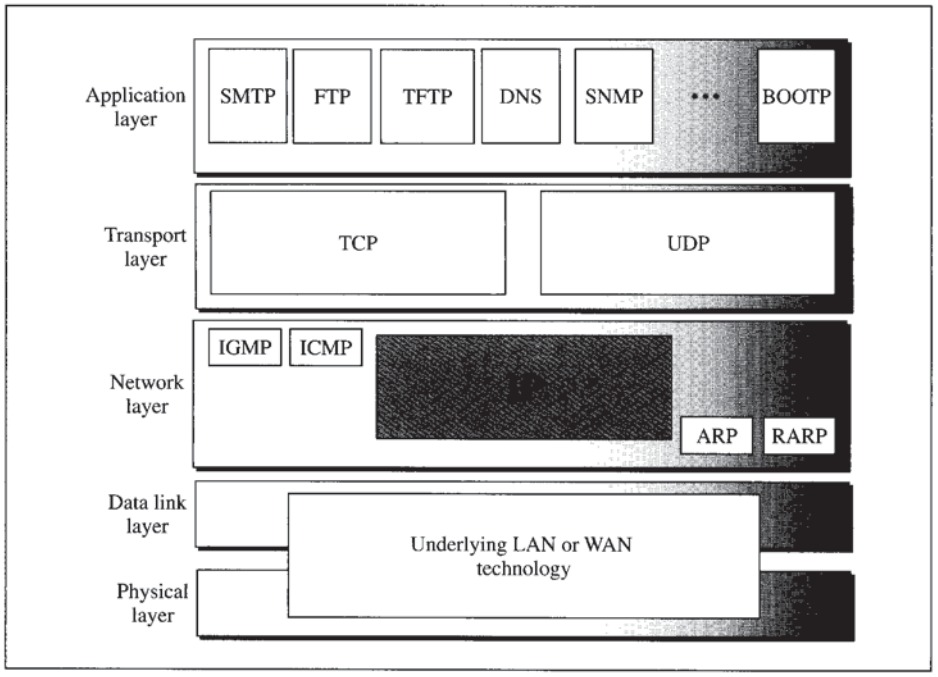
# CHAPTER 8

# *Internet Protocol (IP)*

The Internet Protocol (IP) is the transmission mechanism used by the TCP/IP protocols. Figure 8.1 shows the position of IP in the suite.

**Figure 8.1** *Position of IP in TCP/IP protocol suite*




IP is an unreliable and connectionless datagram protocol—a best-effort delivery service. The term *best-effort* means that IP provides no error checking or tracking. IP assumes the unreliability of the underlying layers and does its best to get a transmission through to its destination, but with no guarantees.

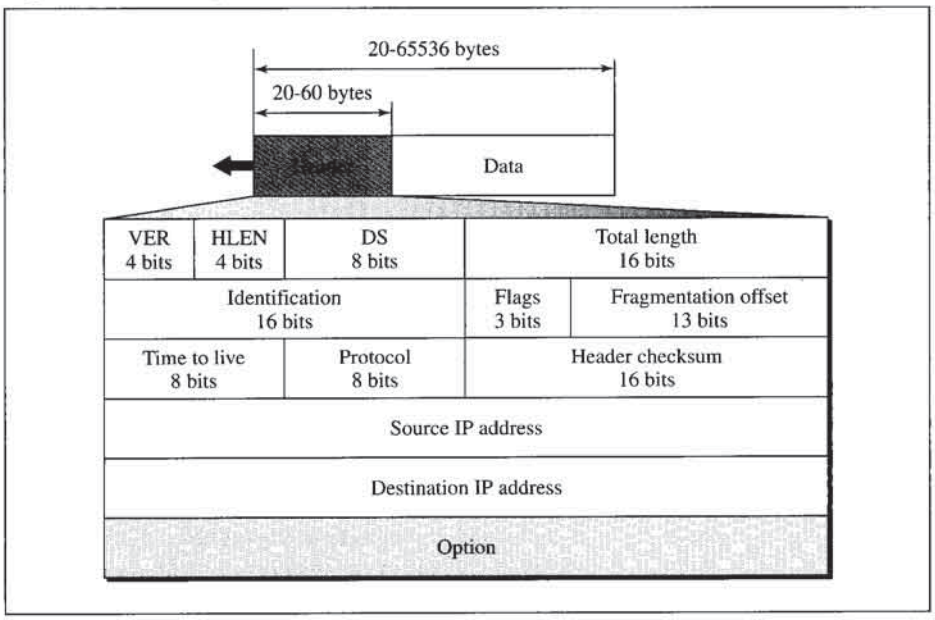
If reliability is important, IP must be paired with a reliable protocol such as TCP. An example of a more commonly understood best-effort delivery service is the post office. The post office does its best to deliver the mail but does not always succeed. If an unregistered letter is lost, it is up to the sender or would-be recipient to discover the loss and rectify the problem. The post office itself does not keep track of every letter and cannot notify a sender of loss or damage.

IP is also a connectionless protocol packaged for a packet switching network that uses the datagram approach (see Chapter 6). This means that each datagram is handled independently, and each datagram can follow a different route to the destination. This implies that datagrams sent by the same source to the same destination could arrive out of order. Also, some could be lost or corrupted during transition. Again, IP relies on a higher level protocol to take care of all these problems.

## 8.1 DATAGRAM

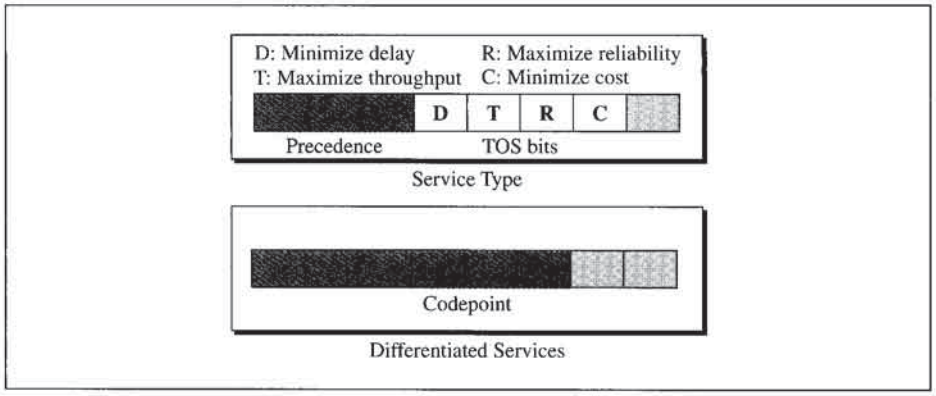
Packets in the IP layer are called **datagrams.** Figure 8.2 shows the IP datagram format. A datagram is a variable-length packet consisting of two parts: header and data. The header is 20 to 60 bytes in length and contains information essential to routing and delivery. It is customary in TCP/IP to show the header in 4-byte sections. A brief description of each field is in order.

**Figure 8.2** *IP datagram*



- **Version (VER).** This 4-bit field defines the version of the IP protocol. Currently the version is 4. However, version 6 (or IPng) may replace version 4 in a few years. This field tells the IP software running in the processing machine that the datagram has the format of version 4. All fields must be interpreted as specified in the fourth version of the protocol. If the machine is using some other version of IP, the datagram is discarded rather than interpreted incorrectly.
- **Header length (HLEN).** This 4-bit field defines the total length of the datagram header in 4-byte words. This field is needed because the length of the header is variable (between 20 and 60 bytes). When there are no options, the header length is 20 bytes, and the value of this field is 5 ($5 \times 4 = 20$). When the option field is at its maximum size, the value of this field is 15 ($15 \times 4 = 60$).
- **Differentiated Services (formerly Service Type).** IETF has recently changed the interpretation and name of this 8-bit field. This field, previously called Service Type, is now called Differentiated Services. We show both interpretations in Figure 8.3.

**Figure 8.3** *Service Type or Differentiated Services*




1. **Service Type**
   In this interpretation, the first 3 bits are called precedence bits. The next 4 bits are called TOS bits and the last bit is not used.
   a. **Precedence** is a 3-bit subfield ranging from 0 (000 in binary) to 7 (111 in binary). The precedence defines the priority of the datagram in issues such as congestion. If a router is congested and needs to discard some datagrams, those datagrams with lowest precedence are discarded first. Some datagrams in the Internet are more important than others. For example, a datagram used for network management is much more urgent and important than a datagram containing optional information for a group. At present, the precedence subfield, however, is not used. It is expected to be functional in future versions.

| The precedence subfield is not used in version 4. |
|---|

b. **TOS bits** is a 4-bit subfield with each bit having a special meaning. Although a bit can be either 0 or 1, one and only one of the bits can have the value of 1 in each datagram. The bit patterns and their interpretations are given in Table 8.1. With only one bit set at a time, we can have five different types of services.

**Table 8.1** *Types of service*

| *TOS Bits* | *Description* |
|---|---|
| 0000 | Normal (default) |
| 0001 | Minimize cost |
| 0010 | Maximize reliability |
| 0100 | Maximize throughput |
| 1000 | Minimize delay |

Application programs can request a specific type of service. The defaults for some applications are shown in Table 8.2.

**Table 8.2** *Default types of service*

| *Protocol* | *TOS Bits* | *Description* |
|---|---|---|
| ICMP | 0000 | Normal |
| BOOTP | 0000 | Normal |
| NNTP | 0001 | Minimize cost |
| IGP | 0010 | Maximize reliability |
| SNMP | 0010 | Maximize reliability |
| TELNET | 1000 | Minimize delay |
| FTP (data) | 0100 | Maximize throughput |
| FTP (control) | 1000 | Minimize delay |
| TFTP | 1000 | Minimize delay |
| SMTP (command) | 1000 | Minimize delay |
| SMTP (data) | 0100 | Maximize throughput |
| DNS (UDP query) | 1000 | Minimize delay |
| DNS (TCP query) | 0000 | Normal |
| DNS (zone) | 0100 | Maximize throughput |

It is clear from the above table that interactive activities, activities requiring immediate attention, and activities requiring immediate response need minimum delay. Those activities that send bulk data require maximum throughput. Management activities need maximum reliability. Background activities need minimum cost.

2. **Differentiated Services**
   In this interpretation, the first 6 bits make up the **codepoint** subfield and the last two bits are not used. The codepoint subfield can be used in two different ways.
   a. When the 3 right-most bits are 0s, the 3 left-most bits are interpreted the same as the precedence bits in the Service Type interpretation. In other words, it is compatible with the old interpretation.
   b. When the 3 right-most bits are not all 0s, the 6 bits define 64 services based on the priority assignment by the Internet or local authorities according to Table 8.3. The first category contains 32 service types; the second and the third each contain 16. The first category (numbers 0, 2, 4, . . . , 62) is assigned by the Internet authorities (IETF). The second category (3, 7, 11, 15, . . . , 63) can be used by local authorities (organizations). The third category (1, 5, 9, . . . , 61) is temporary and can be used for experimental purposes. Note that the numbers are not contiguous. If they were, the first category would range from 0 to 31, the second 32 to 47, and the third 48 to 63. This would be incompatible with the TOS interpretation because XXX000 (which includes 0, 8, 16, 24, 32, 40, 48, and 56) would fall into all three categories. Instead, in this assignment method all these services belong to category 1. Note that these assignments have not yet been finalized.

**Table 8.3** *Values for codepoints*

| *Category* | *Codepoint* | *Assigning Authority* |
|---|---|---|
| 1 | XXXXX0 | Internet |
| 2 | XXXX11 | Local |
| 3 | XXXX01 | Temporary or experimental |

- **Total length.** This is a 16-bit field that defines the total length (header plus data) of the IP datagram in bytes. To find the length of the data coming from the upper layer, subtract the header length from the total length. The header length can be found by multiplying the value in the HLEN field by four.

$$\text{length of data} = \text{total length} - \text{header length}$$

Since the field length is 16 bits, the total length of the IP datagram is limited to 65,535 ($2^{16} - 1$) bytes, of which 20 to 60 bytes are the header and the rest is data from the upper layer.

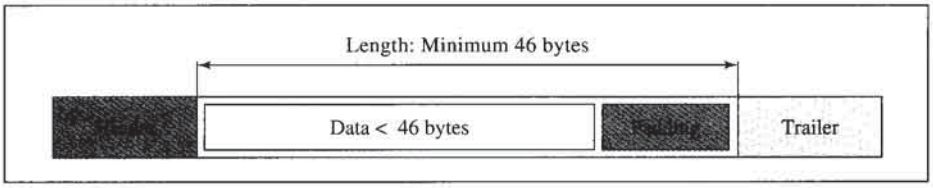
> The total length field defines the total length of the datagram including the header.

Though a size of 65,535 bytes might seem large, the size of the IP datagram may increase in the near future as the underlying technologies allow even more throughput (more bandwidth).

When we discuss fragmentation in the next section, we will see that some physical networks are not able to encapsulate a datagram of 65,535 bytes in their frames. The datagram must be fragmented to be able to pass through those networks.

One may ask why we need this field anyway. When a machine (router or host) receives a frame, it drops the header and the trailer leaving the datagram. Why include an extra field that is not needed? The answer is that in many cases we really do not need the value in this field. However, there are occasions in which the datagram is not the only thing encapsulated in a frame; it may be that padding has been added. For example, the Ethernet protocol has a minimum and maximum restriction on the size of data that can be encapsulated in a frame (46 to 1500 bytes). If the size of an IP datagram is less than 46 bytes, some padding will be added to meet this requirement. In this case, when a machine decapsulates the datagram, it needs to check the total length field to determine how much is really data and how much is padding (see Figure 8.4).

**Figure 8.4** *Encapsulation of a small datagram in an Ethernet frame*



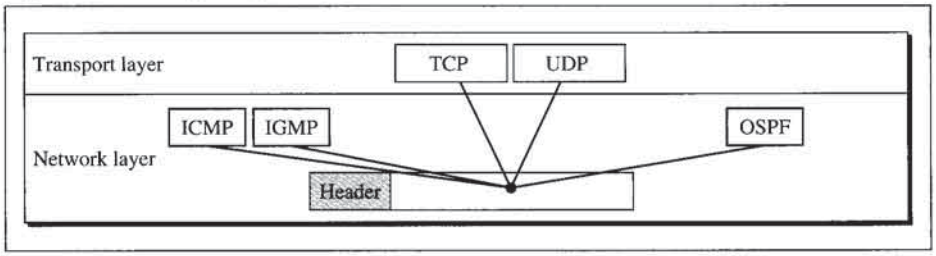

- **Identification.** This field is used in fragmentation (discussed in the next section).
- **Flags.** This field is used in fragmentation (discussed in the next section).
- **Fragmentation offset.** This field is used in fragmentation (discussed in the next section).
- **Time to live.** A datagram has a limited lifetime in its travel through an internet. This field was originally designed to hold a timestamp, which was decremented by each visited router. The datagram was discarded when the value became zero. However, for this scheme, all the machines must have synchronized clocks and must know how long it takes for a datagram to go from one machine to another. Today, this field is mostly used to control the maximum number of hops (routers) visited by the datagram. When a source host sends the datagram, it stores a number in this field. This value is approximately two times the maximum number of routes between any two hosts. Each router that processes the datagram decrements this number by one. If this value, after being decremented, is zero, the router discards the datagram.

  This field is needed because routing tables in the Internet can become corrupted. A datagram may travel between two or more routers for a long time without ever getting delivered to the destination host. Resources may become tied up. This field limits the lifetime for a datagram and prevents old datagrams from popping out of the network and perhaps confusing higher-level protocols (especially TCP).

  Another use of this field is to intentionally limit the journey of the packet. For example, if the source wants to confine the packet to the local network, it can store 1 in this field. When the packet arrives at the first router, this value is decremented to 0, and the datagram is discarded.

- **Protocol.** This 8-bit field defines the higher-level protocol that uses the services of the IP layer. An IP datagram can encapsulate data from several higher level protocols such as TCP, UDP, ICMP, and IGMP. This field specifies the final destination protocol to which the IP datagram should be delivered. In other words, since the IP protocol multiplexes and demultiplexes data from different higher-level protocols, the value of this field helps in the demultiplexing process when the datagram arrives at its final destination (see Figure 8.5).

**Figure 8.5** *Multiplexing*




The value of this field for different higher-level protocols is shown in Table 8.4.

**Table 8.4** *Protocols*

| *Value* | *Protocol* |
|---|---|
| 1 | ICMP |
| 2 | IGMP |
| 6 | TCP |
| 17 | UDP |
| 89 | OSPF |

- **Checksum.** The checksum concept and its calculation are discussed later in this chapter.
- **Source address.** This 32-bit field defines the IP address of the source. This field must remain unchanged during the time the IP datagram travels from the source host to the destination host.
- **Destination address.** This 32-bit field defines the IP address of the destination. This field must remain unchanged during the time the IP datagram travels from the source host to the destination host.

### Example 1

An IP packet has arrived with the first 8 bits as shown:

⟵ 01000010

The receiver discards the packet. Why?

### Solution

There is an error in this packet. The 4 left-most bits (0100) show the version, which is correct. The next 4 bits (0010) show the header length, which means ($2 \times 4 = 8$), which is wrong. The minimum number of bytes in the header must be 20. The packet has been corrupted in transmission.

### Example 2

In an IP packet, the value of HLEN is 1000 in binary. How many bytes of options are being carried by this packet?

### Solution

The HLEN value is 8, which means the total number of bytes in the header is $8 \times 4$ or 32 bytes. The first 20 bytes are the main header, the next 12 bytes are the options.

### Example 3

In an IP packet, the value of HLEN is $5_{16}$ and the value of the total length field is $0028_{16}$. How many bytes of data are being carried by this packet?

### Solution

The HLEN value is 5, which means the total number of bytes in the header is $5 \times 4$ or 20 bytes (no options). The total length is 40 bytes, which means the packet is carrying 20 bytes of data ($40 - 20$).

### Example 4

An IP packet has arrived with the first few hexadecimal digits as shown below:




How many hops can this packet travel before being dropped? The data belong to what upper layer protocol?

### Solution

To find the time-to-live field, we should skip 8 bytes (16 hexadecimal digits). The time-to-live field is the ninth byte, which is 01. This means the packet can travel only one hop. The protocol field is the next byte (02), which means that the upper layer protocol is IGMP (see Table 8.4).

## 8.2 FRAGMENTATION

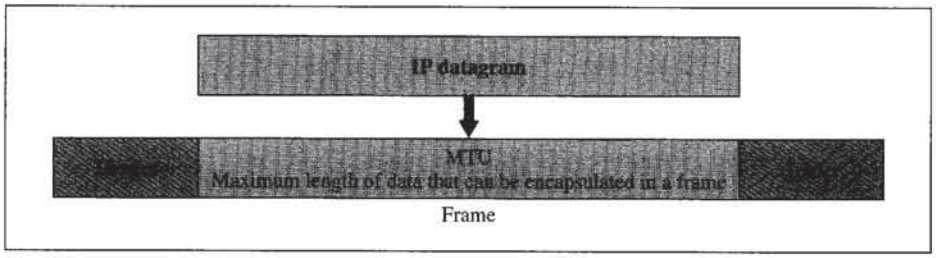
A datagram can travel through different networks. Each router decapsulates the IP datagram from the frame it receives, processes it, and then encapsulates it in another frame. The format and size of the received frame depend on the protocol used by the physical network through which the frame has just traveled. The format and size of the sent frame depend on the protocol used by the physical network through which the frame is going to travel. For example, if a router connects an Ethernet network to a Token Ring network, it receives a frame in the Ethernet format and sends a frame in the Token Ring format.

### Maximum Transfer Unit (MTU)

Each data link layer protocol has its own frame format. One of the fields defined in the format is the maximum size of the data field. In other words, when a datagram is

encapsulated in a frame, the total size of the datagram must be less than this maximum size, which is defined by the restriction imposed by the hardware and software used in the network (see Figure 8.6).

**Figure 8.6** *MTU*




The value of the MTU differs from one physical network protocol to another. Table 8.5 shows the values for different protocols.

**Table 8.5** *MTUs for different networks*

| *Protocol* | *MTU* |
|---|---|
| Hyperchannel | 65,535 |
| Token Ring (16 Mbps) | 17,914 |
| Token Ring (4 Mbps) | 4,464 |
| FDDI | 4,352 |
| Ethernet | 1,500 |
| X.25 | 576 |
| PPP | 296 |

In order to make the IP protocol independent of the physical network, the packagers decided to make the maximum length of the IP datagram equal to the largest maximum transfer unit (MTU) defined so far (65,535 bytes). This makes transmission more efficient if we use a protocol with an MTU of this size. However, for other physical networks, we must divide the datagram to make it possible to pass through these networks. This is called **fragmentation.**

When a datagram is fragmented, each fragment has its own header with most of the fields repeated, but some changed. A fragmented datagram may itself be fragmented if it encounters a network with an even smaller MTU. In other words, a datagram can be fragmented several times before it reaches the final destination.

A datagram can be fragmented by the source host or any router in the path. The reassembly of the datagram, however, is done only by the destination host because each fragment becomes an independent datagram. Whereas the fragmented datagram can travel through different routes, and we can never control or guarantee which route a fragmented datagram may take, all of the fragments belonging to the same datagram

should finally arrive at the destination host. So it is logical to do the reassembly at the final destination.

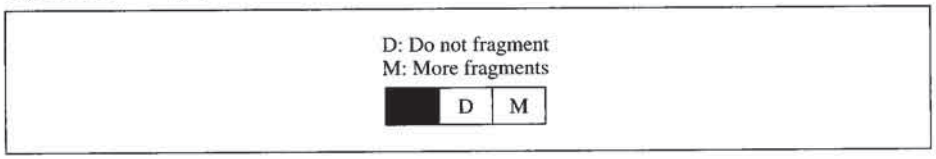
When a datagram is fragmented, required parts of the header must be copied by all fragments. The option field may or may not be copied as we will see in the next section. The host or router that fragments a datagram must change the values of three fields: flags, fragmentation offset, and total length. The rest of the fields must be copied. Of course, the value of the checksum must be recalculated regardless of fragmentation.

## Fields Related to Fragmentation

The fields that are related to fragmentation and reassembly of an IP datagram are the identification, flags, and fragmentation offset fields.

- **Identification.** This 16-bit field identifies a datagram originating from the source host. The combination of the identification and source IP address must uniquely define a datagram as it leaves the source host. To guarantee uniqueness, the IP protocol uses a counter to label the datagrams. The counter is initialized to a positive number. When the IP protocol sends a datagram, it copies the current value of the counter to the identification field and increments the counter by one. As long as the counter is kept in the main memory, uniqueness is guaranteed. When a datagram is fragmented, the value in the identification field is copied into all fragments. In other words, all fragments have the same identification number, which is also the same as the original datagram. The identification number helps the destination in reassembling the datagram. It knows that all fragments having the same identification value should be assembled into one datagram.
- **Flags.** This is a three-bit field. The first bit is reserved. The second bit is called the *do not fragment* bit. If its value is 1, the machine must not fragment the datagram. If it cannot pass the datagram through any available physical network, it discards the datagram and sends an ICMP error message to the source host (see Chapter 9). If its value is 0, the datagram can be fragmented if necessary. The third bit is called the *more fragment* bit. If its value is 1, it means the datagram is not the last fragment; there are more fragments after this one. If its value is 0, it means this is the last or only fragment (see Figure 8.7).

**Figure 8.7** *Flags field*




- **Fragmentation offset.** This 13-bit field shows the relative position of this fragment with respect to the whole datagram. It is the offset of the data in the original datagram measured in units of 8 bytes. Figure 8.8 shows a datagram with a data size of 4000 bytes fragmented into three fragments. The bytes in the original datagram are numbered 0 to 3999. The first fragment carries bytes 0 to 1399. The offset

for this datagram is 0/8 = 0. The second fragment carries bytes 1400 to 2799; the offset value for this fragment is 1400/8 = 175. Finally, the third fragment carries bytes 2800 to 3999. The offset value for this fragment is 2800/8 = 350.

Remember that the value of the offset is measured in units of 8 bytes. This is done because the length of the offset field is only 13 bits long and cannot represent a sequence of bytes greater than 8191. This forces hosts or routers that fragment datagrams to choose the size of each fragment so that the first byte number is divisible by 8.

**Figure 8.8** *Fragmentation example*

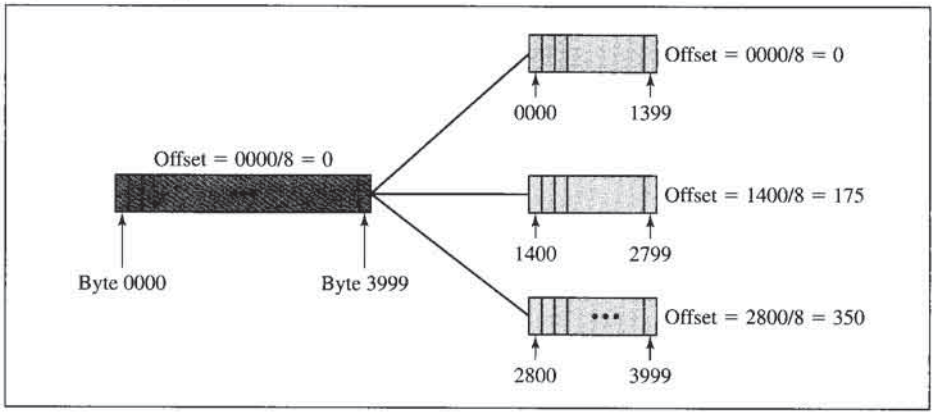


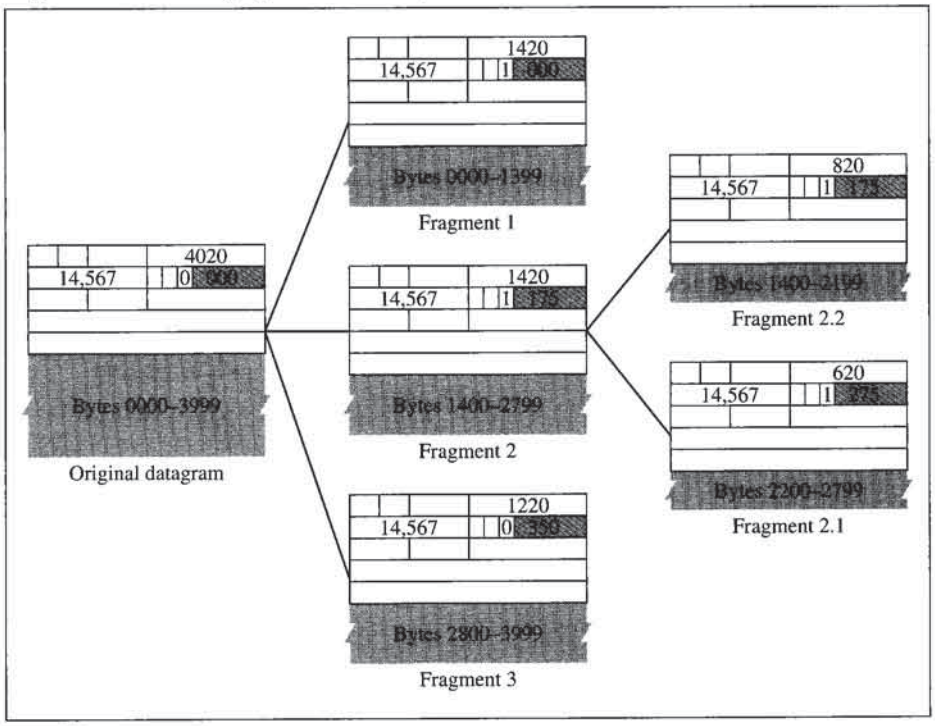

Figure 8.9 shows an expanded view of the fragments in the previous figure. Notice the value of the identification field is the same in all fragments. Notice the value of the flags field with the *more* bit set for all fragments except the last. Also, the value of the offset field for each fragment is shown.

The figure also shows what happens if a fragment itself is fragmented. In this case the value of the offset field is always relative to the original datagram. For example, in the figure, the second fragment is itself fragmented later to two fragments of 800 bytes and 600 bytes, but the offset shows the relative position of the fragments to the original data.

It is obvious that even if each fragment follows a different path and arrives out of order, the final destination host can reassemble the original datagram from the fragments received (if none of them is lost) using the following strategy:

a. The first fragment has an offset field value of zero.
b. Divide the length of the first fragment by 8. The second fragment has an offset value equal to that result.
c. Divide the total length of the first and second fragment by 8. The third fragment has an offset value equal to that result.
d. Continue the process. The last fragment has a *more* bit value of 0.

**Figure 8.9** *Detailed fragmentation example*




### Example 5

A packet has arrived with an *M* bit value of 0. Is this the first fragment, the last fragment, or a middle fragment? Do we know if the packet was fragmented?

### Solution

If the *M* bit is 0, it means that there are no more fragments; the fragment is the last one. However, we cannot say if the original packet was fragmented or not. A nonfragmented packet is considered the last fragment.

### Example 6

A packet has arrived with an *M* bit value of 1. Is this the first fragment, the last fragment, or a middle fragment? Do we know if the packet was fragmented?

### Solution

If the *M* bit is 1, it means that there is at least one more fragment. This fragment can be the first one or a middle one, but not the last one. We don't know if it is the first one or a middle one; we need more information (the value of the fragmentation offset). However, we can definitely say the original packet has been fragmented because the *M* bit value is 1.

### Example 7

A packet has arrived with an *M* bit value of 1 and a fragmentation offset value of zero. Is this the first fragment, the last fragment, or a middle fragment?

**Solution**

Because the *M* bit is 1, it is either the first fragment or a middle one. Because the offset value is 0, it is the first fragment.

**Example 8**

A packet has arrived in which the offset value is 100. What is the number of the first byte? Do we know the number of the last byte?

**Solution**

To find the number of the first byte, we multiply the offset value by 8. This means that the first byte number is 800. We cannot determine the number of the last byte unless we know the length of the data.

**Example 9**

A packet has arrived in which the offset value is 100, the value of HLEN is 5 and the value of the total length field is 100. What is the number of the first byte and the last byte?

**Solution**

The first byte number is $100 \times 8 = 800$. The total length is 100 bytes and the header length is 20 bytes ($5 \times 4$), which means that there are 80 bytes in this datagram. If the first byte number is 800, the last byte number must 879.

## 8.3 OPTIONS

The header of the IP datagram is made of two parts: a fixed part and a variable part. The fixed part is 20 bytes long and was discussed in the previous section. The variable part comprises the options that can be a maximum of 40 bytes.

Options, as the name implies, are not required for every datagram. They are used for network testing and debugging. Although options are not a required part of the IP header, option processing is required of the IP software. This means that all standards must be able to handle options if they are present in the header.

### Format

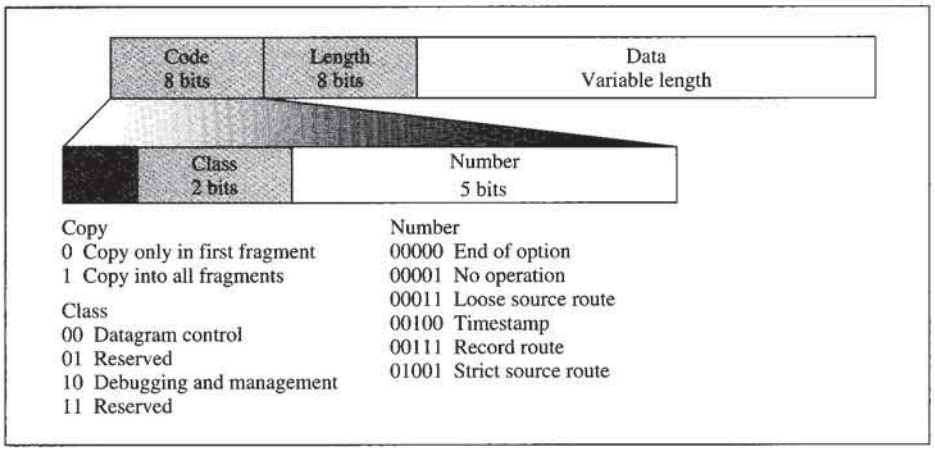
Figure 8.10 shows the format of an option. It is composed of a 1-byte code field, a 1-byte length field, and a variable-sized data field.

#### Code

The **code field** is 8 bits long and contains three subfields: copy, class, and number.

- **Copy.** This 1-bit subfield controls the presence of the option in fragmentation. When its value is 0, it means that the option must be copied only to the first fragment. If its value is 1, it means the option must be copied to all fragments.
- **Class.** This 2-bit subfield defines the general purpose of the option. When its value is 00, it means that the option is used for datagram control. When its value is 10, it means that the option is used for debugging and management. The other two possible values (01 and 11) have not yet been defined.

**Figure 8.10** *Option format*




- **Number.** This 5-bit subfield defines the type of the option. Although 5 bits can define up to 32 different types, currently only 6 types are in use. These will be discussed in a later section.

### Length

The **length field** defines the total length of the option including the code field and the length field itself. This field is not present in all of the option types.

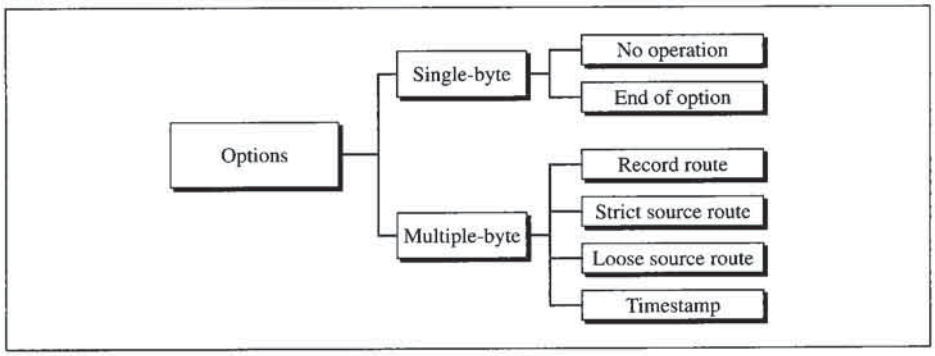
### Data

The **data field** contains the data that specific options require. Like the length field, this field is also not present in all option types.

## Option Types

As mentioned previously, only six options are currently being used. Two of these are 1-byte options, and they do not require the length or the data fields. Four of them are multiple-byte options; they require the length and the data fields (see Figure 8.11).

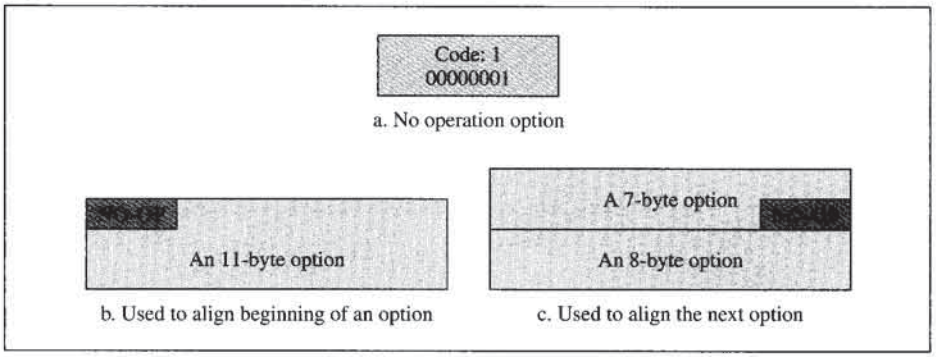
**Figure 8.11** *Categories of options*



### No Operation

A **no operation** option is a 1-byte option used as a filler between options. For example, it can be used to align the next option on a 16-bit or 32-bit boundary (see Figure 8.12).

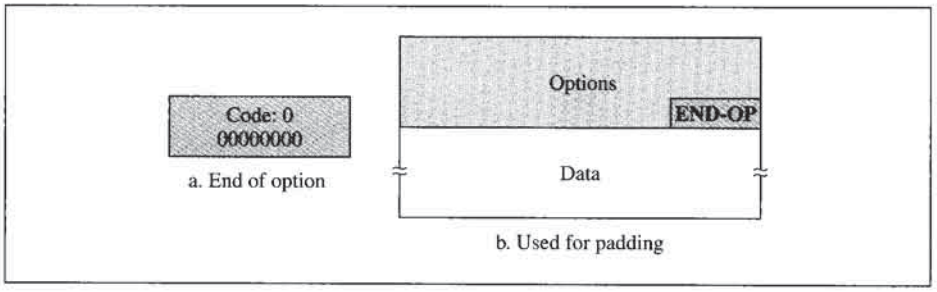
**Figure 8.12** *No operation option*




### End of Option

An **end of option** option is also a 1-byte option used for padding at the end of the option field. It, however, can only be used as the last option. Only one *end of option* option can be used. After this option, the receiver looks for the payload data. This means that if more than 1 byte is needed to align the option field, some no operation options must be used followed by an end of option option (see Figure 8.13).

**Figure 8.13** *End of option option*

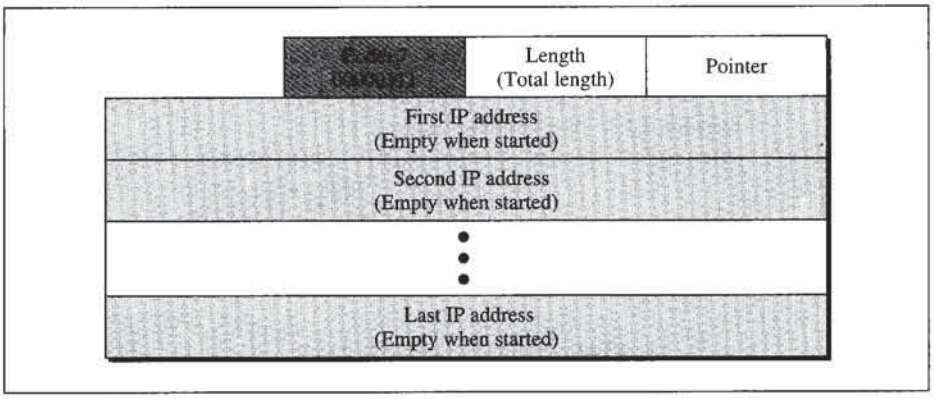



### Record Route

A **record route** option is used to record the internet routers that handle the datagram. It can list up to nine router IP addresses since the maximum size of the header is 60 bytes, which must include 20 bytes for the base header. This implies that only 40 bytes are left over for the option part. The source creates placeholder fields in the option to be filled by the visited routers. Figure 8.14 shows the format of the record route option.

Both the code and length fields have been described above. The **pointer field** is an offset integer field containing the byte number of the first empty entry. In other words, it points to the first available entry.

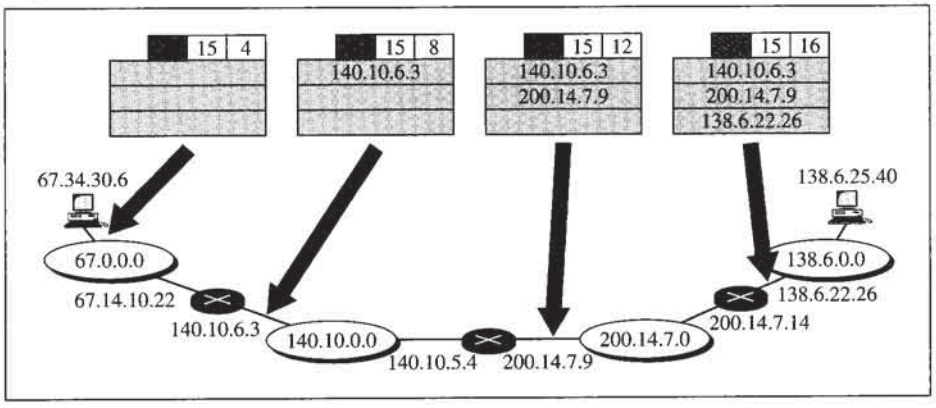
**Figure 8.14** *Record route option*




The source creates empty fields for the IP addresses in the data field of the option. When the datagram leaves the source, all of the fields are empty. The pointer field has a value of 4, pointing to the first empty field.

When the datagram is traveling, each router that processes the datagram compares the value of the pointer with the value of the length. If the value of the pointer is greater than the value of the length, the option is full and no changes are made. However, if the value of the pointer is not greater than the value of the length, the router inserts its outgoing IP address in the next empty field (remember that a router has more than one IP address). In this case, the router adds the IP address of its interface from which the datagram is leaving. The router then increments the value of the pointer by 4. Figure 8.15 shows the entries as the datagram travels left to right from router to router.

**Figure 8.15** *Record route concept*



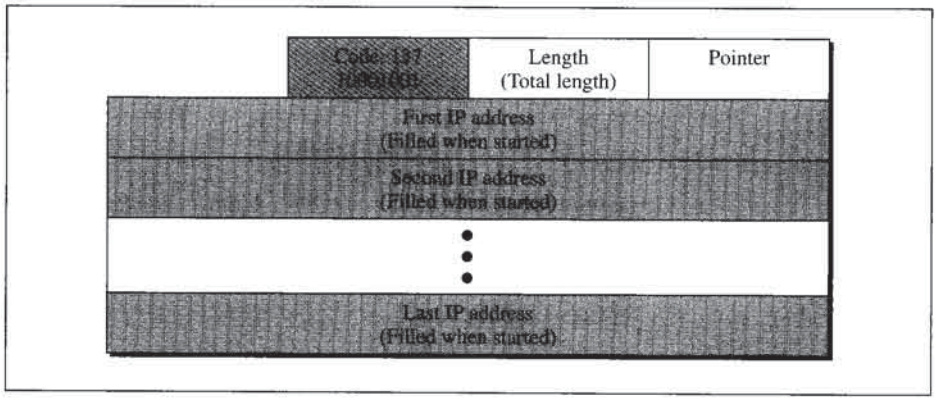
### Strict Source Route

A **strict source route** option is used by the source to predetermine a route for the datagram as it travels through the Internet. Dictation of a route by the source can be useful for several purposes. The sender can choose a route with a specific type of service, such as minimum delay or maximum throughput. Alternatively, it may choose a route that is safer or more reliable for the sender's purpose. For example, a sender can choose a route so that its datagram does not travel through a competitor's network.

If a datagram specifies a strict source route, all of the routers defined in the option must be visited by the datagram. A router must not be visited if its IP address is not listed in the datagram. If the datagram visits a router that is not on the list, the datagram is discarded and an error message is issued. If the datagram arrives at the destination and some of the entries were not visited, it will also be discarded and an error message issued.

Nonprivileged users of the Internet, however, are not usually aware of the physical topology of the Internet. Consequently, strict source routing is not the choice of most users. Figure 8.16 shows the format of the strict source route option.

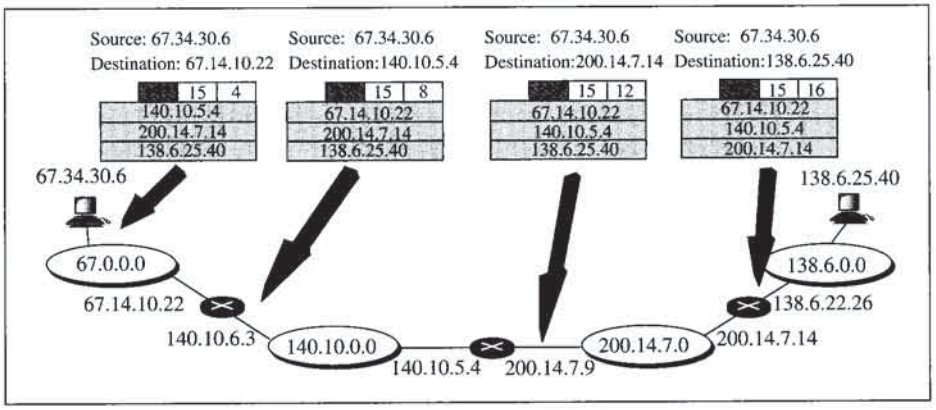
**Figure 8.16** *Strict source route option*




The format is similar to the record route option with the exception that all of the IP addresses are entered by the sender.

When the datagram is traveling, each router that processes the datagram compares the value of the pointer with the value of the length. If the value of the pointer is greater than the value of the length, the datagram has visited all of the predefined routers. The datagram cannot travel anymore; it is discarded and an error message is created. If the value of the pointer is not greater than the value of the length, the router compares the IP address pointed by the pointer with its incoming IP address: If they are equal, it processes the datagram, overwrites the current IP address with its outgoing IP address, increments the pointer value by 4, and forwards the datagram. If they are not equal, it discards the datagram and issues an error message. Figure 8.17 shows the actions taken by each router as a datagram travels from source to destination.

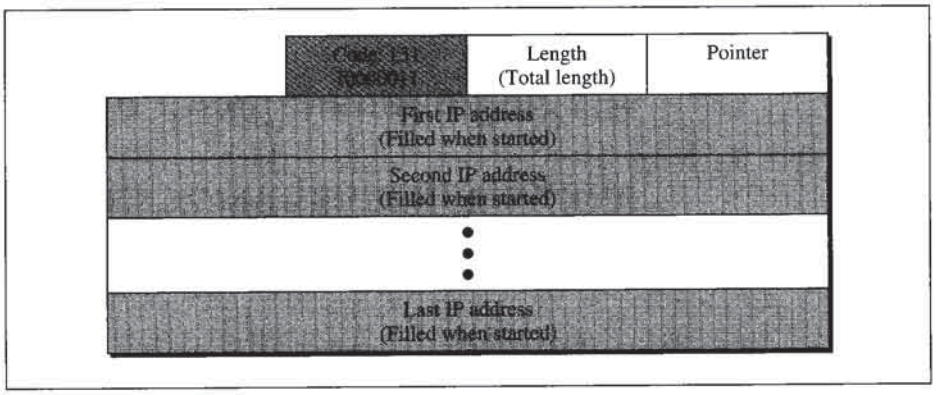
**Figure 8.17** *Strict source route concept*




### Loose Source Route

A **loose source route** option is similar to the strict source route, but it is more relaxed. Each router in the list must be visited, but the datagram can visit other routers as well. Figure 8.18 shows the format of the loose source route option.

**Figure 8.18** *Loose source route option*

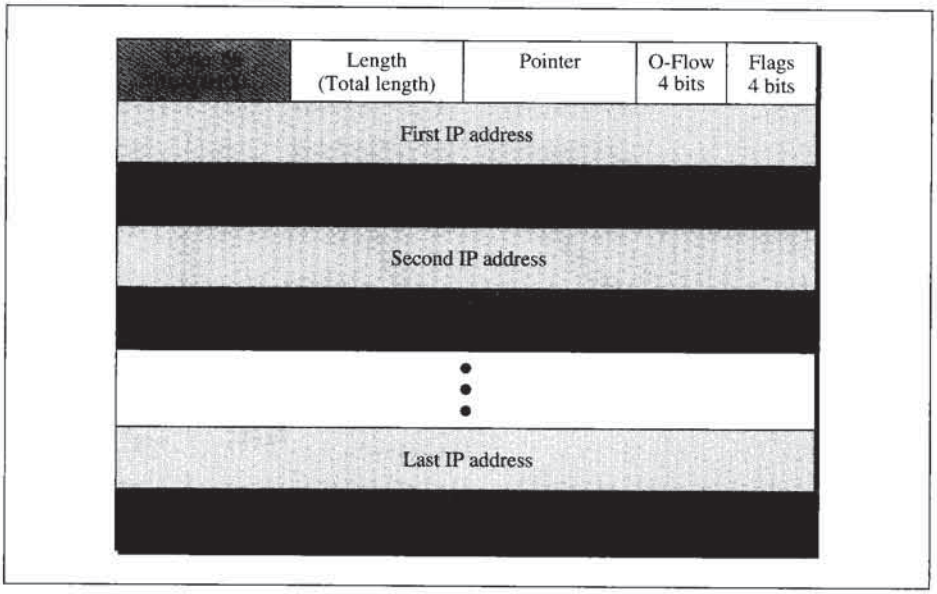



### Timestamp

A **timestamp** option is used to record the time of datagram processing by a router. The time is expressed in milliseconds from midnight, Universal Time. Knowing the time a datagram is processed can help users and managers track the behavior of the routers in the Internet. We can estimate the time it takes for a datagram to go from one router to another. We say *estimate* because, although all routers may use Universal Time, their local clocks may not be synchronized.

However, nonprivileged users of the Internet are not usually aware of the physical topology of the Internet. Consequently, a timestamp option is not a choice for most users. Figure 8.19 shows the format of the timestamp option.

**Figure 8.19** *Timestamp option*




In this figure, the definition of the code and length fields are the same as before. The overflow field records the number of routers that could not add their timestamp because no more fields were available. The flags field specifies the visited router responsibilities. If the flag value is 0, each router adds only the timestamp in the provided field. If the flag value is 1, each router must add its outgoing IP address and the timestamp. If the value is 3, the IP addresses are given, and each router must check the given IP address with its own incoming IP address. If there is a match, the router overwrites the IP address with its outgoing IP address and adds the timestamp (see Figure 8.20).

**Figure 8.20** *Use of flag in timestamp*

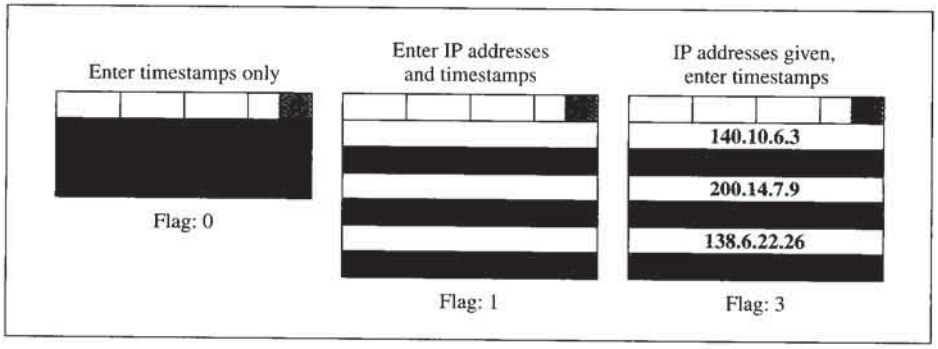
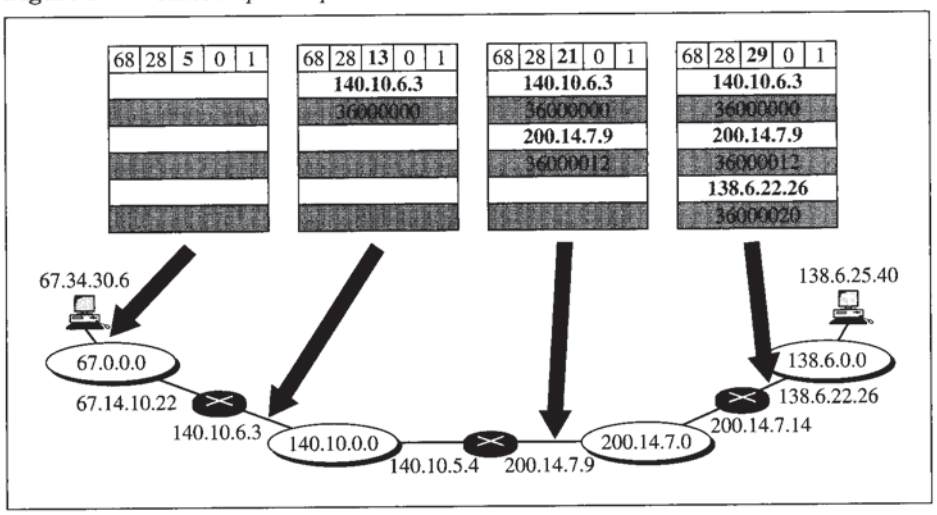



Figure 8.21 shows the actions taken by each router when a datagram travels from source to destination. The figure assumes the flag value of 1.

**Figure 8.21** *Timestamp concept*




### Example 10

Which of the six options must be copied to each fragment?

### Solution

We look at the first (left-most) bit of the code for each option.

a. No operation: Code is **0**0000001; no copy.
b. End of option: Code is **0**0000000; no copy.
c. Record route: Code is **0**0000111; no copy.
d. Strict source route: Code is **1**0001001; it is copied to each fragment.
e. Loose source route: Code is **1**0000011; it is copied to each fragment.
f. Timestamp: Code is **0**1000100; no copy.

### Example 11

Which of the six options are used for datagram control and which are used for debugging and management?

### Solution

We look at the second and third (left-most) bits of the code.

a. No operation: Code is 0**0**000001; datagram control.
b. End of option: Code is 0**0**000000; datagram control.
c. Record route: Code is 0**0**000111; datagram control.
d. Strict source route: Code is 1**0**001001; datagram control.
e. Loose source route: Code is 1**0**000011; datagram control.
f. Time stamp: Code is 0**1**000100; debugging and management control.

## 8.4 CHECKSUM

The error detection method used by most TCP/IP protocols is called the **checksum.** The checksum protects against the corruption that may occur during the transmission of a packet. It is redundant information added to the packet.

The checksum is calculated at the sender and the value obtained is sent with the packet. The receiver repeats the same calculation on the whole packet including the checksum. If the result is satisfactory (see below), the packet is accepted; otherwise, it is rejected.

### Checksum Calculation at the Sender

In the sender, the packet is divided into $n$-bit sections ($n$ is usually 16). These sections are added together using one's complement arithmetic (see Appendix C) such that the sum is also $n$ bits long. The sum is then complemented (all 0s changed to 1s and all 1s to 0s) to produce the checksum.

To create the checksum the sender does the following:

- The packet is divided into $k$ sections, each of $n$ bits.
- All sections are added together using one's complement arithmetic.
- The final result is complemented to make the checksum.

### Checksum Calculation at the Receiver

The receiver divides the received packet into $k$ sections and adds all sections. It then complements the result. If the final result is 0, the packet is accepted; otherwise, it is rejected.

Figure 8.22 shows graphically what happens at the sender and the receiver.

We said when the receiver adds all of the sections and complements the result, it should get zero if there is no error in the data during transmission or processing. This is true because of the rules in one's complement arithmetic.

Assume that we get a number called $T$ when we add all the sections in the sender. When we complement the number in one's complement arithmetic, we get the negative of the number. This means that if the sum of all sections is $T$, the checksum is $-T$.

When the receiver receives the packet, it adds all the sections. It adds $T$ and $-T$ which, in one's complement, is $-0$ (minus zero). When the result is complemented, $-0$ becomes 0. Thus if the final result is 0, the packet is accepted; otherwise, it is rejected (see Figure 8.23).

### Checksum in the IP Packet

The implementation of the checksum in the IP packet follows the same principle discussed above. First, the value of the checksum field is set to 0. Then, the entire header

**Figure 8.22** *Checksum concept*

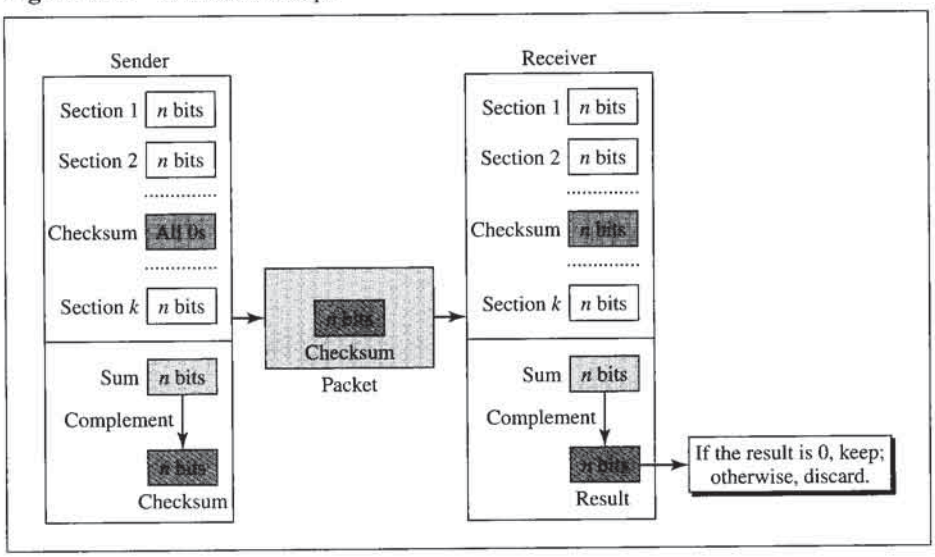
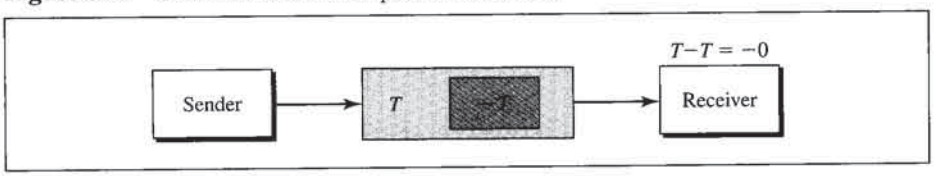



**Figure 8.23** *Checksum in one's complement arithmetic*



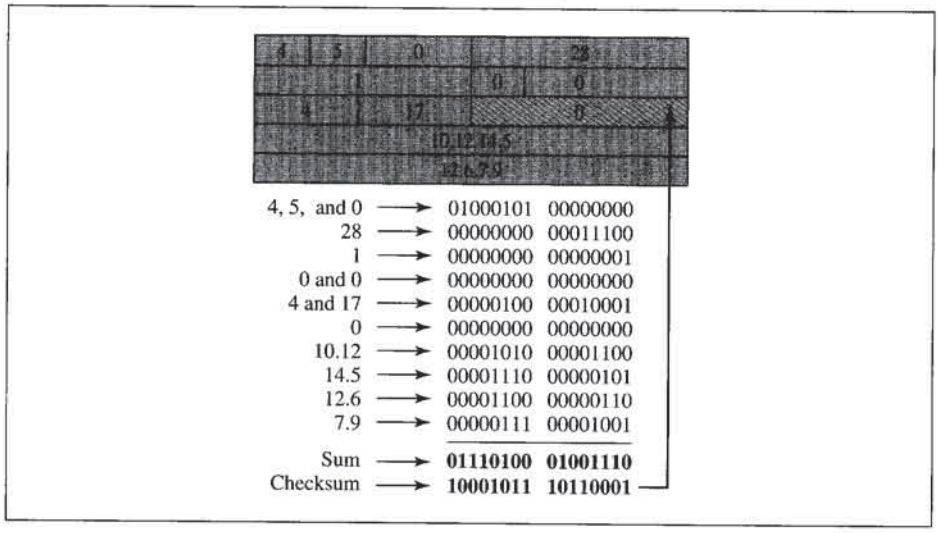

is divided into 16-bit sections and added together. The result (sum) is complemented and inserted into the checksum field.

The checksum in the IP packet covers only the header, not the data. There are two good reasons for this. First, all higher-level protocols that encapsulate data in the IP datagram have a checksum field that covers the whole packet. Therefore, the checksum for the IP datagram does not have to check the encapsulated data. Second, the header of the IP packet changes with each visited router, but the data does not. So the checksum includes only the part that has changed. If the data is included, each router must recalculate the checksum for the whole packet, which means increased processing time for each router.

### Example 12

Figure 8.24 shows an example of a checksum calculation for an IP header without options. The header is divided into 16-bit sections. All the sections are added and the sum is complemented. The result is inserted in the checksum field.

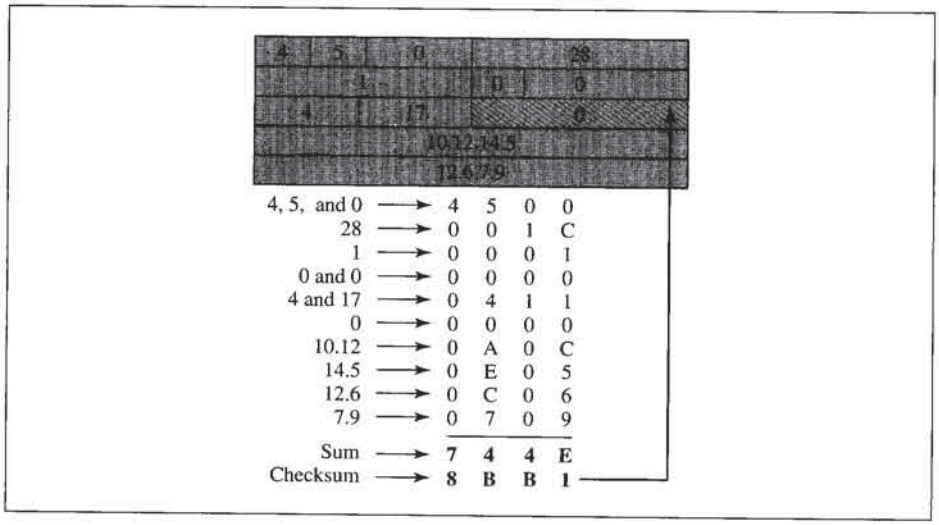
**Figure 8.24** *Example of checksum calculation in binary*




### Example 13

Let us do the same example in hexadecimal. Each row has four hexadecimal digits. We calculate the sum first. Note that if an addition results in more than one hexadecimal digit, the right-most digit becomes the current-column digit and the rest are carried to other columns. From the sum, we make the checksum by complementing the sum. However, note that we subtract each digit from 15 in hexadecimal arithmetic (just as we subtract from 1 in binary arithmetic). This means the complement of E (14) is 1 and the complement of 4 is B (11). Figure 8.25 shows the calculation. Note that the result (8BB1) is exactly the same as in Example 11.

**Figure 8.25** *Example of checksum calculation in hexadecimal*

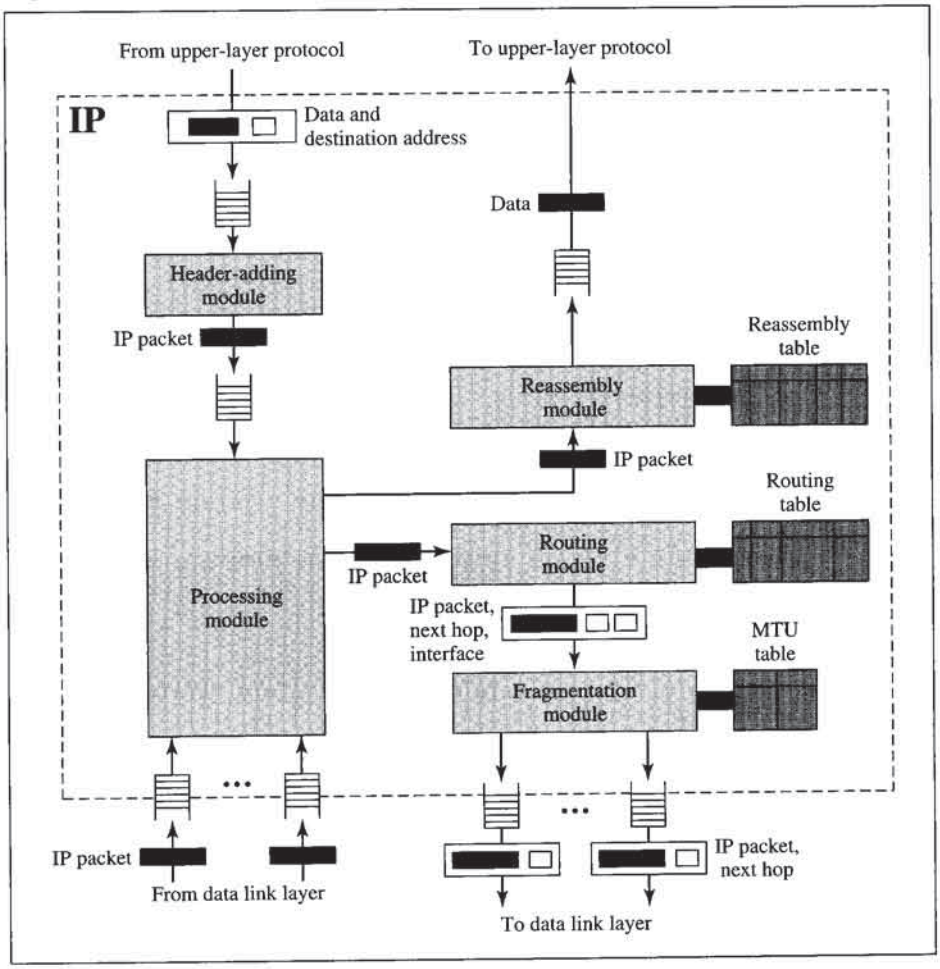


Check Appendix C for a detailed description of checksum calculation and the handling of carries.

## 8.5 IP PACKAGE

In this section, we present a simplified example of a hypothetical IP package. Our purpose is to show the relationships between the different concepts discussed in this chapter. Figure 8.26 shows these eight components and their interactions.

**Figure 8.26** *IP components*



Although IP supports several options, we have omitted option processing in our package to make it easier to understand at this level. In addition, we have sacrificed efficiency for the sake of simplicity.

We can say that the IP package involves eight components: a header-adding module, a processing module, a routing module, a fragmentation module, a reassembly module, a routing table, an MTU table, and a reassembly table. In addition, the package includes input and output queues.

The package receives a packet, either from the data link layer or from a higher-level protocol. If the packet comes from an upper-layer protocol, it is delivered to the data link layer for transmission (unless it has a loopback address of 127.X.Y.Z). If the packet comes from the data link layer, it is either delivered to the data link layer for forwarding (in a router) or it is delivered to a higher-layer protocol if the destination IP address of the packet is the same as the station address.

## Header-Adding Module

The **header-adding module** receives data from an upper-layer protocol along with the destination IP address. It encapsulates the data in an IP datagram by adding the IP header.

| *Header-Adding Module* |
|---|
| Receive: data, destination address |
| 1. Encapsulate the data in an IP datagram. |
| 2. Calculate the checksum and insert it in the checksum field. |
| 3. Send the data to the corresponding input queue. |
| 4. Return. |

## Processing Module

The **processing module** is the heart of the IP package. In our package, the processing module receives a datagram from an interface or from the header-adding module. It treats both cases the same. A datagram must be processed and routed regardless of where it comes from.

The processing module first checks to see if the datagram is a loopback packet (with the destination address of 127.X.Y.Z) or a packet that has reached its final destination. In either case, the packet is sent to the reassembly module.

If the node is a router, it decrements the time-to-live (TTL) field by one. If this value is less than or equal to zero, the datagram is discarded and an ICMP message (see Chapter 9) is sent to the original sender. If the value of TTL is greater than zero after decrement, the processing module sends the datagram to the routing module (see Chapter 8).

*Processing Module*

1. Remove one datagram from one of the input queues.
2. If (destination address is 127.X.Y.Z or matches one of the local addresses)
    1. Send the datagram to the reassembly module.
    2. Return.
3. If (machine is a router)
    1. Decrement TTL.
4. If (TTL less than or equal to zero)
    1. Discard the datagram.
    2. Send an ICMP error message.
    3. Return.
5. Send the datagram to the routing module.
6. Return.

## Queues

Our package uses two types of queues: input queues and output queues. The **input queues** store the datagrams coming from the data link layer or the upper-layer protocols. The **output queues** store the datagrams going to the data link layer or the upper-layer protocols. The processing module dequeues (removes) the datagrams from the input queues. The fragmentation and reassembly modules enqueue (add) the datagrams into the output queues.

## Routing Table

We discussed the routing table in Chapter 6. The routing table is used by the routing module to determine the next-hop address of the packet.

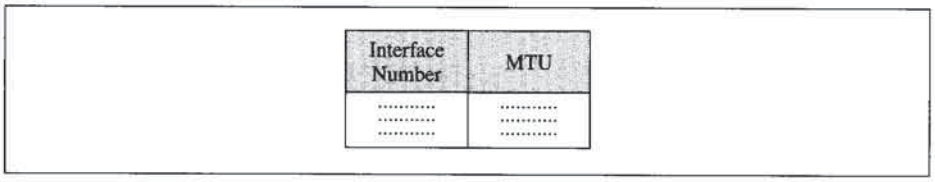
## Routing Module

We discussed the routing module in Chapter 6. The routing module receives an IP packet from the processing module. If the packet is to be forwarded, it is passed to this module. The module finds the IP address of the next station along with the interface number to which the packet should be sent. It then sends the packet with this information to the fragmentation module.

## MTU Table

The MTU table is used by the fragmentation module to find the maximum transfer unit of a particular interface. Figure 8.27 shows the format of an MTU table.

**Figure 8.27** *MTU table*




## Fragmentation Module

In our package, the **fragmentation module** receives an IP datagram from the routing module. The routing module gives the IP datagram, the IP address of the next station (either the final destination in a direct delivery or the next router in an indirect delivery), and the interface number through which the datagram is sent out.

The fragmentation module consults the MTU table to find the MTU for the specific interface number. If the length of the datagram is larger than the MTU, the fragmentation module fragments the datagram, adds a header to each fragment, and sends them to the ARP package (see Chapter 7) for address resolution and delivery.

*Fragmentation Module*

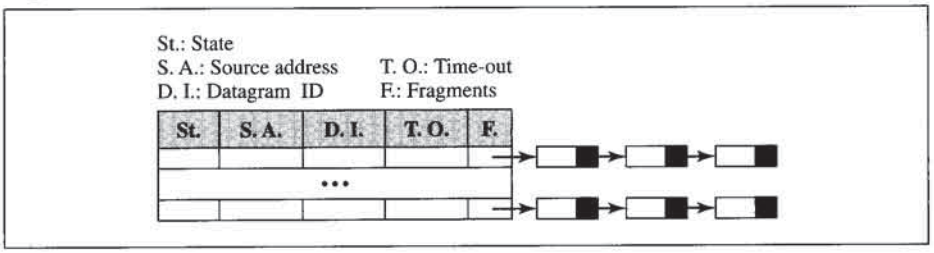
Receive: an IP packet from routing module

1. Extract the size of the datagram.
2. If (size > MTU of the corresponding network)
    1. If [D (*do not fragment*) bit is set]
        1. Discard the datagram.
        2. Send an ICMP error message (see Chapter 9).
        3. Return.
    2. Else
        1. Calculate the maximum size.
        2. Divide the datagram into fragments.
        3. Add header to each fragment.
        4. Add required options to each fragment.
        5. Send the datagrams.
        6. Return.
3. Else
    1. Send the datagram.
4. Return.

## Reassembly Table

The **reassembly table** is used by the reassembly module. In our package, the reassembly table has five fields: state, source IP address, datagram ID, time-out, and fragments (see Figure 8.28).

**Figure 8.28** *Reassembly table*




The value of the state field can be either FREE or IN-USE. The IP address field defines the source IP address of the datagram. The datagram ID is a number that uniquely defines a datagram and all of the fragments belonging to that datagram. The time-out is a predetermined amount of time in which all fragments must arrive. Finally, the fragments field is a pointer to a linked list of fragments.

## Reassembly Module

The **reassembly module** receives, from the processing module, those datagram fragments that have arrived at their final destinations. In our package, the reassembly module treats an unfragmented datagram as a fragment belonging to a datagram with only one fragment.

Because the IP protocol is a connectionless protocol, there is no guarantee that the fragments arrive in order. Besides, the fragments from one datagram can be intermixed with fragments from another datagram. To keep track of these situations, the module uses a reassembly table with associated linked lists, as we described earlier.

The job of the reassembly module is to find the datagram to which a fragment belongs, to order the fragments belonging to the same datagram, and reassemble all fragments of a datagram when all have arrived. If the established time-out has expired and any fragment is missing, the module discards the fragments.

| *Reassembly Module* |
| --- |
| Receive: an IP datagram from the processing module |
| 1. If (offset value is zero and the *M* bit is 0) |
| 1. Send the datagram to the appropriate queue. |
| 2. Return. |
| 2. Search the reassembly table for the corresponding entry. |
| 3. If (not found) |
| 1. Create a new entry. |

*Reassembly Module (continued)*

4. Insert the fragment at the appropriate place in the linked list.
    1. If (all fragments have arrived)
        1. Reassemble the fragments.
        2. Deliver the datagram to the corresponding upper layer protocol.
        3. Return.
    2. Else
        1. Check the time-out.
        2. If (time-out expired)
            1. Discard all fragments.
            2. Send an ICMP error message (see Chapter 9).
5. Return.

## 8.6 KEY TERMS

best-effort delivery
checksum
code field
datagram
destination address
end of option option
fragmentation
fragmentation offset
header
header length
Internet Protocol (IP)
loose source route option
maximum transfer unit (MTU)
no operation option
option
pointer field
precedence
receiver
record route option
router
sender
service type
source address
strict source route option
time to live
timestamp
type of service (TOS)
version