



Data Communications
and Networking

Fourth Edition

Forouzan

Chapter 11

Data Link Control

11-1 FRAMING

*The data link layer needs to pack bits into **frames**, so that each frame is distinguishable from another. Our postal system practices a type of framing. The simple act of inserting a letter into an envelope separates one piece of information from another; the envelope serves as the delimiter.*

Topics discussed in this section:

Fixed-Size Framing

Variable-Size Framing

Figure 11.1 *A frame in a character-oriented protocol*

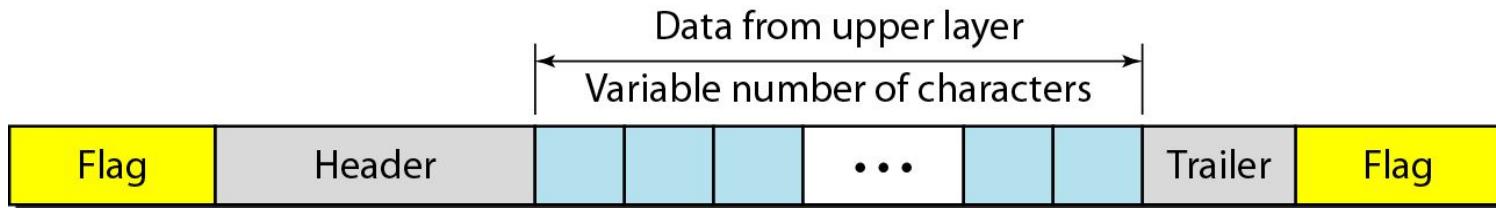
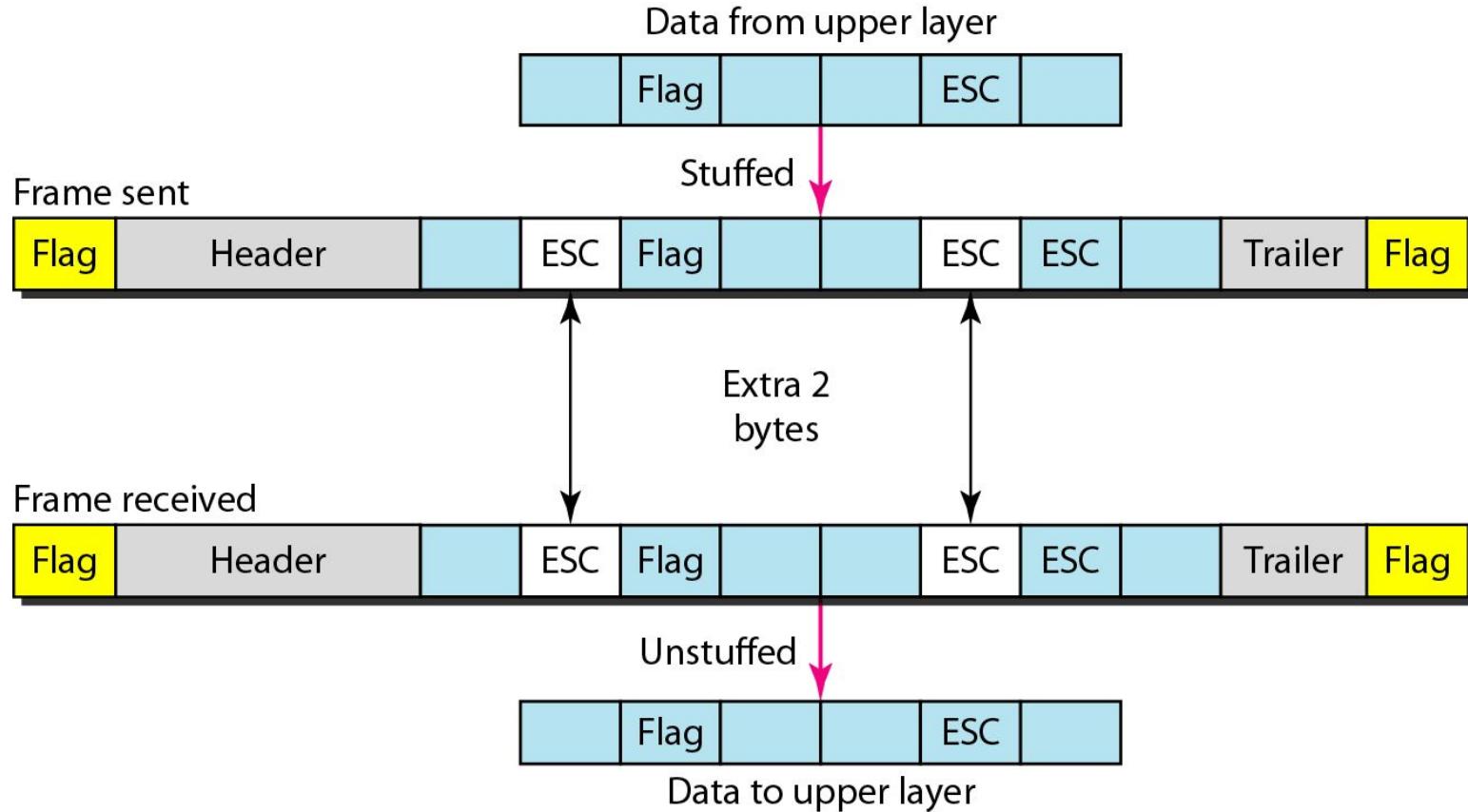
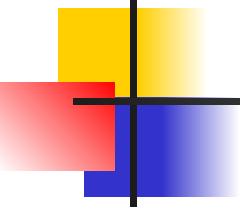


Figure 11.2 *Byte stuffing and unstuffing*

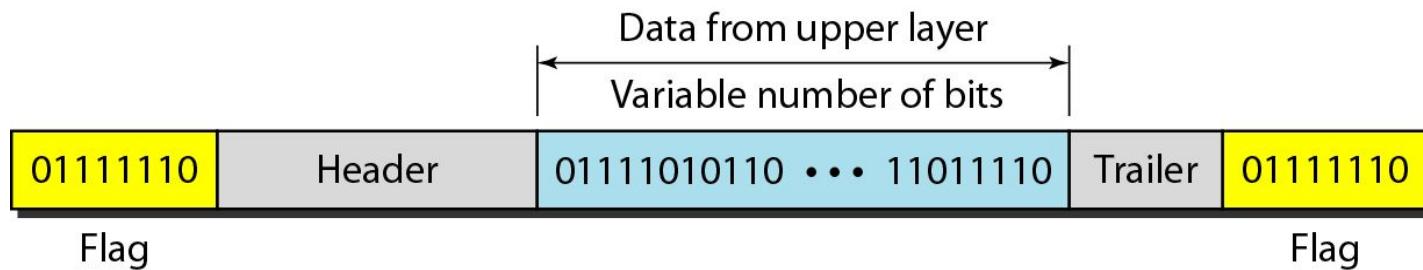


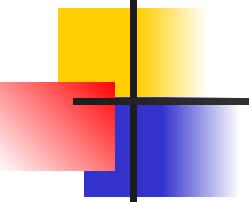


Note

Byte stuffing is the process of adding 1 extra byte whenever there is a flag or escape character in the text.

Figure 11.3 *A frame in a bit-oriented protocol*

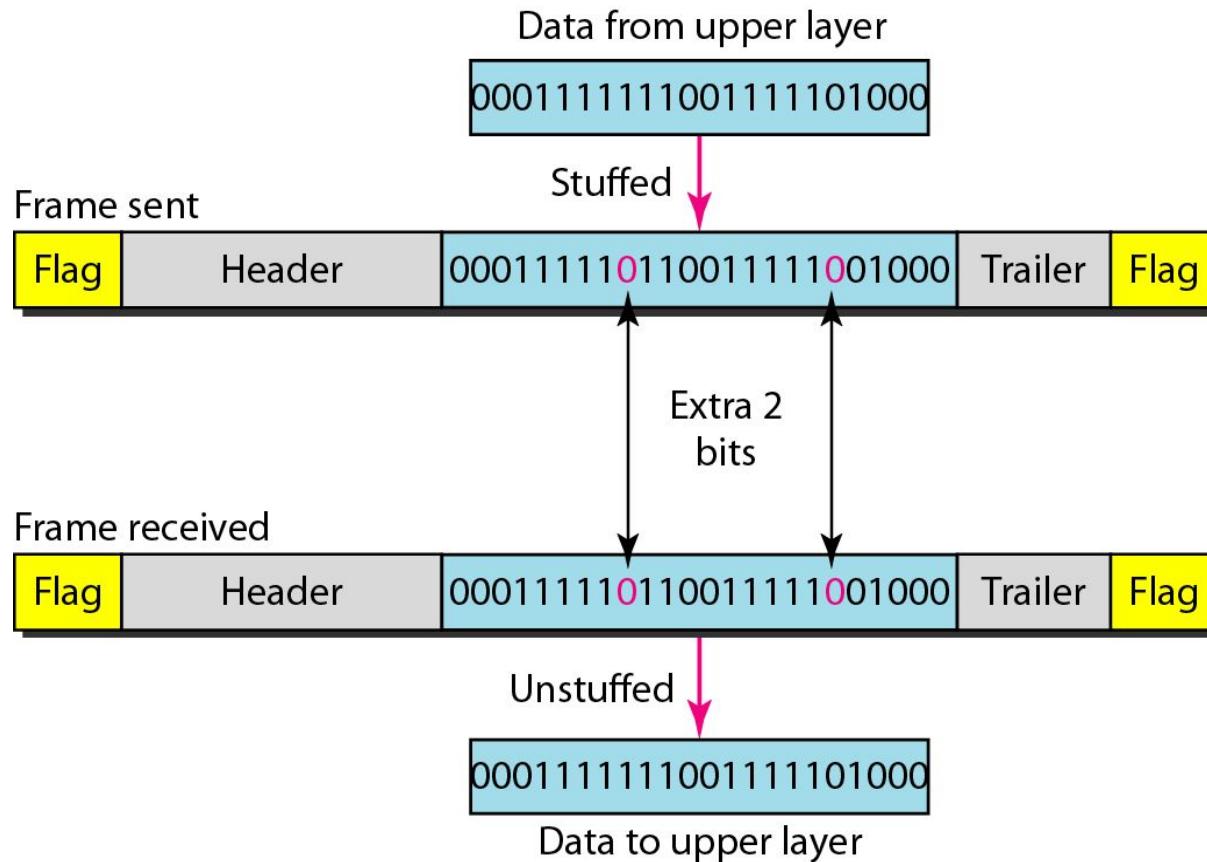




Note

Bit stuffing is the process of adding one extra 0 whenever five consecutive 1s follow a 0 in the data, so that the receiver does not mistake the pattern 0111110 for a flag.

Figure 11.4 Bit stuffing and unstuffing



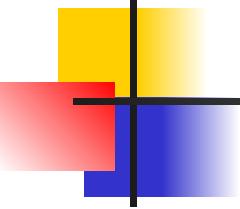
11-2 FLOW AND ERROR CONTROL

*The most important responsibilities of the data link layer are **flow control** and **error control**. Collectively, these functions are known as **data link control**.*

Topics discussed in this section:

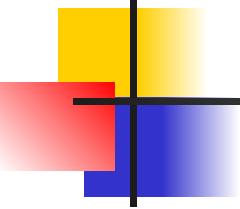
Flow Control

Error Control



Note

Flow control refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment.



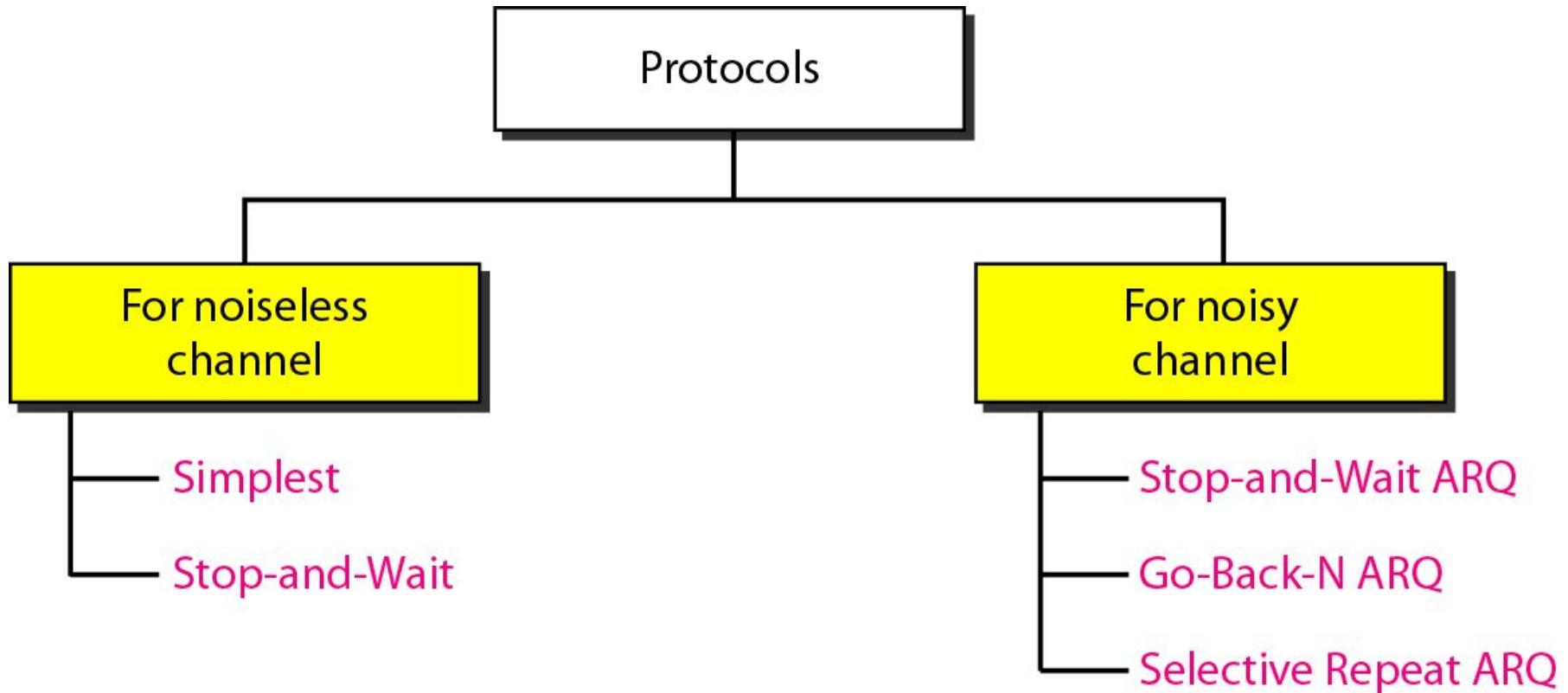
Note

Error control in the data link layer is based on automatic repeat request, which is the retransmission of data.

11-3 PROTOCOLS

Now let us see how the data link layer can combine framing, flow control, and error control to achieve the delivery of data from one node to another. The protocols are normally implemented in software by using one of the common programming languages. To make our discussions language-free, we have written in pseudocode a version of each protocol that concentrates mostly on the procedure instead of delving into the details of language rules.

Figure 11.5 *Taxonomy of protocols discussed in this chapter*



11-4 NOISELESS CHANNELS

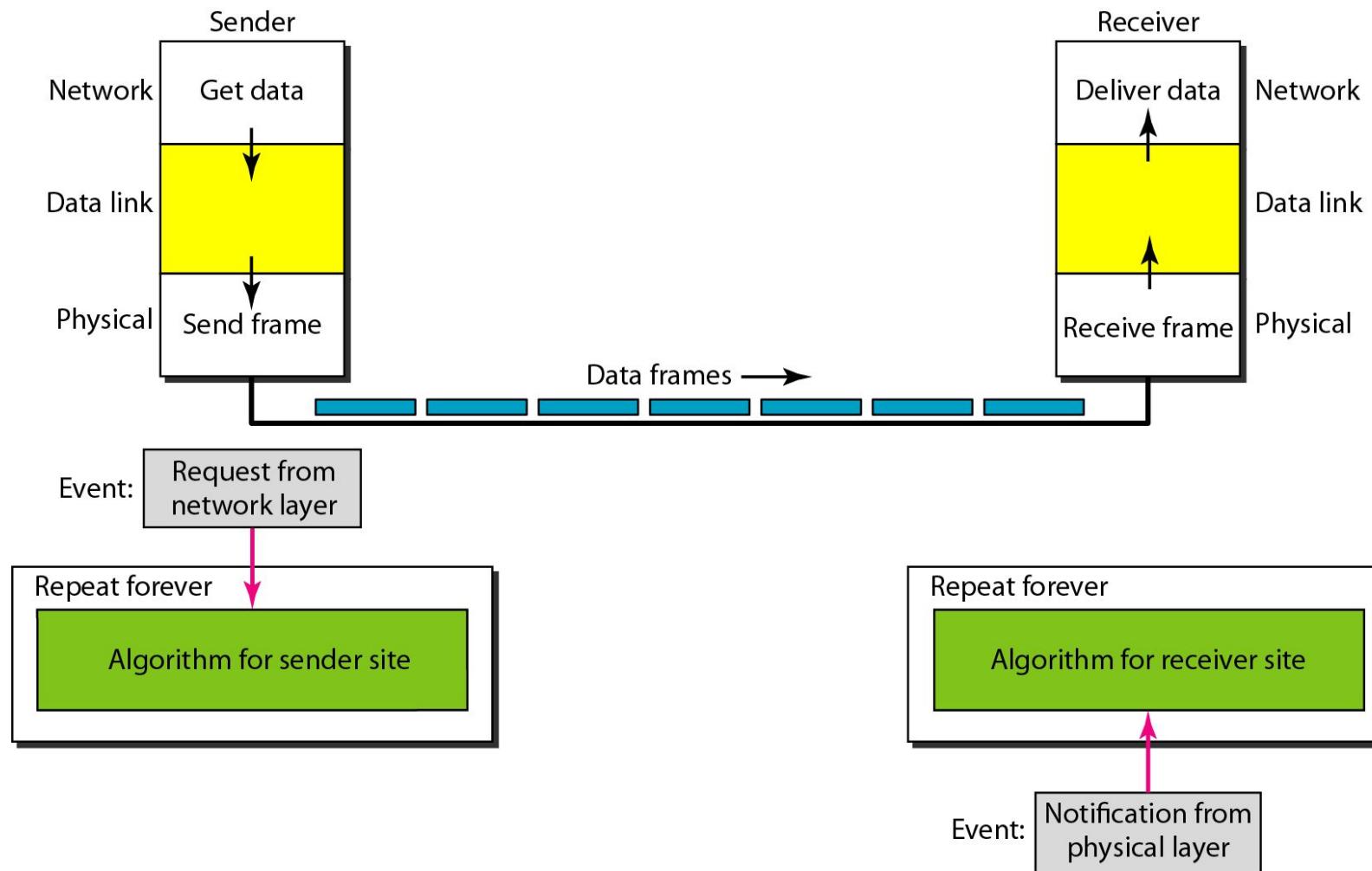
Let us first assume we have an ideal channel in which no frames are lost, duplicated, or corrupted. We introduce two protocols for this type of channel.

Topics discussed in this section:

Simplest Protocol

Stop-and-Wait Protocol

Figure 11.6 The design of the simplest protocol with no flow or error control

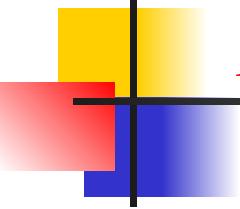


Algorithm 11.1 *Sender-site algorithm for the simplest protocol*

```
1 while(true)          // Repeat forever
2 {
3     WaitForEvent();    // Sleep until an event occurs
4     if(Event(RequestToSend)) //There is a packet to send
5     {
6         GetData();
7         MakeFrame();
8         SendFrame();      //Send the frame
9     }
10 }
```

Algorithm 11.2 *Receiver-site algorithm for the simplest protocol*

```
1 while(true)                                // Repeat forever
2 {
3     WaitForEvent();                         // Sleep until an event occurs
4     if(Event(ArrivalNotification)) //Data frame arrived
5     {
6         ReceiveFrame();
7         ExtractData();
8         DeliverData();                  //Deliver data to network layer
9     }
10 }
```



Example 11.1

Figure 11.7 shows an example of communication using this protocol. It is very simple. The sender sends a sequence of frames without even thinking about the receiver. To send three frames, three events occur at the sender site and three events at the receiver site. Note that the data frames are shown by tilted boxes; the height of the box defines the transmission time difference between the first bit and the last bit in the frame.

Figure 11.7 Flow diagram for Example 11.1

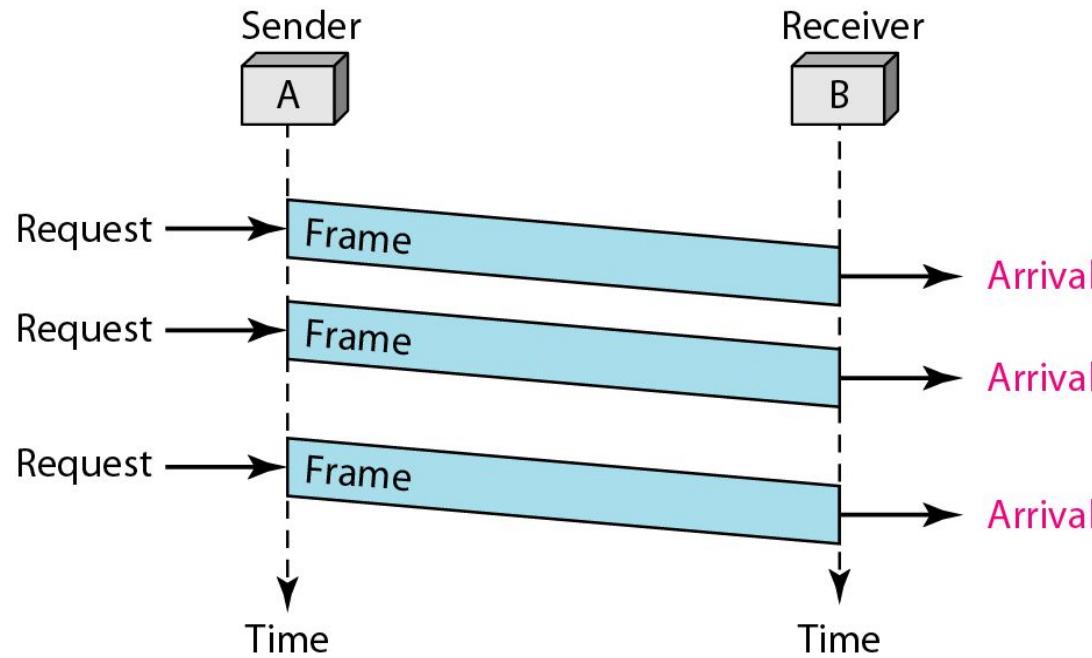
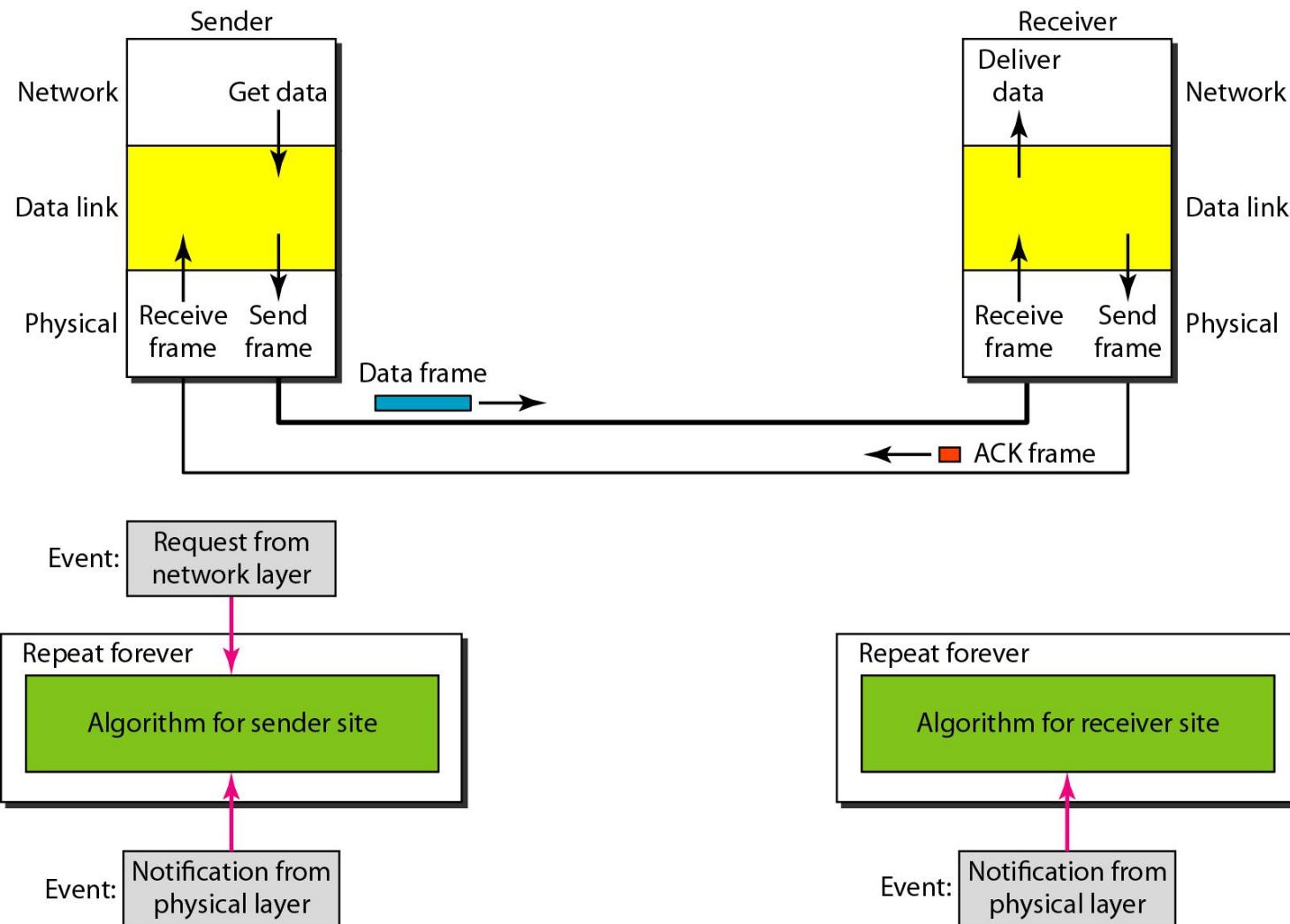


Figure 11.8 Design of Stop-and-Wait Protocol

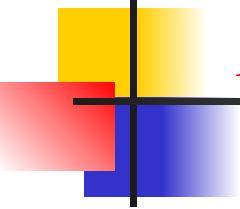


Algorithm 11.3 *Sender-site algorithm for Stop-and-Wait Protocol*

```
1 while(true)                                //Repeat forever
2 canSend = true                            //Allow the first frame to go
3 {
4     WaitForEvent();                      // Sleep until an event occurs
5     if(Event(RequestToSend) AND canSend)
6     {
7         GetData();
8         MakeFrame();
9         SendFrame();                     //Send the data frame
10        canSend = false;               //Cannot send until ACK arrives
11    }
12    WaitForEvent();                      // Sleep until an event occurs
13    if(Event(ArrivalNotification) // An ACK has arrived
14    {
15        ReceiveFrame();                //Receive the ACK frame
16        canSend = true;
17    }
18 }
```

Algorithm 11.4 *Receiver-site algorithm for Stop-and-Wait Protocol*

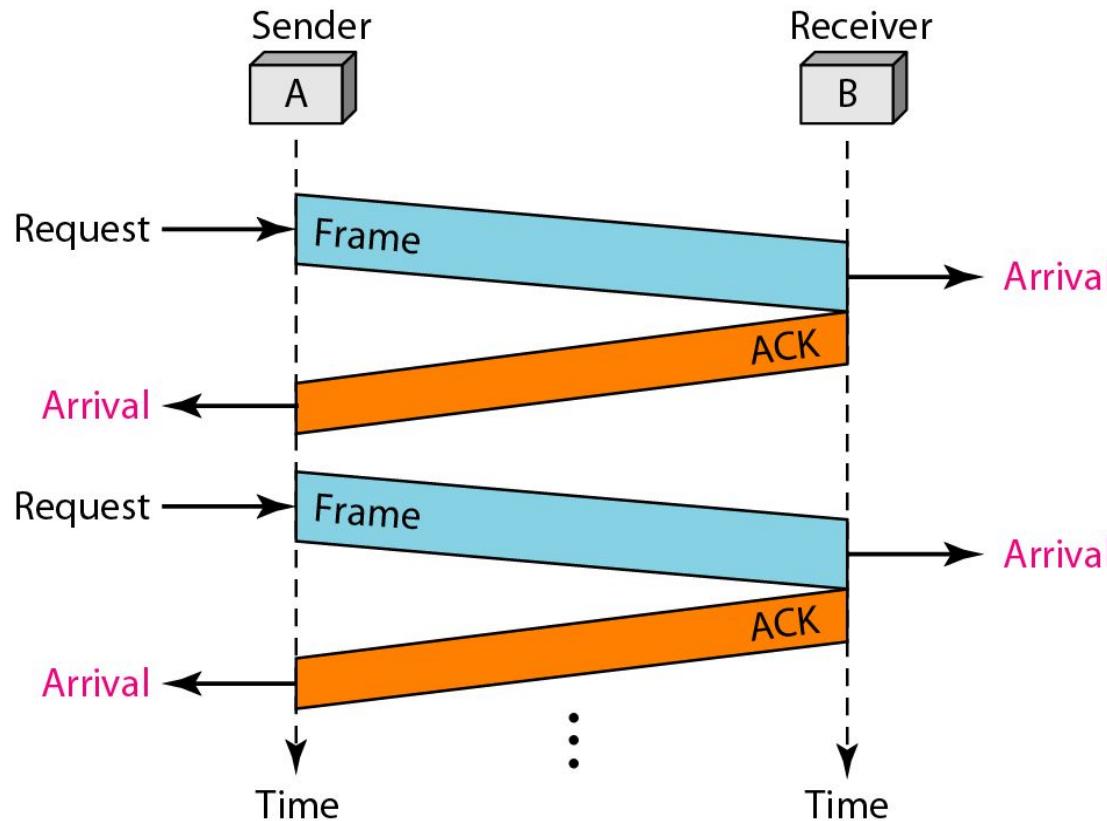
```
1 while(true)                                //Repeat forever
2 {
3     WaitForEvent();                         // Sleep until an event occurs
4     if(Event(ArrivalNotification)) //Data frame arrives
5     {
6         ReceiveFrame();
7         ExtractData();
8         Deliver(data);                  //Deliver data to network layer
9         SendFrame();                  //Send an ACK frame
10    }
11 }
```



Example 11.2

Figure 11.9 shows an example of communication using this protocol. It is still very simple. The sender sends one frame and waits for feedback from the receiver. When the ACK arrives, the sender sends the next frame. Note that sending two frames in the protocol involves the sender in four events and the receiver in two events.

Figure 11.9 Flow diagram for Example 11.2



11-5 NOISY CHANNELS

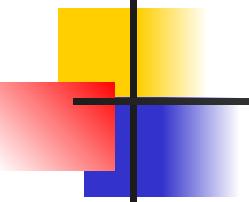
Although the Stop-and-Wait Protocol gives us an idea of how to add flow control to its predecessor, noiseless channels are nonexistent. We discuss three protocols in this section that use error control.

Topics discussed in this section:

Stop-and-Wait Automatic Repeat Request

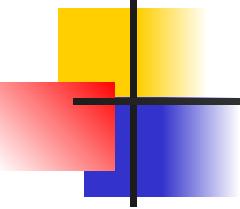
Go-Back-N Automatic Repeat Request

Selective Repeat Automatic Repeat Request



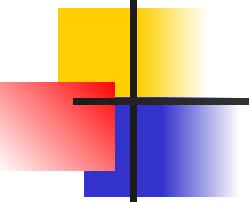
Note

Error correction in Stop-and-Wait ARQ is done by keeping a copy of the sent frame and retransmitting of the frame when the timer expires.



Note

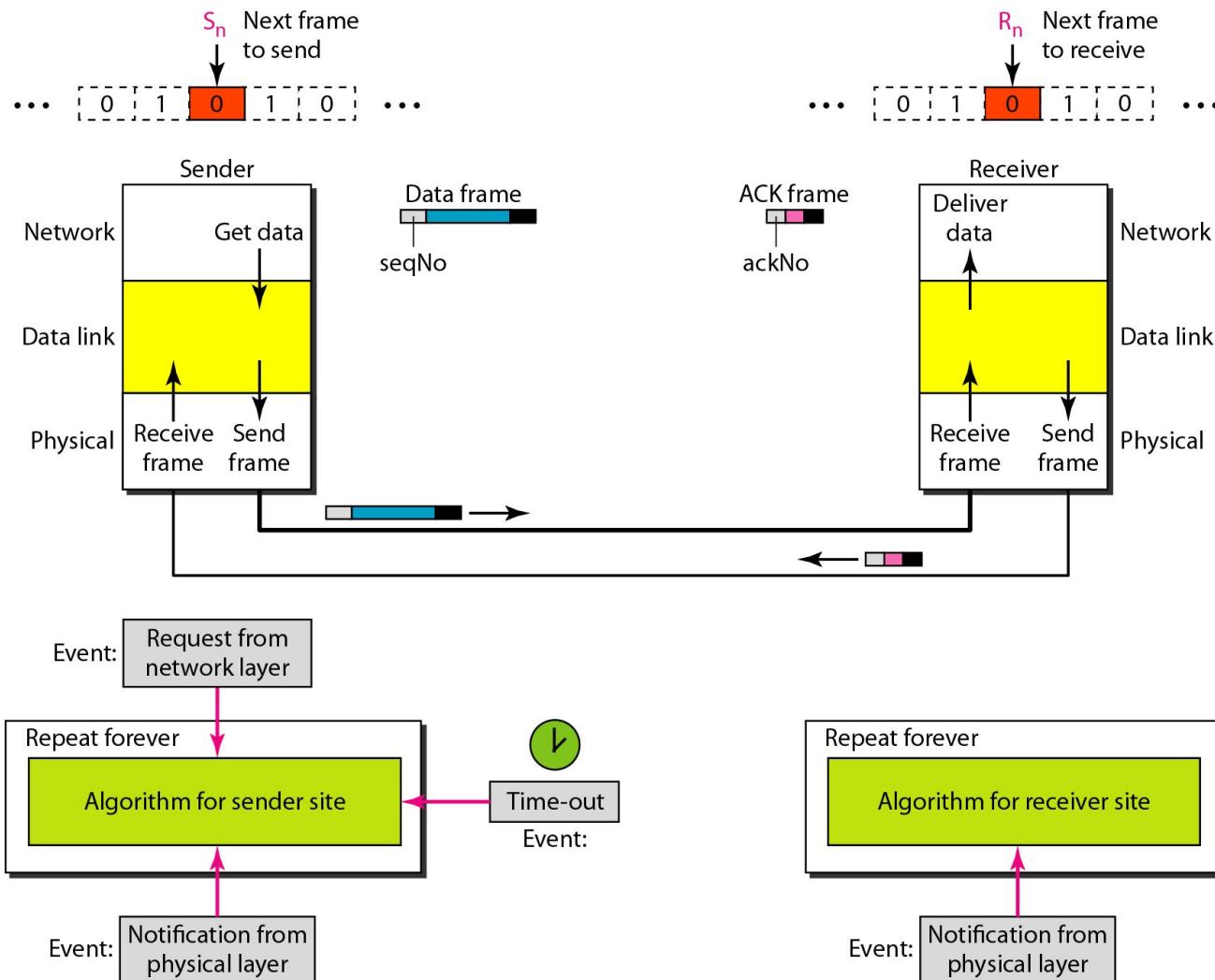
**In Stop-and-Wait ARQ, we use sequence numbers to number the frames.
The sequence numbers are based on modulo-2 arithmetic.**



Note

In Stop-and-Wait ARQ, the acknowledgment number always announces in modulo-2 arithmetic the sequence number of the next frame expected.

Figure 11.10 Design of the Stop-and-Wait ARQ Protocol



Algorithm 11.5 *Sender-site algorithm for Stop-and-Wait ARQ*

```
1 Sn = 0;                                // Frame 0 should be sent first
2 canSend = true;                           // Allow the first request to go
3 while(true)                               // Repeat forever
4 {
5   WaitForEvent();                         // Sleep until an event occurs
6   if(Event(RequestToSend) AND canSend)
7   {
8     GetData();
9     MakeFrame(Sn);                   //The seqNo is Sn
10    StoreFrame(Sn);                  //Keep copy
11    SendFrame(Sn);
12    StartTimer();
13    Sn = Sn + 1;
14    canSend = false;
15  }
16  WaitForEvent();                         // Sleep
```

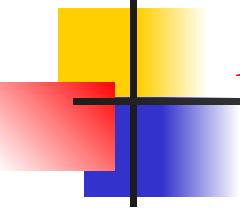
(continued)

Algorithm 11.5 Sender-site algorithm for Stop-and-Wait ARQ (continued)

```
17     if(Event(ArrivalNotification))          // An ACK has arrived
18     {
19         ReceiveFrame(ackNo);           //Receive the ACK frame
20         if(not corrupted AND ackNo == Sn) //Valid ACK
21         {
22             StopTimer();
23             PurgeFrame(Sn-1);        //Copy is not needed
24             canSend = true;
25         }
26     }
27
28     if(Event(TimeOut))                  // The timer expired
29     {
30         StartTimer();
31         ResendFrame(Sn-1);        //Resend a copy check
32     }
33 }
```

Algorithm 11.6 Receiver-site algorithm for Stop-and-Wait ARQ Protocol

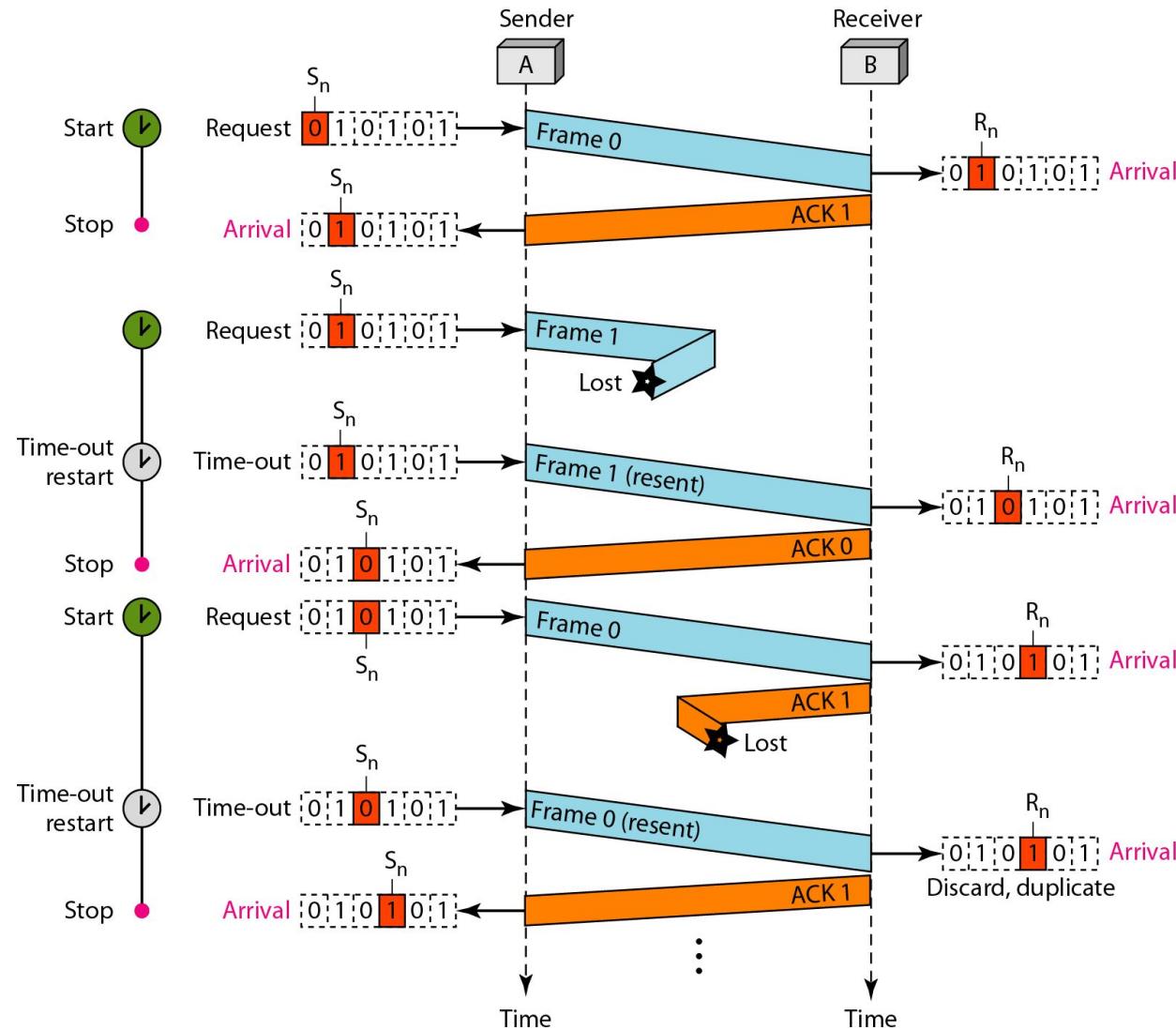
```
1 Rn = 0;                                // Frame 0 expected to arrive first
2 while(true)
3 {
4     WaitForEvent();                      // Sleep until an event occurs
5     if(Event(ArrivalNotification))      //Data frame arrives
6     {
7         ReceiveFrame();
8         if(corrupted(frame));
9             sleep();
10        if(seqNo == Rn)              //Valid data frame
11        {
12            ExtractData();
13            DeliverData();           //Deliver data
14            Rn = Rn + 1;
15        }
16        SendFrame(Rn);           //Send an ACK
17    }
18 }
```

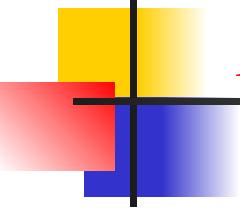


Example 11.3

Figure 11.11 shows an example of Stop-and-Wait ARQ. Frame 0 is sent and acknowledged. Frame 1 is lost and resent after the time-out. The resent frame 1 is acknowledged and the timer stops. Frame 0 is sent and acknowledged, but the acknowledgment is lost. The sender has no idea if the frame or the acknowledgment is lost, so after the time-out, it resends frame 0, which is acknowledged.

Figure 11.11 Flow diagram for Example 11.3





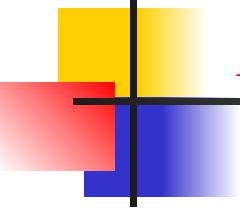
Example 11.4

Assume that, in a Stop-and-Wait ARQ system, the bandwidth of the line is 1 Mbps, and 1 bit takes 20 ms to make a round trip. What is the bandwidth-delay product? If the system data frames are 1000 bits in length, what is the utilization percentage of the link?

Solution

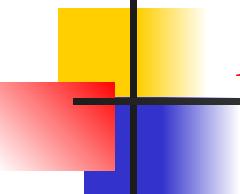
The bandwidth-delay product is

$$(1 \times 10^6) \times (20 \times 10^{-3}) = 20,000 \text{ bits}$$



Example 11.4 (continued)

The system can send 20,000 bits during the time it takes for the data to go from the sender to the receiver and then back again. However, the system sends only 1000 bits. We can say that the link utilization is only $1000/20,000$, or 5 percent. For this reason, for a link with a high bandwidth or long delay, the use of Stop-and-Wait ARQ wastes the capacity of the link.

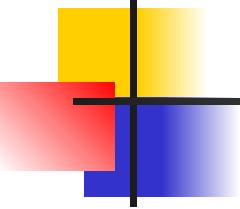


Example 11.5

What is the utilization percentage of the link in Example 11.4 if we have a protocol that can send up to 15 frames before stopping and worrying about the acknowledgments?

Solution

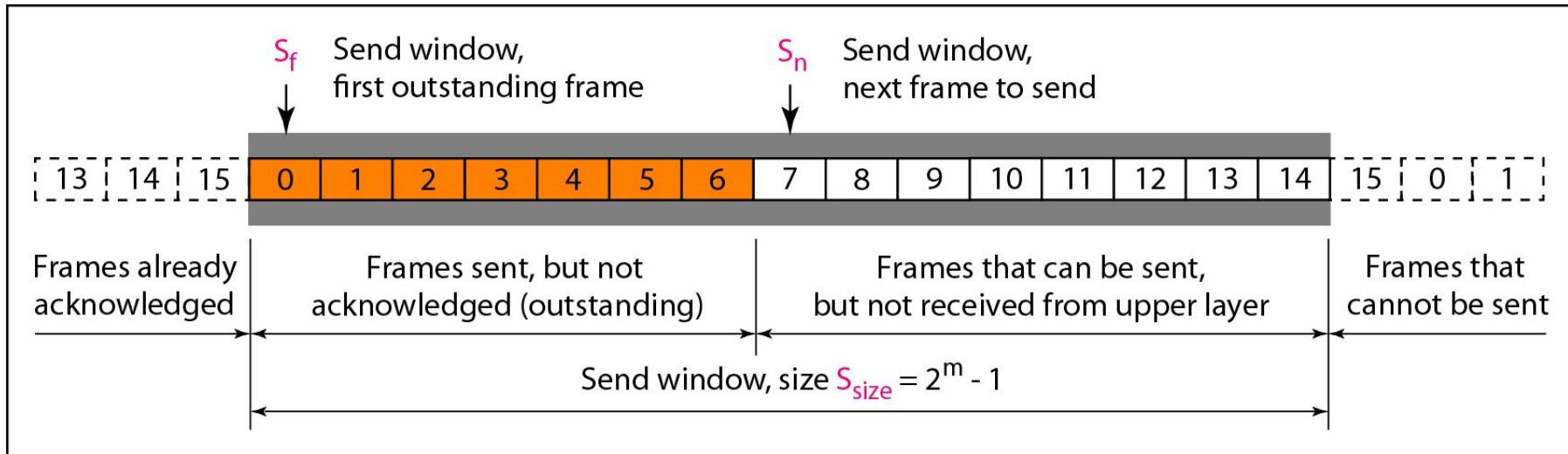
The bandwidth-delay product is still 20,000 bits. The system can send up to 15 frames or 15,000 bits during a round trip. This means the utilization is 15,000/20,000, or 75 percent. Of course, if there are damaged frames, the utilization percentage is much less because frames have to be resent.



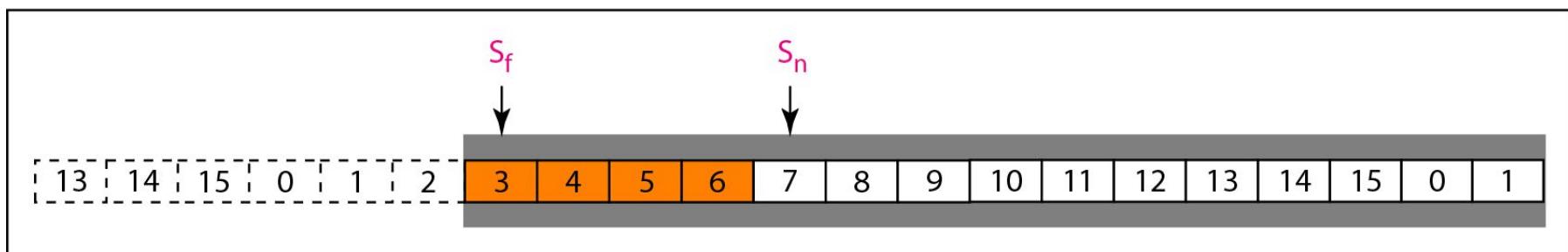
Note

In the Go-Back-N Protocol, the sequence numbers are modulo 2^m , where m is the size of the sequence number field in bits.

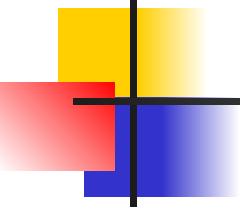
Figure 11.12 Send window for Go-Back-N ARQ



a. Send window before sliding

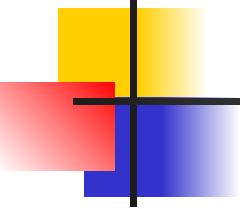


b. Send window after sliding



Note

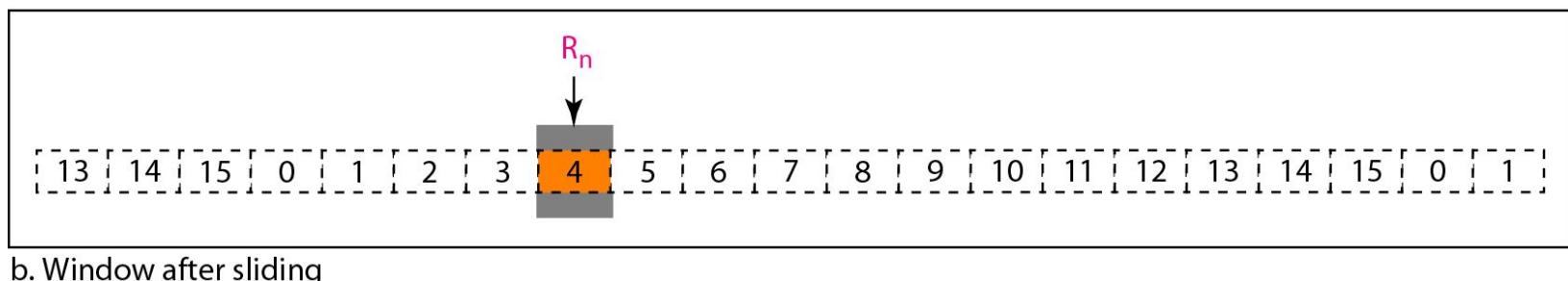
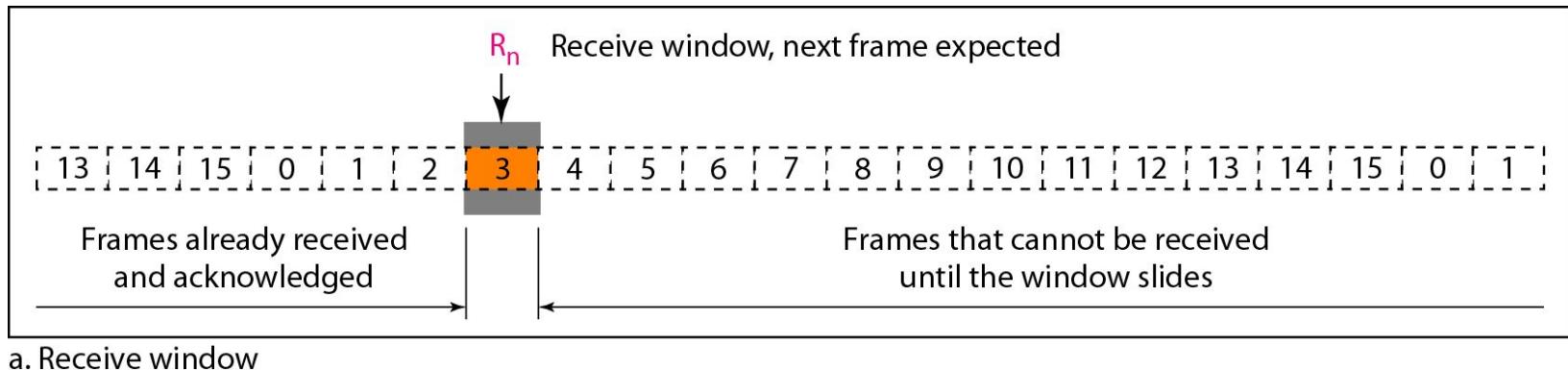
The send window is an abstract concept defining an imaginary box of size $2^m - 1$ with three variables: S_f , S_n , and S_{size} .

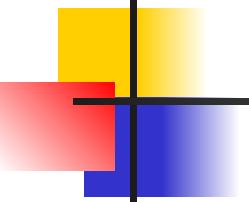


Note

The send window can slide one or more slots when a valid acknowledgment arrives.

Figure 11.13 Receive window for Go-Back-N ARQ





Note

The receive window is an abstract concept defining an imaginary box of size 1 with one single variable R_n .

The window slides when a correct frame has arrived; sliding occurs one slot at a time.

Figure 11.14 Design of Go-Back-N ARQ

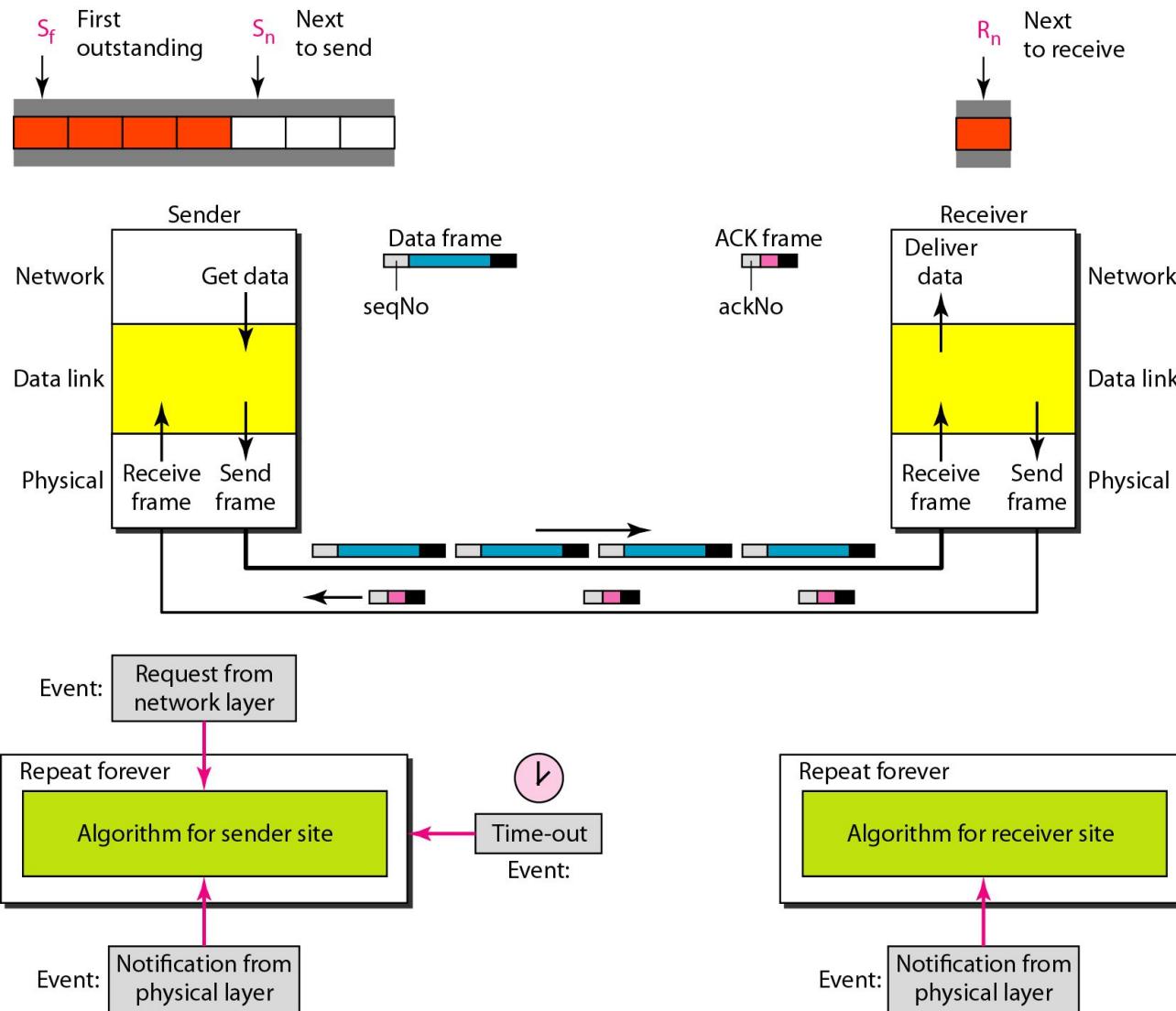
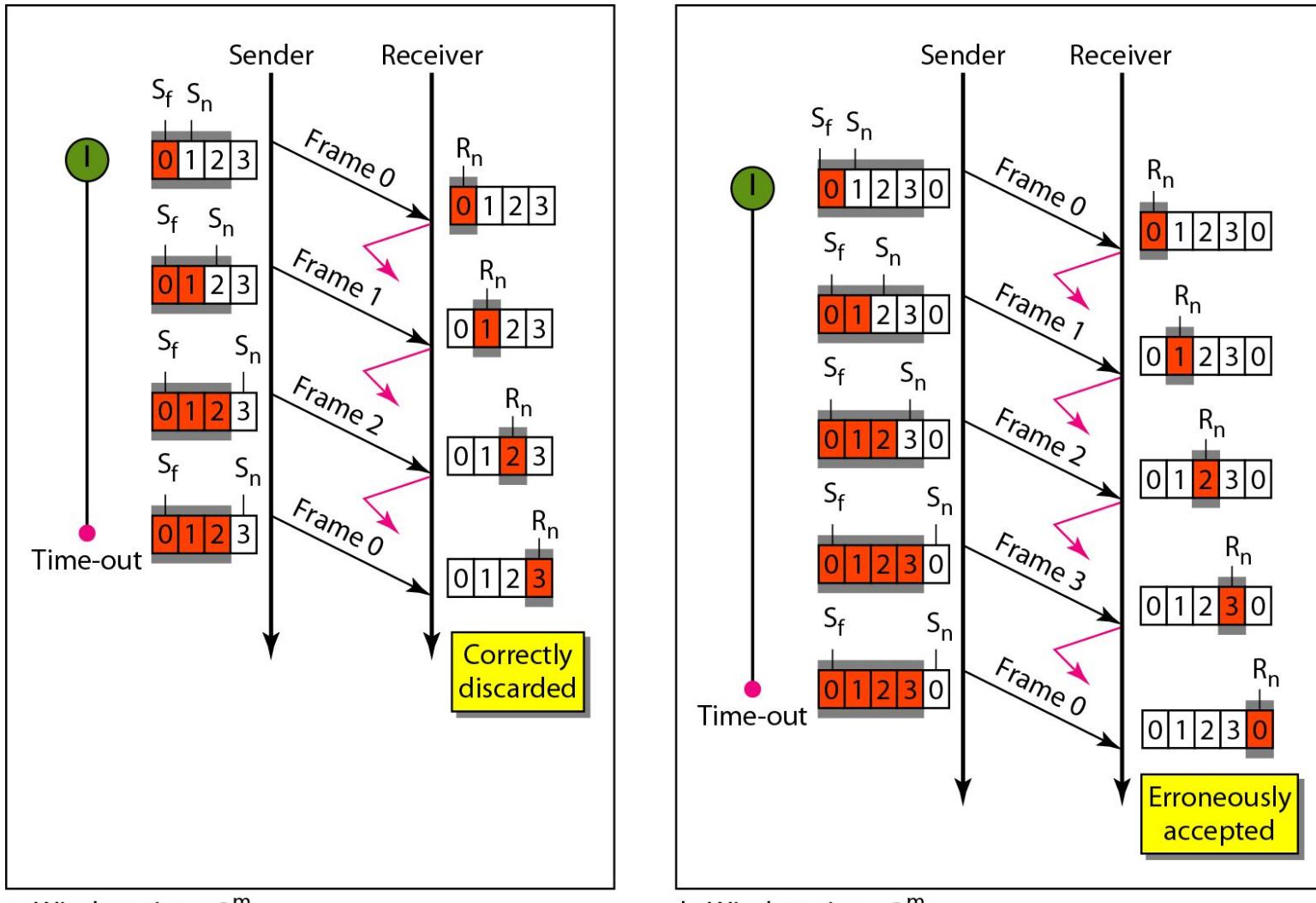
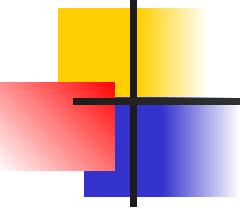


Figure 11.15 Window size for Go-Back-N ARQ





Note

In Go-Back-N ARQ, the size of the send window must be less than 2^m ; the size of the receiver window is always 1.

Algorithm 11.7 Go-Back-N sender algorithm

```
1 Sw = 2m - 1;  
2 Sf = 0;  
3 Sn = 0;  
4  
5 while (true) //Repeat forever  
6 {  
7   WaitForEvent();  
8   if(Event(RequestToSend)) //A packet to send  
9   {  
10     if(Sn-Sf >= Sw) //If window is full  
11       Sleep();  
12     GetData();  
13     MakeFrame(Sn);  
14     StoreFrame(Sn);  
15     SendFrame(Sn);  
16     Sn = Sn + 1;  
17     if(timer not running)  
18       StartTimer();  
19   }  
20 }
```

(continued)

Algorithm 11.7 Go-Back-N sender algorithm

(continued)

```
21  if(Event(ArrivalNotification)) //ACK arrives
22  {
23      Receive(ACK);
24      if(corrupted(ACK))
25          Sleep();
26      if((ackNo>Sf)&&(ackNo<=Sn)) //If a valid ACK
27      While(Sf <= ackNo)
28      {
29          PurgeFrame(Sf);
30          Sf = Sf + 1;
31      }
32      StopTimer();
33  }

34

35  if(Event(TimeOut)) //The timer expires
36  {
37      StartTimer();
38      Temp = Sf;
39      while(Temp < Sn);
40      {
41          SendFrame(Sf);
42          Sf = Sf + 1;
43      }
44  }
45 }
```

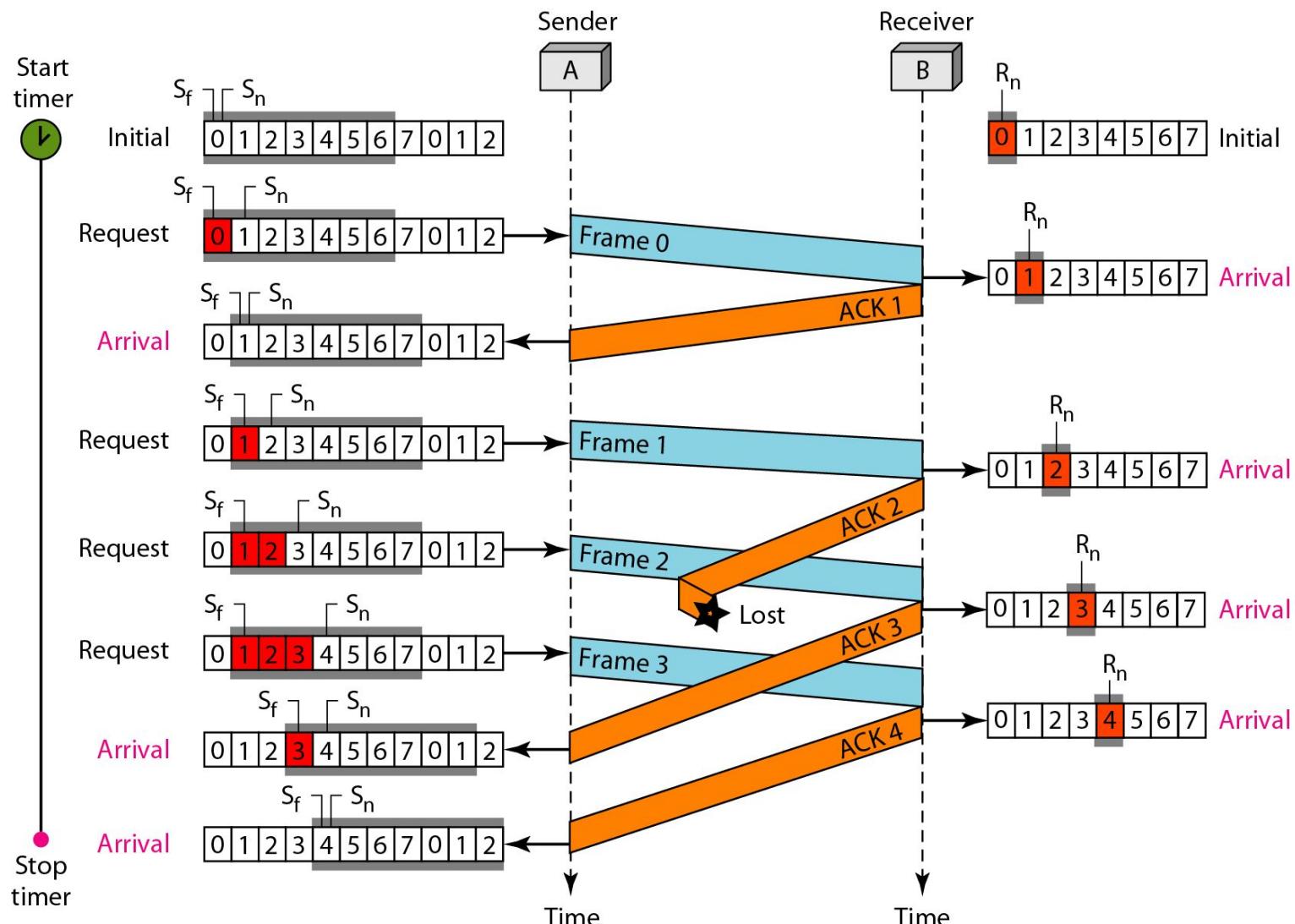
Algorithm 11.8 Go-Back-N receiver algorithm

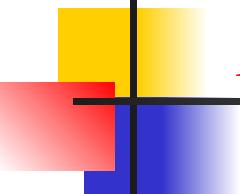
```
1 Rn = 0;  
2  
3 while (true) //Repeat forever  
4 {  
5     WaitForEvent();  
6  
7     if(Event(ArrivalNotification)) /Data frame arrives  
8     {  
9         Receive(Frame);  
10        if(corrupted(Frame))  
11            Sleep();  
12        if(seqNo == Rn) //If expected frame  
13        {  
14            DeliverData(); //Deliver data  
15            Rn = Rn + 1; //Slide window  
16            SendACK(Rn);  
17        }  
18    }  
19}
```

Example 11.6

Figure 11.16 shows an example of Go-Back-N. This is an example of a case where the forward channel is reliable, but the reverse is not. No data frames are lost, but some ACKs are delayed and one is lost. The example also shows how cumulative acknowledgments can help if acknowledgments are delayed or lost. After initialization, there are seven sender events. Request events are triggered by data from the network layer; arrival events are triggered by acknowledgments from the physical layer. There is no time-out event here because all outstanding frames are acknowledged before the timer expires. Note that although ACK 2 is lost, ACK 3 serves as both ACK 2 and ACK 3.

Figure 11.16 Flow diagram for Example 11.6





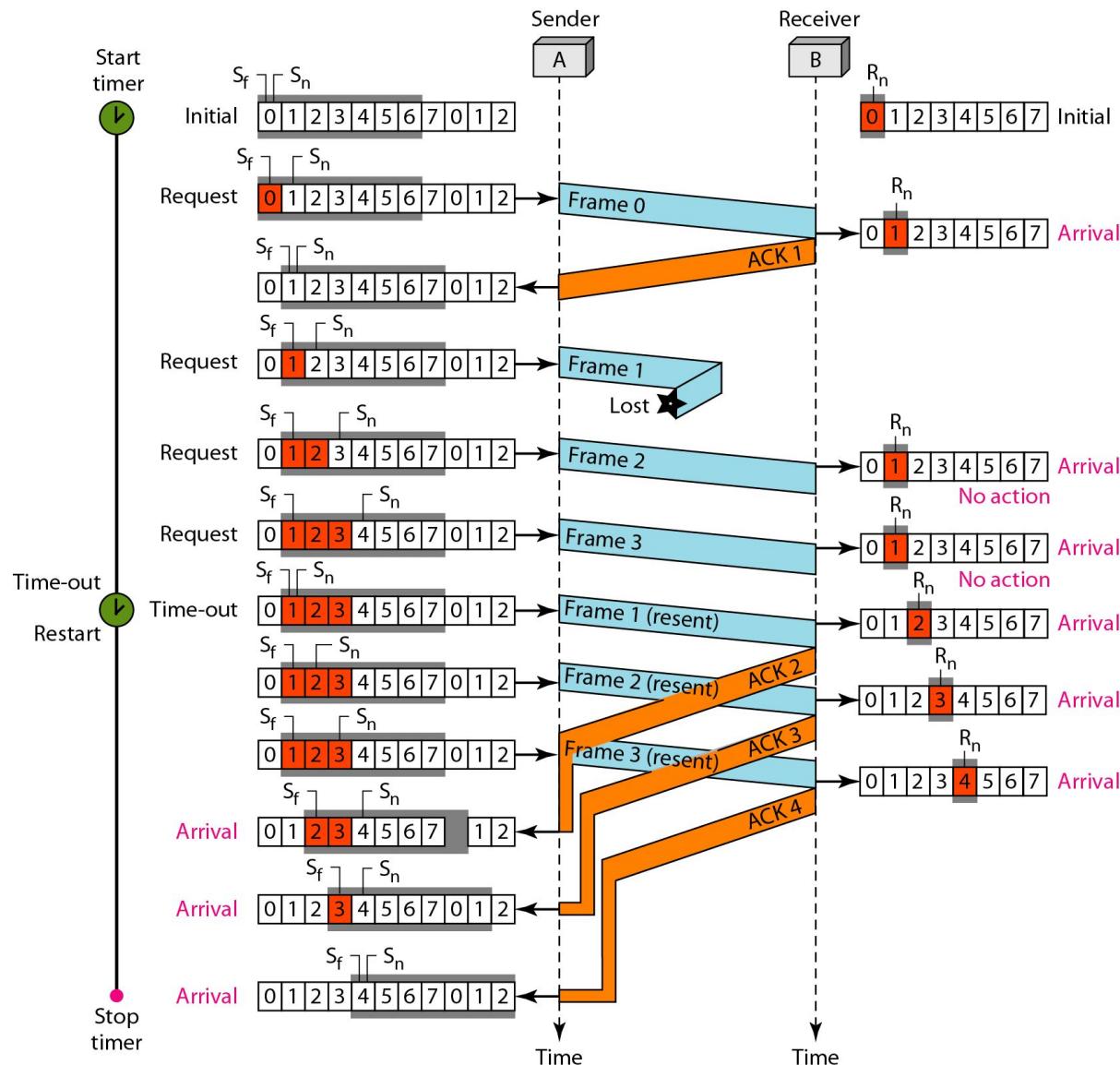
Example 11.7

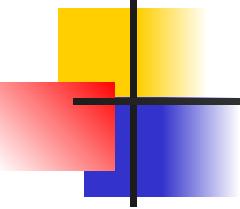
Figure 11.17 shows what happens when a frame is lost. Frames 0, 1, 2, and 3 are sent. However, frame 1 is lost. The receiver receives frames 2 and 3, but they are discarded because they are received out of order. The sender receives no acknowledgment about frames 1, 2, or 3. Its timer finally expires. The sender sends all outstanding frames (1, 2, and 3) because it does not know what is wrong. Note that the resending of frames 1, 2, and 3 is the response to one single event. When the sender is responding to this event, it cannot accept the triggering of other events. This means that when ACK 2 arrives, the sender is still busy with sending frame 3.

Example 11.7 (continued)

The physical layer must wait until this event is completed and the data link layer goes back to its sleeping state. We have shown a vertical line to indicate the delay. It is the same story with ACK 3; but when ACK 3 arrives, the sender is busy responding to ACK 2. It happens again when ACK 4 arrives. Note that before the second timer expires, all outstanding frames have been sent and the timer is stopped.

Figure 11.17 Flow diagram for Example 11.7





Note

Stop-and-Wait ARQ is a special case of Go-Back-N ARQ in which the size of the send window is 1.

Figure 11.18 Send window for Selective Repeat ARQ

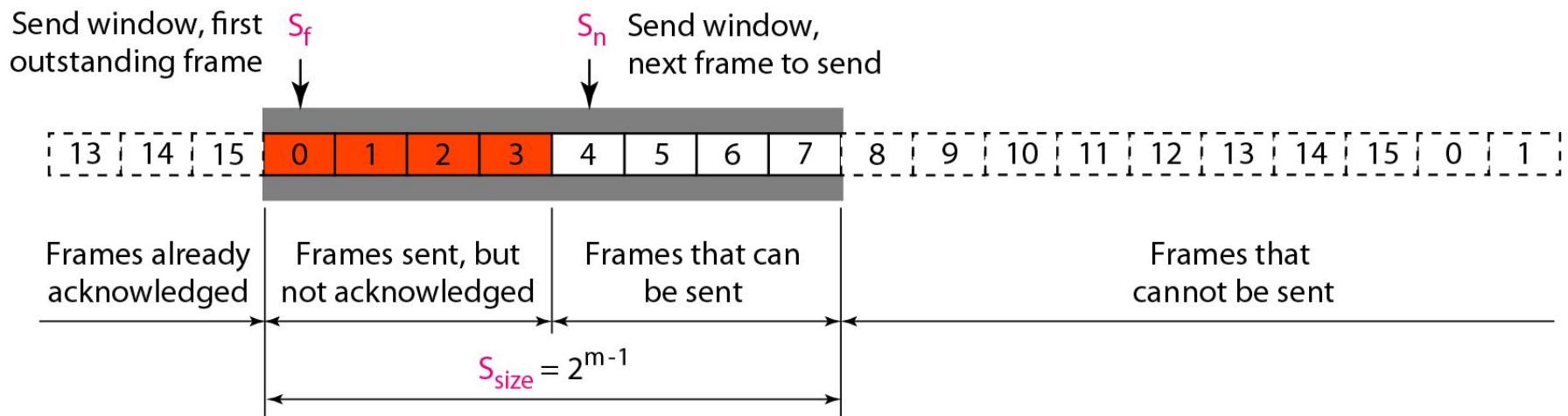


Figure 11.19 Receive window for Selective Repeat ARQ

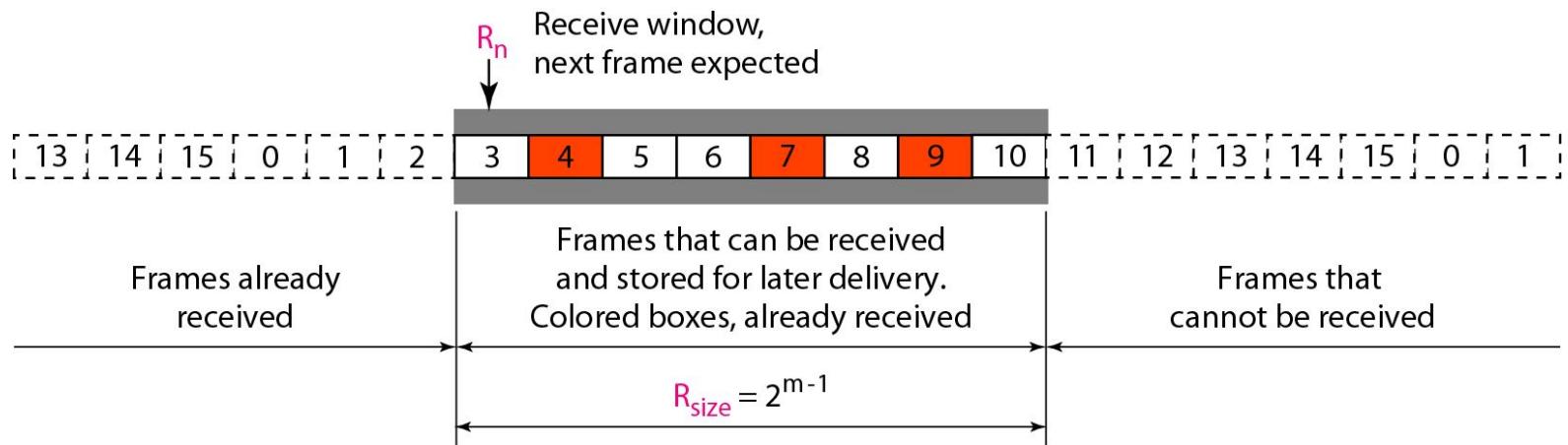


Figure 11.20 Design of Selective Repeat ARQ

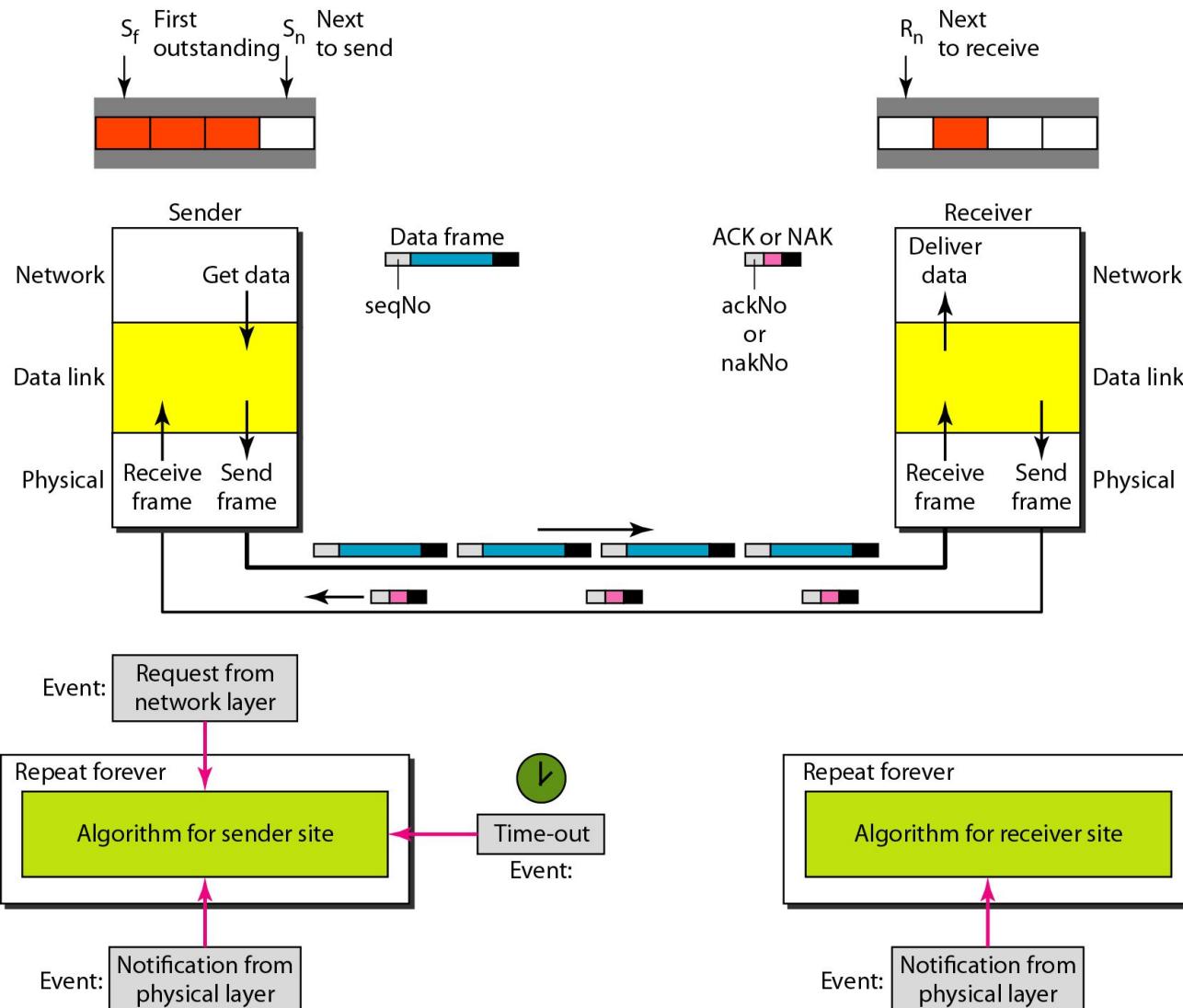
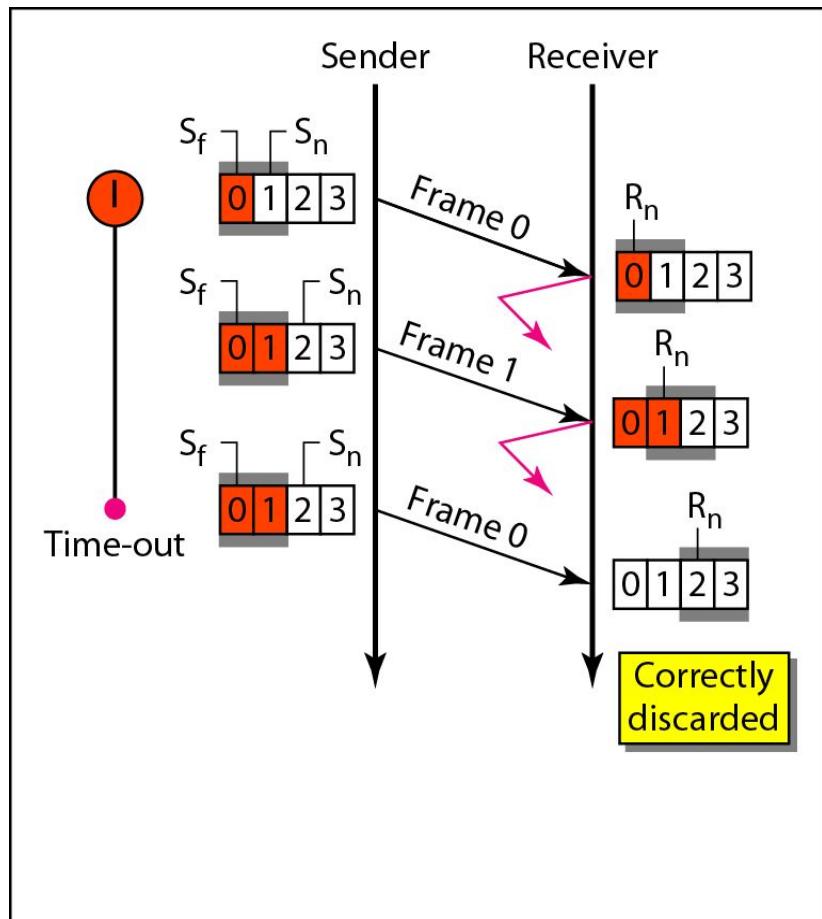
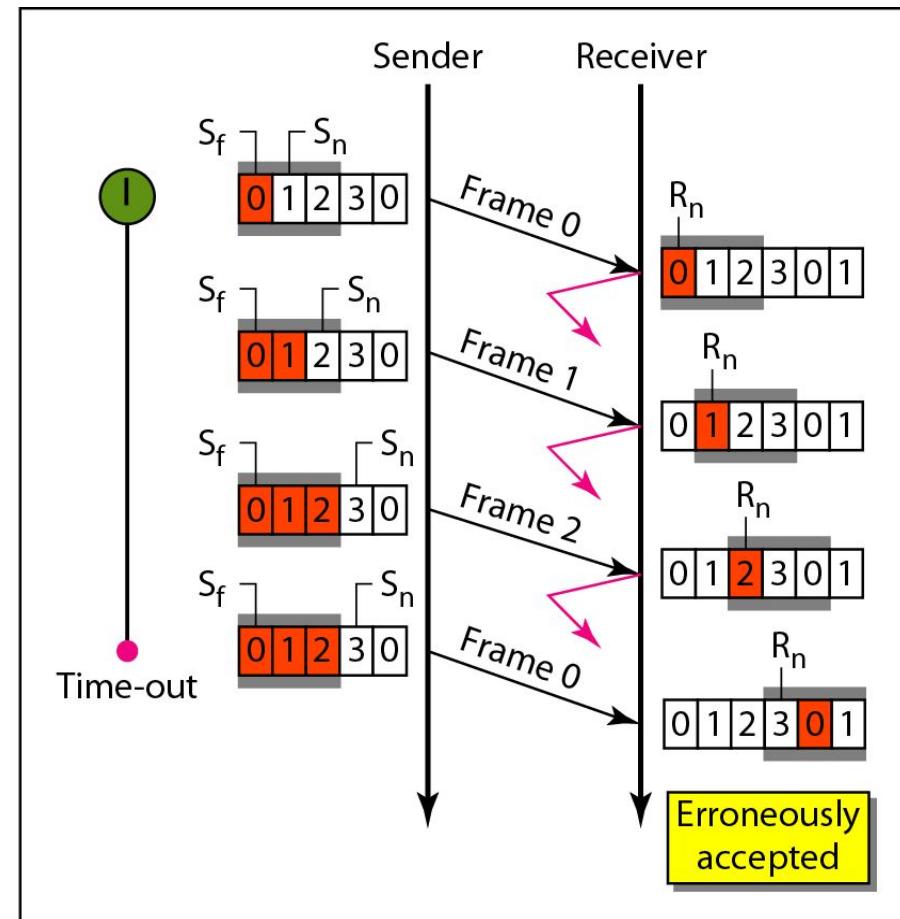


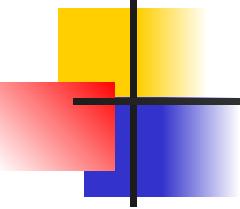
Figure 11.21 Selective Repeat ARQ, window size



a. Window size = 2^{m-1}



b. Window size > 2^{m-1}



Note

In Selective Repeat ARQ, the size of the sender and receiver window must be at most one-half of 2^m .

Algorithm 11.9 *Sender-site Selective Repeat algorithm*

```
1 Sw = 2m-1 ;
2 Sf = 0 ;
3 Sn = 0 ;
4
5 while (true)                                //Repeat forever
6 {
7     WaitForEvent() ;
8     if(Event(RequestToSend))                  //There is a packet to send
9     {
10         if(Sn-Sf >= Sw)                //If window is full
11             Sleep() ;
12         GetData() ;
13         MakeFrame(Sn) ;
14         StoreFrame(Sn) ;
15         SendFrame(Sn) ;
16         Sn = Sn + 1 ;
17         StartTimer(Sn) ;
18     }
19 }
```

(continued)

Algorithm 11.9 Sender-site Selective Repeat algorithm

(continued)

```
20  if(Event(ArrivalNotification)) //ACK arrives
21  {
22      Receive(frame);           //Receive ACK or NAK
23      if(corrupted(frame))
24          Sleep();
25      if (FrameType == NAK)
26          if (nakNo between Sf and Sn)
27          {
28              resend(nakNo);
29              StartTimer(nakNo);
30          }
31      if (FrameType == ACK)
32          if (ackNo between Sf and Sn)
33          {
34              while(sf < ackNo)
35              {
36                  Purge(sf);
37                  StopTimer(sf);
38                  Sf = Sf + 1;
39              }
40          }
41 }
```

Algorithm 11.9 *Sender-site Selective Repeat algorithm*

(continued)

```
42
43     if(Event(TimeOut(t)))          //The timer expires
44     {
45         StartTimer(t);
46         SendFrame(t);
47     }
48 }
```

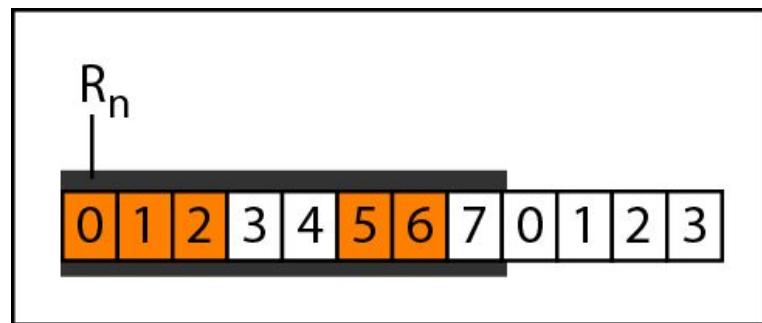
Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

```
1 Rn = 0;
2 NakSent = false;
3 AckNeeded = false;
4 Repeat(for all slots)
5     Marked(slot) = false;
6
7 while (true)                                //Repeat forever
8 {
9     WaitForEvent();
10
11    if(Event(ArrivalNotification))           /Data frame arrives
12    {
13        Receive(Frame);
14        if(corrupted(Frame))&& (NOT NakSent)
15        {
16            SendNAK(Rn);
17            NakSent = true;
18            Sleep();
19        }
20        if(seqNo <> Rn)&& (NOT NakSent)
21        {
22            SendNAK(Rn);
```

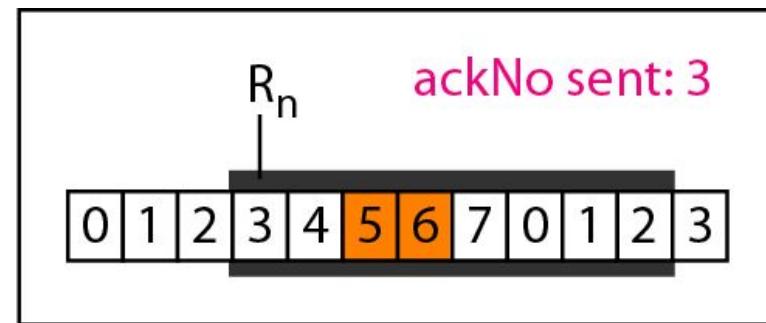
Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

```
23     NakSent = true;
24     if ((seqNo in window) && (!Marked(seqNo)))
25     {
26         StoreFrame(seqNo)
27         Marked(seqNo)= true;
28         while(Marked(Rn))
29         {
30             DeliverData(Rn);
31             Purge(Rn);
32             Rn = Rn + 1;
33             AckNeeded = true;
34         }
35         if(AckNeeded);
36         {
37             SendAck(Rn);
38             AckNeeded = false;
39             NakSent = false;
40         }
41     }
42 }
43 }
44 }
```

Figure 11.22 *Delivery of data in Selective Repeat ARQ*



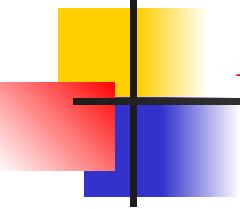
a. Before delivery



b. After delivery

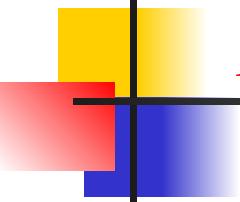
Example 11.8

This example is similar to Example 11.3 in which frame 1 is lost. We show how Selective Repeat behaves in this case. Figure 11.23 shows the situation. One main difference is the number of timers. Here, each frame sent or resent needs a timer, which means that the timers need to be numbered (0, 1, 2, and 3). The timer for frame 0 starts at the first request, but stops when the ACK for this frame arrives. The timer for frame 1 starts at the second request, restarts when a NAK arrives, and finally stops when the last ACK arrives. The other two timers start when the corresponding frames are sent and stop at the last arrival event.



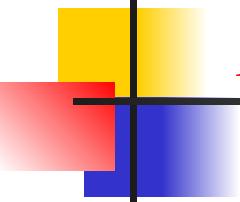
Example 11.8 (continued)

At the receiver site we need to distinguish between the acceptance of a frame and its delivery to the network layer. At the second arrival, frame 2 arrives and is stored and marked, but it cannot be delivered because frame 1 is missing. At the next arrival, frame 3 arrives and is marked and stored, but still none of the frames can be delivered. Only at the last arrival, when finally a copy of frame 1 arrives, can frames 1, 2, and 3 be delivered to the network layer. There are two conditions for the delivery of frames to the network layer: First, a set of consecutive frames must have arrived. Second, the set starts from the beginning of the window.



Example 11.8 (continued)

Another important point is that a NAK is sent after the second arrival, but not after the third, although both situations look the same. The reason is that the protocol does not want to crowd the network with unnecessary NAKs and unnecessary resent frames. The second NAK would still be NAK1 to inform the sender to resend frame 1 again; this has already been done. The first NAK sent is remembered (using the nakSent variable) and is not sent again until the frame slides. A NAK is sent once for each window position and defines the first slot in the window.



Example 11.8 (continued)

The next point is about the ACKs. Notice that only two ACKs are sent here. The first one acknowledges only the first frame; the second one acknowledges three frames. In Selective Repeat, ACKs are sent when data are delivered to the network layer. If the data belonging to n frames are delivered in one shot, only one ACK is sent for all of them.

Figure 11.23 Flow diagram for Example 11.8

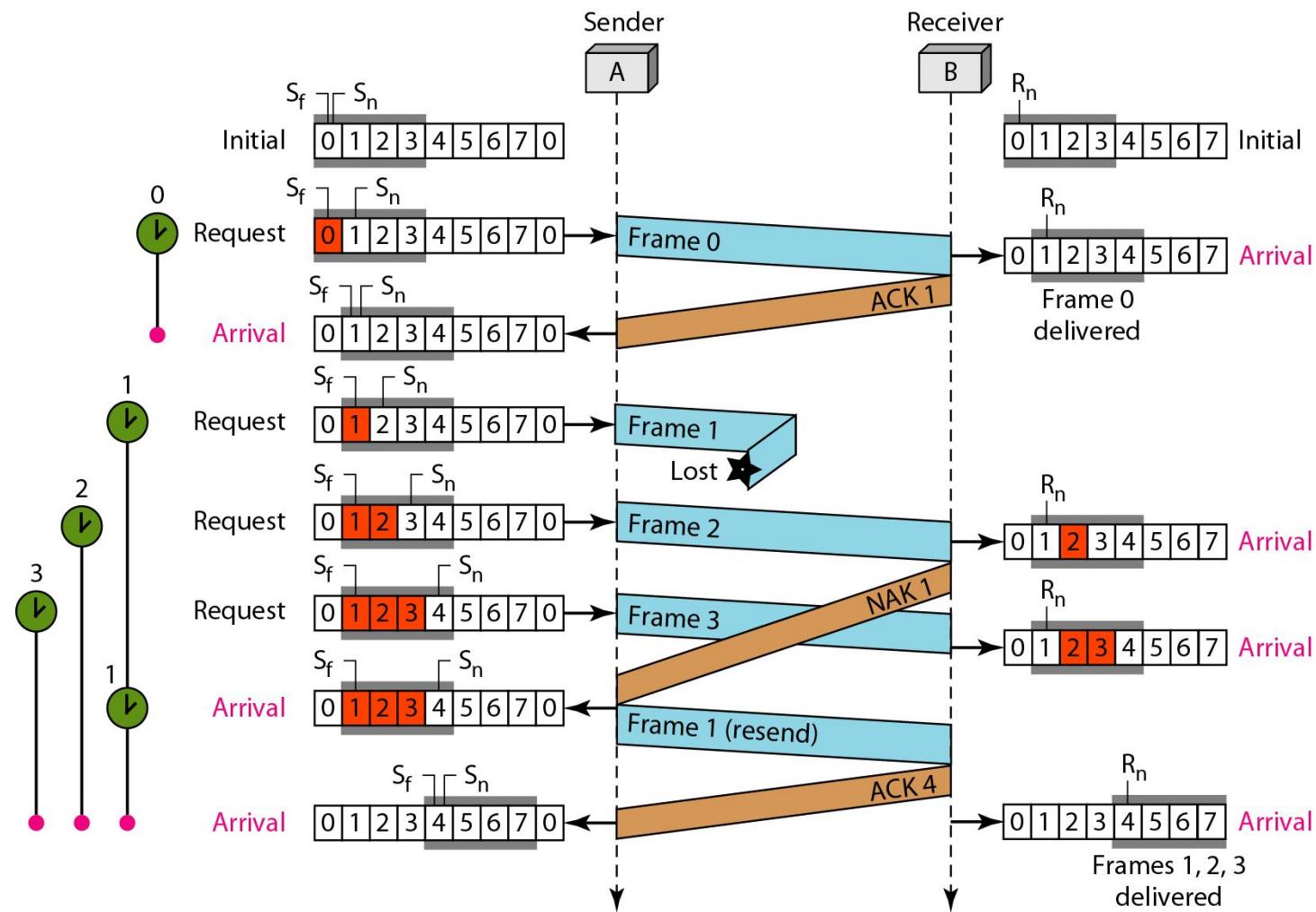
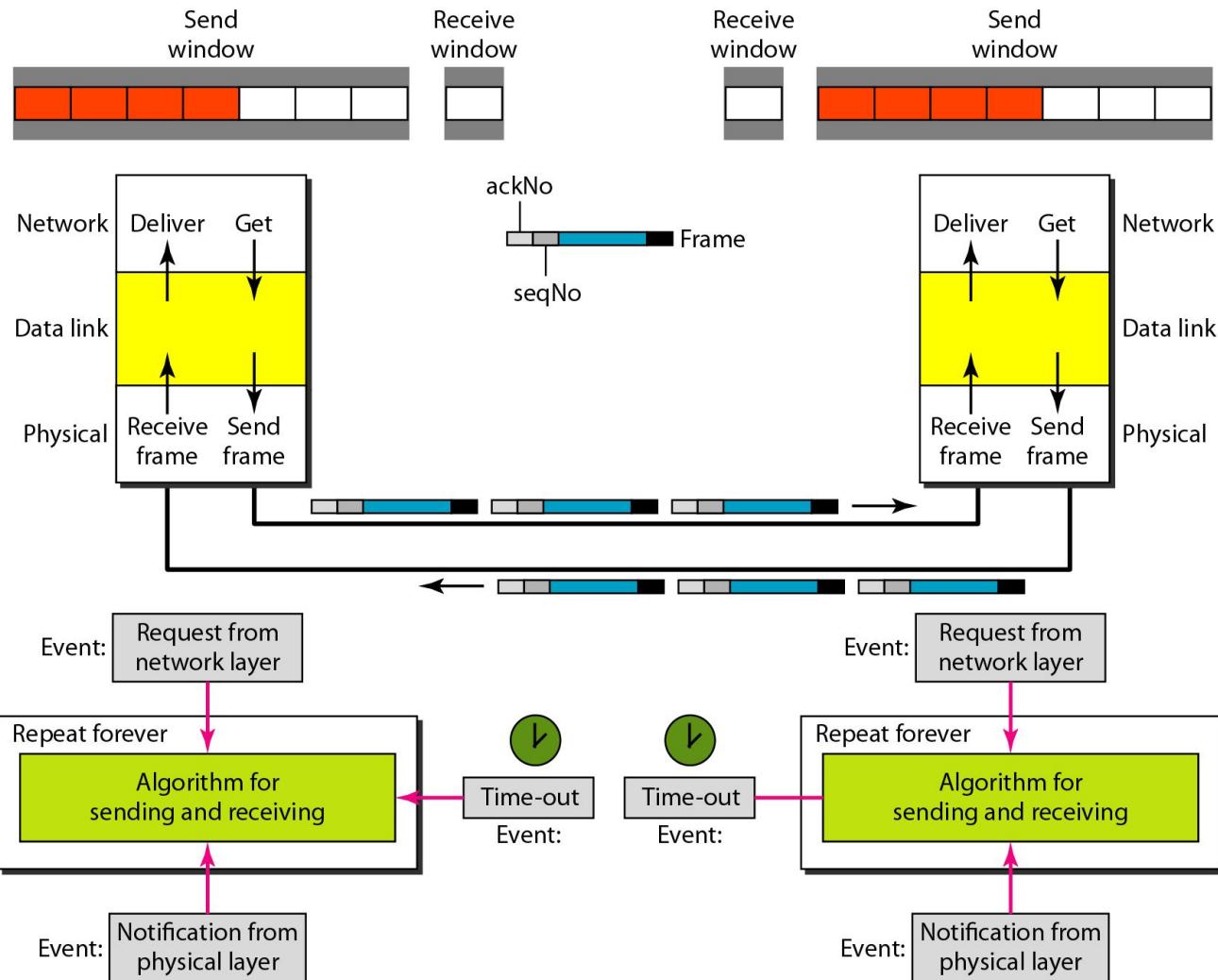


Figure 11.24 Design of piggybacking in Go-Back-N ARQ



11-7 POINT-TO-POINT PROTOCOL

*Although HDLC is a general protocol that can be used for both point-to-point and multipoint configurations, one of the most common protocols for point-to-point access is the **Point-to-Point Protocol (PPP)**. PPP is a byte-oriented protocol.*

Topics discussed in this section:

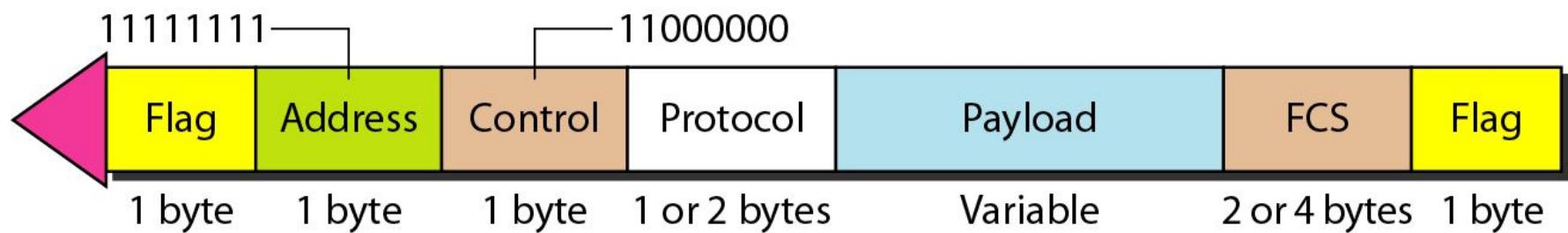
Framing

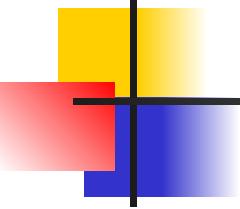
Transition Phases

Multiplexing

Multilink PPP

Figure 11.32 PPP frame format





Note

**PPP is a byte-oriented protocol using
byte stuffing with the escape byte
01111101.**

Figure 11.33 Transition phases

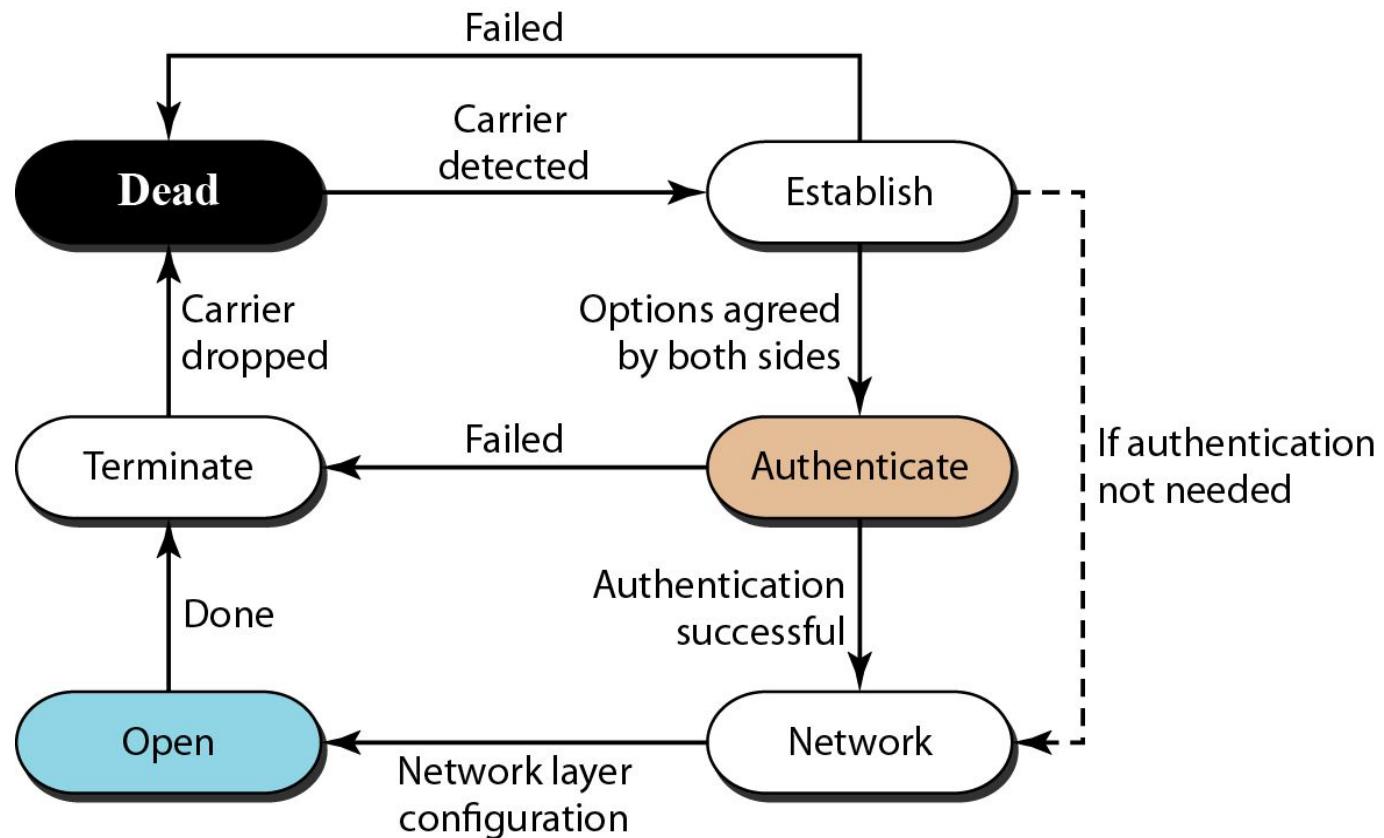


Figure 11.34 Multiplexing in PPP

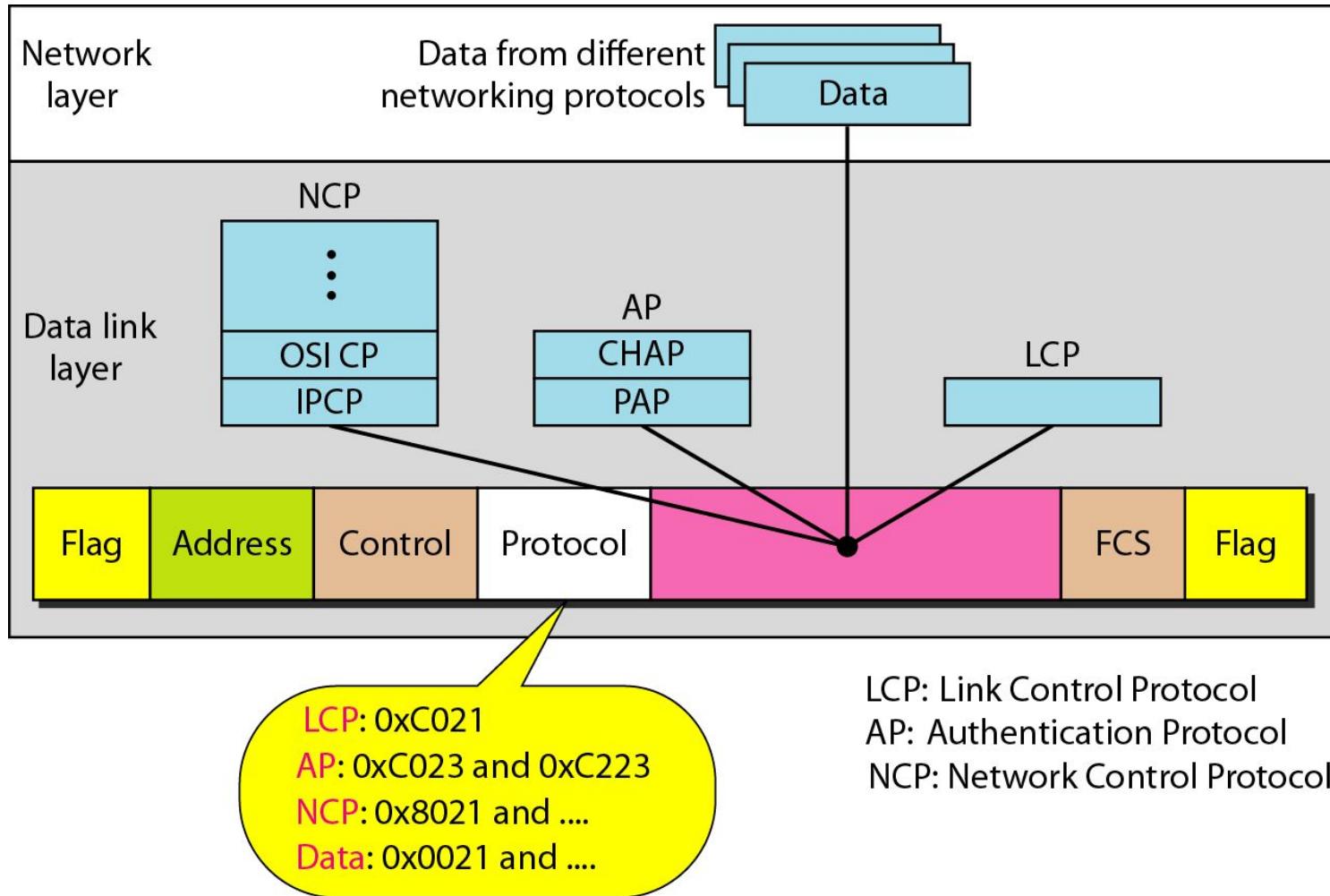


Figure 11.35 LCP packet encapsulated in a frame

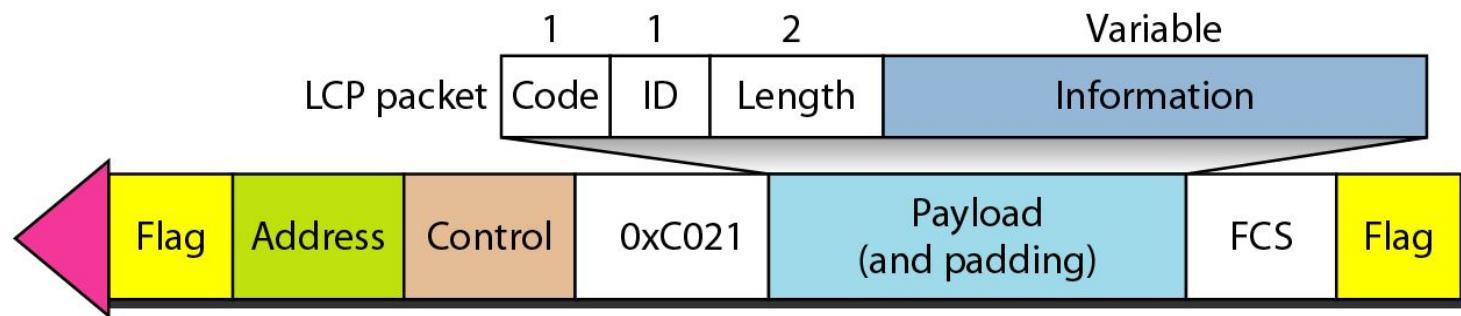


Table 11.2 *LCP packets*

<i>Code</i>	<i>Packet Type</i>	<i>Description</i>
0x01	Configure-request	Contains the list of proposed options and their values
0x02	Configure-ack	Accepts all options proposed
0x03	Configure-nak	Announces that some options are not acceptable
0x04	Configure-reject	Announces that some options are not recognized
0x05	Terminate-request	Request to shut down the line
0x06	Terminate-ack	Accept the shutdown request
0x07	Code-reject	Announces an unknown code
0x08	Protocol-reject	Announces an unknown protocol
0x09	Echo-request	A type of hello message to check if the other end is alive
0x0A	Echo-reply	The response to the echo-request message
0x0B	Discard-request	A request to discard the packet

Table 11.3 *Common options*

<i>Option</i>	<i>Default</i>
Maximum receive unit (payload field size)	1500
Authentication protocol	None
Protocol field compression	Off
Address and control field compression	Off

Figure 11.36 PAP packets encapsulated in a PPP frame

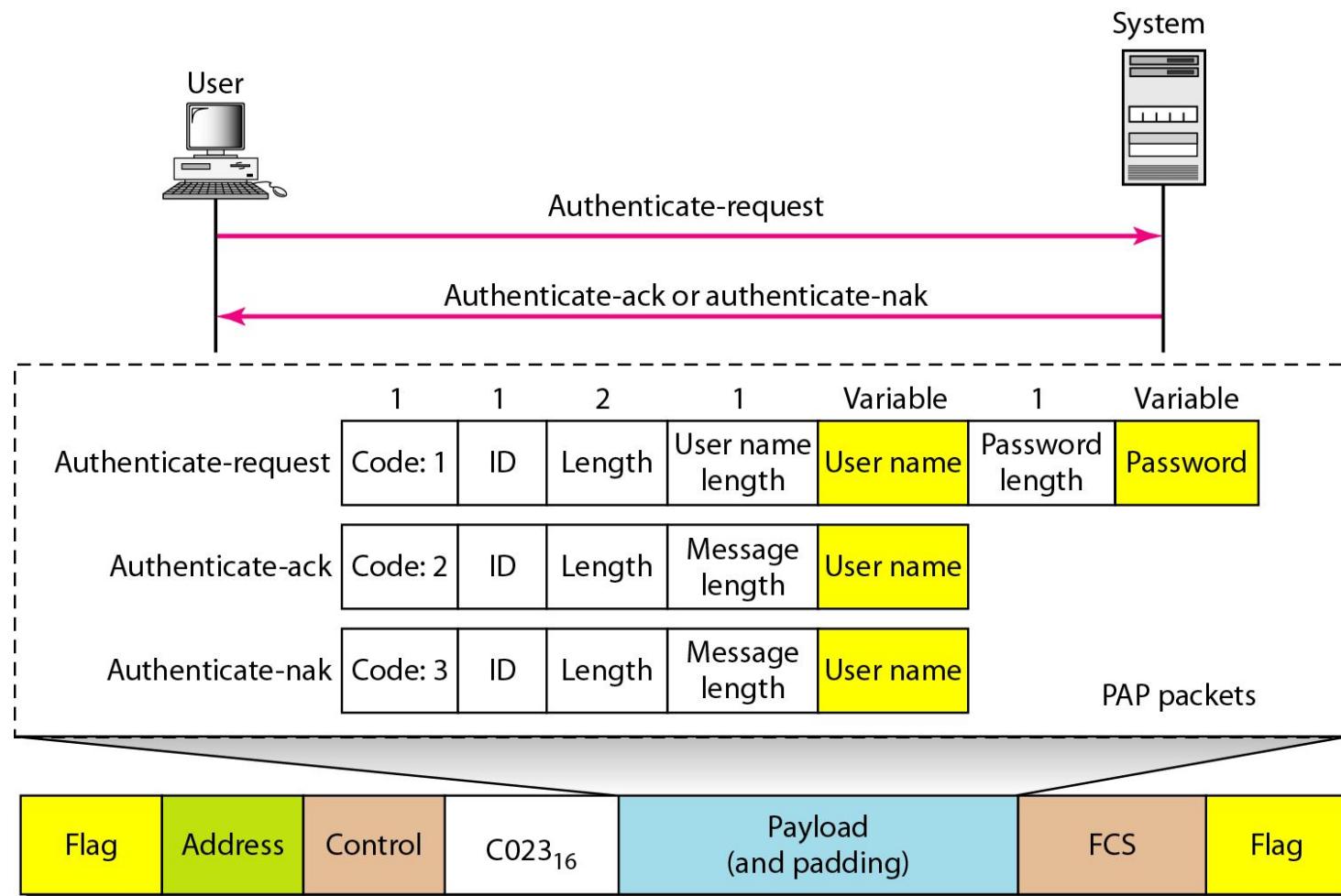


Figure 11.37 CHAP packets encapsulated in a PPP frame

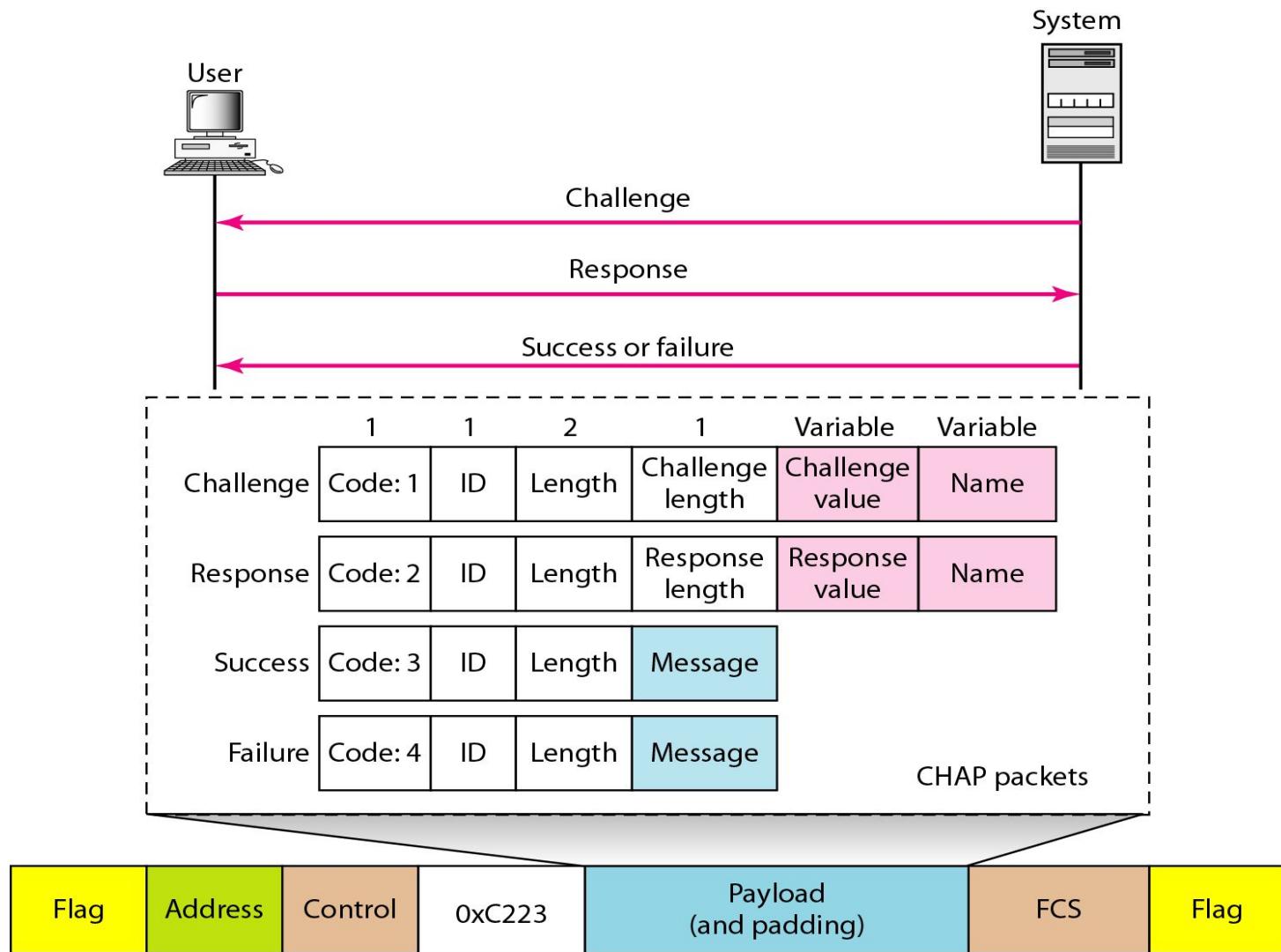


Figure 11.38 *IPCP packet encapsulated in PPP frame*

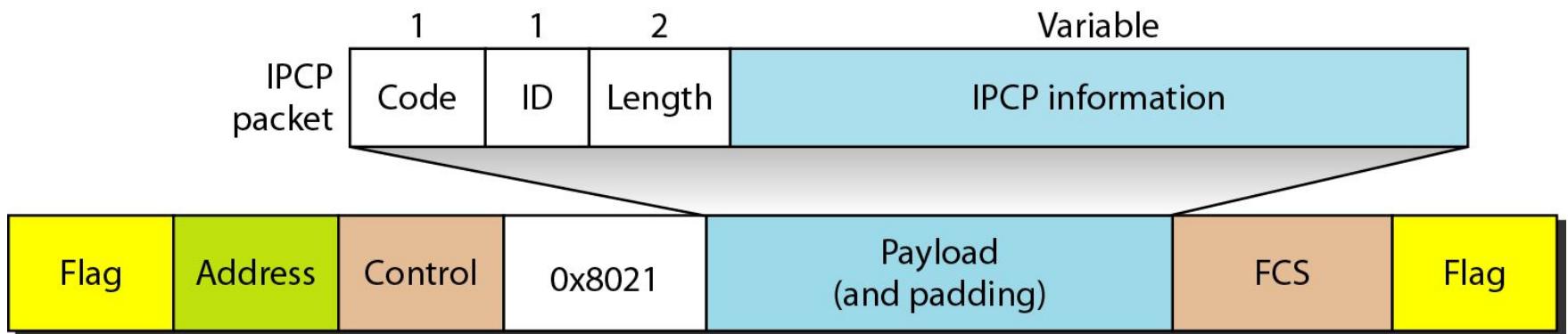


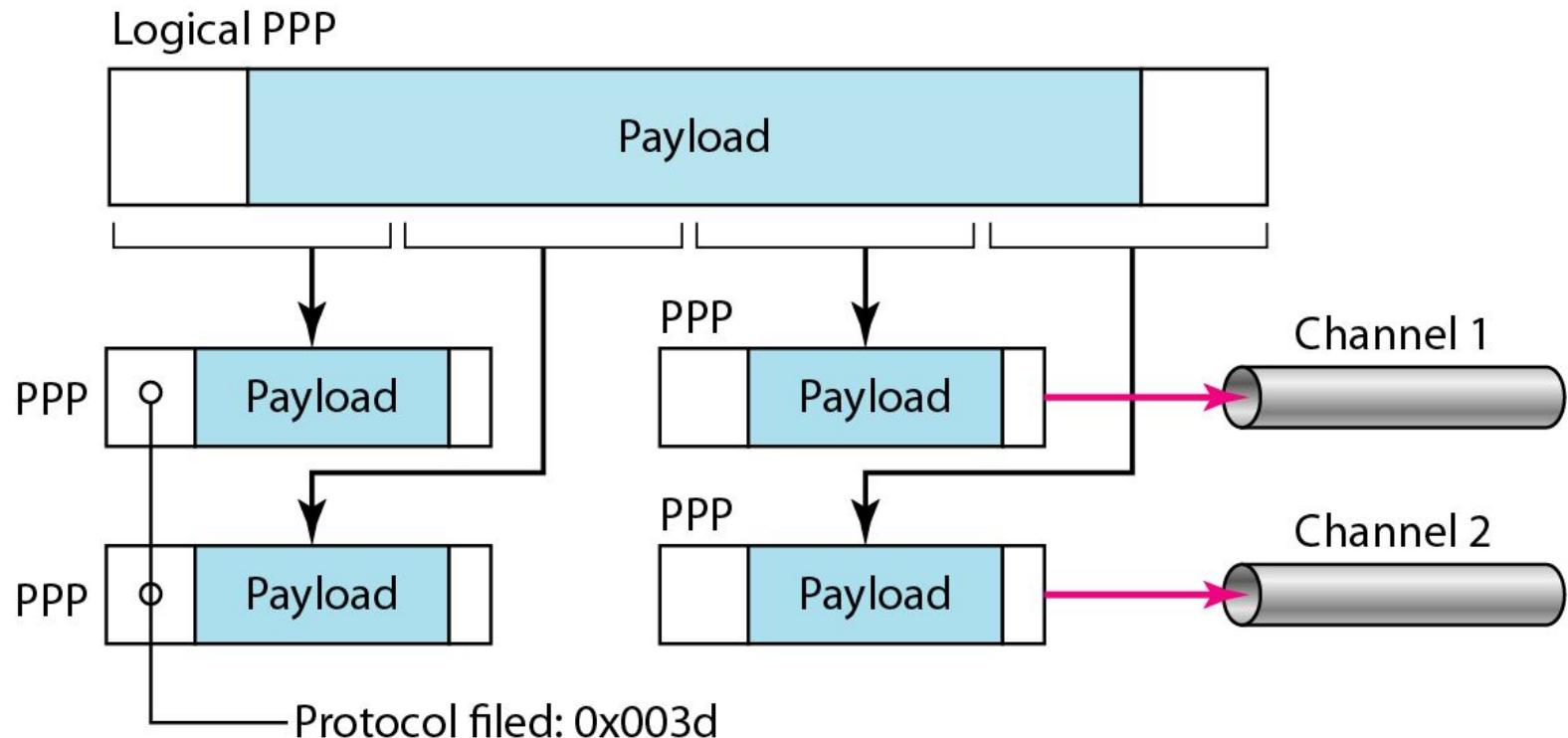
Table 11.4 *Code value for IPCP packets*

<i>Code</i>	<i>IPCP Packet</i>
0x01	Configure-request
0x02	Configure-ack
0x03	Configure-nak
0x04	Configure-reject
0x05	Terminate-request
0x06	Terminate-ack
0x07	Code-reject

Figure 11.39 *IP datagram encapsulated in a PPP frame*



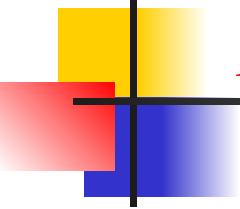
Figure 11.40 Multilink PPP



Example 11.12

Let us go through the phases followed by a network layer packet as it is transmitted through a PPP connection. Figure 11.41 shows the steps. For simplicity, we assume unidirectional movement of data from the user site to the system site (such as sending an e-mail through an ISP).

The first two frames show link establishment. We have chosen two options (not shown in the figure): using PAP for authentication and suppressing the address control fields. Frames 3 and 4 are for authentication. Frames 5 and 6 establish the network layer connection using IPCP.



Example 11.12 (continued)

The next several frames show that some IP packets are encapsulated in the PPP frame. The system (receiver) may have been running several network layer protocols, but it knows that the incoming data must be delivered to the IP protocol because the NCP protocol used before the data transfer was IPCP.

After data transfer, the user then terminates the data link connection, which is acknowledged by the system. Of course the user or the system could have chosen to terminate the network layer IPCP and keep the data link layer running if it wanted to run another NCP protocol.

Figure 11.41 An example

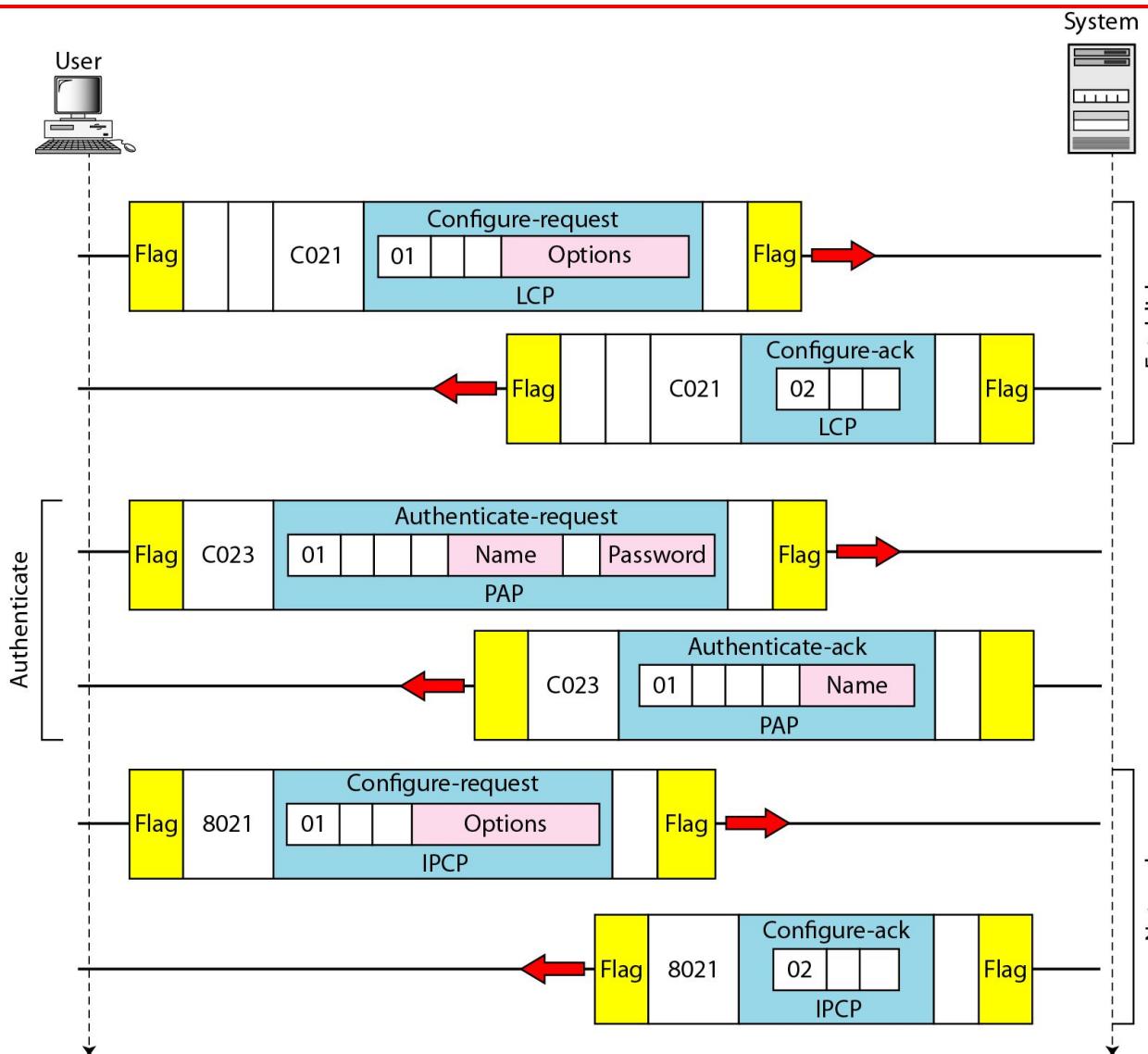


Figure 11.41 An example (continued)

