this value is 12.
Code	1	Identifies the subtype of the problem being communicated. See Table 32-8 for more information about this field as it relates to Parameter Problem messages.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Pointer	1	An offset that points to the byte location in the datagram that caused the Parameter Problem message to be generated. The device receiving the ICMP message can use this value to get an idea of which field in the original message had the problem. This field is used only when the Code value is 0.
Unused	3	3 bytes that are left blank and not used.
Original Datagram Portion	Variable	The full IP header and the first 8 bytes of the payload of the datagram that prompted this error message to be sent.

### **Parameter Problem Message Interpretation Codes and the Pointer Field**

When a Parameter Problem message is generated due to a specific bad field in the original message, the Pointer field is used to show the location of the problem. This meaning of the Parameter Problem message is the one that was defined in the original ICMP standard, RFC 792, and is associated with Code value 0. There are some cases of a parameter problem in which a pointer to a specific field in the original message really wouldn't make sense, so other standards have defined two new Code field values for Parameter Problem messages. Table 32-8 shows the three Code values and provides a brief explanation of each one.

**Table 32-8:** ICMPv4 Parameter Problem Message Interpretation Codes

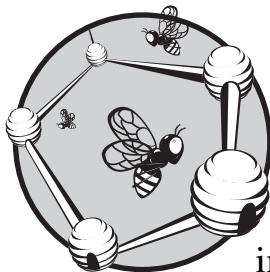
Code Value	Message Subtype	Description
0	Pointer Indicates the Error	This is the normal use of the Parameter Problem message. When this Code value is used, the Pointer field indicates the location of the problem.
1	Missing a Required Option	The IP datagram needed to have an option in it that was missing. Since the option was missing, there is no way to point to it.
2	Bad Length	The length of the datagram overall was incorrect, indicating a general problem with the message as a whole. Again, the Pointer field makes no sense here.

**KEY CONCEPT** The ICMPv4 *Parameter Problem* message is a generic catchall that can be used to convey an error of any type in an IP datagram. A special Pointer field is normally used to indicate to the message's recipient where the problem was in the original datagram.

Note that the Pointer field is only eight bits wide, but since this allows for values of up to 256, it is sufficient for allowing it to point to any location within the IP header. It is possible for the Pointer field to point to a field within an IP option.

# 33

## ICMPV4 INFORMATIONAL MESSAGE TYPES AND FORMATS



The five Internet Control Message Protocol (ICMP) error message types we examined in the previous chapter communicate important information about error or problem conditions encountered during the operation of an Internet Protocol (IP) internetwork. In contrast, the other class of ICMP messages contains those messages that are *informational*. They are not sent in response to some issue with a regular IP datagram, but are used on their own to implement various support functions for IP. Informational messages are used for testing and diagnostic purposes, as well as for allowing devices to share critical information that they need to function correctly.

In this chapter, I describe nine different ICMP version 4 (ICMPv4) informational messages. Because many of these messages are used in functional sets, pairs of related messages are described together. I begin with a discussion of the Echo (Request) and Echo Reply messages used for network testing, and Timestamp (Request) and Timestamp Reply messages used for clock synchronization. I explain the use and format of Router Advertisement and Router Solicitation messages, which allow hosts to discover the identity of

local routers and learn important information about them. I also describe the Address Mask Request and Address Mask Reply messages that communicate subnet mask information. I conclude with a look at the Traceroute message, which implements a more sophisticated version of the traceroute utility.

**NOTE** *The original ICMP standard also defined two more informational message types: Information Request and Information Reply. These were intended to allow devices to determine an IP address and possibly other configuration information. This function was later implemented using host configuration protocols such as the Reverse Address Resolution Protocol (RARP), Boot Protocol (BOOTP), and Dynamic Host Configuration Protocol (DHCP). These message types are now obsolete; therefore, they are not discussed in this chapter.*

## ICMPv4 Echo (Request) and Echo Reply Messages

One of the main purposes of ICMP informational messages is to enable testing and diagnostics in order to help identify and correct problems on an internetwork. The most basic test that can be conducted between two devices is simply checking if they are capable of sending datagrams to each other. The usual way that this is done is to have one device send a test message to a second device, which receives the message and replies back to tell the first device it received the message.

ICMPv4 includes a pair of messages specifically for connection testing. Suppose Device A wants to see if it can reach Device B. Device A begins the test process by sending an ICMPv4 Echo message to Device B. Device B, when it receives the Echo, responds back to Device A with an Echo Reply message. When Device A receives this message, it knows that it is able to communicate (both send and receive) successfully with Device B.

**NOTE** *The name of the first message in this pair is often given as Echo Request. While this does convey the paired nature of the Echo and Echo Reply messages, the formal name used in the standards is simply an Echo message.*

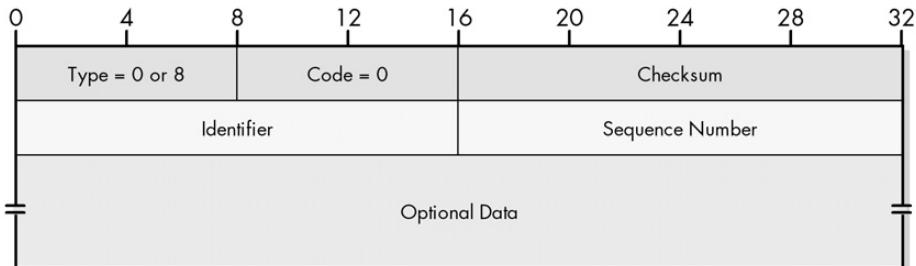
### ICMPv4 Echo and Echo Reply Message Format

Table 33-1 and Figure 33-1 show the format for both ICMPv4 Echo and Echo Reply messages.

**Table 33-1:** ICMPv4 Echo and Echo Reply Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type. For Echo messages, the value is 8; for Echo Reply messages, the value is 0.
Code	1	Not used for Echo and Echo Reply messages; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Identifier	2	An identification field that can be used to help in matching Echo and Echo Reply messages.
Sequence Number	2	A sequence number to help in matching Echo and Echo Reply messages.
Optional Data	Variable	Additional data to be sent along with the message (not specified).

It is possible that a source device may want to send more than one Echo message to either a single destination or multiple destinations. Conversely, a single destination might receive Echo messages from more than one source. It is essential that a device receiving an Echo Reply message knows which Echo message prompted it to be sent.



**Figure 33-1: ICMPv4 Echo and Echo Reply message format** Two special fields are used within the format of these messages. They allow devices to match Echo and Echo Reply messages together, and exchange a sequence of messages. The Identifier field was envisioned as being used as a higher-level label, like a session identifier, while the Sequence Number was seen as something to identify individual test messages within a series. However, the use of these fields is up to the particular implementation. In some cases, the Identifier field is filled in with the process number of the application that is using the Echo or Echo Reply message to allow several users to use utilities like ping without interference.

### Application of Echo and Echo Reply Messages

The most common way that you may use the Echo and Echo Reply messages is through the popular utility *ping*, which is used to test host reachability. While the basic test simply consists of sending an Echo message and waiting for an Echo Reply message, modern versions of ping are quite sophisticated. They allow the user to specify many parameters, including the number of Echo messages sent, how often they are sent, the size of message transmitted, and more. They also provide a great deal of information about the connection, including the number of Echo Reply messages received, the time elapsed for the pair of messages to be exchanged, and a lot more. See the description of ping in Chapter 88 for a full explanation of the utility.

**KEY CONCEPT** ICMPv4 Echo (Request) and Echo Reply messages are used to facilitate network reachability testing. A device can test its ability to perform basic communication with another one by sending an Echo message and waiting for an Echo Reply message to be returned by the other device. The ping utility, a widely used diagnostic tool in TCP/IP internetworks, makes use of these messages.

### ICMPv4 Timestamp (Request) and Timestamp Reply Messages

All of the hosts and routers on an internetwork operate independently of each other. One aspect of this autonomy is that each device maintains a separate system clock. There's a problem, however: Even highly accurate clocks have slight

differences in both how accurately they keep time and the time with which they were initialized at startup. This means that under normal circumstances, no two devices on an internetwork are guaranteed to have exactly the same time.

The creators of TCP/IP recognized that certain applications might not work properly if there were too much differential between the system clocks of a pair of devices. To support this requirement, they created a pair of ICMP messages that allow devices to exchange system time information. The initiating device creates a Timestamp message and sends it to the device with which it wishes to synchronize. That device responds with a Timestamp Reply message. Timestamp fields in these messages are used to mark the times that these messages are sent and received to allow the devices' clocks to be synchronized.

**NOTE** As with the Echo message (described in the previous section), the Timestamp message is sometimes seen as *Timestamp Request*, though the word *Request* doesn't appear in its formal name.

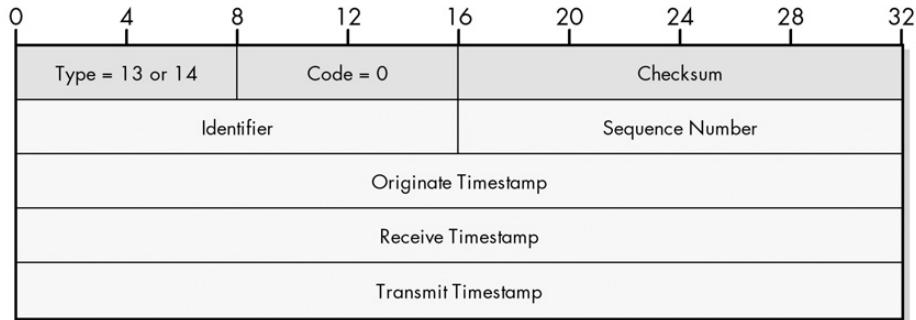
### **ICMPv4 Timestamp and Timestamp Reply Message Format**

The ICMPv4 *Timestamp* and *Timestamp Reply* messages have the same format. The originating device fills in some of the fields, and the replying device fills in others. The format is as shown in Table 33-2 and Figure 33-2.

**Table 33-2:** ICMPv4 Timestamp and Timestamp Reply Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type. For Timestamp messages, the value is 13; for Timestamp Reply messages, the value is 14.
Code	1	Not used for Timestamp and Timestamp Reply messages; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Identifier	2	An identification field that can be used to help in matching Timestamp and Timestamp Reply messages.
Sequence Number	2	A sequence number to help in matching Timestamp and Timestamp Reply messages.
Originate Timestamp	4	A time value filled in by the originating device just before sending the Timestamp message.
Receive Timestamp	4	A time value filled in by the responding device just as it receives the Timestamp message.
Transmit Timestamp	4	A time value filled in by the responding device just before sending back the Timestamp Reply message.

The Identifier and Sequence Number fields are used to match Timestamp and Timestamp Reply messages, exactly as they are used for Echo and Echo Reply messages. The Identifier field is intended as a higher-level label, like a session identifier, while the Sequence Number is often used to identify individual messages within a series. However, the use of these fields is up to the particular implementation.



**Figure 33-2:** ICMPv4 Timestamp and Timestamp Reply message format

All three timestamps are represented as the number of milliseconds since midnight, *Universal Time* (*UT*, also called *Greenwich mean time* or *GMT*). The reason there are three timestamps instead of the two you might ordinarily expect is that the responding device records a separate timestamp when it receives the Timestamp message and when it generates the Timestamp Reply. When the Reply message is received back by the originating device, it then has the times that both the Timestamp and the Timestamp Reply messages were sent. This allows the originating device to differentiate between the time required for transmitting datagrams over the network and the time for the other device to process the Timestamp message and turn it into a Timestamp Reply message.

### **Issues Using Timestamp and Timestamp Reply Messages**

In practice, even with these three timestamp fields, it is difficult to coordinate system clocks over an internetwork, especially a large one like the Internet. The main problem is that the amount of time it takes to send a datagram between any pair of devices varies from one datagram to the next. And again, since IP is unreliable, it's possible that the time for a datagram to be received could be infinite. In fact, it might be lost or dropped by a router.

This means that a simple exchange of Timestamp and Timestamp Reply messages is simply not a method that's reliable enough to ensure that two devices are synchronized on a typical IP internetwork. For this reason, modern devices often use a more sophisticated method for time synchronization, such as the Network Time Protocol (NTP).

Note that unlike many of the other ICMP message types, support for Timestamp and Timestamp Reply messages is optional, for both hosts and routers.

## **ICMPv4 Router Advertisement and Router Solicitation Messages**

In Chapter 23, which described IP routing fundamentals, I discussed a critical aspect of IP internetwork design: the difference between the roles of a router and the roles of a host with regard to routing. Routers are charged with the job of

routing datagrams, and therefore, of knowing routes and exchanging route information. Hosts generally do not know a great deal about routes; they rely on routers to convey datagrams intended for destinations outside the local network.

This dependence means that before a host can really participate on an internetwork, it needs to know the identity of at least one router on the local network. One way to ensure that this is the case is to just manually configure each host with the address of a local router as its default router. This method is simple, but has the typical drawbacks associated with manual processes: It is time-consuming to set up, difficult to maintain, and inflexible.

### ***The Router Discovery Process***

It would be better if there were some method whereby a host could automatically discover the identity of local routers and learn important information about them. In IP, this process is called *Router Discovery* and was first defined in RFC 1256, “ICMP Router Discovery Messages.” The messages referenced in the RFC title are the ICMP Router Advertisement message and the Router Solicitation message. They were added to the ICMP message types that were defined in earlier standards such as RFC 792.

Routers are responsible for sending *Router Advertisement* messages. These messages tell listening devices that the router exists, and they provide important information about the router such as its address (or addresses, if it has more than one) and how long the host should retain information about the router. Routine Router Advertisement messages are sent on a regular basis, and an administrator can configure the time between messages (usually between seven and ten minutes). Hosts listen for these messages; when an advertisement is received, the host processes it and adds the information about the router to its routing table.

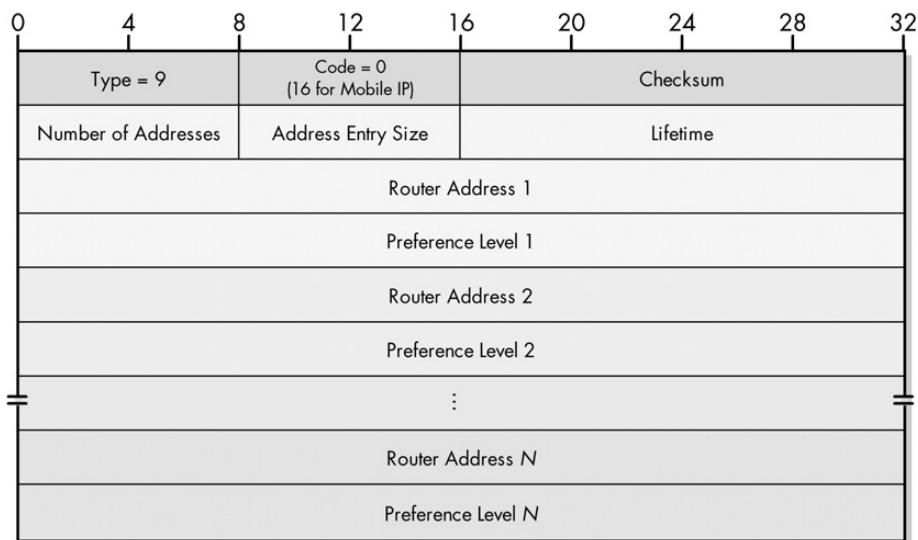
A host that does not have any manually configured routing information will have no knowledge of routers when it first powers on. Having it sit for many minutes while it looks for a routine Router Advertisement message is inefficient. Instead of waiting, the host may send a *Router Solicitation* message on its local network(s). This will prompt any router that hears it to immediately send out an extra Router Advertisement message directly to that host.

### ***ICMPv4 Router Advertisement Message Format***

The ICMPv4 Router Advertisement message format is shown in Table 33-3 and Figure 33-3.

**Table 33-3: ICMPv4 Router Advertisement Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type. For Router Advertisement messages, the value is 9.
Code	1	Normally set to 0. When a Mobile IP agent is sending a Router Advertisement with an Agent Advertisement extension, it may set the value to 16 only if the device is a mobile agent and doesn't intend to handle normal traffic. See the discussion of Mobile IP agent discovery for details (Chapter 30).
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Num Addrs	1	The number of addresses associated with this router that are included in this advertisement.
Addr Entry Size	1	The address entry size—number of 32-bit words of information included with each address. Since in this message format each router address has a 32-bit address and a 32-bit preference level, this value is fixed at 2.
Lifetime	2	The number of seconds that a host should consider the information in this message valid.
Router Address Entries	Value of Num Addrs field * 8	A number of router address entries equal to the value of the Num Addrs field. Each is 8 bytes and has two subfields, each 4 bytes in size. The Router Address subfield is a valid address for an interface to the router sending this message. The Preference Level subfield is the preference level of this address. When more than one address is included in an advertisement, this field indicates which address the router would prefer hosts to use. Higher values mean greater preference.



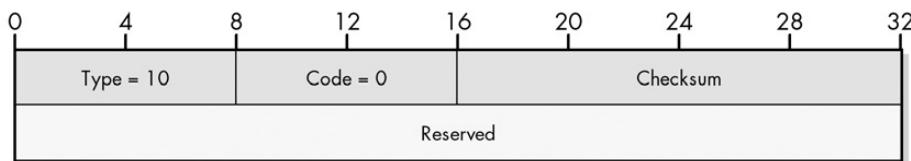
**Figure 33-3: ICMPv4 Router Advertisement Message format**

## **ICMPv4 Router Solicitation Message Format**

ICMPv4 Router Solicitation messages are much simpler, because they need to convey only the following single piece of information: “If you are a router and can hear this, please send a Router Advertisement to me.” The format is therefore just the trivial set of fields shown in Table 33-4 and illustrated in Figure 33-4.

**Table 33-4:** ICMPv4 Router Solicitation Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type. For Router Solicitation messages, the value is 10.
Code	1	Not used; value set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Reserved	4	The 4 reserved bytes sent as 0.



**Figure 33-4:** ICMPv4 Router Solicitation Message format

## **Addressing and Use of Router Advertisement and Router Solicitation Messages**

If possible, both Router Advertisement and Router Solicitation messages are sent out as multicast for efficiency. Router Advertisements use the “all devices” multicast address (224.0.0.1), because they are intended for hosts to hear. *Router Solicitation* messages use the “all routers” multicast address (224.0.0.2). If the local network does not support multicast, messages are instead sent out by broadcast (to address 255.255.255.255).

It is important to remember that just like ICMP Redirect messages, Router Advertisement messages are not a generalized method for exchanging routing information. They are a support mechanism only, used to inform hosts about the existence of routers. Detailed information about routes is communicated between routers using routing protocols, like the Routing Information Protocol (RIP) and Open Shortest Path First (OSPF).

Although Router Discovery is one alternative to manual configuration of a host’s default router, there are other alternatives as well. For example, a host configuration protocol like the Dynamic Host Configuration Protocol (DHCP) can allow a host to learn the address of a default router on the local network.

Finally, note that when Mobile IP is implemented, Router Advertisement messages are used as the basis for Mobile IP-aware routers to send Agent Advertisements. One or more special extensions are added to the regular Router Advertisement format to create an Agent Advertisement. This is discussed extensively in the section on Mobile IP Agent Discovery in Chapter 31.

**KEY CONCEPT** ICMP *Router Advertisement* messages are sent regularly by IP routers to inform hosts of their presence and characteristics. This way, hosts know to use them for delivery of datagrams to distant hosts. A host that is new to a network and wants to find out immediately what routers are present may send a *Router Solicitation* message, which will prompt listening routers to send out Router Advertisement messages.

## ICMPv4 Address Mask Request and Reply Messages

When IP was first developed, IP addresses were based on a simple two-level structure, with a network identifier (network ID) and host identifier (host ID). To provide more flexibility, a technique called *subnetting* was soon developed. Subnetting expands the addressing scheme into a three-level structure, with each address containing a network ID, subnet identifier, and host ID. The *subnet mask* is a 32-bit number that tells devices (and users) which bits are part of the subnet identifier, as compared to the host ID. All of this is described in considerable detail in the part on IP addressing (Part II-3).

To function properly in a subnetting environment, each host must know the subnet mask that corresponds to each address it is assigned. Without the mask, it cannot properly interpret IP addresses. Just as in determining the identity of a local router, a host can be informed of the local network's subnet mask either manually or automatically. The manual method is to simply manually assign the subnet mask to each host. The automatic method makes use of a pair of ICMP messages designed for subnet mask determination, which were defined in RFC 950, the same standard that defined subnetting itself.

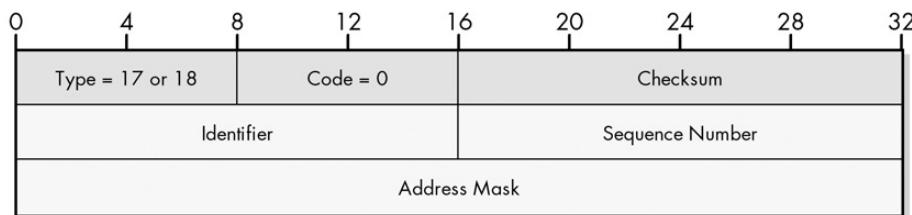
To use this method, a host sends an *Address Mask Request* message on the local network, usually to get a response from a router. If it knows the address of a local router, it may send the request directly (unicast); otherwise, the host will broadcast the request to any listening router. A local router (or other device) will receive this message and respond back with an *Address Mask Reply* message that contains the subnet mask for the local network. This process is somewhat similar to the mechanism used by a host to solicit a router to respond with a *Router Advertisement* message, except that routers do not routinely send subnet mask information—that information must be requested.

### ICMPv4 Address Mask Request and Address Mask Reply Message Format

The Address Mask Request and Address Mask Reply, like some other request and reply pairs, have the same basic format. The host creates the request with all fields filled in except for the subnet mask value itself, and the router supplies the mask and sends the reply back to the host. The format is described in Table 33-5 and illustrated in Figure 33-5.

**Table 33-5: ICMPv4 Address Mask Request and Address Mask Reply Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type. For Address Mask Request messages, the value is 17; for Address Mask Reply messages, it is 18.
Code	1	Not used for either message type; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Identifier	2	An identification field that can be used to help in matching Address Mask Request and Address Mask Reply messages.
Sequence Number	2	A sequence number to help in matching Address Mask Request and Address Mask Reply messages.
Address Mask	4	The subnet mask for the local network, filled in by the router in the Address Mask Reply message.



**Figure 33-5: ICMPv4 Address Mask Request and Address Mask Reply message format**

The Identifier and Sequence Number fields can be used to match up requests and replies, as they are for Echo and Echo Reply messages. However, a host won't normally send multiple requests for subnet masks the way it might send Echo messages for testing. For this reason, the Identifier and Sequence Number fields may be ignored by some implementations.

### **Use of Address Mask Request and Address Mask Reply Messages**

Note that the use of Address Mask Request and Address Mask Reply messages is optional, just as the Router Discovery described in the previous section is. Other methods besides these messages or manual configuration may be used to tell a host what subnet mask to use. Again, a common alternative to ICMP for this is to use a host configuration protocol like DHCP. Routers do need to be able to respond to Address Mask Requests for hosts that choose to send them.

## **ICMPv4 Traceroute Messages**

The Echo and Echo Reply messages you saw earlier in this chapter are used for the most basic type of test that can be conducted between two devices: checking if they can communicate. A more sophisticated test can also be performed in order to see not only if the devices are able to talk, but also to discover the exact sequence of routers used to move datagrams between them. In TCP/IP, this diagnostic is performed using the traceroute (or tracert) utility.

The first implementation of *traceroute* used a clever application of Time Exceeded error messages, as described in the previous chapter. By sending a test message to a destination first with a Time to Live (TTL) value of 1, then 2, then 3, and so on, each router in the path between the source and destination would successively discard the test messages and send back a Time Exceeded message. Each router would then display the sequence of routers between the two hosts. This bit of trickery works well enough in general terms, but is suboptimal in a couple of respects. For example, it requires the source device to send one test message for each router in the path, instead of just a single test message. It also doesn't take into account the possibility that the path between two devices may change during the test.

Recognizing these limitations, a new experimental standard was developed in 1993 that defined a more efficient way to conduct a traceroute: RFC 1393, "Traceroute Using an IP Option." As the title suggests, this method of doing a traceroute works by having the source device send a single datagram to the destination that contains a special Traceroute IP option. Each router that sees that option while the test message is conducted along the route responds back to the original source with an ICMP Traceroute message, which is also defined in RFC 1393.

### **ICMPv4 Traceroute Message Format**

Since the *Traceroute* message was specifically designed for the traceroute utility, it was possible to incorporate extra information in it that a host tracing a route could use. The message format is as shown in Table 33-6 and Figure 33-6.

**Table 33-6:** ICMPv4 Traceroute Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMP message type; in this case, 30.
Code	1	Set to the value 0 if the datagram the source device sent was successfully sent to the next router, or 1 to indicate that the datagram was dropped (meaning the traceroute failed).
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
ID Number	2	An identification field used to match up this Traceroute message to the original message sent by the source (the one containing the Traceroute IP option).
Unused	2	Not used, set to 0.
Outbound Hop Count	2	The number of routers the original message has already passed through.
Return Hop Count	2	The number of routers the return message has passed through.
Output Link Speed	4	The speed of the link over which the Traceroute message is being sent, in bytes per second.
Output Link MTU	4	The maximum transmission unit (MTU) of the link over which the Traceroute message is being sent, in bytes.

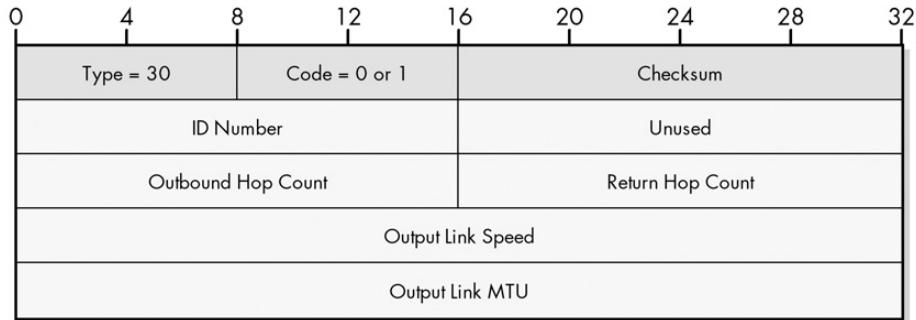


Figure 33-6: ICMPv4 Traceroute message format

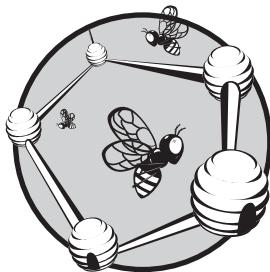
### Use of Traceroute Messages

Although this method of implementing traceroute has advantages over the older Time Exceeded messages method, it has one critical flaw as well: It requires changes to both hosts and routers to support the new IP option and the Traceroute ICMP message. People aren't big on change, especially when it comes to the basic operation of IP. For this reason, RFC 1393 never moved beyond experimental status, and most IP devices still use the older method of implementing traceroute. It is possible that you may encounter ICMP Traceroute messages, however, so it's good that you know they exist.

**KEY CONCEPT** ICMP *Traceroute* messages were designed to provide a more capable way of implementing the traceroute (`traceroute`) utility. However, most TCP/IP implementations still use ICMP Time Exceeded messages for this task.

# 34

## **ICMPV6 ERROR MESSAGE TYPES AND FORMATS**



The original Internet Control Message Protocol (ICMP) defined for version 4 of the Internet Protocol (IPv4) has a number of error messages that allow for the communication of problems on an internetwork. When IP version 6 (IPv6) was developed, the differences between IPv4 and IPv6 were significant enough that a new version of ICMP was also required: version 6 (*ICMPv6*), which is currently specified in RFC 2463. Like ICMPv4, ICMPv6 defines several error messages for informing a source that something has gone wrong.

In this chapter, I describe the four ICMPv6 error messages defined in RFC 2463. I first discuss ICMPv6 Destination Unreachable messages, which are used to tell a device that the datagram it sent could not be delivered for a variety of reasons. I describe Packet Too Big error messages, which are sent when a datagram can't be sent due to being too large for an underlying network it needs to traverse. I explain the use of Time Exceeded messages, which indicate that too much time was taken to accomplish a transmission.

I conclude with a look at Parameter Problem messages, which provide a generalized way of reporting errors that are not described by any of the preceding ICMPv6 error message types.

**NOTE** Three of the four ICMPv6 error messages (all except Packet Too Big) are equivalent to the ICMPv4 error messages that have the same names. However, to allow this chapter to stand on its own, I describe each one fully, in addition to pointing out any significant differences between the ICMPv4 and ICMPv6 version of the message.

## ICMPv6 Destination Unreachable Messages

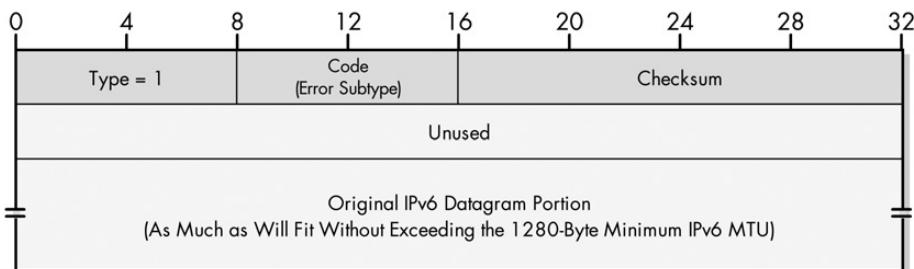
IPv6 includes some important enhancements over the older version 4, but the basic operation of the two protocols is still fundamentally the same. Like IPv4, IPv6 is an unreliable network protocol that makes a best effort to deliver datagrams, but offers no guarantees that they will always get there. Just as they did in IPv4, devices on an IPv6 network must not assume that datagrams sent to a destination will always be received.

When a datagram cannot be delivered, recovery from this condition normally falls to higher-layer protocols like the Transmission Control Protocol (TCP), which will detect the miscommunication and resend the lost datagrams. In some situations, such as a datagram that was dropped due to the congestion of a router, this is sufficient, but in other cases, a datagram may not be delivered due to an inherent problem with how it is being sent. For example, the source may have specified an invalid destination address, which means that even if it were resent many times, the datagram would never get to its intended recipient.

In general, having the source just resend undelivered datagrams while having no idea why they were lost is inefficient. It is better to have a feedback mechanism that can tell a source device about undeliverable datagrams and provide some information about why the datagram delivery failed. As in ICMPv4, in ICMPv6 this is done with *Destination Unreachable* messages. Each message includes a code that indicates the basic nature of the problem that caused the datagram not to be delivered, as well as all or part of the datagram that was undelivered in order to help the source device diagnose the problem.

### ICMPv6 Destination Unreachable Message Format

Table 34-1 and Figure 34-1 show the specific format for ICMPv6 Destination Unreachable messages.



**Figure 34-1:** ICMPv6 Destination Unreachable message format

**Table 34-1:** ICMPv6 Destination Unreachable Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Destination Unreachable messages, this is set to 1.
Code	1	Identifies the subtype of unreachable errors that are being communicated. See Table 34-2 for a full list of codes and what they mean.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Unused	4	The 4 bytes that are left blank and not used.
Original Datagram Portion	Variable	As much of the IPv6 datagram as will fit without causing the size of the ICMPv6 error message (including its own IP header) to exceed the minimum IPv6 maximum transmission unit (MTU) of 1280 bytes.

### ***ICMPv6 Destination Unreachable Message Subtypes***

There are a number of different reasons why a destination may be unreachable. To provide additional information about the nature of the problem to the device that originally tried to send the datagram, a value is placed in the message's Code field. One interesting difference between ICMPv4 and ICMPv6 Destination Unreachable messages is that there are many fewer Code values for ICMPv6. The ICMPv6 Code values were streamlined, mainly because several of the ICMPv4 codes were related to relatively obscure features that aren't applicable to ICMPv6.

Table 34-2 shows the different Code values, corresponding message subtypes, and a brief explanation of each.

**Table 34-2:** ICMPv6 Destination Unreachable Message Subtypes

Code Value	Message Subtype	Description
0	No Route to Destination	The datagram was not delivered because it could not be routed to the destination. Since this means that the datagram could not be sent to the destination device's local network, this is basically equivalent to the Network Unreachable message subtype in ICMPv4.
1	Communication with Destination Administratively Prohibited	The datagram could not be forwarded due to filtering that blocks the message based on its contents. Equivalent to the message subtype with the same name (and Code value 13) in ICMPv4.
3	Address Unreachable	There was a problem attempting to deliver the datagram to the host specified in the destination address. This code is equivalent to the ICMPv4 Host Unreachable code and usually means that the destination address was bad or that there was a problem with resolving it into a layer 2 address.
4	Port Unreachable	The destination port specified in the UDP or TCP header was invalid or does not exist on the destination host.

Note that Code value 2 is not used. Also, Destination Unreachable messages are sent only when there is a fundamental problem with delivering a particular datagram; they are not sent when a datagram is dropped simply due to congestion of a router.

## **Processing of Destination Unreachable Messages**

It is up to the recipient of an ICMPv6 Destination Unreachable message to decide what to do with it. However, just as the original datagram may not reach its destination, the Destination Unreachable message may do the same. Therefore, a device cannot rely on the receipt of one of these error messages to inform it of every delivery problem. This is especially true given that it is possible that some unreachable destination problems may not be detectable.

**KEY CONCEPT** ICMPv6 *Destination Unreachable* messages are used in the same manner as the ICMPv4 Destination Unreachable messages: to inform a sending device of a failure to deliver an IP datagram. The message's Code field provides information about the nature of the delivery problem (though the Code values are different from those in ICMPv4).

## **ICMPv6 Packet Too Big Messages**

One of the most interesting changes made to the operation of IP in version 6 is related to the process of datagram fragmentation and reassembly. In IPv4, a host can send a datagram of any size that's allowed by the IP specification out onto the internetwork. If a router needs to send the datagram over a physical link that has a maximum transmission unit (MTU) size that is too small for the size of the datagram, it will automatically fragment the datagram and send the fragments individually so they will fit. The destination device will receive the fragments and reassemble them. I explain the basics behind this in Chapter 22.

Even though it is convenient for hosts to be able to rely on routers to automatically fragment messages as needed, it is inefficient for routers to spend time doing this. For this reason, in IPv6 developers made the decision to not allow routers to fragment datagrams. This puts the responsibility on each host to ensure that the datagrams they send out are small enough to fit over every physical network between itself and any destination. This is done either by using the IPv6 default minimum MTU of 1280, which every physical link must support, or a special Path MTU Discovery process for determining the minimum MTU between a pair of devices. Again, the full details are in Chapter 22.

If an IPv6 router is not allowed to fragment an IPv6 datagram that is too large to fit on the next physical link over which it must be forwarded, what should the router do with it? The datagram can't be forwarded, so the router has no choice but to discard it. When this happens, the router is required to report this occurrence back to the device that initially sent the datagram, using an ICMPv6 *Packet Too Big* message. The source device will know that it needs to fragment the datagram in order to have it successfully reach its destination.

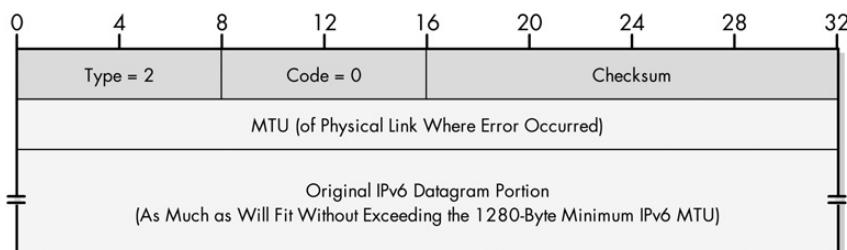
**NOTE** Recall that packet is a synonym for datagram, so you can think of this as the "Datagram Too Big" message.

### **ICMPv6 Packet Too Big Message Format**

Table 34-3 and Figure 34-2 show the format for ICMPv6 Packet Too Big messages.

**Table 34-3: ICMPv6 Packet Too Big Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Packet Too Big messages, this is set to 2.
Code	1	Not used for this message type; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
MTU	4	The MTU size, in bytes, of the physical link over which the router wanted to send the datagram, but was not able to do so due to the datagram's size. Including this value in the Packet Too Big message tells the source device the size it needs to use for its next transmission to this destination in order to avoid this problem in the future (at least for this particular link).
Original Datagram Portion	Variable	As much of the IPv6 datagram as will fit without causing the size of the ICMPv6 message (including its own IP header) to exceed the minimum IPv6 MTU of 1280 bytes.



**Figure 34-2: ICMPv6 Packet Too Big message format**

**KEY CONCEPT** In IPv6, routers are not allowed to fragment datagrams that are too large to send over a physical link to which they are connected. An oversized datagram is dropped, and an ICMPv6 Packet Too Big message is sent back to the datagram's originator to inform it of this occurrence.

### Applications of Packet Too Big Messages

While Packet Too Big is obviously an error message, it also has another use: the implementation of Path MTU Discovery. This process, described in RFC 1981, defines a way for a device to determine the minimum MTU for a path to a destination. To perform Path MTU Discovery, the source device sends a series of test messages, decreasing the size of the datagram until it no longer receives Packet Too Big messages back in response to its tests. See Chapter 27 for a bit more detail on this.

**NOTE** The Packet Too Big message is new to ICMPv6. However, its use is somewhat similar to the use of the Fragmentation Needed and DF Set version of the ICMP4 Destination Unreachable message type, which is used as part of IPv4's Path MTU Discovery feature.

Incidentally, Packet Too Big is an exception to the rule that ICMP messages are sent only in response to unicast datagrams; it may be sent in reply to an oversized multicast datagram. If this occurs, it is important to realize that some of the intended targets of the multicast may still have received it, if the path the multicast took to them did not go through the link with the small MTU that caused the error.

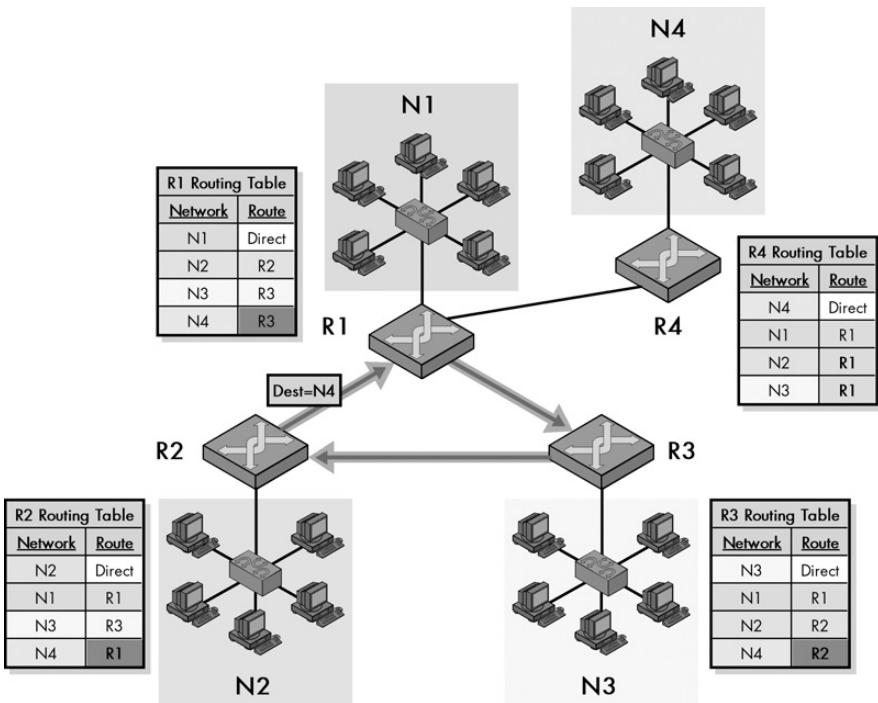
## ICMPv6 Time Exceeded Messages

The engineers who first designed IP recognized that due to the nature of how routing works on an internetwork, there was always a danger that a datagram might get lost in the system and spend too much time being passed from one router to another. They included in IPv4 datagrams a field called *Time to Live (TTL)*, which was intended to be set to a time value by the device sending the datagram and used as a timer to cause the datagram to be discarded if it took too long to get to its destination.

Eventually, the meaning of this field was changed, so it represented not a time in seconds but the number of hops the datagram was allowed to traverse. In IPv6, the new meaning of this field was formalized when it was renamed *Hop Limit*. Regardless of its name, the field still has the same basic purpose: It restricts how long a datagram can exist on an internetwork by limiting the number of times routers can forward it. This is particularly designed to provide protection against router loops that may occur in large or improperly configured internetworks. An example of this situation is where Router A thinks datagrams intended for Network X should next go to Router B, which thinks they should go to Router C, which thinks they need to go to Router A. Without a Hop Limit, such datagrams would circle forever, clogging networks and never accomplishing anything useful. Figure 34-3 illustrates the router loop problem.

Each time a router passes an IPv6 datagram, it decreases the Hop Limit field. If the value ever reaches zero, the datagram expires and is discarded. When this happens, the router that dropped the datagram sends an ICMPv6 Time Exceeded message back to the datagram's originator to inform it that the datagram was dropped. This is basically the same as the ICMPv4 *Time Exceeded* message. As in the ICMPv4 case, the device receiving the message must decide whether and how to respond to receipt of the message. For example, since a device using a Hop Limit that was too low can cause the error, the device may try to resend the datagram with a higher value before concluding that there is a routing problem and giving up. (Chapter 32 for an illustration of how TTL expiration works.)

Just as with the ICMPv4 equivalent, there is also another time expiration situation in which ICMPv6 Time Exceeded messages are used. When an IP message is broken into fragments that are sent independently, the destination device is charged with reassembling the fragments into the original message. One or more fragments may not make it to the destination, however. To prevent the device from waiting forever, it sets a timer when the first fragment arrives. If this timer expires before all of the other fragments are also received, the device gives up on this message. The fragments are tossed out, and a Time Exceeded message is generated.



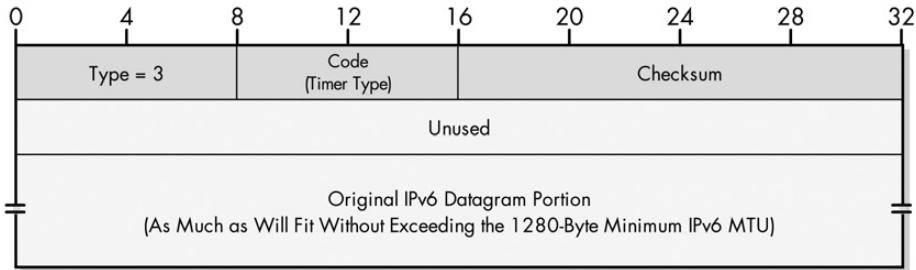
**Figure 34-3: An example of a router loop** This diagram shows a simple internetwork consisting of four networks, each of which is served by a router. It is an adaptation of Figure 23-3 from Chapter 23, but in this case, the routing tables have been set up incorrectly. Router R1 thinks that it needs to route any traffic intended for Network N4 to Router R3, which thinks it goes to Router R2, which thinks it goes back to Router R1. This means that when any device tries to send to Network N4, the datagram will circle this triangle until its Hop Limit is reached, at which point an ICMPv6 Time Exceeded message will be generated.

### ICMPv6 Time Exceeded Message Format

Table 34-4 and Figure 34-4 show the format for ICMPv6 Time Exceeded messages.

**Table 34-4: ICMPv6 Time Exceeded Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Time Exceeded messages, this is set to 3.
Code	1	Identifies the subtype of time error that's being communicated. A value of 0 indicates expiration of the Hop Limit field; a value of 1 indicates that the fragment reassembly time has been exceeded.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Unused	4	The 4 bytes left blank and not used.
Original Datagram Portion	Variable	As much of the IPv6 datagram as will fit without causing the size of the ICMPv6 error message (including its own IP header) to exceed the minimum IPv6 MTU of 1280 bytes.



**Figure 34-4:** ICMPv6 Time Exceeded message format

**KEY CONCEPT** Like their ICMPv4 namesakes, ICMPv6 *Time Exceeded* messages are sent in two different time-related circumstances. The first is if a datagram's *Hop Limit* field is reduced to zero, thereby causing it to expire and the datagram to be dropped. The second is when all the pieces of a fragmented message are not received before the recipient's reassembly timer expires.

### **Applications of Time Exceeded Messages**

In IPv4, ICMP Time Exceeded messages are used both as an error message and in a clever application to implement the TCP/IP traceroute command. This is done by first sending a dummy datagram with a TTL value of 1, thereby causing the first hop in the route to discard the datagram and send back an ICMP Time Exceeded message. Then a second datagram is sent to the same destination with a TTL value of 2, thus causing the second device in the route to report back a Time Exceeded message, and so on.

There is an IPv6 version of traceroute that is sometimes called *traceroute6*. Due to the fact that IPv6 and its protocols and applications are still in development, I have not been able to confirm definitively that traceroute6 is implemented using ICMPv6 Time Exceeded messages in the manner described earlier, but I believe this is the case (and it certainly would make sense). See Chapter 88 for more information about traceroute.

### **ICMPv6 Parameter Problem Messages**

The ICMPv6 Destination Unreachable, Packet Too Big, and Time Exceeded messages described in the previous sections are used to indicate specific error conditions to the original sender of a datagram. Recognizing that a router or host may encounter some other problem in processing a datagram that is not covered by any of these message types, ICMPv6 includes a generic error message type, just as ICMPv4 did. This is called the ICMPv6 *Parameter Problem* message.

As the name suggests, a Parameter Problem message indicates that a device found a problem with a parameter (another name for a datagram field) while attempting to work its way through the header (or headers) in an IPv6 datagram. This message is generated only when the error encountered is serious enough that the device could not make sense of the datagram and had to discard it. So, if an error is found that a device is able to recover from (does not need to drop the datagram), no Parameter Problem message is created.

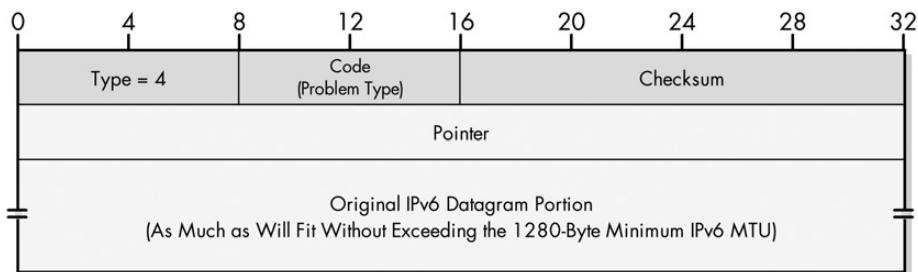
As was the case for the ICMPv4 version of this message, the ICMPv6 message was designed to be generic, so it can indicate an error in basically any field in the original datagram. A special Pointer field is used that points to the place in that datagram where the error was encountered. By looking at the structure of the original message (which, as you may recall, is included up to a certain size in the ICMP message format), the original device can tell which field contained the problem. The Code value is also used to communicate additional general information about the nature of the problem.

### **ICMPv6 Parameter Problem Message Format**

Table 34-5 and Figure 34-5 show the format for ICMPv6 Parameter Problem messages.

**Table 34-5:** ICMPv6 Parameter Problem Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Parameter Problem messages, this is set to 4.
Code	1	Identifies the general class of the parameter problem. See Table 34-6 for more information.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Pointer	4	An offset that points to the byte location in the original datagram that caused the Parameter Problem message to be generated. The device receiving the ICMP message can use this value to get an idea of which field in the original message had the problem.
Original Datagram Portion	Variable	As much of the IPv6 datagram as will fit without causing the size of the ICMPv6 error message (including its own IP header) to exceed the minimum IPv6 MTU of 1280 bytes.



**Figure 34-5:** ICMPv6 Parameter Problem message format

### **Parameter Problem Message Interpretation Codes and the Pointer Field**

The Pointer field, which was only 8 bits wide in ICMPv4, has been widened to 32 bits in ICMPv6 in order to provide more flexibility in isolating the error. The Code value is also used somewhat differently in ICMPv6 than it was in the ICMPv4 version of this message type. In ICMPv4, the Pointer was used only when the Code field was 0, and other code values indicated other problem categories for which the Pointer

field did not make sense. In ICMPv6, the Pointer field is used with all Code types to indicate the general nature of what the problem is. This means the Pointer field tells the recipient of the Parameter Problem message where the problem happened in the message, and the Code field tells it what the nature of the problem is.

Table 34-6 shows the three Code values and provides a brief explanation of each.

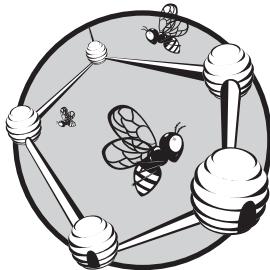
**Table 34-6:** ICMPv6 Parameter Problem Message Interpretation Codes

<b>Code Value</b>	<b>Message Subtype</b>	<b>Description</b>
0	Erroneous Header Field Encountered	The Pointer field points to a header that contains an error or otherwise could not be processed.
1	Unrecognized Next Header Type Encountered	As explained in Chapter 26, in IPv6, a datagram can have multiple headers, each of which contains a Next Header field that points to the next header in the datagram. This code indicates that the Pointer field points to a Next Header field containing an unrecognized value.
2	Unrecognized IPv6 Option Encountered	The Pointer field points to an IPv6 option that was not recognized by the processing device.

**KEY CONCEPT** The ICMPv6 *Parameter Problem* message is a generic error message that can be used to convey an error of any type in an IP datagram. The Pointer field is used to indicate where the problem was in the original datagram to the recipient of the message.

# 35

## ICMPV6 INFORMATIONAL MESSAGE TYPES AND FORMATS



In the previous chapter, we explored a number of Internet Control Message Protocol version 6 (ICMPv6) error messages.

These are sent back to the originator of an Internet Protocol version 6 (IPv6) datagram when the originator detects an error it, thereby making it impossible for the error to be delivered. Like the original version of ICMP (ICMPv4), ICMPv6 also defines another message class: *informational* messages. These ICMPv6 messages are used not to report errors, but to allow the sharing of information required to implement various test, diagnostic, and support functions critical to the operation of IPv6.

In this chapter, I describe eight different ICMPv6 informational messages in five topics (six of these messages are used in matching pairs, and the pairs are described together). I begin by describing ICMPv6 Echo Request and Echo Reply messages, which are used for network connectivity testing. I explain the format of Router Advertisement and Router Solicitation messages, which are used to let hosts discover local routers and learn necessary parameters from them. I then describe ICMPv6 Neighbor Advertisement and

Neighbor Solicitation messages, which are used for various communications between hosts on a local network, including IPv6 address resolution. I discuss IPv6 Redirect messages, which let routers inform hosts of better first-hop routers, and IPv6 Router Renumbering messages.

Several of the ICMPv6 informational messages include additional information that is either optional, recommended, or mandatory, depending on the circumstances under which the message is generated. Some of these are shared between message types, so they are described in a separate topic at the end of the chapter.

In IPv4, the use of many of the ICMP informational messages was described in a variety of different standards. In IPv6, many of the functions using informational messages have been gathered together and formalized in the IPv6 *Neighbor Discovery (ND) protocol*. The solicitation and advertisement of local routers and neighboring hosts, as well as the communication of redirection information are both examples of activities for which ND is responsible. In fact, five of the ICMP messages described in this chapter are actually defined in the ND standard, RFC 2461.

**RELATED INFORMATION** Neighbor Discovery (ND) and ICMPv6 are obviously closely related, given that ND describes the use of several of the ICMP messages: Router Advertisement, Router Solicitation, Neighbor Advertisement, Neighbor Solicitation, and Redirect. Thus, just as ICMPv4 is an important assistant to IPv4, both ICMPv6 and ND are important helpers for IPv6. In this book, I provide most of the description of how these messages are used in the next chapter, which discusses ND. In this chapter, I provide only a brief summary of their use, while focusing primarily on message format and the meaning of each of the fields in that format.

## ICMPv6 Echo Request and Echo Reply Messages

IP is a relatively simple protocol that does not include any method for performing tests between devices to help in diagnosing internetwork problems. This means that this job, like other support tasks, falls to ICMP. The simplest test performed when there is a problem using TCP/IP is usually a check that a pair of devices is able to send datagrams to each other. This is most often done by an initiating device that sends a test message to a second device, which receives it and replies back to tell the first device it received the message.

Like ICMPv4, ICMPv6 includes a pair of messages specifically for connection testing. To use them, Device A begins the test process by sending an ICMPv4 *Echo Request* message to Device B, which responds back to Device A with an *Echo Reply* message. When Device A receives this message, it knows that it is able to communicate (both send and receive) successfully with Device B.

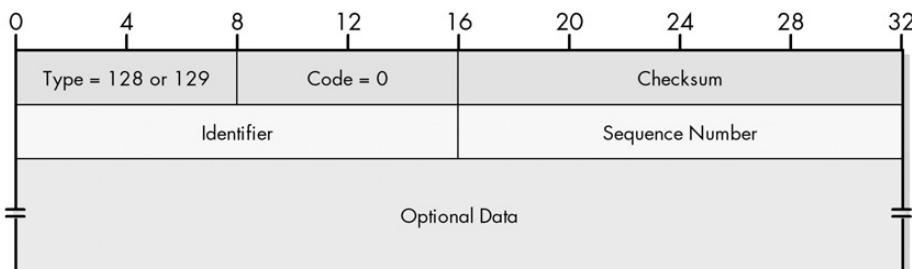
**NOTE** In ICMPv4 the first message type was named just Echo but was often called Echo Request. In ICMPv6, Request is part of the formal message name—a modest but useful improvement from a clarity standpoint.

### ICMPv6 Echo and Echo Reply Message Format

The format for ICMPv6 Echo Request and Echo Reply messages is very similar to that of the ICMPv4 version, as shown in Table 35-1 and Figure 35-1.

**Table 35-1: ICMPv6 Echo Request and Echo Reply Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Echo Request messages, the value is 128, and for Echo Reply messages, it's 129. (In ICMPv6, informational messages always have a Type value of 128 or higher.)
Code	1	Not used; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Identifier	2	An optional identification field that can be used to help in matching Echo Request and Echo Reply messages.
Sequence Number	2	A sequence number to help in matching Echo Request and Echo Reply messages.
Optional Data	Variable	Additional optional data to be sent along with the message. If this is sent in the Echo Request, it is copied into the Echo Reply to be sent back to the source.



**Figure 35-1: ICMPv6 Echo Request and Echo Reply message format**

It is often necessary to match an Echo Reply message with the Echo Request message that led to it being generated. Two special fields are used within the format of these messages to allow Echo Request and Echo Reply messages to be matched together, and to allow a sequence of messages to be exchanged. The Identifier field is provided so that a particular test session can be identified, and the Sequence Number field allows a series of tests in a session to be numbered. The use of both fields is optional.

### **Application of Echo and Echo Reply Messages**

ICMPv6 Echo Request and Echo Reply messages are used via the IPv6 version of the IP ping utility, which is commonly called *ping6*. Like its IPv4 predecessor, this utility allows an administrator to configure a number of test options to perform either a simple or rigorous test of the connection between a pair of devices. See Chapter 88 for a full explanation.

**KEY CONCEPT** ICMPv6 Echo Request and Echo Reply messages are used to facilitate network reachability testing. A device tests its ability to communicate with another by sending it an Echo Request message and waiting for an Echo Reply in response. The *ping* utility, a widely used diagnostic tool in TCP/IP internetworks, makes use of these messages.

## ICMPv6 Router Advertisement and Router Solicitation Messages

At the highest level, we can separate IP devices into two groups: hosts and routers. Both participate in the use of the internetwork, but they have different roles. An important IP principle related to this division is that routers take care of routing—moving data between networks—while hosts generally don't need to worry about this job. Hosts rely on the routers on their local networks to facilitate communication to all other hosts except those on the local network.

The implications of this are clear: A host cannot really use an internetwork until it knows the identity of at least one local router and the method by which that router is to be used. In IPv4, a technique known as *Router Discovery* was invented, which provides a means by which a host can locate a router and learn important parameters related to the operation of the local network. Router Discovery in IPv6 works in a very similar manner by having routers send *Router Advertisement* messages both on a regular basis and in response to hosts prompting for them using *Router Solicitation* messages. The Router Discovery function has been incorporated into the ND protocol, where it is part of a larger class of tools that I call *host-Router Discovery* functions.

### ICMPv6 Router Advertisement Message Format

The ICMPv6 Router Advertisement and Router Solicitation messages are fairly similar to their counterparts in ICMPv4. The main differences are in the parameters that are communicated. Since routers are responsible for a few more functions in IPv6 than they are in IPv4, the Router Advertisement message in ICMPv6 has a few more fields than the older version.

The format of an ICMPv6 Router Advertisement message is described in Table 35-2 and shown in Figure 35-2.

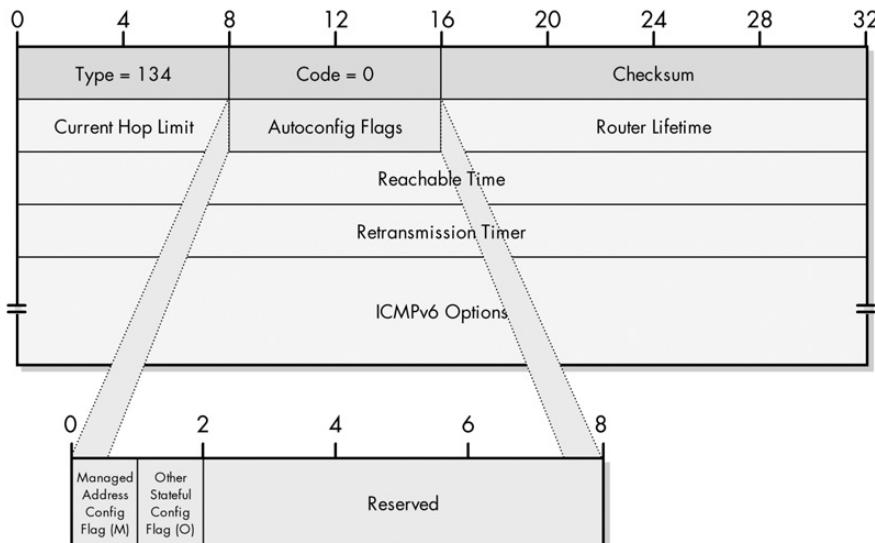


Figure 35-2: ICMPv6 Router Advertisement message format

**Table 35-2:** ICMPv6 Router Advertisement Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Router Advertisement messages, the value is 134.
Code	1	Not used; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Cur Hop Limit	1	Current Hop Limit: This is a default number that the router recommends that hosts on the local network use as a value in the Hop Limit field of datagrams they send. If 0, the router is not recommending a Hop Limit value in this Router Advertisement.
Autoconfig Flags	1	Two flags that let the router tell the host how autoconfiguration is performed on the local network, as described in Table 35-3. (See Chapter 25 for details on IPv6 autoconfiguration.)
Router Lifetime	2	Tells the host receiving this message how long, in seconds, this router should be used as a default router. If 0, it tells the host this router should not be used as a default router. Note that this is an expiration interval only for the status of the router as a default, not for other information in the Router Advertisement message.
Reachable Time	4	Tells hosts how long, in milliseconds, they should consider a neighbor to be reachable after they have received reachability confirmation. (See Chapter 36 for more information.)
Retrans Timer	4	Retransmission Timer: The amount of time, in milliseconds, that a host should wait before retransmitting Neighbor Solicitation messages.
Options	Variable	Router Advertisement messages may contain three possible options (see the “ICMPv6 Informational Message Options” section later in this chapter for more on ICMPv6 options): <ul style="list-style-type: none"><li>• Source Link-Layer Address: Included when the router sending the Advertisement knows its link-layer (layer 2) address.</li><li>• MTU: Used to tell local hosts the MTU of the local network when hosts on the network may not know this information.</li><li>• Prefix Information: Tells local hosts what prefix or prefixes to use for the local network. (You’ll recall that the “prefix” indicates which bits of an IPv6 address are the network identifier when compared to the host identifier; it is thus analogous to an IPv4 subnet mask.)</li></ul>

**Table 35-3:** ICMPv6 Router Advertisement Message Autoconfiguration Flags

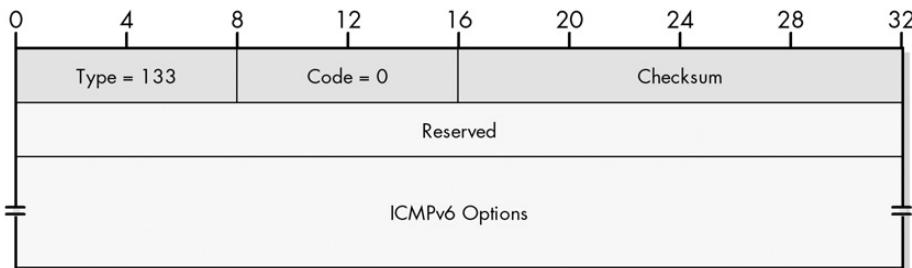
Subfield Name	Size (Bytes)	Description
M	1/8 (1 bit)	Managed Address Configuration Flag: When set, this flag tells hosts to use an administered or stateful method for address autoconfiguration, such as the Dynamic Host Configuration Protocol (DHCP).
O	1/8 (1 bit)	Other Stateful Configuration Flag: When set, this tells hosts to use an administered or stateful autoconfiguration method for information other than addresses.
Reserved	6/8 (6 bits)	Reserved for future use; sent as zeros.

## **ICMPv6 Router Solicitation Message Format**

The format of an ICMPv6 Router Solicitation message is shown in Table 35-4 and Figure 35-3.

**Table 35-4:** ICMPv6 Router Solicitation Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Router Solicitation messages, the value is 133.
Code	1	Not used; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Reserved	4	The 4 reserved bytes set to 0.
Options	Variable	If the device sending the Router Solicitation knows its layer 2 address, it should be included in a Source Link-Layer Address option. Option formats are described in the “ICMPv6 Informational Message Options” section later in this chapter.



**Figure 35-3:** ICMPv6 Router Solicitation message format

## **Addressing of Router Advertisement and Router Solicitation Messages**

Router Solicitation messages are normally sent to the IPv6 “all routers” multicast address; this is the most efficient method, because routers are required to subscribe to this multicast address while hosts will ignore it. A routine (unsolicited) Router Advertisement message is sent to all devices using the “all nodes” multicast address for the local network. A Router Advertisement message that is sent in response to a Router Solicitation message goes in unicast back to the device that sent the solicitation.

**KEY CONCEPT** ICMPv6 Router Advertisement messages are sent regularly by IPv6 routers to inform hosts of their presence and characteristics, and to provide hosts with parameters that they need to function properly on the local network. A host that wants to find out immediately which routers are present may send a Router Solicitation message, which will prompt listening routers to send out Router Advertisements.

## ICMPv6 Neighbor Advertisement and Neighbor Solicitation Messages

The previous section described the Router Advertisement and Router Solicitation messages, which are used to facilitate host–Router Discovery functions as part of the IPv6 ND protocol. The other main group of tasks for which ND is responsible relates to the exchange of information between neighboring hosts on the same network. I call these *host-host communication* or *host-host discovery* functions.

Arguably, the most important additions to the ND protocol are the functions that formalize the exchange of parameters and the methods that determine the existence of neighboring hosts. These tasks include the new method of address resolution in IPv6 as well as the processes of next-hop determination and neighbor unreachability detection. They require the use of two ICMPv6 messages: the *Neighbor Solicitation message* and the *Neighbor Advertisement message*.

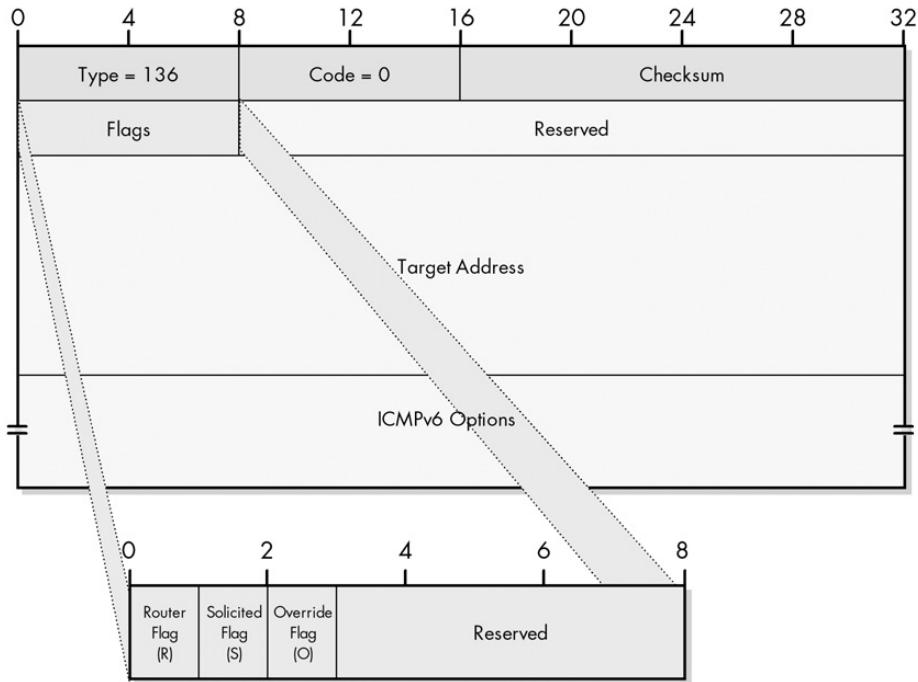
The Neighbor Solicitation message allows a device to check that a neighbor exists and is reachable, and lets a device initiate address resolution. The Neighbor Advertisement message confirms the existence of a host or router, and also provides layer 2 address information when needed. As you can see, these two messages are comparable to the Router Advertisement and Router Solicitation messages, but they are used differently and include different parameters.

### ICMPv6 Neighbor Advertisement Message Format

The format for the Neighbor Advertisement message is shown in Table 35-5 and Figure 35-4.

**Table 35-5:** ICMPv6 Neighbor Advertisement Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Neighbor Advertisement messages, the value is 136.
Code	1	Not used; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Flags	4	Three flags that convey information about the message (and a lot of empty space for future use), as described in Table 35-6.
Target Address	16	If the Neighbor Advertisement is being sent in response to a Neighbor Solicitation, this is the same value as in the Target Address field of the Solicitation. This field will commonly contain the IPv6 address of the device, thereby sending the Neighbor Advertisement, but not in all cases. For example, if a device responds as a proxy for the target of the Neighbor Solicitation, the Target Address field contains the address of the target, not the device sending the response. (See Chapter 13 for details on address resolution proxying.) If the Neighbor Advertisement is being sent unsolicited, then this is the IPv6 address of the device sending it.
Options	Variable	When sent in response to a multicast Neighbor Solicitation, a Neighbor Advertisement message must contain a Target Link-Layer Address option, which carries the link-layer address of the device sending the message. This is a good example of an option that's not really "optional." When the Neighbor Advertisement is sent in response to a unicast Neighbor Solicitation, this option is technically not required (since the sender of the Solicitation must already have the target's link-layer address to have sent it unicast). Despite this, it is still normally included to ensure that the link-layer address of the target is refreshed in the cache of the device that sent the Neighbor Solicitation.



**Figure 35-4:** ICMPv6 Neighbor Advertisement message format

**Table 35-6:** ICMPv6 Neighbor Advertisement Message Flags

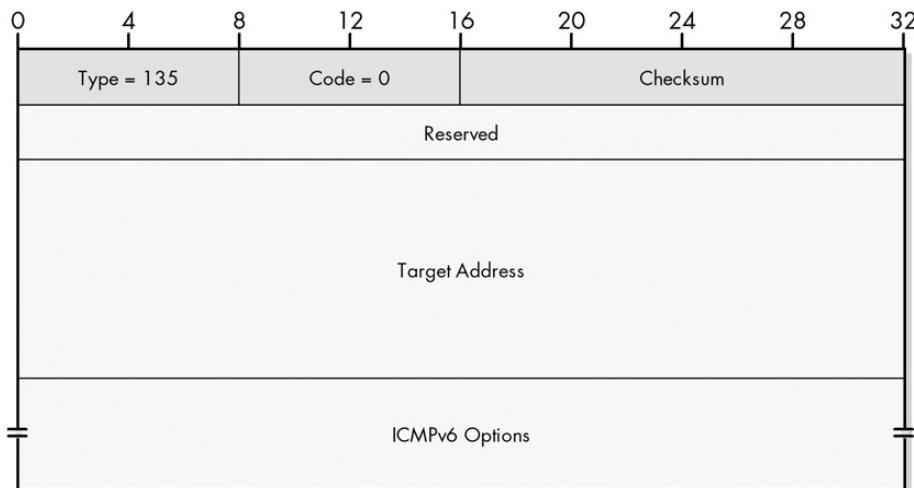
Subfield Name	Size (Bytes)	Description
R	1/8 (1 bit)	Router Flag: Set when a router sends a Neighbor Advertisement, and cleared when a host sends one. This identifies the type of device that sent the datagram, and is also used as part of neighbor unreachability detection to detect when a device changes from acting as a router to functioning as a regular host.
S	1/8 (1 bit)	Solicited Flag: When set, indicates that this message was sent in response to a Neighbor Solicitation message. Cleared for unsolicited Neighbor Advertisements.
O	1/8 (1 bit)	Override Flag: When set, tells the recipient that the information in this message should override any existing cached entry for the link-layer address of this device. This bit is normally set in unsolicited Neighbor Advertisements, since these are sent when a host needs to force a change of information in the caches of its neighbors.
Reserved	3 5/8 (29 bits)	A big set of reserved bits.

### ***ICMPv6 Neighbor Solicitation Message Format***

The Neighbor Solicitation message format is much simpler, as shown in Table 35-7 and Figure 35-5.

**Table 35-7: ICMPv6 Neighbor Solicitation Message Format**

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Neighbor Solicitation messages, the value is 135.
Code	1	Not used; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Reserved	4	The 4 reserved bytes set to 0.
Target Address	16	The IPv6 address of the target of the solicitation. For IPv6 address resolution, this is the actual unicast IP address of the device whose layer 2 (link-layer) address we are trying to resolve.
Options	Variable	If the device sending the Neighbor Solicitation knows both its own IP address and layer 2 address, it should include the layer 2 address in a Source Link-Layer Address option. The inclusion of this option will allow the destination of the Neighbor Solicitation to enter the layer 2 and layer 3 addresses of the source of this message into its own address cache. (See the discussion of IPv6 address resolution in Chapter 25.)



**Figure 35-5: ICMPv6 Neighbor Solicitation message format**

### ***Addressing of Neighbor Advertisement and Neighbor Solicitation Messages***

Neighbor Solicitation messages are sent either unicast to the address of the target device or to the solicited-node multicast address of the target. This latter address is a special type that's used to allow a device to send a multicast that will be heard by the target whose address it is trying to resolve, but won't be heard by most other devices; it is explained in Chapter 25, which describes IPv6 address resolution.

When a Neighbor Advertisement message is generated in response to a Neighbor Solicitation message, it is sent unicast back to the device that sent the Solicitation message, unless that message was sent from the unspecified address, in which case it is multicast to the “all nodes” multicast address. If the Neighbor

Advertisement message is sent unsolicited (for example, by a device that wishes to inform others of a change in link-layer address), it is sent to the “all nodes” multicast address.

**KEY CONCEPT** ICMPv6 *Neighbor Advertisement* and *Neighbor Solicitation* messages are similar in many ways to the Router Advertisement and Router Solicitation messages. However, rather than being used to communicate parameters from routers to hosts, they are used for various types of communication between hosts on a physical network, such as address resolution, next-hop determination, and neighbor unreachability detection.

## ICMPv6 Redirect Messages

Because of the different roles of routers and hosts in an IPv6 internetwork, hosts don’t need to know very much about routes. They send datagrams intended for destinations on the local network directly, while they send those for other networks to their local routers and let them “do the driving,” so to speak.

If a local network has only a single router, it will send all such nonlocal traffic to that router. If it has more than one local router, the host then must decide which router to use for which traffic. In general terms, a host will not know the most efficient choice of router for every type of datagram it may need to send. In fact, many nodes start out with a limited routing table that says to send *everything* to a single default router, even if there are several routers on the network.

When a router receives datagrams destined for certain networks, it may realize that it would be more efficient if a host to a different router on the local network sent such traffic. If so, it will invoke the Redirect function by sending an ICMPv6 *Redirect* message to the device that sent the original datagram. This is the last of the functions that is performed in IPv6 by the ND protocol and is explained in Chapter 36.

**NOTE** In ICMPv6, the *Redirect* message is informational and no longer considered an error message as it was in ICMPv4.

### ICMPv6 Redirect Message Format

The format of ICMPv6 Redirect messages is shown in Table 35-8 and Figure 35-6.

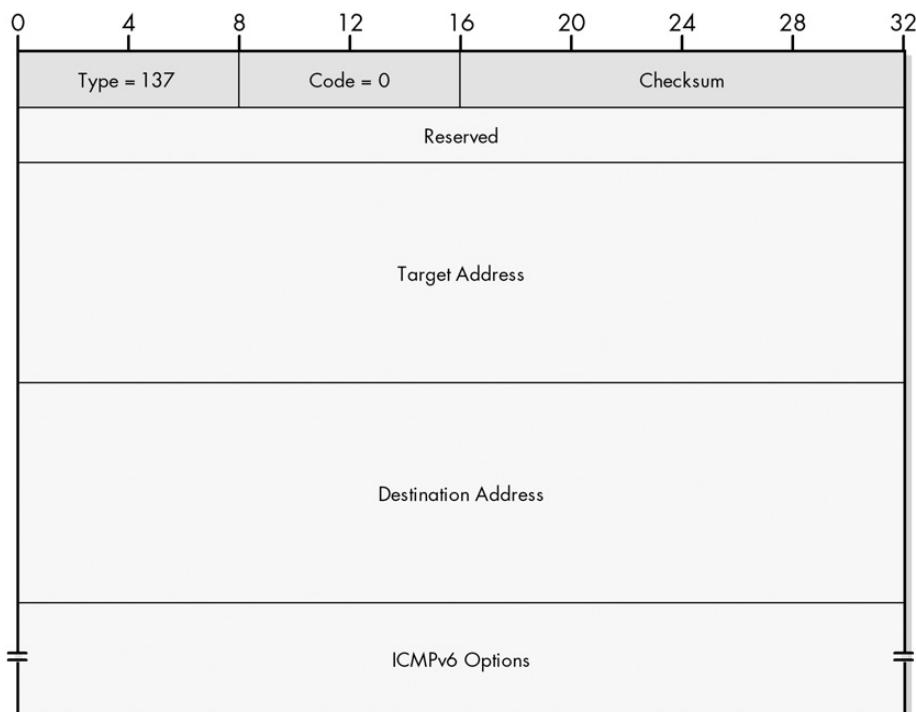
**Table 35-8:** ICMPv6 Redirect Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Redirect messages, the value is 137.
Code	1	Not used; set to 0.
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Reserved	4	The 4 bytes sent as zeros.

(continued)

**Table 35-8:** ICMPv6 Redirect Message Format (continued)

Field Name	Size (Bytes)	Description
Target Address	16	The address of the router that the router creating the Redirect is telling the recipient of the Redirect to use as a first hop for future transmissions to the destination. For example, if Router R2 generated a Redirect telling Host A that, in the future, transmissions to Host B should be sent first to Router R1, then Router R1's IPv6 address would be in this field.
Destination Address	16	The address of the device whose future transmissions are being redirected; this is the destination of the datagram that originally led to the Redirect being generated. Repeating the previous example: If Router R2 generated a Redirect telling Host A that, in the future, transmissions to Host B should be sent first to Router R1, then Host B's IPv6 address would be in this field.
Options	Variable	Redirect messages normally include two ICMPv6 option fields (see the "ICMPv6 Informational Message Options" section later in this chapter): <ul style="list-style-type: none"> <li>• Target Link-Layer Address: The layer 2 address of the Target Address, if known. This saves the recipient of the Redirect message from needing to perform an address resolution on the target.</li> <li>• Redirected Header: As much of the IPv6 datagram that spawned this Redirect as will fit without causing the size of the ICMPv6 error message (including its own IP header) to exceed the minimum IPv6 MTU of 1280 bytes.</li> </ul>



**Figure 35-6:** ICMPv6 Redirect message format

Redirect messages are always sent in unicast to the address of the device that originally sent the datagram that originally created the Redirect message.

## ***Application of Redirect Messages***

The Redirect message has always been somewhat of an oddball. In ICMPv4, it is considered an error message, but this makes it different from other error messages. For one thing, it's not really an error, since it doesn't represent a failure to deliver, only an inefficiency in doing so. For this reason, in ICMPv6 it was moved into the set of informational message types. Here, too, it doesn't really fit in with the others, since it is sent in reaction to a regular IP message, and it also includes a copy of (part of) the datagram that spawned it, as error messages do.

**KEY CONCEPT** ICMPv6 *Redirect* messages are used by a router to inform a host of a better router to use for future datagrams that were sent to a particular host or network. They are not used to alter routes between routers, however.

## ***ICMPv6 Router Renumbering Messages***

One of the more interesting decisions made in IPv6 was the selection of a very large 128-bit address size. This provides an address space far larger than what humans are ever likely to need, and probably larger than needed for IPv6, strictly speaking. What this wealth of bits provides is the flexibility to assign meaning to different bits in the address structure. This, in turn, serves as the basis for important features such as the autoconfiguration and automated renumbering of IPv6 addresses.

### ***IPv6 Router Renumbering***

The renumbering feature in IPv6 is of particular interest to network administrators, since it has the potential to make large network migrations and merges much simpler. In August 2000, the IETF published RFC 2894, “Router Renumbering for IPv6,” which describes a similar technique that allows routers in an autonomous system to be renumbered by giving them new prefixes (network identifiers).

Router renumbering is actually a fairly simple process, especially if we avoid the gory details, which is exactly what I intend to do. A network administrator uses a device on the internetwork to generate one or more *Router Renumbering Command* messages. These messages provide a list of prefixes of routers that are to be renumbered. Each router processes these messages to see if the addresses on any of their interfaces match the specified prefixes. If so, they change the matched prefixes to the new ones specified in the message. Additional information is also included in the Router Renumbering Command message to control how and when the renumbering is done.

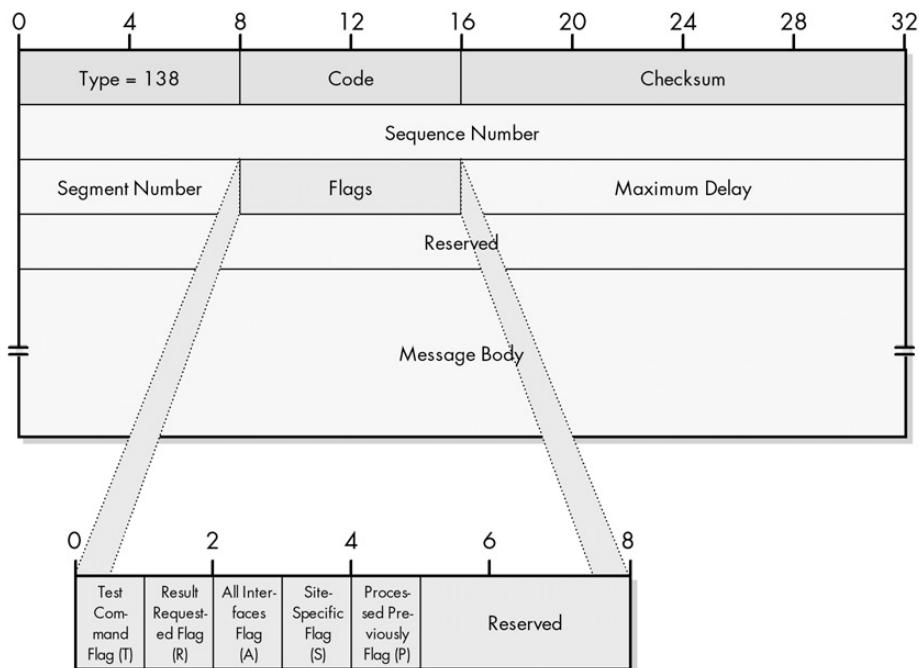
If requested, each router processing a Command message will respond with a *Router Renumbering Result* message. This serves as feedback to the originator of the Command message, indicating whether the renumbering was successful, and what changes, if any, were made.

The router renumbering standard also defines a few important management features. Many of these reflect the great power of something that can mass-renumber routers, and hence, they represent the potential for such power to be abused. Command messages may be sent in a test mode, in which they are processed but the

renumbering is not actually done. Messages include a sequence number to guard against replay attacks, and a special *Sequence Number Reset* message can be used to reset the sequence number information that was previously sent. For added security, the standard specifies that messages be authenticated and have their identity checked.

### **ICMPv6 Router Renumbering Message Format**

The format of Router Renumbering messages is shown in Table 35-9 and Figure 35-7.



**Figure 35-7:** ICMPv6 Router Renumbering message format

**Table 35-9:** ICMPv6 Router Renumbering Message Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 message type; for Router Renumbering messages, the value is 138.
Code	1	Indicates the subtype of Router Renumbering message: 0 = Router Renumbering Command 1 = Router Renumbering Result 255 = Sequence Number Reset
Checksum	2	A 16-bit checksum field for the ICMP header (see Chapter 31).
Sequence Number	4	A 32-bit number that guards against replay attacks by letting a recipient detect stale, duplicate, or out-of-order commands.

(continued)

**Table 35-9:** ICMPv6 Router Renumbering Message Format (continued)

<b>Field Name</b>	<b>Size (Bytes)</b>	<b>Description</b>
Segment Number	1	Differentiates between valid Router Renumbering messages within the same Sequence Number.
Flags	1	Five flags used to control the renumbering process, as described in Table 35-10.
Max Delay	2	Tells a router receiving a message the maximum amount of time (in milliseconds) it is allowed to delay before sending a reply.
Reserved	4	The 4 reserved bytes.
Message Body	Variable	For a Router Renumbering Command, the message body contains two sets of information. The first is a Match-Prefix Part for the prefix being renumbered. The second is one or more Use-Prefix Parts that describe the new prefix for each match. A router receiving a Command checks its own interface addresses, and if they match the Match-Prefix-Part, they use Use-Prefix Parts data to accomplish the renumbering. For a Router Renumbering Result, the message body contains zero or more Match Results entries that describe each prefix that a router has matched from a Router Renumbering Command. Each entry provides information about whether renumbering for a prefix was successful.

Table 35-10 shows the Router Renumbering Message flags. The first four flags (T, R, A, and S) control the operation of Command messages. They are just copied verbatim in a Result message from the Command message that led to the Result message being created. The P flag is used only in Result messages (0 in Command messages).

**Table 35-10:** ICMPv6 Router Renumbering Message Flags

<b>Subfield Name</b>	<b>Size (Bytes)</b>	<b>Description</b>
T	1/8 (1 bit)	Test Command Flag: When set to 1, this flags this Command message as being a test message. This tells the recipient to only simulate processing of the renumbering, not to actually do it.
R	1/8 (1 bit)	Result Requested Flag: When set to 1, requests that a Result message be sent after processing the Command message. When set to 0, says not to send one.
A	1/8 (1 bit)	All Interfaces Flag: When this flag is clear (0), the Command message is not applied to any router interfaces that have been administratively shut down. When 1, it is applied to all interfaces.
S	1/8 (1 bit)	Site-Specific Flag: This flag has meaning only when a router treats its interfaces as belonging to different sites. If so, a value of 1 tells it to apply the Command message only to interfaces on the same site as the interface for which the Command message was received. A value of 0 applies it to all interfaces regardless of site.
P	1/8 (1 bit)	Processed Previously Flag: This flag is normally 0, meaning the Command message was not previously seen and the Result message contains the report of processing it. When 1, this indicates that the recipient of the Command message believes it has seen it before and is not processing it. (Test commands are not included in the assessment of whether a Command message has been seen before.)
Reserved	3/8 (3 bits)	Three bits reserved for future flags.

## ***Addressing of Router Renumbering Messages***

Since Router Renumbering messages are intended for all routers on a site, they are normally sent to the “all routers” multicast address, using either link-local or site-local scope. They may also be sent to local unicast addresses.

## ***ICMPv6 Informational Message Options***

Each of the five ICMPv6 informational message types defined and used by the protocol has an Options field into which one or more options may be inserted. This probably isn’t the best name for these sets of data, since they are only optional in certain cases. In fact, in some cases the option is actually the entire point of the message. For example, a Neighbor Advertisement message containing a link-layer address for address resolution carries it in an Options field, but the message wouldn’t be of much use without it!

Each option has its own structure of subfields based on the classic type, length, and value triplet used in many message formats. The Type subfield indicates the option type, and the Length field indicates its length, so that the device processing the option can determine where it ends. The value may be contained in one or more fields, which hold the actual information for which the option is being used.

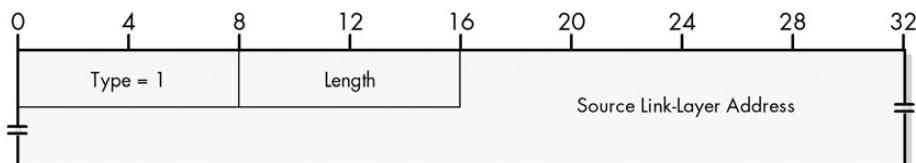
Some options are used for only one kind of ICMPv6 message; others are used for more than one variety. So, they are best thought of as modular components used in different types of messages as needed. I describe the format of each of these five options in the following sections.

### ***Source Link-Layer Address Option Format***

The Source Link-Layer Address Option carries the link-layer address of a device sending an ICMPv6 message, as shown in Table 35-11 and Figure 35-8. It’s used in Router Advertisement, Router Solicitation, and Neighbor Solicitation messages.

**Table 35-11:** ICMPv6 Source Link-Layer Address Option Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 option type. For the Source Link-Layer Address option, the value is 1.
Length	1	The length of the entire option (including the Type and Length fields), expressed in units of 8 octets (64 bits).
Source Link-Layer Address	Variable	The link-layer (layer 2) address of the device sending the ICMPv6 message.



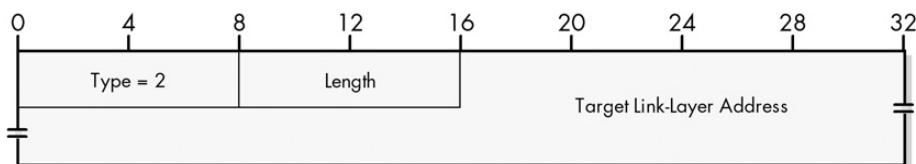
**Figure 35-8:** ICMPv6 Source Link-Layer Address option format

## **Target Link-Layer Address Option Format**

The Target Link-Layer Address option carries the link-layer address corresponding to the Target Address field in Neighbor Advertisement and Redirect messages. Its format is shown in Table 35-12 and Figure 35-9.

**Table 35-12:** ICMPv6 Target Link-Layer Address Option Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 option type. For the Target Link-Layer Address option, the value is 2.
Length	1	The length of the entire option (including the Type and Length fields), expressed in units of 8 octets (64 bits).
Target Link-Layer Address	Variable	The link-layer (layer 2) address of the target device.



**Figure 35-9:** ICMPv6 Target Link-Layer Address option format

## **Prefix Information Option Format**

The *Prefix Information* option provides a prefix and related information in Router Advertisement messages. This is the longest and most complex of the options, as you can see in Table 35-13 and Figure 35-10.

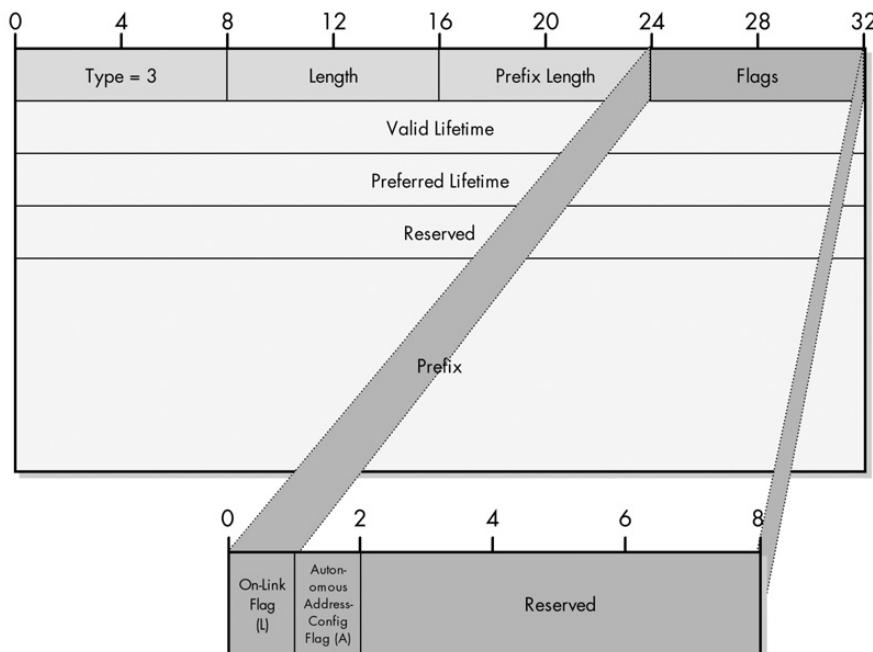
**Table 35-13:** ICMPv6 Prefix Information Option Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 option type. For the Prefix Information option, the value is 3.
Length	1	The length of the entire option (including the Type and Length fields), expressed in units of 8 octets (64 bits). The Prefix Information option is fixed in size at 32 bytes, so the value of the Length field is 4.
Prefix Length	1	The number of bits in the Prefix field that are considered part of the network identifier (the remainder are used for the host identifier and ignored). See Chapter 25 for details on prefix lengths.
Flags	1	A pair of flags that convey information about the prefix, as described in Table 35-14.
Valid Lifetime	4	The amount of time, in seconds, that the recipient of the message containing this option should consider the prefix valid for purposes of on-link determination (see the description of the L flag in Table 35-14). A value of all 1s means infinity (forever).
Preferred Lifetime	4	When the recipient of this prefix uses it to automatically generate addresses using address auto-configuration, this specifies the amount of time, in seconds, that such addresses remain preferred (meaning, valid and freely usable). A value of all 1s means infinity (forever).
Reserved	4	The 4 unused bytes sent as zeros.

(continued)

**Table 35-13: ICMPv6 Prefix Information Option Format (continued)**

Field Name	Size (Bytes)	Description
Prefix	16	The prefix being communicated from the router to the host in the Router Advertisement message. The Prefix Length field indicates how many of the 128 bits in this field are significant (part of the network ID). Only these bits are placed in the Prefix field; the remaining bits are cleared to zero.



**Figure 35-10: ICMPv6 Prefix Information option format**

**Table 35-14: ICMPv6 Prefix Information Option Flags**

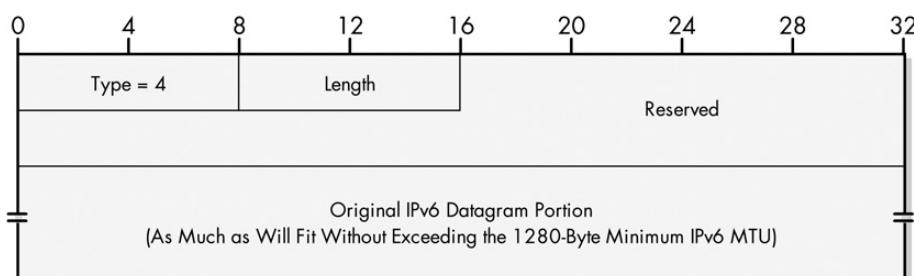
Subfield Name	Size (Bytes)	Description
L	1/8 (1 bit)	On-Link Flag: When set to 1, tells the recipient of the option that this prefix can be used for on-link determination. This means the prefix can be used for deciding whether or not an address is <i>on-link</i> (on the recipient's local network). When 0, the sender is making no statement regarding whether the prefix can be used for this or not.
A	1/8 (1 bit)	Autonomous Address-Configuration Flag: When set to 1, specifies that this prefix can be used for IPv6 address autoconfiguration. (See Chapter 25 for details on IPv6 autoconfiguration.)
Reserved	6/8 (6 bits)	6 leftover bits reserved and sent as zeros.

### ***Redirected Header Option Format***

In a Redirect message, the *Redirected Header* option provides a copy of the original message (or a portion of it) that led to the Redirect message being generated. This option's format is shown in Table 35-15 and Figure 35-11.

**Table 35-15:** ICMPv6 Redirected Header Option Format

Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 option type. For the Redirected Header option, the value is 4.
Length	1	The length of the entire option (including the Type and Length fields), expressed in units of 8 octets (64 bits).
Reserved	6	The 6 reserved bytes sent as zeros.
IP Header + Data	Variable	As much of the original IPv6 datagram as will fit without causing the size of the ICMPv6 error message (including its own IP header) to exceed the minimum IPv6 MTU of 1280 bytes.



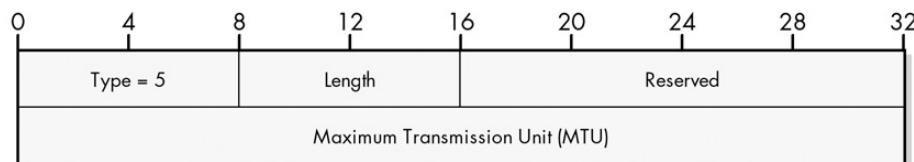
**Figure 35-11:** ICMPv6 Redirected Header option format

### MTU Option Format

The *MTU* option lets a router convey a recommended MTU value in Router Advertisement messages. Its format is shown in Table 35-16 and Figure 35-12.

**Table 35-16:** ICMPv6 MTU Option Format

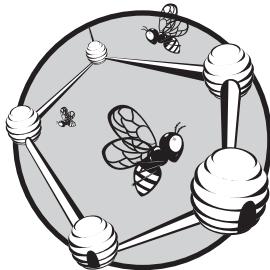
Field Name	Size (Bytes)	Description
Type	1	Identifies the ICMPv6 option type. For the MTU option, the value is 5.
Length	1	The length of the entire option (including the Type and Length fields), expressed in units of 8 octets (64 bits). The MTU option is fixed in length at 8 bytes, so the value of this field is 1.
Reserved	2	The 2 reserved bytes sent as zeros.
MTU	4	The MTU value, in bytes, that the router is recommending for use on the local link.



**Figure 35-12:** ICMPv6 MTU option format

# 36

## **IPV6 NEIGHBOR DISCOVERY (ND) PROTOCOL**



The new Internet Protocol version 6 (IPv6) represents an evolution of the venerable IP.

It maintains the same basic operational principles of IPv4, but makes some important modifications, particularly in the area of addressing. In fact, some of the more significant changes in IPv6 are actually not in IP itself, but in the protocols that support IP. One of the most interesting of these was the creation of an entirely new support protocol for IPv6. It combines several tasks previously performed by other protocols in IPv4, adds some new functions, and makes numerous improvements to the whole package. This new standard is called the IPv6 *Neighbor Discovery (ND)* protocol.

In this chapter, I describe the new ND protocol used in IPv6. I begin with an overview of the protocol, discussing its history, the motivation for its creation, and the standards that define it. I then describe its operation in general terms, listing the fundamental functions that ND performs, the three groups these functions fit into, and the Internet Control Message Protocol version 6 (ICMPv6) message types used to carry them out. I describe the key

differences between ND and the way that its functions were carried out in IPv4. I then provide more information on the three functional groups in ND: those that involve discovery of important internetwork information from routers, those that are related to address resolution and neighbor communication between hosts, and finally, those involved with router redirection.

**BACKGROUND INFORMATION** *This chapter assumes basic comprehension of IPv6, which, in turn, requires understanding IPv4. ND uses ICMPv6 messages, so I reference Chapters 31 to 35, which discuss them. Finally, since ICMP performs some of the functions done by the Address Resolution Protocol (ARP) in IPv4, you may need to refer to Chapter 13 if you're unfamiliar with ARP's operation.*

## IPv6 ND Overview

The purpose of network layer protocols like IP is to provide a means of connecting together individual local networks to create a much larger internetwork. To higher-layer protocols and to users, this internetwork behaves in most respects as if it were a single large network, because the lower layers hide the details that hold together the individual networks. Any device can send information to any other regardless of where it is located, and like magic, it will work—at least most of the time.

The existence of an internetwork means that devices can treat all other devices as peers, at least from the perspective of higher-layer protocols and applications. From the standpoint of lower layers, however, there is a very important difference between devices that are on a host's local network and those that are elsewhere. In a general sense, most devices have a more important relationship with the devices that are on their local network than those that are far away. Some of the most obvious tasks that a device must perform specifically with other devices on its local network include the following:

**Direct Datagram Delivery** Devices deliver data directly to other devices on their local network, while data going to distant devices must be indirectly delivered (routed).

**Layer 2 Addressing** To facilitate direct delivery, devices need to know the layer 2 addresses of the other devices on the local network; they don't need to know them for nonlocal devices.

**Router Identification** To deliver indirectly, a device needs to find a router on its local network that it can talk to.

**Router Communication** The local router must communicate information to each of the local hosts using it, so the hosts know how best to use it.

**Configuration** Hosts will usually look to information provided by local devices to let them perform configuration tasks such as determining their own IP address.

To support these and other requirements, several special protocols and functions were developed along with the original IP (version 4). The IP addressing scheme lets devices differentiate local addresses from distant ones. The Address Resolution Protocol (ARP) lets devices determine layer 2 addresses from layer 3

addresses. ICMP provides a messaging system to support various communication requirements between local devices, including the ability of a host to find a local router and the router to provide information to local hosts.

These features all work properly in IPv4, but they were developed in sort of an ad hoc manner. They are defined not in a single place, but rather in a variety of different Internet standards. There were also some limitations with the way these local device functions were implemented.

### ***Formalizing Local Network Functions: The Neighbor Concept***

IPv6 represents the biggest change in decades to not just the IP itself, but the entire TCP/IP suite. It thus provided an ideal opportunity to formalize and integrate the many disparate functions and tasks related to communication between local devices. The result was the creation of a new protocol: *Neighbor Discovery for IP version 6*, also commonly called the *IPv6 Neighbor Discovery* protocol. Since this protocol is new in IPv6, there is no IPv4 version of it, so the name is usually just seen as the *ND* protocol with no further qualifications; its use with IPv6 is implied.

The term *neighbor* is one that has been used for years in various networking standards and technologies to refer to devices that are local to each other. In the context of the current discussion, two devices are *neighbors* if they are on the same local network, meaning that they can send information to each other directly. The term can refer to either a regular host or a router. I think this is a good analogy to the way humans refer to those who live or work nearby. Just as most of us have a special relationship with people who are our neighbors and communicate more with them than with those who are far away, so do IP devices.

Since a neighbor is a local device, the name of the ND protocol would seem to indicate that ND is all about how neighbors discover each other's existence. In the context of this protocol, however, the term *discovery* has a much more generic meaning: It refers to discovering not just who are neighbors are, but also discovering important information about them. In addition to letting devices identify their neighbors, ND facilitates all the tasks listed earlier, including such functions as address resolution, parameter communication, autoconfiguration, and much more, as you will see in this chapter.

**KEY CONCEPT** The new *IPv6 Neighbor Discovery (ND)* protocol formalizes for IPv6 a number of functions related to communication between devices on a local network that are performed in IPv4 by protocols such as ARP and ICMP. ND is considered another helper protocol for IPv6 and is closely related to ICMPv6.

### ***Neighbor Discovery Standards***

The ND protocol was originally defined in RFC 1970, published in August 1996, and revised in the current defining standard, RFC 2461, published December 1998. Most of the functions of the ND protocol are implemented using a set of five special ICMPv6 control messages, which were discussed in the previous chapter. Thus, to some extent, the operation of ND is partially described by the ICMPv6 standard, RFC 2463. Where ICMPv4 can be considered IPv4's "administrative

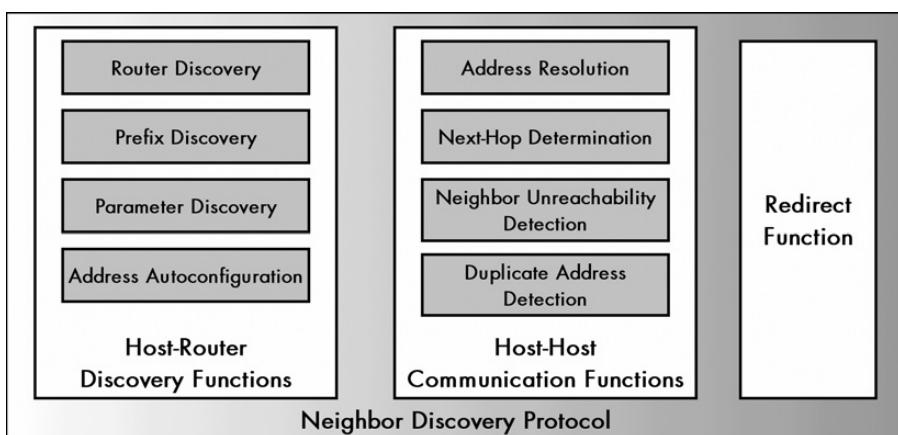
assistant,” IPv6 really has two such assistants working closely together: ICMPv6 and ND. I discuss more of the differences between the ways IPv4 and IPv6 implement ND’s functions later in this chapter.

## IPv6 ND General Operational Overview

As I just mentioned, the name of the ND protocol really does not do it justice. The protocol facilitates not merely the discovery of neighboring devices, but also a substantial number of functions related to local network connectivity, datagram routing, and configuration. Both regular hosts and routers in an IPv6 environment count on the ND protocol to facilitate important exchanges of information that are necessary for proper internetwork operation.

The ND protocol has a number of similarities to ICMP. An important one is that like ICMP, ND is a *messaging* protocol. It doesn’t implement a single specific function, but rather a group of activities that are performed through the exchange of messages. This means I can’t explain the operation of ND through a specific description of what ND does, but rather must define its operation by means of a list of messages that ND provides, and the specific ways that those messages are used.

Any local network on an internetwork will have both regular hosts and routers, and the term *neighbor* can refer to either. Of course, hosts and routers play different roles on a network, and as a result, ND is very different for each. The ND standard describes nine specific functions performed by the protocol. To better understand these functions and how they are related, we can divide them into three functional groups based on communication type and the kinds of devices involved, as illustrated in Figure 36-1.



**Figure 36-1:** Neighbor Discovery (ND) protocol functional groups and functions

Two main groups of functions in ND are those for handling router discovery and those for handling communications between hosts. A third functional group consists of just the Redirect function.

## **Host-Router Discovery Functions**

ND host-router discovery functions are those that facilitate the discovery of local routers and the exchange of information between routers and hosts. This includes four specific functions:

**Router Discovery (RD)** RD is the core function of this group. It's the method by which hosts locate routers on their local network.

**Prefix Discovery** Closely related to the process of RD is Prefix Discovery. Recall that the term *prefix* refers to the network identifier portion of an IP address. Hosts use this function to determine the network they are on, which, in turn, tells them how to differentiate between local and distant destinations and whether to attempt direct or indirect delivery of datagrams.

**Parameter Discovery** Also closely related to RD, this is the method by which a host learns important parameters about the local network and/or routers, such as the maximum transmission unit (MTU) of the local link.

**Address Autoconfiguration** Hosts in IPv6 are designed to be able to automatically configure themselves, but this requires information that is normally provided by a router.

## **Host-Host Communication Functions**

The other main group of functions is associated with information determination and communication directly between nodes, usually hosts. Some of these functions can be performed between hosts and routers, but this group is not specifically related to RD. Host-host communication functions include the following:

**Address Resolution** The process by which a device determines the layer 2 address of another device on the local network from that device's layer 3 (IP) address. This is the job performed by ARP in IPv4.

**Next-Hop Determination** The method for looking at an IP datagram's destination address and determining where it should next be sent.

**Neighbor Unreachability Detection** The process of determining whether or not a neighbor device can be directly contacted.

**Duplicate Address Detection** Determining if an address that a device wishes to use already exists on the network.

## **Redirect Function**

The last functional group contains just one function: *Redirect*. This is the technique whereby a router informs a host of a better next-hop node to use for a particular destination.

**KEY CONCEPT** ND encompasses nine individual functions, many of which are related to each other. They are organized into three functional groups: *host-router discovery functions*, *host-host communications functions*, and the *Redirect function*.

## ***Relationships Between Functions***

The division of ND's overall functionality into nine tasks in three groups is somewhat arbitrary, but provides a good frame of reference for understanding what the protocol does. Some of the functions in different groups are related; next-hop determination uses information obtained as part of Parameter Discovery. The Redirect function is also a form of router-host communication but is distinct from RD.

### ***ICMPv6 Messages Used by ND***

Just as ND is similar to ICMP in its operation, the two protocols are related in another way: the way that they perform messaging. ND actually implements its functions using ICMPv6 messages. A set of five message types is described in the ND standard:

**Router Advertisement Messages** Sent regularly by routers to tell hosts that they exist and provide important prefix and parameter information to them.

**Router Solicitation Messages** Sent by hosts to request that any local routers send a Router Advertisement message so they don't have to wait for the next regular advertisement message.

**Neighbor Advertisement Messages** Sent by hosts to indicate the existence of the host and provide information about it.

**Neighbor Solicitation Messages** Sent to verify the existence of another host and to ask it to transmit a Neighbor Advertisement message.

**Redirect Messages** Sent by a router to tell a host of a better method to route data to a particular destination.

We'll look at how these message types are used later in this chapter. See Chapter 35 for the structures of each of these five ICMPv6 message types used by ND.

## ***IPv6 ND Functions Compared to Equivalent IPv4 Functions***

The IPv6 ND protocol has the distinction of being the only truly new protocol created as part of the core of IPv6; there is no previous version of ND. Of course, most of the services that ND provides to IPv6 were also required in IPv4. They were just provided in a rather diverse set of protocols and standards that the ND protocol has formalized, integrated, and improved.

What this means is that while ND is new, the jobs it does are equivalent to the tasks performed by several other protocols in IPv4. Specifically, the bulk of ND functions correspond to the following set of standards, features, and message types in IPv4:

**ICMPv4 Router Discovery** Most of the functions associated with identifying and obtaining information from routers in ND are based on the use of ICMPv4 Router Advertisement and Router Solicitation messages, as defined in RFC 1256.

**Address Resolution Protocol** ND provides enhanced address resolution capabilities that are similar to the functions provided in IPv4 by ARP.

**ICMPv4 Redirect** ND's Redirect function and Redirect messages are based on similar functionality defined in IPv4 and ICMPv4.

There are other aspects of ND that only somewhat correlate to how things work in IPv4. There are also improvements or new functionality compared to how these IPv4 functions work. Some of these are due to differences in how IPv6 itself operates compared to IPv4. For example, Prefix Discovery in ND is sort of related to the Address Mask Request and Address Mask Reply messaging in ICMPv4.

Overall, ND represents a substantial improvement compared to the way its job was done in IPv4. Like IPv6 itself, ND is generally better suited to the needs of modern networks than the older protocols. Some of the more important specific improvements made in ND compared to how its job was done in IPv4 include the following:

**Formalizing of Router Discovery** In IPv4, the process of RD and solicitation was arguably an afterthought. ND formalizes this process and makes it part of the core of the TCP/IP protocol suite.

**Formalizing of Address Resolution** In a similar manner, address resolution is handled in a superior way in ND, which functions at layer 3 and is tightly tied to IP, just as ICMP is. There is no more need for an ambiguously layered protocol like ARP, whose implementation depends greatly on the underlying physical and data link layers.

**Ability to Perform Functions Securely** ND operates at the network layer, so it can make use of the authentication and encryption capabilities of IPsec for tasks such as address resolution and RD.

**Autoconfiguration** In combination with features built into IPv6, ND allows many devices to automatically configure themselves, without the need for something like a Dynamic Host Configuration Protocol (DHCP) server (though DHCPv6 does also exist).

**Dynamic Router Selection** Devices use ND to detect if neighbors are reachable. If a device is using a router that stops being reachable, it will detect this and automatically switch to another one.

**Multicast-Based Address Resolution** Address resolution is performed using special multicast addresses instead of broadcasts, thereby reducing unnecessary disruption of “innocent bystanders” when resolution messages must be sent.

**Better Redirection** Improvements have been made to the method for generating and using Redirect messages.

## IPv6 ND Host-Router Discovery Functions

Connecting individual networks together creates internetworks. The devices that are responsible for this connection of networks are routers, which send data from one network to the next. A host must rely on a router to forward transmissions to all devices other than those on the local network. For this reason, before a host can properly use an internetwork, it needs to find a local router and learn important information about both the router and the network itself. Enabling this information exchange is one of the most important jobs of the IPv6 ND protocol.

The general term used to describe most of the ND communication between hosts and routers on a local network is *discovery*. As I mentioned earlier in this chapter, the term encompasses not merely discovery of the router, but also communication of important parameters. Most of this communication flows from the routers to the hosts, since routers control the way that each network is used. They provide information to hosts so the hosts know how best to operate.

The various discovery features related to host-router communication are all facilitated by the same exchange of two different ICMPv6 message types. Router Advertisement messages are sent only by routers, and they contain information about the router and the network on which it is located. Router Solicitation messages are optional, and they are sent by hosts when they want to find a local router. The format of each of these messages is described in Chapter 35.

Note that both Router Advertisement and Router Solicitation messages may include an optional layer 2 address of the device sending the message. This is used to update address resolution caches to save time when address resolution is needed later.

The mechanisms for using these messages are not really that complicated. The best way to see how the discovery process works overall is to look at the specific tasks performed both by routers and hosts in ND. Let’s start by looking at the functions that routers perform.

### ***Host-Router Discovery Functions Performed by Routers***

Routers are responsible for the following functions:

**Routine Advertisement** The main job that routers do in ND is to regularly transmit Router Advertisement messages. Each router maintains a timer that controls how often an advertisement is sent out. Advertisements are also sent when any sort of special situation arises. For example, a message will be sent if key information about the router changes, such as its address on the local network. Router Advertisement messages include key information about both the router and the network. See Chapter 35 for a full description of the Router Advertisement message format.

**Parameter Maintenance** Routers are responsible for maintaining key parameters about the local network, so they can be sent in advertisements. These include the default Hop Limit field value that should be used by hosts on the network, a default MTU value for the network, and information such as network prefixes, which are used for both first-hop routing by hosts and autoconfiguration. Again, some more details on these can be found in Chapter 35.

**Solicitation Processing** Routers listen for Router Solicitation messages. When one is received, they will immediately send a Router Advertisement to the requesting host.

### ***Host-Router Discovery Functions Performed by Hosts***

For their part, hosts are responsible for three main functions:

**Advertisement Processing** Hosts listen for advertisements on their local network and process them. They then set appropriate parameters based on the information in these messages. This includes maintaining various data structures such as lists of prefixes and routers, which are updated regularly as new advertisement information comes in.

**Solicitation Generation** Under certain conditions, a host will generate a Router Solicitation and send it out on the local network. This very simple message just requests that any local routers that hear it immediately send a Router Advertisement message back to the device that made the request. This is most often done when a host is first turned on, so it doesn't have to sit waiting for the next routine advertisement.

**Autoconfiguration** When required, and if the network supports the function, the host will use information from the local router to allow it to automatically configure itself with an IP address and other parameters.

**KEY CONCEPT** One of the two main functional groups of ND is the set of *host-router discovery* functions. They allow hosts on a local network to discover the identity of a local router and learn important parameters about how the network is to be used. Host-router discovery operations are performed using ICMPv6 Router Advertisement and Router Solicitation messages.

## **IPv6 ND Host-Host Communication Functions**

The delivery of datagrams in IP can be divided into two methods: direct and indirect. Indirect datagram delivery requires that routers provide help to hosts, which leads to the host-router discovery functions described in the previous section. Direct delivery of datagrams is performed from one host to another on the same network. This doesn't require the use of routers, but necessitates other IPv6 ND protocol functions that involve communication directly between local hosts. These include next-hop determination, address resolution, neighbor unreachability detection, and duplicate address detection.

## **Next-Hop Determination**

The first task that any host must perform when it wants to send a datagram is *next-hop determination*. This is the process by which a device looks at the destination address in a datagram and decides whether direct or indirect delivery is required. In early IPv4, this was done by looking at the class of the address, and later on, by using the subnet mask. In IPv6, the prefix information obtained from local routers is compared to the destination of the datagram to determine if the destination device is local or distant. If it is local, the next hop is the same as the destination address; if it is not local, the next hop is chosen from the device's list of local routers (which are determined either by manual configuration or using the host-router discovery features of ND).

For efficiency purposes, hosts do not perform this next-hop determination for each and every datagram. They maintain a destination cache that contains information about what the next hop should be for recent devices to which datagrams have been sent. Each time a next-hop determination is performed for a particular destination, information from that determination is entered into the cache so that it can be used the next time datagrams are sent to that device.

## **Address Resolution**

If a host determines that the destination of a datagram is local, it will then need to send the datagram to that device. The actual transmission will occur using whatever physical layer and data link layer technology has been used to implement the local network. This requires the host to know the layer 2 address of the destination, even though it generally has only the layer 3 address from the datagram. Getting from the layer 3 address to the layer 2 address is known as the address resolution problem.

In IPv6, the ND protocol is responsible for address resolution. When a host wants to get the layer 2 address of a datagram destination it sends an ICMPv6 Neighbor Solicitation message containing the IP address of the device whose layer 2 address it wishes to determine. That device responds back with a Neighbor Advertisement message that contains its layer 2 address. Instead of using a broadcast that would disrupt each device on the local network, the solicitation is sent using a special multicast to the destination device's solicited-node address. See Chapters 13 and 25 for more information about address resolution in IPv6.

Note also that even though this discussion does concentrate on communication between hosts, address resolution may also be done when a host needs to send a datagram to a local router and has no entry for it in its destination cache. In the context of address resolution, a destination device is just a neighbor. Whether it is a host or a router matters only in terms of what happens after the datagram has been sent and received. In other words, these host-to-host functions are so named only because they are not specific to the communication between hosts and routers like the tasks in the preceding section.

## **Updating Neighbors Using Neighbor Advertisement Messages**

Devices do not routinely send Neighbor Advertisement messages the way that routers send Router Advertisement messages. There really isn't any need for this: Neighbors don't change much over time, and resolution will occur naturally over time as devices send datagrams to each other. In addition, having advertisements sent regularly by so many devices on a network would be wasteful.

A host may, however, send an unsolicited Neighbor Advertisement message under certain conditions where it feels it is necessary to immediately provide updated information to other neighbors on the local network. A good example of this is a hardware failure—in particular, the failure of a network interface card. When the card is replaced, the device's layer 2 (MAC) address will change. Assuming the device's IP layer can detect this, it can send out an unsolicited Neighbor Advertisement message to tell other devices to update their resolution caches with the new MAC address.

## **Neighbor Unreachability Detection and the Neighbor Cache**

Neighbor Solicitation and Neighbor Advertisement messages are most often associated with address resolution, but they also have other purposes. One of these is neighbor unreachability detection. Each device maintains information about each of its neighbors and updates it dynamically as network conditions change. The information is kept for both host and router devices that are neighbors on the local network. Knowing that a device has become unreachable is important because a host can adapt its behavior accordingly. In the case of an unreachable host, a device may wait a certain period of time before trying to send datagrams to an unreachable host, instead of flooding the network with repeated attempts to send to the host. An unreachable router, on the other hand, is a signal that the device needs to find a new router to use, if an alternate is available.

Each host maintains a neighbor cache that contains information about neighboring devices. Each time a host receives a datagram from a neighbor, it knows the neighbor is reachable at that particular moment, so the device makes an entry in the cache for the neighbor to indicate this. Of course, receiving a datagram from a neighbor means only that the neighbor is reachable now; the more time that elapses since the last datagram was received, the greater the chance that something has happened to make the neighbor no longer reachable.

For this reason, neighbor reachability information must be considered temporary. Each time a neighbor is entered into the cache as reachable, a timer is started. When the timer expires, the reachability information for that neighbor is considered stale, and reachability is no longer assumed for that neighbor. When a new datagram is received from the neighbor in question, the timer is reset and the cache is again set to indicate that the device is reachable. The amount of time a host should consider a neighbor reachable before expiring it is communicated by a local router using a field in a Router Advertisement message.

A host can also dynamically seek out a neighbor if it needs to know its reachability status. It sends a Neighbor Solicitation message to the device and waits for a Neighbor Advertisement message in response. It then updates the cache accordingly.

## Duplicate Address Detection

The last use of the two messages we have been discussing here is for duplicate address detection. When a host uses the IPv6 autoconfiguration facility, one of the steps in the process is to ensure that the address it is trying to use doesn't already exist on the network. This is done by sending a Neighbor Solicitation message to the address the device wishes to use. If a Neighbor Advertisement message is received in reply, the address is already in use.

**KEY CONCEPT** The second of the two main functional groups of ND is the set of *host-host communication* functions. Two ICMPv6 messages, Neighbor Advertisement and Neighbor Solicitation, are defined. They enable a variety of types of essential communication between adjacent hosts on a local network. These include address resolution, determining the next hop to which a datagram should be sent, and also the assessment of a neighboring device's reachability.

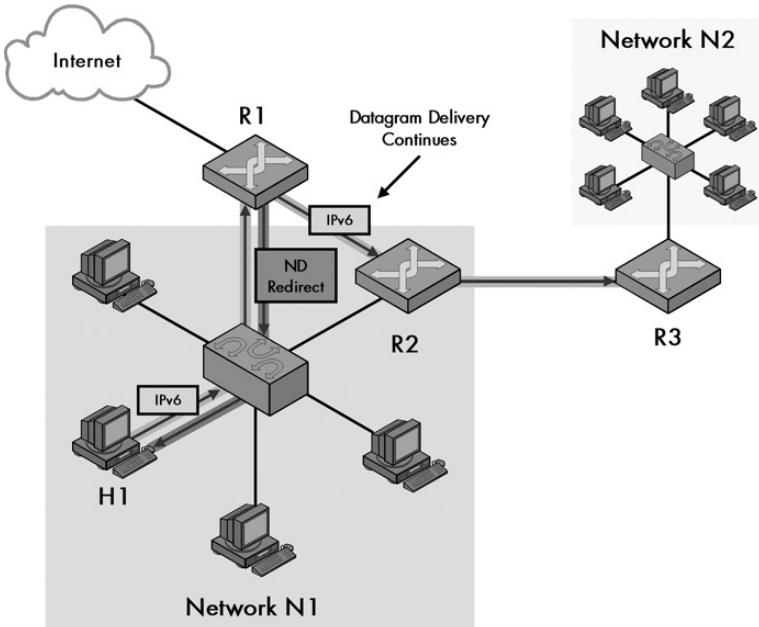
## IPv6 ND Redirect Function

The last of the major responsibilities of the IPv6 ND protocol is the *Redirect* function. This is used by a router to inform a host of a better route to use for datagrams that have been sent to a particular destination. An argument could be made that the Redirect function should be part of the host-router group since it represents a form of communication between routers and regular hosts. However, it is somewhat different from the other discovery functions, and so the standard treats it separately.

Routers are responsible for detecting situations where a host on the local network has made an inefficient first-hop routing decision, and then attempting to correct it. For example, consider a network that has two routers on it, R1 and R2. A Host H1 wants to send a datagram to Device X2 on another network that is connected to Host H1's network through Router R2. If Host H1 sends the datagram to Router R1, that router will know it must go through Router R2, and will send it there. Seeing that Router R2 was also on the local network, Router R1 therefore knows that Host H1 made a poor initial routing decision: The datagram should have been sent to Router R2 directly, not Router R1. If this sounds very similar to ICMPv4's redirect feature, that's because it is!

In response, Router R1 will create a special ICMPv6 Redirect message. This message will tell Host H1 that for any subsequent datagrams that will be sent to Device X2 should be first sent to Router R2, instead of to Router R1. It is also possible that a router may determine other situations where the first hop from a particular host should be different and will advise the host using a Redirect message. This is illustrated in Figure 36-2.

Only routers send Redirect messages, not hosts. Hosts are responsible for looking for these Redirect messages and processing them. A host receiving such a message will look in it to see which destination's datagram led to the redirection notice, and which new first hop the router is saying the host should use in the future for that destination. In this example, Host H1 will see that Router R1 is saying that any further datagrams to Device X2 should be sent to Router R2 instead of Router R1. Host H1 will update its destination cache for Device X2 accordingly.



**Figure 36-2: ND host redirection using an ICMPv6 Redirect message** Host H1 sends to Router R1 an IPv6 datagram destined for a device on Network N2. However, Router R1 notices that Router R2 is on the same network as the source device and is a more direct route to Network N2. It forwards the datagram on to Router R2 but also sends an ICMPv6 Redirect message back to Host H1 to tell it to use Router R2 next time.

**KEY CONCEPT** The ND Redirect function allows a router to tell a host to use a different router for future transmissions to a particular destination. It is similar to the IPv4 redirect feature and is implemented using ICMPv6 Redirect messages.

When a router sends a Redirect message, it may also include in the message the data link layer address of the destination to which it is redirecting. This address is used by the host to update its address resolution cache, if necessary. This may save bandwidth in the future by eliminating an address resolution cycle, when the redirected host tries to send to the new, redirected location. In the example, Router R1 may include Router R2's own layer 2 address in the Redirect message. This can be used by Host H1 the next time it has a datagram for Device X2.

IPv6 also supports the authentication of Redirect messages to prevent unauthorized devices from causing havoc by sending inappropriate Redirect messages. A host may be configured to discard Redirect messages that are not properly authenticated.



# PART II-7

## **TCP/IP ROUTING PROTOCOLS (GATEWAY PROTOCOLS)**

Routing is not just one of the most important activities that take place at the network layer; it is also the function that really *defines* layer 3 of the OSI Reference Model. Routing is what enables small local networks to be linked together to form potentially huge internetworks that can span cities, countries, or even the entire globe. The job of routing is done by special devices called *routers*, which forward datagrams from network to network, allowing any device to send to any other device, even if the source has no idea where the destination is.

Strictly speaking, an argument could be made that some routing protocols don't belong in layer 3. For example, many of these protocols send messages using the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) at layer 4. Despite this, routing is inherently a layer 3 activity, and for this reason, it is traditional to consider routing protocols part of layer 3.

Routing is a complicated subject. The short summary of the process is that routers decide how to forward a datagram based on its destination address, which is compared to information the router keeps in special routing tables. These tables contain entries for each of the networks the router knows about, telling the router which adjacent router the datagram should be sent to in order for it to reach its eventual destination.

As you can imagine, routing tables are critically important to the routing process. It is possible for these tables to be manually maintained by network administrators, but this is tedious and time-consuming and doesn't allow routers to deal with changes or problems in the internetwork. Instead, most modern routers are designed with functionality that lets them share route information with other routers, so they can keep their routing tables up-to-date automatically. This information exchange is accomplished through the use of *routing protocols*.

This part contains five chapters that provide a description of the most common routing (or *gateway*) protocols used in TCP/IP. The first chapter provides an overview of various concepts that are important to know in order to understand how routing protocols work, including an explanation of the difference between interior and exterior routing protocols. This sets the stage for the chapters that follow.

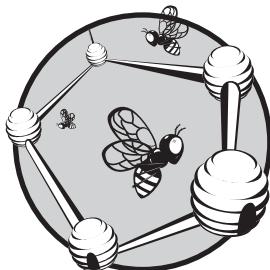
In the second and third chapters, I thoroughly explain the two most commonly used interior routing protocols in TCP/IP: the Routing Information Protocol (RIP) and the Open Shortest Path First (OSPF) protocol. In the fourth chapter, I describe the Border Gateway Protocol (BGP), which is the exterior routing protocol used today on the Internet. The fifth chapter briefly discussing five historical, proprietary, or less commonly used routing protocols.

You may notice that the title of this part refers to both *routing protocols* and *gateway protocols*. These terms are interchangeable, and the word *gateway* appears in the name of several of the protocols. This is an artifact of the historical use of the term *gateway* in early TCP/IP standards to refer to the devices we now call routers. Today, the term *gateway* normally refers not to a router, but to a different type of network interconnection device, so this can be particularly confusing. The term *routing protocol* is now preferred, and it is the one I use.

Like all topics related to routing, routing protocols are generally quite complex. I cover the major ones here in more detail than most general networking references, but even so, I am only scratching the surface, especially of the more complicated ones like OSPF. You can check out the referenced Internet standards (RFCs) for more details if you desire. Also note that there are some routing protocols in use on IP networks that I do not cover here, such as IS-IS (which is actually an OSI protocol and not formally part of TCP/IP).

# 37

## OVERVIEW OF KEY ROUTING PROTOCOL CONCEPTS



Routing protocols play an important part in the overall process of routing in an internetwork. It is therefore easiest to understand them in the scope of an overall discussion of routing. It's difficult to describe the individual TCP/IP routing protocols without some background information on how routing protocols work. For this reason, I feel it is worth taking a brief look at key routing protocol concepts so that you will have more luck making sense of the routing protocols described in the next few chapters.

In this chapter, I will provide an overview of the routing protocol architectures, protocol types, algorithms, and metrics.

### Routing Protocol Architectures

Let's start with a look at routing protocol architectures. In this context, the word *architecture* refers to the way that an internetwork is structured. Once you have some networks and routers that you wish to connect together, there

are any number of ways that you can do this. The architecture you choose is based on the way that routers are linked, and this has an impact on the way that routing is done and how routing protocols operate.

## **Core Architecture**

TCP/IP and the Internet were developed simultaneously, so TCP/IP routing protocols evolved as the Internet itself did. Early architecture of the Internet consisted of a small number of *core* routers that contained comprehensive information about the internetwork. When the Internet was very small, adding more routers to this core expanded it. However, each time the core was expanded, the amount of routing information that needed to be maintained grew.

Eventually, the core became too large, so a two-level hierarchy was formed to allow further expansion. *Noncore* routers were located on the periphery of the core and contained only partial routing information; they relied on the core routers for transmissions that went across the internetwork. A special routing protocol called the *Gateway-to-Gateway Protocol (GGP)* was used within the core of the internetwork, while another protocol called the *Exterior Gateway Protocol (EGP)* was used between noncore and core routers. The noncore routers were sometimes single, stand-alone routers that connected a single network to the core, or they could be sets of routers for an organization.

This architecture served for a while, but it did not scale very well as the Internet grew. The problem was mainly due to the fact that there was only a single level to the architecture: Every router in the core had to communicate with every other router. Even with peripheral routers being kept outside the core, the amount of traffic in the core kept growing.

## **Autonomous System (AS) Architecture**

To resolve the scaling problem, a new architecture was created that moved away from the centralized concept of a core toward an architecture that was better suited to a larger and growing internetwork. This decentralized architecture treats the internetwork as a set of independent groups, with each group called an *autonomous system (AS)*. An AS consists of a set of routers and networks controlled by a particular organization or administrative entity, which uses a single consistent policy for internal routing.

The power of this system is that routing on the internetwork as a whole occurs between ASes and not individual routers. Information is shared between one and maybe a couple of routers in each AS, not every router in each AS. The details of routing within an AS are also hidden from the rest of the internetwork. This provides both flexibility for each AS to do routing as it sees fit (thus the name *autonomous*) and efficiency for the overall internetwork. Each AS has its own number, and the numbers are globally managed to make sure that they are unique across an internetwork (such as the Internet).

**KEY CONCEPT** Large, modern TCP/IP internetworks can contain thousands of routers. To better manage routing in such an environment, routers are grouped into constructs called *autonomous systems (ASes)*, each of which consists of a group of routers managed independently by a particular organization or entity.

## **Modern Protocol Types: Interior and Exterior Routing Protocols**

The different nature of routing within an AS and between ASes can be seen in the fact that the following distinct sets of TCP/IP routing protocols are used for each type:

**Interior Routing Protocols** These protocols are used to exchange routing information between routers within an AS. Interior routing protocols are not used between ASes.

**Exterior Routing Protocols** These protocols are used to exchange routing information between ASes. They may in some cases be used between routers within an AS, but they primarily deal with exchanging information between ASes.

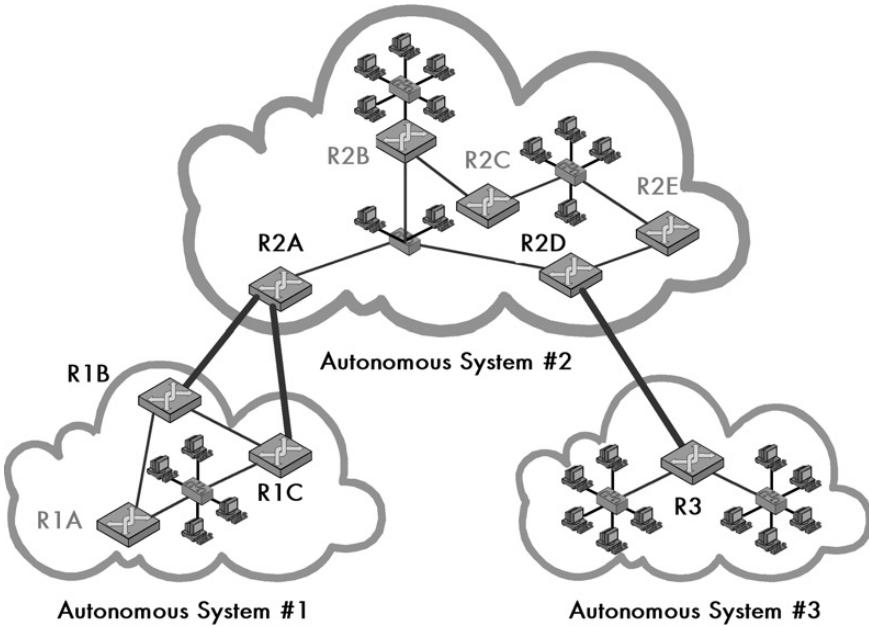
**KEY CONCEPT** Interior routing protocols are used to share routing information within an autonomous system; each AS may use a different interior routing protocol because the system is, as the name says, autonomous. Exterior routing protocols convey routing data between ASes; each AS must use the same exterior protocol to ensure that it can communicate.

Since ASes are just sets of routers, you connect ASes by linking a router in one AS to a router in another AS. Architecturally, an AS consists of a set of routers with two different types of connectivity:

**Internal Routers** Some routers in an AS connect only to other routers in the same AS. These run interior routing protocols.

**Border Routers** Some routers in an AS connect both to routers within the AS and to routers in one or more other ASes. These devices are responsible for passing traffic between the AS and the rest of the internetwork. They run both interior and exterior routing protocols.

Due to its advantages, the AS architecture, an example of which can be seen in Figure 37-1, has become the standard for TCP/IP networks, most notably the Internet. The division of routing protocols into the interior and exterior classifications has thus also become standard, and all modern TCP/IP routing protocols are first subdivided by type in this manner.



**Figure 37-1: TCP/IP autonomous system (AS) routing architecture** This diagram shows a simplified Internet organized into three ASes, each of which is managed independently from the others. Communication within each AS is done using an interior routing protocol chosen by that AS's administrators (thin links). Communication between ASes must be done using a common exterior routing protocol (thick links). Internal routers are shown in lighter text, and border routers are shown in black text.

## Routing Protocol Algorithms and Metrics

Another key differentiation of routing protocols is on the basis of the *algorithms* and *metrics* they use. An algorithm refers to a method that the protocol uses for determining the best route between any pair of networks, and for sharing routing information between routers. A metric is a measure of “cost” that is used to assess the efficiency of a particular route. Since internetworks can be quite complex, the algorithms and metrics of a protocol are very important, and they can be the determining factor in deciding that one protocol is superior to another.

There are two routing protocol algorithms that are most commonly encountered: distance vector and link state. There are also protocols that use a combination of these methods or other methods.

### **Distance-Vector (Bellman-Ford) Routing Protocol Algorithm**

A *distance-vector* routing algorithm, also called a *Bellman-Ford* algorithm after two of its inventors, is one where routes are selected based on the distance between networks. The distance metric is something simple—usually the number of *hops*, or routers, between them.

Routers using this type of protocol maintain information about the distance to all known networks in a table. They regularly send that table to each router they immediately connect with (their *neighbors* or *peers*). These routers then update their

tables and send those tables to their neighbors. This causes distance information to propagate across the internetwork, so that eventually, each router obtains distance information about all networks on the internetwork.

Distance-vector routing protocols are somewhat limited in their ability to choose the best route. They also are subject to certain problems in their operation that must be worked around through the addition of special heuristics and features. Their chief advantages are simplicity and history (they have been used for a long time).

### ***Link-State (Shortest-Path First) Routing Protocol Algorithm***

A *link-state* algorithm selects routes based on a dynamic assessment of the shortest path between any two networks. For that reason, it's also called a *shortest-path first* method.

Using this method, each router maintains a map describing the current topology of the internetwork. This map is updated regularly by testing reachability of different parts of the Internet, and by exchanging link-state information with other routers. The determination of the best route (or shortest path) can be made based on a variety of metrics that indicate the true cost of sending a datagram over a particular route.

Link-state algorithms are much more powerful than distance-vector algorithms. They adapt dynamically to changing internetwork conditions, and they also allow routes to be selected based on more realistic metrics of cost than simply the number of hops between networks. However, they are more complicated to set up and use more computer processing resources than distance-vector algorithms, and they aren't as well established.

### ***Hybrid Routing Protocol Algorithms***

There are also hybrid protocols that combine features from both types of algorithms, and other protocols that use completely different algorithms. For example, the *Border Gateway Protocol (BGP)* is a path-vector algorithm, which is somewhat similar to the distance-vector algorithm, but communicates much more detailed route information. It includes some of the attributes of distance-vector and link-state protocols, but is more than just a combination of the two.

## **Static and Dynamic Routing Protocols**

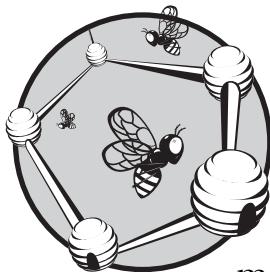
Finally, you may also occasionally see routing protocols categorized by type as *static* and *dynamic*. This terminology is somewhat misleading.

The term *static routing* simply refers to a situation where the routing tables are manually set up so that they remain static. In contrast, *dynamic routing* is the use of routing protocols to dynamically update routing tables. Thus, all routing protocols are dynamic. There is no such thing as a static routing protocol (unless you consider a network administrator who is editing a routing table a protocol).



# 38

## ROUTING INFORMATION PROTOCOL (RIP, RIP-2, AND RIPNG)



The most popular of the TCP/IP interior routing protocols is the *Routing Information Protocol (RIP)*. The simplicity of the name matches the simplicity of the protocol. Of all the routing protocols, RIP is one of the easiest to configure and least demanding of resources. Its popularity is due both to this simplicity and its long history. In fact, support for RIP has been built into operating systems for as long as TCP/IP itself has existed.

There are three versions of RIP: RIP versions 1 and 2 for IP version 4 (IPv4) and RIPng for IP version 6 (IPv6). The basic operation of the protocol is mostly the same for all three versions, but there are also some notable differences between them, especially in terms of the format of messages sent.

RIP was one of the first interior routing protocols used in TCP/IP. More than 20 years later, it continues to be widely used. Even though RIP has important limitations, it continues to have an important place in TCP/IP routing to this day. Evidence that RIP has a future can be seen in the creation of an IPv6 version of the protocol: RIPng.

I will open the examination of RIP with an overall description of its characteristics and how it works in general terms. I start with an overview and history of the protocol, including a brief discussion of its different versions and the standards that define them. I describe the method that RIP uses to determine routes and the metric used to assess route cost. I describe the general operation of the protocol including message types and when they are sent. I then describe the most important limitations and issues with RIP, and the special features that have been added to the protocol to resolve several problems with the basic RIP algorithm. Finally, I take a closer look at each version, showing the message format used for each and discussing version-specific features as well.

## RIP Overview

RIP has been the most popular interior routing protocol in the TCP/IP protocol suite for many years. The history of the protocol and how it came to achieve prominence is a rather interesting one. Unlike many of the other important protocols in the TCP/IP suite, RIP was not first developed formally using the RFC standardization process (see Chapter 3). Rather, it evolved as a de facto industry standard and became an Internet standard later.

The history of RIP has something in common with another networking heavyweight: Ethernet. Like that formidable local area network (LAN) technology, RIP's roots go back to that computing pioneer, Xerox's Palo Alto Research Center (PARC). At the same time that Ethernet was being developed for tying together LANs, PARC created a higher-layer protocol to run on Ethernet called the Xerox PARC Universal Protocol (PUP). PUP required a routing protocol, so Xerox created a protocol called the Gateway Information Protocol (GWINFO). This was later renamed the Routing Information Protocol and used as part of the Xerox Network System (XNS) protocol suite.

RIP entered the mainstream when developers at the University of California at Berkeley adapted it for use in the Berkeley Standard Distribution (BSD) of the UNIX operating system. RIP first appeared in BSD version 4.2 in 1982, where it was implemented as the UNIX program *routed* (pronounced “route-dee,” not “rout-ed”—the “d” stands for “daemon,” a common UNIX term for a server process).

BSD was (and still is) a very popular operating system, especially for machines connected to the early Internet. As a result, RIP was widely deployed and became the industry standard for internal routing protocols. It was used both for TCP/IP and other protocol suites. In fact, a number of other routing protocols, such as the RTP protocol in the AppleTalk suite, were based on this early version of RIP.

## RIP Standardization

For a while, the BSD implementation of *routed* was actually considered the standard for the protocol itself. However, this was not a formally defined standard, and this meant that there was no formal definition of exactly how it functioned. This led to slight differences in various implementations of the protocol over time. To resolve potential interoperability issues between implementations, the Internet Engineering Task Force (IETF) formally specified RIP in the Internet standard RFC 1058, “Routing Information Protocol,” which was published in June 1988. This

RFC was based directly on the BSD routed program. This original version of RIP is now also sometimes called RIP version 1 or RIP-1 to differentiate it from later versions.

RIP's popularity was due in large part to its inclusion in BSD, and it was included in BSD because of the relative simplicity of the protocol.

## ***RIP Operational Overview, Advantages, and Limitations***

RIP uses the distance-vector algorithm to determine routes, as described in Chapter 37. Each router maintains a routing table containing entries for various networks or hosts in the internetwork. Each entry contains two primary pieces of information: the address of the network or host and the distance to it, measured in hops, which is simply the number of routers that a datagram must pass through to get to its destination.

On a regular basis, each router in the internetwork sends out its routing table in a special message on each of the networks to which it is connected, using the User Datagram Protocol (UDP). Other routers receive these tables and use them to update their own tables. This is done by taking each of the routes they receive and adding an extra hop. For example, if Router A receives an indication from Router B that Network N1 is four hops away, since Router A and Router B are adjacent, the distance from Router A to Network N1 is five. After a router updates its tables, it sends out this information to other routers on its local networks. Over time, routing distance information for all networks propagates over the entire internetwork.

RIP is straightforward in operation, easy to implement, and undemanding of router processing power, which makes it especially attractive in smaller autonomous systems (ASes). There are, however, some important limitations that arise due to the simplicity of the protocol. For starters, hops are often not the best metric to use in selecting routes. There are also a number of problems that arise with the algorithm itself. These include slow convergence (delays in having all routers agree on the same routing information) and problems dealing with network link failures. RIP includes several special features to resolve some of these issues, but others are inherent limitations of the protocol. For example, RIP supports a maximum of only 15 hops between destinations, making it unsuitable for very large ASes, and this cannot be changed.

More than two decades after it was first created, RIP continues to be a popular interior routing protocol. Its limitations have led to many internetworking experts hoping that the protocol would eventually be replaced by newer protocols such as Open Shortest Path First (OSPF) that are superior on a strictly technical basis. Some have gone so far as to sarcastically suggest that maybe it would be best if RIP would R. I. P. Once a protocol becomes popular, however, it's hard to resist momentum, and RIP is likely to continue to be used for many years to come.

**KEY CONCEPT** The *Routing Information Protocol (RIP)* is one of the oldest and most popular interior routing protocols. With each router, it uses a distance-vector algorithm that maintains a table, which indicates how to reach various networks in the AS and the distance to it in hops. RIP is popular because it is well established and simple, but it has a number of important limitations.

## **Development of RIP Version 2 (RIP-2) and RIPng for IPv6**

Some other issues with RIP came about as a result of the protocol having been developed in the early 1980s, when TCP/IP was still in its infancy. Over time, as the use of TCP/IP protocols changed, RIP became outdated. In response, *RIP version 2*, or *RIP-2* was created in the early 1990s.

*RIP-2* defines a new message format for RIP and includes a number of new features, including support for classless addressing, authentication, and the use of multicasting instead of broadcasting, which improves network performance. It was first defined in RFC 1388, “RIP Version 2 Carrying Additional Information,” published in January 1993. This RFC was revised in RFC 1723 and finalized in RFC 2453, “RIP Version 2,” published in November 1998.

In order to ensure that RIP can work with TCP/IP in the future, it was necessary to define a version that would work with the IPv6. In 1997, RFC 2080 was published, titled “RIPng for IPv6.” The *ng* stands for *next generation*; you’ll recall that IPv6 is also sometimes called *IPng*.

*RIPng* is not just a new version of RIP, like *RIP-2*, but is defined as a new stand-alone protocol. It is, however, based closely on the original RIP and *RIP-2* standards. A distinct protocol (as opposed to a revision of the original) was needed due to the changes made between IPv4 and IPv6, though RIP and *RIPng* work in the same basic way. *RIPng* is sometimes also called *RIPv6*.

**KEY CONCEPT** The original version of RIP has the fewest features and is now called *RIP-1*. *RIP-2* was created to add support for classless addressing and other capabilities. *RIPng* is the version created for compatibility with IPv6.

## **RIP Route Determination Algorithm and Metric**

As I mentioned in the previous chapter, one of the defining characteristics of any routing protocol is the algorithm it uses for determining routes. RIP falls into the class of protocols that use a distance-vector, or Bellman-Ford, routing algorithm. To help you understand exactly how RIP determines routes, this section presents the specific implementation of the algorithm for RIP and provides an example.

Note that the description presented here is the basic algorithm used by RIP. This is modified in certain ways to address some of the problems that can occur in special circumstances due to how the algorithm works. Later in this chapter, we will explore these problems and the special features RIP includes to address them.

### ***RIP Routing Information and Route Distance Metric***

The job of RIP, like any routing protocol, is to provide a mechanism for exchanging information about routes so routers can keep their routing tables up-to-date. Each router in an RIP internetwork keeps track in its routing table of all networks (and possibly individual hosts) in the internetwork. For each network or host, the device includes a variety of information, of which the following is the most important:

- The address of the network or host
- The distance from that router to the network or host

- The first hop for the route: the device to which datagrams must first be sent to eventually get to the network or host

In theory, the distance metric can be any assessment of cost, but in RIP, distance is measured in hops. As you probably already know, in TCP/IP vernacular, a datagram makes a *hop* when it passes through a router. Thus, the RIP distance between a router and a network measures the number of routers that the datagram must pass through to get to the network. If a router connects to a network directly, then the distance is 1 hop. If it goes through a single router, the distance is 2 hops, and so on. In RIP, a maximum of 15 hops are allowed for any network or host. The value 16 is defined as infinity, so an entry with 16 in it means “this network or host is not reachable.”

### **RIP Route Determination Algorithm**

On a regular basis, each router running RIP will send out its routing table entries to provide information to other routers about the networks and hosts it knows how to reach. Any routers on the same network as the one sending out this information will be able to update their own tables based on the information they receive.

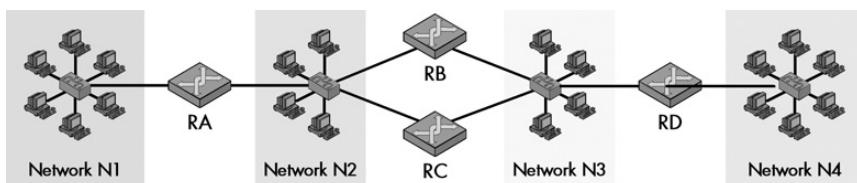
Any router that receives a message from another router on the same network saying it can reach Network X at a cost of  $N$  knows it can reach Network X at a cost of  $N+1$  by sending to the router it received the message from.

### **RIP Route Determination and Information Propagation**

Let’s take a specific example to help you understand how routes are determined and how route information is propagated using RIP. Consider a relatively simple internetwork with four individual networks, connected as follows:

- Router RA connects Network N1 to Network N2.
- Router RB and Router RC connect Network N2 to Network N3.
- Router RD connects Network N3 to Network N4.

This sample AS is illustrated in Figure 38-1.



**Figure 38-1: Sample RIP AS** This is an example of a simple AS that contains four physical networks and four routers.

Now let’s suppose that we just turned on Router RA. It sees that it is directly connected to Network N1 and Network N2, so it will have an entry in its routing table indicating that it can reach Network N1 at a cost of 1, which we can

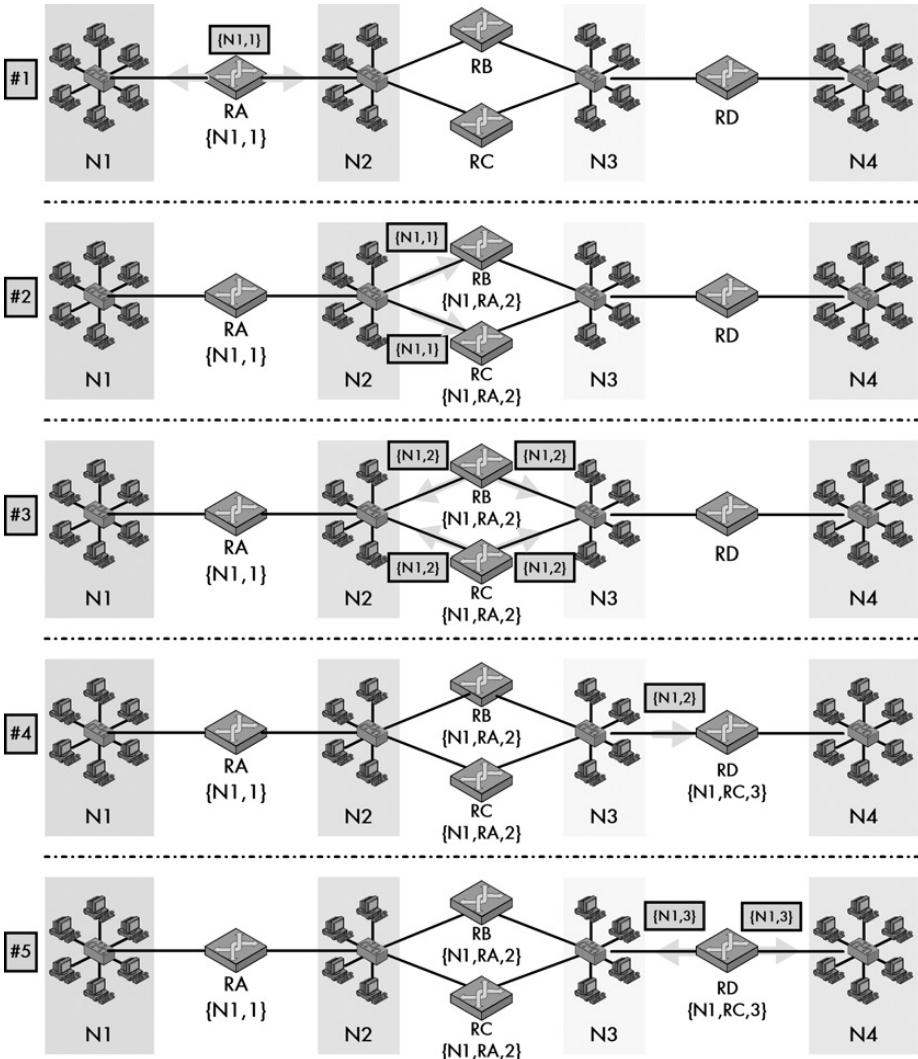
represent as {N1,1}. Information about Network N1 will propagate from Router RA across the internetwork in the following sequence of steps (which are illustrated in Figure 38-2):

1. Router RA sends out an RIP message containing the entry {N1,1} on each of the networks to which it is connected. There are no other routers on Network N1, so nothing happens there. But Routers RB and RC are on Network N2, so they receive the information.
2. Routers RB and RC will look in their routing tables to see if they already have entries for Network N1. Assuming neither does, they will each create a routing table entry {N1,2} for Router RA. This means, “I can reach Network N1 at a cost of 2 hops by sending to Router RA.”
3. Routers RB and RC will each send their own routing tables out over the networks to which they are connected: Networks N2 and N3. This will contain the entry {N1,2}. Router RA will receive that message on Network N2 but will ignore it, since it knows it can reach Network N1 directly (cost of 1, which is less than 2). But Router RD will receive the message on Network N3.
4. Router RD will examine its routing table, and seeing no entry for Network N1, it will add the entry {N1,3} for Routers RB or RC. Either one will work, so whichever is chosen depends entirely on whether Router RD received information about Network N1 first from Router RB or Router RC.
5. Router RD will send the entry {N1,3} on Network N4, but there are no other routers there to hear it.

Note that RIP is designed so that a routing entry is replaced only if information is received about a *shorter* route; ties go to the incumbent, if you will. This means that once Router RD creates an entry for Network N1 with a cost of 3 going through Router RB, if it receives information that it can reach Network N1 at the same cost of 3 through Router RC, it will ignore it. Similarly, if it gets Router RC’s information first, it will ignore the information from Router RB.

Naturally, this same propagation scheme will occur for all the other networks as well. I have shown only how information about Network N1 moves from router to router. For example, Router RA will eventually install an entry for Network N4 saying that it is reachable at a cost of 3 going through either Router RB or RC; this will be either {N4,RB,3} or {N4,RC,3}.

**KEY CONCEPT** Routing information is propagated between routers in RIP using a simple algorithm. On a regular basis, each router sends out RIP messages that specify which networks it can reach and how many hops it takes to reach them. Other routers directly connected to that one know that they can then reach those networks through that router at a cost of one additional hop. So if Router A sends a message saying it can reach Network X for a cost of  $N$  hops, every other router that connects directly to Router A can reach Network X for a cost of  $N+1$  hops. It will put that information into its routing table, unless it knows of an alternate route through another router that has a lower cost.



**Figure 38-2: Propagation of network routing information using RIP** This composite diagram illustrates the five steps in propagating route information about Network N1 from Router RA to the rest of the AS. In step 1, the information is sent from Router RA to both of its connected networks. In step 2, it reaches Routers RB and RC, which then know they can reach Network N1 through Router RA at a cost of one additional hop. In step 3, these two routers send this information on their networks, and in step 4, it reaches Router RD. In step 5, Router RD sends out the information, but no other routers are around to receive it.

This propagation of network routing information occurs on a regular basis, and also when the structure of the network changes (due to intentional changes in topography or failure of links or routers). When this happens, the change information will move through the internetwork so that all routers are eventually updated. For example, suppose a connection were added from Router RC to Network N1. If Router RD previously had the entry {N1, RB, 3}, it would eventually change this to {N1, RC, 2}, since it could now reach Network N1 more quickly by going through Router RC.

## **Default Routes**

In some cases, it is not convenient for every network or host in a large internetwork to be fully specified with its own routing entry. Then it may be advantageous to specify a default route for the network to use in reaching hosts or networks for which they have no information. The most common example of this is when an AS connects to the public Internet through a single router. Except for that router, the rest of the local network doesn't need to know how to access the Internet.

In RIP, information about a default route is communicated by having routers that are intended to handle such traffic send information about a “dummy” network with the address 0.0.0.0. This is treated as if it were a regular network when information about routes is propagated on the internetwork using RIP messages, but other devices recognize this special address and understand that it means a default route.

## **RIP General Operation, Messaging, and Timers**

RIP is a protocol for exchanging routing information, so its operation can best be described in terms of the messages used to exchange this information and the rules for when messages are sent. The RIP software in each router sends messages and takes other actions both in reaction to certain events and in response to triggers set off by timers. Timers are also used to determine when routing information should be discarded if it is not updated.

### **RIP Messages and Basic Message Types**

Communication between RIP software elements in routers on an internetwork is accomplished through the use of *RIP messages*. These messages are sent using the UDP, with the UDP port number 520 reserved for RIP-1 and RIP-2, and 521 for RIPng. Thus, even though RIP is considered part of layer 3 like other routing protocols, it behaves more like an application in terms of how it sends messages. The exact format of the message is version-dependent, and all three formats (RIP, RIP-2, and RIPng) are described in detail later in this chapter. RIP messages can be either sent to a specific device or sent out for multiple devices to receive. If directed to one device, they are sent unicast; otherwise, they are either broadcast (in RIP) or multicast (RIP-2 and RIPng).

There are only two basic message types for all three versions of RIP:

**RIP Request** A message sent by a router to another router asking it to send back all or part of its routing table.

**RIP Response** A message sent by a router containing all or part of its routing table. Note that despite the name, this message is *not* sent just in response to an RIP Request message, as you will see.

**NOTE** The original RIP also defined a few other message types: Traceon, Traceoff, and a special message type reserved for use by Sun Microsystems. These are obsolete and no longer used. They were removed from the RIP-2 and RIPng standards.

## **RIP Update Messaging and the 30-Second Timer**

RIP Request messages are sent under special circumstances when a router requires that it be provided with immediate routing information. The most common example of this is when a router is first powered on. After initializing, the router will typically send an RIP Request message on its attached networks to ask for the latest information about routes from any neighboring routers. It is also possible for RIP Request messages to be used for diagnostic purposes.

A router receiving an RIP Request message will process it and send an RIP Response message containing either all of its routing table or just the entries the Request message asked for, as appropriate. Under normal circumstances, however, routers do not usually send RIP Request messages asking specifically for routing information. Instead, each RIP router has a special timer that goes off every 30 seconds. (This timer is not given a specific name in the RIP standards; it is just the 30-second timer.)

Each time the timer expires, an unsolicited (unrequested) broadcast or multicast is made of an RIP Response message containing the router's entire routing table. The timer is then reset, and 30 seconds later, it goes off again, causing another routine RIP Response message to be sent. This process ensures that route information is regularly sent around the Internet, so routers are always kept up-to-date about routes.

**KEY CONCEPT** RIP uses two basic message types: the RIP Request and RIP Response. Both are sent using the User Datagram Protocol (UDP). RIP Response messages, despite their name, are used both for routine periodic routing table updates as well as to reply to RIP Request messages. Requests are sent only in special circumstances, such as when a router first joins a network.

## **Preventing Stale Information: The Timeout Timer**

When a router receives routing information and enters it into its routing table, that information cannot be considered valid indefinitely. In the example presented earlier in the “RIP Route Determination and Information Propagation” section, suppose that after Router RB installs a route to Network N1 through Router RA, the link between Router RA and Network N2 fails. Once this happens, Network N1 is no longer reachable from Router RB, but Router RB has a route indicating that it can reach Network N1.

To prevent this problem, routes are kept in the routing table for only a limited amount of time. A special Timeout timer is started whenever a route is installed in the routing table. Whenever the router receives another RIP Response message with information about that route, the route is considered refreshed, and its Timeout timer is reset. As long as the route continues to be refreshed, the timer will never expire.

If, however, RIP Response messages containing that route stop arriving, the timer will eventually expire. When this happens, the route is marked for deletion by setting the distance for the route to 16 (which you may recall is RIP infinity and

indicates an unreachable network). The default value for the Timeout timer is usually 180 seconds. This allows several periodic updates of a route to be missed before a router will conclude that the route is no longer reachable.

### ***Removing Stale Information: The Garbage-Collection Timer***

When a route is marked for deletion, a new Garbage-Collection timer is also started. *Garbage collection* is a computer-industry phrase for a task that looks for deleted or invalid information and cleans it up. Thus, this is a timer that counts a number of seconds before the newly invalid route will be actually removed from the table. The default value for this timer is 120 seconds.

The reason for using this two-stage removal method is to give the router that declared the route that's no longer reachable a chance to propagate this information to other routers. Until the Garbage-Collection timer expires, the router will include that route, with the unreachable metric of 16 hops, in its own RIP Response messages, so that the problem with that route is conveyed to the other routers. When the timer expires, the route is deleted. If during the garbage collection period a new RIP Response message for the route is received, then the deletion process is aborted. In this case, the Garbage-Collection timer is cleared, the route is marked as valid again, and a new Timeout timer starts.

### ***Triggered Updates***

In addition to the two situations already described where an RIP Response is sent—in reply to an RIP Request message and on expiration of the 30-second timer—an RIP Response message is also sent out when a route changes.

This action, an enhancement to a basic RIP operation, called a *triggered update*, is intended to ensure that information about route changes is propagated as fast as possible across the internetwork. This will help reduce the slow convergence problem in RIP. For example, in the case of a route timing out and the Garbage-Collection timer starting, a triggered update would be sent out about the now-invalid route immediately. This is described in more detail later in the chapter, in the “RIP Special Features for Resolving RIP Algorithm Problems” section.

## **RIP Problems and Some Resolutions**

The simplicity of RIP is often given as the main reason for its popularity. Simplicity is great most of the time, but an unfortunate price of simplicity in too many cases is that problems crop up, usually in unusual cases or special situations, and so it is with RIP. The straightforward distance-vector algorithm and operation mechanism work well most of the time, but they have some important weaknesses. We need to examine these problems to understand both the limitations of RIP and some of the complexities that have been added to the protocol to resolve them.

## **Issues with RIP's Algorithm**

The most important area where we find serious issues with RIP is with the basic function of the distance-vector algorithm described earlier in this section and the way that messages are used to implement it, as described in the following sections.

### **Slow Convergence**

The distance-vector algorithm is designed so that all routers share all their routing information regularly. Over time, all routers eventually end up with the same information about the location of networks and which are the best routes to use to reach them. This is called *convergence*. Unfortunately, the basic RIP algorithm is rather slow to achieve convergence. It takes a long time for all routers to get the same information, and in particular, it takes a long time for information about topology changes to propagate.

Consider the worst-case situation of two networks separated by 15 routers. Since routers normally send RIP Response messages only every 30 seconds, a change to one of this pair of networks might not be seen by the router nearest to the other one until many minutes have elapsed—an eternity in networking terms.

The slow convergence problem is even more pronounced when it comes to the propagation of route failures. Failure of a route is detected only through the expiration of the 180-second Timeout timer, so that adds up to three minutes more delay before convergence can even begin.

### **Routing Loops**

A routing loop occurs when Router A has an entry telling it to send datagrams for Network 1 to Router B, and Router B has an entry saying that datagrams for Network 1 should be sent to Router A. Larger loops can also exist: Router A says to send to B, which says to send to Router C, which says to send to Router A. Under normal circumstances, these loops should not occur, but they can happen in special situations.

RIP does not include any specific mechanism to detect or prevent routing loops. The best it can do is try to avoid them.

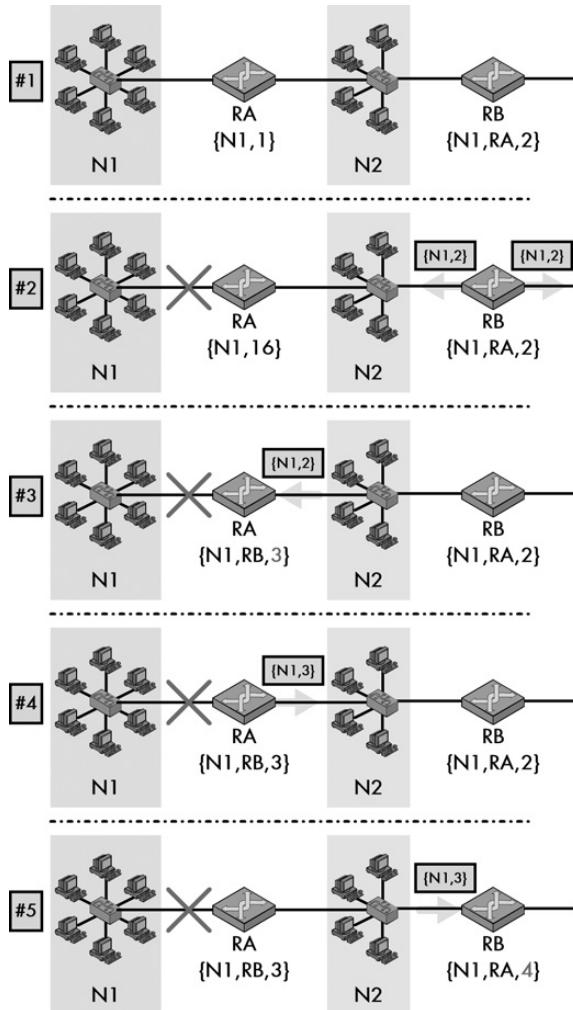
### **Counting to Infinity**

A special case of slow convergence can lead to a routing loop situation where one router passes bad information to another router, which sends more bad information to another router, and so on. This causes a situation where the protocol is sometimes described as unstable. The problem is called *counting to infinity*, for reasons you will soon see.

To understand how this happens, let's modify the example presented earlier in the “RIP Route Determination and Information Propagation” section, as shown in Figure 38-3. Suppose that the internetwork is operating properly for a while.

Router RB has an entry indicating it can reach Network N1 through Router RA at a cost of 2. But let's say the link from Network N1 to Router RA fails. After the Timeout timer for Network N1 expires on Router RA, that router will change the metric for Network N1 to 16 to indicate that it is unreachable. In the absence of any mechanism

to force Router RA to immediately inform other routers of this failure, those routers will not know about the change. Router RB will continue to think it can reach Network N1 through Router RA.



**Figure 38-3: The RIP counting to infinity problem** This composite diagram shows part of the AS illustrated previously in Figure 38-1. The top panel (1) shows the normal state of the network, with Router RB able to reach Network N1 through Router RA at a cost of 2. In panel 2, the link between Router RA and Network N1 is broken. Router RA changes its cost to reach Network N1 to 16 (RIP infinity). In panel 3, before Router RA can send out this update to Router RB, it receives a routine RIP message from Router RB indicating that Network N1 can be reached for a cost of 2. Router RA is then fooled into thinking that it can use Router RB as an alternate route to Network N1, even though Router RB's information originally came from Router RA in the first place. In panel 4, Router RA then sends this bogus information out, which is received by Router RB in panel 5. Router RB then increases its cost to 4, and on its next cycle will send this to Router RA, which will increase its cost to 5, and so on. This cycle will continue, with both routers "counting to infinity" (cost of 16).

Now suppose Router RB's regular 30-second timer goes off before Router RA's next broadcast. Router RB will send its normal routing table, *which contains a route to Network N1 at a cost of 2*. Router RA will see this and say, "Hey look, Router RB has a route to Network N1 with a cost of 2! That means I can get there with a cost of 3, which sure beats my current cost of 16. Let's use that!" So Router RA installs this route and cancels its Timeout timer. Of course, this is bogus information—Router RA doesn't realize that Router RB's claim of being able to reach Network N1 was based on old information from Router RA itself!

It only gets worse from there. When it is time for Router RA's regular routing table update, it will broadcast a route to Network N1 with a cost of 3. Now Router RB will see this and say, "Well, my route to Network N1 is through Router RA. Router RA was saying before that its cost was 1; but now it says the cost is 3. That means I have to change my cost to 4."

Router RB will later send back to Router RA, and back and forth they will go, each incrementing the cost two at a time. This won't stop until the value of infinity cost of 16 is hit—thus the name counting to infinity. At this point, both routers will finally agree that Network N1 is unreachable, but as you can see, it takes a long time for it to happen.

### **Small Infinity**

The use of a relatively small value for the infinity cost limits the slow convergence problem. Even in a situation where we count to infinity, the total amount of time elapsed is at least manageable. (Imagine if infinity were defined as say, 1,000!) Unfortunately, the drawback of this is that it limits the size of the internetwork that can be used for RIP.

Many people balk at the limit of a span of 15 routers in RIP, but to be honest I think it is much ado about, well, if not nothing, then nothing much. The 15 value is not a limit on the total number of routers you can use, but rather a limit on the number of routers between any two networks. Consider that most internetworks are set up hierarchically. Even if you have a rather complex four-level hierarchy, you wouldn't be close to the 15-router limit. In fact, you could create a huge AS with thousands of routers, without having more than 15 routers between any two devices. So this is a limitation for only the very largest of ASes.

On the other hand, RIP's need to send out its entire routing table many times each hour makes it a potentially poor choice for a large internetwork regardless of the infinity=16 issue. In an internetwork with many routers, the amount of traffic RIP generates can become excessive.

**KEY CONCEPT** One of the most important problems with the operation of RIP is slow convergence, which describes the fact that it can take a long time for information about changes to a network to propagate between routers. One specific instance of this problem is the counting to infinity problem, in which out-of-date information causes many bogus RIP messages to be exchanged between routers about an unreachable network.

To be fair, these problems are mostly general to distance-vector routing algorithms and not RIP in particular. Some of them are corrected through the implementation of specific changes to the algorithm or the rules under which RIP

messages are sent, as described in the next section. According to RFC 2453, there was actually a proposal to increase RIP's infinity cost to a number larger than 16, but this would have caused compatibility problems with older devices (which would view any route with a metric of 16 or higher as unreachable), so it was rejected.

### ***Issues with RIP's Metric***

In addition to these concerns with the algorithm itself, RIP is also often criticized because of its choice of metric. The first issue here is RIP's use of hop count as a distance metric. Simply put, hop count is a poor metric of the cost of sending a datagram between two networks. I believe the use of hop count as the metric in RIP is partially due to the desire for simplicity (it's easy to make the protocol work when hop count is all the routers need to consider). But the use of hop count is also partially an artifact of RIP being around for more than 20 years.

Decades ago, computers were slow, so each time a datagram passed through a router there was probably a significant delay. Hop count was not a perfect metric even then, but I think it had more correspondence with how long it took to move a datagram across an internetwork than it does today.

Modern routers are lightning fast, making hop count a flawed way of measuring network distance. The number of hops taken often has no correlation with the actual amount of time it takes to move data across a route. To take an extreme case, consider two networks that are connected by a direct dial-up telephone networking link using 56K modems. Let's say they are also connected by a sequence of three routers using high-speed DS-3 lines. RIP would consider the 56K link a better route because it has fewer hops, even though it clearly is much slower.

Another issue is RIP's lack of support for dynamic (real-time) metrics. Even if RIP were to use a more meaningful metric than hop count, the algorithm requires that the metric should be fixed for each link. There is no way to have RIP calculate the best route based on real-time data about various links the way protocols like OSPF do (see Chapter 39).

Most of these problems are built into RIP and cannot be resolved. Interestingly, some RIP implementations apparently do let administrators "fudge" certain routes to compensate for the limitations of the hop count metric. For example, the routers on either end of the 56K link mentioned earlier could be configured so that they considered the 56K link to have a hop count of ten instead of one. This would cause any routes using the link to be more expensive than the DS-3 path. This is clever, but hardly an elegant or general solution.

Note that in addition to the rather long list of problems that I've mentioned, there were also some specific issues with the first version of RIP. Some of the more important of these include lack of support for Classless Inter-Domain Routing (CIDR), lack of authentication, and a performance reduction caused by the use of broadcasts for messaging. These were mostly addressed through extensions in RIP-2.

### ***RIP Special Features for Resolving RIP Algorithm Problems***

The simplicity of RIP is its most attractive quality, but as you just saw, this leads to certain problems with how it operates. Most of these limitations are related to the basic algorithm used for determining routes, and the method of message passing

that's being used to implement the algorithm. In order for RIP to be a useful protocol, some of these issues needed to be addressed, in the form of changes to the basic RIP algorithm and operational scheme we explored earlier in this section.

The solution to problems that arise due to RIP being too simple is to add complexity in the form of features that add more intelligence to the way that RIP operates. In the following sections, we'll take a look at four of these: *split horizon*, *split horizon with poisoned reverse*, *triggered updates*, and *hold down*.

Note that while I describe these as "features," at least some of them are really necessary to ensure the proper RIP functionality. Therefore, they are generally considered standard parts of RIP, and most were described even in the earliest RIP documents. However, sometimes performance or stability issues may arise when these techniques are used, especially in combination. For this reason different RIP implementations may omit some features. For example, hold down slows down route recovery and may not be needed when other features such as split horizon are used. As always, care must be taken to ensure that all routers are using the same features, or even greater problems may arise.

Also, see the upcoming section on RIP-2's specific features, later in this chapter, for a description of the Next Hop feature, which helps reduce convergence and routing problems when RIP is used.

### **Split Horizon**

The counting to infinity problem is one of the most serious issues with the basic RIP algorithm. In the example in the previous section, the cause of the problem is immediately obvious: After Network N1 fails and Router RA notices it go down, Router RB "tricks" Router RA into thinking it has an alternate path to Network N1 by sending Router RA a route advertisement to Network N1.

If you think about it, it doesn't really make sense under *any* circumstances to have Router RB send an advertisement to Router RA about a network that Router RB can access only through Router RA in the first place. In the case where the route fails, it causes this problem, which is obviously a good reason not do it. But even when the route is operational, what is the point of Router RB telling Router RA about it? Router RA already has a shorter connection to the network and will therefore never send traffic intended for Network N1 to Router RB anyway.

Clearly, the best solution is simply to have Router RB not include any mention of the route to Network N1 in any RIP Response messages it sends to Router RA. We can generalize this by adding a new rule to RIP operation: When a router sends out an RIP Response message on any of the networks to which it is connected, it omits any route information that it originally learned from that network. This feature is called *split horizon*, because the router effectively splits its view of the internetwork, sending different information on certain links than on others.

With this new rule, let's consider the behavior of Router RB. It has an interface on Network N2, which it shares with Router RA. It will therefore not include any information on routes it originally obtained from Router RA when sending on Network N2. This will prevent the counting to infinity loop you saw in the previous section. Similarly, because Router RD is on Network N3, Router RB will not send any information about routes it got from Router RD when sending on Network N3.

Note, however, that split horizon may not always solve the counting to infinity problem, especially in the case where multiple routers are connected indirectly. The classic example would be three routers configured in a triangle. In this situation, problems may still result due to data that is propagated in two directions between any two routers. In this case, the hold down feature, described shortly, may be of assistance.

### **Split Horizon with Poisoned Reverse**

Adding “poisoned reverse” provides an enhancement of the basic split horizon feature. Instead of omitting routes learned from a particular interface when sending RIP Response messages on that interface, we include those routes but set their metric to RIP infinity, or 16. So in the previous example, Router RB *would* include the route to Network N1 in its transmissions on Network N2, but it would say the cost to reach Network N1 was 16 instead of its real cost (which is 2).

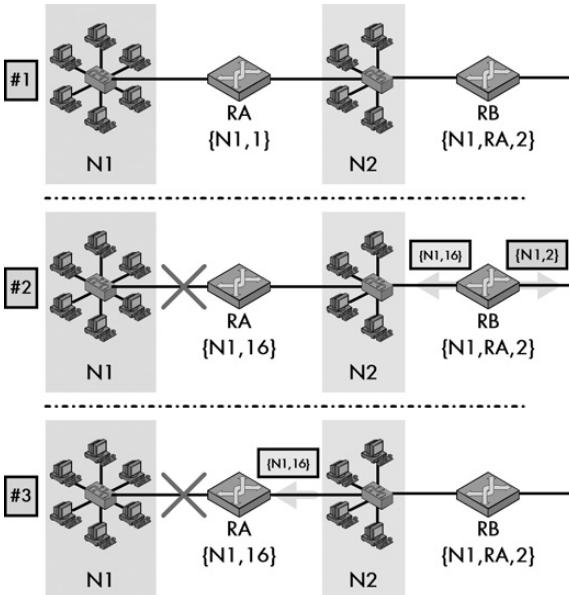
The *poisoned reverse* refers to the fact that we are poisoning the routes that we want to make sure routers on that interface don’t use. Router RA will see Router RB advertise Network N1 but with a cost of 16, which serves as an explicit message to Router RA: “There is absolutely no way for you to get to Network N1 through Router RB.” This provides more insurance than the regular split horizon feature, because if the link from Router RA to Network N1 is broken, Router RA will know for certain that it can’t try to get a new route through Router RB. Figure 38-4 shows how split horizon with poisoned reverse works.

This technique also works in normal circumstances (meaning when there is no issue such as a broken link to a network). In that case, Router RA will receive updates from Router RB with a cost of 16 on a periodic basis, but Router RA will never try to reach Network N1 through Router RB anyway, since it is directly connected to Network N1 (cost of 1).

### **Triggered Updates**

The routing loop problem we looked at earlier in this chapter occurred because Router RB advertised Router RA’s route back to Router RA. There’s another aspect of the problem that is also significant: After Router RA discovered that the link to Network N1 failed, it had to wait up to 30 seconds until its next scheduled transmission time to tell other routers about the failure.

For RIP to work well, when something significant happens, we want to tell other routers on the internetwork immediately. For this reason, a new rule should be added to the basic RIP router operation: Whenever a router changes the metric for a route it is required to (almost) immediately send out an RIP Response message to tell its immediate neighbor routers about the change. If these routers, seeing this change, update their routing information, they are in turn required to send out updates. Thus, the change of any network route information causes cascading updates to be sent throughout the internetwork, significantly reducing the slow convergence problem. Note that this includes the removal of a route due to expiration of its Timeout timer, since the first step in route removal is setting the route’s metric to 16, which triggers an update.



**Figure 38-4: RIP problem solving using split horizon with poisoned reverse** The top panel in this diagram (1) shows the same example as in Figure 38-3. In panel 2, as before, the link between Router RA and Network N1 is broken, just as Router RB is ready to send out its routine update. However, the split horizon with poisoned reverse feature means it sends different messages on its two links. On the network that connects it to Router RA, it sends a route advertisement with a cost of 16. In panel 3, Router RA receives this, which it will discard, ensuring no counting to infinity problem occurs. On Router RA's next cycle, it will update Router RB to tell it that Network N1 is no longer reachable.

You probably noticed that I said that triggered updates were sent “almost” immediately. In fact, before sending a triggered update a route waits a random amount of time, from 1 to 5 seconds. This is done to reduce the load on the internetwork that would result from many routers sending update messages nearly simultaneously.

### Hold Down

Split horizon tries to solve the counting to infinity problem by suppressing the transmission of invalid information about routes that fail. For extra insurance, we can implement a feature that changes how devices receiving route information process it in the case of a failed route. The *hold down* feature works by having each router start a timer when it first receives information about a network that is unreachable. Until the timer expires, the router will discard any subsequent route messages that indicate that the route is in fact reachable. A typical Hold Down timer runs for 60 or 120 seconds.

The main advantage of this technique is that a router won’t be confused by receiving spurious information about a route being accessible when it was just recently told that the route was no longer valid. It provides a period of time for out-of-date information to be flushed from the system, which is valuable especially on complex internetworks. Adding hold down to split horizon can also help in situations where split horizon alone is insufficient for preventing counting to infinity, such as when a trio of routers are linked in a triangle, as discussed earlier.

The main disadvantage of hold down is that it forces a delay in a router responding to a route once it is fixed. Suppose that a route went down for just five seconds for some reason. After the network is up again, routers will want to again know about this. However, the Hold Down timer must expire before the router will try to use that network again. This makes internetworks using hold down relatively slow to respond to corrected routes, and it could lead to delays in accessing networks that fail intermittently.

**KEY CONCEPT** Four special features represent changes to RIP operation that ameliorate or eliminate the problems with the operation of the basic protocol. *Split horizon* and *split horizon with poisoned reverse* prevent a router from sending invalid route information back to the router from which it originally learned the route. *Triggered updates* reduce the slow convergence problem by causing the immediate propagation of changed route information. Finally, the *hold down* feature may be used to provide robustness when information about a failed route is received.

## RIP Version-Specific Message Formats and Features

As I've noted, RIP has been in widespread use for more than two decades. During that time, internetworks and internetworking technologies have changed. To keep up with the times, RIP has also evolved and today has three different versions. The basic operation of all three is fairly similar, and it was described in the previous sections of this chapter. As you might expect, there are also some differences between the versions. One of the more important of these is the format used for RIP messages in each version, and the meaning and use of the fields within that format.

It's now time to take a look at the message format used by each of the three versions of RIP as well as certain specific features not common to all versions. I begin with the original RIP, now also known as *RIP version 1*. I then describe the updated version of RIP called *RIP version 2* or *RIP-2*. Finally, I discuss *RIPng*, also sometimes called *RIPv6*; it's the version of RIP used for IPv6. (Note that this is not technically a new version of the original RIP but a new protocol closely based on the earlier RIP versions.)

### RIP Version 1 (RIP-1) Message Format and Features

RIP evolved as an industry standard and was popularized by its inclusion in the Berkeley Standard Distribution of UNIX (BSD UNIX). This first version of RIP (now sometimes called RIP-1 to differentiate it from later versions) was eventually standardized in RFC 1058. As part of this standard, the original RIP-1 message format was defined, which of course serves *RIP-1* itself, and is also the basis for the format used in later versions.

## RIP-1 Messaging

As explained in the general discussion on RIP operation in the previous sections, route information is exchanged in RIP through the sending of two different types of RIP messages: RIP Request and the RIP Response. These are transmitted as regular TCP/IP messages using UDP, which uses the UDP reserved port number 520. This port number is used as follows:

- RIP Request messages are sent to UDP destination port 520. They may have a source port of 520 or may use an ephemeral port number (see Chapter 43 for an explanation of ephemeral ports).
- RIP Response messages sent in reply to an RIP Request are sent with a source port of 520 and a destination port equal to whatever source port the RIP Request used.
- Unsolicited RIP Response messages (sent on a routine basis and not in response to a request) are sent with both the source and destination ports, which are set to 520.

## RIP-1 Message Format

The basic message format for RIP-1 is described in Table 38-1 and illustrated Figure 38-5.

**Table 38-1:** RIP-1 Message Format

Field Name	Size (Bytes)	Description
Command	1	Command Type: Identifies the type of RIP message being sent. A value of 1 indicates an RIP Request, while 2 means an RIP Response. Originally, three other values and commands were also defined: 3 and 4 for the Traceon and Traceoff commands, and 5, which was reserved for use by Sun Microsystems. These are obsolete and no longer used.
Version	1	Version Number: Set to 1 for RIP version 1.
Must Be Zero	2	Field reserved; value must be set to all zeros.
RIP Entries	20 to 500, in increments of 20	The body of an RIP message consists of 1 to 25 sets of RIP entries. These entries contain the actual route information that the message is conveying. Each entry is 20 bytes long and has the subfields shown in Table 38-2.

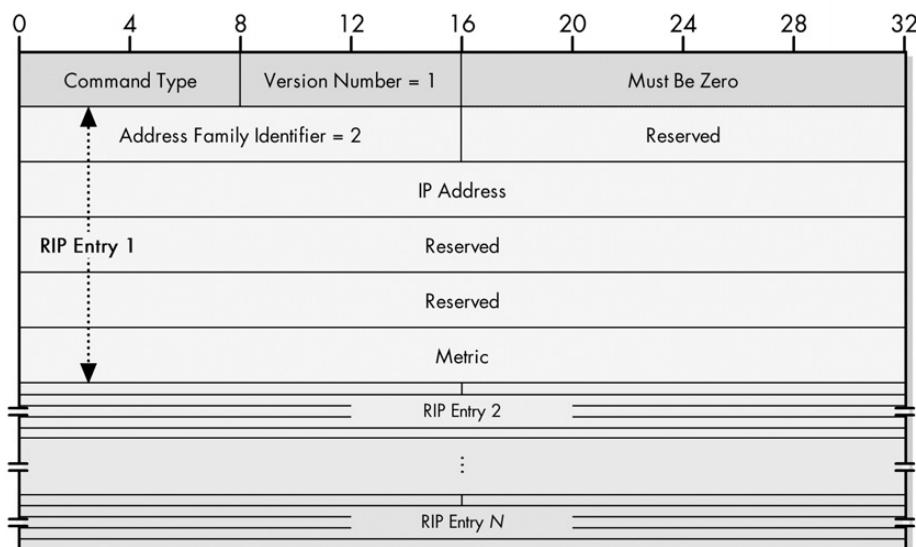
**Table 38-2:** RIP-1 RIP Entries

Subfield Name	Size (Bytes)	Description
Address Family Identifier	2	A fancy name for a field that identifies the type of address in the entry. The routers are using IP addresses, for which this field value is 2.
Must Be Zero	2	Field reserved; value must be set to all zeros.
IP Address	4	The address of the route the routers are sending information about. No distinction is made between addresses of different types of devices in RIP, so the address can be for a network, a subnet, or a single host. It is also possible to send an address of all zeros, which is interpreted as the default route for other devices on the network to use for reaching routes with no specified routing entries. This is commonly used to allow a network to access the Internet.

(continued)

**Table 38-2: RIP-1 RIP Entries (continued)**

Subfield Name	Size (Bytes)	Description
Must Be Zero	4	Field reserved; value must be set to all zeros.
Must Be Zero	4	Field reserved; value must be set to all zeros. (Yes, two of them in a row.)
Metric	4	The distance for the network indicated by the IP address in the IP Address field. Values of 1 to 15 indicate the number of hops to reach the network, while a value of 16 represents infinity (an unreachable destination). See the general discussion of the RIP algorithm earlier in this chapter for more information about the use of metrics.



**Figure 38-5: RIP-1 message format** The RIP-1 message format can contain up to 25 RIP entries. Here, RIP entry 1 is shown with each of its constituent subfields.

If you’re like me, the first thing that comes to mind after looking at this message format is this: What’s with all the extra space? I mean, we have four different fields that are reserved (must be zero), and even most of the other fields are larger than they need to be (a metric of 1 to 16 needs only 4 bits, not 32). The command type and version number could also easily have been made only 4 bits each, if not less. And why bother having a 2-byte field to identify the address type when we are only going to deal with IP addresses anyway?

This seeming wastefulness is actually an artifact of the generality of the original RIP design. The protocol was intended to be able to support routing for a variety of different internetworking protocols, not just Internet Protocol (IP). Remember that it wasn’t even originally developed with IP in mind. So, the Address Family Identifier was included to specify address type, and RIP entries were made large enough to handle large addresses. IP requires only 4 bytes per address, so some of the space is not used.

## RIP-1 Version-Specific Features

Since RIP-1 was the first version of the protocol, its features formed the basis for future RIP versions; it doesn't really have any version-specific features. What RIP-1 has is a number of limitations, such as a lack of support for specifying classless addresses and no means for authentication. RIP-2 was created to address some of RIP-1 shortcomings. As you will see in the next section, RIP-2's features put to good use those "Must Be Zero" bytes in the RIP-1 format!

**KEY CONCEPT** RIP-1 was the first version of RIP and is the simplest in terms of operation and features. The bulk of an RIP-1 message consists of sets of RIP entries that specify route addresses and the distance to the route in hops.

## RIP Version 2 (RIP-2) Message Format and Features

The original RIP (RIP-1) has a number of problems and limitations. As the TCP/IP protocol suite evolved and changed, RIP-1's problems were compounded by it becoming somewhat out of date. It was unable to handle newer IP features. There were some who felt that the existence of newer and better interior routing protocols meant that it would be best to just give up on RIP entirely and move over to something like OSPF.

However, RIP's appeal was never its technical superiority, but its simplicity and ubiquity in the industry. By the early 1990s, RIP was already in use in many thousands of networks. For those who liked RIP, it made more sense to migrate to a newer version that addressed some of RIP-1's shortcomings than to go to an entirely different protocol. To this end, a new version of the protocol, RIP-2 was developed. It was initially published in RFC 1388 in 1993. It is now defined in RFC 2453, "RIP Version 2," which was published in November 1998.

## RIP-2 Version-Specific Features

RIP-2 represents a very modest change to the basic RIP. RIP-2 works in the same basic way as RIP-1. In fact, the new features introduced in RIP-2 are described as *extensions* to the basic protocol, thereby conveying the fact that they are layered upon regular RIP-1 functionality. The five key RIP-2 extensions are as follows:

**Classless Addressing Support and Subnet Mask Specification** When RIP-1 was developed, the use of subnets in IP (as described in RFC 950) had not yet been formally defined. It was still possible to use RIP-1 with subnets through the use of a heuristic to determine if the destination is a network, subnet, or host. However, there was no way to clearly specify the subnet mask for an address using RIP-1 messages. RIP-2 adds explicit support for subnets by allowing a subnet mask within the route entry for each network address. It also provides support for Variable Length Subnet Masking (VLSM; see Chapter 18) and CIDR.

**Next Hop Specification** In RIP-2, each RIP entry includes a space where an explicit IP address can be entered as the next-hop router for datagrams that are intended for the network in that entry. This feature can help improve efficiency of routing by eliminating unnecessary extra hops for datagrams sent to certain destinations. One

common use of this field is when the most efficient route to a network is through a router that is not running RIP. Such a router will not exchange RIP messages and would therefore not normally be selected by RIP routers as a next hop for any network. The explicit Next Hop field allows the router to be selected as the next hop, regardless of this situation.

**Authentication** RIP-1 included no authentication mechanism, which is a problem because it could potentially allow a malicious host to attack an internetwork by sending bogus RIP messages. RIP-2 provides a basic authentication scheme that allows routers to ascertain the identity of a router before it will accept RIP messages from it.

**Route Tag** Each RIP-2 entry includes a Route Tag field where additional information about a route can be stored. This information is propagated along with other data about the route as RIP entries are sent around the internetwork. A common use of this field is when a route is learned from a different AS in order to identify the AS from which the route was obtained.

**Use of Multicasting** To help reduce network load, RIP-2 allows routers to be configured to use multicasts instead of broadcasts for sending out unsolicited RIP Response messages. These datagrams are sent out using the special reserved multicast address 224.0.0.9. All routers on an internetwork must use multicast if this is to work properly.

As you can see, many of these extensions require more information to be included with each advertised route. This is where all that extra space in the message format of RIP-1 routing entries comes in handy, as you will see shortly.

**KEY CONCEPT** RIP-2 is the most recent version of RIP used in IPv4. It includes a number of enhancements over the original RIP-1, including support for subnet masks and classless addressing, explicit next-hop specification, route tagging, authentication, and multicast. For compatibility, it uses the same basic message format as RIP-1, putting the extra information required for its new features into some of the unused fields of the RIP-1 message format.

## RIP-2 Messaging

RIP-2 messages are exchanged using the same basic mechanism as RIP-1 messages. Two different message types exist: RIP Request and RIP Response. They are sent using UDP, which uses the reserved port number 520. The semantics for the use of this port are the same as for RIP-1. For convenience, I'll repeat the description here:

- RIP Request messages are sent to UDP destination port 520. They may have a source port of 520 or may use an ephemeral port number.
- RIP Response messages sent in reply to an RIP Request message are sent with a source port of 520 and a destination port equal to whatever source port the RIP Request message used.
- Unsolicited RIP Response messages (sent on a routine basis and not in response to a request) are sent with both the source and destination ports set to 520.

## RIP-2 Message Format

The basic message format for RIP-2 is also pretty much the same as it was for RIP-1, with the Version field set to 2 in order to clearly identify the message as being RIP-2. Table 38-3 and Figure 38-6 illustrate the RIP-2 message format. The real differences are in the individual RIP entries, as you can see in Table 38-4.

**Table 38-3:** RIP-2 Message Format

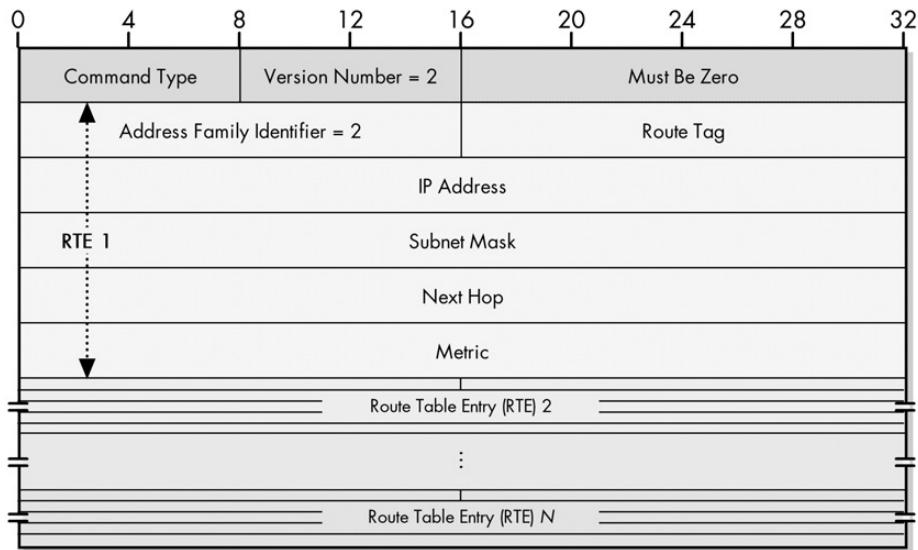
Field Name	Size (Bytes)	Description
Command	1	Command Type: Identifies the type of RIP message being sent. A value of 1 indicates an RIP Request, while 2 means an RIP Response.
Version	1	Version Number: Set to 2 for RIP version 2.
Must Be Zero	2	Field reserved; value must be set to all zeros.
Route Table Entries (RTEs)	20 to 500, in increments of 20	As with RIP-1, the body of an RIP-2 message consists of 1 to 25 sets of route information. In RIP-2 these are labeled Route Table Entries, or RTEs. Each RTE is 20 bytes long and has the subfields shown in Table 38-4.

**Table 38-4:** RIP-2 Route Table Entries (RTEs)

Subfield Name	Size (Bytes)	Description
Address Family Identifier	2	Same meaning as for RIP-1; value is 2 to identify IP addresses.
Route Tag	2	Additional information to be carried with this route.
IP Address	4	Same as in RIP-1: the address of the route the router is sending information about. No distinction is made between the address of different types of devices in RIP, so the address can be for a network, a subnet, or a single host. It is also possible to send an address of all zeros, which is interpreted as the default route, as in RIP-1.
Subnet Mask	4	The subnet mask associated with this address.
Next Hop	4	Address of the device to use as the next hop for the network advertised in this entry.
Metric	4	The distance for the network indicated by the IP address, as in RIP-1. Values of 1 to 15 indicate the number of hops to reach the network (as described in the discussion of the RIP algorithm earlier in this chapter), while a value of 16 represents infinity (an unreachable destination).

As you can see, the unused fields allow the new RIP-2 features to be implemented without changing the basic structure of the RIP entry format. This allows RIP-1 and RIP-2 messages and devices to coexist in the same network. An RIP-2 device can handle both RIP-1 and RIP-2 messages, and will look at the version number to see which version the message is. An RIP-1 device should handle both RIP-2 and RIP-1 messages the same way, simply ignoring the extra RIP-2 fields it doesn't understand.

**NOTE** If authentication is used, one of the RTEs contains authentication information, thus limiting the message to 24 "real" RTEs.



**Figure 38-6: RIP-2 message format** The RIP entries of RIP-1 are called Route Table Entries (RTEs) in RIP-2; the message format can contain up to 25. The format of RTE 1 is shown here with each of its subfields (the others are summarized to save space).

## RIPng (RIPv6) Message Format and Features

The future of TCP/IP is IPv6, which makes some very important changes to IP, especially with regard to addressing. Since IPv6 addresses are different than IPv4 addresses, everything that works with IP addresses must change to function under IPv6. This includes routing protocols, which exchange addressing information.

To ensure a future for the RIP, a new IPv6-compatible version had to be developed. This new version was published in 1997 in RFC 2080, *RIPng for IPv6*, where the *ng* stands for next generation (IPv6 is also sometimes called *IP next generation*).

RIPng, which is also occasionally seen as RIPv6 for obvious reasons, was designed to be as similar as possible to the current version of RIP for IPv4, which is RIP-2. In fact, RFC 2080 describes RIPng as the minimum change possible to RIP to allow it to work on IPv6. Despite this effort, it was not possible to define RIPng as just a new version of the older RIP, as RIP-2 was defined. RIPng is a new protocol, which was necessary because of the significance of the changes between IPv4 and IPv6—especially the change from 32-bit to 128-bit addresses in IPv6, which necessitated a new message format.

### RIPng Version-Specific Features

Even though RIPng is a new protocol, a specific effort was made to make RIPng like its predecessors. Its basic operation is almost entirely the same, and it uses the same overall algorithm and operation, as you saw earlier in this chapter. RIPng also does not introduce any specific new features compared to RIP-2, except those needed to implement RIP on IPv6.

RIPng maintains most of the enhancements introduced in RIP-2; some are implemented as they were in RIP-2, while others appear in a modified form. Here's specifically how the five extensions in RIP-2 are implemented in RIPng:

**Classless Addressing Support and Subnet Mask Specification** In IPv6 all, addresses are classless and specified using an address and a prefix length, instead of a subnet mask. Thus, a field for the prefix length is provided for each entry instead of a subnet mask field.

**Next Hop Specification** This feature is maintained in RIPng, but implemented differently. Due to the large size of IPv6 addresses, if you include a Next Hop field in the format of RIPng, the RTEs would almost double the size of every entry. Since Next Hop is an optional feature, this would be wasteful. Instead, when a Next Hop is needed, it is specified in a separate routing entry.

**Authentication** RIPng does not include its own authentication mechanism. It is assumed that if authentication and/or encryption are needed, they will be provided using the standard IPsec features, which are defined for IPv6 at the IP layer. This is more efficient than having individual protocols like RIPng perform authentication.

**Route Tag** This field is implemented in the same way as it is in RIP-2.

**Use of Multicasting** RIPng uses multicasts for transmissions, specifically the reserved IPv6 multicast address FF02::9.

### RIPng Messaging

There are two basic RIPng message types, RIP Request and RIP Response, which are exchanged using the UDP as with RIP-1 and RIP-2. Since RIPng is a new protocol, it cannot use the same UDP reserved port number 520, which is used for RIP-1/RIP-2. Instead, RIPng uses well-known port number 521. The semantics for the use of this port are the same as those used for port 520 in RIP-1 and RIP-2. For convenience, here are the rules again:

- RIP Request messages are sent to UDP destination port 521. They may have a source port of 521 or may use an ephemeral port number.
- RIP Response messages sent in reply to an RIP Request message are sent with a source port of 521 and a destination port equal to whatever source port the RIP Request message used.
- Unsolicited RIP Response messages (sent on a routine basis and not in response to a request) are sent with both the source and destination ports set to 521.

### RIPng Message Format

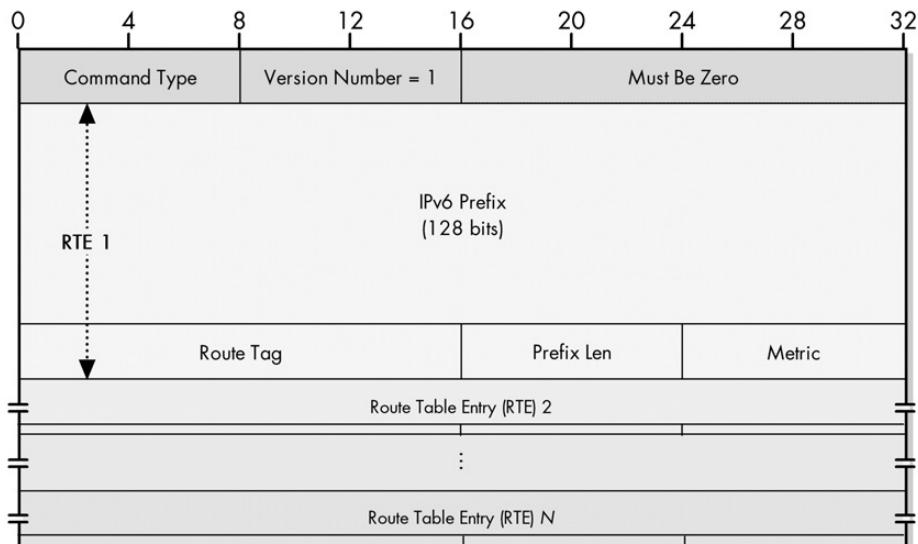
The message format for RIPng is similar to that of RIP-1 and RIP-2, except for the format of the RTEs. It is shown in Table 38-5 and illustrated in Figure 38-7.

**Table 38-5:** RIPng Message Format

Field Name	Size (Bytes)	Description
Command	1	Command Type: Identifies the type of RIPng message being sent. A value of 1 indicates an RIPng Request, while 2 means an RIPng Response.
Version	1	Version Number: Set to 1 (not 6, since this is the first version of the new protocol RIPng).
Must Be Zero	2	Field reserved; value must be set to all zeros.
Route Table Entries (RTEs)	Variable	The body of an RIPng message consists of a variable number of Route Table Entries (RTEs) that contain information about routes. Each entry is 20 bytes long and has the subfields shown in Table 38-6.

**Table 38-6:** RIPng RTEs

Subfield Name	Size (Bytes)	Description
IPv6 Prefix	16	The 128-bit IPv6 address of the network whose information is contained in this RTE.
Route Tag	2	Additional information to be carried with this route, as defined in RIP-2.
Prefix Len	1	The number of bits of the IPv6 address that is the network portion (the remainder being the host portion). This is the number that normally would appear after the slash when specifying an IPv6 network address. It is analogous to an IPv4 subnet mask. See the description of IPv6 prefix notation in Chapter 25 for more details.
Metric	1	The distance for the network indicated by the IP address, as in RIP-1. Values of 1 to 15 indicate the number of hops to reach the network (as described in the general discussion of the RIP algorithm earlier in this chapter) while a value of 16 represents infinity (an unreachable destination).



**Figure 38-7: RIPng message format** RIPng retains the use of RTEs from RIP-2, but their format has been changed to accommodate the much larger IPv6 address size. The limit of 25 entries per message has also been eliminated.

The maximum number of RTEs in RIPng is not restricted to 25 as it is in RIP-1 and RIP-2. It is limited only by the maximum transmission unit (MTU) of the network over which the message is being sent.

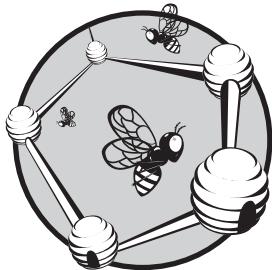
**KEY CONCEPT** RIPng is the version of RIP that was developed for use on IPv6 internetworks. It is technically a distinct protocol from RIP-1 and RIP-2, but is very similar to both. It retains the enhancements to RIP made in RIP-2, making changes to these features and to the RIP message format wherever needed for compatibility with IPv6.

When a Next Hop field needs to be specified, a special RTE is included, as I mentioned earlier. This RTE is included before all the RTEs to which it applies. It has the same basic structure as shown for regular RTEs in Table 38-6, with the IPv6 Prefix subfield containing the next hop address, the Route Tag and Prefix Len fields set to 0, and the Metric field set to 255 (0xFF).



# 39

## **OPEN SHORTEST PATH FIRST (OSPF)**



Interior routing protocols using a distance-vector routing algorithm, such as the Routing Information Protocol (RIP) we explored last chapter, have a long history and work well in a small group of routers. However, they also have some serious limitations in both scalability and performance that make them poorly suited to larger autonomous systems (ASes) or those with specific performance issues. Many organizations that start out using RIP quickly find that its restrictions and issues make it less than ideal.

To solve this problem, a new routing protocol was developed in the late 1980s. This protocol, called *Open Shortest Path First (OSPF)*, uses the more capable (and more complex) link-state or *shortest-path first* routing algorithm. It fixes many of the issues with RIP and allows routes to be selected dynamically based on the current state of the network, not just a static picture of how routers are connected. It also has numerous advanced features, including support for a hierarchical topology and automatic load sharing among

routes. On the downside, it is a complicated protocol, which means it is often not used unless it is really needed. This makes it the complement of RIP and is the reason they both have a place in the spectrum of TCP/IP routing protocols.

In this chapter, I provide a condensed explanation of the concepts and operation behind OSPF. As usual, I begin with an overview of the protocol, discussing how it was developed, its versions, and the standards that define them. I describe the concepts behind OSPF, including basic topology and the link-state database. I then discuss the more complex optional hierarchical topology of routers and the roles routers play when this topology is used. I briefly explain the method used for determining routes in OSPF, and the general operation and messaging used in the protocol, including a description of the five OSPF message types. I conclude with descriptions of the formats used for OSPF messages.

**NOTE** *The difficult thing about networking is that so many protocols and technologies are so involved that each deserves its own book. This is certainly the case with OSPF itself, which is sufficiently complex that the RFC defining OSPF version 2 is more than 240 pages long! Thus, this chapter, despite being fairly comprehensive, is only a high-level description of OSPF.*

## OSPF Overview

In the early days of TCP/IP, RIP became the standard protocol for routing within an autonomous system (AS), almost by default. RIP had two big things going for it: It was simple and easy to use, and it was included in the popular Berkeley Standard Distribution (BSD) of UNIX starting in 1982. Most organizations using TCP/IP started out with relatively small networks and were able to use RIP with some degree of success.

However, as I discussed in Chapter 38, that protocol has some serious technical issues, and they are exacerbated when RIP is used on a larger AS. Many of RIP's problems are due to it being a distance-vector protocol, because the algorithm itself simply limits the ability of RIP to choose the best route and adapt to changing network conditions. Other problems with RIP were based on its implementation, such as the selection of a cost value of 16 for infinity, which makes it impossible to use RIP in a situation where more than 15 hops might occur between devices. Problems such as the lack of classless addressing support were addressed in version 2 of RIP, but the basic difficulties with the protocol as a whole persist.

### ***Development and Standardization of OSPF***

The Internet Engineering Task Force (IETF) recognized that RIP by itself simply would not meet the needs of all ASes on the Internet. It formed a working group in 1988 to develop a new routing protocol based on the more capable link-state algorithm, also called shortest path first (SPF). Research into this type of protocol had already begun as early as the 1970s, with some of it conducted on the ARPAnet, the predecessor of the Internet, upon which much of TCP/IP was developed.

This new protocol's name conveys two of its most important characteristics. The first word refers to the fact that the protocol, like all TCP/IP standards, was developed using the open and public RFC process, so it is not proprietary, and no

license is required to use it. The SPF portion of the name refers to the type of algorithm it uses, which is designed to allow routers to dynamically determine the shortest path between any two networks.

The first version of OSPF was described in RFC 1131, which was published in October 1989. This was quickly replaced by OSPF version 2 in July 1991, which is described in RFC 1247. Since then, there have been several revisions to the OSPF version 2 standard, in RFCs 1583, 2178, and 2328, with the last of these now the current standard. OSPF version 2 is the only version in use today, so it is usually what is meant when people (including myself) refer to OSPF.

## **Overview of OSPF Operation**

The fundamental concept behind OSPF is a data structure called the *link-state database (LSDB)*. Each router in an AS maintains a copy of this database, which contains information in the form of a directed graph that describes the current state of the AS. Each link to a network or another router is represented by an entry in the database, and each has an associated cost (or metric). The metric can be made to include many different aspects of route performance, not just a simple hop count, as is used in RIP.

Information about the AS moves around the AS in the form of *link-state advertisements (LSAs)*, which are messages that let each router tell the others what it currently knows about the state of the AS. Over time, the information that each router has about the AS converges with that of the others, and they all have the same data. When changes occur to the internetwork, routers send updates to ensure that all the routers are kept up-to-date.

To determine actual routes, each router uses its LSDB to construct a shortest-path tree. This tree shows the links from the router to each other router and network and allows the lowest-cost route to any location to be determined. As new information about the state of the internetwork arrives, this tree can be recalculated, so the best route is dynamically adjusted based on network conditions. When more than one route with an equal cost exists, traffic can be shared among the routes.

## **OSPF Features and Drawbacks**

In addition to the obvious benefits of the link-state algorithm, OSPF includes several other features of value, especially to larger organizations. It supports authentication for security and all three major types of IP addressing (classful, subnetted classful, and classless). For very large ASes, OSPF also allows routers to be grouped and arranged into a hierarchical topology. This allows for better organization and improved performance through reduced LSA traffic.

Naturally, the superior functionality and many features of OSPF do not come without a cost. In this case, the primary cost is that of complexity. Where RIP is a simple and easy-to-use protocol, OSPF requires more work and more expertise to properly configure and maintain. This means that even though OSPF is widely considered better than RIP, technically, it's not for everyone. The obvious role for OSPF is as a routing protocol for larger or higher-performance ASes, leaving RIP to cover the smaller and simpler internetworks.

**KEY CONCEPT** Open Shortest Path First (OSPF) was developed in the late 1980s to provide a more capable interior routing protocol for larger or more complex ASes that were not being served well by RIP. It uses the dynamic shortest path first, or link-state, routing algorithm, with each router maintaining a database containing information about the state and topology of the internetwork. As changes to the internetwork occur, routers send out updated state information, which allows each router to dynamically calculate the best route to any network at any point in time. OSPF is a complement to RIP in that RIP is simple but limited, whereas OSPF is more capable but more complicated.

## OSPF Basic Topology and the Link-State Database (LSDB)

OSPF is designed to facilitate routing in both smaller and larger ASes. To this end, the protocol supports two topologies. When there is only a small number of routers, the entire AS is managed as a single entity. This doesn't have a specific name, but I call it *OSPF basic topology* to convey the simple nature of the topology and to contrast it with the hierarchical topology you will explore in the next section.

When OSPF basic topology is used, all the routers in the AS function as peers. Each router communicates routing information with each other one, and each maintains a copy of the key OSPF data structure: the LSDB, which is essentially a computerized representation of the topology of the AS. It is the method by which routers see the state of the links in the AS—thus the name *link-state database* (and for that matter, the name of the entire class of link-state algorithms of which OSPF is a part).

The LSDB is a bit hard to visualize, but is best viewed as a set of data that is equivalent to a graphical picture that shows the topology of an AS. In such a diagram, we typically show routers and networks as nodes, and connections between routers and networks as lines that connect them. The OSPF LSDB takes that information and puts it into a table to allow a router to maintain a virtual picture of all the connections between routers and networks in the AS.

The LSDB therefore indicates which routers can directly reach which other routers and which networks each router can reach. Furthermore, it stores for each of these links a *cost* to reach the network. This cost is an arbitrary metric that can be set up based on any criteria important to the administrator. OSPF is not restricted to the overly simple hop-count metric used in RIP.

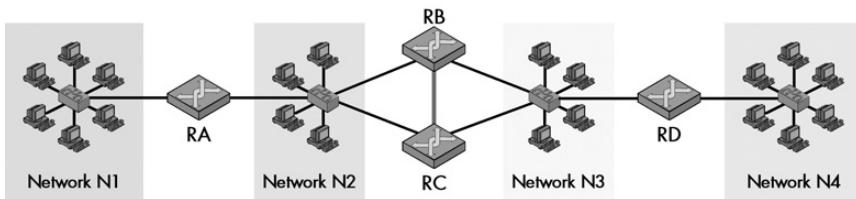
### OSPF Basic Topology

For example, let's consider the same AS that you looked at in the examination of the RIP route determination algorithm in Chapter 38. This internetwork has four individual networks, connected as follows:

- Router RA connects Network N1 to Network N2.
- Routers RB and RC connect Network N2 to Network N3.
- Router RD connects Network N3 to Network N4.

To make this example more interesting, I added a direct link between Routers RB and RC.

The resulting AS is shown in Figure 39-1. Table 39-1 shows what the LSDB for this AS might look like.



**Figure 39-1: Sample OSPF AS** This is the same AS that you looked at in RIP (as shown in Figure 38-1 in Chapter 38), but with the addition of a link between the two Routers RB and RC.

**Table 39-1:** Sample OSPF Link-State Database (LSDB)

To Router/ Network	From Router				From Network			
	RA	RB	RC	RD	N1	N2	N3	N4
RA					0	0		
RB			•		0	0		
RC		•			0	0		
RD						0	0	
N1	•							
N2	•	•	•					
N3		•	•	•				
N4				•				

In practice, each of the bullets (•) in Table 39-1 would be replaced by a metric value indicating the cost to send a datagram from the particular router to another router or network. Note that the chart is symmetric, because if Router RB can reach Router RC, Router RC can reach Router RB. However, the *costs* do not have to be symmetric. It is possible for Router RB to have a metric that is higher for it to send to Router RC than for Router RC to send to Router RB.

Note too that there is no cost to reach a router *from* a network. This ensures that only one cost is applied for a router to send to another router over a network. The cost is to reach the network from the router. This makes sense, since each router is a member of the network on which it is connected.

### LSDB Information Storage and Propagation

An important thing to remember about the LSDB is that even though each router maintains it, the database isn't constructed from the perspective of the individual router. A router's LSDB represents the topology of the entire AS, including links between routers that may be rather distant from it. So, for example, Router RA would keep the entire database in its storage area, including information about Router RC and Router RD, to which it does not connect directly.

Since in the basic topology, all the routers are peers and maintain information for the entire AS, in theory, they should have the exact same LSDB contents. When a router is first turned on, it may have different LSDB information than its neighbors, but this will be corrected through the exchange of update messages containing LSAs. Eventually, all routers should converge to the same information. You will see how this works in the section about OSPF messaging later in this chapter.

OSPF, as an interior routing protocol, is used only within the AS. In most cases, the AS will be connected to other ASes through one or more of its routers. The routers that connect the AS to other ASes are often called *boundary routers*. These devices will use OSPF to communicate within the AS, and an exterior routing protocol (typically BGP) to talk to routers outside the AS. The word *boundary* in its name refers to the fact that these devices are usually located on the periphery of the AS.

**KEY CONCEPT** In basic OSPF topology, each of the routers running OSPF is considered a peer of the others. Each maintains a *link-state database* (LSDB) that contains information about the topology of the entire AS. Each link between a router and network or between two routers is represented by an entry in the LSDB that indicates the cost to send data over the link. The LSDB is updated regularly through the exchange of OSPF *link-state advertisements* (LSAs).

## OSPF Hierarchical Topology

When the number of routers in an AS is relatively small, using the previously described basic topology works well. Each router maintains a common picture of the network topology in the form of an identical LSDB. The routers communicate as peers using LSAs. While changes in the AS may cause a router to temporarily have different information than its peers, routine exchanges of data will keep all the LSDBs synchronized and up-to-date, and not that much information needs to be sent around because the AS is small.

This simpler topology does scale reasonably well, and it can support many smaller and even moderate-sized ASes. However, as the number of routers increases, the amount of communication required to update LSDBs increases as well. In a very large internetwork with dozens or even hundreds of routers, having all the routers be OSPF peers using the basic topology can result in performance degradation. This problem occurs due to the amount of routing information that needs to be passed around and to the need for each router to maintain a large LSDB containing every router and network in the entire AS.

## OSPF Areas

To provide better support for these larger internetworks, OSPF supports the use of a more advanced, hierarchical topology. In this technique, the AS is no longer considered a single, flat structure of interconnected routers all of which are peers. Instead, a two-level hierarchical topology is constructed. The AS is divided into constructs called *areas*, each of which contains a number of contiguous routers and networks. These areas are numbered and managed independently by the routers

within them, so each area is almost as if it were an AS unto itself. The areas are interconnected so that routing information can be shared among areas across the entire AS.

The easiest way to understand this hierarchical topology is to consider each area like a sub-AS within the AS as a whole. The routers within any area maintain an LSDB that contains information about the routers and networks within that area. Routers within more than one area maintain LSDBs about each area that they are a part of, and they also link the areas together to share routing information between them.

**KEY CONCEPT** To allow for better control and management over larger internetworks, OSPF allows a large AS to be structured into a hierarchical form. Contiguous routers and networks are grouped into areas that connect together using a logical backbone. These areas act as the equivalent of smaller ASes within the larger AS, yielding the same benefits of localized control and traffic management that ASes provide for a large internetwork between organizations.

## **Router Roles in OSPF Hierarchical Topology**

The topology just described is hierarchical because the routers in the AS are no longer all peers in a single group. The two-level hierarchy consists of the lower level, which contains individual areas, and the higher level that connects them together, which is called the *backbone* and is designated as Area 0. The routers play different roles, depending on where they are located and how they are connected. There are three different labels applied to routers in this configuration:

**Internal Routers** These are routers that are connected only to other routers or networks within a single area. They maintain an LSDB for only that area and have no knowledge of the topology of other areas.

**Area Border Routers** These are routers that connect to routers or networks in more than one area. They maintain an LSDB for each area of which they are a part. They also participate in the backbone.

**Backbone Routers** These are routers that are a part of the OSPF backbone. By definition, these include all area border routers, since those routers pass routing information between areas. However, a backbone router may also be a router that connects only to other backbone (or area border) routers and is therefore not part of any area (other than Area 0).

To summarize, an *area border router* is also always a *backbone router*, but a backbone router is not necessarily an area border router.

**NOTE** The classifications that I just mentioned are independent of the designation of a router as being a boundary router, as described in the previous section. A boundary router is one that talks to routers or networks outside the AS. A boundary router will also often be an area border router or a backbone router, but this is not necessarily the case. A boundary router could be an internal router in one area.

The point of all this is the same as the point of using AS architecture in the first place. The topology of each area matters only to the devices in that area. This means that changes in that topology need to be propagated only within the area. It also means that internal routers within Area 1 don't need to know about anything that goes on within Area 2 and don't need to maintain information about any area other than their own. Only the backbone routers (which include at least one area border router within each area) need to know the details of the entire AS. These backbone routers condense information about the areas so that only a summary of each area's topology needs to be advertised on the backbone.

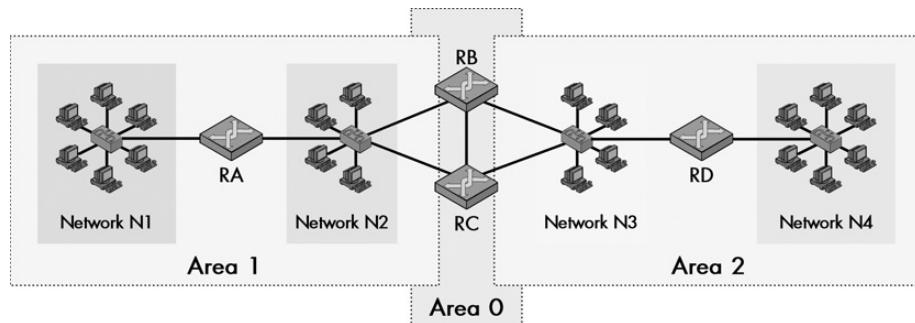
Routing in a hierarchical topology AS is performed in one of two ways, depending on the location of the devices:

- If the source and destination are in the same area, routing occurs only over networks and routers in that area.
- If the source and destination are in different areas, the datagram is routed from the source to an area border router in the source's area, over the backbone to an area border router in the destination's area, and then finally delivered to the destination.

Again, this is analogous to how routing works between ASes in the big-picture internetwork.

Let's take an example to help make things more concrete. We can use the AS in the preceding example. This AS is really small enough that it's unlikely we would use hierarchical topology, but it will suffice for sake of illustration. Let's divide this AS into two areas, as follows (see Figure 39-2):

- Area 1 contains Network N1, Router RA, Network N2, Router RB, and Router RC.
- Area 2 contains Router RB, Router RC, Network N3, Router RD, and Network N4.



**Figure 39-2: Sample OSPF hierarchical topology AS** This is the same AS you saw in Figure 39-1, but it's arranged into OSPF hierarchical topology. The AS has been split evenly into Area 1 and Area 2. Area 0 contains Routers RB and RC, which are area border routers for both Area 1 and Area 2 in this very simple example.

In this example, Router RA and Router RD are internal routers. Router RB and Router RC are area border routers that make up the backbone (Area 0) of the internetwork. Routers RA, RB, and RC will maintain an LSDB describing Area 1, while Routers RB, RC, and RD will maintain an LSDB describing Area 2. Routers

RB and RC maintain a separate LSDB for the backbone. There is no backbone router other than the area border routers RB and RC. However, suppose we had a Router RE that had only direct connections to Routers RB and RC. This would be a backbone router only.

This example has illustrated the chief drawback to hierarchical topology mentioned earlier in this chapter: complexity. For large ASes, however, it has significant advantages over making every router a peer. At the same time, the conceptual complexity is made worse by the need for very careful design, especially of the backbone. If the hierarchy is not set up properly, a single failure of a link between routers could disrupt the backbone and isolate one or more of the areas (including all the devices on all networks within the area!).

## OSPF Route Determination Using SPF Trees

The key data structure maintained by each router in an OSPF AS is the LSDB, which contains a representation of the topology of either the entire AS (in the basic topology) or a single area (in the hierarchical topology). As you have seen, each router in the AS or area has the same LSDB, so it represents a neutral view of the connections between routers and networks.

### ***The SPF Tree***

Each router needs to participate in keeping the LSDB up-to-date, but it also has its own concerns. It needs to be able to determine what routes it should use for datagrams it receives from its connected networks—this is, after all, the entire point of a routing protocol. To find the best route, it must determine the shortest path between itself and each router or network in the AS or area. For this, it needs not a neutral view of the internetwork but a view of it from its own perspective.

The router creates this perspective by taking the information in the LSDB and transforming it into an *SPF tree*. The term *tree* refers to a data structure with a root that has branches coming out that go to other nodes, which also have branches. The structure as a whole looks like an upside-down tree. In this case, the SPF tree shows the topology information of the AS or area with the router that constructs the tree at the top. Each directly connected router or network is one step down in the tree; each router or network connected to these first-level routers or networks is then connected, and so on, until the entire AS or area has been represented.

Again, the router doesn't really *make* the tree; it is just an algorithmic calculation performed by the computer within the router. Once this is done, however, this logical construct can be used to calculate the cost for that router to reach any router or network in the AS (or area). In some cases, there may be more than one way to reach a router or network, so the tree is constructed to show only the shortest (lowest-cost) path to the network.

Each router is responsible only for sending a datagram on the next leg of its journey, and not for what happens to the journey as a whole. After the SPF tree is created, the router will create a routing table with an entry for each network, showing the cost to reach it, and also the next-hop router to use to reach it.

The SPF tree is created dynamically based on the current state of the LSDB. If the LSDB ever changes, the SPF tree and the routing information are recalculated.

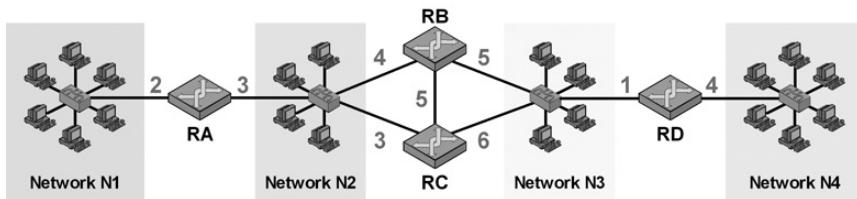
**KEY CONCEPT** To determine what routes it should use to reach networks in its AS, a router generates a *shortest-path first tree (SPF tree)* from its LSDB. This tree contains the same basic information as the LSDB, but presents it from the point of view of the router doing the calculation, so that router can see the costs of various paths to different networks.

## OSPF Route Determination

I can almost see your eyes glazing over, so let's go back to the example we have been using in this chapter. Let's assume that we are looking at the AS as a whole in basic topology, for simplicity. Table 39-2 repeats the LSDB for this AS shown earlier in Table 39-1, but I have taken the liberty of replacing the bullets with cost metrics; these are shown in Figure 39-3 as well. Again, remember that there is no cost to reach a router from a network, so those links have a nonzero cost only going from the router to the network.

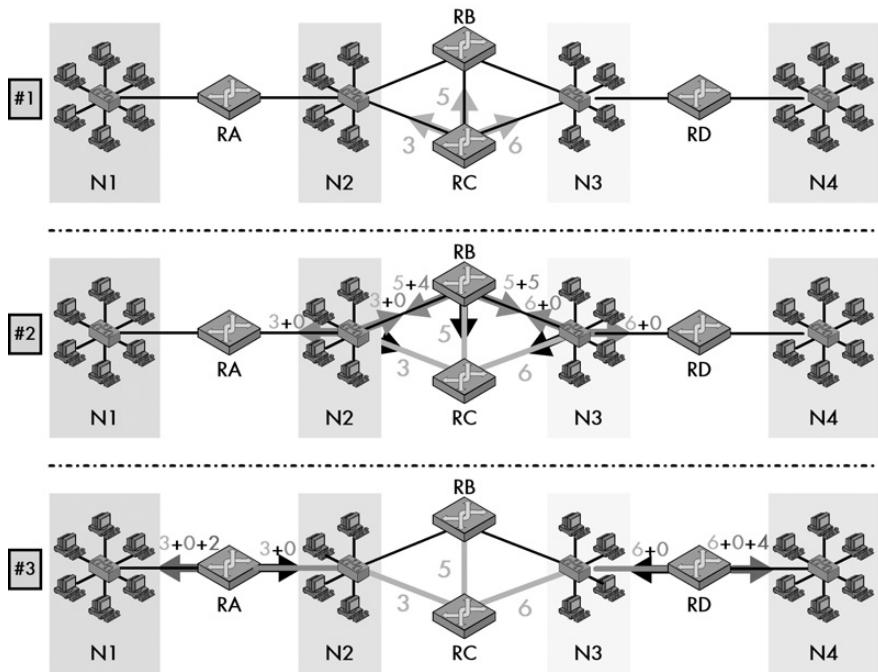
**Table 39-2:** Sample OSPF LSDB with Costs

To Router/ Network	From Router				From Network			
	RA	RB	RC	RD	N1	N2	N3	N4
RA					0	0		
RB			5		0	0		
RC		5			0	0		
RD						0	0	
N1	2							
N2	3	4	3					
N3		5	6	1				
N4				4				



**Figure 39-3: Sample OSPF AS with Costs** This is the same sample AS that is shown in Figure 39-1, but with costs assigned to each of the links between routers and networks. Costs between routers and networks are applied only in the direction from the router to the network.

Now let's construct the SPF tree for RC. We can do this in iterations, as follows (see Figure 39-4).



**Figure 39-4: OSPF route determination using the SPF algorithm** This diagram shows graphically how a router, in this case Router RC, determines the best path to various networks. The arrows here represent not the transfer of data, but rather the examination of various links from a router to other routers or networks. In panel 1, Router RC examines its LSDB and determines the cost for each of its directly linked devices. In panel 2, the second level of the SPF tree is constructed by adding to those numbers the costs of all routers/networks that connect to the routers/networks found in panel 1. (The black arrows represent looking back in the direction we came from in the prior step, which we don't pursue.) In panel 3 the process continues, resulting in the determination of a cost of 5 for Router RC to reach Network N1 and 10 to reach Network N4.

### First Level

To construct the first level of the tree, we look for all devices that Router RC can reach directly. We find the following:

- Router RB, with a cost of 5
- Network N2, with a cost of 3
- Network N3, with a cost of 6

### Second Level

To construct the second level, we look for all devices that the devices on the first level can reach directly. We then add the cost to reach each device on the first level to the cost of each device at the second level.

**RB:** Router RB has a cost of 5 and can reach the following:

- Router RC, with a cost of 5, total cost of 10
- Network N2, with a cost of 4, total cost of 9
- Network N3, with a cost of 5, total cost of 10

**N2:** Network N2 has a cost of 3 and can reach the following:

- Router RA, with a cost of 0, total cost of 3
- Router RB, with a cost of 0, total cost of 3
- Router RC, with a cost of 0, total cost of 3

**N3:** Network N3 has a cost of 6 and can reach the following:

- Router RB, with a cost of 0, total cost of 6
- Router RC, with a cost of 0, total cost of 6
- Router RD, with a cost of 0, total cost of 6

You probably can see immediately that we ended up with a number of different paths to the same devices or networks, some of which make no sense. For example, we don't really care about any path that goes to Router RC, since we *are* Router RC! Similarly, we can weed out certain paths immediately because we already have a shorter path to them. Taking a path through Router RB to Network N3 with a cost of 10 makes no sense when we can go directly at the first level for a cost of 6. So, after separating out the chaff, we end up with the following wheat at the second level:

- Network N2 to Router RA, with a cost of 3
- Network N3 to Router RD, with a cost of 6

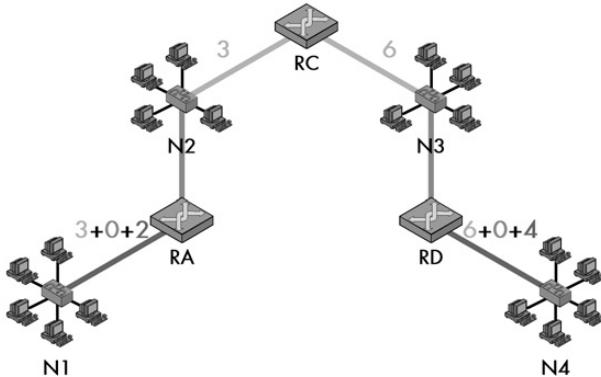
### Third Level

We continue the process by looking for devices that connect to the weeded-out devices that we found on the second level (this time I am only showing the meaningful ones):

**RA:** Router RA connects to Network N1, with a cost of 2, total cost of 5.

**RD:** Router RD connects to Network N4, with a cost of 4, total cost of 10.

In this simple example, we only need three levels to construct the tree for Router RC. (We would need more for Router RA or RD.) The final results would be the tree shown in Figure 39-5 and the routing information for RC to the four networks that is shown in Table 39-3.



**Figure 39-5: OSPF calculated SPF tree** This is a graphical representation of the SPF tree calculated in Figure 39-4, showing only the final results of the calculation process.

**Table 39-3:** Example of Calculated OSPF Routes

Destination Network	Cost	Next Hop
N1	5	RA
N2	3	(local)
N3	6	(local)
N4	10	RD

This is what you would expect in this very simple example. Note that there are no specific entries for other routers, since they are the means to the end of reaching networks. However, if one of the other routers were a boundary router that connected the AS to the outside world, there would be entries for the networks to which the boundary router connected, so Router RC knew to send traffic for those networks to that boundary router.

## OSPF General Operation

As a routing protocol, the main job of OSPF is to facilitate the exchange of routing information between routers. Each router in an OSPF AS that runs OSPF software is responsible for various tasks, such as setting timers to control certain activities that must occur on a regular basis, and the maintenance of important data structures, such as the LSDB. Most important, each OSPF router must both generate and respond to OSPF messages. It is this messaging system that allows important routing information to be shared within an AS or area, which makes it crucial to understanding how OSPF works. So it's worth starting a discussion of OSPF operation by taking a look at the message types and how they are used.

## **OSPF Message Types**

Unlike RIP, OSPF does not send its information using the User Datagram Protocol (UDP). Instead, OSPF forms IP datagrams directly, packaging them using protocol number 89 for the Internet Protocol (IP) Protocol field. OSPF defines five different message types, for various types of communication:

**Hello** As the name suggests, these messages are used as a form of greeting to allow a router to discover other adjacent routers on its local links and networks. The messages establish relationships between neighboring devices (called *adjacencies*) and communicate key parameters about how OSPF is to be used in the AS or area.

**Database Description** These messages contain descriptions of the topology of the AS or area; that is, they convey the contents of the LSDB for the AS or area from one router to another. Communicating a large LSDB may require several messages to be sent; this is done by designating the sending device as a master device and sending messages in sequence, with the slave (recipient of the LSDB information) responding with acknowledgments.

**Link State Request** These messages are used by one router to request updated information about a portion of the LSDB from another router. The message specifies the link(s) about which the requesting device wants more current information.

**Link State Update** These messages contain updated information about the state of certain links on the LSDB. They are sent in response to a Link State Request message, and they are also broadcast or multicast by routers on a regular basis. Their contents are used to update the information in the LSDBs of routers that receive them.

**Link State Acknowledgment** These messages provide reliability to the link-state exchange process by explicitly acknowledging receipt of a Link State Update message.

## **OSPF Messaging**

The use of these messages is approximately as follows. When a router first starts up it will send out a Hello message to see if any neighboring routers are around running OSPF, and it will also send them out periodically to discover any new neighbors that may show up. When an adjacency is set up with a new router, Database Description messages will then be sent to initialize the router's LSDB.

Routers that have been initialized enter a steady state mode. They will each routinely flood their local networks with Link State Update messages, advertising the state of their links. They will also send out updates when they detect a change in topology that needs to be communicated. They will receive Link State Update messages sent by other devices, and respond with Link State Acknowledgments accordingly. Routers may also request updates using Link State Request messages.

When the hierarchical topology is used, internal routers maintain a single LSDB and perform messaging only within an area. Area border routers have multiple LSDBs and perform messaging in more than one area. They, along with any other OSPF backbone routers, also exchange messaging information on the backbone, including summarized link-state information for the areas they border.

Again, all of this is highly simplified; the OSPF standard contains pages and pages of detailed rules and procedures governing the exact timing for sending and receiving messages.

**KEY CONCEPT** The operation of OSPF involves five message types. Hello messages establish contact between routers. Database Description messages initialize a router's LSDB. Routine LSDB updates are sent using Link State Update messages, which are acknowledged using Link State Acknowledgments. A device may also request a specific update using a Link State Request message.

## **OSPF Message Authentication**

The OSPF standard specifies that all OSPF messages are authenticated for security. This is a bit misleading, however, since one of the authentication methods supported is null authentication, meaning no authentication is used. More security is provided by using the optional simple password authentication method, and the most security is available through the use of cryptographic authentication. These methods are described in Appendix D of RFC 2328.

**NOTE** *The Hello messages used in OSPF are also sometimes called the Hello Protocol. This is especially poor terminology, because there is an actual routing protocol called the HELLO Protocol. The two protocols are not related. However, I suspect that the OSPF Hello messages may have been so named because they serve a similar purpose to the messages used in the independent HELLO Protocol.*

## **OSPF Message Formats**

As explained in the previous section, OSPF uses five different types of messages to communicate both link-state and general information between routers within an AS or area. To help illustrate how the OSPF messages are used, it's worth taking a look at the format of each of these messages.

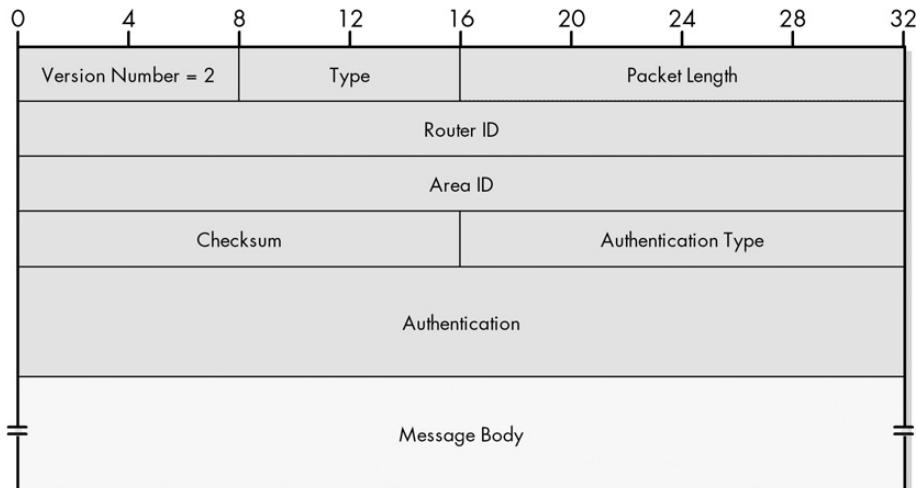
### **OSPF Common Header Format**

Naturally, each type of OSPF message includes a slightly different set of information; otherwise, there wouldn't be different message types. However, all message types share a similar message structure, beginning with a shared 24-byte header. This common header allows certain standard information to be conveyed in a consistent manner, such as the number of the version of OSPF that generated the message. It also allows a device receiving an OSPF message to quickly determine which type of

message it has received, so it knows whether or not it needs to bother examining the rest of the message. Table 39-4 and Figure 39-6 show the common OSPF header format.

**Table 39-4:** OSPF Common Header Format

Field Name	Size (Bytes)	Description
Version #	1	Set to 2 for OSPF version 2.
Type	1	Indicates the type of OSPF message: 1 = Hello 2 = Database Description 3 = Link State Request 4 = Link State Update 5 = Link State Acknowledgment
Packet Length	2	The length of the message, in bytes, including the 24 bytes of this header.
Router ID	4	The ID of the router that generated this message (generally its IP address on the interface over which the message was sent).
Area ID	4	An identification of the OSPF area to which this message belongs, when areas are used.
Checksum	2	A 16-bit checksum computed in a manner similar to a standard IP checksum. The entire message is included in the calculation except for the Authentication field.
AuType	2	Indicates the type of authentication used for this message: 0 = No Authentication 1 = Simple Password Authentication 2 = Cryptographic Authentication
Authentication	8	A 64-bit field used for authentication of the message, as needed.



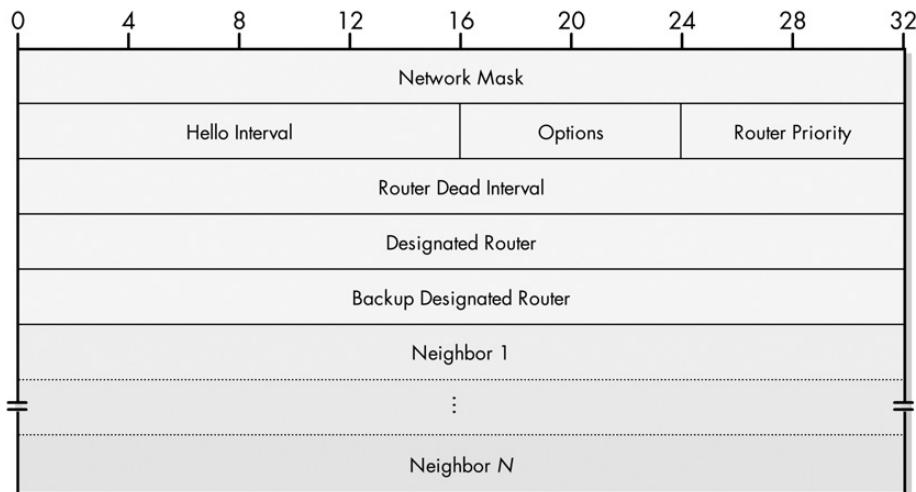
**Figure 39-6: OSPF common header format** Following this header, the body of the message includes a variable number of fields that depend on the message type. Each of the message formats is described in detail in RFC 2328. Since some are quite long, I will describe their fields only briefly here.

## **OSPF Hello Message Format**

Hello messages have a Type value of 1 in the header, and the field structure shown in Table 39-5 and Figure 39-7 in the body of the message.

**Table 39-5:** OSPF Hello Message Format

Field Name	Size (Bytes)	Description
Network Mask	4	The subnet mask of the network the router is sending to.
Hello Interval	2	The number of seconds this router waits between sending Hello messages.
Options	1	Indicates which optional OSPF capabilities the router supports.
Rtr Pri	1	Indicates the router's priority, when electing a backup designated router.
Router Dead Interval	4	The number of seconds a router can be silent before it is considered to have failed.
Designated Router	4	The address of a router designated for certain special functions on some networks. Set to zeros if there is no designated router.
Backup Designated Router	4	The address of a backup designated router. Set to all zeros if there is no backup designated router.
Neighbors	Multiple of 4	The addresses of each router from which this router has received Hello messages recently.



**Figure 39-7:** OSPF Hello message format

## **OSPF Database Description Message Format**

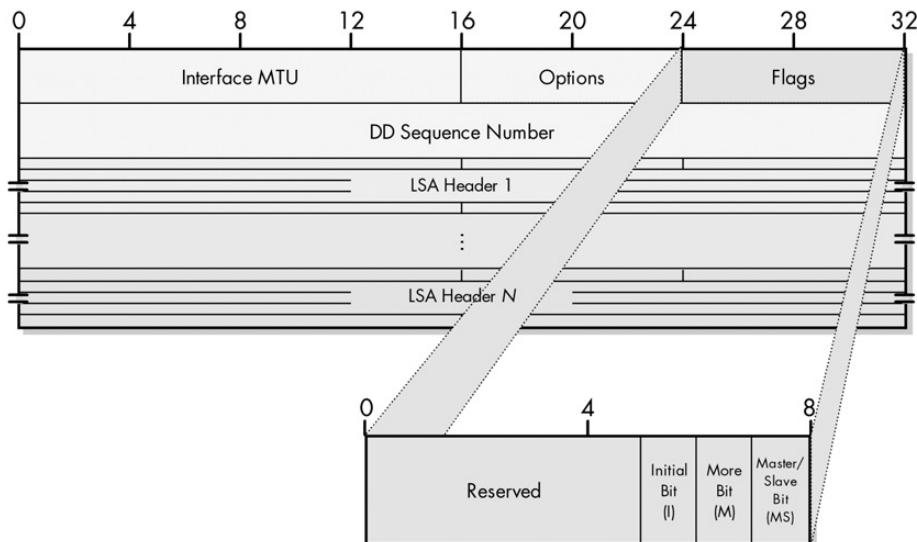
Database Description messages have a Type value of 2 in the header and the body structure depicted in Table 39-6 and Figure 39-8.

**Table 39-6:** OSPF Database Description Message Format

Field Name	Size (Bytes)	Description
Interface MTU	2	The size of the largest IP message that can be sent on this router's interface without fragmentation.
Options	1	Indicates which of several optional OSPF capabilities the router supports.
Flags	1	Special flags used to indicate information about the exchange of Database Description messages, as shown in Table 39-7.
DD Sequence Number	4	Used to number a sequence of Database Description messages so that they are kept in order.
LSA Headers	Variable	Contains LSA headers, which carry information about the LSDB. See the "OSPF Link State Advertisements and the LSA Header Format" section later in this chapter for more information about LSAs. Please add correct cross-ref info.

**Table 39-7:** OSPF Database Description Message Flags

Subfield Name	Size (Bytes)	Description
Reserved	5/8 (5 bits)	Reserved: Sent and received as zero.
I	1/8 (1 bit)	I-Bit: Set to 1 to indicate that this is the first (initial) in a sequence of Database Description messages.
M	1/8 (1 bit)	M-Bit: Set to 1 to indicate that more Database Description messages follow this one.
MS	1/8 (1 bit)	MS-Bit: Set to 1 if the router sending this message is the master in the communication, or 0 if it is the slave.



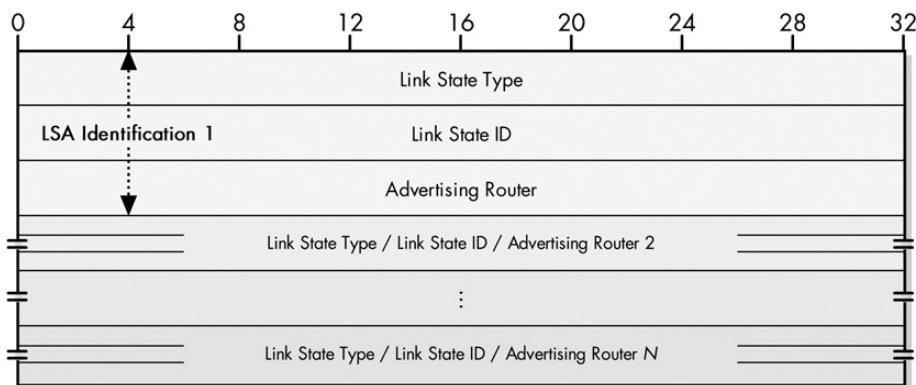
**Figure 39-8:** OSPF Database Description message format

## **OSPF Link State Request Message Format**

Link State Request messages have a Type value of 3 in the header. Following the header comes one or more sets of three fields, each of which identify an LSA for which the router is requesting an update, as shown in Figure 39-9. Each LSA identification has the format described in Table 39-8.

**Table 39-8:** OSPF Link State Request Message Format

Field Name	Size (Bytes)	Description
LS Type	4	The type of LSA being sought.
Link State ID	4	The identifier of the LSA, usually the IP address of either the router or network linked.
Advertising Router	4	The ID of the router that created the LSA whose update is being sought.



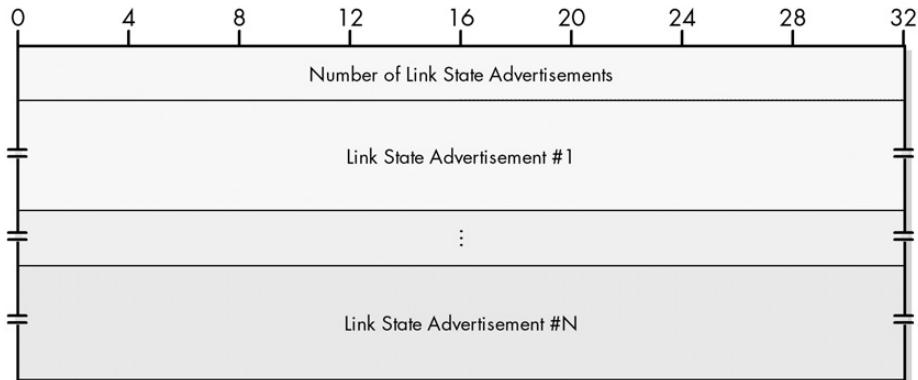
**Figure 39-9:** OSPF Link State Request Message format

## **OSPF Link State Update Message Format**

Link State Update messages have a Type value of 4 in the header and the fields illustrated in Table 39-9 and Figure 39-10.

**Table 39-9:** OSPF Link State Update Message Format

Field Name	Size (Bytes)	Description
# LSAs	4	The number of LSAs included in this message.
LSAs	Variable	One or more LSAs. See the "OSPF Link State Advertisements and the LSA Header Format" section later in this chapter for more details.



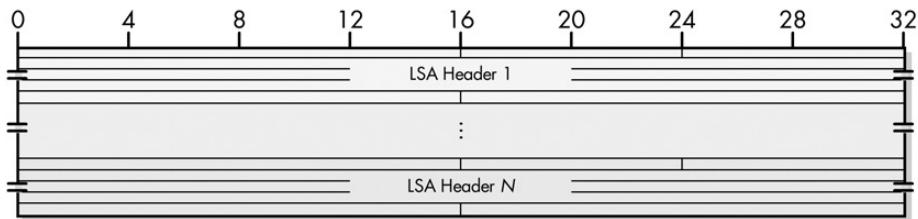
**Figure 39-10:** OSPF Link State Update message format

### **OSPF Link State Acknowledgment Message Format**

Link State Acknowledgment messages have a Type value of 5 in the header. They then contain a list of LSA headers corresponding to the LSAs being acknowledged, as shown in Table 39-10 and Figure 39-11.

**Table 39-10:** OSPF Link State Acknowledgment Message Format

Field Name	Size (Bytes)	Description
LSA Headers	Variable	Contains LSA headers that identify the LSAs acknowledged.



**Figure 39-11:** OSPF Link State Acknowledgment message format

### **OSPF Link State Advertisements and the LSA Header Format**

Several of the previous message types include LSAs, which are the fields that actually carry topological information about the LSDB. There are several types of LSAs, which are used to convey information about different types of links. Like the OSPF messages themselves, each LSA has a common header with 20 bytes and then a number of additional fields that describe the link.

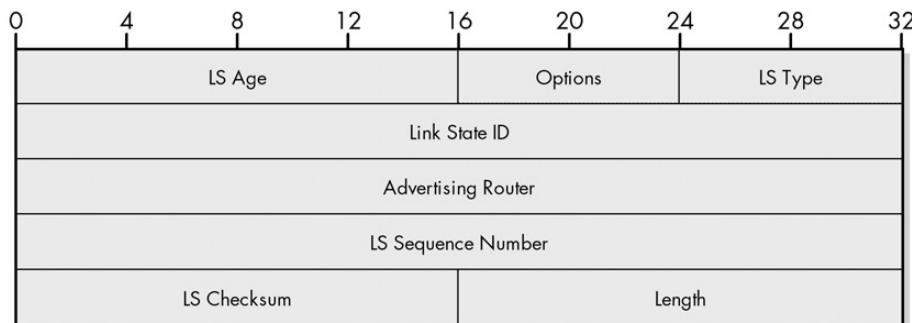
The LSA header contains sufficient information to identify the link. It uses the subfield structure shown in Table 39-11 and Figure 39-12.

**Table 39-11:** OSPF Link State Advertisement Header Format

Subfield Name	Size (Bytes)	Description
LS Age	2	The number of seconds elapsed since the LSA was created.
Options	1	Indicates which of several optional OSPF capabilities the router supports.
LS Type	1	Indicates the type of link this LSA describes, as shown in Table 39-12.
Link State ID	4	Identifies the link. This usually is the IP address of either the router or the network the link represents.
Advertising Router	4	The ID of the router originating the LSA.
LS Sequence Number	4	A sequence number used to detect old or duplicate LSAs.
LS Checksum	2	A checksum of the LSA for data corruption protection.
Length	2	The length of the LSA, including the 20 bytes of the header.

**Table 39-12:** OSPF Link State Advertisement Header LS Types

Value	Link Type	Description
1	Router-LSA	Link to a router.
2	Network-LSA	Link to a network.
3	Summary-LSA (IP Network)	When areas are used, summary information is generated about a network.
4	Summary-LSA (ASBR)	When areas are used, summary information is generated about a link to an AS boundary router.
5	AS-External-LSA	An external link outside the AS.

**Figure 39-12:** OSPF Link State Advertisement header format

Following the LSA header comes the body of the LSA. The specific fields in the body depend on the value of the LS Type field (see Table 39-12). Here is a summary:

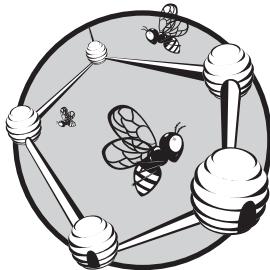
- For normal links to a router, the LSA includes an identification of the router and the metric to reach it, as well as details about the router such as whether it is a boundary or area border router.

- LSAs for networks include a subnet mask and information about other routers on the network.
- Summary LSAs include a metric and a summarized address as well as a subnet mask.
- External LSAs include a number of additional fields that allow the external router to be communicated.

Refer to Appendix A of RFC 2328 if you want all the details about the fields in the LSA body.

# 40

## **BORDER GATEWAY PROTOCOL (BGP/BGP-4)**



Modern TCP/IP internetworks are composed of autonomous systems (ASes) that are run independently. Each may use an interior routing protocol such as Routing Information Protocol (RIP), Open Shortest Path First (OSPF), Interior Gateway Routing Protocol (IGRP), or Enhanced Interior Gateway Routing Protocol (EIGRP) to select routes between networks within the AS. To form larger internetworks, and especially the “mother of all internetworks,” the Internet, these ASes must be connected together. This requires use of a consistent exterior routing protocol that all ASes can agree upon, and in today’s TCP/IP, that protocol is the *Border Gateway Protocol (BGP)*.

If you were to ask the average Internet user, or even the typical network administrator, to make a list of the ten most important TCP/IP protocols, BGP probably wouldn’t show up frequently. Routing protocols are worker bees of the TCP/IP protocol suite, and they just are not very exciting. The reality, however, is that BGP is a critically important protocol to the operation of larger internetworks and the Internet itself. It is the glue that binds

smaller internetworks (ASes) together, and it ensures that every organization is able to share routing information. It is this function that lets us take disparate networks and internetworks and find efficient routes from any host to any other host, regardless of location.

In this chapter, I describe the characteristics, general operation, and detailed operation of BGP. I start, as usual, with an overview of the protocol and discuss its history, standards, and versions, including a discussion of its key features and characteristics. I then cover basic operational concepts, including topology, the notion of BGP speakers, and neighbor relationships. I discuss BGP traffic types and how policies can be used to control traffic flows on the internetwork. I explain how BGP routers store and advertise routes and how Routing Information Bases function. I describe the basic algorithm used by BGP and how path attributes describe routes. I provide a summary of how the BGP route selection process operates. I then give a general description of BGP's operation and its high-level use of various messages. Finally, I present a more detailed analysis of the different message types, how they are used, and their format.

BGP is another in the rather large group of protocols and technologies that is so complex it would take dozens of chapters to do justice. Therefore, I include here my somewhat standard disclaimer that you will find in this chapter only a relatively high-level look at BGP. You will need to refer to the BGP standards (described in the section on BGP standards and versions) if you need more details.

**NOTE** *The current version of BGP is version 4, also called BGP-4. This is the only version widely used today, so unless otherwise indicated, assume that I'm talking about BGP-4 wherever you see BGP.*

## BGP Overview

As I described briefly in the overview of routing protocol concepts in Chapter 37, the way that routers were connected in the early Internet was quite different than it is today. The early Internet had a set of centralized routers functioning as a core AS. These routers used the Gateway-to-Gateway Protocol (GGP) for communication between them within the AS and the aptly named Exterior Gateway Protocol (EGP) to talk to routers outside the core. GGP and EGP are discussed in Chapter 41.

When the Internet grew and moved to AS architecture, EGP was still able to function as the exterior routing protocol for the Internet. However, as the number of ASes in an internetwork grew, the importance of communication between them grew as well. EGP was functional but had several weaknesses that became more problematic as the Internet expanded. It was necessary to define a new exterior routing protocol that would provide enhanced capabilities for use on the growing Internet.

In June 1989, the first version of this new routing protocol was formalized, with the publishing of RFC 1105, “A Border Gateway Protocol (BGP).” This initial version of the BGP standard defined most of the concepts behind the protocol, as well as key fundamentals such as messaging, message formats, and how devices operate in general terms. It established BGP as the Internet’s exterior routing protocol of the future.

## BGP Versions and Defining Standards

Due to the importance of a protocol that spans the Internet, work continued on BGP for many years after the initial standard was published. The developers of BGP needed to correct problems with the initial protocol, refine BGP's operation, improve efficiency, and add features. It was also necessary to make adjustments to allow BGP to keep pace with other changes in the TCP/IP protocol suite, such as the invention of classless addressing and routing.

The result of this ongoing work is that BGP has evolved through several versions and standards. These are sometimes called BGP-*N*, where *N* is the version number. Table 40-1 shows the history of BGP standards, providing the RFC numbers and names, and a brief summary of the changes made in each version.

**Table 40-1:** Border Gateway Protocol (BGP) Versions and Defining Standards

RFC Number	Date	Name	BGP Version	Description
1105	June 1989	A Border Gateway Protocol (BGP)	BGP-1	Initial definition of the BGP.
1163	June 1990	A Border Gateway Protocol (BGP)	BGP-2	This version cleaned up several issues with BGP-1 and refined the meaning and use of several of the message types. It also added the important concept of path attributes, which communicate information about routes. BGP-1 was designed around the notion of a directional topology, with certain routers being up, down, or horizontal relative to each other; BGP-2 removed this concept, making BGP better suited to an arbitrary AS topology. (Note that the RFC title is not a typo; they didn't put "version 2" in the title.)
1267	October 1991	Border Gateway Protocol 3 (BGP-3)	BGP-3	This version optimized and simplified route information exchange, adding an identification capability to the messages used to establish BGP communications, and incorporating several other improvements and corrections. (They left the "A" off the title of this one for some reason.)
1654	July 1994	A Border Gateway Protocol 4 (BGP-4)	BGP-4	Initial standard for BGP-4, revised in RFC 1771.
1771	March 1995	A Border Gateway Protocol 4 (BGP-4)	BGP-4	Current standard for BGP-4. The primary change in BGP-4 is support for Classless Inter-Domain Routing (CIDR). The protocol was changed to allow prefixes to be specified that represent a set of aggregated networks. Other minor improvements were also made to the protocol.

As you might imagine, changing the version of a protocol like BGP is not an easy undertaking. Any modification of the protocol would require the coordination of many different organizations. The larger the Internet grows, the more difficult this would be. As a result, despite frequent version changes in the early 1990s, BGP-4 remains today the current version of the standard and is the one that is widely used. Unless otherwise specified, any mention of BGP in this book refers to BGP-4.

Supplementing RFC 1771 are three other consecutively numbered RFCs published simultaneously with it, which provide supporting information about BGP's functions and use, as shown in Table 40-2.

**Table 40-2:** Additional Defining Standards for BGP-4

RFC Number	Name	Description
1772	Application of the Border Gateway Protocol in the Internet	Provides additional conceptual information on the operation of BGP and how it is applied to and used on the Internet. This document is sometimes considered a companion of RFC 1771 with the pair defining BGP-4.
1773	Experience with the BGP-4 Protocol	Describes the experiences of those testing and using BGP-4 and provides information that justified its acceptance as a standard.
1774	BGP-4 Protocol Analysis	Provides more detailed technical information about the operation of BGP-4.

**KEY CONCEPT** The exterior routing protocol used in modern TCP/IP internetworks is the *Border Gateway Protocol (BGP)*. Initially developed in the late 1980s as a successor to the Exterior Gateway Protocol (EGP), BGP has been revised many times; the current version is 4, so BGP is also commonly called BGP-4. BGP's primary function is the exchange of network reachability information between ASes to allow each AS on an internetwork to send messages efficiently to every other one.

### **Overview of BGP Functions and Features**

If I were to summarize the job of BGP in one phrase, it would be to exchange network reachability information between ASes and from this information determine routes to networks. In a typical internetwork (and in the Internet), each AS designates one or more routers that run BGP software. BGP routers in each AS are linked to those in one or more other ASes. Each BGP stores information about networks and the routes to them in a set of Routing Information Bases (RIBs). This route information is exchanged between BGP routers and propagated throughout the entire internetwork, allowing each AS to find paths to each other AS, and thereby enabling routing across the entire internetwork.

BGP supports an arbitrary topology of ASes, meaning that they can be connected in any manner. An AS must have a minimum of one router running BGP, but can have more than one. It is also possible to use BGP to communicate between BGP routers within the same AS.

BGP uses a fairly complex system for route determination. The protocol goes beyond the limited notion of considering only the next hop to a network the way distance-vector algorithms like RIP function. Instead, the BGP router stores more complete information about the path (sequence of ASes) from itself to a network. Special path attributes describe the characteristics of paths and are used in the process of route selection. Because of its storage of path information, BGP is sometimes called a *path-vector* protocol.

BGP chooses routes using a deterministic algorithm that assesses path attributes and chooses an efficient route, while avoiding router loops and other problem conditions. The selection of routes by a BGP router can also be controlled through a set of BGP policies that specify, for example, whether an AS is willing to carry

traffic from other ASes. However, BGP cannot guarantee the most efficient route to any destination, because it cannot know what happens within each AS and therefore what the cost is to traverse each AS.

BGP's operation is based on the exchange of messages that perform different functions. BGP routers use Open messages to contact neighboring routers and establish BGP sessions. They exchange Update messages to communicate information about reachable networks, sending only partial information as needed. They also use Keepalive and Notification messages to maintain sessions and inform peers of error conditions. The use of these messages is explained thoroughly later in this chapter.

**KEY CONCEPT** BGP supports an arbitrary topology of ASes. Each AS using BGP assigns one or more routers to implement the protocol. These devices then exchange messages to establish contact with each other and share information about routes through the internetwork using the Transmission Control Protocol (TCP). BGP employs a sophisticated path vector route calculation algorithm that determines routes from path attributes that describe how different networks can be reached.

BGP uses the Transmission Control Protocol (TCP) as a reliable transport protocol so that it can take advantage of the many connection setup and maintenance features of that protocol. This also means that BGP doesn't need to worry about issues such as message sequencing, acknowledgments, or lost transmissions. Since unauthorized BGP messages could wreak havoc with the operation of the Internet, BGP includes an authentication scheme for security.

**NOTE** BGP maintains backward compatibility with the older exterior routing protocol, EGP.

## BGP Topology

In the preceding section, I boiled down the function of BGP into this summary: the exchange of network reachability information between ASes of routers and networks, and the determination of routes from this information. The actual method that BGP uses to accomplish this, however, is fairly complex.

One of the most important characteristics of BGP is its flexibility. The protocol can connect together any internetwork of ASes using an arbitrary topology. The only requirement is that each AS have at least one router that is able to run BGP and that this router connect to at least one other AS's BGP router. Beyond that, "the sky is the limit," as they say. BGP can handle a set of ASes connected in a full mesh topology (each AS to each other AS), a partial mesh, a chain of ASes linked one to the next, or any other configuration. It also handles changes to topology that may occur over time.

Another important assumption that BGP makes is that it doesn't know anything about what happens within the AS. This is an important prerequisite to the notion of an AS being autonomous—it has its own internal topology and uses its own choice of routing protocols to determine routes. BGP just takes the information conveyed to it from the AS and shares it with other ASes.

## BGP Speakers, Router Roles, Neighbors, and Peers

Creating a BGP internetwork begins with the designation of certain routers in each AS as ones that will run the protocol. In BGP parlance, these are called *BGP speakers*, since they speak the BGP language. A protocol can reasonably be called a language, but I have not encountered this notion of a speaker in any other protocol, so it's somewhat interesting terminology.

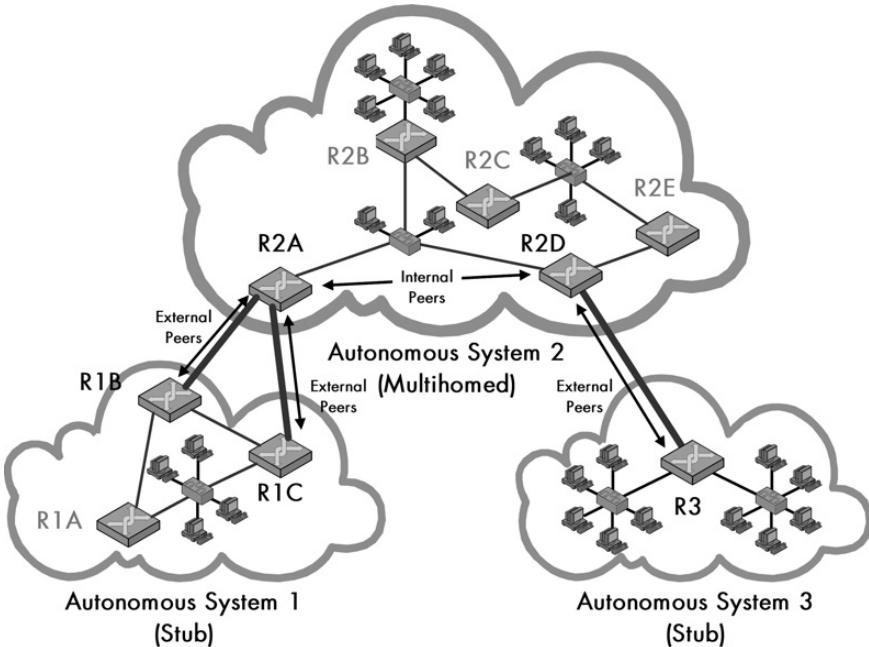
An AS can contain many routers that are connected in an arbitrary topology. We can draw a distinction between routers in an AS that are connected only to other routers within the AS versus those that connect to other ASes. Routers in the former group are usually called *internal routers*, while those in the latter group are called *border routers* in BGP (as well as similar names in other protocols; for example, in OSPF they are called *boundary routers*).

The notion of a border is the basis for the name of the BGP itself. To actually create the BGP internetwork, the BGP speakers bordering each AS are physically connected to one or more BGP speakers in other ASes, in whatever topology the internetwork designer decrees. When a BGP speaker in one AS is linked to a BGP speaker in another AS, they are deemed *neighbors*. The direct connection between them allows them to exchange information about the ASes of which they are a part.

Most BGP speakers will be connected to more than one other speaker. This provides both greater efficiency in the form of more direct paths to different networks and redundancy to allow the internetwork to cope with either device or connection failures. It is possible (and in many cases, likely) for a BGP speaker to have neighbor relationships with other BGP speakers both within its own AS and outside its AS. A neighbor within the AS is called an *internal peer*, while a neighbor outside the AS is an *external peer*. BGP between internal peers is sometimes called *Internal BGP (IBGP)*; use of the protocol between external peers is *External BGP (EBGP)*. The two are similar, but differ in certain areas, especially path attributes and route selection. You can see an example of BGP topology and the designation of internal and external peers in Figure 40-1.

This diagram is a variation on Figure 37-1 in Chapter 37. It shows the names used by BGP to refer to different types of routers and ASes. Internal routers are shown in faint type, while border routers are in bold type. BGP speakers that communicate within an AS are internal peers, while those that communicate between ASes are external peers. This highly simplified internetwork shows two stub ASes, both of which only connect to the multihomed AS 2. A peer connection between BGP speakers can be either a direct link using some form of layer 2 technology or an indirect link using TCP. This allows the BGP speakers to establish BGP sessions and then exchange routing information, using the messaging system you will see later in this chapter. It also is the means by which actual end-user traffic moves between ASes. External peers are normally connected directly, while internal peers are often linked indirectly.

You will see in a moment that the method in which ASes are connected has an important impact on the overall function of the internetwork and how traffic is carried on it.



**Figure 40-1:** Sample BGP topology and designations

**KEY CONCEPT** Each router configured to use BGP is called a *BGP speaker*; these devices exchange route information using the *BGP messaging system*. Routers that connect only to other routers in the same AS are called *internal routers*, while those that connect to other ASes are *border routers*. Neighboring BGP speakers in the same AS are called *internal peers*, while those in different ASes are *external peers*.

## BGP AS Types, Traffic Flows, and Routing Policies

When we connect ASes together to form an internetwork, the paths between AS border routers form the conduit by which messages move from one AS to another. It is very important that the flow of messages between ASes be carefully controlled. Depending on circumstances, we may wish to limit or even prohibit certain types of messages from going to or from a certain AS. These decisions in turn have a direct impact on BGP route determination.

### BGP Traffic Flow and Traffic Types

The flow of messages in an internetwork is sometimes collectively called *traffic*. This term presents a good analogy, because we can consider the matter of traffic flow control in a BGP internetwork in much the same way that we do the streets of a city. You have probably seen signs on residential streets that say “No Through Traffic” or “Local Traffic Only.” These are attempts to control the flow of traffic over those streets. A more extreme example of this would be a street in the neighborhood

where I used to live, where a barricade was intentionally erected in the middle to turn a busy through street into a pair of dead ends. Again, the goal was traffic control.

These measures highlight a key distinction between local traffic and through traffic in a neighborhood. The very same categorization is important in BGP, as shown here:

**Local Traffic** Traffic carried within an AS that either originated in that same AS *or* is intended to be delivered within that AS. This is like local traffic on a street.

**Transit Traffic** Traffic that was generated outside that AS and is intended to be delivered outside the AS. This is like through traffic on streets.

### BGP AS Types

In the previous section, I discussed the distinction between internal routers and border (or boundary) routers in an AS. We can make a similar distinction between different types of ASes, based on how they are interconnected in the overall BGP topology. There are two main types of ASes:

**Stub AS** This is an AS that is connected to only one other AS. It is comparable to a cul-de-sac (dead-end street) in a road analogy; usually, only vehicles coming from or going to houses on the street will be found on that street.

**Multihomed AS** This is an AS that is connected to two or more other ASes. It is comparable to a through street in the road analogy, because it is possible that vehicles may enter the street and pass through it, without stopping at any of the street's houses.

In the example BGP internetwork in Figure 40-1, I have linked border routers in AS 2 to both AS 1 and AS 3. While traffic from AS 2 can flow both to and from AS 1 and AS 3, it is possible that traffic from AS 1 may also flow to AS 3 and vice versa. AS 2 acts as the “through street” for these datagrams.

### BGP AS Routing Policies

The reason why BGP makes a distinction between traffic types and AS types is the same reason why it is done on the streets: Many folks have a dim view of through traffic. In a neighborhood, everyone wants to be able to get from their homes to anywhere they need to go in the city, but they don't want a lot of other people using their streets. Similarly, every AS must use at least one other AS to communicate with distant ASes, but many are less than enthusiastic about being a conduit for a lot of external traffic.

This reluctance really does make sense in many cases, either in the case of a neighborhood or in the case of BGP. Having many cars and trucks on a residential street can be a problem in a number of ways: safety issues, wear and tear on the road, pollution, and so forth. Similarly, if a multihomed AS was forced to carry all transit traffic that other ASes want to send to it, it might become overloaded.

To provide control over the carrying of transit traffic, BGP allows an AS to set up and use routing policies. These are sets of rules that govern how an AS will handle transit traffic. A great deal of flexibility exists in how an AS decides to handle transit traffic. Some of the many options include the following:

**No Transit Policy** An AS can have a policy that it will not handle transit traffic at all.

**Restricted AS Transit Policy** An AS may allow for the handling of traffic from certain ASes but not others. In this case, it tells the ASes it will handle that they may send it traffic, but does not say this to the others.

**Criteria-Based Transit Policy** An AS may use a number of different criteria to decide whether to allow transit traffic. For example, it might allow transit traffic only during certain times or only when it has enough spare capacity.

**NOTE** *An AS that is willing to carry transit traffic is sometimes called a transit AS.*

In a similar manner, policies can be set that control how an AS will have its own traffic handled by other ASes. A stub AS will always connect to the internetwork as a whole using the single AS to which it connects. A multihomed AS, however, may have policies that influence route selection by specifying the conditions under which one AS should be used over another. These policies may be based on considerations of security (if one connecting AS is deemed more secure than another), performance (if one AS is faster than another), reliability, or other factors.

**KEY CONCEPT** One important issue in BGP is how to handle the flow of traffic between ASes. Each AS in a BGP internetwork is either a *stub AS* if it connects to only one other AS, or a *multihomed AS* if it connects to two or more others. BGP allows the administrators of a multihomed AS to establish routing policies that specify under what conditions the AS is willing to handle transit traffic (messages sent over the AS whose source and destination are both external to that AS).

## Issues with Routing Policies and Internetwork Design

What would happen to a city if every street only allowed local traffic? It would be pretty hard to get around. Of course this problem never occurs in well-designed cities, because traffic planners understand the dual need for connectivity and through-traffic avoidance in residential areas. Cities are laid out in a somewhat hierarchical fashion, so local traffic funnels to thoroughfares intended specifically to carry nonlocal traffic.

The same basic situation exists in an internetwork. It wouldn't work very well if every AS declared that it was not interested in carrying transit traffic! Usually, internetworks are designed so that certain ASes are intended to carry large amounts of transit traffic. This is typically the function of high-speed, high-capacity backbone connections that serve other ASes as customers. An AS will usually carry another AS's traffic only if arrangements have been made to allow this.

## BGP Route Storage and Advertisement

The job of the BGP is to facilitate the exchange of route information between BGP devices so that each router can determine efficient routes to each of the networks on an IP internetwork. This means that descriptions of routes are the key data that BGP devices work with. Every BGP speaker is responsible for managing route descriptions according to specific guidelines established in the BGP standards.

### **BGP Route Information Management Functions**

Conceptually, the overall activity of route information management can be considered to encompass four main tasks:

**Route Storage** Each BGP stores information about how to reach networks in a set of special databases. It also uses databases to hold routing information received from other devices.

**Route Update** When a BGP device receives an Update message from one of its peers, it must decide how to use this information. Special techniques are applied to determine when and how to use the information received from peers to properly update the device's knowledge of routes.

**Route Selection** Each BGP uses the information in its route databases to select good routes to each network on the internetwork.

**Route Advertisement** Each BGP speaker regularly tells its peers what it knows about various networks and methods to reach them. This is called *route advertisement* and is accomplished using BGP Update messages. You'll learn more about these messages later in the chapter.

### **BGP Routing Information Bases (RIBs)**

The heart of BGP's system of routing information management and handling is the database where routes are stored. This database is collectively called the *Routing Information Base (RIB)*, but it is not actually a monolithic entity. It is composed of three separate sections that are used by a BGP speaker to handle the input and output of routing information. Two of these sections consist of several individual parts, or copies.

The three RIB sections (using the cryptic names given them by the BGP standards) are as follows:

**Adj-RIBs-In** A set of input database parts that holds information about routes received from peer BGP speakers.

**Loc-RIB** The local RIB. This is the core database that stores routes that have been selected by this BGP device and are considered valid to it.

**Adj-RIBs-Out** A set of output database parts that holds information about routes that this BGP device has selected to be disseminated to its peers.

Thus, the RIB can be considered either a single database or a set of related databases, depending on how you look at it. (The previous divisions are conceptual in nature; the entire RIB can be implemented as a single database with an internal structure representing the different components, or implemented as separate databases.)

The RIB is a fairly complex data structure, not just because of this multisection structure, but also because BGP devices store considerably more information about routes than simpler routing protocols. Routes are also called *paths* in BGP, and the detailed descriptions of them are stored in the form of special BGP path attributes, which we will examine shortly.

The three sections of the RIB are the mechanism by which information flow is managed in a BGP speaker. Data received from Update messages transmitted by peer BGP speakers is held in the Adj-RIBs-In, with each Adj-RIB-In holding input from one peer. This data is then analyzed and appropriate portions of it are selected to update the Loc-RIB, which is the main database of routes this BGP speaker is using. On a regular basis, information from the Loc-RIB is placed into the Adj-RIBs-Out to be sent to other peers using Update messages. This information flow is accomplished as part of the system of route update, selection, and advertisement known as the BGP decision process, which I'll discuss in the "BGP Route Determination and the BGP Decision Process" section later in this chapter.

**KEY CONCEPT** The routine operation of BGP requires BGP speakers to store, update, select, and advertise routing information. The central data structure used for this purpose is the *BGP Routing Information Base (RIB)*. The RIB actually consists of three sections: a set of input databases (*Adj-RIBs-In*) that hold routing information received from peers; a local database (*Loc-RIB*) that contains the router's current routes; and a set of output databases (*Adj-RIBs-Out*) used by the router to send its routing information to other routers.

## BGP Path Attributes and Algorithm Overview

Routing protocols that use a distance-vector algorithm, such as RIP, are relatively simple in large part because the information each device stores about each route is itself simple. Each router only knows that it can reach a network at a specific cost through a particular next-hop router. It doesn't have knowledge of the route that datagrams will take to reach any of these networks. This level of knowledge is simply insufficient for the needs of a protocol like BGP.

In order to handle the calculation of efficient, nonlooping routes in an arbitrary topology of ASes, we need to know not just that we must get Network N7 to send to Router R4, but also the characteristics of the entire path between ourselves and Network N7. By storing this additional information, it is possible to make decisions about how to compute and change routes, using knowledge of the entire path between a router and a network. Thus, instead of advertising networks in terms of a destination and the distance to that destination, BGP devices advertise networks as destination addresses and path descriptions to reach those destinations. This means BGP uses, instead of a distance-vector algorithm, a *path-vector algorithm*.

Each communication of a reachable network provides considerable information about the entire sequence of routers to a destination. Due to this inclusion of topology information, path-vector protocols are sometimes described as a combination of distance-vector and link-state algorithms. This doesn't really do them justice, however, since they do not function in the same way as either of those algorithm types. (If you are interested in additional general information about path-vector algorithms, you can find some in RFC 1322, "A Unified Approach to Inter-Domain Routing." (Warning: do not read before operating heavy machinery.)

The information about the path to each route is stored in the RIB of each BGP speaker in the form of BGP path attributes. These attributes are used to advertise routes to networks when BGP devices send out Update messages. The storing, processing, sending, and receiving of path attributes is the method by which routers decide how to create routes, so understanding them is obviously quite important.

There are several different path attributes, each of which describes a particular characteristic of a route. Attributes are divided into different categories based on their level of importance and specific rules designed to manage their propagation. The most important path attributes are called *well-known attributes*; every BGP speaker must recognize and process these, but only some are required to be sent with every route. Other attributes are optional and may or not be implemented. These are further differentiated based on how they are handled when received by a device that does not recognize them.

## **BGP Path Attribute Classes**

The four formal classifications of path attributes are as follows:

**Well-Known Mandatory** These are the most important path attributes. They must be included in every route description in Update messages, and must be processed by each BGP device receiving them.

**Well-Known Discretionary** A BGP device, if received, must recognize these path attributes, but they may or may not be included in an Update message. Thus, they are optional for a sender of information, but mandatory for a receiver to process.

**Optional Transitive** These path attributes may be recognized by a BGP router and may be included in an Update message. They must be passed on to other BGP speakers when the route is advertised, even if received by a device that does not recognize the attribute.

**Optional Nontransitive** Optional attributes that may be recognized by a BGP device and may be included in an Update message. If received by a device that does not recognize the attribute, it is dropped and not passed on to the next router.

**KEY CONCEPT** Unlike simpler routing protocols that store only limited information about how to reach a network, BGP stores detailed information about complete routes to various networks. This information takes the form of *path attributes* that describe various characteristics of a path (route) through the ASes that connect a router to a destination network.

**NOTE** As you might imagine, all well-known attributes are by definition transitive—they must be passed on from one BGP speaker to the next.

## BGP Path Attribute Characteristics

Table 40-3 provides a summary of the characteristics of each of the most common BGP path attributes used to describe the route to a destination. It also provides a summary of the Attribute Type code assigned to each characteristic in BGP Update messages.

**Table 40-3:** Summary of BGP Path Attributes

BGP Path Attribute	Classification	Attribute Type Value	Description
Origin	Well-Known Mandatory	1	Specifies the origin of the path information. This attribute indicates whether the path came originally from an interior routing protocol, the older exterior routing protocol, or some other source.
AS_Path	Well-Known Mandatory	2	A list of AS numbers that describes the sequence of ASes through which this route description has passed. This is a critically important attribute, since it contains the actual path of ASes to the network. It is used to calculate routes and to detect routing loops.
Next_Hop	Well-Known Mandatory	3	The next-hop router to be used to reach this destination.
Multi_Exit_Disc (MED)	Optional Non-Transitive	4	When a path includes multiple exit or entry points to an AS, this value may be used as a metric to discriminate between them (that is, choose one exit or entry point over the others).
Local_Pref	Well-Known Discretionary	5	Used in communication between BGP speakers in the same AS to indicate the level of preference for a particular route.
Atomic_Aggregate	Well-Known Discretionary	6	In certain circumstances, a BGP speaker may receive a set of overlapping routes whereby one is more specific than the other. For example, consider a route to the network 34.15.67.0/24 and to the network 34.15.67.0/26. The latter network is a subset of the former, which makes it more specific. If the BGP speaker uses the less-specific route (in this case, 34.15.67.0/24), it sets this path attribute to a value of 1 to indicate that this was done.
Aggregator	Optional Transitive	7	Contains the AS number and BGP ID of the router that performed route aggregation; used for troubleshooting.

Some of these path attributes are straightforward; others are fairly cryptic and probably confusing. Delving into any more detail on the path attributes leads us into a full-blown description of detailed inter-AS route calculations. We'll look at that to some degree in the next section.

## BGP Route Determination and the BGP Decision Process

You have now looked at the fundamentals of how BGP devices store and manage information about routes to networks. This included an overview of the four route information management activities performed by BGP speakers: route

storage, update, selection, and advertisement. Route storage is the function of the RIB in each BGP speaker. Path attributes are the mechanism by which BGP stores details about routes and also describes those details to BGP peers.

### **BGP Decision Process Phases**

As you have seen, the RIB also contains sections for holding input information received from BGP peers and for holding output information that each BGP device wants to send to those peers. The functions of route update, selection, and advertisement are concerned with analyzing this input information. They also decide what to include in the local database, update that database, and then choose what routes to send from it to peer devices. In BGP, a mechanism called the *decision process* is responsible for these tasks. It consists of three overall phases:

**Phase 1** Each route received from a BGP speaker in a neighboring AS is analyzed and assigned a preference level. The routes are then ranked according to preference and the best one for each network advertised to other BGP speakers within the AS.

**Phase 2** The best route for each destination is selected from the incoming data based on preference levels, and it's used to update the local routing information base (the Loc-RIB).

**Phase 3** Routes in the Loc-RIB are selected to be sent to neighboring BGP speakers in other ASes.

### **Criteria for Assigning Preferences to Routes**

Obviously, if a BGP speaker only knows of a single route to a network, it will install and use that route (assuming there are no problems with it). The assigning of preferences among routes becomes important only when more than one route has been received by a BGP speaker for a particular network. Preferences can be determined based on a number of different criteria. The following are a few typical ones:

- The number of ASes between the router and the network (fewer generally being better).
- The existence of certain policies that may make certain routes unusable; for example, a route may pass through an AS that as the BGP speaker is not willing to trust with its data.
- The origin of the path—that is, where it came from.

In the case where a set of routes to the same network are all calculated to have the same preference, a tie-breaking scheme is used to select from among them. Additional logic is used to handle special circumstances, such as the case of overlapping networks (see the description of the Atomic\_Aggregate path attribute in Table 40-3 for an example of this).

The selection of routes for dissemination to other routers in phase 3 is based on a rather complex algorithm that I cannot explain adequately here. Route advertisement is guided by the routing policies I discussed earlier in this chapter.

Different rules are used to select routes for advertising to internal peers compared to external peers.

**KEY CONCEPT** The method used by a BGP speaker to determine what new routes to accept from its peers and what routes to advertise back them is called the *BGP decision process*. It is a complex algorithm in three phases that involves the computation of the best route based on both preexisting and incoming path information.

### ***Limitations on BGP's Ability to Select Efficient Routes***

When considering route selection, it's very important to remember that BGP is a routing protocol that operates at the inter-AS level. Thus, routes are chosen between ASes, not at the level of individual routers within an AS. So, for example, when BGP stores information about the path to a network, it stores it as a sequence of ASes, not a sequence of specific routers.

BGP cannot deal with individual routers in an AS because, by definition, the details of what happens within an AS are supposed to be hidden from the outside world. It doesn't know the structure of ASes outside its own. This has an important implication for how BGP selects routes: BGP cannot guarantee that it will pick the fastest, lowest-cost route to every network. It can select a route that minimizes the number of ASes that lie between itself and a particular network, but, of course, ASes are not all the same. Some ASes are large and consist of many slow links; others are small and fast. Choosing a route through two of the latter types of AS will be better than choosing a route through one of the former, but BGP can't know that. Policies can be used to influence AS selection to some extent, but in general, since BGP doesn't know what happens in an AS, it cannot guarantee the efficiency of a route overall. (Incidentally, this is the reason why there is no general cost or distance path attribute in BGP.)

**KEY CONCEPT** As an exterior routing protocol, BGP operates at the AS level. Its routes are calculated based on paths between ASes, not individual routers. Since BGP, by definition, does not know the internal structure of routers within an AS, it cannot know for certain the cost to send a datagram across a given AS. This means that BGP cannot always guarantee that it will select the absolute lowest-cost route between any two networks.

### ***Originating New Routes and Withdrawing Unreachable Routes***

Naturally, a facility exists to allow BGP speakers to originate new routes to networks. A BGP speaker may obtain knowledge about a new route from an interior routing protocol on an AS to which it is directly attached, and then it may choose to share this information with other ASes. It will create a new entry in its RIB for this network and then send information about it out to other BGP peers.

BGP also includes a mechanism for advertising routes it cannot reach. These are called unfeasible or withdrawn routes and are mentioned in Update messages to indicate that a router can no longer reach the specific network.

## BGP General Operation and Messaging

In the previous sections, you have seen how BGP stores information about routes and uses it to determine paths to various networks. Let's now take a high-level look at how BGP operates in general terms. Like many other protocols covered in this book, BGP's operation can be described primarily in the form of messaging. The use of messages is the means by which route information is communicated between BGP peers. This eventually allows the knowledge of how to reach networks to spread throughout the entire internetwork.

### ***Speaker Designation and Connection Establishment***

Before messaging can begin, BGP speakers must be designated and then linked together. The BGP standard does not specify how neighboring speakers are determined; this must be done outside the protocol. Once accomplished, ASes are connected into a BGP-enabled internetwork. Topological linking provides the physical connection and the means for datagrams to flow between ASes. At this point, the dance floor is prepared, but nobody is dancing; BGP can function but isn't yet in operation.

BGP operation begins with BGP peers forming a transport protocol connection. BGP uses TCP for its reliable transport layer, so the two BGP speakers establish a TCP session that remains in place during the course of the subsequent message exchange. When this is done, each BGP speaker sends a BGP Open message. This message is like an invitation to dance, and it begins the process of setting up the BGP link between the devices. In this message, each router identifies itself and its AS, and also tells its peer what parameters it would like to use for the link. This includes an exchange of authentication parameters. Assuming that each device finds the contents of its peer's Open message acceptable, it acknowledges it with a Keepalive message, and the BGP session begins.

Under normal circumstances, most BGP speakers will maintain simultaneous sessions with more than one other BGP speaker, both within the speaker's own AS and outside its AS. Links between ASes are what enable BGP routers to learn how to route through the internetwork. Links within the AS are important to ensure that each BGP speaker in the AS maintains consistent information.

### ***Route Information Exchange***

Assuming the link is initialized, the two peers begin an ongoing process of telling each other what they know about networks and how to reach them. Each BGP speaker encodes information from its RIBs into BGP Update messages. These messages contain lists of known network addresses, as well as information about paths to various networks, as described in the form of path attributes, as you have already seen. This information is then used for the route determination, as described in the preceding section.

When a link is first set up between two peers, those peers ensure that each router holds complete information by exchanging their complete routing tables. Subsequently, Update messages are sent. They contain only incremental updates about routes that have changed. Exchanging only updated information as needed reduces unnecessary bandwidth on the network, thereby making BGP more efficient than it would be if it sent full routing table information on a regular basis.

## **Connectivity Maintenance**

The TCP session between BGP speakers can be kept open for a very long time. Update messages need to be sent only when changes occur to routes, which are usually infrequent. This means many seconds may elapse between the transmission of Update messages.

To ensure that the peers maintain contact with each other, they both send Keepalive messages on a regular basis when they don't have other information to send. These are null messages that contain no data and just tell the peer device "I'm still here." These messages are sent infrequently—no more often than one per second—but regularly enough that the peers won't think the session was interrupted.

## **Error Reporting**

The last type of BGP message is the BGP Notification message. This is an error message; it tells a peer that a problem occurred and describes the nature of the error condition. After sending a BGP Notification message, the device that sent it will terminate the BGP connection between the peers. A new connection will then need to be negotiated, possibly after the problem that led to the Notification message has been corrected.

**KEY CONCEPT** BGP is implemented through the exchange of four different message types between BGP speakers. A BGP session begins with a TCP connection being established between two routers and each sending an Open message to the other. BGP Update messages are the primary mechanism by which routing information is exchanged between devices. Small BGP Keepalive messages are used to maintain communication between devices between periods when they need to exchange information. Finally, Notification messages are used for problem reporting.

## **BGP Detailed Messaging, Operation, and Message Formats**

So far, I have discussed the concepts and general operation of the BGP. To get a better understanding of exactly how BGP works, it is helpful to take a detailed look at its four different message types—Open, Update, Keepalive, and Notification—and how they are used. As we do this, we can examine the fields in each message type, so that you can comprehend not just the way that messaging is accomplished, but the way that routing data is actually communicated. Let's begin with a description of common attributes of BGP message generation and transport, and the general format used for all BGP messages.

### ***BGP Message Generation and Transport***

Each router running BGP generates messages to implement the various functions of the protocol. Some of these messages are created on a regular basis by the BGP software during the course of its normal operation. These are generally controlled by timers that are set and counted down to cause them to be sent. Other messages are sent in response to messages received from BGP peers, possibly after a processing step.

BGP is different from most other routing protocols in that it was designed from the start to operate using a reliable method of message delivery. TCP is present in the software of every Internet Protocol (IP) router, thereby making it the obvious choice for reliable data communication in a TCP/IP Internet, and that's what BGP uses. Routing protocols are usually considered part of layer 3, but this one runs over a layer 4 protocol, thereby making BGP a good example of why architectural models are best used only as a guideline.

TCP provides numerous advantages to BGP by taking care of most of the details of session setup and management, thereby allowing BGP to focus on the data it needs to send. TCP takes care of session setup and negotiation, flow control, congestion handling, and any necessary retransmissions of lost messages, thereby ensuring that messages are received and acknowledged. BGP uses well-known TCP port 179 for connections.

### **BGP General Message Format**

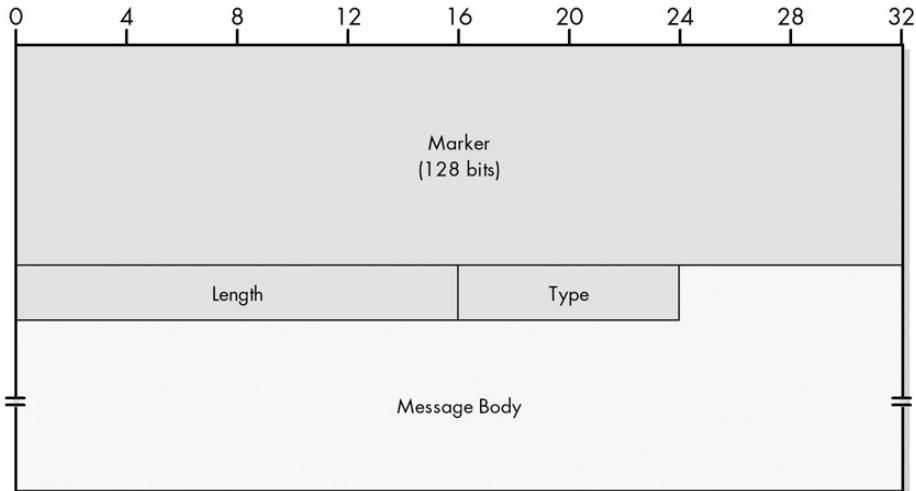
The use of TCP also has an interesting impact on the way BGP messages are structured. One thing that stands out when you look at the BGP message format (as you will see shortly) is that a BGP message can have an odd number of bytes. Most routing protocols are sized in units of 4 or 8 bytes, but since TCP sends data as a stream of octets, there is no need for BGP messages to break on a 32-bit or 64-bit boundary. The other impact is the need of a special Marker field to help ensure that BGP messages can be differentiated from each other in the TCP stream (more about this in a moment).

Like most messaging protocols, BGP uses a common message format for each of its four message types. Each BGP message is conceptually divided into a header and a body (called the *data portion* in the BGP standard). The header has three fields and is fixed in length at 19 bytes. The body is variable in length and is omitted entirely in Keepalive messages, since it is not needed for them.

The general format for all BGP message types is shown in Table 40-4 and illustrated in Figure 40-2.

**Table 40-4:** BGP General Message Format

Field Name	Size (Bytes)	Description
Marker	16	This large field at the start of each BGP message is used for synchronization and authentication.
Length	2	The total length of the message in bytes, including the fields of the header. The minimum value of this field is 19 for a Keepalive message; it may be as high as 4096.
Type	1	Indicates the BGP message type: 1 = Open 2 = Update 3 = Notification 4 = Keepalive
Message Body/ Data Portion	Variable	Contains the specific fields used to implement each message type for Open, Update, and Notification messages.



**Figure 40-2:** BGP general message format

The Marker field is the most interesting one in the BGP message format. It is used for both synchronization and authentication. BGP uses a single TCP session to send many messages in a row. TCP is a stream-oriented transport protocol that sends bytes across the link without any knowledge of what the bytes represent. This means that the protocol using TCP is responsible for deciding where the line is drawn between data units—in this case, BGP messages.

Normally, the Length field tells each BGP device where to draw the line between the end of one message and the start of the next. However, it is possible that, due to various conditions, a device might lose track of where the message boundary is. The Marker field is filled with a recognizable pattern that clearly marks the start of each message; BGP peers keep synchronized by looking for that pattern.

Before a BGP connection is established, the Marker field is filled with all ones. Thus, this is the pattern used for Open messages. Once a BGP session is negotiated, if agreement is reached on an authentication method between the two devices, the Marker field takes on the additional role of authentication. Instead of looking for a Marker field containing all ones, BGP devices look for a pattern generated using the agreed-upon authentication method. Detection of this pattern simultaneously synchronizes the devices and ensures that messages are authentic.

In extreme cases, BGP peers may be unable to maintain synchronization, and if so, a Notification message is generated and the session is closed. This will also happen if the Marker field contains the wrong data when authentication is enabled.

**KEY CONCEPT** All four BGP message types use a general message format that contains three fixed header fields—Marker, Length, and Type—and room for a message body that differs for each message type. The large Marker field is used to denote the start of a new BGP message, and it is also used to facilitate the BGP authentication method.

## BGP Connection Establishment: Open Messages

Before a BGP session can be used to exchange routing information, a connection must first be established between BGP peers. This process begins with the creation of a TCP connection between the devices. Once this is done, the BGP devices will attempt to create a BGP session by exchanging BGP Open messages.

### BGP Open Message Functions

The Open message has two main purposes. The first is identification and initiation of a link between the two devices; it allows one peer to tell the other, “I am a BGP speaker named X on AS Y, and I want to start exchanging BGP information with you.” The second is the negotiation of session parameters. These are the terms by which the BGP session will be conducted. One important parameter negotiated using Open messages is the method that each device wants to use for authentication. The importance of BGP means that authentication is essential in order to prevent bad information or a malicious person from disrupting routes.

Each BGP receiving an Open message processes it. If the message’s contents are acceptable, including the parameters the other device wants to use, it responds with a Keepalive message as an acknowledgment. Each peer must send an Open message and receive a Keepalive acknowledgment for the BGP link to be initialized. If either is not willing to accept the terms of the Open message, the link is not established. In that case, a Notification message may be sent to convey the nature of the problem.

### BGP Open Message Format

The specific format for BGP Open messages is shown in Table 40-5 and Figure 40-3.

**Table 40-5:** BGP Open Message Format

Field Name	Size (Bytes)	Description
Marker	16	This large field at the start of each BGP message is used for synchronization and authentication.
Length	2	The total length of the message in bytes, including the fields of the header. Open messages are variable in length.
Type	1	BGP message type; value is 1 for Open messages.
Version	1	Indicates the BGP version the sender of the Open message is using. This field allows devices to reject connections with devices using versions that they may not be capable of understanding. The current value is 4, for BGP-4, and is used by most, if not all, current BGP implementations.
My Autonomous System	2	Identifies the AS number of the sender of the Open message. AS numbers are centrally managed across the Internet in a manner similar to how IP addresses are administered.
Hold Time	2	The number of seconds that this device proposes to use for the BGP hold timer, which specifies how long a BGP peer will allow the connection to be left silent between receipt of BGP messages. A BGP device may refuse a connection if it doesn’t like the value that its peer is suggesting; usually, however, the two devices agree to use the smaller of the values suggested by each device. The value must be at least 3 seconds, or 0. If 0, this specifies that the hold timer is not used. See the Keepalive message discussion later in this chapter for more on how the hold timer is used.

(continued)

**Table 40-5:** BGP Open Message Format (continued)

Field Name	Size (Bytes)	Description
BGP Identifier	4	Identifies the specific BGP speaker. You'll recall that IP addresses are associated with interfaces, not devices, so each router will have at least two IP addresses. Normally, the BGP identifier is chosen as one of these addresses. Once chosen, this identifier is used for all BGP communications with BGP peers. This includes BGP peers on the interface from which the identifier was chosen, and also BGP peers on other interfaces as well. So, if a BGP speaker with two interfaces has addresses IP1 and IP2, it will choose one as its identifier and use it on both of its interfaces.
Opt Parm Len	1	The number of bytes used for Optional Parameters (see the following entry). If 0, no optional parameters are in this message.
Optional Parameters	Variable	Allows the Open message to communicate any number of extra parameters during BGP session setup. Each parameter is encoded using a rather standard type/length/value triple, as shown in Table 40-6.

**Table 40-6:** BGP Open Message Optional Parameters

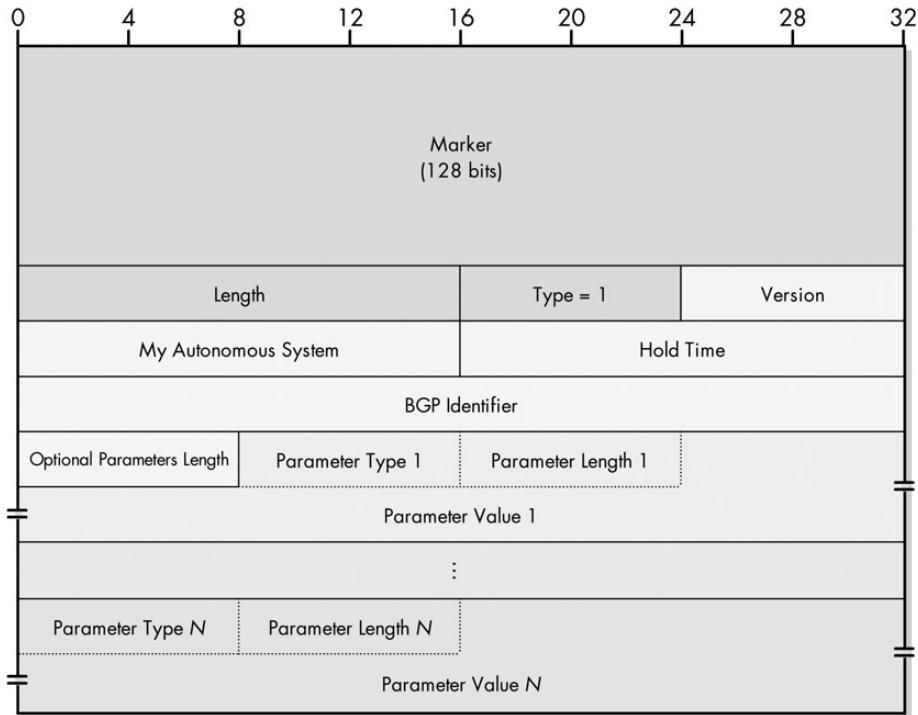
Subfield Name	Size (Bytes)	Description
Parm Type	1	Parameter Type: The type of the optional parameter. At present, only one value is defined, 1, for Authentication Information.
Parm Length	1	Parameter Length: Specifies the length of the Parameter Value subfield (thus, this value is the length of the entire parameter, less 2).
Parm Value	Variable	Parameter Value: The value of the parameter being communicated.

BGP Open messages currently use only one optional parameter: Authentication Information. Its Parameter Value subfield contains a one-byte Authentication Code sub-subfield, which specifies the type of authentication a device wishes to use. Following this is a variable-length Authentication Data sub-subfield. The Authentication Code specifies how authentication is to be performed, including the meaning of the Authentication Data field, and the manner in which Marker fields are to be calculated.

**KEY CONCEPT** BGP sessions begin with each peer in a connection sending the other a BGP Open message. The purpose of this message is to establish contact between devices, identify the sender of the message and its AS, and negotiate important parameters that dictate how the session will be conducted.

### ***BGP Route Information Exchange: Update Messages***

Once BGP speakers have made contact and a link has been established using Open messages, the devices begin the actual process of exchanging routing information. Each BGP router uses the BGP decision process described earlier in this chapter to select certain routes to be advertised to its peer. This information is then placed into BGP Update messages, which are sent to every BGP device for which a session has been established. These messages are the way that network reachability knowledge is propagated around the internetwork.



**Figure 40-3: BGP Open message format**

### BGP Update Message Contents

Each Update message contains either one or both of the following:

**Route Advertisement** The characteristics of a single route.

**Route Withdrawal** A list of networks that are no longer reachable.

Only one route can be advertised in an Update message, but several can be withdrawn. This is because withdrawing a route is simple; it requires just the address of the network for which the route is being removed. In contrast, a route advertisement requires a fairly complex set of path attributes to be described, which takes up a significant amount of space. (Note that it is possible for an Update message to specify only withdrawn routes and not advertise a route at all.)

### BGP Update Message Format

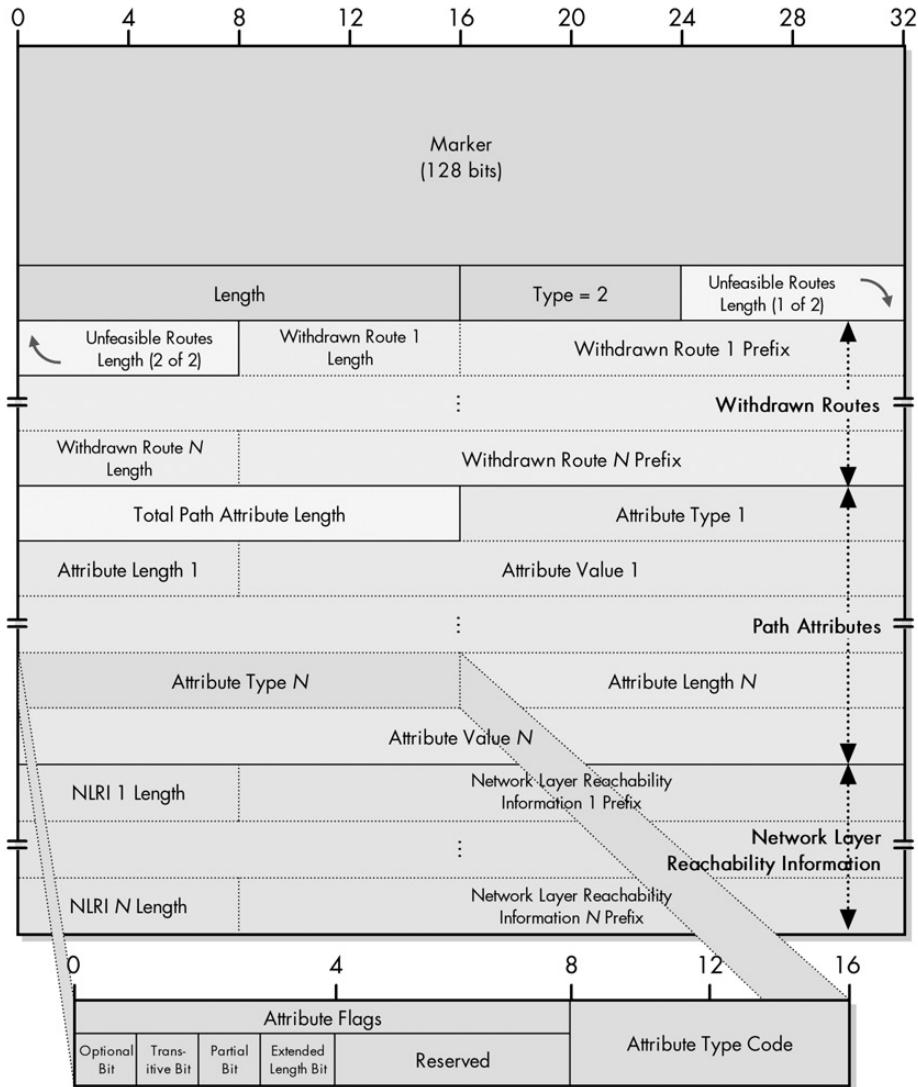
Because of the amount of information it contains and the complexity of that information, BGP Update messages use one of the most complicated structures in TCP/IP. The basic structure of the message is described in Table 40-7 and illustrated in Figure 40-4. As you can see in that table, several of the fields have their own substructure. The Path Attributes field has a complex substructure, which I have shown separately in other tables.

**Table 40-7:** BGP Update Message Format

<b>Field Name</b>	<b>Size (Bytes)</b>	<b>Description</b>
Marker	16	This large field at the start of each BGP message is used for synchronization and authentication.
Length	2	The total length of the message in bytes, including the fields of the header. Update messages are variable in length.
Type	1	BGP message type; value is 2 for Update messages.
Unfeasible Routes Length	2	The length of the Withdrawn Routes field, in bytes. If 0, no routes are being withdrawn and the Withdrawn Routes field is omitted.
Withdrawn Routes	Variable	Specifies the addresses of networks for which routes are being withdrawn from use. Each address is specified using the two subfields. The 1-byte Length field is the number of bits in the IP address Prefix subfield that are significant. The variable-length Prefix subfield is the IP address prefix of the network whose route is being withdrawn. If the number of bits in the prefix is not a multiple of 8, this field is padded with zeros so that it falls on a byte boundary. The length of this field is 1 byte if the preceding Length field is 8 or less; 2 bytes if it is 9 to 16; 3 bytes if it is 17 to 24; and 4 bytes if it is 25 or greater.
Total Path Attribute Length	2	The length of the Path Attributes field, in bytes. If 0, indicates no route is being advertised in this message, so Path Attributes and Network Layer Reachability Information are omitted.
Path Attributes	Variable	Describes the path attributes of the route advertised. Since some attributes require more information than others, attributes are described using a flexible structure that minimizes message size compared to using fixed fields that would often be empty. Unfortunately, it also makes the field structure confusing. Each attribute has the subfields shown in Table 40-8.
Network Layer Reachability Information (NLRI)	Variable	Contains a list of IP address prefixes for the route being advertised. Each address is specified using the same general structure as the one used for Withdrawn Routes. The 1-byte Length subfield is the number of bits in the Prefix subfield that are significant. The variable-length Prefix subfield is the IP address prefix of the network whose route is being advertised. If the number of bits in the prefix is not a multiple of 8, this field is padded with zeros so that it falls on a byte boundary. The length of this field is 1 byte if the preceding Length field is 8 or less; 2 bytes if it is 9 to 16; 3 bytes if it is 17 to 24; and 4 bytes if it is 25 or greater. Unlike most of the other fields in the Update message, the length of the NLRI field is not explicitly stated. It is computed from the overall message Length field, minus the lengths of the other fields that are explicitly specified.

**Table 40-8:** BGP Update Message Path Attributes

<b>Subfield Name</b>	<b>Size (Bytes)</b>	<b>Description</b>
Attribute Type	2	Defines the type of attribute and describes it. This subfield itself has a two-level substructure, with Attribute Type flags and Attribute Type codes, so it won't even fit it here! See Tables 40-9 and 40-10 for details.
Attribute Length	1 or 2	The length of the attribute in bytes. This field is normally 1 byte, thereby allowing for fields with a length up to 255 bytes. For longer attributes, the Extended Length flag is set (see Table 40-9), indicating that this Attribute Length field is 2 bytes, for attributes up to 65,535 bytes.
Attribute Value	Variable	The value of the attribute. The size and meaning of this field depends on the type of path attribute. For example, for an Origin attribute, it is a single integer value indicating the origin of the route; for an AS_Path attribute, this field contains a variable-length list of the ASes in the path to the network.



**Figure 40-4: BGP Update message format** This diagram shows the complete BGP Update message format, including a set of withdrawn routes, path attributes, and NLRI entries. The exploded view shows the substructure of the Attribute Type subfield of the Path Attributes, as described in Tables 40-9 and 40-10.

Table 40-9 shows the structure of the Attribute Flags sub-subfield of the Attribute Type subfield of the Path Attributes field. This subfield contains a set of flags that describe the nature of the attribute and how to process it. You may need to refer to the path attributes description in the “BGP Path Attributes and Algorithm Overview” section earlier in this chapter to make sense of these flags.

**Table 40-9:** BGP Update Message Attribute Flags

Sub-Sub-Subfield Name	Size (Bytes)	Description
Optional	1/8 (1 bit)	Set to 1 for optional attributes; 0 for well-known attributes.
Transitive	1/8 (1 bit)	Set to 1 for optional transitive attributes; 0 for optional nontransitive attributes. Always set to 1 for well-known attributes.
Partial	1/8 (1 bit)	When 1, indicates that information about an optional transitive attribute is partial. This means that since it was optional and transitive, one or more of the routers that passed the path along did not implement that attribute but was forced to pass it along, so information about it may be missing (not supplied by the routers that didn't recognize it but just passed along). If 0, it means information is complete. This bit has meaning only for optional transitive attributes; for well-known or nontransitive attributes, it is 0.
Extended Length	1/8 (1 bit)	Set to 1 for long attributes to indicate that the Attribute Length field is 2 bytes in size. Normally 0, meaning the Attribute Length field is a single byte.
Reserved	4/8 (4 bits)	Set to 0 and ignored.

The Attribute Type Code sub-subfield of the Attribute Type subfield of the Path Attributes field contains a number that identifies the attribute type. Table 40-10 shows the current values.

**Table 40-10:** BGP Update Message Attribute Type Codes

Value	Attribute Type
1	Origin
2	AS_Path
3	Next_Hop
4	Multi_Exit_Disc (MED)
5	Local_Pref
6	Atomic_Aggregate
7	Aggregator

It may seem confusing that there can be more than one prefix in the Network Layer Reachability Information (NLRI) field, even though I said earlier that an Update message advertises only one route. There is, in fact, no inconsistency here. A single route may be associated with more than one networks; to put it another way, multiple networks may have the same path and path attributes. In that case, specifying multiple network prefixes in the same Update message is more efficient than generating a new one for each network.

**KEY CONCEPT** The most important message type in BGP is the Update message, which is used to send detailed information about routes between BGP devices. It uses a complex structure that allows a BGP speaker to efficiently specify new routes, update existing ones, and withdraw routes that are no longer valid. Each message may include the full description of one existing route and may also withdraw from use a list of multiple routes.

## **BGP Connectivity Maintenance: Keepalive Messages**

Once a BGP connection is established using Open messages, BGP peers will initially use Update messages to send each other a large amount of routing information. They will then settle into a routine in which the BGP session is maintained, but Update messages are sent only when needed. Since these updates correspond to route changes, and route changes are normally infrequent, this means many seconds may elapse between the receipt of consecutive Update messages.

### **The BGP Hold Timer and Keepalive Message Interval**

While a BGP peer is waiting to hear the next Update message, it remains sort of like a person who has been put on hold on the telephone. Now seconds may not seem like much to us, but to a computer, they are a very long time. Like you, a BGP speaker that is put on hold for too long might become impatient and might start to wonder if maybe the other guy hung up. Computers don't get offended at being put on hold, but they might wonder if perhaps a problem arose that led to the connection being interrupted.

To keep track of how long it has been on hold, each BGP device maintains a special *hold timer*. This hold timer is set to an initial value each time its peer sends a BGP message. The timer then counts down until the next message is received, and then it is reset. If the hold timer ever expires, the connection is assumed to have been interrupted and the BGP session is terminated.

The length of the hold timer is negotiated as part of session setup using Open messages. It must be at least three seconds long, or may be negotiated as a value of zero. If zero, the hold timer is not used; this means the devices are infinitely patient and don't care how much time elapses between messages.

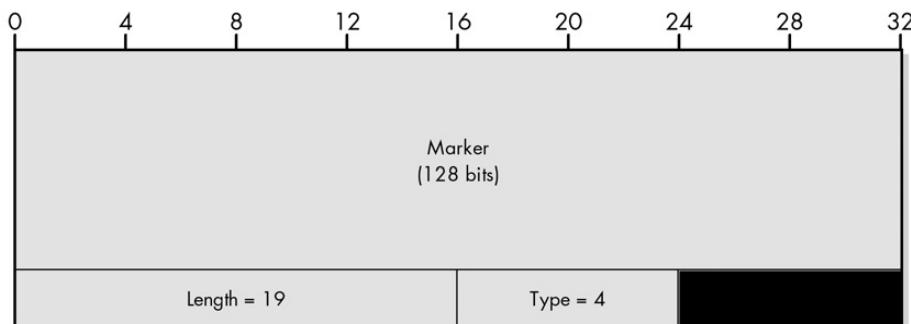
To ensure that the timer doesn't expire even when no Update messages need to be sent for a long while, each peer periodically sends a BGP Keepalive message. The name says it all: The message just keeps the BGP connection alive. The rate at which Keepalive messages are sent depends on the implementation, but the standard recommends that they be sent with an interval of one-third the value of the hold timer. So if the hold timer has a value of three seconds, each peer sends a Keepalive message every second (unless it needs to send some other message type in that second). To prevent excess bandwidth use, Keepalive messages must be sent no more often than once per second, so that is the minimum interval, even if the hold timer is shorter than three seconds.

### **BGP Keepalive Message Format**

The point of a Keepalive message is the message itself; there's no data to be communicated. In fact, we want to keep the message short and sweet. Thus, it is really a dummy message that contains only a BGP header—a nice change after that incredibly long Update message format! The format of the Keepalive message is shown in Table 40-11 and Figure 40-5.

**Table 40-11: BGP Keepalive Message Format**

Field Name	Size (Bytes)	Description
Marker	16	This large field at the start of each BGP message is used for synchronization and authentication.
Length	2	The total length of the message in bytes, including the fields of the header. Keepalive messages are fixed in length at 19 bytes.
Type	1	BGP message type; value is 4 for Keepalive messages.



**Figure 40-5: BGP Keepalive message format**

There is also a special use for Keepalive messages: They acknowledge the receipt of a valid Open message during the initial BGP session setup.

**KEY CONCEPT** BGP Keepalive messages are sent periodically during idle periods when no real information needs to be sent between connected BGP speakers. They serve only to keep the session alive, and thus contain only a BGP header and no data.

## **BGP Error Reporting: Notification Messages**

Once established, a BGP session will remain open for a considerable period of time, allowing routing information to be exchanged between devices on a regular basis. During the course of operation, certain error conditions may crop up that may interfere with normal communication between BGP peers.

### **BGP Notification Message Functions**

Some of the error conditions that arise are serious enough that the BGP session must be terminated. When this occurs, the device detecting the error will inform its peer of the nature of the problem by sending it a BGP Notification message, and then it will close the connection.

Of course, having someone tell you, “I found an error, so I quit” is not of much value. Therefore, the BGP Notification message contains a number of fields that provide information about the nature of the error that caused the message to be sent. This includes a set of primary error codes as well as subcodes within some of these error codes. Depending on the nature of the error, an additional data field may also be included to aid in diagnosing the problem.

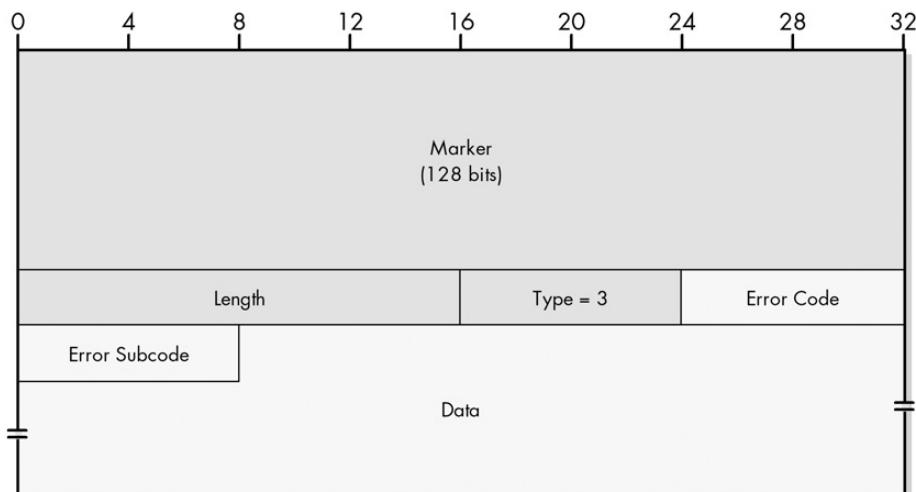
In addition to the use of Notification messages to convey the occurrence of an error, this message type is also used for other purposes. For example, one may be sent if two devices cannot agree on how to negotiate a session, which isn't, strictly speaking, an error. A Notification message is also used to allow a device to tear down a BGP session for reasons that have nothing to do with an error.

### BGP Notification Message Format

The format for the BGP Notification messages is detailed in Table 40-12 and illustrated in Figure 40-6.

**Table 40-12:** BGP Notification Message Format

Field Name	Size (Bytes)	Description
Marker	16	This large field at the start of each BGP message is used for synchronization and authentication.
Length	2	The total length of the message in bytes, including the fields of the header. Notification messages are variable in length.
Type	1	BGP message type; value is 3 for Notification messages.
Error Code	1	Specifies the general class of the error. Table 40-13 shows the possible error types with a brief description of each.
Error Subcode	1	Provides a more specific indication of the cause of the error for three of the Error Code values. The possible values of this field for each Error Code value are shown in Table 40-14.
Data	Variable	Contains additional information to help diagnose the error. Its meaning depends on the type of error specified in the Error Code and Error Subcode fields. In most cases, this field is filled in with whatever bad value caused the error to occur. For example, for "Message Header Error / Bad Message Type," the value of the bad Type field is placed here.



**Figure 40-6:** BGP Notification message format

Tables 40-13 and 40-14 show the values permitted for the Error Code and Error Subcode fields, respectively, and thus provide a good summary of the types of errors that Notification messages can report. They also demonstrate the other nonerror uses of the message type.

**Table 40-13:** BGP Notification Message Error Codes

Error Code Value	Code Name	Description
1	Message Header Error	A problem was detected either with the contents or length of the BGP header. The Error Subcode field provides more details on the nature of the problem.
2	Open Message Error	A problem was found in the body of an Open message. The Error Subcode field describes the problem in more detail. Note that authentication failures or the inability to agree on a parameter such as hold time are included here.
3	Update Message Error	A problem was found in the body of an Update message. Again, the Error Subcode field provides more information. Many of the problems that fall under this code are related to issues detected in the routing data or path attributes sent in the Update message, so these messages provide feedback about such problems to the device sending the erroneous data.
4	Hold Timer Expired	A message was not received before the hold time expired. See the description of the Keepalive message earlier in this chapter for details on this timer.
5	Finite State Machine Error	The BGP finite state machine refers to the mechanism by which the BGP software on a peer moves from one operating state to another based on events (see the TCP finite state machine description in Chapter 47 for some background on this concept). If an event occurs that is unexpected for the state the peer is currently in, it will generate this error.
6	Cease	Used when a BGP device wants to break the connection to a peer for a reason not related to any of the error conditions described by the other codes.

**Table 40-14:** BGP Notification Message Error Subcodes

Error Type	Error Subcode Value	Subcode Name	Description
Message Header Error (Error Code 1)	1	Connection Not Synchronized	The expected value in the Marker field was not found, indicating that the connection has become unsynchronized. See the description of the Marker field in Table 40-12.
	2	Bad Message Length	The message was less than 19 bytes, greater than 4096 bytes, or not consistent with what was expected for the message type.
	3	Bad Message Type	The Type field of the message contains an invalid value.

(continued)

**Table 40-14:** BGP Notification Message Error Subcodes (continued)

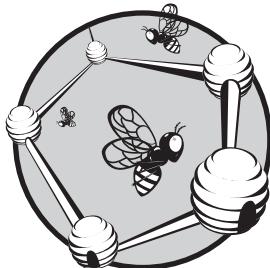
Error Type	Error Subcode Value	Subcode Name	Description
Open Message Error (Error Code 2)	1	Unsupported Version Number	The device does not “speak” the version number its peer is trying to use.
	2	Bad Peer AS	The router doesn’t recognize the peer’s AS number or is not willing to communicate with it.
	3	Bad BGP Identifier	The BGP Identifier field is invalid.
	4	Unsupported Optional Parameter	The Open message contains an optional parameter that the recipient of the message doesn’t understand.
	5	Authentication Failure	The data in the Authentication Information optional parameter could not be authenticated.
	6	Unacceptable Hold Time	The router refuses to open a session because the proposed hold time its peer specified in its Open message is unacceptable.
Update Message Error (Error Code 3)	1	Malformed Attribute List	The overall structure of the message’s path attributes is incorrect, or an attribute has appeared twice.
	2	Unrecognized Well-Known Attribute	One of the mandatory well-known attributes was not recognized.
	3	Missing Well-Known Attribute	One of the mandatory well-known attributes was not specified.
	4	Attribute Flags Error	An attribute has a flag set to a value that conflicts with the attribute’s type code.
	5	Attribute Length Error	The length of an attribute is incorrect.
	6	Invalid Origin Attribute	The Origin attribute has an undefined value.
	7	AS Routing Loop	A routing loop was detected.
	8	Invalid Next_Hop Attribute	The Next_Hop attribute is invalid.
	9	Optional Attribute Error	An error was detected in an optional attribute.
	10	Invalid Network Field	The Network Layer Reachability Information field is incorrect.
	11	Malformed AS_Path	The AS_Path attribute is incorrect.

Note that, perhaps ironically, no mechanism exists to report an error in a Notification message itself. This is likely because the connection is normally terminated after such a message is sent.

**KEY CONCEPT** BGP Notification messages are used for error reporting between BGP peers. Each message contains an Error Code field that indicates what type of problem occurred. For certain Error Code fields, an Error Subcode field provides additional details about the specific nature of the problem. Despite these field names, Notification messages are also used for other types of special nonerror communication, such as terminating a BGP connection.

# 41

## OTHER ROUTING PROTOCOLS



The Routing Information Protocol (RIP), Open Shortest Path First (OSPF), and Border Gateway Protocol (BGP)—detailed in the preceding chapters—are the three most well-known routing protocols used in the TCP/IP protocol suite. But there are several other TCP/IP routing protocols, and they fall into one of two categories. Some protocols are no longer in use today but are nevertheless interesting from a historical perspective. Others are proprietary RIP and OSPF alternatives that you may occasionally encounter in today's networking world.

In this chapter, I provide a brief description of five additional TCP/IP routing protocols. I begin with a look at two obsolete interior routing protocols that played an important role in the early Internet: the Gateway-to-Gateway Protocol (GGP) and the HELLO Protocol. I then describe two interior routing protocols (developed by Cisco Systems) that are sometimes seen in the industry today as alternatives to RIP and OSPF: the Interior Gateway Routing Protocol (IGRP) and the Enhanced Interior Gateway Routing Protocol (EIGRP). I conclude with a discussion of the Exterior Gateway Protocol (EGP), the exterior routing protocol that preceded BGP.

## TCP/IP Gateway-to-Gateway Protocol (GGP)

In Chapter 37, I described the evolution of TCP/IP routing architectures. The modern Internet is based on the concept of independent autonomous systems (ASes), which run interior routing protocols within them and exterior routing protocols between them. The early Internet, however, was somewhat simpler. It consisted of a relatively small number of core routers that carried detailed information about the Internet as a whole, as well as noncore routers that knew only partial information about the whole internetwork and were located around the core.

These core routers used a special routing protocol to communicate called the *Gateway-to-Gateway Protocol (GGP)*. Bolt, Beranek, and Newman, one of the pioneers of the Internet and TCP/IP, originally developed GGP in the early 1980s. It was documented in RFC 823, “The DARPA Internet Gateway,” published September 1982. This protocol is now obsolete, but it played an important role in the early Internet by introducing certain concepts that developers used in later routing protocols.

GGP is similar in general operation to RIP (described in Chapter 38) in that it uses a distance-vector algorithm to determine the best routes between devices. Like RIP, the metric is a simple hop count, so GGP will select a route with the shortest number of hops. Although you have seen that hop count is not always the best metric of cost for a router in RIP, it was actually a pretty good method of route determination back then. This is because the early Internet used both computers and links that would be considered glacially slow by today’s standards, thereby making each hop fairly expensive (in terms of the time required to send data) compared to modern routing.

A router using GGP initially starts out in a null state. It then tests the status of its local networks by seeing if it can send and receive messages on the network. Every 15 seconds, the router sends a GGP Echo message to each of its neighbors. If the neighbor receives the message, it responds with a GGP Echo Reply message. The router sending the Echo messages considers the neighbor up if it receives replies to a certain percentage of messages (the default is 50 percent).

**NOTE** These messages serve a similar function to the Internet Control Message Protocol version 4 (ICMPv4) Echo and Echo Reply messages (described in Chapter 33), but are not the same.

Actual routing information is communicated by sending GGP Routing Update messages. These are similar in nature to RIP Response messages used in RIP. Each Routing Update message contains the information in the sending router’s routing table, which specifies the networks the router can reach and what the cost (in hops) will be for each.

A router that receives a Routing Update message knows that it can reach the router that sent the update. Because of that, it can reach all of the other routers’ reachable networks at the cost of an additional hop. The router uses the information to update its own internal tables of destinations and metrics, and then it sends out its own Routing Update on its own attached networks. This way, it can propagate the information it acquired from other routers on its own networks. This process continues until eventually, routes to all GGP routers spreads across the internetwork, just as this process occurs in RIP.

One interesting difference between GGP and RIP is that in GGP networks and costs aren't sent in pairs. Instead, a GGP router sends its routing table in groups. If it has three networks it can communicate with directly at a cost of 1, it sends those in a group with a distance value of 1. Next, if the GGP router has a few networks it can reach at a cost of 2, it sends those in a group with a distance value of 2, and so on.

Another difference is that GGP Routing Update messages are acknowledged. Each Routing Update message is sent with a sequence number, which ensures that out-of-date information is not propagated. If the Routing Update is received and it has a new sequence number (indicating that it contains recent information), the router processing returns a GGP Acknowledgment message to the originator. If the sequence number indicates the message is stale, a Negative Acknowledgment is sent instead and the message is discarded.

As a distance-vector algorithm using hop count as a metric, GGP shared most of the same pros and cons as RIP. It had simplicity on its side, but it had numerous problems such as slow convergence and the counting to infinity issue. GGP was a much more rudimentary protocol than RIP, however, and did not include many of the features included in RIP to handle such issues, such as split horizon. GGP was also limited to unsubnetted classful networks, due to its age.

When Internet architecture moved to the use of ASes, GGP was obsoleted. While it was an important part of TCP/IP history, it is today not formally considered a part of the TCP/IP protocol suite.

**KEY CONCEPT** The *Gateway-to-Gateway Protocol (GGP)* was used to communicate route information between core routers on the early Internet. It is a distance-vector protocol that operates in a manner that's very similar to RIP. Each router periodically sends out its routing table to neighboring routers so that each router can learn the cost, in hops, to reach every network in the AS. GGP is now considered a historical protocol and is no longer part of TCP/IP.

## The **HELLO** Protocol (**HELLO**)

The TCP/IP Internet as we know it today evolved over the course of decades. It began as an experimental research project started by the United States Defense Advanced Research Projects Agency (DARPA or ARPA). Called the ARPAnet, the project grew through the addition of other networks, such as the important NSFnet developed by the National Science Foundation (NSF). The NSFnet backbone grew over the course of many years and was instrumental to the eventual creation of the modern Internet.

The original NSFnet backbone consisted of six Digital Equipment Corporation (absorbed by Compaq years ago) LSI-11 computers located across the United States. These computers ran special software that was colloquially called "fuzzball." This software enabled the computers to function as routers. These fuzzball routers connected various networks to the NSFnet and the ARPAnet.

The six NSFnet routers worked as an AS, and like any AS, used an interior routing protocol to exchange routing information. The routing protocol used in these early routers was called the *HELLO Protocol*. Developed in the early 1980s, it was documented in RFC 891, "DCN Local-Network Protocols," which was published

in December 1983. The name *HELLO* is capitalized, but it is not an acronym; it simply refers to the word *hello*, because the protocol uses messages that are sort of analogous to the routers greeting each other.

**NOTE** *The OSPF routing protocol has a message type called Hello. The use of these messages is sometimes referred to as the Hello protocol. OSPF is not directly related to the HELLO Protocol described in this section, other than the fact that an AS could use both protocols for routing. OSPF may have borrowed the name Hello from the HELLO Protocol.*

The HELLO Protocol uses a distance-vector algorithm, like RIP and GGP. What's interesting about it, however, is that unlike RIP and GGP, HELLO does not use hop count as a metric. Instead, it attempts to select the best route by assessing network delays and choosing the path with the shortest delay.

One of the key jobs of routers that use HELLO is to compute the time delay required to send and receive datagrams to and from its neighbors. On a regular basis, routers exchange HELLO messages that contain clock and timestamp information. By using a special algorithm to compare the clock value and timestamp in the message to its own clock, a receiving device can compute an estimate for the amount of time it takes to send a datagram over the link.

Like RIP and GGP messages, HELLO messages also contain routing information in the form of pairs of destinations and metrics. These represent places that the sending router is able to reach and a cost to communicate with each one. However, in HELLO, the metric is an estimate of the round-trip delay cost for each destination. This information is added to the computed round-trip delay time for the link over which the message was received, and it is used to update the receiving router's own routing table.

This seems a bit confusing, but is really similar to the way a hop-count distance-vector protocol like RIP works. Router A, which is using RIP to receive an RIP Response message from Router B, knows it can reach every destination Router B can, but at a cost of one extra hop (the hop from Router A to Router B). Similarly, Router A, which receives a HELLO message from Router B, knows it can reach every destination that Router B can, but at an additional cost of the computed delay for the link between Router A and Router B.

In theory, using delay calculations should result in more efficient route selection than simply using a hop-count algorithm, but this comes at the cost of more complexity. This makes HELLO very interesting indeed, especially for a protocol that is more than 20 years old. However, since the latency of a link is often unrelated to its bandwidth, using time delay as a link metric may lead to spurious results.

Furthermore, it is normal for the delay on any link to vary over time; for example, if two routes are similar in cost, fluctuations in the delay for each route could result in rapid changes between routes (a phenomenon sometimes called *route flapping*). Adjustments are needed to the basic overview of the operation of the HELLO Protocol in order to avoid these sorts of problems.

Like other early routing protocols, HELLO does not include anything fancy like authentication. Such features were not needed in the early days of the Internet, when the internetworks were small and could easily be controlled. As the Internet grew, newer routing protocols such as RIP eventually replaced HELLO. It is now considered a historical protocol (in other words, obsolete) and is no longer used.

**KEY CONCEPT** The *HELLO Protocol* was used on very early routers on the precursors of the Internet to exchange routing information. It is a distance-vector protocol like RIP and GGP, but differs because it uses calculated delay instead of hop count as a metric. Like GGP, it is now considered a historical protocol and is no longer part of TCP/IP.

## Interior Gateway Routing Protocol (IGRP)

I greatly prefer universal, open standards to proprietary standards. I explain the reasons why in Chapter 3, which discusses networking standards. I am not alone in this view, and it's no exaggeration to say that much of the success of TCP/IP and the Internet is tied to the fact that they were both developed, and still are being developed, with the open RFC process.

That said, in certain situations, a proprietary protocol can be a benefit and can even achieve considerable success if a minimum of two factors is true:

- There is a lack of a suitable open protocol or a gap in the feature coverage of existing open protocols, creating an opportunity for a proprietary protocol to succeed.
- The proprietary protocol must be either initiated or strongly supported by a big player in the industry. This helps to ensure that other companies will take notice and give the protocol a chance to become a standard.

This situation arose in the 1980s in the world of routing protocols. At that time, the most popular interior routing protocol was RIP, which does a basically good job, but has a number of limitations and problems that are inherent to the protocol and are not easily resolved. In the mid-1980s, open alternatives like OSPF did not yet exist; even if they had, OSPF is much more complex than RIP and therefore sometimes not a good alternative to it.

Cisco Systems—definitely one of the big names in networking, internetworking, and routing—decided to develop a new routing protocol that would be similar to RIP but would provide greater functionality and solve some of RIP's inherent problems. Called the *Interior Gateway Routing Protocol (IGRP)*, it conveniently uses the words *gateway* and *routing* in its name, illustrating that these two words are used interchangeably in internetworking standards. Cisco designed it as a replacement for RIP. It is similar in many ways and keeps RIP's simplicity, one of its key strengths. At the same time, IGRP overcomes two key limitations of RIP: the use of hop count solely as a routing metric and the hop count limit of 15.

Like RIP, IGRP is a distance-vector routing protocol designed for use with an AS, and thus uses the same basic mechanism for route determination. Each router routinely sends out a message on each attached local network that contains a copy of its routing table. This message contains pairs of reachable networks and costs (metrics) to reach each network. A router receiving this message knows it can reach all the networks in the message as long as it can reach the router that sent the message. It computes the cost to reach those networks by adding what it costs to reach the router that sent the message to the networks' costs. The routers update their tables accordingly and send this information out in their next routine update. Eventually, each router in the AS will have information about the cost to reach each network in it.

There's an important difference between RIP and IGRP, however. RIP allows the cost to reach a network to be expressed only in terms of hop count; IGRP provides a much more sophisticated metric. In IGRP, the overall cost to reach a network is computed based on several individual metrics, including internetwork delay, bandwidth, reliability, and load. An administrator can customize the calculation of cost by setting relative weightings to the component metrics that reflect the priorities of that AS. So, if a particular administrator feels that emphasizing reliability over bandwidth would best minimize route cost, he can do this. Such a system provides tremendous flexibility over the rigid hop-count system of RIP. Unlike RIP, IGRP also does not have an inherent limit of 15 hops between networks.

To this basic algorithm, IGRP adds a feature called *multipath routing*. This allows multiple paths between routes to be used automatically, with traffic shared between them. The traffic can either be shared evenly or apportioned unevenly based on the relative cost metric of each path. This provides improved performance and flexibility.

Since IGRP is a distance-vector protocol like RIP, it shares many of RIP's algorithmic issues. Unsurprisingly, then, IGRP must incorporate many of the same stability features as RIP, including the use of split horizon, split horizon with poisoned reverse (in certain circumstances), and the employment of hold-down timers. Like RIP, IGRP also uses timers to control how often updates are sent, how long routers are held down, and how long routes are held in the routing table before they expire.

Cisco originally developed IGRP for Internet Protocol (IP) networks, and since IP is predominant in the industry, these networks are where it is most often seen. IGRP is not specific to IP, however, and can be used with other internetworking protocols if implemented for them. As you will see, Cisco also used IGRP as the basis for an improved routing protocol called EIGRP, which it developed several years after the original.

**KEY CONCEPT** In the 1980s, Cisco Systems created the *Interior Gateway Routing Protocol (IGRP)* as an improvement over the industry standard protocol, RIP. Like RIP, IGRP is a distance-vector protocol, but it includes several enhancements. Most important, it eliminates the 15-hop limit between routers and provides the ability to use metrics other than hop count to determine optimal routes.

## Enhanced Interior Gateway Routing Protocol (EIGRP)

As discussed in the previous section, IGRP represented a substantial improvement over RIP, but like any successful company, Cisco was not content to rest on its laurels. Cisco developers knew that IGRP had significant room for improvement, so they set to work on creating a better version of IGRP in the early 1990s. The result was the *Enhanced Interior Gateway Routing Protocol (EIGRP)*.

Compared to the original protocol, EIGRP is more of an evolution than a revolution. EIGRP is still a distance-vector protocol, but it is more sophisticated than other distance-vector protocols like IGRP or RIP, and it includes certain features that are more often associated with link-state routing protocols like OSPF

than distance-vector algorithms. Also, since the Cisco developers realized that many of the organizations that had decided to use EIGRP would be migrating to it from IGRP, they took special steps to ensure compatibility between the two.

The chief differences between IGRP and EIGRP are not in what they do, but how they do it. In an effort to improve the efficiency and speed of route convergence (that is, to improve the agreement between different routers in the internetwork), EIGRP changes the way that routes are calculated. EIGRP is based on a new route calculation algorithm called the *Diffusing Update Algorithm (DUAL)*, developed at SRI International by Dr. J. J. Garcia-Luna-Aceves.

DUAL differs from a typical distance-vector algorithm primarily in that it maintains more topology information about the internetwork than RIP or IGRP do. It uses this information to automatically select least-cost, loop-free routes between networks. EIGRP uses a metric that combines an assessment of the bandwidth of a link with the total delay to send over the link. (Other metrics are configurable as well, though not recommended.) When a neighboring router sends changed metric information, routes are recalculated and updates sent as needed. DUAL will query neighboring routers for reachability information if needed (for example, if an existing route fails).

This “as needed” aspect of operation highlights an important way that EIGRP improves performance over IGRP. EIGRP does not send routine route updates, but instead sends only partial updates as required, thereby reducing the amount of traffic generated between routers. Furthermore, these updates are designed so that only the routers that need the updated information receive them.

In order to build the tables of information that it needs to calculate routes, EIGRP requires routers to make and maintain contact with other routers on their local networks. To facilitate this, EIGRP incorporates a neighbor discovery and recovery process. This system involves the exchange of small Hello messages that let routers discover the other routers on the local network and periodically check to see whether they’re reachable. This is very similar to the way the identically named Hello messages are used in OSPF (as described in Chapter 39) and has a low impact on bandwidth use because the messages are small and infrequently sent.

Some of the features in IGRP carry through to its successor, such as the use of split horizon with poisoned reverse for improved stability. In addition to the basic improvements of efficiency and route convergence that accrue from the algorithm itself, EIGRP includes some other features. These include support for Variable Length Subnet Masks (VLSM) as well as support for multiple network-layer protocols. This means that EIGRP could be configured to function on a network that is running IP as well as another layer 3 protocol.

**KEY CONCEPT** Developed in the 1990s, the *Enhanced Interior Gateway Routing Protocol (EIGRP)* is an improved version of Cisco’s IGRP. It is similar to IGRP in many respects, but it uses a more sophisticated route calculation method called the *Diffusing Update Algorithm (DUAL)*. EIGRP also includes several features that make it more intelligent with regard to how it computes routes; it borrows concepts from link-state routing protocols and uses more efficient partial updates, rather than sending out entire routing tables.

## TCP/IP Exterior Gateway Protocol (EGP)

In the days of the early Internet, a small number of centralized core routers that maintained complete information about network reachability did the routing. These core routers exchanged information using the historical interior routing protocol, GGP, which we examined earlier in this chapter. Other noncore routers located around the periphery of this core, both stand-alone and in groups, exchanged network reachability information with the core routers using the first TCP/IP exterior routing protocol: the *Exterior Gateway Protocol (EGP)*.

Internet pioneers Bolt, Beranek, and Newman developed EGP in the early 1980s. It was first formally described in an Internet standard in RFC 827, “Exterior Gateway Protocol (EGP),” published in October 1982, which was later superseded by RFC 904, “Exterior Gateway Protocol Formal Specification,” in April 1984. Like GGP, EGP is now considered obsolete, having been replaced by BGP. However, like GGP, it is an important part of the history of TCP/IP routing, so it is worth examining briefly.

**NOTE** As I explained in Chapter 37, routers were in the past often called gateways. As such, exterior routing protocols were exterior gateway protocols. The EGP protocol discussed here is a specific instance of an exterior gateway protocol (also known as EGP). Thus, you may occasionally see BGP also called an exterior gateway protocol or an EGP, which is the generic use of the term.

EGP is responsible for the communication of network reachability information between neighboring routers that may or may not be in different ASes. The operation of EGP is somewhat similar to that of BGP (discussed in Chapter 40). Each EGP router maintains a database of information about which networks it can reach and how to reach them. It sends this information out on a regular basis to each router to which it is directly connected. Routers receive these messages and update their routing tables, and then use this new information to update other routers. Information about how to reach each network propagates across the entire internetwork.

The actual process of exchanging routing information involves several steps that discover neighbors and then set up and maintain communications. The steps are as follows:

1. **Neighbor Acquisition** Each router attempts to establish a connection to each of its neighboring routers by sending Neighbor Acquisition Request messages. A neighbor hearing a request can respond with a Neighbor Acquisition Confirm message, which says that it recognized the request and wishes to connect. It may reject the acquisition by replying with a Neighbor Acquisition Refuse message. For an EGP connection to be established between a pair of neighbors, each message must first successfully acquire the other with a Confirm message.

2. **Neighbor Reachability** After acquiring a neighbor, a router checks to make sure the neighbor is reachable and functioning properly on a regular basis. This is done by sending an EGP Hello message to each neighbor for which a connection has been established. The neighbor replies with an I Heard You (IHU) message. These messages are somewhat analogous to the BGP Keepalive message, but they are used in matched pairs.
3. **Network Reachability Update** A router sends Poll messages on a regular basis to each of its neighbors. The neighbor responds with an Update message, which contains details about the networks that it is able to reach. This information is used to update the routing tables of the device that sent the Poll message.

A neighbor can decide to terminate a connection (called *neighbor deacquisition*) by sending a Cease message; the neighbor responds with a Cease-ack (acknowledge) message.

An Error message, similar to the BGP Notification message in role and structure (see Chapter 40), is also defined. A neighbor may send this message in response to the receipt of an EGP message either when the message itself has a problem (such as a bad message length or unrecognized data in a field) or to indicate a problem with how the message is being used (such as receipt of Hello or Poll messages at a rate deemed excessive). Unlike with the BGP Notification message, an EGP router does not necessarily close the connection when sending an Error message.

**KEY CONCEPT** The *Exterior Gateway Protocol (EGP)* was the first TCP/IP exterior routing protocol and was used with GGP on the early Internet. It functions in a manner similar to BGP. For example, an EGP router makes contact with neighboring routers and exchanges routing information with them. A mechanism is also provided to maintain a session and report errors. EGP is more limited than BGP in capability and is now considered a historical protocol.

The early Internet was designed to connect peripheral routers or groups of routers to the Internet core. It was therefore designed under the assumption that the internetwork was connected as a hierarchical tree, with the core as the root. EGP was designed based on this assumption of a tree structure and, for that reason, cannot handle an arbitrary topology of ASes like BGP. It likewise cannot guarantee the absence of routing loops if such loops exist in the interconnection of neighboring routers. This is part of why BGP needed to be developed as the Internet moved to a more arbitrary structure of AS connections, where loops would be possible if steps weren't taken to avoid them.



# PART II-8

## TCP/IP TRANSPORT LAYER PROTOCOLS

The first three layers of the OSI Reference Model—the physical layer, data link layer, and network layer—are very important layers for understanding how networks function. The physical layer moves bits over wires; the data link layer moves frames on a network; and the network layer moves datagrams on an internetwork. Taken as a whole, they are the parts of a protocol stack that are responsible for the actual nuts and bolts of getting data from one place to another.

Immediately above these three layers is the fourth layer of the OSI Reference Model: the *transport layer*, called the host-to-host transport layer in the TCP/IP model. This layer is interesting in that it resides in the very architectural center of the model. Accordingly, it represents an important transition point between the hardware-associated layers below it that do the grunt work and the layers above that are more software-oriented and abstract.

Protocols running at the transport layer are charged with providing several important services to enable software applications in higher layers to work over an internetwork. They are typically responsible for allowing connections to be established and maintained between software services on possibly distant machines. Many higher-layer applications need to send data in a reliable way, without needing to worry about error correction, lost data, or flow management. However, network layer protocols are typically unreliable and

unacknowledged. Transport layer protocols are often very tightly tied to the network layer protocols directly below them and designed specifically to take care of functions that are not dealt with by those protocols.

This part describes transport layer protocols and related technologies used in the TCP/IP protocol suite. There are two main protocols at this layer: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). UDP is the simpler of the two and doesn't take a great deal of time to explain. In contrast, TCP is a rather complex protocol that is also a very important part of the TCP/IP protocol suite, and thus it requires considerably more explanation.

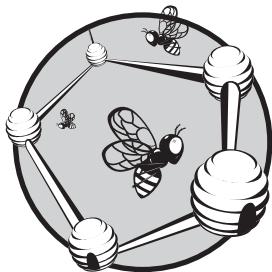
The first chapter in this part provides a quick overview of the roles of these two protocols in the TCP/IP protocol suite, a discussion of why they are both important, and a summary that compares their key attributes. The second chapter describes the method that both protocols employ for addressing, using transport layer ports and sockets. The third chapter contains a discussion of UDP.

The remaining five chapters encompass a comprehensive description of the concepts, characteristics, and functions of TCP. The fourth chapter in this part provides an overview of TCP, describing its history, what it does, and how it works. The fifth chapter covers some important background information that is necessary to understanding how TCP operates, explaining key concepts such as streams and segments, sliding windows, and TCP ports and connections. The sixth chapter describes the process used by TCP to establish, maintain, and terminate sessions. The seventh chapter describes TCP messages and how they are formatted and transferred. Finally, the last chapter in this part shows how TCP provides reliability and other important transport layer functions, such as flow control, retransmission of lost data, and congestion avoidance.

Since TCP is built on top of the Internet Protocol (IP), in describing TCP, I assume that you have at least a basic familiarity with IP (covered in Part II-3; specifically, see Chapters 15 and 16 for descriptions of basic IP concepts).

# 42

## OVERVIEW AND COMPARISON OF TCP AND UDP



TCP/IP is the most important inter-networking protocol suite in the world. It is the basis for the Internet and the “language” spoken by the vast majority of the world’s networked computers. TCP/IP includes a large set of protocols that operate at the network layer and the layers above it. The suite as a whole is anchored at layer 3 by the Internet Protocol (IP), which many people consider the single most important protocol in the world of networking.

Of course, there’s a bit of *architectural distance* between the network layer and the applications that run at the layers well above that layer. IP is the protocol that performs the bulk of the functions needed to make an internetwork work, but it does not include some capabilities that many applications need. In TCP/IP, a pair of protocols that operate at the transport layer performs these tasks. The protocols are the *Transmission Control Protocol (TCP)* and the *User Datagram Protocol (UDP)*.

Of these two, TCP gets the most attention. It is the transport layer protocol that is most often associated with TCP/IP. It is also the transport protocol that many of the Internet's most popular applications use.

UDP, on the other hand, gets second billing. However, UDP and TCP are really peers that play the same role in TCP/IP. They function very differently and provide different benefits for and drawbacks to the applications that use them. Yet they are both important to the protocol suite as a whole. This chapter introduces what TCP and UDP do and highlights the similarities and differences between them.

## Two Protocols for TCP/IP Transport Layer Requirements

The transport layer in a protocol suite is responsible for a specific set of functions. For this reason, you might expect that the TCP/IP suite would have a single main transport protocol that performs those functions, just as it has IP as its core protocol at the network layer. But there are *two* different widely used TCP/IP transport layer protocols, an arrangement that is probably one of the best examples of the power of protocol layering (showing that it was worth all the time you spent learning to understand that pesky OSI Reference Model back in Chapters 5 through 7).

Let's start with a look back at layer 3. In my overview of the key operating characteristics of IP in Chapter 15, I described several limitations of IP. The most important limitations are that IP is *connectionless*, *unreliable*, and *unacknowledged*. Using a best-effort paradigm, data is sent over an IP internetwork without first establishing a connection. Messages *usually* get where they need to go, but there are no guarantees, and the sender usually doesn't even know if the data arrived at its destination.

These characteristics present serious problems for software. Many, if not most, applications need to be able to count on the fact that the data they send will get to its destination without loss or error. Applications also want the connection between two devices to be automatically managed, with problems such as congestion and flow control taken care of as needed. Unless some mechanism is provided for this at lower layers, every application would need to perform these jobs, and that would be a massive duplication of effort.

In fact, you might argue that establishing connections, providing reliability, and handling retransmissions, buffering, and data flow are sufficiently important that it might have been best to simply build these abilities directly into IP. Interestingly, that was exactly the case in the early days of TCP/IP. In the beginning there was just a single protocol called TCP. It combined the tasks of IP with the reliability and session management features that I just mentioned. There's a big problem with this, however: Establishing connections, providing a mechanism for reliability, managing flow control, managing acknowledgments, and managing retransmissions all come at a cost of time and bandwidth. Building all of these capabilities into a single protocol that spanned layers 3 and 4 would mean that all applications would receive the benefits of reliability, but would also take on the costs. While this approach would be fine for many applications, there are others that either don't need the reliability or can't afford the overhead required to provide it.

The solution was simple: Let the network layer (IP) take care of basic data movement on the internetwork, and define two protocols at the transport layer. One protocol would provide a rich set of services for applications that need that functionality, and the understanding would be that some overhead would be required when using this protocol. The other protocol would be simpler, providing little in the way of classic layer 4 functions, but it would be fast and easy to use. Thus, the result was two TCP/IP transport layer protocols:

**Transmission Control Protocol (TCP)** TCP is a full-featured, connection-oriented, reliable transport protocol for TCP/IP applications. It provides transport layer addressing that allows multiple software applications to simultaneously use a single IP address, and it allows a pair of devices to establish a virtual connection and then pass data bidirectionally. Transmissions are managed using a special *sliding window* system, with unacknowledged transmissions detected and automatically retransmitted. Additional functionality allows the flow of data between devices to be managed, and special circumstances to be addressed.

**User Datagram Protocol (UDP)** In contrast, UDP is a very simple transport protocol that provides transport layer addressing like TCP, but little else. UDP is barely more than a wrapper protocol that provides a way for applications to access IP. No connection is established, transmissions are unreliable, and data can be lost.

**KEY CONCEPT** Many TCP/IP applications require different transport requirements, thus two TCP/IP transport layer protocols are necessary. The *Transmission Control Protocol (TCP)* is a full-featured, connection-oriented protocol that provides the acknowledged delivery of data while managing traffic flow and handling issues such as congestion and transmission loss. The *User Datagram Protocol (UDP)*, in contrast, is a much simpler protocol that concentrates only on delivering data in order to maximize the speed of communication when the features of TCP are not required.

## Applications of TCP and UDP

To use an analogy, TCP is a fully loaded luxury performance sedan with a chauffeur and a satellite tracking/navigation system. It provides a lot of frills, comfort, and performance. It virtually guarantees that you will get where you need to go without any problems, and any concerns that do arise can be corrected. In contrast, UDP is a stripped-down race car. Its goal is simplicity and speed; everything else is secondary. You will probably get where you need to go, but you can have trouble keeping race cars up and running.

Having two transport layer protocols with such complementary strengths and weaknesses provides considerable flexibility to the creators of networking software.

### TCP Applications

Most typical applications need the reliability and other services provided by TCP, and most applications don't care about the loss of a small amount of performance due to TCP's overhead requirements. For example, most applications that transfer files or important data between machines use TCP, because the loss of any portion

of the file renders the data useless. Examples include such well-known applications as the Hypertext Transfer Protocol (HTTP), which is used by the World Wide Web (WWW), the File Transfer Protocol (FTP), and the Simple Mail Transfer Protocol (SMTP). I describe TCP applications in more detail in Section III.

## **UDP Applications**

What sort of application doesn't care if its data gets there, and why would anyone want to use such an unreliable application? You might be surprised. A lot of TCP/IP protocols use UDP. It is a good match when the application doesn't really care if some of the data gets lost, such as if you are streaming video or multimedia. The application won't notice one lost byte of data. UDP is also a good match when the application itself chooses to provide some other mechanism to make up for the lack of functionality in UDP.

Applications that send very small amounts of data often use UDP and assume that the client will just send a new request later on if a request is sent and a reply is not received. This provides enough reliability without the overhead of a TCP connection. I discuss some common UDP applications in Chapter 44.

**KEY CONCEPT** Most typical applications, especially ones that send files or messages, require that data be delivered reliably, and therefore use TCP for transport. The loss of a small amount of data usually is not a concern to applications that use UDP or that use their own application-specific procedures for dealing with potential delivery problems.

Note that even though TCP is often described as being *slower* than UDP, this is a *relative* measurement. TCP is a very well-written protocol that is capable of highly efficient data transfers. It is slow only compared to UDP because of the overhead of establishing and managing connections. The difference can be significant, but it is not enormous.

Incidentally, if you want a good real-world illustration of why it's valuable to have both UDP and TCP, consider message transport under the Domain Name System (DNS). As described in Chapter 57, DNS actually uses UDP for certain types of communication and TCP for others.

## **Summary Comparison of UDP and TCP**

In the next few chapters, we will explore both UDP and TCP in further detail. I will help you to understand much better the strengths and drawbacks of both protocols. While informative, these chapters are time-consuming to read. Thus, for your convenience, I have included Table 42-1, which describes the most important attributes of both protocols and how they contrast with each other.

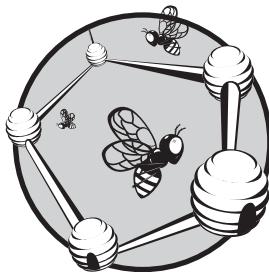
**Table 42-1:** Summary Comparison of UDP and TCP

<b>Characteristic/ Description</b>	<b>UDP</b>	<b>TCP</b>
General Description	Simple, high-speed, low-functionality wrapper that interfaces applications to the network layer and does little else	Full-featured protocol that allows applications to send data reliably without worrying about network layer issues
Protocol Connection Setup	Connectionless; data is sent without setup	Connection-oriented; connection must be established prior to transmission
Data Interface to Application	Message-based; the application sends data in discrete packages	Stream-based; the application sends data with no particular structure
Reliability and Acknowledgments	Unreliable, best-effort delivery without acknowledgments	Reliable delivery of messages; all data is acknowledged
Retransmissions	Not performed; application must detect lost data and retransmit if needed	Delivery of all data is managed, and lost data is retransmitted automatically
Features Provided to Manage Flow of Data	None	Flow control using sliding windows; window size adjustment heuristics; congestion-avoidance algorithms
Overhead	Very low	Low, but higher than UDP
Transmission Speed	Very high	High, but not as high as UDP
Data Quantity Suitability	Small to moderate amounts of data (up to a few hundred bytes)	Small to very large amounts of data (up to a few gigabytes)
Types of Applications That Use the Protocol	Applications where data delivery speed matters more than completeness, where small amounts of data are sent, or where multicast/broadcast are used	Most protocols and applications sending data that must be received reliably, including most file and message transfer protocols
Well-Known Applications and Protocols	Multimedia applications, DNS, BOOTP, DHCP, TFTP, SNMP, RIP, NFS (early versions)	FTP, Telnet, SMTP, DNS, HTTP, POP, NNTP, IMAP, BGP, IRC, NFS (later versions)



# 43

## TCP AND UDP ADDRESSING: PORTS AND SOCKETS



Internet Protocol (IP) addresses are the main form of addressing used on a TCP/IP network. These network layer addresses uniquely identify each network interface, and as such, they serve as the mechanism by which data is routed to the correct network on the internetwork and then to the correct device on that network.

But there is an additional level of addressing that occurs at the transport layer in TCP/IP, above that of the IP address. Both of the TCP/IP transport protocols—the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP)—use the concepts of *ports* and *sockets* for virtual software addressing. Ports and sockets enable many applications to function simultaneously on an IP device.

In this chapter, I describe the special mechanism used for addressing in both TCP and UDP. I begin with a discussion of TCP/IP application processes, including the client/server nature of communication, which provides a background for explaining how ports and sockets are used. I then give an overview of the concept of ports and how they enable the multiplexing of data over an

IP address. I describe the way that port numbers are categorized in ranges and assigned to server processes for common applications. I explain the concept of ephemeral port numbers used for clients. I then discuss sockets and their use for connection identification, including the means by which multiple devices can talk to a single port on another device. I then provide a summary table of the most common registered port numbers.

## **TCP/IP Processes, Multiplexing, and Client/Server Application Roles**

The most sensible place to start learning about how the TCP/IP protocol suite works is by examining IP itself, as well as the support protocols that function in tandem with it at the network layer. IP is the foundation upon which most of the rest of TCP/IP is built. It is the mechanism by which data is packaged and routed throughout a TCP/IP internetwork.

It makes sense, then, that when we examine the operation of TCP/IP from the perspective of IP, we talk very generically about sending and receiving datagrams. To the IP layer software that sends and receives IP datagrams, the higher-level application that datagrams come from and go to is really unimportant. To IP, a datagram is a datagram. All datagrams are packaged and routed in the same way, and IP is mainly concerned with lower-level aspects of moving them between devices in an efficient manner. It's important to remember, however, that this is really an abstraction for the convenience of describing a layer 3 operation. It doesn't consider how datagrams are really generated and used above layer 3.

Layer 4 represents a transition point between the OSI model hardware-related layers (1, 2, and 3) and the software-related layers (5 to 7). This means that the TCP/IP transport layer protocols, TCP and UDP, need to pay attention to the way that software uses TCP/IP, even if IP really does not.

Ultimately, the entire point of having networks, internetworks, and protocol suites like TCP/IP is to enable networking *applications*. Most Internet users employ these applications on a daily basis. In fact, most of us run many different applications simultaneously. For example, you might use a web browser to check the news, a File Transfer Program (FTP) client to upload some pictures to share with family, and an Internet Relay Chat (IRC) program to discuss something with a friend or colleague. In fact, it is common to have multiple instances of a single application. The most common example is having multiple web browser windows open (I sometimes find myself with as many as 30 going at one time!).

### ***Multiplexing and Demultiplexing***

Most communication in TCP/IP takes the form of exchanges of information between a program running on one device and a matching program running on another device. Each instance of an application represents a copy of that application's software that needs to send and receive information. These application instances are commonly called *processes*. A TCP/IP application process is any piece of networking software that sends and receives information using the TCP/IP protocol suite. This includes classic end-user applications such as the ones

described earlier, and support protocols that behave like applications when they send messages. Examples of the latter would include a network management protocol like the Simple Network Management Protocol (SNMP; see Chapters 65 through 69), or even the routing protocol Border Gateway Protocol (BGP; see Chapter 40), which sends messages using TCP the way an application does.

So, a typical TCP/IP host has multiple processes, and each one needs to send and receive datagrams. All of these datagrams, however, must be sent using the same interface to the internetwork, using the IP layer. This means that the data from all applications (with some possible exceptions) is initially funneled down to the transport layer, where TCP or UDP handles it. From there, messages pass to the device's IP layer, where they are packaged in IP datagrams and sent out over the internetwork to different destinations. The technical term for this is *multiplexing*. This term simply means combining, and its use here is a software analog to the way multiplexing is done with signals (such as how individual telephone calls are packaged).

A complementary mechanism is responsible for the receipt of datagrams. At the same time that the IP layer multiplexes datagrams to send from many application processes, it receives many datagrams that are intended for different processes. The IP layer must take this stream of unrelated datagrams and pass them to the correct process (through the transport layer protocol above it). This is *demultiplexing*, the opposite of multiplexing.

You can see an illustration of the concept behind TCP/IP process multiplexing and demultiplexing in Figure 43-1.

**KEY CONCEPT** TCP/IP is designed to allow many different applications to send and receive data simultaneously using the same IP software on a given device. To accomplish this, it is necessary to *multiplex* transmitted data from many sources as it is passed down to the IP layer. As a stream of IP datagrams is received, it is *demultiplexed* and the appropriate data passed to each application software instance on the receiving host.

## TCP/IP Client Processes and Server Processes

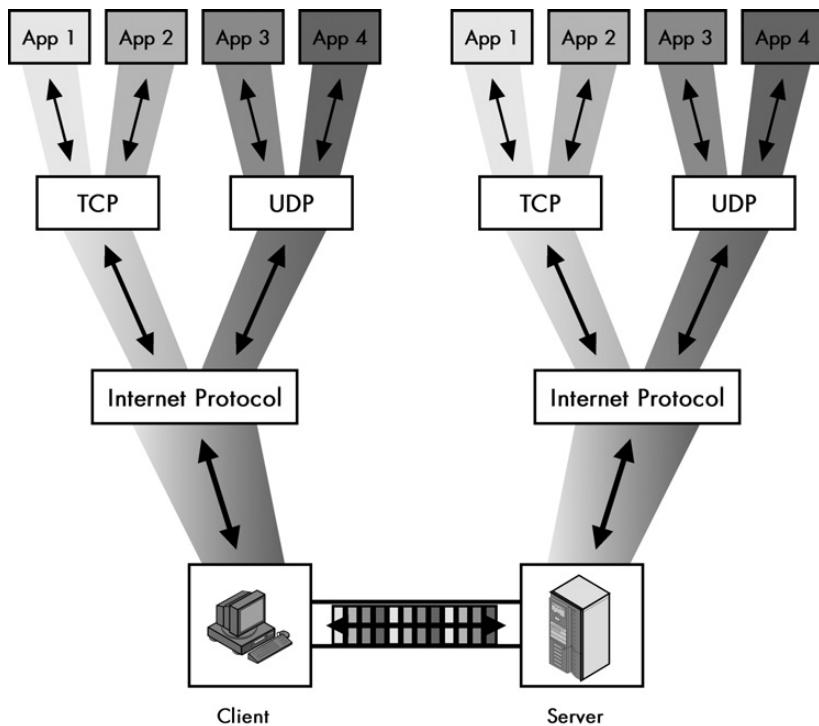
TCP/IP software is generally *asymmetric*. This means that when a TCP/IP application process on one computer tries to talk to an application process on another computer, the two processes are usually not exactly the same. They are instead *complements* of each other, designed to function together as a team.

As I explained in the overview description of TCP/IP in Chapter 8, most networking applications use a *client/server* model of operation. This term can be used to refer to the roles of computers, where a *server* is a relatively powerful machine that provides services to a large number of user-operated *clients*. It also applies to software. In a software context, a *client process* usually runs on a client machine and initiates contact to perform some sort of function. A *server process* usually runs on a hardware server, listens for requests from clients, and responds to them.

The classic example of this client/server operation is the World Wide Web (WWW). The Web uses the Hypertext Transfer Protocol (HTTP; see Chapter 80), which is a good example of an application protocol. A web browser is an HTTP client that normally runs on an end-user client machine. It initiates an exchange of HTTP

(web) data by sending a request to a web (HTTP) server. A server process on that web server hears the request and replies either with the requested item(s)—a web page or other data—or an error message. The server is usually specifically designed to handle many incoming client requests, and in many cases, has no other use.

Why am I telling you all of this in a section that is supposed to explain TCP and UDP ports? I started here because many application processes run simultaneously and have their data multiplexed for transmission. The simultaneity of application processes and the multiplexing of data are the impetus for why higher-level addressing is a necessity in TCP/IP. The client/server arrangement used by TCP/IP has an important impact on the way that ports are used and the mechanisms for how they are assigned. The next two sections explore these concepts more completely.



**Figure 43-1: Process multiplexing and demultiplexing in TCP/IP** In a typical machine that is running TCP/IP, there are many different protocols and applications running simultaneously. This example shows four different applications communicating between a client and server machine. All four are multiplexed for transmission using the same IP software and physical connection; received data is demultiplexed and passed to the appropriate application. IP, TCP, and UDP provide a way of keeping the data from each application distinct.

## TCP/IP Ports: TCP/UDP Addressing

A typical host on a TCP/IP internetwork has many different software application processes running concurrently. Each process generates data that it sends to either TCP or UDP, which then passes it to IP for transmission. The IP layer sends out this multiplexed stream of datagrams to various destinations. Simultaneously, each

device's IP layer is receiving datagrams that originated in numerous application processes on other hosts. These datagrams need to be demultiplexed so that they end up at the correct process on the device that receives them.

## ***Multiplexing and Demultiplexing Using Ports***

The question is how do we demultiplex a sequence of IP datagrams that need to go to many different application processes? Let's consider a particular host with a single network interface bearing the IP address 24.156.79.20. Normally, every datagram received by the IP layer will have this value in the IP Destination Address field. The consecutive datagrams that IP receives may contain a piece of a file you are downloading with your web browser, an email your brother sent to you, and a line of text a buddy wrote in an IRC chat channel. How does the IP layer know which datagrams go where if they all have the same IP address?

The first part of the answer lies in the Protocol field included in the header of each IP datagram. This field carries a code that identifies the protocol that sent the data in the datagram to IP. Since most end-user applications use TCP or UDP at the transport layer, the Protocol field in a received datagram tells IP to pass data to either TCP or UDP as appropriate. Of course, this just defers the problem to the transport layer.

Many applications use both TCP and UDP at once. This means that TCP or UDP must figure out which process to send the data to. To make this possible, an additional addressing element is necessary. This address allows a more specific location—a software process—to be identified within a particular IP address. In TCP/IP, this transport layer address is called a *port*.

## ***Source Port and Destination Port Numbers***

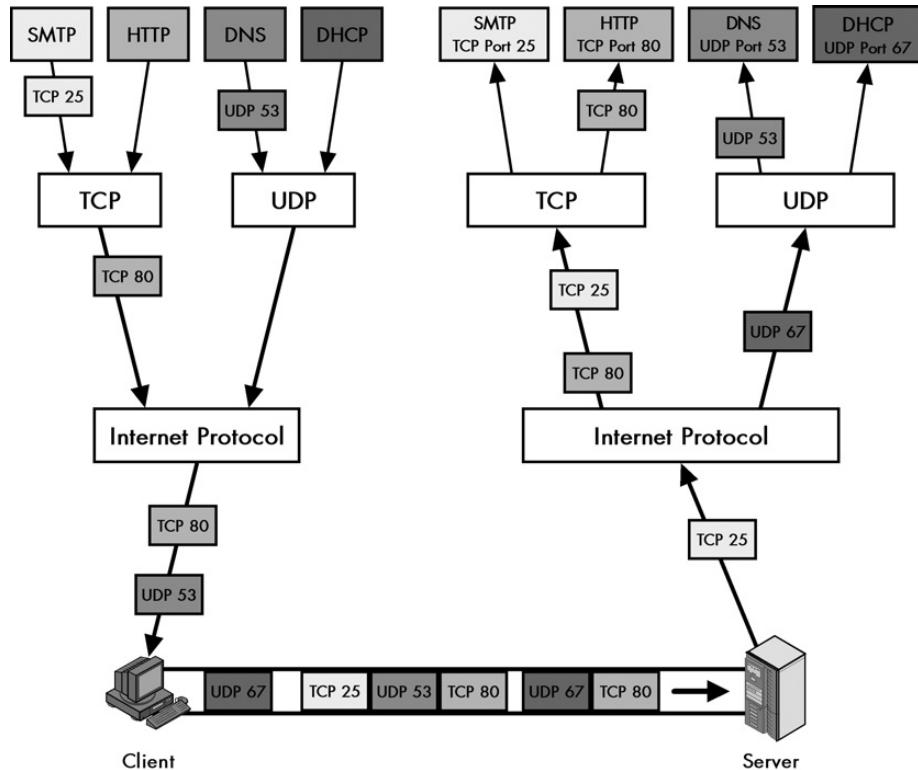
In UDP and TCP messages two addressing fields appear: a *source port* and a *destination port*. These are analogous to the source address and destination address fields at the IP level, but at a higher level of detail. They identify the originating process on the source machine and the destination process on the destination machine. The TCP or UDP software fills them in before transmission, and they direct the data to the correct process on the destination device.

**NOTE** *The term port has many meanings aside from this one in TCP/IP. For example, a physical outlet in a network device is often called a port. Usually, you can discern whether the port in question refers to a hardware port or a software port from the context.*

TCP and UDP port numbers are 16 bits in length. Valid port numbers can theoretically take on values from 0 to 65,535. As you will see in the next section, these values are divided into ranges for different purposes, with certain ports reserved for particular uses.

One fact that is sometimes a bit confusing is that both UDP and TCP use the same range of port numbers, but they are independent. In theory, it is possible for UDP port number 77 to refer to one application process and TCP port number 77 to refer to an entirely different one. There is no ambiguity, at least to the computers, because as mentioned earlier, each IP datagram contains a Protocol field that specifies whether it is carrying a TCP message or a UDP message. IP passes the

datagram to either TCP or UDP, which sends the message on to the right process using the port number in the TCP or UDP header. This mechanism is illustrated in Figure 43-2.



**Figure 43-2: TCP/IP process multiplexing/demultiplexing using TCP/UDP ports** A more concrete version of Figure 43-1, this figure shows how TCP and UDP ports accomplish software multiplexing and demultiplexing. Again there are four different TCP/IP applications communicating, but this time I am showing only the traffic going from the client to the server. Two of the applications use TCP, and two use UDP. Each application on the client sends messages using a specific TCP or UDP port number. The server's UDP or TCP software uses these port numbers to pass the datagrams to the appropriate application process.

In practice, having TCP and UDP use different port numbers is confusing, especially for the reserved port numbers that common applications use. To avoid confusion, by convention, most reserved port numbers are reserved for both TCP and UDP. For example, port 80 is reserved for HTTP for both TCP and UDP, even though HTTP only uses TCP. We'll examine this in greater detail in the following section.

**KEY CONCEPT** TCP/IP transport layer addressing is accomplished by using TCP and UDP ports. Each port number within a particular IP device identifies a particular software process.

## **Summary of Port Use for Datagram Transmission and Reception**

Here's how transport layer addressing (port addressing) works in TCP and UDP:

**Sending Datagrams** An application specifies the source and destination port it wishes to use for the communication. The port numbers are encoded into the TCP or UDP header, depending on which transport layer protocol the application is using. When TCP or UDP passes data to IP, IP indicates the protocol type that's appropriate for TCP or UDP in the Protocol field of the IP datagram. The source and destination port numbers are encapsulated as part of the TCP or UDP message, within the IP datagram's data area.

**Receiving Datagrams** The IP software receives the datagram, inspects the Protocol field, and decides which protocol the datagram belongs to (in this case, TCP or UDP). TCP or UDP receives the datagram and passes its contents to the appropriate process based on the destination port number.

**KEY CONCEPT** Application process multiplexing and demultiplexing in TCP/IP is implemented using the IP Protocol field and the UDP/TCP Source Port and Destination Port fields. Upon transmission, the Protocol field is given a number to indicate whether TCP or UDP was used, and the port numbers are filled in to indicate the sending and receiving software process. The device receiving the datagram uses the Protocol field to determine whether TCP or UDP was used and then passes the data to the software process that the destination port number indicates.

## **TCP/IP Application Assignments and Server Port Number Ranges**

The port numbers I just discussed provide a method of transport layer addressing that allows many applications to use TCP and UDP simultaneously. By specifying the appropriate destination port number, an application sending data can be sure that the right process on the destination device will receive the message. Unfortunately, there's still a problem to be solved.

Let's go back to using the World Wide Web. You fire up a web browser, which is client software that sends requests using HTTP. You need to know the IP address of the website you want to access, or you may have the Domain Name System (DNS) supply the IP address to you automatically. Once you have the address, the web browser can generate an HTTP message and send it to the website's IP address.

This HTTP message is intended for the web server process on the site you are trying to reach. The problem is how does the web browser (client process) know which port number has been assigned to the server process on the website? Port numbers can range from 0 to 65535, which means a lot of choices. And, in theory, every website could assign a different port number to its web server process.

There are a couple of different ways to resolve this problem. TCP/IP takes what is probably the simplest possible approach: It *reserves* certain port numbers for particular applications.

## **Reserved Port Numbers**

Server processes, which listen for requests for that application and then respond to them, assign each common application a specific port number. To avoid chaos, the software that implements a particular server process normally uses the same reserved port number on every IP device so that clients can find it easily.

In the example of accessing a website with a web browser, the reserved port number for HTTP is 80. Every web browser knows that web designers design websites to listen for requests sent to port 80. The web browser will thus use this value in requests to ensure that the IP and TCP software on the web browser directs these HTTP messages to the web server software. It is possible for a particular web server to use a different port number, but in this case, the web server must inform the user of this number somehow, and must explicitly tell the web browser to use it instead of the default port number (80).

**KEY CONCEPT** To allow client devices to establish connections to TCP/IP servers more easily, server processes for common applications use universal server port numbers. Clients are preprogrammed to know to use the port numbers by default.

In order for this system to work well, universal agreement on port assignments is essential. Thus, this becomes another situation where a central authority is needed to manage a list of port assignments that everyone uses. For TCP/IP, it is the same authority responsible for the assignment and coordination of other centrally managed numbers, including IP addresses, IP protocol numbers, and so forth: the Internet Assigned Numbers Authority (IANA; see Chapter 3).

## **TCP/UDP Port Number Ranges**

As you have seen, there are 65,536 port numbers that can be used for processes. But there are also a fairly large number of TCP/IP applications, and the list grows every year. IANA needs to carefully manage the port number address space in order to ensure that port numbers are not wasted on protocols that won't be widely used. IANA also needs to provide flexibility for organizations that must make use of obscure applications. To this end, the full spectrum of TCP and UDP port numbers is divided into three ranges:

**Well-Known (Privileged) Port Numbers (0 to 1023)** IANA manages these port numbers and reserves them for only the most universal TCP/IP applications. The IANA assigns these port numbers only to protocols that have been standardized using the TCP/IP RFC process, protocols that are in the process of being standardized, or protocols that are likely to be standardized in the future. On most computers, only server processes run by system administrators or privileged users use these port numbers. These processes generally correspond to processes that implement key IP applications, such as web servers, FTP servers, and the like. For this reason, these processes are sometimes called *system port numbers*.

**Registered (User) Port Numbers (1024 to 49151)** There are many applications that need to use TCP/IP, but are not specified in RFCs or are not as universally used as other applications, so they do not warrant a worldwide well-known port number. To ensure that these various applications do not conflict with each other, IANA uses the bulk of the overall port number range for registered port numbers. Anyone who creates a viable TCP/IP server application can request to reserve one of these port numbers, and if the request is approved, the IANA will register that port number and assign it to the application. Any user on a system can generally access registered port numbers; thus they are sometimes called *user port numbers*.

**Private/Dynamic Port Numbers (49152 to 65535)** IANA neither reserves nor maintains these ports. Anyone can use them for any purpose without registration, so they are appropriate for a private protocol that only a particular organization uses.

**KEY CONCEPT** IANA manages port-number assignments to ensure universal compatibility around the global Internet. The numbers are divided into three ranges: well-known port numbers used for the most common applications, registered port numbers for other applications, and private/dynamic port numbers that can be used without IANA registration.

Use of these ranges ensures that there will be universal agreement on how to access a server process for the most common TCP/IP protocols. They also allow flexibility for special applications. Most of the TCP/IP applications and application protocols use numbers in the well-known port number range for their servers. These port numbers are not generally used for client processes, but there are some exceptions. For example, port 68 is reserved for a client using the Bootstrap Protocol (BOOTP) or Dynamic Host Configuration Protocol (DHCP).

## TCP/IP Client (Ephemeral) Ports and Client/Server Application Port Use

The significance of the asymmetry between clients and servers in TCP/IP becomes evident when you examine in detail how port numbers are used. Since clients initiate application data transfers using TCP and UDP, they need to know the port number of the server process. Consequently, servers are required to use universally known port numbers. Thus, well-known and registered port numbers identify server processes. Clients that send requests use the well-known or registered port number as the destination port number.

In contrast, servers respond to clients; they do not initiate contact with them. Thus, the client doesn't need to use a reserved port number. In fact, this is really an understatement. A server shouldn't use a well-known or registered port number to send responses back to clients because it is possible for a particular device to have client and server software from the same protocol running on the same machine. If a server received an HTTP request on port 80 of its machine and sent the reply back to port 80 on the client machine, the server would be sending the reply to the client machine's HTTP server process (if present), rather than the client process that sent the initial request.

To know where to send the reply, the server must know the port number the client is using. The client supplies the port number as the *source port* in the request, and then the server uses the source port as the destination port to send the reply. Client processes don't use well-known or registered ports. Instead, each client process is assigned a temporary port number for its use. This is commonly called an *ephemeral port number*.

**NOTE** Your \$10 word for the day: ephemeral: "short-lived; existing or continuing for a short time only." — Webster's Revised Unabridged Dictionary.

## Ephemeral Port Number Assignment

The TCP/IP software assigns ephemeral port numbers as needed to processes. Obviously, each client process that's running concurrently needs to use a unique ephemeral port number, so the TCP and UDP layers must keep track of which ones are in use. The TCP/IP software generally assigns these port numbers in a *pseudo-random* manner from a reserved pool of numbers. I say pseudo-random because there is no specific meaning to an ephemeral port number that is assigned to a process, so the TCP/IP software could select a random one for each client process. However, since it is necessary to reuse the port numbers in this pool over time, many implementations use a set of rules to minimize the chance of confusion due to reuse.

Consider a client process that used only ephemeral port number 4121 to send a request. The client process received a reply and then terminated. Suppose you immediately reallocate 4121 to some other process. However, the prior user of port 4121 accesses the server, which for some reason sends an extra reply. The reply would go to the new process, thereby creating confusion. To avoid this, it is wise to wait as long as possible before reusing port number 4121 for another client process. Some implementations will therefore cycle through the port numbers in order to ensure that the maximum amount of time elapses between consecutive uses of the same ephemeral port number.

**KEY CONCEPT** Well-known and registered port numbers are needed for server processes since a client must know the server's port number to initiate contact. On the other hand, client processes can use any port number. Each time a client process initiates a UDP or TCP communication, the TCP/IP software assigns it a temporary, or *ephemeral*, port number to use for that conversation. The TCP/IP software assigns these port numbers in a *pseudo-random* way because the exact number that the software uses is not important as long as each process has a different number.

## Ephemeral Port Number Ranges

The range of port numbers that TCP/IP software uses for ephemeral ports on a device also depends on the implementation. The TCP/IP implementation in Berkeley Standard Distribution (BSD) UNIX established the classic ephemeral port range. BSD UNIX defined it as 1024 to 4999, thereby providing 3,976 ephemeral ports. This seems like a very large number, and it is indeed usually more than enough for a typical client. However, the size of this number can be deceiving.

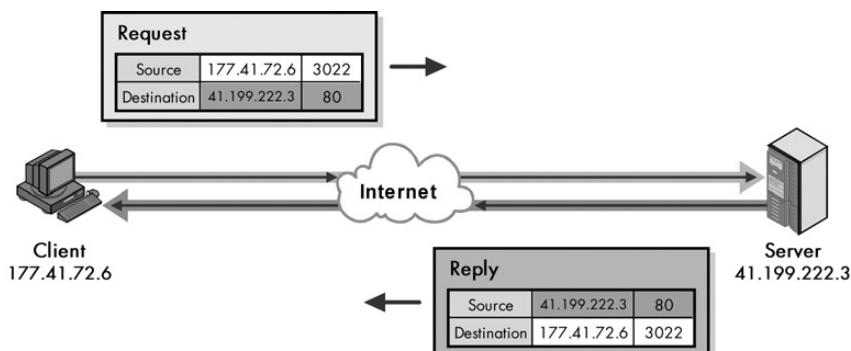
Many applications use more than one process, and it is theoretically possible to run out of ephemeral port numbers on a very busy IP device. For this reason, most of the time, the ephemeral port number range can be changed. The default range may be different for other operating systems.

Just as well-known and registered port numbers are used for server processes, ephemeral port numbers are for client processes only. This means that the use of a range of addresses from 1024 to 4999 does not conflict with the use of that same range for registered port numbers. I discussed this in the previous section, “Ephemeral Port Number Assignment.”

### Port Number Use During a Client/Server Exchange

Now let’s return to the matter of client/server application message exchange. Once a client is assigned an ephemeral port number, that port number is used as the source port in the client’s request TCP/UDP message. The server receives the request and then generates a reply. In forming this response message, the server *swaps* the source and destination port numbers, just as it does the source and destination IP addresses. So the server’s reply is sent from the well-known or registered port number on the server process back to the ephemeral port number on the client machine.

Now back to the web browser example. The web browser, with IP address 177.41.72.6, wants to send an HTTP request to a particular website at IP address 41.199.222.3. The TCP/IP software sends the HTTP request with a *destination port* number of 80 (the one reserved for HTTP servers). The TCP/IP software allocates the *source port* number from a pool of ephemeral ports; let’s say it’s port 3022. When the HTTP request arrives at the web server, it is conveyed to port 80 where the HTTP server receives it. That process generates a reply and sends it back to 177.41.72.6, using *destination port* 3022 and *source port* 80. The two processes can exchange information back and forth each time the TCP/IP software swaps the source port number and destination port number along with the source and destination IP addresses. This example is illustrated in Figure 43-3.



**Figure 43-3: TCP/IP client/server application port mechanics** This highly simplified example shows how clients and servers use port numbers for a request/reply exchange. The client is making an HTTP request and sends it to the server at HTTP’s well-known port number, 80. Its port number for this exchange is the pseudo-randomly selected port 3022. The server sends its reply back to that port number, which it reads from the request.

**KEY CONCEPT** In most TCP/IP client/server communications, the client uses a random ephemeral port number and sends a request to the appropriate reserved port number at the server's IP address. The server sends its reply back to whatever port number it finds in the Source Port field of the request.

## TCP/IP Sockets and Socket Pairs: Process and Connection Identification

In this chapter, I have discussed the key difference between addressing at the level of IP and addressing with regard to application processes. To summarize, at layer 3 an IP address is all that is really important for properly transmitting data between IP devices. In contrast, application protocols must be concerned with the port assigned to each instance of the application so that the protocols can properly use TCP or UDP.

So, the overall identification of an application process actually uses the combination of the IP address of the host it runs on—or the network interface over which it is talking, to be more precise—and the port number that has been assigned to it. This combined address is called a *socket*. Sockets are specified using the notation *<IP Address>:<Port Number>*. For example, if you have a website running on IP address 41.199.222.3, the socket corresponding to the HTTP server for that site would be *41.199.222.3:80*.

**KEY CONCEPT** The overall identifier of a TCP/IP application process on a device is the combination of its IP address and port number, which is called a *socket*.

You will also sometimes see a socket specified using a host name instead of an IP address, like this: *<Host Name>:<Port Number>*. To use this descriptor, the web browser must first resolve the name to an IP address using DNS. For example, you might find a website URL such as <http://www.thisisagreatsite.com:8080>. This tells the web browser to *resolve* the name [www.thisisagreatsite.com](http://www.thisisagreatsite.com) first to an IP address using DNS. Then it tells the browser to send a request to that address using the nonstandard server port 8080, which the browser occasionally uses instead of port 80. (See Chapter 70's discussion of application layer addressing using URLs for more information.)

The *socket* is a fundamental concept to the operation of TCP/IP application software. In fact, it is the basis for an important TCP/IP application program interface (API) with the same name: *sockets*. A version of this API for Windows is called *Windows Sockets* or *Winsock*, which you may have heard of before. These APIs allow application programs to easily use TCP/IP to communicate.

So the exchange of data between a pair of devices consists of a series of messages sent from a socket on one device to a socket on the other. Each device will normally have multiple simultaneous conversations going on. In the case of TCP, a connection is established for each pair of devices for the duration of the communication session. These connections must be managed, and this requires that they be uniquely identified. This is done using the socket identifiers for each of the two devices that are connected.

**KEY CONCEPT** Each device may have multiple TCP connections active at any given time. Each connection is uniquely identified using the combination of the client socket and server socket, which in turn contains four elements: the client IP address and port, and the server IP address and port.

Let's return to the example in Figure 43-3. You are sending an HTTP request from your client at 177.41.72.6 to the website at 41.199.222.3. The server for that website will use well-known port number 80, so its socket is 41.199.222.3:80, as you saw before. You have been ephemeral port number 3022 for the web browser, so the client socket is 177.41.72.6:3022. The overall connection between these devices can be described using this socket pair: (41.199.222.3:80, 177.41.72.6:3022).

For much more on how TCP identifies connections, see the topic on TCP ports and connection identification in Chapter 46.

Unlike TCP, UDP is a connectionless protocol, so it obviously doesn't use connections. The pair of sockets on the sending and receiving devices can still be used to identify the two processes that are exchanging data, but because there are no connections, the socket pair doesn't have the significance that it does in TCP.

## Common TCP/IP Applications and Well-Known and Registered Port Numbers

The great popularity of the TCP/IP protocol suite has led to the development of literally thousands of different applications and protocols. Most of these use the client/server model of operation that I discussed earlier in this chapter. Server processes for a particular application are designed to use a particular reserved port number, and clients use an ephemeral (temporary) port number to initiate a connection to the server. To ensure that everyone agrees on which port numbers each server application should use for each application, port numbers are centrally managed by the IANA.

Originally, IANA kept the list of well-known and registered port numbers in a lengthy text document along with all the many other parameters for which IANA was centrally responsible (such as IP Protocol field numbers, Type and Code field values for ICMP, and so on). These port numbers were published on a periodic basis in Internet (RFC) standards documents titled "Assigned Numbers." This system worked fine in the early days of the Internet, but by the mid-1990s, these values were changing so rapidly that using the RFC process was not feasible. It was too much work to keep publishing them, and the RFC was practically out of date the day after it was published.

The last "Assigned Numbers" standard was RFC 1700, which was published in October 1994. After that time, IANA moved to a set of World Wide Web documents that contained the parameters they manage. This allowed IANA to keep the lists constantly up-to-date, and enabled TCP/IP users to get more current information. RFC 1700 was officially obsoleted in 2002.

You can find complete information on all the parameters that IANA maintains at <http://www.iana.org/numbers.html>. The URL of the file that contains TCP/UDP port assignments is <http://www.iana.org/assignments/port-numbers>.

This document is the definitive list of all well-known and registered TCP and UDP port assignments. Each port number is assigned a short *keyword* with a brief description of the protocol that uses it. There are two problems with this document. First, it is incredibly long; it contains over 10,000 lines of text. Most of the protocols mentioned in those thousands of lines are for obscure applications that you have probably never heard of before (I certainly have never heard of most of them!). This makes it hard to easily see the port assignments for the protocols that are most commonly used.

The other problem with this document is that it shows the same port number as reserved for both TCP and UDP for an application. As I mentioned earlier, TCP and UDP port numbers are actually independent, so, in theory, one port number could assign TCP port 80 to one server application type and UDP port 80 to another. It was believed that this would lead to confusion, so with very few exceptions, the same port number is shown in the list for the same application for both TCP and UDP. Nevertheless, showing this in the list has a drawback: You can't tell which protocol the application actually uses, and which has just been reserved for consistency.

Given all that, I have decided to include a couple of summary tables that show the well-known and registered port numbers for the most common TCP/IP applications. I have indicated whether or not the protocol uses TCP, UDP, or both. Table 43-1 lists the well-known port numbers for the most common TCP/IP application protocols.

**Table 43-1:** Common TCP/IP Well-Known Port Numbers and Applications

<b>Port #</b>	<b>TCP/UDP</b>	<b>Keyword</b>	<b>Protocol Abbreviation</b>	<b>Application or Protocol Name/Comments</b>
7	TCP + UDP	echo	—	Echo Protocol
9	TCP + UDP	discard	—	Discard Protocol
11	TCP + UDP	systat	—	Active Users Protocol
13	TCP + UDP	daytime	—	Daytime Protocol
17	TCP + UDP	qotd	QOTD	Quote of the Day Protocol
19	TCP + UDP	chargen	—	Character Generator Protocol
20	TCP	ftp-data	FTP (data)	File Transfer Protocol (default data port)
21	TCP	ftp	FTP (control)	File Transfer Protocol (control/commands)
23	TCP	telnet	—	Telnet Protocol
25	TCP	smtp	SMTP	Simple Mail Transfer Protocol
37	TCP + UDP	time	—	Time Protocol
43	TCP	nicname	—	Whois Protocol (also called Nicname)
53	TCP + UDP	domain	DNS	Domain Name Server (Domain Name System)
67	UDP	bootps	BOOTP/DHCP	Bootstrap Protocol/Dynamic Host Configuration Protocol (server)
68	UDP	bootpc	BOOTP/DHCP	Bootstrap Protocol/Dynamic Host Configuration Protocol (client)

*(continued)*

**Table 43-1:** Common TCP/IP Well-Known Port Numbers and Applications (continued)

<b>Port #</b>	<b>TCP/UDP</b>	<b>Keyword</b>	<b>Protocol Abbreviation</b>	<b>Application or Protocol Name/Comments</b>
69	UDP	tftp	TFTP	Trivial File Transfer Protocol
70	TCP	gopher	—	Gopher Protocol
79	TCP	finger	—	Finger User Information Protocol
80	TCP	http	HTTP	Hypertext Transfer Protocol (World Wide Web)
110	TCP	pop3	POP	Post Office Protocol (version 3)
119	TCP	nntp	NNTP	Network News Transfer Protocol
123	UDP	ntp	NTP	Network Time Protocol
137	TCP + UDP	netbios-ns	—	NetBIOS (Name Service)
138	UDP	netbios-dgm	—	NetBIOS (Datagram Service)
139	TCP	netbios-ssn	—	NetBIOS (Session Service)
143	TCP	imap	IMAP	Internet Message Access Protocol
161	UDP	snmp	SNMP	Simple Network Management Protocol
162	UDP	snmptrap	SNMP	Simple Network Management Protocol (Trap)
179	TCP	bgp	BGP	Border Gateway Protocol
194	TCP	irc	IRC	Internet Relay Chat
443	TCP	https	HTTP over SSL	Hypertext Transfer Protocol over Secure Sockets Layer
500	UDP	isakmp	IKE	IPsec Internet Key Exchange
520	UDP	router	RIP	Routing Information Protocol (RIP-1 and RIP-2)
521	UDP	ripng	RIPng	Routing Information Protocol - Next Generation

The registered port numbers are by definition for protocols that are not standardized using the RFC process, so they are mostly esoteric applications, and I don't think it's necessary to list all of them. Table 43-2 shows a few that I feel are of particular interest.

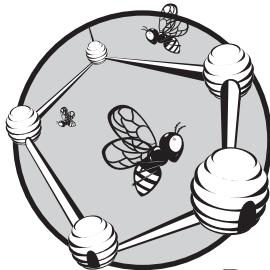
**Table 43-2:** Common TCP/IP Registered Port Numbers and Applications

<b>Port #</b>	<b>TCP/UDP</b>	<b>Keyword</b>	<b>Protocol Abbreviation</b>	<b>Application or Protocol Name/Comments</b>
1512	TCP + UDP	wins	WINS	Microsoft Windows Internet Naming Service
1701	UDP	l2tp	L2TP	Layer 2 Tunneling Protocol
1723	TCP	pptp	PPTP	Point-to-Point Tunneling Protocol
2049	TCP + UDP	nfs	NFS	Network File System
6000–6063	TCP	x11	X11	X Window System



# 44

## TCP/IP USER DATAGRAM PROTOCOL (UDP)



The very fact that the TCP/IP protocol suite bears the name of the Internet Protocol (IP) and the Transmission Control Protocol (TCP) suggests that these are the two key protocols in the suite. IP resides at the network layer, and TCP is at the transport layer. It's no wonder that many people don't even realize that there is a second transport layer protocol in TCP/IP.

Like a shy younger brother, the *User Datagram Protocol (UDP)* sits in the shadows while TCP gets the glory. The fancier sibling deserves much of this limelight, since TCP is arguably the more important of the two. However, UDP fills a critical niche in the TCP/IP protocol suite, because it allows many applications to work at their best when using TCP would be less than ideal.

In this chapter, I discuss UDP, the simpler and less-known TCP/IP transport protocol. I begin with an overview of the protocol and describe its history and standards. I outline how UDP operates, and explain the format used for UDP messages. I conclude with a discussion of what kinds of applications use UDP and the well-known or registered ports that are assigned to them.

## UDP Overview, History, and Standards

I suppose the sibling rivalry analogy I mentioned in the introduction to this section may be a bit silly. I highly doubt that protocols lie awake at night worrying about how much we use them. However, it's interesting to discover just how important UDP really is, given how little attention it gets compared to TCP. In fact, in true older-sibling, spotlight-stealing fashion, you can't even really understand the history of UDP without first discussing TCP.

In Chapter 8, where I described the history of TCP/IP, I explained that very early on in the development of the protocol suite, there was only one protocol that handled the functions IP and TCP perform. This protocol, called TCP, provided network layer connectivity like IP, and also established connections, offered reliability, and took care of the typical transport layer quality requirements that you associate with modern TCP, such as flow control and retransmission handling.

It didn't take long before the developers of the fledgling combined TCP protocol quickly realized that mixing these functions together was a mistake. While most conventional applications needed the classic transport layer reliability functions, some did not. These features introduced overhead, which was added whether or not applications actually needed the reliability features. Worse, there were some applications for which the features not only were of no value, but also were a detriment, since even a small amount of lost performance due to the overhead would be a problem.

The solution was to separate the original protocol into IP and TCP. IP would do basic internetworking, and TCP would do the reliability features. This paved the way for the creation of an alternative transport layer protocol—UDP—for applications that didn't want or need the features that TCP provided.

There are two main attributes that are always associated with UDP: simple and fast. UDP is a simple protocol that uses a very straightforward messaging structure that is similar to the message format that many other TCP/IP protocols use (in contrast to the more complex data structures—streams and segments—that TCP uses). In fact, when you boil it down, UDP's only real goal is to serve as an interface between networking application processes that are running at the higher layers, and the internetworking capabilities of IP.

Like TCP, UDP layers a method of transport layer addressing (and hence, process identification) on top of IP through the use of UDP port numbers. UDP includes an optional checksum capability for error detection, but adds virtually no other functionality.

The best way to see the simplicity of UDP is to look at the standards that define it. Or rather, I should say *standard* in the singular, because there is only one. UDP was defined in RFC 768, “User Datagram Protocol,” in 1980. This document is three pages in length, and no one has ever needed to revise it.

UDP is a fast protocol specifically because it doesn't have all the bells and whistles of TCP. This makes it unsuitable for use by many, if not most, typical networking applications. But for some applications, this speed is exactly what the applications want from a transport layer protocol, namely something that takes the applications' data and quickly shuffles it down to the IP layer with minimal fuss. In choosing to use UDP, the application writer takes it upon himself to take care of issues such as reliability and retransmissions, if necessary. This can be a recipe for success or failure, depending on the application and how carefully the writer uses UDP.

**KEY CONCEPT** *The User Datagram Protocol (UDP) was developed for use by application protocols that do not require reliability, acknowledgment, or flow control features at the transport layer. It is designed to be simple and fast. It provides only transport layer addressing (in the form of UDP ports), an optional checksum capability, and little else.*

## UDP Operation

UDP is so simple that I can't say a great deal about how it works. It is designed to do as little as possible.

### What UDP Does

UDP's only real task is to take data from higher-layer protocols and place it in UDP messages, which are then passed down to IP for transmission. The basic steps for transmission using UDP are as follows:

1. **Higher-Layer Data Transfer** An application sends a message to the UDP software.
2. **UDP Message Encapsulation** The higher-layer message is encapsulated into the Data field of a UDP message. The headers of the UDP message are filled in, including the Source Port field of the application that sent the data to UDP and the Destination Port field of the intended recipient. The checksum value may also be calculated.
3. **Transfer Message to IP** The UDP message is passed to IP for transmission.

And that's about it. Of course, when the destination device receives the message, this short procedure is reversed.

### What UDP Does Not Do

UDP is so simple that its operation is often described in terms of what it does not do, instead of what it does. As a transport protocol, UDP does not do the following:

- Establish connections before sending data. It just packages the data and sends it off.
- Provide acknowledgments to show that data was received.
- Provide any guarantees that its messages will arrive.
- Detect lost messages and retransmit them.
- Ensure that data is received in the same order that it was sent.
- Provide any mechanism to handle congestion or manage the flow of data between devices.

**KEY CONCEPT** UDP is probably the simplest protocol in all of TCP/IP. It takes application layer data that has been passed to it, packages it in a simplified message format, and sends it to IP for transmission.

If these limitations sound similar the ones for IP, then you’re paying attention. UDP is basically IP with transport layer port addressing. (For this reason, UDP is sometimes called a *wrapper protocol*, since all it does is wrap application data in its simple message format and send it to IP.)

However, despite the previous list, there are a couple of limited feedback and error-checking mechanisms that do exist within UDP. One is the optional checksum capability, which can allow for the detection of an error in transmission or the situation in which a UDP message is delivered to the wrong place (see the next section, “UDP Message Format” for details). The other is Internet Control Message Protocol (ICMP) error reporting (see Chapter 31). For example, if a UDP message is sent that contains a destination port number that the destination device does not recognize, the destination host will send an ICMP Destination Unreachable message back to the original source. Of course, ICMP exists for all IP errors of this sort, so I’m stretching a bit here.

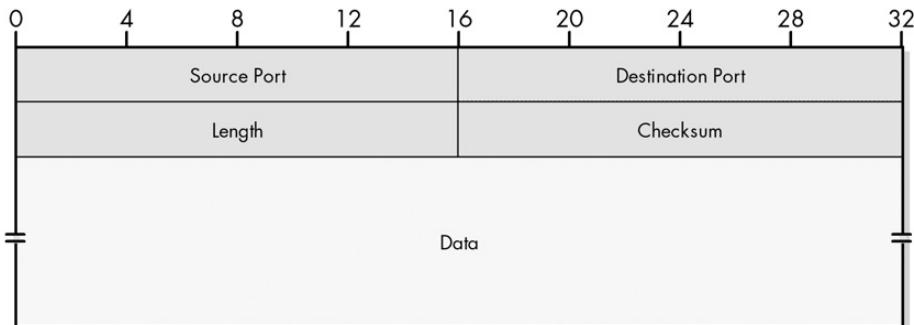
## UDP Message Format

What’s the magic word when it comes to UDP? It’s *simple*. This is true of the operation of the protocol, and it is also true of the format used for UDP messages. Interestingly, however, there is one aspect of UDP that is not simple.

In keeping with the goal of efficiency, the UDP header is only 8 bytes in length. You can contrast this with the TCP header size of 20 bytes or more. Table 44-1 and Figure 44-1 show the format of UDP messages.

The UDP Checksum field is the one area where the protocol is a bit confusing. The concept of a checksum itself is nothing new; checksums are used widely in networking protocols to provide protection against errors. What’s a bit odd is this notion of computing the checksum over the regular datagram as well as a pseudo header. So instead of calculating the checksum over only the fields in the UDP datagram, the UDP software first constructs a fake additional header that contains the following fields:

- IP Source Address field
- IP Destination Address field
- IP Protocol field
- UDP Length field

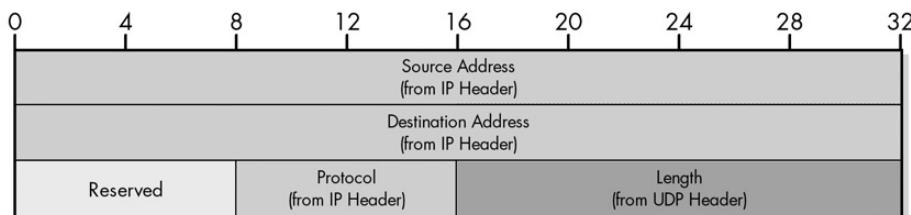


**Figure 44-1:** UDP message format

**Table 44-1:** UDP Message Format

Field Name	Size (Bytes)	Description
Source Port	2	The 16-bit port number of the process that originated the UDP message on the source device. This will normally be an ephemeral (client) port number for a request that a client sends to a server or a well-known/registered (server) port number for a reply that a server sends to a client. (See Chapter 43 for details.)
Destination Port	2	The 16-bit port number of the process that is the ultimate intended recipient of the message on the destination device. This will usually be a well-known/registered (server) port number for a client request or an ephemeral (client) port number for a server reply. (See Chapter 43 for details.)
Length	2	The length of the entire UDP datagram, including both header and Data fields.
Checksum	2	An optional 16-bit checksum computed over the entire UDP datagram plus a special pseudo header of fields. See below for more information.
Data	Variable	The encapsulated higher-layer message that will be sent.

The UDP pseudo header format is illustrated in Figure 44-2.



**Figure 44-2:** UDP pseudo header format

The total length of this pseudo header is 11 bytes. It is padded to 12 bytes with a byte of zeros and then prepended to the real UDP message. The checksum is then computed over the combination of the pseudo header and the real UDP message, and the value is placed into the Checksum field. The pseudo header is used only for this calculation and is then discarded; it is not actually transmitted. The UDP software in the destination device creates the same pseudo header when calculating its checksum in order to compare it to the one transmitted in the UDP header.

Computing the checksum over the regular UDP fields protects the UDP message against bit errors. Adding the pseudo header allows the checksum to also protect the UDP message against other types of problems as well, most notably the accidental delivery of a message to the wrong destination. The checksum calculation in UDP, including the use of the pseudo header, is exactly the same as the method used in TCP (except that the Length field is different in TCP). See Chapter 48 for a full description of why the pseudo header is important, as well as some of the interesting implications of using IP fields in transport layer datagram calculations.

**KEY CONCEPT** UDP packages application layer data into a very simple message format that includes only four header fields. One of these is an optional Checksum field. When the Checksum field is used, the checksum is computed over both the real header and a pseudo header of fields from the UDP and IP headers, in a manner that's very similar to the way the TCP checksum is calculated.

Note that the use of the Checksum field is optional in UDP. If UDP doesn't use the Checksum field, UDP sets it to a value of all zeros. This could potentially create confusion, however, since when UDP uses the checksum, the calculation can sometimes result in a value of zero. To avoid having the destination think that UDP didn't use the checksum in this case, UDP instead represents this zero value as a value of all ones (65,535 decimal).

## UDP Common Applications and Server Port Assignments

As you have seen, UDP contains very little functionality. With the exception of the important addressing capability that UDP ports represent, using UDP is very much like using IP. This means that UDP has most of the same disadvantages that IP has. It doesn't establish a lasting connection between devices; it doesn't acknowledge received data or retransmit lost messages; and it certainly isn't concerned with obscurities such as flow control and congestion management.

The absence of those features makes UDP simply unsuitable for the majority of classic networking applications. These applications usually need to establish a connection so that the two devices can exchange data. Many applications also must have the ability to occasionally, or even regularly, send very large amounts of data that must be received intact for it to be of value. For example, consider a message transfer protocol like the Hypertext Transfer Protocol (HTTP). If only part of a web page gets from a server back to a web browser, it's useless. HTTP and other file and message transfer protocols like it need the capabilities of TCP.

### Why Some TCP/IP Applications Use UDP

So what applications use UDP then? UDP's classic limitation is that because it doesn't provide reliability features, an application that uses UDP is responsible for those functions. In reality, if an application needs the features that TCP provides but not the ones that UDP provides, it's inefficient to allow the application to implement those features, except in special cases. If the application needs what TCP provides, it should just use TCP! However, applications that only need some of what TCP implements are sometimes better off using UDP and implementing that limited set of functionality at the application level.

So, the applications that run over UDP are normally the ones that do not require all or even most of the features that TCP has. These applications can benefit from the increased efficiency that comes about from avoiding the setup and overhead associated with TCP. Applications usually (but not always) meet this description because the data they send falls into one of two categories:

**Data Where Performance Is More Important Than Completeness** The classic example of this category is a multimedia application. For streaming a video clip over the Internet, the most important feature is that the stream starts flowing quickly and keeps flowing. Human beings notice only significant disruptions in the flow of this type of information, so a few bytes of data missing due to a lost

datagram is not a big problem. Furthermore, even if someone used TCP for something like this and noticed and retransmitted a lost datagram, it would be useless, because the lost datagram would belong to a part of the clip that is long past—and the time spent in that retransmission might make the current part of the clip arrive late. Clearly, UDP is best for this situation.

**Data Exchanges That Are “Short and Sweet”** There are many TCP/IP applications in which the underlying protocol consists of only a very simple request/reply exchange. A client sends a short request message to a server, and a short reply message goes back from the server to the client. In this situation, there is no real need to set up a connection the way that TCP does. Also, if a client sends only one short message, a single IP datagram can carry the message. This means that there is no need to worry about data arriving out of order, flow control between the devices, and so forth. How about the loss of the request or the reply? These can be handled simply at the application level using timers. If a client sends a request and the server doesn’t get it, the server won’t reply, and the client will eventually send a replacement request. The same logic applies if the server sends a response that never arrives.

These are the most common cases where UDP is used, but there are other reasons. For example, if an application needs to multicast or broadcast data, it must use UDP, because TCP is supported only for unicast communication between two devices.

**KEY CONCEPT** A protocol uses UDP instead of TCP in two situations. The first is when an application values timely delivery over reliable delivery, and when TCP’s retransmission of lost data would be of limited or even no value. The second is when a simple protocol can handle the potential loss of an IP datagram itself at the application layer using a timer/retransmit strategy, and when the other features of TCP are not required. Applications that require multicast or broadcast transmissions also use UDP, because TCP does not support those transmissions.

Incidentally, I have read about problems that have occurred in the past in applications using UDP. Sometimes, programmers don’t realize how little UDP does, how it leaves the application responsible for handling all the potential vagaries of an internetworking environment. Someone writing a UDP-based application must always keep in mind that no one can make assumptions about how or even whether a destination will receive any message. Insufficient testing can lead to disaster in worst-case scenarios on a larger internetwork, especially the Internet.

### **Common UDP Applications and Server Port Use**

Table 44-2 shows some of the more interesting protocols that use UDP and the well-known and registered port numbers used for each one’s server processes. It also provides a very brief description of why these protocols use UDP instead of TCP.

## ***Applications That Use Both UDP and TCP***

There are some protocols that use both of the TCP/IP transport layer protocols. This is often the case either for utility protocols that are designed to accept connections using both transport layer protocols, or for applications that need the benefits of TCP in some cases but not others.

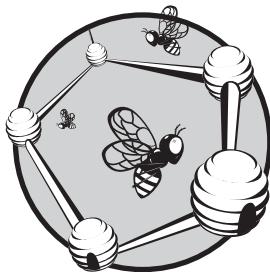
The classic example of the latter is the TCP/IP Domain Name System (DNS), which normally uses UDP port 53 for simple requests and replies, which are usually short. Larger messages requiring reliable delivery, such as zone transfers, use TCP port 53 instead. Note that in Table 44-2, I have omitted some of the less-significant protocols such as the ones used for diagnostic purposes (Echo, Discard, CharGen, and so on). For a full list of all common applications, see Chapter 43.

**Table 44-2:** Common UDP Applications and Server Port Assignments

<b>Port #</b>	<b>Keyword</b>	<b>Protocol</b>	<b>Comments</b>
53	domain	Domain Name Server (DNS)	Uses a simple request/reply messaging system for most exchanges (but also uses TCP for longer ones).
67 and 68	bootps/ bootpc	Bootstrap Protocol (BOOTP) and Dynamic Host Configuration Protocol (DHCP)	Host configuration protocols that consist of short request and reply exchanges.
69	tftp	Trivial File Transfer Protocol (TFTP)	TFTP is a great example of a protocol that was specifically designed for UDP, especially when you compare it to regular FTP. The latter protocol uses TCP to establish a session between two devices and then makes use of its own large command set and TCP's features in order to ensure the reliable transfer of possibly very large files. In contrast, TFTP is designed for the quick and easy transfer of small files. To avoid file corruption, TFTP includes simple versions of some of TCP's features, such as acknowledgments.
161 and 162	snmp	Simple Network Management Protocol	An administrative protocol that uses relatively short messages.
520 and 521	router/ ripng	Routing Information Protocol (RIP-1, RIP-2, RIPng)	Unlike more complex routing protocols like BGP, RIP uses a simple request/reply messaging system, doesn't require connections, and does require multicasts/broadcasts. This makes it a natural choice for UDP. If a routing update is sent due to a request and is lost, it can be replaced by sending a new request. Routine (unsolicited) updates that are lost are replaced in the next cycle.
2049	nfs	Network File System	NFS is an interesting case. Since it is a file-sharing protocol, you would think that it would use TCP instead of UDP, but it was originally designed to use UDP for performance reasons. There were many people who felt that this was not the best design decision, and later versions moved to the use of TCP. The latest version of NFS uses only TCP.

# 45

## TCP OVERVIEW, FUNCTIONS, AND CHARACTERISTICS



As I mentioned in Chapter 42, the Transmission Control Protocol (TCP) is a critically important part of the TCP/IP protocol suite. It's also a fairly complicated protocol, with a lot of important concepts and mechanisms that you need to understand. The old joke says the “best way to eat an elephant is one bite at a time.” Similarly here, you can best comprehend the operation of this complicated protocol by going slowly, starting with a high-level look at it, where it came from, and what it does.

In this chapter, I begin by introducing you to TCP. I first provide an overview and history of TCP and then describe the standards that define it. Then I illustrate what TCP actually does by listing its functions and explaining how TCP works by describing its most important characteristics. This will give you a feel for what TCP is all about, and it will set the stage for the more complex technical discussions in subsequent chapters.

## TCP Overview, History, and Standards

Between them, layers 3 and 4 of the OSI Reference Model represent the interface between networking software (the applications that need to move data across networks) and networking hardware (the devices that carry the data over networks). Any protocol suite must have a protocol or set of protocols that handles these layer 3 and layer 4 functions.

The TCP/IP protocol suite is named for the two main protocols that provide these capabilities. Both TCP and the Internet Protocol (IP) allow software to run on an internetwork. IP deals with internetwork datagram delivery and routing, while TCP handles connections and provides reliability. What's interesting, however, is that in the early days of the protocol suite, there was, in fact, no TCP/IP at all.

### TCP History

Due to its prominent role in the history of networking, TCP is impossible to describe without going back to the early days of the protocol suite. In the early 1970s, what we know today as the global Internet was a small research internetwork called the *ARPAnet*, an acronym that came from the United States Defense Advanced Research Projects Agency (DARPA or ARPA). This network used a technology called the *Network Control Protocol (NCP)*, which allowed hosts to connect to each other. NCP did approximately the same job that TCP and IP do together today.

Due to limitations in NCP, development began on a new protocol that would be better suited to a growing internetwork. This new protocol, first formalized in RFC 675, was called the Internet *Transmission Control Program (TCP)*. Like its predecessor NCP, TCP was responsible for basically everything that was needed to allow applications to run on an internetwork. Thus, TCP was at first both TCP and IP.

As I explain in detail in the description of the history of TCP/IP as a whole in Chapter 8, several years were spent adjusting and revising TCP, with version 2 of the protocol documented in 1977. While the functionality of TCP was steadily improved, there was a problem with the basic concept behind the protocol. Having TCP handle datagram transmissions, routing (layer 3 functions), and connections, reliability, and data-flow management (layer 4 functions) meant that TCP violated key concepts of protocol layering and modularity. TCP forced all applications to use the layer 4 functions in order to use the layer 3 functions. This made TCP inflexible and poorly suited to the needs of applications that required only the lower-level functions and not the higher-level ones.

As a result, the decision was made to split TCP into two: the layer 4 functions were retained, with TCP renamed the *Transmission Control Protocol* (as opposed to Transmission Control Program). The layer 3 functions became the Internet Protocol. This split was finalized in version 4 of TCP, and so the first IP was given “version 4” as well, for consistency. RFC 793, “Transmission Control Protocol,” published in September 1981, defined TCP version 4, and it is still the current version of the standard.

Even though it is more than 20 years old and is the first version most people have ever used, version 4 was the result of several years' work and many earlier TCP versions tested on the early Internet. It is therefore a very mature protocol for its

age. A precocious protocol, you might say. (To be fair, other standards have described many additional features and modifications to TCP, rather than upgrading the main document.)

## **Overview of TCP Operation**

TCP is a full-featured transport layer protocol that provides all the functions that a typical application needs for the reliable transportation of data across an arbitrary internetwork. It provides transport layer addressing for application processes in the form of TCP ports and allows machines to use these ports in order to establishing connections between them. Once the devices have connected to each other, they can pass data bidirectionally between them. Applications can send data to TCP as a simple stream of bytes, and TCP takes care of packaging and sending the data as segments that TCP packages into IP datagrams. The receiving device's TCP implementation reverses the process, passing up to the application the stream of data that the device originally sent.

TCP includes an extensive set of mechanisms. These mechanisms ensure that data gets from source to destination reliably, consistently, and in a timely fashion. The key to its operation in this regard is the *sliding window acknowledgment system*, which allows each device to keep track of the bytes of data it has sent and to confirm the receipt of data received from the other device in the connection. Unacknowledged data is eventually retransmitted automatically, and the parameters of the system can be adjusted to the needs of the devices and the connection. This same system also provides buffering and flow control capabilities between devices. These capabilities handle uneven data delivery rates and other problems.

**KEY CONCEPT** The primary transport layer protocol in the TCP/IP protocol suite is the *Transmission Control Protocol (TCP)*. TCP is a connection-oriented, acknowledged, reliable, full-featured protocol designed to provide applications with a reliable way to send data using the unreliable Internet Protocol (IP). It allows applications to send bytes of data as a stream of bytes and automatically packages them into appropriately sized segments for transmission. It uses a special *sliding window acknowledgment system* to ensure that its recipient receives all data, handles necessary retransmissions, and provides flow control so that each device in a connection can manage the rate at which other devices send data to it.

Because of TCP's many capabilities, it's likely that the protocol will satisfy just about any application that requires reliable, connection-oriented data delivery. A primary goal of TCP, reliable data delivery means that higher-layer applications don't need to provide TCP's common functions. Because the majority of conventional message-passing applications employ it, the TCP/IP transport protocol is the most widely used transport protocol.

## **TCP Standards**

RFC 793 is the defining standard for TCP, but it doesn't include all the details about how modern TCP operates. Several other standards include additional information about how the protocol works and describe enhancements to the basic

TCP mechanisms that were developed over the years. Some of these are fairly esoteric, but they are useful for gaining a more complete understanding of TCP. I have listed some of them in Table 45-1.

**Table 45-1:** Supplementary TCP Standards

RFC #	Name	Description
813	Window and Acknowledgment Strategy in TCP	Discusses the TCP sliding window acknowledgment system, describes certain problems that can occur with it, and offers methods to correct them.
879	The TCP Maximum Segment Size and Related Topics	Discusses the important maximum segment size (MSS) parameter that controls the size of TCP messages, and then relates this parameter to IP datagram size.
896	Congestion Control in IP/TCP Internetworks	Talks about congestion problems and how you can use TCP to handle them. Note the interesting inversion of the normal protocol suite name: IP/TCP.
1122	Requirements for Internet Hosts — Communication Layers	Describes important details of how TCP should be implemented on hosts.
1146	TCP Alternate Checksum Options	Specifies a mechanism for having TCP devices use an alternative method of checksum generation.
1323	TCP Extensions for High Performance	Defines extensions to TCP for high-speed links and new TCP options.
2018	TCP Selective Acknowledgment Options	An enhancement to basic TCP functionality that allows TCP devices to selectively specify specific segments for retransmission.
2581	TCP Congestion Control	Describes four algorithms used for congestion control in TCP networks: slow start, congestion avoidance, fast retransmit, and fast recovery.
2988	Computing TCP's Retransmission Timer	Discusses issues related to setting the TCP retransmission timer, which controls how long a device waits for acknowledgment of sent data before retransmitting it.

There are hundreds of higher-layer application protocols that use TCP and whose defining standards therefore make at least glancing reference to it.

TCP is designed to use IP, since they were developed together and as you have seen, were even once part of the same specification. They were later split up in order to respect the principles of architectural layering. For this reason, TCP tries to make as few assumptions as possible regarding the underlying protocol over which it runs. It is not as strictly tied to the use of IP as you might imagine, and you can even adapt it for use over other network layer protocols. For our purposes, however, this should be considered mainly an interesting aside.

## TCP Functions

You have now seen where TCP comes from and the standards that describe it. As I said in the introduction to this chapter, TCP is a complicated protocol, so it will take some time to explain how it works. Here, I'll describe what TCP does and what it doesn't do.

## **Functions That TCP Performs**

Despite the TCP’s complexity, I can simplify its basic operation by describing its primary functions. The following are what I believe to be the six main tasks that TCP performs:

**Addressing/Multiplexing** Many different applications use TCP for a transport protocol. Therefore, like its simpler sibling, the User Datagram Protocol (UDP), multiplexing the data that TCP receives from these different processes so that the data can be sent out using the underlying network layer protocol is an important job for TCP. At the same time, these higher-layer application processes are identified using TCP ports. Chapter 43 contains a great deal of detail about how this addressing works.

**Establishing, Managing, and Terminating Connections** TCP provides a set of procedures that devices can follow in order to negotiate and establish a TCP connection over which data can travel. Once a connection is opened, TCP includes logic for managing the connection and handling problems that may result with the connection. When a device is finished with a TCP connection, a special process is followed to terminate it.

**Handling and Packaging Data** TCP defines a mechanism by which applications are able to send data to TCP from higher layers. This data is then packaged into messages that will be sent to the destination TCP software. The destination software unpackages the data and gives it to the application on the destination machine.

**Transferring Data** Conceptually, the TCP implementation on a transmitting device is responsible for the transfer of packaged data to the TCP process on the other device. Following the principle of layering, this transfer is done by having the TCP software on the sending machine pass the data packets to the underlying network layer protocol, which again normally means IP.

**Providing Reliability and Transmission Quality Services** TCP includes a set of services and features that allows an application to consider the protocol a reliable means of sending of data. This means that normally a TCP application doesn’t need to worry about data being sent and never showing up or arriving in the wrong order. It also means that other common problems that might arise if IP were used directly are avoided.

**Providing Flow Control and Congestion Avoidance Features** TCP allows the flow of data between two devices to be controlled and managed. It also includes features that deal with congestion that devices may experience during communication between each other.

## **Functions That TCP Doesn’t Perform**

Clearly, TCP is responsible for a fairly significant number of key functions. The items listed in the preceding section may not seem that impressive, but this is just a high-level look at the protocol. When you look at these functions in detail, you will see that each one actually involves a rather significant amount of work for TCP to do.

Conversely, sometimes TCP is described as doing everything an application needs in order to use an internetwork. However, the protocol doesn't do everything. It has limitations and certain areas that its designers specifically did not address. The following are some of the notable functions that TCP does not perform:

**Specifying Application Use** TCP defines the transport protocol. It does not specifically describe how applications should use TCP. That is up to the application protocol.

**Providing Security** TCP does not provide any mechanism for ensuring the authenticity or privacy of data that it transmits. If authenticity and privacy are important to applications, they must accomplish them using some other means, such as IPsec, for example.

**Maintaining Message Boundaries** TCP sends data as a continuous stream rather than discrete messages. It is up to the application to specify where one message ends and the next begins.

**Guaranteeing Communication** Wait a minute; isn't TCP supposed to guarantee that data will get to its destination? Well, yes and no. TCP will detect unacknowledged transmissions and resend them if needed. However, if some sort of problem prevents reliable communication, all TCP can do is keep trying. It can't make any guarantees, because there are too many things out of its control. Similarly, it can attempt to manage the flow of data, but it cannot resolve every problem.

This last point might seem a bit pedantic, but it is important to keep in mind, especially since many people tend to think of TCP as bulletproof. The overall success of communication depends entirely on the underlying internetwork and the networks that constitute it. A chain is as strong as its weakest link, and if there is a problem at the lower layers, nothing TCP can do will guarantee successful data transfer.

**KEY CONCEPT** TCP provides reliable communication only by detecting failed transmissions and resending them. It cannot guarantee any particular transmission, because it relies on IP, which is unreliable. All it can do is keep trying if an initial delivery attempt fails.

## TCP Characteristics

In many ways, it is more interesting to look at how TCP does its job than the functions of the job. By examining the most important attributes of TCP and its operation, you can get a better handle on the way TCP works. You can also see the many ways that it contrasts with its simpler transport layer sibling, UDP.

TCP has the following characteristics, which allow it to perform its functions:

**Connection-Oriented** TCP requires that devices first establish a connection with each other before they send data. The connection creates the equivalent of a circuit between the units; it is analogous to a telephone call. A process of negotiation occurs, and that process establishes the connection, thereby ensuring that both devices agree on how they will exchange data.

**Bidirectional** Once a connection is established, TCP devices send data bidirectionally. Both devices on the connection can send and receive, regardless of which one initiated the connection.

**Multiply Connected and Endpoint Identified** The pair of sockets used by the two devices in the connection identifies the endpoints of the TCP connection. This identification method allows each device to have multiple connections opened, either to the same IP device or different IP devices, and to handle each connection independently without conflicts.

**Reliable** Communication using TCP is said to be reliable because TCP keeps track of data that has been sent and received to ensure that all the data gets to its destination. As you saw in the previous section earlier, TCP can't really guarantee that data will always be received. However, it can guarantee that all data sent will be checked for reception, checked for data integrity, and then retransmitted when needed.

**Acknowledged** A key to providing reliability is that TCP acknowledges all transmissions at the TCP layer. Furthermore, TCP cannot guarantee that the remote application will receive all such transmissions. The recipient must tell the sender, "Yes, I got that" for each piece of data transferred. This is in stark contrast to typical messaging protocols in which the sender never knows what happened to its transmission. As you will see, this acknowledgment is fundamental to the operation of TCP as a whole.

**Stream-Oriented** Most lower-layer protocols are designed so that higher-layer protocols must send them data in blocks in order to use them. IP is the best example of this; you send it a message to be formatted and IP puts that message into a datagram. UDP works the same way. In contrast, TCP allows applications to send it a continuous stream of data for transmission. Applications don't need to worry about dividing this stream into chunks for transmission; TCP does it.

**Unstructured Data** An important consequence of TCP's stream orientation is that there are no natural divisions between data elements in the application's data stream. When multiple messages are sent over TCP, applications must provide a way of differentiating one message (data element, record, and so on) from the next.

**Managed Data Flow** TCP does more than just package data and send it as fast as possible. A TCP connection is managed to ensure that data flows evenly and smoothly and that connection includes the ability to deal with problems that arise along the way.

You'll notice that I have not listed "slow" as one of TCP's characteristics. It is true that applications use UDP for performance reasons when they don't want to deal with the overhead that TCP incorporates for connections and reliability. That, however, should not lead you to conclude that TCP is glacially slow. It is in fact quite efficient—were it not, it's unlikely that it would have ever achieved such widespread use.

**KEY CONCEPT** To summarize TCP's key characteristics, we can say that it is connection-oriented, bidirectional, multiply connected, reliable, acknowledged, stream-oriented, and flow-managed.

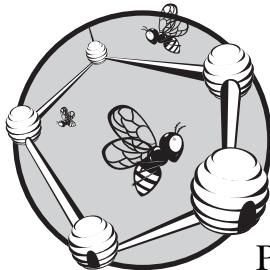
## The Robustness Principle

The TCP standard says that TCP follows the *robustness principle*, which is described in this way: “Be conservative in what you do; be liberal in what you accept from others.” This rule means that every TCP implementation tries to avoid doing anything that would cause a problem for another device’s TCP layer. At the same time, every TCP implementation is also trying to anticipate problems that another TCP may cause and attempting to deal with those problems gracefully.

This principle represents a “belt and suspenders” approach that helps provide extra protection against unusual conditions in TCP operation. In fact, this general principle is applied to many other protocols in the TCP/IP protocol suite, which is part of the reason why it has proven to be so capable over the years. The principle allows TCP and other protocols to deal with unanticipated problems that might show up in the difficult environment of a large internetwork such as the Internet.

# 46

## **TRANSMISSION CONTROL PROTOCOL (TCP) FUNDAMENTALS AND GENERAL OPERATION**



Many people have a difficult time understanding how the Transmission Control Protocol (TCP) works. After spending dozens of hours writing almost 100 pages on the protocol, I am quite sympathetic! I think a main reason for the difficulty is that many descriptions of the protocol leap too quickly from a brief introduction straight into the mind-boggling details of TCP's operation. The problem is that TCP works in a very particular way. Its operation is built around a few very important fundamentals that you absolutely must understand before the details of TCP operation will make much sense.

In this chapter, I describe some of the key operating fundamentals of TCP. I begin with a discussion of how TCP handles data and introduce the concepts of streams, segments, and sequences. I then describe the very important TCP sliding window system, which is used for acknowledgment, reliability, and data flow control. I discuss how TCP uses ports and how it identifies connections. I also describe the most important applications that use TCP and what ports they use for server applications.

## TCP Data Handling and Processing

One of the givens in the operation of most of the protocols you'll find at upper layers in the OSI Reference Model is that the protocols are oriented around the use of messages. These messages are analogous to a written letter in an envelope that contains a specific piece of information. They are passed from higher layers down to lower ones, where they are encapsulated in the lower layers' headers (like putting them in another envelope), and then passed down further until they are actually sent out at the physical layer.

You can see a good example of this by looking at the User Datagram Protocol (UDP), TCP's transport layer peer. To use UDP, an application passes it a distinct block of data that is usually fairly short. The block is packaged into a UDP message, then sent to the Internet Protocol (IP). IP packs the message into an IP datagram and eventually passes it to a layer 2 protocol such as Ethernet. There, IP places it into a frame and sends it to layer 1 for transmission.

### ***Increasing the Flexibility of Application Data Handling: TCP's Stream Orientation***

The use of discrete messaging is pretty simple, and it obviously works well enough since most protocols make use of it. However, it is inherently limiting because it forces applications to create discrete blocks of data in order to communicate. There are many applications that need to send information continuously in a manner that doesn't lend itself well to creating "chunks" of data. Others need to send data in chunks that are so large that applications could never send them as a single message at the lower layers.

To use a protocol like UDP, many applications would be forced to artificially divide their data into messages of a size that has no inherent meaning to them. This would immediately introduce new problems that would require more work for the application. The application would have to keep track of what data is in what message, and replace any data that was lost. It would need to ensure that the messages could be reassembled in the correct order, since IP might deliver them out of order.

Of course, you could program applications to do this, but it would make little sense, because these functions are already ones that TCP is charged with handling. Instead, the TCP designers took the very smart approach of generalizing TCP so that it could accept application data of any size and structure without requiring the data to be in discrete pieces. More specifically, TCP treats data coming from an application as a *stream*—thus, the description of TCP as *stream-oriented*. Each application sends the data it wishes to transmit as a steady stream of octets (bytes). The application doesn't need to carve the data into blocks or worry about how lengthy streams will get across the internetwork. It just "pumps bytes" to TCP.

### ***TCP Data Packaging: Segments***

TCP must take the bytes it gets from an application and send them using a network layer protocol, which is IP in this case. IP is a message-oriented protocol; it is not stream-oriented. Thus, we have simply "passed the buck" to TCP, which must take the stream from the application and divide it into discrete messages for IP. These messages are called *TCP segments*.

**NOTE** Segment is one of the most confusing data structure names in the world of networking. From a dictionary definition standpoint, referring to a piece of a stream as a segment is sensible, but most people working with networks don't think of a message as being a segment. In the industry, the term also refers to a length of cable or a part of a local area network (LAN), among other things, so watch out for that.

IP treats TCP segments like all other discrete messages for transmission. IP places them into IP datagrams and transmits them to the destination device. The recipient unpackages the segments and passes them to TCP, which converts them back to a byte stream in order to send them to the application. This process is illustrated in Figure 46-1.

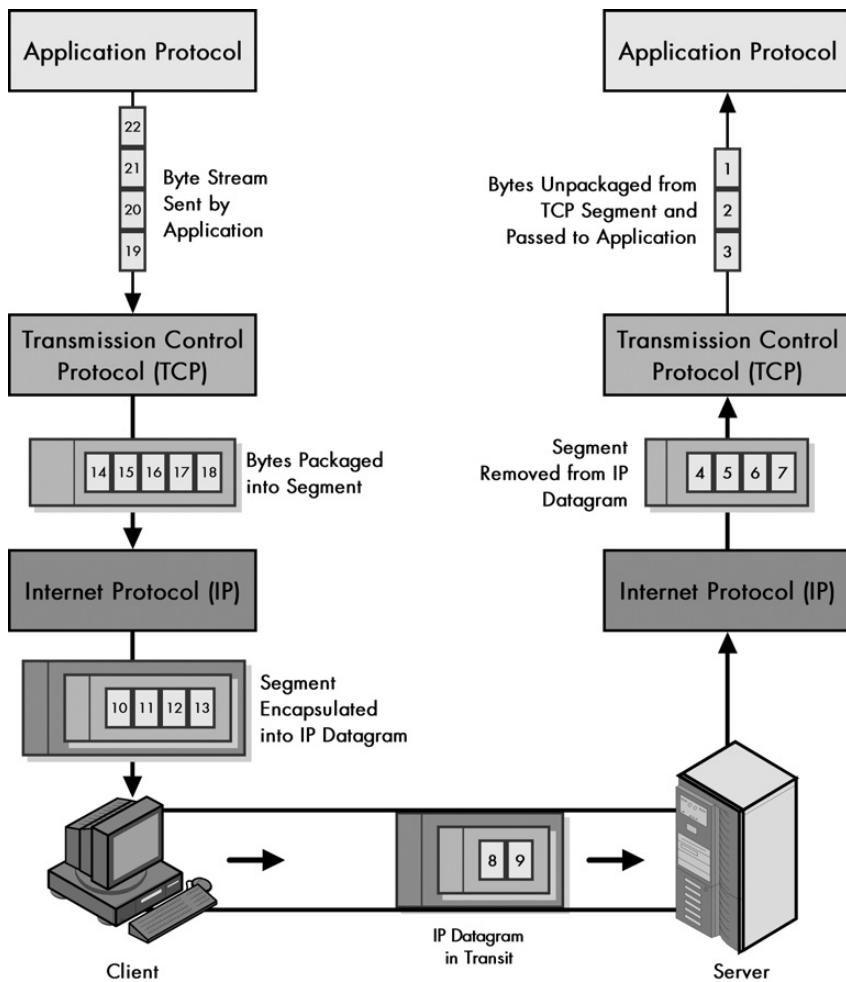
**KEY CONCEPT** TCP is designed to have applications send data to it as a stream of bytes, rather than requiring fixed-size messages to be used. This provides maximum flexibility for a wide variety of uses, because applications don't need to worry about data packaging and can send files or messages of any size. TCP takes care of packaging these bytes into messages called *segments*.

The TCP layer on a device accumulates data that it receives from the application process stream. On regular intervals, the TCP layer forms segments that it will transmit using IP. Two primary factors control the size of the segment. First, there is an overall limit to the size of a segment, chosen to prevent unnecessary fragmentation at the IP layer. A parameter called the *maximum segment size (MSS)* governs this size limit. The MSS is determined during connection establishment. Second, TCP is designed so that once a connection is set up, each of the devices tells the other how much data it is ready to accept at any given time. If the data is lower than the MSS value, the device must send a smaller segment. This is part of the sliding window system described a little later in this chapter.

### TCP Data Identification: Sequence Numbers

The fact that TCP treats data coming from an application as a stream of octets has a couple of very significant implications for the operation of the protocol. The first is related to data identification. Since TCP is reliable, it needs to keep track of all the data it receives from an application so it can make sure that the destination receives all the data. Furthermore, TCP must make sure that the destination receives the data in the order that the application sent it, and the destination must retransmit any lost data.

If a device conveyed data to TCP in block-like messages, it would be fairly simple to keep track of the data by adding an identifier to each message. Because TCP is stream-oriented, however, that identification must be done for each byte of data! This may seem surprising, but it is actually what TCP does through the use of sequence numbers. Each byte of data is assigned a sequence number that is used to keep track of it through the process of transmission, reception, and acknowledgment (though in practice, blocks of many bytes are managed using the sequence numbers of bytes at the start and end of the block). These sequence numbers are used to ensure that the sending application transmits and reassembles the segmented data into the original stream of data. The sequence numbers are required to implement the sliding window system, which enables TCP to provide reliability and data flow control.



**Figure 46-1: TCP data stream processing and segment packaging** TCP is different from most protocols because it does not require applications that use it to send data to it in messages. Once a TCP connection is set up, an application protocol can send TCP a steady stream of bytes that does not need to conform to any particular structure. TCP packages these bytes into segments that are sized based on a number of different parameters. These segments are passed to IP, where they are encapsulated into IP datagrams and transmitted. The receiving device reverses the process: Segments are removed from IP datagrams, and then the bytes are taken from the segments and passed up to the appropriate recipient application protocol as a byte stream.

**KEY CONCEPT** Since TCP works with individual bytes of data rather than discrete messages, it must use an identification scheme that works at the byte level to implement its data transmission and tracking system. This is accomplished by assigning a sequence number to each byte that TCP processes.

## **The Need for Application Data Delimiting**

When TCP treats incoming data as a stream, the data the application using TCP receives is called *unstructured*. For transmission, a stream of data goes into TCP on the sending device, and on reception, a stream of data goes back to the application on the receiving device. Even though TCP breaks the stream into segments for transmission, these segments are TCP-level details that remain hidden from the application. When a device wants to send multiple pieces of data, TCP provides no mechanism for indicating where the dividing line is between the pieces, since TCP doesn't examine the meaning of the data. The application must provide a means for doing this.

Consider, for example, an application that is sending database records. It needs to transmit record 579 from the Employees database table, followed by record 581 and record 611. It sends these records to TCP, which treats them all collectively as a stream of bytes. TCP will package these bytes into segments, but in a way that the application cannot predict. It is possible that each byte will end up in a different segment, but more likely that they will all be in one segment, or that part of each will end up in different segments, depending on their length. The records must have some sort of explicit markers so that the receiving device can tell where one record ends and the next starts.

**KEY CONCEPT** Since applications send data to TCP as a stream of bytes as opposed to prepackaged messages, each application must use its own scheme to determine where one application data element ends and the next begins.

## **TCP Sliding Window Acknowledgment System**

What differentiates TCP from simpler transport protocols like UDP is the quality of the manner in which it sends data between devices. Rather than just sticking data in a message and saying, “off you go,” TCP carefully keeps track of the data it sends. This management of data is required to facilitate the following two key requirements of the protocol:

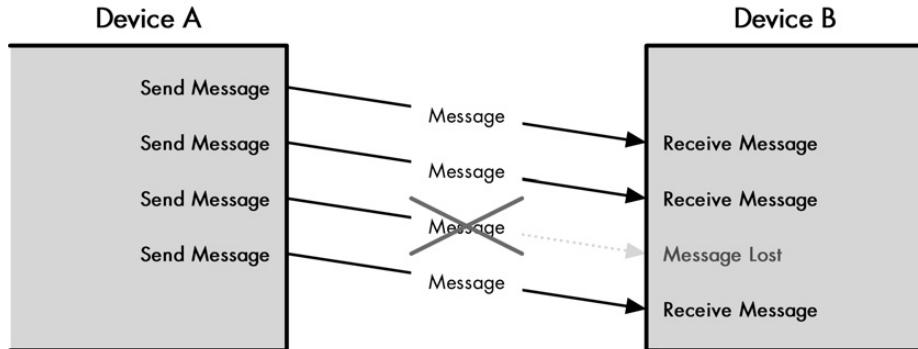
**Reliability** Ensuring that data that is sent actually arrives at its destination, and if it doesn't arrive, detecting this and resending it.

**Data Flow Control** Managing the rate at which data is sent so that it does not overwhelm the device that is receiving it.

To accomplish these tasks, the entire operation of the protocol is oriented around something called the *sliding window acknowledgment system*. It is no exaggeration to say that comprehending how sliding windows work is critical to understanding just about everything else in TCP. It is also, unfortunately, a bit hard to follow if you try to grasp it all at once. I wanted to make sure that I explained the mechanism thoroughly without assuming that you already understood it. For this reason, I am going to start by explaining the concepts behind sliding windows, particularly how the technique works and why it is so powerful.

## The Problem with Unreliable Protocols: Lack of Feedback

A simple “send and forget” protocol like IP is unreliable and includes no flow control for one main reason: It is an open-loop system in which the transmitter receives no feedback from the recipient. (I am ignoring error reports using ICMP and the like for the purpose of this discussion.) A datagram is sent, and it may or may not get there, but the transmitter will never have any way of knowing because there is no mechanism for feedback. This concept is illustrated in Figure 46-2.



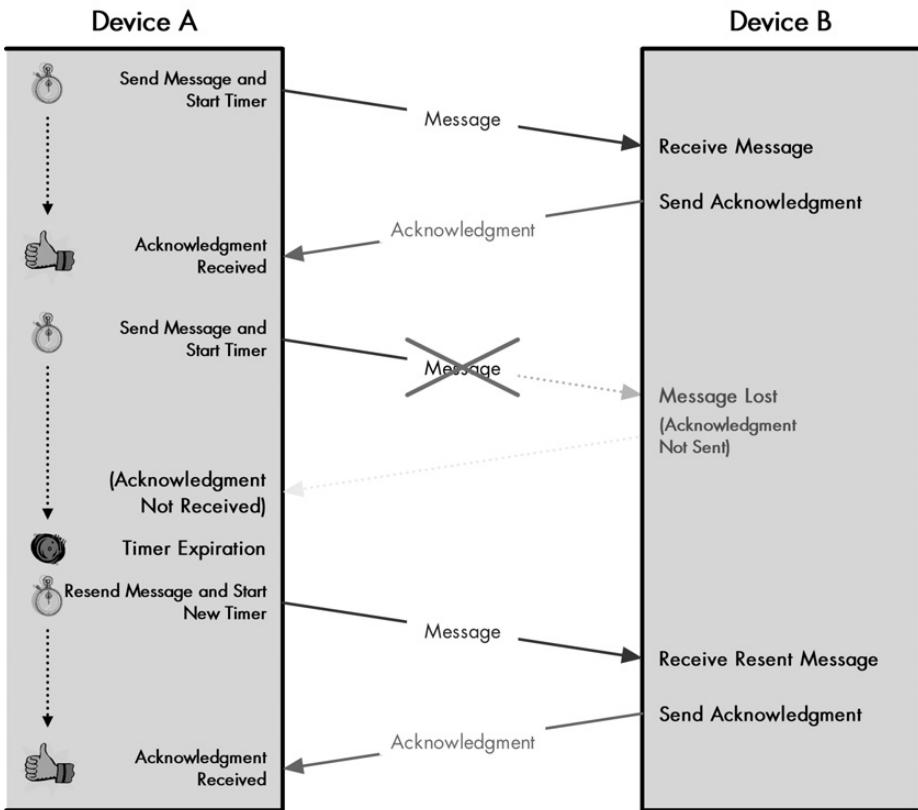
**Figure 46-2: Operation of an unreliable protocol** In a system such as the one that IP uses, if a message gets to its destination, that's great; otherwise, nobody will have a clue. Some external mechanism is needed to take care of the lost message, unless the protocol doesn't really care whether a few bits and pieces are missing from its message stream.

## Providing Basic Reliability Using Positive Acknowledgment with Retransmission (PAR)

Basic reliability in a protocol running over an unreliable protocol like IP can be implemented by closing the loop so the recipient provides feedback to the sender. This is most easily done with a simple acknowledgment system. Device A sends a piece of data to Device B, which receives the data and sends back an acknowledgment saying, “Device A, I received your message.” Device A then knows its transmission was successful.

But since IP is unreliable, that message may in fact never get to where it is going. Device A will sit and wait for the acknowledgment and never receive it. Conversely, it is also possible that Device B gets the message from Device A, but the acknowledgment itself vanishes somehow. In either case, we don't want Device A to sit forever waiting for an acknowledgment that is never going to arrive.

To prevent this from happening, Device A starts a timer when it first sends the message to Device B, which allows sufficient time for the message to get to Device B and for the acknowledgment to travel back, plus some reasonable time to allow for possible delays. If the timer expires before the acknowledgment is received, Device A assumes that there was a problem and retransmits its original message. Since this method involves positive acknowledgments (“Yes, I got your message”) and a facility for retransmission when needed, it is commonly called *positive acknowledgment with retransmission (PAR)*, as shown in Figure 46-3.



**Figure 46-3: Basic reliability: positive acknowledgment with retransmission (PAR)** This diagram shows one of the most common and simple techniques for ensuring reliability. Each time Device A sends a message, it starts a timer. Device B sends an acknowledgment back to Device A when it receives a message, so that Device A knows that it successfully transmitted the message. If a message is lost, the timer goes off, and Device A retransmits the data. Note that only one message can be outstanding at any time, making this system rather slow.

**KEY CONCEPT** A basic technique for ensuring reliability in communications uses a rule that requires a device to send back an acknowledgment each time it successfully receives a transmission. If a device doesn't acknowledge the transmission after a period of time, its sender retransmits the acknowledgment. This system is called *positive acknowledgment with retransmission (PAR)*. One drawback with this basic scheme is that the transmitter cannot send a second message until after the first device has acknowledged the first.

PAR is a technique that is used widely in networking and communications for protocols that exchange relatively small amounts of data, or protocols that exchange data infrequently. The basic method is functional, but it is not well suited to a protocol like TCP. One main reason is that it is *inefficient*. Device A sends a message, and then waits for the acknowledgment. Device A cannot send another message to Device B until it hears that Device B received its original message, which is very wasteful and would make the protocol extremely slow.

## **Improving PAR**

The first improvement we can make to the PAR system is to provide some means of identification to the messages that were sent, as well as the acknowledgments. For example, we could put a message ID field in the message header. The device sending the message would uniquely identify it, and the recipient would use this identifier in the acknowledgment. For example, Device A might send a piece of data in a message with the message ID 1. Device B would receive the message and then send its own message back to Device A, saying “Device A, I received your message 1.” The advantage of this system is that Device A can send multiple messages at once. It must keep track of each one that it sends, and whether or not Device B sent an acknowledgment. Each device also requires a separate timer, but that’s not a big problem.

Of course, we also need to consider this exchange from the standpoint of Device B. Before, Device B had to deal with only one message at a time from Device A. Now it may have several show up all at once. What if it is already busy with transmissions from another device (or ten)? We need some mechanism that lets Device B say, “I am only willing to handle the following number of messages from you at a time.” We could do that by having the acknowledgment message contain a field, such as send limit, which specifies the maximum number of unacknowledged messages Device A was allowed to have in transit to Device B at one time.

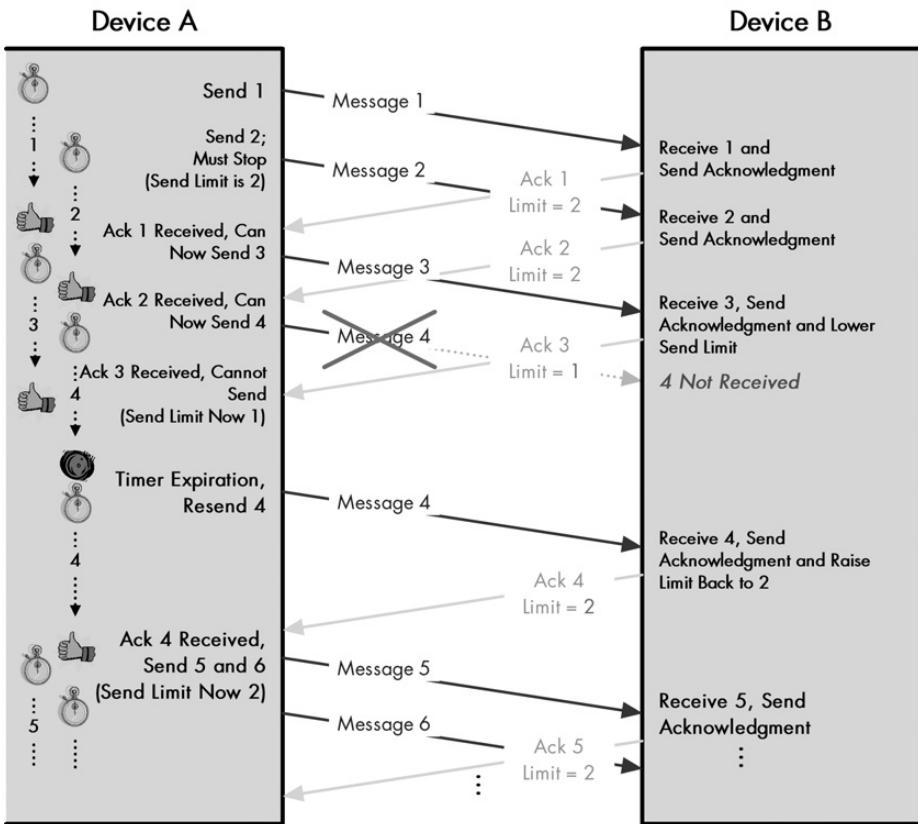
Device A would use this send limit field to restrict the rate at which it sent messages to Device B. Device B could adjust this field depending on its current load and other factors to maximize performance in its discussions with Device A. This enhanced system would thus provide reliability, efficiency, and basic data flow control, as illustrated in Figure 46-4.

**KEY CONCEPT** The basic PAR reliability scheme can be enhanced by identifying each message to be sent, so multiple messages can be in transit at once. The use of a send limit allows the mechanism to also provide flow control capabilities, by allowing each device to control the rate at which other devices send data to it.

## **TCP’s Stream-Oriented Sliding Window Acknowledgment System**

So does TCP use this variation on PAR? Of course not! That would be too simple. Conceptually, the TCP sliding window system is very similar to this method, which is why it is important that you understand it. However, it requires some adjustment. The main reason has to do with the way TCP handles data: the matter of stream orientation compared to message orientation discussed earlier in this chapter. The technique I have outlined involves explicit acknowledgments and (if necessary) retransmissions for messages. Thus, it would work well for a protocol that exchanged reasonably large messages on a fairly infrequent basis.

TCP, on the other hand, deals with individual bytes of data as a stream. Transmitting each byte one at a time and acknowledging each one at a time would quite obviously be absurd. It would require too much work, and even with overlapped transmissions (that is, not waiting for an acknowledgment before sending the next piece of data), the result would be horribly slow.



**Figure 46-4: Enhanced PAR** This diagram shows two enhancements to the basic PAR scheme from Figure 46-3. First, each message now has an identification number; each can be acknowledged individually, so more than one message can be in transit at a given time. Second, Device B regularly communicates to Device A a send limit parameter, which restricts the number of messages Device A can have outstanding at once. Device B can adjust this parameter to control the flow of data from Device A.

This slowness is why TCP does not send bytes individually but divides them into segments. All of the bytes in a segment are sent together and received together, and thus acknowledged together. TCP uses a variation on the method I described earlier, in which the sequence numbers I discussed earlier identify the data sent and acknowledged. Instead of acknowledging the use of something like a message ID field, we acknowledge data using the sequence number of the last byte of data in the segment. Thus, we are dealing with a range of bytes in each case, and the range represents the sequence numbers of all the bytes in the segment.

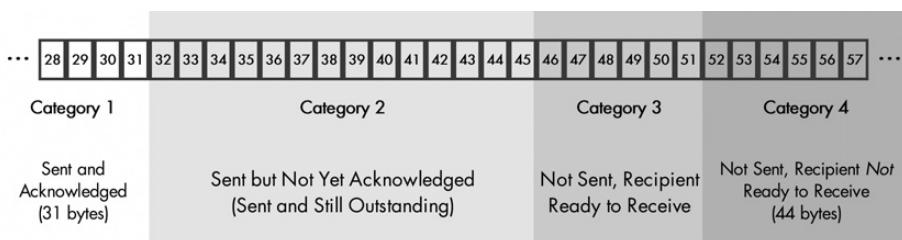
### Conceptual Division of TCP Transmission Stream into Categories

Imagine a newly established TCP connection between Device A and Device B. Device A has a long stream of bytes that it will transmit, but Device B can't accept them all at once, so it limits Device A to sending a particular number of bytes at once in segments, until the bytes in the segments already sent have been

acknowledged. Then Device A is allowed to send more bytes. Each device keeps track of which bytes have been sent and which have not, and which have been acknowledged.

At any point in time, we can take a “snapshot” of the process. If we do, we can conceptually divide the bytes that the sending TCP has in its buffer into the following four categories, and view them as a timeline (see Figure 46-5):

1. **Bytes Sent and Acknowledged** The earliest bytes in the stream will have been sent and acknowledged. These bytes are basically viewed from the standpoint of the device sending data. In the example in Figure 46-5, 31 bytes of data have already been sent and acknowledged. These would fall into category 1.
2. **Bytes Sent but Not Yet Acknowledged** These are the bytes that the device has sent but for which it has not yet received an acknowledgment. The sender cannot consider these handled until they are acknowledged. In Figure 46-5, there are 14 bytes here, in category 2.
3. **Bytes Not Yet Sent for Which Recipient Is Ready** These are bytes that the device has not sent, but which the recipient has room for based on its most recent communication to the sender regarding how many bytes it is willing to handle at once. The sender will try to send these immediately (subject to certain algorithmic restrictions that you’ll explore later). In Figure 46-5, there are 6 bytes in category 3.
4. **Bytes Not Yet Sent for Which Recipient Is Not Ready** These are the bytes further down the stream, which the sender is not yet allowed to send because the receiver is not ready. In Figure 46-5, there are 44 bytes in category 4.



**Figure 46-5: Conceptual division of TCP transmission stream into categories**

**NOTE** I am using very small numbers here to keep the example simple and to make the diagrams a bit easier to construct! TCP does not normally send tiny numbers of bytes around for efficiency reasons.

The receiving device uses a similar system in order to differentiate between data received and acknowledged, data not yet received but ready to receive, and data not yet received and not yet ready to be received. In fact, both devices maintain a separate set of variables to keep track of the categories into which bytes fall in the stream they are sending, as well as the stream they are receiving. This is explored further in Chapter 48’s section named “TCP Sliding Window Data Transfer and Acknowledgment Mechanics,” which describes the detailed sliding window data transfer procedure.

**KEY CONCEPT** The TCP *sliding window system* is a variation on the enhanced PAR system, with changes made to support TCP's stream orientation. Each device keeps track of the status of the byte stream that it needs to transmit. The device keeps track by dividing the byte streams into four conceptual categories: bytes sent and acknowledged, bytes sent but not yet acknowledged, bytes not yet sent but that can be sent immediately, and bytes not yet sent that cannot be sent until the recipient signals that it is ready for them.

## Sequence Number Assignment and Synchronization

The sender and receiver must agree on the sequence numbers that they will assign to the bytes in the stream. This is called *synchronization* and is done when the TCP connection is established. For simplicity, let's assume that the first byte was sent with sequence number 1 (this is not normally the case). Thus, in the example shown in Figure 46-5, the byte ranges for the four categories are as follows:

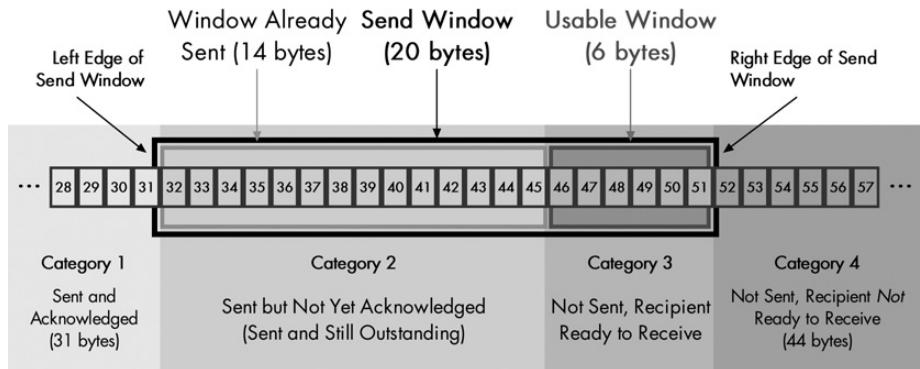
1. The bytes sent and acknowledged are bytes 1 to 31.
2. The bytes sent but not yet acknowledged are bytes 32 to 45.
3. The bytes not yet sent for which the recipient is ready are bytes 46 to 51.
4. The bytes not yet sent for which the recipient is not ready are bytes 52 to 95.

## The Send Window and Usable Window

The key to the operation of the entire process is the number of bytes that the recipient is allowing the transmitter to have unacknowledged at one time. This is called the *send window*, or often, just the *window*. The window is what determines how many bytes the sender is allowed to transmit, and is equal to the sum of the number of bytes in category 2 and category 3. Thus, the dividing line between the last two categories (bytes not sent that the recipient is ready for and bytes the recipient is not ready for) is determined by adding the window to the byte number of the first unacknowledged byte in the stream. In the example shown in Figure 46-5, the first unacknowledged byte is 32. The total window size is 20.

The term *usable window* is defined as the amount of data the transmitter is still allowed to send given the amount of data that is outstanding. It is thus exactly equal to the size of category 3. You may also commonly hear the *edges* of the window mentioned. The left edge marks the first byte in the window (byte 32). The right edge marks the last byte in the window (byte 51). See Figure 46-6 for an illustration of these concepts.

**KEY CONCEPT** The *send window* is the key to the entire TCP sliding window system. It represents the maximum number of unacknowledged bytes that a device is allowed to have outstanding at one time. The *usable window* is the amount of the send window that the sender is still allowed to send at any point in time; it is equal to the size of the send window less the number of unacknowledged bytes already transmitted.

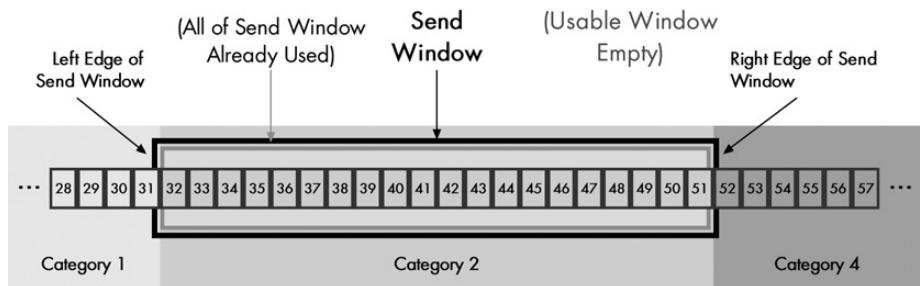


**Figure 46-6: TCP transmission stream categories and send window terminology** This diagram shows the same categories as the ones in Figure 46-5, except that it shows the send window as well. The black box is the overall send window (categories 2 and 3 combined); the light gray box represents the bytes already sent (category 2), and the dark gray box is the usable window (category 3).

### Changes to TCP Categories and Window Sizes After Sending Bytes in the Usable Window

Now let's suppose that in the example shown in Figure 46-6 there is nothing stopping the sender from immediately transmitting the 6 bytes in category 3 (the usable window). When the sender transmits them, the 6 bytes will shift from category 3 to category 2. The byte ranges will now be as follows (see Figure 46-7):

1. The bytes sent and acknowledged are bytes 1 to 31.
2. The bytes sent but not yet acknowledged are bytes 32 to 51.
3. The bytes not yet sent for which the recipient is ready are none.
4. The bytes not yet sent for which the recipient is not ready are bytes 52 to 95.



**Figure 46-7: TCP stream categories and window after sending usable window bytes** This diagram shows the result of the device sending all the bytes that it is allowed to transmit in its usable window. It is the same as Figure 46-6, except that all the bytes in category 3 have moved to category 2. The usable window is now zero and will remain so until it receives an acknowledgment for bytes in category 2.

### Processing Acknowledgments and Sliding the Send Window

Some time later, the destination device sends back a message to the sender and provides an acknowledgment. The destination device will not specifically list out the bytes that it has acknowledged, because as I said earlier, listing the bytes would

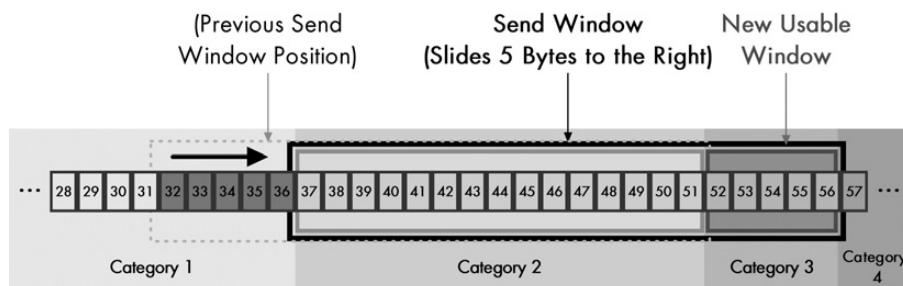
be inefficient. Instead, the destination device will acknowledge a range of bytes that represents the longest contiguous sequence of bytes it has received since the ones it had previously acknowledged.

For example, let's suppose that the bytes already sent but not yet acknowledged at the start of the example (bytes 32 to 45) were transmitted in four different segments. These segments carried bytes 32 to 34, 35 to 36, 37 to 41, and 42 to 45, respectively. The first, second, and fourth segments arrived, but the third did not. The receiver will send back an acknowledgment only for bytes 32 to 36 (32 to 34 and 35 to 36). The receiver will hold bytes 42 to 45 but won't acknowledge them, because this would imply that the receiver has received bytes 37 to 41, which have not shown up yet. This is necessary because TCP is a cumulative acknowledgment system that can use only a single number to acknowledge data. That number is the number of the last contiguous byte in the stream that was successfully received. Let's also say that the destination keeps the window size the same at 20 bytes.

**NOTE** An optional feature called selective acknowledgments does allow noncontiguous blocks of data to be acknowledged. This is explained in Chapter 49's section named "TCP Noncontiguous Acknowledgment Handling and Selective Acknowledgment (SACK)"; we'll ignore this complication for now.

When the sending device receives this acknowledgment, it will be able to transfer some of the bytes from category 2 to category 1, because they have now been acknowledged. When it does so, something interesting will happen. Since 5 bytes have been acknowledged, and the window size didn't change, the sender is allowed to send 5 more bytes. In effect, the window shifts or slides over to the right in the timeline. At the same time 5 bytes move from category 2 to category 1, 5 bytes move from category 4 to category 3, creating a new usable window for subsequent transmission. So, after the groups receive the acknowledgment, they will look like what you see in Figure 46-8. The byte ranges will be as follows:

1. The bytes sent and acknowledged are bytes 1 to 36.
2. The bytes sent but not yet acknowledged are bytes 37 to 51.
3. The bytes not yet sent for which the recipient is ready are bytes 52 to 56.
4. The bytes not yet sent for which the recipient is not ready are bytes 57 to 95.



**Figure 46-8: Sliding the TCP send window** After receiving acknowledgment for bytes 32 to 36, the bytes move from category 2 to 1 (shown in dark shading). The send window shown in Figure 46-7 slides right by 5 bytes; shifting 5 bytes from category 4 to 3, and opening a new usable window.

This process will occur each time an acknowledgment is received, thereby causing the window to slide across the entire stream in order to be transmitted. And thus, ladies and gentlemen, you have the TCP sliding window acknowledgement system!

It is a very powerful technique that allows TCP to easily acknowledge an arbitrary number of bytes using a single acknowledgment number. It provides reliability to the byte-oriented protocol without spending time on an excessive number of acknowledgments. For simplicity, the example I've used here leaves the window size constant, but in reality, it can be adjusted to allow a recipient to control the rate at which data is sent, thereby enabling flow control and congestion handling.

**KEY CONCEPT** When a device gets an acknowledgment for a range of bytes, it knows the destination has successfully received them. It moves them from the "sent but unacknowledged" to the "sent and acknowledged" category. This causes the send window to slide to the right, allowing the device to send more data.

### Dealing with Missing Acknowledgments

But what about bytes 42 through 45 in the example shown in Figure 46-8? Until segment 3 (containing bytes 37 to 41) shows up, the receiving device will not send an acknowledgment for those bytes, and it won't send any others that show up after it. The sending device will be able to send the new bytes that were added to category 3, namely, bytes 52 to 56. The sending device will then stop, and the window will be stuck on bytes 37 to 41.

**KEY CONCEPT** TCP acknowledgments are cumulative and tell a transmitter that the receiving device successfully received all the bytes up to the sequence number indicated in the acknowledgment. Thus, if the receiving device receives bytes out of order, the device cannot acknowledge them until all the preceding bytes are received.

Like the PAR system, TCP includes a system for timing transmissions and retransmitting. Eventually, the TCP device will resend the lost segment. Unfortunately, one drawback of TCP is that since it does not separately acknowledge segments, it may have to retransmit other segments that the recipient actually received (such as the segment with bytes 42 to 45). This starts to get very complex, as I discussed in the topic on TCP retransmissions in Chapter 49.

### More Information on TCP Sliding Windows

Despite the length of this explanation, the preceding is just a summary description of the overall operation of sliding windows. This chapter does not include all of the modifications used in modern TCP! As you can see, the sliding window mechanism is at the heart of the operation of TCP as a whole. In the chapter that describes segments and discusses data transfer, you will see in more detail how TCP transmitters decide how and when to create segments for transmission. Chapter 49 provides much more information about how sliding windows enable a device to manage the flow of data to it on a TCP connection. It also discusses special problems that can

arise if window size is not carefully managed and how you can avoid problems such as congestion in TCP implementations through key changes to the basic sliding window mechanism described in this section.

## TCP Ports, Connections, and Connection Identification

The two TCP/IP transport layer protocols, TCP and UDP, play the same architectural role in the protocol suite, but do it in very different ways. In fact, one of the few functions that the two have in common is that they both provide a method of transport layer addressing and multiplexing. Through the use of *ports*, both protocols allow the data from many different application processes to be aggregated and sent through the IP layer, and then returned up the stack to the proper application process on the destination device. I explain TCP ports in detail in Chapter 43.

Despite this commonality, TCP and UDP diverge somewhat even in how they deal with processes. UDP is a connectionless protocol, which means that devices do not set up a formal connection before sending data. UDP does not have to use sliding windows or keep track of how long it has been since UDP sent a transmission and so forth. When the UDP layer on a device receives data, it just sends it to the process that the destination port indicates, and that's that. UDP can seamlessly handle any number of processes that are sending it messages because UDP handles them all identically.

In contrast, since TCP is connection-oriented, it has many more responsibilities. Each TCP software layer needs to be able to support connections to several other TCPs simultaneously. The operation of each connection is separate from of each other connection, and the TCP software must manage each operation independently. TCP must be sure that it not only routes data to the right process, but that it also manages transmitted data on each connection without any overlap or confusion.

The first consequence of this is that TCP must uniquely identify each connection. It does this by using the pair of socket identifiers that correspond to the two endpoints of the connection, where a socket is simply the combination of the IP address and the port number of each process. This means a socket pair contains four pieces of information: source address, source port, destination address, and destination port. Thus, TCP connections are sometimes said to be described by this addressing quadruple.

I introduced this concept in Chapter 43, where I gave the example of a Hypertext Transfer Protocol (HTTP) request that a client sends at 177.41.72.6 to a website at 41.199.222.3. The server for that website will use well-known port number 80, so the server's socket is 41.199.222.3:80. If the server assigns a client ephemeral port number 3022 for the web browser, the client socket is 177.41.72.6:3022. The overall connection between these devices can be described using this socket pair: (41.199.222.3:80, 177.41.72.6:3022).

This identification of connections using both client and server sockets is what provides the flexibility in allowing multiple connections between devices that we probably take for granted on the Internet. For example, busy application server processes (such as web servers) must be able to handle connections from more than one client; otherwise, the Web would be pretty much unusable. Since the client and server's socket identify the connection, this is no problem. At the same

time that the web server maintains the connection, it can easily have another connection to say, port 2199 at IP address 219.31.0.44. The connection identifier that represents this as follows: (41.199.222.3:80, 219.31.0.44:2199).

In fact, you can have multiple connections from the same client to the same server. Each client process will be assigned a different ephemeral port number, so even if they all try to access the same server process (such as the web server process at 41.199.222.3:80), they will all have a different client socket and represent unique connections. This difference is what lets you make several simultaneous requests to the same website from your computer.

Again, TCP keeps track of each of these connections independently, so each connection is unaware of the others. TCP can handle hundreds or even thousands of simultaneous connections. The only limit is the capacity of the computer running TCP, and the bandwidth of the physical connections to it—the more connections running at once, the more each one has to share limited resources.

**KEY CONCEPT** Each device can handle simultaneous TCP connections to many different processes on one or more devices. The socket numbers of the devices in the connection, called the connection's *endpoints*, identify each connection. Each endpoint consists of the device's IP address and port number, so the four-way communication between client IP address and port number, and server IP address and port number identifies each connection.

## TCP Common Applications and Server Port Assignments

In the overview of TCP in Chapter 45, you saw that the protocol originally included the functions of both modern TCP and IP. TCP was split into TCP and IP in order to allow applications that didn't need TCP's complexity to bypass it, using the much simpler UDP as a transport layer protocol instead. This bypass was an important step in the development of the TCP/IP protocol suite, since there are several important protocols for which UDP is ideally suited, and even some for which TCP is more of a nuisance than a benefit.

Most commonly, however, UDP is used only in special cases. I describe the two types of applications that may be better suited to UDP than TCP in Chapter 44: applications where speed is more important than reliability, and applications that send only short messages infrequently. The majority of TCP/IP applications do not fall into these categories. Thus, even though the layering of TCP and IP means that most protocols aren't required to use TCP, most of them do anyway. The majority of the protocols that use TCP employ all, or at least most, of the features that it provides. The establishment of a persistent connection is necessary for many interactive protocols, such as Telnet, as well as for ones that send commands and status replies, like HTTP. Reliability and flow control are essential for protocols like the File Transfer Protocol (FTP) or the email protocols, which send large files.

Table 46-1 shows some of the more significant application protocols that run on TCP. For each protocol, I have shown the well-known or registered port number that's reserved for that protocol's server process (clients use ephemeral ports, not the port numbers in the table). I have also shown the special keyword shortcut for each port assignment and provided brief comments on why the protocol is well matched to TCP.

**Table 46-1:** Common TCP Applications and Server Port Assignments

Port #	Keyword	Protocol	Comments
20 and 21	ftp-data/ftp	File Transfer Protocol (FTP, data and control)	Used to send large files, so it is ideally suited for TCP.
23	telnet	Telnet Protocol	Interactive session-based protocol. Requires the connection-based nature of TCP.
25	smtp	Simple Mail Transfer Protocol (SMTP)	Uses an exchange of commands, and sends possibly large files between devices.
53	domain	Domain Name Server (DNS)	An example of a protocol that uses both UDP and TCP. For simple requests and replies, DNS uses UDP. For larger messages, especially zone transfers, DNS uses TCP.
70	gopher	Gopher Protocol	A messaging protocol that has been largely replaced by the WWW.
80	http	Hypertext Transfer Protocol (HTTP/World Wide Web)	The classic example of a TCP-based messaging protocol.
110	pop3	Post Office Protocol (POP version 3)	Email message retrieval protocols that use TCP to exchange commands and data.
119	nntp	Network News Transfer Protocol (NNTP)	Used for transferring NetNews (Usenet) messages, which can be lengthy.
139	netbios-ssn	NetBIOS Session Service	A session protocol, clearly better suited to TCP than UDP.
143	imap	Internet Message Access Protocol (IMAP)	Another email message retrieval protocol.
179	bgp	Border Gateway Protocol (BGP)	While interior routing protocols like RIP and OSPF use either UDP or IP directly, BGP runs over TCP. This allows BGP to assume reliable communication even as it sends data over potentially long distances.
194	irc	Internet Relay Chat (IRC)	IRC is like Telnet in that it is an interactive protocol that is strongly based on the notion of a persistent connection between a client and server.
2049	nfs	Network File System (NFS)	NFS was originally implemented using UDP for performance reasons. Given that it is responsible for large transfers of files and given UDP's unreliability, NFS was probably not the best idea, so developers created TCP versions. The latest version of NFS uses TCP exclusively.
6000–6063	TCP	x11	Used for the X Window graphical system. Multiple ports are dedicated to allow many sessions.

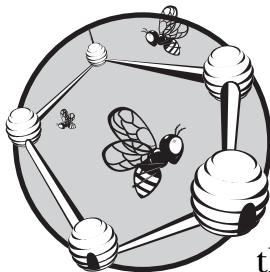
A couple of the protocols in Table 46-1 use both TCP and UDP in order to get the best of both worlds. UDP can send short, simple messages, while TCP moves larger files. Many of the protocols that use both TCP and UDP are actually utility/diagnostic protocols (such as Echo, Discard, and the Time Protocol). These are special cases, because they developers designed them to use both UDP and TCP specifically to allow their use for diagnostics on both protocols.

I have not included an exhaustive list of TCP applications in Table 46-1. See Chapter 42 for common TCP/IP applications and port numbers, and also a reference to the full (massive) list of well-known and registered TCP server ports.



# 47

## TCP BASIC OPERATION: CONNECTION ESTABLISHMENT, MANAGEMENT, AND TERMINATION



While I have described the Transmission Control Protocol (TCP) as *connection-oriented*, this term isn't just any old characteristic of TCP. The overall operation of the entire protocol can be described in terms of how TCP software prepares, negotiates, establishes, manages, and terminates connections. TCP implementations certainly do more than handle connections, but the other major tasks they perform, such as data handling and providing reliability and flow control, can occur only over a stable connection. This stability makes connections the logical place to begin exploring the details of how TCP works.

In this chapter, I describe TCP connections from start to finish. I begin with an overview of TCP's operation by providing a summary of the *finite state machine* that formally defines the stages of a connection. State machines can be a bit mind-boggling when you read about them in standards, but a simplified, explained version provides an excellent high-level view of the life of a connection, so it is a good place to start.

From there, I move on to provide details about TCP’s handling of connections. I describe how you prepare and set up connections and *transmission control blocks* (*TCBs*), and explain the difference between a passive and an active socket Open. I explain the three-way handshake that you can use to create a connection and the method by which parameters are exchanged and sequence numbers synchronized. I talk about how an established connection is managed, including the method by which TCP handles problem conditions and resets the connection when necessary. Finally, I describe how a connection can be terminated when it is no longer needed.

**BACKGROUND INFORMATION** *The following detailed sections assume that you’re familiar with the concepts in the previous chapter, especially the notion of sequence numbers.*

## TCP Operational Overview and the TCP Finite State Machine (FSM)

It is essential that all devices that implement a networking protocol do so in a consistent manner. Otherwise, one device might behave in a manner that the other would not expect. Naturally, this inconsistency is why there are standards that describe the operation of each protocol. The problem with a protocol like TCP is that it performs so many tasks that it is difficult to specify the exact operation of all aspects of the protocol succinctly.

One way that computer scientists describe how a complex protocol works is through a theoretical tool called a *finite state machine* (*FSM*). An FSM attempts to describe a protocol or algorithm by considering it like a virtual machine that progresses through a series of stages of operation in response to various occurrences.

### **Basic FSM Concepts**

You need to understand the following four essential concepts to comprehend the workings of an FSM:

**State** The particular circumstance or status that describes the protocol software on a machine at a given time.

**Transition** The act of moving from one state to another.

**Event** Something that causes a transition to occur between states.

**Action** Something a device does in response to an event before it transitions to another state.

An FSM describes the protocol by explaining all the different states the protocol can be in, the events that can occur in each state, what actions are taken in response to the events, and what transitions happen as a result. The protocol usually starts in a particular *beginning state* when it is first run. It then follows a sequence of steps that get it into a regular operating state, and moves to other states in response to particular types of input or other circumstances. The state machine is called *finite* because there are a limited number of states.

## The Simplified TCP FSM

In the case of TCP, the FSM describes the life stages of a connection. Each connection between one TCP device and another begins in a null state where there is no connection and then proceeds through a series of states until a connection is established. The connection remains in that state until something occurs to cause the connection to be closed again, at which point it proceeds through another sequence of transitional states and returns to the closed state.

**KEY CONCEPT** Many computer scientists use the *finite state machine (FSM)* to describe the operation of a protocol or algorithm. The FSM describes the different actions that a piece of software takes over time by defining a finite number of operating *states*, *events* that can cause *transitions* between states, and *actions* taken in response to events.

The full description of the states, events, and transitions in a TCP connection is lengthy and complicated. This is not surprising, because those three elements would cover much of the entire TCP standard. That level of detail would be a good cure for insomnia, but not much else. However, a simplified look at the TCP FSM will help give you a nice overall feel for how TCP establishes connections and then functions when a connection has been created.

Table 47-1 briefly explains each of the TCP states in a TCP connection, the main events that occur in each state, and what actions and transitions occur as a result. For brevity, three abbreviations are used for the three types of messages that control transitions between states, which correspond to the TCP header flags that are set to indicate that a message is serving that function. These are as follows:

**SYN** A *Synchronize* message; initiates and establishes a connection. It is so named since one of its functions is to synchronize sequence numbers between devices.

**FIN** A *Finish* message, which is a TCP segment with the FIN bit set; it indicates that a device wants to terminate the connection.

**ACK** An *Acknowledgment message*; indicates receipt of a message such as a SYN or a FIN.

Again, I have not shown every possible transition, just the ones normally followed in the life of a connection. Error conditions also cause transitions, but including these would move us well beyond a simplified state machine. The FSM, including how state transitions occur, is illustrated in Figure 47-1.

It's important to remember that this state machine is followed for each connection. This means that, at any given time, TCP may be in one state for one connection to socket X, and in another for its connection to socket Y. Also, the typical movement between states for the two processes in a particular connection is not symmetric, because the roles of the devices are not symmetric. For example, one device initiates a connection, and the other responds; one device starts termination, and the other replies. There is also an alternate path taken for connection establishment and termination if both devices initiate simultaneously (which is unusual, but can happen). This is shown by the shading in Figure 47-1.

**Table 47-1: TCP Finite State Machine (FSM) States, Events, and Transitions**

State	State Description	Event and Transition
CLOSED	The default state that each connection starts in before the process of establishing it begins. The state is called "fictional" in the standard because this state represents the situation in which there is no connection between devices. It either hasn't been created yet or has just been destroyed (if that makes sense).	Passive Open: A server begins the process of connection setup by doing a passive open on a TCP port. At the same time, it sets up the data structure (transmission control block, or TCB) that it needs in order to manage the connection. It then transitions to the LISTEN state.
		Active Open, Send SYN: A client begins the connection setup by sending a SYN message, and it sets up a TCB for this connection. It then transitions to the SYN-SENT state.
LISTEN	A device (normally a server) is waiting to receive a SYN message from a client. It has not yet sent its own SYN message.	Receive Client SYN, Send SYN+ACK: The server device receives a SYN from a client. It sends back a message that contains its own SYN and acknowledges the one it received. The server moves to the SYN-RECEIVED state.
SYN-SENT	The device (normally a client) has sent a SYN message and is waiting for a matching SYN from the other device (usually a server).	Receive SYN, Send ACK: If the device that has sent its SYN message receives a SYN from the other device but not an ACK for its own SYN, it acknowledges the SYN it receives and then transitions to SYN-RECEIVED in order to wait for the acknowledgment to its own SYN.
		Receive SYN+ACK, Send ACK: If the device that sent the SYN receives both an acknowledgment to its SYN and a SYN from the other device, it acknowledges the SYN received and then moves straight to the ESTABLISHED state.
SYN-RECEIVED	The device has received a SYN (connection request) from its partner and sent its own SYN. It is now waiting for an ACK to its SYN in order to finish the connection setup.	Receive ACK: When the device receives the ACK to the SYN that it sent, it transitions to the ESTABLISHED state.
ESTABLISHED	The steady state of an open TCP connection. Both devices can exchange data freely once both devices in the connection enter this state. This will continue until they close the connection.	Close, Send FIN: A device can close the connection by sending a message with the FIN bit sent, and then it can transition to the FIN-WAIT-1 state.
		Receive FIN: A device may receive a FIN message from its connection partner asking that the connection be closed. It will acknowledge this message and transition to the CLOSE-WAIT state.
CLOSE-WAIT	The device has received a close request (FIN) from the other device. It must now wait for the application on the local device to acknowledge this request and generate a matching request.	Close, Send FIN: The application using TCP, having been informed that the other process wants to shut down, sends a close request to the TCP layer on the machine on which it is running. TCP then sends a FIN to the remote device that already asked to terminate the connection. This device now transitions to LAST-ACK.
LAST-ACK	A device that has already received a close request and acknowledged has sent its own FIN and is waiting for an ACK to this request.	Receive ACK for FIN: The device receives an acknowledgment for its close request. We have now sent our FIN and had it acknowledged, and received the other device's FIN and acknowledged it, so we go straight to the CLOSED state.
FIN-WAIT-1	A device in this state is waiting for an ACK for a FIN it has sent, or is waiting for a connection termination request from the other device.	Receive ACK for FIN: The device receives an acknowledgment for its close request. It transitions to the FIN-WAIT-2 state.
		Receive FIN, Send ACK: The device does not receive an ACK for its own FIN, but receives a FIN from the other device. It acknowledges it and then moves to the CLOSING state.

(continued)

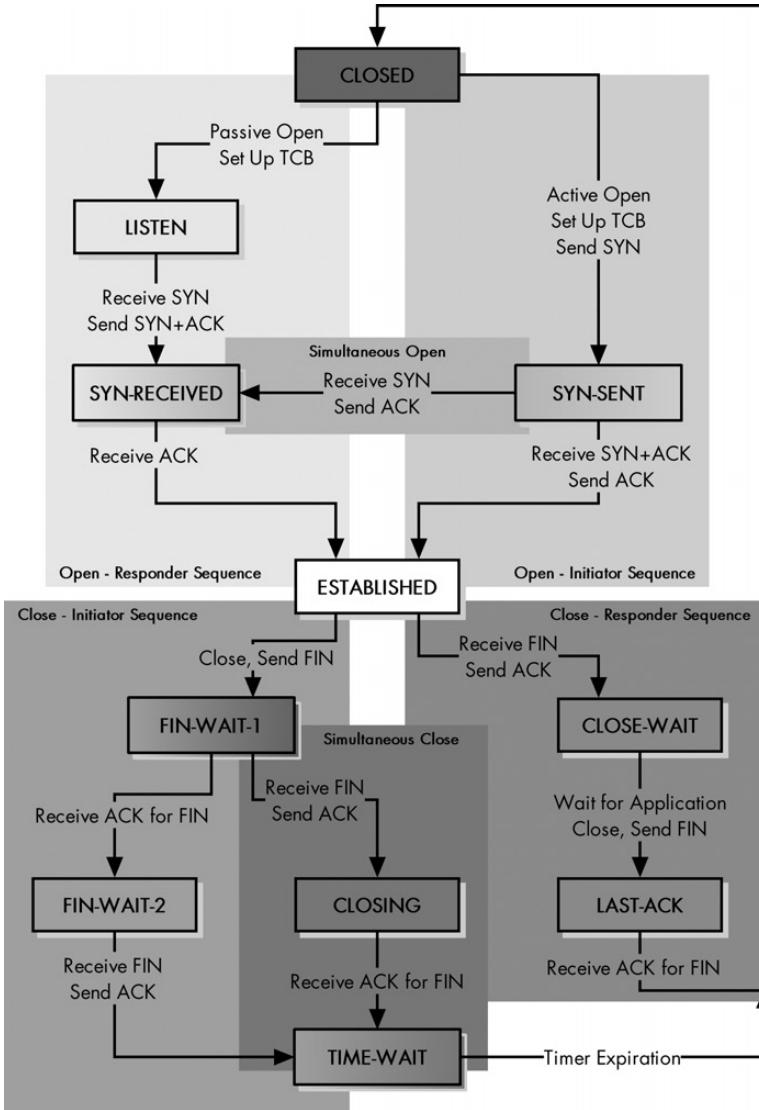
**Table 47-1:** TCP Finite State Machine (FSM) States, Events, and Transitions (continued)

State	State Description	Event and Transition
FIN-WAIT-2	A device in this state has received an ACK for its request to terminate the connection and is now waiting for a matching FIN from the other device.	Receive FIN, Send ACK: The device receives a FIN from the other device. It acknowledges it and then moves to the TIME-WAIT state.
CLOSING	The device has received a FIN from the other device and has sent an ACK for it, but has not yet received an ACK for its own FIN message.	Receive ACK for FIN: The device receives an acknowledgment for its close request. It transitions to the TIME-WAIT state.
TIME-WAIT	The device has now received a FIN from the other device and acknowledged it, and sent its own FIN and received an ACK for it. We are finished, except for waiting to ensure the ACK is received and preventing potential overlap with new connections. (See the “TCP Connection Termination” section later in this chapter for more details on this state.)	Timer Expiration: After a designated wait period, the device transitions to the CLOSED state.

Thus, for example, at the start of connection establishment, the two devices will take different routes to get to the ESTABLISHED state. One device (the server usually) will pass through the LISTEN state, while the other (the client) will go through SYN-SENT state. Similarly, one device will initiate connection termination and take the path through the FIN-WAIT-1 state in order to get back to the CLOSED state; the other will go through the CLOSE-WAIT and LAST-ACK states. However, if both try to open at once, they each proceed through SYN-SENT and SYN-RECEIVED states, and if both try to close at once, they go through FIN-WAIT-1, CLOSING, and TIME-WAIT states roughly simultaneously.

Although FSM may seem a bit intimidating at first, if you spend a few minutes with it, you can get a good handle on how TCP works. The FSM will be of great use in making sense of the connection establishment and termination processes discussed later in this chapter, and reading those sections will help you make sense of the FSM.

**KEY CONCEPT** The TCP finite state machine (FSM) describes the sequence of steps that both devices take in a TCP session as they establish, manage, and close the connection. Each device may take a different path through the states, because under normal circumstances, the operation of the protocol is not symmetric—one device initiates connection establishment or termination, and the other responds.



**Figure 47-1: The TCP finite state machine (FSM)** This diagram illustrates the simplified TCP FSM. The shadings are not an official part of the definition of the FSM; I have added them to show more clearly the sequences the two devices took to open and close a link. For establishment and termination, there is a regular sequence, in which the initiating and responding devices go through different states, and a simultaneous sequence, in which each uses the same sequence.

## TCP Connection Preparation

In Chapter 43, I raised an important point about TCP operation, particularly that it must be capable of handling many connections simultaneously. For this reason, we must uniquely identify each connection using the *quadruple* of the socket identifiers

(IP address and port number) for each of the two devices on the connection. The process of setting up, managing, and terminating a connection is performed independently for each connection.

### ***Storing Connection Data: The Transmission Control Block (TCB)***

Since each connection is distinct, we must maintain data about each connection separately. TCP uses a special data structure for this purpose, called a *transmission control block (TCB)*. The TCB contains all the important information about the connection, such as the two socket numbers that identify it and pointers to buffers that hold incoming and outgoing data. The TCB also implements the sliding window mechanism. It holds variables that keep track of the number of bytes received and acknowledged, bytes received and not yet acknowledged, current window size, and so forth. Each device maintains its own TCB for the connection.

Before the process of setting up a TCP connection can begin, the devices on each end must perform some “prep work.” One of the tasks required in order to prepare for the connection is to set up the TCB that will be used to hold information about it. This is done right at the very start of the connection establishment process, when each device transitions out of the CLOSED state.

### ***Active and Passive Opens***

TCP/IP is based on the client/server model of operation, and TCP connection setup is based on the existence of these roles as well. The client and server each prepare for the connection by performing an *Open* operation. However, there are two different kinds of Open operations:

**Active Open** A client process using TCP takes the active role and initiates the connection by sending a TCP message to start the connection (a SYN message).

**Passive Open** A server process designed to use TCP takes a more “laid-back” approach. It performs a *passive Open* by contacting TCP and saying, “I’m here, and I’m waiting for clients that may wish to talk to me to send me a message on the following port number.” The Open is called *passive* because, aside from indicating that the process is listening, the server process does nothing. A passive Open can specify that the server is waiting for an active Open from a specific client, though not all TCP/IP APIs support this capability. More commonly, a server process is willing to accept connections from all comers. Such a passive Open is said to be *unspecified*.

**KEY CONCEPT** A client process initiates a TCP connection by performing an *active Open*, sending a SYN message to a server. A server process using TCP prepares for an incoming connection request by performing a *passive Open*. For each TCP session, both devices create a data structure, called a *transmission control block (TCB)*, that is used to hold important data related to the connection.

## **Preparation for Connection**

Both the client and the server create the TCB for the connection at the time that they perform the Open. The client already knows the IP addresses and port numbers for both the client process and the server process it is trying to reach, so it can use these to uniquely identify the connection and the TCB that goes with it.

For the server, the concept of a TCB at this stage of the game is a bit more complex. If the server is waiting for a particular client, it can identify the connection using its own socket and the socket of the client for which it is waiting. Normally, however, the server doesn't know which client is trying to reach it. In fact, more than one client could contact it at nearly the same time.

In this case, the server creates a TCB with an unspecified (zero) client socket number and waits to receive an active Open. It then *binds* the socket number of the client to the TCB for the passive Open as part of the connection process. To allow the server to handle multiple incoming connections, the server process may perform several unspecified passive Opens simultaneously.

The TCB for a connection is maintained throughout the connection and destroyed when the connection is completely terminated, and the device returns to the CLOSED state. TCP does include a procedure that handles the situation in which both devices perform an active Open simultaneously, as I discuss the next section.

## **TCP Connection Establishment Process: The Three-Way Handshake**

Before TCP can be employed for any actually useful purpose—that is, sending data—a connection must be set up between the two devices that wish to communicate. This process, usually called *connection establishment*, involves an exchange of messages that transitions both devices from their initial connection state (CLOSED) to the normal operating state (ESTABLISHED).

### **Connection Establishment Functions**

The connection establishment process actually accomplishes the following tasks as it creates a connection suitable for data exchange:

**Contact and Communication** The client and server make contact with each other and establish communication by sending each other messages. The server usually doesn't even know which client it will be talking to before this point, so it discovers this during connection establishment.

**Sequence Number Synchronization** Each device lets the other know what initial sequence number it wants to use for its first transmission.

**Parameter Exchange** The two devices exchange certain parameters that control the operation of the TCP connection.

I'll discuss the sequence number synchronization and parameter exchange tasks in the "TCP Connection Establishment Sequence Number Synchronization and Parameter Exchange" section later in this chapter.

## **Control Messages Used for Connection Establishment: SYN and ACK**

TCP uses control messages to manage the process of contact and communication. There aren't, however, any special TCP control message types; all TCP messages use the same segment format. A set of control flags in the TCP header indicates whether a segment is being used for control purposes or just to carry data. As I introduced in the discussion of the TCP FSM earlier in the chapter, two control message types are used in connection setup, which are specified by setting the following two flags:

**SYN** Indicates that the segment is being used to initialize a connection. SYN stands for *synchronize*, in reference to the sequence number synchronization task in the connection establishment process.

**ACK** Indicates that the device sending the segment is conveying an *acknowledgment* for a message it has received (such as a SYN).

There are also other control bits (*FIN*, *RST*, *PSH*, and *URG*) that aren't important to connection establishment, so I will discuss them in other topics. In common TCP parlance, a message with a control bit set is often named for that bit. For example, if the SYN control bit is set, the segment is often called a SYN message. Similarly, a segment with the ACK bit set is an ACK message, or even just an ACK.

## **Normal Connection Establishment: The Three-Way Handshake**

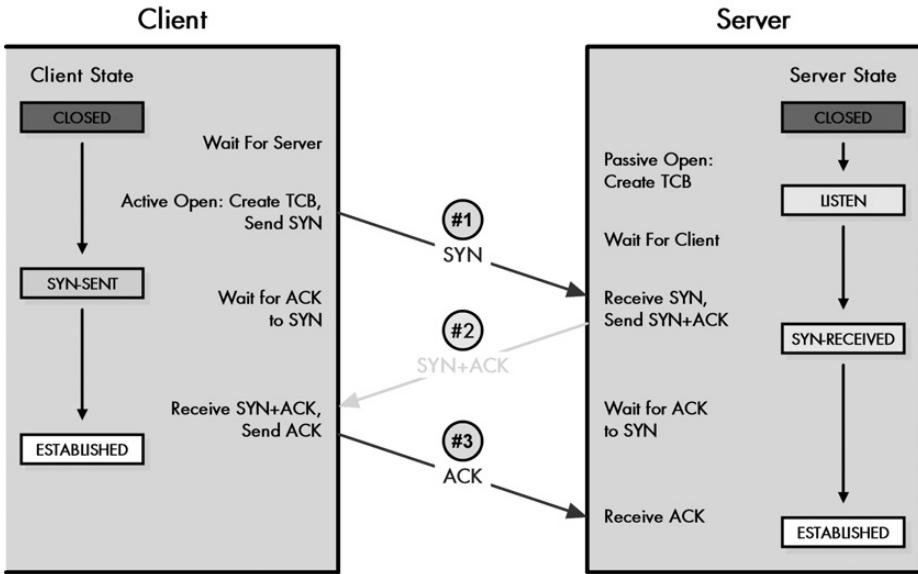
To establish a connection, each device must send a SYN message and receive an ACK message for it from the other device. Thus, conceptually, we need to have four control messages pass between the devices. However, it's inefficient to send a SYN and an ACK in separate messages when one could communicate both simultaneously. Thus, in the normal sequence of events in connection establishment, one of the SYNs and one of the ACKs are sent together by setting both of the relevant bits (a message sometimes called a *SYN+ACK*). This makes a total of three messages, and for this reason the connection procedure is called a *three-way handshake*.

**KEY CONCEPT** The normal process of establishing a connection between a TCP client and server involves the following three steps: The client sends a SYN message. The server sends a message that combines an ACK for the client's SYN and contains the server's SYN. And the client sends an ACK for the server's SYN. This is called the *TCP three-way handshake*.

Table 47-2 describes in detail how the three-way handshake works (including a summary of the preparation discussed in the previous section). It is adapted from the table describing the TCP FSM (Table 47-1), but shows what happens for both the server and the client over time. Each row shows the state the device begins in, what action it takes in that state, and the state to which it transitions. The transmit and receive parts of each of the three steps of the handshake process are shown as well. The same process is also illustrated in 47-2.

**Table 47-2:** TCP Three-Way Handshake Connection Establishment Procedure

Client			Server		
Start State	Action	Move to State	Start State	Action	Move to State
CLOSED	The client cannot do anything until the server has performed a passive Open and is ready to accept a connection.	—	CLOSED	The server performs a passive Open, creating a TCB for the connection and readying itself for the receipt of a connection request (SYN) from a client.	LISTEN
CLOSED	Step 1 Transmit: The client performs an active Open, creating a TCB for the connection and sending a SYN message to the server.	SYN-SENT	LISTEN	The server waits for contact from a client.	—
SYN-SENT	The client waits to receive an ACK to the SYN that it has sent, as well as the server's SYN.	—	LISTEN	Step 1 Receive, Step 2 Transmit: The server receives the SYN from the client. It sends a single SYN+ACK message back to the client that contains an ACK for the client's SYN, as well as the server's own SYN.	SYN- RECEIVED
SYN-SENT	Step 2 Receive, Step 3 Transmit: The client receives from the server the SYN+ACK containing the ACK to the client's SYN, and the SYN from the server. It sends the server an ACK for the server's SYN. The client is now finished with the connection establishment.	ESTABLISHED	SYN- RECEIVED	The server waits for an ACK to the SYN it sent previously.	—
ESTABLISHED	The client is waiting for the server to finish connection establishment so they can operate normally.	—	SYN- RECEIVED	Step 3 Receive: The server receives the ACK to its SYN and is now finished with connection establishment.	ESTABLISHED
ESTABLISHED	The client is ready for normal data transfer operations.	—	ESTABLISHED	The server is ready for normal data transfer operations.	—



**Figure 47-2: TCP three-way handshake connection establishment procedure** This diagram illustrates how a client and server establish a conventional connection. It shows how the three messages sent during the process and how each device transitions from the CLOSED state through intermediate states until the session is in the ESTABLISHED state.

### Simultaneous Open Connection Establishment

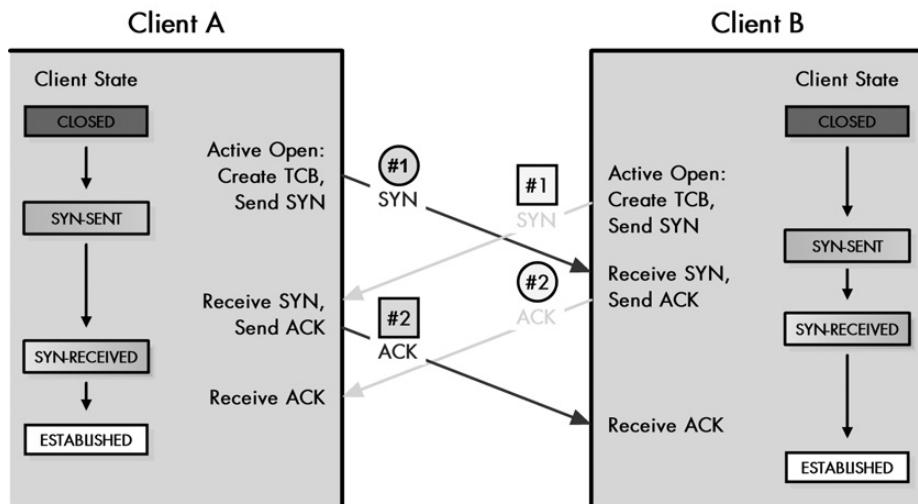
TCP is also set up to handle the situation in which both devices perform an active Open instead of one doing a passive Open. This may occur if two clients are trying to reach each other instead of a client and a server. It is uncommon, however, and only happens under certain circumstances. Simultaneous connection establishment can also happen only if one of the devices uses a well-known port as its source port.

In the case of simultaneous open connection establishment, the steps are different for both devices. Each client will perform an active Open and will then proceed through both the SYN-SENT and SYN-RECEIVED states until the clients acknowledge each other's SYNs. This means that there is no three-way handshake; instead, there is something like two simultaneous two-way handshakes. Each client sends a SYN, receives the other's SYN, acknowledges the SYN with an ACK it, and then waits for its own ACK.

I have described the transaction for establishing open connections simultaneously, in a simplified way, in Table 47-3 and illustrated it in Figure 47-3. To limit the table size, I have shown the activities performed by the two devices occurring simultaneously (in the same row). In reality, the actions don't need to occur at exactly the same time and probably won't. All that must happen for the simultaneous procedure to be followed is that each device receives a SYN before getting an ACK for its own SYN, as Figure 47-3 shows.

**Table 47-3: TCP Simultaneous Open Connection Establishment Procedure**

Client A			Client B		
Start State	Action	Move to State	Start State	Action	Move to State
CLOSED	Client A Step 1 Transmit: Client A performs an active Open, creating a TCB and sending a SYN to the server.	SYN-SENT	CLOSED	Client B Step 1 Transmit: Client B performs an active Open, creating a TCB and sending a SYN to the server.	SYN-SENT
SYN-SENT	Client B Step 1 Receive and Step 2 Transmit: Client A receives Client B's SYN and sends it an ACK. It is still waiting for an ACK to its own SYN.	SYN-RECEIVED	SYN-SENT	Client A Step 1 Receive and Step 2 Transmit: Client B receives Client A's SYN and sends it an ACK. It is still waiting for an ACK to its own SYN.	SYN-RECEIVED
SYN-RECEIVED	Client A Step 2 Receive: Client A receives the ACK from Client B for its SYN and finishes connection establishment.	ESTABLISHED	SYN-RECEIVED	Client B Step 2 Receive: Client B receives the ACK from Client A for its SYN and finishes connection establishment.	ESTABLISHED



**Figure 47-3: TCP simultaneous open connection establishment procedure** This diagram shows what happens when two devices try to open a connection to each other at the same time. In this case, instead of a three-way handshake, each sends a SYN and receives an ACK. They each follow the same sequence of states, which differs from both sequences in the normal three-way handshake.

**KEY CONCEPT** If one device setting up a TCP connection sends a SYN and then receives a SYN from the another device before it acknowledges its SYN, the two devices perform a **simultaneous OPEN**, which consists of the exchange of two independent SYN and ACK message sets. The end result is the same as the conventional three-way handshake, but the process of getting to the ESTABLISHED state is different.

## TCP Connection Establishment Sequence Number Synchronization and Parameter Exchange

The TCP three-way handshake describes the mechanism of message exchange that allows a pair of TCP devices to move from a closed state to one that is a ready-to-use, established connection. Connection establishment is about more than just passing messages between devices in order to establish communication. The TCP layers on the devices must also exchange information about the sequence numbers each device wants to use for its first data transmission. The layers must also exchange information about the parameters that will control how the connection operates. The sequence numbers exchange is usually called *sequence number synchronization*, and it is such an important part of connection establishment that the messages that each device sends to start the connection are called *SYN (synchronization)* messages.

You may recall from the TCP fundamentals discussion in Chapter 46 that TCP refers to each byte of data individually and uses sequence numbers to keep track of which bytes have been sent and received. Since each byte has a sequence number, we can acknowledge each byte, or more efficiently, use a single number to acknowledge a range of bytes received.

In the example I gave in Chapter 46, I assumed that each device would start a connection by giving the first byte of data sent between them sequence number 1. A valid question is why wouldn't we *always* just start off each TCP connection by sending the first byte of data with a sequence number of 1? The sequence numbers are arbitrary, after all, and this is the simplest method. In an ideal world, this would probably work, but we don't live in an ideal world.

The problem with starting off each connection with a sequence number of 1 is that it introduces the possibility of segments from different connections getting mixed up. Suppose we established a TCP connection and sent a segment containing bytes 1 through 30. However, a problem with the internetwork caused a delay with this segment, and eventually, the TCP connection itself was terminated. We then started up a new connection and again used a starting sequence number of 1. As soon as this new connection was started, however, the old segment with bytes labeled 1 to 30 showed up. The other device would erroneously think those bytes were part of the *new* connection.

This is but one of several similar problems that could occur. To avoid them, each TCP device, at the time a connection is initiated, chooses a 32-bit *initial sequence number (ISN)* for the connection. Each device has its own ISN, and those ISNs normally won't be the same.

### Initial Sequence Number Selection

Traditionally, each device chose the ISN by making use of a timed counter, like a clock of sorts, that was incremented every 4 microseconds. TCP initialized the counter when it started up, and then the counter's value increased by one every 4 microseconds until it reached the largest 32-bit value possible (4,294,967,295), at which point it wrapped around to 0 and resumed incrementing. Any time a new connection was set up, the ISN was taken from the current value of this timer. Since

it takes over 4 hours to count from 0 to 4,294,967,295 at 4 microseconds per increment, this virtually ensured that each connection would not conflict with any previous ones.

One issue with this method is that it made ISNs predictable. A malicious person could write code to analyze ISNs and then predict the ISN of a subsequent TCP connection based on the ISNs used in earlier ones. Malicious hackers have exploited this security risk in the past (such as in the case of the famous Mitnick attack). To defeat the malicious hackers, implementations now use a random number in their ISN selection process.

### TCP Sequence Number Synchronization

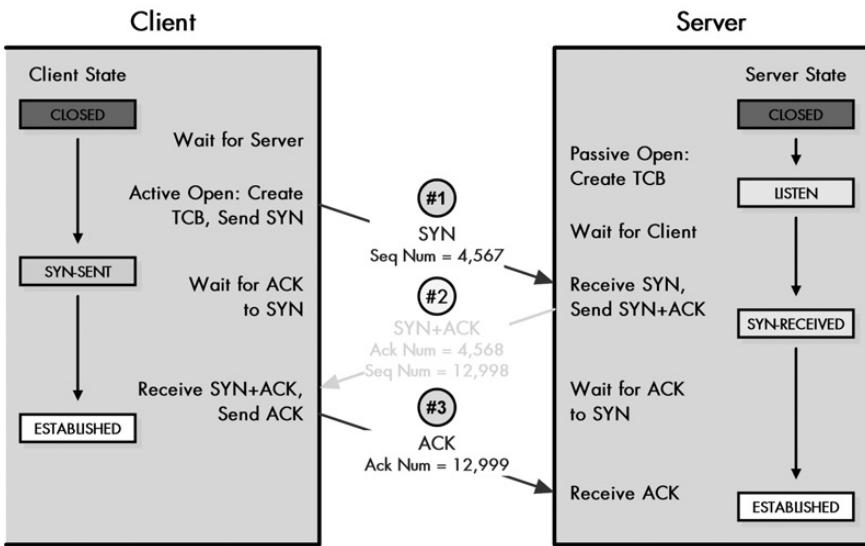
Once each device chooses its ISN, it sends the ISN value to the other device in the Sequence Number field in the device's initial SYN message. The device receiving the SYN responds with an ACK message that acknowledges the SYN (which may also contain its own SYN, as in step 2 of the three-way handshake). In the ACK message, the Acknowledgment Number field is set to the value of the ISN that is received from the other device *plus one*. This represents the next sequence number the device expects to receive from its peer; the ISN actually represents the sequence number of the last byte received (fictitious in this case, since the connection is new and nothing yet has been received).

**KEY CONCEPT** As part of the process of connection establishment, each of the two devices in a TCP connection informs the other of the sequence number it plans to use for its first data transmission. Each device informs the other by putting the preceding sequence number in the Sequence Number field of its SYN message. The other device confirms this by incrementing that value and putting it into the Acknowledgment Number field of its ACK message, telling the other device that it is the sequence number it is expecting for the first data transmission. This process is called *sequence number synchronization*.

Here's a simplified example of the three-way handshake steps (see Figure 47-4). I chose small ISNs for readability, but remember that ISNs can be any 32-bit number.

1. **Connection Request by Client** The client chooses an ISN for its transmissions of 4,567. It sends a SYN with the Sequence Number field set to 4,567.
2. **Acknowledgment and Connection Request by Server** The server chooses an ISN for its transmissions of 12,998. It receives the client's SYN. It sends a SYN+ACK with an Acknowledgment Number field value of 4,568 (one more than the client's ISN). This message has a Sequence Number field value of 12,998.
3. **Acknowledgment by Client** The client sends an ACK with the Acknowledgment Number field set to 12,999.

With the connection now established, the client will send data whose first byte will be given sequence number 4,568. The server's first byte of data will be numbered 12,999.



**Figure 47-4: TCP sequence number synchronization** This diagram illustrates the same three-way handshake connection establishment procedure that I introduced in Figure 47-2, except this time I have shown the Sequence Number and Acknowledgment Number fields in each message, so that you can see how each of the two devices use them to establish initial sequence numbers for data exchange.

### TCP Parameter Exchange

In addition to the initial sequence numbers, SYN messages also are designed to convey important parameters about how the connection should operate. TCP includes a flexible scheme for carrying these parameters, in the form of a variable-length *Options* field in the TCP segment format, which can be expanded to carry multiple parameters. In RFC 793, only a single parameter is defined to be exchanged during connection setup: *maximum segment size (MSS)*. I explain the significance of this parameter in the TCP data transfer discussion in Chapter 48.

Each device sends the other the MSS that it wants to use for the connection; that is, if the device wishes to use a nondefault value. When receiving the SYN, the server records the MSS value that the client sent, and it will never send a segment larger than that value to the client. The client does the same for the server. The client and server MSS values are independent, so they can establish a connection where the client can receive larger segments than the server or vice versa.

Later RFCs have defined additional parameters that may be exchanged during connection setup. Some of these include the following:

**Window Scale Factor** Allows a pair of devices to specify larger window sizes than would normally be possible given the 16-bit size of the TCP *Window* field.

**Selective Acknowledgment Permitted** Allows a pair of devices to use the optional selective acknowledgment feature to allow only certain lost segments to be retransmitted.

**Alternate Checksum Method** Lets devices specify an alternative method of performing checksums than the standard TCP checksum mechanism.

## TCP Connection Management and Problem Handling

Once both of the devices in a TCP connection have completed connection setup and have entered the ESTABLISHED state, the TCP software is in its normal operating mode. The TCP software will package bytes of data into segments for transmission using the mechanisms described in Chapter 48. TCP will use the sliding windows scheme to control segment size and to provide flow control, congestion handling, and retransmissions as needed.

Once in the sliding windows mode, both devices can remain there indefinitely. Some TCP connections can be very long-lived—in fact, some users maintain certain connections like Telnet sessions for hours or even days at a time. The following two circumstances can cause a connection to move out of the ESTABLISHED state:

**Connection Termination** Either of the devices decides to terminate the connection. This involves a specific procedure that I cover in the “TCP Connection Termination” section later in this chapter.

**Connection Disruption** A problem of some sort occurs and interrupts the connection.

### The TCP Reset Function

In order for it to live up to its job of being a reliable and robust protocol, TCP includes intelligence that allows it to detect and respond to various problems that can occur during an established connection. One of the most common is the *half-open connection*. This situation occurs when, due to some sort of problem, one device closes or aborts the connection without the other one knowing about it. This means one device is in the ESTABLISHED state, while the other may be in the CLOSED state (no connection) or some other transient state. This could happen if, for example, one device had a software crash and someone restarted it in the middle of a connection, or if some sort of glitch caused the states of the two devices to become unsynchronized.

To handle half-open connections and other problem situations, TCP includes a special *reset function*. A *reset* is a TCP segment that TCP sends with the *RST* flag set to 1 in its header. Generally speaking, the TCP software generates a reset whenever something unexpected happens. The following are some of the most common cases in which the TCP software generates a reset:

- Receipt of any TCP segment from any device with which the device receiving the segment does not currently have a connection (other than a SYN requesting a new connection)
- Receipt of a message with an invalid or incorrect Sequence Number or Acknowledgment Number field, indicating that the message may belong to a prior connection or is spurious in some other way
- Receipt of a SYN message on a port where there is no process listening for connections

## **Handling Reset Segments**

When a device receives a segment with the RST bit, it tells the other device to reset the connection so that the device can reestablish the connection. Like all segments, the reset itself must be checked to ensure that it is valid (by looking at the value of its Sequence Number field). This check prevents a spurious reset from shutting down a connection. Assuming the reset is valid, the handling of the message depends on the state of the device that receives it, as follows:

- If the device is in the LISTEN state, it ignores the reset and remains in that state.
- If the device is in the SYN-RECEIVED state but was previously in the LISTEN state (which is the normal course of events for a server setting up a new connection), it returns to the LISTEN state.
- In any other situation, the reset causes the device to abort the connection and the device returns to the CLOSED state for that connection. The device will advise the higher-layer process that is using TCP that it has closed the connection.

**KEY CONCEPT** TCP includes a special *connection reset feature* that allows devices to deal with problem situations, such as *half-open connections* or the receipt of unexpected message types. To use the feature, the device detecting the problem sends a TCP segment with the RST (reset) flag set to 1. The receiving device either returns to the LISTEN state, if it was in the process of connection establishment, or closes the connection and returns to the CLOSED state pending a new session negotiation.

## **Idle Connection Management and Keepalive Messages**

One final connection management issue in TCP is how to handle an idle connection; that is, a TCP session that is active but that has no data being transmitted by either device for a prolonged period of time. The TCP standard specifies that the appropriate action to take in this situation is nothing. The reason is that, strictly speaking, there is no need to do anything to maintain an idle connection in TCP. The protocol is perfectly happy to allow both devices to stop transmitting for a very long period of time. Then it simply allows both devices to resume transmissions of data and acknowledgment segments when each one has data to send.

However, in the same way that people become antsy when they are on a telephone call and don't hear anything for a long time, some TCP implementors were concerned that an idle TCP connection might mean that something had broken the connection.

Thus, TCP software often includes an unofficial feature that allows a device with a TCP link to periodically send a null segment, which contains no data, to its peer on the connection. If the connection is still valid, the other device responds with a segment that contains an acknowledgment; if it is not, the other device will reply with a connection reset segment as I described earlier. These segments are sometimes called TCP *keepalive messages*, or just *keepalives*. They are analogous to Border Gateway Protocol (BGP) Keepalive messages (described in Chapter 40).

The use of these messages is quite controversial, and therefore, not universal. Those who oppose using them argue that they are not really necessary, and that sending them represents a waste of internetwork bandwidth and a possible additional cost on metered links (those that charge for each datagram sent). Their key point is that if the connection is not presently being used, it doesn't matter if it is still valid or not; as soon as the connection is used again, if it has broken, in the meantime, TCP can handle that using the reset function mentioned earlier.

Sending a keepalive message can, in theory, break a good TCP session unnecessarily. This may happen if the keepalive is sent during a time when there is an intermittent failure between the client and server. The failure might otherwise have corrected itself by the time the next piece of real data must be sent. In addition, some TCP implementations may not properly deal with the receipt of these segments.

Those in favor of using keepalives point out that each TCP connection consumes a certain amount of resources, and this can be an issue, especially for busy servers. If many clients connect to such a server and don't terminate the TCP connection properly, the server may sit for a long time with an idle connection, using system memory and other resources that it could apply elsewhere.

Since there is no wide acceptance on the use of this feature, devices implementing it include a way to disable it if necessary. Devices are also programmed so that they will not terminate a connection simply because they did not receive a response to a single keepalive message. They may terminate the connection if they do not receive a reply after several such messages have been sent over a period of time.

## TCP Connection Termination

As the saying goes, all good things must come to an end, and so it is with TCP connections. The link between a pair of devices can remain open for a considerable period of time, assuming that a problem doesn't force the device to abort the connection. Eventually, however, one or both of the processes in the connection will run out of data to send and will shut down the TCP session, or the user will instruct the device to shut down.

### ***Requirements and Issues In Connection Termination***

Just as TCP follows an ordered sequence of operations in order to establish a connection, it also includes a specific procedure for terminating a connection. As with connection establishment, each of the devices moves from one state to the next in order to terminate the connection. This process is more complicated than you might imagine. In fact, an examination of the TCP FSM shows that there are more distinct states involved in shutting down a connection than in setting one up.

The reason that connection termination is complex is that during normal operation, both devices are sending and receiving data simultaneously. Usually, connection termination begins with one device indicating to TCP that it wants to close the connection. The matching process on the other device may not be aware that its peer wants to end the connection at all. Several steps are required to ensure that both devices shut down the connection gracefully and that no data is lost in the process.

Ultimately, shutting down a TCP connection requires the application processes on both ends of the connection to recognize that “the end is nigh” for the connection and that they should stop sending data. For this reason, connection termination is implemented so that each device terminates its end of the connection separately. The act of closing the connection by one device means that device will no longer send data, but can continue to receive it until the other device has decided to stop sending. This allows all data that is pending to be sent by both sides of the communication to be flushed before the connection is ended.

### **Normal Connection Termination**

In the normal case, each side terminates its end of the connection by sending a special message with the FIN (finish) bit set. The FIN message serves as a connection termination request to the other device, while also possibly carrying data like a regular segment. The device receiving the FIN responds with an acknowledgment to the FIN that indicates that it received the acknowledgment. Neither side considers the connection terminated until they both have sent a FIN and received an ACK, thereby finishing the shutdown procedure.

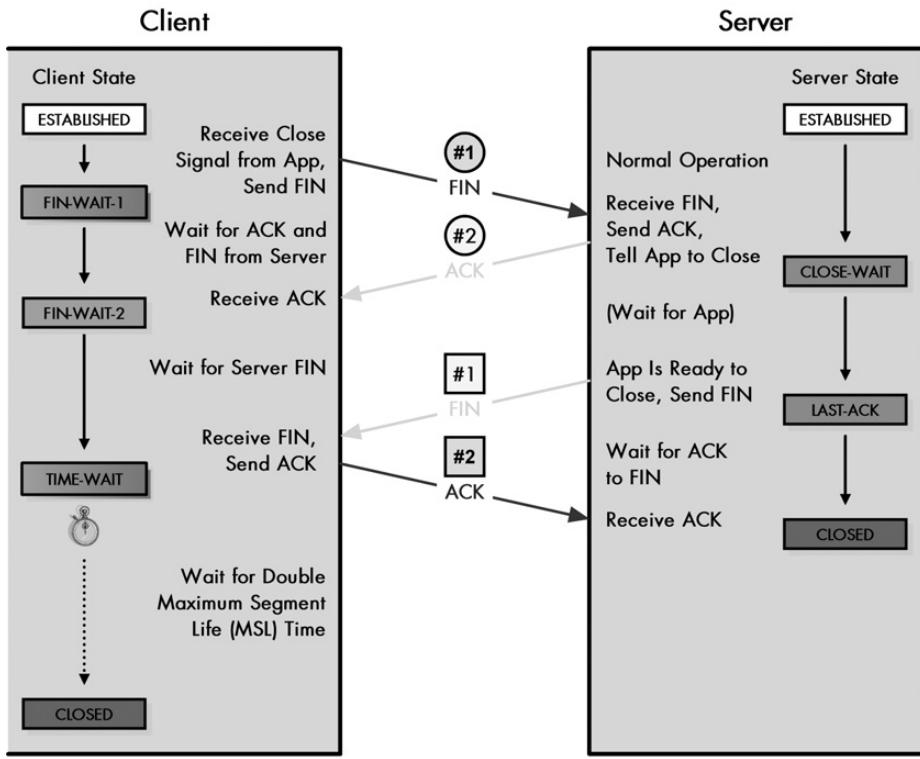
Thus, termination isn’t a three-way handshake as with establishment. It is a pair of two-way handshakes. The states that the two devices in the connection move through during a normal connection shutdown are different because the device initiating the shutdown must behave differently than the one that receives the termination request. In particular, the TCP on the device receiving the initial termination request must inform its application process and wait for a signal that the process is ready to proceed. The initiating device doesn’t need to do this, since the application started the ball rolling in the first place.

**KEY CONCEPT** A TCP connection is terminating using a special procedure by which each side independently closes its end of the link. The connection normally begins with one of the application processes signaling to its TCP layer that the session is no longer needed. That device sends a FIN message to tell the other device that it wants to end the connection, which the other device acknowledges. When the responding device is ready, it too sends a FIN that the other device acknowledges; after waiting a period of time for the device to receive the ACK, the device closes the session.

Table 47-4 describes in detail how the connection termination process works. You can also see the progression of states and messages exchanged in Figure 47-5. The table is adapted from Table 47-1, which describes the TCP FSM, but shows what happens for both the server and the client over time during connection shutdown. Either device can initiate connection termination; in this example, I am assuming the client does it. Each row shows the state each device begins in, what action it takes in that state, and what state it transitions to. I have also shown the send and receive stages of both of the steps for each of the client and server’s close operations.

**Table 47-4:** TCP Connection Termination Procedure

Client			Server		
Start State	Action	Transitions to State	Start State	Action	Transitions to State
ESTABLISHED	Client Close Step 1 Transmit: The application using TCP signals that the connection is no longer needed. The client TCP sends a segment with the FIN bit set to request that the connection be closed.	FIN-WAIT-1	ESTABLISHED	At this stage the server is still in normal operating mode.	—
FIN-WAIT-1	The client, having sent a FIN, is waiting for a device to acknowledge it and for the server to send its own FIN. In this state, the client can still receive data from the server but will no longer accept data from its local application to be sent to the server.	—	ESTABLISHED	Client Close Step 1 Receive and Step 2 Transmit: The server receives the client's FIN. It sends an ACK to acknowledge the FIN. The server must wait for the application using it to be told that the other end is closing, so the application here can finish what it is doing.	CLOSE-WAIT
FIN-WAIT-1	Client Close Step 2 Receive: The client receives the ACK for its FIN. It must now wait for the server to close.	FIN-WAIT-2	CLOSE-WAIT	The server waits for the application process on its end to signal that it is ready to close.	—
FIN-WAIT-2	The client is waiting for the server's FIN.	—	CLOSE-WAIT	Server Close Step 1 Transmit: The server's TCP receives a notice from the local application that it is done. The server sends its FIN to the client.	LAST-ACK
FIN-WAIT-2	Server Close Step 1 Receive and Step 2 Transmit: The client receives the server's FIN and sends back an ACK.	TIME-WAIT	LAST-ACK	The server is waiting for an ACK for the FIN that it sent.	—
TIME-WAIT	The client waits for a period of time equal to double the maximum segment life (MSL) time; this wait ensures that the ACK it sent was received.	—	LAST-ACK	Server Close Step 2 Receive: The server receives the ACK to its FIN and closes the connection.	CLOSED
TIME-WAIT	The timer expires after double the MSL time.	CLOSED	CLOSED	The connection is closed on the server's end.	—
CLOSED	The connection is closed.	—	CLOSED	The connection is closed.	—



**Figure 47-5: TCP connection termination procedure** This diagram shows the conventional termination procedure for a TCP session, with one device initiating termination and the other responding. In this case, the client initiates; it sends a FIN, which the server acknowledges. The server waits for the server process to be ready to close and then sends its FIN, which the client acknowledges. The client waits for a period of time in order to ensure that the device receives its ACK, before proceeding to the CLOSED state.

The device receiving the initial FIN may have to wait a fairly long time (in networking terms) in the CLOSE-WAIT state for the application it is serving to indicate that it is ready to shut down. TCP cannot make any assumptions about how long this will take. During this period of time, the server in the previous example may continue sending data, and the client will receive it. However, the client will not send data to the server.

Eventually, the second device (the server in the example) will send a FIN to close its end of the connection. The device that originally initiated the close (the client) will send an ACK for this FIN. However, the client cannot immediately go to the CLOSED state right after sending that ACK because it must allow time for the ACK to travel to the server. Normally, this will be quick, but delays might slow it down somewhat.

### The TIME-WAIT State

The TIME-WAIT state is required for two main reasons:

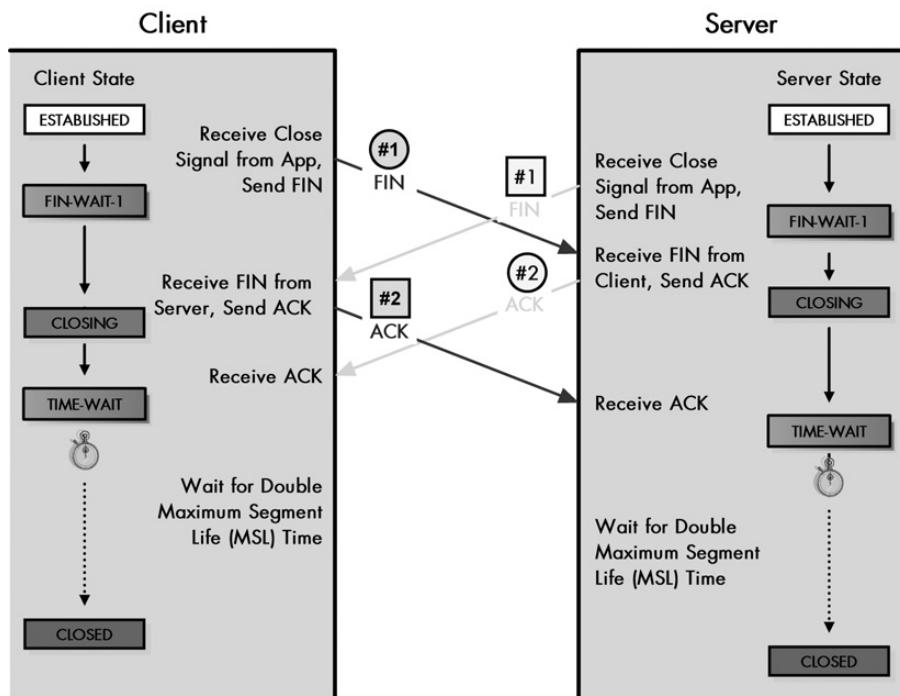
- To provide enough time to ensure that the other device receives the ACK, and to retransmit it if it is lost

- To provide a buffering period between the end of this connection and any subsequent ones. If not for this period, it is possible that packets from different connections could be mixed, thereby creating confusion.

The standard specifies that the client should wait double a particular length of time, called the *maximum segment lifetime (MSL)*, before closing the connection. The TCP standard defines MSL as being a value of 120 seconds (2 minutes). In modern networks, this is an eternity, so TCP allows implementations to choose a lower value if it believes that will lead to better operation.

### **Simultaneous Connection Termination**

Just as it is possible to change the normal connection establishment process if two devices decide to actively open a connection to each other, it is also possible for two devices to try to terminate a connection simultaneously. This term *simultaneously* does not mean that they both decide to shut down at exactly the same time—variances in network delays mean nothing can be simultaneous on an internetwork anyway. It simply means that, in the previous example, the client decides to shut down and sends a FIN, but the server sends its own FIN before the client's FIN shows up at the server. In that case, a different procedure is followed, as described in Table 47-5 and illustrated in Figure 47-6.



**Figure 47-6: TCP simultaneous connection termination procedure** Under certain circumstances, both devices may decide to terminate a connection simultaneously, or nearly simultaneously. In this case, each sends a FIN and, before getting an ACK for it, receives the other device's FIN. Each acknowledges the other's FIN and waits for a period of time before closing the connection. Note the transition through the CLOSING state, which is used only as part of simultaneous termination.

**Table 47-5:** TCP Simultaneous Connection Termination Procedure

Client			Server		
Start State	Action	Transitions to State	Start State	Action	Transitions to State
ESTABLISHED	Client Close Step 1 Transmit: The application using TCP signals that the connection is no longer needed. The TCP on the client sends the next segment with the FIN bit set, indicating a request to close the connection.	FIN-WAIT-1	ESTABLISHED	Server Close Step 1 Transmit: Before the server can receive the FIN that the client sent, the application on the server also signals a close. The server also sends a FIN.	FIN-WAIT-1
FIN-WAIT-1	Server Close Step 1 Receive and Step 2 Transmit: The client has sent a FIN and is waiting for it to be acknowledged. Instead, it receives the FIN that the server sends. It acknowledges the server's close request with an ACK and continues to wait for its own ACK.	CLOSING	FIN-WAIT-1	Client Close Step 1 Receive and Step 2 Transmit: The server has sent a FIN and is waiting for it to be acknowledged. Instead, it receives the FIN that the client sends. It acknowledges the client's close request with an ACK and continues to wait for its own ACK.	CLOSING
CLOSING	Client Close Step 2 Receive: The client receives the ACK for its FIN.	TIME-WAIT	CLOSING	Server Close Step 2 Receive: The server receives the ACK for its FIN.	TIME-WAIT
TIME-WAIT	The client waits for a period of time equal to double the MSL time. This gives enough time to ensure that the ACK it sent to the server was received.	—	TIME-WAIT	The server waits for a period of time equal to double the MSL time. This gives enough time to ensure the ACK it sent to the client was received.	—
TIME-WAIT	The timer expires after double the MSL time.	CLOSED	TIME-WAIT	The timer expires after double the MSL time.	CLOSED
CLOSED	The connection is closed.	—	CLOSED	The connection is closed.	—

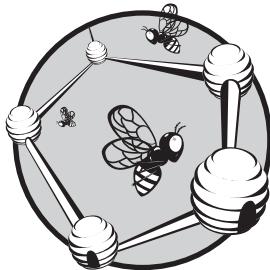
As you can see, the process is much more symmetric in this case, with both devices transitioning through the same states. In either case the end result is the same, with the connection in the CLOSED state—meaning no connection. Each TCP will make sure all outstanding data is sent to the application, sometimes referred to as an implied *push* (see the description of the push function in Chapter 48 for an explanation of this term). The TCBs established for the connection in both devices are destroyed when the connection is closed down.

**KEY CONCEPT** Just as two devices can simultaneously open a TCP session, they can terminate it simultaneously as well. In this case, a different state sequence is followed, with each device responding to the other's FIN with an ACK, then waiting for receipt of its own ACK, and pausing for a period of time to ensure that the other device received its ACK before ending the connection.



# 48

## TCP MESSAGE FORMATTING AND DATA TRANSFER



The previous chapter described how two devices using the Transmission Control Protocol (TCP) establish a TCP connection, as well as how that connection is managed and eventually terminated. While connections are a key part of how TCP works, they are really just a means to the ultimate end of the protocol: sending data. By employing the TCP sliding window mechanism, a special segment format, and several features, TCP devices are able to package and send data over the connection, enabling applications to communicate.

This chapter describes the actual mechanism by which TCP messages are formatted and data is transferred between devices. I begin with a look at the important *TCP segment format*, which describes the fields in each TCP message and how they are used. Next, I provide a description of the method used to calculate the checksum in TCP (as well as UDP) messages, and explain the reason why a special pseudo header is used. Then I discuss the maximum segment size (MSS) parameter and its significance. Following that, I talk about exactly how the sliding window mechanism is used to transfer and

acknowledge data. I conclude with a description of two special data transfer features: the push feature, for immediate data transfer, and the urgent feature for priority data transfer.

**BACKGROUND INFORMATION** *This chapter assumes that you are already familiar with TCP concepts such as sequence numbers, segments, and the basics of the TCP sliding window mechanism. If you are not, read Chapter 46 before proceeding with this one.*

## TCP Message (Segment) Format

In the TCP overview in Chapter 45, I described one of the most interesting jobs that TCP performs: It allows an application to send data as an unstructured sequence of bytes, transparently packaging that data in distinct messages as required by the underlying protocol that TCP uses (normally IP, of course). TCP messages are called *segments*, the name referring to the fact that each is a portion of the overall data stream passing between the devices.

TCP segments are very much jack-of-all-trades messages—they are flexible and serve a variety of purposes. A single field format is used for all segments, with a number of header fields that implement the many functions and features for which TCP is responsible. One of the most notable characteristics of TCP segments is that they are designed to carry both control information and data simultaneously. This reduces the number of segments sent, since a segment can perform more than one function.

For example, there is no need to send separate acknowledgments in TCP, because each TCP message includes a field for an acknowledgment byte number. Similarly, one can request that a connection be closed while sending data in the same message. The nature of each TCP segment is indicated through the use of several special control bits. More than one bit can be sent to allow a segment to perform multiple functions, such as when a bit is used to specify an initial sequence number (ISN) and acknowledge receipt of another such segment at the same time.

The price we pay for this flexibility is that the TCP header is large: 20 bytes for regular segments and more for those carrying options. This is one of the reasons why some protocols prefer to use the User Datagram Protocol (UDP) if they don't need TCP's features. The TCP header fields are used for the following general purposes:

**Process Addressing** The processes on the source and destination devices are identified using port numbers.

**Implementing the Sliding Window System** Sequence Number, Acknowledgment Number, and Window Size fields implement the TCP sliding window system (discussed in the “TCP Sliding Window Data Transfer and Acknowledgment Mechanics” section later in this chapter).

**Setting Control Bits and Fields** These are special bits that implement various control functions and fields that carry pointers and other data needed for them.

**Carrying Data** The Data field carries the actual bytes of data being sent between devices.

**Performing Miscellaneous Functions** These include a checksum for data protection and options for connection setup.

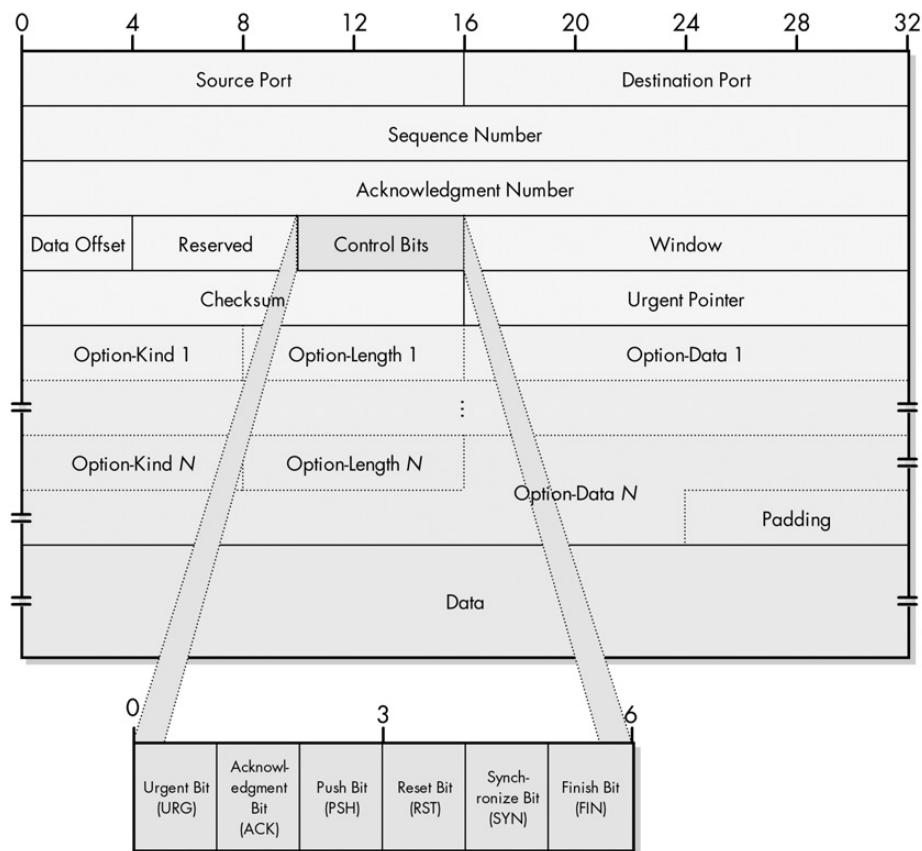
The format for TCP messages (segments) is described fully in Tables 48-1 through 48-3 and illustrated in Figure 48-1.

**Table 48-1:** TCP Segment Format

Field Name	Size (Bytes)	Description
Source Port	2	This is the 16-bit port number of the process that originated the TCP segment on the source device. This will normally be an ephemeral (client) port number for a request sent by a client to a server, or a well-known/registered (server) port number for a reply from a server to a client.
Destination Port	2	This is the 16-bit port number of the process that is the ultimate intended recipient of the message on the destination device. This will usually be a well-known/registered (server) port number for a client request, or an ephemeral (client) port number for a server reply.
Sequence Number	4	For normal transmissions, this is the sequence number of the first byte of data in this segment. In a connection request (SYN) message, this carries the ISN of the source TCP. The first byte of data will be given the next sequence number after the contents of this field, as described in Chapter 47.
Acknowledgment Number	4	When the ACK bit is set, this segment is serving as an acknowledgment (in addition to other possible duties), and this field contains the sequence number the source is next expecting the destination to send. See the “TCP Sliding Window Data Transfer and Acknowledgment Mechanics” section later in this chapter for details.
Data Offset	1/2 (4 bits)	This specifies the number of 32-bit words of data in the TCP header. In other words, this value times four equals the number of bytes in the header, which must always be a multiple of four. It is called a <i>data offset</i> since it indicates by how many 32-bit words the start of the data is offset from the beginning of the TCP segment.
Reserved	3/4 (6 bits)	This field is 6 bits reserved for future use; sent as zero.
Control Bits	3/4 (6 bits)	TCP does not use a separate format for control messages. Instead, certain bits are set to indicate the communication of control information. The 6 bits are described in Table 48-2.
Window	2	This indicates the number of octets of data the sender of this segment is willing to accept from the receiver at one time. This normally corresponds to the current size of the buffer allocated to accept data for this connection. In other words, this field is the current receive window size for the device sending this segment, which is also the send window for the recipient of the segment. See the “TCP Sliding Window Data Transfer and Acknowledgment Mechanics” section later in this chapter for details.
Checksum	2	This is a 16-bit checksum for data integrity protection, computed over the entire TCP datagram, plus a special pseudo header of fields. It is used to protect the entire TCP segment against errors in transmission as well as errors in delivery. Optional alternate checksum methods are also supported.
Urgent Pointer	2	This is used in conjunction with the URG control bit for priority data transfer (see Table 48-2). This field contains the sequence number of the last byte of urgent data. See the “TCP Priority Data Transfer: Urgent Function” section later in this chapter for details.
Options	Variable	TCP includes a generic mechanism for including one or more sets of optional data in a TCP segment. Each of the options can be either one byte in length or variable in length. The first byte is the Option-Kind subfield, and its value specifies the type of option, which in turn indicates whether the option is just a single byte or multiple bytes. Options that are many bytes consist of three fields, which are described in Table 48-3.
Padding	Variable	If the Options field is not a multiple of 32 bits in length, enough zeros are added to pad the header so it is a multiple of 32 bits.
Data	Variable	This is the bytes of data being sent in the segment.

**Table 48-2: TCP Segment Control Bits**

Subfield Name	Size (Bytes)	Description
URG	1/8 (1 bit)	Urgent bit: When set to 1, indicates that the priority data transfer feature has been invoked for this segment, and that the Urgent Pointer field is valid.
ACK	1/8 (1 bit)	Acknowledgment bit: When set to 1, indicates that this segment is carrying an acknowledgment, and the value of the Acknowledgment Number field is valid and carrying the next sequence expected from the destination of this segment.
PSH	1/8 (1 bit)	Push bit: The sender of this segment is using the TCP push feature, requesting that the data in this segment be immediately pushed to the application on the receiving device.
RST	1/8 (1 bit)	Reset bit: The sender has encountered a problem and wants to reset the connection.
SYN	1/8 (1 bit)	Synchronize bit: This segment is a request to synchronize sequence numbers and establish a connection; the Sequence Number field (see Table 48-1) contains the ISN of the sender of the segment.
FIN	1/8 (1 bit)	Finish bit: The sender of the segment is requesting that the connection be closed.



**Figure 48-1: TCP segment format**

**Table 48-3:** TCP Segment Option Subfields

<b>Subfield Name</b>	<b>Size (Bytes)</b>	<b>Description</b>
Option-Kind	1	This specifies the option type.
Option-Length	1	This is the length of the entire option in bytes, including the Option-Kind and Option-Length fields.
Option-Data	Variable	This field contains the option data itself. In at least one oddball case, this field is omitted (making Option-Length equal to 2).

Table 48-4 shows the main options currently defined for TCP

**Table 48-4:** Some TCP Options

<b>Option-Kind</b>	<b>Option-Length</b>	<b>Option-Data</b>	<b>Description</b>
0	—	—	End of Option List: A single-byte option that marks the end of all options included in this segment. This needs to be included only when the end of the options doesn't coincide with the end of the TCP header.
1	—	—	No-Operation: A "spacer" that can be included between options to align a subsequent option on a 32-bit boundary if needed.
2	4	Maximum Segment Size Value	Maximum Segment Size: Conveys the size of the largest segment the sender of the segment wishes to receive. Used only in connection request (SYN) messages.
3	3	Window Size Shift Bits	Window Scale: Implements the optional window scale feature, which allows devices to specify much larger window sizes than would be possible with the normal Window field. The value in Option-Data specifies the power of 2 that the Window field should be multiplied by to get the true window size the sender of the option is using. For example, if the value of Option-Data is 3, this means values in the Window field should be multiplied by 8, assuming both devices agree to use this feature. This allows very large windows to be advertised when needed on high-performance links. See the "TCP Sliding Window Data Transfer and Acknowledgment Mechanics" section later in this chapter for details.
4	2	—	Selective Acknowledgment Permitted: Specifies that this device supports the selective acknowledgment (SACK) feature. This was implemented as a 2-byte option with no Option-Data field, instead of a single-byte option like End of Option List or No-Operation. This was necessary because it was defined after the original TCP specification, so an explicit option length needed to be indicated for backward compatibility.
5	Variable	Blocks of Data Selectively Acknowledged	Selective Acknowledgment: Allows devices supporting the optional selective acknowledgment feature to specify noncontiguous blocks of data that have been received so they are not retransmitted if intervening segments do not show up and need to be retransmitted.
14	3	Alternate Checksum Algorithm	Alternate Checksum Request: Lets a device request that a checksum-generation algorithm other than the standard TCP algorithm be used for this connection. Both devices must agree to the algorithm for it to be used.
15	Variable	Alternate Checksum	Alternate Checksum: If the checksum value needed to implement an alternate checksum is too large to fit in the standard 16-bit Checksum field, it is placed in this option.

The table does not include every TCP option; it just shows the basic ones defined in RFC 793 and a few others that are interesting and correspond to features described elsewhere in this book. Note that most options are sent only in connection request (SYN) segments. This includes the Maximum Segment Size, Window Scale, Selective Acknowledgment Permitted, and Alternate Checksum Request options. In contrast, the Selective Acknowledgment and Alternate Checksum options appear in regular data segments when they are used.

## TCP Checksum Calculation and the TCP Pseudo Header

TCP is designed to provide reliable data transfer between a pair of devices on an IP internetwork. Much of the effort required to ensure reliable delivery of data segments is focused on the problem of ensuring that data is not lost in transit. But there's another important critical impediment to the safe transmission of data: the risk of *errors* being introduced into a TCP segment during its travel across the internetwork.

### Detecting Transmission Errors Using Checksums

If the data gets where it needs to go but is corrupted, and we do not detect the corruption, this is in some ways worse than it never showing up at all. To provide basic protection against errors in transmission, TCP includes a 16-bit Checksum field in its header. The idea behind a checksum is very straightforward: Take a string of data bytes and add them all together, then send this sum with the data stream and have the receiver check the sum. In TCP, the device sending the segment uses a special algorithm to calculate this checksum. The recipient then employs the same algorithm to check the data it received and ensure that there were no errors.

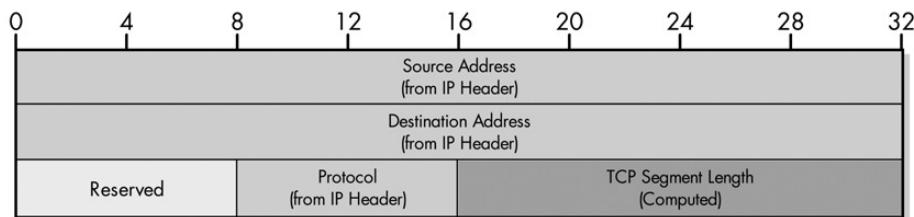
The checksum calculation used by TCP is a bit different than a regular checksum algorithm. A conventional checksum is performed over all the bytes that the checksum is intended to protect, and it can detect most bit errors in any of those fields. The designers of TCP wanted this bit-error protection, but they also wanted protection against other types of problems. To this end, a change was made in how the TCP checksum is computed. This special TCP checksum algorithm was eventually also adopted for use by UDP; see Chapter 44.

### Increasing the Scope of Detected Errors: The TCP Pseudo Header

Instead of computing the checksum over only the actual data fields of the TCP segment, a 12-byte TCP *pseudo header* is created prior to checksum calculation. This header contains important information taken from fields in both the TCP header and the Internet Protocol (IP) datagram into which the TCP segment will be encapsulated (see Chapter 21 for a description of the IP datagram format). The TCP pseudo header has the format described in Table 48-5 and illustrated in Figure 48-2.

**Table 48-5: TCP Pseudo Header for Checksum Calculations**

Field Name	Size (Bytes)	Description
Source Address	4	This is the 32-bit IP address of the originator of the datagram, taken from the IP header.
Destination Address	4	This is the 32-bit IP address of the intended recipient of the datagram, also from the IP header.
Reserved	1	This consists of 8 bits of zeros.
Protocol	1	This is the Protocol field from the IP header. This indicates the higher-layer protocol that is carried in the IP datagram. Of course, we already know that this protocol is TCP. So, this field will normally have the value 6.
TCP Length	2	This is the length of the TCP segment, including both header and data. Note that this is not a specific field in the TCP header; it is computed.

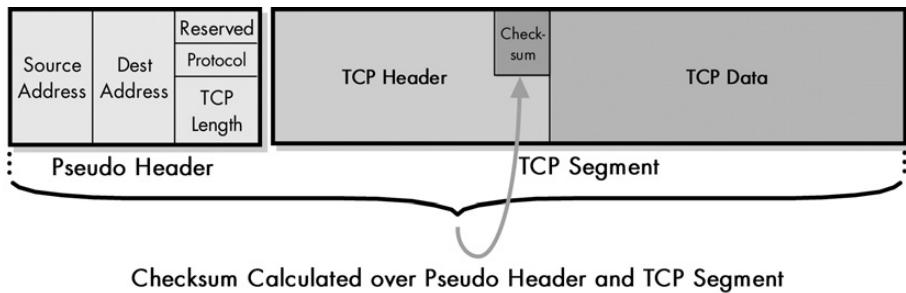


**Figure 48-2: TCP pseudo header for checksum calculation**

Once this 96-bit header has been formed, it is placed in a buffer, followed by the TCP segment itself. Then the checksum is computed over the entire set of data (pseudo header plus TCP segment). The value of the checksum is placed in the Checksum field of the TCP header, and the pseudo header is discarded; it is *not* an actual part of the TCP segment and is not transmitted. This process is illustrated in Figure 48-3.

**NOTE** *The Checksum field is itself part of the TCP header and thus one of the fields over which the checksum is calculated, creating a “chicken-and-egg” situation of sorts. This field is assumed to be all zeros during calculation of the checksum.*

When the TCP segment arrives at its destination, the receiving TCP software performs the same calculation. It forms the pseudo header, prepends it to the actual TCP segment, and then performs the checksum (setting the Checksum field to zero for the calculation as before). If there is a mismatch between its calculation and the value the source device put in the Checksum field, this indicates that an error of some sort occurred, and the segment is normally discarded.



**Figure 48-3: TCP header checksum calculation** To calculate the TCP segment header's Checksum field, the TCP pseudo header is first constructed and placed, logically, before the TCP segment. The checksum is then calculated over both the pseudo header and the TCP segment. The pseudo header is then discarded.

### Advantages of the Pseudo Header Method

So, why bother with this pseudo header? The source and destination devices both compute the checksum using the fields in this pseudo header. This means that if, for any reason, the two devices don't use the same values for the pseudo header, the checksum will fail. When we consider what's in the header, we find that this means the checksum now protects against not just errors in the TCP segment fields, but also against the following problems:

**Incorrect Segment Delivery** If there is a mismatch in the Destination Address between what the source specified and what the destination that received the segment used, the checksum will fail. The same will happen if the Source Address does not match.

**Incorrect Protocol** If a datagram is routed to TCP that actually belongs to a different protocol for whatever reason, this can be immediately detected.

**Incorrect Segment Length** If part of the TCP segment has been omitted by accident, the lengths the source and destination used won't match, and the checksum will fail.

What's clever about the pseudo header is that by using it for the checksum calculation, we can provide this protection without actually needing to send the fields in the pseudo header itself. This eliminates duplicating the IP fields used in the pseudo header within the TCP header, which would be redundant and wasteful of bandwidth. The drawback of the pseudo header method is that it makes checksum calculation take more time and effort (though this is not much of an issue today).

**KEY CONCEPT** TCP checksums are computed over not just the TCP segment, but also over a TCP *pseudo header* that contains the length of the TCP segment as well as the IP Source Address, Destination Address, and Protocol fields. Since these fields are part of the checksum, if the segment is received by the wrong device or has the incorrect Protocol field or segment length, it will be rejected. The technique is clever because the checksum can provide this protection, even though the pseudo header itself is not actually transmitted.

In the context of today's modern, high-speed, highly reliable networks, the use of the pseudo header sometimes seems archaic. How likely is it that a datagram will be delivered to the wrong address? Not very. At the time TCP was created, however, there was significant concern that there might not be proper end-to-end checking of the delivery of datagrams at the IP level. Including IP information in the TCP checksum was seen as a useful additional level of protection.

**NOTE** *There is one interesting implication of the TCP pseudo header: It violates the architectural layering principles that the designers of TCP sought to respect in splitting up TCP and IP. For the checksum, TCP must know IP information that technically it shouldn't know. TCP checksum calculation requires, for example, that the protocol number from the IP header be given to the TCP layer on the receiving device from the IP datagram that carried the segment. The TCP pseudo header is a good example of a case where strict layering was eschewed in favor of practicality.*

TCP also supports an optional method of having two devices agree on an alternative checksum algorithm. This must be negotiated during connection establishment.

## TCP Maximum Segment Size (MSS)

TCP *segments* are the messages that carry data between TCP devices. The Data field is where the actual data being transmitted is carried, and since the length of the Data field in TCP is variable, this raises an interesting question: How much data should we put into each segment? TCP accepts data as a constant stream from the applications that use it, which means that it must decide how many bytes to put into each message that it sends.

A primary determinant of how much data to send in a segment is the current status of the sliding window mechanism on the part of the receiver. When Device A receives a TCP segment from Device B, it examines the value of the Window field to know the limit on how much data Device B is allowing Device A to send in its next segment. (This process is described in the “TCP Sliding Window Data Transfer and Acknowledgment Mechanics” section later in this chapter.) There are also important issues in the selection and adjustment of window size that impact the operation of the TCP system as a whole, which are discussed in Chapter 46.

In addition to the dictates of the current window size, each TCP device also has associated with it a *ceiling* on TCP size—a segment size that will never be exceeded, regardless of how large the current window is. This is called the *maximum segment size (MSS)*. When deciding how much data to put into a segment, each device in the TCP connection will choose the amount based on the current window size, in

conjunction with the various algorithms described in Chapter 46, but it will never be so large that the amount of data exceeds the MSS of the device to which it is sending.

**NOTE** *The name maximum segment size is misleading. The value actually refers to the maximum amount of data that a segment can hold. It does not include the TCP headers. So if the MSS is 100, the actual maximum segment size could be 120 (for a regular TCP header) or larger (if the segment includes TCP options).*

## MSS Selection

The selection of the MSS is based on the need to balance various competing performance and implementation issues in the transmission of data on TCP/IP networks. The main TCP standard, RFC 793, doesn't say much about MSS, so there was potential for confusion about how the parameter should be used. RFC 879 was published a couple of years after the TCP standard to clarify this parameter and the issues surrounding it.

Some issues with the MSS are fairly mundane; for example, certain devices are limited in the amount of space they have for buffers to hold TCP segments, and therefore may wish to limit segment size to a relatively small value. In general, though, the MSS must be chosen by balancing two competing performance issues:

**Overhead Management** The TCP header takes up 20 bytes of data (or more if options are used); the IP header also uses 20 or more bytes. This means that between them, a minimum of 40 bytes is needed for headers, and all of that is nondata overhead. If we set the MSS too low, this results in very inefficient use of bandwidth. For example, if we set it to 40 bytes, a *maximum* of 50 percent of each segment could actually be data; the rest would just be headers. Many segment datagrams would be even worse in terms of efficiency.

**IP Fragmentation** TCP segments will be packaged into IP datagrams. As you saw in Chapter 22, datagrams have their own size limit issues: the matter of the maximum transmission unit (MTU) of an underlying network. If a TCP segment is too large, it will lead to an IP datagram that is too large to be sent without fragmentation. Fragmentation reduces efficiency and increases the chances of part of a TCP segment being lost, resulting in the entire segment needing to be retransmitted.

## TCP Default MSS

The solution to the two competing issues of overhead management and IP fragmentation was to establish a default MSS for TCP that was as large as possible, while avoiding fragmentation for most transmitted segments. This was computed by starting with the minimum MTU for IP networks of 576 bytes. All networks are required to be able to handle an IP datagram of this size without fragmenting. From this number, we subtract 20 bytes for the TCP header and 20 bytes for the IP header, leaving 536 bytes. This is the standard MSS for TCP.

**KEY CONCEPT** TCP is designed to restrict the size of the segments it sends to a certain maximum limit, to reduce the likelihood that segments will need to be fragmented for transmission at the IP level. The TCP *maximum segment size (MSS)* specifies the maximum number of bytes in the TCP segment's Data field, regardless of any other factors that influence segment size. The default MSS for TCP is 536 bytes, which is calculated by starting with the minimum IP MTU of 576 bytes and subtracting 20 bytes each for the IP and TCP headers.

The selection of this MSS value was a compromise of sorts. It means that most TCP segments will be sent unfragmented across an IP internetwork. However, if any TCP or IP options are used, the minimum MTU of 576 bytes will be exceeded, and fragmentation will occur. Still, it makes more sense to allow some segments to be fragmented, rather than use a much smaller MSS to ensure that none are ever fragmented. If we chose, say, an MSS of 400 bytes, we would probably never have fragmentation, but we would lower the data/header ratio from 536:40 (93 percent data) to 400:40 (91 percent data) for all segments.

### ***Nondefault MSS Value Specification***

Naturally, there will be cases where the default MSS is not ideal. TCP provides a means for a device to specify that the MSS it wants to use is either smaller or larger than the default value of 536 bytes. A device can inform the other device of the MSS it wants to use through parameter exchange during the connection establishment process. A device that chooses to do so includes in its SYN message the TCP option called, appropriately, Maximum Segment Size. The other device receives this option and records the MSS for the connection. Each device can specify the MSS it wants for the segments it receives independently.

**NOTE** *The exchange of MSS values during setup is sometimes called MSS negotiation. This is actually a misleading term, because it implies that the two devices must agree on a common MSS value, which is not the case. The MSS value used by each may be different, and there is no negotiation at all.*

Devices may wish to use a larger MSS if they know that the MTUs of the networks the segments will pass over are larger than the IP minimum of 576 bytes. This is most commonly the case when large amounts of data are sent on a local network. The process of MTU path discovery, as described in Chapter 22, is used to determine the appropriate MSS. Devices might use a smaller MSS if they know that TCP segments use a particular optional feature that would consistently increase the size of the IP header, such as when the segments employ IPsec for security (see Chapter 29).

**KEY CONCEPT** Devices can indicate that they wish to use a different MSS value from the default by including a Maximum Segment Size option in the SYN message they use to establish a connection. Each device in the connection may use a different MSS value.

## TCP Sliding Window Data Transfer and Acknowledgment Mechanics

The TCP connection establishment process is employed by a pair of devices to create a TCP connection between them. Once all the setup is done—transmission control blocks (TCBs) have been set up, parameters have been exchanged, and so forth—the devices are ready to get down to the business of transferring data.

The sending of data between TCP devices on a connection is accomplished using the sliding window system we explored in Chapter 46. Here, we will take a more detailed look at exactly how sliding windows are implemented to allow data to be sent and received. For ease of explanation, we'll assume that our connection is between a client and a server—this is easier than the whole “Device A/Device B” business.

### ***Sliding Window Transmit and Receive Categories***

Each of the two devices on a connection must keep track of the data it is sending, as well as the data it is receiving from the other device. This is done by conceptually dividing the bytes into *categories*. For data being transmitted, there are four transmit categories:

**Transmit Category 1** Bytes sent and acknowledged

**Transmit Category 2** Bytes sent but not yet acknowledged

**Transmit Category 3** Bytes not yet sent for which recipient is ready

**Transmit Category 4** Bytes not yet sent for which recipient is not ready

For data being received, there is no need to separate into “received and acknowledged” and “received and unacknowledged,” the way the transmitter separates its first two categories into “sent and acknowledged” and “sent but not yet acknowledged.” The reason is that the transmitter must wait for acknowledgment of each transmission, but the receiver doesn’t need acknowledgment that it received something. Thus, one receive category corresponds to Transmit Categories 1 and 2, while the other two correspond to Transmit Category 3 and Transmit Category 4, respectively, for a total of three receive categories. To help make more clear how the categories relate, I number them as follows:

**Receive Category 1+2** Bytes received and acknowledged. This is the receiver’s complement to Transmit Categories 1 and 2.

**Receive Category 3** Bytes not yet received for which recipient is ready. This is the receiver’s complement to Transmit Category 3.

**Receive Category 4** Bytes not yet received for which recipient is not ready. This is the receiver’s complement to Transmit Category 4.

## **Send (SND) and Receive (RCV) Pointers**

Both the client and server must keep track of both streams being sent over the connection. This is done using a set of special variables called *pointers*, which carve the byte stream into the categories described in the previous section.

The four transmit categories are divided using three send (SND) pointers. Two of the pointers are absolute (refer to a specific sequence number), and one is an offset that is added to one of the absolute pointers, as follows:

**Send Unacknowledged (SND.UNA)** The sequence number of the first byte of data that has been sent but not yet acknowledged. This marks the first byte of Transmit Category 2; all previous sequence numbers refer to bytes in Transmit Category 1.

**Send Next (SND.NXT)** The sequence number of the next byte of data to be sent to the other device (the server, in this case). This marks the first byte of Transmit Category 3.

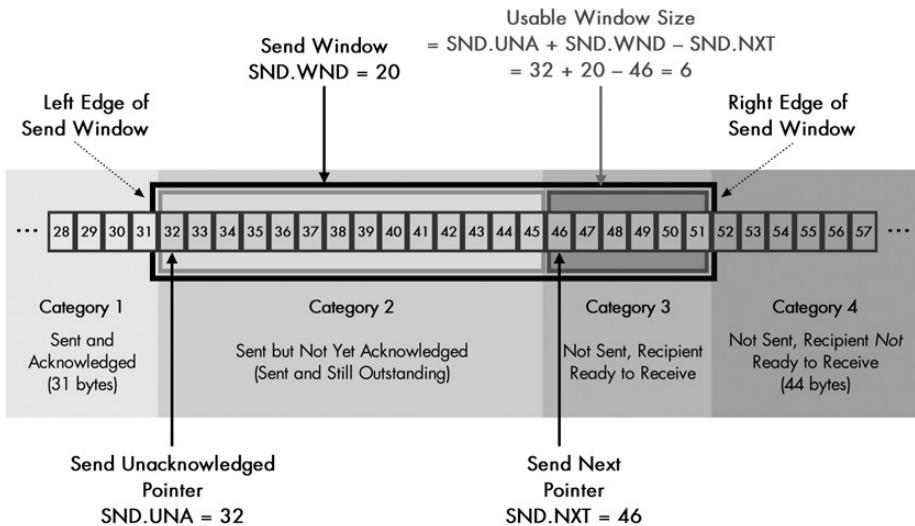
**Send Window (SND.WND)** The size of the send window. Recall that the window specifies the total number of bytes that any device may have outstanding (*unacknowledged*) at any one time. Thus, adding the sequence number of the first unacknowledged byte (SND.UNA) and the send window (SND.WND) marks the first byte of Transmit Category 4.

Another way of looking at these pointers is how they indicate the number of bytes a transmitting device can send at any point in time—that is, the number of bytes in Transmit Category 3. The start of Transmit Category 3 is marked by SND.NXT. The end is marked by the first byte of Transmit Category 4, given by SND.UNA+SND.WND. Thus, the number of bytes in Transmit Category 3 is given by the following formula:

$$\text{SND.UNA} + \text{SND.WND} - \text{SND.NXT}$$

This is called the *usable window*, since it indicates how many bytes the transmitter can use at any point in time. When data is acknowledged, this causes bytes to move from Transmit Category 2 to Transmit Category 1, by increasing the value of SND.UNA. Assuming that the send window size doesn't change, this causes the window to *slide* to the right, permitting more data to be sent. Figure 48-4 illustrates the SND pointers.

**KEY CONCEPT** The TCP sliding windows scheme uses three pointers that keep track of which bytes are in each of the four transmit categories. SND.UNA points to the first unacknowledged byte and indicates the start of Transmit Category 2; SND.NXT points to the next byte of data to be sent and marks the start of Transmit Category 3. SND.WND contains the size of the send window; it is added to SND.NXT to mark the start of Transmit Category 4. Adding SND.WND to SND.UNA and then subtracting SND.NXT yields the current size of the usable transmit window.



**Figure 48-4: TCP transmission categories, send window, and pointers** This diagram is the same as Figure 46-6 (in Chapter 46), but shows the TCP send pointers.  $SND.UNA$  points to the start of Transmit Category 2,  $SND.NXT$  points to the start of Transmit Category 3, and  $SND.WND$  is the size of the send window. The size of the usable window (the hatched rectangle) can be calculated as shown from those three pointers.

The three receive categories are divided using two pointers:

**Receive Next (RCV.NXT)** The sequence number of the next byte of data that is expected from the other device. This marks the first byte in Receive Category 3. All previous sequence numbers refer to bytes already received and acknowledged, in Receive Categories 1 and 2.

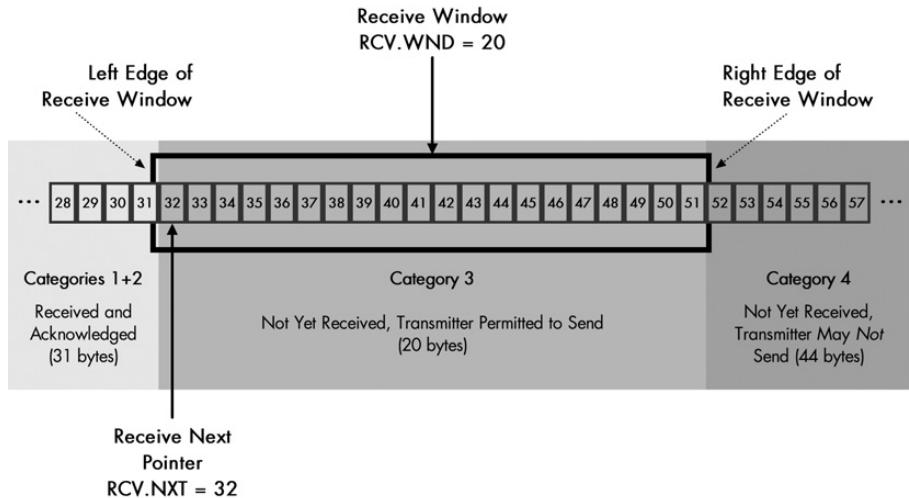
**Receive Window (RCV.WND)** The size of the receive window advertised to the other device. This refers to the number of bytes the device is willing to accept at one time from its peer, which is usually the size of the buffer allocated for receiving data for this connection. When added to the RCV.NXT pointer, this pointer marks the first byte of Receive Category 4.

The receive categories and pointers are illustrated in Figure 48-5.

The SND and RCV pointers are complementary, just as the categories are, with each device managing both the sending of its data and receiving of data from its peer. Assuming we have a client and a server, the relationship between these pointers is as follows:

**Client** The SND pointers keep track of the client's outgoing data stream; the RCV pointers refer to the data coming in from the server. The client's SND categories correspond to the server's RCV categories.

**Server** The SND pointers keep track of the server's outgoing data stream; the RCV pointers refer to the data being received from the client. The server's SND categories correspond to the client's RCV categories.



**Figure 48-5: TCP receive categories and pointers** This diagram is the complement of Figure 48-4, showing how the categories are set up for the receiving device. Categories 1 and 2 have been combined since there is no differentiation between “received and unacknowledged” and “received and acknowledged.” This example shows the state of the receiving device prior to receipt of the 14 bytes that in Figure 48-4 have already been sent.

**KEY CONCEPT** A set of receive (*RCV*) pointers is maintained by each device. These receive pointers are the complement of the send (*SND*) pointers. A device’s send pointers keep track of its outgoing data, and its receive pointers keep track of the incoming data. The two receive pointers are *RCV.NXT*, which indicates the number of the next byte of data expected from the other device, and *RCV.WND*, which is the size of the receive window for that device. The *RCV.WND* of one device equals the *SND.WND* of the other device on the connection.

Since the *SND* and *RCV* values are complementary, the send window of one device is the receive window of the other, and vice versa. Note, however, that the values of the pointers do not always match exactly on the two devices, because at any given time, some bytes may be in transit between the two. Figure 48-5, for example, shows the receive pointers of the recipient *prior* to receiving bytes 32 to 45, which are shown in transit in Figure 48-4.

### TCP Segment Fields Used to Exchange Pointer Information

Both *SND* and *RCV* pointers are maintained in the TCB for the connection held by each device. As data is exchanged, the pointers are updated, and information about the state of the send and receive streams is exchanged using control fields in the TCP segment format. The following are the three most important TCP segment fields used to exchange pointer information:

**Sequence Number** Identifies the sequence number of the first byte of data in the segment being transmitted. This will normally be equal to the value of the *SND.UNA* pointer at the time that data is sent.

**Acknowledgment Number** Acknowledges the receipt of data by specifying the sequence number that the sender of the segment expects in the segment recipient's next transmission. This field will normally be equal to the RCV.NXT pointer of the device that sends it.

**Window** The size of the receive window of the device sending the segment (and thus, the send window of the device receiving the segment).

The Acknowledgment Number field is critical because a device uses this field to tell its peer which segments it has received. The system is *cumulative*. The Acknowledgment Number field says, "I have received all data bytes with sequence numbers less than this value." This means if a client receives many segments of data from a server in rapid succession, it can acknowledge all of them using a single number, as long as they are contiguous. If they are not contiguous, then things get more complicated; see "TCP Noncontiguous Acknowledgment Handling and Selective Acknowledgment (SACK)" in Chapter 49.

**KEY CONCEPT** Three essential fields in the TCP segment format are used to implement the sliding windows system. The Sequence Number field indicates the number of the first byte of data being transmitted. The Acknowledgment Number is used to acknowledge data received by the device sending this segment. The Window field tells the recipient of the segment the size to which it should set its send window.

### An Example of TCP Sliding Window Mechanics

To see how all of this works, let's consider an example of a client and server using a mythical file-retrieval protocol. This protocol specifies that the client sends a request and receives an immediate response from the server. The server then sends the file requested when it is ready.

The two devices will first establish a connection and synchronize sequence numbers. For simplicity, let's say the client uses an ISN of 0, and the server uses an ISN of 240. The server will send the client an ACK with an Acknowledgment Number of 1, indicating it is the sequence number it expects to receive next. Let's say the server's receive window size is set to 360, so this is the client's send window size. The client will send its ACK with an Acknowledgment Number of 241. Let's say its receive window size is 200 (and the server's client window size is thus 200). Let's assume that both devices maintain the same window size throughout the transaction. This won't normally happen, especially if the devices are busy, but the example is complicated enough. Let's also say the MSS is 536 bytes in both directions. This means that the MSS won't affect the size of actual segments in this example (since the MSS is larger than the send window sizes for both devices).

We'll follow a sample transaction to show how the send and receive pointers are created and changed as messages are exchanged between client and server. Table 48-6 describes the process in detail, showing for each step what the send and receive pointers are for both devices. It is rather large, so beware. The transaction is also graphically illustrated in Figures 48-6 and 48-7. Both illustrate the same exchange of messages, using the step numbers of Table 48-6, but from the perspective of one of the devices. Figure 48-6 shows the server's send pointers and client's receive pointers. Figure 48-7 shows the client's send pointers and server's receive pointers. (I would have put them all in one diagram, but they wouldn't fit!)

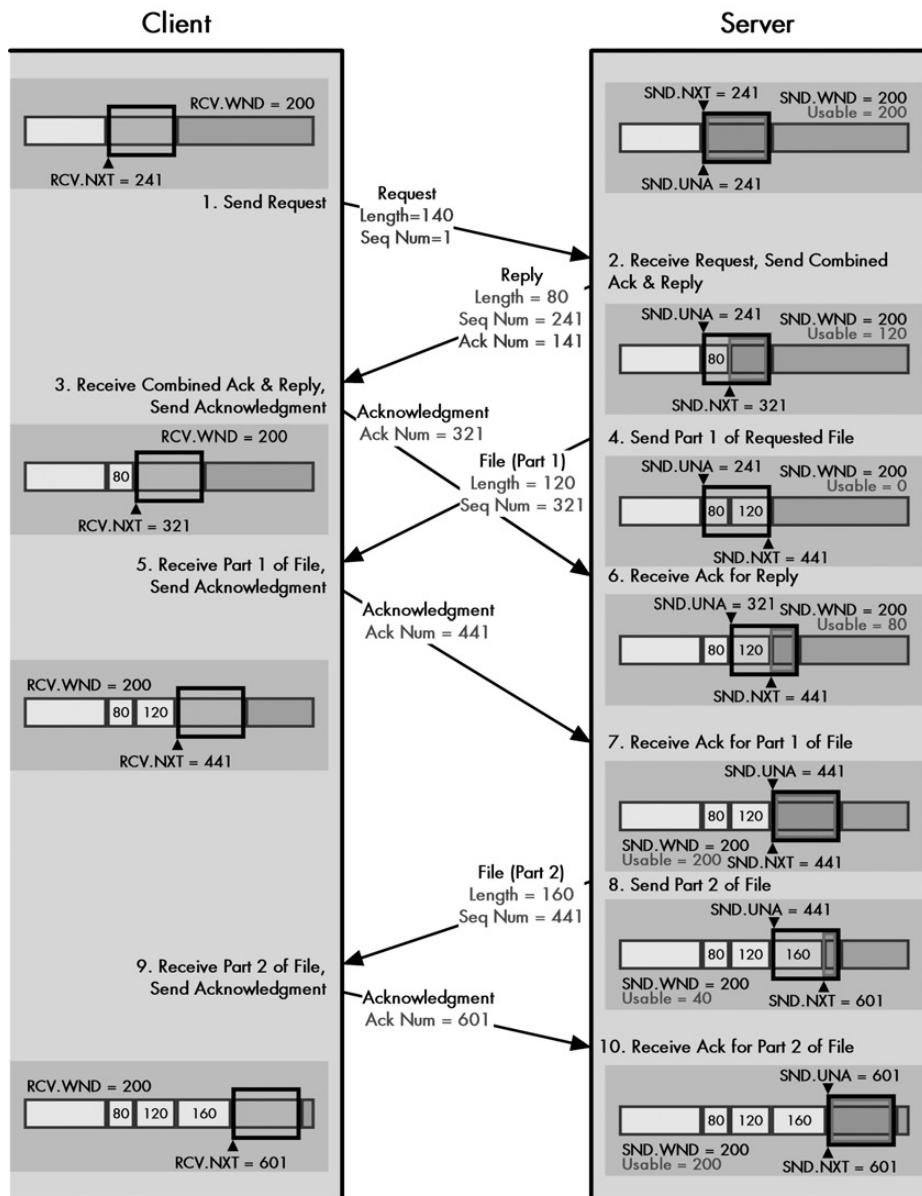
**Table 48-6:** TCP Transaction Example with Send and Receive Pointers

Client						Server					
Process Step	SND. UNA	SND. NXT	SND. WND	RCV. NXT	RCV. WND	Process Step	SND. UNA	SND. NXT	SND. WND	RCV. NXT	RCV. WND
Description						Description					
(setup)	1	1	360	241	200	(setup)	241	241	200	1	360
During connection establishment, the client sets up its pointers based on the parameters exchanged during setup. Notice that the SND.UNA and SND.NXT values are the same. No data has been sent yet, so nothing is unacknowledged. RCV.NXT is the value of the first byte of data expected from the server.						The server sets up its pointers just as the client does. Notice how its values are the complement of the client's.					
1. Send Request	1	141	360	241	200	(wait)	241	241	200	1	360
The client transmits a request to the server. Let's say the request is 140 bytes in length. It will form a segment with a data field of this length and transmit it with the Sequence Number set to 1, the sequence number of the first byte. Once this data has been sent, the client's SND.NXT pointer will be incremented to the value 141 to indicate this is the next data to be sent to the server.						The server does nothing, waiting for a request.					
(wait)	1	141	360	241	200	2. Receive Request, Send Ack & Reply	241	321	200	141	360
At this point, the client hasn't received an acknowledgment for its request. At present, SND.UNA+SND.WND is 361, while SND.NXT is 141. This means the current usable window is 220 bytes. The client could send up to 220 more bytes of data before getting back an acknowledgment. For now, let's say it has nothing more to transmit.						The server receives the 140-byte request from the client. The server sends back an 80-byte response that also acknowledges the client's TCP segment. The Sequence Number field will be 241, the first sequence number of the server's 80 bytes of data. The Acknowledgment Number will be 141, telling the client that is the next sequence number the server expects to hear, and thereby implicitly acknowledging receipt of bytes 1 through 140.  The server increases its RCV.NXT pointer to 141 to reflect the 140 bytes of data received. It increases its SND.NXT pointer by 80.					
3. Receive Ack & Reply, Send Ack	141	141	360	321	200	4. Send Part 1 of File	241	441	200	141	360
The client receives the server's response. It sees the Acknowledgment Number of 141 and knows bytes 1 to 140 were successfully received. It increases its SND.UNA to 141, effectively "sliding the send window" by 140.  The client also accepts the 80 bytes of data the server sent, increasing its RCV.NXT pointer by 80. Assuming it has no more data to send, it sends back a TCP segment that is a pure acknowledgment of the server's response. This segment has no data and an Acknowledgment Number value of 321.						While the client was receiving its response, the server's TCP was supplied with a 280-byte file to be sent to the client. It cannot send all this in one segment, however. The current value of SND.UNA+SND.WND is 441, while SND.NXT is 321. Thus, the server's usable window contains 120 bytes of data. It creates a TCP segment with this much data and a Sequence Number of 321. It increases the SND.NXT pointer to 441. The server has now filled the send window.  Note that the server does not need to wait for an acknowledgment to the reply it sent in step 2. This is a key factor in TCP's ability to ensure high throughput.					

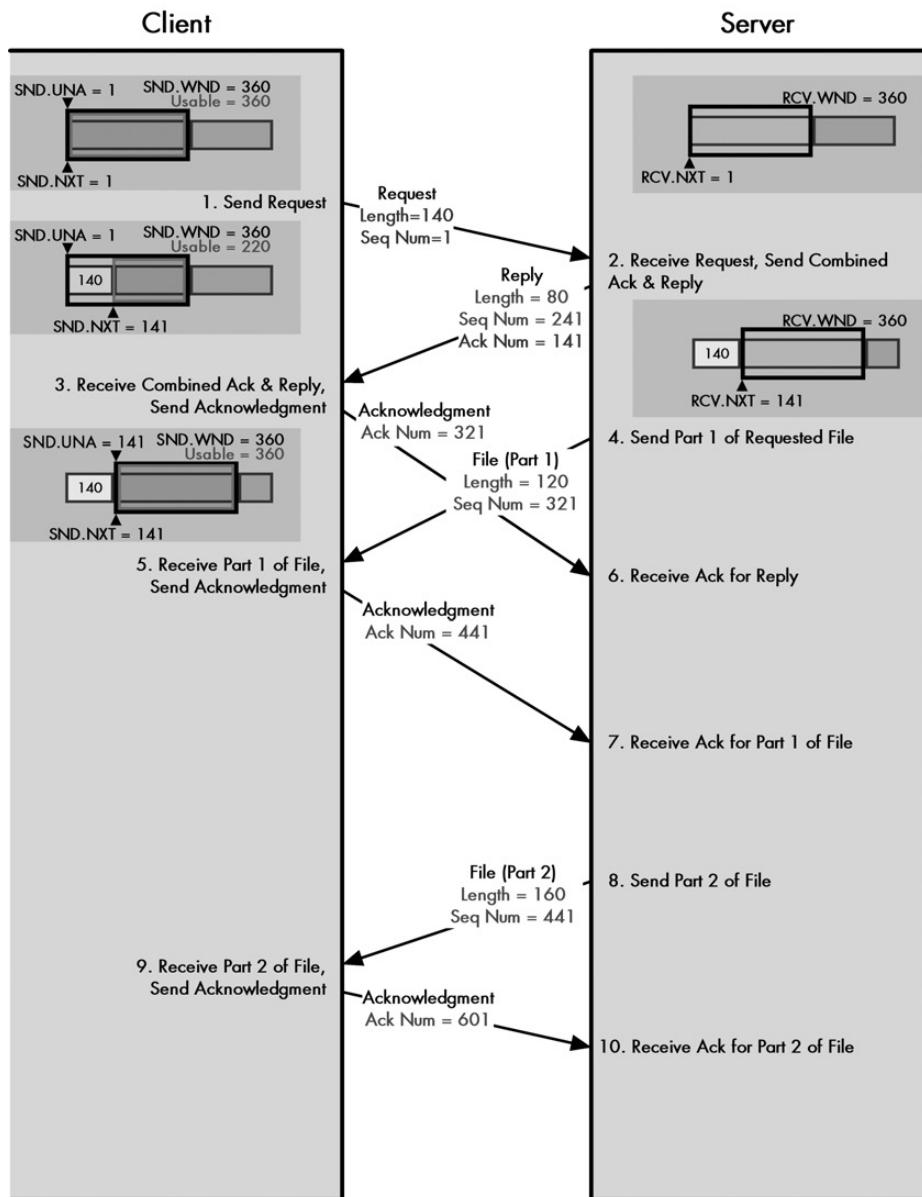
(continued)

**Table 48-6:** TCP Transaction Example with Send and Receive Pointers (continued)

Client						Server					
Process Step	SND. UNA	SND. NXT	SND. WND	RCV. NXT	RCV. WND	Process Step	SND. UNA	SND. NXT	SND. WND	RCV. NXT	RCV. WND
Description						Description					
<b>5. Receive Part 1 of File, Send Ack</b>	141	141	360	441	200	<b>6. Receive Ack for Reply</b>	321	441	200	141	360
The client receives the first 120-byte part of the file the server was sending. It increases the RCV.NXT pointer to 441 and sends an acknowledgment back with an Acknowledgment Number of 441. Again, if it had another request to make of the server, it could include it here, but we'll assume it does not.						The server receives the client's acknowledgment of its earlier 80-byte response (sent in step 2). It increases its SND.UNA to 321. Since it just received acknowledgment of 80 bytes (and the client's window didn't change), the server's usable window is now 80 bytes. However, as we will see in Chapter 49, sending small segments like this can lead to performance issues. Let's say the server has been programmed to not send segments under 100 bytes when it has a lot of data to transmit. It decides to wait.					
(wait)	141	141	360	441	200	<b>7. Receive Ack for Part 1 of File</b>	441	441	200	141	360
The client waits for the rest of the file.						The server receives the acknowledgment for the first part of the file. It increases SND.UNA to 441. This now restores the full 200-byte window.					
(still waiting?)	141	141	360	441	200	<b>8. Send Part 2 of File</b>	441	601	200	141	360
The client continues to wait for the rest of the file.						The server sends the remaining 160 bytes of data in the file in one segment. It increases SND.NXT by 160, and sends the data with a Sequence Number value of 441.					
<b>9. Receive Part 2 of File, Send Ack</b>	141	141	360	601	200	(wait)	441	601	200	141	360
The client receives the rest of the file and acknowledges it. It increases RCV.NXT to 601 and sends back a segment with an Acknowledgment Number of 601.						The server is done for now. It waits for the acknowledgment of the second part of the file.					
(done)	141	141	360	601	200	<b>10. Receive Ack for Part 2 of File</b>	601	601	200	141	360
The client is done with this exchange.						The server receives the second acknowledgment and slides its send window forward by 160 bytes. The transaction is now completed.					



**Figure 48-6: TCP transaction example showing the server's send pointers** The transaction of Table 48-6 from the perspective of the server. See Figure 48-7 for the client's pointers.



**Figure 48-7: TCP transaction example showing client's send pointers** The transaction of Table 48-6 from the perspective of the client. See Figure 48-6 for the server's pointers.

## **Real-World Complications of the Sliding Window Mechanism**

I'm sure the process outlined in the previous section seems rather complicated, but in fact, the example is highly *simplified*, to show you how the basic data transfer mechanism works without too much going on. Scary, isn't it? A real-world connection would include several complications:

**Overlapping Transmissions** I intentionally showed only one request from the client and the response from the server. In reality, the client and server could be pumping many requests and responses at each other in rapid-fire succession. The client would be acknowledging segments received from the server with segments that themselves contained new requests, and so on.

**Acknowledgment of Multiple Segments** I also didn't show a case where two segments are received by a device and acknowledged with a single acknowledgment, although this can certainly happen. Suppose that, in the example, the two parts of the 280-byte file were sent at once and received by the client at the same time. The client would acknowledge both by sending a single segment with an Acknowledgment Number of 601. Remember that this field is a *cumulative* acknowledgment of all segments containing data through the number preceding it, so this would acknowledge all data up to byte 600.

**Fluctuating Window Sizes for Flow Control** The window sizes in the example remained constant, but in a real connection, this will not always be the case. A very busy server may not be able to process and remove data from its buffer as fast as it acknowledges it. It may need to shrink its receive window to reduce the amount of data the client sends it, and then increase the window when more space becomes available. This is how TCP implements flow control, as you will see in the next chapter.

**Lost Transmissions** In a real connection, some transmitted segments will be lost and need to be retransmitted. This is handled by TCP's retransmission scheme (described in Chapter 49).

**Avoiding Small Window Problems** I hinted in the description of the example that we don't necessarily always want to send data as fast as we can, to avoid sending a very small segment. The reason is that this can lead to performance degradation, including a phenomenon called *silly window syndrome*. This will also be explored in the next chapter, where we will see how handling it requires that we change the simple sliding windows scheme we examined so far.

**Congestion Handling and Avoidance** The basic sliding window mechanism has been changed over the years to avoid having TCP connections cause internetwork congestion and to have them handle congestion when it is detected. Congestion issues are discussed, as you may have guessed, in the next chapter.

## TCP Immediate Data Transfer: Push Function

The fact that TCP takes incoming data from a process as an unstructured stream of bytes gives it great flexibility in meeting the needs of most applications. There is no need for an application to create blocks or messages; it just sends the data to TCP when it is ready for transmission. For its part, TCP has no knowledge or interest in the meaning of the bytes of data in this stream. They are just bytes, and TCP sends them without any real concern for their structure or purpose.

This has a couple of interesting effects on how applications work. One is that TCP does not provide any natural indication of the dividing point between pieces of data, such as database records or files. The application must take care of this. Another result of TCP's byte orientation is that TCP cannot decide when to form a segment and send bytes between devices based on the contents of the data. TCP will generally accumulate data sent to it by an application process in a buffer. It chooses when and how to send data based solely on the sliding window system discussed in the previous section, in combination with logic that helps to ensure efficient operation of the protocol.

This means that while an application can control the rate and timing with which it sends data to TCP, it cannot inherently control the timing with which TCP itself sends the data over the internetwork. Now, if we are sending a large file, for example, this isn't a big problem. As long as we keep sending data, TCP will keep forwarding it over the internetwork. It's generally fine in such a case to let TCP fill its internal transmit buffer with data and form a segment to be sent when TCP feels it is appropriate.

However, there are situations where letting TCP accumulate data before transmitting it can cause serious application problems. The classic example of this is an interactive application such as the Telnet protocol (see Chapter 87). When you are using such a program, you want each keystroke to be sent immediately to the other application; you don't want TCP to accumulate hundreds of keystrokes and then send them all at once. The latter may be more efficient, but it makes the application unusable, which is really putting the cart before the horse.

Even with a more mundane protocol that transfers files, there are many situations in which we need to say, "Send the data *now*." For example, many protocols begin with a client sending a request to a server—like the hypothetical one in the preceding example or a request for a web page sent by a web browser. In that case, we want the client's request sent immediately; we don't want to wait until enough requests have been accumulated by TCP to fill an optimal-sized segment.

Naturally, the designers of TCP realized that we needed a way to handle these situations. When an application has data that it needs to have sent across the internetwork immediately, it sends the data to TCP, and then uses the TCP *push* function. This tells the sending TCP to immediately "push" all the data it has to the recipient's TCP as soon as it is able to do so, without waiting for more data.

When this function is invoked, TCP will create a segment (or segments) that contains all the data it has outstanding and then transmit it with the PSH control bit set to 1. The destination device's TCP software, seeing this bit sent, will know that it should not just take the data in the segment it received and buffer it, but rather push it through directly to the application.

**KEY CONCEPT** TCP includes a special *push* function to handle cases where data given to TCP needs to be sent immediately. An application can send data to its TCP software and indicate that it should be pushed. The segment will be sent right away rather than being buffered. The pushed segment's PSH control bit will be set to 1 to tell the receiving TCP that it should immediately pass the data up to the receiving application.

It's important to realize that the push function only forces immediate delivery of data. It does not change the fact that TCP provides no boundaries between data elements. It may seem that an application could send one record of data and then push it to the recipient, then send the second record and push that, and so on. However, the application cannot assume that because it sets the PSH bit for each piece of data it gives to TCP, each piece of data will be in a single segment. It is possible that the first push may contain data given to TCP earlier that wasn't yet transmitted, and it's also possible that two records pushed in this manner may end up in the same segment anyway.

## TCP Priority Data Transfer: Urgent Function

As noted earlier, the fact that TCP treats data to be transmitted as just an unstructured stream of bytes has some important implications on how it used. One aspect of this characteristic is that since TCP doesn't understand the content of the data it sends, it normally treats all the data bytes in a stream as *equals*. The data is sent to TCP in a particular sequence, and it is transmitted in that same order. This makes TCP, in this regard, like those annoying voice mail systems that tell you not to hang up because they will answer calls in the order received.

Of course, while waiting on hold is irritating, this *first-in, first-out* behavior is usually how we want TCP to operate. If we are transmitting a message or a file, we want to be able to give TCP the bytes that compose that file and have TCP transmit that data in the order we gave it. However, just as special circumstances can require the use of the push function described in the previous section, there are cases where we may not want to always send all data in the exact sequence it was given to TCP.

The most common example of this is when it is necessary to interrupt an application's data transfer. Suppose we have an application that sends large files in both directions between two devices. The user of the application realizes that the wrong file is being transferred. When she tells the application to stop the file being sent, she wants this to be communicated to the other end of the TCP connection immediately. She doesn't want the abort command to just be placed at the end of the line after the file she is trying to send!

TCP provides a means for a process to prioritize the sending of data in the form of its *urgent* function. To use it, the process that needs to send urgent data enables the function and sends the urgent data to its TCP layer. TCP then creates a special TCP segment that has the URG control bit set to 1. It also sets the Urgent Pointer field to an offset value that points to the last byte of urgent data in the segment. So, for example, if the segment contained 400 bytes of urgent data followed by 200 bytes of regular data, the URG bit would be set, and the Urgent Pointer field would have a value of 400.

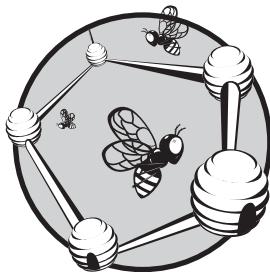
Upon receipt of a segment with the URG flag set to 1, the receiving device looks at the Urgent Pointer and from its value determines which data in the segment is urgent. It then forwards the urgent data to the process with an indication that the data is marked as urgent by the sender. The rest of the data in the segment is processed normally.

**KEY CONCEPT** To deal with situations where a certain part of a data stream needs to be sent with a higher priority than the rest, TCP incorporates an *urgent* function. When critical data needs to be sent, the application signals this to its TCP layer, which transmits it with the URG bit set in the TCP segment, bypassing any lower-priority data that may have already been queued for transmission.

Since we typically want to send urgent data, well, urgently, it makes sense that when such data is given to TCP, the push function is usually also invoked. This ensures that the urgent data is sent as soon as possible by the transmitting TCP and also forwarded up the protocol stack right away by the receiving TCP. Again, we need to remember that this does not guarantee the contents of the urgent segment. Using the push function may mean the segment contains only urgent data with no regular data following, but again, an application cannot assume that this will always be the case.

# 49

## TCP RELIABILITY AND FLOW CONTROL FEATURES



The main task of the Transmission Control Protocol (TCP) is simple: packaging and sending data. Of course, almost every protocol packages and sends data! What distinguishes TCP from these protocols is the sliding window mechanism we explored in the previous chapter, which controls the flow of data between devices. This system not only manages the basic data transfer process, but it also ensures that data is sent reliably and manages the flow of data between devices to transfer data efficiently, without either device sending data faster than the other can receive it.

To enable TCP to provide the features and quality of data transfer that applications require, the protocol needed to be enhanced beyond the simplified data transfer mechanism we saw in preceding chapters. The developers needed to give extra “smarts” to the protocol to handle potential problems and make changes to the basic way that devices send data, to avoid inefficiencies that might otherwise have resulted.

In this chapter, I describe how TCP ensures that devices on a TCP connection communicate in a reliable and efficient manner. I begin with an explanation of the basic method by which TCP detects lost segments and retransmits them. I discuss some of the issues associated with TCP's acknowledgment scheme and an optional feature for improving its efficiency. I then describe the system by which TCP adjusts how long it will wait before deciding that a segment is lost. I discuss how the window size can be adjusted to implement flow control and some of the issues involved in window size management. This includes a look at the infamous “silly window syndrome” problem and special heuristics for addressing issues related to small window size that modify the basic sliding windows scheme. I conclude with a discussion of TCP's mechanisms for handling and avoiding congestion.

**BACKGROUND INFORMATION** *This section assumes that you are already familiar with TCP sequence numbers and segments, and the basics of the TCP sliding window mechanism. It also assumes you have already read the section on TCP message formatting and data transfer. If not, you may want to review at least the section about TCP data transfer mechanics in Chapter 48. Several of the sections in this chapter extend that simplified discussion of TCP data transfer to show what happens in nonideal conditions.*

## TCP Segment Retransmission Timers and the Retransmission Queue

TCP's basic data transfer and acknowledgment mechanism uses a set of variables maintained by each device to implement the sliding window system. These pointers keep track of the bytes of data sent and received by each device, as well as differentiating between acknowledged and unacknowledged transmissions. In the preceding chapter, I described this mechanism and gave a simplified example showing how a client and server use it for basic data transfer.

One of the reasons why that example is simplified is that every segment that was transmitted by the server was received by the client and vice versa. It would be nice if we could always count on this happening, but as we know, in an Internet environment, this is not realistic. Due to any number of conditions—such as hardware failure, corruption of an Internet Protocol (IP) datagram, or router congestion—a TCP segment may be sent but never received. To qualify as a reliable transport protocol, TCP must be able detect lost segments and *retransmit* them.

### Managing Retransmissions Using the Retransmission Queue

The method for detecting lost segments and retransmitting them is conceptually simple. Each time we send a segment, we start a *retransmission timer*. This timer starts at a predetermined value and counts down over time. If the timer expires before an acknowledgment is received for the segment, we retransmit the segment.

TCP uses this basic technique, but implements it in a slightly different way. The reason for this is the need to efficiently deal with many segments that may be unacknowledged at once, to ensure that they are each retransmitted at the appropriate time if needed. The TCP system works according to the following specific sequence.

**Placement on Retransmission Queue, Timer Start** As soon as a segment containing data is transmitted, a copy of the segment is placed in a data structure called the *retransmission queue*. A retransmission timer is started for the segment when it is placed on the queue. Thus, at some point, *every* segment is placed in this queue. The queue is kept sorted by the time remaining in the retransmission timer, so the TCP software can keep track of which timers have the least time remaining before they expire.

**Acknowledgment Processing** If an acknowledgment is received for a segment before its timer expires, the segment is removed from the retransmission queue.

**Retransmission Timeout** If an acknowledgment is *not* received before the timer for a segment expires, a *retransmission timeout* occurs, and the segment is automatically retransmitted.

Of course, we have no more guarantee that a retransmitted segment will be received than we had for the original segment. For this reason, after retransmitting a segment, it remains in the retransmission queue. The retransmission timer is reset, and the countdown begins again. If an acknowledgment is not received for the retransmission, the segment will be retransmitted again and the process repeated.

Certain conditions may cause even repeated retransmissions of a segment to fail. We don't want TCP to just keep retransmitting forever, so TCP will retransmit a lost segment only a certain number of times before concluding that there is a problem and terminating the connection.

**KEY CONCEPT** To provide basic reliability for sent data, each device's TCP implementation uses a *retransmission queue*. Each sent segment is placed in the queue and a *retransmission timer* started for it. When an acknowledgment is received for the data in the segment, it is removed from the retransmission queue. If the timer goes off before an acknowledgment is received, the segment is retransmitted and the timer restarted.

## **Recognizing When a Segment Is Fully Acknowledged**

But how do we know when a segment has been fully acknowledged? Retransmissions are handled on a segment basis, but TCP acknowledgments, as we have seen, are done on a cumulative basis using sequence numbers. Each time a segment is sent by Device A to Device B, Device B looks at the value of the Acknowledgment Number field in the segment. All bytes with sequence numbers lower than the value of this field have been received by Device A. Thus, a segment sent by Device B to Device A is considered acknowledged when all of the bytes that were sent in the segment have a lower sequence number than the latest Acknowledgment Number sent by Device B to Device A. This is determined by calculating the last sequence number of the segment using its first byte number (in the Sequence Number field) and length of the segment's Data field.

**KEY CONCEPT** TCP uses a *cumulative acknowledgment system*. The Acknowledgment Number field in a segment received by a device indicates that all bytes of data with sequence numbers less than that value have been successfully received by the other device. A segment is considered acknowledged when all of its bytes have been acknowledged; in other words, when an Acknowledgment Number containing a value larger than the sequence number of its last byte is received.

Let's use the example illustrated in Figure 49-1 to clarify how acknowledgments and retransmissions work in TCP. Suppose the server in a connection sends out four contiguous segments (numbered starting with 1 for clarity):

**Segment 1** Sequence Number field is 1 and segment length is 80. So the last sequence number in Segment 1 is 80.

**Segment 2** Sequence Number field is 81 and segment length is 120. The last sequence number in Segment 2 is 200.

**Segment 3** Sequence Number field is 201 and segment length is 160. The last sequence number in Segment 3 is 360.

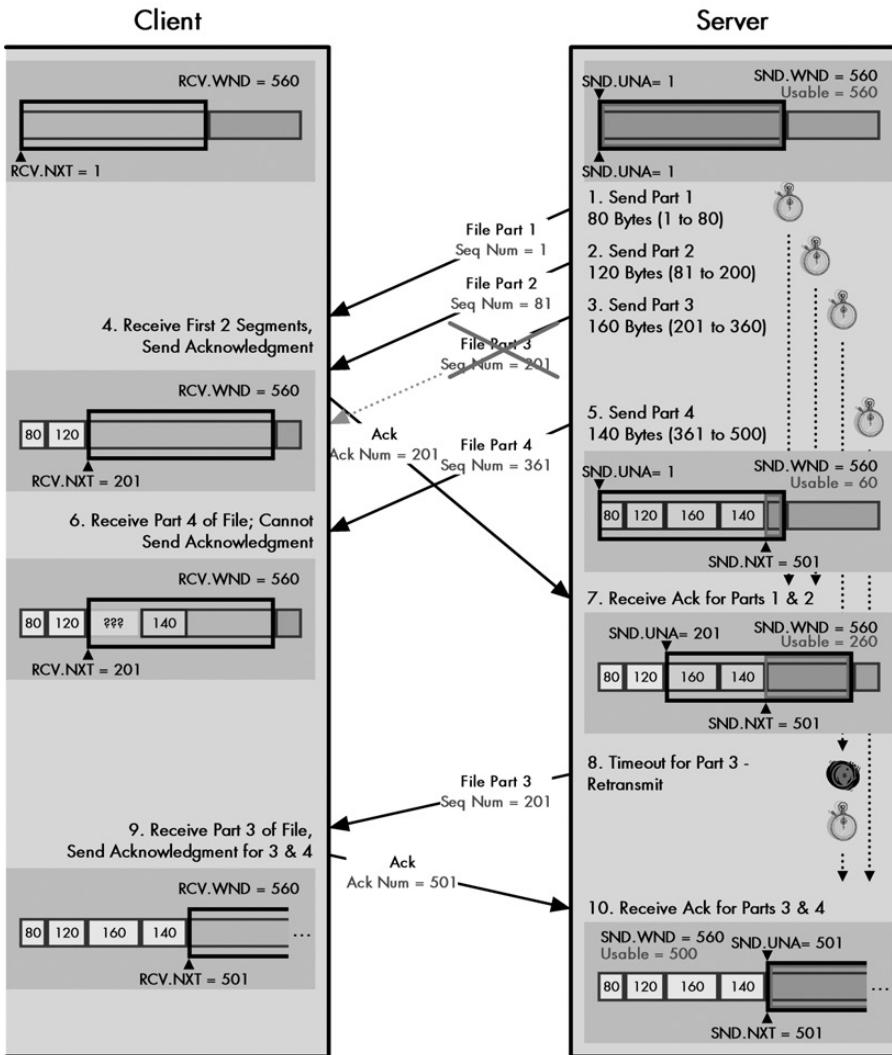
**Segment 4** Sequence Number field is 361 and segment length is 140. The last sequence number in Segment 4 is 500.

Again, these segments can be sent one after the other, without needing to wait for each preceding transmission to be acknowledged. This is a major benefit of TCP's sliding window mechanism.

Now let's say the client receives the first two transmissions. It will send back an acknowledgment with an Acknowledgment Number field value of 201. This tells the server that the first two segments have been successfully received by the client; they will be removed from the retransmission queue (and the server's send window will slide 200 bytes to the right). Segment 3 will remain on the retransmission queue until a segment with an Acknowledgment Number field value of 361 or higher is received; Segment 4 requires an acknowledgment value of 501 or greater.

Now, let's further suppose in this example that Segment 3 gets lost in transit, but Segment 4 is received. The client will store Segment 4 in its receive buffer, but will not be able to acknowledge it, because of TCP's cumulative acknowledgment system—acknowledging Segment 4 would imply receipt of Segment 3 as well, which never showed up. So, the client will need to wait for Segment 3. Eventually, the retransmission timer that the server started for Segment 3 will expire. The server will then retransmit Segment 3. It will be received by the client, which will then be able to acknowledge both Segments 3 and 4 to the server.

There's another important issue here, however: How exactly should the server handle Segment 4? While the client is waiting for the missing Segment 3, the server is receiving no feedback, so it doesn't know that Segment 3 is lost, and it also doesn't know what happened to Segment 4 (or any subsequent transmissions). It is possible that the client has already received Segment 4 but just couldn't acknowledge it. Then again, maybe Segment 4 got lost as well. Some implementations may choose to resend only Segment 3, while some may choose to resend both Segments 3 and 4. This is an important issue that we will discuss next.



**Figure 49-1: TCP transaction example with retransmission** This diagram illustrates a simple transaction and shows the server's send pointers and client's receive pointers. The server sends three segments to the client in rapid succession, setting a retransmission timer for each. Parts 1 and 2 are received, and the client sends an acknowledgment for them. Upon receipt of this ACK, Parts 1 and 2 are taken off the retransmission queue. However, Part 3 is lost in transit. When Part 4 is received, the client cannot acknowledge it; this would imply receipt of the missing Part 3. Eventually, the retransmission timer for Part 3 expires and it is retransmitted, at which time both Part 3 and Part 4 are acknowledged.

A final issue is what value we should use for the retransmission timer when we put a segment on the retransmission queue. If it is set too low, excessive retransmissions occur; if set too high, performance is reduced due to extraneous delays in resending lost segments. In fact, TCP cannot use a single number for this value. It must determine the value dynamically using a process called adaptive retransmission, which we will examine later in the chapter.

## TCP Noncontiguous Acknowledgment Handling and Selective Acknowledgment (SACK)

Computer science people sometimes use the term *elegant* to describe a simple but effective solution to a problem or need. I think the term applies fairly well to the cumulative acknowledgment method that is part of the TCP sliding window system. With a single number, returned in the Acknowledgment Number field of a TCP segment, the device sending the segment can acknowledge not just a single segment it has received from its connection peer, but possibly several of them. We saw how this works in the discussion of the fundamentals of sliding windows in Chapter 46, and again in the previous discussion of retransmissions.

Even the most elegant technique has certain weaknesses, however. In the case of the TCP acknowledgment system, it is the inability to effectively deal with the receipt of *noncontiguous* TCP segments. The Acknowledgment Number specifies that *all* sequence numbers lower than its value have been received by the device sending that number. If we receive bytes with sequence numbers in two noncontiguous ranges, there is no way to specify this with a single number.

This can lead to potentially serious performance problems, especially on internetworks that operate at high speed or over inherently unreliable physical networks. To see what the problem is, let's go back to the example illustrated in Figure 49-1. There, the server sent four segments and received back an acknowledgment with an Acknowledgment Number value of 201. Segment 1 and Segment 2 were thus considered acknowledged. They would be removed from the retransmission queue, and this would also allow the server's send window to slide 80+120 bytes to the right, allowing 200 more bytes of data to be sent.

However, let's again imagine that Segment 3, starting with sequence number 201, is somehow lost in transit. Since the client never receives this segment, it can never send back an acknowledgment with an Acknowledgment Number higher than 201. This causes the sliding window system to get stuck. The server can continue to send additional segments until it fills up the client's receive window, but until the client sends another acknowledgment, the server's send window will not slide.

The other problem we saw is that if Segment 3 gets lost, the client has no way to tell the server that it has received any *subsequent* segments. It's entirely possible that the client has received the server's Segment 4 and later segments, until the window filled up. But the client can't send an acknowledgment with a value of 501 to indicate receipt of Segment 4, *because this implies receipt of Segment 3 as well*.

**NOTE** *In some cases, the client may still send an acknowledgment upon receipt of Segment 4, but containing only a repeated acknowledgment of the bytes up to the end of Segment 2. See the coverage of congestion avoidance later in this chapter for an explanation.*

And here we see the drawback of the single-number, cumulative acknowledgment system of TCP. We could imagine a worst-case scenario, in which the server is told it has a window of 10,000 bytes, and sends 20 segments of 500 bytes each. The first segment is lost, and the other 19 are received. But since it is the first segment that never showed up, none of the other 19 segments can be acknowledged!

**KEY CONCEPT** TCP's acknowledgment system is *cumulative*. This means that if a segment is lost in transit, no subsequent segments can be acknowledged until the missing one is retransmitted and successfully received.

## **Policies for Dealing with Outstanding Unacknowledged Segments**

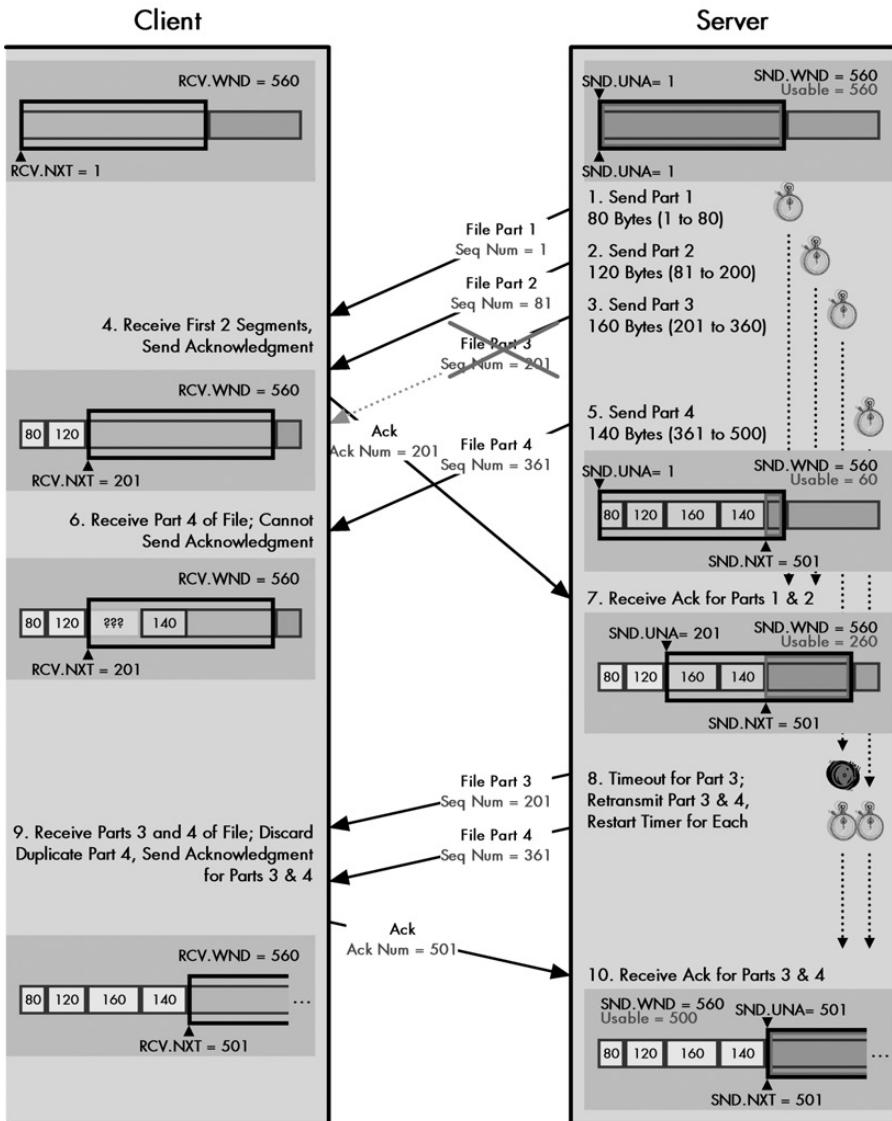
How do we handle retransmissions when there are subsequent segments outstanding beyond the lost segment? In our example, when the server experiences a retransmission timeout on Segment 3, it must decide what to do about Segment 4, when it simply doesn't know whether or not the client received it. In our worst-case scenario, we have 19 segments that may or may not have shown up at the client after the first one that was lost.

We have two possible ways to handle this situation:

**Retransmit Only Timed-Out Segments** This is the more conservative, or if you prefer, optimistic approach. We retransmit only the segment that timed out, hoping that the other segments beyond it were successfully received. This method is best if the segments after the timed-out segment actually showed up. It doesn't work so well if they did not. In the latter case, each segment would need to time out individually and be retransmitted. Imagine that in our worst-case scenario, all twenty 500-byte segments were lost. We would need to wait for Segment 1 to time out and be retransmitted. This retransmission would be acknowledged (we hope), but then we would get stuck waiting for Segment 2 to time out and be resent. We would need to do this many times.

**Retransmit All Outstanding Segments** This is the more aggressive, or pessimistic, method. Whenever a segment times out, we resend not only that segment, but all other segments that are still unacknowledged. This method ensures that any time there is a holdup with acknowledgments, we refresh all outstanding segments to give the other device an extra chance at receiving them, in case they, too, were lost. In the case where all 20 segments were lost, this saves substantial amounts of time over the alternative, optimistic approach. The problem here is that these retransmissions may not be necessary. If the first of 20 segments was lost and the other 19 were actually received, we would be resending 9500 bytes of data (plus headers) for no reason.

Since TCP doesn't know whether these other segments showed up, it cannot know which method is better. It must simply make an executive decision to use one approach or the other and hope for the best. In the example shown in Figure 49-1, I demonstrated the conservative, optimistic approach: Only the lost segment of the file was retransmitted. Figure 49-2 illustrates the alternative aggressive, pessimistic approach to retransmission.



**Figure 49-2: TCP aggressive retransmission example** This example is the same as that shown in Figure 49-1, except that here the server is taking an “aggressive” approach to retransmitting lost segments. When Segment 3 times out, both Segments 3 and 4 are retransmitted, and their retransmission timers restarted. (In this case, Segment 4 already arrived, so this extra transmission was not useful.)

**KEY CONCEPT** There are two approaches to handling retransmission in TCP. In the more conservative approach, only the segments whose timers expire are retransmitted. This saves bandwidth, but it may cause performance degradation if many segments in a row are lost. The alternative is that when a segment’s retransmission timer expires, both it and all subsequent unacknowledged segments are retransmitted. This provides better performance if many segments are lost, but it may waste bandwidth on unnecessary retransmissions.

This lack of knowledge about noncontiguous segments is the core of the problem. The solution is to extend the basic TCP sliding window algorithm with an optional feature that allows a device to acknowledge noncontiguous segments individually. This feature, introduced in RFC 1072 and refined in RFC 2018, is called TCP *selective acknowledgment*, abbreviated SACK.

## A Better Solution: Selective Acknowledgment (SACK)

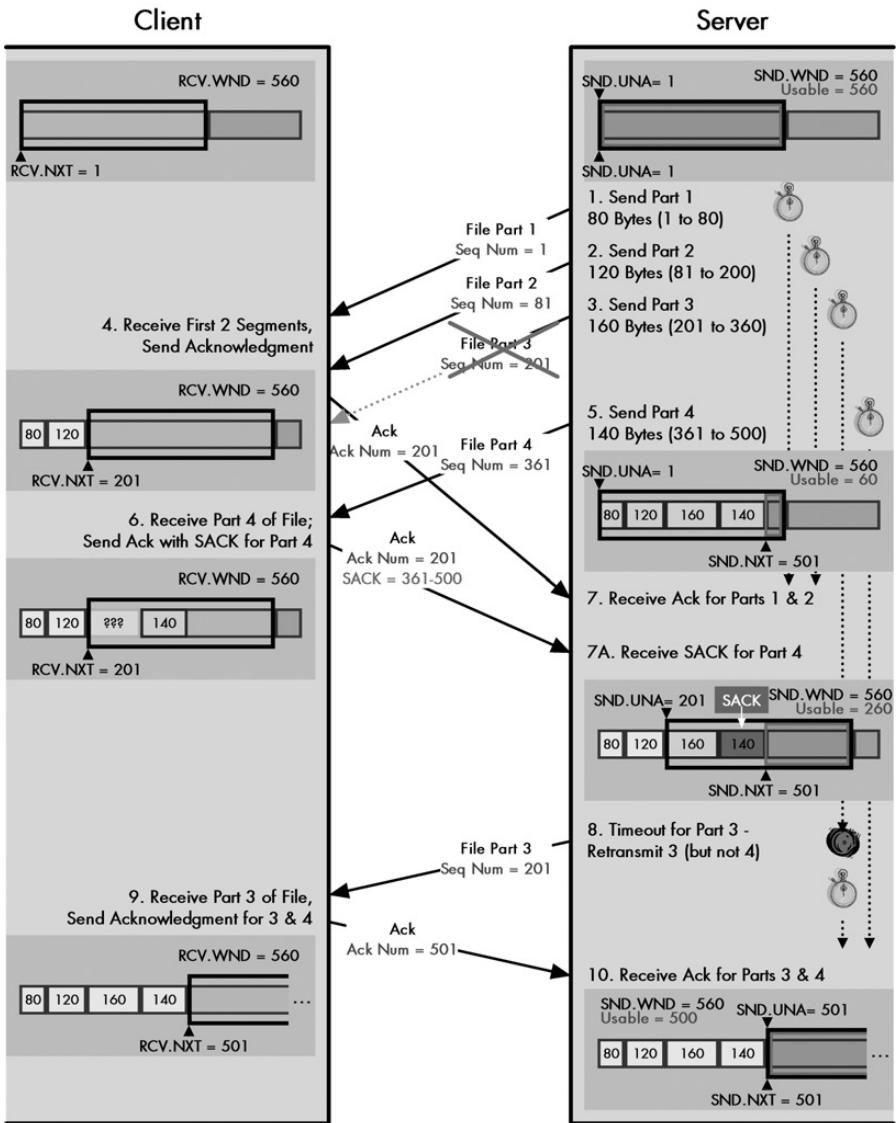
To use SACK, the two devices on the connection must both support the feature, and must enable it by negotiating the Selective Acknowledge Permitted (SACK-Permitted) option in the SYN segment they use to establish the connection. Assuming this is done, either device is then permitted to include in a regular TCP segment a Selective Acknowledgment (SACK) option. This option contains a list of sequence number ranges of segments of data that have been received but have not been acknowledged, since they are noncontiguous.

Each device modifies its retransmission queue so that each segment includes a flag that is set to 1 if the segment has been selectively acknowledged—the SACK bit. The device then uses a modified version of the aggressive method illustrated in Figure 49-2, where upon retransmission of a segment, all later segments are also retransmitted *unless* their SACK bits are set to 1.

**KEY CONCEPT** The optional TCP *selective acknowledgment* feature provides a more elegant way of handling subsequent segments when a retransmission timer expires. When a device receives a noncontiguous segment, it includes a special *Selective Acknowledgment (SACK)* option in its regular acknowledgment that identifies noncontiguous segments that have already been received, even if they are not yet acknowledged. This saves the original sender from needing to retransmit them.

For example, in our four-segment case, if the client receives Segment 4 but not Segment 3, when it sends back a segment with an Acknowledgment Number field value of 201 (for Segments 1 and 2), it can include a SACK option that specifies, “I have received bytes 361 through 500, but they are not yet acknowledged.” This can also be done in a second acknowledgment segment if Segment 4 arrives well after Segments 1 and 2. The server recognizes this as the range of bytes for Segment 4, and turns on the SACK bit for Segment 4. When Segment 3 is retransmitted, the server sees that the SACK bit for Segment 4 is on and does not retransmit it. This is illustrated in Figure 49-3.

After Segment 3 is retransmitted, the SACK bit for Segment 4 is cleared. This is done for robustness, to handle cases where, for whatever reason, the client changes its mind about having received Segment 4. The client *should* send an acknowledgment with an Acknowledgment Number of 501 or higher, officially indicating receipt of Segments 3 and 4. If this does not happen, the server must receive another selective acknowledgment for Segment 4 to turn its SACK bit back on. Otherwise, it will be automatically resent when its timer expires or when Segment 3 is retransmitted.



**Figure 49-3: TCP retransmission with selective acknowledgement (SACK)** This is the example from Figures 49-1 and 49-2, changed to use the optional selective acknowledgement feature. After receiving Parts 1, 2, and 4 of the file, the client sends an acknowledgment for 1 and 2 that includes a SACK for Part 4. This tells the server not to resend Part 4 when Part 3's timer expires.

## TCP Adaptive Retransmission and Retransmission Timer Calculations

Whenever a TCP segment is transmitted, a copy of it is also placed on the retransmission queue. When the segment is placed on the queue, a retransmission timer is started for the segment, which starts from a particular value and counts down to zero. This timer controls how long a segment can remain unacknowledged before the sender gives up, concludes that the segment is lost, and sends it again.

The length of time we use for retransmission timer is thus very important. If it is set too low, we might start retransmitting a segment that was actually received, because we didn't wait long enough for the acknowledgment of that segment to arrive. Conversely, if we set the timer too long, we waste time waiting for an acknowledgment that will never arrive, reducing overall performance.

Ideally, we would like to set the retransmission timer to a value just slightly larger than the *round-trip time (RTT)* between the two TCP devices; that is, the typical time it takes to send a segment from a client to a server and the server to send an acknowledgment back to the client (or the other way around, of course). The problem is that there *is* no such typical RTT. There are two main reasons for this:

**Differences in Connection Distance** Suppose you are at work in the United States, and during your lunch hour, you transfer a large file between your workstation and a local server connection using 100 Mbps Fast Ethernet. At the same time, you are downloading a picture of your nephew from your sister's personal website, which is connected to the Internet using an analog modem to an ISP in a small town near Lima, Peru. Would you want both of these TCP connections to use the same retransmission timer value? I certainly hope not!

**Transient Delays and Variability** The amount of time it takes to send data between any two devices will vary over time due to various happenings on the internetwork: fluctuations in traffic, router loads, and so on. To see an example of this for yourself, try typing `ping www.tcpipguide.com` from the command line of an Internet-connected PC, and you'll see how the reported times can vary.

It is for these reasons that TCP does not attempt to use a static, single number for its retransmission timers. Instead, TCP uses a dynamic, or *adaptive* retransmission scheme.

### Adaptive Retransmission Based on RTT Calculations

TCP attempts to determine the approximate RTT between the devices and adjusts it over time to compensate for increases or decreases in the average delay. The practical issues of how this is done are important, but they are not covered in much detail in the main TCP standard. However, RFC 2988, “Computing TCP’s Retransmission Timer,” discusses the issue extensively.

RTTs can bounce up and down, so we want to aim for an *average* RTT value for the connection. This average should respond to consistent movement up or down in the RTT, without overreacting to a few very slow or fast acknowledgments. To allow this to happen, the RTT calculation uses a *smoothing* formula:

$$\text{New RTT} = (\alpha * \text{Old RTT}) + ((1-\alpha) * \text{Newest RTT Measurement})$$

where  $\alpha$  (alpha) is a *smoothing factor* between 0 and 1. Higher values of  $\alpha$  (closer to 1) provide better smoothing and avoiding sudden changes as a result of one very fast or very slow RTT measurement. Conversely, this also slows down how quickly TCP reacts to more sustained changes in RTT. Lower values of alpha (closer to 0) make the RTT change more quickly in reaction to changes in measured RTT, but can cause overreaction when RTTs fluctuate wildly.

## Acknowledgment Ambiguity

Measuring the RTT between two devices is simple in concept: Note the time that a segment is sent, note the time that an acknowledgment is received, and subtract the two. The measurement is more tricky in actual implementation, however.

One of the main potential “gotchas” occurs when a segment is assumed lost and is retransmitted. The retransmitted segment carries nothing that distinguishes it from the original. When an acknowledgment is received for this segment, it’s unclear whether this corresponds to the retransmission or the original segment. Even though we decided the segment was lost and retransmitted it, it’s possible the segment eventually got there, after taking a long time, or that the segment got there quickly but the *acknowledgment* took a long time!

This is called *acknowledgment ambiguity*, and it is not trivial to resolve. We can’t just decide to assume that an acknowledgment always goes with the oldest copy of the segment sent, because this makes the RTT appear too high. We also don’t want to just assume an acknowledgment always goes with the latest sending of the segment, as that may artificially lower the average RTT.

## Refinements to RTT Calculation and Karn’s Algorithm

TCP’s solution is based on the use of a technique called *Karn’s algorithm*, after its inventor, Phil Karn. The main change this algorithm makes is the separation of the calculation of average RTT from the calculation of the value to use for timers on retransmitted segments.

The first change made under Karn’s algorithm is to not use measured RTT for any segments that are retransmitted in the calculation of the overall average RTT for the connection. This completely eliminates the problem of acknowledgment ambiguity.

However, this by itself, would not allow increased delays due to retransmissions to affect the average RTT. For this, we need the second change: incorporation of a *timer backoff* scheme for retransmitted segments. We start by setting the retransmission timer for each newly transmitted segment based on the current average RTT. When a segment is retransmitted, the timer is not reset to the same value it was set for the initial transmission. It is “backed off,” or increased, using a multiplier (typically 2) to give the retransmission more time to be received. The timer

continues to be increased until a retransmission is successful, up to a certain maximum value. This prevents retransmissions from being sent too quickly and further adding to network congestion.

Once the retransmission succeeds, the RTT is kept at the longer (backed-off) value until a valid RTT can be measured on a segment that is sent and acknowledged without retransmission. This permits a device to respond with longer timers to occasional circumstances that cause delays to persist for a period of time on a connection, while eventually having the RTT settle back to a long-term average when normal conditions resume.

**KEY CONCEPT** TCP uses an *adaptive* retransmission scheme that automatically adjusts the amount of time to which retransmission timers are set, based on the average amount of time it takes to send segments between devices. This helps avoid retransmitting potentially lost segments too quickly or too slowly.

## TCP Window Size Adjustment and Flow Control

We have seen the importance of the concept of *window size* to TCP's sliding window mechanism. In a connection between a client and a server, the client tells the server the number of bytes it is willing to receive at one time from the server; this is the client's *receive window*, which becomes the server's *send window*. Likewise, the server tells the client how many bytes of data it is willing to take from the client at one time; this is the server's *receive window* and the client's *send window*.

The use of these windows is demonstrated in Chapter 48, where we discussed TCP's basic data transfer and acknowledgment mechanism. However, just as the example in that chapter was simplified because I didn't show what happens with lost segments, there's another way that it doesn't reflect the real-world conditions of an actual Internet: the send and receive window sizes never changed during the course of communication.

To understand why the window size may fluctuate, we need to understand what it represents. The simplest way of considering the window size is that it indicates the size of the device's receive buffer for the particular connection. That is, window size represents how much data a device can handle from its peer at one time before it is passed to the application process. Let's consider the example in Chapter 48. I said that the server's window size was 360. This means the server is willing to take no more than 360 bytes at a time from the client.

When the server receives data from the client, it places it into this buffer. The server must then do two distinct things with this data:

**Acknowledgment** The server must send an acknowledgment back to the client to indicate that the data was received.

**Transfer** The server must process the data, transferring it to the destination application process.

It is critically important that we differentiate between these two activities. Unfortunately, the TCP standards don't do a great job in this regard, which makes them very difficult to understand. The key point is that in the basic sliding window

system, data is acknowledged when received, but *not necessarily* immediately transferred out of the buffer. This means that it is possible for the buffer to fill up with received data faster than the receiving TCP can empty it. When this occurs, the receiving device may need to adjust the window size to prevent the buffer from being overloaded.

Since the window size can be used in this manner to manage the rate at which data flows between the devices at the ends of the connection, it is the method by which TCP implements *flow control*, one of the classic jobs of the transport layer. Flow control is vitally important to TCP, as it is the method by which devices communicate their status to each other. By reducing or increasing window size, the server and client each ensure that the other device sends data just as fast as the recipient can deal with it.

### ***Reducing Send Window Size to Reduce the Rate Data Is Sent***

To understand window size adjustment, let's go back to our earlier example in Chapter 48, but with a few changes. First, to keep things simple, let's just look at the transmissions made from the client to the server, not the server's replies (other than acknowledgments)—this is illustrated in Figure 48-7. As before, the client sends 140 bytes to the server. After sending the 140 bytes, the client has 220 bytes remaining in its usable window: 360 bytes in the send window less the 140 bytes it just sent.

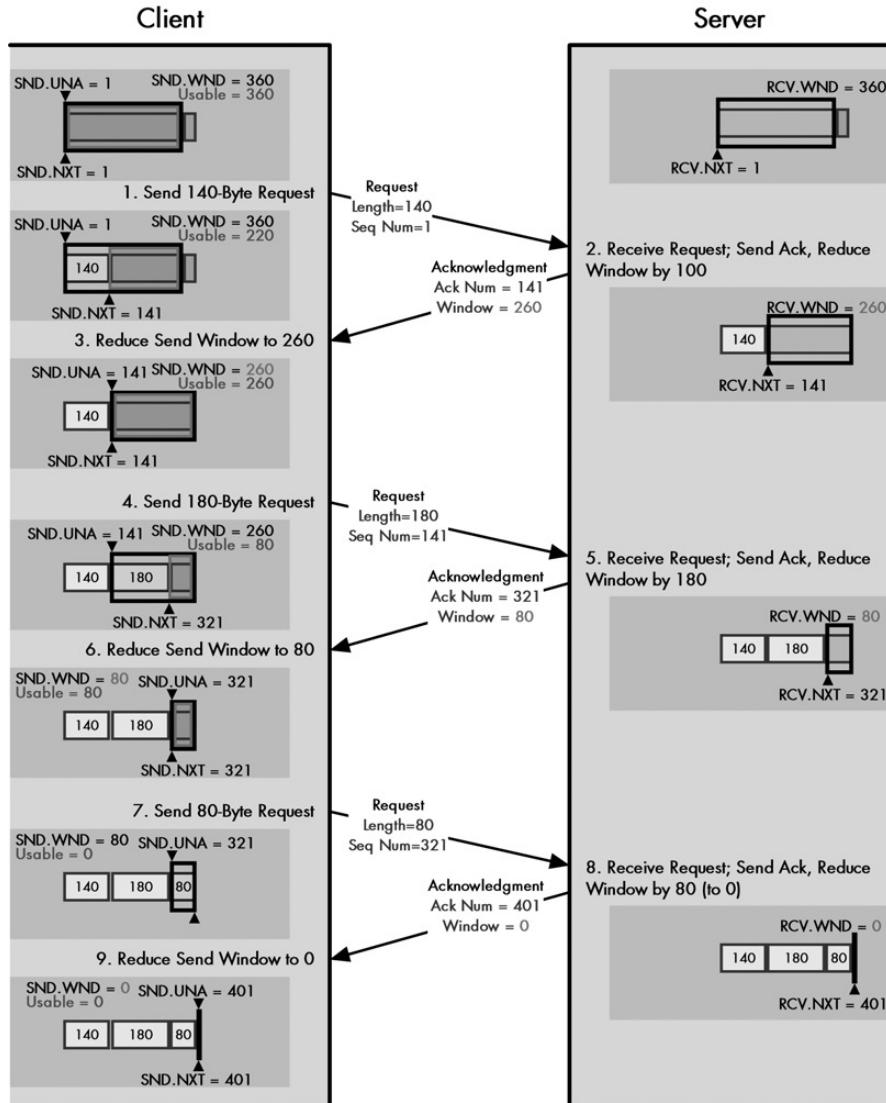
Sometime later, the server receives the 140 bytes and puts them in the buffer. Now, in an ideal world, the 140 bytes go into the buffer, and they are acknowledged and immediately removed from the buffer. Another way of thinking of this is that the buffer is of infinite size and can hold as much as the client can send. The buffer's free space remains 360 bytes in size, so the same window size can be advertised back to the client. This was the simplification in the previous example.

As long as the server can process the data as fast as it comes in, it will keep the window size at 360 bytes. The client, upon receipt of the acknowledgment of 140 bytes and the same window size it had before, slides the full 360-byte window 140 bytes to the right. Since there are now 0 unacknowledged bytes, the client can now once again send 360 bytes of data. These correspond to the 220 bytes that were formerly in the usable window, plus 140 new bytes for the ones that were just acknowledged.

In the real world, however, that server might be dealing with dozens, hundreds, or even thousands of TCP connections. TCP might not be able to process the data immediately. Alternatively, it is possible the application itself might not be ready for the 140 bytes for whatever reason. In either case, the server's TCP may not be able to immediately remove all 140 bytes from the buffer. If so, upon sending an acknowledgment back to the client, the server will want to change the window size that it advertises to the client, to reflect the fact that the buffer is partially filled.

Suppose that we receive 140 bytes, but are able to send only 40 bytes to the application, leaving 100 bytes in the buffer. When we send back the acknowledgment for the 140 bytes, the server can reduce its send window by 100 bytes, to 260 bytes. When the client receives this segment from the server, it will see the acknowledgment of the 140 bytes sent and slide its window 140 bytes to the right. However, as it slides this window, it reduces its size to only 260 bytes. We can consider this as

sliding the *left edge* of the window 140 bytes, but the *right edge* only 40 bytes. The new, smaller window ensures that the server receives a maximum of 260 bytes from the client, which will fit in the 260 bytes remaining in its receive buffer. This is illustrated in the first exchange of messages (steps 1 through 3) at the top of Figure 49-4.



**Figure 49-4: TCP window size adjustments and flow control** This diagram shows three message cycles, each of which results in the server reducing its receive window. In the first cycle, the server reduces it from 360 to 260 bytes, so the client's usable window can increase by only 40 bytes when it gets the server's acknowledgment. In the second and third cycles, the server reduces the window size by the amount of data it receives, which temporarily freezes the client's send window size, halting it from sending new data.

## **Reducing Send Window Size to Stop the Sending of New Data**

What if the server is so bogged down that it cannot process *any* of the bytes received? Let's suppose that the next transmission from the client is 180 bytes in size, but the server is so busy it cannot remove any of them.

In this case, the server could buffer the 180 bytes and, in the acknowledgment it sends for those bytes, reduce the window size by the same amount: from 260 to 80 bytes. When the client received the acknowledgment for 180 bytes, it would see the window size had reduced by 180 bytes as well. It would slide its window by the same amount as the window size was reduced! This is effectively like the server saying, "I acknowledge receipt of 180 bytes, but I am not allowing you to send any new bytes to replace them." Another way of looking at this is that the left edge of the window slides 180 bytes, while the right edge remains fixed. And as long as the right edge of the window doesn't move, the client cannot send any more data than it could before receipt of the acknowledgment. This is the middle exchange (steps 4 to 6) in Figure 49-4.

### **Closing the Send Window**

This process of window adjustment can continue, and, of course, can be done by both devices, even though we are considering only the client-sends-to-server side of the equation here. If the server continues to receive data from the client faster than it can pump it out to the application, it will continue to reduce the size of its receive window.

To continue our example, suppose that after the send window is reduced to 80 bytes, the client sends a third request, this one 80 bytes in length, but the server is still busy. The server then reduces its window all the way down to 0, which is called *closing* the window. This tells the client the server is very overloaded, and it should stop routine sending of data entirely, as shown in the bottom third of Figure 49-4. Later on, when the server is less loaded down, it can increase the window size for this connection again, permitting more data to be transferred.

**KEY CONCEPT** The TCP sliding window system is used not just for ensuring reliability through acknowledgments and retransmissions, but it is also the basis for TCP's *flow control* mechanism. By increasing or reducing the size of its receive window, a device can raise or lower the rate at which its connection partner sends it data. In the case where a device becomes extremely busy, it can even reduce the receive window to zero. This will close the window and halt any further transmissions of data until the window is reopened.

While conceptually simple, flow control using window size adjustment can be very tricky. If we aren't careful about how we make changes to window size, we can introduce serious problems in the operation of TCP. There are also special situations that can occur, especially in cases where the window size is made small in response to a device becoming busy. The next two sections explore window management issues and changes that need to be made to the basic sliding window system to address them.

## TCP Window Management Issues

Each of the two devices on a TCP connection can adjust the window size it advertises to the other, to control the flow of data over the connection. Reducing the size of the window forces the other device to send less data; increasing the window size lets more data flow. In theory, we should be able to just let the TCP software on each of the devices change the window size as needed to match the speed at which data both enters the buffer and is removed from it to be sent to the receiving application.

Unfortunately, certain changes in window size can lead to undesirable consequences. These can occur both when the size of the window is reduced and when it is increased. For this reason, there are a few issues related to *window size management* that we need to consider. As in previous sections, we'll use for illustration a modification of the same client/server example introduced in Chapter 48.

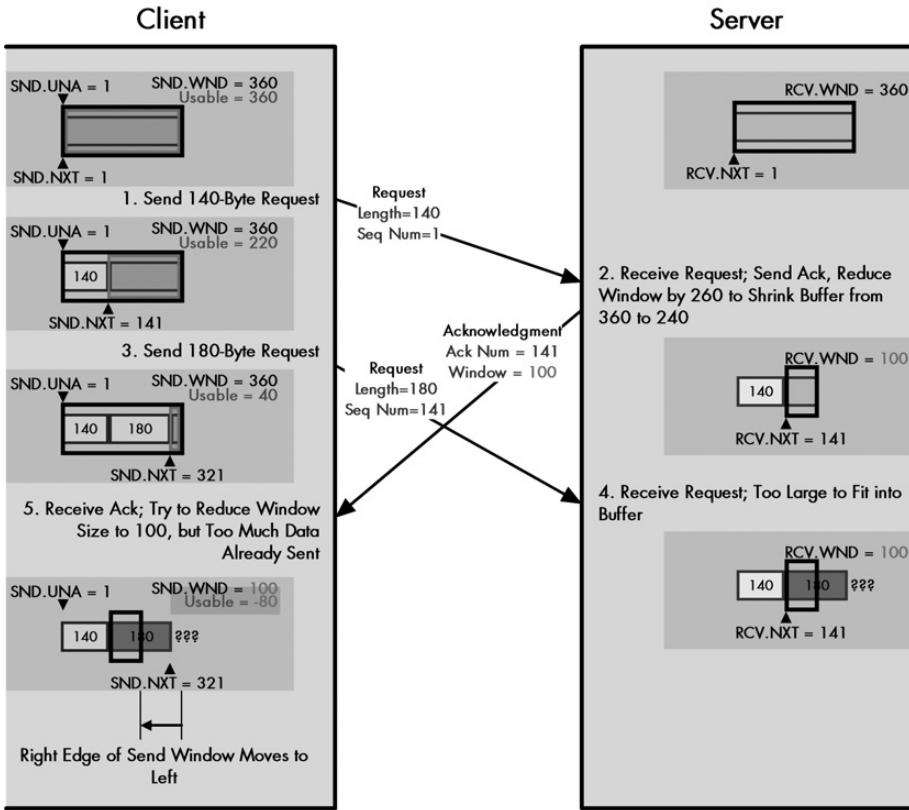
### Problems Associated with Shrinking the TCP Window

One window size management matter is related to just how quickly a device reduces the size of its receive window when it gets busy. Let's say the server starts with a 360-byte receive window, as in the aforementioned example, and receives 140 bytes of data that it acknowledges but cannot remove from the buffer immediately. The server can respond by reducing the size of the window it advertises back to the client. In the case where no bytes can be removed from the buffer at all, the window size is reduced by the same 140 bytes that were added to the buffer. This freezes the right edge of the client's send window, so it cannot send any additional data when it gets an acknowledgment.

What if the server were so overloaded that we actually needed to reduce the size of the *buffer* itself? Say memory was short and the operating system said, "I know you have 360 bytes allocated for the receive buffer for this connection, but I need to free up memory, so now you only have 240." The server still cannot immediately process the 140 bytes it received, so it would need to drop the window size it sent back to the client all the way from 360 bytes down to 100 bytes (240 in the total buffer less the 140 already received).

In effect, doing this actually moves the right edge of the client's send window *back to the left*. It says, "Not only can't you send more data when you receive this acknowledgment, but you now can send *less* when you do send data." In TCP parlance, this is called *shrinking the window*.

There's a very serious problem with doing this, however: While the original 140 bytes were in transit from the client to the server, the client still thought it had 360 bytes of total window, of which 220 bytes were *usable* (360 less 140). The client may well have already sent some of those 220 bytes of data to the server before it got the notification that the server had shrunk the window! If so, and the server reduced its buffer to 240 bytes with 140 used, when those 220 bytes showed up at the server, only 100 would fit, and any additional ones would need to be discarded. This would force the client to need to retransmit that data, which is inefficient. Figure 49-5 illustrates graphically how this situation would play out.



**Figure 49-5: The problem with shrinking the TCP window** In this modification of the example of Figure 49-4, the client begins with a usable window size of 360 bytes. It sends a 140-byte segment and then a short time thereafter sends one of 180 bytes. The server is busy, however, and when it receives the first transmission, it decides to reduce its buffer to 240 bytes. It holds the 140 bytes just received and reduces its receive window all the way down to 100 bytes. When the client's 180-byte segment arrives, there is room for only 100 of the 180 bytes in the server's buffer. When the client gets the new window size advertisement of 100, it will have a problem, because it already has 180 bytes sent but not acknowledged.

### Reducing Buffer Size Without Shrinking the Window

To prevent the problems associated with shrinking windows from occurring, TCP adds a simple rule to the basic sliding window mechanism: A device is not allowed to shrink the window.

Note that there is a potential terminology ambiguity here. The words *shrinking* and *reducing* are sometimes used synonymously in colloquial discussions. As we've seen, there's nothing wrong with *reducing* the size of the window. The problem of *shrinking* the window refers only to the case where we reduce the window size so much that we contradict a prior window advertisement by *taking back* permission to send a certain number of bytes.

Another way of looking at this is that *shrinking* occurs whenever the server sends back a window size advertisement smaller than what the client considers its usable window size to be at that time. In this case, the server shrunk the window, because at the time it was acknowledging the 140 bytes, it sent back a window size of 100, which is less than the 220-byte usable window the client had then.

Of course, there may be cases where we *do* need to reduce a buffer, so how should this be handled? Instead of shrinking the window, the server must be more patient. In the example in the previous section, where the buffer needs to be reduced to 240 bytes, the server must send back a window size of 220, freezing the right edge of the client's send window. The client can still fill the 360-byte buffer, but it cannot send more than that. As soon as 120 bytes are removed from the server's receive buffer, the buffer can then be reduced in size to 240 bytes with no data loss. Then the server can resume normal operation, increasing the window size as bytes are taken from the receive buffer.

**KEY CONCEPT** A phenomenon called *shrinking the window* occurs when a device reduces its receive window so much that its partner device's usable transmit window shrinks in size (meaning that the right edge of its send window moves to the left). Since this can result in data already in transit needing to be discarded, devices must instead reduce their receive window size more gradually.

### ***Handling a Closed Window and Sending Probe Segments***

Another special window management problem is how to deal with the case where a device must reduce the send window size all the way down to zero. As noted earlier, this is called *closing the receive window*. Since the server's receive window is the client's send window, reducing its size to zero means the client cannot send any more data. This situation continues until the client receives from the server a new acknowledgment with a nonzero Window field, which reopens the window. Then the client is able to send again.

The problem with this situation is that the client must depend on receipt of the “window opening” segment from the server. Like all TCP segments, this segment is carried over IP, which is unreliable. Remember that TCP is reliable only because it acknowledges sent data and retransmits lost data if necessary, but it can never guarantee that any particular segment gets to its destination. This means that when the server tries to reopen the window with an acknowledgment segment containing a larger Window field, it's possible that the client will never get the message. The client might conclude that a problem had occurred and terminate the connection.

To prevent this from happening, the client can regularly send special *probe* segments to the server. The purpose of these probes is to prompt the server to send back a segment containing the current window size. The probe segment can contain either zero or one byte of data, even when the window is closed. The probes will continue to be sent periodically until the window reopens, with the particular implementation determining the rate at which the probes are generated.

**KEY CONCEPT** A device that reduces its receive window to zero is said to have *closed* the window. The other device's send window is thus closed; it may not send regular data segments. It may, however, send *probe* segments to check the status of the window, thus making sure it does not miss notification when the window reopens.

When the server decides to reopen the closed window, there is another potential pitfall: opening the window to too small a value. In general, when the receive window is too small, this leads to the generation of many small segments, greatly reducing the overall efficiency of TCP. The next section explores this well-known problem and how it is resolved through changes to the basic sliding window mechanism.

## TCP Silly Window Syndrome

In the description of TCP's maximum segment size (MSS) parameter in Chapter 48, I explained the trade-off in determining the optimal size of TCP segments. If segments are too large, we risk having them become fragmented at the IP level. If they're too small, we get greatly reduced performance, because we are sending a small amount of data in a segment with at least 40 bytes of header overhead. We also use up valuable processing time that is required to handle each of these small segments.

The MSS parameter ensures that we don't send segments that are too large; TCP is not allowed to create a segment larger than the MSS. Unfortunately, the basic sliding windows mechanism doesn't provide any *minimum* size of segment that can be transmitted. In fact, not only is it *possible* for a device to send very small, inefficient segments, the simplest implementation of flow control using unrestricted window size adjustments *ensures* that under conditions of heavy load, window size will become small, leading to significant performance reduction!

### How Silly Window Syndrome Occurs

To see how the *silly window syndrome* (SWS) can happen, let's consider an example that is a variation on the one we've been using so far in this section. We'll assume the MSS is 360 bytes and a client/server pair where the server's initial receive window is set to this same value, 360. This means the client can send a full-sized segment to the server. As long as the server can keep removing the data from the buffer as fast as the client sends it, we should have no problem. (In reality, the buffer size would normally be larger than the MSS.)

Now, imagine that instead, the server is bogged down for whatever reason while the client needs to send it a great deal of data. For simplicity, let's say that the server is able to remove only 1 byte of data from the buffer for every 3 bytes it receives. Let's say it also removes 40 additional bytes from the buffer during the time it takes for the next client's segment to arrive. Here's what will happen:

1. The client's send window is 360 bytes, and it has a lot of data to send. It immediately sends a 360-byte segment to the server. This uses up its entire send window.

2. When the server gets this segment, it acknowledges it. However, it can remove only 120 bytes, so the server reduces the window size from 360 to 120 bytes. It sends this in the Window field of the acknowledgment.
3. The client receives an acknowledgment of 360 bytes and sees that the window size has been reduced to 120. It wants to send its data as soon as possible, so it sends off a 120-byte segment.
4. The server has removed 40 more bytes from the buffer by the time the 120-byte segment arrives. The buffer thus contains 200 bytes (240 from the first segment, less the 40 removed). The server is able to immediately process one-third of those 120 bytes, or 40 bytes. This means 80 bytes are added to the 200 that already remain in the buffer, so 280 bytes are used up. The server must reduce the window size to 80 bytes.
5. The client will see this reduced window size and send an 80-byte segment.
6. The server started with 280 bytes and removed 40, so 240 bytes remain. It receives 80 bytes from the client and removes one-third, so 53 are added to the buffer, which becomes 293 bytes. It reduces the window size to 67 bytes (360 minus 293).

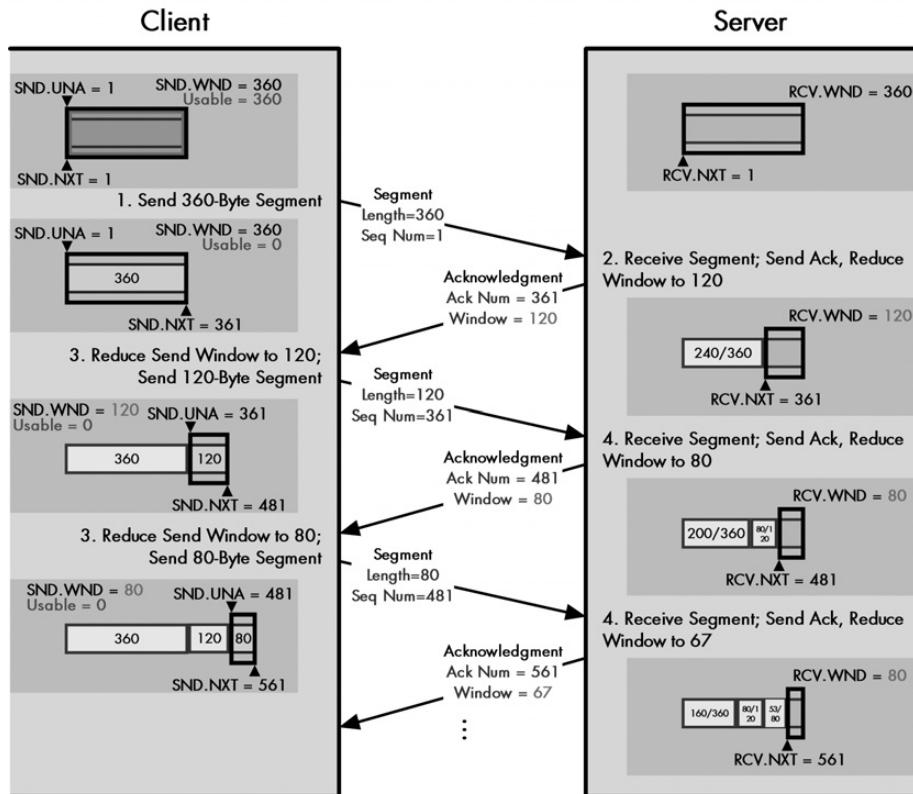
This process, which is illustrated in Figure 49-6, will continue for many rounds, with the window size getting smaller and smaller, especially if the server gets even more overloaded. Its rate of clearing the buffer may decrease even more, and the window may close entirely.

Let's suppose this happens. Now, eventually, the server will remove some of the data from this buffer. Let's say it removes 40 bytes by the time the first closed-window probe from the client arrives. The server then reopens the window to a size of 40 bytes. The client is still desperate to send data as fast as possible, so it generates a 40-byte segment. And so it goes, with likely all the remaining data passing from the client to the server in tiny segments, until either the client runs out of data or the server clears the buffer more quickly.

Now imagine the worst-case scenario. This time, it is the application process on the server that is overloaded. It is drawing data from the buffer one byte at a time. Every time it removes a byte from the server's buffer, the server's TCP opens the window with a window size of exactly 1 and puts this in the Window field in an acknowledgment to the client. The client then sends a segment with exactly one byte, refilling the buffer until the application draws off the next byte.

None of this represents a *failure per se* of the sliding window mechanism. It is working properly to keep the server's receive buffer filled and to manage the flow of data. The problem is that the sliding window mechanism is concerned only with managing the buffer. It doesn't take into account the inefficiency of the small segments that result when the window size is micromanaged in this way. In essence, by sending small window size advertisements, we are winning the battle but losing the war.

Early TCP/IP researchers who discovered this phenomenon called it *silly window syndrome (SWS)*, a play on the phrase *sliding window system*, which expresses their opinion on how it behaves when it gets into this state.



**Figure 49-6: TCP silly window syndrome (SWS)** This diagram shows one example of how the phenomenon known as TCP silly window syndrome can arise. The client is trying to send data as fast as possible to the server, which is very busy and cannot clear its buffers promptly. Each time the client sends data, the server reduces its receive window. The size of the messages the client sends shrinks until it is sending only very small, inefficient segments. Note that in this diagram, I have shown the server's buffer fixed in position, rather than sliding to the right, as in the other diagrams in this chapter. This way, you can see the receive window decreasing in size more easily.

The examples discussed show how SWS can be caused by the advertisement of small window sizes by a receiving device. It is also possible for SWS to happen if the sending device isn't careful about how it generates segments for transmission, regardless of the state of the receiver's buffers. For example, suppose the client TCP in the example shown in Figure 49-6 was receiving data from the sending application in blocks of 10 bytes at a time. However, the sending TCP was so impatient to get the data to the client that it took each 10-byte block and immediately packaged it into a segment, even though the next 10-byte block was coming shortly thereafter. This would result in a needless swarm of inefficient 10-byte segments.

**KEY CONCEPT** The basic TCP sliding window system sets no minimum size on transmitted segments. Under certain circumstances, this can result in a situation where many small, inefficient segments are sent, rather than a smaller number of large ones. Affectionately termed *silly window syndrome (SWS)*, this phenomenon can occur either as a result of a recipient advertising window sizes that are too small or a transmitter being too aggressive in immediately sending out very small amounts of data.

## **Silly Window Syndrome Avoidance Algorithms**

Since SWS is caused by the basic sliding window system not paying attention to the result of decisions that create small segments, dealing with SWS is conceptually simple: Change the system so that we avoid small window size advertisements, and at the same time, also avoid sending small segments. Since both the sender and recipient of data contribute to SWS, changes are made to the behavior of both to avoid SWS. These changes are collectively termed *SWS avoidance algorithms*.

### **Receiver SWS Avoidance**

Let's start with SWS avoidance by the receiver. As we saw in the previous example, the receiver contributed to SWS by reducing the size of its receive window to smaller and smaller values. This caused the right edge of the sender's send window to move by ever-smaller increments, leading to smaller and smaller segments. To avoid SWS, we simply make the rule that the receiver may not update its advertised receive window in such a way that this leaves too little usable window space on the part of the sender. In other words, we restrict the receiver from moving the right edge of the window by too small an amount. The usual minimum that the edge may be moved is either the value of the MSS parameter or one-half the buffer size, whichever is less.

Let's see how we might use this in the example shown in Figure 49-6. When the server receives the initial 360-byte segment from the client and can process only 120 bytes, it does not reduce the window size to 120. It reduces it all the way to zero, closing the window. It sends this back to the client, which will then stop and not send a small segment. Once the server has removed 60 more bytes from the buffer, it will now have 180 bytes free, half the size of the buffer. It now opens the window up to 180 bytes in size and sends the new window size to the client.

It will continue to advertise only either 0 bytes or 180 or more bytes, not smaller values in between. This seems to slow down the operation of TCP, but it really doesn't. Because the server is overloaded, the limiting factor in overall performance of the connection is the rate at which the server can clear the buffer. We are just exchanging many small segments for a few larger ones.

### **Sender SWS Avoidance and Nagle's Algorithm**

SWS avoidance by the sender is accomplished generally by imposing "restraint" on the part of the transmitting TCP. Instead of trying to immediately send data as soon as we can, we wait to send it until we have a segment of a reasonable size. The specific method for doing this is called *Nagle's algorithm*, named for its inventor, John Smith. (Just kidding, it was John Nagle.) Simplified, this algorithm works as follows:

- As long as there is no unacknowledged data outstanding on the connection, as soon as the application wants, data can be immediately sent. For example, in the case of an interactive application like Telnet, a single keystroke can be pushed in a segment.
- While there *is* unacknowledged data, all subsequent data to be sent is held in the transmit buffer and not transmitted until either all the unacknowledged data is acknowledged or we have accumulated enough data to send a full-sized (MSS-sized) segment. This applies even if a push is requested by the user.

This might seem strange, especially the part about buffering data despite a push request! You might think this would cause applications like Telnet to break. In fact, Nagle's algorithm is a very clever method that suits the needs of both low-data-rate interactive applications like Telnet and high-bandwidth file-transfer applications.

If you are using something like Telnet where the data is arriving very slowly (humans are very slow compared to computers), the initial data (first keystroke) can be pushed right away. The next keystroke must wait for an acknowledgment, but this will probably come reasonably soon relative to how long it takes to hit the next key. In contrast, more conventional applications that generate data in large amounts will automatically have the data accumulated into larger segments for efficiency.

Nagle's algorithm is actually far more complex than this description, but this section is already getting too long. RFC 896 discusses it in (much) more detail.

**KEY CONCEPT** Modern TCP implementations incorporate a set of *SWS avoidance algorithms*. When receiving, devices are programmed not to advertise very small windows, waiting instead until there is enough room in the buffer for one of a reasonable size. Transmitters use *Nagle's algorithm* to ensure that small segments are not generated when there are unacknowledged bytes outstanding.

## TCP Congestion Handling and Congestion Avoidance Algorithms

By changing the window size that a device advertises to a peer on a TCP connection, the device can increase or decrease the rate at which its peer sends it data. This is how the TCP sliding window system implements flow control between the two connected devices. We've seen how this works in this chapter, including the changes required to the basic mechanism to ensure performance remains high by reducing the number of small segments sent.

Flow control is a very important part of regulating the transmission of data between devices, but it is limited in the following respect: It considers only what is going on within each of the devices on the connection, and *not* what is happening in devices between them. In fact, this "self-centeredness" is symptomatic of architectural layering. Since we are dealing with how TCP works between a typical server and client at layer 4, we don't worry about how data gets between them; that's the job of IP at layer 3.

### Congestion Considerations

In practice, what is going on at layer 3 can be quite important. Considered from an abstract point of view, our server and client may be connected directly using TCP, but all the segments we transmit are carried across an internetwork of networks and routers between them. These networks and routers are also carrying data from many other connections and higher-layer protocols. If the internetwork becomes

very busy, the speed at which segments are carried between the endpoints of our connection will be reduced, and they could even be dropped. This is called *congestion*.

Again, at the TCP level, there is no way to directly comprehend what is causing congestion or why. It is perceived simply as inefficiencies in moving data from one device to another, through the need for some segments to be retransmitted. However, even though TCP is mostly oblivious to what is happening on the internetwork, it *must* be smart enough to understand how to deal with congestion and not exacerbate it.

Recall that each segment that is transmitted is placed in the retransmission queue with a retransmission timer. Now, suppose congestion dramatically increased on the internetwork, and there were no mechanisms in place to handle congestion. Segments would be delayed or dropped, which would cause them to time out and be retransmitted. This would increase the amount of traffic on the internetwork between our client and server. Furthermore, there might be thousands of other TCP connections behaving similarly. Each would keep retransmitting more and more segments, increasing congestion further, leading to a vicious circle. Performance of the entire internetwork would decrease dramatically, resulting in a condition called *congestion collapse*.

The message is clear: TCP cannot just ignore what is happening on the internetwork between its connection endpoints. To this end, TCP includes several specific algorithms that are designed to respond to congestion or avoid it in the first place. Many of these techniques can be considered, in a way, to be methods by which a TCP connection is made less selfish; that is, it tries to take into account the existence of other users of the internetwork over which it operates. While no single connection by itself can solve congestion of an entire internetwork, having all devices implement these measures collectively reduces congestion due to TCP.

The first issue is that we need to know when congestion is taking place. By definition, congestion means intermediate devices—routers—are overloaded. Routers respond to overloading by dropping datagrams. When these datagrams contain TCP segments, the segments don't reach their destination, and they are therefore left unacknowledged and will eventually expire and be retransmitted. This means that when a device sends TCP segments and does not receive acknowledgments for them, it can be assumed that, in most cases, they have been dropped by intermediate devices due to congestion. By detecting the rate at which segments are sent and not acknowledged, a TCP device can infer the level of congestion on the network between itself and its TCP connection peer.

## **TCP Congestion-Handling Mechanisms**

After getting information about congestion, we must then decide what to do with that information. The main TCP standard, RFC 793, includes very little information about TCP congestion-handling issues. That is because early versions of TCP based solely on this standard didn't include congestion-handling measures. Problems with these early implementations led to the discovery that congestion was an important issue. The measures used in modern devices were developed over the years, and eventually documented in RFC 2001, “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms.”

**KEY CONCEPT** TCP flow control is an essential part of regulating the traffic flow between TCP devices, but takes into account only how busy the two TCP endpoints are. It is also important to take into account the possibility of *congestion* of the networks over which any TCP session is established, which can lead to inefficiency through dropped segments. To deal with congestion and avoid contributing to it unnecessarily, modern TCP implementations include a set of Congestion Avoidance algorithms that alter the normal operation of the sliding window system to ensure more efficient overall operation.

RFC 2001 refers to four algorithms: Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. In practice, these features are all related to each other. Slow Start and Congestion Avoidance are distinct algorithms but are implemented using a single mechanism, involving the definition of a *congestion window* that limits the size of transmissions and whose size is increased or decreased depending on congestion levels. Fast Retransmit and Fast Recovery are implemented as changes to the mechanism that implements Slow Start and Congestion Avoidance.

The following sections provide simplified summaries of how these algorithms work. My goal is simply to help you get a feel for how congestion is handled in TCP in general terms.

**NOTE** *Congestion handling is a rather complex process. If you want to learn more, RFC 2001 contains the technical details, showing how each of the algorithms is implemented in each device.*

### **Slow Start**

In the original implementation of TCP, as soon as a connection was established between two devices, they could each go “hog wild,” sending segments as fast as they liked as long as there was room in the other device’s receive window. In a busy internetwork, the sudden appearance of a large amount of new traffic could exacerbate any existing congestion. To alleviate this, modern TCP devices are restrained in the rate at which they initially send segments.

Each sender is at first restricted to sending only an amount of data equal to one full-sized segment—that is, equal to the MSS value for the connection. Each time an acknowledgment is received, the amount of data the device can send is increased by the size of another full-sized segment. Thus, the device starts slow in terms of how much data it can send, with the amount it sends increasing until either the full window size is reached or congestion is detected on the link. In the latter case, the Congestion Avoidance feature, described next, is used.

### **Congestion Avoidance**

When potential congestion is detected on a TCP link, a device responds by throttling back the rate at which it sends segments. A special algorithm is used that allows the device to drop the rate at which segments are sent quickly when congestion occurs. The device then uses the Slow Start algorithm to gradually increase the transmission rate back up again to try to maximize throughput without congestion occurring again.

## **Fast Retransmit**

We've already seen in our look at TCP segment retransmission that when segments are received by a device out of order (noncontiguously), the recipient will acknowledge only the ones received contiguously. The Acknowledgment Number field will specify the sequence number of the byte it expects to receive next. So, in the example given in that section, Segments 1 and 2 were acknowledged, while Segment 4 was not because Segment 3 was not received.

It is possible for a TCP device to respond with an acknowledgment when it receives an out-of-order segment, simply reiterating that it is stuck waiting for a particular byte number. So, when the client in that example receives Segment 4 and not Segment 3, it could send back an acknowledgment saying, "I am expecting the first byte of Segment 3 next."

Now, suppose this happens over and over. The server, not realizing that Segment 3 was lost, sends Segments 5, 6, and so on. Each time a segment is received, the client sends back an acknowledgment specifying the first byte number of Segment 3. Eventually, the server can reasonably conclude that Segment 3 is lost, even if its retransmission timer has not expired.

The Fast Retransmit feature dictates that if three or more of these acknowledgments are received, all saying, "I want the segment starting with byte  $N$ ," then it's probable that the segment starting with byte  $N$  has been lost, usually because it was dropped due to congestion. In this case, the device will immediately retransmit the missing segment, without going through the normal retransmission queue process. This improves performance by eliminating delays that would suspend effective data flow on the link.

## **Fast Recovery**

When Fast Retransmit is used to resend a lost segment, the device using it performs Congestion Avoidance, but does not use Slow Start to increase the transmission rate back up again. The rationale for this is that since multiple ACKs were received by the sender, all indicating receipt of out-of-order segments, this indicates that several segments have already been removed from the flow of segments between the two devices. For efficiency reasons, then, the transmission rate can be increased more quickly than when congestion occurs in other ways. This improves performance compared to using the regular Congestion Avoidance algorithm after Fast Retransmit.



# **SECTION III**

## **TCP/IP APPLICATION LAYER PROTOCOLS**

The OSI Reference Model is used to describe the architecture of networking protocols and technologies and to show how they relate to one another. In the chapter describing OSI Reference Model concepts (Chapter 5), I mentioned that its seven layers could be organized into two layer groupings: the lower layers (1 through 4) and the upper layers (5 through 7). While there are certainly other ways to divide the layers, this split best reflects the different roles that the layers play in a network.

The lower layers are concerned primarily with the mechanics of formatting, encoding, and sending data over a network. These layers involve software elements, but they are often closely associated with networking hardware devices. In contrast, the upper layers are concerned mainly with user interaction and the implementation of software applications, protocols, and services that let us actually use the network. These elements generally don't need to worry about details, relying on the lower layers to ensure that data gets to where it needs to go reliably.

In this section, I describe the details of the many protocols and applications that occupy the upper layers in TCP/IP. The organization of this section is quite different from the previous section's organization. Since the TCP/IP protocol suite uses an architecture that lumps all the higher layers together, even attempting to differentiate between these layers is not

worthwhile. For these reasons, this section is divided by functions, rather than by layers. It contains ten parts: four that discuss application layer protocols that support the operation of TCP/IP, and six that discuss actual application protocols.

The first part discusses naming systems, especially the TCP/IP Domain Name System (DNS). The second part overviews file and resource sharing protocols, with a focus on the Network File System (NFS). The third part covers TCP/IP host configuration and the host configuration protocols: the Boot Protocol (BOOTP) and the Dynamic Host Configuration Protocol (DHCP). The fourth part describes the TCP/IP network management framework, including the Simple Network Management Protocol (SNMP) and Remote Network Monitoring (RMON).

The fifth part introduces TCP/IP applications with a look at application layer addressing and an overview of file and message transfer applications. The sixth part covers the general file transfer protocols: the File Transfer Protocol (FTP) and the Trivial File Transfer Protocol (TFTP). The seventh part explains the many related protocols that together form TCP/IP's electronic mail application. The eighth part covers the Web and the important Hypertext Transfer Protocol (HTTP). The ninth part describes Usenet (network news) and Gopher. Finally, the tenth part discusses interactive and administrative protocols.

# PART III-1

## **NAME SYSTEMS AND TCP/IP NAME REGISTRATION AND NAME RESOLUTION**

Humans and computers first started dealing with each other several decades ago. The relationship between man (and woman!) and machine has been a pretty good one overall, and this is reflected in the fact that while computers were once just the province of techies, they are now *mainstream*. However, there are areas where humans and computers simply don't see eye to eye. One of these is in the way that we deal with information.

Computers work best with numbers, while most people prefer not to work with numbers. This fundamental difference represented a problem for the designers of networking technology. It made sense from a technical standpoint to design addressing schemes for networks and internetworks using simple numeric identifiers, for simplicity and efficiency. Unfortunately, identifying computers using numeric addresses is cumbersome for people and becomes more so as the number of devices on a network increases. To solve this problem, the techies went to work and came up with *name systems* for networks. These mechanisms allow computers to continue to use simple, efficient numeric addresses, while letting humans specify names to identify network devices.

This part includes eight chapters that explain both the theory and practice behind networking name systems. The first chapter describes the motivation for name systems and the important concepts and techniques behind how they work. The second chapter provides an introduction to name systems on TCP/IP and a brief description of the simple host table name system.

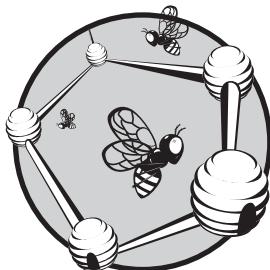
The remaining chapters describe the very important Domain Name System (DNS). The third chapter provides an overview of DNS, including a description of its characteristics and components. The fourth chapter discusses the DNS name space and architecture, and the fifth chapter covers the DNS name registration process, including hierarchical authorities and administration. The sixth chapter describes DNS name servers and how they represent, manage, and provide data when resolution is invoked. The seventh chapter describes DNS clients, called *resolvers*, how they initiate resolution, and the steps involved in the resolution process. Finally, the eighth chapter ties together the information about name servers and resolvers by providing a look at message exchange between these units, and describing the formats of messages, resource records, and DNS master files. This chapter includes a brief look at the changes made to DNS to support the new version 6 of the Internet Protocol (IPv6) and its much longer addresses.

Note that even though the abbreviation *DNS* usually stands for *Domain Name System*, you will also sometimes see the *S* stand for other words, especially *Service* or *Server*. Also, some documents refer to this name system as *the DNS*. Most people just say *DNS*, without the definite article, and that's the convention I follow here as well.

A set of related TCP/IP utilities called *nslookup*, *host*, and *dig* can be used by an administrator to query DNS name servers for information. They are useful for a variety of purposes, including manually determining the IP address of a host, checking for specific resource records maintained for a DNS name, and verifying the name resolution function. You can find more information about these utilities in Part III-10.

# 50

## **NAME SYSTEM ISSUES, CONCEPTS, AND TECHNIQUES**



Name systems can be considered as the diplomats of the networking protocol stack. Just as a political diplomat is skilled at speaking multiple languages and ensuring good communications between those who may view the world in different ways, name systems bridge the gulf between the numeric addresses that computers like to use and the simpler names that humans prefer.

Before looking at specific name systems, it makes sense to discuss them generally. This will help you to understand the reasons why these systems are important and the concepts that underlie all name systems, regardless of their specific implementation.

I begin this chapter with an overview of name systems and a discussion of why they were created. I then discuss the three main functions of a name system: the name space, name registration, and name resolution. I then

expand on this functional overview, illustrating how name spaces and architectures work, the issues behind name registration and administration, and finally, name resolution techniques and the practical issues in the resolution process.

This chapter provides an introduction to name systems and doesn't discuss specific name systems. I like to use examples to explain concepts and, for this purpose, do make reference to the TCP/IP Domain Name System (DNS) at times. However, you do not need to be familiar with DNS to follow this chapter.

## Name System Overview

One of several important differences between humans and computers is how we prefer to deal with information. Computers work with numbers, while very few humans like to do so. This distinction becomes particularly important when we look at how identifiers, or addresses, are assigned to network devices.

### ***Symbolic Names for Addressing***

To a computer, there is no problem with simply giving a number to each device on the network and using those numbers to move information around. Your computer would be perfectly happy if you assigned a number like 341,481,178,295 to it and all the other machines on your network, and then issued commands such as, "Send this file to machine 56,712,489,901." However, most humans don't want to use a network in this manner. These long, cryptic numbers don't mean anything to them. They want to tell their machine, "Send this file to Joe's computer," or "Print this on the color laser in the Sales department," or "Check the latest headlines on CNN's website."

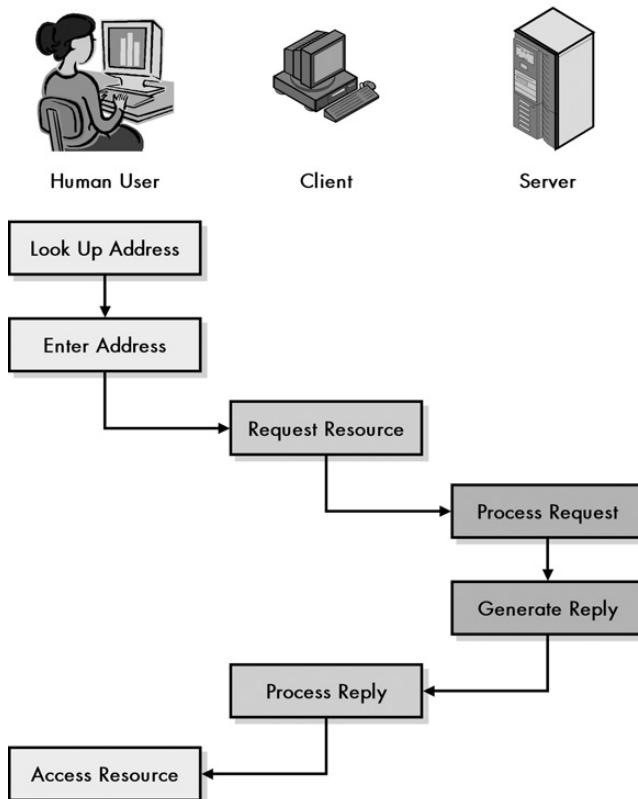
This difference led to the development of *name systems*. These technologies allow computers on a network to be given both a conventional numeric address and a more user-friendly, human-readable name, composed of letters, numbers, and other special symbols. Sometimes called a *symbolic name*, this can be used as an alternative form of addressing for devices. The name system takes care of the functions necessary to manage this system, including ensuring that names are unique, translating from names to numbers, and managing the list of names and numbers.

### ***A Paradox: Name Systems Are Both Essential and Unnecessary***

What's interesting about name systems is that they are extremely important to networks, but at the same time, they often aren't strictly necessary for a network to operate. This seeming paradox is due again to the difference between humans and computers. Computers need only the numeric addressing scheme, not the names assigned to them. So, without name systems, the computers and the network can still work, but it will be much harder for people to use them!

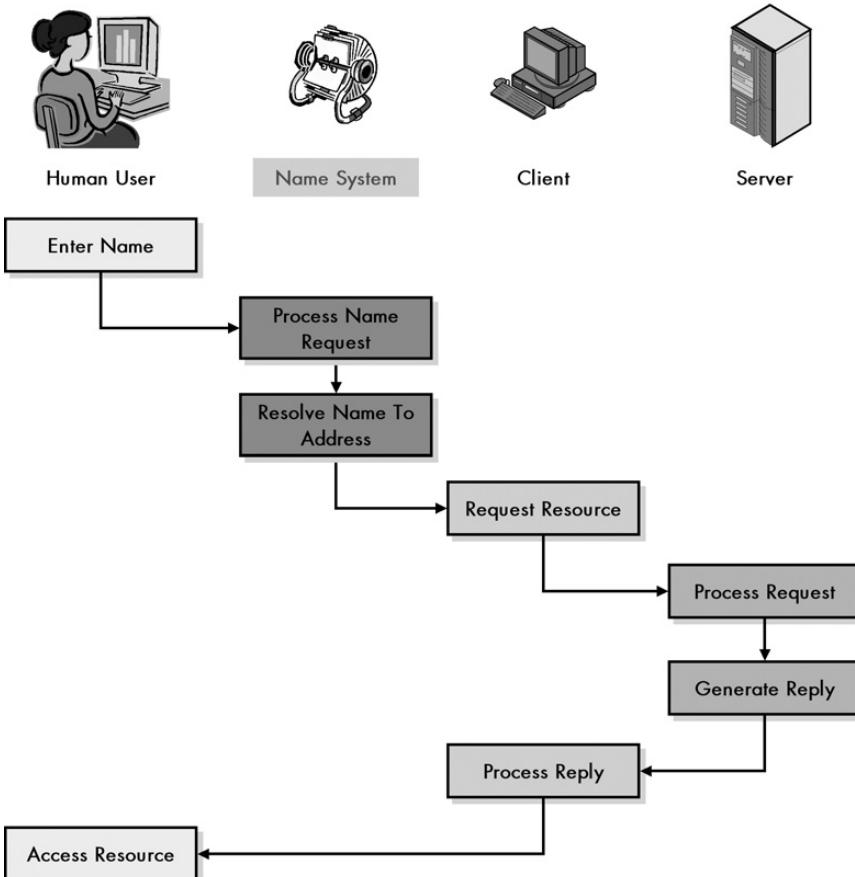
An example of this can most readily be seen when a problem disables the operation of a part of DNS used to provide naming services on the Internet. Technically, DNS isn't needed to use most parts of the Internet, because all communications use IP addresses. This means that even though you might normally access CNN's website at [www.cnn.com](http://www.cnn.com), you could instead just use the IP address 64.236.16.20.

The problem is that prior to reading this, you probably had no idea what the IP address of CNN's website is, and that's true of almost everyone else who uses the site as well. Also, you might want to check not just CNN's website, but perhaps 1, 2, or 20 other news sites. It would be difficult to remember the numbers for even a small percentage of the thousands of different websites on the Internet, so each time you wanted to access a resource, you would need to manually look up its address, as shown in Figure 50-1.



**Figure 50-1: Internetwork access without a name system** When there is no name system, a user must know the address of any device he or she wishes to access on the internetwork. Since most of us have limited memories for numbers, this means each access must be preceded by an inefficient, tedious, manual address lookup.

In contrast, it's much easier to remember the names of resources. With a name system, you just enter the name of a device, and the name system converts it to an address, as shown in Figure 50-2. This is why name systems are so important, even if they aren't needed by the networking technologies themselves. In fact, the reliance on name systems like DNS is so significant that many people don't even realize they can enter IP addresses into their web browsers!



**Figure 50-2:** Internetwork access with a name system

When an internetwork is equipped with a name system, the user no longer needs to know the address of a device to access it. He or she enters the name, and the name system converts it into an address automatically, like a computerized Rolodex, as shown here. The name system then passes the address to the client software, which uses that address to access the requested resource as if the user had entered it directly.

### **Factors That Determine the Necessity of a Name System**

More generally, the importance of a name system depends greatly on the characteristics of the network where it is used. The following are the three main issues in determining the need for a name system:

**Network Size** With a really small network and only a handful of computers, having human users remember the numeric addresses for these machines is at least feasible, if not ideal. For example, a small home network with two or three machines doesn't really *need* a name system, in theory. If you have thousands or millions of devices, however, the name system becomes essential.

**Address Size and Complexity** The more complex the numeric addressing scheme, or the larger the numbers used, the more difficult it is for humans to remember the numbers. This makes having a name system all the more essential for the users of those addresses.

**User Base Size and Skill** In the early days of networks, a small number of highly skilled and well-trained engineers used them, and these people sometimes just memorized the numbers of the machines they worked with every day. In modern networks with thousands or millions of regular users, expecting the average person to remember device numbers is not reasonable.

**KEY CONCEPT** Networking name systems are important because they allow devices to be assigned efficient numeric addresses, while still enabling humans to access them using names that are easier to remember. Name systems become more important as you increase the size of the network, the address, or the user base. They are also more essential when the user base is limited in skill or experience.

Looking at these issues, we can see that the trends in today's networks are all in the direction of increasing the importance of name systems. Our networks, both private and public, are growing larger, and we have more people using them, including more people without a technical background. We are also increasingly moving from small addresses to larger ones. The best example of this is the upcoming change to IP. While DNS is important for the 32-bit addresses used in IPv4, it's even *more* important for dealing with the enormous 128-bit addresses of IPv6 (see Part II-4).

## ***Basic Name System Functions: Name Space, Name Registration, and Name Resolution***

While the difference between numeric addresses and symbolic names is very significant to the users of network devices, it's important to remember that both numbers and names really serve the same basic purpose: *device identification*. Even when we use a name system to make devices easier to access, the computers themselves will still normally need to use the underlying numeric identifier. In essence, every device will end up with (at least) two identifiers: a number *and* a name.

The fact that devices end up with multiple identifiers is what allows both people and their machines to use the method of identification they prefer. However, it means that there must be ways of managing the assignment of names to devices and converting between them. A name system involves more than just slapping names on computers. It must be a complete *system* that allows names to be used by the humans while numbers continue to be used by the devices.

At the highest level, a name system must handle three basic functions:

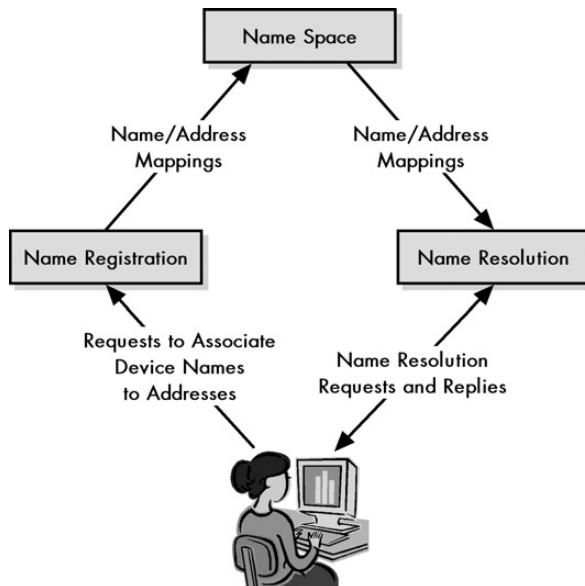
**Name Space Definition** The name system defines a *name space* for the networking system on which it runs. The name space, also sometimes called a *name architecture*, describes the rules for how names are structured and used. It also defines how the

name of one device is related to the names of other devices in the system and how to ensure that there are no invalid names that would cause problems with the system as a whole.

**Name Registration** To implement the name system, a name must be assigned to each device on the network. Like any addressing system, a name system cannot work properly unless every name on the system is unique. We need some way of managing how the names are assigned so the result is sensible. The process of linking specific names to particular devices is usually called *name registration*.

**Name Resolution** Even though humans like symbolic names, computers usually have little use for them. It is necessary to define a mechanism by which a device's symbolic name can be translated into its numeric address. This process is usually called *name resolution*.

The name space is more of a descriptive function, which defines how names work in the system. Name registration and resolution are more active functions, with each name system including one or more specific procedures for how these jobs are carried out. Name registration and resolution are in some ways complements of each other, so certain registration techniques are most often associated with particular resolution methods. In turn, the types of registration and resolution methods that are possible depend on the name space, and in particular, its architecture. These relationships are shown in simplified form in Figure 50-3.



**Figure 50-3: Name system functions**

This diagram shows the relationships between the three main functions of a name system. The *name space* defines the structure of the name system and the rules for creating names. The name space is used as the basis for the *name registration* method, which defines the mappings between names and addresses. When a user wants to access a device by name, a *name resolution* method is used to consult the

name space, determine what address is associated with a name, and then convert the name to an address. The processes of registration and resolution can be either quite plain or fairly complicated, depending on the type of name system used. Simple name systems are largely manual in operation, easy to understand, and best used in smaller networks. Larger, more complex networks and internetworks require more sophisticated methods of registration and resolution, which involve less administrator intervention and *scale* better as new machines are added to the network.

Although name registration and name resolution work as functions at the highest level, they are probably better thought of as *sets* of functions. Name registration is necessarily tied to issues such as name system administration and management, and understanding resolution requires that we look at a number of important implementation issues in the areas of efficiency and reliability. The rest of this chapter expands on this overview by considering each of these three functions in more detail.

**KEY CONCEPT** A name system consists of three theoretical high-level functions: the *name space*, which describes how names are created and organized; the *name registration* technique, which is used to set up relationships between names and addresses; and the *name resolution* method, which is responsible for translating names to addresses.

## Name Spaces and Name Architectures

The main idea of a name system is to provide a way to identify devices using symbolic names. Like any identification mechanism, before it can be used, we must define the way that identification will be performed. Numeric addressing schemes (like IP addresses) have rules for how addresses are created and assign addresses to each device from their *address space*. In a similar way, devices in a name system are given names from the system's *name space*.

### Name Space Functions

Of the three main components of a name system, the name space is the most abstract. It is also the most fundamental part of the system, since it actually describes how the names are created. There are several aspects to what the name space defines in a name system:

**Name Size and Maximum Number of Names** The name space specifies the number of characters (symbols) that compose names. It also defines the maximum number of names that can appear in the system.

**Name Rules and Syntax** The name space specifies which characters and symbols are allowed in a name. This is used to allow legal names to be chosen for all devices, while avoiding illegal names.

**Name Architecture and Semantics** Each name space uses a specific *architecture* or *structure*, which describes how names are constructed and interpreted.

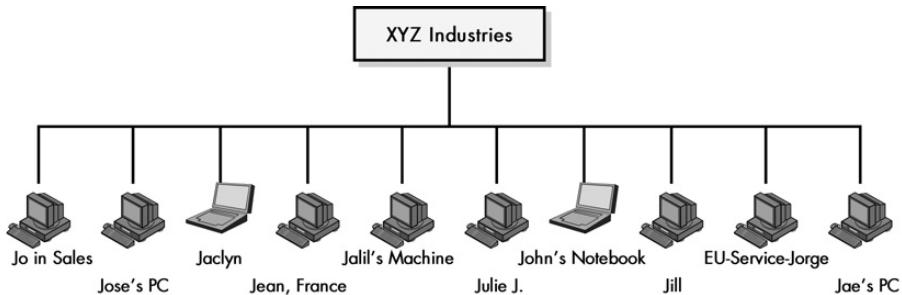
The concepts of name size and name syntax are relatively straightforward. The *name architecture* is probably the most important differentiating characteristic of name systems. For this reason, name spaces are sometimes even *called* name architectures. The architecture of the name space determines whether names are assigned and used as a simple unstructured set of symbols or have a more complex internal structure. In the latter case, the name space also must define how elements of a particular name are related to each other.

Theoretically, many different name architectures are possible. In practice, most fall into one of two categories: flat and hierarchical.

### **Flat Name Architecture (Flat Name Space)**

In a *flat name architecture*, names are assigned as a sequence of symbols that are interpreted as a single, whole label without any internal structure. There is no clear relationship between any name and any other name.

An example of this sort of architecture would be a name system where computers are given unstructured names like Engineering Workstation 1 or Joanne's PC, as shown in the example in Figure 50-4.



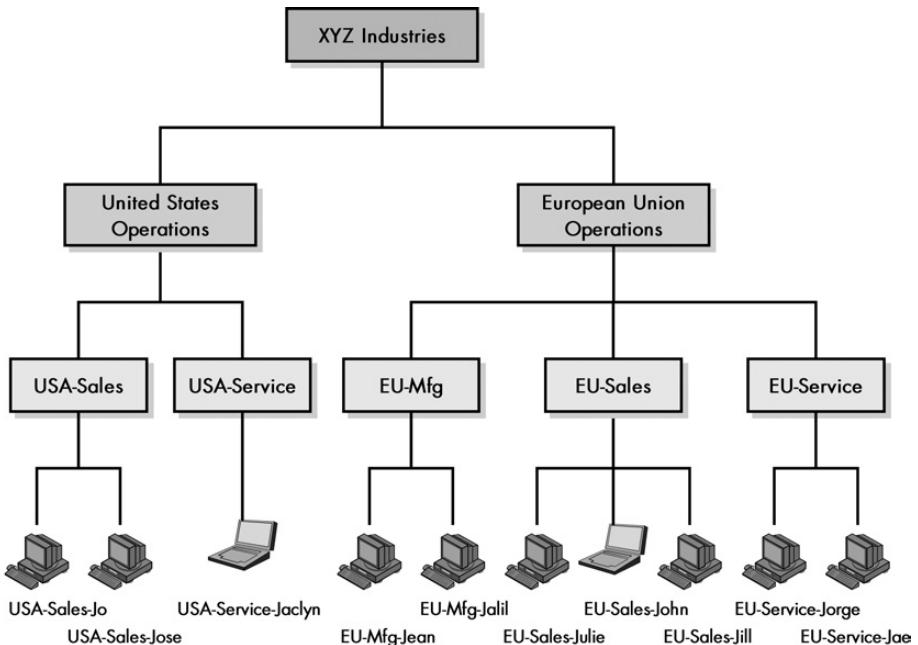
**Figure 50-4: Flat name architecture (flat name space)** This diagram shows an example of a flat name architecture. There is no structure that organizes the names or dictates how they must be constructed. Logically, each device is a peer of each of the others.

### **Hierarchical Name Architecture (Structured Name Space)**

In a *hierarchical name architecture*, or structured name space, the names are a sequence of symbols, but these symbols are assigned using a specific and clear structure. Each name consists of discrete elements that are related to each other, usually by using hierarchical parent/child semantics. There are many naming architectures in various contexts that use this type of hierarchical structure. For example, consider how a large company might set up an organization chart and name the executives and officers in the organization. One hypothetical example of a hierarchical name architecture is illustrated in Figure 50-5.

The best-known real-world example of a hierarchical name space is the name space of DNS (see Chapter 53), which uses text labels separated by periods (or *dots*) to form an internal structure. All the names in the system are organized into a structure, and a particular device's place in the structure can be determined by

looking at its name. For example, [www.tcpipguide.com](http://www.tcpipguide.com) refers to the World Wide Web server for *The TCP/IP Guide*, which is named under the umbrella of commercial (.com) companies.



**Figure 50-5:** Hierarchical name architecture (structured name space)

This diagram contains the same devices as Figure 50-4, but they have been arranged using a hierarchical, structured name architecture. In this case, the organization has chosen to structure its device names first by facility location, and then by department. Each name starts with something like USA-Service- or EU-Mfg-. This has immediate benefits by providing local control over device naming without risk of conflicts. If someone named John were hired into the USA sales force, his machine could be named USA-Sales-John, without conflicting with the machine owned by John of the European sales force (EU-Sales-John). The structure also makes it easier to know immediately where a device can be found within the organization.

### Comparing Name Architectures

As you will see in the next two sections in this chapter, the architecture of the name space is intimately related to how names are registered and managed, and ultimately, how they are resolved. A flat name space requires a central authority of some sort to assign names to all devices in the system to ensure uniqueness. A hierarchical name architecture is ideally suited to a more distributed registration scheme that allows many authorities to share in the registration and administrative process.

All of this means that the advantages and disadvantages of each of these architectures are not a great mystery. Flat name spaces have the advantage of simplicity and the ability to create short and easily remembered names, as shown in Figure 50-4. However, they do not scale well to name systems containing hundreds or thousands

of machines, due to the difficulties in ensuring each name is unique. For example, what happens if there are four people named John who all try to name their computers John's PC? Another issue is the overhead needed to centrally manage these names.

In contrast, hierarchical name spaces are more sophisticated and flexible, because they allow names to be assigned using a logical structure. We can name our machines using a hierarchy that reflects our organization's structure, for example, and give authority to different parts of the organization to manage parts of the name space. As long as each department is named uniquely and that unique department name is part of each machine name, we don't need to worry about each assigned name being unique across the entire organization; it just needs to be unique within the department. Thus, we can have four different machines named with their department name and John, as Figure 50-5 demonstrates. The price of this flexibility is the need for longer names and more complexity in name registration and resolution.

**KEY CONCEPT** The two most common types of name architecture are the flat name space and the hierarchical name space. Names in a flat name space are all peers with no relationship. In a hierarchical architecture, a multiple-level structure is used to organize names in a specific way. The flat system is simpler and satisfactory for small networks. The hierarchical name space is more flexible and powerful, and better suited to larger networks and internetworks.

## Name Registration Methods, Administration, and Authorities

It seems obvious that for our name system to be implemented, we need some method of assigning names to each of the devices that will use the system. Just as a name system has a name space that is comparable to an addressing system's address space, it also must implement a set of rules and procedures for assigning names, comparable to how an addressing system assigns addresses. This is called *name registration*.

### Name Registration Functions

In general, name registration encompasses the following four concepts and tasks:

**Name Assignment and Guarantee of Uniqueness** The core task of the name registration process is assigning names to devices. Like all identification schemes, a key requirement of name registration is ensuring that each name is unique. Duplicated names cause ambiguity and make consistent name resolution impossible.

**Central Registration Authority Designation** Ensuring uniqueness of names requires that there be someone in charge of the name assignment process. This *central registration authority* may be a single individual that maintains a file containing names, or it may be an organization that is responsible for the overall name registration process. The authority is also charged with resolving problems and conflicts that may arise in registrations.

**Registration Authority Delegation** In smaller name systems, the central registration authority may be responsible for the actual registration process for all devices. In larger, hierarchical name systems, having this process centralized is impractical. Instead, the central registration authority will divide the name space and *delegate* authority for registering names in different parts of it to subordinate organizations. This requires a delegation policy to be developed and implemented.

**Hierarchical Structure Definition** When a hierarchical name space is used, the central authority is responsible for defining how the structure will look. This, in turn, dictates how names can be registered in different parts of the hierarchy, and of course, also impacts how authority is delegated.

The complexity of the name registration process depends to a great extent on the size and complexity of the name system as a whole, and, in particular, on the architecture of the name space. In a simple name system using a flat name space, registration is usually accomplished using a single authority. There is no structure and usually no delegation of authority, so there isn't much to registration. For hierarchical name systems, name registration is tied tightly to the hierarchy used for names.

### ***Hierarchical Name Registration***

The central authority defines the structure of the hierarchy and decides how the hierarchy is to be *partitioned* into subsets that can be independently administered by other authorities. Those authorities may, in turn, delegate subsets of their name spaces as well, creating a flexible and extensible system.

This ability to delegate authority for name registration is one of the most powerful benefits of a hierarchical name space. For example, in DNS, a central authority is responsible for name registration as a whole. This central authority is in charge of deciding which top-level domains—such as .com, .edu, .info, and .uk—are allowed to exist. Authority for managing each of these subsets of the worldwide hierarchy is then delegated to other organizations. These organizations continue the process of dividing the hierarchy as they see fit. Eventually, each organization is able to decide how it will name its own internal systems independently; for example, IBM can register names in any way it sees fit within the ibm.com name.

### ***Name Registration Methods***

There are several common methods by which the actual process of registration is carried out. These include table name registration, broadcast name registration, and database registration. Each of these has its strengths and weaknesses, and again, some are better suited to flat name spaces and some to hierarchical ones.

#### **Table Name Registration**

Using table name registration, name assignments are maintained in a table by an administrator. When names need to be added, deleted, or changed, the table is edited.

This technique is usually associated with small, flat name space name systems. It has the same benefits and drawbacks as flat architecture in general: It is simple and easy to implement, but doesn't scale well to larger systems. With a dozen machines, having someone edit name registration tables is practical; with thousands of machines, it is not. It is also not conducive to a hierarchical system where there are multiple authorities, because the table needs to be kept in one place.

In larger internetworks, tables may be used as an adjunct to one of the other, more sophisticated, registration techniques.

### Broadcast Name Registration

Broadcast name registration is a trial-and-error technique. A device that wants to use a particular name sends out a message to all other devices on the network, asking if anyone else is already using it. If so, it chooses a different name. If not, the name is considered registered and can then be used.

This technique is more sophisticated than using tables, but it is still limited to use in relatively small systems. It is not practical to attempt to broadcast to thousands of systems, and this method could not be used over the Internet, since there is no way to broadcast to every device on an internetwork.

### Database Registration

With database registration, a database of name assignments is maintained. To register a name, a request must be made to have the name assignment added to the database. If the authority for the name system is entirely centralized, the database will be centralized and maintained by that authority. If authority for parts of the hierarchy is delegated, then a *distributed database* is used for registration, with each authority maintaining the part of the database describing its section of the hierarchy.

This is the most sophisticated technique and one normally associated with hierarchical name systems like DNS. It has several benefits, including flexibility, reliability, and distribution of maintenance effort. Its main drawback is complexity.

**KEY CONCEPT** *Name registration* is the process by which names are linked to addresses in a name system. It encompasses activities such as central registry authority designation and delegation, and name space structure management. The most common methods of name registration, in order of both increasing capability and complexity, are manual table maintenance, broadcast registration, and database registration.

## Name Resolution Techniques and Elements

As we discussed earlier in this chapter, using a name system creates two parallel identification systems for computers: the numbers used by machines and the names used by people. The job of the name system is to integrate these two schemes. Name registration allows humans to specify which machines use which names. This is only half the process, however; we also need a way for machines to take a name given to them by a human and translate it into the numeric address it can actually use for communication. This is called *name resolution*.

Name resolution, also sometimes called *name translation*, *mapping*, or *binding*, is the most well-known aspect of name systems, because it is where most of the “heavy lifting” of a name system occurs. The name space is generally set up once, and name registration occurs infrequently—only when names must be created or changed. In contrast, every user of a name system instructs the machines he or she uses to perform name resolution, hundreds or even thousands of times a day.

## **Name Resolution Methods**

Several different techniques can be used for name resolution. How this function is implemented depends a great deal on the other two name system functions: name space and name registration. As you might imagine, a simple name system with a simple name registration method will most often use a simple resolution method as well. Complex hierarchical systems with distributed databases require more sophistication in how names are resolved. There are three common name resolution methods: table name resolution, broadcast name resolution, and client/server name resolution.

### **Table-Based Name Resolution**

The table used for table-based name registration is consulted by a device when resolution needs to be performed. The table tells the device how to transform the name of the machine it needs to contact into an address.

This technique obviously corresponds to table name registration. It is the simplest and least capable of the three methods. Table name resolution is suitable for stand-alone use only in very small name systems, but it can be a supplement to other methods as well.

### **Broadcast Name Resolution**

When a device needs to resolve a name, it broadcasts a query that says something to this effect: “I need to send to the device named *X*. Who is that?” The device whose name is *X* responds, “I’m *X*, and my numeric address is *N*.”

This is the complement of broadcast name registration. It, too, can be used only in simple systems where every device can hear a broadcast. The use of broadcasts also makes it wasteful of network bandwidth.

### **Client/Server Name Resolution**

With client/server name resolution, servers are programmed with software that allows them to respond to name resolution requests sent by clients. These servers take the name in the request, look up the associated numeric identifier in a database, and send it back in a response.

This technique is generally used in conjunction with database name registration. It is the most complex name resolution method, but it is also the most efficient and the only one that can really work properly on a large, distributed hierarchical name system.

## **Client/Server Name Resolution Functional Elements**

Client/server name resolution is the method used for most large, modern name systems. The client/server method of request/reply resolution is similar to how many other protocols function. One thing that is unique about name resolution, however, is that name resolution isn't often invoked directly by the client. It's rare, for example, for a human user to say, "Please resolve the following name." We also certainly wouldn't want users to need to manually resolve a name to an address each time they wished to contact a device, as this would be cumbersome.

Instead, the system is automated by having software accept machine names input by users. The software resolves the name by passing it to a *name resolver* software component. The resolver acts as the client in the name resolution process. It contacts a *name server*, which responds to the request. The name resolver and name server constitute the two main functional elements in name resolution.

**KEY CONCEPT** *Name resolution* is arguably the most important of the main functional elements of a name system, because it is the part of the system that actually converts names into addresses. The two main components of name resolution are *name resolvers*, which act as clients in the resolution process, and *name servers*. The three main name resolution methods—table-based, broadcast, and client/server—correspond closely to the table, broadcast, and database methods of name registration.

In a distributed database for a hierarchical name system, multiple requests may be required, since name servers will contain only information for certain machines and not others. Resolvers follow a special procedure to travel the hierarchy until they find the server that has the information they want. Again, DNS's name resolution is the best example of this method.

## **Efficiency, Reliability, and Other Name Resolution Considerations**

As described in the previous section, the primary function of name resolution is allowing humans to identify devices using names, then converting these names into numbers so that computers can use the numbers instead. This basic task is conceptually quite simple, but it can become quite complex in implementation. The reason for this is the key characteristic that makes name resolution so different from the other tasks performed by a name system: the frequency with which it is done.

Name registration is seldom done, but name resolution is done very often. If you consider a large internetwork with thousands of users running various applications, millions of names must be resolved every day. Now, consider something like the Internet, which must process billions of client/server requests and replies daily! Ensuring that such systems work requires that we do more than just implement a resolution process; we must add facilities to ensure that resolution is done as effectively as possible.

## **Efficiency Considerations**

The first major concern with name resolution is *efficiency*. Name resolution uses up system resources, especially with resolution techniques that require requests and replies to be sent. This means we want to minimize the number of times resolution is performed, if at all possible. Now, consider that many people will frequently access the same machines over and over again. For example, if you go to a website called [www.thisisasite.com](http://www.thisisasite.com) for the first time, your system will need to resolve that name. After the home page for that site loads, if you click a link to another page on that site, the page will also be found at that same name: [www.thisisasite.com](http://www.thisisasite.com). So, it would be wasteful to need to resolve that name a second time.

To avoid this, name systems almost always include some sort of *caching* capability, which allows devices to remember recent name resolutions and retain the mapping from name to address for a period of time. Whenever a name needs to be resolved, the cache is first checked before going through the formal process of resolution. The use of caching eliminates the vast majority of actual name resolution requests that would otherwise be required.

The drawbacks of caching are that it requires some system resources of its own and that it adds complexity to the system. One issue is deciding how long to retain data in the cache. If we keep it too short a time, we generate extra unnecessary resolution requests. If we keep it too long, we risk having the mapping become stale if the name assignment for the machine changes. These are issues that a sophisticated name system must handle. A typical solution is to allow each name registration to specify how long information about that name-to-address link may be cached.

## **Reliability Considerations**

The next main concern after efficiency is name resolution *reliability*. As I said earlier in this chapter, having a name system isn't strictly necessary for the computers, but it's very important for the users, especially on a large network like the Internet.

While having a single central place that maintains all information about a name system may make administration simpler, it creates a dangerous single point of failure. If anything happens to the device storing the information, the entire name system fails. Modern name systems employ redundancies to prevent having the entire system rely on any particular device for resolution. A typical approach in a client/server system is to have multiple servers in different locations (or attached to different networks) that can respond to name resolution requests.

**KEY CONCEPT** Since name resolution is the part of a name system that is used most often, it is here that we must pay careful attention to implementation issues. The two most important ones are efficiency and reliability. Efficiency is essential due to the many thousands or millions of resolutions performed every day on a large system. Reliability is a consideration because users of the name system quickly come to rely on it, so we must make sure it is robust.

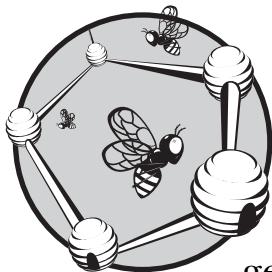
## **Other Considerations**

An optional feature in some name resolution systems is *load balancing*. When properly implemented, load balancing allows a single name to map to more than one underlying address. This allows requests sent to a particular virtual device to actually be directed to a number of different actual physical devices, spreading the load over multiple machines. A common use of this feature is for very popular websites that are visited often.

Finally, while name resolution is obviously designed to allow names to be mapped to addresses, there are cases where we may wish to go in the other direction: given a numeric address, find the name that goes with it. This process, called *reverse resolution*, is analogous to having a phone number and trying to find the name of the person or company to which it belongs. Just as we can't easily find the name matching a phone number using a conventional phone book (we would need to scan every page looking for the number), reverse resolution requires special support on the part of the name system. This is especially true if the name system data is distributed over many servers.

# 51

## **TCP/IP NAME SYSTEMS OVERVIEW AND THE HOST TABLE NAME SYSTEM**



TCP/IP has become sufficiently popular that many people—even those who aren't geeks—are fairly comfortable working with its numeric identifiers (IP addresses). Even so, it's a lot easier to work with names than numbers, and it's certainly easier to remember names. We can consider also that name systems become more important when used on larger networks, and TCP/IP is used to implement the Internet, the world's largest internetwork. Having a good name system is vital to the operation of the Internet, and thus, has become an important element of TCP/IP as a whole.

In this chapter, I begin the discussion of TCP/IP's name systems with a look at the history of the use of host names in TCP/IP and the early development of its name systems. I then provide a description of the simple host table name system, the first one used in the protocol suite. I discuss why host tables were replaced by the Domain Name System (DNS) and how, even today, they can be used to complement DNS functions.

**BACKGROUND INFORMATION** This chapter assumes that you are already familiar with the general concepts and issues of name systems explained in the preceding chapter.

## A Brief History of TCP/IP Host Names and Name Systems

In the previous chapter, I described an interesting paradox: Even though name systems aren't strictly necessary for the functioning of a networking system, they make using a network so much easier for people that they are considered an essential part of most networks. No better evidence of this can be found than in the history of name system development in TCP/IP.

### ***Developing the First Name System: ARPAnet Host Name Lists***

The history of name systems in the TCP/IP protocol suite actually goes back well before the Transmission Control Protocol (TCP) and Internet Protocol (IP) were themselves even created. In the late 1960s and early 1970s, when the predecessor of the Internet, called the *ARPAnet*, was being developed, it used older networking protocols that served the same function that TCP and IP do today.

The ARPAnet was very small by today's standards, containing at first only a few machines, referred to as *hosts*, just as TCP/IP machines often are called today. The addressing scheme was also very simple, consisting of just the combination of a computer number and a port number for each host. With only a handful of machine names, it was easy to memorize addresses, but as the ARPAnet grew to several dozen machines, this scheme became untenable.

As early as 1971, it was apparent to the engineers designing the ARPAnet that symbolic names were much easier for everyone to work with than numeric addresses. They began to assign simple host names to each of the devices on the network. Each site managed its own *host table*, which listed the mappings of names to addresses.

Naturally, the ARPAnet engineers immediately recognized the dangers of having each site maintain a list of possibly inconsistent host names. Since the internetwork was just a small "club" at this point, they used the Request for Comment (RFC) process itself to document standard host-name-to-address mappings. RFC 226, "Standardization of Host Mnemonics," is the first RFC I could find showing how host names were assigned. It was published on September 20, 1971.

This initial name system was about as manual as a system could be. As additions and changes were made to the network, the list of host names was updated in a new RFC, leading to a series of RFCs being published in the 1970s. Each host administrator still maintained his or her own host table, which was updated when a new RFC was published. During this time, the structure of host names was still under discussion, and changes were made to just about every aspect of the name system as new ideas were explored and refined.

### ***Storing Host Names in a Host Table File***

This early name system worked fine while the ARPAnet was very small, but it presented many problems as the internetwork grew. One problem was that it was extremely slow in responding to network modifications, because additions and

changes would be entered into device tables only after a new list was published. Also, even with the centralized list, there were still potential consistency issues, because a site manager might forget to update a file or make a typographical error.

The first improvement was to make the list of host name assignments a standard “master” text file, which was centrally managed and could be downloaded using network protocols like the File Transfer Protocol (FTP). The file was maintained at the Network Information Center (NIC) at Stanford University. The process for defining and using this file was described in RFCs 606 and 608, both entitled “Host Names On-Line,” published in December 1973 and January 1974, respectively. These documents also formally specified the syntax for the TCP/IP host table name system, described later in this chapter.

The use of a centrally managed host table continued through the 1970s. When TCP/IP was developed, the system was maintained, and the mappings were made between host names and 32-bit IP addresses. RFC 810, “DoD Internet Host Table Specification,” shows how host tables were defined for use with IP addresses. It was published in March 1982.

### ***Outgrowing the Host Table Name System and Moving to DNS***

The continuing growth of the ARPAnet/Internet made it apparent that the simple host table name system would eventually become unmanageable. With at first dozens, and then hundreds and thousands of new hosts connecting to the internetwork, a single text file maintained in a central location just wasn’t up to the task.

The idea of moving to a hierarchical name system based on the concept of *domains* was first introduced in September 1981 in RFC 799, “Internet Name Domains.” Considerable discussion and development of this concept occurred in the early 1980s. By 1983, a plan was put in place to migrate from the flat host table name system to the new Domain Name System (DNS). The detailed history of the development of this name system is continued in the overview of DNS in Chapter 52.

## **The TCP/IP Host Table Name System**

The pioneers of the modern Internet made the first name system for the TCP/IP suite when they created simple files containing the names and addresses of the machines in the early ARPAnet, as explained in the preceding section. This system was so simple that it originally wasn’t even formally specified as a name system per se. Since the files contained names for network hosts, the process for relating names to addresses was simply called the *host name* mechanism. Later, these files were called *host tables*, and for this reason, this technology is commonly called the *TCP/IP host table name system*.

As a system, it is extremely simple, since it consists of nothing more than a text file maintained on each machine on the network. This file is normally called /etc/hosts on a UNIX system and HOSTS on a Windows system (usually residing in the main Windows directory). The file usually begins with some comment lines and then lists pairs of IP addresses and host names. A very simplified example (using the modern table structure, which is slightly different from the original host table format) is shown in Listing 51-1.

---

```
# Host Database
# This file should contain the addresses and aliases
# for local hosts that share this file.
#
# Each line should take the form:
# <address>          <host name>
#
127.0.0.1           localhost
209.68.14.80         www.pcguide.com
216.92.177.143       www.desktopscenes.com
198.175.98.64        ftp.intel.com
```

---

**Listing 51-1:** Example TCP/IP host table

The name space and architecture for the host table name system is theoretically flat, with each name being able to take any form, without any real structure. Despite this, for consistency, certain rules were eventually put in place regarding how names should be created, as discussed in Chapter 53. As you will learn later in this chapter, it's also possible to use host tables to support the implementation of a hierarchical name space, which would mean that the names would need to be created using that name space's structural rules.

### **Host Table Name Resolution**

Name resolution in the host table name system is very simple. Each device reads the host table into memory when it starts up. Users of the system can refer to the names in that host table by using names, instead of a numeric IP addresses, in their invocation of various applications. When the software detects a name has been used in this manner, it refers the name to the internal resolver routine in the device, which looks up the name in the host table in memory and returns its address. There is no need for any transmissions or servers to be contacted; resolution is entirely local.

### **Host Table Name Registration**

Now, here is the part where I am supposed to say that name registration in the host table name system is simple as well, right? Well, yes and no. From a purely technical standpoint, it certainly is simple. A name is registered on a particular device when the name and corresponding IP address are entered into the device's host table, and that's it.

However, name registration is much more complicated from an administrative standpoint, and this is where we find the major weakness of using host tables. Each network device maintains its own host table independent of the others, usually stored as a file on its local hard disk. This is in contrast to database registration systems (see Chapter 50), where the data is centrally stored and managed. This approach to name registration leads to two very important concerns:

**Consistency** Since every device has its own host table, how do we ensure that information is consistent throughout all the tables on the different devices?

**Modifications** How do we ensure that information about new device mappings and changes to existing ones are propagated to all devices?

As explained earlier in this chapter, the original mechanism for name registration was simply hand-editing, with administrators consulting updated published lists of device names. This was a very inefficient method that was prone to error and slow to acknowledge changes to the network. The revised system used a centrally managed master file that was downloaded by all sites on a regular basis. Name registration in this method required that the name/address mapping be submitted to the authority managing the central file, the NIC.

### ***Weaknesses of the Host Table Name System***

The use of a centralized master file for name registration certainly worked better than using the equivalent of interoffice memos to publish host name lists, but it was practical only in the early days of TCP/IP. As the internetwork grew, more weaknesses of the host table system became apparent:

**Central Administration Overload** The changes to the central file became more frequent, increasing the administrative load on the individual managing the master file, to the point where changes were being made many times per day. As the Internet continued to grow, it would eventually have become impossible for human beings to enter the changes as fast as they were being submitted.

**Growth in the Master File Size** Every host needed a line in the master file. When the Internet grew to be thousands and eventually millions of devices, the file size would have become excessive.

**Excessive Bandwidth Use** Since the master file was changing so often, this also meant that all the devices on the network needed to keep downloading this master file repeatedly to stay current. At the same time, the file was also growing in size. Frequent downloads of a big file meant large amounts of network bandwidth were being consumed on something that was, in essence, an overhead activity.

**Flat Name Space Problems** The lack of a hierarchical name space led to conflicts when users chose identical names for their devices, and this further increased the workload on the central administrator. These issues were ameliorated in part by using naming conventions, such as using a prefix with a location before each individual machine name (like the example we saw in Chapter 50), but this was not an ideal solution.

All of these are reasons why the designers of the Internet eventually moved away from using host tables for the entire Internet to the more capable DNS.

**KEY CONCEPT** The *host table name system* was the original mechanism used for implementing names on the early Internet. It consists simply of a set of tables containing mappings between names and addresses maintained on each machine in the internetwork. When a name needs to be resolved, the table is consulted to determine the appropriate address. This system is extremely simple, but not very capable and not well suited to a large global Internet, which is why it was eventually abandoned in favor of DNS.

## **Use of the Host Table Name System in Modern Networking**

Although the host table name system has critical weaknesses, it has not gone away entirely. There are two circumstances in which this technique is still of value, as explained in this section.

### **Small “Island” Networks**

If you are setting up a small local area network (LAN) using TCP/IP, and you don’t need the names of your devices to be accessible by those outside your network, then guess what: You have the equivalent, of sorts, of the early Internet. In that case, the host table system is as applicable to you as it was to the Internet in the 1970s. You can simply set up host tables on each device and manage them manually.

As long as the LAN is small enough that editing these files periodically is not a hassle, this is actually a fast and effective name system, because no exchange of messages is needed for resolution. You can even maintain a master file on one machine and copy it to the others when changes are required using a script, to save time.

### **Local Name Mappings to Supplement DNS**

Even though modern systems use DNS for most name resolution, they also usually still support the use of host table files. You can manually enter common name mappings into this file, even for devices that are on the global Internet. Your system can then be set up to consult this list before making use of its assigned DNS server.

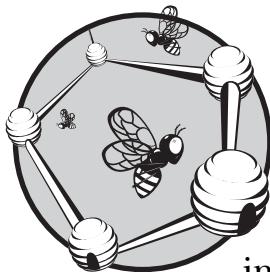
The use of the HOSTS file in conjunction with DNS allows you to manually specify mappings for commonly accessed sites, which may provide a slight performance improvement since there is no need to access a server. Since the HOSTS file doesn’t enforce any particular structure to names, it is naturally quite possible to put DNS-style hierarchical names into the file, as I showed in Listing 51-1. The file is loaded into memory and used to *override* the normal DNS process for names listed in it.

Of course, you then subject yourself to all the potential maintenance headaches of manually edited files. You must update these files as host names or addresses are changed in the DNS system. For this reason, this second use of the HOSTS file for Internet sites served by DNS is less popular than the use of the file for local machines.

**KEY CONCEPT** Even though the host table name system is not the primary mechanism used for TCP/IP naming, it is still used in two circumstances. The first is to implement a basic name system in a small local TCP/IP internetwork. The second is as an adjunct to DNS, where it allows manual mappings to be created that override the DNS process when needed.

# 52

## DOMAIN NAME SYSTEM (DNS) OVERVIEW, FUNCTIONS, AND CHARACTERISTICS



The creation of host tables to map computer names to addresses greatly improved the usability of the early Internet and the TCP/IP protocol suite that implemented it. Unfortunately, while the host table name system worked well when the internetwork was small, it did not scale particularly well as the Internet started to grow in size and complexity. The name system had to stay, but the use of host tables had to be dispensed with in favor of a newer, more capable system.

Over the period of several years, many engineers worked to create a system that would meet not just the needs of TCP/IP internetworks of the time, but also those of the future. The new name system was based on a hierarchical division of the network into groups and subgroups, with names reflecting this structure. It was designed to store data in a distributed fashion to facilitate decentralized control and efficient operation, and included flexible and extensible mechanisms for name registration and resolution. This new name system for TCP/IP was called the *Domain Name System (DNS)*.

We'll begin our look at DNS in this introductory chapter. I start by providing an overview of DNS's development, history, and standards, continuing the history begun in the overall look at TCP/IP name systems. I discuss the design goals and objectives of the creators of DNS, to help you understand better what its designers were trying to do. I then talk about the main components of DNS and the functions it performs, relating these to the basic functions explained in the overview section on name systems.

## DNS Overview, History, and Standards

The aversion that most people have to trying to remember numeric identifiers led to the very quick adoption of a name system for devices on the predecessors of what we now call the Internet. In the 1960s and early 1970s, names were given to machines, and these names were maintained in host tables. The TCP/IP host table name system (described in Chapter 51) worked well for a number of years, with a centrally maintained master list used by device administrators to ensure a consistent view of the network.

Unfortunately, such a system works well only when the number of devices is small. As the budding Internet grew, numerous weaknesses became apparent in the host table method, as I detailed in Chapter 51. Furthermore, the problems with the system weren't something that could be easily patched with small changes; the problems were structural, part of the basic idea of host tables as a whole. A completely new approach was needed for how names would be used on the Internet.

### ***Early DNS Development and the Move to Hierarchical Domains***

The most important paradigm shift made by the TCP/IP engineers was the decision to change the name system from one that used a single, centralized list of names to a more decentralized system. The idea was to create a structured topology where names were organized into *domains*. This idea was first introduced in RFC 799, "Internet Name Domains," published in September 1981.

RFC 799 actually describes more the mechanics of delivering electronic mail messages between domains than the domains themselves. Interestingly, the standard assumes a flat structure of domains in its discussion, while mentioning the possibility of creating a hierarchical structure instead. It was the decision to go to such a hierarchical name space for domains that led to the creation of DNS in the form in which we know it today.

Many RFC documents describing the development of different aspects of DNS were published in the early 1980s. The first real milestone in DNS's history was probably the publishing, in November 1983, of three initial documents discussing DNS concepts:

- RFC 881, "Domain Names Plan and Schedule," discusses the issues involved in implementing the new DNS and how to migrate from the older host table system.

- RFC 882, “Domain Names: Concepts and Facilities,” describes the concepts and functional elements of DNS in fairly extensive detail. It includes a discussion of the name space, resource records, and how name servers and resolvers work.
- RFC 883, “Domain Names: Implementation Specification,” provides the nitty-gritty details on DNS messaging and operation.

### ***Standardization of DNS and Initial Defining Standards***

The three “Domain Names” RFC documents published in November 1983 were discussed frequently over the months that followed, and the basic DNS mechanism was revised many times. Several subsequent RFCs were published, updating the DNS transition plan and schedule. Finally, in November 1987, agreement on the operation of the system was finalized, and four new RFCs were published that formalized the DNS system for the first time:

- RFC 1032, “Domain Administrators Guide,” specifies administrative procedures and policies for those running a domain.
- RFC 1033, “Domain Administrators Operations Guide,” provides technical details on how to operate a DNS server, including how to maintain portions of the DNS distributed database of names.
- RFC 1034, “Domain Names - Concepts and Facilities,” replaces RFC 882, providing an introduction and conceptual description of DNS.
- RFC 1035, “Domain Names - Implementation and Specification,” is an update to RFC 883, specifying how DNS works in detail, including resource record definitions, message types, master file format, and resolver and name server implementation details.

These last two documents, RFCs 1034 and 1035, are considered the definitive original specification for the operation of DNS. While they are now many years old, they still provide the essential description of how DNS works.

As the Internet has grown to include thousands and then millions of sites, the importance of DNS has grown as well. Today, most people use DNS almost every time they use TCP/IP to access the Internet. It has gone from an alternative form of addressing for applications to one that is preferred by most users. It is also an important building block of the more complete application layer addressing scheme developed for TCP/IP: Uniform Resource Identifiers (URIs) (described in Chapter 70).

The hierarchical nature of the DNS name space has allowed the Internet to grow by making the assignment and mapping of names manageable. The authority structure (which defines who is in charge of parts of the name space) is also hierarchical, giving local administrators control over the names of devices they manage, while ensuring name consistency across the hierarchy as a whole. The distribution of data using many name servers and a standardized resolution technique following a standard message protocol provides efficiency and reliability. These concepts will become clearer as we explore DNS more completely in later sections of this chapter.

## **DNS Evolution and Important Additional Standards**

TCP/IP and the Internet have both changed a lot since 1987, of course, and DNS has also had to change. Many RFCs have been written since the base documents were published in the late 1980s, most of which further clarify the operation of DNS, expand on its capabilities, or define new features for it. You can find all of these by searching for “domain” or “DNS” in a list of RFCs. There are dozens of these. The following are a few of the more interesting ones:

- RFC 1183, “New DNS RR Definitions,” defines several new experimental resource record types. Other subsequent RFCs have also defined new resource records.
- RFC 1794, “DNS Support for Load Balancing,” discusses load balancing for greater performance in DNS servers.
- RFC 1995, “Incremental Zone Transfer in DNS,” specifies a new feature that allows only part of a zone to be transferred to a secondary name server for efficiency.
- RFC 1996, “A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY),” adds a new message type to DNS to allow primary (authoritative) DNS servers to tell secondary servers that information has changed in the main database.
- RFC 2136, “Dynamic Updates in the Domain Name System (DNS UPDATE),” describes a technique for dynamically making resource record changes in the DNS database (also called *Dynamic DNS*).
- RFC 2181, “Clarifications to the DNS Specification,” discusses several issues with the main DNS standards as defined in RFCs 1034 and 1035 and how to address them.
- RFC 2308, “Negative Caching of DNS Queries (DNS NCACHE),” specifies the operation of negative caching, a feature that allows a server to maintain information about names that do not exist more efficiently.

## **DNS Adaptation for Internet Protocol Version 6**

Version 6 of the Internet Protocol (IPv6, covered in Part II-4) was developed starting in the mid-1990s and brought with it the need to make changes and enhancements to the operation of DNS. (Even though DNS operates at the higher layers, it deals intimately with addresses, and addresses have changed in IPv6, as discussed in Chapter 25.) The modifications required to allow DNS to support IPv6 were first defined in RFC 1886, “IPv6 DNS Extensions,” which was part of a group of RFCs that laid out the fundamentals of IPv6. Several subsequent standards have been published since that time; these are discussed in the section on IPv6 DNS near the end of Chapter 57.

The rest of this chapter provides a more complete overview of DNS and its development, by discussing the design goals of its creators and the protocol’s key characteristics.

## DNS Design Goals, Objectives, and Assumptions

As we just saw, the elapsed time from the first RFC discussing TCP/IP domain names to the publishing of the official standards describing the operation of DNS was more than six years. This is a very long time for the development of a system, but it isn't surprising. A lot of thought had to go into the creation of DNS, to be certain that it would meet all of the many demands that would be placed on it.

The first problem was that the creators of DNS needed to worry about both how to define the new system and how to migrate from the old one. Considerable time was spent figuring out how all the existing hosts would be moved over to the new DNS name space and how the new protocols for exchanging DNS information would be implemented on them.

The creators of DNS knew they were making the new system because the old one didn't scale very well. They also knew that if migration was a difficult problem with the small number of hosts in existence at that time, it would be much more difficult if they needed to go to another new system in the future. This made the key challenge in DNS to create a system that would meet the needs of the Internet not just the day it was introduced, or the following year, but even ten years or more down the road.

### ***DNS Design Goals and Objectives***

Back in the 1980s, no one had any idea how the Internet would grow as it has in the last decade. That DNS still works as well as it does is a testament to the skill of its designers. Much of this success is due to the early groundwork put into the design of the system. DNS engineers documented some of what they considered to be the main design goals in creating it, which can help us understand not just what DNS does, but also why. These design goals and objectives are as follows:

**Creation of a Global, Scalable, Consistent Name Space** The name space needed to be capable of spanning a large, global internetwork containing millions of machines. It was necessary that it provide a consistent and predictable method for naming devices and resources, so they could be easily found. It was also, obviously, essential that name duplication be avoided, even when conflicts could potentially be between devices on different continents.

**Local Control over Local Resources** Administrators of networks and small internetworks on the Internet as a whole needed to be able to have control over the naming of their own devices. It would not be acceptable to need to go through a central authority for naming every single object, nor would it be acceptable for every administrator to need to know the names of everyone else's networks and machines.

**Distributed Design to Avoid Bottlenecks** The designers of DNS knew that they would need to abandon the idea of a centralized database in favor of a distributed approach to data storage, to avoid the bottlenecks that would result in using DNS with many devices.

**Application Universality** The system needed to be general enough that it would support a wide variety of applications. For example, it needed to support host identification, mail delivery, and other functions.

**Multiple Underlying Protocol Support** DNS needed to be inherently able to support different underlying protocols. Many people don't realize, for example, that DNS can support not just IP addresses, but other types of addresses, simply because IP is so dominant in networking today.

**Hardware Universality** Both large and small computers needed to be able to use the system.

Keep these objectives in mind as you learn more about DNS, and they will help you understand better why certain design attributes were chosen. For example, if we consider the first two objectives listed, they seem almost contradictory: How can we have a global name space with unique names if individual administrators were able to assign local names? As you will see, this is where the power of the DNS hierarchical name space shines through.

### **DNS Design Assumptions**

The design goals tell us what DNS's creators wanted to make sure the new system addressed. In addition, the engineers that worked on the protocol's implementation details needed to make decisions based on certain assumptions of how it would be used:

**Rapidly Growing Database Size** By the mid-1980s, it was obvious that the DNS database of names would start out rather small but would grow quickly. The system needed to be capable of handling this rapid growth.

**Variable Data Modification Rate** Most of the data in the name database would change only infrequently, but some data would change more often than that. This meant flexibility would be required in how data changes were handled and how information about those changes was communicated.

**Delegatable Organizational Responsibility** Responsibility for portions of the name database would be delegated primarily on the basis of organizational boundaries. Many organizations would also run their own hardware and software to implement portions of the overall system.

**Relative Importance of Name Information Access** It was assumed that the most important thing about DNS was providing reliable name resolution, so the system was created so that it was always possible for a user to access a name and determine its address. A key decision in creating the system was deciding that even if the information were slightly out of date, it was better than no information at all. If a name server were unable to provide the latest data to fill a request, it would return the best information it had available.

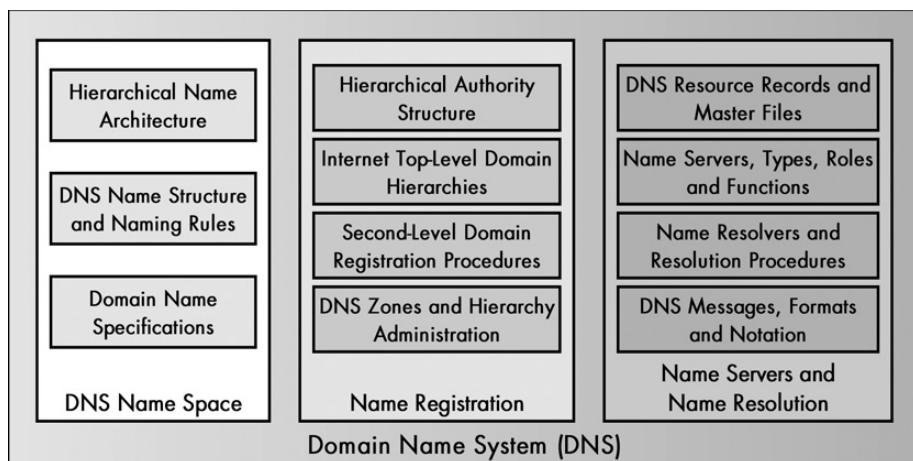
**Handling of Requests for Missing Information** Since the name data was to be distributed, a particular name server might not have the information requested by a user. In this case, the name server should not just say, “I don’t know.” It should provide a referral to a more likely source of the information or take care of finding the data by issuing its own requests. This led to the creation of the several DNS name resolution techniques: local, iterative, and recursive.

**Use of Caching for Performance** From the start, it was assumed that DNS would make extensive use of caching to avoid unnecessary queries to servers containing parts of the distributed name database.

Arguably, a lot more assumptions were made in creating this system, as is the case in the development of every system. For example, DNS needed to make assumptions about how exactly data would be stored, the transport mechanism for sending messages, the role of administrators, and so on. You’ll learn more about these as we go through our look at the system.

## DNS Components and General Functions

To meet the many objectives set for it by its designers, DNS requires a great deal of functionality. It is a true name system with the emphasis on *system*, and as such, is considerably more complex than the host table name system used earlier in TCP/IP. In Chapter 50, I divided the many tasks of a full-featured name system into three categories. DNS includes functions in all of these categories, and so using these categories is a good way to take a high-level look at the way DNS works (see Figure 52-1).



**Figure 52-1: DNS functions** DNS consists of three main functional categories: name space, name registration, and name servers/resolution. Each of these consists of a number of specific tasks and responsibilities.

## **DNS Name Space**

DNS uses a hierarchical name space consisting of a single, complex, multiple-level structure into which all names in the system fit. The name space is organized starting from a single root into which containers (called *domains*) are placed. Each can contain either individual device names or more specific subcontainers. The overall structure is somewhat analogous to how a directory system on a computer organizes files from general to specific, using an arbitrary structure that can be optimized to various needs. A specific syntax is used to define valid names, and special terminology is used to describe parts of the structure and identify domain names, from the root down to the device level.

### **Name Registration (Including Administration and Authorities)**

DNS name registration is used to enter individual names into the DNS distributed database. DNS uses a hierarchical arrangement of authorities that complements the hierarchical name space. A centralized authority determines the overall shape and structure of the name space and handles registration of names at the highest level. Authority is then *delegated* to different organizations to manage various parts of the name space. A set of universal policies controls the registration process and deals with problems and conflicts.

### **Name Resolution**

DNS uses a powerful, distributed, client/server name resolution mechanism. This is probably the area where the most attention needed to be put into the design of DNS, to ensure that it could scale to handle millions and eventually billions of name resolution requests each day.

The name resolution process is implemented using two basic software elements that play the role of server and client: name servers and name resolvers.

DNS name servers are special programs running on hardware servers that are the heart of DNS. Servers are maintained by organizations that have administrative control over part of the DNS name space. They contain *resource records* that describe names, addresses, and other characteristics of those portions of the name space. As such, the servers themselves are arranged into a hierarchy analogous to that of the name space, although not identical in structure.

The main job of name servers is to receive requests for name resolution and respond with either the data requested from the database or with the name of another name server that will lead to the requested information. Name servers are also responsible for data caching and other administrative tasks to ensure efficient operation of the system as a whole.

Name resolvers are the usual clients in the name resolution process. When a user makes reference to a name in a networking application, the name is passed to the resolver, which issues a request to a name server. Depending on the configuration, more than one request may be needed, and several different resolution processes may be combined to find the needed information. Resolvers also may employ caching or implement other features.

**NOTE** *The division between resolvers and servers is based on roles. As you'll see when we look at name resolution, name servers may also function as clients in certain exchanges of data. See Chapter 56 for an explanation of this apparent paradox.*

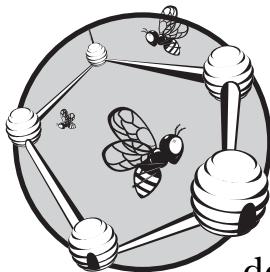
If this seems a lot like the classic description of a name system that I gave in Chapter 50, that's not a coincidence. DNS is considered *the* name system against which most others are usually compared. If you understand these high-level descriptions, then you already know the basics of how DNS works. The next three chapters delve into each of these three functional areas in more detail and will help you really learn how DNS does its thing.

**KEY CONCEPT** As a complete name system, DNS provides numerous capabilities that implement each of the three basic name system functions. The DNS *name space* is hierarchical and is organized using a multilevel structure with particular naming rules. The DNS *name registration system* is based on the idea of a hierarchy of domains and registration authorities responsible for them. DNS *name resolution* is similarly hierarchical, and it is designed around interaction between *name resolver* and *name server* software components that consult databases of DNS *resource records* and communicate using a special messaging protocol to answer client queries.



# 53

## DNS NAME SPACE, ARCHITECTURE, AND TERMINOLOGY



The name space is the most fundamental part of any name system, since it is what defines the ways that the names themselves are created. The name space tells us what form names may take and provides the rules for how they are created. Most important, it specifies the *architecture* of the names—the internal structure of names themselves. This, in turn, has a critical influence on how name registration and resolution work, making an examination of name space and architecture issues the obvious place to start in learning the details of the Domain Name System (DNS).

In this chapter, I describe the concepts behind the DNS name space and its structure. I begin with an overview of the DNS name space and description of the hierarchical architecture it uses. I then explain the terminology often used to refer to parts of the name space. Next, I provide a formal description of DNS labels and the official and unofficial rules for creating domain names. I conclude with a description of domain name specifications, and I explain the concept of qualification and how fully qualified and partially qualified names differ.

## DNS Domains and the DNS Hierarchical Name Architecture

The most important element of a name system's name space is its *name architecture*, which describes how names are constructed and interpreted. The architecture of DNS is, unsurprisingly, based on the concept of an abstraction called a *domain*. This is obviously a good place to start in explaining how DNS works.

### ***The Essential Concept in the DNS Name Space: Domains***

Dictionary definitions of the word *domain* generally convey the notion of a sphere of influence or an area of control or rulership. An essential concept is that in various contexts, control or authority can be exerted at many different levels. One sphere of influence may contain smaller ones, which can, in turn, contain still smaller ones. This means that such domains are naturally arranged in a hierarchy.

As an example, consider geopolitical domains. We have no centralized “world government” on earth, but we do have the United Nations, which deals with worldwide issues. At the next level down, we have individual countries. Some of these countries have divisions such as states and provinces. Still lower levels have counties, municipalities, neighborhoods, and individual residences or businesses. The “domains” are inherently hierarchical in organization.

DNS uses the word *domain* in a manner very similar to this, and it employs a hierarchical structure that works in much the same way as the geopolitical example. In DNS, a *domain* is defined as either a single object or a set of objects that have been collected together based on some type of commonality. Usually, in DNS, that commonality is that they are all administered by the same organization or authority, which makes the name hierarchy tightly linked to the notion of the DNS hierarchical authority structure (see Chapter 54).

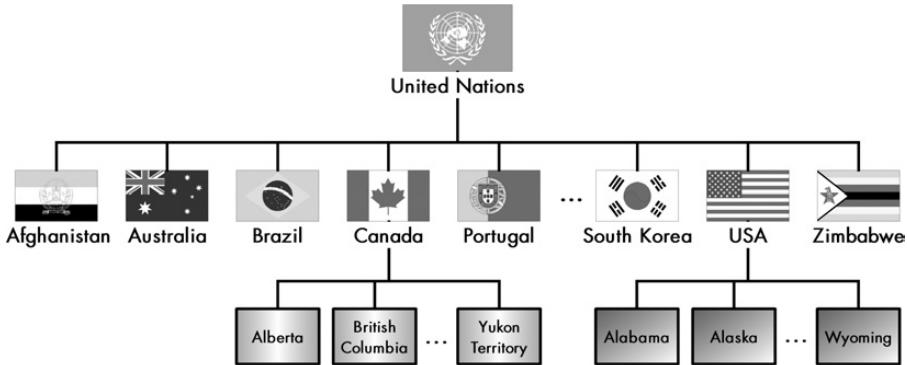
**NOTE** *The term domain is also used in other contexts in the world of networking. The most notable example of this is in Microsoft networking, where domain is also used to represent the notion of a collection of objects under common authority. However, the two types of domains are completely different and not related beyond this conceptual level.*

### ***The DNS Hierarchical Tree Structure of Names***

We could construct a tree diagram with the United Nations on top, with lines pointing to each of the countries in the world. Then, within the United States, we could draw lines to each of the states. Within each state, we could draw lines to each county, and so on. The result would be something that looks like an upside-down tree, as illustrated in Figure 53-1. This is called a *tree structure*.

Tree structures are common in computing and networking. For example, trees are a type of topology used to connect networks into a local area network.

For understanding DNS, the best example of a tree structure is the directory tree used to store files on a computer's hard disk. The root directory is at the top of the structure and may contain named files and/or named directories. Each directory can itself contain individual files or subdirectories, which can, in turn, contain their own subdirectories, and so on. The domain name structure in DNS is conceptually arranged in the same way, but instead of dealing with files, DNS deals with named objects, usually devices like Internet Protocol (IP) hosts.



**Figure 53-1: Example of a global hierarchical domain architecture** This diagram shows an example of hierarchical architecture, based on political divisions. The United Nations is an umbrella organization representing (to one extent or another) all of the world's nations. It is the root of the tree; underneath it we find individual nations. Each nation then is further subdivided in a manner it chooses. For example, Canada has provinces and territories, and the United States has individual states. These can be further subdivided in any number of ways.

The highest level is still the *root* of the tree. It contains a number of domains, each of which can contain individual objects (names) and/or lower-level domains. Lower-level domains can, in turn, have still lower-level domains, allowing the tree as a whole to take on an arbitrary structure.

Like a directory structure, the DNS hierarchical name architecture allows names to be organized from most general to most specific. It also has complete flexibility, allowing us to arrange the structure in any way that we want. For example, we could make a name system that is structured exactly paralleling the geopolitical organization chart shown in Figure 53-1. We could have the root of the name structure represent the United Nations and create a domain for each country. Then, for those countries that have states, we could create state domains within those country domains. Smaller countries not needing those domains could have city domains directly under the country domain. The hierarchy is flexible, because at each level, it can be given a suitable substructure.

**KEY CONCEPT** The DNS name space is arranged into a *hierarchy of domains* shaped like an inverted tree. It is structurally similar to the directory structure of a file system, with a root that contains domains, each of which can contain subdomains and so forth.

It's important to remember that every stand-alone internetwork can have its own name space and unique hierarchical structure. Many times, people conflate the idea of *a* DNS name space with *the* DNS name space. The latter refers to the DNS hierarchy used for the global Internet, and it's obvious that this deserves a great deal of attention. But it is just one possible arrangement, although an important one, of an infinite number of possible structures.

**NOTE** Chapter 54 provides more specific information about the Internet's DNS hierarchy. As you'll see, geopolitical structures are, in fact, used to assign names to some of the Internet's computers, but other parts of the hierarchy are different.

## DNS Structural Elements and Terminology

Now that we've reviewed the fundamentals of the DNS name space, let's look at its structure in more detail. At the same time, I'll define the many different terms used to refer to parts of the DNS domain name hierarchy.

### DNS Tree-Related Terminology

As I explained in the previous section, the DNS name structure is shaped somewhat like a tree. The comparison between structured elements and trees is a common one in networking. The main difference between technology and biology is that DNS trees grow from the top down, instead of reaching for the sky. The analogy to a tree naturally leads to the use of several tree-related terms in describing the DNS name structure, some of which are illustrated in Figure 53-2:

**Root** This is the conceptual top of the DNS name structure. The *root* domain in DNS contains the entire structure. By definition, it has no name; it is *null*.

**Branch** A *branch* is any contiguous portion of DNS hierarchy. It consists of a domain and all the domains and objects within it. All branches connect together to the root, just as in a real tree. (Yes, it would be better if the root were called the *trunk*, but computer science majors apparently don't take botany electives.)

**Leaf** This is an end object in the structure; that is, a domain that doesn't have anything underneath it. The analogy to a leaf being at the end of a sequence of branches is apt.

There is no specific term to refer to a domain that is not a leaf. These are sometimes called *interior nodes*, meaning that they are in the middle of the structure. A *node* is the generic computing term for an object in a topology or structure. So, in DNS, every node is a domain, and it may be an interior node that contains additional domains and/or objects or a leaf that is a specific, named device. The term *domain* is thus somewhat ambiguous, as it can refer to either a collection of objects that represents a branch of the tree or to a specific leaf.

### DNS Domain-Related Terminology

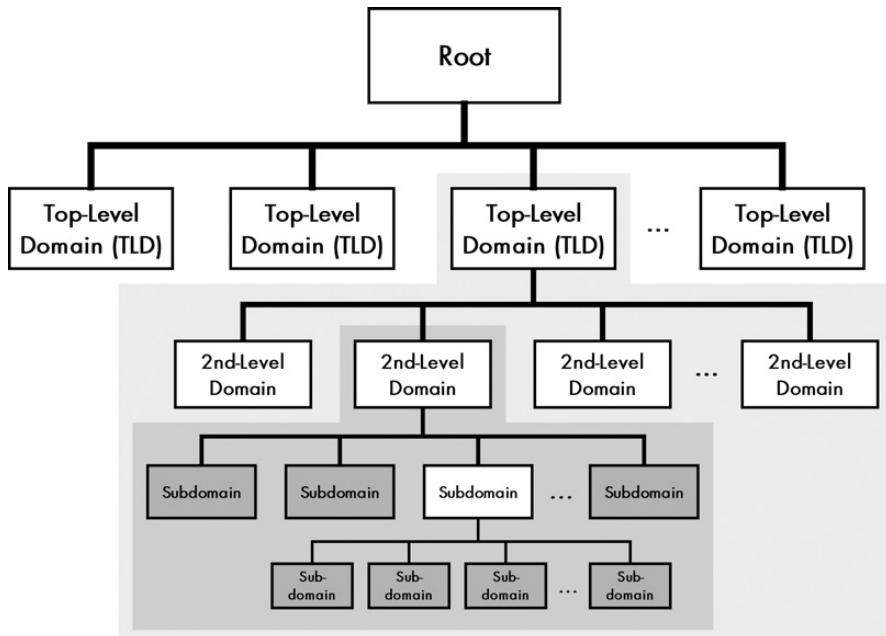
There are also several domain-like terms that are often used to refer to domains at different levels of the hierarchy. These terms are also shown in Figure 53-2:

**Root Domain** This is the root of the tree.

**Top-Level Domains (TLDs)** These are the highest-level domains directly under the root of the tree. They are also sometimes called *first-level domains*.

**Second-Level Domains** Shockingly enough, these are the domains located directly below the top-level domains.

**Subdomains** In some contexts, this term refers only to domains that are located directly below the second-level domains.



**Figure 53-2: DNS tree-related and domain-related terminology** The top of the DNS name space is the root of the tree, and it has no name. Under the root comes any number of top-level domains (TLDs). Within each of these can be placed second-level domains, then within those subdomains, and so forth. Some of the tree terminology used in DNS is also shown here. The portion of the tree with the light shading is one branch; the darker area highlights a smaller subbranch within that branch. The darkest nodes within that area are the leaves of that smaller branch of the tree.

**KEY CONCEPT** The top of the DNS name space is the root. Under the root come *top-level domains*, and within these are *second-level domains* and then *subdomains*. In theory, any number of levels of subdomains can be created. A *branch* is any contiguous portion of the DNS tree. A *leaf* is a domain with nothing underneath it in the structure, and it usually represents a single device.

The term *subdomain* can also be used generically, like the word *domain* itself. In that case, it refers simply to the relationship between two domains, with a subdomain being under another domain in the structure. This means, for example, that top-level domains can be said to be subdomains of the root; every second-level domain is a subdomain of a top-level domain, and so on. But again, sometimes *subdomain* means specifically a third-level or lower domain.

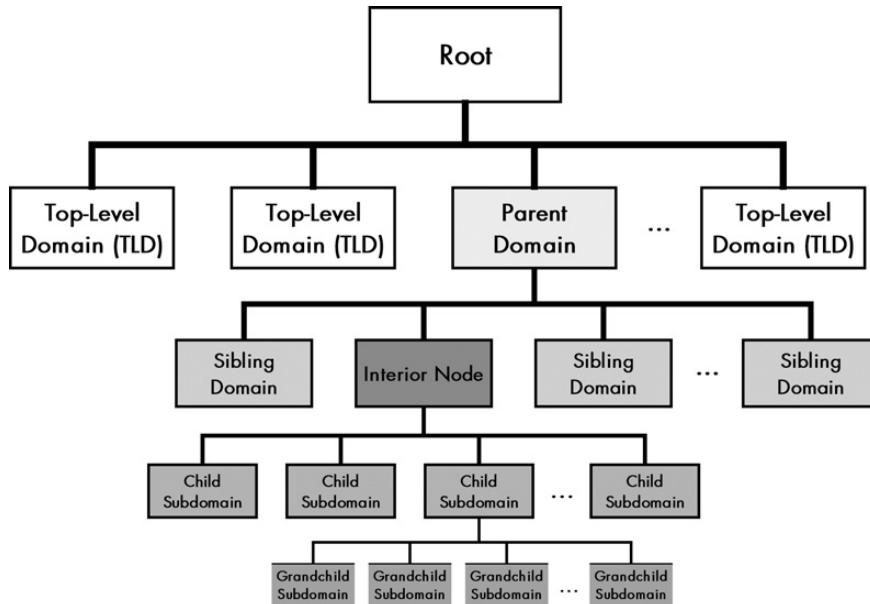
## DNS Family-Related Terminology

Another set of terminology related to DNS compares the tree structure not to a living tree, but to another analogy: a family tree. These terms are most often used to describe how a particular domain relates to the other domains or subdomains around it, so they are relative terms. The following family-related terms are common (see Figure 53-3).

**Parent Domain** This is the domain that is above this one in the hierarchy. For example, the root domain is the parent of all top-level domains.

**Child** This is a domain at the next level down from this one in the hierarchy. Thus, the top-level domains are *children* of the root.

**Sibling** This is a peer at the same level as this one in the hierarchy, with the same parent. Thus, all top-level domains are *siblings* with the root as a parent; all second-level domains within a particular top-level domain are siblings, and so on.



**Figure 53-3: DNS name space “family tree”** This diagram is similar to Figure 53-2, but the nodes are labeled to show the family-oriented terminology sometimes used in DNS. In this case, the names are relative to the interior node shown in the darker shade. The domain immediately above it is its parent node. Other nodes on the same level are siblings, and subdomains within it are children of that node.

**KEY CONCEPT** The domain above a given domain in the DNS name space is called its *parent domain*. Domains at the same level within the same parent are *siblings*. Subdomains are called *children* of that domain.

Like a real tree, the DNS name structure must be a true tree in its structure. Every domain can have only one parent (except the root), just as every branch of a tree connects to only one limb (except the root/trunk). Also, no loops can appear in the structure; you cannot have a domain whose child is also its parent, for example.

**KEY CONCEPT** A DNS name space must be arranged as a true topological tree. This means each domain can have only one parent, and no loops are permitted in the structure.

Keep in mind that even though the name hierarchy represents an arrangement of named devices, it is only a logical structure. There is no necessary correspondence to the physical location of devices. A domain with 10 children may represent 11 devices in 11 different countries. We'll explore this more when we look at DNS authority structures in the next chapter.

## DNS Labels, Names, and Syntax Rules

We've seen how the DNS name space hierarchy allows us to arrange domains into a virtual tree that reflects the characteristics of how the devices themselves are organized. While using a hierarchical name space is inherently more complex than a flat name space, it yields a powerful result: the ability to specify names that can be locally managed while remaining globally unique. At the same time, the complexity of the tree yields the benefit of relatively simple name construction using domain identifiers.

### DNS Labels and Label Syntax Rules

Naming in DNS begins with giving each domain, or node, in the DNS name space a text *label*. The label identifies the domain within the structure and must follow several syntax rules:

**Length** Each label can theoretically be from 0 to 63 characters in length. In practice, a length of 1 to about 20 characters is most common, with a special exception for the label assigned to the root of the tree (which is 0 characters in length, as explained in the next section).

**Symbols** Letters and numbers are allowed, as well as the dash character (-). No other punctuation is permitted. For example, an underscore (\_) cannot be used in a label.

**Case** Labels are not case-sensitive. For example, *Jabberwocky* and *jabberwocky* are equivalent domain name labels.

Every label must be unique within its parent domain. So, for example, if we have a top-level domain called Rocks, we can have only one subdomain within Rocks called Crystal. Due to the case-insensitivity of labels, we cannot have both CRYSTAL and Crystal within Rocks, because they are considered the same.

It is this concept of *local uniqueness* within a parent domain that ensures the uniqueness of names as a whole, while allowing local control over naming. Whoever is in charge of the Rocks domain can assign names to as many individual objects or subdomains as he likes, as long as those names are unique within the domain. Someone else, say, the maintainer of the Glass domain, can also create a subdomain called Crystal within Glass. There will not be a conflict, because the Glass and Rocks domains are separate. Of course, since all top-level domains have the same parent (the root), all top-level domains must be unique.

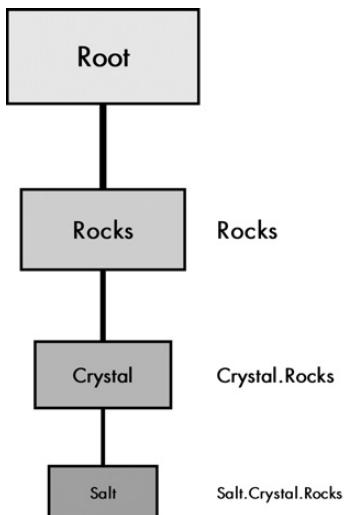
**KEY CONCEPT** Each node in the DNS name space is identified by a *label*. Each label must be unique within a parent domain, but it does not need to be unique across domains. This enables each domain to have local control over the names of subdomains, without causing any conflicts in the full domain names created on a global level.

## Domain Name Construction

Each individual domain within the domain name structure can be uniquely identified using the sequence of labels that starts from the root of the tree and progresses down to that domain. The labels at each level in the hierarchy are listed in sequence, starting with the highest level, from right to left, separated by dots. The result is the formal definition of a *domain name*.

The root of the name space is given a zero-length, null name by default; that is, the label for the root exists, but it's empty. This is done because the root technically is part of every domain name, so it must be included in every domain name. If it were something long like Root, we would need to include that at the end of every domain name. This would simply make every name longer, while not really adding any useful information—we already know every domain name is under the root.

Consider the example of a top-level domain called Rocks, within which is a second-level domain Crystal. The domain name of Rocks is “Rocks.”, with the dot separating Rocks and the empty label (the null root). In practice, the trailing dot is often omitted, so the domain name of the top-level domain Rocks can be considered as just “Rocks”. The subdomain Crystal within Rocks has the domain name “Crystal.Rocks”. If we had a device named Salt within the Crystal.Rocks domain, it would be called “Salt.Crystal.Rocks”. This is fairly straightforward, as you can see in Figure 53-4.



**Figure 53-4: DNS labels and domain name construction** Each node in the DNS name space has a label (except the root, whose label is null). The domain name for a node is constructed simply by placing in order the sequence of labels from the top of the tree down to the individual domain, going from right to left, separating each label with a dot (period).

We can use these names to easily identify subdomains of a particular domain. For example, if we start with Salt.Crystal.Rocks, it's obvious that Crystal.Rocks is its parent domain. It's also clear that both Crystal.Rocks and Salt.Crystal.Rocks are subdomains of Rocks; one is a single level down from Rocks, and the other is two levels down.

Note that there is a maximum limit of 255 characters for a complete domain name, for implementation purposes. In practice, most domain names are much shorter than this limit, as it would violate the whole purpose of domain names if we let them get so long that no one could remember them.

**KEY CONCEPT** A *domain name* is a string of text that uniquely identifies a particular node in the name space. The domain name for a node is constructed by concatenating in right-to-left order all the labels in the branch of the DNS tree, starting from the top of the tree down to the particular node, separating each by a dot (period).

Finally, note that in many protocols, it is possible to specify a particular resource within a domain name by providing a directory structure after a name. This is done using the standard TCP/IP URL syntax, where a path is indicated using slashes to separate subdirectories. For example, a specific file at Salt.Crystal.Rocks might be located at Salt.Crystal.Rocks/chem/composition. While DNS names are case-insensitive, the labels in a path are case-sensitive. So, this example would be different from Salt.Crystal.Rocks/chem/Composition. See the discussion of URL syntax in Chapter 70 for more details.

## Absolute (Fully Qualified) and Relative (Partially Qualified) Domain Name Specifications

As explained in the previous section, we can specify the domain name of any node in the DNS name hierarchy by simply starting at the root node and following the sequence of subdomains down to the node in question, listing each level's labels separated by a dot. When we do this, we get a single name that uniquely identifies a particular device. In practice, domain names can be specified by their fully qualified names or their partially qualified names.

### Fully Qualified Domain Names

Technically, if a top-level domain A contains a subdomain B that contains subdomain C, the full domain name for C is "C.B.A.". This is called the *fully qualified domain name (FQDN)* for the node. Here, the word *qualified* is synonymous with *specified*. The domain name C.B.A. is fully qualified because it gives the full location of the specific domain that bears its name within the whole DNS name space.

FQDNs are also sometimes called *absolute* domain names. This term reflects the fact that you can refer unambiguously to the name of any device using its FQDN from any other portion of the name space. Using the FQDN always instructs the person or software interpreting the name to start at the root, and then follow the sequence of domain labels from right to left, going top to bottom within the tree.

## **Partially Qualified Domain Names**

There are also some situations in which you may refer to a device using an incomplete name specification. This is called a *partially qualified domain name (PQDN)*, which means that the name only partially specifies the location of the device. By definition, a PQDN is ambiguous, because it doesn't give the full path to the domain. Thus, you can use a PQDN only within the context of a particular parent domain, whose absolute domain name is known.

We can find the FQDN of a partially specified domain name by appending the partial name to the absolute name of the parent domain. For example, if we have the PQDN Z within the context of the FQDN “Y.X.”, we know the FQDN for Z is “Z.Y.X.”.

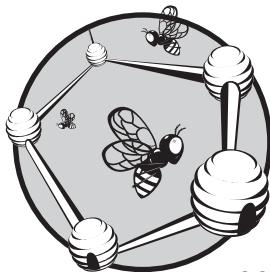
Why bother with this? The answer is convenience. An administrator for a domain can use PQDNs as a shorthand to refer to devices or subdomains without needing to repeat the entire full name. For example, suppose you are in charge of the computer science department at the University of Widgetopia. The domain name for the department as a whole is “cs.widgetopia.edu.”, and the individual hosts you manage are named after fruit. In the DNS files you maintain, you could refer to each device by its FQDN every time; for example, “apple.cs.widgetopia.edu.”, “banana.cs.widgetopia.edu.”, and so on. But it's easier to tell the software, “If you see a name that is not fully qualified, assume it is in the cs.widgetopia.edu domain.” Then you can just call the machines apple, banana, and so on. Whenever the DNS software sees a PQDN such as kiwi, it will treat it as “kiwi.cs.widgetopia.edu.”.

**KEY CONCEPT** A *fully qualified domain name (FQDN)* is a complete domain name that uniquely identifies a node in the DNS name space by giving the full path of labels from the root of the tree down to that node. It defines the absolute location of a domain. In contrast, a *partially qualified domain name (PQDN)* specifies only a portion of a domain name. It is a relative name that has meaning only within a particular context. The partial name must be interpreted within that context to fully identify the node.

I mentioned earlier in this chapter that the trailing dot for the null root domain is usually omitted. This is true in common parlance and when users specify a domain name in an application. You don't use the trailing dot in your web browser, for instance. However, the dot is used to clearly distinguish a FQDN from a PQDN within DNS master files. This allows us to use both FQDNs and PQDNs together. In our example, apple would refer to “apple.cs.widgetopia.edu.”, but “apple.com.” would refer to the FQDN for Apple Computer, Inc. You must be careful about watching the dots here, because apple.com (without a trailing period) would be a PQDN and would refer to “apple.com.cs.widgetopia.edu.”, not the domain of Apple Computer.

# 54

## DNS NAME REGISTRATION, PUBLIC ADMINISTRATION, ZONES, AND AUTHORITIES



The previous chapter explained how the Domain Name System (DNS) name space consists of a hierarchy of domains and subdomains. From the root, we have a number of top-level domains, then second-level domains below them, and still lower-level domains below that. The obvious questions then become: How do we determine the shape and structure of the name space, and who will manage it? More specifically, who will control the root of the tree and decide what the top-level domains will be called? How will we then subdivide control over the rest of the name space? How do we ensure there are no conflicts in choosing the names of sibling subdomains within a domain?

DNS can be used on private networks controlled by a single organization, and if so, that organization is obviously in charge of the name space. We'll discuss private naming, but in reality, it's just not that interesting. The vast majority of DNS use occurs on the public Internet. Here, we have a much greater challenge, because we need to construct a name space that

spans the globe and covers millions of machines managed by different organizations. For this, we need a very capable *name registration* process and administration methods to support it.

In this chapter, I will describe the process of name registration and how authorities are managed within DNS, focusing on the public Internet. I begin with a description of the DNS hierarchical authority structure and how it relates to the hierarchical name space, and a discussion of the concepts behind the DNS distributed name database. I describe the Internet's organizational and geopolitical top-level domains, and how they are administered by various authorities. I then discuss how authority is delegated to the second-level and lower-level domains, and how public registration of domain names works, including how public registration issues and problems are resolved. I explain how the DNS name space is partitioned into administrative zones of authority, and then I conclude with a brief discussion of private DNS name registration.

**RELATED INFORMATION** Most TCP/IP implementations include a special utility called *whois* that can be used to interrogate the DNS distributed name database to obtain registration information about domains. This application can be very useful for troubleshooting. For details, see the section discussing *whois* in Chapter 88.

## DNS Hierarchical Authority Structure and the Distributed Name Database

In the previous chapter, I explained that the central concept of naming in DNS is based on *domains*. Each domain can be considered akin to a sphere of influence or control. A domain “spreads its wings” over all the objects and subdomains that it contains. Due to this concept of influence, when we consider any DNS name space, we see that it is hierarchical because it reflects a hierarchy of organizations that control domains and the nodes within them. This means that there is a *hierarchical authority structure* that complements the hierarchical name structure in DNS.

The primary reason why the name space hierarchy leads to an authority hierarchy is the requirement that sibling subdomains be unique within a domain. As soon as we have a need for uniqueness, we must have some sort of authority or process that ensures that each subdomain or object picks a different name within that domain. This is what name registration is all about.

This concept of a hierarchical authority structure is a bit abstract, but it's easier to understand if we examine a sample DNS name space and discuss the issues involved in assigning names within it. Naturally, we want to start at the top of the name hierarchy, with the root domain, null.

### ***The DNS Root Domain Central Authority***

To start off the name space, we must create top-level domains (TLDs) within the root. Now, each of these must be unique, so one authority must manage the creation of all TLDs. This means that the authority that controls the root domain controls the entire name space.

In the case of the Internet, this central authority is ultimately responsible for every name in DNS. The central DNS authority for the Internet, which controls the

creation of TLDs, was initially called the *Network Information Center*. It was later the *Internet Assigned Numbers Authority (IANA)*, which is also responsible for protocol numbers, IP addresses, and more. These functions are now shared by IANA and the *Internet Corporation for Assigned Names and Numbers (ICANN)*. We'll discuss the specific TLDs of the Internet in the next few chapters; IANA, ICANN, and related organizations are discussed in the section on Internet registration authorities in Chapter 3.

## **TLD Authorities**

At the next level down in the authority hierarchy, we create second-level domains within each of the TLDs. Each TLD must itself be managed using a coordinating authority, however, this is not necessarily the organization that runs the root (IANA). IANA *delegates* authority for some of the TLDs to other organizations.

IANA may delegate control for each TLD to a different authority at this level of the hierarchy. In fact, there can be completely different rules for managing the creation of second-level domains in one TLD than there are in another. And in some TLDs, there are multiple authorities that work together on name registration.

## **Lower-Level Authority Delegation**

This process of authority delegation continues as we move down the name space hierarchy. At each level, the name space becomes more specific.

If we use an organizational hierarchy, like the .COM TLD, we generally delegate authority for each second-level domain to the organization whose name it represents. So, for example, IBM.COM is managed by IBM. Since IBM is huge, it may itself subdivide the authority structure further, but smaller organizations probably won't.

## **Authority Hierarchy's Relationship to the Name Hierarchy**

The authority hierarchy is complementary to the name hierarchy; they are not exactly the same. It is not necessary that there be a different authority for every level of the hierarchy. In many cases, a single authority may manage a section of the name space that spans more than one level of the structure. For example, IANA manages the Internet root domain (null) and also the .INT TLD, but other TLDs are managed by other organizations. The name hierarchy is divided into *zones of authority* that reflect the hierarchy of authorities that manage parts of the name space.

Also, authority over a domain doesn't necessarily imply physical control. A domain can contain subdomains that are managed by organizations on different continents, and a single subdomain can contain named devices that are on different continents as well.

## **The DNS Distributed Name Database**

Of course, with authority comes responsibility, and the main responsibility an authority has for a domain is registering names within the domain. When a name is registered, a set of data is created for it, which can then be used by internetwork devices to resolve the name into an address or perform other functions.

The set of all the data describing all DNS domains constitutes the DNS *name database*. Just as registration authority is distributed and hierarchical, this database is distributed and hierarchical. In other words, there is no single place where all DNS name information is stored. Instead, DNS servers carry resource records (see Chapter 57) that describe the domains for which they have authority. As you'll see, the fact that this database is distributed has major implications on how name resolution is carried out.

**KEY CONCEPT** The name space of the public Internet is managed by a *hierarchy of authorities* that is similar in structure to the hierarchical DNS name space, though not identical. The top of the hierarchy is centrally managed by IANA/ICANN, which delegates authority to other organizations for registering names in various other parts of the hierarchy. The information about name registrations is maintained in resource records stored in various locations, which form a distributed name database on the Internet.

## DNS Organizational (Generic) TLDs and Authorities

The top of the DNS name hierarchy is managed by a central authority, which controls the entire name space by virtue of deciding which TLDs are allowed to exist. Obviously, it is very important that a great deal of thought go into how the TLDs are chosen. A poor design at this top level would make the entire hierarchy poorly reflect the actual structure of organizations using the name space.

The creators of DNS could have chosen any number of ways to structure the Internet's name hierarchy. One obvious possibility is to structure the Internet based on geopolitical boundaries: countries, states, and so forth. Another sensible idea is to structure the name space based on types of organizations.

The beauty of the hierarchical name space is that we don't need to choose between different methods of structuring the name space. We can use more than one technique at the same time, and this is exactly what was done when DNS was first implemented. Both the organization type and geography structures were used for TLDs. This gives multiple options for name registration for most groups and individuals.

I'll begin here by discussing organizational TLDs, and then we'll look at geopolitical ones. As you'll see, although there are only a handful of organizational TLDs, there is no doubt that they have been much more popular than the geopolitical ones.

### ***Original Generic TLDs***

The initial deployment of DNS featured a set of seven top-level domains that are called *generic* TLDs. The idea was that each company or organization could choose a name within one of these TLDs; they were generic enough that every organization would find a place that suited them. I prefer to call them *organizational*, because they divide the generic portion of the name space by organization type.

The initial TLDs and their original intended organization types were as follows:

- .ARPA** A temporary domain used many years ago for transition to DNS; today, this domain is used for reverse name resolution (see Chapter 56).
- .COM** Corporations and businesses
- .EDU** Universities and other educational organizations
- .GOV** Government agencies
- .MIL** Military organizations
- .NET** Organizations that implement, deal with, or manage networking technologies and/or the Internet
- .ORG** Other organizations that don't fit into any of the previous classifications

At first glance, this seems like a reasonable way to cover the organizations of the world. However, since the .ARPA domain (whose name refers to the ARPAnet, the precursor of the modern Internet, as described in Chapter 8) was temporary, this left only six categories for all other organizations. Also, the TLDs weren't all used as was originally foreseen. For example, the .GOV and .MIL domains were not used for all types of government and military organizations, but primarily for the United States federal government and military. The .EDU domain ended up being used only for universities, again in the United States.

This left only three common TLDs—.COM, .NET, and .ORG—for almost all other groups and companies that wanted to use the organizational hierarchy. Since there were only three such TLDs, they quickly became very crowded, especially the .COM domain. A new fourth domain, .INT for international organizations, was added fairly soon to the original seven. However, it was intended only for a small number of organizations, such as international standards bodies.

Of course, there was no inherent reason why the generic domains should be limited to only the few that were originally created.

### **New Generic TLDs**

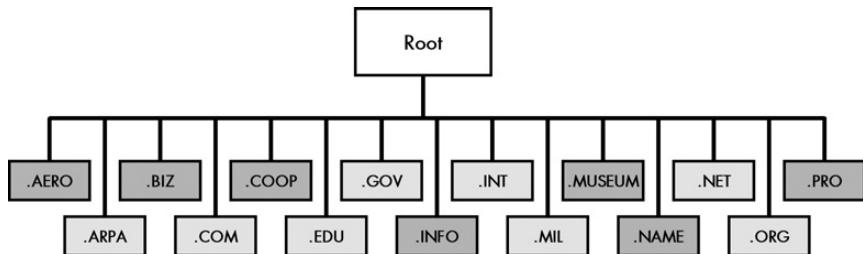
Over the years, many suggestions were made for new generic TLDs that would expand the number of possible second-level domain names and also provide better categorization for different organization types—that is, to make the generic TLDs less generic. There was some resistance at first to adopting these new names, especially because there were so many different ideas about what new TLDs should be created.

IANA took input from a lot of people and followed a complex procedure to determine what new TLDs should be made. In 2001 and 2002, approval was given for the creation of several new TLDs, and decisions were made about authorities for administering them.

Of the new TLDs approved in the past few years, the number that has achieved widespread popularity is, to my knowledge, zero. Humans are creatures of inertia, and most people are still used to names ending in .COM, .NET, or .ORG. In time this may change, but it will probably take a few years.

**NOTE** Some people actually felt that adding new generic TLDs was a bad idea, since it makes organizations potentially more difficult to locate (due to the possibility of a name ending in a variety of different TLDs). This is debatable, however, especially since the exhaustion of address space in the existing TLDs means many companies have needed to choose unintuitive domain names anyway.

Table 54-1 shows all the current generic TLDs and describes how they are used, and it lists the current central authority that manages each. The original TLDs are highlighted in italics (I am including .INT as an original TLD, since it was created long before the “new” ones). Figure 54-1 shows the 15 generic TLDs in graphical form.



**Figure 54-1: Internet DNS organizational (generic) TLDs** There are 15 generic TLDs currently defined for the Internet. They are shown here in alphabetical order, with the original TLDs shown in light shading and the new ones added in 2001/2002 in darker shading.

**Table 54-1:** Internet DNS Organizational (Generic) Top-Level Domains

Generic TLD	Abbreviation For	Authority	Description
.AERO	Aerospace	Société Internationale de Télécommunications Aéronautiques (SITA)	Used for members of the aerospace industry, such as airlines and airports. (Yes, that is French!)
.ARPA	Address and Routing Parameter Area	IANA/ ICANN	First defined as a temporary domain for migration from the older host table system, the ARPA of course originally stood for the Advanced Research Projects Agency, creators of the predecessors of the Internet. Today, the .ARPA domain is used for internal Internet management purposes; the expanded name shown in this table was, I believe, chosen to fit the acronym. The best-known use of this domain is for reverse DNS lookups.
.BIZ	Business	NeuLevel, Inc.	Used for businesses. Intended as a competitor to .COM.
.COM	Commercial Organizations	VeriSign, Inc.	Originally intended for corporations and other commercial interests, .COM is also widely used for other purposes, including small businesses and even individuals who like the popularity of the .COM domain.

(continued)

**Table 54-1:** Internet DNS Organizational (Generic) Top-Level Domains (continued)

Generic TLD	Abbreviation For	Authority	Description
.COOP	Cooperative Associations	Dot Cooperation, LLC	Used for cooperative associations.
.EDU	Education	Educause	Originally intended for all types of educational organizations, .EDU is now used only for degree-granting higher-education institutions accredited in the US. Other educational institutions such as public schools usually use the country code TLDs.
.GOV	Government	U.S. General Services Administration	Reserved for the U.S. federal government.
.INFO	Information	Afilias, Ltd.	A very generic TLD designed for information resources of various sorts. It is unrestricted, in that anyone can register any sort of organization in .INFO. It's also positioned as an alternative to .COM.
.INT	International	IANA .int Domain Registry	Used only for large organizations established by international treaty.
.MIL	Military	U.S. DoD Network Information Center	Reserved for the U.S. military.
.MUSEUM	Museum	Museum Domain Management Association	Take a guess. See <a href="http://index.museum">http://index.museum</a> for a complete list of museums using this TLD.
.NAME	Names	Global Name Registry	In the original generic hierarchy, there was no place set aside for individuals to register names for themselves, so people would create domains like <code>jonesfamily.org</code> . This was not ideal, so .NAME was created as a place for individuals and families to register a domain for their names. .NAME also competes with the country code TLDs.
.NET	Network	VeriSign, Inc.	This was supposed to be used only for Internet service providers (ISPs) and other organizations working intimately with the Internet or networking. Due to the exhaustion of name spaces in .COM and .ORG, many .NET domains are registered to other organizations, however.
.ORG	Organizations	Public Interest Registry	Originally intended for organizations not fitting into the other generic TLDs, .ORG quickly became associated with professional and nonprofit organizations. It is possible, however, to have a for-profit company use an .ORG name.
.PRO	Professional	RegistryPro	Reserved for credentialed professionals such as lawyers and doctors.

**KEY CONCEPT** One of the two ways in which the Internet's DNS name space is divided is using a set of generic TLDs. These TLDs are intended to provide a place for all companies and organizations to be named based on their organization type. There were originally six such domains, but this has been expanded so that there are now 15.

## DNS Geopolitical (Country Code) TLDs and Authorities

In theory, the generic TLDs would be sufficient to meet the needs of all the individuals, companies, and groups in the world. This is especially true since .ORG, by definition, is a catchall that can include anyone or anything. Thus, in an ideal world, everyone in the world would have been able to find a place in those simple domains.

However, back at the beginning of DNS, its creators recognized that the generic TLDs might not meet the needs of everyone around the world, especially in certain cases. There are several reasons for this:

**Americentrism of the Generic Domains** I don't mean this as a criticism (I'm an American citizen and love my country!). It is indisputable, however, that United States organizations and companies dominate the generic TLDs. This is not surprising, given that the Internet was first developed in the United States, but it still presents a problem for certain groups. For example, if the United States military controls the .MIL domain, where does, say, Great Britain's military fit into the name space?

**Language** Most of the generic domains are populated by organizations that primarily do business in English. There are hundreds of languages in the world, however, and it's easier for the speakers of those tongues if they can more readily locate resources they can understand.

**Local Control** Countries around the world rarely agree on much, and they certainly differ on how organizations within their nations should have their Internet presence arranged. There was a desire on the parts of many to allow nations to have the ability to set up subsets of the name space for their own use.

For these and other reasons, the Internet's name space was set up with a set of *country code* TLDs paralleling the generic ones, sometimes called *ccTLDs*. I call these *geopolitical* TLDs, since they are based on geopolitical divisions of the world (similar to the example I used in the overview of the DNS name space in Chapter 53). In this hierarchy, every country of the world is assigned a particular two-letter code as a TLD, with a specific authority put in charge of administering the domain. For example, the ccTLD for Great Britain is .UK, the one for Canada is .CA, and the one for Japan is .JP. The codes often are more meaningful in the local language than in English. For example, Germany's is .DE, and Switzerland's is .CH.

### ***Country Code Designations***

When I said that countries rarely agree on anything, I wasn't kidding. In fact, they can't even agree on what's a country! Real shooting wars have been fought over whether or not a particular territory was independent or part of another nation, and the creators of DNS wanted no part of this sort of controversy. As the IANA website says, "The IANA is not in the business of deciding what is and what is not a country, nor what code letters are appropriate for a particular country."

To remain neutral, IANA's ccTLD codes are taken directly from the standard country abbreviations maintained by the International Organization for Standardization (ISO) in ISO Standard 3166-1. When a country is recognized by the ISO and a code assigned to it on this list, IANA creates it as a TLD. There are presently more than 200 different geopolitical TLDs. You can find the current list of IANA country code TLDs at <http://www.iana.org/cctld/cctld-whois.htm>.

**KEY CONCEPT** Due to the limitations of the generic TLDs, a set of *country code* top-level domains was created. This *geopolitical hierarchy* allows each nation on earth to set up its own name system based on its own requirements and to administer it in the manner it sees fit. The IANA determines what is a country based on official decisions made by ISO.

## ***Country Code TLD Authorities***

Each country has the authority to set up its TLD with whatever internal substructure it chooses; again, this is the power of a hierarchical structure.

Some countries enforce a further geographical substructure at the lower levels. For example, the .US domain for the United States was originally set up so that all second-level domains were two-letter state abbreviations (this was later changed).

Other countries may actually use organizational subdomains within their country code. For example, Great Britain has .CO.UK for companies in the country (like .COM but for the UK only; they left off the *M*), and .COM.AU is for corporations in Australia.

Other countries may not have any particular substructure at all, especially if they are small.

## ***Leasing/Sale of Country Code Domains***

Interestingly, some very small countries with recognizable country codes, especially to English speakers, have used their codes for very creative purposes, including selling or renting the name space to enterprising companies.

A good example is the .TV domain, which technically belongs to the island nation of *Tuvalu*. Of course, to most people, "TV" means something quite different. Some folks thought that domain names ending in .TV might be popular in the English-speaking world, so they formed a company called The .TV Corporation and negotiated with the government of Tuvalu to use the .TV domain. Today, the authority for this TLD is this corporation, headquartered in California! Similar arrangements can be found with the .CC, .NU, .TO, and other TLDs.

This serves as a good reminder that the name space is logical and not physical. Obviously, the many computers with .TV names are not actually located on a remote island in the South Pacific. Similarly, if a website ends with .CA, for example, it probably represents a Canadian organization, but that doesn't necessarily mean the website itself is actually hosted in Canada.

## **Drawbacks of the Geopolitical TLDs**

The geopolitical domains have been very popular for certain uses. National governments and other official institutions like to use them, for obvious reasons. Typing `www.gov.xx` or `www.government.xx`, where `xx` is a country code is likely to bring you to the national government website of most countries. Some companies and organizations use the ccTLDs because they allow them to choose a name already taken in the generic hierarchies or simply to express national pride.

For many other companies and organizations, however, the generic TLDs have been much more popular than the country codes. I think the most important reason for this is that organizations are easier to locate using the generic domains.

Here's a good example of what I mean. In the town near where I live, a new grocery store called Aldi recently opened. I like the store and wanted to learn more about it, so I fired up my web browser and sought out its website. Yes, I could have typed it into a search engine, but like most people, I'm lazy. It was much easier to just enter `www.aldi.com` into my browser, and lo and behold, up popped the website of Aldi International.

Now, Aldi is actually headquartered in Germany, and the company does have a website at `www.aldi.de` as well. But I didn't know that. I found them easily by going to `www.aldi.com`, because I didn't need to know their physical location, and because I know that most large companies have a .COM domain. Of course, being findable is very important, especially for commercial organizations trying to do business.

Another good example is the United States, which has its own country code, .US, in addition to dominating the generic TLDs. The authority in charge of this domain initially chose to make it follow a strict geographical hierarchy, so every domain had to be of the form `organization.city.state-code.us`. So, to use this part of the name space, a company in Boston must be within the `.boston.ma.us` domain. That's very neat and logical, but it makes names both longer and harder to guess than the generic equivalents.

Suppose you wanted to get information on metals giant Alcoa. If you're in the industry, you might know Alcoa is located in Pittsburgh, but if not, which is easier to find: `www.alcoa.pittsburgh.pa.us` or `www.alcoa.com`? Anyone here know how to spell Albuquerque?

It is for this reason that the .US domain achieved success in certain segments of society but not in others, especially commercial entities (corporations). The strict hierarchy does have some real advantages, such as avoiding name space conflicts, but its disadvantages were such that the rules were recently relaxed in the .US domain.

## **Public Registration for Second-Level and Lower Domains**

The IANA is in charge of deciding which TLDs exist in the Internet name space, and as such, they are ultimately responsible for all names in the Internet. The entire point of the authority hierarchy, however, is that IANA should not be responsible for the whole name space. So, while IANA maintains control over certain TLDs, such as .INT and .ARPA, control for managing the others is delegated to secondary authorities for each TLD.

Just as IANA had the choice of how to delegate authority to the subdomains of the root domain, the organization in charge of each TLD gets to make the same decision about how second-level domains are to be created under the TLD.

In many TLDs, especially the generic ones, second-level domains are assigned directly to individuals or organizations. For example, a company named XYZ Industries might want to get the domain xyzindustries.com.

In other TLDs, second-level domains are set up in a particular structure, like the state codes used in the .US domain. There, you need to go down more levels, but eventually you get to the point where companies and people register their own domains. For example, in the .US domain, XYZ Industries might want to register xyz.phoenix.az.us if it were headquartered in Phoenix.

This transition point between the authorities granted responsibility for parts of the name space and the regular people and groups who want to get names is important. A process of *public registration* had to be established to allow such name assignment to occur in a consistent and manageable way. This was not that difficult to accomplish back when the original generic TLDs and country code TLDs were first created. The Internet was quite small, and it made sense to just have the authority in charge of each TLD perform registrations within that TLD. This ensured that there was no duplication of names within a TLD with a minimum of fuss.

## ***Registration Authority***

For very important generic TLDs such as .COM, .NET, and .ORG, the authority in charge of registration was the Internet Network Information Center (the InterNIC). The InterNIC was set up as a service administered by the United States government, who later granted the contract to manage it to Network Solutions Inc. (NSI). NSI was eventually purchased by VeriSign, who later spun it off as a separate venture. (Things change quickly in the networking world!)

NSI single-handedly performed all registrations within the .COM, .NET, and .ORG TLDs for many years. The popularity of the original generic TLDs, however, led to an explosion in demand for name registration in these domains in the 1990s. Having a single company in charge of registration led to this becoming another bottleneck in the Internet's Domain Name System. There were also many folks who didn't like the lack of accountability and competition that came with having a single monopoly in charge of registration. The InterNIC could set its own price and originally charged \$35 per year per domain name, then later \$50 per year.

In the late 1990s, responsibility for name registration was given to ICANN. The registration process was *deregulated*, to borrow a term referring to removal of monopolies from industries like power generation. As of December 1999, there was still a single authority with overall responsibility for each TLD, including .COM, .NET, and .ORG.

Today, NSI is still the authority running .COM and .NET. However, it isn't the only organization that registers names within these TLDs. It further delegates registration authority to a multitude of other companies, called *accredited registrars*. Any registrar can register names within the TLD(s) for which they are accredited.

## ***Registration Coordination***

Naturally, coordination becomes much more of a concern when you have multiple companies registering names in a TLD. A special set of technical and administrative procedures is followed to ensure that there are no problems, such as two registrars trying to grab a name at the same time.

The system has worked well, and those who wish to use TLDs where competition exists now can choose from a variety of registering companies. The most noticeable result of this was also the most predictable one: the cost of registering a domain name in the deregulated generic TLDs is usually much lower than the fees originally charged by the InterNIC.

Once a company, individual, or organization has a registered lower-level domain, he/she/it becomes the authority for that domain. Use of the domain then becomes private, but depending on how the domain is used, further public name registration may be required. See the discussion of private registration, near the end of this chapter, for more information.

## **DNS Public Registration Disputes and Dispute Resolution**

The Internet started off as a medium for research into networking, evolved into a system for interconnecting scientists, and ended up as a global communications tool used by just about everyone. As part of this evolution, the Internet also became a very important part of how business is done in the world. Money started to come into the Internet picture in the early 1990s, and just a few short years later, its impact on the Internet was so significant that the growth of the stock market to dizzying heights in the late 1990s is now often called “the Internet bubble.”

### ***Public Registration Disputes***

Unfortunately, the increasing importance of the Internet to commercial interests crashed headlong into the noncommercial original design of Internet technology, and nowhere was this more evident than in DNS. Since there were only a few generic TLDs, each name within a TLD had to be unique, and humans are often confrontational creatures, it didn’t take long before arguments broke out over who should be able to use what name and why. And, of course, from there, it didn’t take long before lawsuits and other unpleasantries were common.

There are a surprising number of significant problems associated with public registration of domain names:

**Corporate Name Conflicts** The .COM domain is for corporations, but many corporations have the same name. The ACME Furniture Company, the ACME Restaurant Supply Corporation, and ACME Footwear, Inc., probably all would like to have the acme.com domain. But there can be only one such domain within .COM. (These are fictional examples; acme.com is actually owned by an organization called *Acme Labs.*)

**Corporate/Individual/Small Business Name Conflicts** There are many corporations that have names similar to or even identical to the names of individuals, leading to potential conflicts. For example, suppose your first name is Wendy and you own a small fabric store called Wendy's Fabrics. But you are Internet savvy and decide you want to register [wendys.com](http://wendys.com) as soon as you hear about the Internet in 1993. Then this big hamburger chain comes along and has a problem with that.

**NOTE** *To my knowledge, no such issue arose with respect to Wendy's, but there actually was a widely publicized case that shows just how recently most corporations were out of the loop with respect to domain naming. In 1994, a writer for Wired magazine was astonished to find that the mcdonalds.com domain name was unregistered! To show just how unregulated the registration process was, he registered it himself, and caused a bit of a stir as a result. The Golden Arches folks eventually acquired the domain from him in an amicable arrangement, where he relinquished the name and they made a donation to charity.*

**Corporate Warfare** A particularly ugly type of conflict is when companies intentionally try to take business from each other by registering names that have nothing to do with their own companies. An example would be if Burger King had tried to register [mcdonalds.com](http://mcdonalds.com) and use it to advertise Burger King products. (Which they didn't do, I might add, so please nobody sue me!) Another example is when companies try to use alternate TLDs, such as registering [burgerking.org](http://burgerking.org) to confuse people trying to find [burgerking.com](http://burgerking.com). In fact, many companies have taken the step of registering their names in many different TLDs to prevent this sort of thing from happening.

**Cybersquatting** Some ambitious (to choose a nice term) individuals, recognizing early on the potential value of certain names, registered large volumes of names with the hopes of reselling them. Many people condemned this as exploitative, and the term *cybersquatting* was created to refer to this type of activity. Unfortunately, a lot of money was made this way, and there are many domain names that, to this day, cannot be used because they have been reserved indefinitely by people or individuals who will never use them.

**Deceptive Naming Practices** Another type of somewhat diabolic creativity has been displayed by people who seek to take advantage of the inability of some of us to spell. For example, if you were a competitor of a large company called Superb Transceivers Inc., which registered [superbtransceivers.com](http://superbtransceivers.com), you might register [superbtranscievers.com](http://superbtranscievers.com) and redirect traffic from there to your own domain. Another example takes advantage of the common mix-up between the letter *O* and *0* (zero). For example, a software company once registered [micros0ft.com](http://micros0ft.com), much to the chagrin of the Redmond, Washington software giant.

Incidentally, it was all this nonsense that led, in part, to the clamor for new generic TLDs. Even though the more complicated schemes used by TLDs like [.US](http://.US) are not very popular, they have a huge advantage over the generic domains. Since all these registrations are geographic, there are far fewer conflicts. For example, the ACME Furniture Company might use [acme.seattle.wa.us](http://acme.seattle.wa.us), the ACME Restaurant Supply Corporation might have [acme.mendocino.ca.us](http://acme.mendocino.ca.us), and ACME Footwear, Inc., could go with [acme.anchorage.ak.us](http://acme.anchorage.ak.us). A dispute would arise only

when organizations have the same name and also are in the same state and town. You could still have three Joe's Pizza Parlors in Chicago duke it out, but it's not likely to be a problem for big companies.

### ***Methods of Registration Dispute Resolution***

So, how do we resolve these situations? As the saying goes, it can be done either the easy way or the hard way. Here are some methods that have been used for dispute resolution:

**Domain Name Sharing** Sometimes, the antagonists agree on a productive solution. One particularly constructive idea is to agree to *share* the domain name. For example, the three different ACME companies could each create their own more specifically named domains, such as acmefurniture.com, acmerestaurantsupply.com, and acmefootwear.com. Then they might agree to have the www.acme.com registered to nobody, by having one company register it and not use it for anything. Even better, they could set it up with a simple web page that says the domain is shared, with a link to the three sites. Unfortunately, it seems grade school children understand the concept of sharing better than most corporate executives do, so this type of resolution is rare.

**Domain Name Purchase** Another option is purchase. If a big company wants a domain name already registered by an individual or a small business, it will often just purchase the name, as this is the easiest thing to do. During the height of the Internet mania, there were domain names that sold for *millions* of dollars—just for the right to use the name! Many cybersquatters and other speculators got rich selling names.

**Litigation** Often, the combatants don't play nice, and the usual occurs: threats, intimidation, lawsuits, and so forth. Sometimes, a letter from a lawyer is enough to resolve the issue, especially when some poor individual owning a website gets threatened with legal action by a large company—this has happened many times. However, often the disagreeing parties stick to their guns, especially if two companies lock horns and their lawyers refused to back down. Usually, the matter then ends up in the courts, where it is eventually resolved one way or the other. Usually, claims of trademark infringement would be used by a company challenging a prior domain name registration.

### ***The Uniform Domain Name Dispute Resolution Policy***

Lawsuits are expensive and time-consuming, so there was a desire that some other mechanism exist for resolving these conflicts as well. Since the authority for each TLD controls what happens within it, it also has the right to create its own policies for how to deal with these sorts of issues. For the generic TLDs, the original registering authority, the InterNIC, had a dispute resolution policy that allowed people with a complaint to challenge a domain name registration if they had a trademark interest in that name. The policy was controversial for a number of reasons, not the least of which because it led to some domain names being successfully challenged, even if there was no proof of trademark infringement.

The current authority for the generic TLDs, IANA/ICANN, created a new Uniform Domain Name Dispute Resolution Policy (UDRP) in 1999, to better handle domain name conflicts. This policy specifies a procedure whereby a company that has a valid trademark can challenge a domain name if it infringes on the trademark, is confusingly similar to it, or was registered by someone else in bad faith. At the same time, it also lists ways that the original registrant can prove that the registration is valid and should be maintained. This new system eliminates many of the problems associated with public registration of domain names—such as deceptive naming, corporate warfare, and cybersquatting—while not automatically allowing a second-comer to shut down a legitimate domain.

## **DNS Name Space Administrative Hierarchy Partitioning: DNS Zones of Authority**

I explained earlier in this chapter that the DNS name space is arranged in a hierarchy and that there is also a hierarchy of authorities that is related to that hierarchical name structure. However, the two hierarchies are not exactly the same. If they were the same, we would need a separate authority for every domain at every level of the tree, and that's something we are very unlikely to want to have everywhere in the structure.

At the very top levels of the DNS tree, it seems reasonable that we might want to designate a separate authority at each level of the structure. Consider the geopolitical name hierarchy; IANA/ICANN manages the root domain, but each of the ccTLDs is managed by a distinct national authority.

However, when we get to the lower levels of the structure, it is often inconvenient or downright impossible to have each level correspond to a separate authority. As an example, let's suppose you are in charge of the Googleplex University IT department, which runs its own DNS servers for the googleplex.edu domain. Suppose there were only two schools at this university, teaching fine arts and computer science. Suppose also that the name space for the computers were divided into three subdomains: finearts.googleplex.edu, compsci.googleplex.edu, and admin.googleplex.edu (for central administrative functions, including the IT department itself).

Most likely, you don't want or need the Fine Arts department running its own DNS servers. The same is likely true of the administration machines. However, it's possible that the Computer Science department does want to run its own DNS servers, because this department probably has many more computers than the other departments, and the staff might use running a DNS server as part of the curriculum.

In this case, you might want yourself, the administrator for googleplex.edu, to maintain authority for the finearts.googleplex.edu and admin.googleplex.edu subdomains and everything within them, while delegating authority for compsci.googleplex.edu to whomever in the Computer Science department is designated for the task. DNS is specifically designed to allow these divisions between the name hierarchy and the authority structure to be created.

## **Methods of Dividing a Name Space into Zones of Authority**

The complete DNS name structure is divided by making *cuts* (as RFC 1034 calls them) between adjacent nodes to create groups of contiguous nodes in the structure. Each group is called a *zone of authority*, or more commonly, just a *zone*. Each zone is usually identified by the domain name of the highest-level node in the zone; that is, the one closest to the root. The zones in DNS are by definition *non-overlapping*—every domain or subdomain is in exactly one zone. The division of the name space into zones can be made in an arbitrary way. At one extreme, we could place a cut between every node, and thereby divide the entire name space so each domain (and subdomain, and so on) was a separate zone. If we did this, the name hierarchy and authority hierarchy would indeed be the same for the entire DNS tree. At the other end of the scale, we could use no cuts at all, defining a single zone encompassing the entire DNS structure. This would mean the root was the authority for the entire tree.

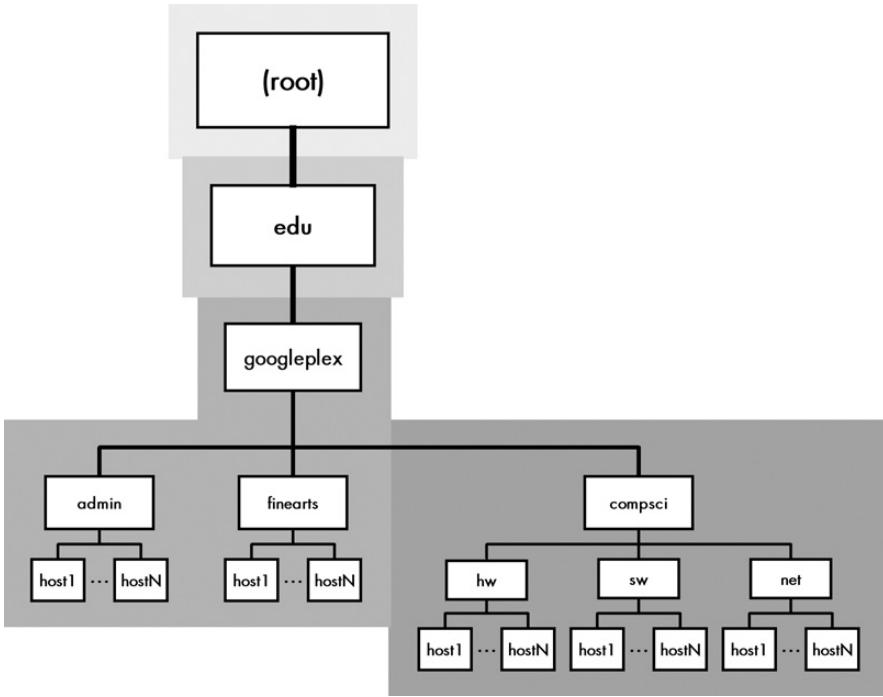
Of course in practice, neither of these methods is particularly useful, as neither reflects how the real-world administration of DNS works. Instead, we generally divide the name structure in a variety of places, depending on the needs of different parts of the name space. There are many cases where we might want to create a subdomain that is responsible for its own DNS server operation; there are others where we might not want to do that. The significance of a cut in the name hierarchy is that making such a cut represents, in essence, a *declaration of DNS independence* by the node below the cut from the one above the cut.

Returning to our example, if googleplex.edu is in charge of its own DNS servers, then there would be a cut in the name space between googleplex.edu and .EDU at the next-higher level. This means that the DNS server for .EDU is no longer in charge of DNS for the googleplex.edu domain; instead, either the university itself or someone hired as a third party must provide DNS for it. In this case, we are assuming the folks at Googleplex U. themselves run their own DNS. Without making any other cuts, the googleplex.edu domain would be a single zone containing everything below that name, including both finearts.googleplex.edu and compsci.googleplex.edu.

In our example, however, we would make another cut, between googleplex.edu and compsci.googleplex.edu. This, in effect, *liberates* compsci.googleplex.edu, allowing its administrators to be in charge of their own DNS server. In doing this, we end up with two distinct zones: one encompassing googleplex.edu, finearts.googleplex.edu, and admin.googleplex.edu (and everything underneath them) and another for compsci.googleplex.edu (and everything below it). This is illustrated in Figure 54-2.

## **The Impact of Zones on Name Resolution: Authoritative Servers**

The concept of zones is critical to understanding how DNS name servers work, and therefore, how name resolution is performed. All of the information about the subdomains and individual devices in the zone is represented using a set of resource records stored on a DNS name server. Usually, this name server is associated with the highest-level domain name in the zone. A name server that contains the definitive information for the zone is said to be *authoritative* for the zone.



**Figure 54-2: DNS zones of authority** Cuts can be made between nodes in the DNS name tree to create an arbitrary hierarchy of name authorities. This example shows the DNS tree branch for googleplex.edu, with each zone indicated using a different shading. IANA/ICANN is responsible for the root domain, and a separate authority named Educause takes care of .EDU. The third zone covers much of googleplex.edu, except that a cut has been made between googleplex and compsci to create an independent zone of authority for compsci.googleplex.edu.

An authoritative server for a zone is one that maintains the official information about the zone, and the one that is ultimately responsible for providing name resolution information about it. We'll discuss this in the section on DNS servers and name resolution in Chapter 56.

**KEY CONCEPT** The DNS name registration hierarchy is divided into regions called *zones of authority*. Each zone represents an area that is administered independently and consists of a contiguous segment of the DNS name tree.

Every DNS zone has a set of authoritative servers, which are usually a pair called the *primary* (or *master*) and *secondary* (or *slave*) servers. However, it is also possible for a single DNS name server to be authoritative for more than one zone.

As mentioned earlier, it is not always necessary for the actual owner of a domain to provide DNS services for it. Very often, especially for the domains owned by small businesses or individuals, DNS services are provided by a third party, often an ISP. For example, I have had pcguide.com registered as a domain since 1997, but my long-time web-hosting provider, pair Networks, has provided DNS services for

me since the beginning. This means that pair's DNS servers in the pair.com hierarchy are responsible for pcguide.com. They are also responsible for many other domains for the company's customers.

## DNS Private Name Registration

We have now reviewed the hierarchical nature of the DNS name space and the authority structure that administers it. Name registration begins with the generic and country code TLDs within the root of the name hierarchy, then proceeds to second-level domains within the TLDs and then lower-level subdomains below those. As we progress down the name tree, we move from the most general, public authority (IANA/ICANN, which runs all of DNS), through the high-level TLD authorities, and eventually down to the level of individual organizations, corporations, and individuals.

This dividing line between public authorities and private authorities occurs in many different places in the name structure. Wherever it does occur, below that line, responsibility for the domain becomes that of the organization that registered it. The organization can further subdivide the name space, granting parts of it to other organizations, or even reselling it. Alternatively, an organization may decide to use the name space to create a purely internal structure. I call this *private name registration*, in contrast to the *public name registration* described earlier in this chapter.

For example, if a company called XYZ Industries registers xyzindustries.com, that company becomes the owner of not just that domain name, but any subdomain structure or named items within it that the company may choose to create. This is the beauty and power of authority delegation and the hierarchical structure. The company has an important decision to make, however: whether they want to create names that are part of the global DNS name structure or use names within the structure purely privately.

### Using Publicly Accessible Private Names

If an organization's administrators want names within their domain to be part of the global DNS name structure, they must perform the work required to properly set up and manage these names so they fit into DNS. The most common example is creating a public World Wide Web server. Most companies name such servers beginning with www, so XYZ Industries would probably wish to have the name www.xyzindustries.com for its web server address.

Obviously, the XYZ Industries owners want and need anyone on the Internet to be able to locate this server. Thus, even though they have private control of the xyzindustries.com domain, and own the name www.xyzindustries.com, they must follow proper procedures for ensuring that DNS resource records are set up for their www subdomain so everyone on the Internet can find it. They may do this themselves, if they run their own DNS servers, or may have an ISP or other third party do it for them, as described earlier.

## **Using Private Names for Internal Use**

The alternative is to create purely private names for use only within the organization. For example, it is likely that even if XYZ wants a public web server, the administrators may wish to name many other machines that are to be accessed only within the company itself. In this case, they don't need to set up these machines so they are publicly recognizable. They can create private machine names and manage them internally within their own network.

**KEY CONCEPT** Once an organization registers a particular domain name, it becomes the owner of that name and can decide whether and how to create a substructure within that domain. If an organization wants objects in the domain to be accessible on the public Internet, it must structure its domain to be consistent with Internet DNS standards. Alternately, it can create a purely private domain using any structure and rules it prefers.

One common way to do this is to make use of the older host table name system. This system is now archaic for large internetworks, but is often still used in smaller companies due to its simplicity. A name is *registered* by being added to the host tables on each of the computers within the organization, and *resolved* when the operating system on a host checks this file prior to using standard DNS resolution methods. The host table supplements DNS in this case (it is not really a part of DNS). The two systems are complementary and can work together, as explained in Chapter 51.

## **Using Private Names on Networks Not Connected to the Internet**

Note that if you are running a purely private network not connected to the Internet at all, you can actually set up your own entirely private name hierarchy and run DNS yourself. In this case, you are in charge of the DNS root and can use any naming system you like.

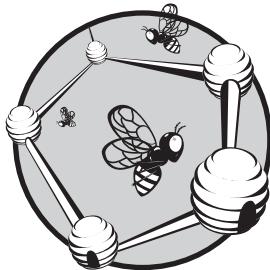
This approach is sometimes considered attractive, because you can then use very simple machine names on small networks, without needing to perform any public registration or use names that correspond to the global hierarchy. Instead of the accounting computer in XYZ Industries being named accounting.xyzindustries.com, internally it could be named accounting. You can mix these with real DNS names, too, when accessing resources. For example, Joe's machine could be called just joe, while the website of UPS would, of course, still be www.ups.com.

The most common example of this mixing of private and public names is the definition of the private local name for the loopback address of a computer. Most Windows and UNIX machines define the name *localhost* to be the address 127.0.0.1, which means "this computer" on any TCP/IP machine.



# 55

## DNS NAME SERVER CONCEPTS AND OPERATION



Of all the components and functional elements that combine to form the Domain Name System (DNS), name servers are arguably the most important. These servers, which may be either dedicated devices or software processes running on machines that also perform other tasks, are the workhorses of DNS. They store and manage information about domains, and respond to resolution requests for clients—in some cases, millions of times each day. Understanding how they perform this most basic task and the many support jobs for which they are also responsible is crucial to understanding DNS as a whole.

In this chapter, I describe the concepts related to DNS name servers and explain how they operate. I begin with an overview of DNS name server functions and general operation. I describe the way that DNS name server data is stored in resource records and the role of classes. I discuss the different roles of name servers in DNS and explain the all-important root name servers. I discuss how DNS zones are managed, the notions of domain contacts and zone transfers, and how caching and load balancing are used to improve efficiency

in DNS. I conclude with a brief outline of several enhancements to basic DNS server operation, including the new Notify and Update message types and incremental zone transfers.

**RELATED INFORMATION** *The information in this section should be considered complementary to that in the following chapter on DNS resolvers.*

## DNS General Operation

The three major functions of a name system are creating a name space, performing name registration, and providing name resolution services. The previous chapters describe how DNS uses a hierarchical tree structure for its name space (Chapter 53), and a hierarchical tree for name authorities and registration (Chapter 54). I'm sure that, given this, you will have to struggle to contain your surprise when I tell you that name resolution is also oriented around the notion of a hierarchical structure.

The devices that are primarily charged with performing the functions required to enable name resolution are *name servers*. They are arranged in a hierarchy that is closely related to the authority structure of the name system. Just as the authority structure complements the name structure but is not exactly the same as it, the name server architecture complements both the authority structure and the name structure, but may be different from them in its actual composition.

### **DNS Name Server Architecture and the Distributed Name Database**

In a large DNS implementation, information about domains is not centralized in a single database run by one authority. Instead, it is *distributed* across many different authorities that manage particular top-level domains (TLDs), second-level domains, or lower-level subdomains. In the case of the global Internet, literally millions of different authorities, many of them responsible only for their own local domain space, participate cooperatively in running the DNS system.

With authority for registration distributed in this manner, the information about domains is similarly spread among many entities, resulting in a *distributed database*. A key concept in DNS name resolution is that each entity that maintains responsibility for a part of the name space must also arrange to have that information stored on a DNS server. This is required so that the server can provide the information about that part of the name space when resolution is performed. As you can see, the existence of a structured hierarchy of authorities directly implies the need for a hierarchy of servers that store that hierarchical name information.

Each DNS zone of authority is required to have one or more DNS servers that are in charge of managing information about that zone. These servers are said to be *authoritative* for the zone. Storing information about the domains, subdomains, and objects in the zone is done by recording the data in special resource records that are read from DNS master lists maintained by administrators. Servers then respond to requests for this information.

**KEY CONCEPT** DNS public name information is stored in a *distributed database* of DNS name servers that are structured in a hierarchy comparable to the hierarchy of authorities. Each zone has one or more DNS name servers in charge of the zone's information, called *authoritative name servers*.

Since information in DNS is stored in a distributed form, there is no single server that has information about every domain in the system. As you'll see in the next chapter, the process of resolution instead relies on the hierarchy of name servers. At the top of the DNS hierarchy is the *root* domain, and in that domain are root name servers. These are the most important servers, because they maintain information about the TLDs within the root. They also have knowledge of the servers that can be used to resolve domains one level below them. Those servers, in turn, are responsible for the TLDs and can reference servers that are responsible for second-level domains. Thus, a DNS resolution may require that requests be sent to more than one server.

## DNS Server Support Functions

The storing and serving of name data (through responses to requests from DNS resolvers) is the main function of a DNS server. However, other support jobs are also typically required of a DNS server:

**Interacting with Other Servers** Because the DNS resolution process often requires that multiple servers be involved, servers must maintain not just name information, but information about the existence of other servers. Depending on the type of DNS request, servers may themselves become clients and generate requests to other servers.

**Zone Management and Transfers** The server must provide a way for DNS information within the zone to be managed. A facility also exists to allow a *zone transfer* to be performed between the master (primary) server for a zone and slave (secondary) servers.

**Performance Enhancement Functions** Due to the large number of requests servers handle, they employ numerous techniques to reduce the time required to respond to queries. The most important of these is caching of name information. A variation of regular caching called *negative caching* may also be used to improve performance, and load balancing is a feature that can be used to improve efficiency of busy devices registered within the DNS system.

**Administration** Various other administrative details are required of name servers, such as storing information about the different types of contacts (humans) who are responsible for certain tasks related to management of a domain or zone.

As you'll see later in this chapter, not all name servers perform all of these tasks described; some perform only a subset.

## **The Logical Nature of the DNS Name Server Hierarchy**

Like the other hierarchies, the name server hierarchy is logical in nature. I already mentioned that it often is not exactly the same as the authority hierarchy. For one thing, it is common for a single DNS name server to be the authoritative server for a number of domains. Even if a particular group has authority for a subdomain of a particular domain, it's possible they will share the DNS servers with the authority of their parent domain for efficiency reasons. For example, a university might delegate control over parts of its domain space to different groups (as in the example of DNS zones in Chapter 54) but still manage all subdomains on the same server. In practice, the lower the level of the subdomain in the DNS name hierarchy, the less likely that subdomain has its own DNS server.

Another important aspect of the logical nature of the name server hierarchy is that there is no necessary relationship between the structure of the name servers and their location. In fact, in many cases, name servers are specifically put in different places for reliability reasons. The best example of this is the set of root name servers. These are all at the top of the DNS server architecture, but they are spread around the globe to prevent a single problem from taking all of them out. Also remember not to be fooled by the structure of a name in the geopolitical DNS name hierarchy (as discussed in Chapter 53). A name server called ns1.blahblah.ca might be in Canada, but it very well might not be located there.

**KEY CONCEPT** The DNS name server hierarchy is logical in nature and not exactly the same as the DNS name server tree. One server may be responsible for many domains and subdomains. Also, the structure of the DNS name server hierarchy doesn't necessarily indicate the physical locations of name servers.

## **DNS Name Server Data Storage**

One of the most important jobs performed by name servers is the storage of name data. Since the authority for registering names is distributed across the internetwork using DNS, the database of name information is likewise distributed. An *authoritative* server is responsible for storing and managing all the information for the zones of authority it is assigned.

Each DNS server is, in essence, a type of database server. The database contains many kinds of information about the subdomains and individual devices within the domain or zone for which the server is responsible. In DNS, the database entries that contain this name information are called *resource records (RRs)*. A specific set of RRs is associated with each node within the zone.

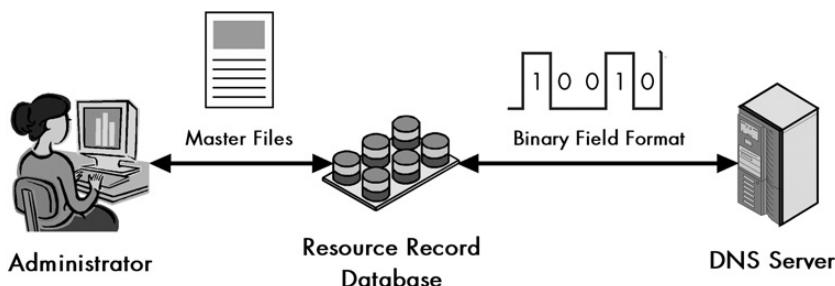
### ***Binary and Text Representations of Resource Records***

The entire point of DNS is to allow humans to work with names and computers to work with numbers. This principle is further reflected in the two very different representations that exist for the DNS RRs themselves (see Figure 55-1):

**RR Field Format (Binary) Representation** Name servers are required to respond to queries for name information by sending RRs within DNS messages. Obviously, we want to do this in as efficient a way as possible, so each RR is internally stored

using a special field format that is similar to the many field formats used for messages in other protocols. All RRs use a general field format for some of their fields and then have a unique portion that is specific to the RR type.

**Master File (Text) Representation** Computers are happy to exchange binary-encoded field formats and have no problem remembering that, for example, RR type 15 corresponds to a mail exchange (MX) record. However, human administrators want to be able to quickly and easily maintain DNS information without needing to remember cryptic codes or work with binary values. For this reason, DNS uses a *master file* format for its user-machine interface, which allows RRs to be specified in text form for easier maintenance.



**Figure 55-1:** DNS RR master file and binary field formats

To meet the needs of humans and computers, DNS uses two representations for the data stored in RRs. Administrators enter and maintain information in textual DNS master files. These are read by DNS server software and internally stored in binary format for answering DNS requests.

### **Use of RRs and Master Files**

Each node may have a variable number of records, depending on the node type and what information is being kept for it. The RRs are added, changed, or deleted when DNS information changes, by administrators who make modifications to the text master files on the server computer. These files are then read into memory by the DNS server software, parsed (interpreted), and converted into binary form. Then they are ready for use in resolving DNS name requests and other queries. I describe both the binary RR field formats and master file format in Chapter 57.

**KEY CONCEPT** DNS name servers store DNS information in the form of *resource records (RRs)*. Each RR contains a particular type of information about a node in the DNS tree. There are two representations for RRs: Conventional binary field formats are used for communication between DNS name servers and resolvers, and text *master files* are edited by administrators to manage DNS zones.

## Common RR Types

The main DNS standards, RFC 1034 and 1035, defined a number of RR types. Over time, the list has changed, with new RR types being created in subsequent standards and the use of others changed. Like other Internet parameters, the list of DNS RR types is maintained in a file by the Internet Assigned Numbers Authority (IANA). Also like other Internet parameters, there are actually several dozen defined RRs in DNS, but only a few are commonly used; others are now obsolete, used for special purposes, or experimental in nature. The current list of DNS resource records is maintained in a file that can be found at <http://www.iana.org/assignments/dns-parameters>.

Table 55-1 summarizes the most important RR types. For each, I have shown the numeric Type value for the record, which is used to identify the RR type in message exchanges, and the text code used for the RR in master files.

**Table 55-1:** Summary of Common DNS Resource Records

RR Type Value	RR Text Code	RR Type	Description
1	A	Address	Contains a 32-bit IP address. This is the “meat and potatoes” of DNS, since it is where the address of a node is stored for name resolution purposes.
2	NS	Name Server	Specifies the name of a DNS name server that is authoritative for the zone. Each zone must have at least one NS record that points to its primary name server, and that name must also have a valid Address (A) record.
5	CNAME	Canonical Name	Used to allow aliases to be defined that point to the real name of a node. The CNAME record provides a mapping between this alias and the canonical (real) name of the node. It is commonly used to hide changes in the internal DNS structure from outside users, by letting them use an unchanging alias, while the internal names are modified based on the needs of the organization. See the discussion of name resolution in Chapter 56 for an example.
6	SOA	Start Of Authority	Used to mark the start of a DNS zone and provide important information about it. Every zone must have exactly one SOA record, which contains the name of the zone, its primary (master) authoritative server name, and technical details such as the email address of its administrator and parameters for how often slave (secondary) name servers are updated.
12	PTR	Pointer	Provides a pointer to another location in the name space. These records are best known for their use in reverse resolution through the IN-ADDR.ARPA domain (described in Chapter 54).
15	MX	Mail Exchange	Specifies the location (device name) that is responsible for handling email sent to the domain.
16	TXT	Text String	Allows arbitrary additional text associated with the domain to be stored.

All of these RRs are used in different ways to define zones and devices within them and then permit name resolution and other functions to take place. You’ll see how they are used in more detail in Chapter 56, which covers name resolution. You can also find a more lengthy description of some of them in the section in Chapter 57 devoted to RR field formats.

**RELATED INFORMATION** See the topic on IPv6 DNS support near the end of Chapter 57 for IPv6-specific RR types.

## **RR Classes**

Finally, I would like to mention a historical note about RRs. When DNS was first created, its inventors wanted it to be as generic as possible. To that end, they designed it so that a DNS server could, theoretically, provide name service for more than one type of underlying protocol; that is, DNS could support TCP/IP as well as other protocols simultaneously.

Of course, protocols have different addressing schemes and also varying needs for name resolution. Therefore, DNS was defined so that each protocol could have a distinct set of RR types. Each set of RR types was called a *class*. Technically, an RR must be identified using both a class identifier and an RR type. Like the RR types, classes have a numeric code number and a text abbreviation. The class for TCP/IP uses the number 1, with the text code IN (for Internet).

In practice, this notion of multiple classes of RRs never took off. Today, DNS is, to my knowledge, used only for TCP/IP. (There may be some obscure exceptions.) Several other classes have been defined by RFC 1035 and are in the IANA DNS parameters list, but they are for relatively obscure, experimental, or obsolete network types, with names such as CSNET, CHAOS, and Hesiod. You'll still see this concept of class in the specification of DNS message and RR formats, but there really is only class today: IN for TCP/IP. For this reason, in most cases, the class name can be omitted in DNS-related commands and data entries, and IN will be assumed by default.

**KEY CONCEPT** The DNS standards were originally created to allow them to work with multiple protocols, by specifying the class of each RR. Today, the only class commonly used is that for TCP/IP, which is called IN (for Internet).

## **DNS Name Server Types and Roles**

So far, we have looked at the functions of DNS servers, focusing on the important job of storing name server information. There are many thousands of DNS servers on the Internet, and not all are used in the same way. Each DNS server has a particular role in the overall operation of the name system. The different kinds of servers also interact with each other in a variety of ways.

### ***Master (Primary)/Slave (Secondary) Servers***

Every zone needs to have at least one DNS name server that is responsible for it. These DNS name servers are called *authoritative* servers for the zone, because they contain the full set of RRs that describe the zone. When any device on the Internet wants to know something about a zone, it consults one of its authoritative servers.

From a strictly theoretical perspective, having one name server for each zone or domain is sufficient to provide name resolution services for the entire DNS name structure. From an implementation standpoint, however, having only one name server for each part of the name space is not a wise idea. Instead, each zone usually

has associated with it at least two name servers: one *primary* or *master* name server, and one *secondary* or *slave* name server. Some zones may have more than one secondary name server.

**NOTE** *The terms primary and secondary are used often in the DNS standards to refer to the roles of the two authoritative servers for a zone. However, master and slave are now the preferred terms, because primary and secondary are somewhat ambiguous and used in other contexts. You should be prepared to see both terms used.*

The master name server is obviously the most essential server. It is on this name server that the master files for the zone's RRs are maintained, so the master name server is the final word on information on the zone. However, there are several reasons why slave servers are also important:

**Redundancy** If there were only one name server and it failed, no one would be able to resolve names such as www.xyzindustries.com into IP addresses, and that would be a Bad Thing. Slave name servers act as a backup for the masters they support. Redundancy is the most important consideration in setting up master and slave name servers. Sticking two machines side by side in a server room, plugged into the same electrical service, both connected to the Internet with the same Internet service provider (ISP), and making one your master DNS server and the other your slave is not a smart move. Ideally, the primary and secondary servers should be as independent as possible; they should be physically distant and have separate connections to the Internet.

**Maintenance** With more than one server, we can easily take the primary server down for maintenance when needed without name resolution service being disrupted.

**Load Handling** Busy zones can use multiple servers to spread the load of name resolution requests to improve performance.

**Efficiency** There are many cases where there is an advantage to positioning a name server in a particular geographical location for the sake of efficiency. For example, a company may have an office in a distant location connected using a low-speed wide area network link. To reduce name resolution traffic across that link, it makes sense to have that zone's information available in a name server on both sides of the connection, which would require two physical servers.

Just as the names *master* and *slave* suggest, the secondary name servers are not the original source of information about a zone. They normally obtain their RRs not from human-edited master files, but from updates from the master server. This is accomplished using a process called a *zone transfer*. These transfers are performed on a regular basis to ensure that the slave servers are kept up-to-date. The slaves can then respond to name resolution requests with current information. Both the master and the slave are considered authoritative for the zone.

## Name Server Roles

The master and slave roles for a zone are logical and do not always correspond to individual physical hardware devices. A single physical name server can play multiple roles in the following cases:

- It can be the master name server for more than one zone. Each zone in this case has a distinct set of RRs maintained in separate master files.
- It can be a slave name server for more than one zone.
- It can be a slave name server for certain zones as well as a primary for others.

Note, however, that a single physical name server cannot be a master and a slave server for the same zone.

**KEY CONCEPT** The master DNS server for a zone is its primary server, which maintains the master copy of DNS information. Most DNS zones also have at least one slave or secondary DNS server. These are important because they serve as backups for the primary server, and they can also help share the load of responding to requests in busy zones. Secondary name servers get their information from primary servers on a routine basis. Both master and slave servers are considered authoritative for the zones whose data they maintain.

## Caching-Only Name Servers

For efficiency, all DNS servers—both masters and slaves—perform caching of DNS information so it can be used again if requested in the near future. (Caching is described in the “Name Server Caching” section later in this chapter.) The importance of caching is so significant that some servers are set up only to cache information from other DNS servers. Unsurprisingly, these are called *caching-only* name servers.

These name servers are not authoritative for any zone or domain, and they don’t maintain any RRs of their own. They can answer name resolution requests only by contacting other name servers that *are* authoritative and then relaying the information. They then store the information for future requests. Why bother? The reason is performance. Through strategic placement, a caching-only server can increase DNS resolution performance substantially in some networks by cutting down on requests to authoritative servers.

**KEY CONCEPT** There are DNS servers that do not maintain DNS RRs of their own but solely hold recently used information from other zones. These are called *caching-only* name servers and are not authoritative for any zone.

## DNS Zone Management, Contacts, and Zone Transfers

The authority for a particular DNS zone is responsible for performing a variety of tasks to manage it. *Zone management* encompasses the entire gamut of jobs related to a zone: deciding on the name hierarchy within the zone, specifying procedures for name registration, technical work related to keeping DNS servers running, and

other administrative overhead of all sorts. This job can be either very small or incredibly large, depending on the type of organization. A small domain owned by an individual doesn't require much work to manage, while one for a huge company might require a dedicated staff to maintain.

## **Domain Contacts**

It is important that it be possible for anyone on an internetwork to be able to determine who the owner of a domain is, so that person can be reached for whatever reason. On the Internet, each DNS domain has associated with it a set of three *contacts* that are responsible for different facets of managing a domain:

**Administrative Contact** The main contact, responsible for the domain as a whole. This individual or organization is considered the overall owner of the domain.

**Billing Contact** A contact responsible for handling payment for domain services and other accounting matters.

**Technical Contact** A contact who handles the technical details of setting up DNS for the domain and making sure it works.

For smaller domains, there usually is no separate billing contact; it is the same as the administrative contact. In contrast, the technical contact is often different from the administrative contact in both large and small domains. Large organizations will make the technical contact someone in their information technology department. Small organizations often let their ISP provide DNS services, and in that case, the technical contact will be someone at that ISP.

**KEY CONCEPT** Each DNS domain has associated with it a set of three contact names that indicate who is responsible for managing it. The *administrative contact* is the person with overall responsibility for the domain. The *billing contact* is responsible for payment issues; this may be the same as the administrative contact. The *technical contact* is in charge of technical matters for the domain and is often a different person than the administrative contact, especially when DNS services are outsourced.

## **Zone Transfers**

The ultimate purpose of zone management is to ensure that information about the zone is kept current on the zone's master and slave name servers, so it can be efficiently provided to name resolvers. Thus, the management of a zone begins with decision-making and administrative actions that result in changes to the RRs for the zone. These are reflected in changes made to the DNS master files on the master (primary) DNS server for the zone.

In contrast, each zone's secondary DNS server(s) act as slaves to the master primary server. They carry information about the zone, but do not load it from local master files that are locally edited. Instead, they obtain their information from the master name server on a regular basis. The procedure responsible for this is called a *zone transfer*.

The records on the master name server can be updated at any time. As soon as the master name server's records have been changed, the information at the slave name servers becomes partially out-of-date. This is not generally a big deal, because most of the data will still be accurate, and the secondary server will continue to respond to resolution requests using the most current information it has. However, it is obviously important that we update the slave servers on a regular basis; if this is not done, eventually their data will become stale and unreliable. To this end, it is necessary that zone transfers be performed on a regular basis.

### **Control of When Zone Transfers Occur**

Controlling when zone transfers happen requires implementation of a communication process between the servers that consists of two basic parts. First, we need a mechanism to allow slave servers to regularly check for changes to the data on the master. Second, we must have a mechanism for copying the RRs for the zone from the primary name server to the secondary server when needed.

Both mechanisms make use of standard DNS query/response facilities and special fields in the RRs for the zone. Of particular importance is the Start Of Authority (SOA) record for the zone, which contains several parameters that control zone status checking and zone transfers. While the formal description of these parameters can be found in the description of RR formats in Chapter 57, I'll discuss how they are used here.

When a slave name server starts up, it may have no information about the zone at all, or it may have a copy of the zone's RRs stored on its local storage, from the last time it was running. In the former case, it must immediately perform a full zone transfer, since it has no information. In the latter case, it will read its last-known copy of the zone from local storage; it may immediately perform a *poll* on the master server to see if the data has changed, depending on configuration. A poll is done by requesting the SOA RR for the zone.

The Serial field in the SOA record contains a serial number (which may be arbitrary or may be encoded so it has a particular meaning) that acts as the version number of the master server's zone database. Each time the master file for the zone is modified (either manually by editing or automatically through another means), this serial number is increased. Therefore, a slave server can detect when changes have been made on the master by seeing if the Serial field in the most recent SOA record is greater than the one the slave stored the last time it polled the master. If the serial number has changed, the slave begins a zone transfer.

Three other fields in the SOA record control the timing that slave name servers use for polling and updating their information:

**Refresh** This field specifies how many seconds a slave server waits between attempts to check for an update on the master. Assuming the slave can make contact, this is the longest period of time that data on a slave will become stale when the master changes.

**Retry** This field controls how long the slave must wait before trying again to check in with the master if its last attempt failed. This is used to prevent rapid-fire attempts to contact a master that may clog the network.

**Expire** If, for whatever reason, the slave name server is not able to make contact with the master for a number of seconds given by this field's value, it must assume that the information it has is stale and stop using it. This means that it will stop acting as an authoritative name server for the zone until it receives an update.

The fact that these parameters are part of the SOA record for the zone gives the administrator of the zone complete control over how often master name servers are updated. In a small zone where changes are rare, the interval between checks made by the slave servers can be increased; for larger zones or ones that are changed often, the Refresh interval can be decreased.

### Zone Transfer Mechanism

When a zone transfer is required, it is accomplished using a DNS query sent to the master server using the regular DNS query/response messaging method used for name resolution (discussed in the next section). A special DNS question type, called AXFR (address transfer) is used to initiate the zone transfer. The server will then transfer the RRs for the zone using a series of DNS response messages (assuming that the server that requested the transfer is authorized to do so). Since it's important that zone transfers be received reliably, and since the amount of data to be sent is large and needs to be managed, a Transmission Control Protocol (TCP) session must first be established and used for zone transfers. This is in contrast to the simpler User Datagram Protocol (UDP) transport used for regular DNS messages (as described in the section discussing the use of UDP and TCP for DNS at the start of Chapter 57).

Once the zone transfer is complete, the slave name server will update its database and return to regular operation. It will continue to perform regular polls of the master server every Refresh seconds. If it has a problem with a regular poll, it will try again after Retry seconds. Finally, if an amount of time equal to Expires seconds elapses, the slave name server will stop serving data from the zone until it reestablishes contact with the primary name server.

**KEY CONCEPT** Slave name servers do not have their DNS information managed directly by an administrator. Instead, they obtain information from their master name server on a periodic basis through a process called a *zone transfer*. Several fields in the Start Of Authority (SOA) DNS RR control the zone transfer process, including specifying how often transfers are done and how slave name servers handle problem conditions such as an inability to contact the master server.

Note that the DNS *Notify* feature is an enhancement to the basic zone status check/zone transfer model. It allows the master server to notify a slave server when the master's database has changed. Another new feature allows only part of a zone to be transferred instead of the entire zone. See the discussion of DNS name server enhancements later in this chapter for more information.

## DNS Root Name Servers

DNS is strongly oriented around the notion of hierarchical structure. The name space, registration authorities, and name servers are all arranged in a tree structure. Like these structures, the name resolution process is also hierarchical. As explained in Chapter 53, a fully qualified domain name (FQDN) is resolved by starting with the least specific domain name element (label) and working toward the most specific one.

Naturally, the least specific portion of every name is the root node under which the entire DNS structure exists. This means that, absent caching and other performance enhancements, all name resolution begins with the root of the name tree. We find here a set of name servers that are responsible for name server functions for the DNS root: the DNS *root name servers*.

Like all name servers, DNS root name servers store information about and provide name resolution services for all the nodes within the root zone. This includes certain specific TLDs and subdomains. Most TLDs, however, are in their own zones. The root name servers are used as the “go-to” spot to obtain the names and addresses of the authoritative servers for each of these TLDs. For example, if we want to resolve the name www.xyzindustries.co.uk, the root name servers are where a resolver would find the identity of the name server that is responsible for .UK.

### ***Root Name Server Redundancy***

Clearly, these root name servers are extremely important to the functioning of the DNS system as a whole. If anything were to ever happen to cause the root name servers to stop operating, the entire DNS system would essentially shut down. For this reason, there obviously isn’t just one root server, nor are there two or three; there are (at present) thirteen different root name servers.

In fact, there are actually far more than 13 physical servers. Most of the 13 name servers are implemented as clusters of several independent physical hardware servers. Some are distributed collections of servers that are in different physical locations. The best example is the F root server, which has been implemented as a set of more than a dozen *mirrors* in various places around the world, to provide better service.

The principles of redundancy that are a good idea for choosing a secondary name server for a regular domain obviously apply that much more to the root. This is why the various physical devices that compose the 13 root servers are all located in different places all around the globe. Many of them are in the United States, but even these are in many locations throughout the country (albeit concentrated in a couple of hot spots in California and near Washington, DC) and are set up to use different networks to connect to the Internet.

The root name servers are, of course, rather powerful. Despite there being several dozen pieces of hardware to spread the load, they must each handle large amounts of data, 24 hours a day. They are run by networking professionals who ensure that they function efficiently. An Internet standard, RFC 2870, “Root Name Server Operational Requirements,” spells out the basic rules and practices

for the operation of these name servers. It specifies extensive procedures for ensuring the security of the servers and for avoiding performance problems due to their pivotal role.

**KEY CONCEPT** Information about the DNS root and its TLDs is managed by a set of *root name servers*. These servers are essential to the operation of DNS. They are arranged into 13 groups and physically distributed around the world.

Despite all the efforts taken to ensure that the root servers are widely distributed and secure, they still collectively represent a point of weakness in the global Internet. Millions and millions of people depend on these servers. There have been incidents in the past where rogue elements on the Internet have attempted to disrupt DNS by attacking the root name servers. One widely publicized incident was a denial-of-service (DoS) attack against the root servers on October 21, 2002. The attack failed, but it significantly raised awareness of the importance of these servers and how essential DNS security is.

### **Current Root Name Servers**

Originally, the root name servers were given domain names reflecting the organizations that ran them. In these historical names, we can see a veritable who's who of the big players in the development of the Internet: the Information Sciences Institute (ISI), National Aeronautics and Space Administration (NASA), United States military, and others. Several of the servers are still run by government agencies or the United States military, where added security can be put into place to protect them. For convenience, however, all the root name servers are now given alphabetical letter names in the special domain root-servers.net.

Table 55-2 shows the most current information about the DNS root name servers as of the date of publishing of this book. For your interest and amusement, I have also mapped the locations of these servers in Figure 55-2 .

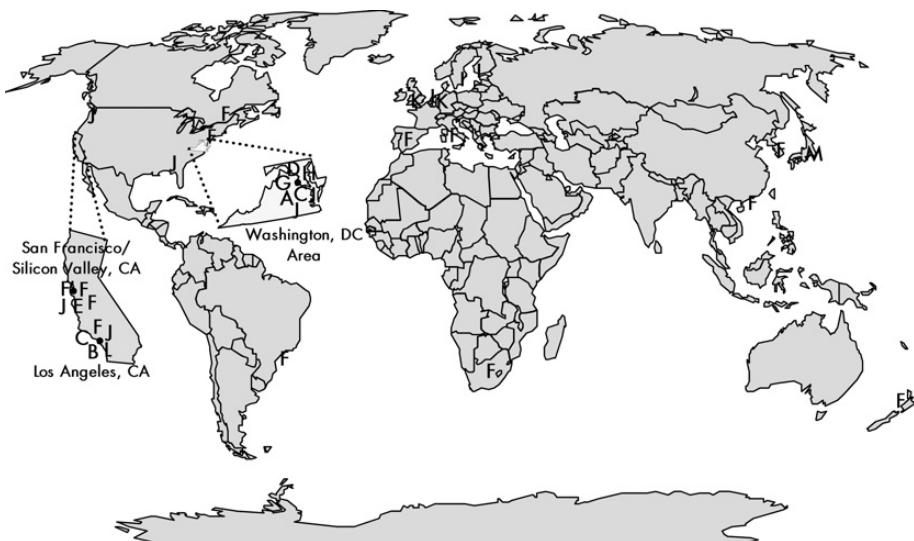
**Table 55-2:** Internet DNS Root Name Servers

<b>Root Server Name</b>	<b>IP Address</b>	<b>Historical Name</b>	<b>Location(s)</b>
a.root-servers.net	198.41.0.4	ns.internic.net	Dulles, VA, U.S.
b.root-servers.net	128.9.0.107	ns1.isi.edu	Marina Del Rey, CA, U.S.
c.root-servers.net	192.33.4.12	c.psi.net	Herndon, VA and Los Angeles, CA, U.S.
d.root-servers.net	128.8.10.90	terp.umd.edu	College Park, MD, U.S.
e.root-servers.net	192.203.230.10	ns.nasa.gov	Mountain View, CA, U.S.
f.root-servers.net	192.5.5.241	ns.isc.org	Auckland, New Zealand; Sao Paulo, Brazil; Hong Kong, China; Johannesburg, South Africa; Los Angeles, CA, U.S.; New York, NY, U.S.; Madrid, Spain; Palo Alto, CA, U.S.; Rome, Italy; Seoul, Korea; San Francisco, CA, U.S.; San Jose, CA, U.S.; Ottawa, ON, Canada

*(continued)*

**Table 55-2: Internet DNS Root Name Servers (continued)**

<b>Root Server Name</b>	<b>IP Address</b>	<b>Historical Name</b>	<b>Location(s)</b>
g.root-servers.net	192.112.36.4	ns.nic.ddn.mil	Vienna, VA, U.S.
h.root-servers.net	128.63.2.53	aos.arl.army.mil	Aberdeen, MD, U.S.
i.root-servers.net	192.36.148.17	nic.nordu.net	Stockholm, Sweden; Helsinki, Finland
j.root-servers.net	192.58.128.30	—	Dulles, VA, U.S.; Mountain View, CA, U.S.; Sterling, VA, U.S.; Seattle, WA, U.S.; Atlanta, GA, U.S.; Los Angeles, CA, U.S.; Amsterdam, The Netherlands
k.root-servers.net	193.0.14.129	—	London, UK; Amsterdam, The Netherlands
l.root-servers.net	198.32.64.12	—	Los Angeles, CA, U.S.
m.root-servers.net	202.12.27.33	—	Tokyo, Japan



**Figure 55-2: Geographic locations of Internet DNS root name servers**

The current list of root name servers can be found in the file `ftp://ftp.rs.internic.net/domain/named.root`. You can also find the information in a more user-friendly format at <http://www.root-servers.org>.

## DNS Name Server Caching

Most of the grunt work done by name servers is responding to name resolution requests. Busy servers—like the root name servers, the ones that carry zone information for the TLDs, and ones that serve very busy zones—must handle hundreds or even thousands of name resolution requests each *second*. Each of these requests takes time and resources to resolve and takes internetwork bandwidth away from the business of transferring data. It is essential, therefore, that DNS server implementations employ mechanisms to improve their efficiency and cut down on unnecessary name resolution requests. One of the most important of these is *caching*.

## **Name Server Caching**

The word *cache* refers to a store, or a place where something is kept. In the computer world, the term usually refers to an area of memory set aside for storing information that has been recently obtained so it can be used again. In the case of DNS, caching is used by DNS name servers to store the results of recent name resolution and other requests, so that if the request occurs again, it can be satisfied from the cache without requiring another complete run of the name resolution process. Due to how most people use computers, a particular request is often followed by another request for the same name, so caching can significantly reduce the number of requests that result in complete name resolution procedures.

An example is the best way to illustrate this. Suppose you are using a host on your company's local network. This host is probably configured to use your company's DNS name server to handle resolution requests. You type `www.xyzindustries.com` into your web browser, which causes a resolution attempt to be made for that address. Most likely, your local DNS server doesn't know that name, so it will follow the complete name resolution process (described in Chapter 56) to get its address. After doing this, your local DNS server will *cache* the name `www.xyzindustries.com` and the address associated with it.

If you click a link for a page at that website, that new page will also probably be somewhere at the `www.xyzindustries.com` site. This will result in another DNS resolution request being sent off to your local DNS server. However, this time, the local server will not need to perform a resolution. It remembers that this name is in its cache and returns the saved address for the name immediately. Voilà! You get your answer faster, and unnecessary Internet traffic is avoided.

**KEY CONCEPT** *Caching* is an essential efficiency feature that reduces DNS message traffic by eliminating unnecessary requests for recently resolved names. Whenever a name is resolved, the resulting DNS information is cached so it can be used for subsequent requests that occur shortly thereafter.

Of course, things aren't entirely this simple. One very important issue that comes up with every caching system, including the one used in DNS, is the matter of the *freshness* of the cache.

### ***Caching Data Persistence and the Time to Live Interval***

Suppose your local DNS server resolves the name `www.xyzindustries.com`, and then caches its address. In this example, where you click a link a few seconds after the XYZ Industries home page loads, you aren't likely to be too concerned about how fresh the DNS data is. But how about if you shut down your computer to go on vacation for two weeks, and then come back to work and type the name into your browser again. If your local server still has the name in its cache, how do you know the IP address of `www.xyzindustries.com` hasn't changed during that two-week period?

Two different mechanisms are used to address this issue. The first is that when data is cached, the caching server also makes a note of the authoritative server from which it came. When a resolver (client) asks for a name resolution and the address is drawn from the cache, the server marks the answer as *non-authoritative* to clearly tell the client that the name came from the cache. The server also supplies the name of the authoritative server that originally supplied the data.

The client then has a choice: It can either use the non-authoritative answer or issue a request for a fresh name resolution from the authoritative server. This is a trade-off between performance (using the cached data) and currency (asking for a fresh resolution each time). Usually, the cached data can be used safely, because DNS information doesn't change very often.

The second technique for ensuring that caching data doesn't get too old is a procedure for limiting the *persistence* of DNS cached data. Each RR has associated with it a time interval, called the *Time to Live (TTL)*. Whenever an RR is read from a server, the TTL for the record is also read. Any server caching the record is supposed to discard the record after that time interval expires.

Each zone also has associated with it a default value for the TTL field to be applied to all records in the zone. This allows an administrator to select a TTL value for all records in a zone without needing to enter TTL numbers for each record individually. At the same time, the administrator can assign an override TTL value to any records that need a number that is different from the default. This default TTL was originally kept in the special SOA RR for each zone, but is now handled using a special directive in the zone's master file.

**NOTE** *This Time to Live (TTL) field is not related to the one used in Internet Protocol (IP) datagrams (see Chapter 21). Obviously, IP and DNS are totally different protocols, but more than that, the TTL fields in IP and DNS don't have the same meaning at all.*

It's worth emphasizing that DNS gives control over caching to the owner of the record, not whoever is running the DNS server doing the caching. While it is possible for a particular caching server to override the TTL and specify how long data will be held in its own cache, DNS is not supposed to work that way. The ability to specify a TTL on a record-by-record basis allows the persistence of cache data to be tailored to the needs of the individual data elements. Data that changes often can be given a small TTL value; infrequently modified records can be given a higher TTL. Selecting the TTL value must be done carefully. This is another trade-off between performance (which is optimized with higher TTL values, reducing the number of queries made for cached data) and freshness of the data (which increases as the TTL values are lowered).

**KEY CONCEPT** Cached information can become stale over time and result in incorrect responses sent to queries. Each RR can have associated with it a time interval, called the *Time to Live (TTL)*, that specifies how long the record may be held in a cache. The value of this field is controlled by the owner of the RR, who can tailor it to the specific needs of each RR type.

## **Negative Caching**

Classic DNS caching stores only the results of successful name resolutions. It is also possible for DNS servers to cache the results of *unsuccessful* name resolution attempts; this is called *negative caching*. To extend the example we've been using in this section, suppose you mistakenly thought the name of the company's website was www.xyz-industries.com and typed that into your browser. Your local DNS server would be unable to resolve the name and would mark that name as unresolvable in its cache—a negative cache entry.

Suppose you typed the name in incorrectly because someone mistyped it on an internal memo. If a colleague later tried the same name, the DNS server would say, "I already know this is a bogus name," and not try to resolve it again. Since there is no RR for an invalid name, the server itself must decide how long to cache this negative information. Negative caching improves performance because resolving a name that doesn't exist takes resources, just as resolving an existing one does. Note that regular caching is sometimes called *positive caching* to contrast it with negative caching.

The value to be used for negative caching in a zone is now specified by the Minimum field in the SOA RR for each zone. As mentioned in the previous section, this was formerly used to specify the default TTL for a zone.

## **DNS Name Server Load Balancing**

The Address (A) RR is the most fundamental one in DNS, since it records an actual mapping between a domain name and an IP address. Let's consider for a moment one of the words in that sentence in more detail. No, I don't mean *address* or *RR* or *mapping*. I mean the word *an*!

The Address record mentions only a single address for each domain name. This means that each domain name maps to only a single physical hardware device. When the number of requests that a particular server or other device needs to handle is relatively small, this is not a problem; the function can usually be implemented using a single physical hardware device. If the server gets busier, the usual solution is to throw more hardware at the problem—get a bigger machine.

However, some hosts on a large internetwork, especially the Internet, feature servers that must handle tremendous amounts of traffic from many clients. There simply is no single hardware device that can readily handle the traffic of a site like www.cnn.com or www.microsoft.com, for example, without becoming unwieldy. Sites like these must use a technique called *load balancing* to spread requests across multiple hardware servers.

### **Using Multiple Address Records to Spread Out Requests to a Domain**

One simple way to do load balancing would be to have multiple machine names. For example, CNN could create several different websites called www1.cnn.com, www2.cnn.com, and so on, each pointing to a different hardware device. DNS certainly supports this type of solution. The problem with this solution is that it is cumbersome; it requires users to remember multiple server names.

It would be better if we could balance the load automatically. DNS supports this by providing a simple way to implement load balancing. Instead of specifying a single Address RR for a name, we can create several such records, thereby associating more than one IP address with a particular DNS name. When we do this, each time the authoritative name server for the zone in which that name exists resolves that name, it sends all the addresses on the list back to the requester. The server changes the order of the addresses supplied in the response, choosing the order randomly or in a sequential, round-robin fashion. The client will usually use the first address in the list returned by the server, so by changing the list, the server ensures that requests for that device's name are resolved to multiple hardware units.

**KEY CONCEPT** Rather than creating a single Address (A) RR for a DNS domain name, it is possible to create multiple ones. This associates several IP addresses with one name, which can be used to spread a large number of requests for one domain name over many physical IP devices. This allows DNS to implement load balancing for busy Internet servers.

As Internet traffic increases, load balancing is becoming more popular. In early 2003, I saw a survey that indicated approximately 10 percent of Internet names at that time used load balancing—a fairly significant number. Most employed either two or three addresses, but some used as many as sixty addresses! Incidentally, at last check, [www.cnn.com](http://www.cnn.com) was associated with eight different IP addresses. (You can check the number of addresses associated with a name using the host command, as described in Chapter 88.)

### **Using Multiple DNS Servers to Spread Out DNS Requests**

The term *DNS load balancing* also has a completely different meaning from what I described in the previous section. In the discussion of DNS server roles, I talked about how each zone should have at least one slave (secondary) DNS server in addition to the master (primary) server. The usually stated main reason for this is redundancy, in case something happens to cause the master server to fail. However, having a slave server can also allow the load of DNS resolution requests to be balanced between the servers. In fact, some busy domains have more than two servers specifically for this reason.

Thus, *DNS load balancing* can refer to either using DNS to spread the load of requests (such as web page requests) to a device that is named using DNS or to spreading the load of DNS requests themselves.

## **DNS Name Server Enhancements**

The fundamentals of operation of DNS servers, as explained in the preceding sections in this chapter, are specified in the main DNS standards, RFC 1034 and 1035. These documents are pretty old by computer industry standards; they were published in 1987. To the credit of the designers of DNS, most of what they originally put into the DNS protocol is still valid and in use today. The creators of DNS

knew that it had to be able to scale to a large size, and the system has successfully handled the expansion of the Internet to a degree far beyond what anyone could have imagined 15 or so years ago.

As originally defined, DNS requires that DNS information be updated manually by editing master files on the master server for a zone. The zone is then copied in its entirety to slave servers using the polling/zone-transfer mechanism described earlier in this chapter. This method is satisfactory when the internetwork is relatively small and changes to a zone are made infrequently. However, in the modern Internet, large zones may require nearly constant changes to their RRs. Hand-editing and constantly copying master files can be impractical, especially when they grow large, and having slave servers get out of date between zone transfers may lead to reliability and performance concerns. For these reasons, several enhancements to the operation of DNS servers have been proposed over the years. We'll take a closer look at three of them here: DNS Notify, incremental zone transfers, and Dynamic DNS.

### ***Automating Zone Transfers: DNS Notify***

The first problem that many DNS administrators wanted to tackle was the reliance on polling for updating slave name servers. Imagine that you placed an order for a new music CD at your favorite online music store, but it was out of stock—backordered. Which makes more sense: having you call them every six hours to ask if your CD has arrived yet, or having the store simply call you when it shows up?

The answer is so obvious that the question seems ridiculous. Yet DNS uses the first model: slave name servers must constantly call up their zone masters and ask them, “Has anything changed yet?” This both generates unnecessary traffic and results in the slave name server being out of date from the time the master *does* change until the next poll is performed. Tweaking the Refresh time for the zone allows only the choice between more polls or more staleness when changes happen; neither is really good.

To improve this situation, a new technique was developed and formalized in RFC 1996, published in 1996 (weird coincidence!). This standard, “A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY),” defines a new DNS message type called *Notify* and describes a protocol for its use. The Notify message is a variation on the standard DNS message type, with some of the fields redefined to support this new feature.

If both the master and slave name servers support this feature, when a modification is made to an RR, the master server will automatically send a Notify message to its slave server(s), saying, “Your CD has arrived!” er... “The database has changed.” The slave then acts as if its Refresh timer had just expired. Enabling this feature allows the Refresh interval to be dramatically increased, since slave servers don’t need to constantly poll the master for changes.

**KEY CONCEPT** The optional DNS *Notify* feature allows a master name server to inform slave name servers when changes are made to a zone. This has two advantages: It cuts down on unnecessary polling by the slave servers to find out if changes have occurred to DNS information, and it also reduces the amount of time that slave name servers have out-of-date records.

## **Improving Zone Transfer Efficiency: Incremental Transfers**

The second issue with regular DNS is the need to transfer the entire zone whenever a change to any part of it is made. There are many zones on the Internet that have truly enormous master files that change constantly. Consider the master files for the .COM zone, for example. Having to copy the entire database to slave name servers every time there is a change to even one record is beyond inefficient—it's downright insane!

RFC 1995, “*Incremental Zone Transfer in DNS*,” specifies a new type of zone transfer called an *incremental zone transfer*. When this feature is implemented on master and slave name servers in a zone, the master server keeps track of the most recent changes made to the database. Each time a slave server determines that a change has occurred and the slave’s database needs to be updated, it sends an IXFR (incremental transfer) query to the master, which contains the serial number of the slave’s current copy of the database. The master then looks to see what RRs have changed since that serial number was the current one and sends only the updated RRs to the slave server.

To conserve storage, the master server obviously doesn’t keep all the changes made to its database forever. It will generally track the last few modifications to the database, with the serial number associated with each. If the slave sends an IXFR request that contains a serial number for which recent change information is still on the master server, only the changes are sent in reply. If the request has a serial number so old that the master server no longer has information about some of the changes since that version of the database, a complete zone transfer is performed instead of an incremental one.

**KEY CONCEPT** The DNS *incremental zone transfer* enhancement uses a special message type that allows a slave name server to determine what changes have occurred since it last synchronized with the master server. By transferring only the changes, the amount of time and bandwidth used for zone transfers can be significantly reduced.

## **Dealing with Dynamic IP Addresses: DNS Update/Dynamic DNS**

The third problem with classic DNS is that it assumes changes are made infrequently to zones, so they can be handled by hand-editing master files. Some zones are so large that hand-editing of the master files would be nearly continuous. However, the problem goes beyond just inconvenience. Regular DNS assumes that the IP address for a host is relatively static. Modern networks, however, make use of host technologies such as the Dynamic Host Configuration Protocol (DHCP) (described in Part III-3), to assign IP addresses dynamically to devices. When DHCP is used, the IP address of each host in a zone could change on a weekly, daily, or even hourly basis! Clearly, there would be no hope of keeping up with this rate of change using a human being and a text editor.

In April 1997, RFC 2136, “*Dynamic Updates in the Domain Name System (DNS UPDATE)*,” was published. This standard describes an enhancement to basic DNS operation that allows DNS information to be dynamically updated. When this feature is implemented, the resulting system is sometimes called *Dynamic DNS (DDNS)*.

RFC 2136 defines a new DNS message type: the Update message. Like the Notify message, the Update message is designed around the structure of regular DNS messages, but with changes to the meanings of several of the fields. As the name implies, Update messages allow RRs to be selectively changed within the master name server for a zone. Using a special message syntax, it is possible to add, delete, or modify RRs.

Obviously, care must be taken in how this feature is used, since we don't want just anyone to be making changes willy-nilly to our master records. The standard specifies a detailed process for verifying Update messages, as well as security procedures that must be put into place so the server accepts such messages from only certain individuals or systems.

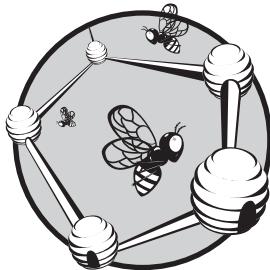
Dynamic DNS allows administrators to make changes much more easily, but its true power becomes evident only when it is used to integrate DNS with other address-related protocols and services. Dynamic DNS solves a major weakness with traditional DNS: the inability to easily associate a host name with an address assigned using a protocol like DHCP.

With DNS servers supporting this feature, DNS and DHCP can be integrated, allowing automatic address and name assignment, and automatic update of DNS records when a host's IP address changes. One common application of Dynamic DNS is to allow the use of DNS names by those who access the Internet using a service provider that dynamically assigns IP addresses. Dynamic DNS is similarly used by certain directory services, notably Microsoft's Active Directory, to associate addresses with device names.

**KEY CONCEPT** An enhancement to DNS, commonly called *Dynamic DNS (DDNS)*, allows DNS information in a server's database to be updated automatically, rather than always requiring hand-editing of master files. This can not only save time and energy on the part of administrators, but it also allows DNS to better handle dynamic address assignment, such as the type performed by host configuration protocols like DHCP.

# 56

## DNS RESOLUTION CONCEPTS AND RESOLVER OPERATIONS



In the preceding three chapters, I have described the Domain Name System (DNS) name space, authorities, registration mechanism, and name servers. These elements can all be considered part of the infrastructure of DNS; they are the parts of the system that must be established first to enable it to be used. Once we have these components in place, we can actually get down to the business at hand: name resolution. This is accomplished using a specific set of procedures carried out by DNS clients called *resolvers*.

In this chapter, I describe DNS name resolvers and the process of name resolution itself. I begin with an overview of the functions performed by DNS resolvers and how they work in general terms. I then describe the two fundamental methods of name resolution used in DNS: iterative and recursive resolution. I discuss the way that resolvers improve efficiency through local resolution and caching. I describe the steps in the actual name resolution algorithm. I then cover two special cases of name resolution: reverse name resolution using the special IN-ADDR.ARPA domain, and the way that DNS provides mail support using Mail Exchange resource records.

**RELATED INFORMATION** The information in this section complements that in the previous chapter on DNS name servers. I assume in the topics here that you have at least basic familiarity with DNS servers.

## DNS Resolver Functions and General Operation

Name servers are arguably the most important part of the DNS system as a whole. After all, they store all the data on the system and actually provide the addresses we need when names are given to them. Without these servers, there would be no DNS at all. Of course, what use is a server if nobody is asking for service? The clients in the system, called *resolvers*, are also important, because they initiate the process of name resolution. Resolvers are where the rubber meets the road, so to speak.

The operation of DNS resolvers is explained in the two main DNS standards. RFC 1034 describes the functions performed by resolvers and how they work in general terms. This includes a discussion of the algorithm used to conduct name resolution. RFC 1035 deals more with the implementation details of resolvers and the fine points of how they do their jobs. Several subsequent standards have modified these base standards, changing some of the ways that resolvers work in different ways.

### **Name Resolution Services**

Just as the main job of a DNS server is to store DNS name data and serve it when it receives requests, the main job of a DNS resolver is to, well, resolve. While most people think of name resolution as only the process of transforming a DNS name into an IP address, this is just one of several types of resolution services performed by DNS. The following are a few of the most typical types of DNS resolution:

**Standard Name Resolution** Taking a DNS name as input and determining its corresponding IP address.

**Reverse Name Resolution** Taking an IP address and determining what name is associated with it.

**Electronic Mail Resolution** Determining where to send electronic mail (email) messages based on the email address used in a message.

### **Functions Performed by Name Resolvers**

There are other types of resolution activities as well, though again, most name resolution requests are of the standard variety, making it the primary focus in our discussion. To accomplish this task, name resolvers perform a number of related functions:

**Providing the User Interface** Normal name resolution usually doesn't involve explicitly running a piece of resolver software. In your web browser, you don't have to say, "Please find the IP address for www.xyzindustries.com," and then say, "Please connect to this IP address for XYZ Industries." You just type www.xyzindustries.com, and the name resolution happens. There is no magic involved. The resolver is just called *implicitly* instead of explicitly. The web browser recognizes that a name has been entered instead of an IP address and feeds it to the resolver, saying, "I need

you to resolve this name, please.” (Hey, it never hurts to be polite.) The resolver takes care of resolution and provides the IP address to the web browser, which connects to the site. Thus, the resolver is the interface between the user (both the human user and the software user, the browser) and the DNS system.

**Forming and Sending Queries** Given a name to resolve, the DNS resolver must create an appropriate query using the DNS messaging system, determine what type of resolution to perform, and send the query to the appropriate name server.

**Processing Responses** The resolver must accept back responses from the DNS server to which it sent its query and decide what to do with the information within the reply. As you’ll see, it may be necessary for more than one server to be contacted for a particular name resolution.

**KEY CONCEPT** The primary clients in DNS are software modules called DNS *name resolvers*. They are responsible for accepting names from client software, generating resolution requests to DNS servers, and processing and returning responses.

These tasks seem fairly simple, and they are in some ways, but implementation can become rather complicated. The resolver may need to juggle several outstanding name resolutions simultaneously. It must keep track of the different requests, queries, and responses and make sure everything is kept straight.

Name resolvers don’t need to perform nearly as many administrative jobs as name servers do; clients are usually simpler than servers in this regard. One important support function that many name resolvers do perform, however, is caching. Like name servers, name resolvers can cache the results of the name resolutions they perform to save time if the same resolution is required again. (Not all resolvers perform caching, however.)

Even though resolvers are the DNS components that are most associated with name resolution, name servers can also act as clients in certain types of name resolution. In fact, it is possible to set up a network so that the resolvers on each of the client machines do nothing more than hand resolution requests to a local DNS server and let the server take care of it. In this case, the client resolver becomes little more than a shell, sometimes called a *stub resolver*. This has the advantage of centralizing name resolution for the network, but a potential disadvantage of performance reduction.

## DNS Name Resolution Techniques: Iterative and Recursive Resolution

Conventional name resolution transforms a DNS name into an IP address. At the highest level, this process can be considered to have two phases. In the first phase, we locate a DNS name server that has the information we need: the address that goes with a particular name. In the second phase, we send that server a request containing the name we want to resolve, and it sends back the address required.

Somewhat ironically, the second phase (the actual mapping of the name into an address) is fairly simple. It is the first phase—finding the right server—that is potentially difficult and represents most of the work in DNS name resolution.

While perhaps surprising, this is a predictable result of how DNS is structured. Name information in DNS is not centralized, but rather distributed throughout a hierarchy of servers, each of which is responsible for one zone in the DNS name space. This means we must follow a special sequence of steps to find the server that has the information we need.

The formal process of name resolution parallels the treelike hierarchy of the DNS name space, authorities, and servers. Resolution of a particular DNS name starts with the most general part of the name and proceeds to the most specific part. Naturally, the most general part of every name is the *root* of the DNS tree, represented in a name as a trailing dot (.), sometimes omitted. The next most specific part is the top-level domain (TLD), then the second-level domain, and so forth. The DNS name servers are linked in that the DNS server at one level knows the name of the servers that are responsible for subdomains in zones below it at the next level.

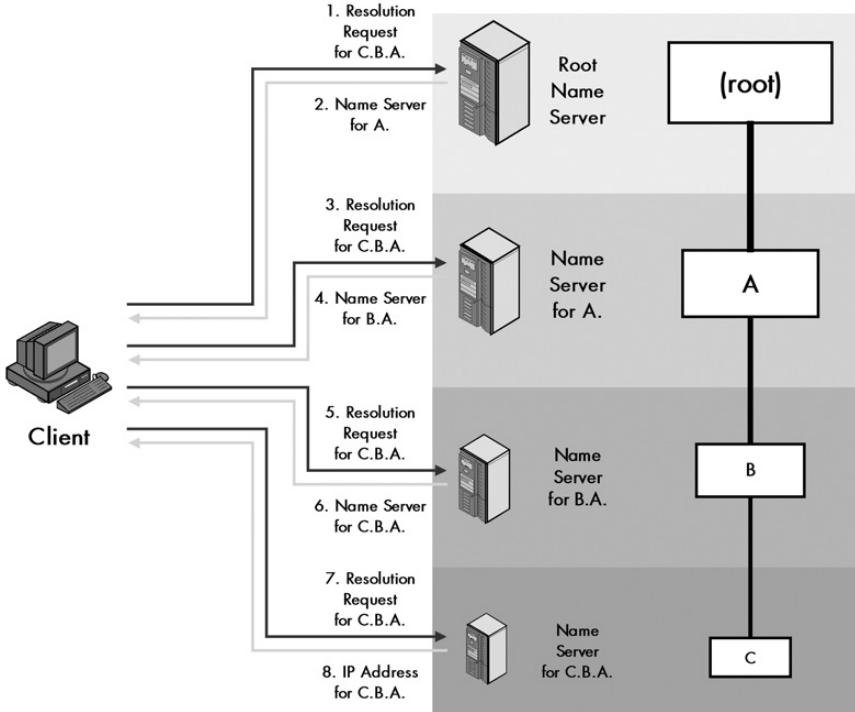
Suppose we start with C.B.A. as the fully qualified domain name (FQDN). Formally, every name resolution begins with the root of the tree—this is why the root name servers are so important. It's possible that the root name servers are authoritative for this name, but this is probably not the case; that's not what the root name servers are usually used for. What the root name server does know is the name of the server responsible for the TLD: A.. The name server for A. may have the information to resolve C.B.A., but it's still fairly high level, so C.B.A. is probably not directly within its zone. In that case, it will not know the address we seek, but it will know the name of the server responsible for B.A.. In turn, that name server may be authoritative for C.B.A., or it may just know the address of the server for C.B.A., which will have the information we need. As you can see, it is very possible that several different servers may be needed in a name resolution.

**KEY CONCEPT** Since DNS name information is stored as a distributed database spread across many servers, name resolution cannot usually be performed using a single request/response communication. It is first necessary to find the server that has the information that the resolver requires. This usually requires a sequence of message exchanges, starting from a root name server and proceeding down to the specific server containing the resource records (RRs) that the client requires.

The DNS standards actually define two distinct ways of following this hierarchy of servers to discover the correct one. They both eventually lead to the right device, but they differ in how they assign responsibility for resolution when it requires multiple steps. The two techniques are *iterative resolution* and *recursive resolution*.

### ***Iterative Resolution***

When a client sends an iterative request to a name server, the server responds with either the answer to the request (for a regular resolution, the IP address we want) *or* the name of another server that has the information or is closer to it. The original client must then *iterate* by sending a new request to this referred server, which again may either answer it or provide another server name. The process continues until the correct server is found. The iterative resolution method is illustrated in Figure 56-1.



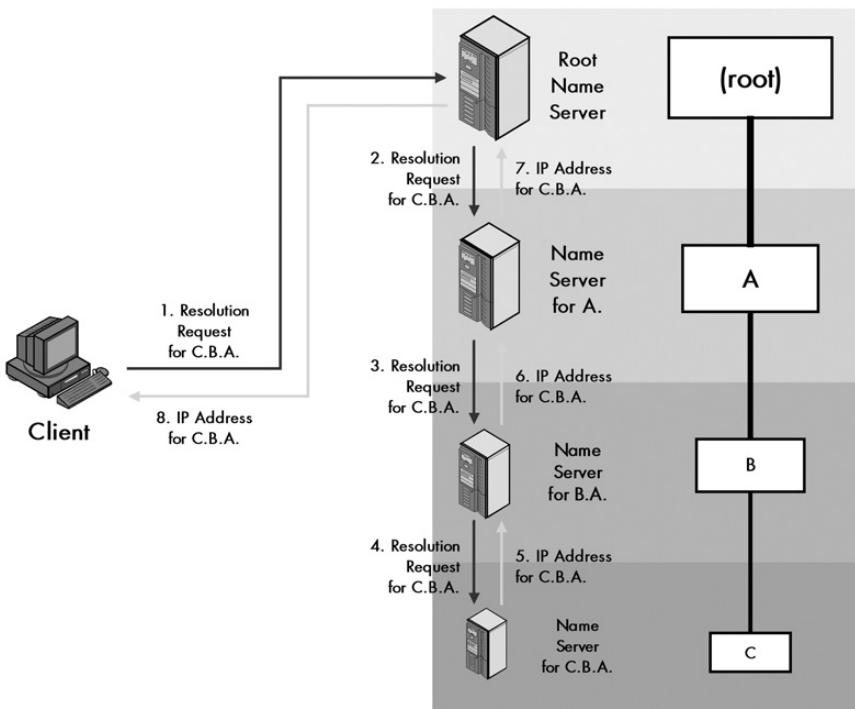
**Figure 56-1: Iterative DNS name resolution** In this example, the client is performing a name resolution for C.B.A. using strictly iterative resolution. It is thus responsible for forming all DNS requests and processing all replies. It starts by sending a request to the root name server for this mythical hierarchy. That server doesn't have the address of C.B.A., so it instead returns the address of the name server for A. The client then sends its query to that name server, which points the client to the server for B.A. That name server refers the client to the name server that actually has the address for C.B.A., which returns it to the client. Contrast this to Figure 56-2.

### Recursive Resolution

When a client sends a recursive request to a name server, the server responds with the answer if it has the information sought. If it doesn't, the server takes responsibility for finding the answer by becoming a client on behalf of the original client and sending new requests to other servers. The original client sends only one request and eventually gets the information it wants (or an error message if it is not available). This technique is shown in Figure 56-2.

### Contrasting Iterative and Recursive Resolution

To help explain the difference between iterative and recursive resolution, let's take a side trip to a real-world case. Suppose you are trying to find the phone number of your old friend Carol, with whom you haven't spoken in years. You call your friend Joe. He doesn't have Carol's number, but he gives you John's number, suggesting you call him. So you call John. He doesn't have the information, but he knows the number of Carol's best friend, Debbie, and gives that to you. You call Debbie, and she gives you Carol's information. This is an example of an iterative process.



**Figure 56-2: Recursive DNS name resolution** This is the same theoretical DNS resolution shown in Figure 56-1, but this time, the client asks for the name servers to perform recursive resolution, and they agree to do so. As in the iterative case, the client sends its initial request to the root name server. That server doesn't have the address of C.B.A., but instead of merely returning to the client the address of the name server for A., it sends a request to that server itself. That name server sends a request to the server for B.A., which sends a request to the server for C.B.A.. The address of C.B.A. is then carried back up the chain of requests, from the server of C.B.A. to that of B.A., then A., then the root, and then finally, back to the client.

In contrast, suppose you call Joe and Joe says, “I don’t know, but I think I know how to find out.” He calls John, and then Debbie, and then calls you back with the phone number. That would be like recursive resolution.

So, in essence, iteration is like doing the job yourself, while recursion is like passing the buck. You might think that everyone would always want to use recursion since it makes the other guy do the work. This is true, but passing the buck is not considered good form if it is not done with permission. Not all name servers support recursion, especially servers near the top of the hierarchy. Obviously, we don’t want to bog down certain name servers—such as the root name servers, the ones that handle .COM, and other critical TLDs—with doing recursion. It is for this reason that clients must request that name servers perform recursion for them. One place where recursion is often used is with the local name server on a network. Rather than making client machine resolvers perform iterative resolution, it is common for the resolver to generate a recursive request to the local DNS server, which then generates iterative requests to other servers as needed. As you can see, recursive and iterative requests can be combined in a single resolution, providing significant flexibility to the process as a whole. This is demonstrated in a more realistic example in the “DNS Name Resolution Process” section later in this chapter.

Again, remember that for the purpose of understanding resolution, a DNS server can act as a client. As soon as a DNS server accepts a recursive request for resolution on a name it cannot resolve itself, it becomes a client in the process. Also, it is common for resolvers to know the names of not one, but two local DNS servers, so if a problem occurs reaching the first, they can try the second.

**KEY CONCEPT** The two methods of name resolution in DNS are *iterative resolution* and *recursive resolution*. In iterative resolution, if a client sends a request to a name server that does not have the information the client needs, the server returns a pointer to a different name server, and the client sends a new request to that server. In recursive resolution, if a client sends a request to a server that doesn't have the requested information, that server takes on the responsibility for sending requests to other servers to find the necessary records, and then returns them to the client. A server doing this takes on the role of client for its requests to other servers.

## DNS Name Resolution Efficiency Improvements: Caching and Local Resolution

The basic resolution techniques—iterative and recursive—can be considered complete from an algorithmic standpoint. By starting at the top (root) and working our way down, we are “guaranteed” to always eventually arrive at the server that has the information we need. I put *guaranteed* in quotation marks because, as always, there are no real guarantees in networking—we might have asked for a nonexistent name, or a server might have bad data, for example. But in the absence of such atypical problems, the process leads to the information eventually.

The problem is that last word: *eventually*. Both iterative and recursive resolution will get us to the right server, but they take a long time to do it, especially if the name we are trying to resolve is in a deep part of the DNS hierarchy (for example, F.E.D.C.B.A.). Since resolution is done so often, it is helpful to define changes to the basic resolution process that improve efficiency as much as possible.

### ***The Motivation for Caching: Locality of Reference***

A computer science principle called *locality of reference* describes two common phenomena related to how computers (and networks) are used. The first, sometimes called *spatial locality of reference*, observes that a resource is more likely to be referenced if it is near another resource that was recently referenced. The second, *temporal locality of reference*, says a resource is more likely to be accessed if it was recently accessed.

We can observe both of these phenomena by using the example of browsing the Web. To observe spatial locality of reference, notice what happens when you visit a site such as <http://www.tcpipguide.com>. The initial request asks the server for the main index document of *The TCP/IP Guide*. However, that document contains links to several images and other items, all of which are also located at the domain [tcpipguide.com](http://www.tcpipguide.com). When your browser asks for the main document, it will shortly thereafter also ask for a number of graphics. As you navigate the site, you will click links to go to other web pages. Again, most of these will be at the same domain, [tcpipguide.com](http://www.tcpipguide.com).

What this means is that if we resolve a particular domain name, it is likely that we will need to resolve it again very soon in the future. It would be silly to need to interrogate the same domain server dozens of times, asking it to resolve the same name each time.

The second phenomenon, *temporal locality of reference*, is one you have probably noticed yourself. You are far more likely to access a resource you have used recently than one you have not looked at in a year. This means that maintaining information about recently used resources can be inherently advantageous.

These two phenomena are the rationale for caching in the computer world in general, and as you have seen in Chapter 55, in DNS servers in particular. The same advantages apply to resolvers, and many of them perform caching also, in a way rather similar to how it is done in servers.

### **Name Resolver Caching**

On a particular client computer, once a particular name is resolved, it is cached and remains ready for the next time it is needed. Again, this eliminates traffic and load on DNS servers. (Note, however, that not all resolvers perform caching.)

You might be wondering why we bother having caching on both resolvers and servers. This is not redundant, as it may appear. Or rather, it's redundant, but in a good way. To understand why, we must recognize that a fundamental trade-off in caching is that a cache provides better performance the closer it is the requester of the data, but better coverage the farther it is from the user.

If resolvers didn't cache results but our local server did, we could get the information from the server's cache, but it would require waiting for the exchange of a query and response. The resolver's cache is closer to the user and so more efficient. At the same time, this doesn't obviate the need for caching at our network's local DNS server. The server is farther away from the user than the resolver, but its cache is shared by many machines. They can all benefit from its cache. For example, if you look up a particular name, and then someone else does a few minutes later, she can use your cached resolution, even though she is typing it for the first time.

**KEY CONCEPT** In addition to the caching performed by DNS name servers, many (but not all) DNS resolvers also cache the results of recent resolution requests. This cache is checked prior to beginning a name resolution, to save time when multiple requests are made for the same name.

Caching by name resolvers follows the same general principles and rules as caching by name servers, outlined in Chapter 55. The amount of time a resource record (RR) is held in the cache is specified by its Time to Live (TTL) value. Also, resolvers will not cache the results of certain queries, such as reverse lookups, and may also not cache a resolution if they suspect (for whatever reason) that the data returned is unreliable or corrupted.

### **Local Resolution**

One other area where resolution efficiency can be improved is the special case where we are trying to resolve the names of computers in our own organizations.

Suppose that you, an employee at XYZ Industries, want to get some sales information using the File Transfer Protocol (FTP) from sales.xyzindustries.com. Your FTP client will invoke your local resolver to resolve that name, by sending it to your local DNS server. Now, would it be smart for that server, which is here inside the company, to start the resolution process up at the root name server? Not really.

The local DNS server that accepts local resolution requests from resolvers on the network may be the authoritative name server for sales.xyzindustries.com. In other cases, it may know how to answer certain resolution requests directly. Obviously, it makes sense for the server to check to see if it can answer a resolver's query before heading up to the root server, since this provides a faster answer to the client and saves internetwork traffic. This is called *local resolution*.

Most DNS servers will perform this check to see if they have the information needed for a request before commencing the formal top-down resolution process. The exception is DNS servers that do not maintain information about any zones: *caching-only servers*. In some cases, DNS resolvers on client machines may also have access to certain local zone information, in which case, they can use it instead of sending a resolution query at all.

**NOTE** *Most operating systems support the use of the old host table mechanism (described in Chapter 51), which can be useful for local machines on a network. If a host has a host table, the resolver will check the host table to see if it can find a mapping for a name before it will bother with the more time-consuming DNS resolution process. This is not technically part of DNS, but is often used in conjunction with it.*

## DNS Name Resolution Process

In the first half of this chapter, I have described what name resolvers do, explained the basic top-down resolution process using iterative and recursive resolution, and discussed how local resolution and caching are used to improve resolution performance. Now it's time to tie all this background material together and see how the name resolution process works as a whole.

As usual, the best way to do this is by example. Here, I will actually combine two examples I have used earlier: the fictitious company XYZ Industries and the nonexistent college, Googleplex University.

### A Simple Example of DNS Name Resolution

Let's say that XYZ Industries runs its own DNS servers for the xyzindustries.com zone. The master name server is called ns1.xyzindustries.com, and the slave is ns2.xyzindustries.com. These are also used as local DNS servers for resolvers on client machines. We'll assume for this example that, as is often the case, our DNS servers will accept recursive requests from machines within our company, but we will not assume that other machines will accept such requests. Let's also assume that both the server and resolver perform caching, and that the caches are empty.

Let's say that Googleplex University runs its own DNS servers for the googleplex.edu domain, as in the example in Chapter 54. There are three subdomains: finearts.googleplex.edu, compsci.googleplex.edu, and admin.googleplex.edu. Of these, compsci.googleplex.edu is in a separate zone with dedicated servers, while the other subdomains are in the googleplex.edu zone (see Figure 54-2).

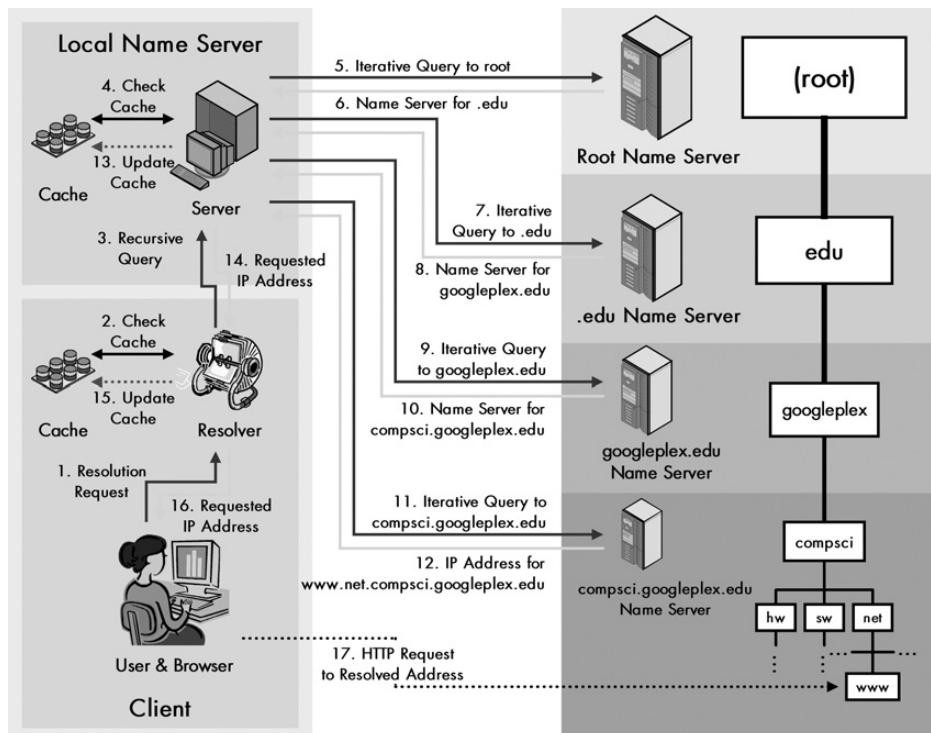
Now, suppose you are an employee within XYZ Industries and one of your clients is in charge of the networking department at Googleplex U. You type into your web browser the address of that department's web server, www.net.compsci.googleplex.edu. In simplified terms, the procedure would involve the following steps (Figure 56-3 shows the process graphically):

1. Your web browser recognizes the request for a name and invokes your local resolver, passing to it the name www.net.compsci.googleplex.edu.
2. The resolver checks its cache to see if it already has the address for this name. If it does, it returns it immediately to the web browser, but in this case, we are assuming that it does not. The resolver also checks to see if it has a local host table file. If so, it scans the file to see if this name has a static mapping. If so, it resolves the name using this information immediately. Again, let's assume it does not, since that would be boring.
3. The resolver generates a recursive query and sends it to ns1.xyzindustries.com (using that server's IP address, of course, which the resolver knows).
4. The local DNS server receives the request and checks its cache. Again, let's assume it doesn't have the information needed. If it did, it would return the information, marked non-authoritative, to the resolver. The server also checks to see if it has in its zone resource records that can resolve www.net.compsci.googleplex.edu. Of course, it does not in this case, since they are in totally different domains.
5. ns1.xyzindustries.com generates an iterative request for the name and sends it to a root name server.
6. The root name server does not resolve the name. It returns the name and address of the name server for the .edu domain.
7. ns1.xyzindustries.com generates an iterative request and sends it to the name server for .edu.
8. The name server for .edu returns the name and address of the name server for the googleplex.edu domain.
9. ns1.xyzindustries.com generates an iterative request and sends it to the name server for googleplex.edu.
10. The name server for googleplex.edu consults its records. It sees, however, that this name is in the compsci.googleplex.edu subdomain, which is in a separate zone. It returns the name server for that zone.
11. ns1.xyzindustries.com generates an iterative request and sends it to the name server for compsci.googleplex.edu.
12. The name server for compsci.googleplex.edu is authoritative for www.net.compsci.googleplex.edu. It returns the IP address for that host to ns1.xyzindustries.com.
13. ns1.xyzindustries.com caches this resolution.
14. The local name server returns the resolution to the resolver on your local machine.
15. Your local resolver also caches the information.

16. The local resolver gives the address to your browser.
17. Your browser commences an HTTP request to the Googleplex machine's IP address.

This seems rather complicated and slow. Of course, computers work faster than you can read (or I can type, for that matter). Even given that, the benefits of caching are obvious—if the name were in the cache of the resolver or the local DNS server, most of these steps would be avoided.

Note that this example is highly simplified and also shows only one possible way that servers might be set up. For one thing, it is possible that even though compsci.googleplex.edu is in a separate zone from googleplex.edu, they might use the same server. In that case, one iteration in the process would be skipped. The example also doesn't show what happens if an error occurs in the process. Also, if the name entered were an alias, indicated by a CNAME record, this would change the processing as well.



**Figure 56-3: Example of the DNS name resolution process** This fairly complex example illustrates a typical DNS name resolution using both iterative and recursive resolution. The user types a DNS name (`www.net.compsci.googleplex.edu`) into a web browser, which causes a DNS resolution request to be made from her client machine's resolver to a local DNS name server. That name server agrees to resolve the name recursively on behalf of the resolver, but uses iterative requests to accomplish it. These requests are sent to a DNS root name server, followed in turn by the name servers for `.edu`, `googleplex.edu`, and `compsci.googleplex.edu`. The IP address is then passed to the local name server and then back to the user's resolver, and finally, to her web browser software.

### **Changes to Resolution to Handle Aliases (CNAME Records)**

CNAME records are used to allow a constant name for a device to be presented to the outside world, while allowing the actual device that corresponds to the name to vary inside the organization. When a CNAME is used, it changes the name resolution process by adding an extra step: First we resolve the alias to the canonical name, and then we resolve the canonical name.

For example, web servers are almost always named starting with www., so at XYZ Industries, we want people to be able to find our website at www.xyzindustries.com. However, the web server may be shared with other services on bigserver.xyzindustries.com. We can set up a CNAME record to point www.xyzindustries.com to bigserver.xyzindustries.com. Resolution of www will result in a CNAME pointing to bigserver, which is then itself resolved. If in the future, our business grows and we decide to upgrade our web service to run on biggerserver.xyzindustries.com, we just change the CNAME record, and users are unaffected.

## **DNS Reverse Name Resolution Using the IN-ADDR.ARPA Domain**

If most people were asked to identify the core job of DNS to one function, they would probably say it was converting the names of objects into the numeric IP addresses associated with them. (Well, they would if they knew much about DNS.) For this reason, DNS is sometimes compared to a telephone book, or to telephone 411 (information) service. There are certain problems with this analogy, but at the highest level, it is valid. In both cases, we take a name, consult a database (of one type or another), and produce from it a number that matches that name.

In the real world, there are sometimes situations where you don't want to find the phone number that goes with a name, but rather, you have a phone number and want to know what person it belongs to. For example, this might happen if your telephone records the number of incoming calls but you don't have caller ID to display the name associated with a number. You might also find a phone number on a piece of paper and not remember whose number it is.

Similarly, in the networking world, there are many situations where we have an IP address and want to know what name goes with it. For example, a World Wide Web server records the IP address of each device that connects to it in its server logs, but these numbers are generally meaningless to humans, who prefer to see the names that go with them. A more serious example might be a hacker trying to break into your computer; by converting the IP address into a name, you might be able to find out what part of the world he is from, what Internet service provider (ISP) he is using, and so forth. There are also many reasons why a network administrator might want to find out the name that goes with an address, for setup or troubleshooting purposes.

DNS originally included a feature called *inverse querying* that would allow this type of “opposite” resolution.

## ***The Original Method: Inverse Querying***

For inverse querying, a resolver could send a query which, instead of having a name filled in and a space for the server to fill in the IP address, had the IP address and a space for the name. The server would check its RRs and return the name to the resolver.

This works fine in theory, and even in practice, if the internetwork is very small. However, remember that due to the distributed nature of DNS information, the biggest part of the job of resolution is finding the right server. Now, in the case of regular resolution, we can easily find the right server by traversing the hierarchy of servers. This is possible because the servers are connected together following a hierarchy of names.

DNS servers are not, however, arranged based on IP address. This means that to use inverse queries, we need to use the right name server for the IP address we want to resolve into a name, with no easy way to find out what it is. Sure, we could try sending the inverse query to the authoritative DNS server for every zone in the hierarchy. If you tried, it would probably take you longer than it took to write this book, so let's not go there. The end result of all of this is that inverse queries were never popular, except for local server troubleshooting. They were formally removed from DNS in November 2002 through the publishing of RFC 3425.

So, what to do? Well, the problem is that the servers are arranged by name and not by IP address. The solution, therefore, is as simple as it sounds: Arrange the servers by IP address. This doesn't mean we remove the name hierarchy, or duplicate all the servers, or anything silly like that. Instead, we create an additional, numerical hierarchy that coexists with the name hierarchy. We then use this to find names from numbers, using a process commonly called *reverse name resolution*.

## ***The IN-ADDR.ARPA Name Structure for Reverse Resolution***

The name hierarchy for the Internet is implemented using a special domain called *IN-ADDR.ARPA*, located within the reserved .ARPA TLD (*IN-ADDR* stands for *INternet ADDRess*). Recall from the discussion in Chapter 54 that .ARPA was originally used to transition old Internet hosts to DNS and is now used by the folks that run the Internet for various purposes.

A special numerical hierarchy is created within IN-ADDR.ARPA that covers the entire IP address space (see Figure 56-4):

- At the first level within IN-ADDR.ARPA there are 256 subdomains called 0, 1, 2, and so on, up to 255; for example, 191.IN-ADDR.ARPA. (Actually, there may not be all 256 of these, since some IP addresses are reserved, but let's ignore that for now.)
- Within each of the first-level subdomains, there are 256 further subdomains at the second level, numbered the same way. So, for example, one of these would be 27.191.IN-ADDR.ARPA.
- Again, there are 256 subdomains at the third level within each of the second-level subdomains, such as 203.27.191.IN-ADDR.ARPA.
- Finally, there are 256 subdomains at the fourth level within each of the third-level subdomains, such as 8.203.27.191.IN-ADDR.ARPA.

As you can see, within IN-ADDR.ARPA, we have created a name space that parallels the address space of the Internet Protocol (IP). Yes, this means there are several billion nodes and branches in this part of the Internet DNS name space!

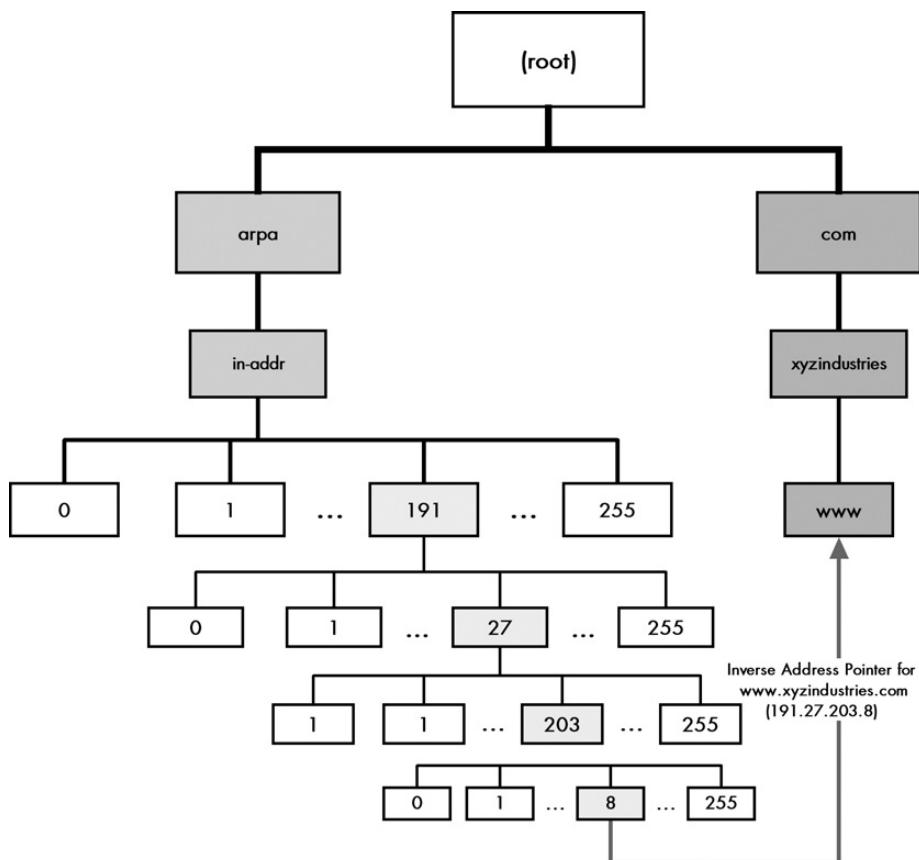
### ***RR Setup for Reverse Resolution***

With this structure in place, we can now associate one entry in this name space with each entry in the real DNS name space. We do this using the Pointer (PTR) RR type. For example, if www.xyzindustries.com has the IP address 191.27.203.8, then the DNS server for its zone will have an Address (A) RR indicating this. In master file text format, it will say something like this:

---

```
www.xyzindustries.com. A 191.27.203.8
```

---



**Figure 56-4: The DNS IN-ADDR.ARPA reverse name resolution hierarchy** The special IN-ADDR.ARPA hierarchy was created to allow easy reverse lookups of DNS names. IN-ADDR.ARPA contains 256 sub-domains numbered 0 to 255, each of which has 256 sub-domains numbered 0 to 255, and so forth, down to four levels. Thus, each IP address is represented in the hierarchy. This example shows the DNS domain name www.xyzindustries.com. It would have a conventional RR pointing to its IP address, 191.27.203.8, as well as a reverse resolution record at 8.203.27.191.IN-ADDR.ARPA, pointing to the domain name www.xyzindustries.com.

However, there will also be the following entry for it within the IN-ADDR.ARPA domain:

---

8.203.27.191.IN-ADDR.ARPA PTR www.xyzindustries.com

---

**NOTE** Remember that DNS names are case-insensitive, so IN-ADDR.ARPA could also be given as in-addr.arpa.

Once this is done, reverse name resolution can be easily performed by doing a name resolution on 8.203.27.191.in-addr.arpa. If we do this, a server for the IN-ADDR.ARPA domain will return to us the name www.xyzindustries.com. This is shown in Figure 56-4.

**KEY CONCEPT** Most name resolutions require that we transform a DNS domain name into an IP address. However, there are cases where we want to perform a *reverse name resolution*, by starting with an IP address and finding out what domain name matches it. This is difficult to do using the conventional DNS distributed name hierarchy, because there is no easy way to find the DNS server containing the entries for a particular IP address. To this end, a special hierarchy called IN-ADDR.ARPA was set up for reverse name lookups. This hierarchy contains four levels of numerical subdomains structured so that each IP address has its own node. The node for an IP address contains an entry that points to the DNS domain name associated with that address.

I'm sure you've noticed that the numbers are backward in the IN-ADDR.ARPA domain. We've already seen the reason for this: Name resolution proceeds from the least specific to the most specific element, going from right to left. In contrast, IP addresses have the least specific octet on the left and the most specific on the right. Thus, we reverse them to maintain consistency with the DNS name space.

This immediately yields one extra benefit. Just as we can delegate authority for portions of the regular name space, for example, letting XYZ Industries be in charge of everything in xyzindustries.com, we can also delegate authority for parts of the IN-ADDR.ARPA name space. For example, since the Massachusetts Institute of Technology (MIT) owns all IP addresses with a first octet of 18 (at least, I think it still does), it is possible that if MIT wanted to, it could control the 18.IN-ADDR.ARPA domain as well for reverse queries. This would not be possible without reversing the octets.

Note that for this system to work reliably, it is essential that the data in the regular name space and the reverse name space remain consistent. Whenever a new DNS name is registered, an appropriate entry must be made within IN-ADDR.ARPA as well. Special procedures have been put into place to allow these pointer entries to be created automatically.

**RELATED INFORMATION** A similar scheme using a different reverse domain is used for DNS under version 6 of the Internet Protocol (IPv6). See the end of Chapter 57 for more information.

## DNS Electronic Mail Support and Mail Exchange (MX) Resource Records

Most savvy users of the Internet know that DNS exists, and they usually associate it with the most common Internet applications. Of these applications, the “Big Kahuna” is the World Wide Web. It’s probably the case that the majority of DNS name resolution requests are spawned as a result of web server domain names being typed into browsers billions of times a day, as well as requests for named pages generated by both user mouse clicks and web-based applications.

Of course, DNS is not tied specifically to any one application. We can specify names in any place where an IP address would go. For example, you can use a DNS name instead of an address for an FTP client, or even for a troubleshooting utility like traceroute or ping (see Chapter 88). The resolver will, in each case, take care of translating the name for you.

There’s one application that has always used DNS, but it’s one that doesn’t usually spring to mind when you think about DNS: electronic mail (discussed in Part III-7). Electronic mail (email) is, in fact, more reliant on DNS than just about any other TCP/IP application. Consider that while you may sometimes type an IP address for a command like traceroute, or even type it into a browser, you probably have never sent anyone mail by entering joe@14.194.29.60 into your email client. You type something like joe@xyzindustries.com, and DNS takes care of figuring out where email for XYZ Industries is to go.

### ***Special Requirements for Email Name Resolution***

Name resolution for email addresses is different from other applications in DNS, for three reasons (which I describe in more detail in the discussion of TCP/IP email addressing and address resolution in Chapter 75):

- We may not want email to go to the exact machine specified by the address.
- We need to be able to change server names without changing everyone’s email address.
- We need to be able to support multiple servers for handling mail.

For example, XYZ Industries might want to use a dedicated mail server called mail.xyzindustries.com to handle incoming mail, but actually construct all of its email addresses to use @xyzindustries.com. This makes addresses shorter and allows the server’s name to be changed without affecting user addresses. If the company wishes, it might decide to use two servers, mail1.xyzindustries.com and mail2.xyzindustries.com, for redundancy, and again have just @xyzindustries.com for addresses.

To allow the flexibility needed for these situations, a special DNS RR type, called a *Mail Exchange (MX)* record, is defined.

## **The Mail Exchange (MX) Record and Its Use**

Each MX record specifies a particular mail server that is to be used to handle incoming email for a particular domain. Once this record is established, resolution of email messages is pretty much similar to regular resolution. Suppose you want to send a message to joe@xyzindustries.com. The basic process is as follows:

1. Your email client invokes the resolver on your local machine to perform an email resolution on xyzindustries.com.
2. Your local resolver and local DNS server follow the process described earlier in this chapter to find the authoritative server for xyzindustries.com, which is ns1.xyzindustries.com.
3. ns1.xyzindustries.com finds the MX record for xyzindustries.com and replies back indicating that mail.xyzindustries.com should be used for email.

The email client can't actually send anything to mail.xyzindustries.com; it needs its IP address. So, it would then need to resolve that name. This resolution request will likely end up right back at the same DNS name server that just handled the MX request. To eliminate the inefficiency of two separate resolutions, the DNS name server can combine the information. In our example, ns1.xyzindustries.com will include the A (Address) RR for mail.xyzindustries.com in the Additional section of the DNS message that it sends in step 3.

**NOTE** *RFC 1035 originally defined several other RR types related to email as well: Mailbox (MB), Mail Group (MG), and Mail Rename (MR). These are called "experimental" in the standard. I think the experiment failed, whatever it was, because I don't believe these are used today. There are also two even older mail-related RRs, Mail Destination (MD) and Mail Forwarder (MF), which must have been used at one time but were already obsolete at the time RFC 1035 itself was written.*

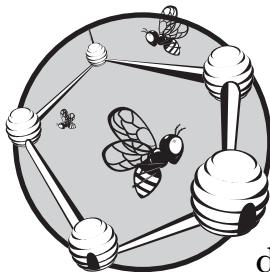
It is also possible to specify multiple MX records for a particular domain, each pointing to a different mail server's name. This provides redundancy, so if there is a problem with one mail server, another can pick up the slack. DNS allows each mail server to be specified with a *preference* value, so you can clearly indicate which is the main mail server, which is the first backup, the second backup, and so on. The DNS server will choose the mail server with the lowest preference value first, then the next highest one, and so on.

**KEY CONCEPT** Since email is sent using host names and not IP addresses, DNS contains special provisions to support the transfer of email between sites. Special *Mail Exchange (MX)* DNS RRs are set up that contain the names of mail servers that a domain wants to use for handling incoming email. Before sending email to a site, a device performs a name resolution to get that site's MX record, so it knows where to send the message.



# 57

## DNS MESSAGING AND MESSAGE, RESOURCE RECORD, AND MASTER FILE FORMATS



Networking is all about the communication of information between connected devices. In the case of the Domain Name

System (DNS), information about names and objects on the internetwork is exchanged during each of the many types of operations DNS performs. This involves sending *messages* between devices. Like most protocols, DNS uses its own set of messages with distinct field formats, and it follows a particular set of rules for generating them and transporting them over the internetwork.

In this chapter, I explain how messages are generated and sent in DNS, and I describe the formats used for messages and resource records (RRs). I begin with an overview discussion of DNS messages and how they are generated and transported. I provide an overview of the general DNS message format and the five sections it contains. I describe the notation used for names and the special compression method that helps keep DNS messages down in size. I then show the fields in the DNS message Header and Question

section. I illustrate the common field format used for all RRs and the specific fields in the most important record types. I also provide a description of the format used for DNS text master files.

I conclude with a brief discussion of the changes made to DNS to support Internet Protocol version 6 (IPv6). Most of these changes (but not all of them) are associated with message formats and RRs, the subject of this chapter.

**BACKGROUND INFORMATION** *This chapter assumes that you are already familiar with DNS concepts and operation as described in Chapters 52 through 56.*

## DNS Message Generation and Transport

In the preceding chapters in this part of the book, we have explored the many different tasks that servers and resolvers perform: regular name resolution, reverse name resolution, email resolution, zone transfers, and more. Each of these operations requires that information be exchanged between a pair of DNS devices. Like so many other TCP/IP protocols, DNS is designed to accomplish this information transfer using a *client/server* model. All DNS exchanges begin with a client sending a request and a server responding with an answer.

### DNS Client/Server Messaging Overview

In Chapter 8's overview of TCP/IP's client/server nature, I explained a potential source of confusion regarding these terms: the fact that they refer to hardware roles, software roles, and transactional roles. This issue definitely applies when it comes to DNS. You've already seen that DNS implementation consists of two sets of software elements: resolvers that act as clients and name servers that are the servers. Resolver software usually runs on client machines like PCs, while name server software often runs on dedicated server hardware. However, these designations are based on the overall role of the hardware and software.

From a messaging viewpoint, the client is the initiator of the communication, regardless of what type of machine does this initiating, and the server is the device that responds to the client. A resolver usually acts as a client and a name server as a server. However, in a particular exchange, a DNS name server can act as a client, in at least two cases. First, in recursive name resolution, a server generates requests to others servers and therefore acts as a client. Second, in administrative functions like zone transfers, one server acts as a client and sends a request to another server. (There are no cases in DNS that I know of where a resolver acts as a server, incidentally.)

Most transactions in DNS consist of the exchange of a single query message and a single response message. The device acting as client for the transaction creates the query and sends it to the server; the server then sends back a reply. In certain cases where a great deal of data needs to be sent, such as zone transfers, the server may send back multiple messages. Multiple such transactions may be required to perform a complete name resolution, as the example of the DNS resolution process in the previous chapter demonstrated.

## DNS Message Transport Using UDP and TCP

TCP/IP has two different transport layer protocols: the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) (see Part II-8). UDP and TCP share layer 4 in the TCP/IP model, because they are so different in terms of capabilities and operation. Some application layer protocols need the services of TCP and can use it to take advantage of them, while others are better off with the simpler UDP. DNS is itself a perfect example of the valid reasons for having both UDP and TCP in the protocol suite (see Chapter 42), because it uses both.

UDP is a simple connectionless protocol that provides no real features but is very fast. It is ideally suited for small, quick exchanges of information and can be faster than TCP because there is no need to establish a connection. This makes it a good choice for most of the conventional queries used in DNS, because they are normally very short, and fast data exchange is important. For this reason, the DNS standards recommend use of UDP for queries and replies as part of regular and reverse name resolution. UDP DNS messages are limited to 512 bytes; longer messages are truncated, and a special bit in the header is set to indicate that this has occurred. If a message being truncated causes a problem for its recipient, the query must be repeated using TCP.

**NOTE** *The 512-byte limit on DNS UDP messages can be surpassed if the optional Extension Mechanisms for DNS (EDNS0) are implemented. These are described in RFC 2671.*

Since UDP does not provide reliable delivery of messages, DNS clients must keep track of requests they have sent. If no response is received after a particular amount of time, the request must be retransmitted. The need to take care of these details is considered an acceptable trade-off for the lower setup costs involved with UDP, such as not requiring a connection. The rate at which retransmissions are sent is usually set at a minimum of two to five seconds to prevent excessive DNS traffic on the internetwork.

For certain special DNS transactions, UDP is simply inappropriate. The most common example of such a transaction is a zone transfer. While the query for a zone transfer is small in size, the amount of data sent in response can be quite large. The limit of 512 bytes for UDP is not even close to enough. Furthermore, we really do need to make sure that a zone transfer is accomplished reliably and with flow control and other data transfer management features, or we risk having corrupted zone information in our secondary DNS server databases.

The solution is to use TCP for these types of exchanges. TCP allows messages to be of arbitrary length, and as a connection-oriented, acknowledged, reliable protocol, automatically provides the mechanisms we need to ensure that zone transfers and other lengthy operations complete successfully. The cost is the small amount of overhead needed to establish the connection, but since zone transfers are infrequent (compared to the sheer volume of regular name resolutions), this is not a problem.

You can see how DNS nicely illustrates the roles of both TCP and UDP in TCP/IP. Since both transport protocols can be used, name servers listen for UDP and TCP requests on the same well-known port number, 53. The device acting as the client uses an ephemeral port number for the transaction. All DNS messages are sent unicast from one device directly to another.

**KEY CONCEPT** DNS uses both UDP and TCP to send messages. Conventional message exchanges are short, and thus well suited to the use of the very fast UDP; DNS itself handles the detection and retransmission of lost requests. For larger or more important exchanges of information, especially zone transfers, TCP is used—both for its reliability and its ability to handle messages of any size.

## DNS Message Processing and General Message Format

As we've just discussed, DNS message exchanges are all based on the principle of client/server computing. In a particular exchange, one device acts as a client, initiating the communication by sending a query; the other acts as the server by responding to the query with an answer. This query/response behavior is an integral part of DNS, and it is reflected in the format used for DNS messages.

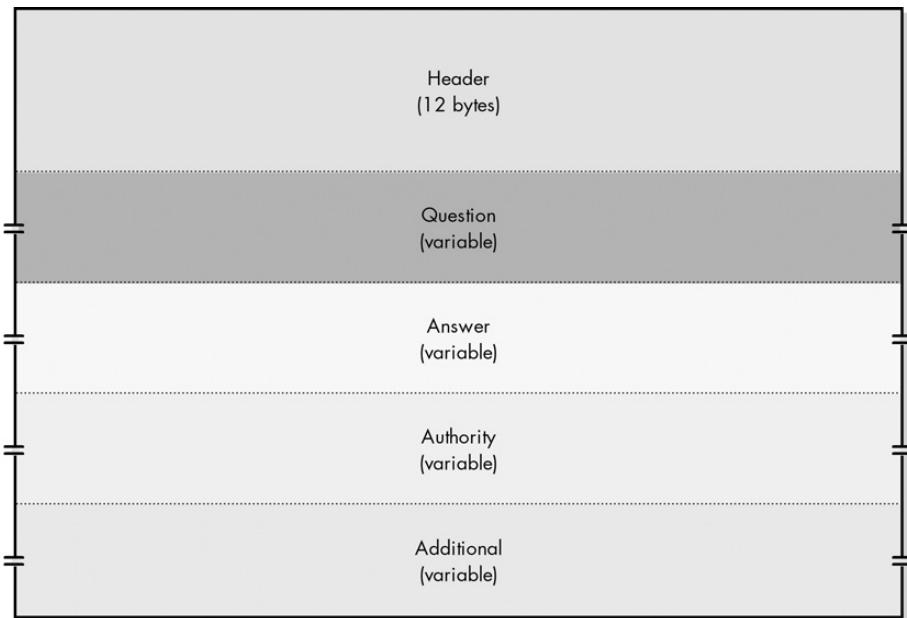
A common message format is used for DNS queries and responses. This message format contains five sections that provide a place for the query asked by the client, the answer(s) provided by the server, and header information that controls the entire process. Table 57-1 describes the DNS general message format, providing a brief summary of each of its sections and how they are used. You can also see a simplified illustration of the message format in Figure 57-1.

**Table 57-1:** DNS General Message Format

Section Name	Description
Header	Contains fields that describe the type of message and provide important information about it. Also contains fields that indicate the number of entries in the other sections of the message.
Question	Carries one or more questions—that is, queries for information being sent to a DNS name server.
Answer	Carries one or more RRs that answer the question(s) indicated in the Question section.
Authority	Contains one or more RRs that point to authoritative name servers that can be used to continue the resolution process.
Additional	Conveys one or more RRs that contain additional information related to the query that is not strictly necessary to answer the queries (questions) in the message.

The Header section is always present in all messages and is fixed in length. In addition to containing important DNS control information, it has a flag (QR) that indicates whether a message is a query or a response. It also has four “count” fields that tell the recipient the number of entries in the other four sections.

When a client initiates a query, it creates a message with the fields in the Header section filled in, and one or more queries (requests for information) in the Question section. It sets the QR flag to 0 to indicate that this is a query, and it places a number in the QDCount field of the header that indicates the number of questions in the Question section. The number of entries in the other sections are usually 0, so their count fields (ANCount, NSCount, and ARCount) are set to 0 in the header. (Although more than one question can be put into a query, usually only one is included.)



**Figure 57-1: DNS general message format**

When the server receives the query, it processes it and performs the information retrieval operation requested (if it can). It then uses the query as the basis for its response message. The Header and Question sections are copied to the response message, and the QR flag is set to 1 to indicate that the message is a reply. Certain fields are also changed in the Header section to provide information back to the client. For example, the server sets the RCode (Response Code) field to indicate whether the query was successful or if an error occurred, and if one did occur, to indicate what the problem was. The next section of this chapter illustrates all the Header fields and indicates how each is used by both client and server.

The server is also responsible for filling in the other three sections of the message: Answer, Authority, and Additional. These sections share the same basic format, each carrying one or more RRs that use a common record format. The number of records in each section is indicated using the count fields in the message header. The sections differ only in terms of the types of records they carry. Answer records are directly related to the question asked, while Authority records carry RRs that identify other name servers. Authority records are thus the means by which name servers are hierarchically linked when the server doesn't have the information the client requested.

The Additional section exists for the specific purpose of improving DNS efficiency. There are cases where a server supplies an answer to a query that it has reason to believe will lead to a subsequent question that the server can also answer. For example, suppose a server provides the name of another name server in the Authority section (an NS RR). The client may not have the address for that server, which would mean it must perform an extra name resolution to contact the referenced server. If the server providing the NS record already knows the IP address

for this name server, it can include it in the Additional section. The same goes for a server providing an MX record as I explained in the discussion of DNS mail support in the previous chapter.

**KEY CONCEPT** DNS uses a general message format for all messages. It consists of a fixed 12-byte header, a Question section that contains a query, and then three additional sections that can carry RRs of different types. The Answer section usually contains records that directly answer the question of the message; the Authority section holds the names of name servers being sent back to the client; and the Additional section holds extra information that may be of value to the client, such as the IP address of a name server mentioned in the Authority section.

Another optimization by DNS is a special compression technique used to reduce the size of DNS messages. This is explained in the “DNS Name Notation and Message Compression” section later in this chapter.

Note that the special Notify and Update messages use a different format than the regular DNS query/response messages. These special messages (whose use is described in the section about DNS server enhancements in Chapter 55) are based on the regular format but with the meanings of certain fields changed. You can find these field formats in RFC 1996 and RFC 2136, respectively.

The client/server information exchange in DNS is facilitated using query/response messaging. Both queries and responses have the same general format, containing up to five individual sections carrying information. Of these, two are usually found in both queries and responses: the Header section and the Question section. We will look at these two sections first, and then examine the RR formats used by servers for the other three message sections.

## DNS Message Header Format

The header is the most important part of any message, since it is where critical control fields are carried. In DNS messages, the Header section carries several key control flags, and it also indicates which of the other sections are used in the message. Examining the Header section can help you understand several of the nuances of how messaging works in DNS.

The format of the Header section used in all DNS messages is illustrated in Figure 57-2 and described in detail in Tables 57-2, 57-3, and 57-4. Where fields are used differently by the client and server in an exchange, I have mentioned in Table 57-2 how the use is differentiated between the two.

Note that the current lists of valid question types, query operation codes, and response codes are maintained by the Internet Assigned Numbers Authority (IANA) as one of its many lists of Internet parameters. Response codes 0 to 5 are part of regular DNS and are defined in RFC 1035; codes 6 to 10 implement Dynamic DNS and are defined in RFC 2136.

**Table 57-2:** DNS Message Header Format

<b>Field Name</b>	<b>Size (Bytes)</b>	<b>Description</b>
ID	2	Identifier: A 16-bit identification field generated by the device that creates the DNS query. It is copied by the server into the response, so it can be used by that device to match that query to the corresponding reply received from a DNS server. This is used in a manner similar to how the Identifier field is used in many of the Internet Control Message Protocol (ICMP) message types.
QR	1/8 (1 bit)	Query/Response Flag: Differentiates between queries and responses. Set to 0 when the query is generated; changed to 1 when that query is changed to a response by a replying server.
OpCode	1/2 (4 bits)	Operation Code: Specifies the type of query the message is carrying. This field is set by the creator of the query and copied unchanged into the response. See Table 57-3 for the OpCode values.
AA	1/8 (1 bit)	Authoritative Answer Flag: This bit is set to 1 in a response to indicate that the server that created the response is authoritative for the zone in which the domain name specified in the Question section is located. If it is 0, the response is non-authoritative.
TC	1/8 (1 bit)	Truncation Flag: When set to 1, indicates that the message was truncated due to its length being longer than the maximum permitted for the type of transport mechanism used. TCP doesn't have a length limit for messages; UDP messages are limited to 512 bytes, so this bit being sent usually is an indication that the message was sent using UDP and was too long to fit. The client may need to establish a TCP session to get the full message. On the other hand, if the portion truncated was part of the Additional section, it may choose not to bother.
RD	1/8 (1 bit)	Recursion Desired: When set in a query, requests that the server receiving the query attempt to answer the query recursively, if the server supports recursive resolution. The value of this bit is not changed in the response.
RA	1/8 (1 bit)	Recursion Available: Set to 1 or cleared to 0 in a response to indicate whether the server creating the response supports recursive queries. This can then be noted by the device that sent the query for future use.
Z	3/8 (3 bits)	Zero: Three reserved bits set to 0.
RCode	1/2 (4 bits)	Response Code: Set to 0 in queries, then changed by the replying server in a response to convey the results of processing the query. This field is used to indicate if the query was answered successfully or if some sort of error occurred. See Table 57-4 for the RCode values.
QDCount	2	Question Count: Specifies the number of questions in the Question section of the message.
ANCount	2	Answer Record Count: Specifies the number of RRs in the Answer section of the message.
ARCount	2	Additional Record Count: Specifies the number of RRs in the Additional section of the message.

**Table 57-3:** Header OpCode Values

<b>OpCode Value</b>	<b>Query Name</b>	<b>Description</b>
0	Query	A standard query.
1	IQuery	An inverse query; now obsolete. RFC 1035 defines the inverse query as an optional method for performing inverse DNS lookups; that is, finding a name from an IP address. Due to implementation difficulties, the method was never widely deployed, however, in favor of reverse mapping using the IN-ADDR.ARPA domain. Use of this OpCode value was formally obsoleted in RFC 3425, November 2002.
2	Status	A server status request.
3	Reserved	Reserved, not used.

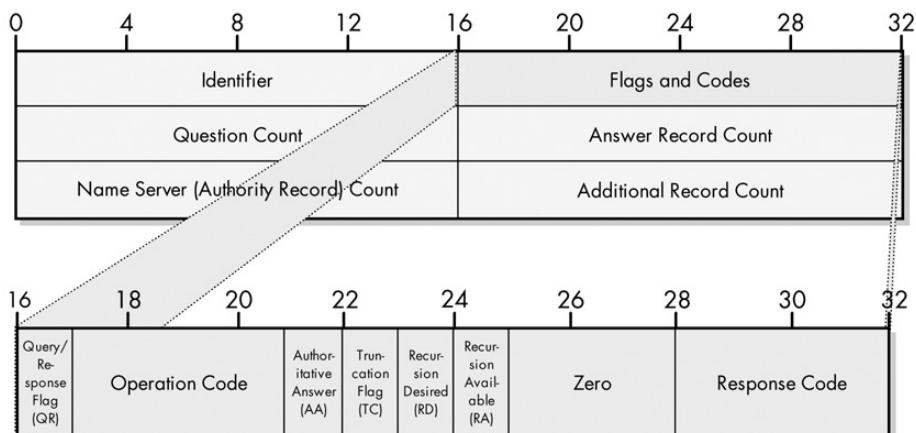
(continued)

**Table 57-3:** Header OpCode Values (continued)

OpCode Value	Query Name	Description
4	Notify	A special message type added by RFC 1996. It is used by a primary (master, authoritative) server to tell secondary servers that data for a zone has changed and prompt them to request a zone transfer. See the discussion of DNS server enhancements in Chapter 55 for more details.
5	Update	A special message type added by RFC 2136 to implement Dynamic DNS. It allows RRs to be added, deleted, or updated selectively. See the discussion of DNS server enhancements in Chapter 55 for more details.

**Table 57-4:** Header RCode Values

RCode Value	Response Code	Description
0	No Error	No error occurred.
1	Format Error	The server was unable to respond to the query due to a problem with how it was constructed.
2	Server Failure	The server was unable to respond to the query due to a problem with the server itself.
3	Name Error	The name specified in the query does not exist in the domain. This code can be used by an authoritative server for a zone (since it knows all the objects and subdomains in a domain) or by a caching server that implements negative caching.
4	Not Implemented	The type of query received is not supported by the server.
5	Refused	The server refused to process the query, generally for policy reasons and not technical ones. For example, certain types of operations, such as zone transfers, are restricted. The server will honor a zone transfer request only from certain devices.
6	YX Domain	A name exists when it should not.
7	YX RR Set	An RR set exists that should not.
8	NX RR Set	An RR set that should exist does not.
9	Not Auth	The server receiving the query is not authoritative for the zone specified.
10	Not Zone	A name specified in the message is not within the zone specified in the message.



**Figure 57-2:** DNS message header format

## DNS Question Section Format

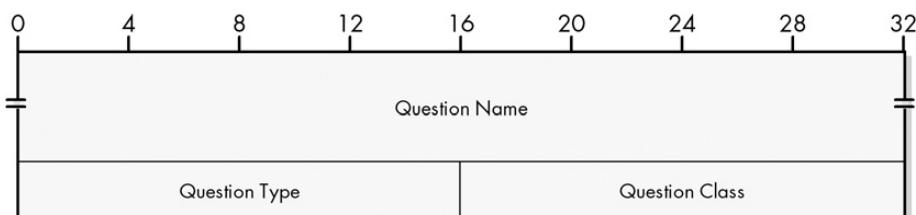
DNS queries always contain at least one entry in the Question section that specifies what the client in the exchange is trying to find out. These entries are copied to the response message unchanged, for reference on the part of the client if needed. The format used for each entry in the Question section of a DNS message described in detail in Tables 57-5 and 57-6, and illustrated in Figure 57-3.

**Table 57-5:** DNS Message Question Section Format

Field Name	Size (Bytes)	Description
QName	Variable	Question Name: Contains the object, domain, or zone name that is the subject of the query, encoded using standard DNS name notation, which is explained later in this chapter.
QType	2	Question Type: Specifies the type of question being asked by the device acting as a client. This field may contain a code number corresponding to a particular type of RR being requested. (Table 55-1 in Chapter 55 contains the numbers for the most common RRs.) If so, this means the client is asking for that type of record to be sent for the domain name listed in QName. The QType field may also contain one of the codes listed in Table 57-6, corresponding to a special type of requests.
QClass	2	Question Class: Specifies the class of the RR being requested, normally the value 1 for Internet (IN). See the discussion of classes and RR types in Chapter 56 for an explanation. In addition, the QClass value 255 is defined to have the special meaning "any class."

**Table 57-6:** Question Section QType Values

QType Value	Question Type	Description
251	IXFR	A request for an incremental (partial) zone transfer, per RFC 1995
252	AXFR	A request for a zone transfer
253	MAILB	A request for mailbox-related records (RR types MB, MG, or MR; now obsolete)
254	MAILA	A request for mail agent RR (now obsolete; MX records are used instead)
255	* (asterisk)	A request for all records



**Figure 57-3:** DNS message Question section format

## DNS Message Resource Record Field Formats

As you've learned in this and the previous chapter, the exchange of information in DNS consists of a series of client/server transactions. Clients send requests, or *queries*, to servers, and the servers send back *responses*. DNS servers are database servers, and

they store DNS name database information in the form of RRs. The questions asked by clients are requests for information from a DNS server's database, and they are answered by the DNS server looking up the requested RRs and putting them into the DNS response message.

The Answer, Authority, and Additional sections of the overall DNS message format are the places where servers put DNS RRs to be sent back to a client. Each section consists of zero or more records, and in theory, any record can be placed in any section. The sections differ only in the semantics (meaning) that the client draws from a record being in one section rather than in another section.

RRs have two representations: binary and text. The text format is used for master files edited by humans and is discussed in the "DNS Master File Format" section later in this chapter. The binary representation consists of regular numeric and text fields, just like the other fields in the DNS message format.

### DNS Common RR Format

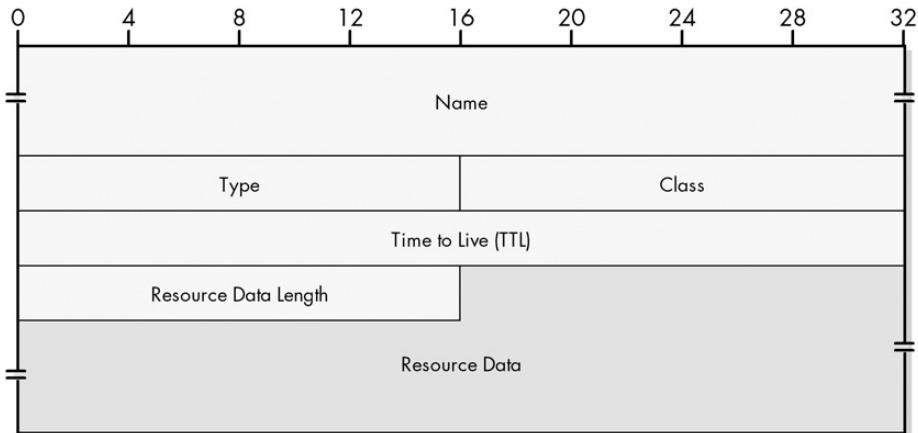
There are certain types of information that are common to all RRs and other types that are unique to each type of record. To handle this, all RRs are represented using a common field format, which contains a single RData field that varies by record type. The common RR format is described in Table 57-7 and illustrated in Figure 57-4.

**Table 57-7:** DNS Common Resource Record Format

Field Name	Size (Bytes)	Description
Name	Variable	Name: Contains the object, domain, or zone name that is the subject of the RR, encoded using standard DNS name notation, which is explained later in this chapter. All of the information in the RR is associated with this object, which I call the <i>named object</i> for the record.
Type	2	Type: A code value specifying the type of resource record. The type values for the most common kinds of RRs are shown in Table 55-1, in Chapter 55 and also in the following sections of this chapter.
Class	2	Class: Specifies the class of the RR being requested, normally the value 1 for Internet (IN). See Chapter 55 for an explanation.
TTL	4	Time to Live: Specifies the number of seconds that the record should be retained in the cache of the device reading the record. See the discussion of DNS name server caching in Chapter 55 for a full explanation. A value of 0 means to use this information for the current name resolution only; do not cache it.
RDLength	2	Resource Data Length: Indicates the size of the RData field, in bytes.
RData	Variable	Resource Data: The data portion of the RR.

### RData Field Formats for Common RRs

The RData field consists of one or more subfields that carry the actual payload for the RR. The following sections present the most common RR types. For each, I have indicated the RR text code, name, and type value; provided a brief summary of the RR's use; and shown the structure of the RData field in a table.



**Figure 57-4:** DNS common RR format

### A (Address) RR (Type Value 1)

A (Address) is the primary RR type in DNS. It contains a 32-bit IP address associated with a domain name, as shown in Table 57-8.

**Table 57-8:** DNS Address RR Data Format

Subfield Name	Size (Bytes)	Description
Address	4	Address: The 32-bit IP address corresponding to this record's named object.

### NS (Name Server) RR (Type Value 2)

The NSDName data field carries the domain name of a name server, as shown in Table 57-9.

**Table 57-9:** DNS Name Server RR Data Format

Subfield Name	Size (Bytes)	Description
NSDName	Variable	Name Server Domain Name: A variable-length name of a name server that should be authoritative for this record's named object. Like all names, this name is encoded using standard DNS name notation. A request for this RR type normally results in an A record for the name server specified also being returned in the Additional section of the response, if available.

### CName (Canonical Name) RR (Type Value 5)

The CName data field contains the real name of a named object that has been referenced using an alias, as shown in Table 57-10.

**Table 57-10:** DNS Canonical Name RR Data Format

Subfield Name	Size (Bytes)	Description
CName	Variable	Canonical Name: The canonical (real) name of the named object. This name is then resolved using the standard DNS resolution procedure to get the address for the originally specified name.

**SOA (Start Of Authority) RR (Type Value 6)**

The SOA record marks the start of a DNS zone and contains key information about how it is to be managed and used. The SOA record is the most complex of the DNS RR types. Its format is explained in Table 57-11 and illustrated in Figure 57-5. See the discussion of zone transfers in Chapter 55 for information about how the fields in this RR are used.

**Table 57-11:** DNS Start Of Authority RR Data Format

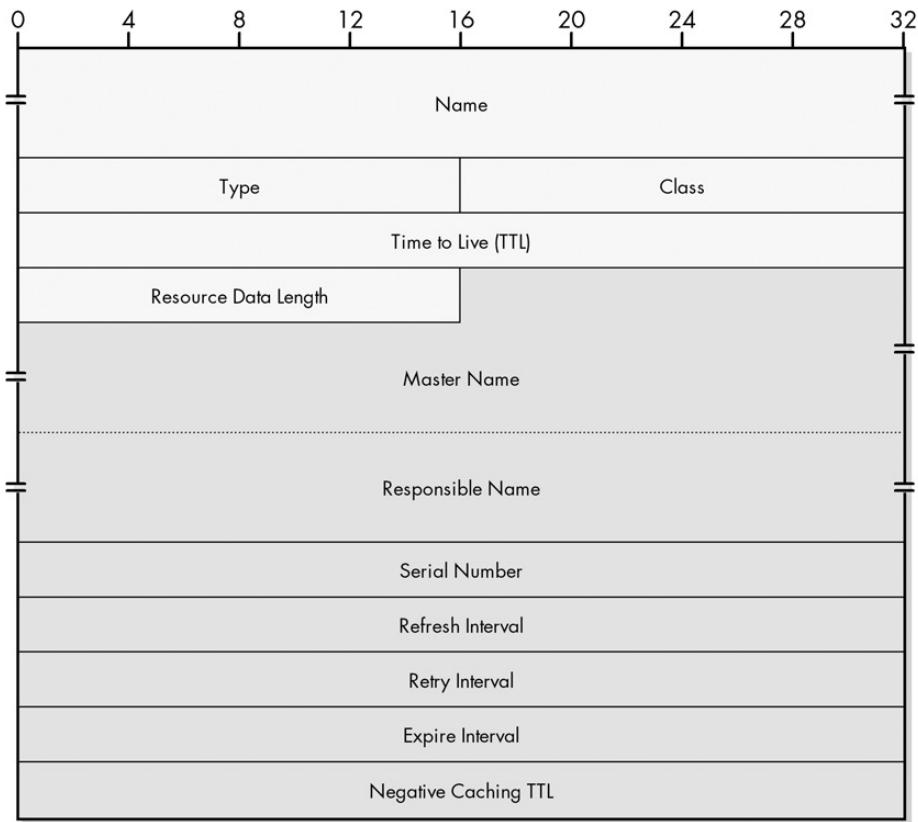
Subfield Name	Size (Bytes)	Description
MName	Variable	Master Name: The domain name of the name server that is the source of the data for the zone. This is normally the primary authoritative server for the zone. It is encoded using the standard DNS name format.
RName	Variable	Responsible Name: The email address of the person responsible for this zone. Email addresses in DNS are encoded using a special variation of the regular DNS name notation, discussed later in this chapter.
Serial	4	Serial Number: The serial number, or version number, of the RR database for this zone. Used to determine when changes have been made to the database to trigger zone transfers.
Refresh	4	Refresh Interval: The number of seconds that secondary name servers for this zone will wait between attempts to check for changes made to the zone database on the primary name server.
Retry	4	Retry Interval: The number of seconds a secondary name server waits before trying again to check with a primary for changes if its previous attempt failed.
Expire	4	Expire Interval: The number of seconds that can elapse between successful contacts with the primary name server before a secondary name server must consider the information it holds stale.
Minimum	4	Negative Caching TTL: Originally carried the default TTL value for records where no explicit TTL value was specified. Now represents the zone's negative cache TTL. See the discussion of DNS name server caching in Chapter 55.

**PTR (Pointer) RR (Type Value 12)**

The PTR record carries a pointer to an RR. It's used for reverse address lookups. It contains one data field, shown in Table 57-12.

**Table 57-12:** DNS Pointer RR Data Format

Subfield Name	Size (Bytes)	Description
PTRDName	Variable	Pointer Domain Name: A variable-length domain name. This is a name pointed to by the RR. See the description of reverse resolution in Chapter 56 for the most common way that this record type is used.



**Figure 57-5:** DNS Start Of Authority (SOA) RR data format

### MX (Mail Exchange) RR (Type Value 15)

The special MX record contains information about the mail server(s) to be used for sending email to the domain (see Chapter 56). Each record contains two fields, as shown in Table 57-13.

**Table 57-13:** DNS Mail Exchange RR Data Format

Subfield Name	Size (Bytes)	Description
Preference	2	Preference Value: The preference level for this mail exchange. Lower values signify higher preference.
Exchange	Variable	Exchange Domain Name: The domain name, encoded using standard DNS name notation, of a host willing to provide mail exchange services for this named object.

### TXT (Text) RR (Type Value 16)

The TXT record contains additional descriptive information about the named object, as shown in Table 57-14.

**Table 57-14:** DNS Text RR Data Format

Subfield Name	Size (Bytes)	Description
TXT-Data	Variable	Text Data: Variable-length descriptive text.

## DNS Name Notation and Message Compression

Obviously, the entire DNS protocol is oriented around dealing with names for domains, subdomains, and objects. As you've seen in the preceding topics, there are many fields in DNS messages and RRs that carry the names of objects, name servers, and so forth. DNS uses a special notation for encoding names in RRs and fields, a variation of this notation for email addresses, and a special compression method that reduces the size of messages for efficiency.

### **Standard DNS Name Notation**

In Chapter 53, you learned how DNS names are constructed. Each node in the name hierarchy has a label associated with it. The fully qualified domain name (FQDN) for a particular device consists of the sequence of labels that starts from the root of the tree and progresses down to that device. The labels at each level in the hierarchy are listed in sequence, starting with the highest level, from right to left, separated by dots. This results in the domain names we are used to working with, such as `www.xyzindustries.com`.

It would be possible to encode these names into RRs or other DNS message fields directly: Put the letter *w* into each of the first three bytes of the name, then put a dot (.) into the fourth byte, an *x* into the fifth byte, and so on. The disadvantage of this is that as a computer was reading the name, it wouldn't be able to tell when each name was finished. We would need to include a length field for each name.

Instead, DNS uses a special notation for DNS names. Each label is encoded, one after the next, in the name field. Before each label, a single byte is used that holds a binary number indicating the number of characters in the label. Then the label's characters are encoded, one per byte. The end of the name is indicated by a null label, representing the root; this has a length of zero, so each name ends with just a 0 character, indicating this zero-length root label.

Note that the dots between the labels aren't necessary, since the length numbers delineate the labels. The computer reading the name also knows how many bytes are in each label as it reads the name, so it can easily allocate space for the label as it reads it from the name.

For example, `www.xyzindustries.com` would be encoded as follows:

---

[3] w w w [13] x y z i n d u s t r i e s [3] c o m [0]

---

I have shown the label lengths in square brackets to distinguish them. Remember that these label lengths are binary encoded numbers, so a single byte can hold a value from 0 to 255; that [13] is one byte, not two, as you can see in Figure 57-6. Labels are actually limited to a maximum of 63 characters, and you'll see shortly why this is significant.

0	4	8	12	16	20	24	28	32
	3		w		w		w	
	13		x		y		z	
i		n		d		u		
s		t		r		i		
e		s		3		c		
o		m		0				

**Figure 57-6: DNS standard name notation** In DNS, every named object or other name is represented by a sequence of label lengths and then labels, with each label length taking one byte and each label taking one byte per character. This example shows the encoding of the name www.xyzindustries.com.

### DNS Electronic Mail Address Notation

Email addresses are used in certain DNS resource records, such as the RName field in the SOA RR. Email addresses take the form <name>@<domain-name>. DNS encodes these in exactly the same way as regular DNS domains, simply treating the @ like another dot. So, johnny@somewhere.org would be treated as johnny.somewhere.org and encoded as follows:

---

[6] j o h n n y [9] s o m e w h e r e [3] o r g [0]

---

Note that there is no specific indication that this is an email address. The name is interpreted as an email address instead of a device name based on context.

### DNS Message Compression

A single DNS message may contain many domain names. Now, consider that when a particular name server sends a response containing multiple domain names, they are all usually in the same zone or are related to the zone. Most of these names will have common elements to their names.

Consider our previous mail example of a client asking for an MX record for xyzindustries.com. The response to this client will contain, among other things, these two records:

**MX Record** An MX record that has xyzindustries.com in the Name field of the record and mail.xyzindustries.com in the RData field.

**A Record** Assuming the name server knows the IP address of mail.xyzindustries.com, the Additional section will contain an A record that has mail.xyzindustries.com in the Name field and its address in the RData field.

This is just one small example of name duplication. It can be much more extreme with other types of DNS messages, with certain string patterns being repeated many times. Normally, this would require that each name be spelled out fully using the encoding method described here. But this would be wasteful, since a large portion of these names is common. To cut down on this duplication, a special technique called *message compression* is used.

### Using Message Compression to Avoid Duplication of a Full Name

Using message compression, instead of a DNS name encoded using the combination of labels and label lengths, a two-byte subfield represents a *pointer* to another location in the message where the name can be found. The first two bits of this subfield are set to 1 (the value 11 in binary), and the remaining 14 bits contain an *offset* that specifies where in the message the name can be found, counting the first byte of the message (the first byte of the ID field) as 0.

Let's go back to our example. Suppose that in the DNS message, the RData field of the MX record, containing mail.xyzindustries.com, begins at byte 47. In this first instance, we would find the name encoded in full as follows:

---

```
[4] m a i l [13] x y z i n d u s t r i e s [3] c o m [0]
```

---

However, in the second instance, where mail.xyzindustries.com shows up in the Name field of the A record, we would instead put two 1 bits, followed by the number 47 encoded in binary. So, this would be the 16-bit binary pattern 11000000 00101111, or two numeric byte values 192 and 47. This second instance now takes 2 bytes instead of duplicating the 24 bytes needed for the first instance of the name.

How does a device reading a Name field differentiate a pointer from a real name? This is the reason that 11 is used at the start of the field. Doing this guarantees that the first byte of the pointer will always have a value of 192 or larger. Since labels are restricted to a length of 63 or less, when the host reads the first byte of a name, if it sees a value of 63 or less in a byte, it knows this is a real name; a value of 192 or more means it is a pointer.

### Using Message Compression to Avoid Duplication of Part of a Name

The previous example shows how pointers can be used to eliminate duplication of a whole name: The name mail.xyzindustries.com was used in two places, and a pointer was used instead of the second. Pointers are even more powerful than this,

however. They can also be used to point to only part of a real name or can be combined with additional labels to provide a compressed representation of a name related to another name in a RR. This provides even greater space savings.

In the previous example, this means that even the first instance of mail xyzindustries.com can be compressed. Recall that the MX record will have xyzindustries.com in the Name field and mail.xyzindustries.com in the RData field. If the Name field of that record starts at byte 19, then we can encode the RData field as follows:

---

```
[4] m a i l [pointer-to-byte-19]
```

---

The device reading the record will get “mail” for the first label and then read “xyzindustries.com” from the Name field to get the complete name, mail xyzindustries.com.

Similarly, suppose we had a record in this same message that contained a reference to the parent domain for xyzindustries.com, which is “com.” This could simply be encoded as follows:

---

```
[pointer-to-byte-33]
```

---

The reason is that byte 33 is where we find the [3] c o m [0] part of the Name field containing [13] x y z i n d u s t r i e s [3] c o m [0].

## DNS Master File Format

DNS servers answer queries from clients by sending reply messages containing RRs. You have already seen in this chapter the binary message formats used to encode these RRs. These message formats are great for transmitted messages, because they are compact and efficient. Computers have no problem reading fields very quickly and knowing how to interpret a particular string of ones and zeros.

Humans, on the other hand, don’t deal well with cryptic codes in binary. Before an RR can be provided by a server, it is necessary for a human administrator to tell the server what those records are and what information they contain. To make this job easier, DNS includes a special text representation for zones and RRs. Administrators edit special *master files* that describe the zone and the records it contains. These files are then read into memory by the server’s DNS software and converted into binary form for responding to client requests. This is described in more detail in Chapter 56.

Each master file consists of a simple, flat text file that can be created with any sort of text editor. Each file contains a number of lines expressed using a simple set of syntax rules that describe a zone and the records within it. The basic syntactic rules for DNS master files are specified in RFC 1035, Section 5.1. Certain DNS implementations use their own variations on the syntax in the standard, though they are all pretty similar.

## DNS Common Master File Record Format

Just as all RRs are stored internally using a common field format, they also use a common master file format. Each record normally appears on a separate line of the file. This format is as follows, with optional fields shown in square brackets:

---

```
<domain-name> [<ttl>] <class> <type> <rdata>
```

---

The fields are as follows:

**<domain-name>** A DNS domain name, which may be either an FQDN or a partially qualified name (PQDN).

**<ttl>** A TTL value, in seconds, for the record. If omitted, the default TTL value for the zone is used. In fact, most RRs do not have a specified TTL and just use the default provided by the SOA record.

**<class>** The RR class. For modern DNS, this field is optional, and it defaults to IN, for Internet.

**<type>** The RR type, specified using a text code such as A or NS, not the numeric code.

**<rdata>** RR data, which is a set of space-separated entries that depends on the record type.

The **<rdata>** can be either a single piece of information or a set of entries, depending on the record type. In the case of longer record types, especially the SOA record, multiple entry **<rdata>** fields are spread over several lines and enclosed in parentheses; the parentheses make all the entries act as if they were on a single line. Note that if the **<ttl>** field is present, the order of it and the **<class>** field may be switched without any problems, because one is a number and the other text (IN).

## Use and Interpretation of Partially Qualified Domain Names (PQDNs)

Domain names may be mixed between FQDNs and PQDNs (described in Chapter 53). PQDNs are used to make master files faster to create and more readable, by cutting down on the common parts of names. They are sort of the human equivalent of DNS message compression. An FQDN is shown as a full domain name ending in a dot (.) to represent the DNS name tree root. A PQDN is given as just a partial name with no root, and is interpreted as an FQDN by the software reading the master file. (See the description of the \$ORIGIN directive in the next section for more information.)

It is important to remember the trailing dot to mark FQDNs. If the origin is xyzindustries.com and in its zone file the name bigisp.net appears, the server will read this as bigisp.net.xyzindustries.com—probably not what you want. Also, email addresses, such as the **<r-name>** field in the SOA record, have the @ of the email address converted to a dot, following the standard DNS convention.

## **Master File Directives**

In addition to RRs, most master file implementations also support the use of *directives*. These are commands that specify certain important pieces of information to guide how the master file is to be interpreted. The following are three of the most common directives:

**\$ORIGIN** Specifies the domain name that is appended to unqualified specifications. This is the base used to convert PQDNs to FQDNs. For example, if the origin is xyzindustries.com., then a PQDN such as “sales” will be interpreted as sales.xyzindustries.com. Once defined, the origin can be referenced by just using @ in place of a name, as you will see in the example of a sample master file shown at the end of this section.

**\$TTL** Specifies the default TTL value to be used for any RRs that do not specify a TTL value in the record itself. (This value was formerly specified by the Minimum field in the SOA record.)

**\$INCLUDE** Allows one master file to include the contents of another. This is sometimes used to save the duplication of certain entries that are common between zones.

## **Syntax Rules for Master Files**

There are a few other syntax rules for DNS master files, some of which are intended to save time or energy on the part of administrators:

**Multiple-Record Shorthand** If multiple consecutive records pertain to the same domain, the *<domain-name>* is specified for the first one and can be then be left blank for the subsequent ones. The server will assume that any RRs without a *<domain-name>* indicated apply to the last *<domain-name>* it saw.

**Comments** A semicolon (;) marks a comment. Any text from the semicolon until the end of the line is ignored.

**Escape Character** A backslash (\) is used to “escape” the special meaning of a character. For example, a double-quotation (quote) mark ("") is used to delimit text strings; a literal double-quote character is indicated by a backslash–double-quote combination (\").

**White Space** Tabs and spaces are used as delimiters and blank lines are ignored. For readability, most smart administrators indent using tabs to clarify which records belong with which names, and group records using blank lines and comments.

**Case** Like DNS domain names, master file entries are case-insensitive.

## **Specific RR Syntax and Examples**

The following sections show the specific formats and examples for each of the common RR types. The fields are basically the same as the ones explained in the DNS binary record formats. The examples include explanatory comments using the DNS comment format. Assume that these examples are for the zone googleplex.edu.

### **A (Address) RR**

The format for an A record is as follows:

---

```
<domain-name> [<ttl>] IN A <ip-address>
```

---

Here is an example:

---

```
admin1.googleplex.edu IN A 204.13.100.3 ; An FQDN  
admin2 IN A 204.13.100.44 ; A PQDN equivalent to  
                           ; admin2.googleplex.edu
```

---

### **NS (Name Server) RR**

The format for an NS record is as follows:

---

```
<domain-name> [<ttl>] IN NS <name-server-name>
```

---

Here is an example:

---

```
< googleplex.edu. IN NS custns.bigisp.net ; Secondary NS
```

---

### **CName (Canonical Name) RR**

The format for a CName record is as follows:

---

```
<domain-name> [<ttl>] IN CNAME <canonical-name>
```

---

Here is an example:

---

```
www IN CNAME bigserver ; www.googleplex.edu is really  
                           ; bigserver.googleplex.edu.
```

---

### **SOA (Start Of Authority) RR**

The format for an SOA record is as follows:

---

```
<domain-name> [<ttl>] IN SOA <m-name> <r-name> (  
    <serial-number>  
    <refresh-interval>
```

```
<retry-interval>
<expire-interval>
<default-ttl>)
```

---

Here is an example:

---

```
< googleplex.edu. IN SOA ns1.googleplex.edu it.googleplex.edu (
    42      ; Version 42 of the zone.
    21600   ; Refresh every 6 hours.
    3600    ; Retry every hour.
    604800  ; Expire after one week.
    86400)  ; Negative Cache TTL is one day.
```

---

### **PTR (Pointer) RR**

The format for a PTR record is as follows:

---

```
<reverse-domain-name> [<ttl>] IN PTR <domain-name>
```

---

Here is an example:

---

```
3.100.13.204.IN-ADDR.ARPA. IN PTR admin1.googleplex.edu.
```

---

Note that the PTR record would actually be in the IN-ADDR.ARPA domain.

### **MX (Mail Exchange) RR**

The format of an MX record is as follows:

---

```
<domain-name> [<ttl>] IN MX <preference-value> <exchange-name>
```

---

Here is an example:

---

```
googleplex.edu.      IN MX 10 mainmail.googleplex.edu.
                      IN MX 20 backupmail.googleplex.edu
```

---

### **TXT (Text) RR**

The format of a TXT record is as follows:

---

```
<domain-name> [<ttl>] IN TXT <text-information>
```

---

Here is an example:

---

```
googleplex.edu. IN TXT "Contact Joe at X321 for more info."
```

---

## **Sample Master File**

The following is a real-world example of a DNS master file, taken from my own pcguide.com server (slightly modified), hosted by (and DNS information provided by) the fine folks at pair.com. Note the use of @ as a shortcut to mean “this domain” (pcguide.com).

---

```
$ORIGIN pcguide.com.
@ IN SOA ns23.pair.com. root.pair.com. (
    2001072300 ; Serial
    3600        ; Refresh
    300         ; Retry
    604800      ; Expire
    3600 )       ; Minimum

@ IN NS ns23.pair.com.
@ IN NS ns0.ns0.com.

localhost IN A 127.0.0.1
@           IN A 209.68.14.80
@           IN MX 50 qs939.pair.com.

www   IN CNAME  @
ftp   IN CNAME  @
mail  IN CNAME  @
relay IN CNAME  relay.pair.com.
```

---

## **DNS Changes to Support IPv6**

Version 4 of the Internet Protocol (IPv4) is the basis of today’s Internet and the foundation upon which the TCP/IP protocol suite is built. While IPv4 has served us well for over two decades, it has certain important drawbacks that would limit internetworks of the future if it were to continue to be used. For this reason, the next generation of IP, IP version 6 (IPv6), has been in development for many years. IPv6 will eventually replace IPv4 and take TCP/IP into the future.

The change from IPv4 to IPv6 will have effects that ripple to other TCP/IP protocols, including DNS. DNS is a higher-level protocol, so you might think that based on the principle of layering, a change to IP should not affect it. However, this is another example of how strict layering doesn’t always apply. DNS works directly with IP addresses, and one of the most significant modifications that IPv6 makes to IP is in the area of addressing, so this means that using DNS on IPv6 requires some changes to how the protocol works.

## **IPv6 DNS Extensions**

In fact, because DNS is so architecturally distant from IP down there at layer 3, the changes required are not extensive. RFC 1886, “IPv6 DNS Extensions,” published in December 1995, was the Internet Engineering Task Force’s (IETF’s) first formalized attempt to describe the changes needed in DNS to support IPv6. It defines three specific modifications to DNS for IPv6:

**New RR Type—AAAA (IPv6 Address)** The regular DNS Address (A) RR is defined for a 32-bit IPv4 address, so a new one was created to allow a domain name to be associated with a 128-bit IPv6 address. The four As (AAAA) are a mnemonic to indicate that the IPv6 address is four times the size of the IPv4 address. The AAAA record is structured in very much the same way as the A record in both binary and master file formats; it is just much larger. The DNS RR Type value for AAAA is 28.

**New Reverse Resolution Hierarchy** A new hierarchical structure similar to IN-ADDR.ARPA is defined for IPv6 reverse lookups, but the IETF put it in a different top-level domain (TLD). The new domain is *IP6.INT* and is used in a way similar to how IN-ADDR.ARPA works. However, since IPv6 addresses are expressed in hexadecimal instead of dotted-decimal, IP6.INT has 16 subdomains 0 through F, and each of those has 16 subdomains 0 through F, and so on, 16 layers deep. Yes, this leads to a potentially frightfully large reverse resolution database!

**Changes to Query Types and Resolution Procedure** All query types that work with A records or result in A records being included in the Additional section of a reply must be changed to also handle AAAA records. Also, queries that would normally result in A records being returned in the Additional section must return the corresponding AAAA records only in the Answer section, not in the Additional section.

**KEY CONCEPT** Even though DNS resides far above IP in the TCP/IP protocol suite architecture, it works intimately with IP addresses. For this reason, changes are required to allow it to support the new IPv6. These changes include the definition of a new IPv6 address RR (AAAA), a new reverse resolution domain hierarchy, and certain changes to how messaging is performed.

## **Proposed Changes to the IPv6 DNS Extensions**

In 2000, the IETF published RFC 2874, “DNS Extensions to Support IPv6 Address Aggregation and Renumbering.” This standard proposed a replacement for the IPv6 support introduced in RFC 1886, using a new record type, A6, instead of RFC 1886’s AAAA. The main difference between AAAA and A6 records is that the former are just whole addresses like A records, while A6 records can contain either a whole or partial address.

The idea behind RFC 2874 was that A6 records could be set up in a manner that complements the IPv6 format for unicast addresses (see Chapter 25). Then name resolution would involve a technique called *chaining* to determine a full address for a name from a set of partially specified address components. In essence, this would make the addresses behave much the way hierarchical names themselves work, providing some potential flexibility benefits.

For a couple of years, both RFC 1886 and RFC 2874 were proposed standards, and this led to considerable confusion. In August 2002, RFCs 3363 and 3364 were published, which clarified the situation with these two proposals. RFC 3363 represents the “Supreme Court decision,” which was that RFC 2874 and the A6 record be changed to experimental status and the AAAA record of RFC 1886 be kept as the DNS IPv6 standard.

The full explanation for the decision can be found in RFC 3364. In a nutshell, it boiled down to the IETF believing that there were significant potential risks in the successful implementation of RFC 2874. While the capabilities of the A6 record were interesting, it was not clear that they were needed, and given those risks, the IETF felt that sticking with RFC 1886 was the better move.

# PART III-2

## **NETWORK FILE AND RESOURCE SHARING PROTOCOLS**

To the typical end user, networks were created for one main reason: to permit the sharing of information. Most information on computers exists in the form of files that reside on storage devices such as hard disks; thus, one primary purpose of networks is to let users share files. File transfer and message transfer protocols allow users to manually move files from one place to the next, but a more automated method is preferable in many cases. Internetworking protocols provide such capabilities in the form of *network file and resource sharing protocols*.

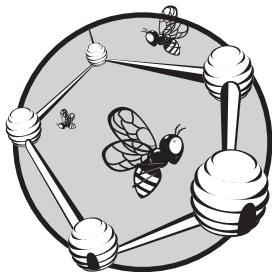
In this brief part, I describe network file and resource sharing protocols from the standpoint of TCP/IP networks. The one chapter here provides an overview of the concepts and operation of this class of protocols, discussing some of the elements common to the different types. It then describes the most common one defined specifically for TCP/IP: the Network File System (NFS).

Obviously, network file and resource sharing protocols and services are closely related to the file and message transfer protocols I mentioned earlier. For example, NFS can be used to accomplish tasks similar to those performed by TCP/IP file and message transfer applications such as the File Transfer Protocol (FTP) and the Hypertext Transfer Protocol (HTTP). I consider

those protocols more like specific end-user applications unto themselves, and therefore describe them in later parts on application protocols (FTP in Part III-6 and HTTP in Part III-8). I realize that this distinction between manual and automatic file transfer is somewhat arbitrary, but then, so are a lot of other things in the great world of networking.

# 58

## **NETWORK FILE AND RESOURCE SHARING AND THE TCP/IP NETWORK FILE SYSTEM (NFS)**



File and resource sharing protocols are important because they let users seamlessly share files over a network. Due to the dominance of Microsoft operating systems in the industry, many people are familiar with the way Microsoft networking can be used in this way. However, Microsoft is somewhat of a “Johnny come lately” to file sharing protocols. Long before Microsoft Windows even existed, the *Network File System (NFS)* was letting users share files over a network using the UNIX operating system.

In this chapter, I provide a brief look at network file and resource sharing in TCP/IP, with a focus on the operation of NFS. I begin with a general look at file and resource sharing protocol concepts. Then I provide an overview and history of NFS, and discuss its common versions and standards. I describe the architecture of NFS and the three components that compose it. I then describe the NFS file system model and how data is encoded using the *External Data Representation (XDR)* standard. I explain the client/server operation

of NFS using *Remote Procedure Calls (RPCs)*. I then list the procedures and operations used in NFS, and conclude with a description of the separate NFS Mount protocol, used to attach network resources to a device.

## File and Resource Sharing Concepts and Components

A primary reason why networks and internetworks are created is to allow files and other resources to be shared among computers. Thus, in any internetworking protocol stack, we need some mechanism by which users can easily move files across a network in a simple way. Application layer file and message transfer protocols like the File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP) were created for just this purpose: to let users access resources across a network while hiding the details of how the network operates at the layers below them.

However, even though these protocols hide the lower layers, they are somewhat *manual* in nature. They require a user to invoke an application protocol and use specific commands that accomplish network-based resource access. In fact, the problem with such protocols isn't so much that they require manual intervention, but that they make sharing more difficult because they don't allow a file to be used directly on another resource.

Consider a protocol like FTP. It does let you share files between machines, but it draws a clear distinction between a file that is yours and a file that is someone else's. If you want to use a file on Joe's machine, you must transfer it to your machine, use it, and then transfer it back. Also, if you don't transfer the file back, Joe might never even see the updated version.

### The Power of File and Resource Sharing Protocols

The ultimate in file and resource sharing is achieved when we can hide even the details of where the files are located and the commands required to move them around. Such a system would use an *automatic* sharing protocol that lets files and resources be used over a network seamlessly. Once set up, a network resource in such a scheme can be used in much the same way that one on a local computer is. Such protocols are sometimes called *network file and resource sharing protocols*.

It is this blurring of the line between a local file and a remote one that makes file and resource sharing protocols so powerful. Once the system is set up, users can access resources on another host as readily as on their own host. This is an extremely useful capability, especially in the modern era of client/server computing. For example, it allows a company to store information that is used by many individuals in a common place, such as in a directory on a server, where each of those individuals can access it. In essence, there is a virtual file system that spans network devices, instead of being simply on one storage device on a single computer.

### Components of a File and Resource Sharing Protocol

File and transfer protocols allow users to share files effortlessly, but that doesn't mean no work is involved. The work is still there, but it's shouldered by those who

write the protocol and those who administer its operation. Generally speaking, these protocols require at least the following general components:

**File System Model and Architecture** A mechanism for defining resources and files to be shared, and for describing how the virtual file system works.

**Resource Access Method** Procedures that describe how users can attach or detach a distant resource from their local host.

**Operation Set** A set of operations for accomplishing various tasks that the users need to perform on files on other hosts.

**Messaging Protocol** Message formats that carry operations to be performed, status information, and more, and a protocol for exchanging these messages between devices.

**Administrative Tools** Miscellaneous functionality needed to support the operation of the protocol and tie the other elements together.

## NFS Design Goals, Versions, and Standards

The histories of TCP/IP and the Internet are inextricably linked, as I discussed in Chapter 8. However, there is a third partner that is less often mentioned but very much part of the development history of these technologies. That is the operating system that ran on the machines in the early Internet and is still used on a large percentage of Internet servers today: the *UNIX* operating system.

Sun Microsystems was one of the early pioneers in the development of UNIX and in TCP/IP networking. Early in the evolution of TCP/IP, certain tools were created to allow a user to access another machine over the network—after all, this is arguably the entire point of networking. Remote-access protocols such as Telnet allowed a user to log in to another host computer and use resources there. FTP allowed people to copy a file from a distant machine to their own and edit it. However, neither of these solutions really fit the bill of allowing a user to access a file on a remote machine in a way similar to how a local file is used. To fill this need, Sun created the *Network File System (NFS)*.

### NFS Design Goals

NFS was specifically designed with the goal of eliminating the distinction between a local and a remote file. To a user, after the appropriate setup is performed, a file on a remote computer can be used as if it were on a hard disk on the user’s local machine. Sun also crafted NFS specifically to be vendor-independent, to ensure that both hardware made by Sun and that made by other companies could interoperate.

One of the most important design goals of NFS was performance. Obviously, even if you set up a file on a distant machine as if it were local, the actual read and write operations must travel across a network. Usually, this takes more time than simply sending data within a computer, so the protocol itself needed to be as lean and mean as possible. This decision led to some interesting choices, such as the use

of the unreliable User Datagram Protocol (UDP) for transport in TCP/IP, instead of the reliable Transmission Control Protocol (TCP), as with most file transfer protocols. This, in turn, has interesting implications on how the protocol works as a whole.

Another key design goal for NFS was simplicity (which of course is related to performance). NFS servers are said to be *stateless*, which means that the protocol is designed so that servers do not need to keep track of which files have been opened by which clients. This allows requests to be made independently of each other, and allows a server to gracefully deal with events such as crashes without the need for complex recovery procedures.

The protocol is also designed so that if requests are lost or duplicated, file corruption will not occur.

**KEY CONCEPT** The Network File System (NFS) was created to allow client hosts to access files on remote servers as if they were local. It was designed primarily with the goals of performance, simplicity, and cross-vendor compatibility.

## NFS Versions and Standards

Since it was initially designed and marketed by Sun, NFS began as a de facto standard. The first widespread version of NFS was version 2 (NFSv2), and this is still the most common version of the protocol. NFSv2 was eventually codified as an official TCP/IP standard when RFC 1094, “NFS: Network File System Protocol Specification,” was published in 1989.

NFS version 3 (NFSv3) was subsequently developed, and it was published in 1995 as RFC 1813, “NFS Version 3 Protocol Specification.” It is similar to NFSv2, but makes a few changes and adds some new capabilities. These include support for larger files and file transfers, better support for setting file attributes, and several new file access and manipulation procedures.

NFS version 4 (NFSv4) was published in 2000 as RFC 3010, “NFS Version 4 Protocol.” Where NFSv3 contained only relatively small changes to the previous version, NFSv4 is virtually a rewrite of NFS. It includes numerous changes, most notably the following:

- Reflecting the needs of modern internetworking, NFSv4 puts greater emphasis on security.
- NFSv4 introduces the concept of a *compound* procedure, which allows several simpler procedures to be sent from a client to a server as a group.
- NFSv4 almost doubles the number of individual procedures that a client can use in accessing a file on an NFS server.
- NFSv4 makes a significant change in messaging, with the specification of TCP as the transport protocol for NFS.
- NFSv4 integrates the functions of the Mount protocol into the basic NFS protocol, eliminating it as a separate protocol as it is in previous versions.

The NFSv4 standard also has a lot more details about implementation and optional features than the earlier standards—it's 275 pages long. So much for simplicity! RFC 3010 was later updated by RFC 3530, “Network File System (NFS) Version 4 Protocol,” in April 2003. This standard makes several further revisions and clarifications to the operation of NFSv4.

## NFS Architecture and Components

NFS follows the classic TCP/IP client/server model of operation. A hard disk or a directory on a storage device of a particular computer can be set up by an administrator as a shared resource. This resource can then be accessed by client computers, which *mount* the shared drive or directory, causing it to appear as if it were a local directory on the client machine. Some computers may act as only servers or only clients; others may be both, sharing some of their own resources and accessing resources provided by others.

Considered from the perspective of the TCP/IP protocol suite as a whole, NFS is a single protocol that resides at the application layer of the TCP/IP (DOD) model (described in Chapter 8). This TCP/IP layer encompasses the session, presentation, and application layers of the OSI Reference Model (described in Chapter 6). As I have said before in this book, I don’t see much value in trying to differentiate between layers 5 through 7 most of the time. In some situations, however, these layers can be helpful in understanding the architecture of a protocol, and that’s the case with NFS.

### NFS Main Components

The operation of NFS is defined in the form of three main components that can be viewed as logically residing at each of the three OSI model layers corresponding to the TCP/IP application layer, as illustrated in Figure 58-1:

**Remote Procedure Call (RPC)** RPC is a generic session layer service used to implement client/server internetworking functionality. It extends the notion of a program calling a local procedure on a particular host computer to the calling of a procedure on a remote device across a network.

**External Data Representation (XDR)** XDR is a descriptive language that allows data types to be defined in a consistent manner. XDR conceptually resides at the presentation layer. Its universal representations allow data to be exchanged using NFS between computers that may use very different internal methods of storing data.

**NFS Procedures and Operations** The actual functionality of NFS is implemented in the form of procedures and operations that conceptually function at layer 7 of the OSI model. These procedures specify particular tasks to be carried out on files over the network, using XDR to represent data and RPC to carry the commands across an internetwork.

These three key “subprotocols,” if you will, compose the bulk of the NFS protocol. Each is described in more detail in a separate section in this chapter.

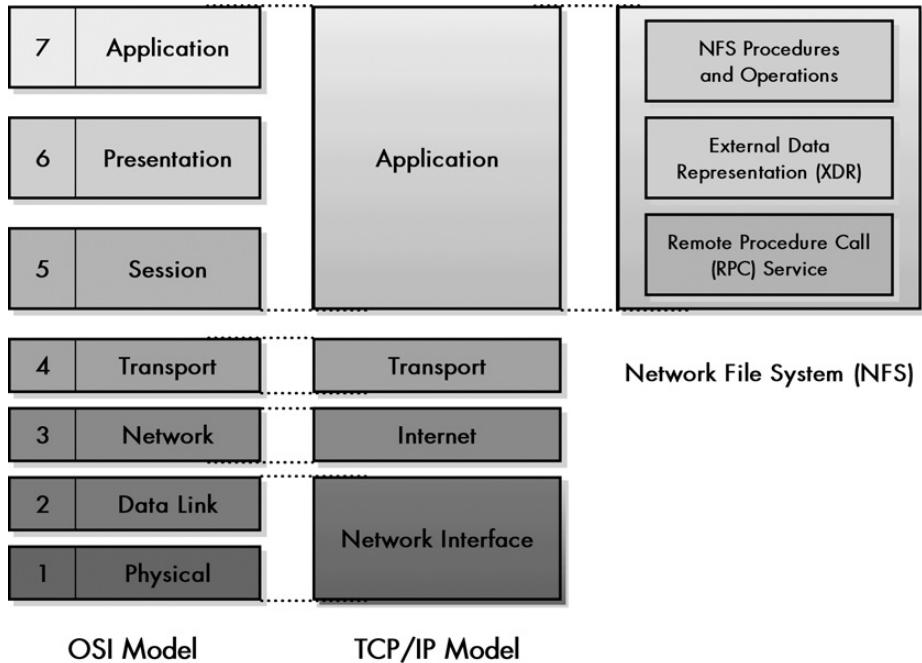


Figure 58-1: NFS architectural components

**KEY CONCEPT** NFS resides architecturally at the TCP/IP application layer. Even though in the TCP/IP model no clear distinction is made generally between the functions of layers 5 through 7 of the OSI Reference Model, NFS's three subprotocols correspond well to those three layers as shown. NFS resides architecturally at the application layer of the TCP/IP model. Its functions are implemented primarily through three distinct functional components that implement the functions of layers 5 through 7 of the OSI Reference Model: the *Remote Procedure Call (RPC)*, which provide session-layer services; the *External Data Representation (XDR)* standard, which manages data representation and conversion; and *NFS procedures and operations*, which allow application layer tasks to be performed using the other two components.

### Other Important NFS Functions

Aside from its three main components, the NFS protocol as a whole involves a number of other functions, most notably the following:

**Mount Protocol** A specific decision was made by the creators of NFS to not have NFS deal with the particulars of file opening and closing. Instead, a separate protocol called the *Mount* protocol is used for this purpose. Accessing a file or other resource over the network involves first *mounting* it using this protocol. The Mount protocol is architecturally distinct, but obviously closely related to NFS, and is even defined in an appendix of the NFS standard. I describe it in the last section in this chapter. (Note that in NFSv4, the functions of the Mount protocol have been incorporated into NFS proper.)

**NFS File System Model** NFS uses a particular model to implement the directory and file structure of the systems that use it. This model is closely based on the file system model of UNIX, but is not specific to only that operating system. It is discussed in conjunction with the explanation of the Mount protocol at the end of this chapter.

**Security** Versions 2 and 3 of NFS include only limited security provisions. They use UNIX-style authentication to check permissions for various operations. NFSv4 greatly increases the security options available for NFS implementations. These include provisions for multiple authentication and encryption algorithms, and many changes to the protocol as a whole to make it more secure.

## **NFS Data Definition with the External Data Representation (XDR) Standard**

The overall idea behind NFS is to allow you to read from or write to a file on another computer as readily as you do on your local machine. Of course, the files on your local machine are all stored in the same file system, using the same file structure and the same means of representing different types of data. You can't be sure that this will be the case when accessing a remote device, and this creates a bit of a Tower of Babel problem.

One approach would be to simply restrict access only to remote files on machines that use the same operating system. However, this would remove much of the effectiveness of NFS. It would also be highly impractical to require every computer to understand the internal representation of every other one. A more general method is needed to allow even very dissimilar machines to share data. To this end, the creators of NFS defined NFS so that it deals with data using a universal data description language. This language is called the *External Data Representation (XDR)* standard and was originally described in RFC 1014. It was updated in RFC 1832, “XDR: External Data Representation Standard,” in 1995.

### **A Method of Universal Data Exchange: XDR**

The idea behind XDR is simple, and it can be easily understood in the form of an analogy. If you had delegates speaking 50 different languages at a convention, they would have a hard time communicating. You could hire translators to facilitate, but you would never find translators to handle all the different possible combinations of languages. A more practical solution is to declare one language, such as English, to be a common language. You then need only 49 translators: one to translate from English to each of the non-English languages and back again. To translate from Swedish to Portuguese, you translate from Swedish to English and then from English to Portuguese. The common language could be French, Spanish, or something else, as long as a translator could be found from all the other languages.

XDR works in the same manner. When information about how to access a file is to be transferred from Device A to Device B, Device A first converts it from Device A's internal representation to the XDR representation of those data types. The information is transmitted across the network using XDR encoding. Then Device B translates from XDR back to its own internal representation, so it can be presented

to the user as if it were on the local file system. Each device needs to know only how to convert from its own language to XDR and back again; Device A doesn't need to know Device B's internal details and vice versa. This sort of translation is a classic job of the presentation layer, which is where XDR resides in the OSI Reference Model. XDR is itself based on an International Organization for Standardization (ISO) standard called "Abstract Syntax Notation."

**NOTE** *The idea behind XDR is also used in other protocols to allow the exchange of data independent of the nature of the underlying systems. For example, a similar idea is behind the way management information is exchanged using the Simple Network Management Protocol (SNMP), which is described in Chapter 66. The same basic idea underlies the important Network Virtual Terminal (NVT) paradigm used in the Telnet protocol, which is described in Chapter 87.*

**KEY CONCEPT** The purpose of the *External Data Representation (XDR)* standard is to define a common method for representing common data types. Using this universal representation, data can be exchanged between devices, regardless of what internal file system each uses. This enables NFS to exchange file data between clients and servers that may be implemented using very different hardware and software platforms.

## XDR Data Types

For XDR to be universal, it must allow the description of all the common types of data that are used in computers. For example, it must allow integers, floating-point numbers, strings, and other data constructs to be exchanged. The XDR standard describes the structure of many data types using a notation somewhat similar to the C programming language. As you may know, this is one of the most popular languages in computing history, and it is closely associated with UNIX (and thus, certain TCP/IP technologies as well).

Table 58-1 shows the data types defined by XDR, which can be used by NFS in exchanging data between the client and server. For each, I have included the data type code, its size in bytes, and a brief description.

**Table 58-1:** NFS External Data Representation (XDR) Data Types

Data Type Code	Size (Bytes)	Description
int	4	Signed integer: A 32-bit signed integer in two's complement notation, capable of holding a value from -2,147,483,648 to +2,147,483,647.
unsigned int	4	Unsigned integer: A 32-bit unsigned integer, from 0 to 4,294,967,295.
enum	4	Enumeration: An alternate way of expressing a signed integer where some of the integer values are used to stand for particular constant values. For example, you could represent the colors of the rainbow, by defining the value 1 to stand for PURPLE, 2 to stand for BLUE, and so on.
bool	4	Boolean: A logical representation of an integer, analogous to a two-level enumeration where a value of 0 is defined as FALSE and 1 is TRUE.
hyper	8	Signed hyper integer: Same as a regular signed integer, but 8 bytes wide to allow much larger numbers.
unsigned hyper	8	Unsigned hyper integer: Same as a regular unsigned integer but 8 bytes wide to allow much larger numbers.

(continued)

**Table 58-1: NFS External Data Representation (XDR) Data Types (continued)**

<b>Data Type Code</b>	<b>Size (Bytes)</b>	<b>Description</b>
float	4	Floating-point number: A 32-bit signed floating-point number. 1 bit holds the sign (positive or negative), 8 bits hold the exponent (power), in base 2, and 23 bits hold the mantissa (fractional part of the number).
double	8	Double-precision floating-point number: The same as float but with more bits to allow greater precision. 1 bit is for the sign, 11 bits for the exponent, and 52 bits for the mantissa.
quadruple	16	Quadruple-precision floating-point number: The same as float and double but with still more bits to allow greater precision. 1 bit is for the sign, 15 bits for the exponent, and 112 bits for the mantissa.
opaque	Variable	Opaque data: Data that is to be passed between devices without being given a specific representation using XDR. The term <i>opaque</i> means that the data is treated like a “black box” whose insides cannot be seen. Obviously, any machines using this data type must themselves know how to deal with it, since NFS does not.
string	Variable	String: A variable-length string of ASCII characters.
(array)	Variable	Arrays: A group of any single type of the elements above, such as integers, floating-point numbers, and so on, may be specified in an array to allow many to be referenced as a single unit. They are not indicated using a separate data type code.
struct	Variable	Structure: An arbitrary structure containing other data elements from this table. This allows the definition of complex data types.
union	Variable	Discriminated union: A complex data type where a code value called a “discriminant” is used to determine the nature of the rest of the structure. See section 3.14 of RFC 1014 for details.
void	0	Void: A null data type that contains nothing.
const	0	Constant: A constant value used in other representations.

As you can see, XDR provides considerable data description capabilities. If you know the C language, much of what is in Table 58-1 is probably familiar to you. Unfortunately, I can’t really describe many of the more complex data types without turning this into a guide to C programming.

XDR also provides a means of defining new data types and a method for specifying optional data. This offers even more flexibility beyond the large number of specific types already specifically described. Each version of NFS has a slightly different list of data types it supports.

## **NFS Client/Server Operation Using Remote Procedure Calls (RPCs)**

Almost all applications deal with files and other resources. When a software program on a particular computer wants to read a file, write a file, or perform related tasks, it needs to use the correct software instructions for this purpose. It would be inefficient to require each software program to contain a copy of these instructions, so instead, they are encoded as standardized software modules, sometimes called *procedures*. To perform an action, a piece of software *calls* the procedure. The procedure temporarily takes over for the main program and

performs a task such as reading or writing data. The procedure then returns control of the program back to the software that called it, and optionally, returns data as well.

Since the key concept of NFS was to make remote file access look like local file access, it was designed around the use of a network-based version of this procedure calling method. A software application that wants to do something with a file still makes a procedure call, but it makes the call to a procedure on a different computer instead of the local one. A special set of routines is used to handle the transmission of the call across the network, in a way largely invisible to software performing the call.

This functionality could have been implemented directly in NFS, but instead Sun created a separate session-layer protocol component called the *Remote Procedure Call (RPC)* specification, which defines how this works. RPC was originally created as a subcomponent of NFS, but it is generic enough and useful enough that it has been used for other client/server applications in TCP/IP. For this reason, it is really considered in many respects a distinct protocol.

Because RPC is the actual process of communicating in NFS, NFS itself is different from many other TCP/IP protocols. Its operation can't be described in terms of specific message exchanges and state diagrams the way a protocol like HTTP or the Dynamic Host Configuration Protocol (DHCP), or even TCP can, because RPC does all of that. NFS is defined in terms of a set of RPC server procedures and operations that an NFS server makes available to NFS clients. These procedures and operations each allow a particular type of action to be taken on a file, such as reading from it, writing to it, or deleting it.

### **RPC Operation and Transport Protocol Usage**

When a client wants to perform some type of action on a file on a particular machine, it uses RPC to make a call to the NFS server on that machine. The server accepts the request and performs the action required, then returns a result code and possibly data back to the client, depending on the request. The result code indicates if the action was successful. If it was, the client can assume that whatever it asked to be done was completed. For example, in the case of writing data, the client can assume the data has been successfully written to long-term storage.

**KEY CONCEPT** NFS does not use a dedicated message format, like most other protocols do. Instead, clients and servers use the *Remote Procedure Call (RPC)* protocol to exchange file operation requests and data.

NFS can operate over any transport mechanism that has a valid RPC implementation at the session layer. NFS has seen an evolution of sorts in its use of transport protocol. The NFSv2 standard says that it operates normally using UDP, and this is still a common way that NFS information is carried. NFSv3 says that either UDP or TCP may be used, but NFSv4 specifies TCP to carry data. The nominal registered port number for use by NFS is 2049, but other port numbers are sometimes used for NFS, through the use of RPC's *port mapper* capability.

## **Client and Server Responsibilities in NFS**

Since UDP is unreliable, the use of that protocol to transport important information may seem strange. For example, we obviously don't want data that we are trying to write to a file to be lost in transit. Remember, however, that UDP doesn't preclude the use of measures to ensure reliable communications; it simply doesn't provide those capabilities itself. UDP can be used by NFS because the protocol itself is designed to tolerate loss of transmitted data and to recover from it.

Consistent with this concept, the general design of NFS puts most of the responsibility for implementing the protocol on the client, not the server. As the NFSv3 standard says, "NFS servers are dumb, and NFS clients are smart." What this means is that the servers focus only on responding to requests, while clients must take care of most of the nitty-gritty details of the protocol, including recovery from failed communications. This is a common requirement when UDP is used, because if a client request is lost in transit, the server has no way of knowing that it was ever sent.

As mentioned in the NFS overview earlier in this chapter, NFS servers are designed to be stateless. In simplified terms, this means that the NFS server does not keep track of the state of the clients using it from one request to another. Each request is independent of the previous one, and the server in essence has no memory of what it did before when it gets a new command from a client. This again requires more intelligence to be put into the clients, but has the important advantage of simplifying recovery in the case that the server crashes. Since there is nothing that the server was keeping track of for the client, there's nothing that can be lost. This is an important part of ensuring that files are not damaged as a result of network problems or congestion.

## **Client and Server Caching**

Both NFS clients and servers can make use of caching to improve performance. Servers may use caching to store recently requested information in case it is needed again. They may also use *predictive* caching, sometimes called *prefetching*. In this technique, a server that receives a request to read a block of data from a file may load into memory the next block after it, on the theory that it will likely be requested next.

Client-side caching is used to satisfy repeat NFS requests from applications while avoiding additional RPC calls. Like almost everything else about NFS, caching is implemented much more thoroughly in NFSv4 than in the previous versions.

**KEY CONCEPT** NFS is designed to be a *stateless* protocol, with intelligent clients and relatively dumb servers that respond to requests and do not maintain status information about what files are in use. NFS was originally designed to use UDP for transport, for efficiency purposes. This requires that NFS clients take care of detecting lost requests and retransmitting them. NFSv4 uses TCP to take advantage of TCP's reliability and other features.

## NFS Server Procedures and Operations

The actual exchange of information between an NFS client and server is performed by the underlying RPC protocol. NFS functionality is therefore described not in terms of specific protocol operations, but by delineating the different actions that a client may take on files residing on a server. In the original version of NFS, NFSv2, these are called NFS *server procedures*.

Each procedure represents a particular action that a client may perform, such as reading from a file, writing to a file, or creating or removing a directory. The operations performed on the file require that the file be referenced using a data structure called a *file handle*. As the name suggests, the file handle, like the handle of a real object, lets the client and server “grasp” the file. The Mount protocol, described later in this chapter, is used to mount a file system, to enable a file handle to be accessed for use by NFS procedures.

NFSv3 uses the same basic model for server procedures, but makes certain changes. Two of the NFSv2 procedures were removed, and several new ones added to support new functionality. The numbers assigned to identify each procedure were also changed.

### NFS Version 2 and Version 3 Server Procedures

Table 58-2 shows the server procedures defined in versions 2 and 3 of NFS. The table shows the procedure numbers for both NFSv2 and NFSv3, as well as the name of each procedure and a description of what it does. I have kept the descriptions short so the table can serve as a useful summary of what NFS can do. They are listed in order of the procedure number used in NFSv2.

**Table 58-2:** NFS Version 2 and Version 3 Server Procedures

Procedure No. (v2)	Procedure No. (v3)	Procedure Name	Procedure Summary	Description
0	0	null	Do nothing	Dummy procedure provided for testing purposes.
1	1	getattr	Get file attributes	Retrieves the attributes of a file on a remote server.
2	2	setattr	Set file attributes	Sets (changes) the attributes of a file on a remote server.
3	—	root	Get file system root (obsolete)	This procedure was originally defined to allow a client to find the root of a remote file system, but is now obsolete. This function is instead now implemented as part of the Mount protocol. It was removed in NFSv3.
4	3	lookup	Look up filename	Returns the file handle of a file for the client to use.
5	5	readlink	Read from symbolic link	Reads the name of a file specified using a symbolic link.
6	6	read	Read from file	Reads data from a file.
7	—	writecache	Write to cache	Proposed for future use in NFSv2 but abandoned and removed from NFSv3.

(continued)

**Table 58-2:** NFS Version 2 and Version 3 Server Procedures (continued)

<b>Procedure No. (v2)</b>	<b>Procedure No. (v3)</b>	<b>Procedure Name</b>	<b>Procedure Summary</b>	<b>Description</b>
8	7	write	Write to file	Writes data to a file.
9	8	create	Create file	Creates a file on the server.
10	12	remove	Remove file	Deletes a file from the server.
11	14	rename	Rename file	Changes the name of a file.
12	15	link	Create link to file	Creates a hard (nonsymbolic) link to a file.
13	10	symlink	Create symbolic link	Creates a symbolic link to a file.
14	9	mkdir	Create directory	Creates a directory on the server.
15	13	rmdir	Remove directory	Deletes a directory.
16	16	readdir	Read from directory	Reads the contents of a directory.
17	—	statfs	Get file system attributes	Provides to the client general information about the remote file system, including the size of the file system and the amount of free space remaining. In NFSv3, this was replaced by fsstat and fsinfo.
—	4	access	Check access permission	Determines the access rights that a user has for a particular file system object. This is new in NFSv3.
—	11	mknod	Create a special device	Creates a special file such as a named pipe or device file. This is new in NFSv3.
—	17	readdirplus	Extended read from directory	Retrieves additional information from a directory. This is new in NFSv3.
—	18	fsstat	Get dynamic file system information	Returns volatile (dynamic) file system status information such as the current amount of file system free space and the number of free file slots. This is new in NFSv3.
—	19	fsinfo	Get static file system information	Returns static information about the file system, such as general data about how the file system is used and parameters for how requests to the server should be structured. This is new in NFSv3.
—	20	pathconf	Retrieve POSIX information	Retrieves additional information for a file or directory. This is new in NFSv3.
—	21	commit	Commit cached data on a server to stable storage	Flushes any data that the server is holding in a write cache to storage. This is used to ensure that any data that the client has sent to the server but that the server has held pending write to storage is written out. This is new in NFSv3.

It is common that a client may want to perform multiple actions on a file, such as several consecutive reads. One of the problems with the server procedure system in NFSv2 and NFSv3 is that each client action required a separate procedure call. This was somewhat inefficient, especially when NFS was used over a high-latency link.

## NFS Version 4 Server Procedures and Operations

To improve the efficiency of server procedures, NFSv4 makes a significant change to the way that server procedures are implemented. Instead of each client action being a separate procedure, a single procedure, called a *compound procedure*, is defined. Within this compound procedure, a large number of *server operations* are encapsulated. These are all sent as a single unit, and the server interprets and follows the instructions in each operation in sequence.

This change means there are actually only two RPC procedures in NFSv4, as shown in Table 58-3.

**Table 58-3:** NFS Version 4 Server Procedures

Procedure Number	Procedure Name	Procedure Summary	Description
0	null	Do nothing	Dummy procedure provided for testing purposes.
1	compound	Compound operations	Combines a number of NFS operations into a single request.

All the real client actions are defined as operations within the compound procedure, as shown in Table 58-4. You'll notice that the number of NFSv4 operations is much larger than the number of procedures in NFSv2 and NFSv3. This is due both to the added features in NFSv4 and the fact that it incorporates functions formerly performed by the separate Mount protocol.

**Table 58-4:** NFS Version 4 Server Operations

Operation Number	Operation Name	Operation Summary	Description
3	access	Check access rights	Determines the access rights a user has for an object.
4	close	Close file	Closes a file.
5	commit	Commit cached data	Flushes any data that the server is holding in a write cache to storage, to ensure that any pending data is permanently recorded.
6	create	Create a nonregular file object	This is similar to the mknod procedure in NFSv3; it creates a "nonregular" (special) object file. (Regular files are created using the open operation.)
7	delepurge	Purge delegations awaiting recovery	NFSv4 has a feature where a server may delegate to a client responsibility for certain files. This operation removes delegations awaiting recovery from a client.
8	delegreturn	Return delegation	Returns a delegation from a client to the server that granted it.
9	getattr	Get attributes	Obtains the attributes for a file.
10	getfh	Get current file handle	Returns a file handle, which is a logical object used to allow access to a file.
11	link	Create link to a file	Creates a hard (nonsymbolic) link to a file.
12	lock	Create lock	Creates a lock on a file. Locks are used to manage access to a file—for example, to prevent two clients from trying to write to a file simultaneously and thus corrupting it.

(continued)

**Table 58-4:** NFS Version 4 Server Operations (continued)

<b>Operation Number</b>	<b>Operation Name</b>	<b>Operation Summary</b>	<b>Description</b>
13	lockt	Test for lock	Tests for the existence of a lock on an object and returns information about it.
14	locku	Unlock file	Removes a lock previously created on a file.
15	lookup	Look up filename	Looks up or finds a file.
16	lookupp	Look up parent directory	Returns the file handle of an object's parent directory.
17	nverify	Verify difference in attributes	Checks to see if attributes have changed on a file.
18	open	Open a regular file	Opens a file.
19	openattr	Open named attribute directory	Opens an attribute directory associated with a file.
20	open_confirm	Confirm open	Confirms information related to an opened file.
21	open_downgrade	Reduce open file access	Adjusts the access rights for a file that is already open.
22	putfh	Set current file handle	Replaces one file handle with another.
23	putpubfh	Set public file handle	Sets the current file handle to be the public file handle of the server. This may or may not be the same as the root file handle.
24	putrootfh	Set root file handle	Sets the current file handle to be the root of the server's file system.
25	read	Read from file	Reads data from a file.
26	readdir	Read directory	Reads the contents of a directory.
27	readlink	Read symbolic link	Reads the name of a file specified using a symbolic link.
28	remove	Remove file system object	Removes (deletes) an object.
29	rename	Rename directory entry	Changes the name of an object.
30	renew	Renew a lease	Renews an NFS delegation made by a server. (Note that these leases have nothing to do with DHCP leases, which are discussed in Chapter 61.)
31	restorefh	Restore saved file handle	Allows a file handle previously saved to be made the current file handle.
32	savefh	Save current file handle	Allows a file handle to be saved so it can later be restored when needed.
33	secinfo	Obtain available security	Retrieves NFS security information.
34	setattr	Set attributes	Changes one or more attributes of a file.
35	setclientid	Negotiate client ID	Allows a client to communicate information to the server regarding how the client wants to use NFS.
36	setclientid_confirm	Confirm client ID	Used to confirm the results of a previous negotiation using setclientid.

(continued)

**Table 58-4:** NFS Version 4 Server Operations (continued)

Operation Number	Operation Name	Operation Summary	Description
37	verify	Verify same attributes	Allows a client to verify certain attributes before proceeding with a particular action.
38	write	Write to file	Writes data to a file.
39	release_lockowner	Release lock owner state	Used by a client to tell a server to release certain information related to file locks.
10044	illegal	Illegal operation	A placeholder (dummy) operation used to support error reporting when an invalid operation is used in a request from a client.

**KEY CONCEPT** File operations in NFS are carried out using NFS *server procedures*. In versions 2 and 3 of NFS, each procedure performs one action, such as reading data from a file. In NFSv4, a special *compound* action is defined that allows many individual *operations* to be sent in a single request to a server.

## NFS File System Model and the Mount Protocol

Since NFS is used by a client to simulate access to remote directories of files as if they were local, the protocol must present the files from the remote system to the local user. Just as files on a local storage device are arranged using a particular file system, NFS uses a *file system model* to represent how files are shown to a user.

### The NFS File System Model

The file system model used by NFS is the same one that most of us are familiar with: a hierarchical arrangement of directories that contain files and subdirectories. The top of the hierarchy is the *root*, which contains any number of files and first-level directories. Each directory may contain more files or other directories, allowing an arbitrary tree structure to be created.

A file can be uniquely specified by using its *filename* and a *path name* that shows the sequence of directories one must traverse from the root to find the file. Since NFS is associated with UNIX, files in NFS discussions are usually shown in UNIX notation; for example, */etc/hosts*. The same basic tree idea can also be expressed using the method followed by Windows operating systems: *C:\WINDOWS\HOSTS*.

### The Mount Protocol

Before NFS can be used to allow a client to access a file on a remote server, the client must be given a way of accessing the file. This means that a portion of the remote file system must be made available to the client, and the file opened for access. A specific decision was made when NFS was created to not put file access, opening, and closing functions into NFS proper. Instead, a separate protocol was created to work with NFS, so that if the method of providing file access needed to be changed later, it wouldn't require changes to NFS itself. This separate mechanism is called the *Mount protocol* and is described in Appendix A of RFC 1094 (NFSv2). Note that while its functionally distinct, Mount is considered part of the overall NFS package.

When NFS was revised to version 3, the Mount protocol was similarly modified. The NFSv3 version of the Mount protocol is defined in Appendix I of RFC 1813 (NFSv3). It contains some changes to how the protocol works, but the overall operation of the two versions of Mount is pretty much the same.

The term *mount* is actually an analog to a hardware term that refers to making a physical storage volume available. In the past, storage devices were usually removable disk packs, and to use one, you mounted it onto a drive unit. In a similar manner, NFS resources are logically mounted using the Mount protocol, which makes the shared file system available to the client. A file can then be opened and a file handle returned to the NFS client, so it can reference the file for operations such as reading and writing.

**KEY CONCEPT** Versions 2 and 3 of NFS do not include procedures for opening or closing resources on a remote server. Before NFS tasks can be accomplished on these versions, the special *Mount* protocol must be employed to mount a file system and create a file handle to access a file on it. The protocol is also used to unmount the file system when no longer required. The Mount protocol is implemented in a manner similar to NFS itself, defining a sequence of procedures that use RPC and XDR. In NFSv4, the Mount protocol is no longer needed, because the tasks it performs have been implemented as NFSv4 operations.

The actual implementation of the Mount protocol is very similar to that of NFS itself. Like NFS, the Mount protocol uses XDR to define data types to be exchanged between the client and server and RPC to define a set of server procedures that clients may use to perform different operations. The main difference between Mount and NFS is simply that Mount defines procedures related to opening and closing file systems, rather than file access operations. Table 58-5 shows the server procedures used in the Mount protocol.

**Table 58-5:** NFS Mount Protocol Server Procedures

Procedure Number	Procedure Name	Procedure Summary	Description
0	null	Do nothing	Dummy procedure provided for testing purposes.
1	mnt	Add mount entry	Performs a mount operation by mapping a path on a server to a file handle for the client to use.
2	dump	Return mount entries	Returns a list of remotely mounted file systems.
3	umnt	Remove mount entry	Performs an unmount operation by removing a mount entry. (Yes, it should be <i>dismount</i> ; techies usually aren't English majors.)
4	umntall	Remove all mount entries	Removes all mount entries, thus eliminating all mounted file systems between server and client.
5	export	Return export list	Returns a list of exported file systems and indicates which clients are allowed to mount them. This is used to let the client see which served file systems are available for use.

Again, NFSv4 does away with the notion of a separate Mount protocol, incorporating file mounting operations into NFS directly.



# PART III-3

## **HOST CONFIGURATION AND TCP/IP HOST CONFIGURATION PROTOCOLS**

Each host that is placed on a network or internetwork must be set up and configured before it can be used. Configuration ensures that the host functions properly and that it is told the parameters needed for it to successfully communicate with other hosts and devices. In the good old days, administrators would manually set up each host as it was added to the network, and they would also manually make changes to the configuration as required.

Modern networks, however, are very large, and manual configuration of hosts is a time-consuming chore. Furthermore, we often need features that only automated configuration can provide, particularly for special hosts that have no internal storage. It is for these reasons that *host configuration* protocols were developed.

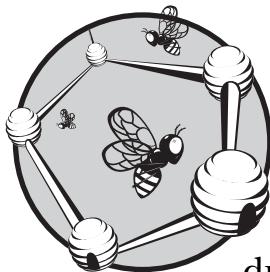
This part includes six chapters that describe the concepts behind host configuration protocols and then illustrate the operation of two of the most important ones in use today. The first chapter is an overview of host configuration concepts and issues, which will help you understand why these protocols are so important. In the second chapter, I describe the TCP/IP Bootstrap Protocol (BOOTP), the first truly capable automated configuration tool for Internet Protocol (IP) hosts.

The remaining chapters in this part cover BOOTP's successor, the feature-filled Dynamic Host Configuration Protocol (DHCP). The third chapter introduces DHCP and talks about the different ways DHCP can assign addresses, with a focus on dynamic addressing. The fourth chapter discusses how DHCP operates, including a look at configuration parameter management and the procedures for allocating addresses and managing those allocations. The fifth chapter describes DHCP messaging and illustrates the DHCP message format. The final chapter details DHCP clients and servers, looks at special features and issues with DHCP, and describes DHCP changes to support the new IP version 6 (IPv6).

Technically, the very first host configuration protocol for TCP/IP was the Reverse Address Resolution Protocol (RARP). RARP is a simple, crude protocol that allows very basic host configuration to be performed, but little else. RARP is very different from BOOTP and DHCP, not only because of its more limited capabilities, but because it operates between layers 2 and 3, like the Address Resolution Protocol (ARP) on which it is based. It is therefore covered in Part II-2, which also describes ARP.

# 59

## **HOST CONFIGURATION CONCEPTS, ISSUES, AND MOTIVATION**



Putting a host on an internetwork requires that certain setup and configuration procedures be followed. Hardware must be selected and set up, and software must be chosen and installed on the hardware. Once the software is set up, we aren't finished, however. We must also perform other configuration tasks that tell the software how we want it to operate and give it certain parameters, so it knows its role on the network and how to function.

In this brief chapter, I discuss the purpose of host configuration, the problems associated with it, and host configuration protocols.

### **The Purpose of Host Configuration**

Probably the most important configuration task that must be performed for each host on an internetwork is to give it an *identity*, in the form of an address that is unique to it alone. In TCP/IP networks, each device must be given an

IP address. Hosts also often require other parameters to ensure that they operate properly. For a TCP/IP network, we might want to tell each host some of the following:

- The address of a default router on the local network
- The network mask the host should use
- The addresses of servers providing particular services to the host, such as a mail server or a Domain Name System (DNS) name server
- The maximum transmission unit (MTU) of the local network (see Chapter 22)
- What Time to Live (TTL) value to use for IP datagrams (see Chapter 21)

There may be a lot more information that must be relayed to the host. Dozens of different parameters must be set up for certain networks. Many of these may be common to all the machines on a network, but IP addresses must be unique. The administrator must therefore ensure that each IP address is assigned to only one computer, even as machines are added to and removed from the network.

## The Problems with Manual Host Configuration

If you're an administrator in charge of a small local area network (LAN) with ten hosts, performing setup and configuration is simple. For each host, you set up the hardware, install the software, and then configure the software. Even making changes and keeping track of IP addresses wouldn't be a big deal; a single sheet of paper would suffice. However, what happens when your network has a hundred computers, or a thousand computers, or even ten thousand?

As the size of the network grows, the work needed for manual configuration grows with it. And while initial hardware setup may be time-consuming, at least it is done mainly when the host is first set up, and rarely changed thereafter. This is not the case with configuration parameters. If the address of the local router changes on a network with a thousand hosts, do you really want to go to each host to edit a configuration file?

The drudge work associated with manual configuration is significant, but the problems go well beyond the inefficiency issue. There are situations where manual configuration is not just inconvenient, but is actually impossible:

**Remote Configuration** An administrator cannot be everywhere; modern networks can span cities or nations. Unless we want to train every user on how to configure network hosts, we must use an automated protocol.

**Mobile Device Configuration** IP was designed when computers were large and attached to each other using heavy cables; today, we have computers that fit in a shirt pocket and communicate using radio waves. IP addresses must be assigned based on the network to which they are attached, and this makes reconfiguration required when a device is moved. This is not conducive to manual configuration at all.

**Dumb Host Configuration** Most of the hosts we use today are full-fledged computers, with their own internal storage. We can assign such a device an address by entering it into a file that the device reads when it starts up. There are certain devices, however, that do not include any form of storage. Since they are mass-produced, they

are all identical and cannot have individualized parameters stored within them. Such a device relies on a configuration protocol to learn what it needs to function on a network—especially including its individual identity.

**Address Sharing** The proliferation of devices attached to the global Internet has led to a situation where IP addresses must be carefully managed to ensure that they are not wasted on devices that aren't using them. Some organizations even find themselves with more potential hosts than they have addresses. A host configuration protocol can allow an address to be automatically assigned to a host when needed, and then have that address returned to a common pool for reuse when the host leaves the network. This permits addresses to be shared and reduces the need for more address space.

## Automating the Process: Host Configuration Protocols

Even though most of us don't have robots that can automate the hardware portions of the setup and configuration job, we can employ tools that will make the rest of the job easier. This includes the use of special *host configuration* protocols. These protocols allow hosts to be automatically configured when they are set up and to have additional parameters assigned when needed.

Host configuration protocols generally function by having a host send a request for an address and other parameters, which is satisfied by a response from a server. The information in the response is used by the client to set its address, identify a local router, and perform other necessary setup so it can communicate.

The use of an automated protocol solves all of the problems associated with manual configuration. We can configure devices remotely, rather than needing to walk to each one. We can instantly assign a valid address to mobile devices. We can have dumb hosts boot up and obtain the information they need to operate. Finally, we can maintain a pool of addresses that is shared by a group of hosts.

**KEY CONCEPT** *Host configuration protocols* enable administrators to set up hosts so that they can automatically determine their address and other key parameters. They are useful not only because of the effort they save over manual configuration, but because they enable the automatic setup of remote, storageless, or mobile devices.

## The Role of Host Configuration Protocols in TCP/IP

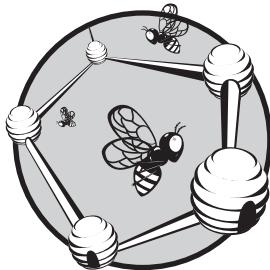
You might find it strange that host configuration protocols would exist in the lofty heights of the application layer. It certainly sounds like host configuration is a function related more to the network layer, where internetwork addresses such as IP addresses function. In fact, some host configuration protocols, like the rudimentary Reverse Address Resolution Protocol (RARP, discussed in Chapter 14), do exist down at that level.

However, there are advantages to having host configuration protocols reside at higher levels. A major one is that the operation of the protocol does not depend on the hardware on which it runs, making it more universal. Another is being able to convey host configuration messages between networks, which is not possible with a low-level protocol operating on the local network.



# 60

## **TCP/IP BOOTSTRAP PROTOCOL (BOOTP)**



Before a device on a TCP/IP network can effectively communicate, it needs to know its IP address. While a conventional network host can read this information from its internal disk, some devices have no storage, so they do not have this luxury. They need help from another device on the network to provide them with an IP address and the other information and/or software they need to become active Internet Protocol (IP) hosts. This problem of getting a new machine up and running is commonly called *bootstrapping*, and to provide this capability to IP hosts, the *TCP/IP Bootstrap Protocol (BOOTP)* was created.

In this chapter, I provide a detailed look at BOOTP. I begin with an overview and history of the protocol and a look at the standards that define it. I then discuss the general client/server nature of BOOTP and how addressing is done in communication between the client and the server. I describe the operation of BOOTP step by step and illustrate the format of BOOTP messages. I conclude with a description of BOOTP vendor extensions, which

are used to allow the information sent in BOOTP messages to be customized, and a discussion of BOOTP relay agents, which allow the protocol to operate even when the BOOTP server and client are on different networks.

**RELATED INFORMATION** *BOOTP was the predecessor of the Dynamic Host Configuration Protocol (DHCP). DHCP was built to be substantially compatible with BOOTP, and so the two protocols have a fair degree of commonality. To avoid duplication, certain information has been included only in the following chapters about DHCP (with references to this chapter where appropriate). On the other hand, some of the historical background information behind features like vendor information extensions and relay agents, which were first developed for BOOTP and adopted by DHCP, is in this chapter and referenced from the DHCP chapters. If you plan to read about DHCP as well as BOOTP, I recommend reading this section first. If you don't plan to read up on DHCP, you may wish to check the discussion of DHCP/BOOTP interoperability in Chapter 64.*

## **BOOTP Overview, History, and Standards**

The TCP/IP protocol suite has been with us for over two decades, and the problem of how to automate the configuration of parameters on IP hosts has been around almost as long. Back in the early 1980s, networks were small and relatively simple, so manual configuration wasn't that difficult. Automated host configuration was primarily needed because it was the only way to configure devices like diskless workstations.

As I discussed in Chapter 59, without a form of internal storage, a device must rely on someone or something to tell it "who it is" (its address) and how to function each time it is powered up. When a device like this is turned on, it is in a difficult position: It needs to use IP to communicate with another device that will tell it how to communicate using IP! This process, called *bootstrapping* or *booting*, comes from an analogy to a person "pulling himself up using his own bootstraps." You've likely encountered this term before, if at no other time then when some tech support person has told you to "reboot" your computer.

### ***BOOTP: Correcting the Weaknesses of RARP***

The Reverse Address Resolution Protocol (RARP) was the first attempt to resolve this bootstrap problem. Created in 1984, RARP is a direct adaptation of the low-level Address Resolution Protocol (ARP) that binds IP addresses to link-layer hardware addresses (see Chapter 13). RARP is capable of providing a diskless device with its IP address, using a simple client/server exchange of a request and reply between a host and an RARP server.

The difficulty with RARP is that it has so many limitations. It operates at a fairly low level using hardware broadcasts, so it requires adjustments for different hardware types. An RARP server is also required on every physical network to respond to layer 2 broadcasts. Each RARP server must have address assignments manually provided by an administrator. And perhaps worst of all, RARP provides only an IP address to a host and none of the other information a host may need. (I describe these issues in detail in Chapter 14.)

RARP clearly wasn't sufficient for the host configuration needs of TCP/IP. To support both diskless hosts and other situations where the benefits of autoconfiguration were required, BOOTP was created. BOOTP was standardized in RFC 951,

published in September 1985. This relatively straightforward protocol was designed specifically to address the shortcomings of RARP:

- BOOTP is still based on a client/server exchange, but is implemented as a higher-layer software protocol, using the User Datagram Protocol (UDP) for message transport (see Chapter 44). It is not dependent on the particular hardware of the network as RARP is.
- It supports sending additional configuration information to a client beyond just an IP address. This extra information can usually be sent in one message for efficiency.
- It can handle having the client and server on different networks of an internet-work. This allows the administration of the server providing IP addresses to be more centralized, saving money as well as administrative time and hassle.

It should be noted that, even though the name of BOOTP implies that it defines everything needed for a storageless device to boot, this isn't really the case. As the BOOTP standard itself describes, bootstrapping generally requires two phases. In the first, the client is provided with an address and other parameters. In the second, the client downloads software, such as an operating system and drivers, that let it function on the network and perform other tasks. BOOTP really deals with only the first of these phases: address assignment and configuration. The second is assumed to take place using a simple file transfer protocol like the Trivial File Transfer Protocol (TFTP, discussed in Chapter 73).

**KEY CONCEPT** The first widely used host configuration protocol for TCP/IP was the *Boot Protocol (BOOTP)*. It was created specifically to enable host configuration while addressing many of the weaknesses of RARP. BOOTP is intended to be used as the first phase of a two-phase boot procedure for storageless devices. After obtaining an IP address and other configuration parameters using BOOTP, the device employs a protocol such as TFTP to download software necessary to function on the network.

### **Vendor-Specific Parameters**

One smart decision made when BOOTP was created was the inclusion of a *vendor-specific area*. This was intended to provide a place where hardware vendors could define parameters relevant to their own products. As the complexity of TCP/IP increased, it was realized that this field could be used to define a method of communicating certain parameters that were commonly needed by IP hosts, and were in fact vendor-independent. This was first proposed in RFC 1048, “BOOTP Vendor Information Extensions,” published in February 1988.

The fact that BOOTP can be used to provide information to a client beyond just an IP address makes it useful even in cases where a device already knows its address. BOOTP can be used to send parameters that the administrator wants all hosts to have, to ensure that they use the network in a consistent manner. Also, in the case of devices that do have local storage (and therefore do not need BOOTP to get an IP address), BOOTP can still be used to let these devices get the name of a boot file for phase two of bootstrapping, in which the client downloads software.

## **Changes to BOOTP and the Development of DHCP**

BOOTP was the TCP/IP host configuration protocol of choice from the mid-1980s through the end of the 1990s. The vendor extensions introduced in RFC 1048 were popular, and over the years, additional vendor extensions were defined. RFC 1048 was replaced by RFCs 1084, 1395, and 1497 in succession.

Some confusion also resulted over the years in how some sections of RFC 951 should be interpreted and how certain features of BOOTP work. RFC 1542, “Clarifications and Extensions for the Bootstrap Protocol,” was published in October 1993 to address this and also to make some slight changes to the protocol’s operation. (RFC 1542 is actually a correction of the nearly identical RFC 1532, which had some small errors.)

While BOOTP was obviously quite successful, it also had certain weaknesses. One of the most important of these was the lack of support for *dynamic* address assignment. The need for dynamic assignment became much more pronounced when the Internet really started to take off in the late 1990s. This led directly to the development of the Dynamic Host Configuration Protocol (DHCP).

While DHCP replaced BOOTP as the TCP/IP host configuration protocol of choice, it would be inaccurate to say that BOOTP is gone. It is still used to this day in some networks. Furthermore, DHCP was based directly on BOOTP, and they share many attributes, including a common message format. BOOTP vendor extensions were used as the basis for DHCP *options*, which work in the same way but include extra capabilities. In fact, the successor to RFC 1497 is RFC 1533, which officially merges BOOTP vendor extensions and DHCP options into the same standard.

## **BOOTP Client/Server Messaging and Addressing**

While BOOTP can be used for a variety of devices, one of the primary motives behind its creation was to provide a way to automatically configure “dumb” devices that have no storage. Most of these devices are relatively limited in their capabilities, so requiring them to support a fancy boot protocol would not make sense. BOOTP is thus an uncomplicated protocol, which accomplishes host configuration without a lot of complicated concepts or implementation requirements.

Like so many other TCP/IP protocols, BOOTP is client/server in nature. The operation of the protocol consists of a single exchange of messages between a *BOOTP client* and a *BOOTP server*. A BOOTP client can be any type of device that needs to be configured. A BOOTP server is a network device that has been specially set up to respond to BOOTP client requests, and has been programmed with addressing and other information it can provide to clients when required.

The BOOTP server maintains a special set of information about the clients it serves. One key part of this is a table that maps the hardware (layer 2, the data link layer) addresses of each client to an assigned IP address for that device. The client specifies its hardware address in its request, and the server uses that address to look up the client’s IP address and return it to the client. (Other techniques can also be used, but a mapping table is most common.)

## ***BOOTP Messaging and Transport***

BOOTP messaging uses UDP as its layer 4 transport protocol, for a couple of reasons:

- UDP is a lot less complex than the other layer 4 transport protocol, the Transmission Control Protocol (TCP), and is ideal for simple request/reply protocols like BOOTP.
- Since the client obviously doesn't know the address of a BOOTP server, the request is broadcast on its local network. UDP supports broadcasts; TCP does not.

UDP uses a special well-known (reserved) port number for BOOTP servers: UDP port 67. BOOTP servers listen on port 67 for these broadcast BOOTP requests sent by clients. After processing the request, the server sends a reply back to the client. How this is handled depends on whether or not the client knows its own address.

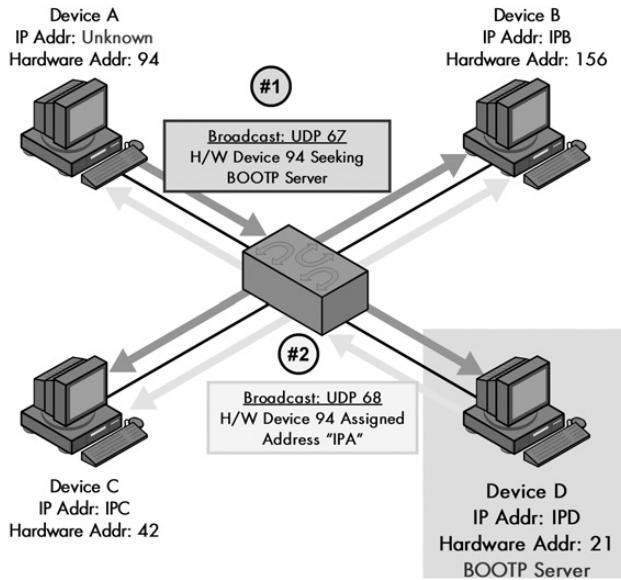
BOOTP is often used to provide an IP address to a client that doesn't know its address. This is sometimes called a chicken-and-egg problem, because it represents a loop of sorts like the old conundrum of which came first, the chicken or the egg? To resolve this dilemma, the BOOTP server has two choices. If the operating system supports it, the server can use the client's hardware address to create an ARP entry for the device, and then use a layer 2 unicast to deliver the reply. Otherwise, it must send the reply as a broadcast as well on the local network.

However, in the case where the BOOTP client already knows its own address, that address can be used by the BOOTP server to send back its reply directly.

## ***BOOTP Use of Broadcasts and Ports***

The fact that BOOTP servers may need to broadcast back to the client necessitates a bit of a change from the way most TCP/IP protocols use client ports. Recall that normally, the client in a client/server transaction using UDP or TCP generates a temporary, or ephemeral, port number that it uses as the source port in its request. The server sends the reply back to the client's IP address using that ephemeral port number.

Ephemeral port numbers must be unique for a particular IP address, but may not necessarily be unique across all the devices on a network. For example, Device A may be using ephemeral port number 1248 for an HTTP request to a web server, while Device B may be using port number 1248 on its TCP/IP stack to send a Domain Name System (DNS) request. Since the server in BOOTP is broadcasting, it is not targeting a particular device with a unicast transmission. This means it cannot safely send to an ephemeral port number. This is because some other device on the network may have selected the same ephemeral port number for some other transaction and may mistake the BOOTP server's response as being intended for itself. To avoid this problem, another well-known port number is used just for BOOTP clients: UDP port 68. Clients listen on this port for broadcast or unicast transmissions; devices that have not sent a BOOTP request will ignore it. This dual-broadcast BOOTP communication process is illustrated in Figure 60-1.



**Figure 60-1:** General operation of BOOTP

**KEY CONCEPT** BOOTP is a relatively simple client/server protocol that relies on broadcasts to permit communication with devices that do not have an assigned IP address. In this example, Device A is trying to determine its IP address and other parameters. It broadcasts a BOOTP request on the local network using UDP port 67 and then listens for a reply on port 68. Device D is configured as a BOOTP server and listens on this port. When it receives the request, it sends a broadcast on port 68 telling Device A what its IP address is. A BOOTP client uses broadcasts to send its requests to any listening BOOTP server. In most cases, the BOOTP client device does not know its own IP address when it uses the protocol. For this reason, a BOOTP server will also typically use broadcast in sending its reply, to be sure it reaches the client.

### ***Retransmission of Lost Messages***

The drawback of the simplicity of using UDP for BOOTP messaging is that UDP is unreliable, which means a BOOTP request might be lost before it gets to the server, or the server's response may not get back to the client. Like many other protocols using UDP, BOOTP clients take care of this by using a retransmission timer. If after a certain period of time the client has not received a response, it resends its request.

However, BOOTP clients must take care in how they implement their retransmission strategy. Consider a scenario where a network with 200 BOOTP clients loses power. These machines are all pretty much the same, so when the power comes back on, they all restart and try to send BOOTP requests at about the same time. Most likely, problems will occur due to all these requests: Some will be lost, or the server may drop some due to overload. If all the clients use the same amount of time for retransmission, then after that time elapses, a whole bunch of machines will again send requests and re-create the original problem.

To avoid retransmission problems, the BOOTP standard recommends using an exponential backoff scheme for retransmissions, starting with a retransmission

interval of 4 seconds and doubling it for successive tries. A randomness element is also added to prevent many devices from overlapping their retransmissions. The idea is very similar to the backoff method used by Ethernet (in fact, the standard even refers to the Ethernet specification). For example, the first retransmission would occur after a random period of time between 0 and 4 seconds (plus or minus a random amount); a second retransmission, if needed, after a random time interval between 0 and 8 seconds, plus or minus, and so forth. This helps reduce the chances of retransmissions being lost and also helps ensure BOOTP traffic doesn't bog down the network.

**KEY CONCEPT** BOOTP uses UDP for transport, which provides no reliability features. For this reason, the BOOTP client must detect when its requests are lost and, if necessary, retransmit them.

## BOOTP Detailed Operation

Now that you have seen how BOOTP messaging works in general terms, let's take a closer look at the detailed operation of the protocol. This will clarify how clients and servers create and process messages, and also help make sense of some of the important fields in the BOOTP message field format. Understanding the basic operation of BOOTP will also be of use when we examine BOOTP relay agents later in this chapter, and even when we discuss DHCP in the following chapters.

### ***BOOTP Bootstrapping Procedure***

The following are the basic steps performed by the client and server in a regular BOOTP bootstrapping procedure (see Figure 60-2).

**Client Creates Request** The client machine begins the procedure by creating a BOOTP request message. In creating this message, it fills in the following information:

- It sets the message type (Op) to the value 1, for a BOOTREQUEST message.
- If it knows its own IP address that it plans to keep using, it specifies it in the CIAddr (Client IP Address) field. Otherwise, it fills this field with zeros. (The CIAddr field is discussed in more detail in the next section.)
- It puts its own layer 2 hardware address in the CHAddr field. This is used by the server to determine the right address and other parameters for the client.
- It generates a random transaction identifier and puts this in the XID field.
- The client may specify a particular server that it wants to send it a reply and put that in the SName field. It may also specify the name of a particular type of boot file that it wants the server to provide in the File field.
- The client may specify vendor-specific information, if programmed to do so.

**Client Sends Request** The client broadcasts the BOOTREQUEST message by transmitting it to address 255.255.255.255. Alternatively, if it already knows the address of a BOOTP server, it may send the request unicast.

**Server Receives Request and Processes It** A BOOTP server, listening on UDP port 67, receives the broadcasted request and processes it. If a name of a particular server was specified and this name is different from the name of this server, the server may discard the request. This is especially true if the server knows that the server the client asked for is also on the local network. If no particular server is specified, or this particular server was the one the client wanted, the server will reply.

**Server Creates Reply** The server creates a reply message by copying the request message and changing several fields:

- It changes the message type (Op) to the value 2, for a BOOTREPLY message.
- It takes the client's specified hardware address from the CHAddr field and uses it in a table lookup to find the matching IP address for this host. It then places this value into the YIAddr (Your IP Address) of the reply.
- It processes the File field and provides the filename type the client requested, or if the field was blank, the default filename.
- It puts its own IP address and name in the SIAddr and SName fields.
- It sets any vendor-specific values in the Vend field.

**Server Sends Reply** The server sends the reply. The method it uses depends on the contents of the request:

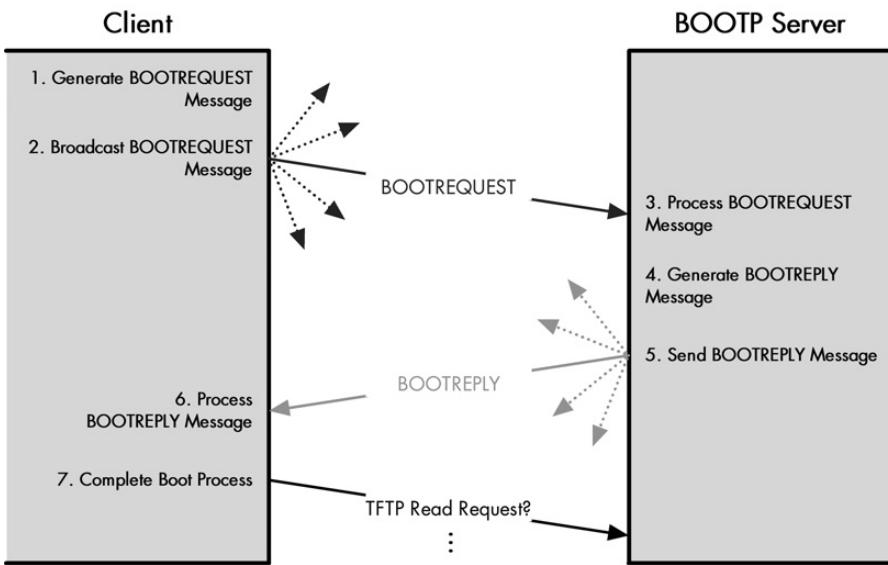
- If the B (Broadcast) flag is set, this indicates that the client can't have the reply sent unicast, so the server will broadcast it.
- If the CIAddr field is nonzero, the server will send the reply unicast back to that CIAddr.
- If the B flag is zero and the CIAddr field is also zero, the server may either use an ARP entry or broadcast, as described earlier.

**Client Processes Reply** The client receives the server's reply and processes it, storing the information and parameters provided. (See the next section for one important issue related to this processing.)

**Client Completes Boot Process** Once configured, the client proceeds to phase two of the bootstrapping process, by using a protocol such as TFTP to download its boot file containing operating system software, using the filename the server provided.

### ***Interpretation of the Client IP Address (CIAddr) Field***

A complication can arise when a client chooses to specify an IP address in the CIAddr field in its request. The problem is how exactly to interpret this field. Does it mean that the client is already using this IP address? Or is it just the one it used last time it was booted? Then there is the related problem of what to do if the server supplies an address in the YIAddr that is different from the one the client is using. Should the server's provided address override the client's address? Or should the client ignore it? Who makes the decision, the server or the client?



**Figure 60-2: BOOTP operation** BOOTP uses a simple two-step message exchange consisting of a broadcast request and broadcast reply. After the client receives configuration information from the BOOTP server, it completes the bootstrapping process using a protocol such as TFTP.

Much confusion occurred due to the vagueness of the original standard in this regard, and this led to nonuniformity in how different implementations chose to handle this issue. There were even some implementations that used the CIAddr to mean “the client requests this IP address,” which was never part of BOOTP functionality. This is an especially bad idea since it could lead to BOOTP replies never reaching the client.

RFC 1542 was written in part to try to clean up this mess. It suggests that the following is the best way to handle the meaning of these fields:

- If a client is willing to accept whatever IP address the server provides, it sets CIAddr to all zeros, even if it knows a previous address.
- If the client fills in a value for the field, it is saying it will use this address, and it must be prepared to receive unicast messages sent to that address.
- If the client specifies an address in CIAddr and receives a different address in the YIAddr field, the server-provided address is ignored.

Note that not all hardware devices may necessarily agree with this interpretation as provided by RFC 1542, so there are still potential interoperability concerns here with older equipment. Then again, RFC 1542 was written in 1993, so this is probably no longer much of an issue!

## BOOTP Message Format

The exchange of information in BOOTP takes the form of a request sent by a client and a reply sent back by the server. BOOTP, like a number of other request/reply protocols, uses a common message format for requests and replies. The client starts

by setting aside memory space for the message and clearing it to all zeros. It then fills in the fields of the message and sends the request, as you saw in the previous section. The server creates its reply not from scratch, but by copying the request and changing certain fields.

BOOTP messages contain a considerable number of fields, so the message format is rather large. It is described in Tables 60-1 and 60-2, and illustrated in Figure 60-3.

**Table 60-1:** BOOTP Message Format

Field Name	Size (Bytes)	Description
Op	1	Operation Code: Specifies the type of message. A value of 1 indicates a request (BOOTREQUEST message). A value of 2 is a reply (BOOTREPLY message).
HType	1	Hardware Type: This field specifies the type of hardware used for the local network and is used in exactly the same way as the equivalent field (HRD) in the ARP message format (see Chapter 13). Some of the most common values for this field are shown in Table 60-2.
HLen	1	Hardware Address Length: Specifies how long hardware addresses are in this message. For Ethernet or other networks using IEEE 802 MAC addresses, the value is 6. This is the same as the field with a similar name (HLN) in the ARP field format.
Hops	1	Hops: Set to 0 by a client before transmitting a request and used by BOOTP relay agents to control the forwarding of BOOTP messages.
XID	4	Transaction Identifier: A 32-bit identification field generated by the client, to allow it to match up the request with replies received from BOOTP servers.
Secs	2	Seconds: According to RFC 951, the client enters into this field the number of seconds “elapsed since [the] client started trying to boot.” This is supposed to provide information to BOOTP servers to help them decide which requests to respond to first. Unfortunately, it isn’t clear if this meant the amount of time since the machine was powered on or since the first BOOTREQUEST message was sent. In addition, some devices incorrectly implemented this field. As a result, it is not always used.
Flags	2	Flags: In the original BOOTP standard (RFC 951), this was an empty 2-byte field. RFC 1542 changed this to a Flags field, which at present contains only one flag. It has a B (Broadcast) flag subfield, 1 bit in size, which is set to 1 if the client doesn’t know its own IP address at the time it sends its BOOTP request. This serves as an immediate indicator to the BOOTP server or relay agent that receives the request that it definitely should send its reply by broadcast. The other subfield is Reserved, which is 15 bits, set to 0, and not used.
CIAddr	4	Client IP Address: If the client has a current IP address that it plans to keep using, it puts it in this field. By filling in this field, the client is committing to responding to unicast IP datagrams sent to this address. Otherwise, it sets this field to all Os to tell the server it wants an address assigned. (See the previous section in this chapter for important information about this field.)
YIAddr	4	Your IP Address: The IP address that the server is assigning to the client. This may be different than the IP address currently used by the client.
SIAddr	4	Server IP Address: The IP address of the BOOTP server sending a BOOTREPLY message.
GIAddr	4	Gateway IP Address: Used to route BOOTP messages when BOOTP relay agents facilitate the communication of BOOTP requests and replies between a client and a server on different subnets or networks. To understand the name, remember that the old TCP/IP term for router is <i>gateway</i> ; BOOTP relay agents are typically routers. Note that this field is set to 0 by the client and should be ignored by the client when processing a BOOTREPLY. It specifically does not represent the server giving the client the address of a default router address to be used for general IP routing purposes.

(continued)

**Table 60-1:** BOOTP Message Format (continued)

Field Name	Size (Bytes)	Description
CHAddr	16	Client Hardware Address: The hardware (layer 2) address of the client sending a BOOTREQUEST. It is used to look up a device's assigned IP address and also possibly in delivery of a reply message.
SName	64	Server Name: The server sending a BOOTREPLY may optionally put its name in this field. This can be a simple text nickname or a fully qualified DNS domain name (such as myserver.organization.org). Note that a client may specify a name in this field when it creates its request. If it does so, it is saying that it wants to get a reply only from the BOOTP server with this name. This may be done to ensure that the client is able to access a particular boot file stored on only one server.
File	128	Boot Filename: Contains the full directory path and filename of a boot file that can be downloaded by the client to complete its bootstrapping process. The client may request a particular type of file by entering a text description here, or it may leave the field blank and the server will supply the filename of the default file.
Vend	64	Vendor-Specific Area: Originally created to allow vendors to customize BOOTP to the needs of different types of hardware, this field is now also used to hold additional vendor-independent configuration information. The next section, on BOOTP vendor information extensions, contains much more detail on this field. It may be used by the client and/or the server.

**Table 60-2:** BOOTP Message HTYPE Values

HTYPE Value	Hardware Type
1	Ethernet (10 Mb)
6	IEEE 802 Networks
7	ARCNet
15	Frame Relay
16	Asynchronous Transfer Mode (ATM)
17	High-Level Data Link Control (HDLC)
18	Fibre Channel
19	ATM
20	Serial Line

As I mentioned earlier in this chapter, both requests and replies are encapsulated into UDP messages for transmission. The BOOTP standard specifies that the use of UDP checksums is optional. Using the checksum provides protection against data-integrity errors and is thus recommended. This may cause unacceptable processing demands on the part of very simple clients, so the checksum can legally be skipped.

Similarly, for simplicity, BOOTP assumes that its messages will not be fragmented. This is to allow BOOTP clients to avoid the complexity of reassembling fragmented messages. Since BOOTP messages are only 300 bytes in length, under the maximum transmission unit (MTU) required for all TCP/IP links, this is not normally an issue.

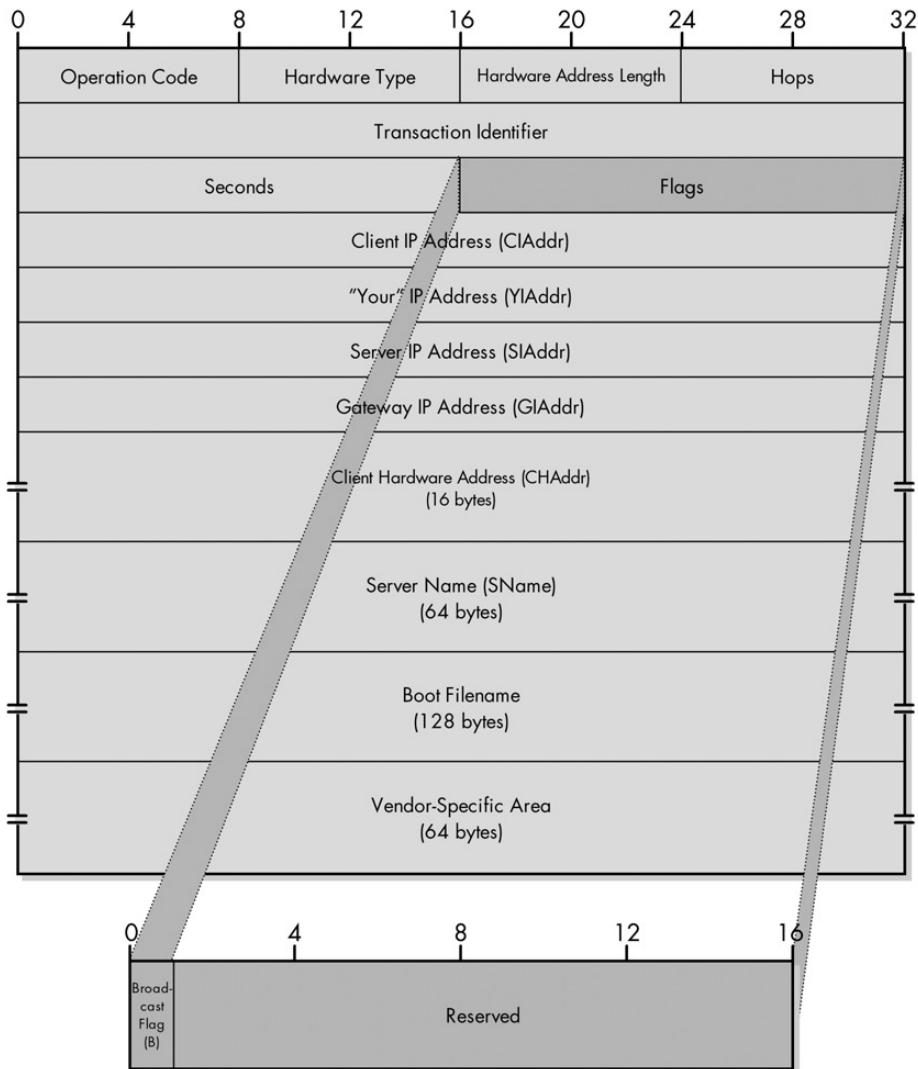


Figure 60-3: BOOTP message format

## BOOTP Vendor-Specific Area and Vendor Information Extensions

The creators of BOOTP realized that certain types of hardware might require additional information to be passed from the server to the client in order for the client to boot. For this reason, they put into the BOOTP field format the 64-byte Vend field, also called the Vendor-Specific Area. Including this field makes BOOTP flexible, since it allows vendors to decide for themselves how they want to use the protocol and to tailor it to their needs.

A client can use the Vend field by asking for certain types of information in the field when composing its request. The server can then respond to these requests, and it may also include parameters it wants the client to have, even if they were not requested. The original BOOTP protocol does not define any structure for the Vendor-Specific Area, leaving this up to each manufacturer to decide.

Obviously, there is nothing preventing a client made by one manufacturer from trying to send a request to a server made by another one. If each one is expecting the Vend field to contain something different, the results will be less than satisfactory. Thus, for the Vend field to be used properly, both devices must be speaking the same language when it comes to the meaning of this field. This is done by setting the first four bytes of the field to a special value. Each manufacturer chooses its own *magic number*, sometimes called a *magic cookie*, for this four-byte subfield.

**NOTE** Why is it called a *magic cookie*? I'm not sure, to be honest. I have heard that its origin may be the cookie that Alice ate to grow or shrink in the story Alice in Wonderland.

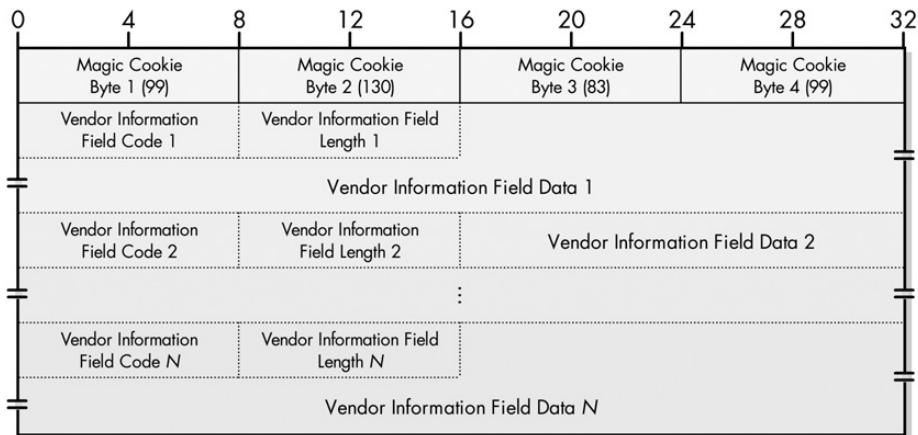
### **BOOTP Vendor Information Extensions**

Including the Vend field in BOOTP gives the protocol extensibility for vendor-specific information. Unfortunately, the original field format didn't include any way of extending the information sent from a server to a client for *generic*, nonvendor-specific TCP/IP information. This was a significant oversight in the creation of the protocol, because there are many types of information that a TCP/IP host needs when it starts up that really have nothing to do with its vendor. For example, when a host boots, we probably want it to be told the address of a default router, the subnet mask for its local subnet, the address of a local DNS server, the MTU of the local network, and much more. None of these things are vendor-specific, but there is no place to put them in the BOOTP reply message.

Since there was no nonvendor-specific area field in BOOTP, the decision was made to define a way of using the Vendor-Specific Area (Vend field) for communicating this additional generic information. This was first standardized in RFC 1048 and then refined in later RFCs, as I explained in the BOOTP overview earlier in this chapter. This scheme basically represents one particular way of using the Vend field that most TCP/IP BOOTP implementations have chosen to adopt, regardless of their vendor. This enhancement is formally referred to as *BOOTP vendor information extensions*.

**KEY CONCEPT** The BOOTP message format includes a Vend field that was originally intended for vendor-specific customized fields. It was later changed to a place where additional generic information could be sent from a BOOTP server to a BOOTP client. Each such parameter is carried in a *BOOTP vendor information field*.

To clearly mark that this particular meaning of the Vend field is being used, a special, universal magic cookie value of 99.130.83.99 is inserted into the first four bytes of the field. Then the remaining 60 bytes can contain a sequence of one or more *vendor information fields*. The overall structure of the Vendor-Specific Area when vendor information extensions are used is shown in Figure 60-4.



**Figure 60-4:** BOOTP Vendor-Specific Area format showing vendor information fields

**NOTE** The BOOTP Vendor-Specific Area begins with the four-byte magic cookie and then contains a number of variable-length vendor information fields, each of which has the format shown above and in Table 60-3. Despite the use of IP dotted-decimal notation to represent the value 99.130.83.99, this is not an IP address. It's just a marker—a magic number that is universally recognized.

### BOOTP Vendor Information Fields

Each vendor information field specifies a particular type of information to be communicated, and it is encoded using a special subfield structure that specifies the field's type, length, and value. This is a common method of specifying options, called *TLV-encoding* (for *type, length, value*). The same basic method is used for encoding Internet Protocol versions 4 and 6 (IPv4 and IPv6) options. Table 60-3 shows the structure and the common names for the subfields of each vendor information field.

**Table 60-3:** BOOTP Vendor Information Field Format

Subfield Name	Size (Bytes)	Description
Code	1	Vendor Information Field Code: A single octet that specifies the vendor information field type.
Len	1	Vendor Information Field Length: The number of bytes in this particular vendor information field. This does not include the two bytes for the Code and Len fields.
Data	Variable	Vendor Information Field Data: The data being sent, which has a length indicated by the Len subfield, and which is interpreted based on the Code subfield.

There are two special cases that violate the field format shown in Table 60-3. A Code value of 0 is used as padding when subfields need to be aligned on word boundaries; it contains no information. The value 255 is used to mark the end of the vendor information fields. Both of these codes contain no actual data. To save space, when either is used, just the single Code value is included, and the Len and

Data fields are omitted. A device seeing a Code value of 0 just skips it as filler; a device seeing a Code value of 255 knows it has reached the end of the vendor information fields in this Vend field.

The vendor information extensions of BOOTP have become so popular that the use of this field for sending extra generic information is pretty much standard. In fact, I am not sure if anyone today still uses the Vend field solely for vendor-specific information.

When the vendor information extensions were introduced, one was created that points to a file where vendor-specific information can be found. This lets devices have the best of both worlds—they can use the standard vendor-independent fields and can incorporate vendor-specific fields (through the referenced file) where needed. Later, another field type was created that lets vendor-specific fields be mixed with vendor-independent ones directly in a BOOTP message.

When DHCP was created, the same vendor extension mechanism was maintained and enhanced further, but instead of the field being called vendor information extensions, it was renamed to *Options*. (A much better name!) The BOOTP vendor information fields were retained in DHCP, and new DHCP-specific options were defined. To avoid duplication, I have listed all the BOOTP vendor information fields and DHCP options in a set of tables in Chapter 63, which covers DHCP messaging. This includes a discussion of how vendor-specific and vendor-independent information can be mixed. You may also want to read the section in Chapter 63 that describes DHCP options, which discusses how they were created from BOOTP vendor information extensions.

## BOOTP Relay Agents (Forwarding Agents)

One reason why RARP was quickly replaced by BOOTP is that RARP required the client being configured and the server providing it with an IP address to be on the same physical network. This is fine when you run a small organization with ten machines, which are probably all on the same physical network. Larger networks must be divided into multiple physical networks for efficiency, however. RARP would require a separate RARP server for each network, meaning needing to duplicate all the functions of a single server onto multiple machines. Worse yet, all the configuration information would also be duplicated, and any changes would need to be made to all the different servers each time.

BOOTP is designed to allow the BOOTP server and the clients it serves to be on different networks. This centralizes the BOOTP server and greatly reduces the amount of work required of network administrators. However, implementing this feature means increasing the complexity of the protocol. In particular, we need to involve a third-party device in the configuration process.

You might rightly wonder why this would be the case. RARP is a low-level protocol that works at the link layer, so that explains why it would have problems putting the client and server on different physical networks. But wasn't the whole point of making BOOTP a high-level protocol that it was able to use IP? And if BOOTP uses IP, can't we send from one network to another arbitrarily, just like any IP-based messaging protocol?

The answer is that even though we are indeed using IP and UDP, BOOTP still has one of the same issues that RARP had: a reliance on *broadcasts*. The client usually doesn't know the address of a server, so it must send out its request as a broadcast, saying in essence, "Can anyone hear this and give me the information I need?" For efficiency reasons, routers do not route such broadcasts, as they would clog the network. This means that if the server and client are not on the same network, the server can't hear the client's broadcast. Similarly, if the server ever did get the request and broadcast its reply back to the client, the client would never get it anyway. To make this all work, we need something to act as an intermediary between the client and the server: a *BOOTP relay agent*.

### ***The Function of BOOTP Relay Agents***

The job of a BOOTP relay agent is to sit on a physical network where BOOTP clients may be located and act as a proxy for the BOOTP server. The agent gets its name because it relays messages between the client and server, and thus enables them to be on different networks.

**NOTE** *BOOTP relay agents were originally called forwarding agents. RFC 1542 changed the name to make explicit the fact that BOOTP relaying was not the same as conventional IP datagram forwarding by regular routers.*

In practice, a BOOTP relay agent is not usually a separate piece of hardware. Rather, it's a software module that runs on an existing piece of hardware that performs other functions. It is common for BOOTP relay agent functionality to be implemented on an IP router. In that case, the router is acting both as a regular router and also playing the role of a BOOTP agent. The forwarding functions required of a BOOTP relay agent are distinct from the normal IP datagram forwarding tasks of a router.

**KEY CONCEPT** Since BOOTP uses broadcasts, the BOOTP client and BOOTP server must be on the same physical network to be able to hear each other's broadcasted transmissions. For a client and server on different networks to communicate, a third party is required to facilitate the transaction: a *BOOTP relay agent*. This device, which is often a router, listens for transmissions from BOOTP clients and relays them to the BOOTP server. The server responds back to the agent, which then sends the server's response back to the client.

Naturally, the placement of the client and server on different networks and the presence of a relay agent change the normal request/reply process of BOOTP significantly. A couple of specific fields in the BOOTP message format are used to control the process. RFC 951 was rather vague in describing how this process works, so RFC 1542 described it in much more detail.

## **Normal BOOTP Operation Using a Relay Agent**

The following shows, in simplified form, a revised set of BOOTP operation steps when a relay agent is involved. To keep the size of this discussion manageable, I have omitted the details of the basic request/reply process to focus on the relaying functionality, which you can also see graphically in Figure 60-5.

**Client Creates Request** The client machine creates its request normally. The existence of a relay agent is totally transparent to the client.

**Client Sends Request** The client broadcasts the BOOTREQUEST message by transmitting it to address 255.255.255.255. (Note that in the case where a client already knows both its own address and the address of a BOOTP server, we don't need the relay agent at all—both the request and reply can be sent unicast over an arbitrary internetwork.)

**Relay Agent Receives Request and Processes It** The BOOTP relay agent on the physical network where the client is located is listening on UDP port 67 on the server's behalf. It processes the request as follows:

- It checks the value of the Hops field. If the value is less than or equal to 16, it increments it. If the value is greater than 16, it discards the request and does nothing further.
- It examines the contents of the GIAddr field. If this field is all zeros, it knows it is the first relay agent to handle the request and puts its own IP address into this field. (If the agent is a router, it has more than one IP address, so it chooses the one of the interface on which it received the request.)

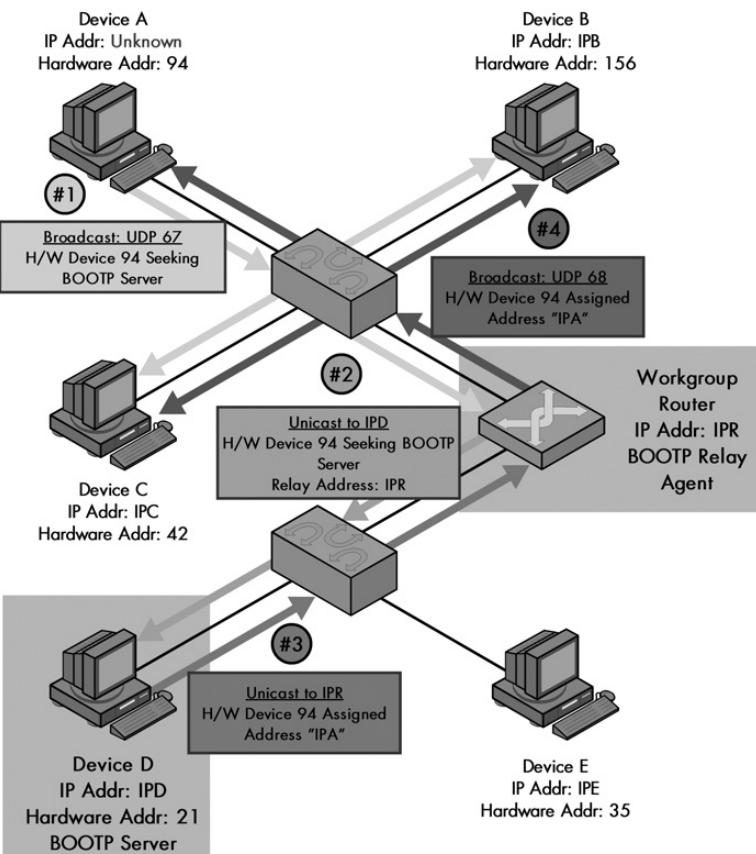
**Relay Agent Relays Request** The relay agent sends the BOOTP request to the BOOTP server. If the relay agent knows the server's IP address, it will send it unicast directly to the server. Otherwise, if the agent is a router, it may choose to broadcast the request on a different interface from the one on which it received the request. In the latter case, it is possible that multiple relay agents may be required to convey the request to the server. See the next section for more on this.

**Server Receives Request and Processes It** The BOOTP server receives the relayed request from the BOOTP relay agent. It processes it as normal.

**Server Creates Reply** The server creates a reply message as normal.

**Server Sends Reply** Seeing that the GIAddr field in the request was nonzero, the server knows the request was relayed. Instead of trying to send its reply back to the client that sent the request, it transmits the reply unicast back to the relay agent specified in GIAddr.

**Relay Agent Relays Reply** The BOOTP relay agent transmits the BOOTREPLY message back to the client. It does this either unicast or broadcast, depending on the value of the CIAddr field and the B (Broadcast) flag, just as a server does in the nonrelay case.



**Figure 60-5: BOOTP operation using a relay agent** In this example, Device A is trying to access a BOOTP server, but the only one is on a different network; the two are connected by a workgroup router that is configured to act as a BOOTP relay agent. Device A broadcasts its request, which the router receives. It relays the request to the BOOTP server, Device D, and puts its own IP address (IPR) into the BOOTP GIAddr field. The BOOTP server sends the reply back to the router using address IPR. The router then broadcasts it on Device A's local network so that Device A can receive it.

### Relaying BOOTP Requests Using Broadcasts

The simplest case of relaying is when each network has a relay agent that knows the IP address of the BOOTP server. The relay agent captures the request in step 3 of the procedure described in the preceding section, and sends it directly to the BOOTP server, wherever it may be on the network. The request is relayed as a regular unicast UDP message and routed to the BOOTP server. The BOOTP server's reply is routed back to the BOOTP relay agent, just like any UDP message in an IP datagram, and the relay agent forwards the reply.

It is also possible to set up BOOTP relay agents to relay requests even if they don't know the BOOTP server's address. These agents take requests received on one network and relay them to the next, where they expect another agent to continue the relaying process until a BOOTP server is reached. For example, suppose we have a set of three networks. Network N1 is connected to Network N2 using Router RA, and Network N2 connects to Network N3 using Router RB. Both of these routers function as relay agents but don't know the IP address of the BOOTP server. Here's what would happen if a client on Network N1 sent a request and the server was on Network N3:

1. The client would send its request.
2. Router RA would capture the request and put its address into GIAddr. It would increment the Hops field to a value of 1 and then broadcast the request out on Network N2.
3. Router RB would capture this request. It would see there is already an address in GIAddr, so it would leave that alone. It would increment the Hops field to 2 and broadcast the request on Network N3.
4. The BOOTP server would receive the request, process it, and return the reply directly back to Router RA.
5. Router RA would relay the reply back to the client.

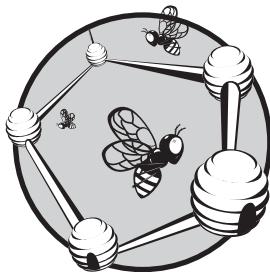
As you can see, the purpose of the Hops field is to ensure that errant requests don't circle around the network endlessly. Each relay agent increments it, and if the value of 16 is ever exceeded, the request is dropped. You can also see that any relay agents other than the first are involved only for handling the request; the reply is sent unicast back to the agent closest to the client.

Incidentally, if this multiple-step relaying process sounds like IP routing (only using broadcasts), and the Hops field sounds like the Time to Live (TTL) field in an IP datagram, then you've been paying attention. It is essentially the same idea (as explained in Chapter 21).



# 61

## DHCP OVERVIEW AND ADDRESS ALLOCATION CONCEPTS



In some ways, technological advancement can be considered more a journey than a destination. When a particular technology is refined or replaced with a superior one, it's usually only a matter of time before it, too, is replaced with something better. And so it was with the TCP/IP Boot Protocol (BOOTP), described in the previous chapter. While BOOTP was far more capable than the protocol it replaced, Reverse Address Resolution Protocol (RARP), after a number of years BOOTP, itself was replaced with a new TCP/IP configuration protocol: the *Dynamic Host Configuration Protocol (DHCP)*.

Where BOOTP represented a revolutionary change from RARP, DHCP is more of an evolution of BOOTP. It was built using BOOTP as a foundation, with the same basic message format. The most significant addition in DHCP is the ability to *dynamically* assign addresses to clients and to centrally

manage them. It is this capability that makes DHCP so powerful. Today, DHCP is the standard TCP/IP host configuration protocol and is used in everything from single-client home networks to enterprise-class internetworks.

In this first chapter on DHCP, I provide an overview of the protocol and a description of the concepts behind DHCP address assignment and leasing. I take a high-level look at how DHCP address assignment works and give a description of the three DHCP address allocation mechanisms. I then delve into DHCP leases and the policies and techniques used to decide how to implement DHCP leasing. I provide an overview of the lease life cycle from start to finish and describe the two DHCP lease timers that help control the process. Finally, I describe DHCP lease address pools and ranges, and the general concepts behind address management.

**RELATED INFORMATION** Since DHCP builds on BOOTP, they have a number of things in common. For example, DHCP makes use of BOOTP relay agent functionality, and DHCP options are basically the same as BOOTP vendor information fields. Since DHCP is the more common of the two protocols, I have tried to be complete in describing the operation of these features here, highlighting especially any differences between how they work for DHCP and in BOOTP. However, I have avoided duplicating the history and reasoning for the existence of many of these features. Since BOOTP came first, I have placed more of the historical information in the previous chapter. In general, if you plan to read about DHCP as well as BOOTP, I recommend reading the chapter on BOOTP first. If you don't plan to read up on BOOTP, you may wish to check the topic on DHCP/BOOTP interoperability in Chapter 64 instead.

## DHCP Overview, History, and Standards

As you learned in the previous chapter, BOOTP represents a significant improvement over RARP because it solves so many of RARP's problems. BOOTP is a higher-layer protocol, not hardware-dependent like RARP. It can support sending extra information beyond an IP address to a client to enable customized configuration. Also, through the use of BOOTP relay agents, it allows a large organization to use just one or two BOOTP servers to handle clients spread out over many physical networks. In so doing, BOOTP effectively solves one of the major classes of problems that administrators have with manual configuration: the "I have to go configure each host myself" issue. It allows "dumb" (storageless) hosts to configure themselves automatically and saves administrators the hassles of needing to trek to each host individually to specify important configuration parameters.

BOOTP normally uses a static method of determining what IP address to assign to a device. When a client sends a request, it includes its hardware address, which the server looks up in a table to determine the IP address for that client. (It is possible for BOOTP to use other methods of determining the relationship between an IP and hardware address, but static mapping is usually used.) This means BOOTP works well in relatively static environments, where changes to the IP addresses assigned to different devices are infrequent. Such networks were basically the norm in the 1980s and early 1990s.

Over time, many networks quickly started to move away from this model, for a number of reasons. As computers became smaller and lighter, it was more common for them to move from one network to another, where they would require a different address using the new network's network ID. Laptop and even palmtop computers could literally move from one network to another many times per day. Another

major issue was the looming exhaustion of the IP address space (see Chapter 17). For many organizations, permanently assigning a static IP address to each and every computer that might connect to their network was a luxury they could not afford.

In many organizations, trying to keep track of constant IP address changes became a daunting task in and of itself. BOOTP, with its static table of mappings between hardware addresses and IP addresses, simply wasn't up to the task. It also offered no way to reuse addresses; once an address had been assigned, a device could keep it forever, even if it were no longer needed.

## **DHCP: Building on BOOTP's Strengths**

A new host configuration protocol was needed to serve modern networks, which would move away from static, permanent IP address assignment. The Internet Engineering Task Force (IETF) supplied this in the form of DHCP, first formalized in RFC 1541, published in October 1993. (Actually, it was really originally specified in RFC 1531 in that same month, but due to minor errors in 1531, the standard was quickly revised and 1541 published.)

Because BOOTP worked well within its limitations and was also already widely deployed, it would not have made sense to start over from scratch with DHCP. This was especially so given that such a decision would have meant dealing with the inevitable painful transition, as well as compatibility problems associated with having both BOOTP and DHCP around for many years.

So, instead of tossing out BOOTP, DHCP was built on it as a foundation. In its simplest form, DHCP consists of two major components: an address allocation mechanism and a protocol that allows clients to request configuration information and servers to provide it. DHCP performs both functions in a manner similar to BOOTP, but with improvements.

## **Overview of DHCP Features**

The most significant differences between BOOTP and DHCP are in the area of address allocation, which is enhanced through the support for *dynamic* address assignment. Rather than using a static table that absolutely maps hardware addresses to IP addresses, a *pool* of IP addresses is used to dynamically allocate addresses. Dynamic addressing allows IP addresses to be efficiently allocated, and even shared among devices. At the same time, DHCP still supports static mapping of addresses for devices where this is needed.

The overall operation and communication between clients and servers are similar to that used by BOOTP, but with changes. The same basic request/reply protocol using UDP was retained for communicating configuration information, but additional message types were created to support DHCP's enhanced capabilities. BOOTP relay agents can be used by DHCP in a manner very similar to how they are used by BOOTP clients and server. The vendor information extensions from BOOTP were retained as well, but were formalized, renamed *DHCP options*, and extended to allow the transmission of much more information.

The result of all of this development effort is a widely accepted, universal host configuration protocol for TCP/IP that retains compatibility with BOOTP while significantly extending its capabilities. Today, DHCP is found on millions of networks

worldwide. It is used for everything from assigning IP addresses to corporate networks with thousands of hosts, to allowing a home Internet access router to automatically providing the correct Internet configuration information to a single user's computer.

**KEY CONCEPT** The *Dynamic Host Configuration Protocol (DHCP)* is the host configuration protocol currently used on modern TCP/IP internetworks. It was based on BOOTP and is similar to its predecessor in many respects, including the use of request/reply message exchanges and a nearly identical message format. However, DHCP includes added functionality, the most notable of which is *dynamic address assignment*, which allows clients to be assigned IP addresses from a shared pool managed by a DHCP server.

The original DHCP specification was revised in March 1997 with the publishing of RFC 2131, also titled “Dynamic Host Configuration Protocol.” This standard defined another new DHCP message (DHCPINFORM) type to allow active IP hosts to request additional configuration information. It also made several other small changes to the protocol. Since that time, numerous other DHCP-related RFCs have been published, most of which either define new DHCP option types (other kinds of information DHCP servers can send to DHCP clients) or slightly refine the way that DHCP is used in particular applications.

## DHCP Address Assignment and Allocation Mechanisms

The two main functions of DHCP are to provide a mechanism for assigning addresses to hosts and to provide a method by which clients can request addresses and other configuration data from servers. Both functions are based on the ones implemented in DHCP’s predecessor, BOOTP, but the changes are much more significant in the area of address assignment than they are in communication. It makes sense to start our look at DHCP here, since this will naturally lead us into a detailed discussion of defining characteristic of DHCP: *dynamic addressing*.

### DHCP Address Allocation

Providing an IP address to a client is the most fundamental configuration task performed by a host configuration protocol. To provide flexibility for configuring addresses on different types of clients, the DHCP standard includes three different address allocation mechanisms: manual, automatic, and dynamic.

I don’t really care for the names *automatic* and *dynamic* allocation, because they don’t do a good job of clearly conveying the differences between these methods. Both methods can be considered automatic, because in each, the DHCP server assigns an address without requiring any administrator intervention. The real difference between them is only in how long the IP address is retained, and therefore, whether a host’s address varies over time. I think better names would be *static* or *permanent* automatic allocation and *dynamic* or *temporary* automatic allocation.

Regardless of what you call them, all three of these methods exist for configuring IP hosts using DHCP. It is not necessary for administrators to choose one over the others. Instead, they will normally combine the methods, using each where it makes the most sense.

## **DHCP Manual Allocation**

With manual allocation, a particular IP address is preallocated to a single device by an administrator. DHCP communicates only the IP address to the device.

Manual allocation is the simplest method, and it is equivalent to the method BOOTP uses for address assignment, described in the previous chapter. Each device has an address that an administrator gives it ahead of time, and all DHCP does is look up the address in a table and send it to the client for which it is intended. This technique makes the most sense for devices that are mainstays of the network, such as servers and routers. It is also appropriate for other devices that must have a stable, permanent IP address.

Okay, now here's a fair question you might have. DHCP acts basically like BOOTP in the case of manual allocation. But BOOTP was created for devices that needed help with configuration. Servers and routers are complex devices with their own internal storage, and they obviously don't need a DHCP server to tell them their IP address as a diskless workstation does, so why bother using DHCP for them at all?

Well, in fact, you could just manually assign the address to the device directly and tell DHCP to ignore those addresses. However, using DHCP for manual assignments yields a different benefit: an administrative one. It keeps all the IP address information centralized in the DHCP address database, instead of requiring an administrator to go from machine to machine checking addresses and ensuring there are no duplicates. Updates can be made in a single place as well.

## **DHCP Dynamic Allocation**

While manual allocation is possible in DHCP, dynamic allocation is its real *raison d'être*. With dynamic allocation, DHCP assigns an IP address from a pool of addresses for a limited period of time chosen by the server, or until the client tells the DHCP server that it no longer needs the address. An administrator sets up the *pool* (usually a range or set of ranges) of IP addresses that are available for use. Each client that is configured to use DHCP contacts the server when it needs an IP address. The server keeps track of which IP addresses are already assigned, and it *leases* one of the free addresses from the pool to the client. The server decides the amount of time that the lease will last. When the time expires, the client must either request permission to keep using the address (*renew* the lease) or must get a new one.

Dynamic allocation is the method used for most client machines in modern DHCP-enabled IP internetworks. It offers numerous benefits, such as the following:

**Automation** Each client can be automatically assigned an IP address when it is needed, without any administrator intervention. Administrators do not need to manually decide which address goes with which client.

**Centralized Management** All the IP addresses are managed by the DHCP server. An administrator can easily look to see which devices are using which addresses and perform other network-wide maintenance tasks.

**Address Reuse and Sharing** By limiting the amount of time that each device holds an IP address, the DHCP server can ensure that the pool of IP addresses is used only by devices actively using the network. After a period of time, addresses no longer being used are returned to the pool, allowing other devices to use them. This allows an internetwork to support a total number of devices larger than the number of IP addresses available, as long as not all the devices connect to the internetwork at the same time.

**Portability and Universality** BOOTP (and DHCP manual allocation) both require that the DHCP server know the identity of each client that connects to it, so the server can find the client's assigned address. With dynamic allocation, there are no predefined allocations, so any client can request an IP address. This inherently makes dynamic allocation the ideal choice for supporting mobile devices that travel between networks.

**Conflict Avoidance** Since IP addresses are all assigned from a pool that is managed by the DHCP server, IP address conflicts are avoided. This, of course, assumes that all the clients use DHCP. The administrator must ensure that the address pool is not used by non-DHCP devices.

### **DHCP Automatic Allocation**

With the automatic allocation method, DHCP automatically assigns an IP address permanently to a device, selecting it from a pool of available addresses. This method can be used in cases where there are enough IP addresses for each device that may connect to the network, but where devices don't really care which IP address they use. Once an address is assigned to a client, that device will keep using it. Automatic allocation can be considered a special case of dynamic allocation: It is essentially dynamic allocation where the time limit on the use of the IP address by a client (the lease length) is forever.

In practice, automatic allocation is not used nearly as much as dynamic allocation, for a simple reason: Automatically assigning an IP address to a device permanently is a risky move. Most administrators feel it is better to use manual allocation for the limited number of machines that really need a permanent IP address assignment and to use dynamic addressing for others.

**KEY CONCEPT** DHCP defines three basic mechanisms for address assignment. *Dynamic allocation* is the method most often used, and it works by having each client *lease* an address from a DHCP server for a period of time. The server chooses the address dynamically from a shared address pool. *Automatic allocation* is like dynamic allocation, but the address is assigned permanently instead of being leased. *Manual allocation* preassigns an address to a specific device, just as BOOTP does, and is normally used only for servers and other permanent, important hosts.

## DHCP Leases

Of the three address allocation methods supported by DHCP, dynamic address allocation is by far the most popular and important. The significance of the change that dynamic addressing represents to how IP addresses are used in TCP/IP can be seen in the semantics of how addresses are treated in DHCP. Where conventionally a host was said to *own* an IP address, when dynamic address allocation is used, hosts are said instead to *lease* an address.

The notion of a lease conveys very accurately the difference between dynamic allocation and the other types. A host no longer is strictly entitled to a particular address, with a server merely telling it what the address is. In DHCP, the server remains the real owner of all the IP addresses in the address pool, and it merely gives permission for a client to use the address for a period of time. The server guarantees that it will not try to use the address for another client only during this time. The client is responsible for taking certain actions if it wants to continue using the address. If it does not successfully reacquire permission for using the address after a period of time, it must stop using it or risk creating an IP address conflict on the network.

**KEY CONCEPT** DHCP's most significant new feature is dynamic allocation, which changes the way that IP addresses are managed. Where in traditional IP each device owns a particular IP address, in DHCP the server owns all the addresses in the address pool, and each client leases an address from the server, usually for only a limited period of time.

### DHCP Lease Length Policy

When dynamic address allocation is used, the administrator of the network must provide parameters to the DHCP server to control how leases are assigned and managed. One of the most important decisions to be made is the *lease length policy* of the internetwork: how long the administrator wants client leases to last. This choice will depend on the network, the server, and the clients. The choice of lease time, like so many other networking parameters, comes down to a trade-off between *stability* and *allocation efficiency*.

The primary benefit of using long lease times is that the addresses of devices are relatively stable. A device doesn't need to worry about its IP address changing all the time, and neither does its user. This is a significant advantage in many cases, especially when it is necessary for the client to perform certain server functions, accept incoming connections, or use a DNS domain name (ignoring for the moment dynamic DNS capabilities). In those situations, having the IP address of a device moving all over the place can cause serious complications.

The main drawback of using long leases is that they substantially increase the amount of time that an IP address, once it is no longer needed, is tied up before it can be reused. In the worst-case scenario, the amount of wasted time for an allocation can be almost as long as the lease itself. If we give a device a particular address for six months and after two weeks the device is shut down and no longer used, the IP address that it was using is still unavailable for another five and a half more months.

For this reason, many administrators prefer to use short leases. This forces a client to continually renew the lease as long as it needs it. When it stops asking for permission, the address is quickly put back into the pool. This makes shorter leases a better idea in environments where the number of addresses is limited and must be conserved. The drawback is the opposite of the benefit of long leases: constantly changing IP addresses.

Administrators do not need to pick from short and long lease durations. They can compromise by choosing a number that best suits the network. The following are some examples of lease times and the reasoning behind them:

**One Hour or Less** Ensures maximum IP address allocation efficiency in a very dynamic environment where there are many devices connecting and disconnecting from the network, and the number of IP addresses is limited.

**One Day** Suitable for situations where guest machines typically stay for a day, to increase IP efficiency when many employees work part time, or otherwise to ensure that every day each client must ask again for permission to use an address.

**Three Days** The default used by Microsoft, which alone makes it a popular choice.

**One Week** A reasonable compromise between the shorter and longer times.

**One Month** Another compromise, closer to the longer end of the lease time range.

**Three Months** Provides reasonable IP address stability so that addresses don't change very often in reasonably static environments. Also a good idea if there are many IP addresses available and machines are often turned off for many days or weeks at a time. For example, this duration may be used in a university setting to ensure that IP addresses of returning students are maintained over the summer recess.

**One Year** An approximation of an infinite lease.

Not only is the administrator not restricted to a limited number of possible lease durations, it is not necessary for the administrator to choose a constant lease length policy for all clients. Depending on the capabilities of the DHCP server, an administrator may select different lease lengths for certain clients. For example, the administrator may decide to use long leases for desktop computers that are permanently assigned to a particular subnet and not moved, and a pool of short-leased addresses for notebook computers and visitors. In some DHCP implementations, this can be done by assigning clients to particular classes. Of course, this requires more work (and may even require multiple servers).

In selecting a lease time policy, the administrator must also bear in mind that, by default, after half the length of a lease, the client will begin attempting to renew the lease. This may make it more advisable to use a longer lease time, to increase the amount of time between when a client tries to renew the lease and when the lease expires. For example, in a network with a single DHCP server, an administrator may want to use leases no shorter than eight hours. This provides a four-hour window for maintenance on the server without leases expiring.

When a lease is very short, such as minutes or hours, it will typically expire when a client machine is turned off for a period of time, such as overnight. Longer leases will persist across reboots. The client in this case will still contact the DHCP server each time it is restarted to *reallocate* the address—confirm that it may continue using the address it was assigned.

**KEY CONCEPT** A key decision that a network administrator using DHCP must make is what the network's *lease length policy* will be. Longer leases allow devices to avoid changing addresses too often; shorter leases are more efficient in terms of reallocating addresses that are no longer required. An administrator can choose from a variety of different lease times and may choose longer leases for some devices than for others.

## Issues with Infinite Leases

In addition to choosing a particular lease length number, it is possible to specify an infinite lease length duration for certain clients. This effectively turns dynamic allocation into automatic allocation for a particular client. As I said earlier, however, this is generally not done. The reason is that an infinite lease never expires, and as the old saw goes, “Never is a long time.”

Permanently assigning an IP address from a pool is a somewhat risky move, because once assigned, if anything occurs that causes that address to be no longer used, it can never be recovered. A worst-case scenario would be a visitor to a company site who plugs a notebook computer in to the network to check email or transfer a file. If that machine is assigned an IP address using automatic allocation, the visitor will take it with him when he leaves. Obviously, this is not a great idea.

For this reason, most administrators prefer to use dynamic allocation instead, with addresses set to a very long time frame, such as a year or two years. This is considered near enough to infinity that it approximates a permanent assignment, but allows an IP address to eventually be recovered if a device stops using it. In such a policy, anything that really, truly needs a permanent assignment is given an address using *manual assignment*, which requires a conscious decision to dedicate the address to a particular device.

**RELATED INFORMATION** For a little more information related to lease length selection, see the section on DHCP server implementation problems and issues in Chapter 64.

## DHCP Lease Life Cycle and Lease Timers

The use of dynamic address allocation in DHCP means a whole new way of thinking about addresses. A client no longer owns an address, but rather leases it. This means that when a client machine is set to use DHCP dynamic addressing, it can never assume that it has an address on a permanent basis. Each time it powers up, it must engage in communications with a DHCP server to begin or confirm the lease of an address. It also must perform other activities over time to manage this lease and possibly terminate it.

Calling dynamic address assignments leases is a good analogy, because a DHCP IP address lease is similar to a real-world lease in a number of respects. For example, when you rent an apartment, you sign the lease. Then you use the apartment for a

period of time. Typically, assuming you are happy with the place, you will *renew* the lease before it expires, so you can keep using it. If by the time you get near the end of the lease the owner of the apartment has not allowed you to renew it, you will probably lease a different apartment to ensure you have somewhere to live. And if you decide, say, to move out of the country, you may terminate the lease and not get another.

## DHCP Lease Life Cycle Phases

DHCP leases follow a lease life cycle that generally consists of the following six phases:

**Allocation** A client begins with no active lease, and hence, no DHCP-assigned address. It acquires a lease through a process of *allocation*.

**Reallocation** If a client already has an address from an existing lease, then when it reboots or starts up after being shut down, it will contact the DHCP server that granted it the lease to confirm the lease and acquire operating parameters. This is sometimes called *reallocation*; it is similar to the full allocation process but shorter.

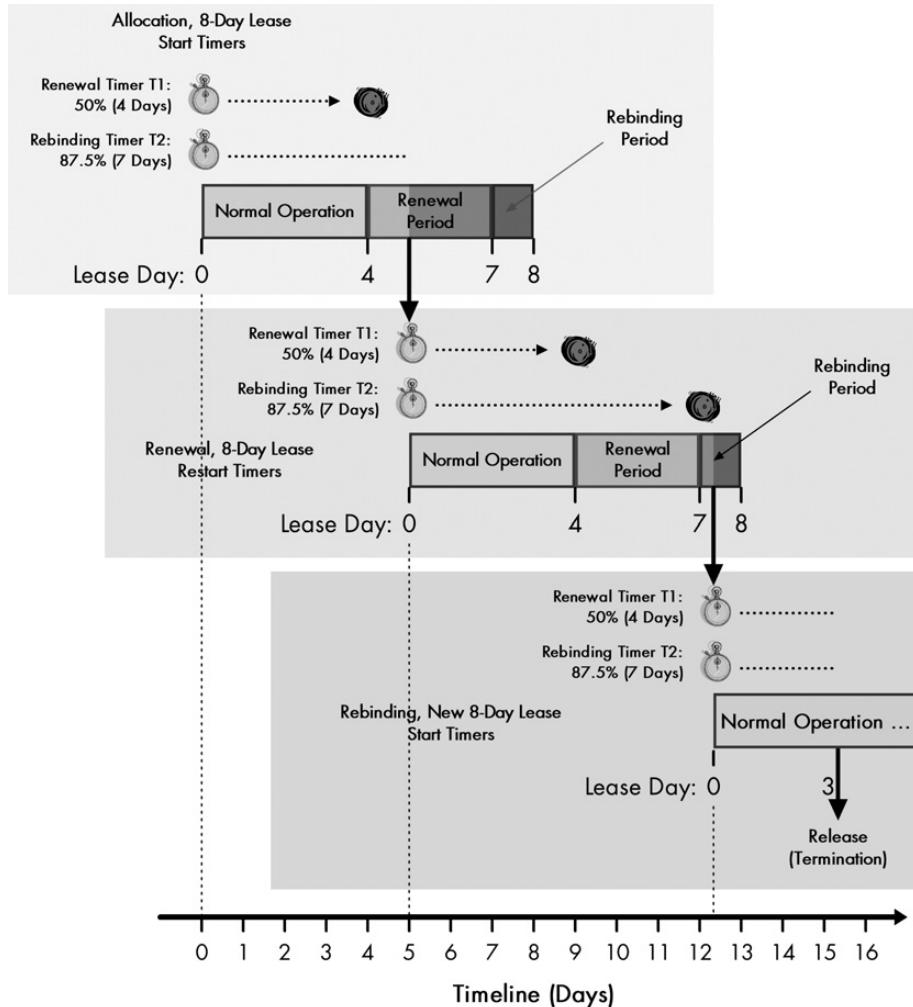
**Normal Operation** Once a lease is active, the client functions normally, using its assigned IP address and other parameters during the main part of the lease. The client is said to be *bound* to the lease and the address.

**Renewal** After a certain portion of the lease time has expired, the client will attempt to contact the server that initially granted the lease to *renew* the lease, so it can keep using its IP address.

**Rebinding** If renewal with the original leasing server fails (because, for example, the server has been taken offline), the client will try to *rebind* to any active DHCP server, in an attempt to extend its current lease with any server that will allow it to do so.

**Release** The client may decide at any time that it no longer wishes to use the IP address it was assigned, and may terminate the lease, *releasing* the IP address. Like the apartment renter moving out of the country, this may be done if a device is moving to a different network, for example. (Of course, unlike DHCP servers, landlords usually don't let you cancel a lease at your leisure, but hey, no analogy is perfect.)

Figure 61-1 illustrates the DHCP life cycle using an example that spans three individual leases.



**Figure 61-1: DHCP life cycle example** In this example, the initial lease has a duration of 8 days and begins at day 0. The T1 and T2 timers are set for 4 days and 7 days, respectively. When the T1 timer expires, the client enters the renewal period and successfully renews at day 5 with a new 8-day lease. When this second lease's T1 timer expires, the client is unable to renew with the original server. It enters the rebinding period when its T2 timer goes off, and it is granted a renewed 8-day lease with a different server. Three days into this lease, it is moved to a different network and no longer needs its leased address, so it voluntarily releases it.

## **Renewal and Rebinding Timers**

The processes of renewal and rebinding are designed to ensure that a client's lease can be extended before it is scheduled to end, so no loss of functionality or interruption occurs to the user of the client machine. Each time an address is allocated or reallocated, the client starts two timers that control the renewal and rebinding process:

**Renewal Timer (T1)** This timer is set by default to 50 percent of the lease period. When it expires, the client will begin the process of renewing the lease. It is simply called *T1* in the DHCP standards.

**Rebinding Timer (T2)** This timer is set by default to 87.5 percent of the length of the lease. When it expires, the client will try to rebind, as described in the previous section. It is given the snappy name *T2* in the DHCP standards.

Naturally, if the client successfully renews the lease when the T1 timer expires, this will result in a fresh lease, and both timers will be reset. T2 comes into play only if the renewal is not successful. It is possible to change the amount of time to which these timers are set, but obviously T1 must expire before T2, which must expire before the lease itself ends. These usually are not changed from the default, but may be modified in certain circumstances.

**KEY CONCEPT** DHCP leases follow a conceptual *life cycle*. The lease is first assigned to the client through a process of *allocation*; if the device later reboots, it will *reallocate* the lease. After a period of time controlled by the *renewal timer (T1)*, the device will attempt to *renew* its lease with the server that allocated it. If this fails, the *rebinding timer (T2)* will go off, and the device will attempt to *rebind* the lease with any available server. The client may also *release* its IP address if it no longer needs it.

The lease life cycle is described in the DHCP standards in the form of states that the client moves through as it acquires a lease, uses it, and then either renews or ends it. The next chapter describes these states and the specific exchanges of messages between a client and server to accomplish different lease activities.

## **DHCP Lease Address Pools, Ranges, and Address Management**

Simpler host configuration methods such as BOOTP (or DHCP manual allocation for that matter) associate a single IP address with each client machine. DHCP dynamic addressing removes this one-to-one correspondence, in favor of flexible address mapping to clients on an as-needed basis. The clients no longer own the addresses, but lease them from the true owner, the server. Thus, a primary job of both a DHCP server and the administrator of that server is to maintain and manage these client addresses.

## **Address Pool Size Selection**

The set of all addresses that a DHCP server has available for assignment is most often called the *address pool*. The first issue related to address management is ensuring that the address pool is large enough to serve all the clients that will be using the server. The number of addresses required depends on several factors:

**Number of Clients** This is an obvious factor.

**Stability and Frequency of Use of Clients** If most clients are left on and connected to the network all the time, you will probably need to plan on an address for each one. In contrast, if you are serving part-time employees or consultants who frequently travel, you can get away with sharing a smaller number of addresses.

**Consequences of Overallocation** If having certain clients be unable to get a free address is a problem, you need to more carefully manage the address pool to ensure that you don't run out of IP addresses. If having a client not get an address is *never* acceptable, make sure you have as many or more addresses as clients.

I'm sure you've probably noticed that these issues are similar to those that I raised in discussing lease lengths earlier in this chapter. In fact, the two matters are intimately related. Generally speaking, having more addresses gives the administrator the luxury of using longer leases. If you are short on addresses, you probably need to use shorter leases to reduce the chances of any unused addresses continuing to be allocated to devices not needing them.

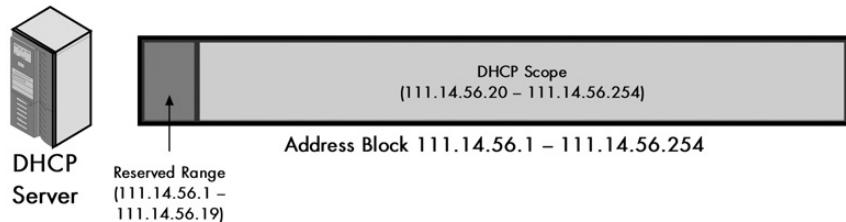
## **Lease Address Ranges (Scopes)**

In its simplest form, the address pool takes the form of a list of all addresses that the DHCP server has reserved for dynamic client allocation. Along with each address, the server stores certain parameters, such as a default lease length for the address and other configuration information to be sent to the client when it is assigned that address (for example, a subnet mask and the address of a default router). All of this data is stored in a special database on the server.

Of course, many clients will request addresses from this pool. Most of these clients are equals as far as the DHCP server is concerned, and it doesn't matter which address each individual client gets. This means most of the information stored with each of the addresses in a pool may be the same, except for the address number itself. Due to this similarity, it would be inefficient to need to specify each address and its parameters individually. Instead, a *range* of addresses is normally handled as a single group defined for a particular network or subnet. These are not given any particular name in the DHCP standards, but are commonly called *scopes*. This term has been popularized by Microsoft in its DHCP server implementations. Other operating systems sometimes just call these blocks of addresses *ranges*, but I prefer scope.

**KEY CONCEPT** Each DHCP server maintains a set of IP addresses that it uses to allocate leases to clients. These are usually contiguous blocks of addresses assigned to the server by an administrator, called DHCP *address ranges* or *scopes*.

The exact method for setting up scopes depends on the particular operating system and DHCP server software. However, each scope definition typically begins by specifying a range of addresses using a starting and an ending IP address. For example, if a company was assigned the IP address block 111.14.56.0/24, the administrator might set up a scope encompassing addresses 111.14.56.20 through 111.14.56.254, as shown in Figure 61-2. Then for that scope, the administrator can set up various parameters to be specified to each client assigned an address from the scope.



**Figure 61-2: DHCP scope** A single DHCP server scope, encompassing addresses 111.14.56.1 through 111.14.56.254.

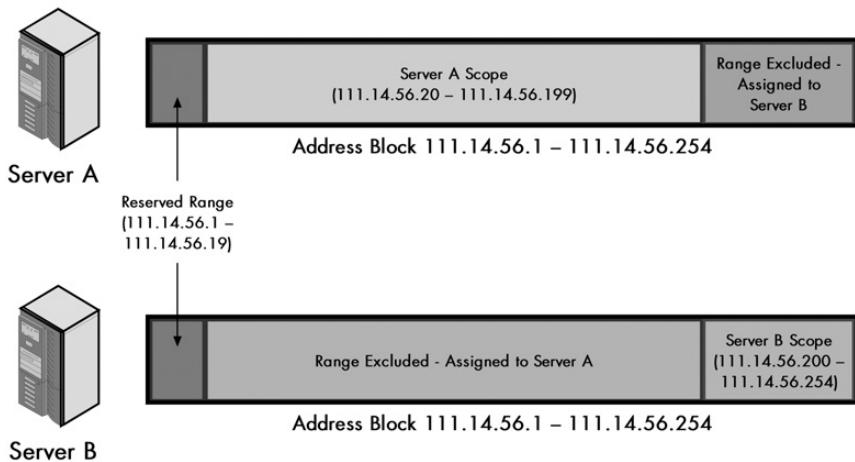
Why not start at 111.14.56.1? Usually, we will want to set aside certain IP addresses for manual configuration of servers, routers, and other devices requiring a fixed address. One easy way to do that is to simply reserve a block of addresses that aren't used by DHCP. Alternatively, most DHCP server software will allow you to specify a range but *exclude* an address or set of addresses from the range. So, we could specify 111.14.56.1 through 111.14.56.254 and individually mark as not available addresses we manually assign. Or we could specify that 111.14.56.1 through 111.14.56.19 are reserved.

Instead of putting all of its addresses (except excluded ones) in a single scope, a server may use *multiple* scopes. One common reason for the latter approach is to support more than one subnet on a server. Multiple scopes are also commonly used when multiple DHCP servers are used to serve the same clients. There are two ways to do this: by having either *overlapping* or *non-overlapping* scopes.

Overlapping scopes allows each server to assign any address from the same pool. However, the DHCP standard doesn't specify any way for servers to communicate with each other when they assign an address, so if both servers were told they could assign addresses from the same address pool, this could result in both servers trying to assign a particular address to two different devices. As a result, if you are using two DHCP servers (as is often recommended for redundancy reasons), the administrator generally gives them different, non-overlapping scope assignments. Alternatively, the same scope is given to each server, with each server told to exclude from use the addresses the other server is assigning.

For example, suppose we have two DHCP servers: Server A (the main server) and Server B (the backup). We want to assign most of the addresses to Server A and a few as backup to Server B. We could give both Server A and Server B the scope 111.14.56.1 through 111.14.56.254. We would exclude 111.14.56.1 through 111.14.56.19 from both. Then we would exclude from Server A the range 111.14.56.200 through 111.14.56.254 and exclude from Server B the range 111.14.20 through 111.14.56.199. Figure 61-3 shows how this would work. The

main advantage of this method is that if one server goes down, the administrator can quickly remove the exclusion and let the remaining server access all addresses. Also, if one server runs out of addresses while the other has plenty, the allocations can be shifted easily.



**Figure 61-3: DHCP multiple-server non-overlapping scopes** DHCP Servers A and B have been assigned non-overlapping scopes to ensure that they do not conflict. This has been done by starting with the same scope definition for both. The common reserved range is excluded from each. Then Server A has Server B's address range excluded (hatched area at right in the top bar), and Server B has Server A's range excluded (hatched area in the middle at bottom).

### Other Issues with Address Management

There are many other issues related to address management, which start to get into the guts of DHCP server implementation. For example, as was the case with BOOTP, we may need to use relay agents when the DHCP server is responsible for addresses on a subnet different from its own. There are also special DHCP features that affect how addresses are managed. For example, the DHCP conflict detection feature can actually allow two servers to have overlapping scopes, despite what I said in the previous section. Chapter 64, which covers DHCP implementation and features, describes these issues in more detail.

**KEY CONCEPT** If a site has multiple DHCP servers, they can be set up with either *overlapping* or *non-overlapping* scopes. Overlapping scopes allow each server to assign from the same pool, providing flexibility, but raising the possibility of two clients being assigned the same address unless a feature such as *server conflict detection* is employed. Non-overlapping scopes are safer because each server has a dedicated set of addresses for its use, but this means one server could run out of addresses while the other still has plenty, and if a server goes down, its addresses will be temporarily unallocatable.

