IP v4
Part 2 Lecture

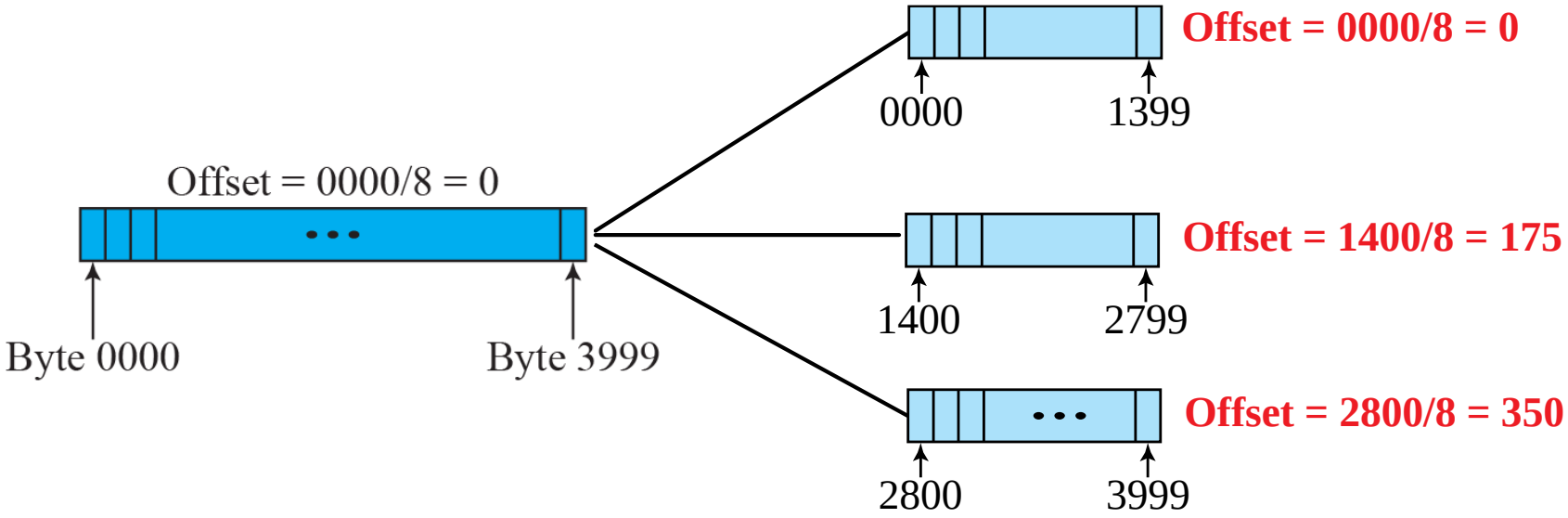
Fields Related to Fragmentation

❑ **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 (not used). The second bit is called the *do not fragment* bit. If its value is 1, the machine must not fragment the datagram. 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.

❑ **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 7.8 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

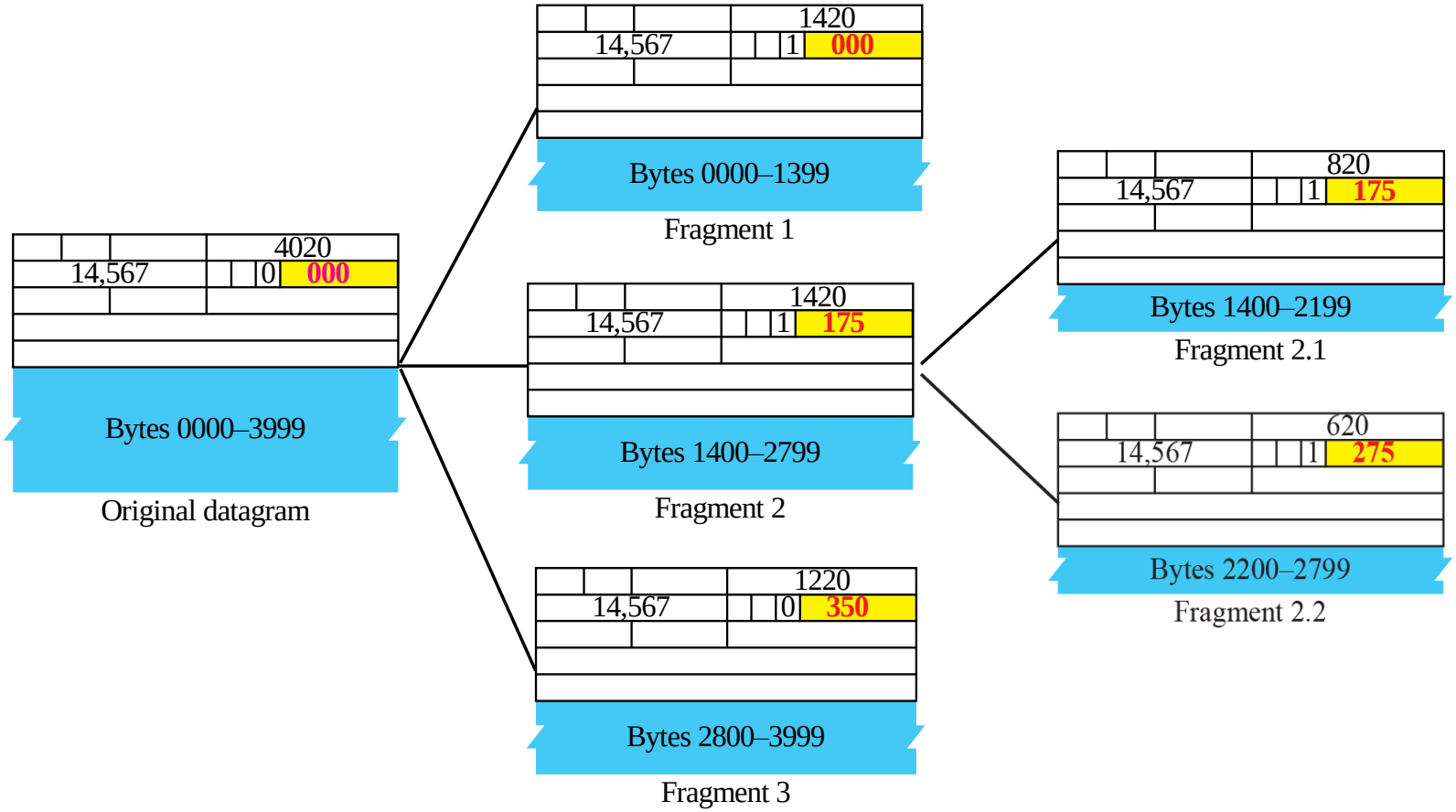
Figure 7.8 ***Fragmentation example***



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 7.9 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.

Figure 7.9 *Detailed fragmentation example*



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.

Example 7.8

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 7.9

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

7-4 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, which can be a maximum of 40 bytes.

Options, as the name implies, are not required for a datagram. They can be 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.

7-5 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.

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.**

Note

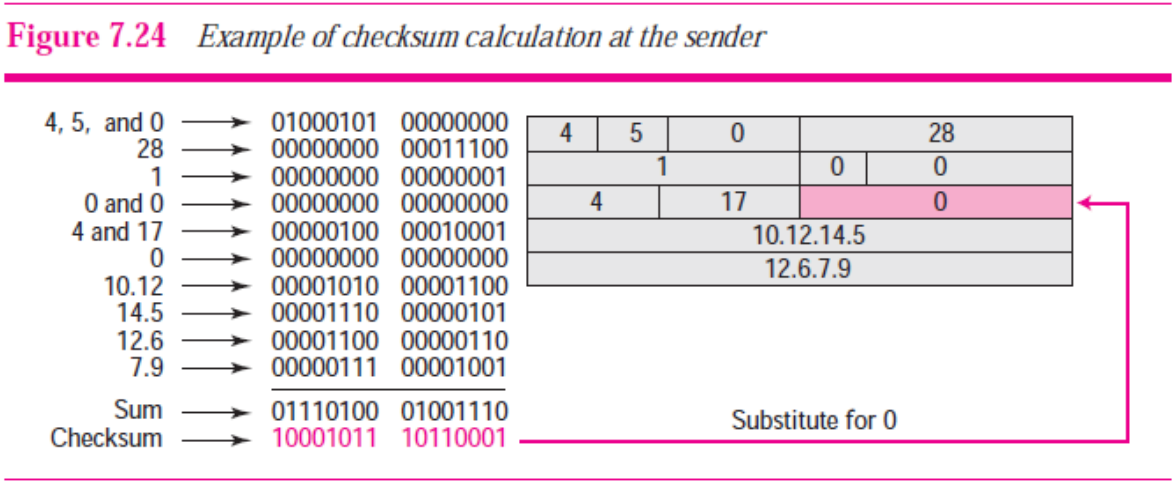
Checksum in IP covers only the header, not the data.

Example 7.17

Figure 7.24 shows an example of a checksum calculation at the sender site 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 7.24 *Example of checksum calculation at the sender side*

Packet contents in hexadecimal: 45 00 00 1C 00 01 00 00 04 17 00 00 0A 0C 0E 05 0C 06 07 09

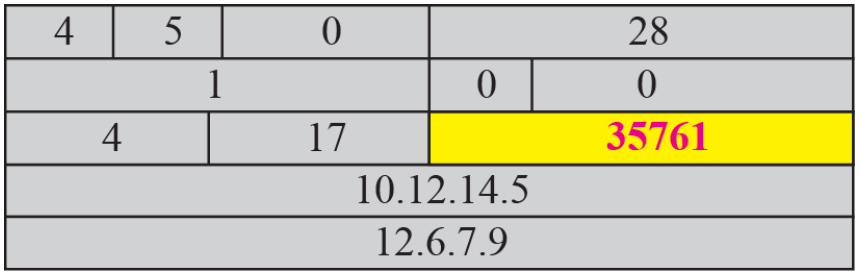


So the checksum value in hexadecimal is 8BB1 or 35761 in decimal

Example 7.18

Figure 7.25 shows the checking of checksum calculation at the receiver site (or intermediate router) assuming that no errors occurred in the header. The header is divided into 16-bit sections. All the sections are added and the sum is complemented. Since the result is 16 0s, the packet is accepted.

Figure 7.25 *Example of checksum calculation at the receiver*



4, 5, and 0	01000101	00000000
28	00000000	00011100
1	00000000	00000001
0 and 0	00000000	00000000
4 and 17	00000100	00010001
Checksum	**10001011**	**10110001**
10.12	00001010	00001100
14.5	00001110	00000101
12.6	00001100	00000110
7.9	00000111	00001001
Sum	**1111 1111**	**1111 1111**
Checksum	**0000 0000**	**0000 0000**

Note

Appendix D gives an algorithm for checksum calculation.

7-8 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.

Topics Discussed in the Section

- ✓ **Header-Adding Module**
- ✓ **Processing Module**
- ✓ **Queues**
- ✓ **Routing Table**
- ✓ **Forwarding Module**
- ✓ **MTU Table**
- ✓ **Fragmentation Module**
- ✓ **Reassembly Table**
- ✓ **Reassembly Module**

Figure 7.29 *IP components*

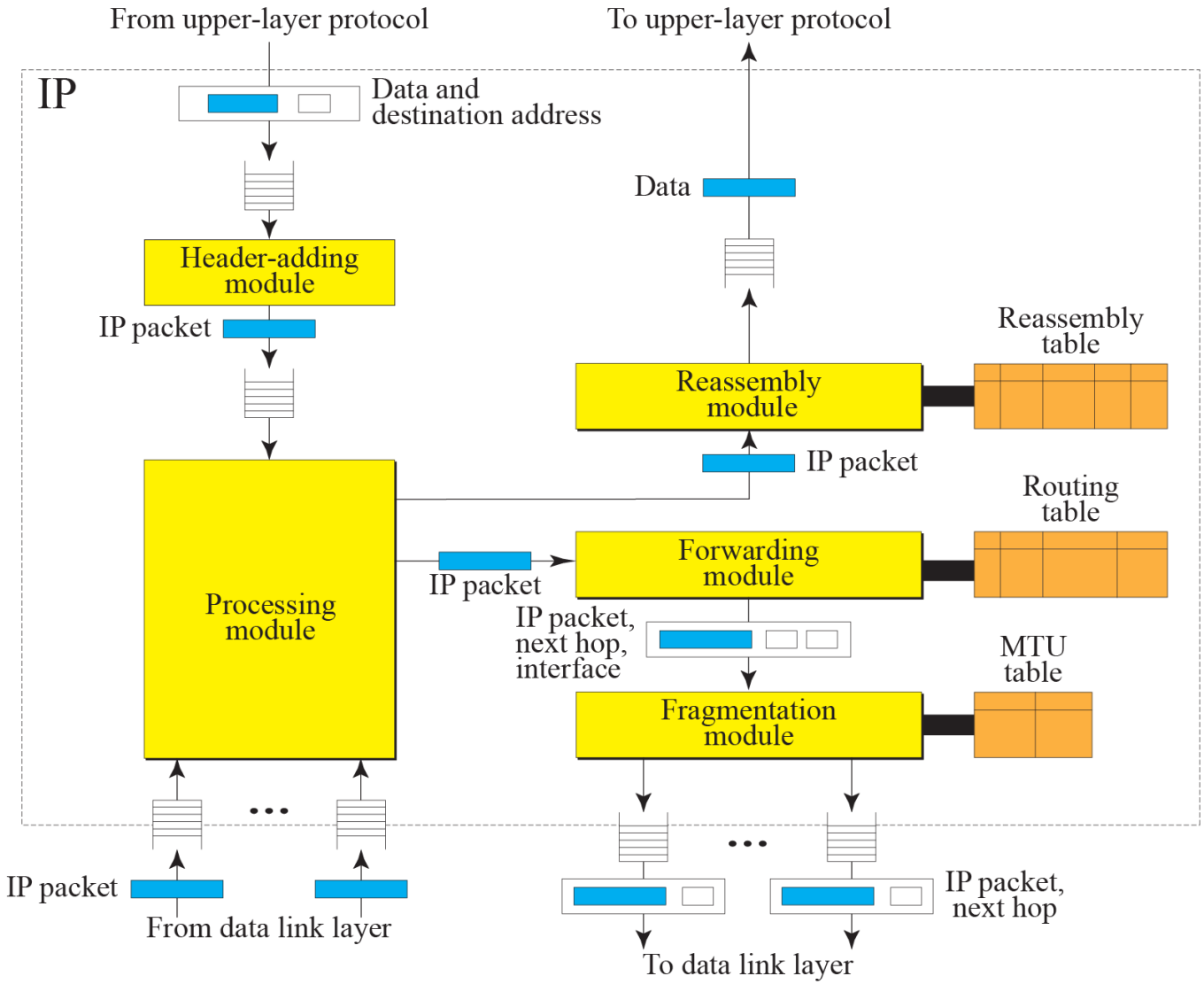


Table 7.3 *Adding module*

```
1  IP_Adding_Module (data, destination_address)
2  {
3       Encapsulate data in an IP datagram
4       Calculate checksum and insert it in the checksum field
5       Send data to the corresponding queue
6       Return
7  }
```

Table 7.4 *Processing module*

```
1   IP_Processing_Module (Datagram)
2   {
3         Remove one datagram from one of the input queues.
4         If (destination address matches a local address)
5         {
6             Send the datagram to the reassembly module.
7             Return.
8         }
9         If (machine is a router)
10        {
11            Decrement TTL.
12        }
13        If (TTL less than or equal to zero)
14        {
15            Discard the datagram.
16            Send an ICMP error message.
17            Return.
18        }
19        Send the datagram to the forwarding module.
20        Return.
21  }
```

Table 7.5 *Fragmentation module*

```
1   IP_Fragmentation_Module (datagram)
2   {
3        Extract the size of datagram
4        If (size > MTU of the corresponding network)
5        {
6             If (D bit is set)
7             {
8                  Discard datagram
9                  Send an ICMP error message
10                 return
11            }
12            Else
13            {
14                 Calculate maximum size
15                 Divide the segment into fragments
16                 Add header to each fragment
17                 Add required options to each fragment
```

Table 7.5 *Fragmentation module (continued)*

```
18                    Send fragment
19                    return
20               }
21          }
22          Else
23          {
24               Send the datagram
25          }
26          Return.
27    }
```

Figure 7.30 *Reassembly table*

St.: State
S. A.: Source address
D. I.: Datagram ID
T. O.: Time-out
F.: Fragments

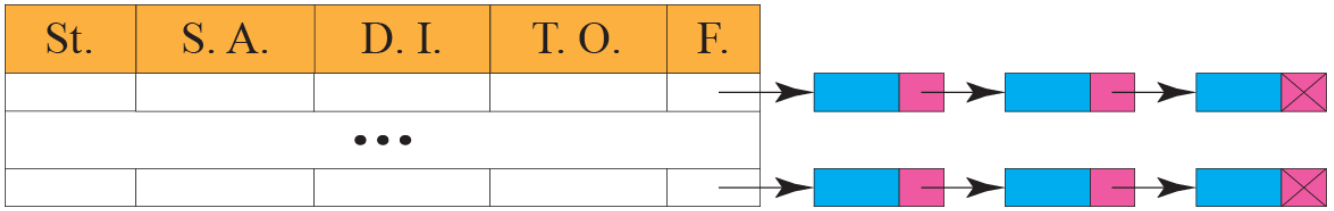


Table 7.6 *Reassembly module*

```
1   IP_Reassembly_Module (datagram)
2   {
3        If (offset value = 0 AND M = 0)
4        {
5             Send datagram to the appropriate queue
6             Return
7        }
8        Search the reassembly table for the entry
9        If (entry not found)
10       {
11            Create a new entry
12       }
13       Insert datagram into the linked list
14       If (all fragments have arrived)
15       {
16            Reassemble the fragment
17            Deliver the fragment to upper-layer protocol
18            return
19       }
20       Else
21       {
22            If (time-out expired)
23            {
24                 Discard all fragments
25                 Send an ICMP error message
26            }
27       }
28       Return.
29  }
```