COMPUTER COMMUNICATION NETWORKS

(15EC64)

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*

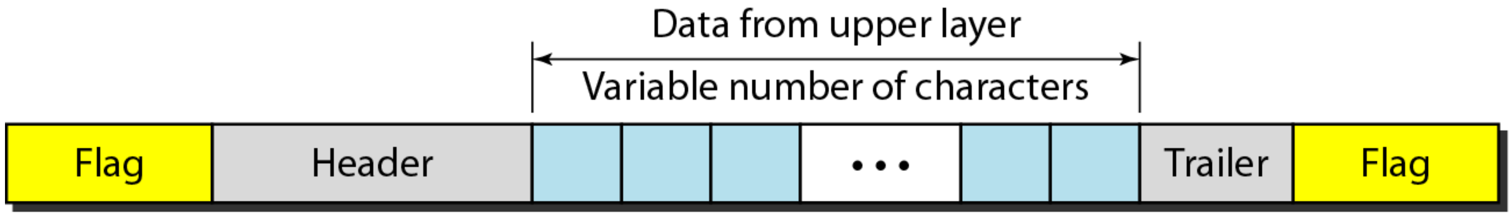
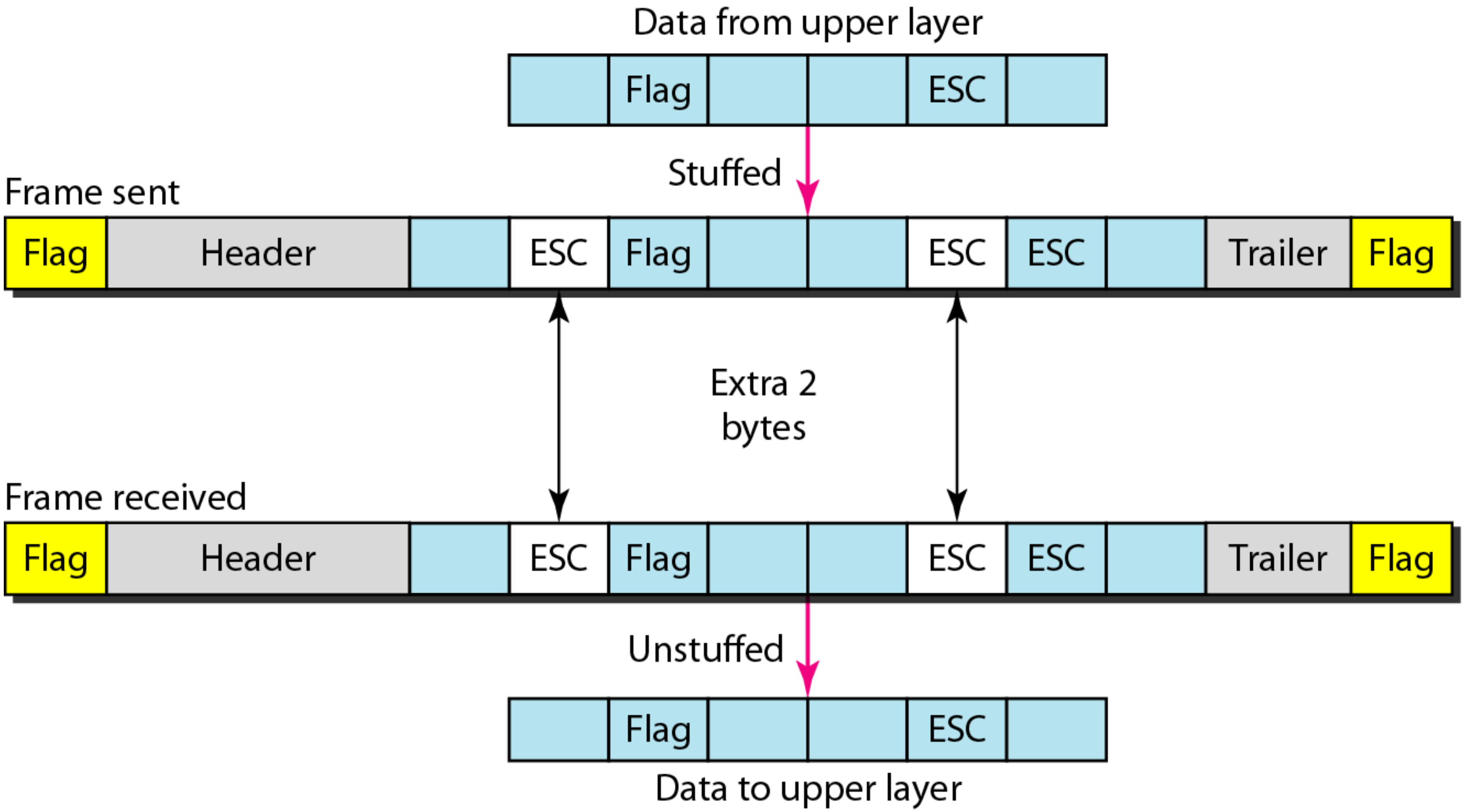


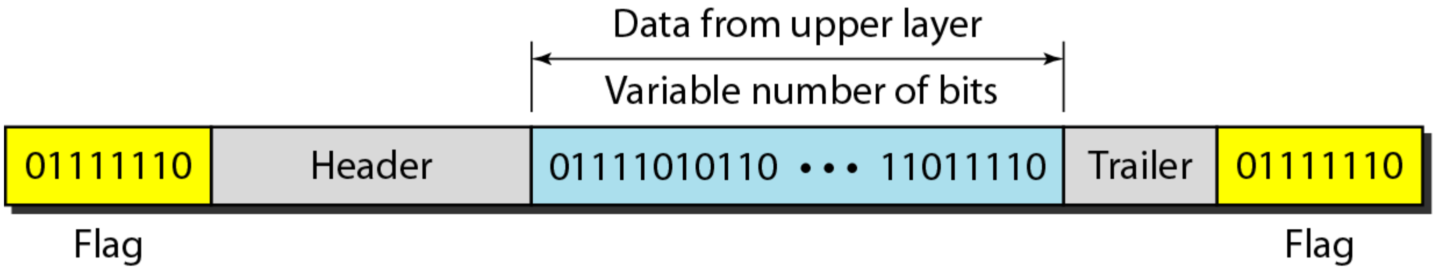
Figure 11.2 *Byte stuffing and unstuffing*



Note

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

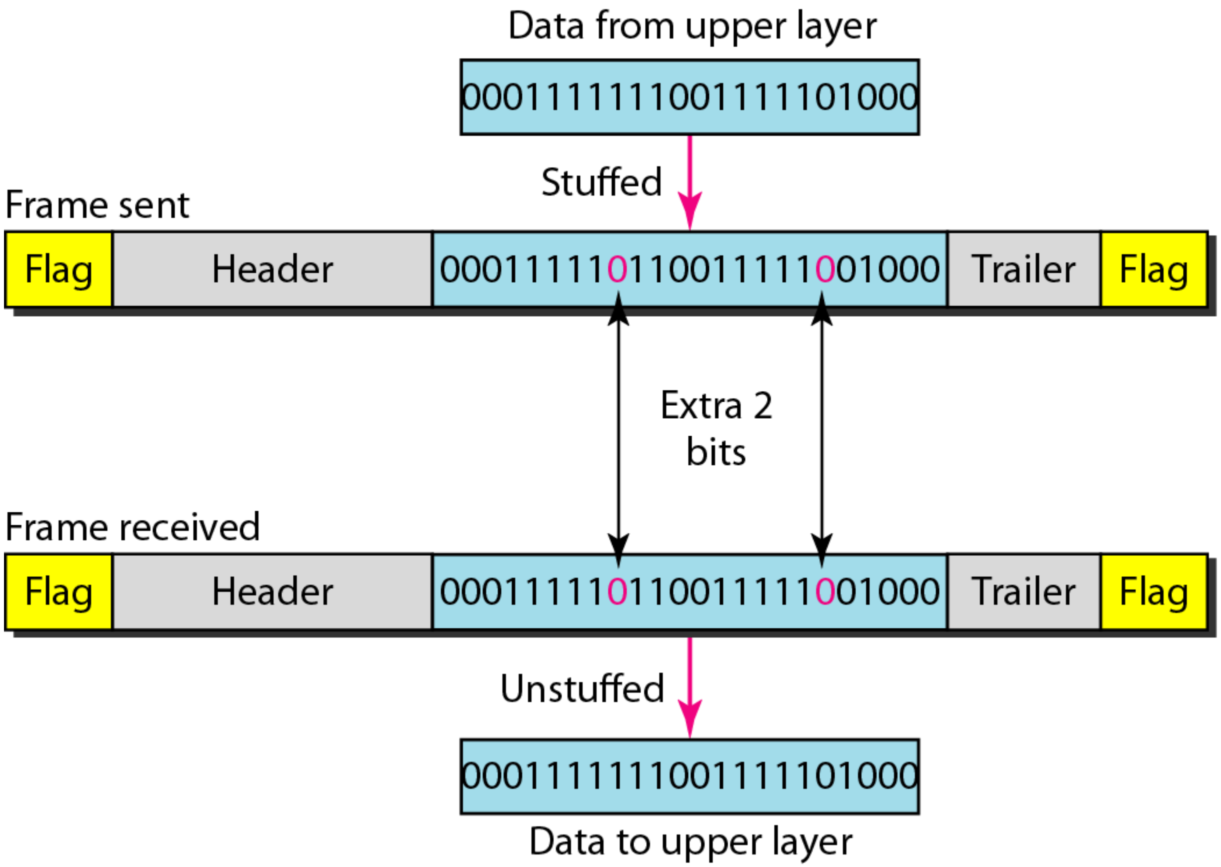
Figure 11.3 *A frame in a bit-oriented protocol*



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*



Data from upper layer

0001111111001111101000

Stuffed

Frame sent

Flag | Header | 000111110110011111001000 | Trailer | Flag

Extra 2 bits

Frame received

Flag | Header | 000111110110011111001000 | Trailer | Flag

Unstuffed

0001111111001111101000

Data to upper layer

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.

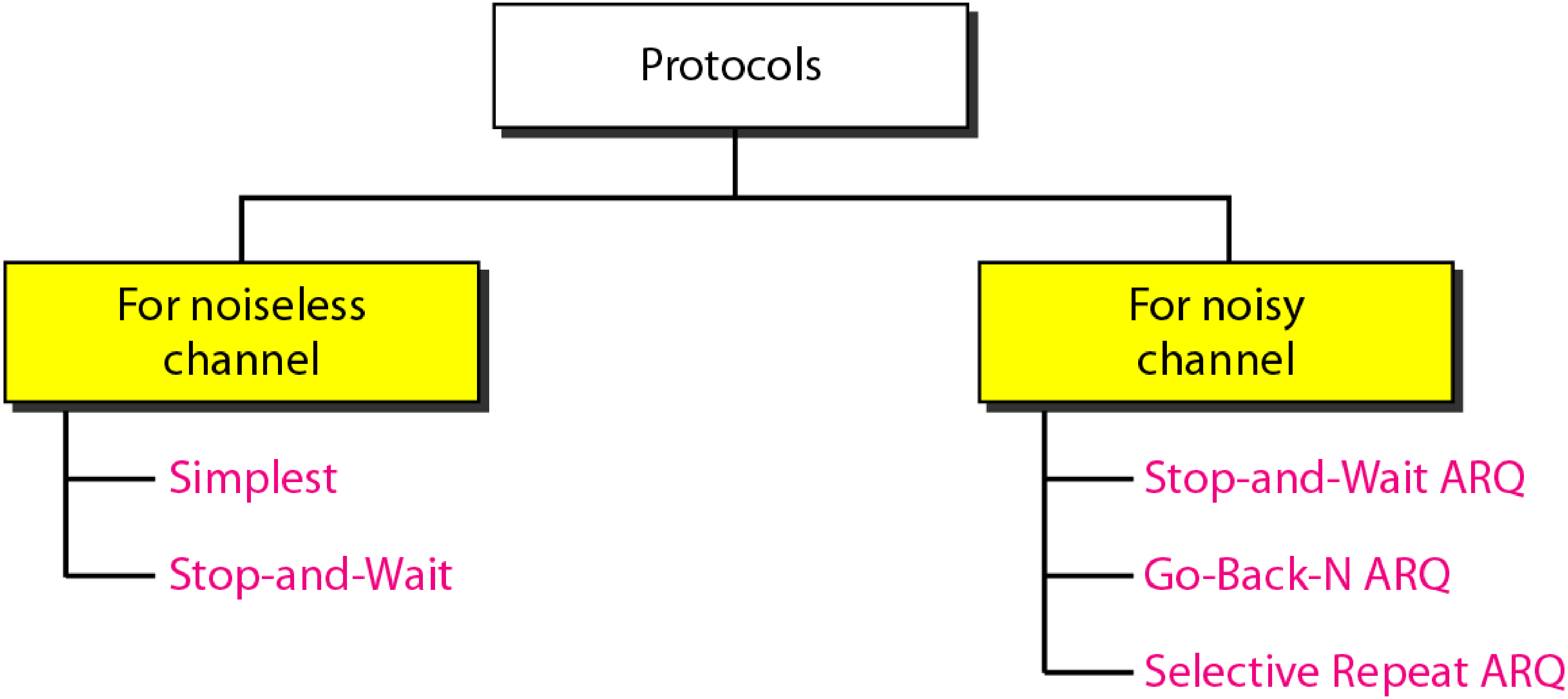
Note

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*



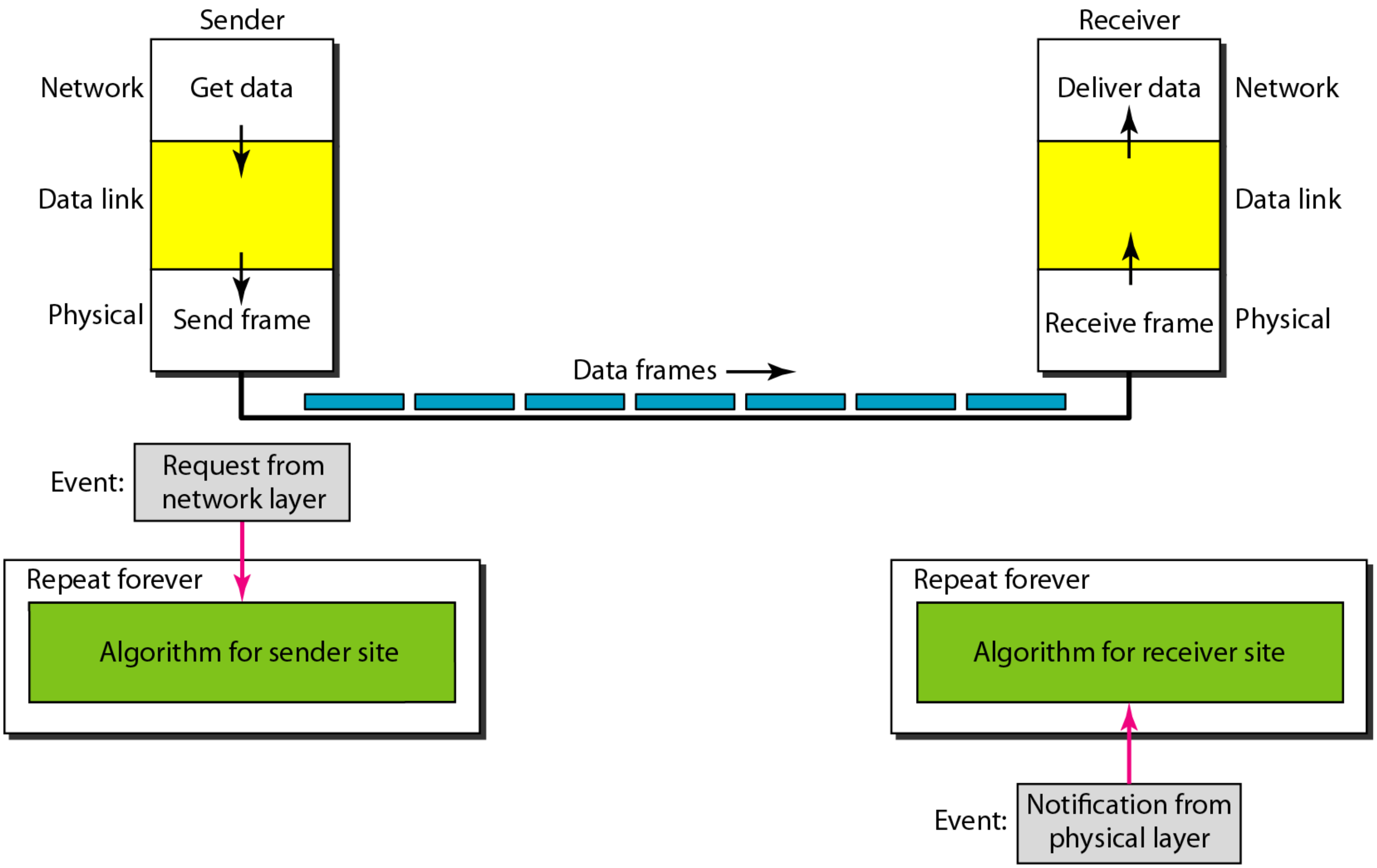
11-4 NOISELESS CHANNELS

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

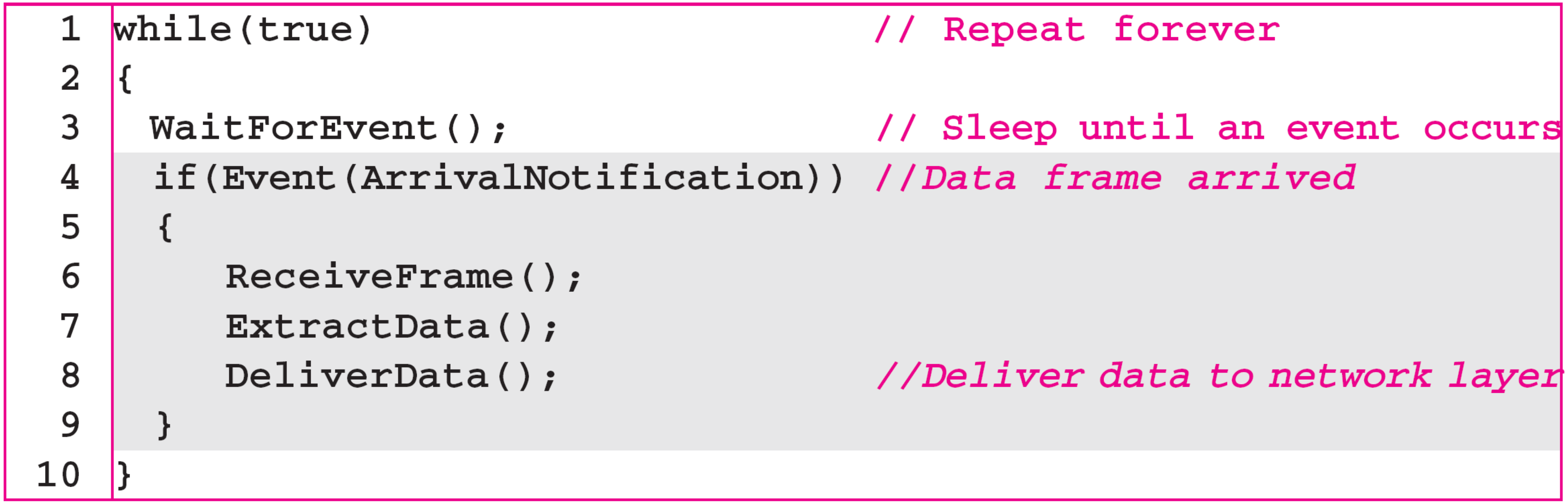
Figure 11.6 *The design of the simplest protocol with no flow or error control*



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

```
while(true)                    // Repeat forever
{
  WaitForEvent();              // Sleep until an event occurs
  if(Event(RequestToSend))     //There is a packet to send
  {
     GetData();
     MakeFrame();
     SendFrame();              //Send the frame
  }
}
```

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



```
while(true)                      // Repeat forever
{
  WaitForEvent();                // Sleep until an event occurs
  if(Event(ArrivalNotification)) //Data frame arrived
  {
      ReceiveFrame();
      ExtractData();
      DeliverData();             //Deliver data to network layer
  }
}
```

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*

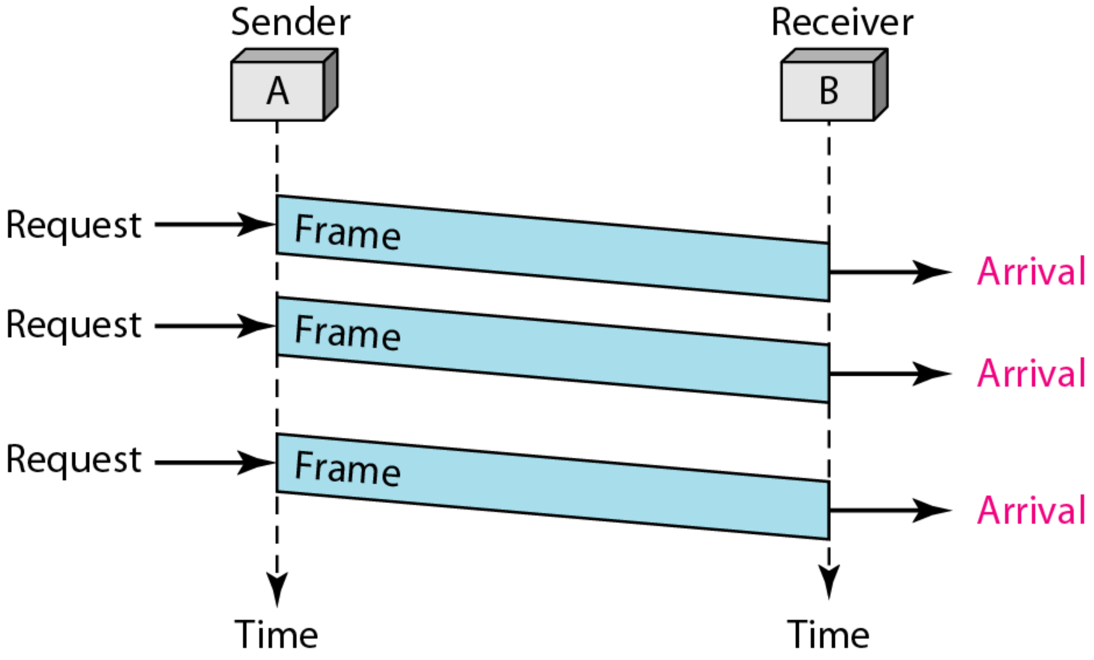
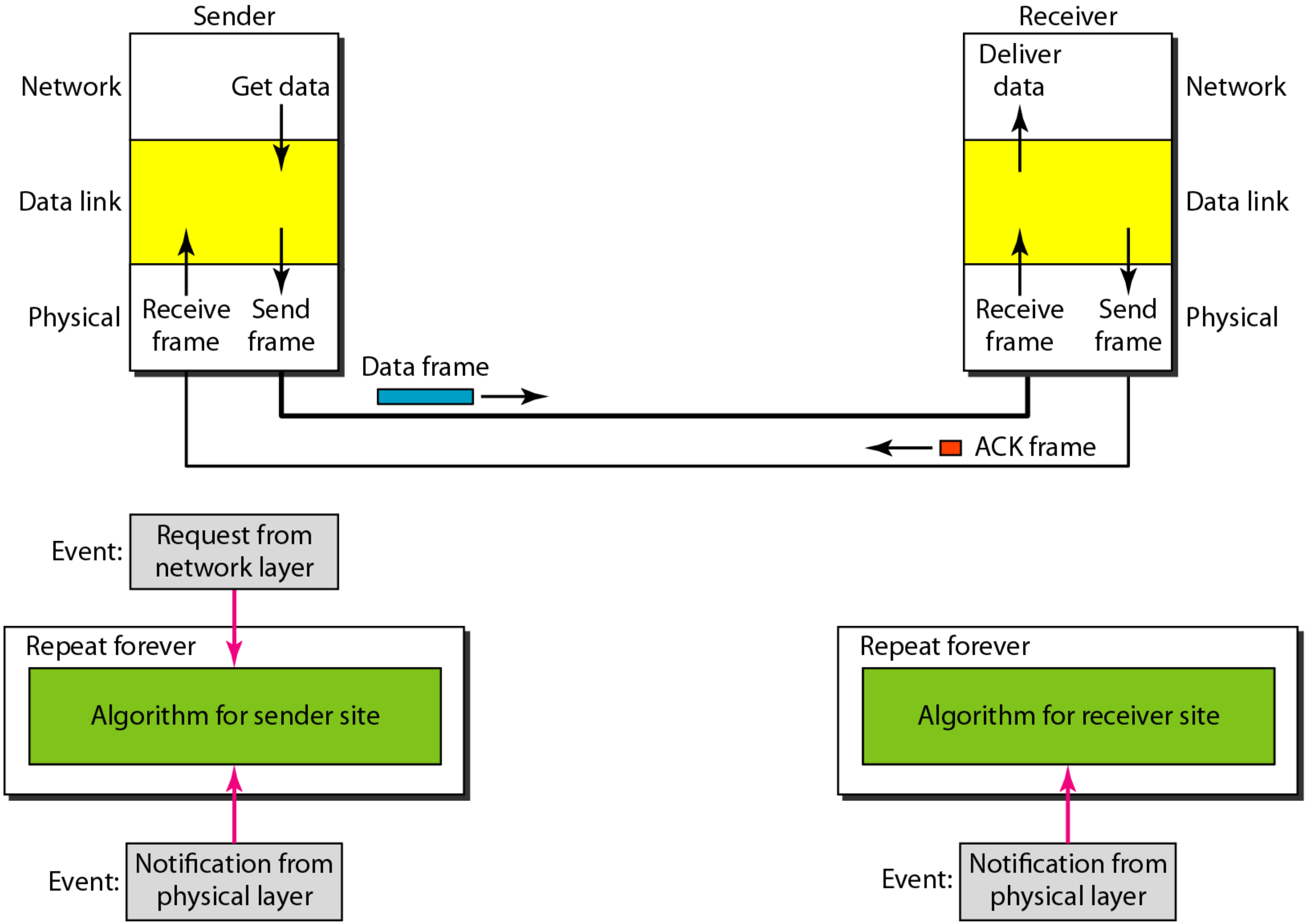


Figure 11.8 *Design of Stop-and-Wait Protocol*



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

```
while(true)                              //Repeat forever
canSend = true                           //Allow the first frame to go
{
  WaitForEvent();                        // Sleep until an event occurs
  if(Event(RequestToSend) AND canSend)
  {
      GetData();
      MakeFrame();
      SendFrame();                       //Send the data frame
      canSend = false;                   //Cannot send until ACK arrives
  }
  WaitForEvent();                        // Sleep until an event occurs
  if(Event(ArrivalNotification)          // An ACK has arrived
   {
      ReceiveFrame();                    //Receive the ACK frame
      canSend = true;
   }
}
```

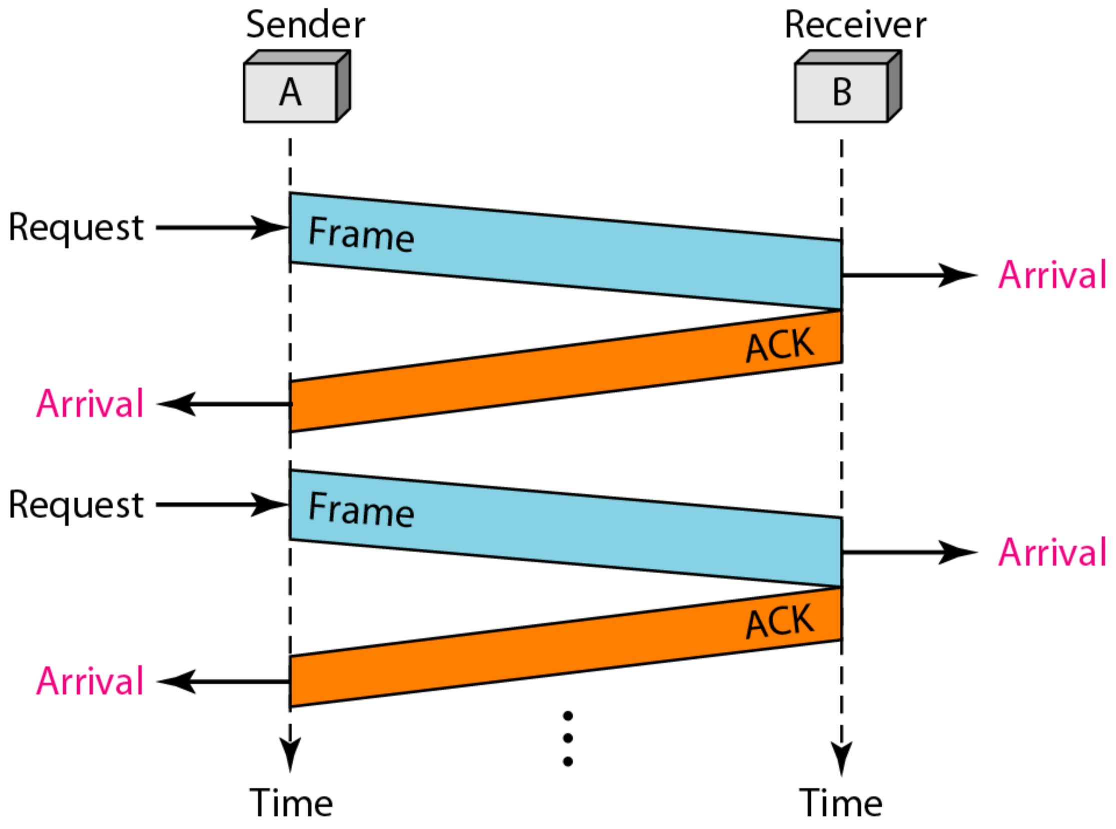
Algorithm 11.4 *Receiver-site algorithm for Stop-and-Wait Protocol*

```
while(true)                      //Repeat forever
{
  WaitForEvent();                // Sleep until an event occurs
  if(Event(ArrivalNotification)) //Data frame arrives
  {
      ReceiveFrame();
      ExtractData();
      Deliver(data);             //Deliver data to network layer
      SendFrame();               //Send an ACK frame
  }
}
```

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.

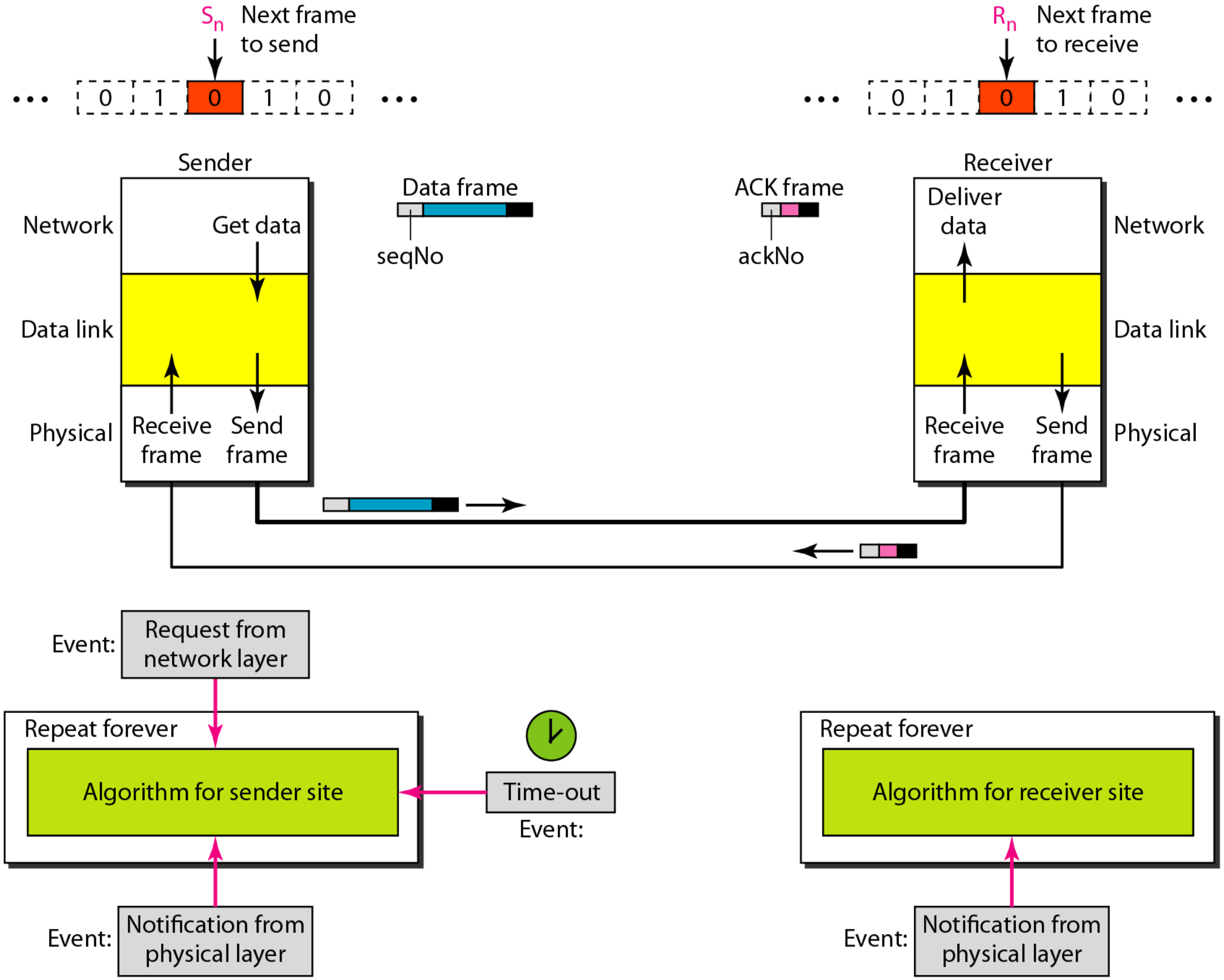
Note

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

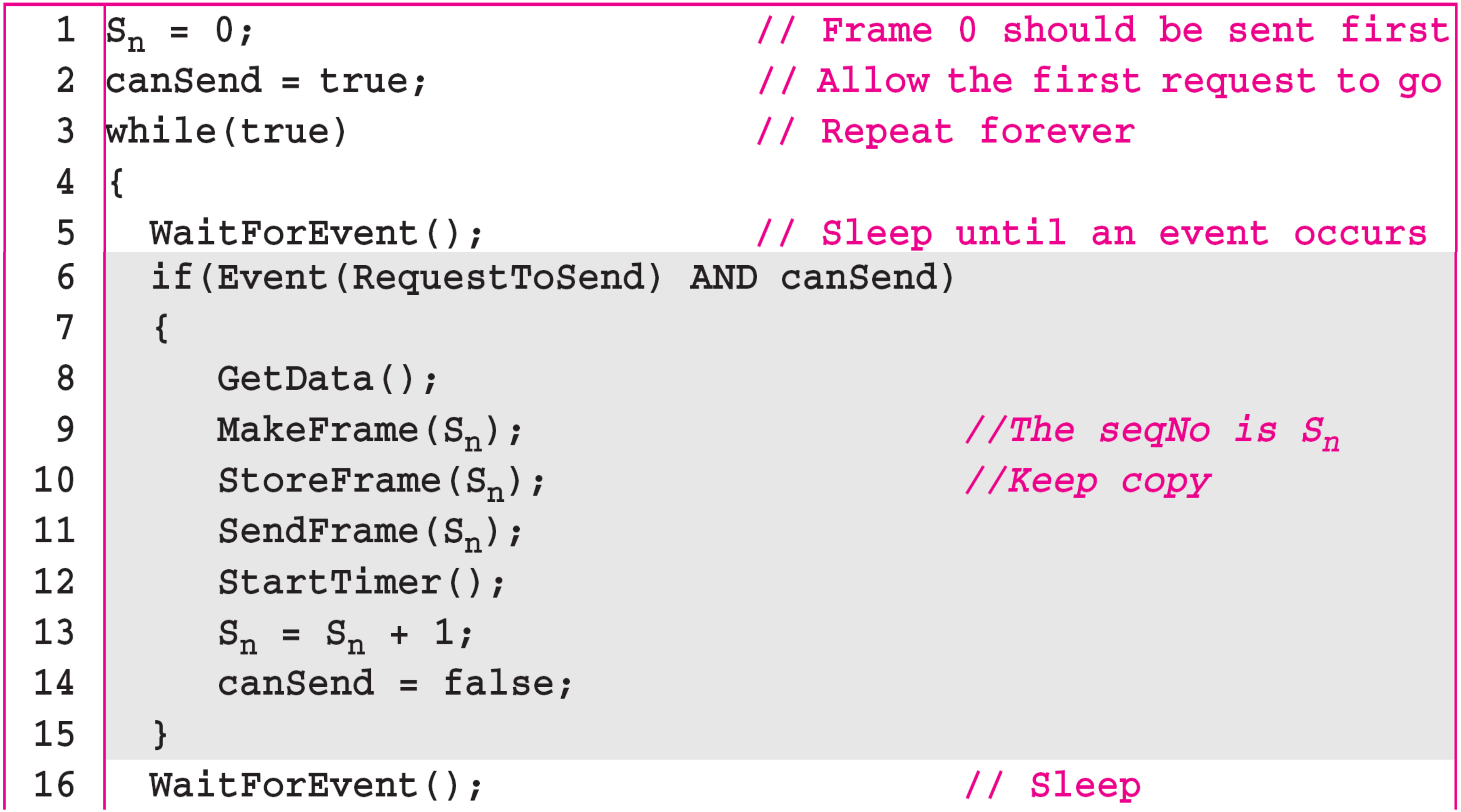
Note

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*



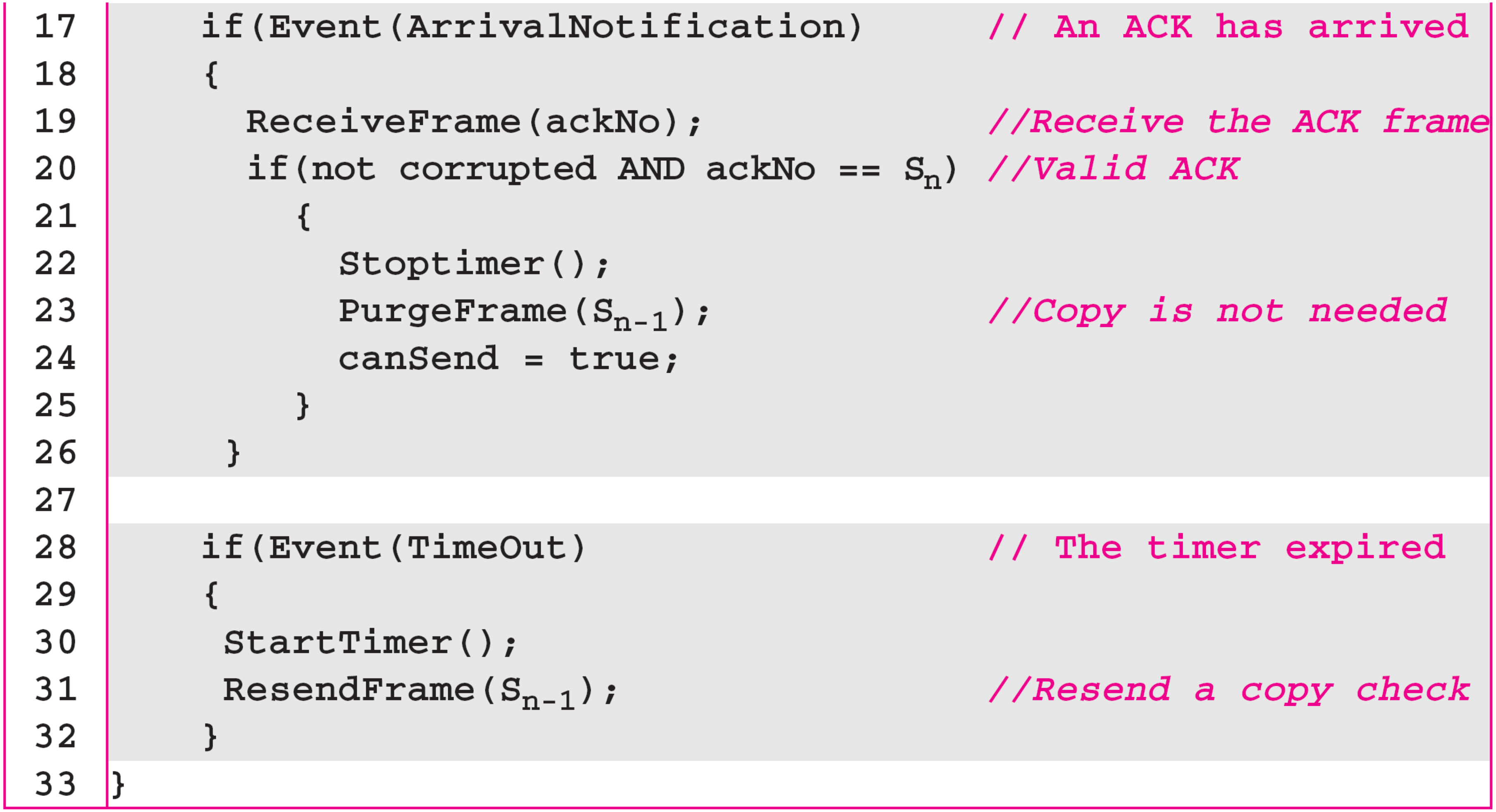
Algorithm 11.5 *Sender-site algorithm for Stop-and-Wait ARQ*



```
$S_n$ = 0;                                 // Frame 0 should be sent first
canSend = true;                            // Allow the first request to go
while(true)                                // Repeat forever
{
  WaitForEvent();                          // Sleep until an event occurs
  if(Event(RequestToSend) AND canSend)
  {
     GetData();
     MakeFrame($S_n$);                              //The seqNo is $S_n$
     StoreFrame($S_n$);                             //Keep copy
     SendFrame($S_n$);
     StartTimer();
     $S_n$ = $S_n$ + 1;
     canSend = false;
  }
  WaitForEvent();                                   // Sleep
```

(continued)

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



```
    if(Event(ArrivalNotification)                // An ACK has arrived
    {
      ReceiveFrame(ackNo);                       //Receive the ACK frame
      if(not corrupted AND ackNo == $S_n$)         //Valid ACK
        {
          Stoptimer();
          PurgeFrame($S_{n-1}$);                     //Copy is not needed
          canSend = true;
        }
     }

    if(Event(TimeOut)                            // The timer expired
    {
     StartTimer();
     ResendFrame($S_{n-1}$);                         //Resend a copy check
    }
}
```

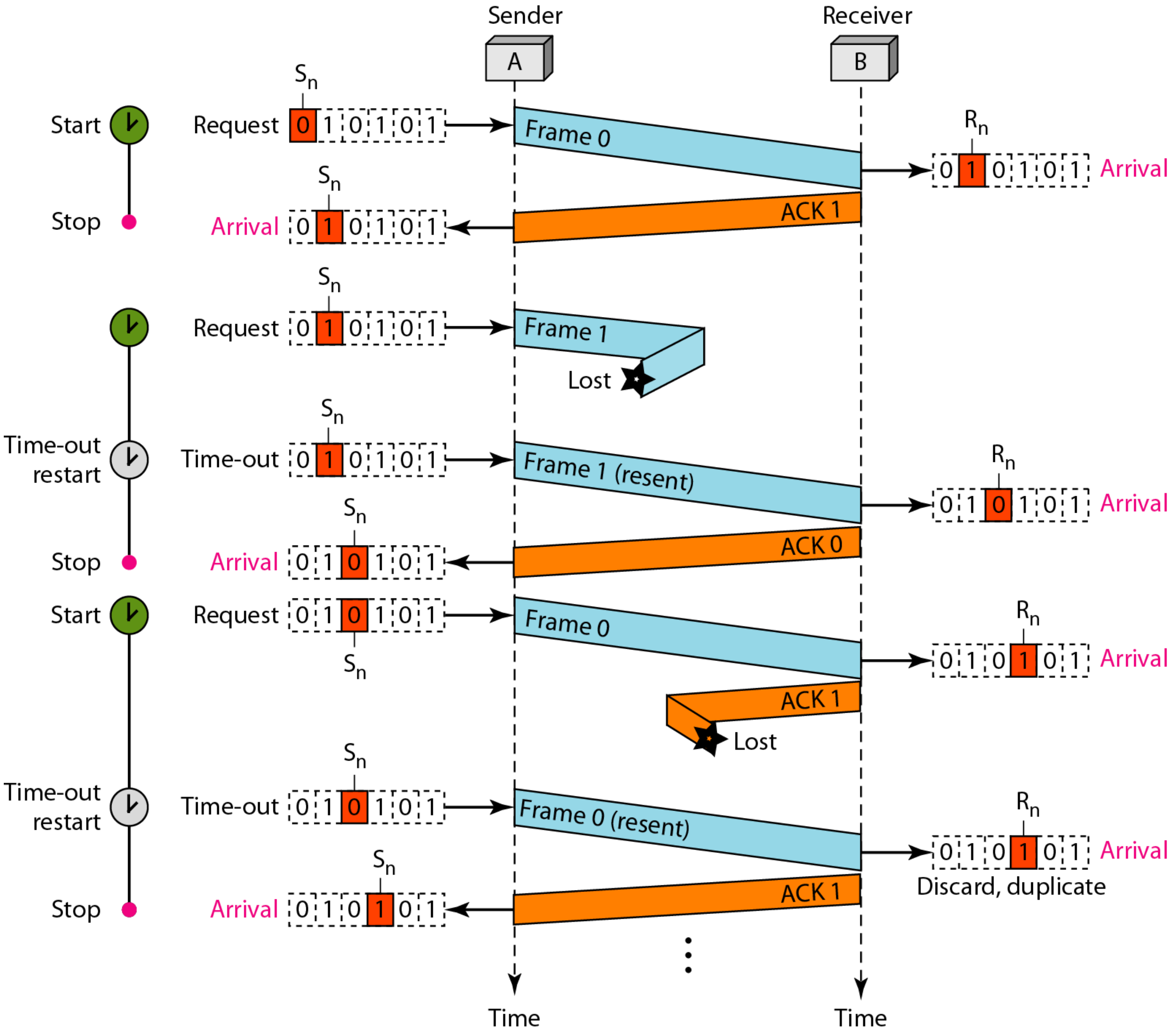
Algorithm 11.6 *Receiver-site algorithm for Stop-and-Wait ARQ Protocol*

```
$R_n$ = 0;                 // Frame 0 expected to arrive first
while(true)
{
  WaitForEvent();          // Sleep until an event occurs
  if(Event(ArrivalNotification))   //Data frame arrives
  {
     ReceiveFrame();
     if(corrupted(frame));
        sleep();
     if(seqNo == $R_n$)                   //Valid data frame
     {
      ExtractData();
       DeliverData();                  //Deliver data
       $R_n$ = $R_n$ + 1;
     }
      SendFrame($R_n$);                   //Send an ACK
  }
}
```

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

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

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?

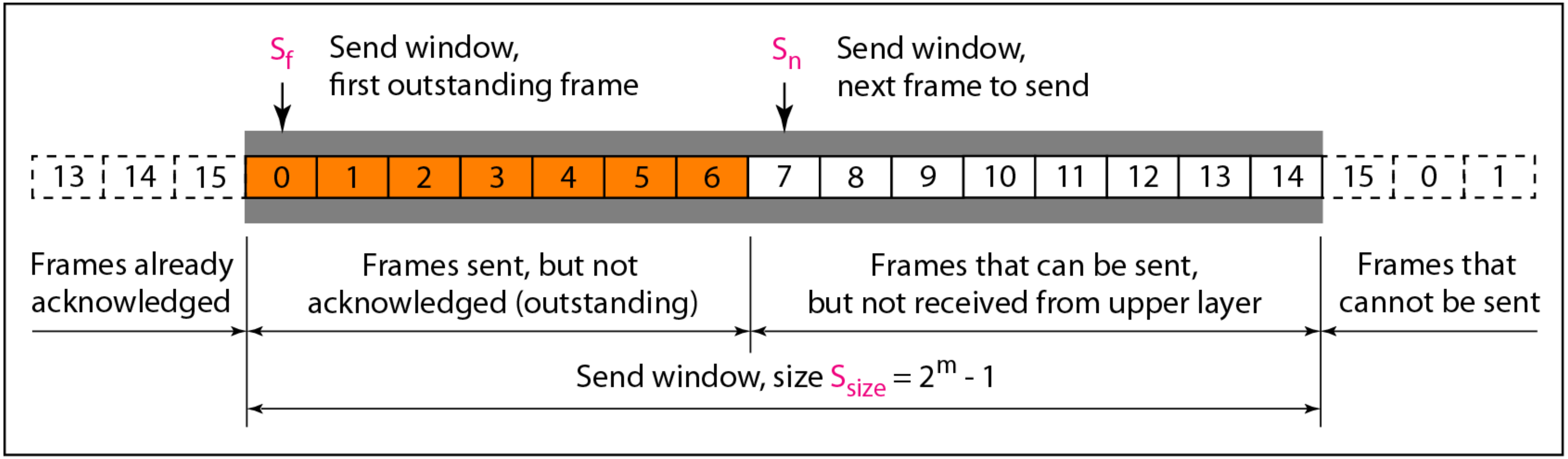
Solution

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.

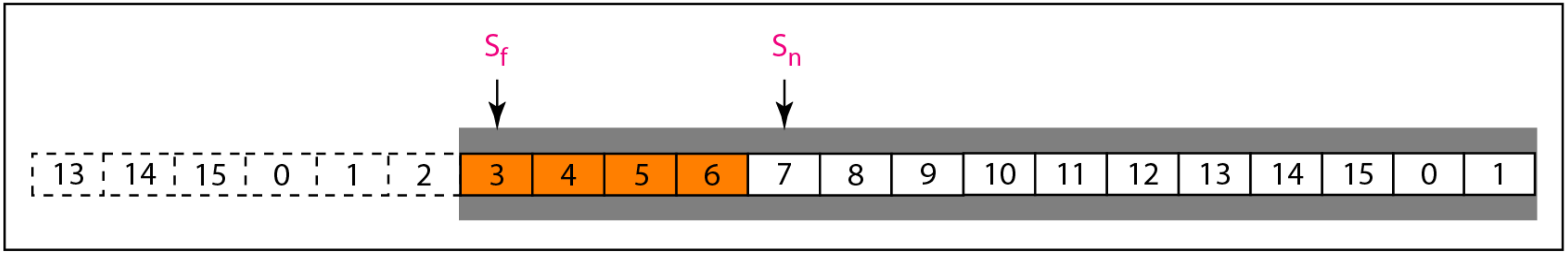
Note

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*


S_f
Send window, first outstanding frame
S_n
Send window, next frame to send
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1
Frames already acknowledged
Frames sent, but not acknowledged (outstanding)
Frames that can be sent, but not received from upper layer
Frames that cannot be sent
Send window, size $S_{size} = 2^m - 1$

a. Send window before sliding



b. Send window after sliding

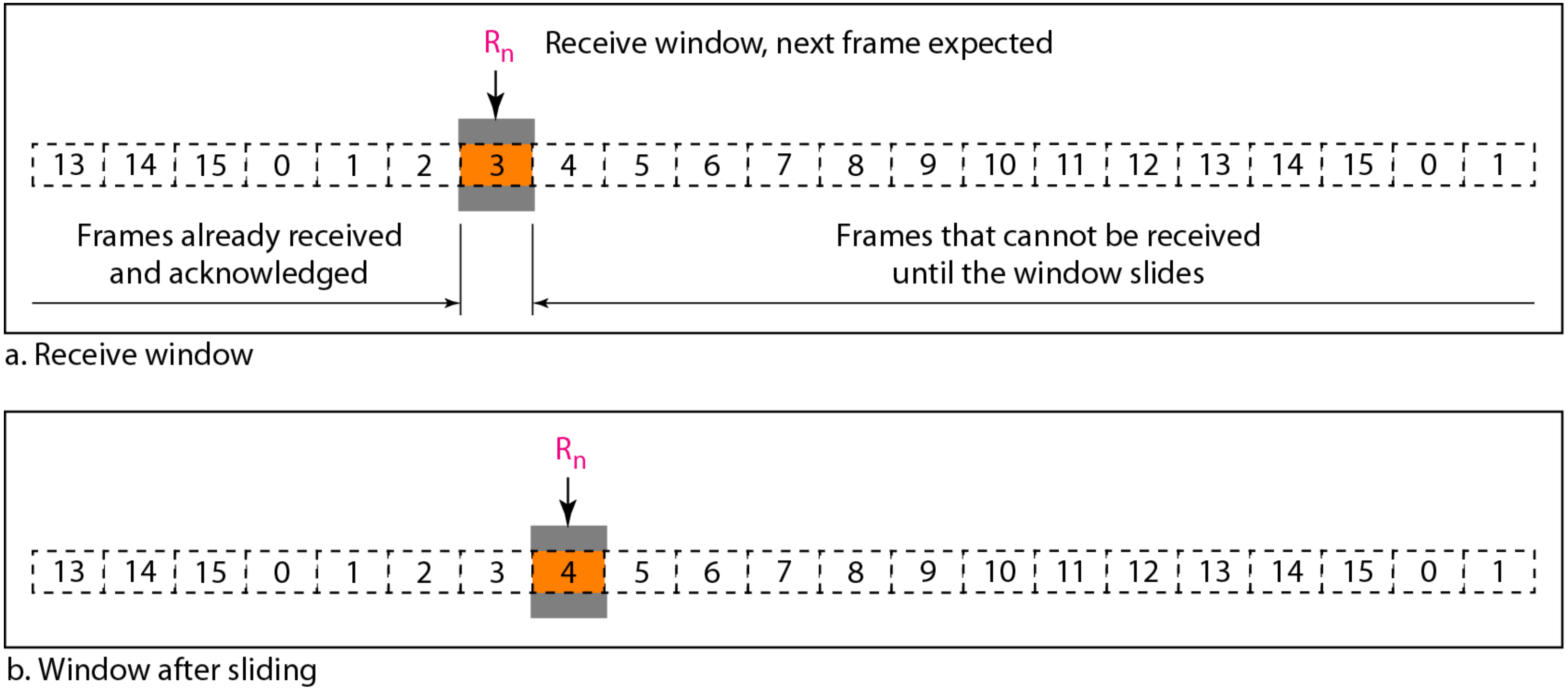
Note

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*


R_n
Receive window, next frame expected
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1
Frames already received and acknowledged
Frames that cannot be received until the window slides
R_n
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1

a. Receive window

b. Window after sliding

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*

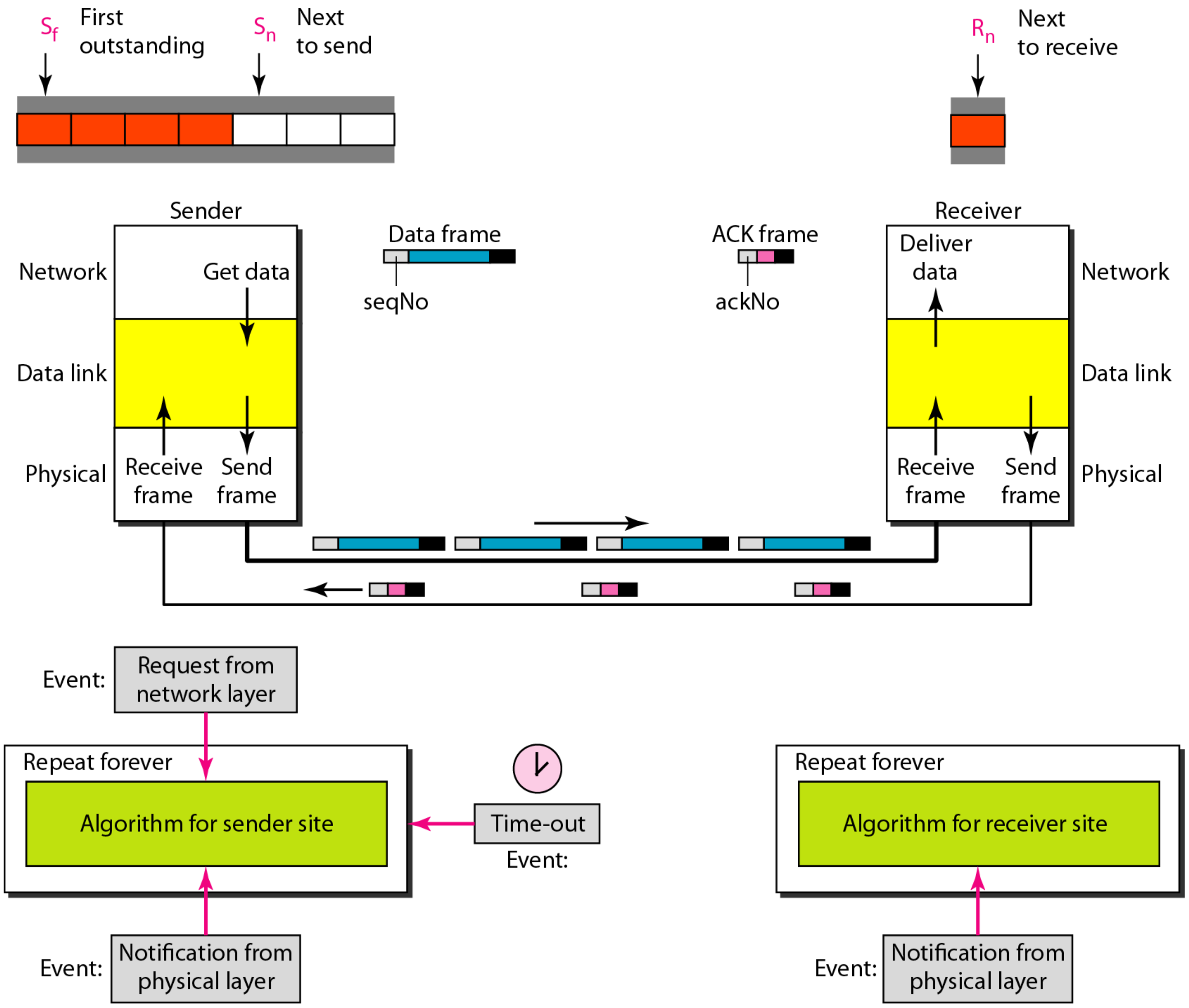
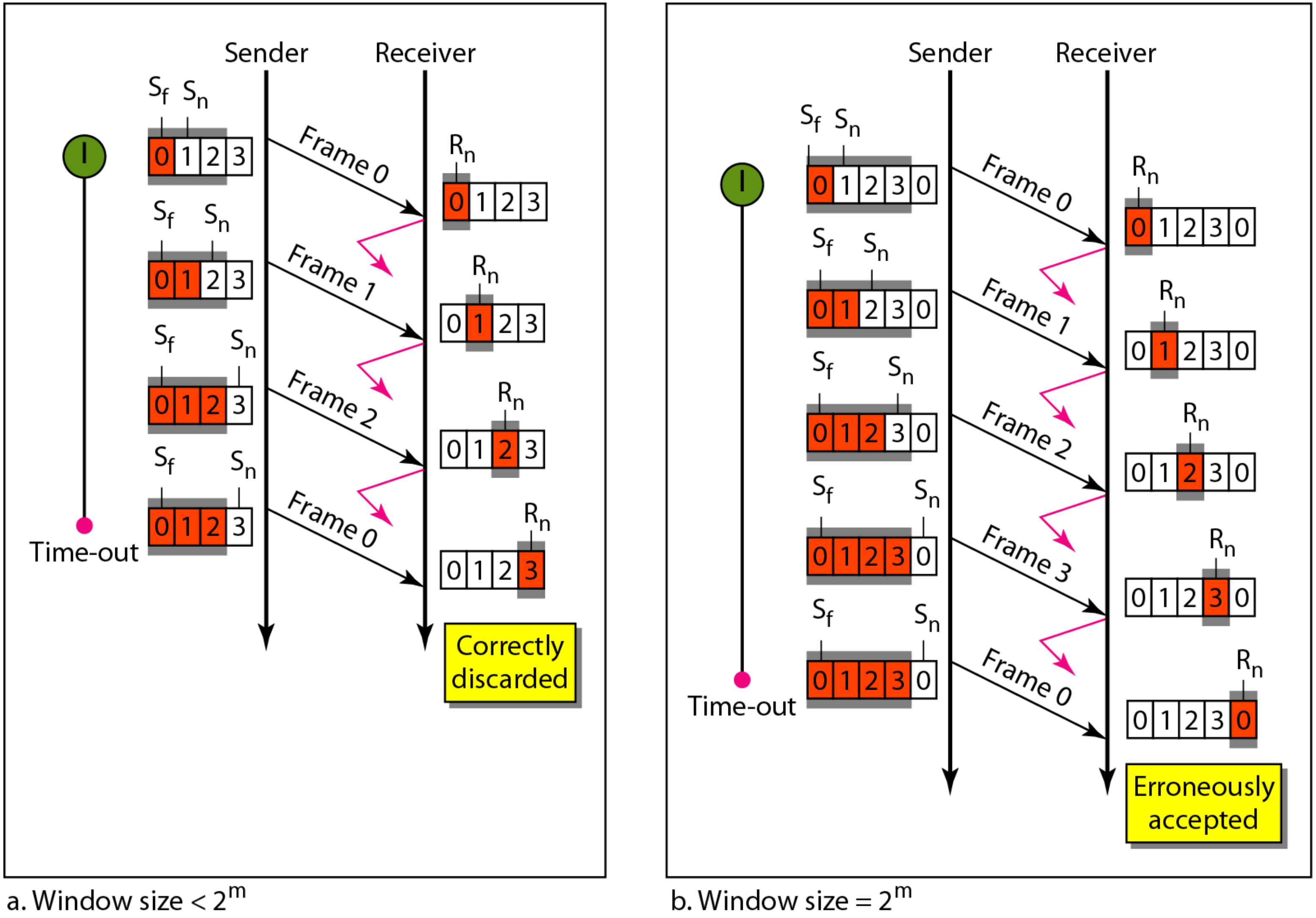


Figure 11.15 *Window size for Go-Back-N ARQ*



a. Window size < 2^m

b. Window size = 2^m

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*

```
$S_w = 2^m - 1$;
$S_f = 0$;
$S_n = 0$;

while (true)                                   //Repeat forever
{
 WaitForEvent();
  if(Event(RequestToSend))                     //A packet to send
  {
     if($S_n - S_f >= S_w$)                    //If window is full
           Sleep();
     GetData();
     MakeFrame($S_n$);
     StoreFrame($S_n$);
     SendFrame($S_n$);
     $S_n = S_n + 1$;
     if(timer not running)
          StartTimer();
  }

```

(continued)

Algorithm 11.7 *Go-Back-N sender algorithm* *(continued)*

```
  if(Event(ArrivalNotification))  //ACK arrives
  {
     Receive(ACK);
     if(corrupted(ACK))
          Sleep();
     if((ackNo>$S_f$)&&(ackNo<=$S_n$))  //If a valid ACK
     While($S_f$ <= ackNo)
      {
       PurgeFrame($S_f$);
       $S_f$ = $S_f$ + 1;
      }
      StopTimer();
  }

  if(Event(TimeOut))              //The timer expires
  {
   StartTimer();
   Temp = $S_f$;
   while(Temp < $S_n$);
    {
     SendFrame($S_f$);
     $S_f$ = $S_f$ + 1;
    }
  }
}
```

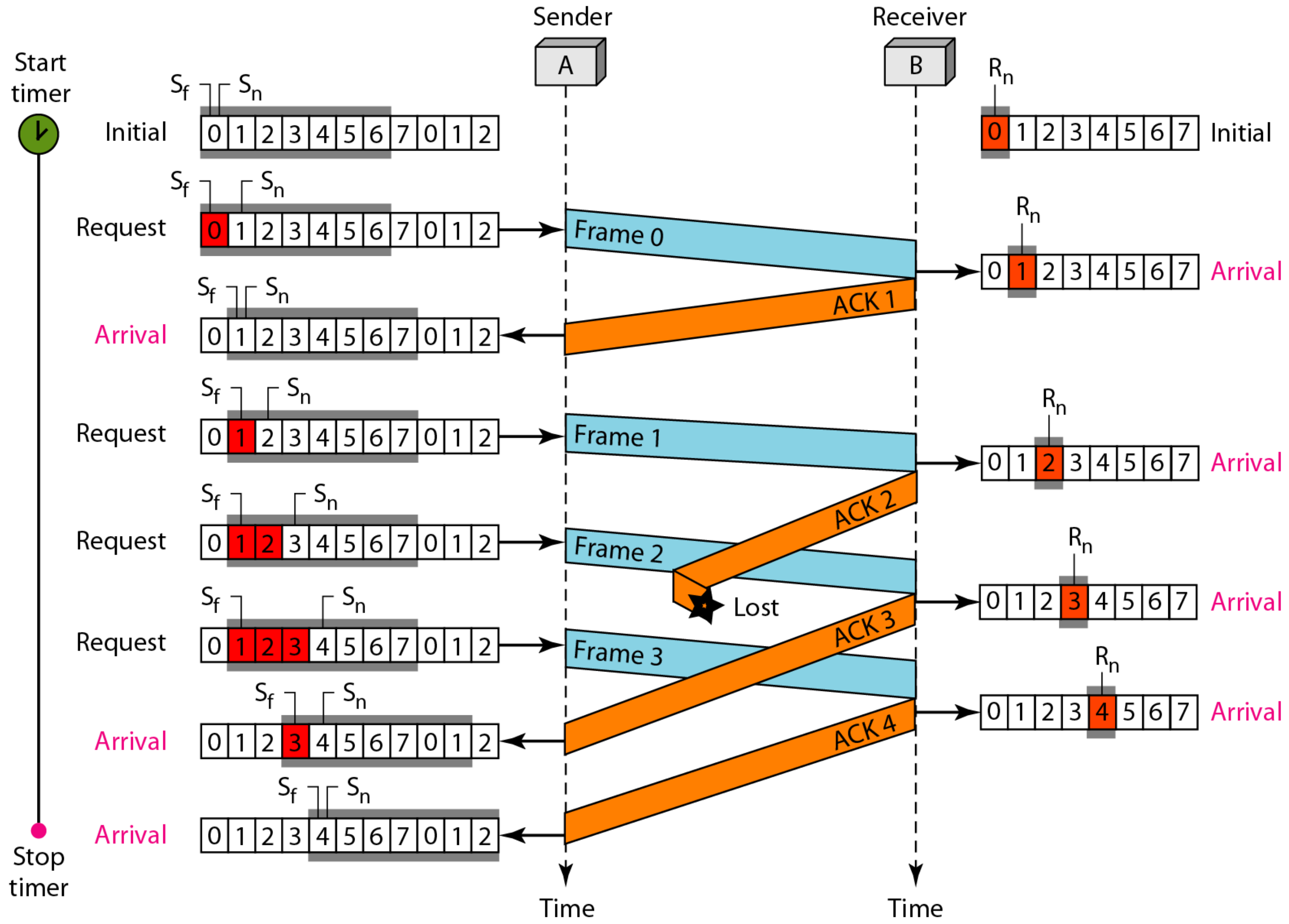
Algorithm 11.8 *Go-Back-N receiver algorithm*

```
$R_n$ = 0;

while (true)                      //Repeat forever
{
  WaitForEvent();

  if(Event(ArrivalNotification)) /Data frame arrives
  {
     Receive(Frame);
     if(corrupted(Frame))
          Sleep();
     if(seqNo == $R_n$)           //If expected frame
     {
       DeliverData();             //Deliver data
       $R_n$ = $R_n$ + 1;         //Slide window
       SendACK($R_n$);
     }
  }
}
```

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*



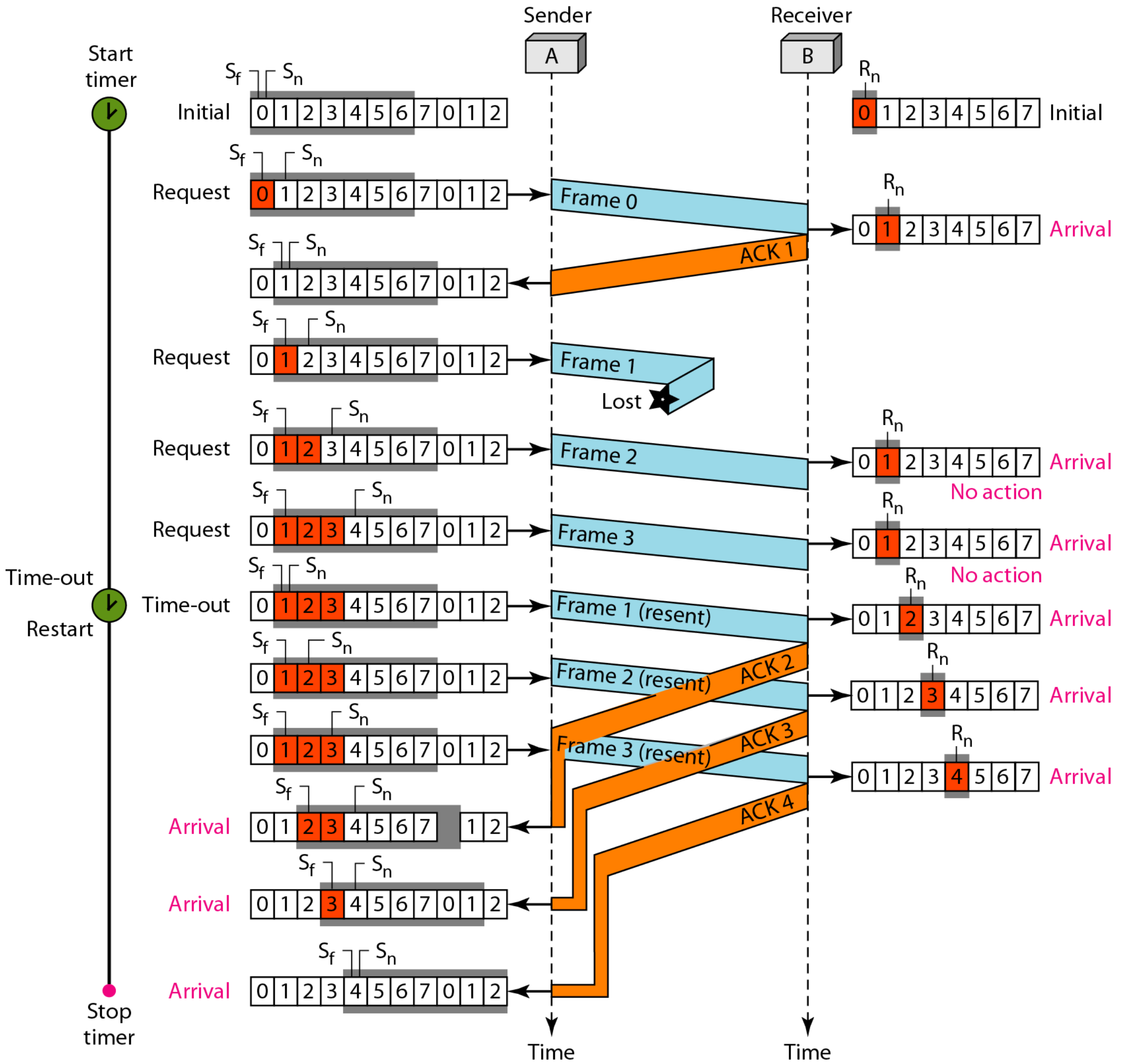
Example 11.7

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*

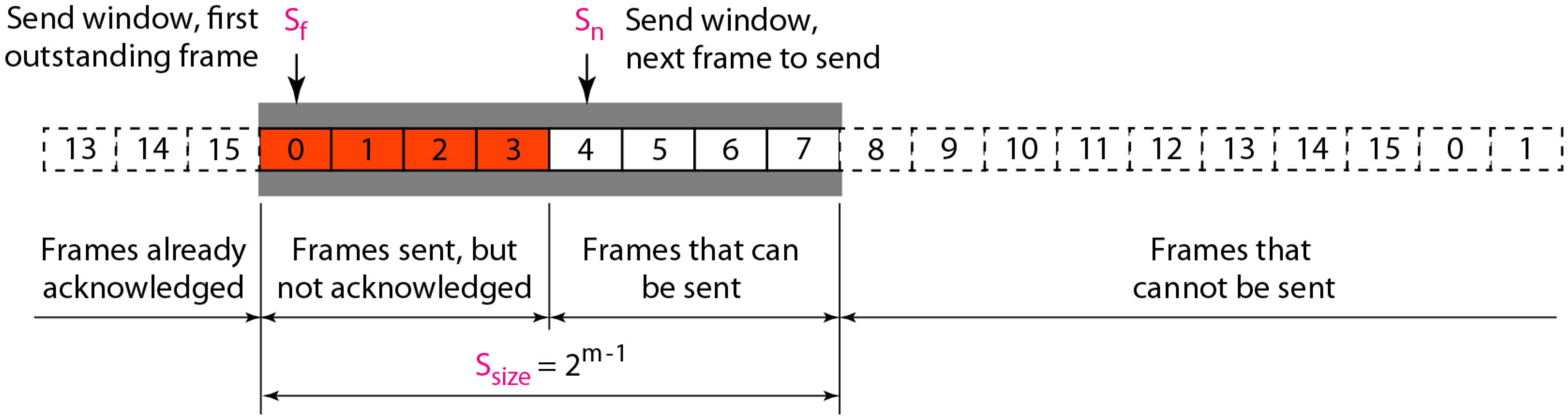

Send window, first outstanding frame
S_f
S_n
Send window, next frame to send
13
14
15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
Frames already acknowledged
Frames sent, but not acknowledged
Frames that can be sent
Frames that cannot be sent
$S_{size} = 2^{m-1}$

Figure 11.19 *Receive window for Selective Repeat ARQ*

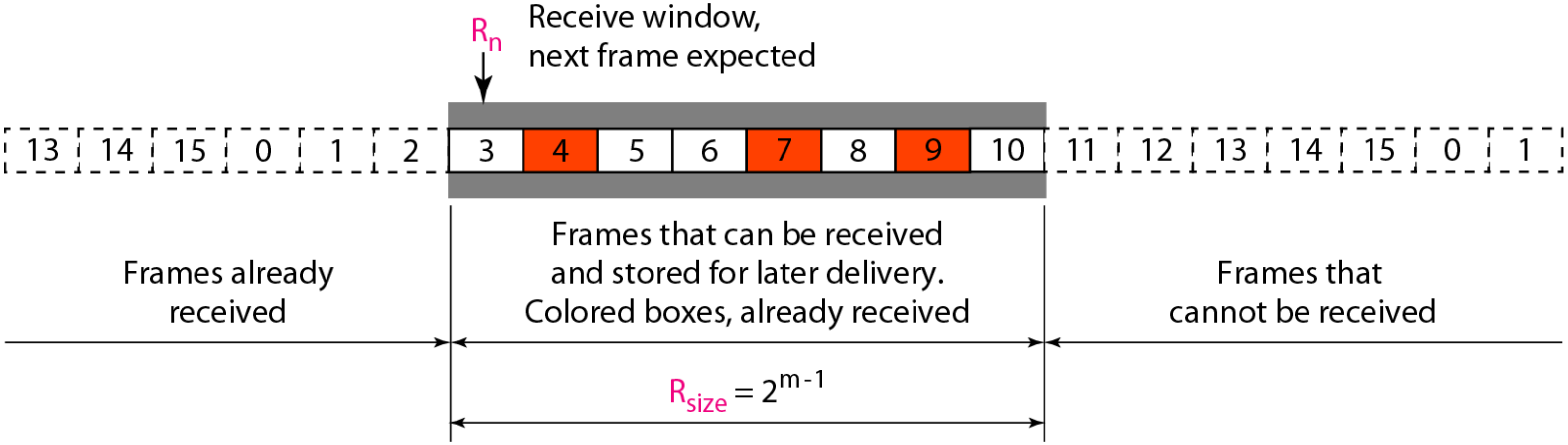


Figure 11.20 *Design of Selective Repeat ARQ*

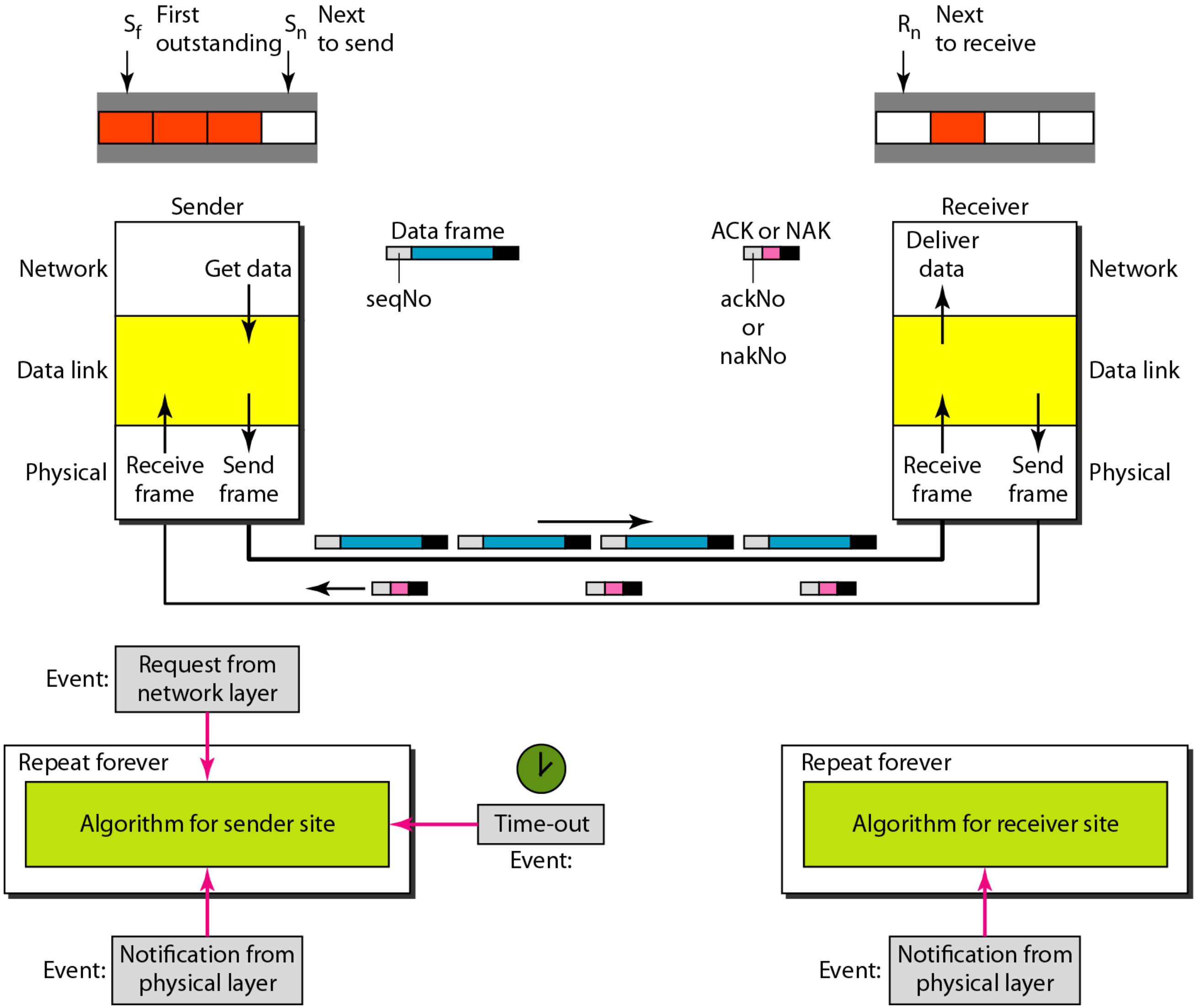
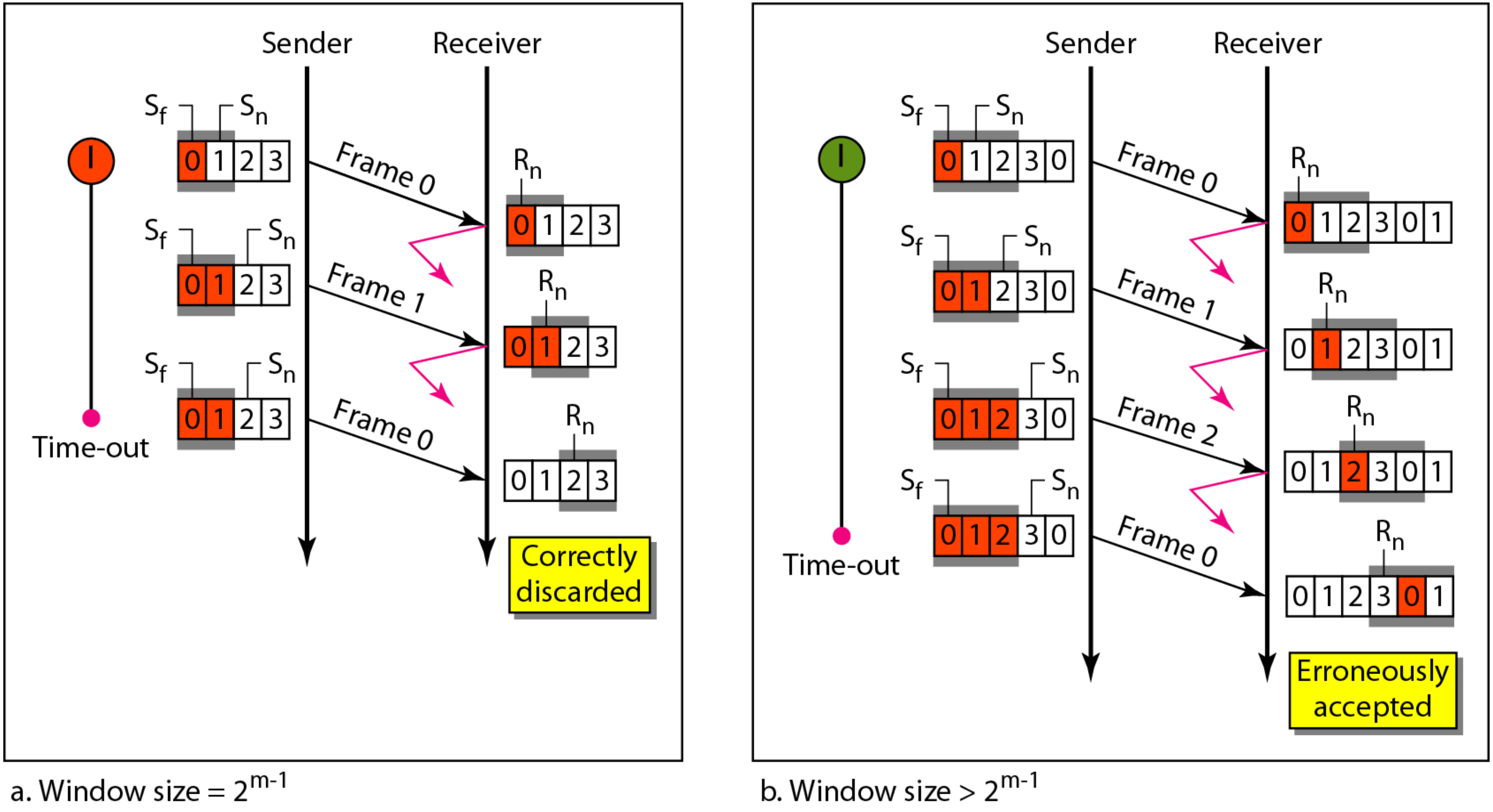


Figure 11.21 *Selective Repeat ARQ, window size*



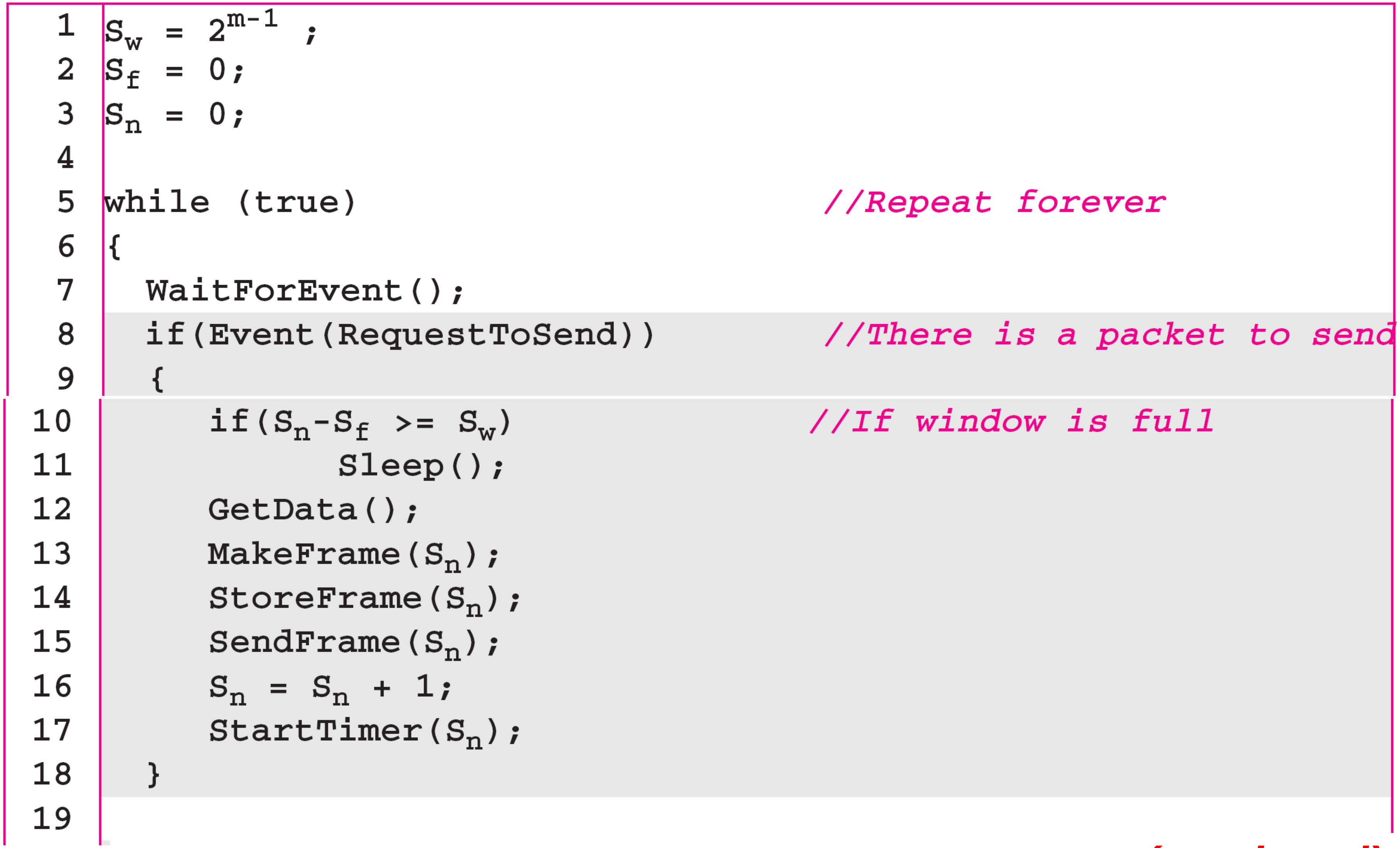
a. Window size = 2^{m-1}

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*



```
$S_w = 2^{m-1}$ ;
$S_f = 0$;
$S_n = 0$;

while (true)                                       //Repeat forever
{
  WaitForEvent();
  if(Event(RequestToSend))                         //There is a packet to send
  {
     if($S_n-S_f >= S_w$)                          //If window is full
            Sleep();
     GetData();
     MakeFrame($S_n$);
     StoreFrame($S_n$);
     SendFrame($S_n$);
     $S_n = S_n + 1$;
     StartTimer($S_n$);
  }

```

(continued)

Algorithm 11.9 *Sender-site Selective Repeat algorithm* *(continued)*

```
if(Event(ArrivalNotification)) //ACK arrives
{
   Receive(frame);             //Receive ACK or NAK
   if(corrupted(frame))
        Sleep();
   if (FrameType == NAK)
      if (nakNo between $S_f$ and $S_n$)
      {
       resend(nakNo);
       StartTimer(nakNo);
      }
   if (FrameType == ACK)
      if (ackNo between $S_f$ and $S_n$)
      {
        while($s_f$ < ackNo)
        {
         Purge($s_f$);
         StopTimer($s_f$);
         $S_f$ = $S_f$ + 1;
        }
      }
}
```

(continued)

Algorithm 11.9 *Sender-site Selective Repeat algorithm* (continued)

```

  if(Event(TimeOut(t)))          //The timer expires
  {
   StartTimer(t);
   SendFrame(t);
  }
}
```

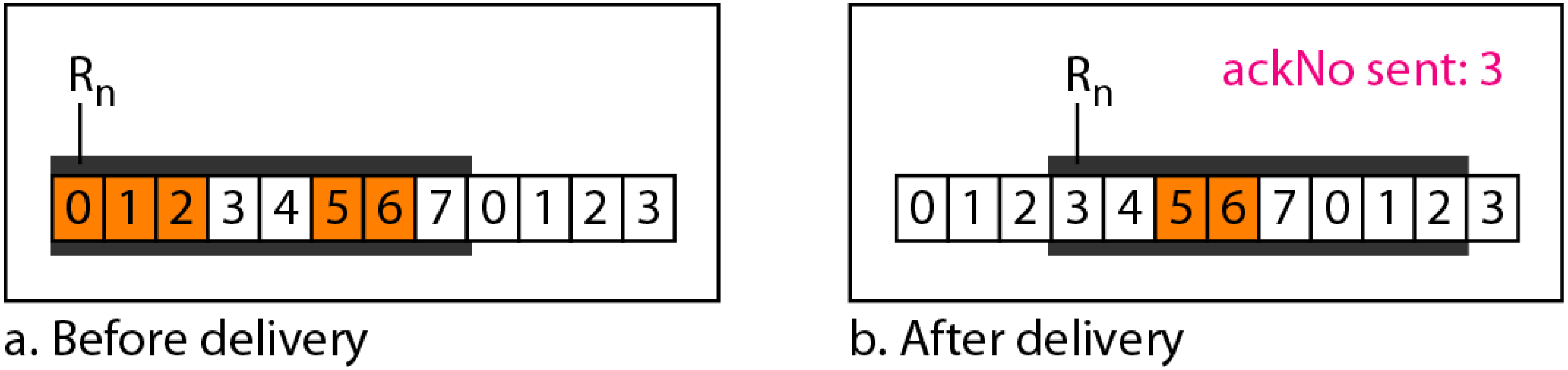
Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

```
$R_n$ = 0;
NakSent = false;
AckNeeded = false;
Repeat(for all slots)
    Marked(slot) = false;

while (true)                                             //Repeat forever
{
  WaitForEvent();

  if(Event(ArrivalNotification))                         /Data frame arrives
  {
     Receive(Frame);
     if(corrupted(Frame))&& (NOT NakSent)
     {
      SendNAK($R_n$);
      NakSent = true;
      Sleep();
     }
     if(seqNo <> $R_n$)&& (NOT NakSent)
     {
      SendNAK($R_n$);
```

Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

```
      NakSent = true;
      if ((seqNo in window)&&(!Marked(seqNo))
      {
       StoreFrame(seqNo)
       Marked(seqNo)= true;
       while(Marked($R_n$))
       {
        DeliverData($R_n$);
        Purge($R_n$);
        $R_n$ = $R_n$ + 1;
        AckNeeded = true;
       }
        if(AckNeeded);
        {
        SendAck($R_n$);
        AckNeeded = false;
        NakSent = false;
        }
      }
     }
  }
}
```

Figure 11.22 *Delivery of data in Selective Repeat ARQ*



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*

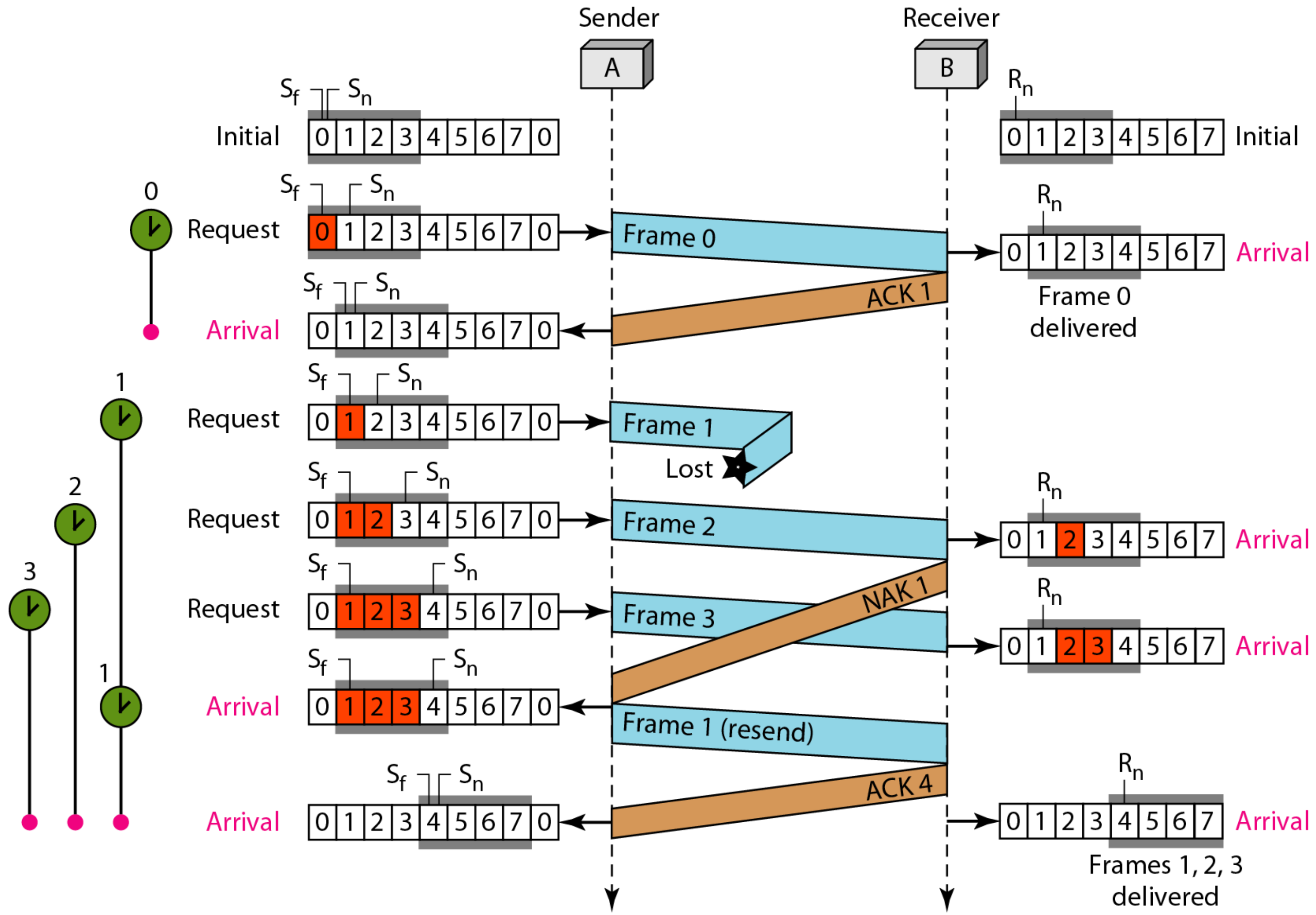
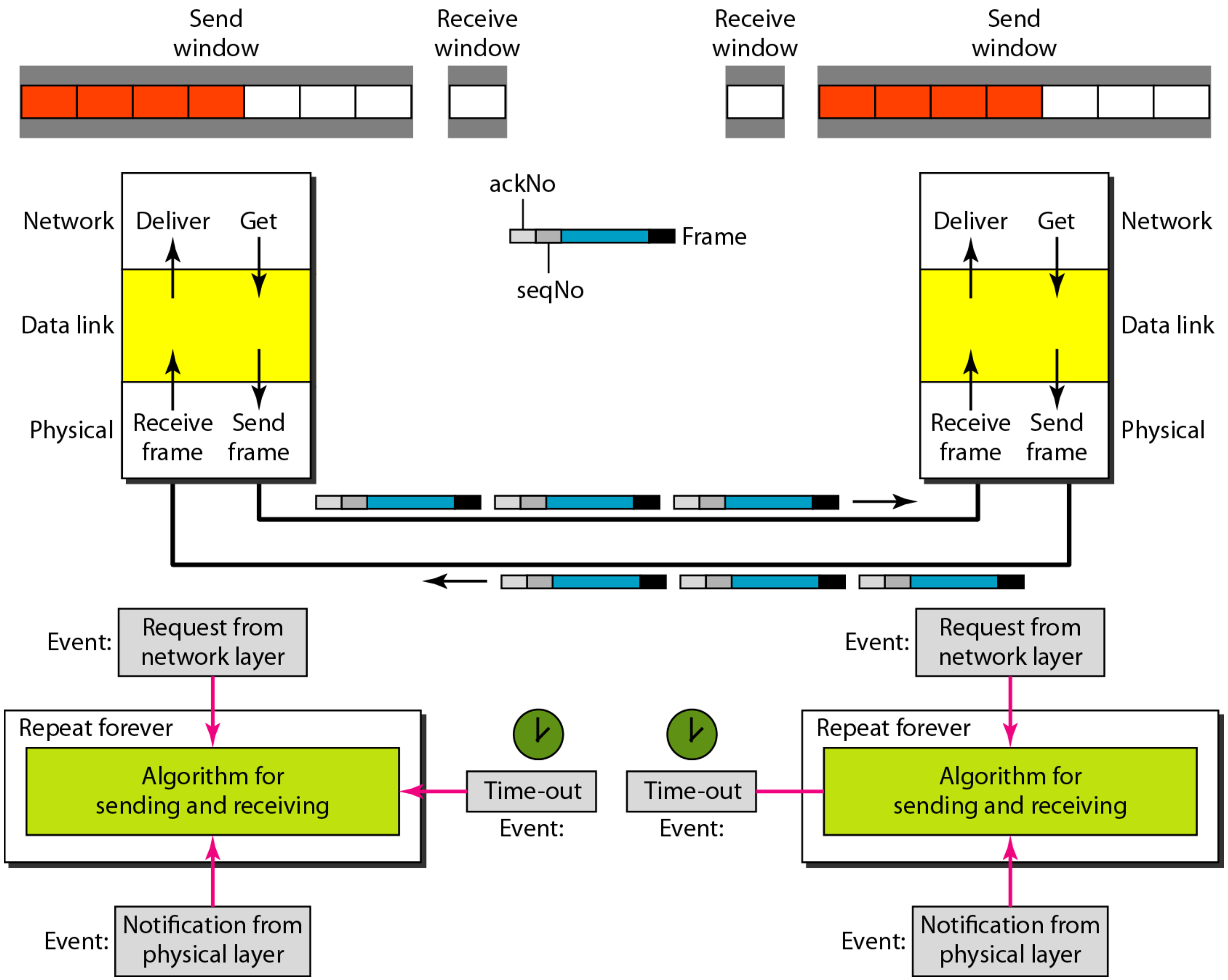


Figure 11.24 *Design of piggybacking in Go-Back-N ARQ*



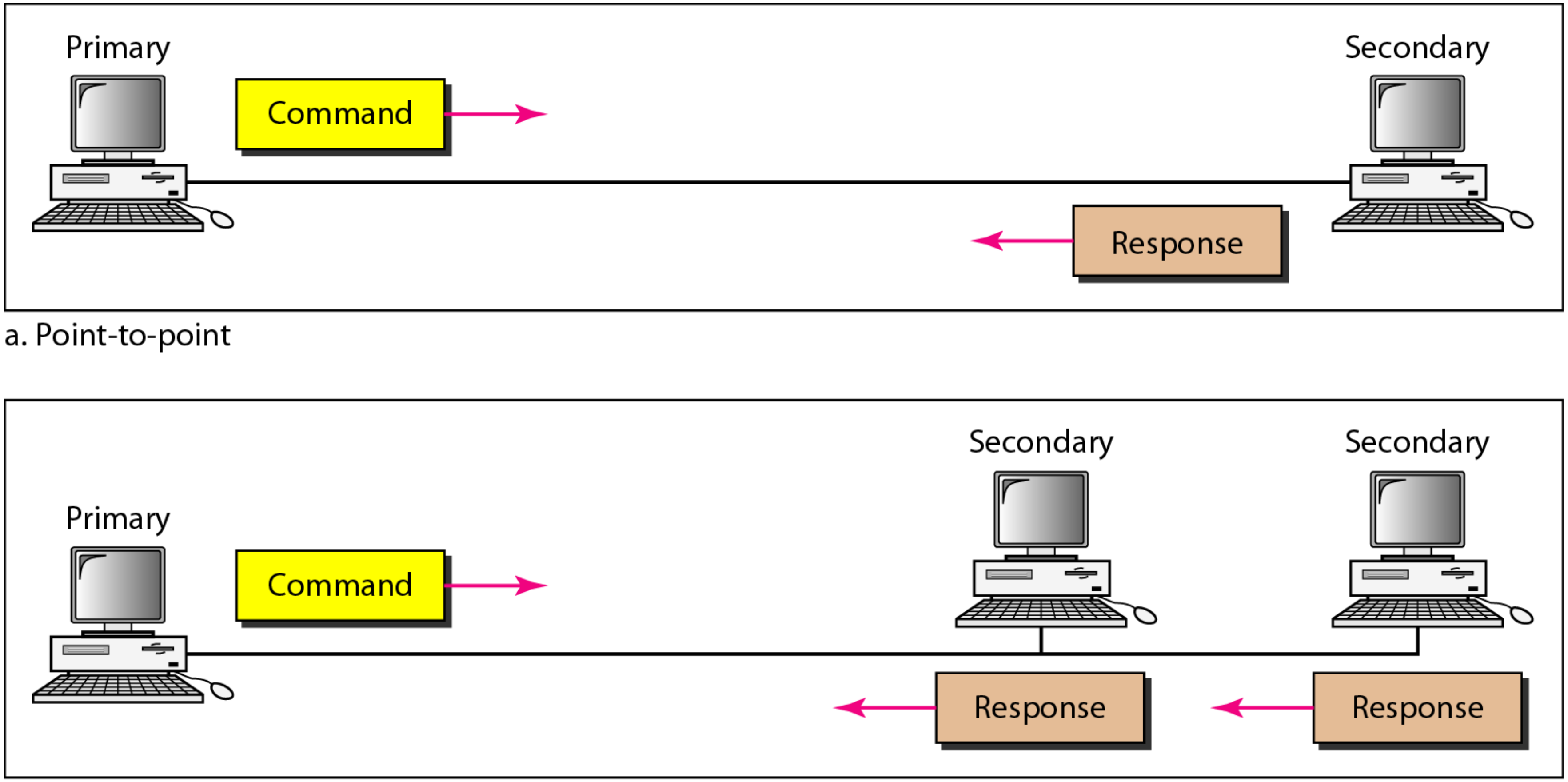
11-6 HDLC

***High-level Data Link Control (HDLC)* is a *bit-oriented* protocol for communication over point-to-point and multipoint links. It implements the ARQ mechanisms we discussed in this chapter.**

Topics discussed in this section:

Configurations and Transfer Modes
Frames
Control Field

Figure 11.25 *Normal response mode*



a. Point-to-point

b. Multipoint

Figure 11.26 *Asynchronous balanced mode*

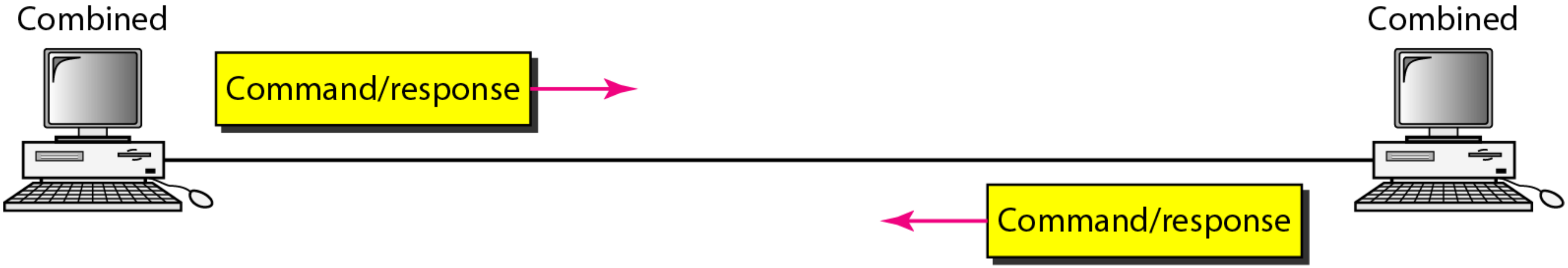


Figure 11.27 *HDLC frames*

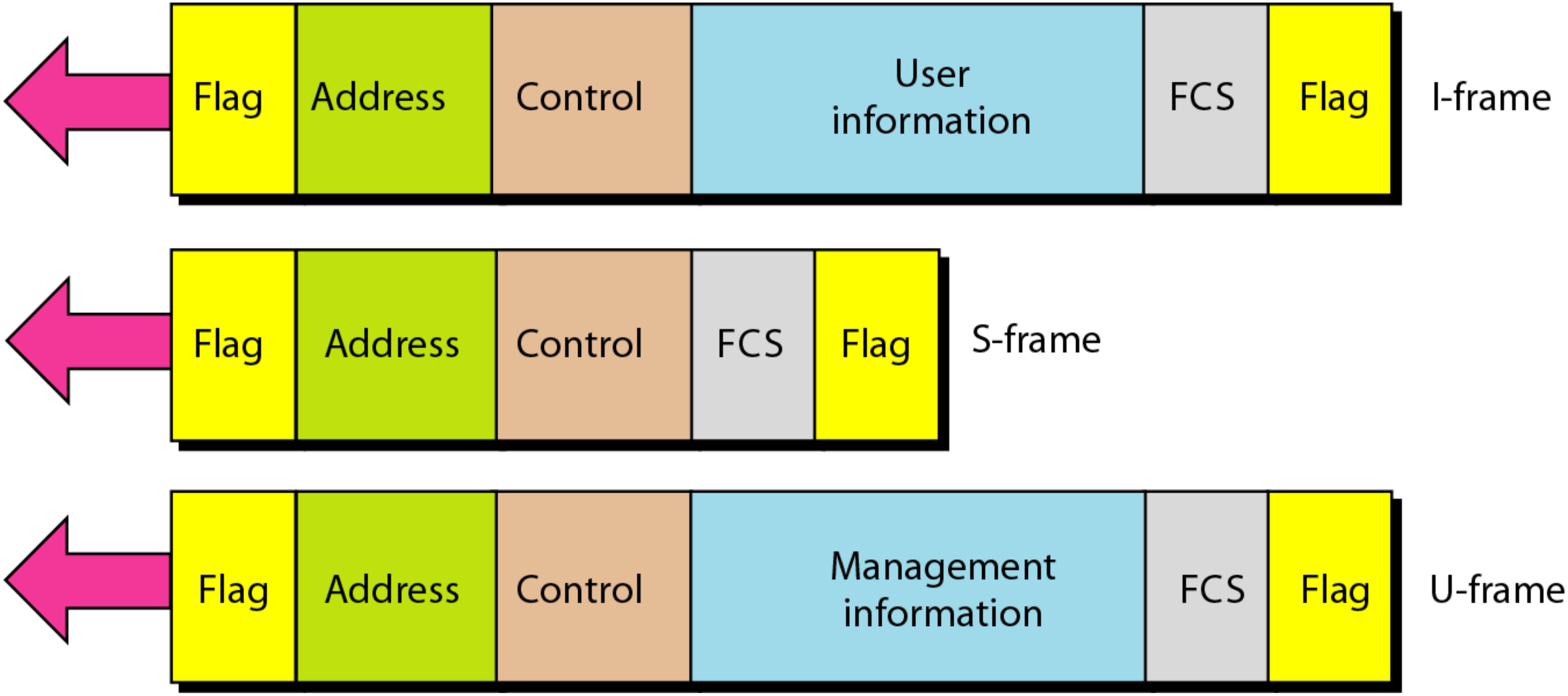


Figure 11.28 *Control field format for the different frame types*

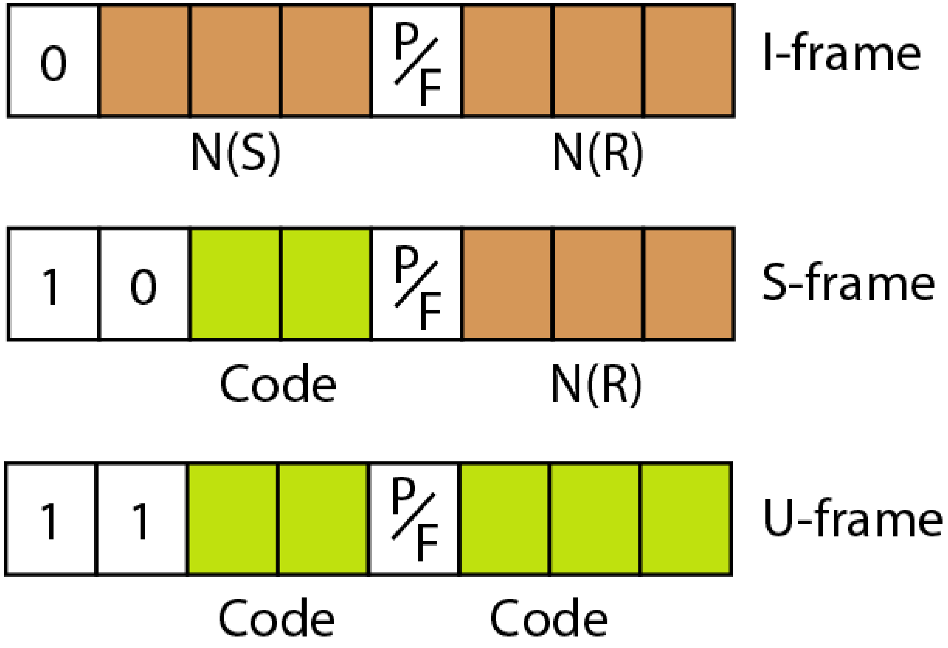


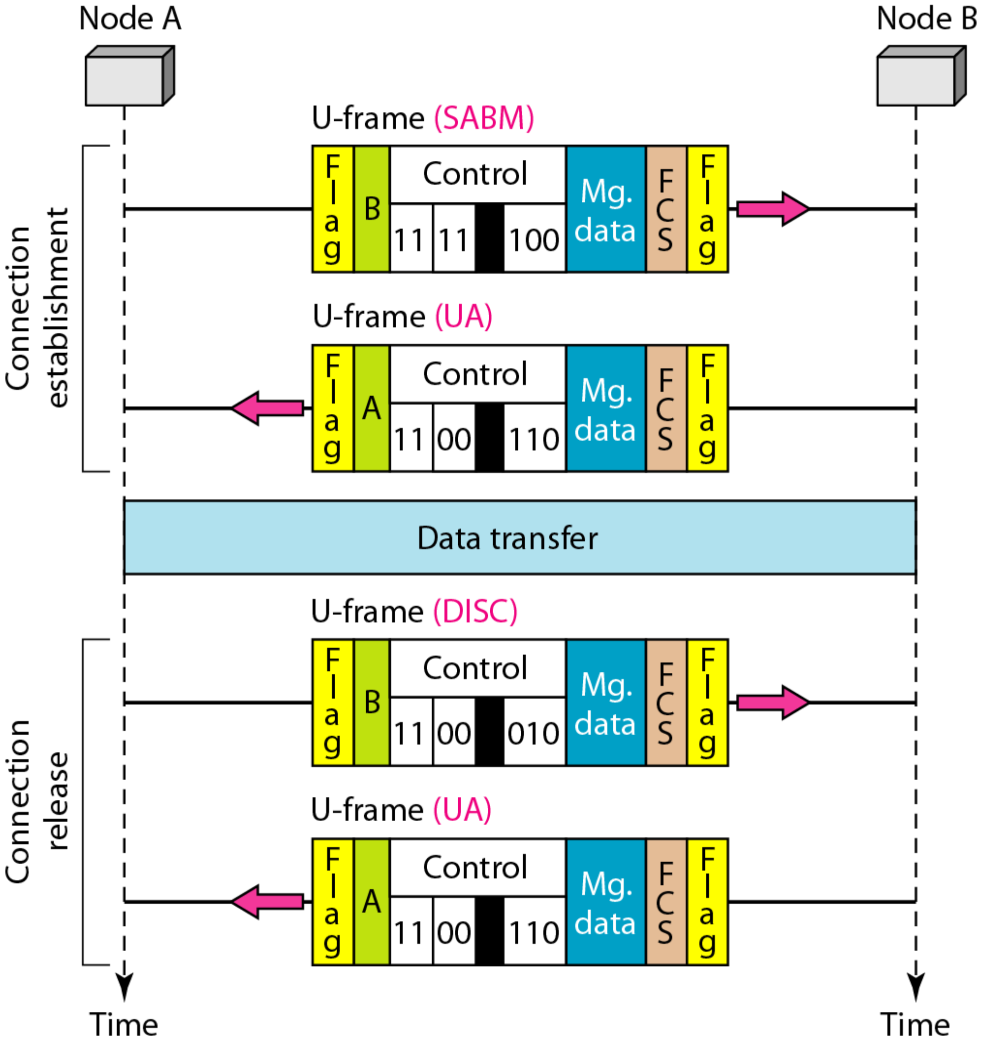
Table 11.1 *U-frame control command and response*

Code	*Command*	*Response*	*Meaning*
00 001	SNRM		Set normal response mode
11 011	SNRME		Set normal response mode, extended
11 100	SABM	**DM**	Set asynchronous balanced mode or **disconnect mode**
11 110	SABME		Set asynchronous balanced mode, extended
00 000	UI	**UI**	Unnumbered information
00 110		**UA**	**Unnumbered acknowledgment**
00 010	DISC	**RD**	Disconnect or **request disconnect**
10 000	SIM	**RIM**	Set initialization mode or **request information mode**
00 100	UP		Unnumbered poll
11 001	RSET		Reset
11 101	XID	**XID**	Exchange ID
10 001	FRMR	**FRMR**	Frame reject

Example 11.9

Figure 11.29 shows how U-frames can be used for connection establishment and connection release. Node A asks for a connection with a set asynchronous balanced mode (SABM) frame; node B gives a positive response with an unnumbered acknowledgment (UA) frame. After these two exchanges, data can be transferred between the two nodes (not shown in the figure). After data transfer, node A sends a DISC (disconnect) frame to release the connection; it is confirmed by node B responding with a UA (unnumbered acknowledgment).

Figure 11.29 *Example of connection and disconnection*



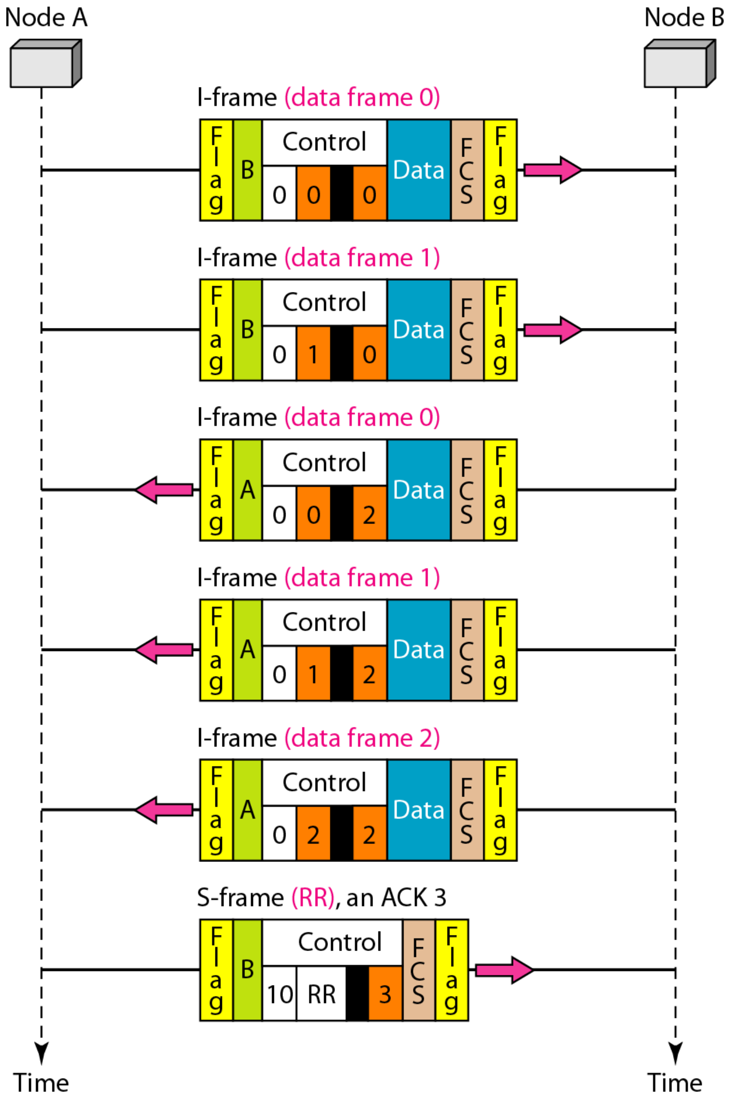
Example 11.10

Figure 11.30 shows an exchange using piggybacking. Node A begins the exchange of information with an I-frame numbered 0 followed by another I-frame numbered 1. Node B piggybacks its acknowledgment of both frames onto an I-frame of its own. Node B's first I-frame is also numbered 0 [N(S) field] and contains a 2 in its N(R) field, acknowledging the receipt of A's frames 1 and 0 and indicating that it expects frame 2 to arrive next. Node B transmits its second and third I-frames (numbered 1 and 2) before accepting further frames from node A.

Example 11.10 (continued)

Its N(R) information, therefore, has not changed: B frames 1 and 2 indicate that node B is still expecting A's frame 2 to arrive next. Node A has sent all its data. Therefore, it cannot piggyback an acknowledgment onto an I-frame and sends an S-frame instead. The RR code indicates that A is still ready to receive. The number 3 in the N(R) field tells B that frames 0, 1, and 2 have all been accepted and that A is now expecting frame number 3.

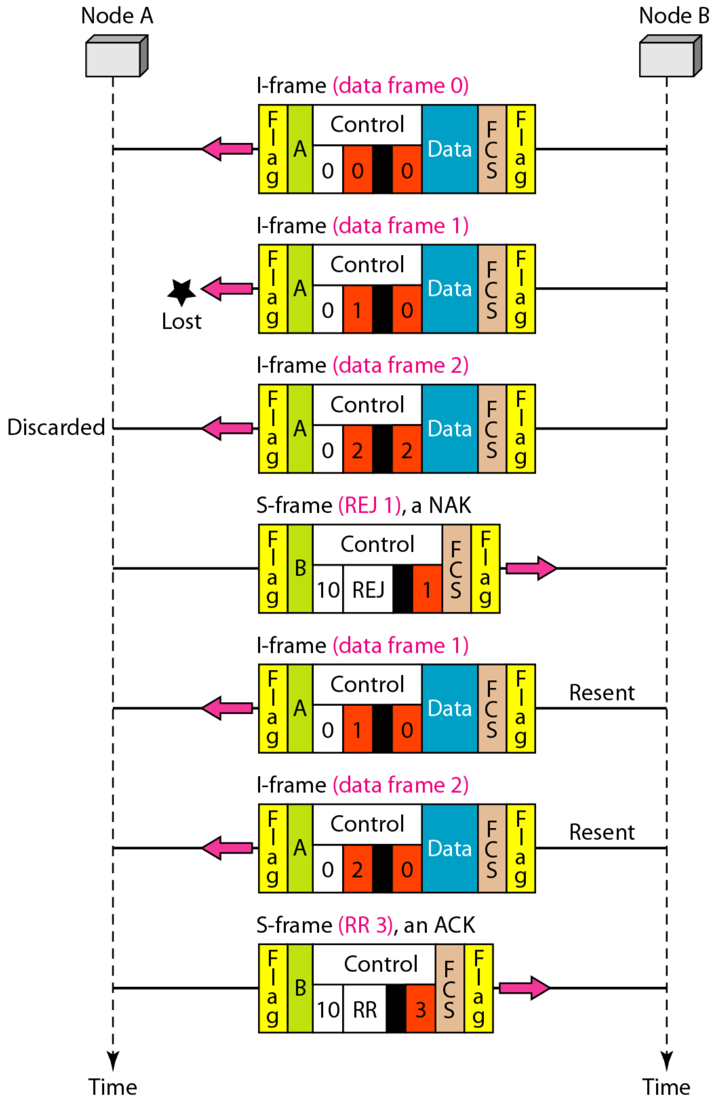
Figure 11.30 *Example of piggybacking without error*


Node A
Node B
I-frame (data frame 0)
Flag B Control 0 0 0 Data FCS Flag
I-frame (data frame 1)
Flag B Control 0 1 0 Data FCS Flag
I-frame (data frame 0)
Flag A Control 0 0 2 Data FCS Flag
I-frame (data frame 1)
Flag A Control 0 1 2 Data FCS Flag
I-frame (data frame 2)
Flag A Control 0 2 2 Data FCS Flag
S-frame (RR), an ACK 3
Flag B Control 10 RR 3 FCS Flag
Time
Time

Example 11.11

Figure 11.31 shows an exchange in which a frame is lost. Node B sends three data frames (0, 1, and 2), but frame 1 is lost. When node A receives frame 2, it discards it and sends a REJ frame for frame 1. Note that the protocol being used is Go-Back-N with the special use of an REJ frame as a NAK frame. The NAK frame does two things here: It confirms the receipt of frame 0 and declares that frame 1 and any following frames must be resent. Node B, after receiving the REJ frame, resends frames 1 and 2. Node A acknowledges the receipt by sending an RR frame (ACK) with acknowledgment number 3.

Figure 11.31 *Example of piggybacking with error*



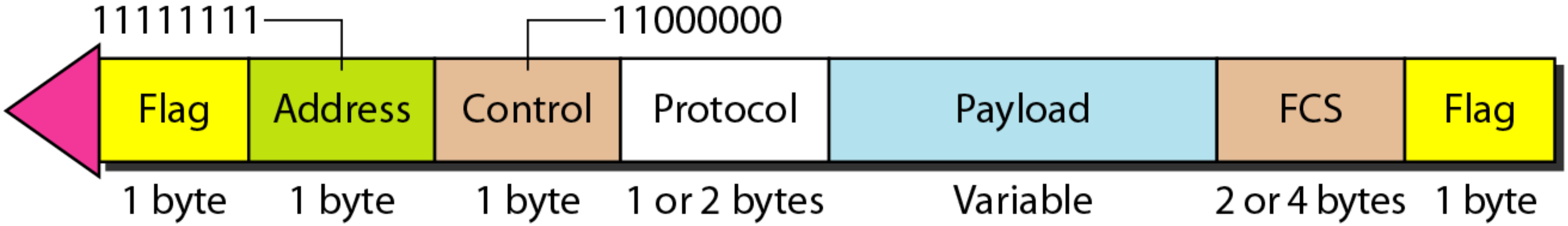
11-7 POINT-TO-POINT PROTOCOL

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*

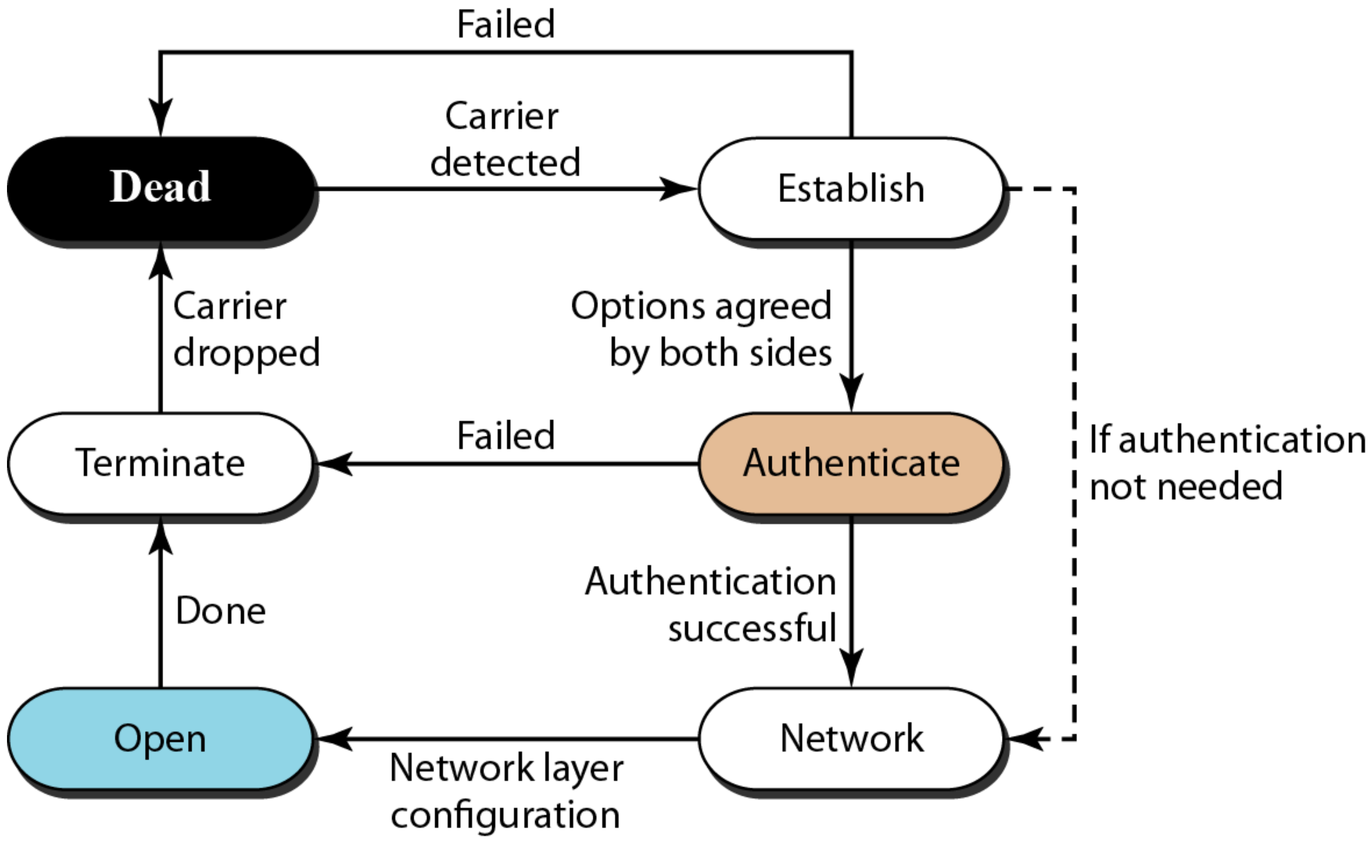
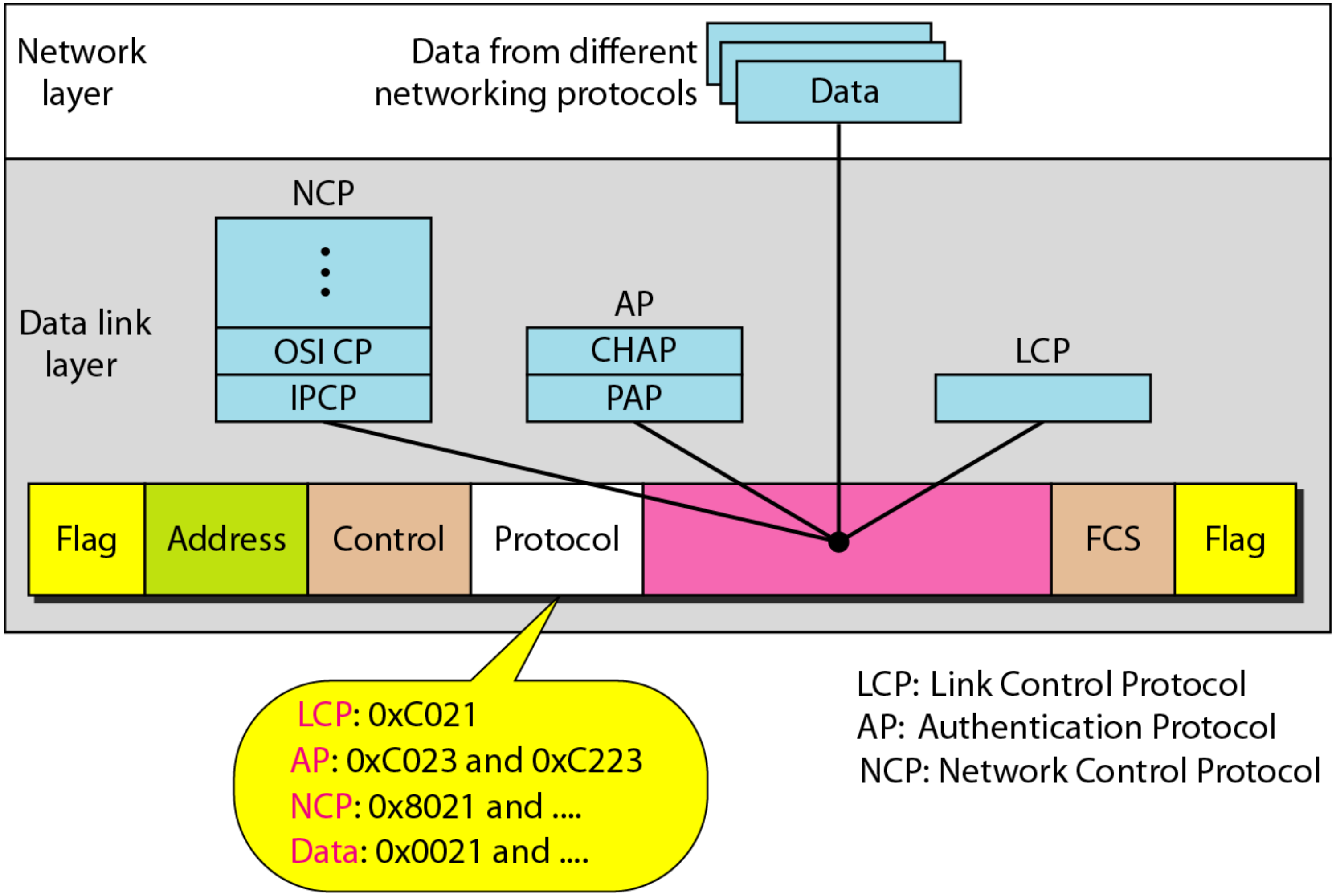

Failed
Carrier detected
Dead
Establish
Carrier dropped
Options agreed by both sides
Terminate
Failed
Authenticate
If authentication not needed
Done
Authentication successful
Open
Network
Network layer configuration

Figure 11.34 *Multiplexing in PPP*



LCP: Link Control Protocol
AP: Authentication Protocol
NCP: Network Control Protocol

Figure 11.35 *LCP packet encapsulated in a frame*

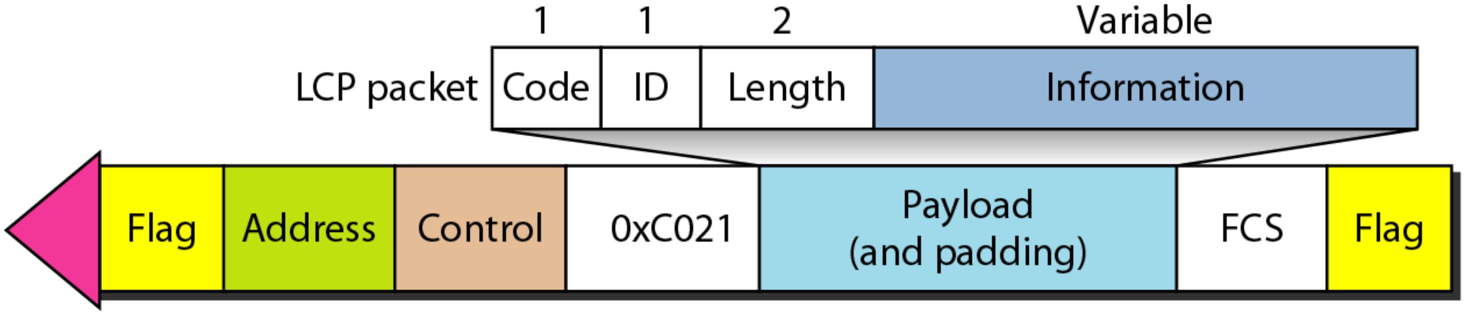
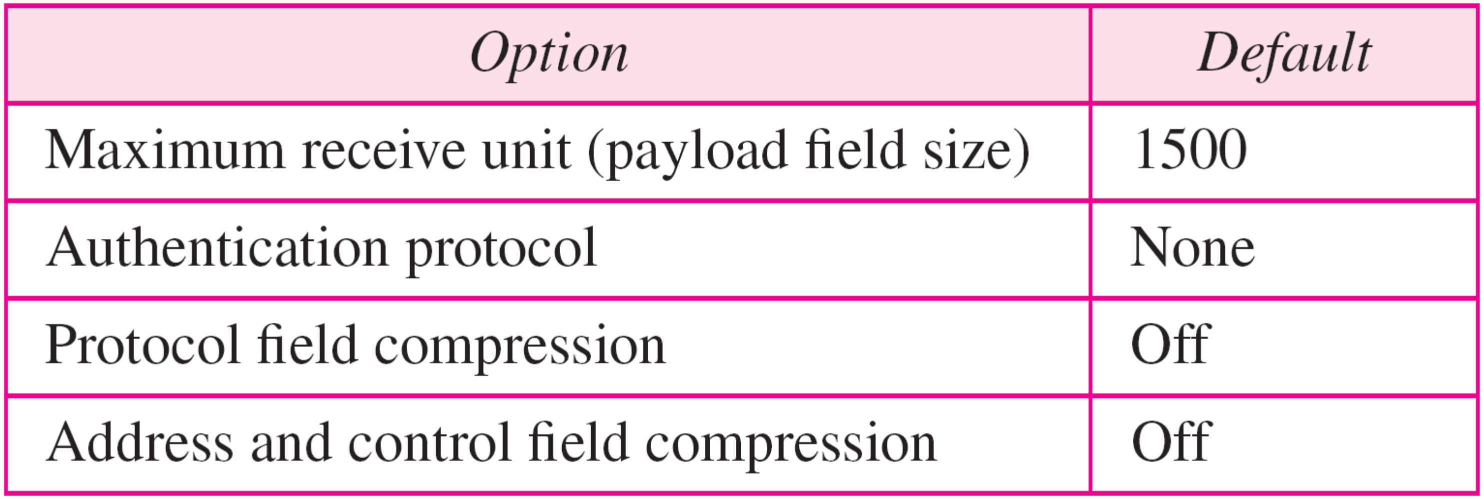


Table 11.2 *LCP packets*

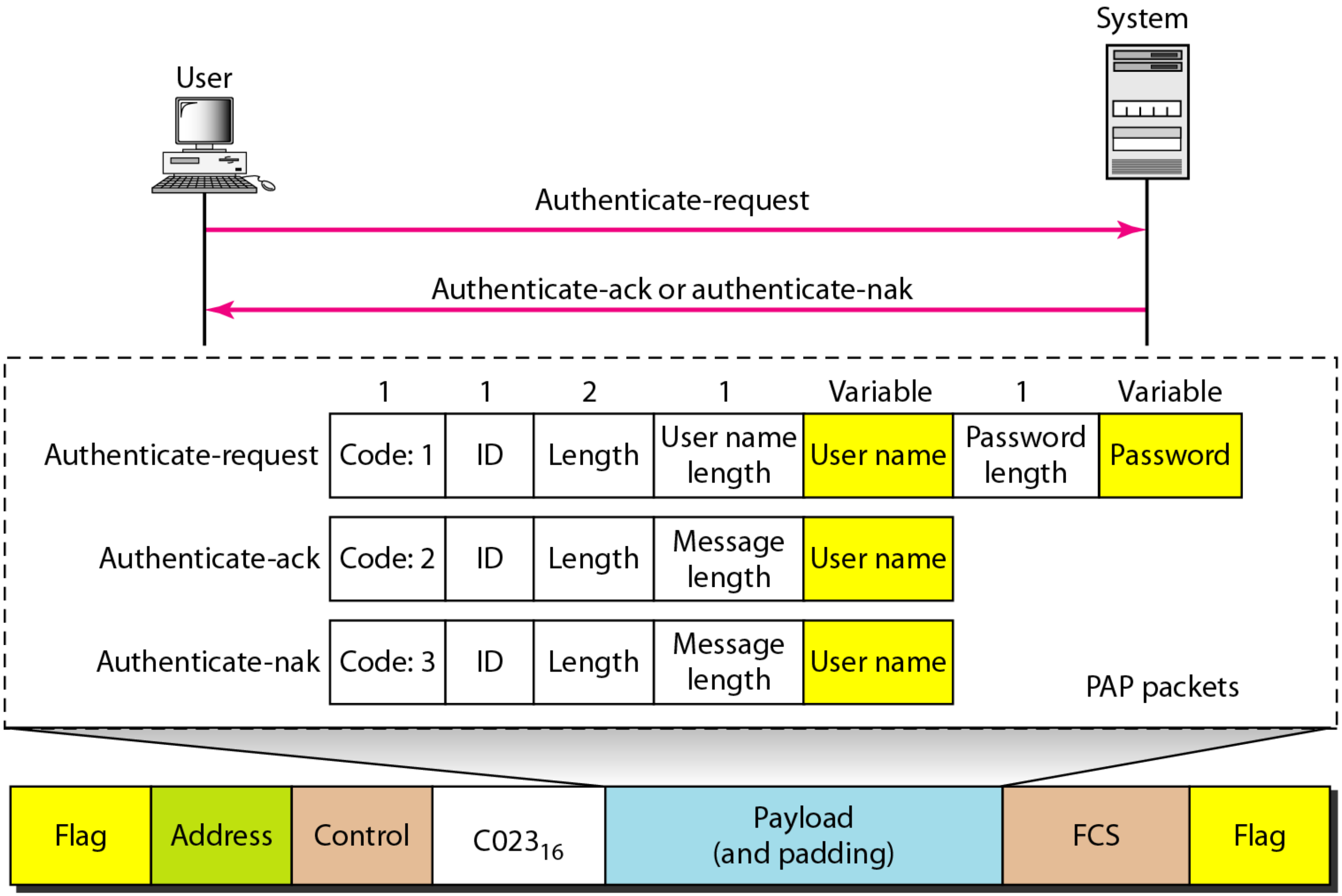
Code	*Packet Type*	*Description*
0x01	Configure-request	Contains the list of proposed options and their values
0x02	Configure-ack	Accepts all options proposed
0x03	Configure-nak	Announces that some options are not acceptable
0x04	Configure-reject	Announces that some options are not recognized
0x05	Terminate-request	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***



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

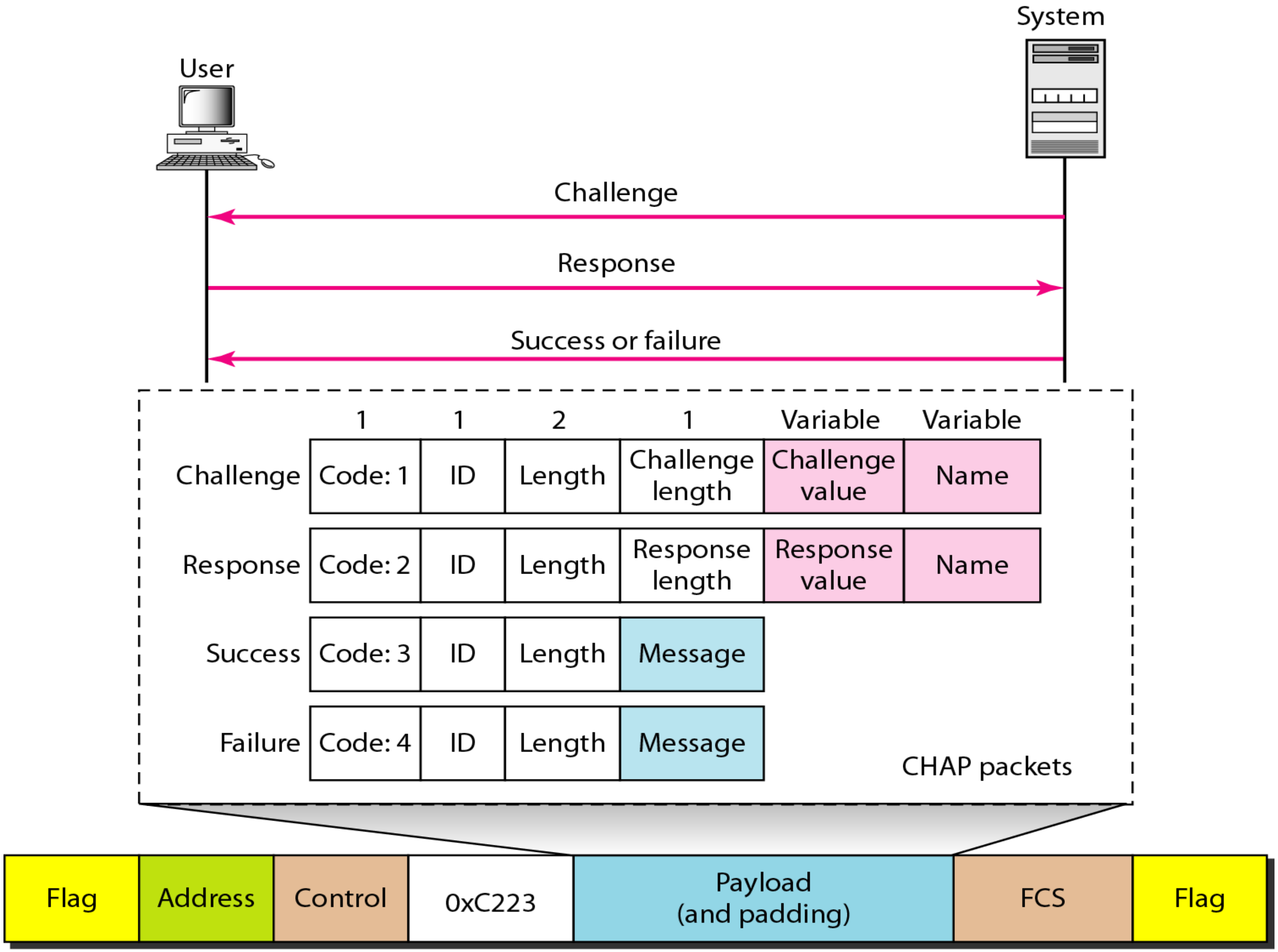
Figure 11.36 *PAP packets encapsulated in a PPP frame*



	1	1	2	1	Variable	1	Variable
Authenticate-request	Code: 1	ID	Length	User name length	User name	Password length	Password
Authenticate-ack	Code: 2	ID	Length	Message length	User name		
Authenticate-nak	Code: 3	ID	Length	Message length	User name		

Flag	Address	Control	$C023_{16}$	Payload (and padding)	FCS	Flag

Figure 11.37 *CHAP packets encapsulated in a PPP frame*



	1	1	2	1	Variable	Variable
Challenge	Code: 1	ID	Length	Challenge length	Challenge value	Name
Response	Code: 2	ID	Length	Response length	Response value	Name
Success	Code: 3	ID	Length	Message		
Failure	Code: 4	ID	Length	Message		

Flag	Address	Control	0xC223	Payload (and padding)	FCS	Flag

Figure 11.38 *IPCP packet encapsulated in PPP frame*

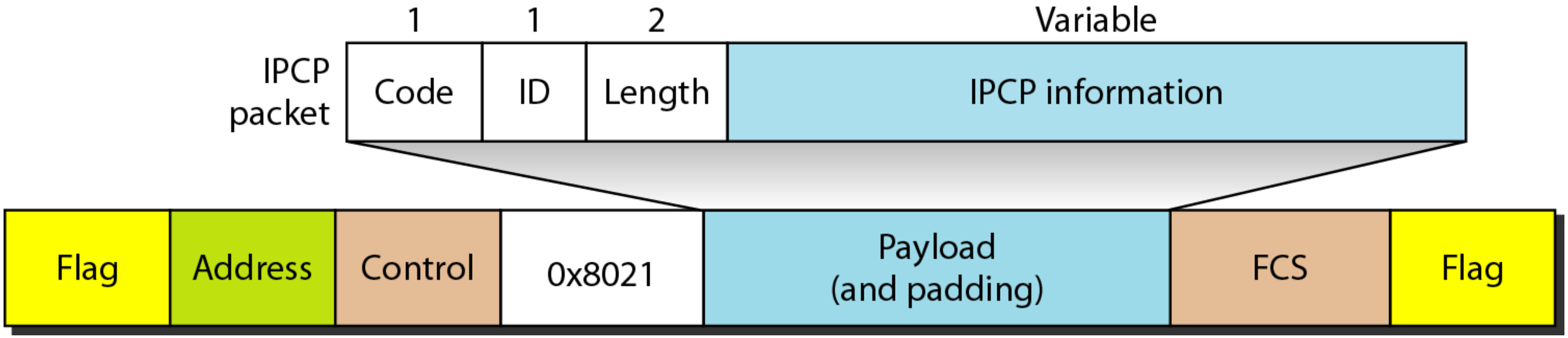
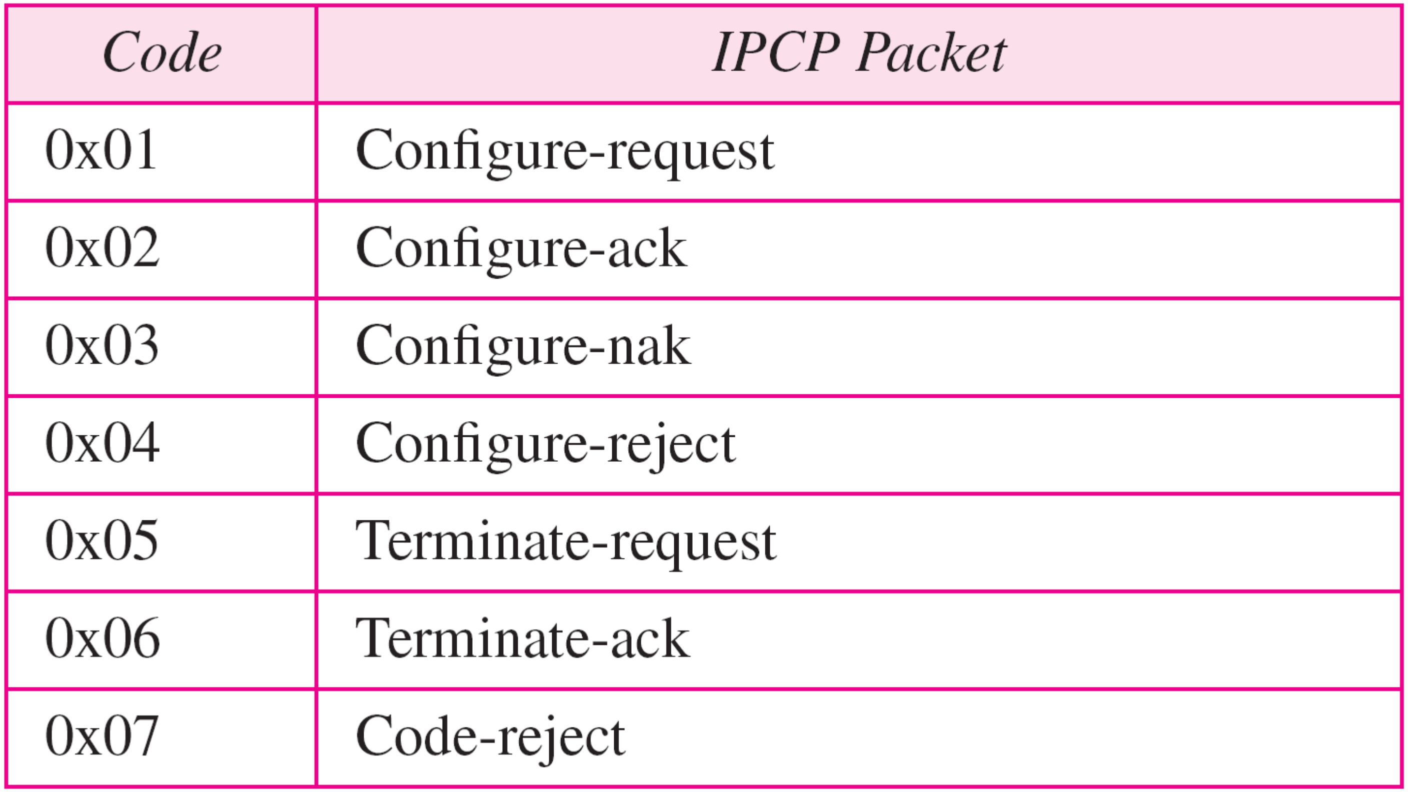


Table 11.4 ***Code value for IPCP packets***



Code	*IPCP Packet*
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*

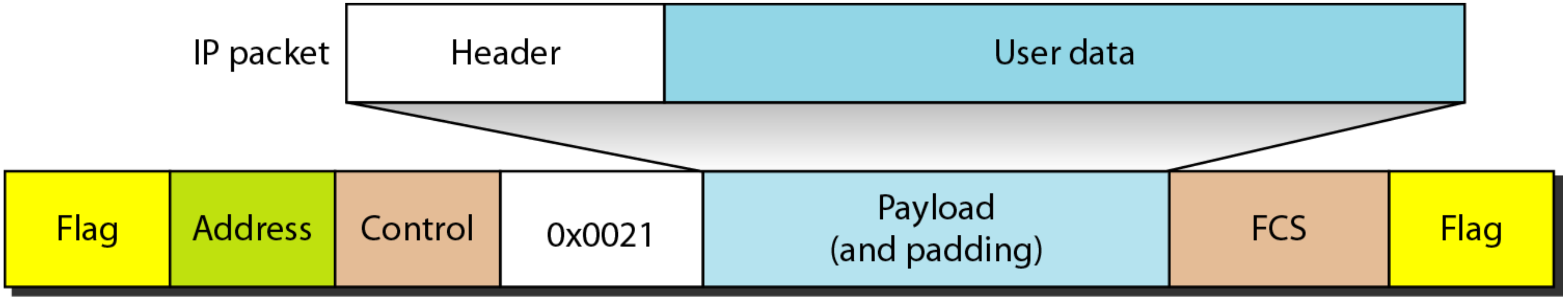
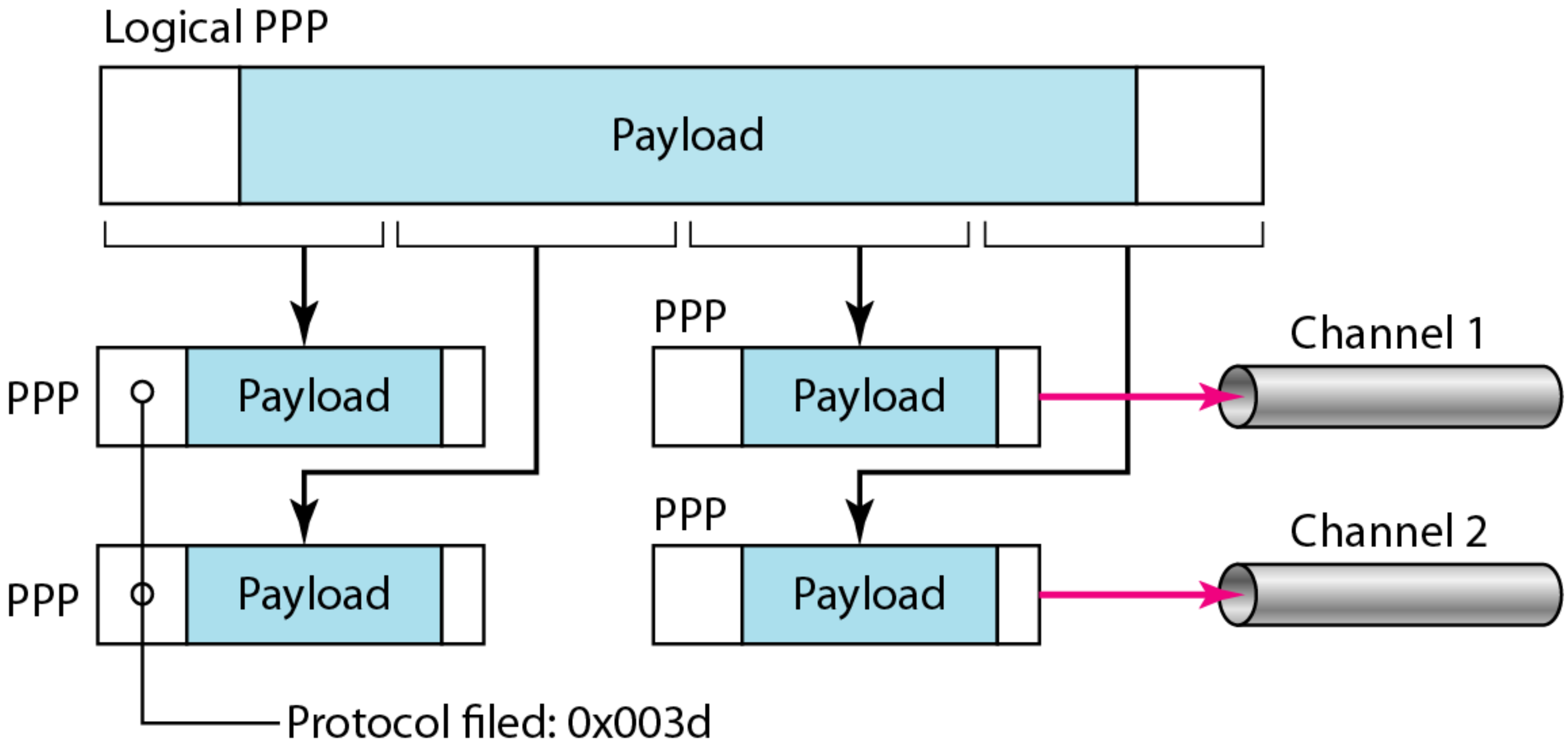


Figure 11.40 *Multilink PPP*



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*

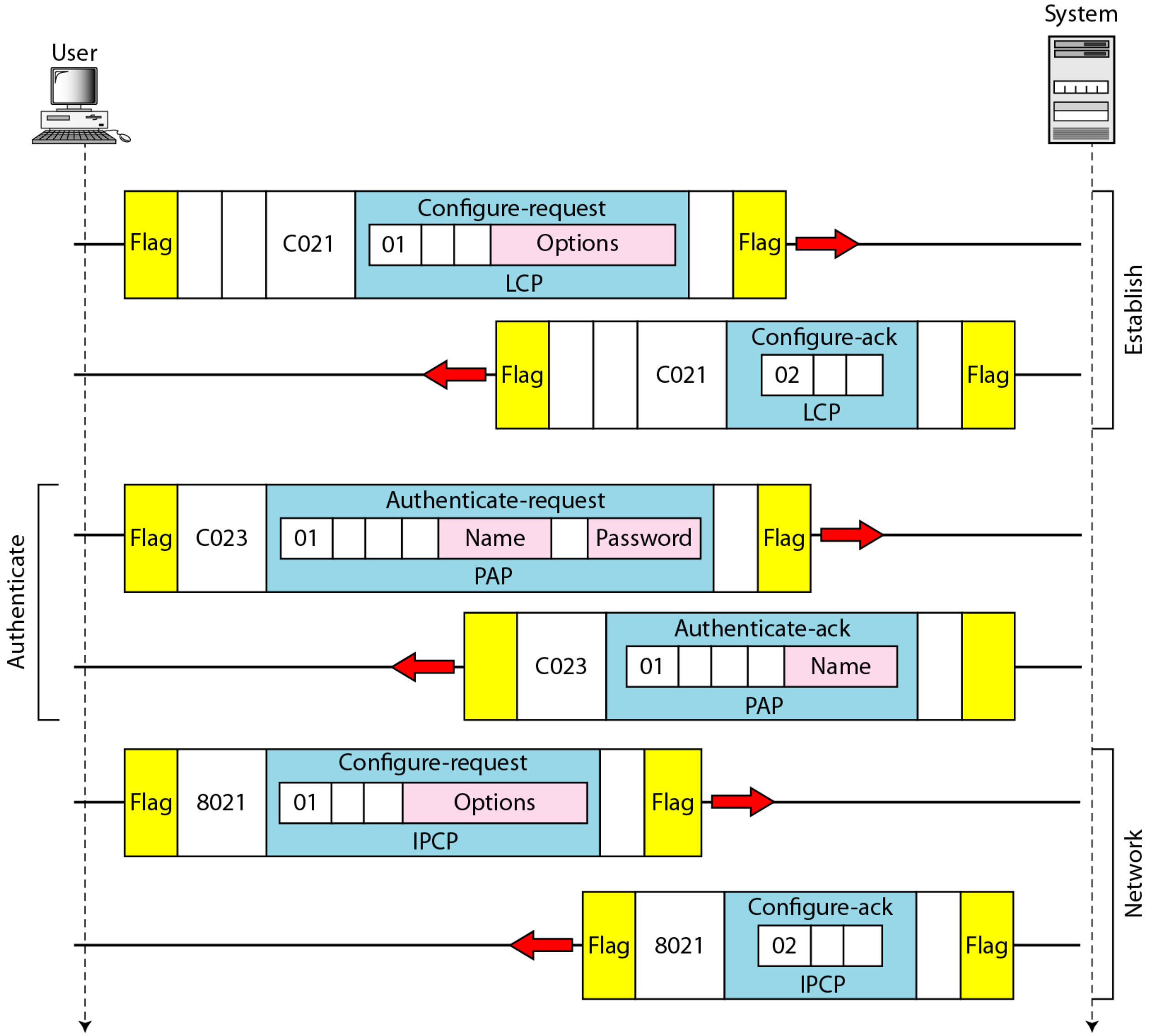
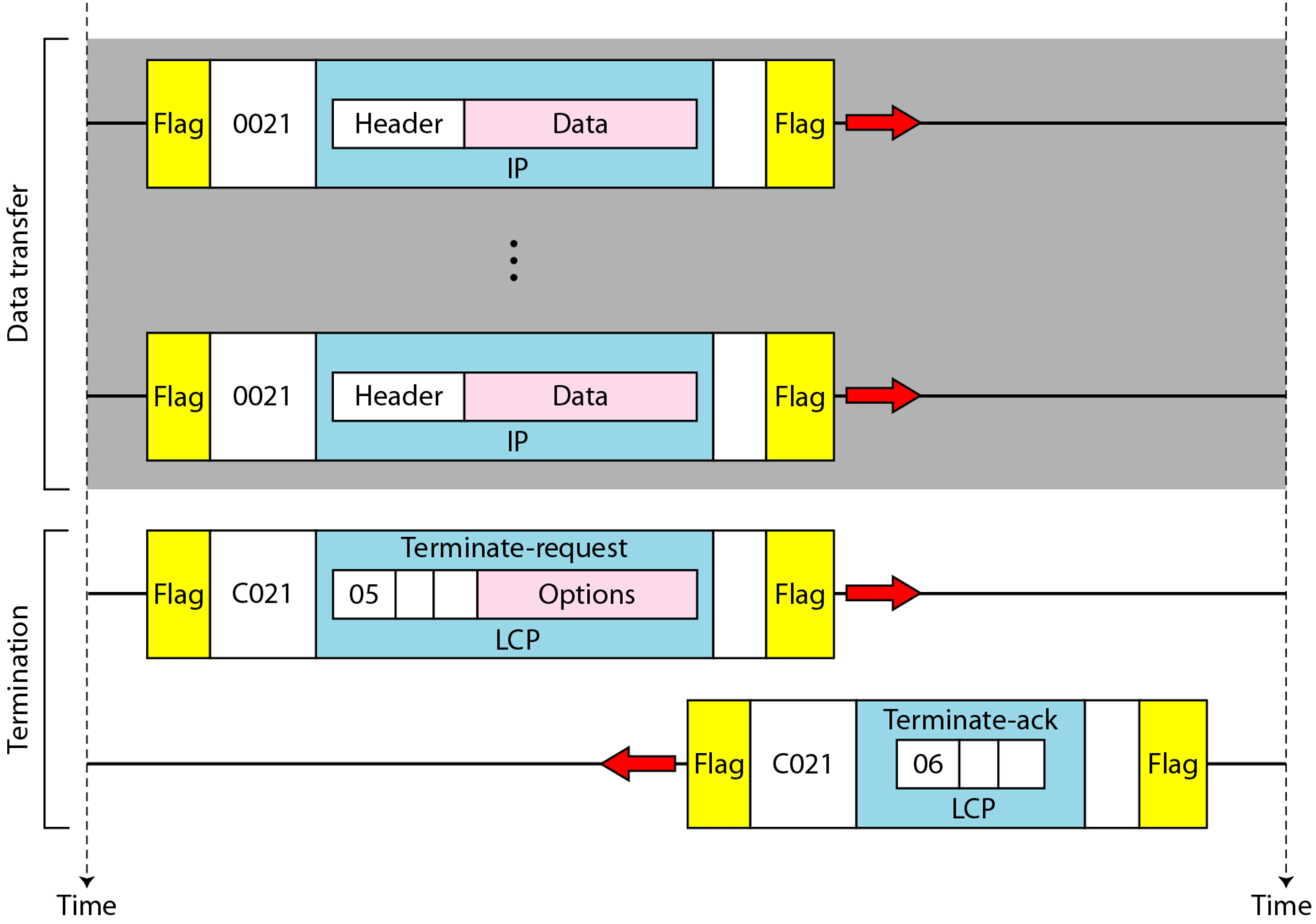


Figure 11.41 *An example (continued)*



COMPUTER COMMUNICATION NETWORKS

(15EC64)

Chapter 12

Multiple Access

Figure 12.1 *Data link layer divided into two functionality-oriented sublayers*

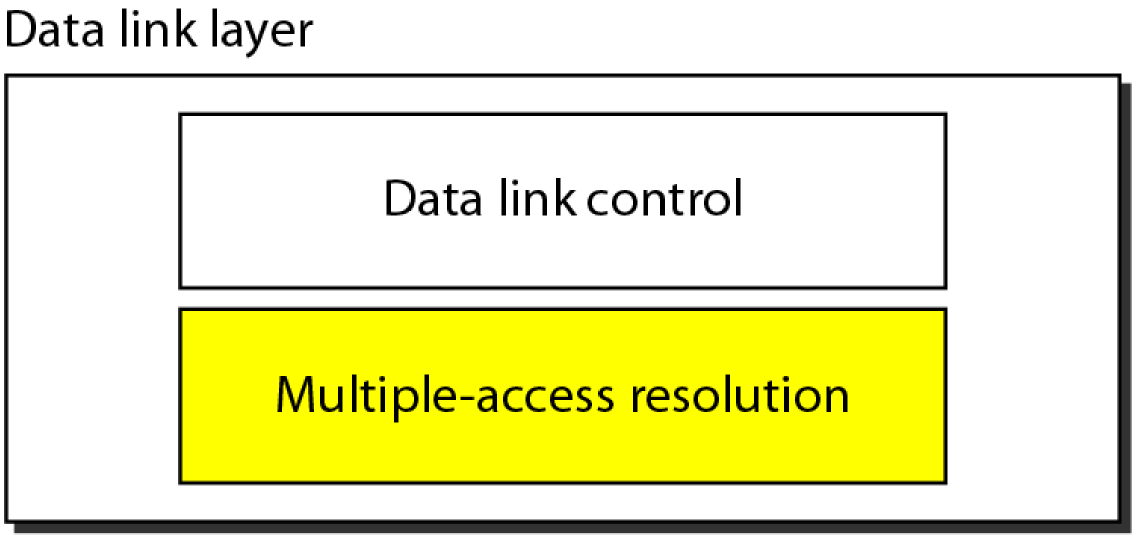
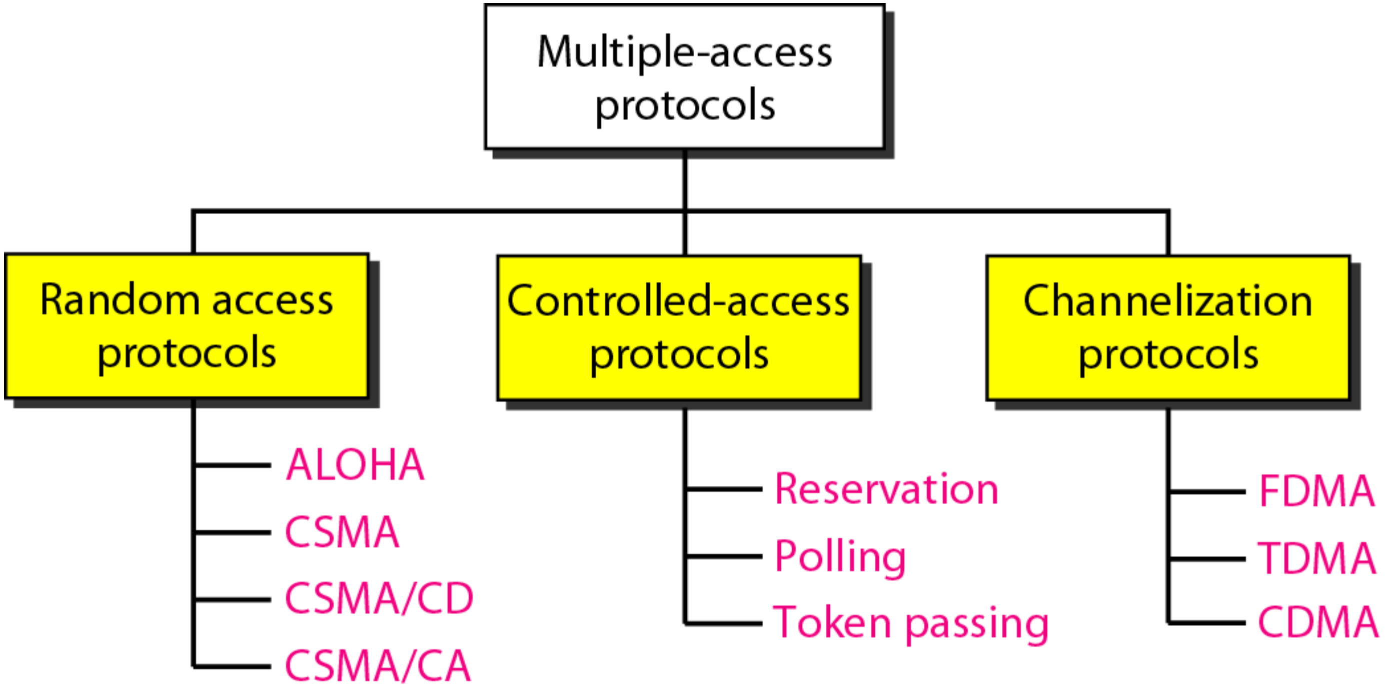


Figure 12.2 *Taxonomy of multiple-access protocols discussed in this chapter*



12-1 RANDOM ACCESS

In random access or contention methods, no station is superior to another station and none is assigned the control over another. No station permits, or does not permit, another station to send. At each instance, a station that has data to send uses a procedure defined by the protocol to make a decision on whether or not to send.

Topics discussed in this section:

ALOHA
Carrier Sense Multiple Access
Carrier Sense Multiple Access with Collision Detection
Carrier Sense Multiple Access with Collision Avoidance

Figure 12.3 *Frames in a pure ALOHA network*

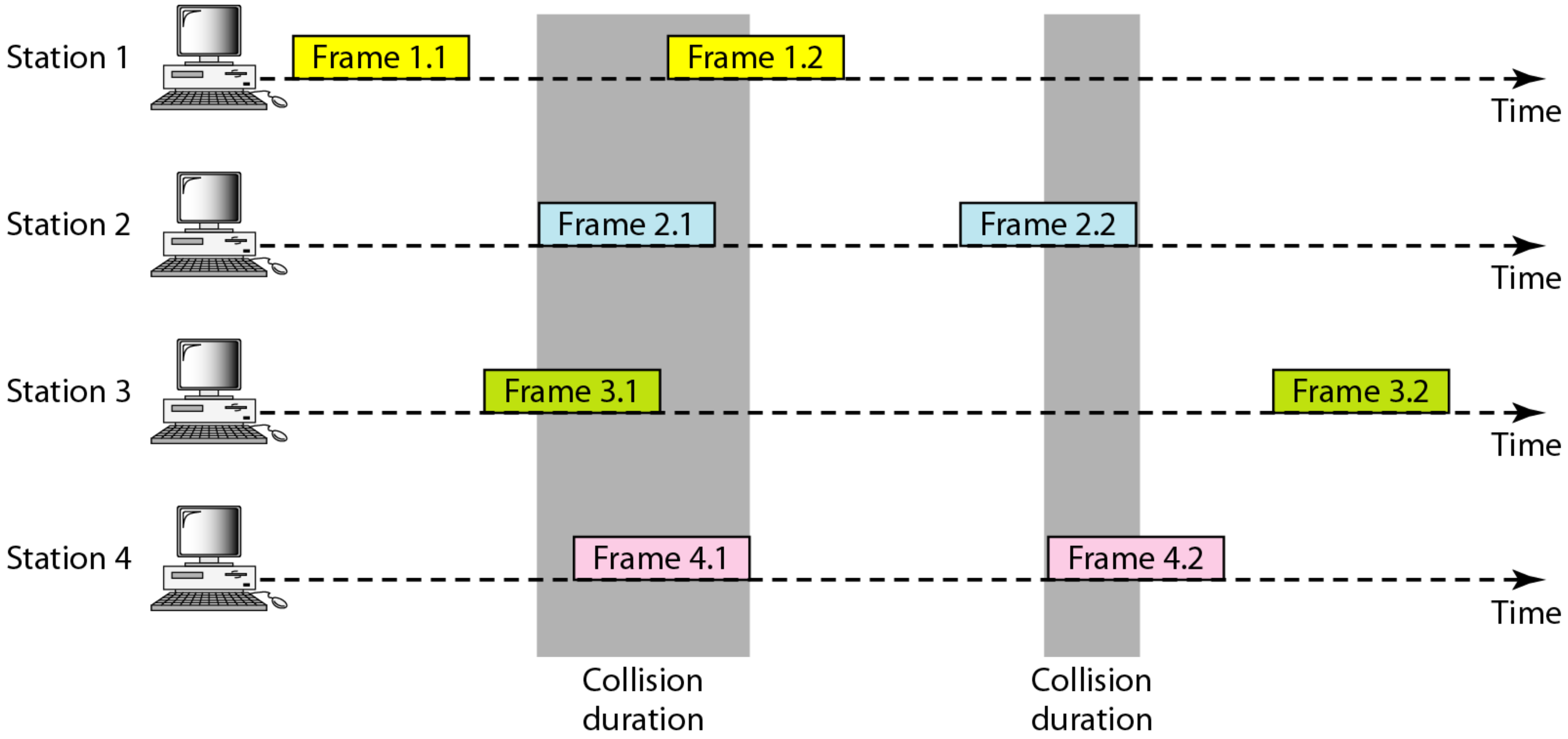
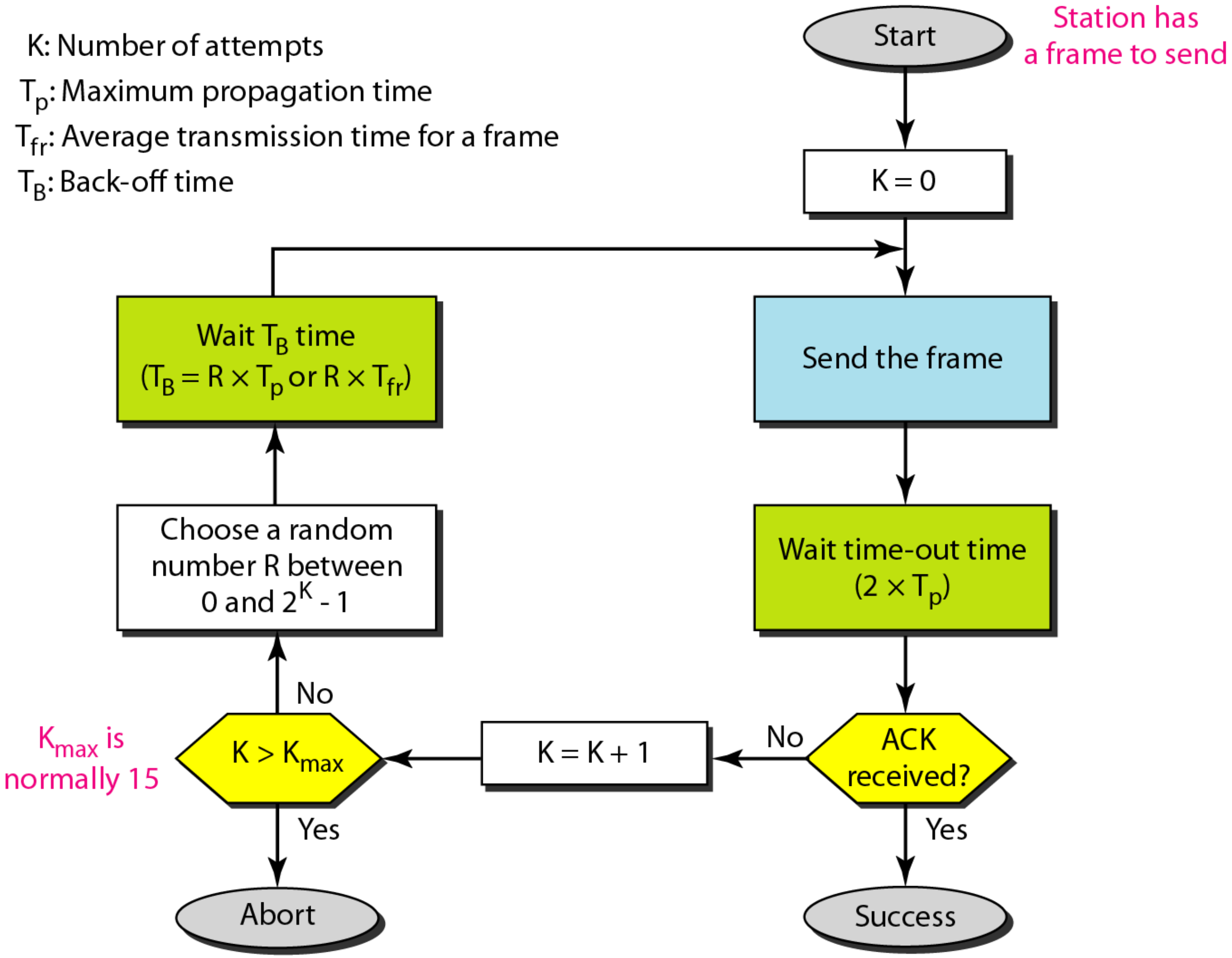


Figure 12.4 *Procedure for pure ALOHA protocol*


K: Number of attempts
T_p: Maximum propagation time
T_{fr}: Average transmission time for a frame
T_B: Back-off time
Start
Station has a frame to send
K = 0
Send the frame
Wait time-out time ($2 \times T_p$)
ACK received?
No
Yes
Success
K = K + 1
$K > K_{max}$
K_{max} is normally 15
No
Yes
Abort
Choose a random number R between 0 and 2^K - 1
Wait T_B time ($T_B = R \times T_p$ or $R \times T_{fr}$)

Example 12.1

The stations on a wireless ALOHA network are a maximum of 600 km apart. If we assume that signals propagate at 3 × 10^8 m/s, we find

$$T_p = (600 \times 10^5) / (3 \times 10^8) = 2 \text{ ms}.$$

Now we can find the value of T_B for different values of K .

a. For K = 1, the range is {0, 1}. The station needs to generate a random number with a value of 0 or 1. This means that T_B is either 0 ms (0 × 2) or 2 ms (1 × 2), based on the outcome of the random variable.

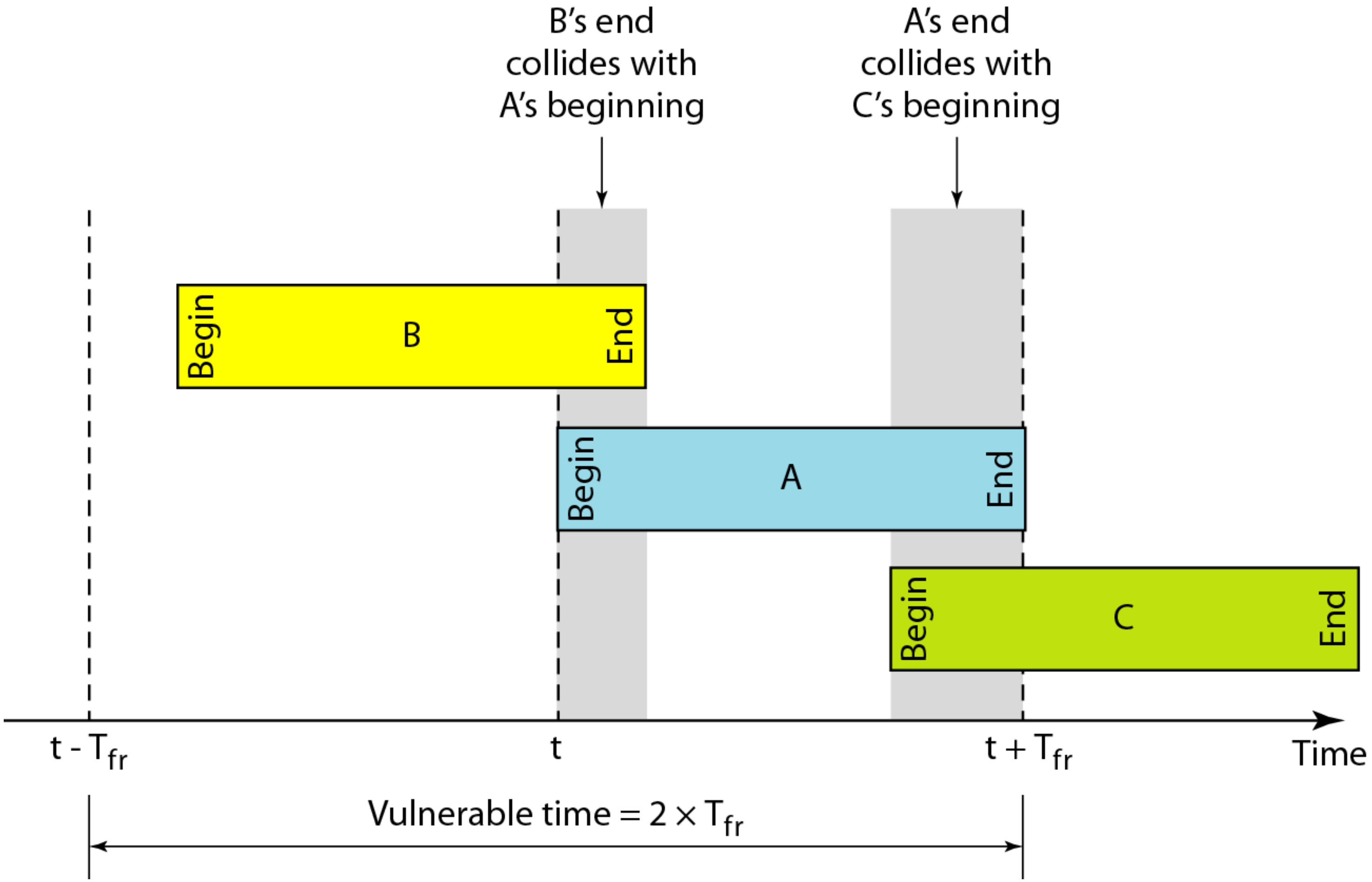
Example 12.1 (continued)

***b.** For K = 2, the range is {0, 1, 2, 3}. This means that T_B can be 0, 2, 4, or 6 ms, based on the outcome of the random variable.*

***c.** For K = 3, the range is {0, 1, 2, 3, 4, 5, 6, 7}. This means that T_B can be 0, 2, 4, . . . , 14 ms, based on the outcome of the random variable.*

***d.** We need to mention that if K > 10, it is normally set to 10.*

Figure 12.5 *Vulnerable time for pure ALOHA protocol*


B's end collides with A's beginning
A's end collides with C's beginning
Begin
B
End
Begin
A
End
Begin
C
End
t - T_{fr}
t
t + T_{fr}
Time
Vulnerable time = 2 × T_{fr}

Example 12.2

A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the requirement to make this frame collision-free?

Solution

Average frame transmission time T_{fr} is 200 bits/200 kbps or 1 ms. The vulnerable time is 2 × 1 ms = 2 ms. This means no station should send later than 1 ms before this station starts transmission and no station should start sending during the one 1-ms period that this station is sending.

Note

The throughput for pure ALOHA is $S = G \times e^{-2G}$.

The maximum throughput $S_{max} = 0.184$ when G= (1/2).

Example 12.3

A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces

a. 1000 frames per second b. 500 frames per second

c. 250 frames per second.

Solution

The frame transmission time is 200/200 kbps or 1 ms.

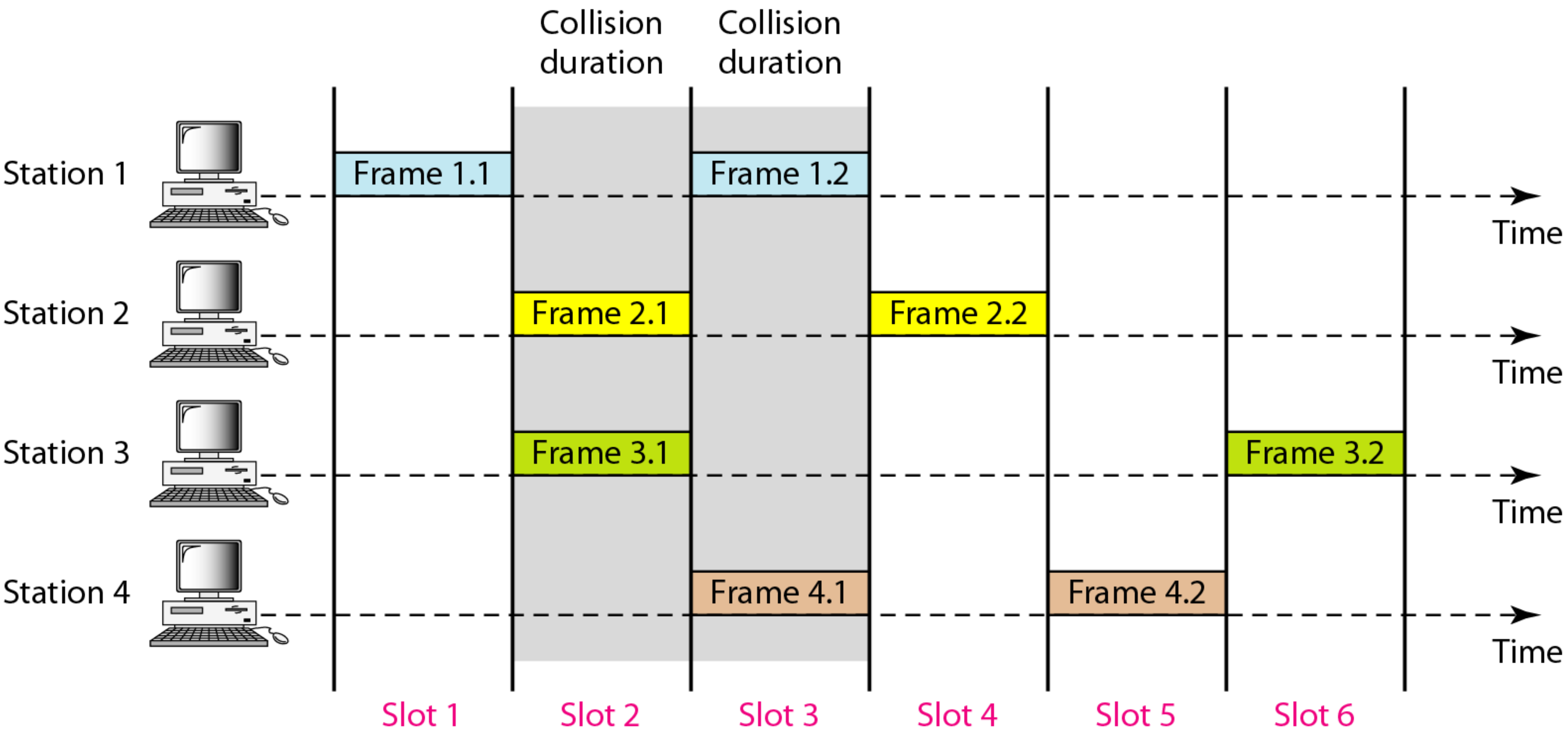
a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case $S = G \times e^{-2G}$ or $S = 0.135$ (13.5 percent). This means that the throughput is 1000 × 0.135 = 135 frames. Only 135 frames out of 1000 will probably survive.

Example 12.3 (continued)

***b.** If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case $S = G \times e^{-2G}$ or S = 0.184 (18.4 percent). This means that the throughput is 500 × 0.184 = 92 and that only 92 frames out of 500 will probably survive. Note that this is the maximum throughput case, percentagewise.*

***c.** If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case $S = G \times e^{-2G}$ or S = 0.152 (15.2 percent). This means that the throughput is 250 × 0.152 = 38. Only 38 frames out of 250 will probably survive.*

Figure 12.6 *Frames in a slotted ALOHA network*

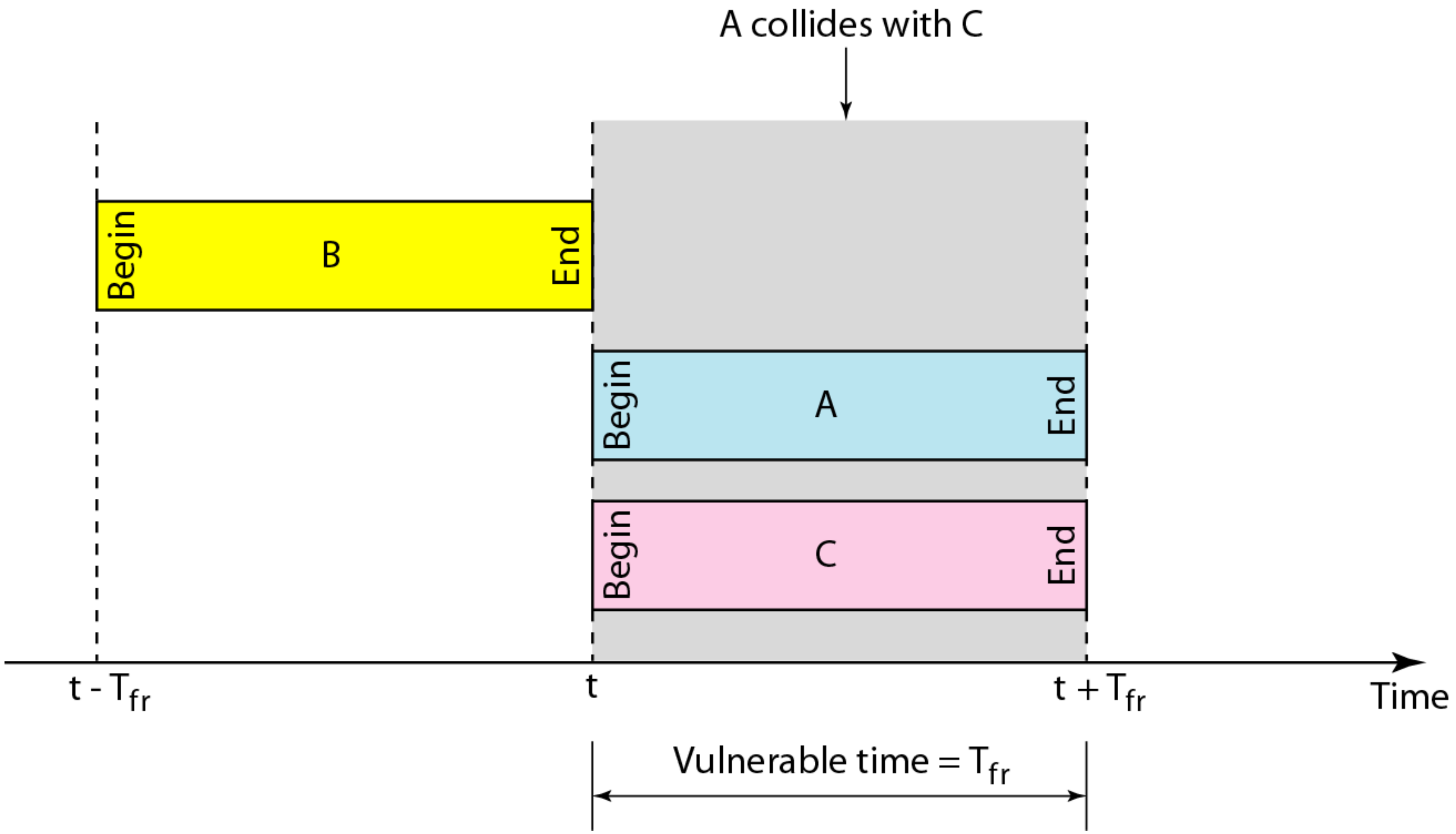


Note

The throughput for slotted ALOHA is $S = G \times e^{-G}$.

The maximum throughput $S_{max} = 0.368$ when G = 1.

Figure 12.7 *Vulnerable time for slotted ALOHA protocol*



Example 12.4

A slotted ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces
a. 1000 frames per second b. 500 frames per second
c. 250 frames per second.

Solution

The frame transmission time is 200/200 kbps or 1 ms.

a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case S = G× e^{-G} or S = 0.368 (36.8 percent). This means that the throughput is 1000 × 0.0368 = 368 frames. Only 386 frames out of 1000 will probably survive.

Example 12.4 (continued)

b. If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case $S = G \times e^{-G}$ or $S = 0.303$ (30.3 percent). This means that the throughput is 500 × 0.0303 = 151. Only 151 frames out of 500 will probably survive.

c. If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case $S = G \times e^{-G}$ or $S = 0.195$ (19.5 percent). This means that the throughput is 250 × 0.195 = 49. Only 49 frames out of 250 will probably survive.

Figure 12.8 *Space/time model of the collision in CSMA*

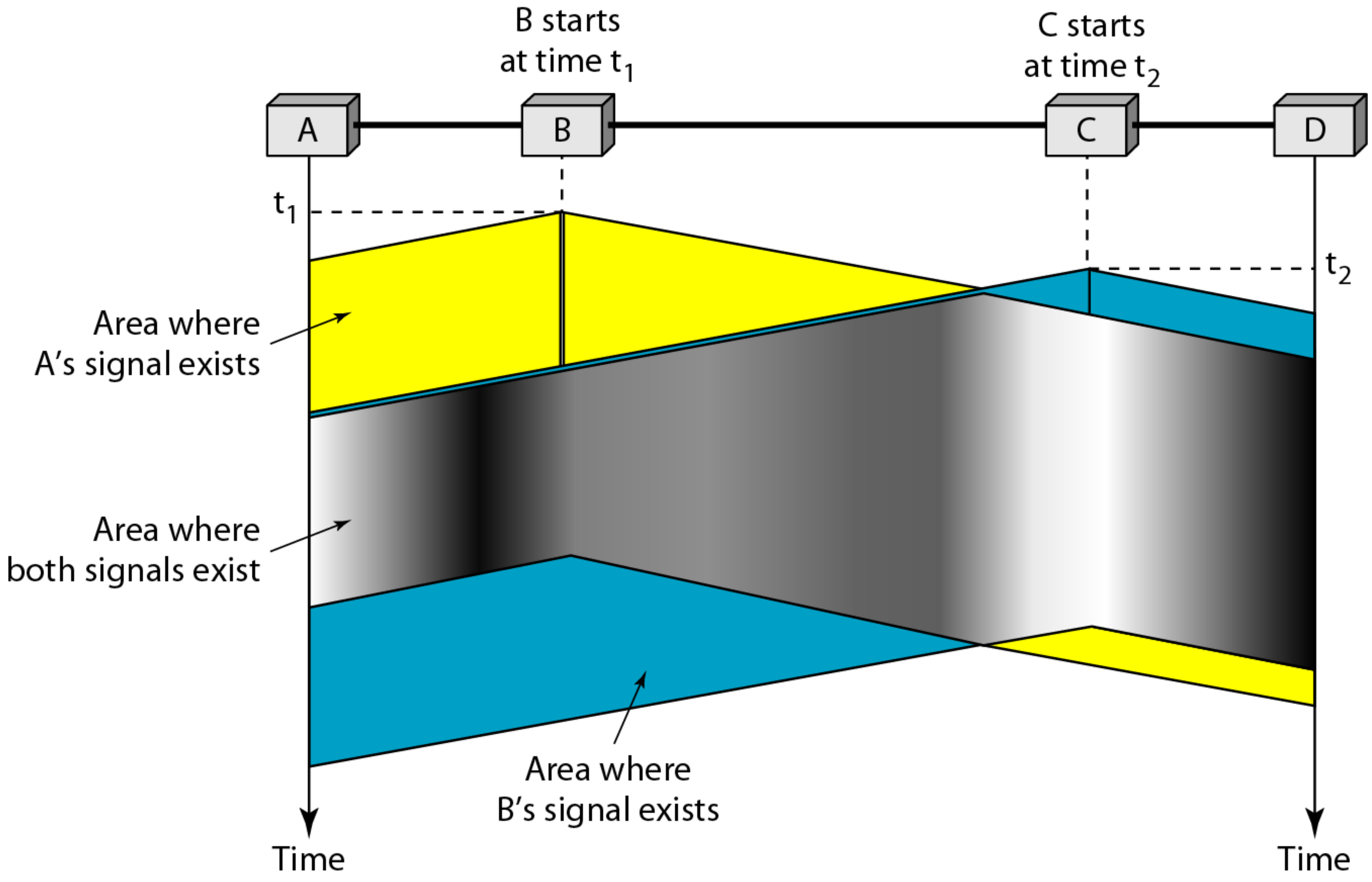


Figure 12.9 *Vulnerable time in CSMA*

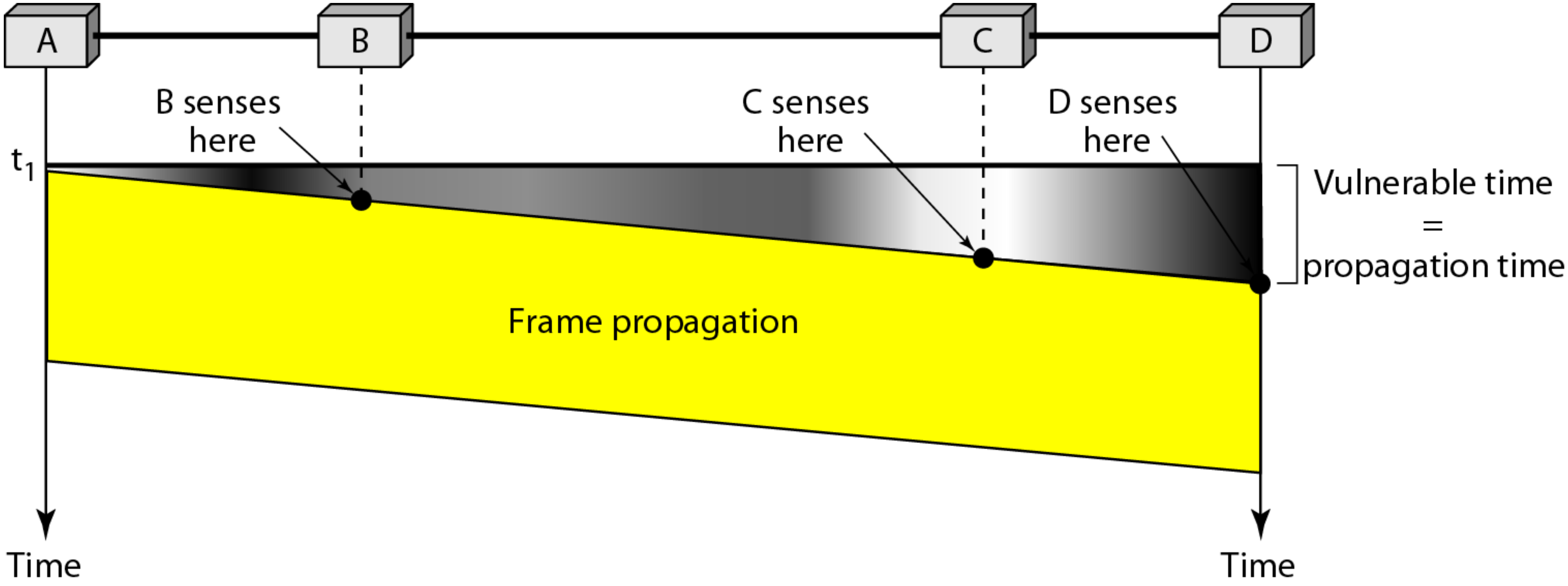
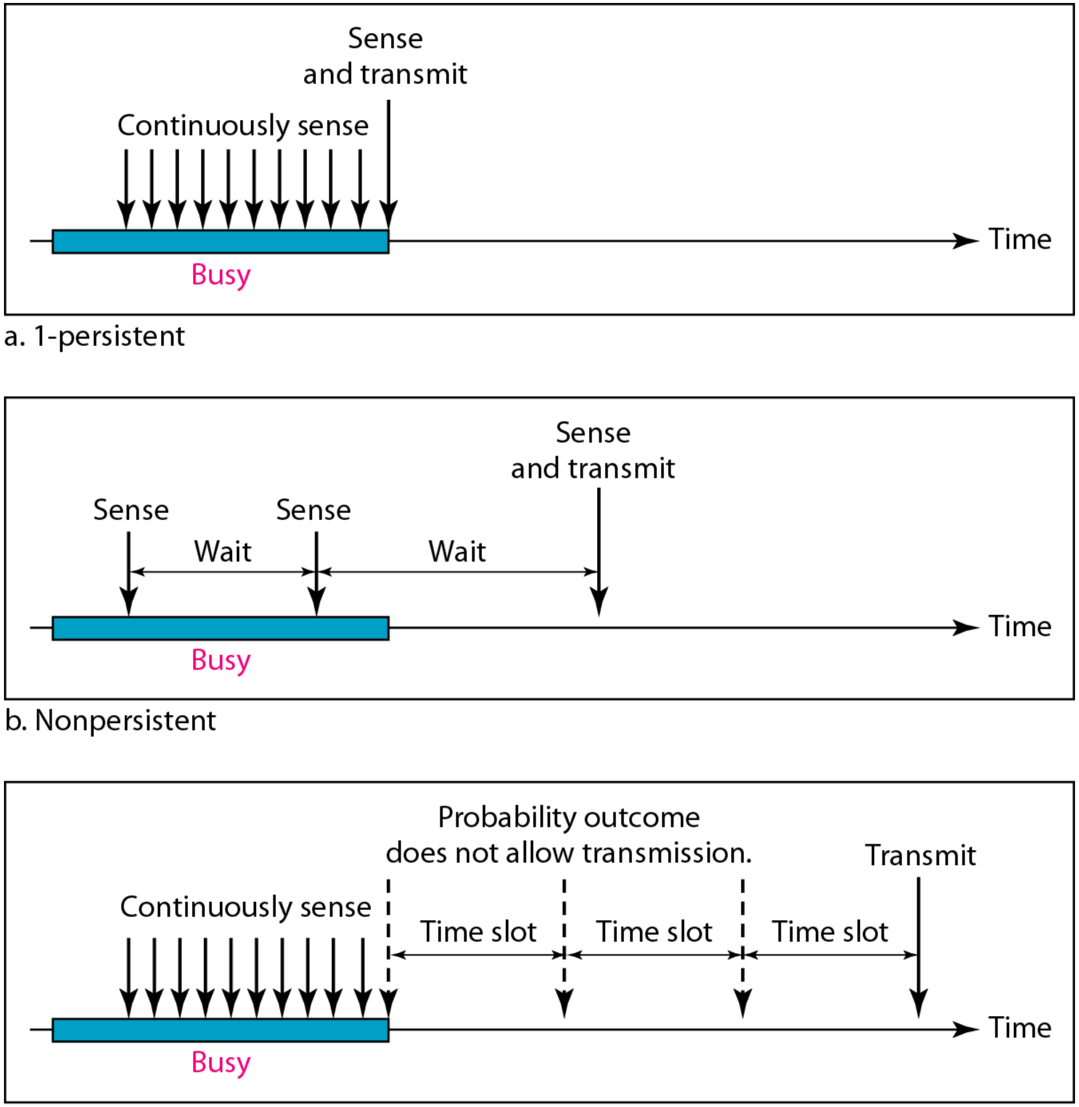


Figure 12.10 *Behavior of three persistence methods*

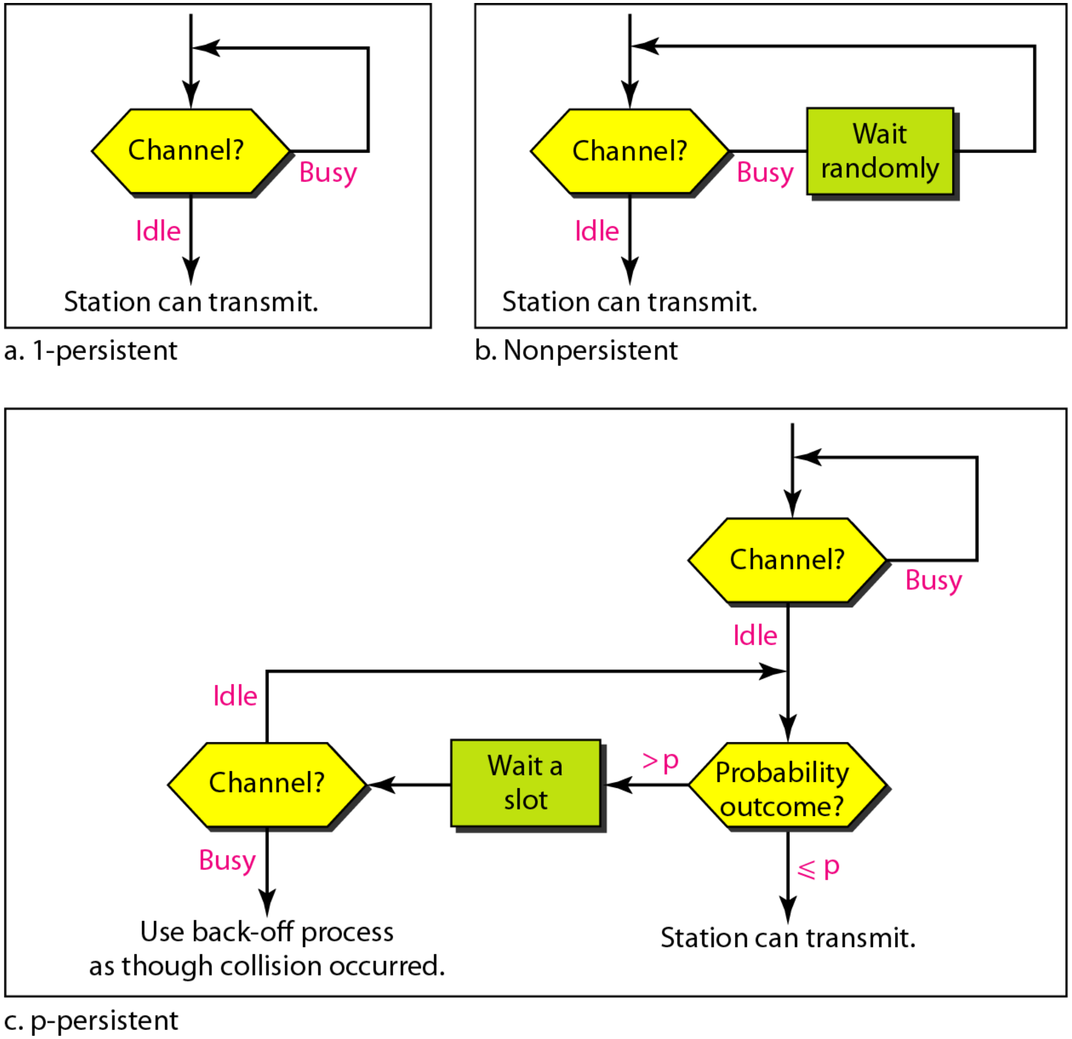


a. 1-persistent

b. Nonpersistent

c. p-persistent

Figure 12.11 *Flow diagram for three persistence methods*



a. 1-persistent

b. Nonpersistent

c. p-persistent

Figure 12.12 *Collision of the first bit in CSMA/CD*

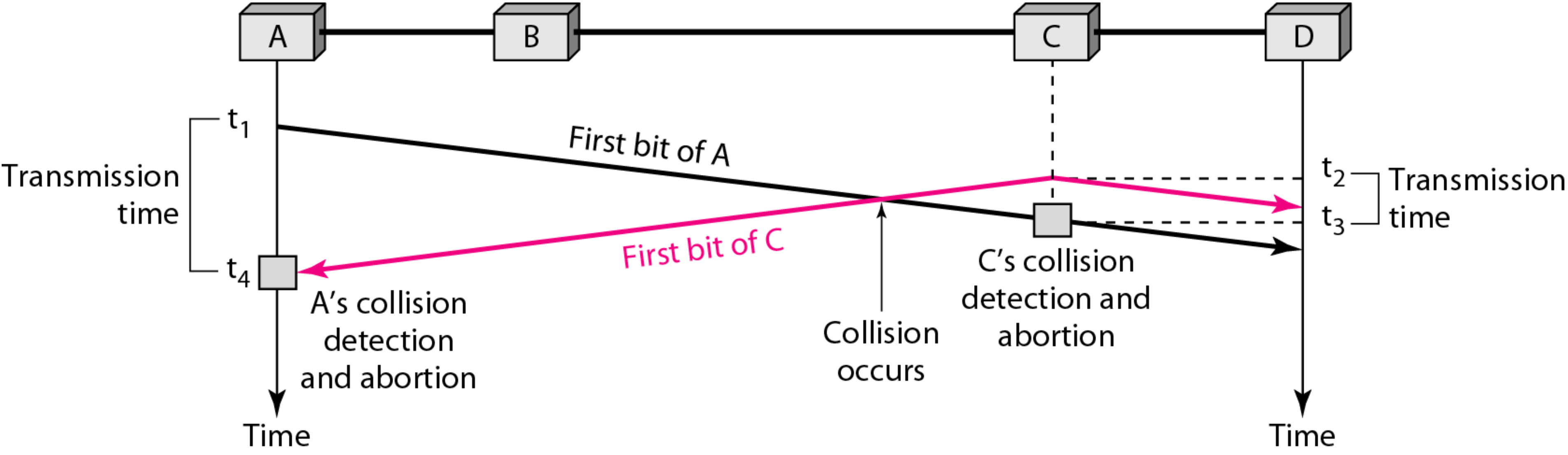
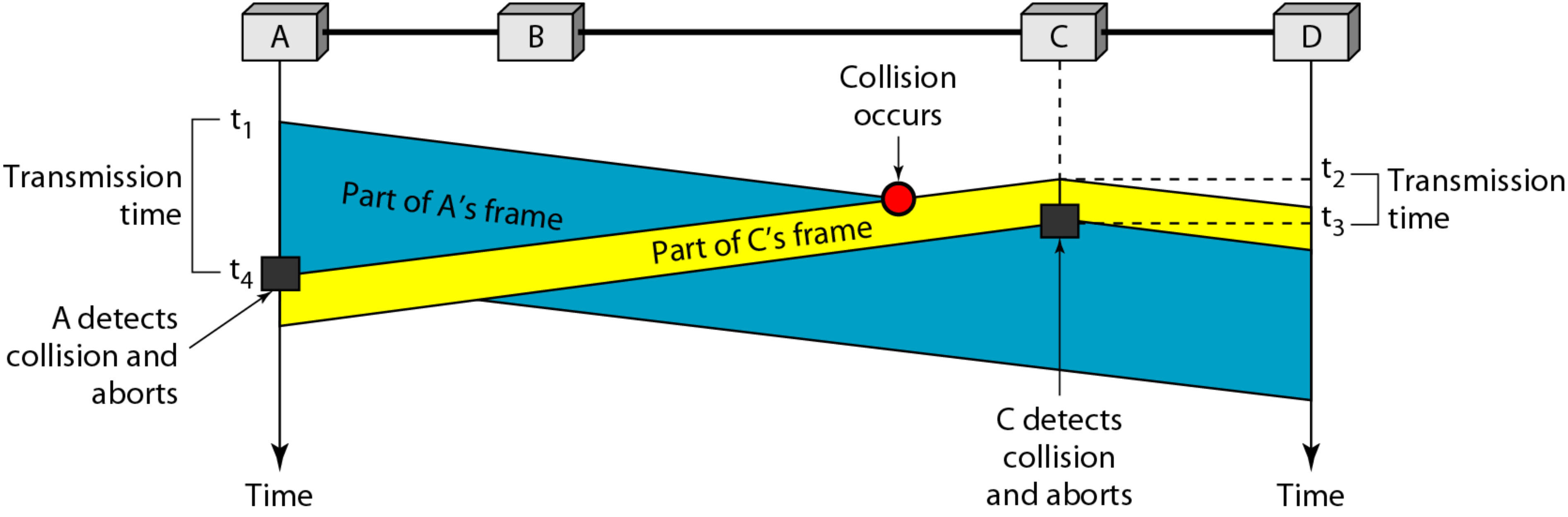


Figure 12.13 *Collision and abortion in CSMA/CD*



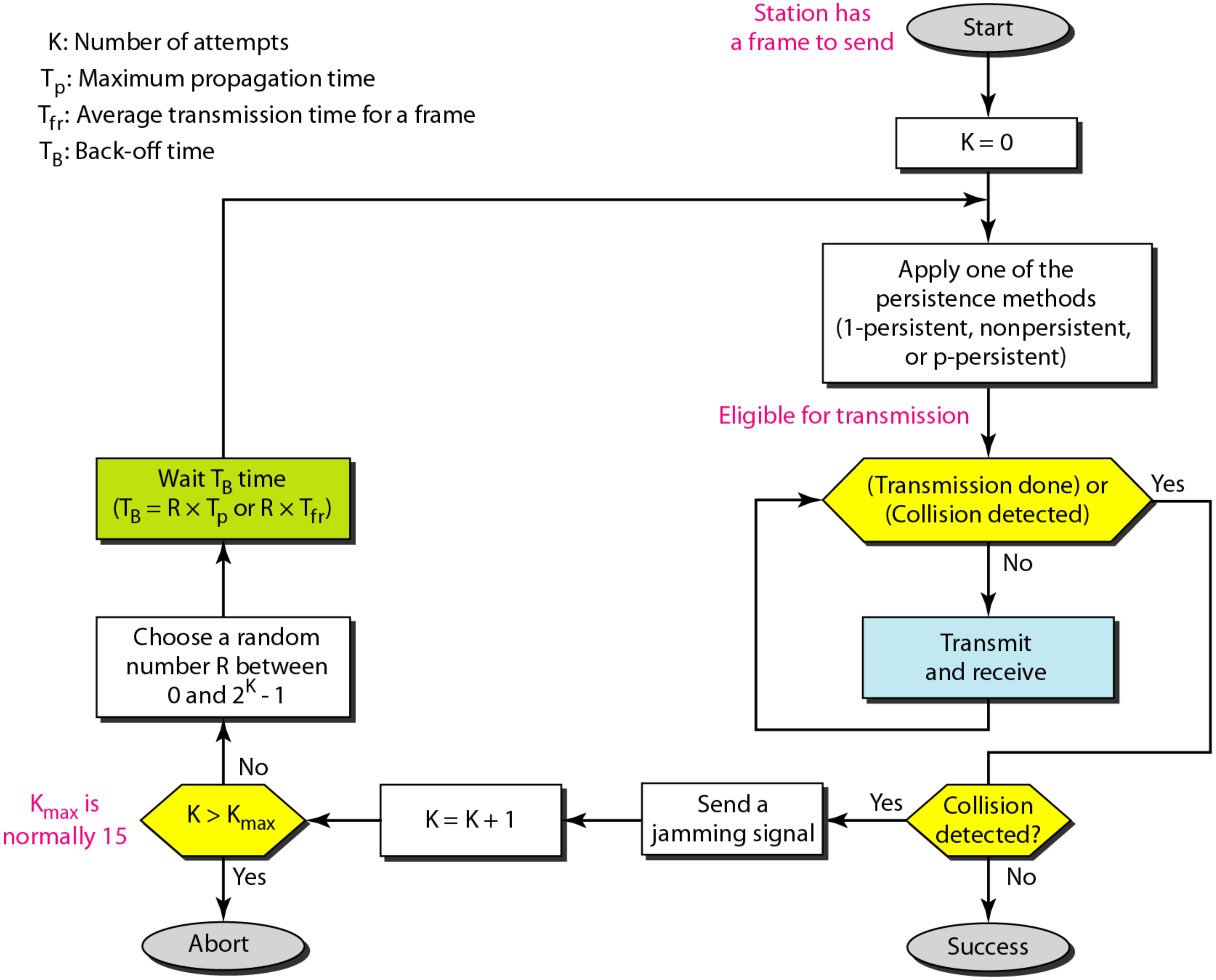
Example 12.5

A network using CSMA/CD has a bandwidth of 10 Mbps. If the maximum propagation time (including the delays in the devices and ignoring the time needed to send a jamming signal, as we see later) is 25.6 μs, what is the minimum size of the frame?

Solution

The frame transmission time is T_{fr} = 2 × T_p = 51.2 μs. This means, in the worst case, a station needs to transmit for a period of 51.2 μs to detect the collision. The minimum size of the frame is 10 Mbps × 51.2 μs = 512 bits or 64 bytes. This is actually the minimum size of the frame for Standard Ethernet.

Figure 12.14 *Flow diagram for the CSMA/CD*



K: Number of attempts
T_p: Maximum propagation time
T_{fr}: Average transmission time for a frame
T_B: Back-off time
Station has a frame to send
Start
K = 0
Apply one of the persistence methods (1-persistent, nonpersistent, or p-persistent)
Eligible for transmission
(Transmission done) or (Collision detected)
Yes
No
Transmit and receive
Collision detected?
Yes
No
Success
Send a jamming signal
K = K + 1
$K > K_{max}$
K_{max} is normally 15
No
Yes
Abort
Choose a random number R between 0 and $2^K - 1$
Wait T_B time ($T_B = R \times T_p$ or $R \times T_{fr}$)

Figure 12.15 *Energy level during transmission, idleness, or collision*

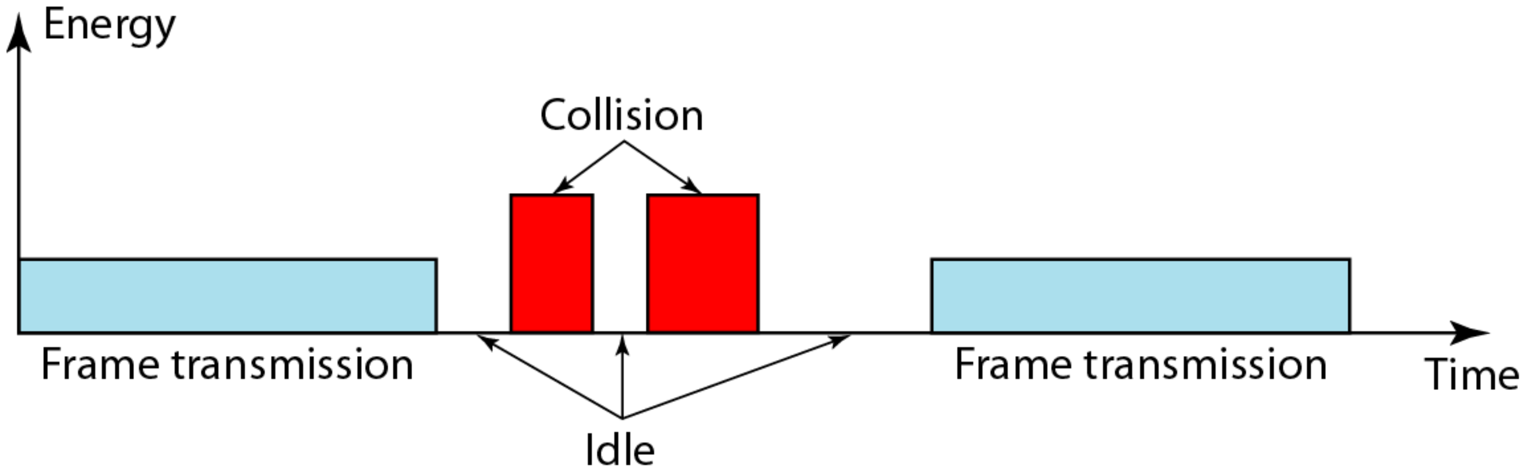
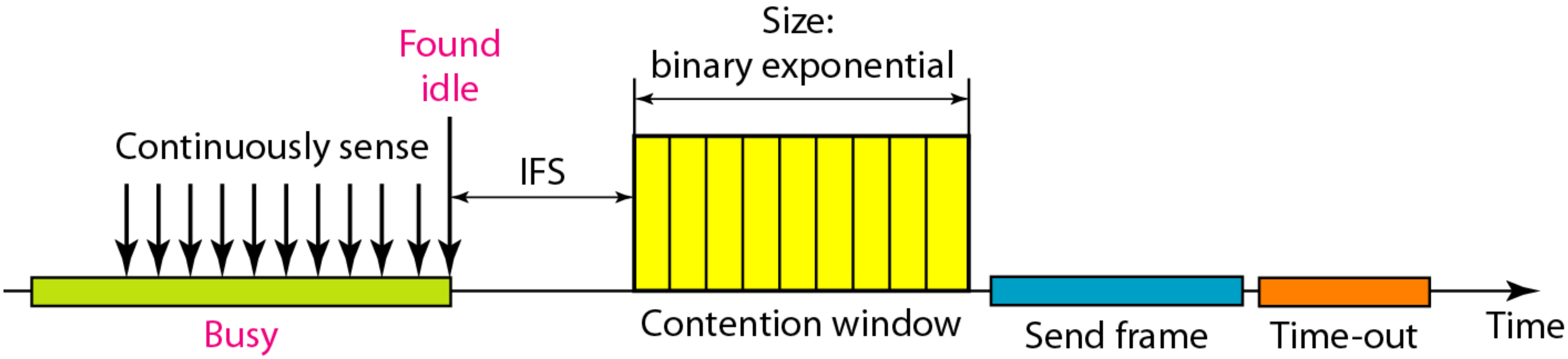


Figure 12.16 *Timing in CSMA/CA*



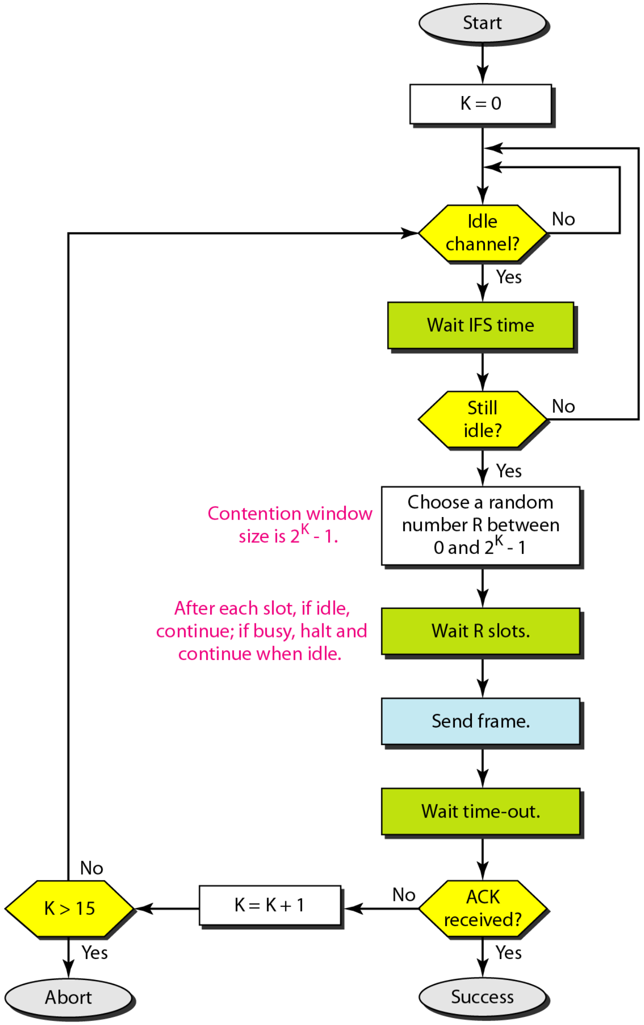
Note

In CSMA/CA, the IFS can also be used to define the priority of a station or a frame.

Note

In CSMA/CA, if the station finds the channel busy, it does not restart the timer of the contention window;
it stops the timer and restarts it when the channel becomes idle.

Figure 12.17 *Flow diagram for CSMA/CA*



Contention window size is $2^K - 1$.

After each slot, if idle, continue; if busy, halt and continue when idle.

12-2 CONTROLLED ACCESS

In controlled access, the stations consult one another to find which station has the right to send. A station cannot send unless it has been authorized by other stations. We discuss three popular controlled-access methods.

Topics discussed in this section:

Reservation
Polling
Token Passing

Figure 12.18 *Reservation access method*

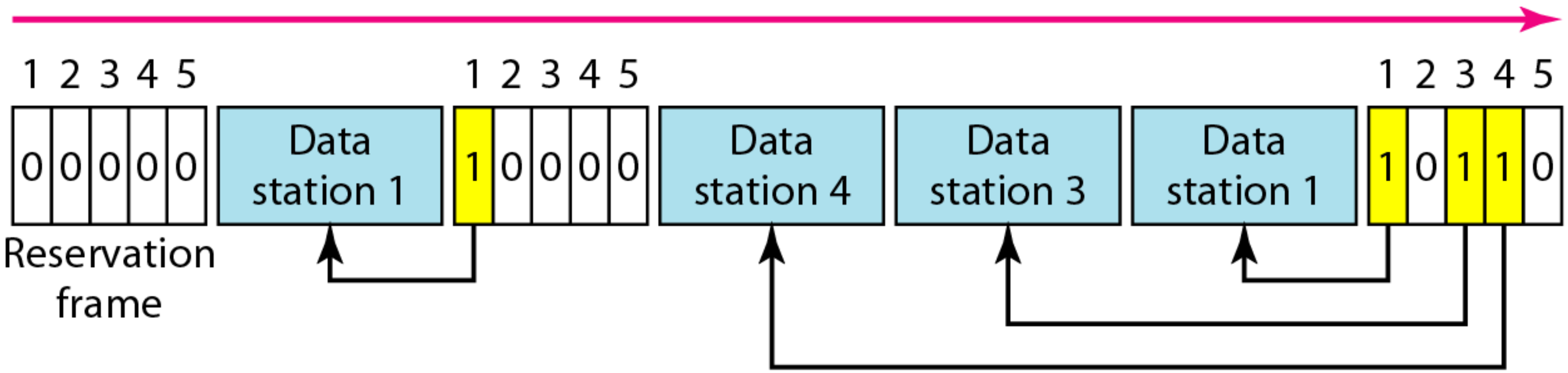


Figure 12.19 *Select and poll functions in polling access method*

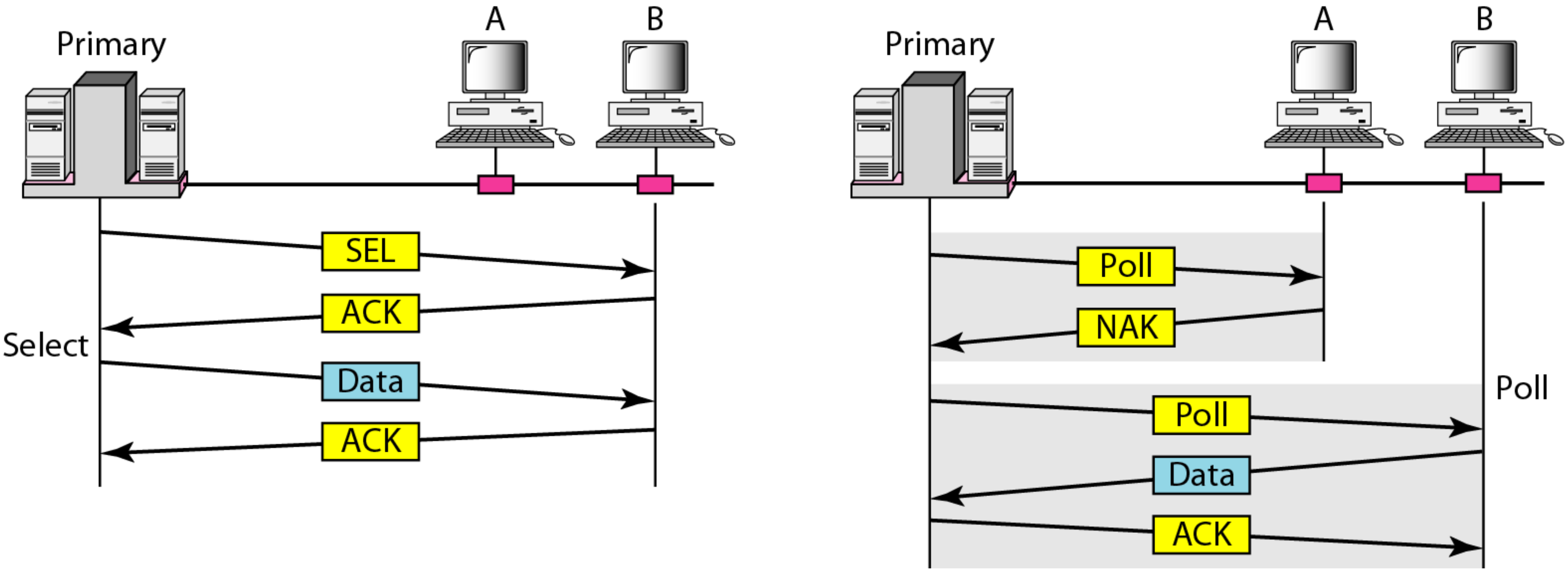
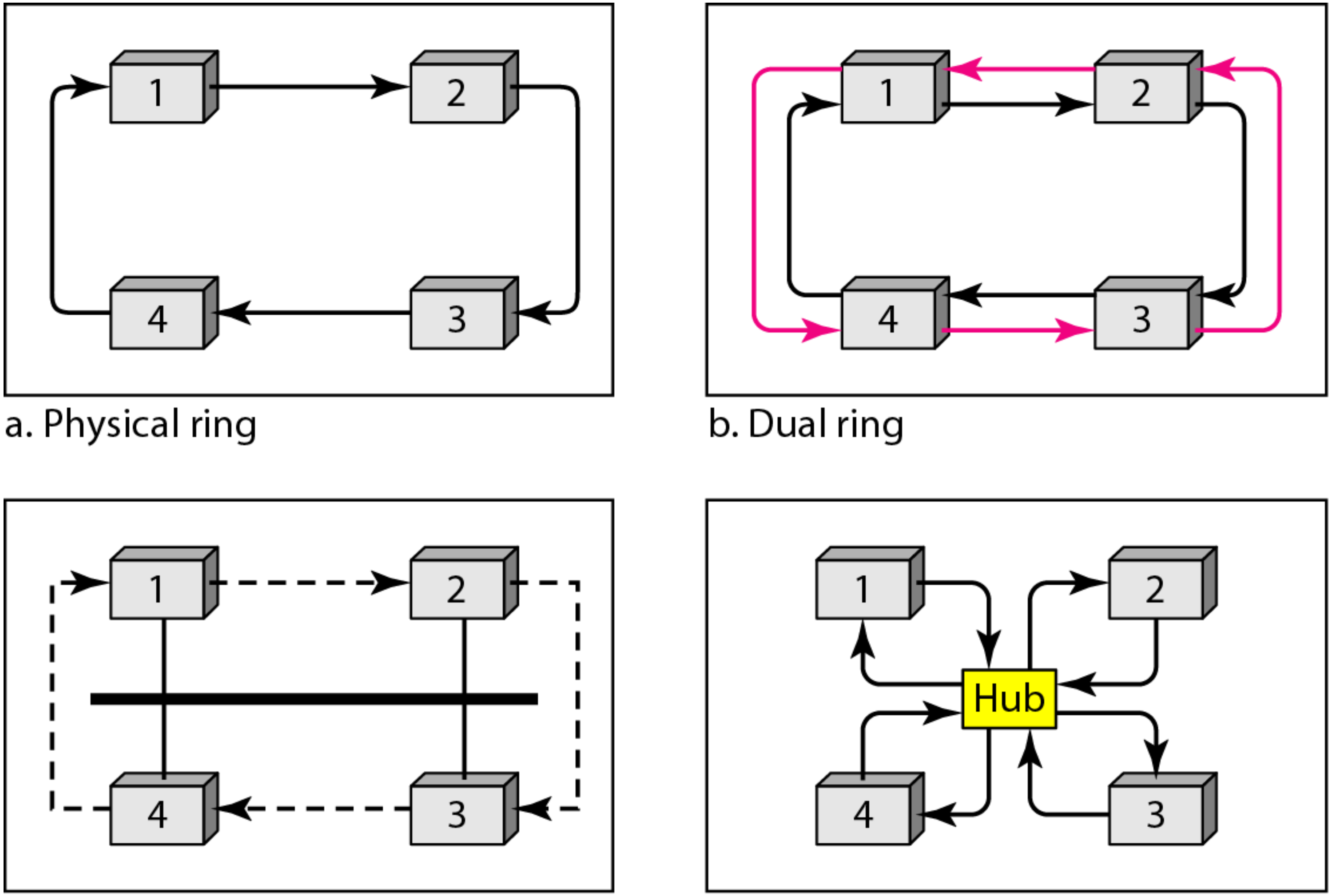


Figure 12.20 *Logical ring and physical topology in token-passing access method*



a. Physical ring

b. Dual ring

c. Bus ring

d. Star ring

12-3 CHANNELIZATION

Channelization is a multiple-access method in which the available bandwidth of a link is shared in time, frequency, or through code, between different stations. In this section, we discuss three channelization protocols.

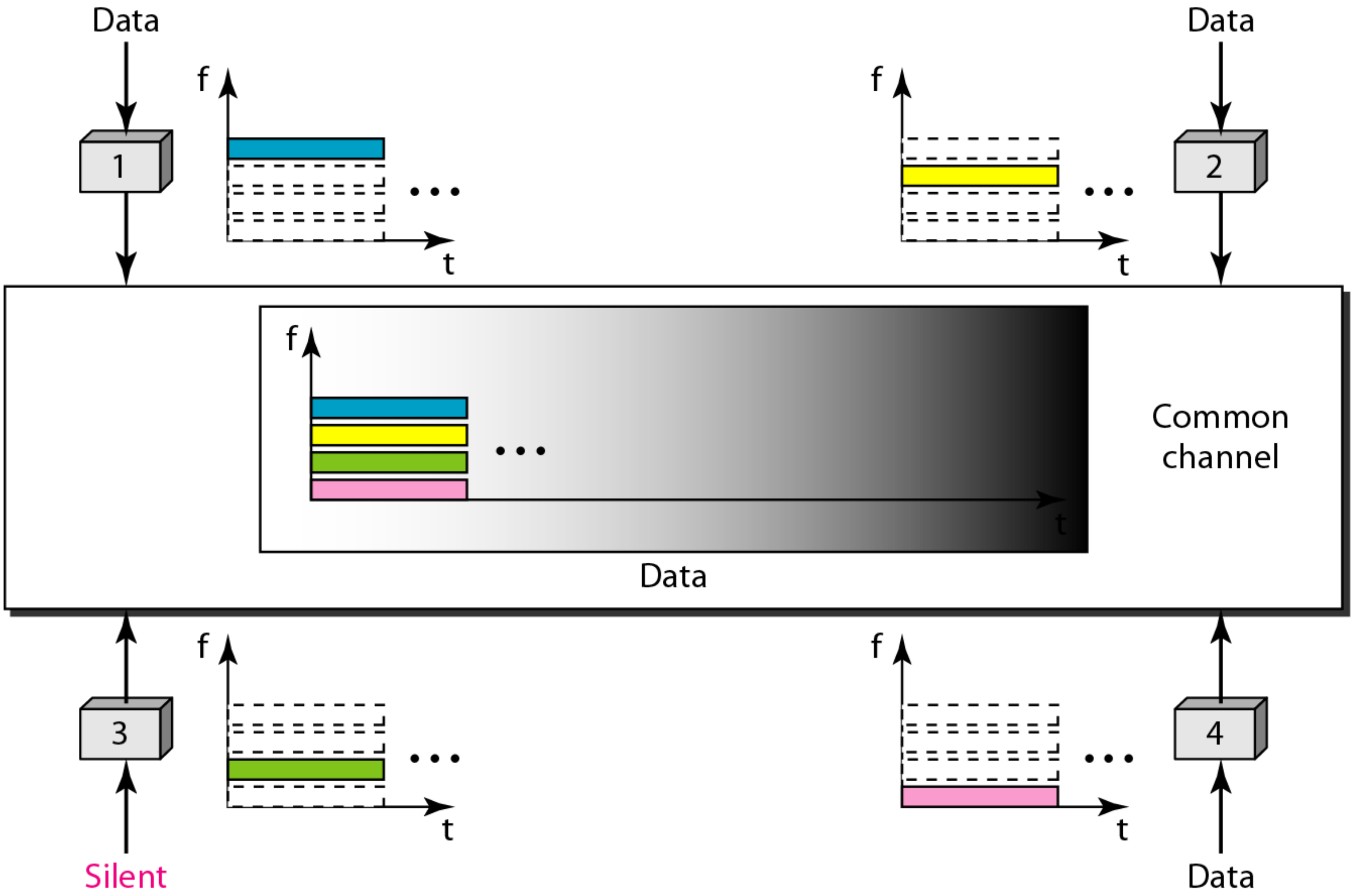
Topics discussed in this section:

Frequency-Division Multiple Access (FDMA)
Time-Division Multiple Access (TDMA)
Code-Division Multiple Access (CDMA)

Note

We see the application of all these methods in Chapter 16 when we discuss cellular phone systems.

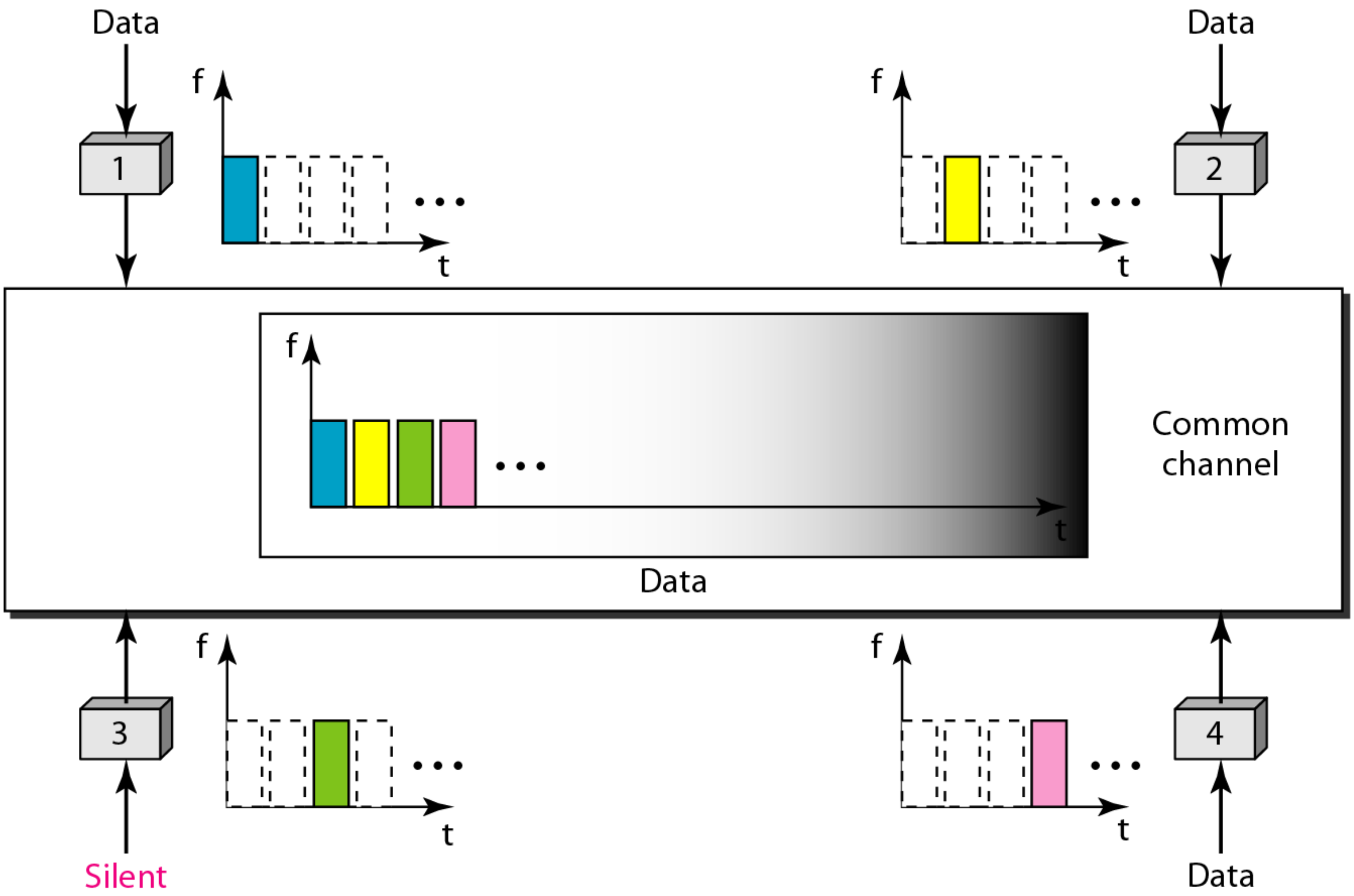
Figure 12.21 *Frequency-division multiple access (FDMA)*



Note

In FDMA, the available bandwidth of the common channel is divided into bands that are separated by guard bands.

Figure 12.22 *Time-division multiple access (TDMA)*



Note

In TDMA, the bandwidth is just one channel that is timeshared between different stations.

Note

In CDMA, one channel carries all transmissions simultaneously.

Figure 12.23 *Simple idea of communication with code*

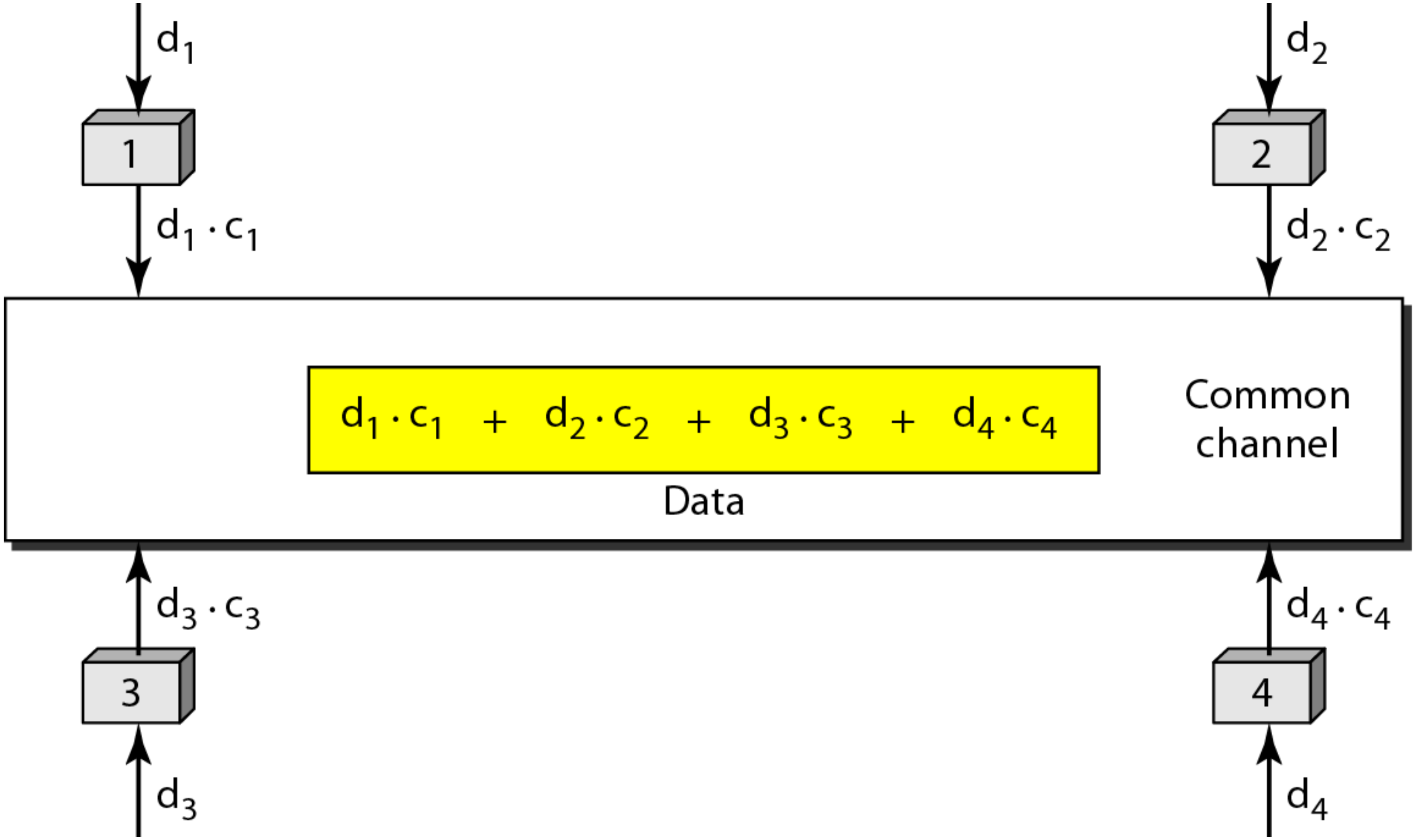
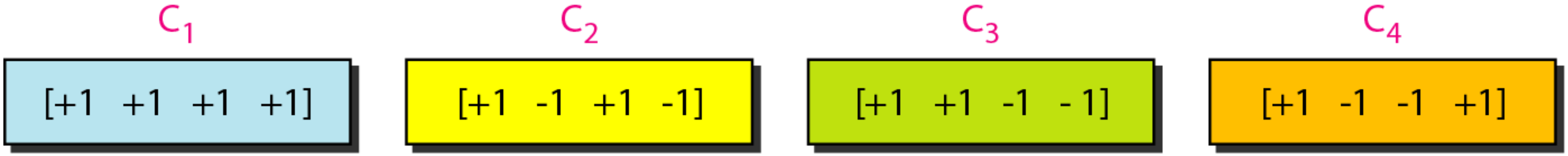


Figure 12.24 *Chip sequences*



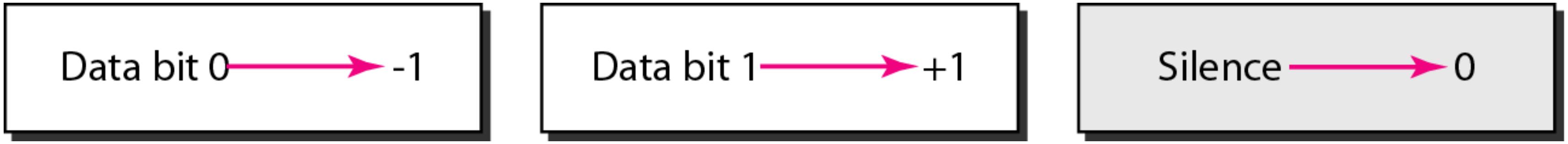
C_1 [+1 +1 +1 +1]

C_2 [+1 -1 +1 -1]

C_3 [+1 +1 -1 - 1]

C_4 [+1 -1 -1 +1]

Figure 12.25 *Data representation in CDMA*



Data bit 0 → -1

Data bit 1 → +1

Silence → 0

Figure 12.26 *Sharing channel in CDMA*

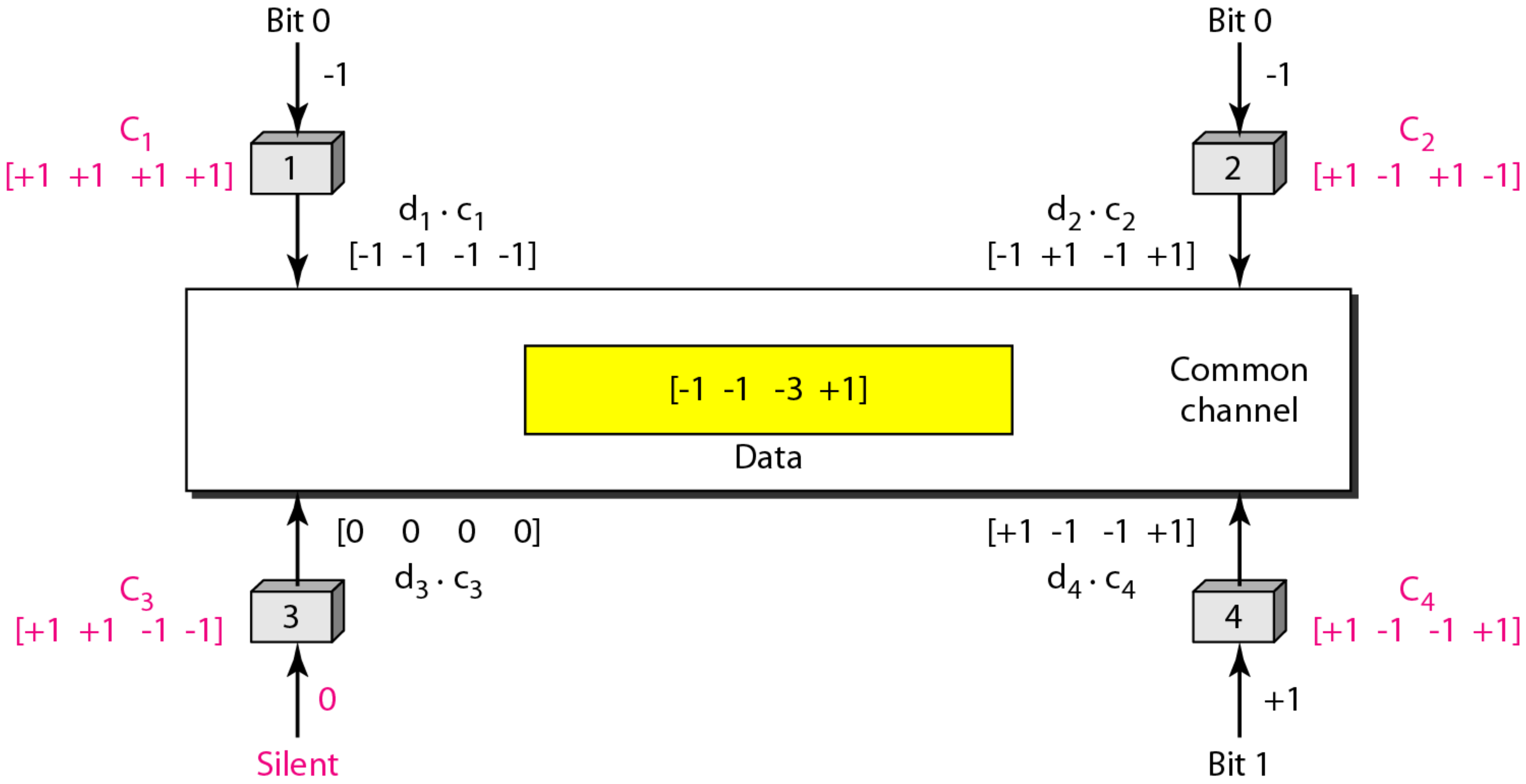


Figure 12.27 *Digital signal created by four stations in CDMA*

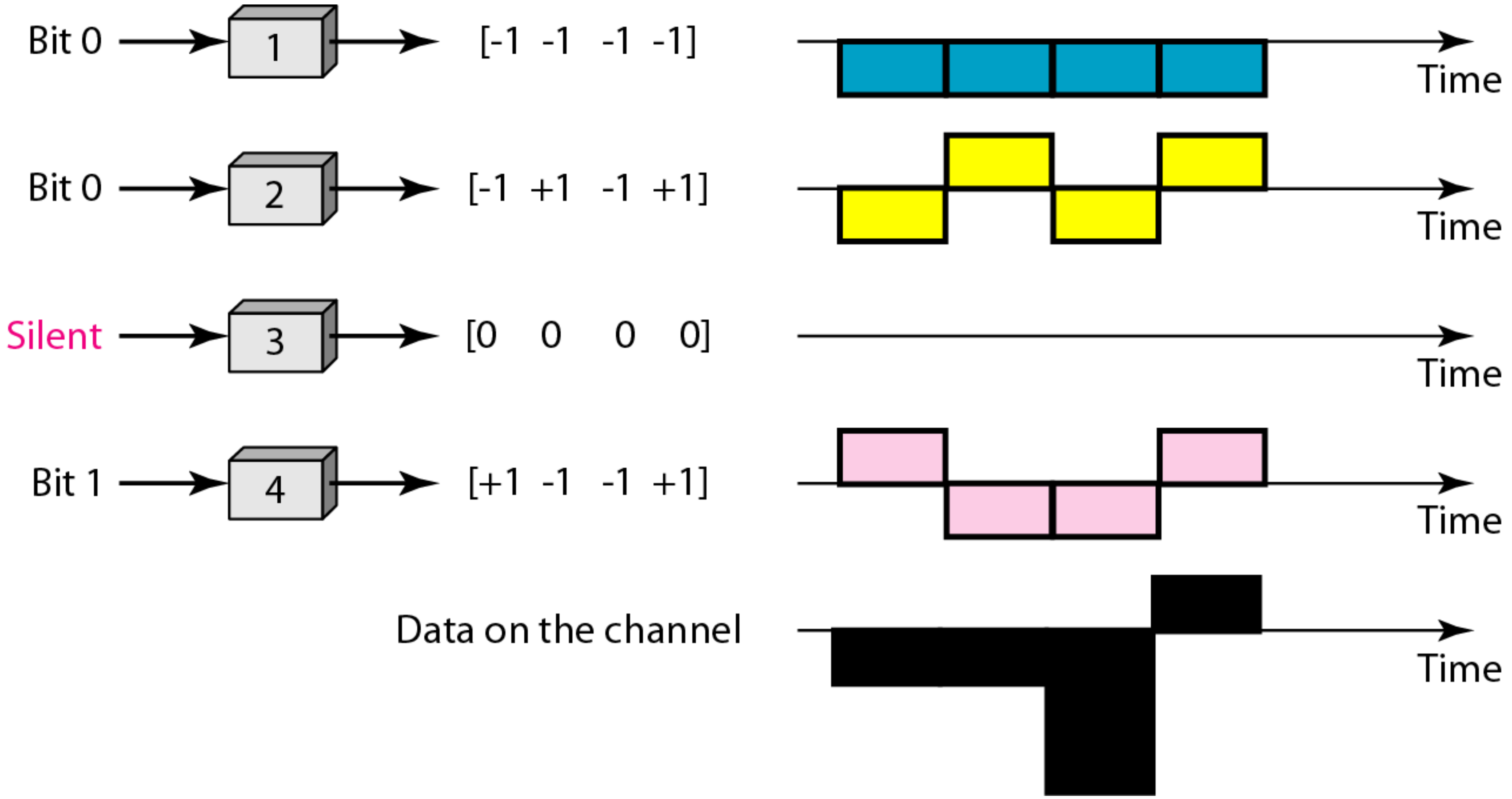


Figure 12.28 *Decoding of the composite signal for one in CDMA*

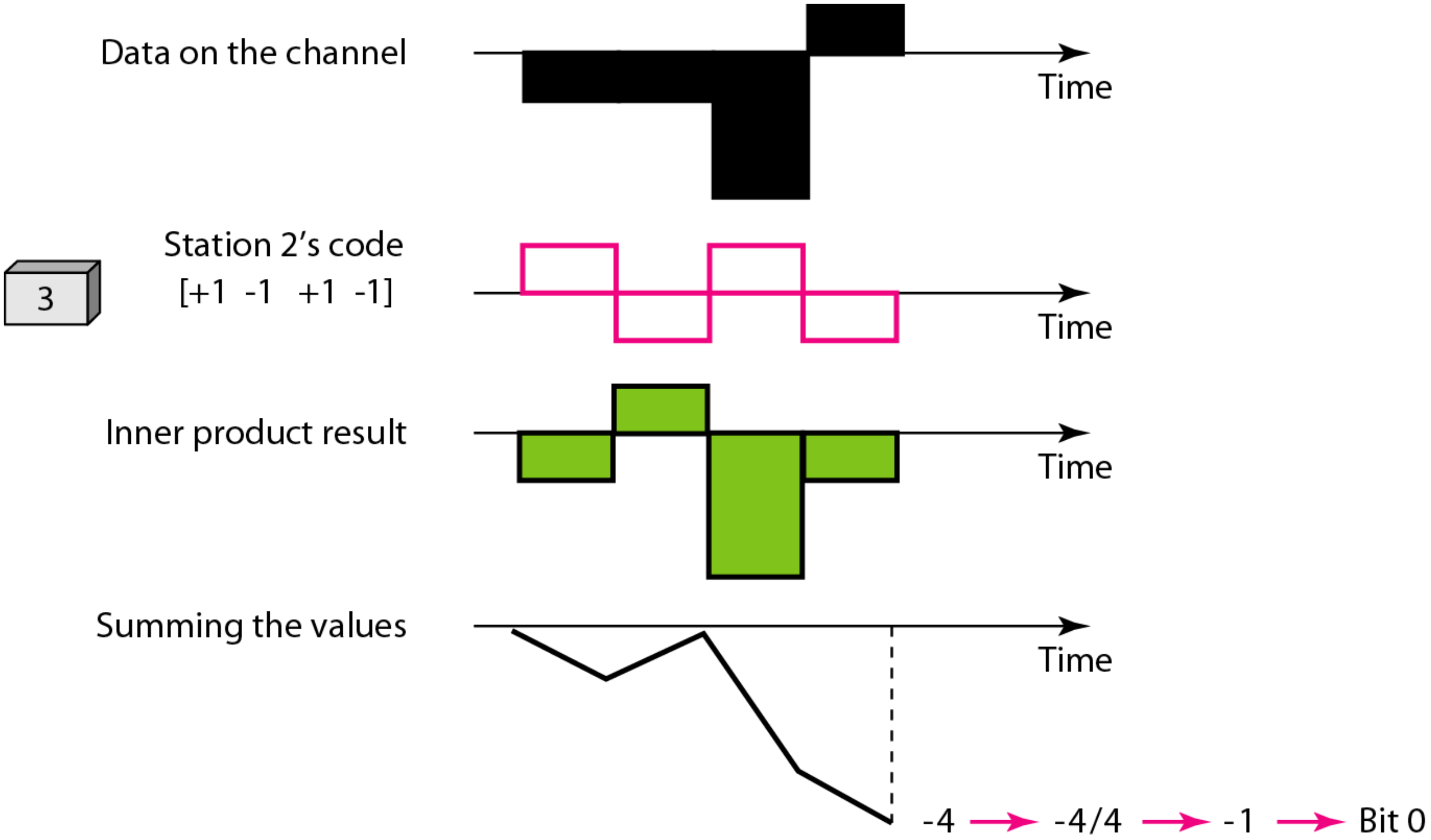


Figure 12.29 *General rule and examples of creating Walsh tables*

$$W_1 = \begin{bmatrix} +1 \end{bmatrix} \qquad W_{2N} = \begin{bmatrix} W_N & W_N \\ W_N & \overline{W_N} \end{bmatrix}$$

a. Two basic rules

$$W_1 = \begin{bmatrix} +1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \qquad W_4 = \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix}$$

b. Generation of W_1, W_2, and W_4

Note

The number of sequences in a Walsh table needs to be N = 2^m.

Example 12.6

Find the chips for a network with

a. Two stations ***b. Four stations***

Solution

We can use the rows of W_2 and W_4 in Figure 12.29:

a. For a two-station network, we have

[+1 +1] and [+1 −1].

b. For a four-station network we have

[+1 +1 +1 +1], [+1 −1 +1 −1],

[+1 +1 −1 −1], *and* [+1 −1 −1 +1].

Example 12.7

What is the number of sequences if we have 90 stations in our network?

Solution

The number of sequences needs to be 2^m. We need to choose m = 7 and N = 2^7 or 128. We can then use 90 of the sequences as the chips.

Example 12.8

Prove that a receiving station can get the data sent by a specific sender if it multiplies the entire data on the channel by the sender's chip code and then divides it by the number of stations.

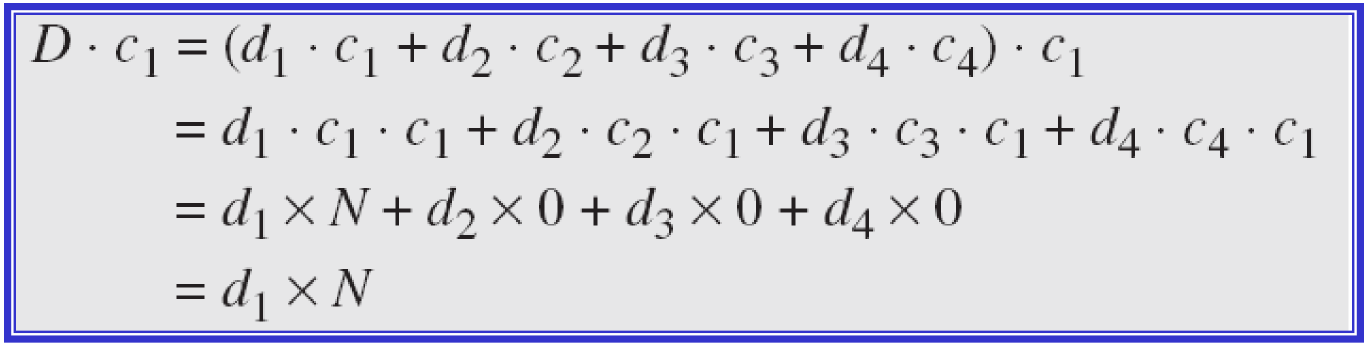
Solution

Let us prove this for the first station, using our previous four-station example. We can say that the data on the channel

$$D = (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4).$$

The receiver which wants to get the data sent by station 1 multiplies these data by c_1.

Example 12.8 (continued)



$$\begin{aligned} D \cdot c_1 &= (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4) \cdot c_1 \\ &= d_1 \cdot c_1 \cdot c_1 + d_2 \cdot c_2 \cdot c_1 + d_3 \cdot c_3 \cdot c_1 + d_4 \cdot c_4 \cdot c_1 \\ &= d_1 \times N + d_2 \times 0 + d_3 \times 0 + d_4 \times 0 \\ &= d_1 \times N \end{aligned}$$

When we divide the result by N, we get d_1 .

COMPUTER COMMUNICATION NETWORKS

(15EC64)

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*

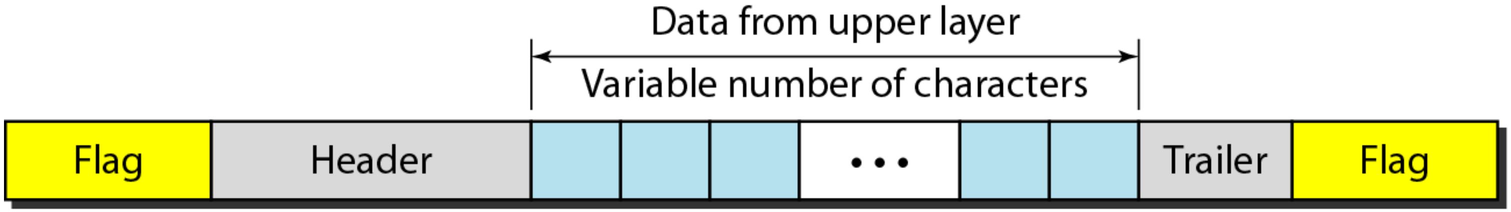
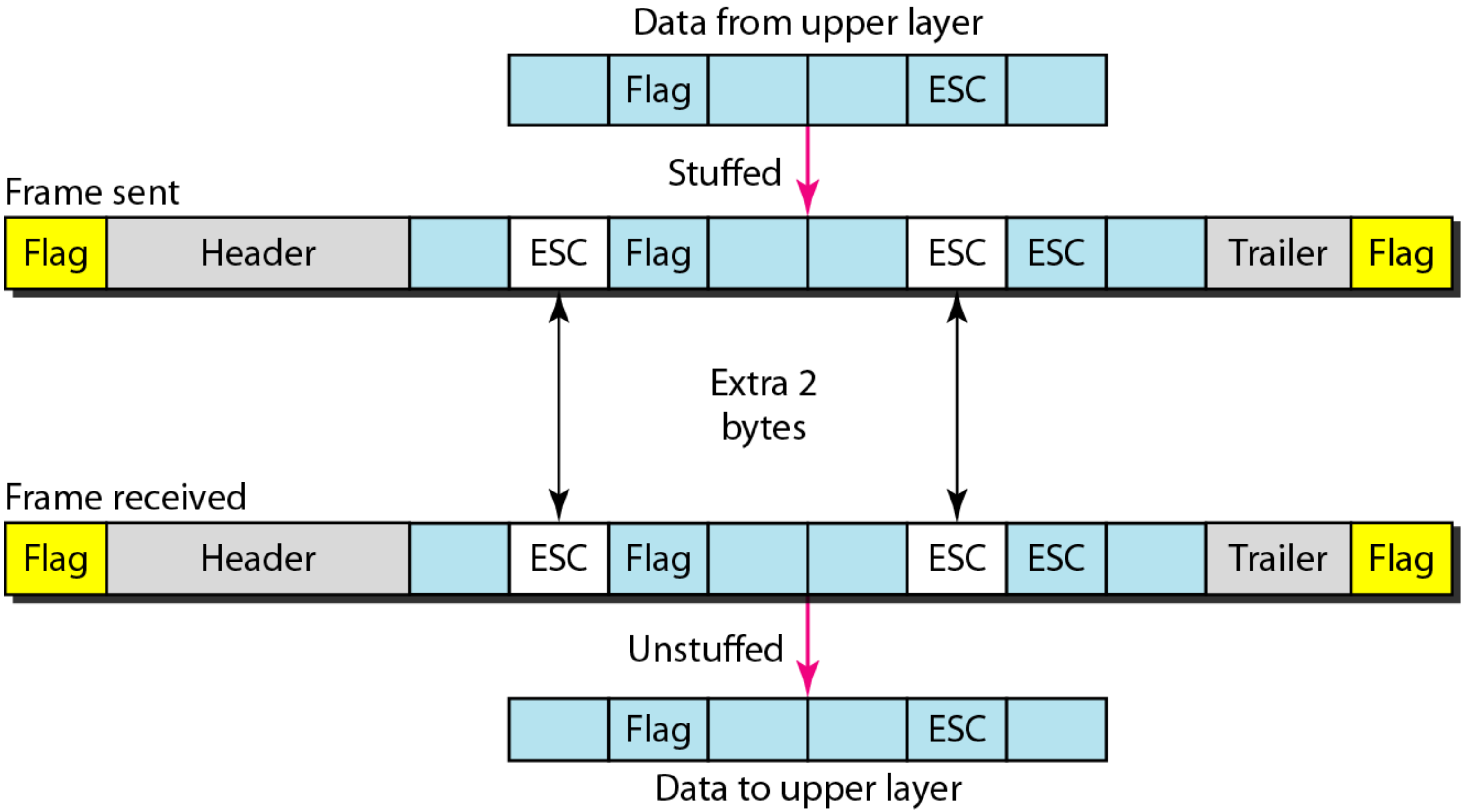


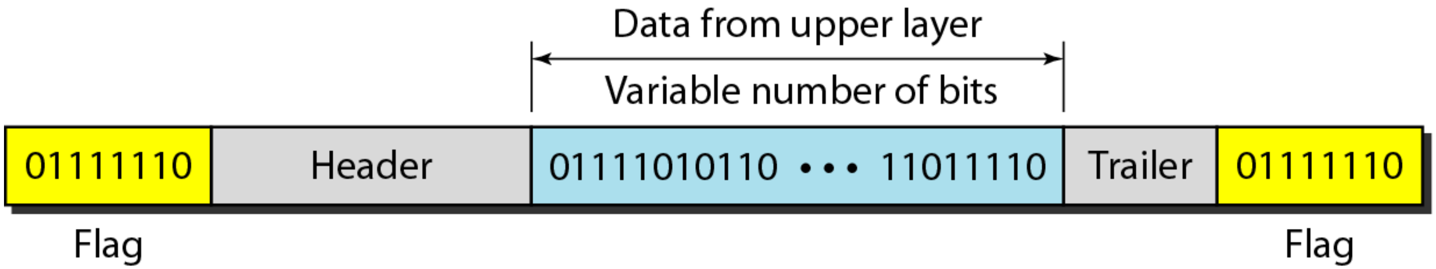
Figure 11.2 *Byte stuffing and unstuffing*



Note

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

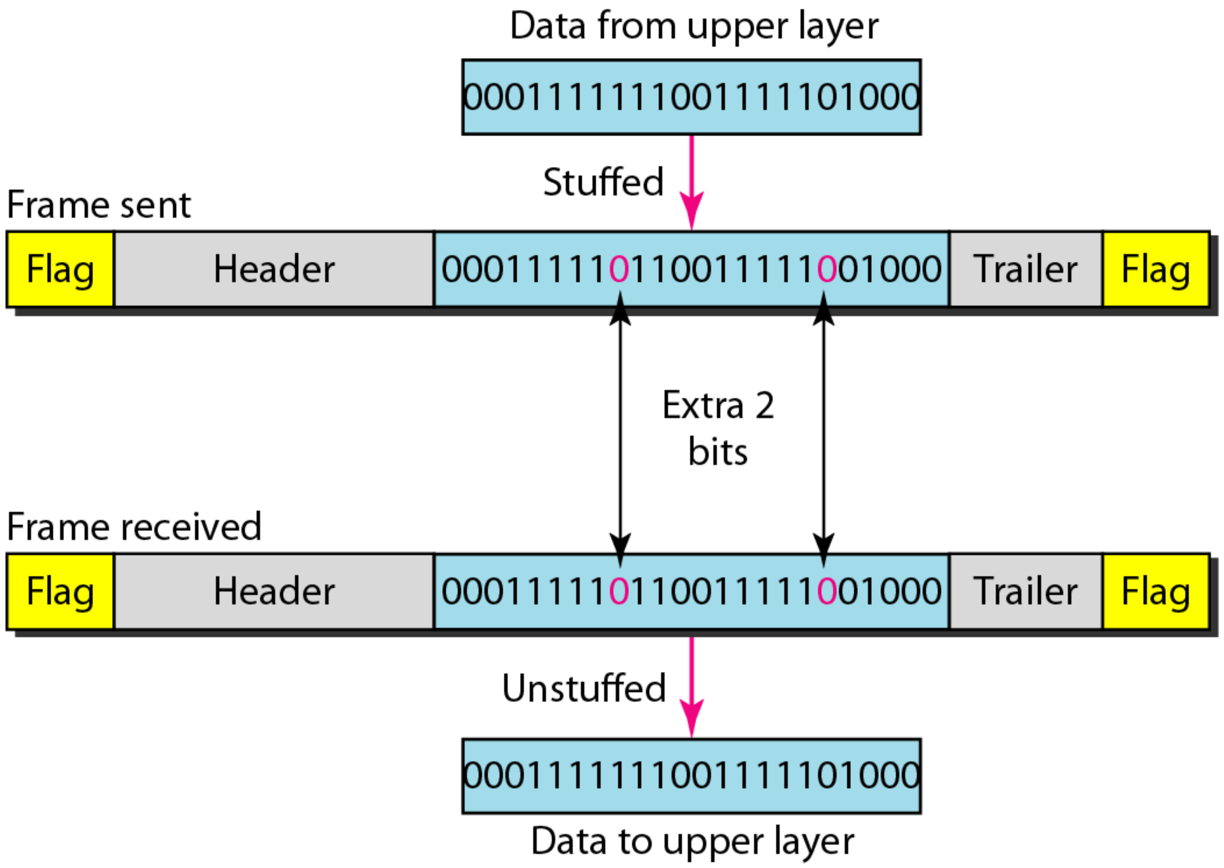
Figure 11.3 *A frame in a bit-oriented protocol*



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.

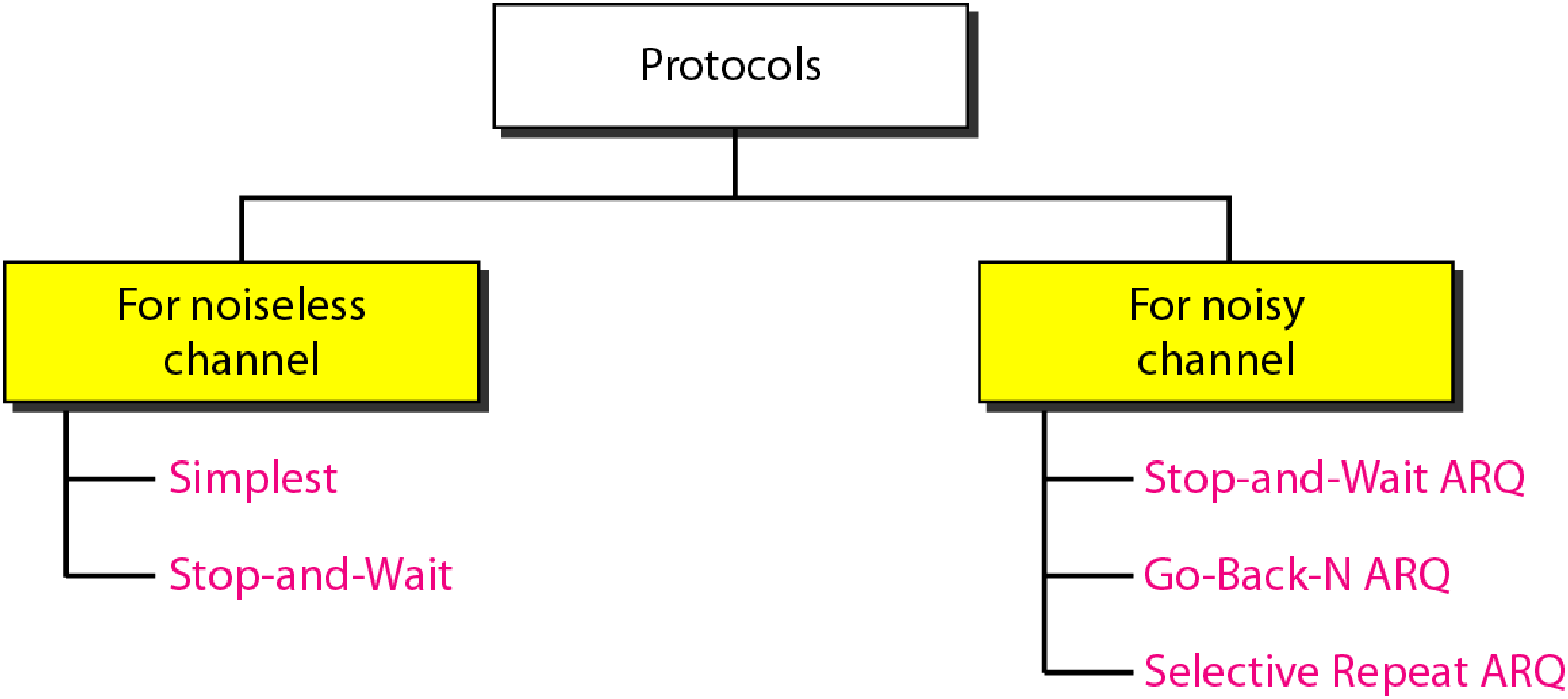
Note

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*



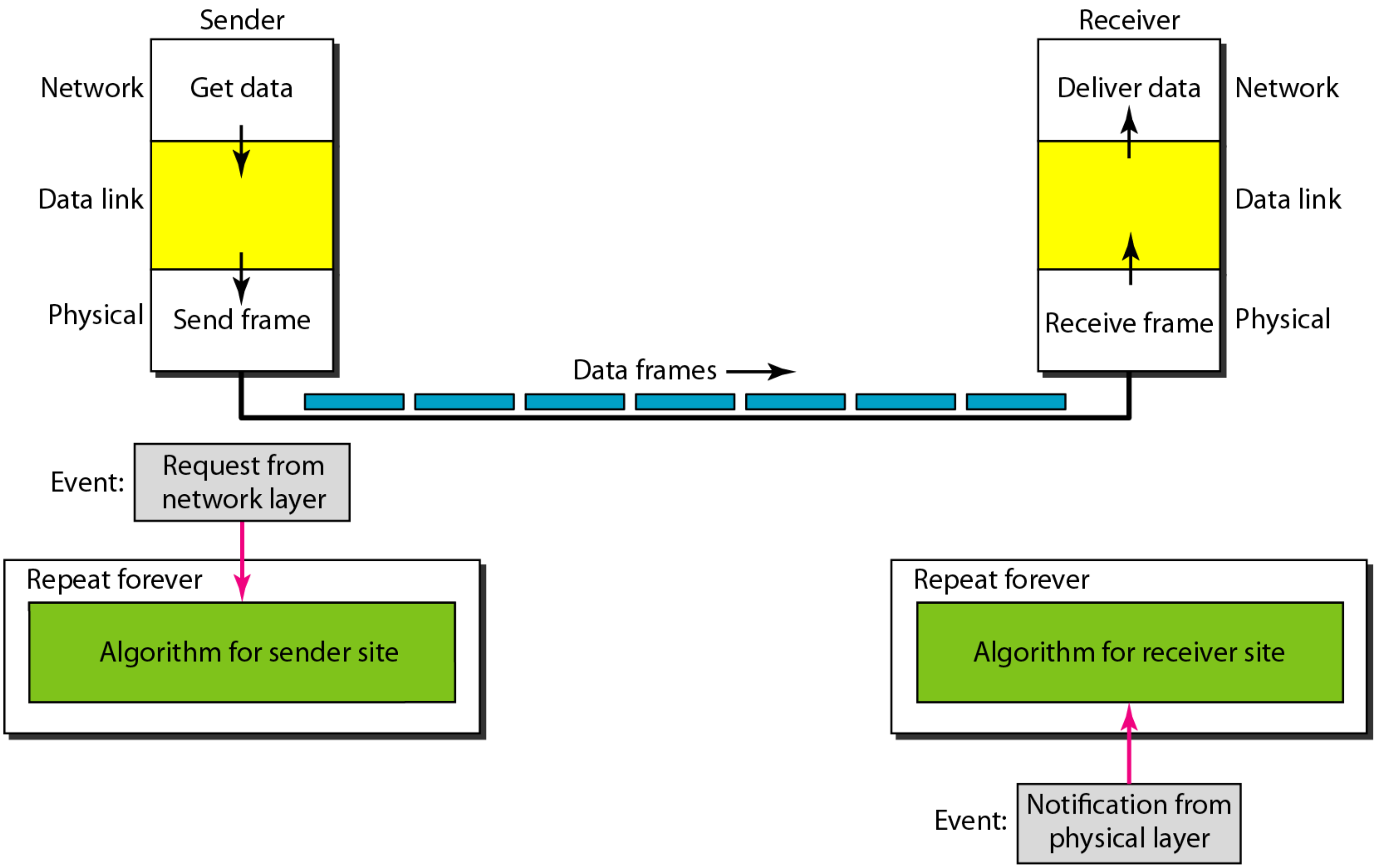
11-4 NOISELESS CHANNELS

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*

```
while(true)                    // Repeat forever
{
  WaitForEvent();              // Sleep until an event occurs
  if(Event(RequestToSend))     //There is a packet to send
  {
     GetData();
     MakeFrame();
     SendFrame();              //Send the frame
  }
}
```

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

```
while(true)                         // Repeat forever
{
  WaitForEvent();                   // Sleep until an event occurs
  if(Event(ArrivalNotification))   //Data frame arrived
  {
     ReceiveFrame();
     ExtractData();
     DeliverData();                 //Deliver data to network layer
  }
}
```

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*

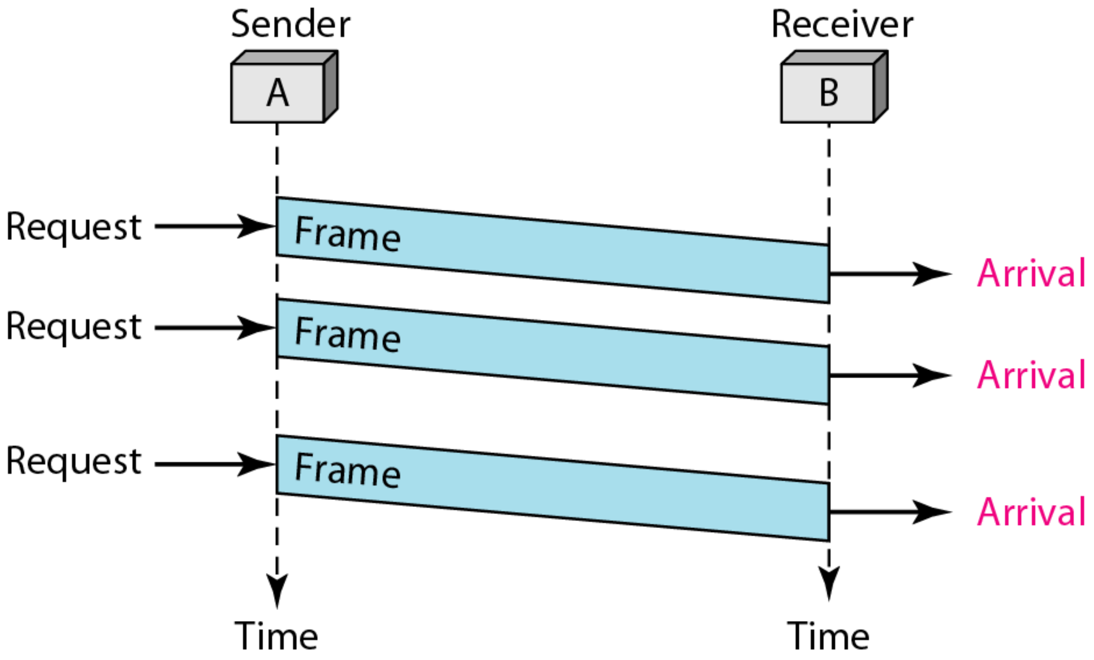
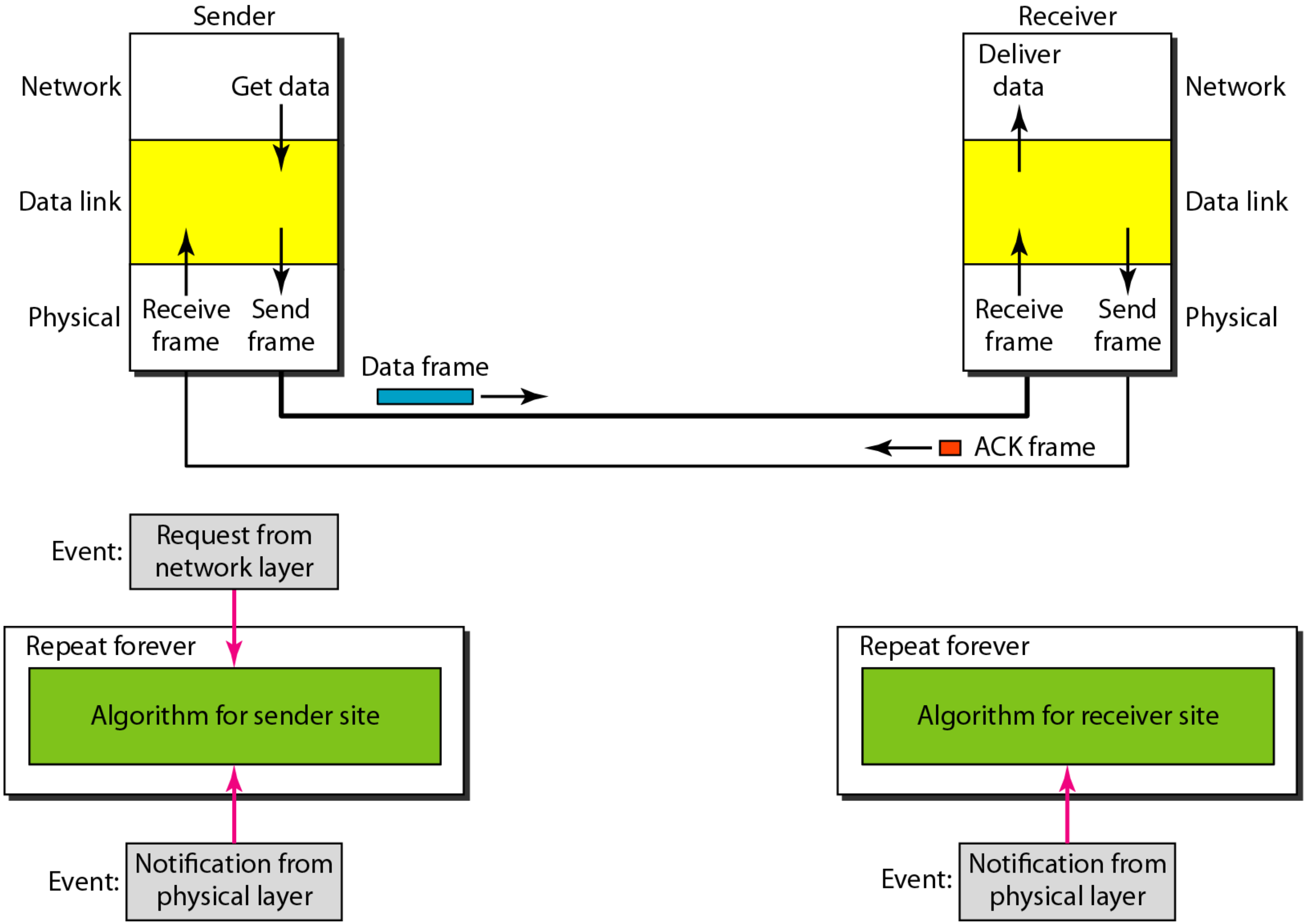


Figure 11.8 *Design of Stop-and-Wait Protocol*



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

```
while(true)                          //Repeat forever
canSend = true                       //Allow the first frame to go
{
 WaitForEvent();                     // Sleep until an event occurs
  if(Event(RequestToSend) AND canSend)
  {
     GetData();
     MakeFrame();
     SendFrame();                    //Send the data frame
     canSend = false;                //Cannot send until ACK arrives
  }
  WaitForEvent();                    // Sleep until an event occurs
  if(Event(ArrivalNotification)      // An ACK has arrived
   {
     ReceiveFrame();                 //Receive the ACK frame
     canSend = true;
   }
}
```

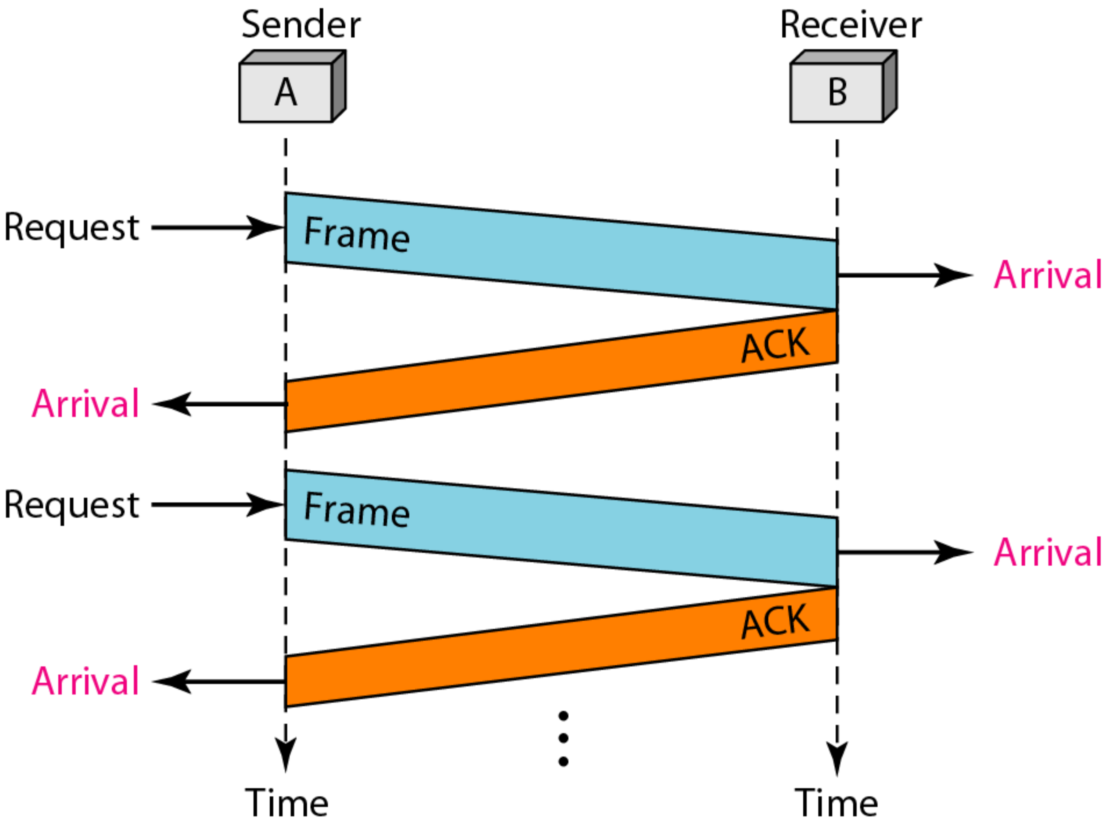
Algorithm 11.4 *Receiver-site algorithm for Stop-and-Wait Protocol*

```
while(true)                               //Repeat forever
{
  WaitForEvent();                         // Sleep until an event occurs
  if(Event(ArrivalNotification))          //Data frame arrives
  {
     ReceiveFrame();
     ExtractData();
     Deliver(data);                       //Deliver data to network layer
     SendFrame();                         //Send an ACK frame
  }
}
```

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.

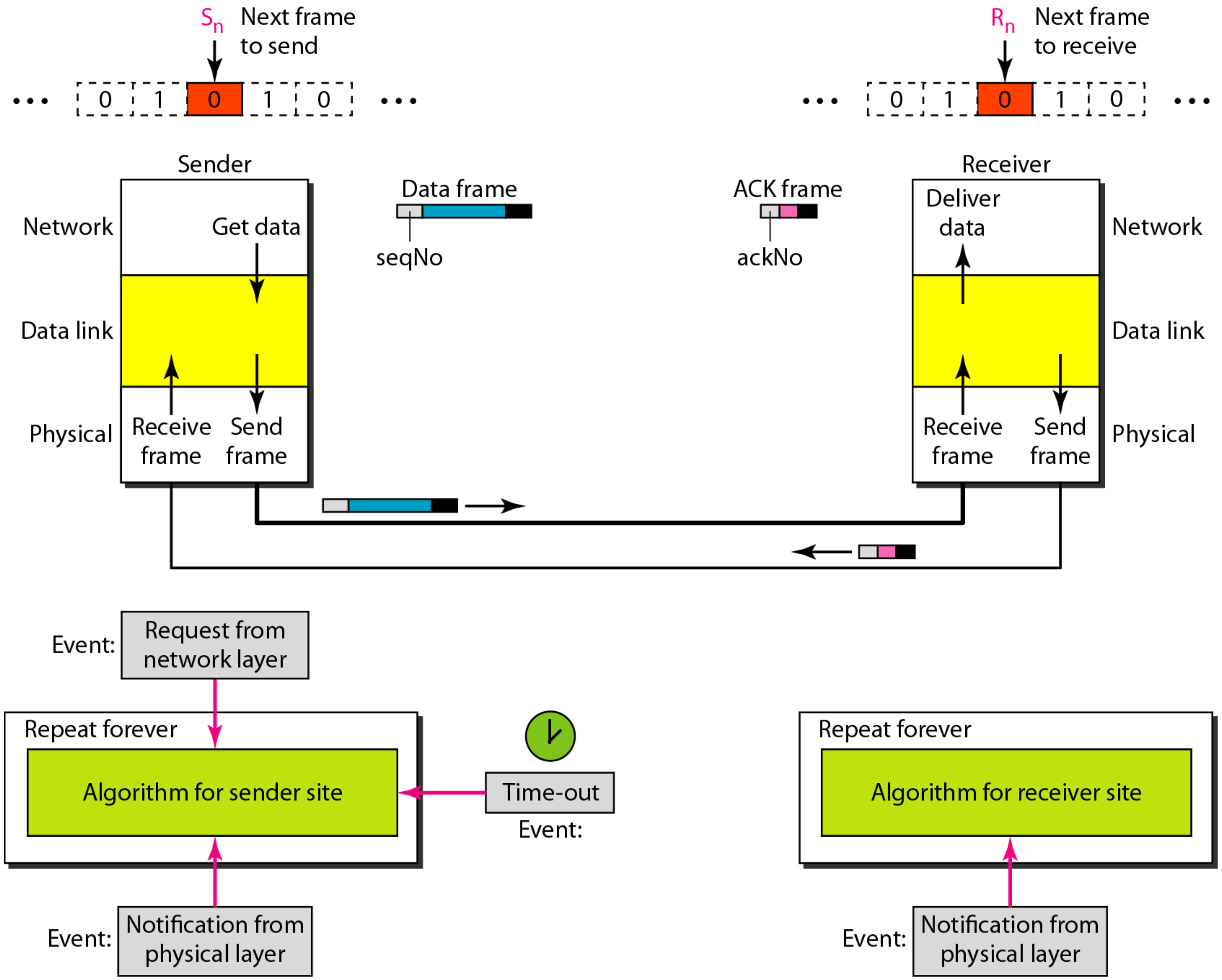
Note

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*

```
$S_n$ = 0;                           // Frame 0 should be sent first
canSend = true;                      // Allow the first request to go
while(true)                          // Repeat forever
{
  WaitForEvent();                    // Sleep until an event occurs
  if(Event(RequestToSend) AND canSend)
  {
     GetData();
     MakeFrame($S_n$);                        //The seqNo is $S_n$
     StoreFrame($S_n$);                       //Keep copy
     SendFrame($S_n$);
     StartTimer();
     $S_n$ = $S_n$ + 1;
     canSend = false;
  }
  WaitForEvent();                             // Sleep
```

(continued)

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

```
    if(Event(ArrivalNotification)     // An ACK has arrived
    {
      ReceiveFrame(ackNo);            //Receive the ACK frame
      if(not corrupted AND ackNo == $S_n$) //Valid ACK
        {
          Stoptimer();
          PurgeFrame($S_{n-1}$);            //Copy is not needed
          canSend = true;
        }
     }

    if(Event(TimeOut)                 // The timer expired
    {
     StartTimer();
     ResendFrame($S_{n-1}$);                //Resend a copy check
    }
}
```

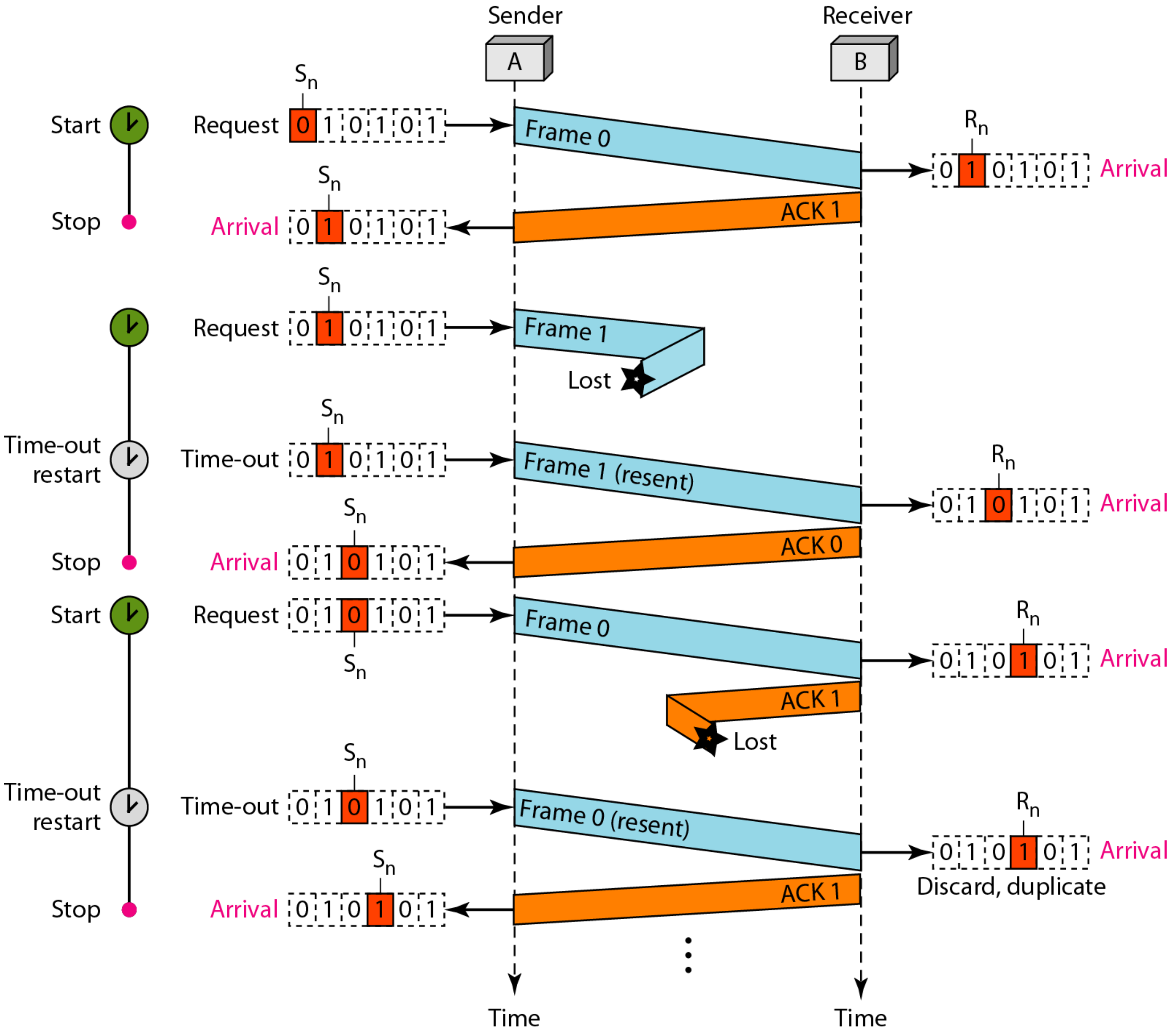
Algorithm 11.6 *Receiver-site algorithm for Stop-and-Wait ARQ Protocol*

```
$R_n$ = 0;                   // Frame 0 expected to arrive first
while(true)
{
  WaitForEvent();            // Sleep until an event occurs
  if(Event(ArrivalNotification))  //Data frame arrives
  {
     ReceiveFrame();
     if(corrupted(frame));
        sleep();
     if(seqNo == $R_n$)               //Valid data frame
     {
      ExtractData();
       DeliverData();                //Deliver data
       $R_n$ = $R_n$ + 1;
     }
      SendFrame($R_n$);               //Send an ACK
  }
}
```

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

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

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?

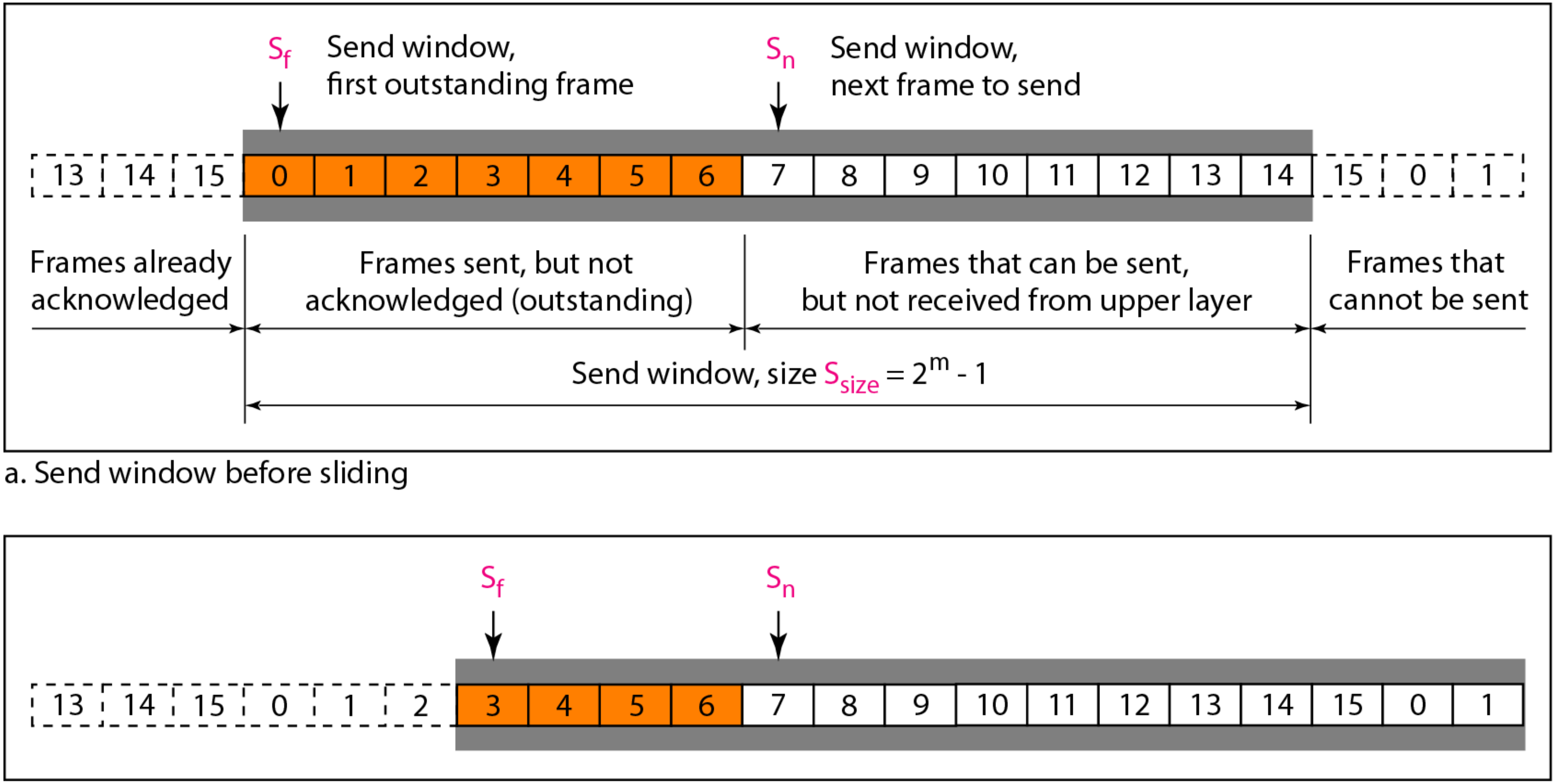
Solution

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*


S_f
Send window, first outstanding frame
S_n
Send window, next frame to send
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1
Frames already acknowledged
Frames sent, but not acknowledged (outstanding)
Frames that can be sent, but not received from upper layer
Frames that cannot be sent
Send window, size $S_{size} = 2^m - 1$
S_f
S_n
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1

a. Send window before sliding

b. Send window after sliding

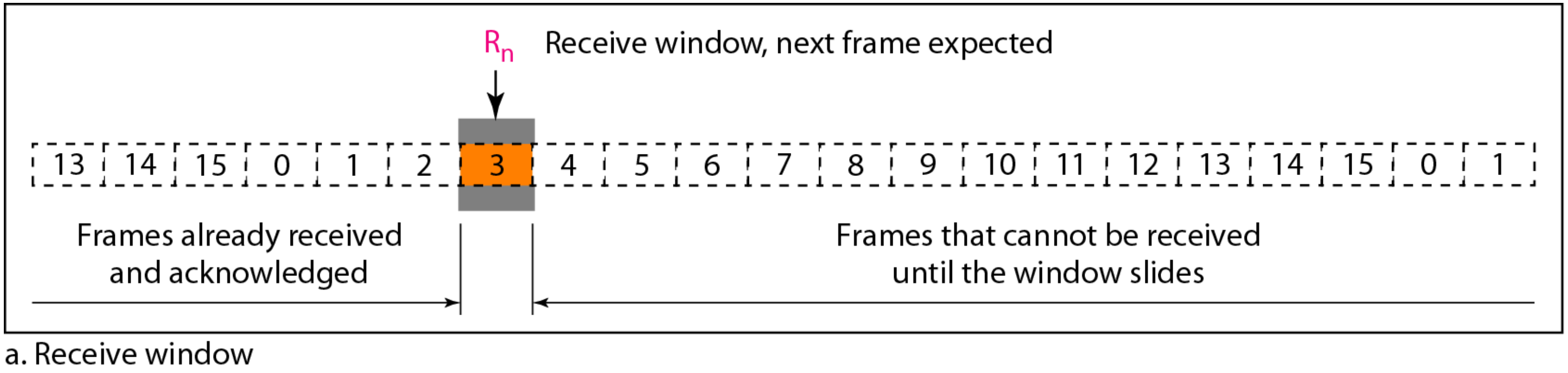
Note

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}.

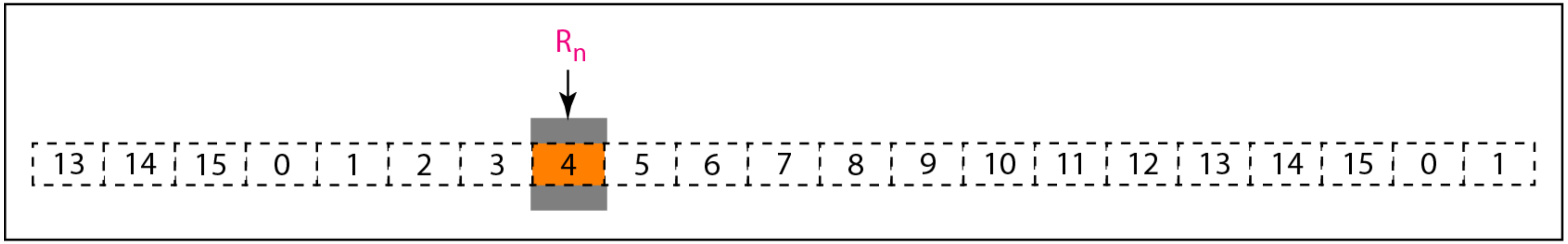
Note

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

Figure 11.13 *Receive window for Go-Back-N ARQ*


R_n
Receive window, next frame expected
13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1
Frames already received and acknowledged
Frames that cannot be received until the window slides

a. Receive window



b. Window after sliding

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*

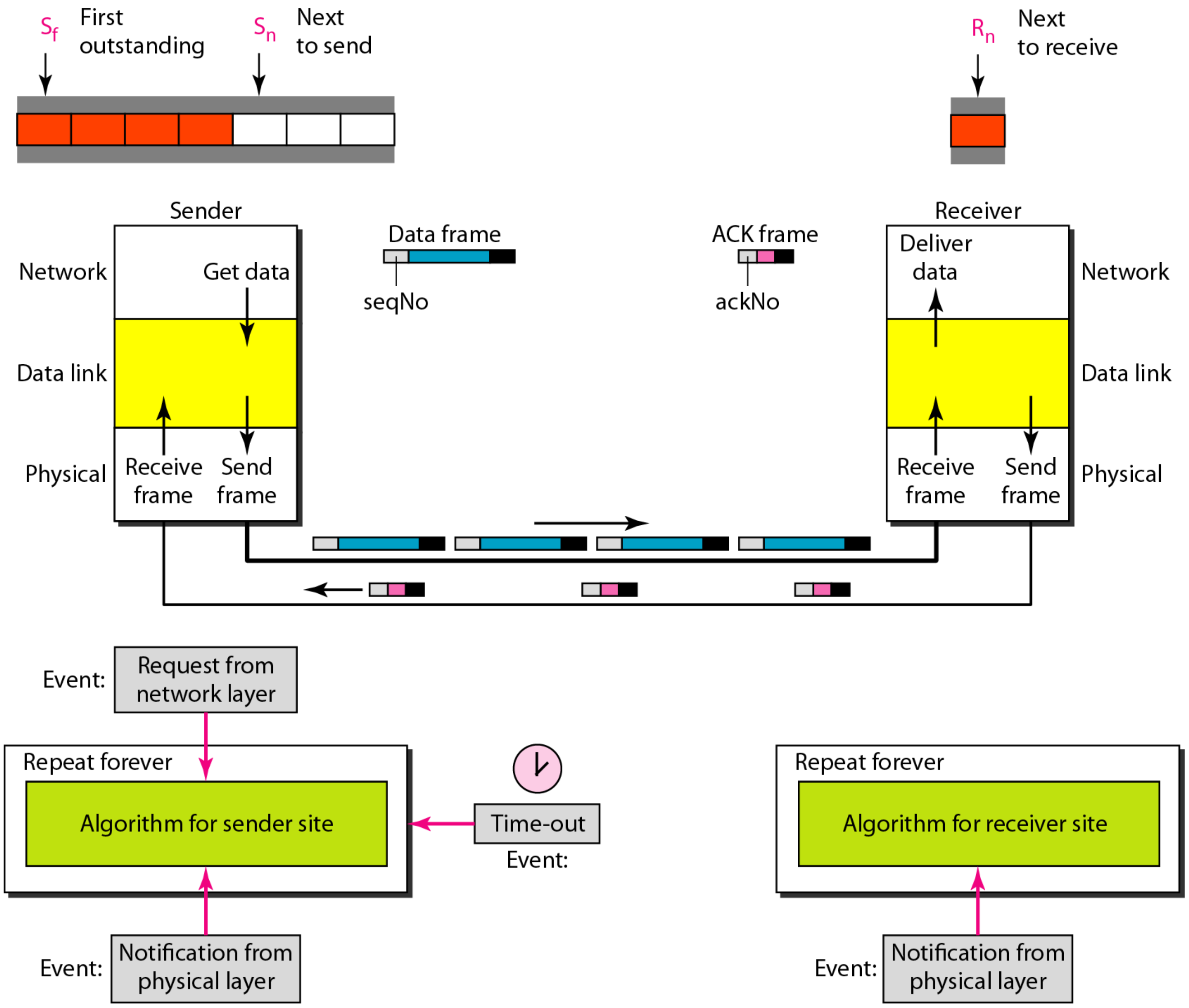
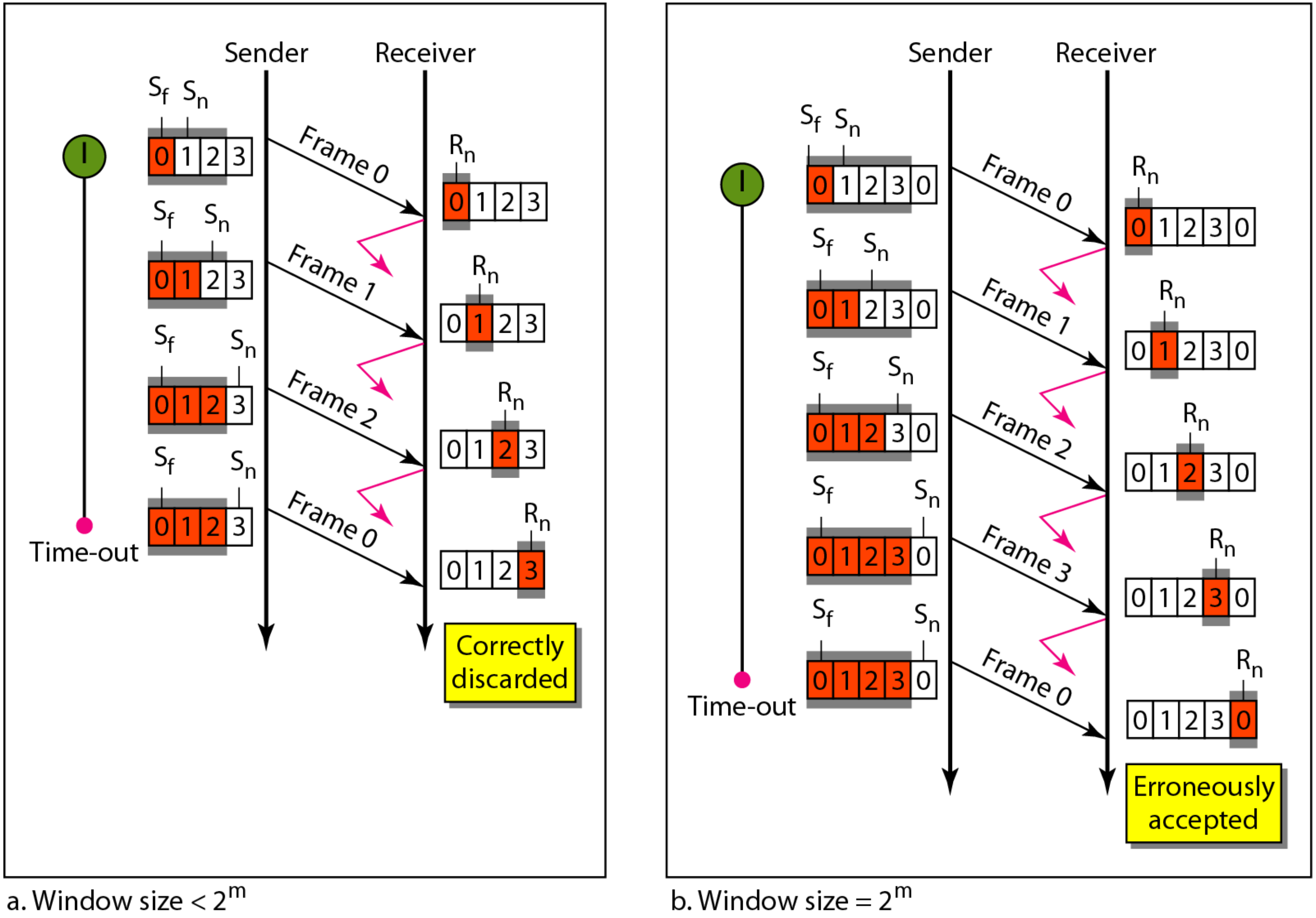


Figure 11.15 *Window size for Go-Back-N ARQ*



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*

```
$S_w = 2^m - 1$;
$S_f = 0$;
$S_n = 0$;

while (true)                                    //Repeat forever
{
 WaitForEvent();
   if(Event(RequestToSend))                     //A packet to send
   {
      if($S_n$-$S_f$ >= $S_w$)                  //If window is full
            Sleep();
      GetData();
      MakeFrame($S_n$);
      StoreFrame($S_n$);
      SendFrame($S_n$);
      $S_n = S_n + 1$;
      if(timer not running)
           StartTimer();
   }

```

(continued)

Algorithm 11.7 *Go-Back-N sender algorithm* *(continued)*

```
  if(Event(ArrivalNotification))  //ACK arrives
  {
     Receive(ACK);
     if(corrupted(ACK))
          Sleep();
     if((ackNo>$S_f$)&&(ackNo<=$S_n$))  //If a valid ACK
     While($S_f$ <= ackNo)
      {
       PurgeFrame($S_f$);
       $S_f$ = $S_f$ + 1;
      }
      StopTimer();
  }

  if(Event(TimeOut))              //The timer expires
  {
   StartTimer();
   Temp = $S_f$;
   while(Temp < $S_n$);
    {
     SendFrame($S_f$);
     $S_f$ = $S_f$ + 1;
    }
  }
}
```

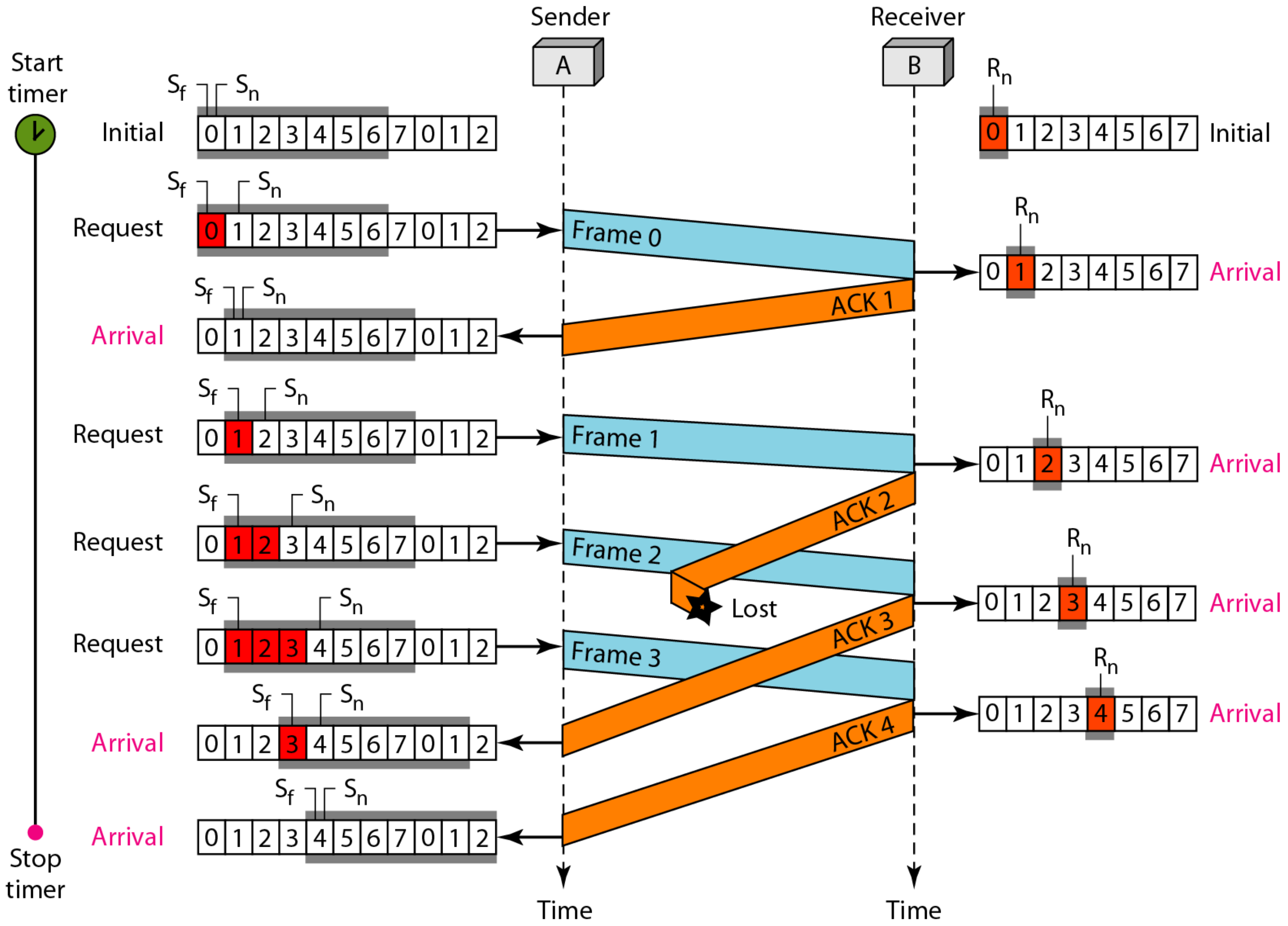
Algorithm 11.8 *Go-Back-N receiver algorithm*

```
$R_n$ = 0;

while (true)                      //Repeat forever
{
  WaitForEvent();

  if(Event(ArrivalNotification)) /Data frame arrives
  {
     Receive(Frame);
     if(corrupted(Frame))
          Sleep();
     if(seqNo == $R_n$)                //If expected frame
     {
       DeliverData();             //Deliver data
       $R_n$ = $R_n$ + 1;                //Slide window
       SendACK($R_n$);
     }
  }
}
```

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*



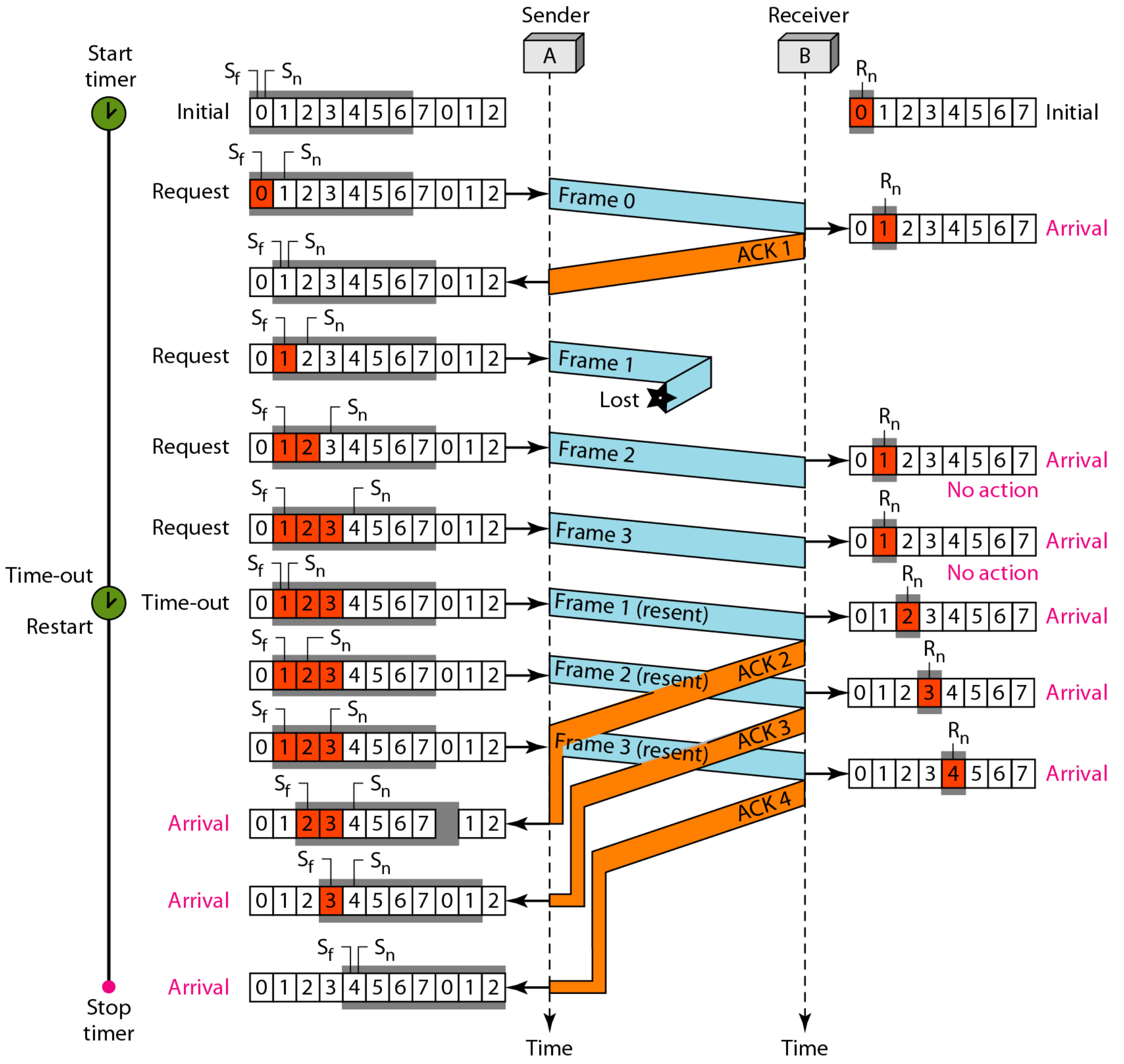
Example 11.7

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*

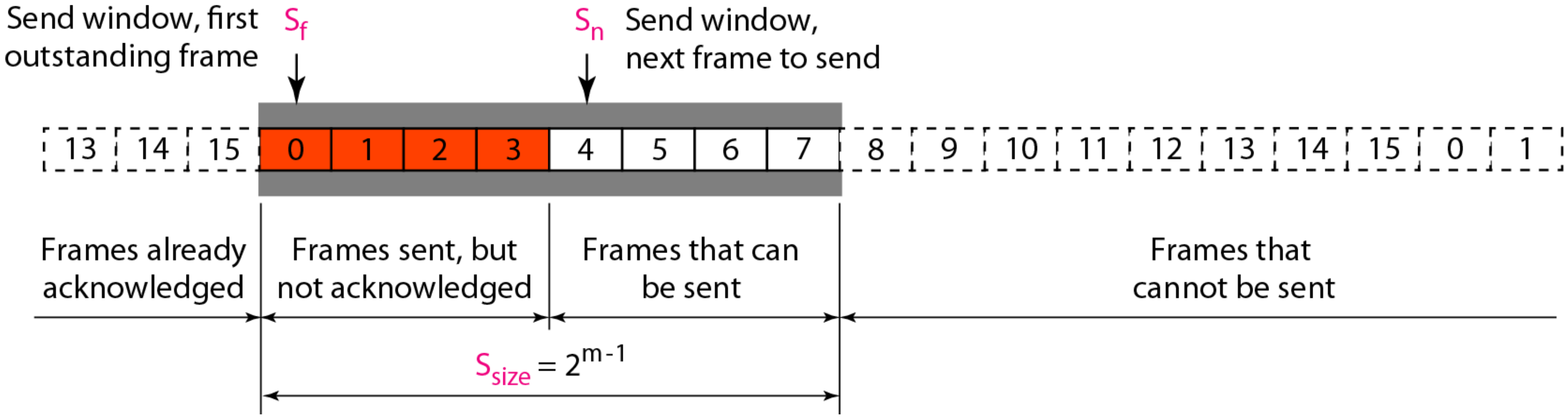

Send window, first outstanding frame
S_f
S_n
Send window, next frame to send
13
14
15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
Frames already acknowledged
Frames sent, but not acknowledged
Frames that can be sent
Frames that cannot be sent
$S_{size} = 2^{m-1}$

Figure 11.19 *Receive window for Selective Repeat ARQ*

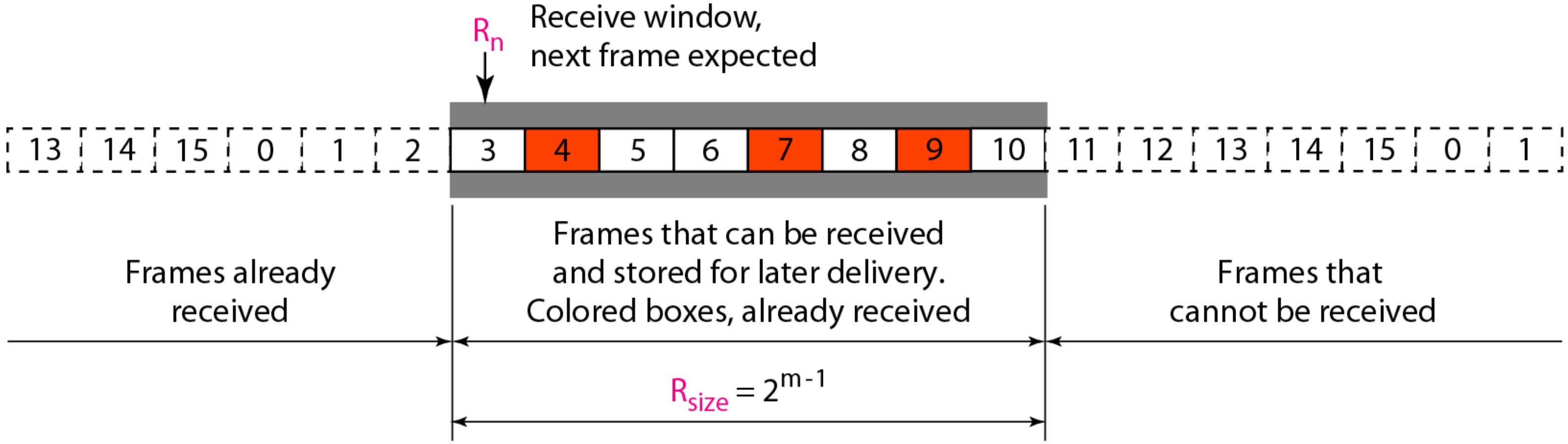

R_n
Receive window,
next frame expected
13
14
15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0
1
Frames already received
Frames that can be received and stored for later delivery. Colored boxes, already received
Frames that cannot be received
$R_{size} = 2^{m-1}$

Figure 11.20 *Design of Selective Repeat ARQ*

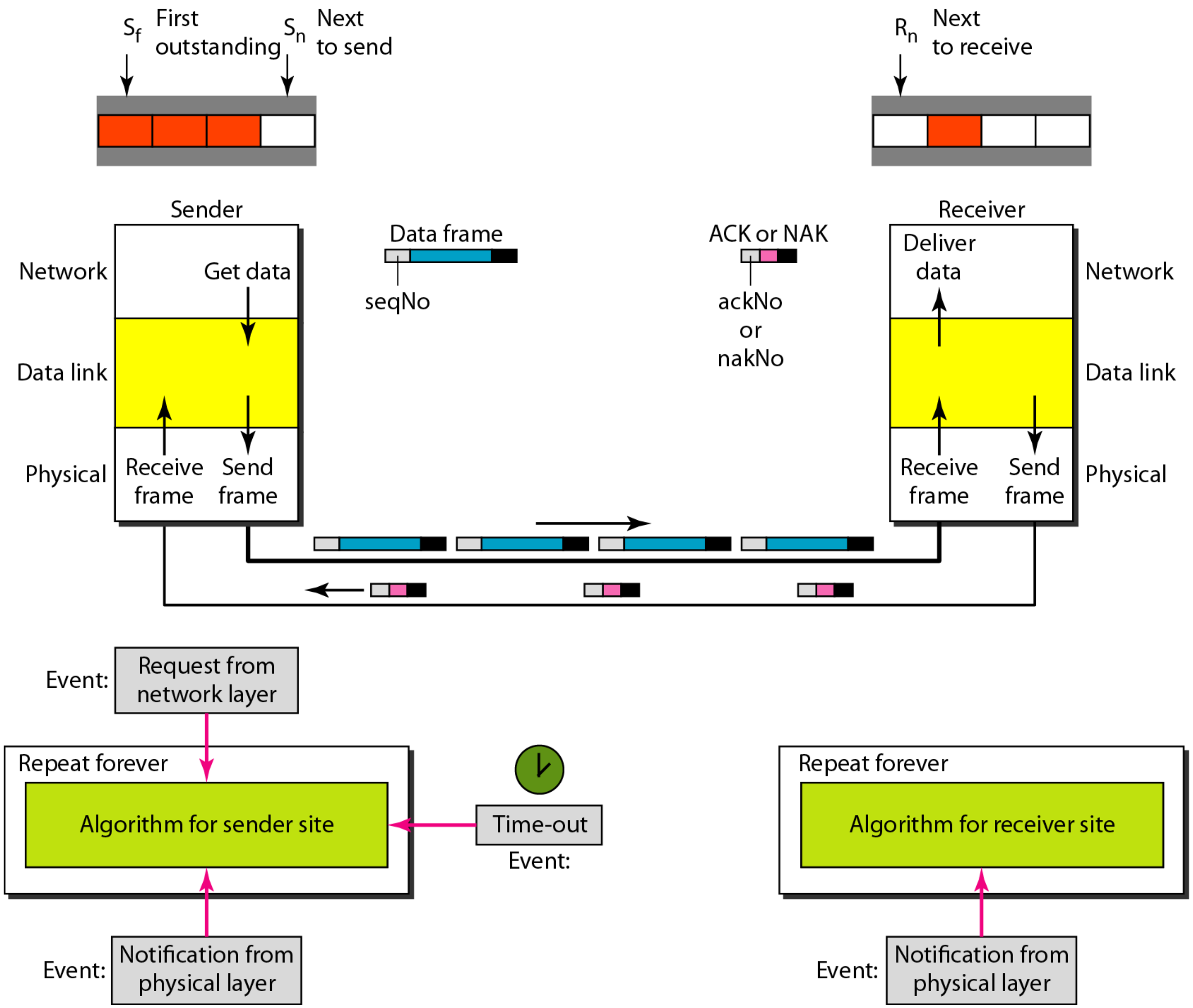
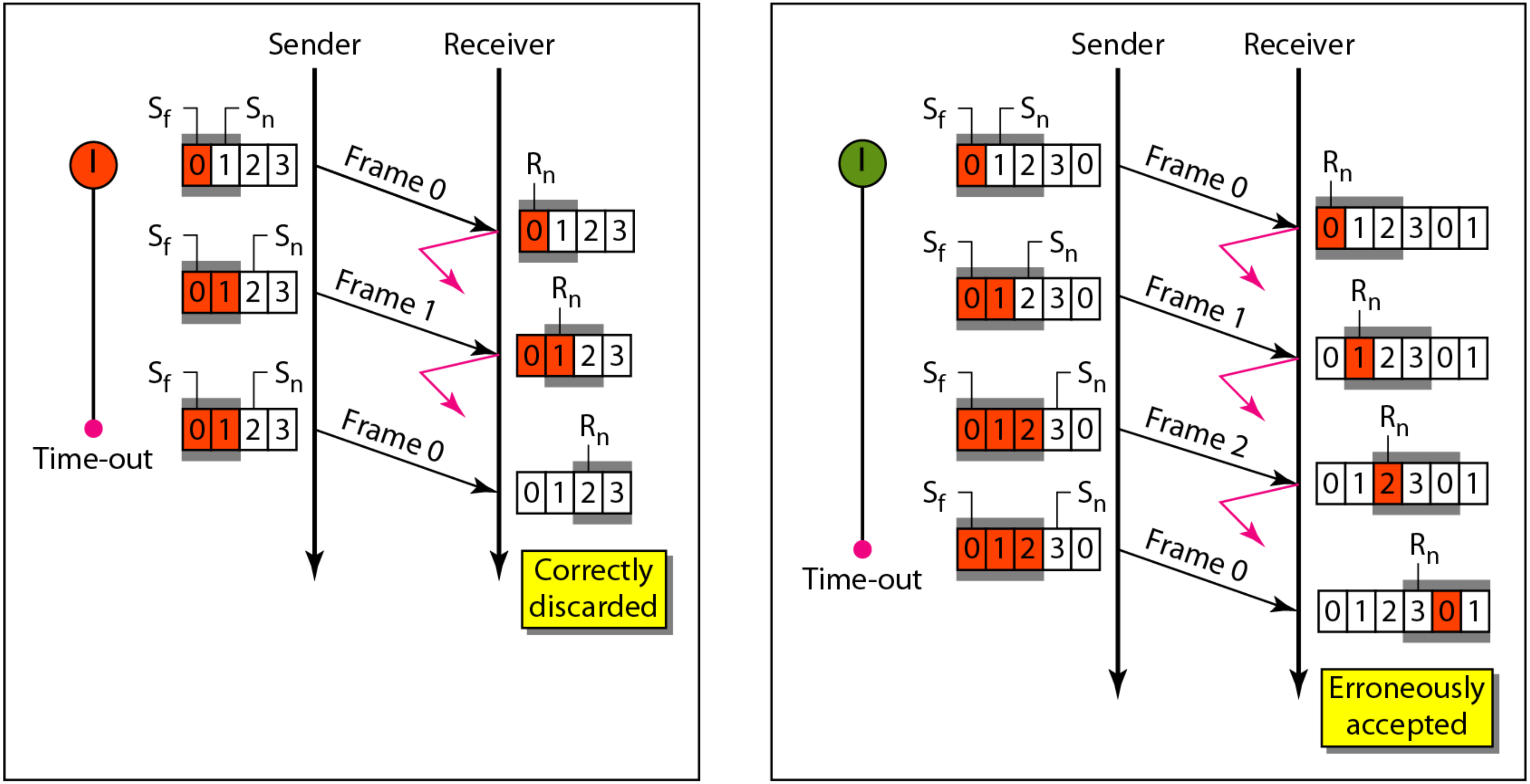


Figure 11.21 *Selective Repeat ARQ, window size*



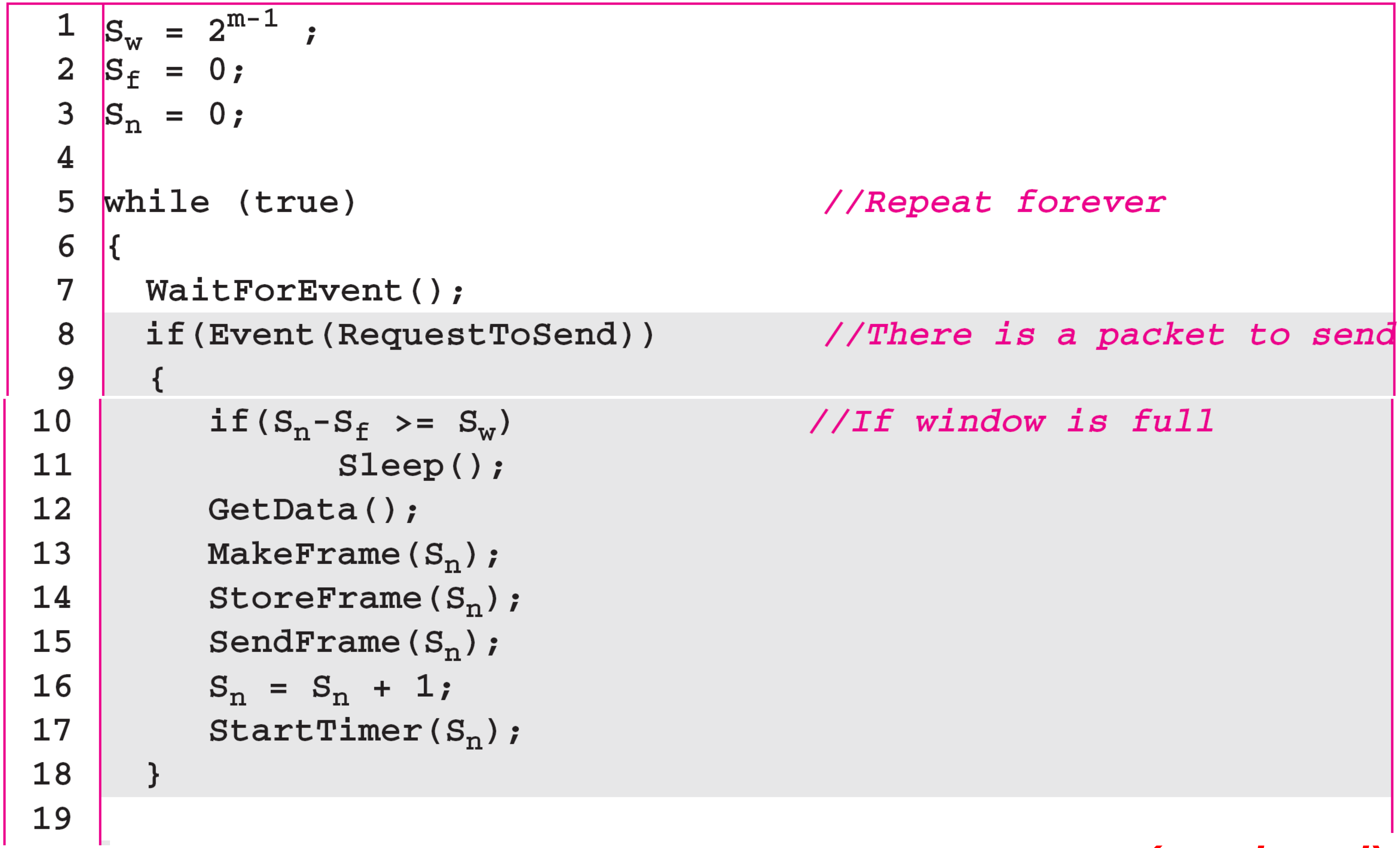
a. Window size = 2^{m-1}

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*



```
$S_w = 2^{m-1}$ ;
$S_f = 0$;
$S_n = 0$;

while (true)                                 //Repeat forever
{
  WaitForEvent();
  if(Event(RequestToSend))                   //There is a packet to send
  {
     if($S_n-S_f >= S_w$)                    //If window is full
            Sleep();
     GetData();
     MakeFrame($S_n$);
     StoreFrame($S_n$);
     SendFrame($S_n$);
     $S_n = S_n + 1$;
     StartTimer($S_n$);
  }

```

(continued)

Algorithm 11.9 *Sender-site Selective Repeat algorithm* *(continued)*

```
if(Event(ArrivalNotification)) //ACK arrives
{
   Receive(frame);             //Receive ACK or NAK
   if(corrupted(frame))
        Sleep();
   if (FrameType == NAK)
      if (nakNo between $S_f$ and $S_n$)
      {
       resend(nakNo);
       StartTimer(nakNo);
      }
   if (FrameType == ACK)
      if (ackNo between $S_f$ and $S_n$)
      {
        while($s_f$ < ackNo)
        {
         Purge($s_f$);
         StopTimer($s_f$);
         $S_f$ = $S_f$ + 1;
        }
      }
}
```

(continued)

Algorithm 11.9 *Sender-site Selective Repeat algorithm* *(continued)*

```

  if(Event(TimeOut(t)))               //The timer expires
  {
   StartTimer(t);
   SendFrame(t);
  }
}
```

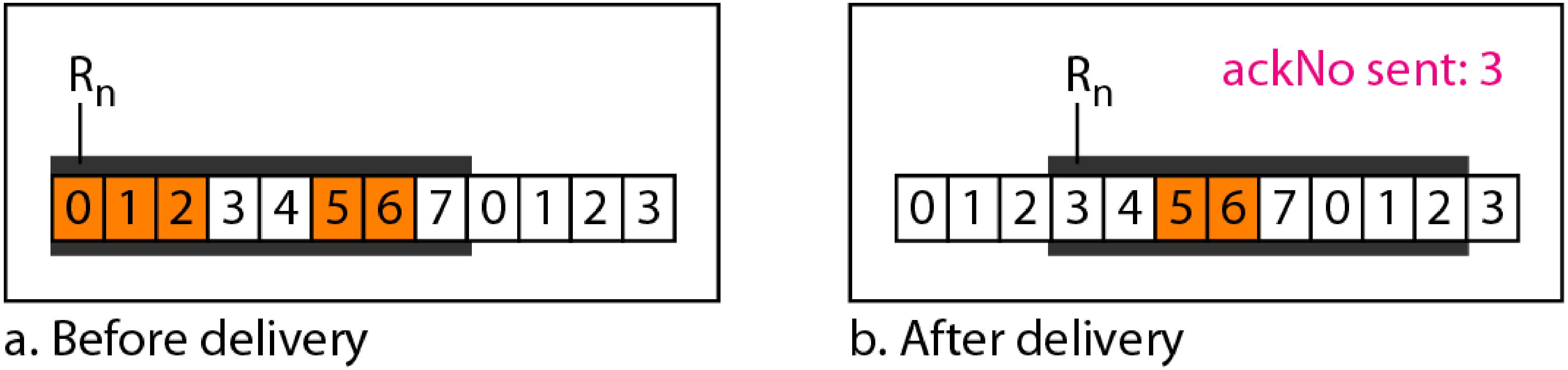
Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

```
$R_n$ = 0;
NakSent = false;
AckNeeded = false;
Repeat(for all slots)
    Marked(slot) = false;

while (true)                                     //Repeat forever
{
  WaitForEvent();

  if(Event(ArrivalNotification))                 /Data frame arrives
  {
     Receive(Frame);
     if(corrupted(Frame))&& (NOT NakSent)
     {
      SendNAK($R_n$);
      NakSent = true;
      Sleep();
     }
     if(seqNo <> $R_n$)&& (NOT NakSent)
     {
      SendNAK($R_n$);
```

Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

```
      NakSent = true;
      if ((seqNo in window)&&(!Marked(seqNo))
      {
       StoreFrame(seqNo)
       Marked(seqNo)= true;
       while(Marked($R_n$))
       {
        DeliverData($R_n$);
        Purge($R_n$);
        $R_n$ = $R_n$ + 1;
        AckNeeded = true;
       }
        if(AckNeeded);
        {
        SendAck($R_n$);
        AckNeeded = false;
        NakSent = false;
        }
      }
     }
  }
}
```

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*

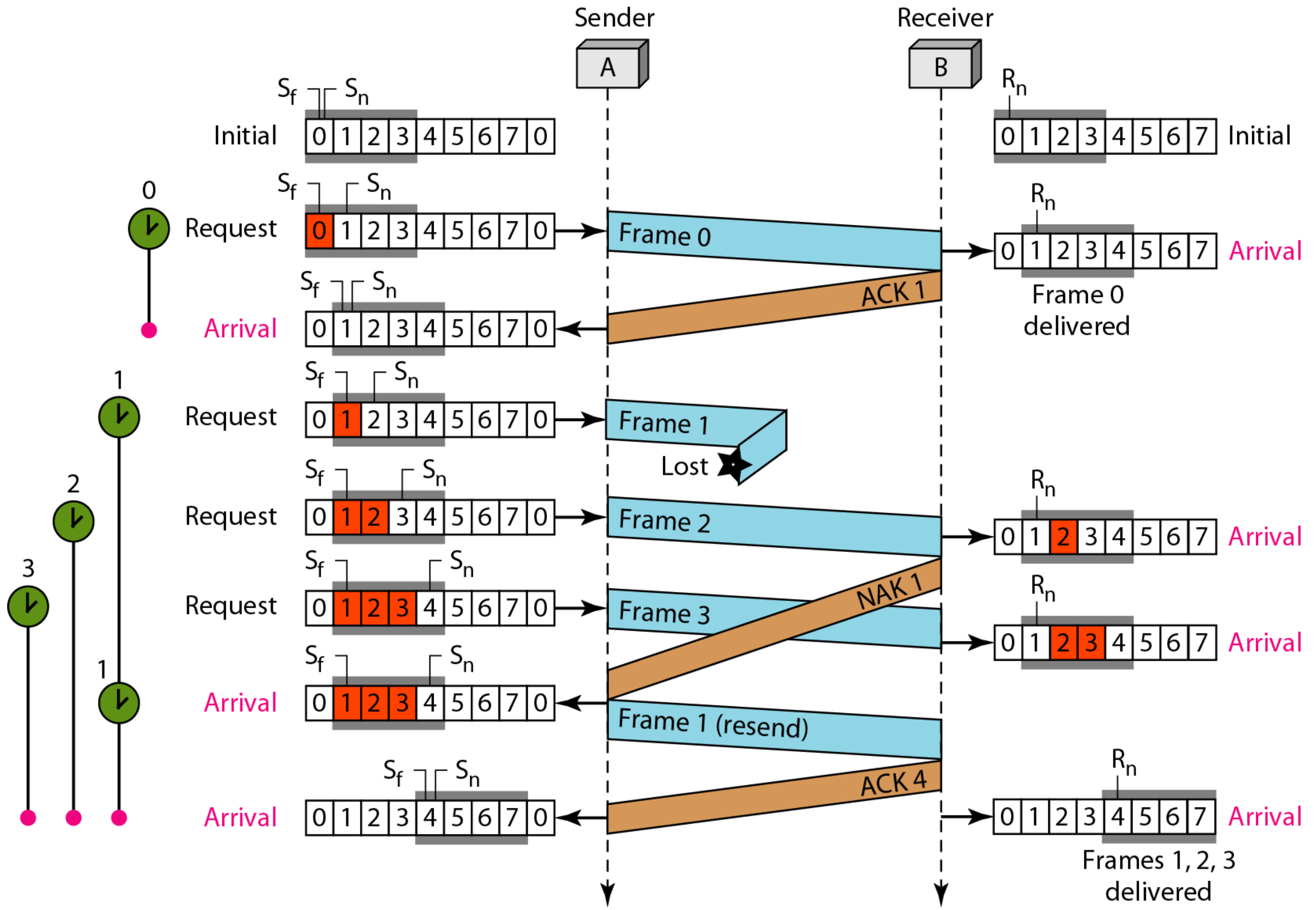
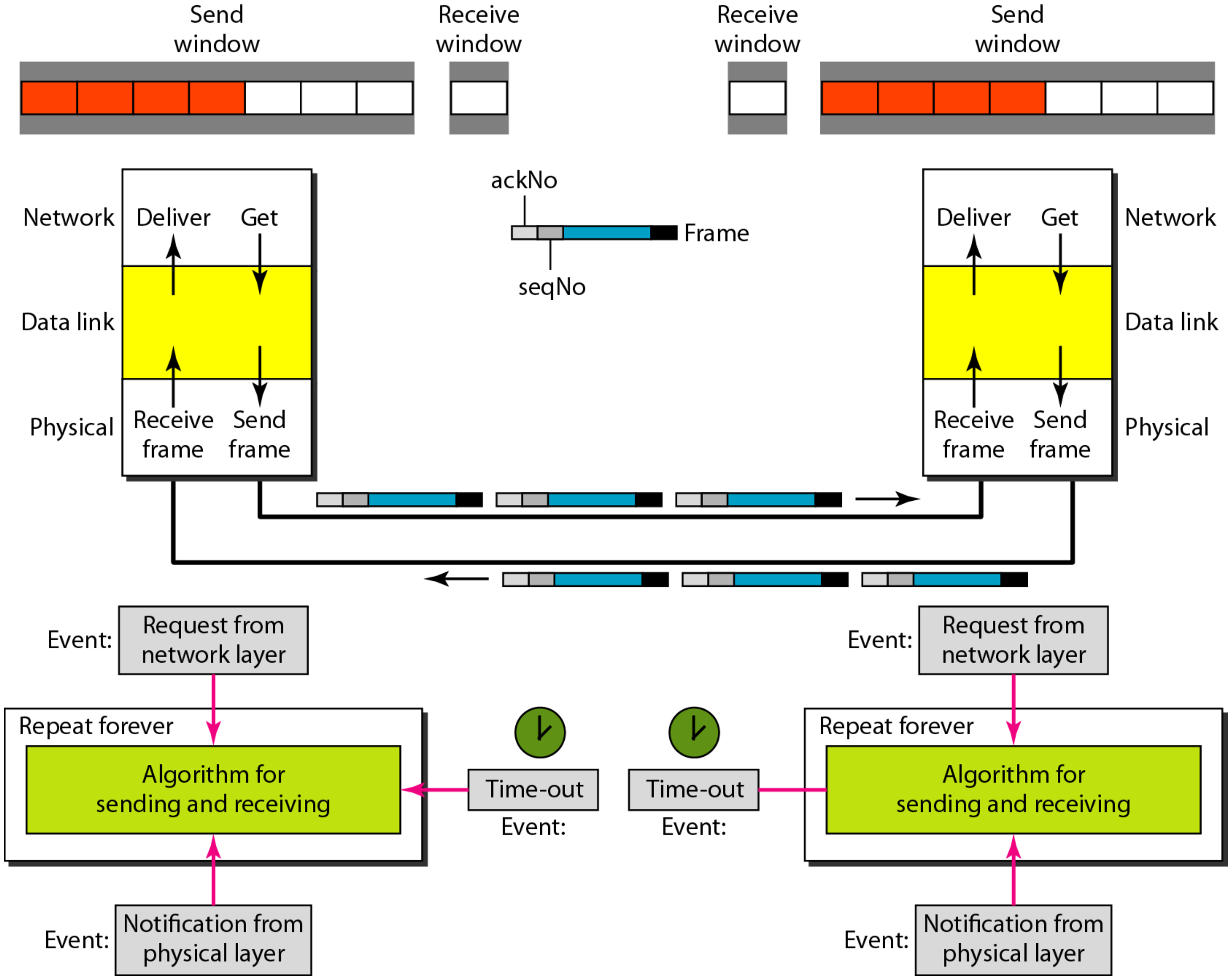


Figure 11.24 *Design of piggybacking in Go-Back-N ARQ*



11-6 HDLC

***High-level Data Link Control (HDLC)* is a *bit-oriented* protocol for communication over point-to-point and multipoint links. It implements the ARQ mechanisms we discussed in this chapter.**

Topics discussed in this section:

Configurations and Transfer Modes
Frames
Control Field

Figure 11.25 *Normal response mode*

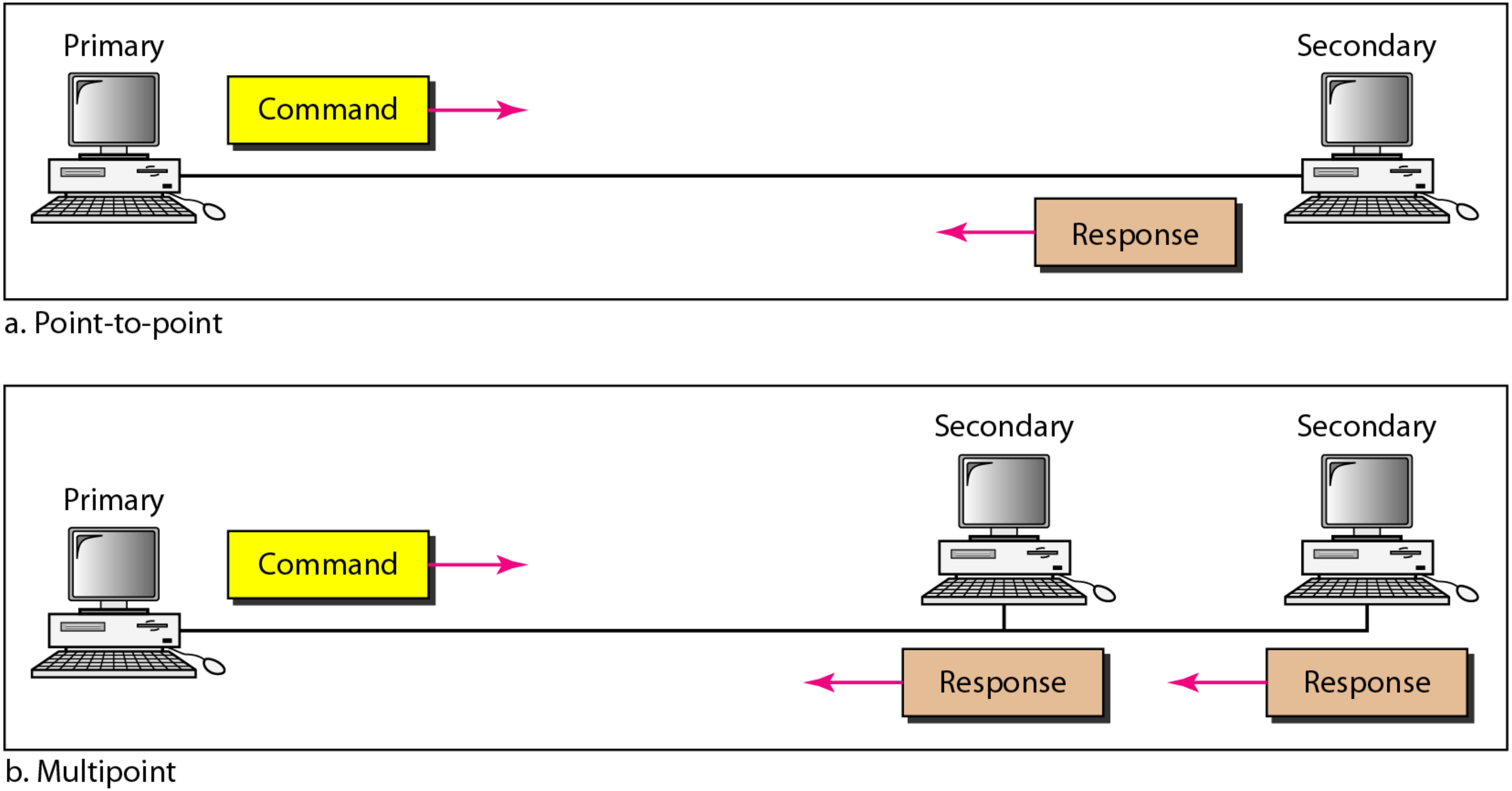


Figure 11.26 *Asynchronous balanced mode*

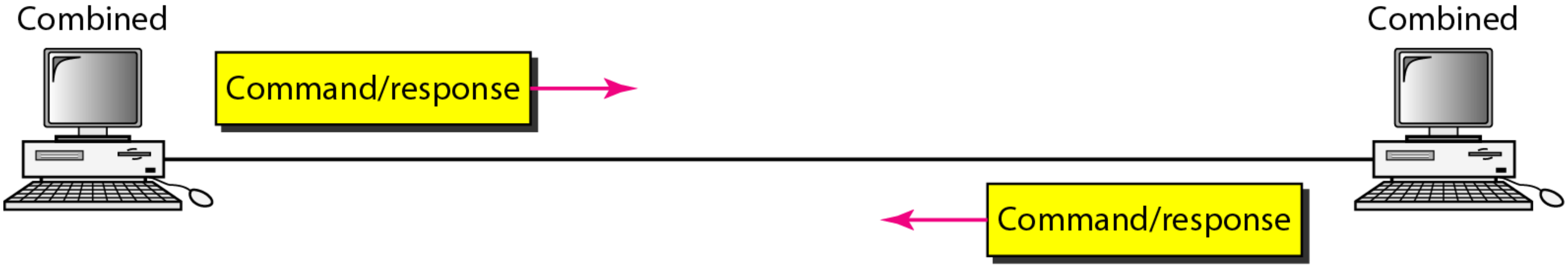


Figure 11.27 ***HDLC frames***

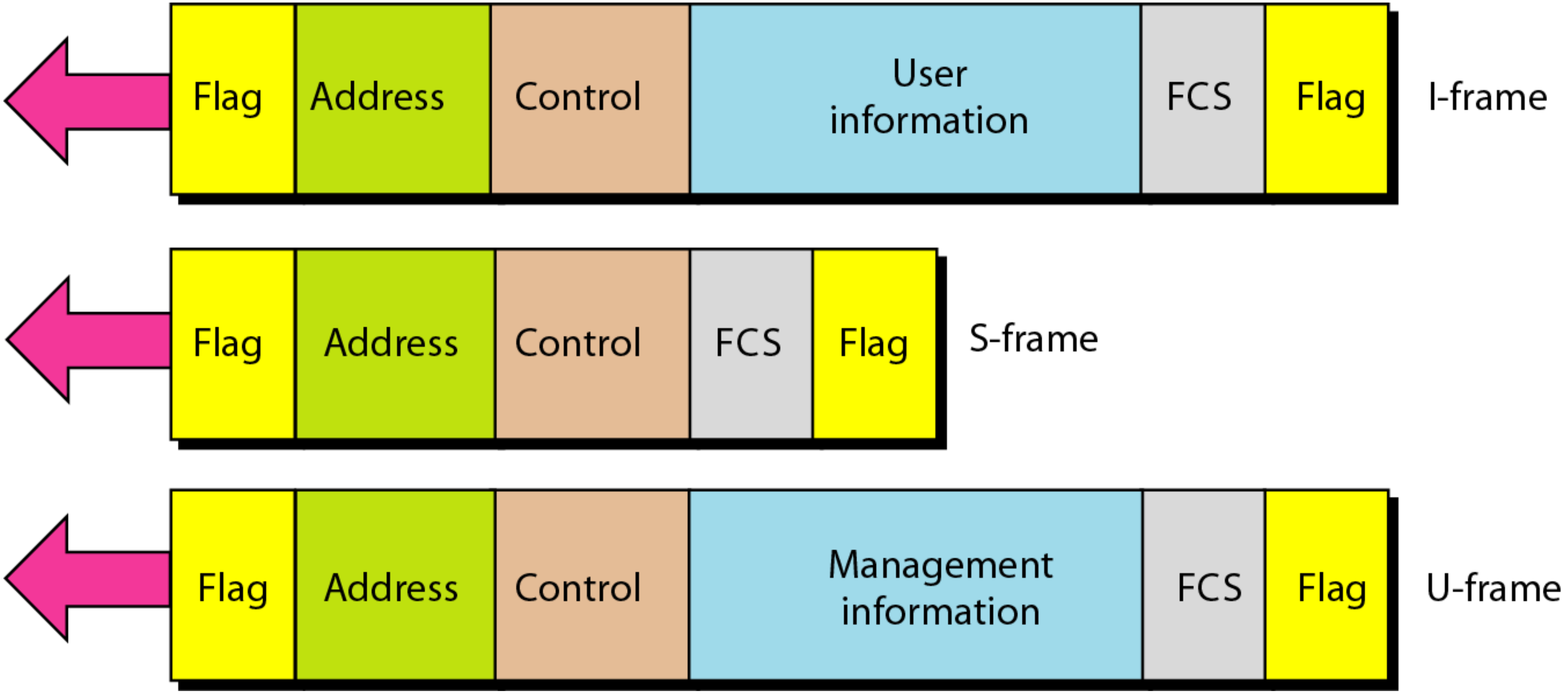


Figure 11.28 *Control field format for the different frame types*

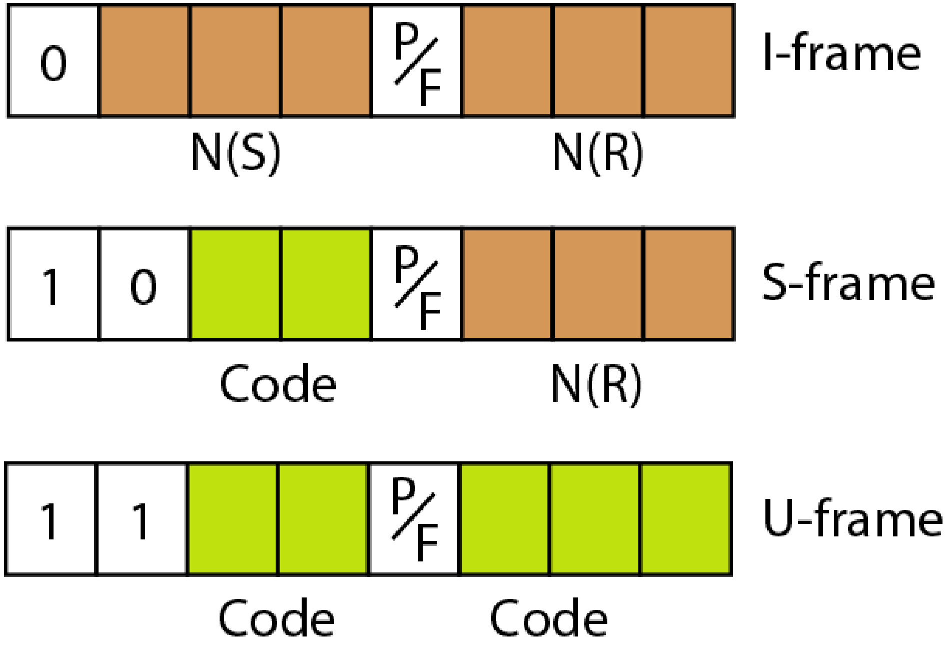


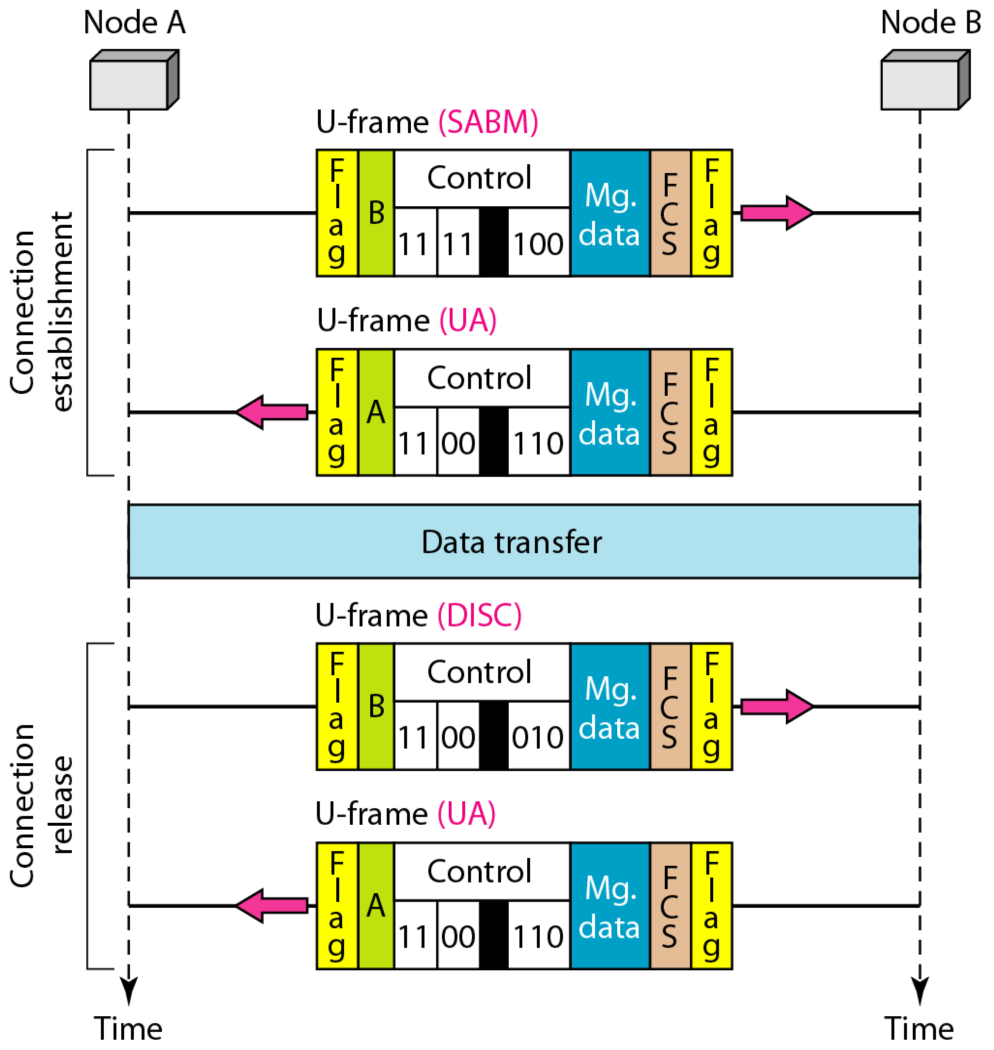
Table 11.1 *U-frame control command and response*

Code	*Command*	*Response*	*Meaning*
00 001	SNRM		Set normal response mode
11 011	SNRME		Set normal response mode, extended
11 100	SABM	**DM**	Set asynchronous balanced mode or **disconnect mode**
11 110	SABME		Set asynchronous balanced mode, extended
00 000	UI	**UI**	Unnumbered information
00 110		**UA**	**Unnumbered acknowledgment**
00 010	DISC	**RD**	Disconnect or **request disconnect**
10 000	SIM	**RIM**	Set initialization mode or **request information mode**
00 100	UP		Unnumbered poll
11 001	RSET		Reset
11 101	XID	**XID**	Exchange ID
10 001	FRMR	**FRMR**	Frame reject

Example 11.9

Figure 11.29 shows how U-frames can be used for connection establishment and connection release. Node A asks for a connection with a set asynchronous balanced mode (SABM) frame; node B gives a positive response with an unnumbered acknowledgment (UA) frame. After these two exchanges, data can be transferred between the two nodes (not shown in the figure). After data transfer, node A sends a DISC (disconnect) frame to release the connection; it is confirmed by node B responding with a UA (unnumbered acknowledgment).

Figure 11.29 *Example of connection and disconnection*



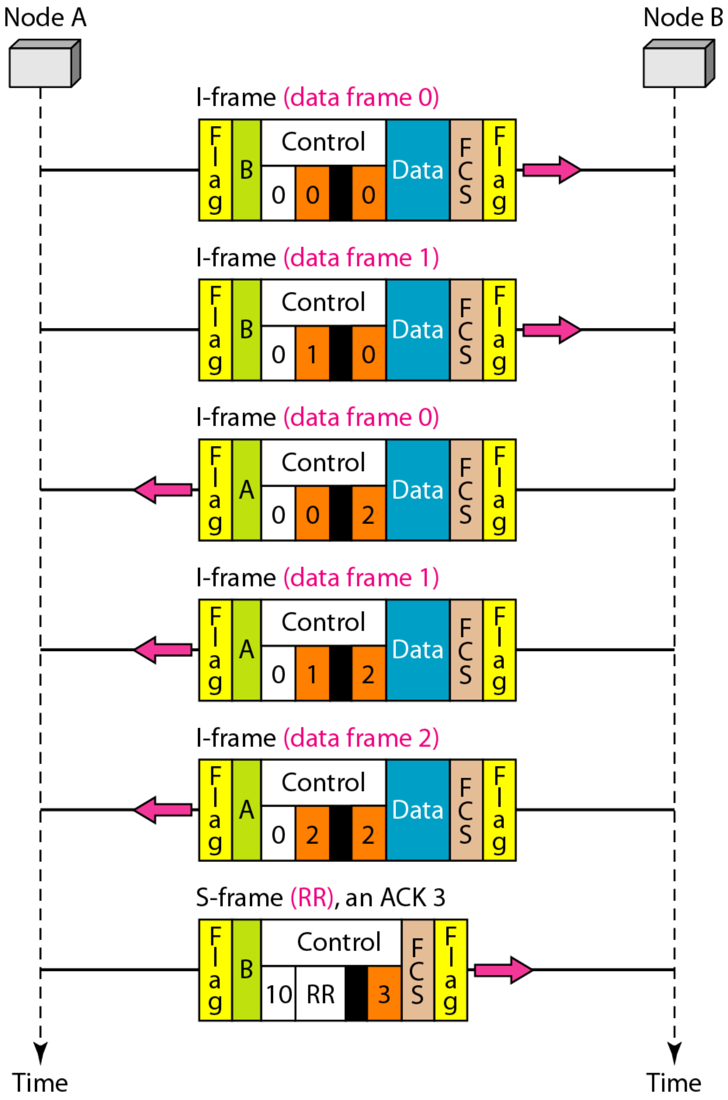
Example 11.10

Figure 11.30 shows an exchange using piggybacking. Node A begins the exchange of information with an I-frame numbered 0 followed by another I-frame numbered 1. Node B piggybacks its acknowledgment of both frames onto an I-frame of its own. Node B's first I-frame is also numbered 0 [N(S) field] and contains a 2 in its N(R) field, acknowledging the receipt of A's frames 1 and 0 and indicating that it expects frame 2 to arrive next. Node B transmits its second and third I-frames (numbered 1 and 2) before accepting further frames from node A.

Example 11.10 (continued)

Its N(R) information, therefore, has not changed: B frames 1 and 2 indicate that node B is still expecting A's frame 2 to arrive next. Node A has sent all its data. Therefore, it cannot piggyback an acknowledgment onto an I-frame and sends an S-frame instead. The RR code indicates that A is still ready to receive. The number 3 in the N(R) field tells B that frames 0, 1, and 2 have all been accepted and that A is now expecting frame number 3.

Figure 11.30 *Example of piggybacking without error*

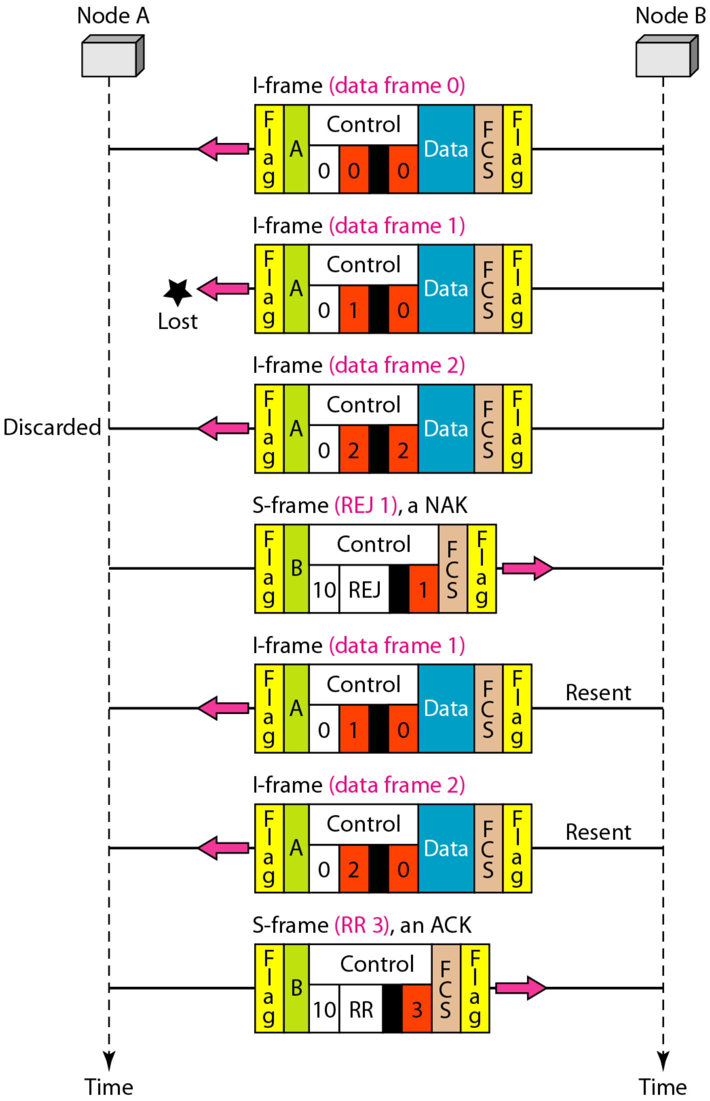


Node A
Node B
I-frame (data frame 0)
Flag B Control 0 0 0 Data FCS Flag
I-frame (data frame 1)
Flag B Control 0 1 0 Data FCS Flag
I-frame (data frame 0)
Flag A Control 0 0 2 Data FCS Flag
I-frame (data frame 1)
Flag A Control 0 1 2 Data FCS Flag
I-frame (data frame 2)
Flag A Control 0 2 2 Data FCS Flag
S-frame (RR), an ACK 3
Flag B Control 10 RR 3 FCS Flag
Time
Time

Example 11.11

Figure 11.31 shows an exchange in which a frame is lost. Node B sends three data frames (0, 1, and 2), but frame 1 is lost. When node A receives frame 2, it discards it and sends a REJ frame for frame 1. Note that the protocol being used is Go-Back-N with the special use of an REJ frame as a NAK frame. The NAK frame does two things here: It confirms the receipt of frame 0 and declares that frame 1 and any following frames must be resent. Node B, after receiving the REJ frame, resends frames 1 and 2. Node A acknowledges the receipt by sending an RR frame (ACK) with acknowledgment number 3.

Figure 11.31 *Example of piggybacking with error*



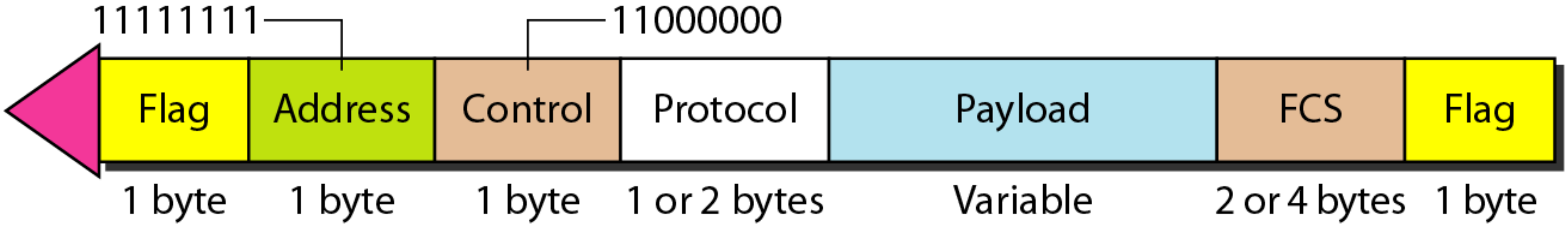
11-7 POINT-TO-POINT PROTOCOL

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*

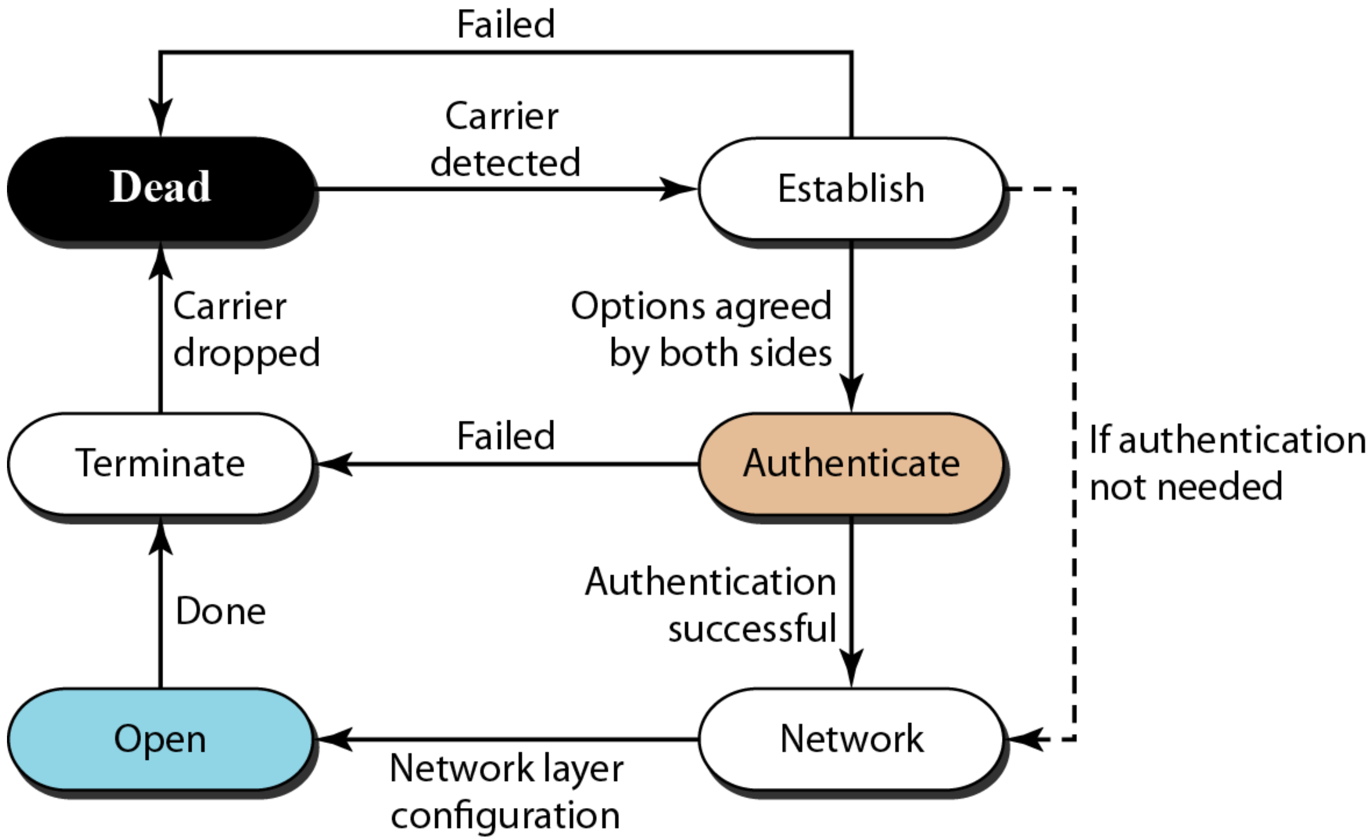
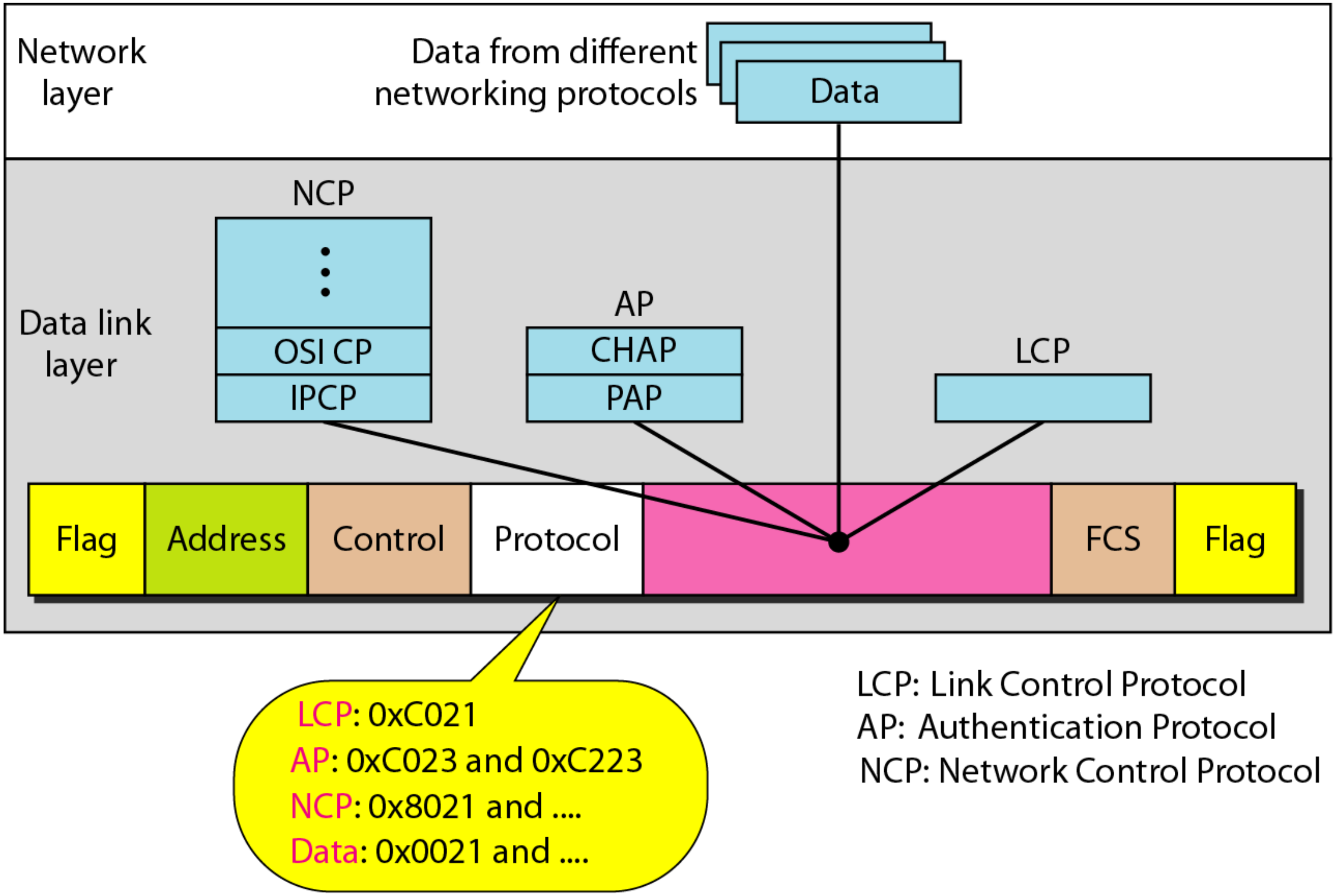

Failed
Carrier detected
Dead
Establish
Carrier dropped
Options agreed by both sides
Terminate
Failed
Authenticate
If authentication not needed
Done
Authentication successful
Open
Network
Network layer configuration

Figure 11.34 *Multiplexing in PPP*



LCP: 0xC021
AP: 0xC023 and 0xC223
NCP: 0x8021 and ....
Data: 0x0021 and ....

LCP: Link Control Protocol
AP: Authentication Protocol
NCP: Network Control Protocol

Figure 11.35 *LCP packet encapsulated in a frame*

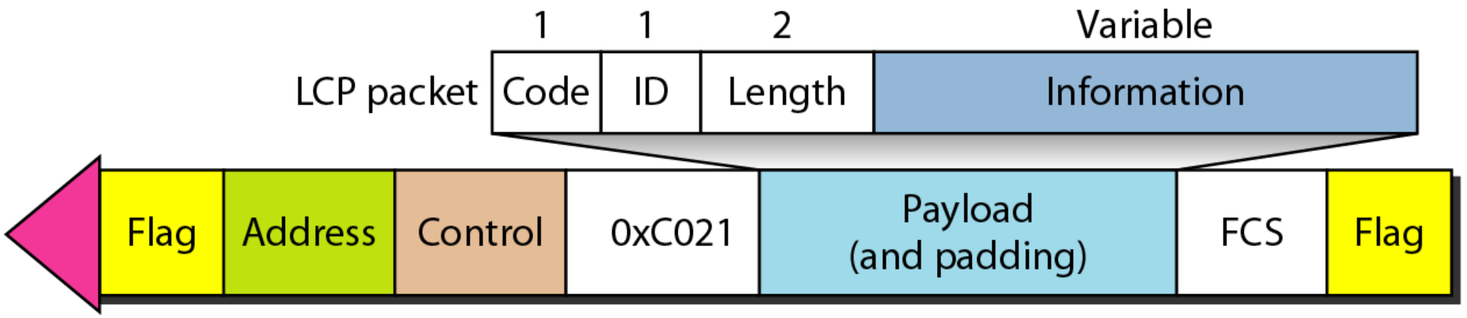


Table 11.2 ***LCP packets***

Code	*Packet Type*	*Description*
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

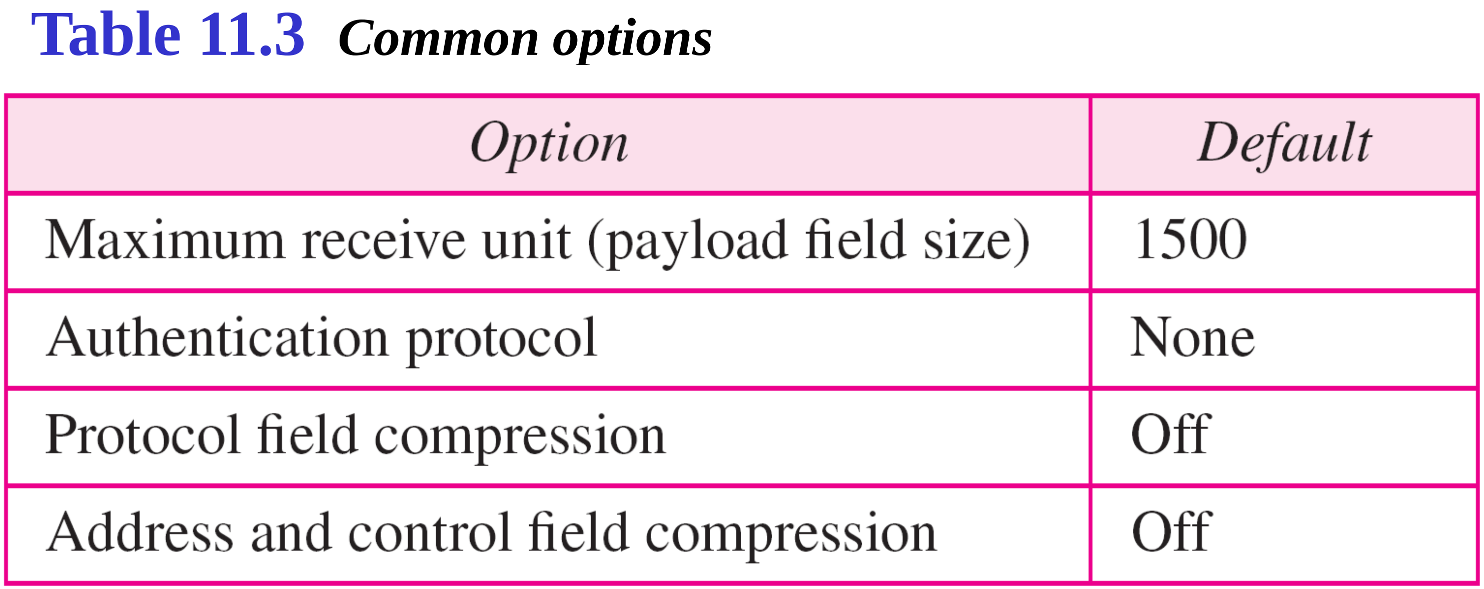
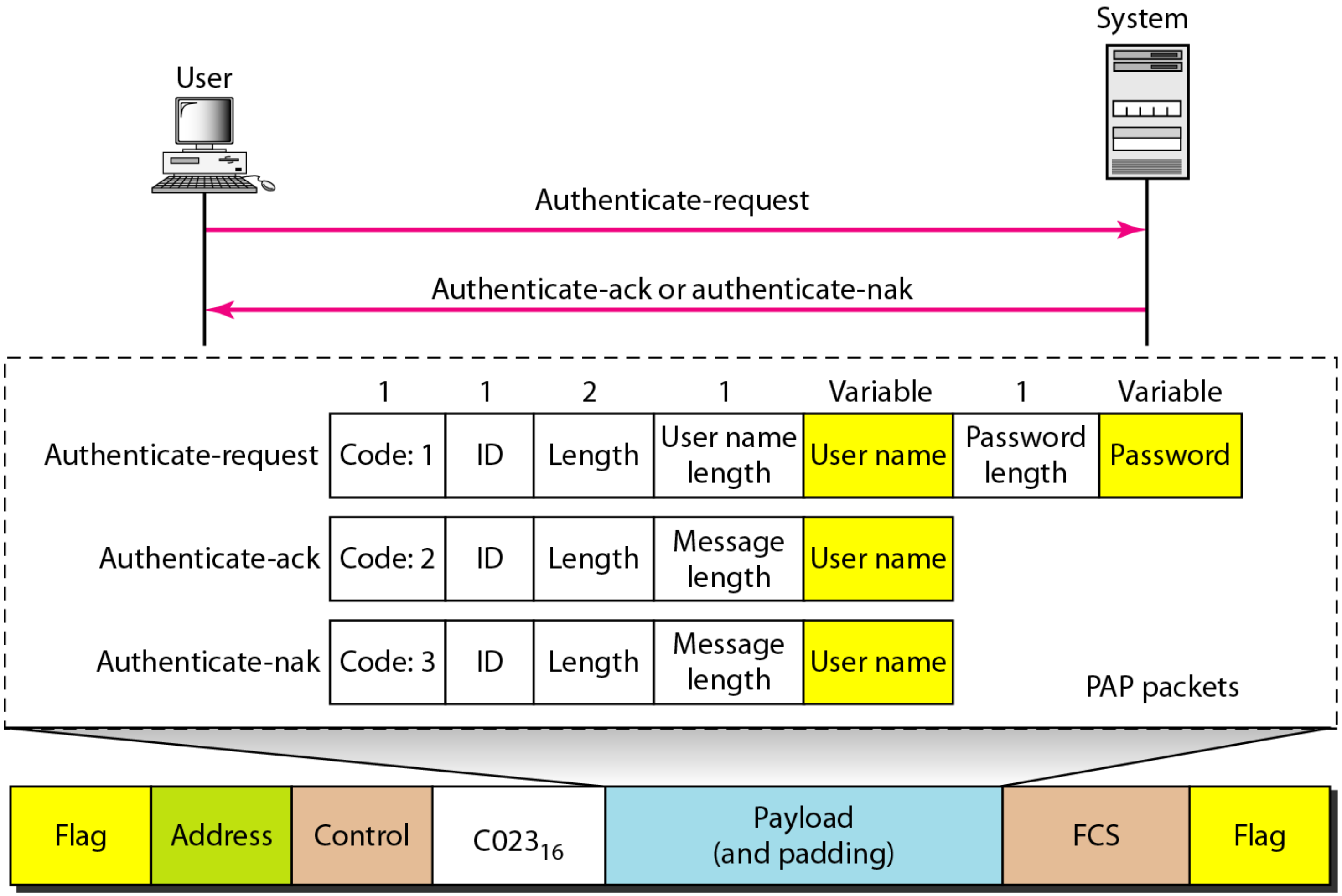


Table 11.3 ***Common options***

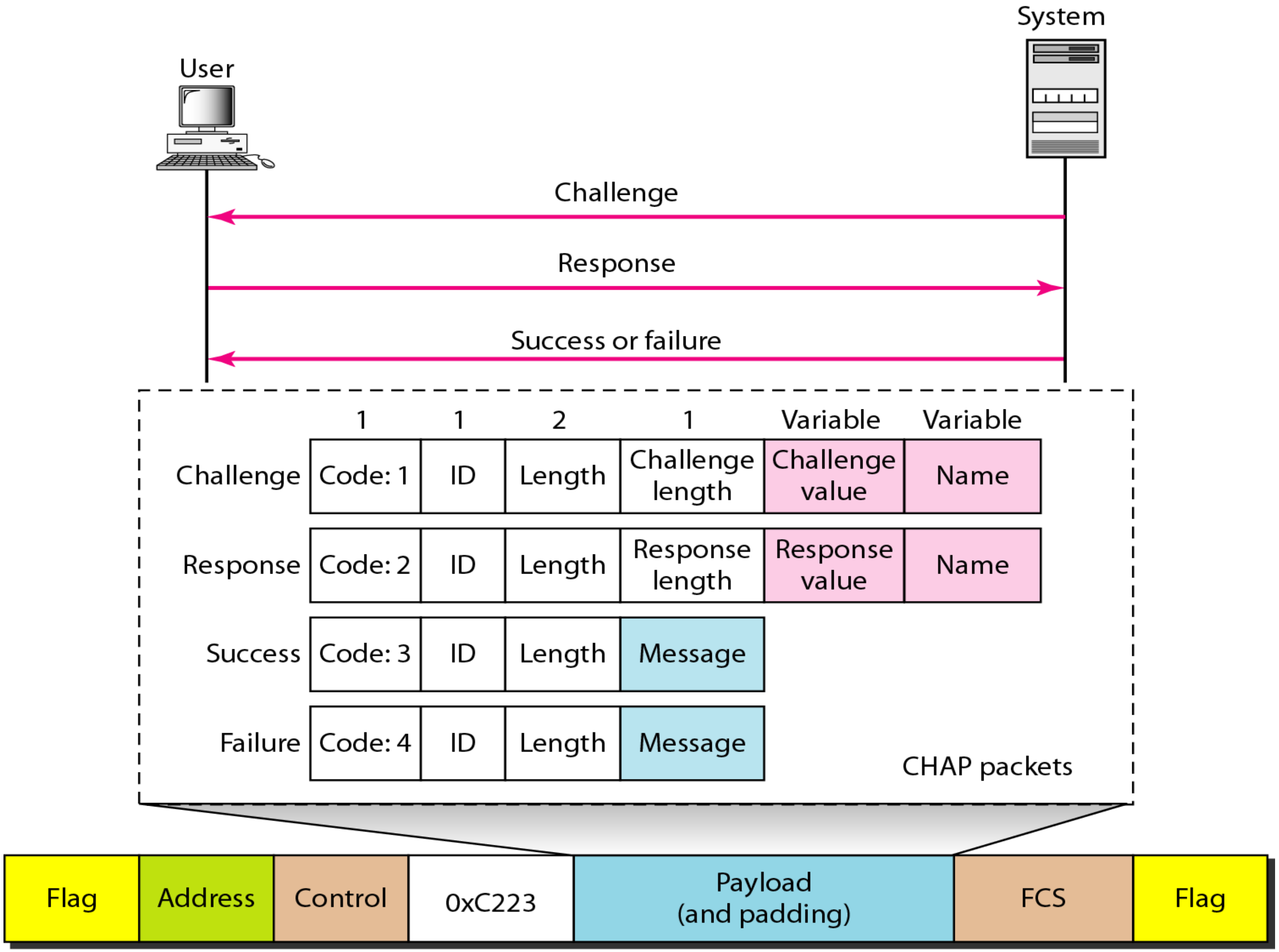
Option	*Default*
Maximum receive unit (payload field size)	1500
Authentication protocol	None
Protocol field compression	Off
Address and control field compression	Off

Figure 11.36 *PAP packets encapsulated in a PPP frame*



	1	1	2	1	Variable	1	Variable
Authenticate-request	Code: 1	ID	Length	User name length	User name	Password length	Password
Authenticate-ack	Code: 2	ID	Length	Message length	User name		
Authenticate-nak	Code: 3	ID	Length	Message length	User name		

Figure 11.37 *CHAP packets encapsulated in a PPP frame*



	1	1	2	1	Variable	Variable
Challenge	Code: 1	ID	Length	Challenge length	Challenge value	Name
Response	Code: 2	ID	Length	Response length	Response value	Name
Success	Code: 3	ID	Length	Message		
Failure	Code: 4	ID	Length	Message		

Flag	Address	Control	0xC223	Payload (and padding)	FCS	Flag

Figure 11.38 *IPCP packet encapsulated in PPP frame*

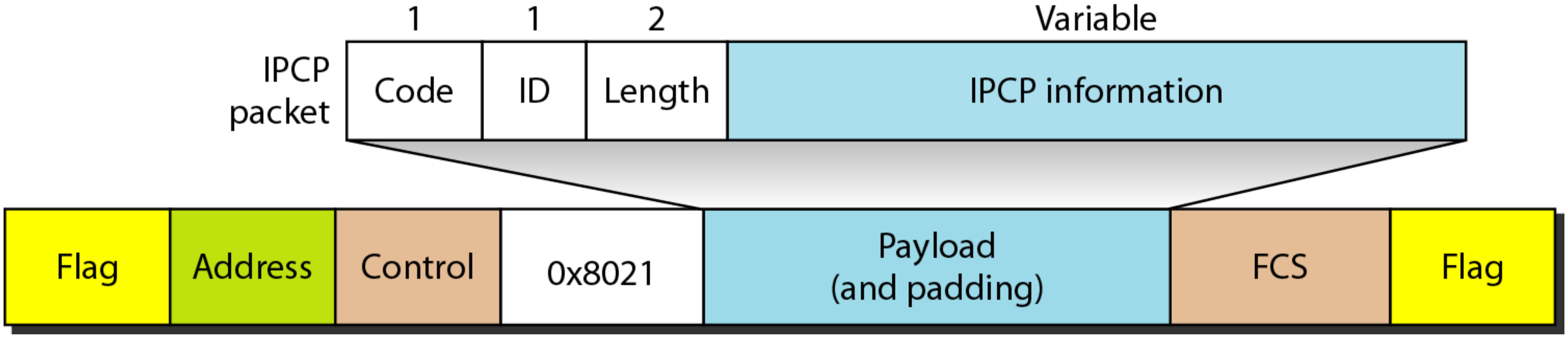


Table 11.4 ***Code value for IPCP packets***

Code	*IPCP Packet*
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*

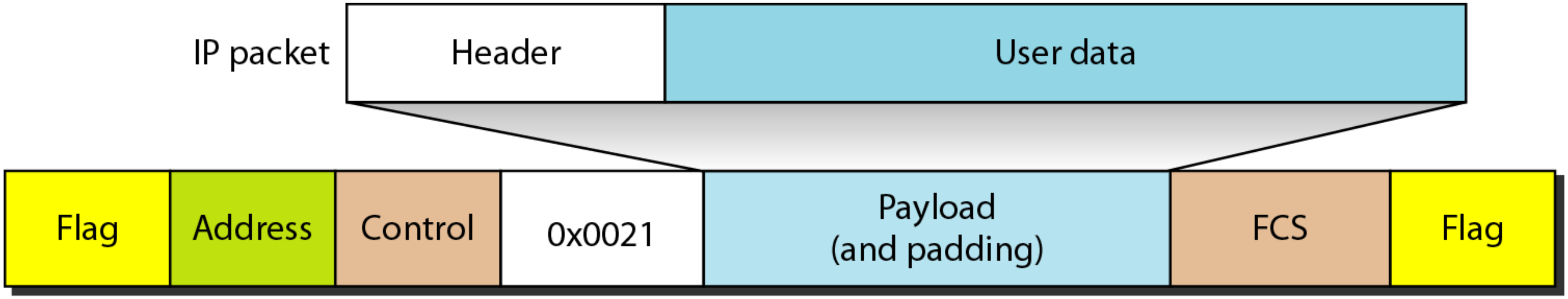
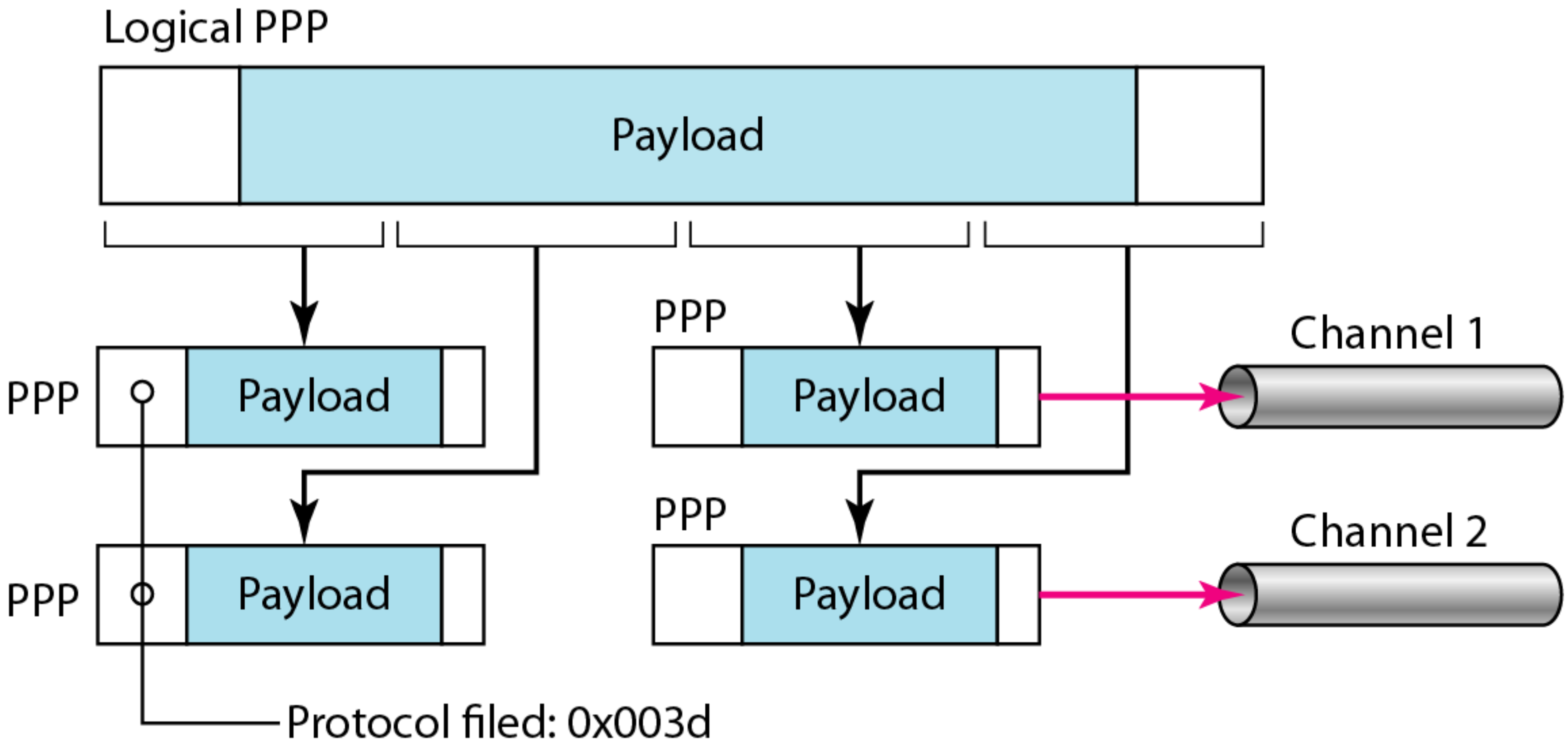


Figure 11.40 *Multilink PPP*



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*

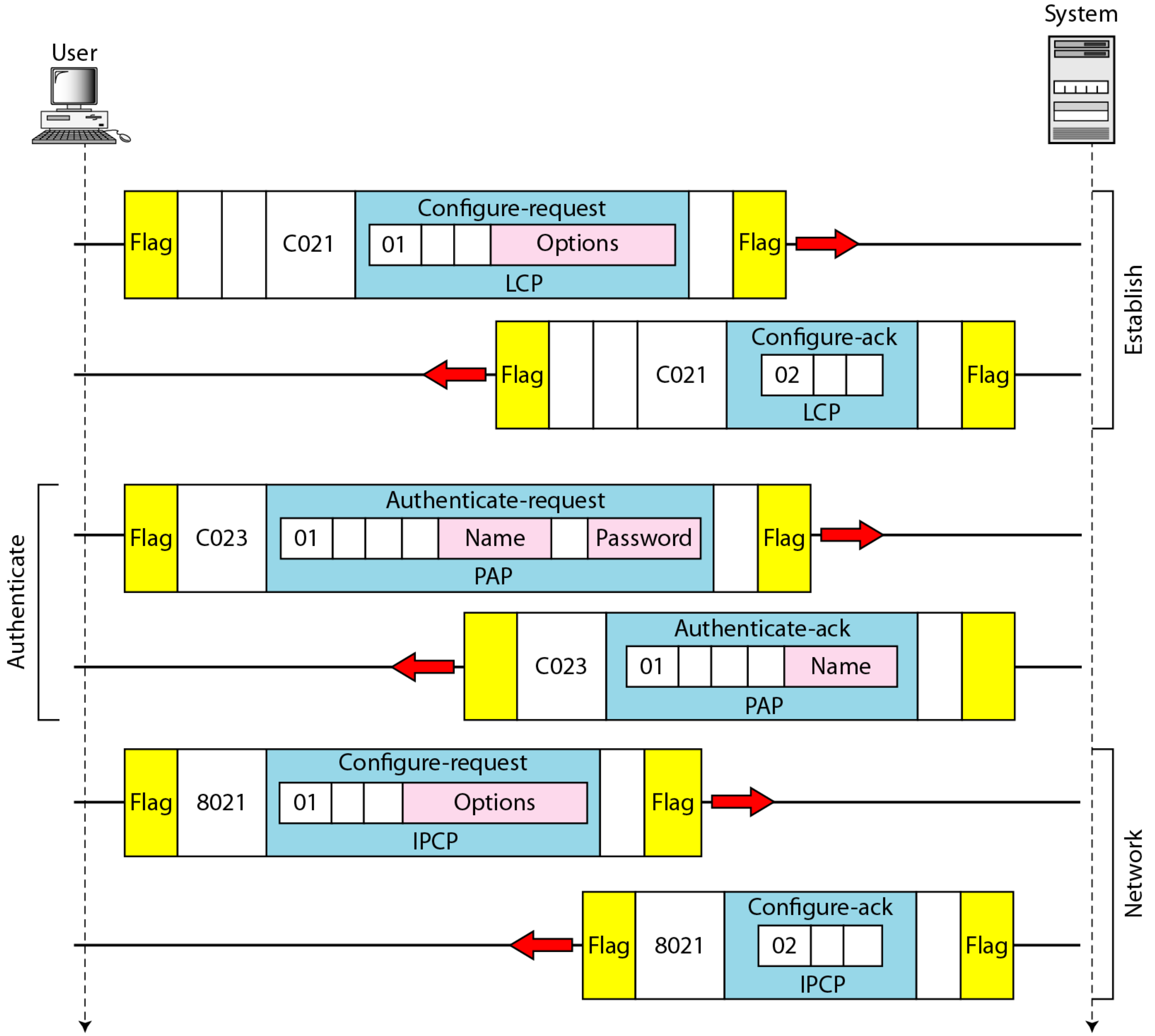
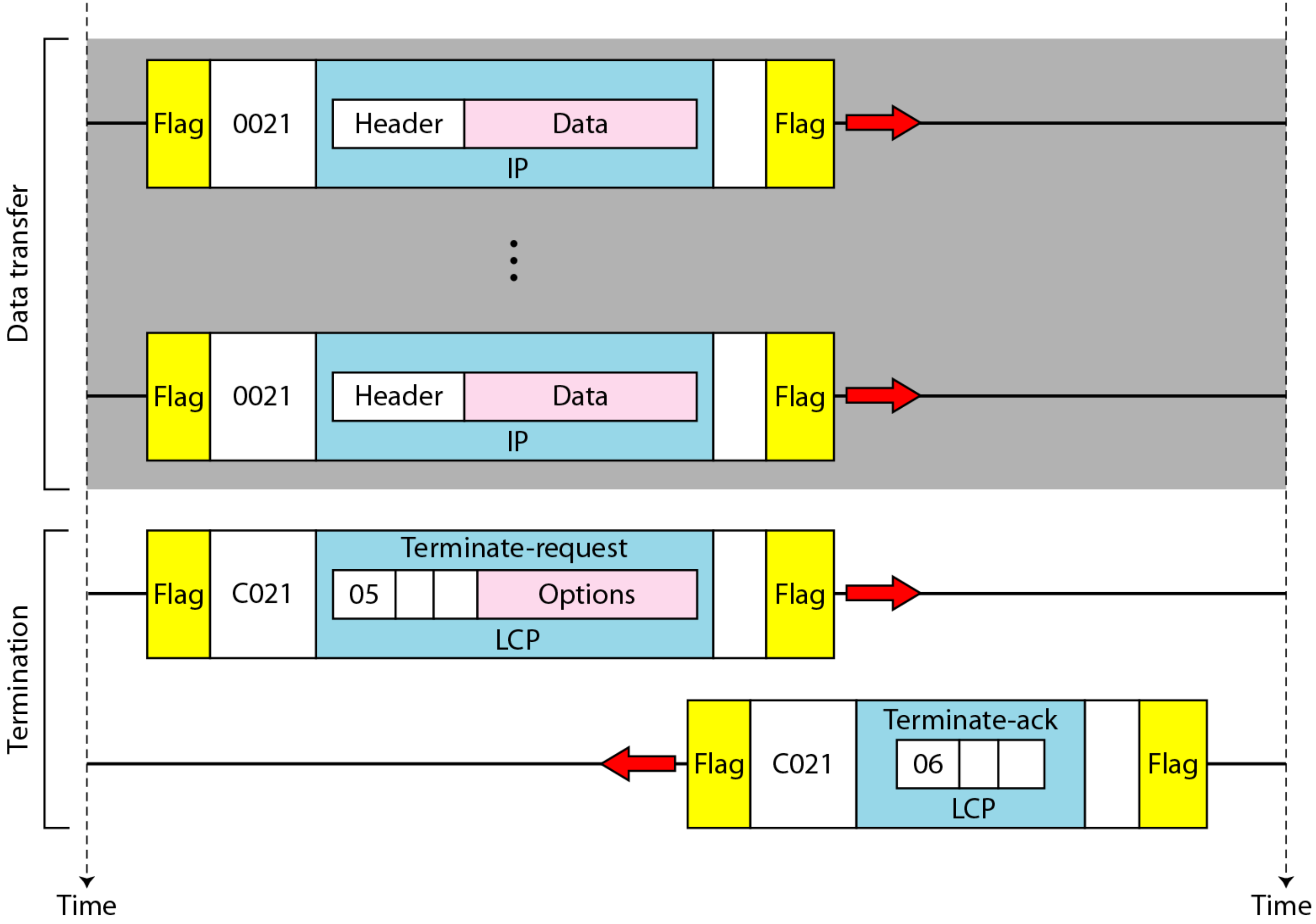


Figure 11.41 *An example (continued)*



COMPUTER COMMUNICATION NETWORKS

(15EC64)

Chapter 12

Multiple Access

Figure 12.1 *Data link layer divided into two functionality-oriented sublayers*

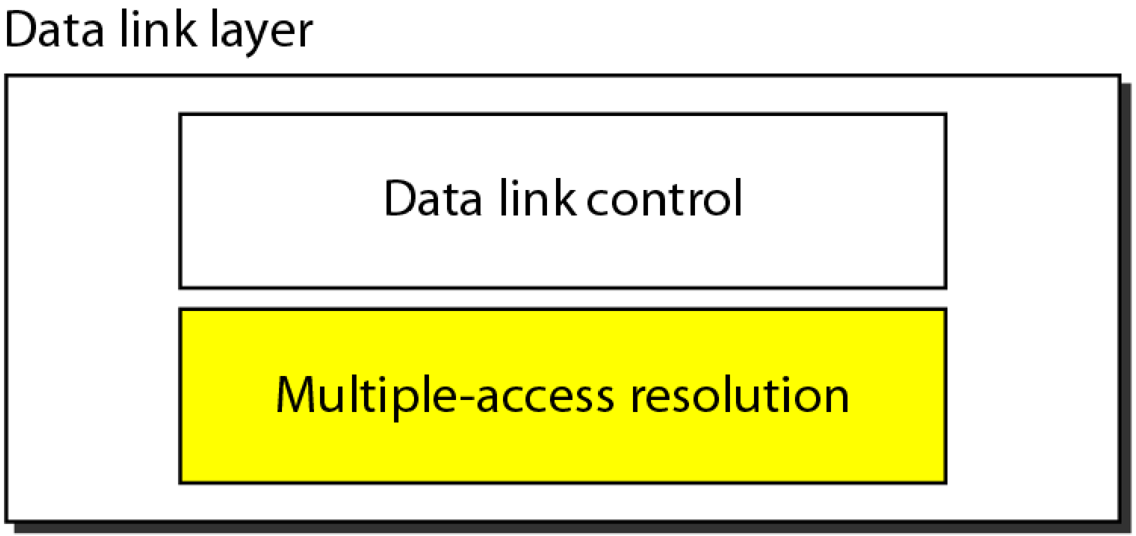
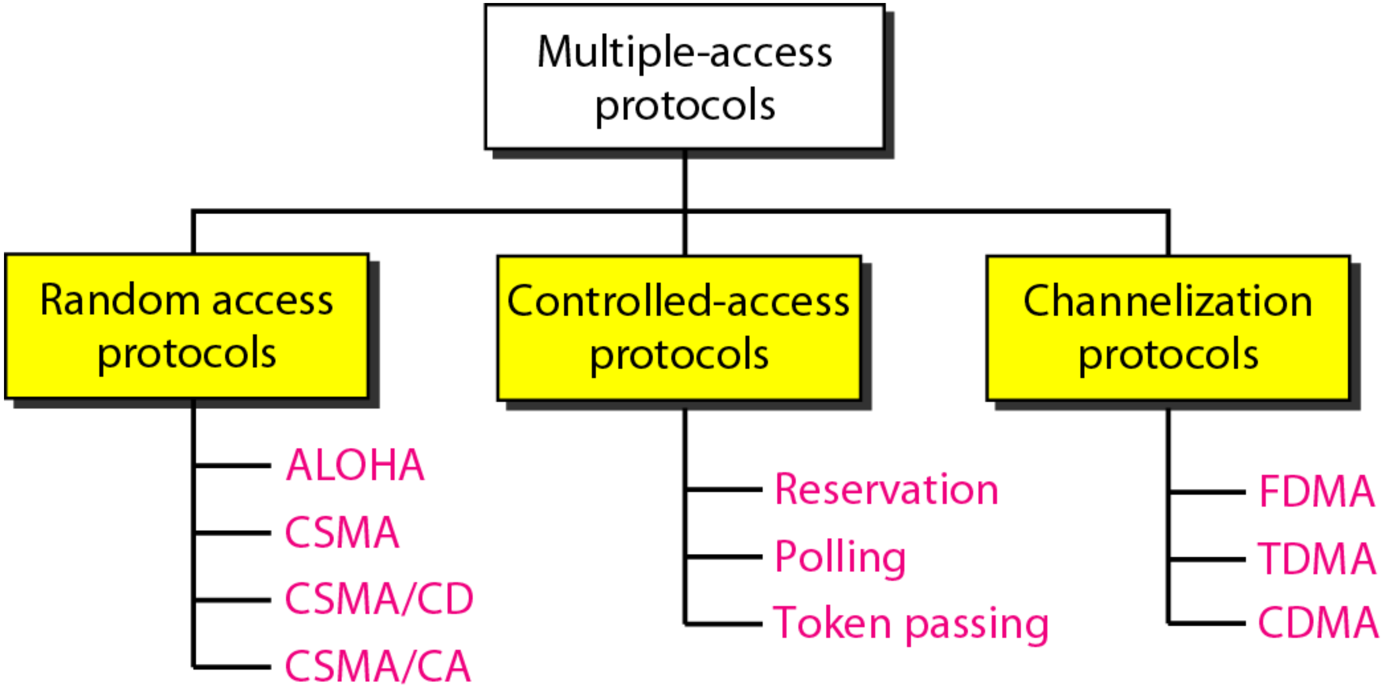


Figure 12.2 *Taxonomy of multiple-access protocols discussed in this chapter*



12-1 RANDOM ACCESS

In random access or contention methods, no station is superior to another station and none is assigned the control over another. No station permits, or does not permit, another station to send. At each instance, a station that has data to send uses a procedure defined by the protocol to make a decision on whether or not to send.

Topics discussed in this section:

ALOHA
Carrier Sense Multiple Access
Carrier Sense Multiple Access with Collision Detection
Carrier Sense Multiple Access with Collision Avoidance

Figure 12.3 *Frames in a pure ALOHA network*

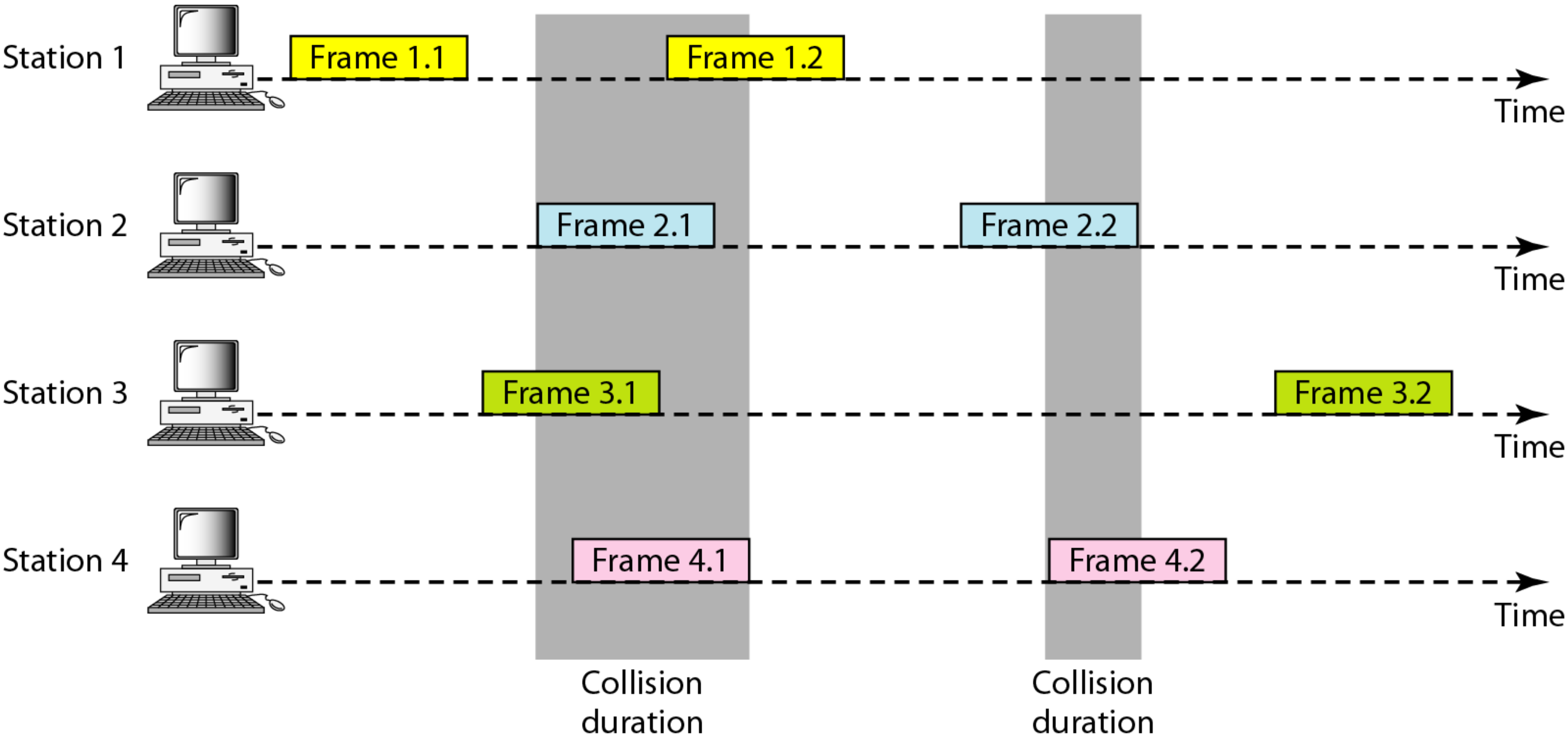
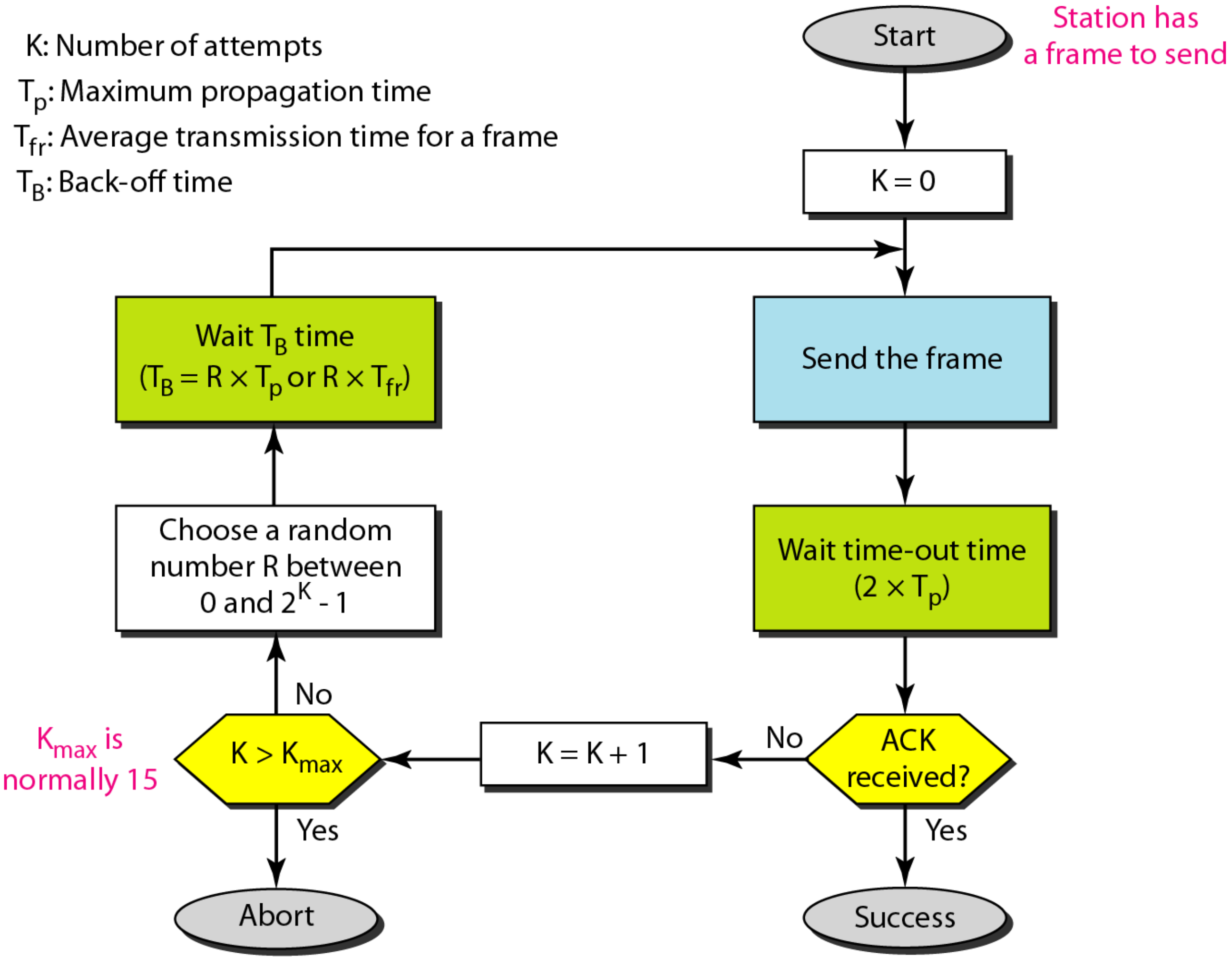


Figure 12.4 *Procedure for pure ALOHA protocol*


K: Number of attempts
T_p: Maximum propagation time
T_{fr}: Average transmission time for a frame
T_B: Back-off time
Start
Station has a frame to send
K = 0
Wait T_B time ($T_B = R \times T_p$ or $R \times T_{fr}$)
Send the frame
Choose a random number R between 0 and $2^K - 1$
Wait time-out time ($2 \times T_p$)
No
K_{max} is normally 15
$K > K_{max}$
K = K + 1
No
ACK received?
Yes
Yes
Abort
Success

Example 12.1

The stations on a wireless ALOHA network are a maximum of 600 km apart. If we assume that signals propagate at 3 × 10⁸ m/s, we find

$$T_p = (600 \times 10^5) / (3 \times 10^8) = 2 \text{ ms.}$$

Now we can find the value of T_B for different values of K .

a. For K = 1, the range is {0, 1}. The station needs to| generate a random number with a value of 0 or 1. This means that T_B is either 0 ms (0 × 2) or 2 ms (1 × 2), based on the outcome of the random variable.

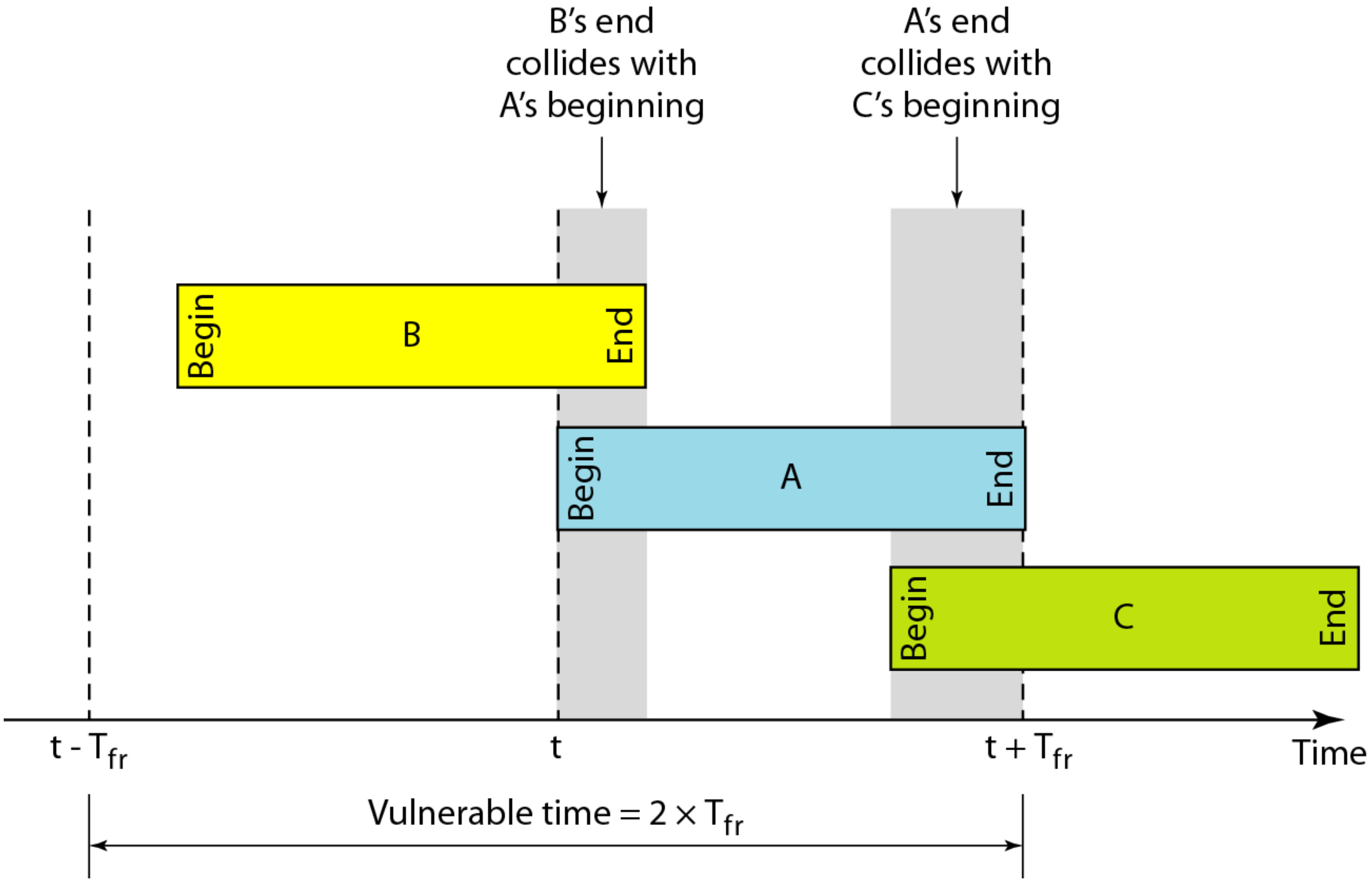
Example 12.1 (continued)

***b.** For K = 2, the range is {0, 1, 2, 3}. This means that T_B can be 0, 2, 4, or 6 ms, based on the outcome of the random variable.*

***c.** For K = 3, the range is {0, 1, 2, 3, 4, 5, 6, 7}. This means that T_B can be 0, 2, 4, . . . , 14 ms, based on the outcome of the random variable.*

***d.** We need to mention that if K > 10, it is normally set to 10.*

Figure 12.5 *Vulnerable time for pure ALOHA protocol*


B's end collides with A's beginning
A's end collides with C's beginning
Begin
B
End
Begin
A
End
Begin
C
End
$t - T_{fr}$
t
$t + T_{fr}$
Time
Vulnerable time = $2 \times T_{fr}$

Example 12.2

A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the requirement to make this frame collision-free?

Solution

Average frame transmission time T_{fr} is 200 bits/200 kbps or 1 ms. The vulnerable time is 2 × 1 ms = 2 ms. This means no station should send later than 1 ms before this station starts transmission and no station should start sending during the one 1-ms period that this station is sending.

Note

The throughput for pure ALOHA is $S = G \times e^{-2G}$.

The maximum throughput $S_{max} = 0.184$ when G= (1/2).

Example 12.3

A pure ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces
a. 1000 frames per second b. 500 frames per second
c. 250 frames per second.

Solution

The frame transmission time is 200/200 kbps or 1 ms.

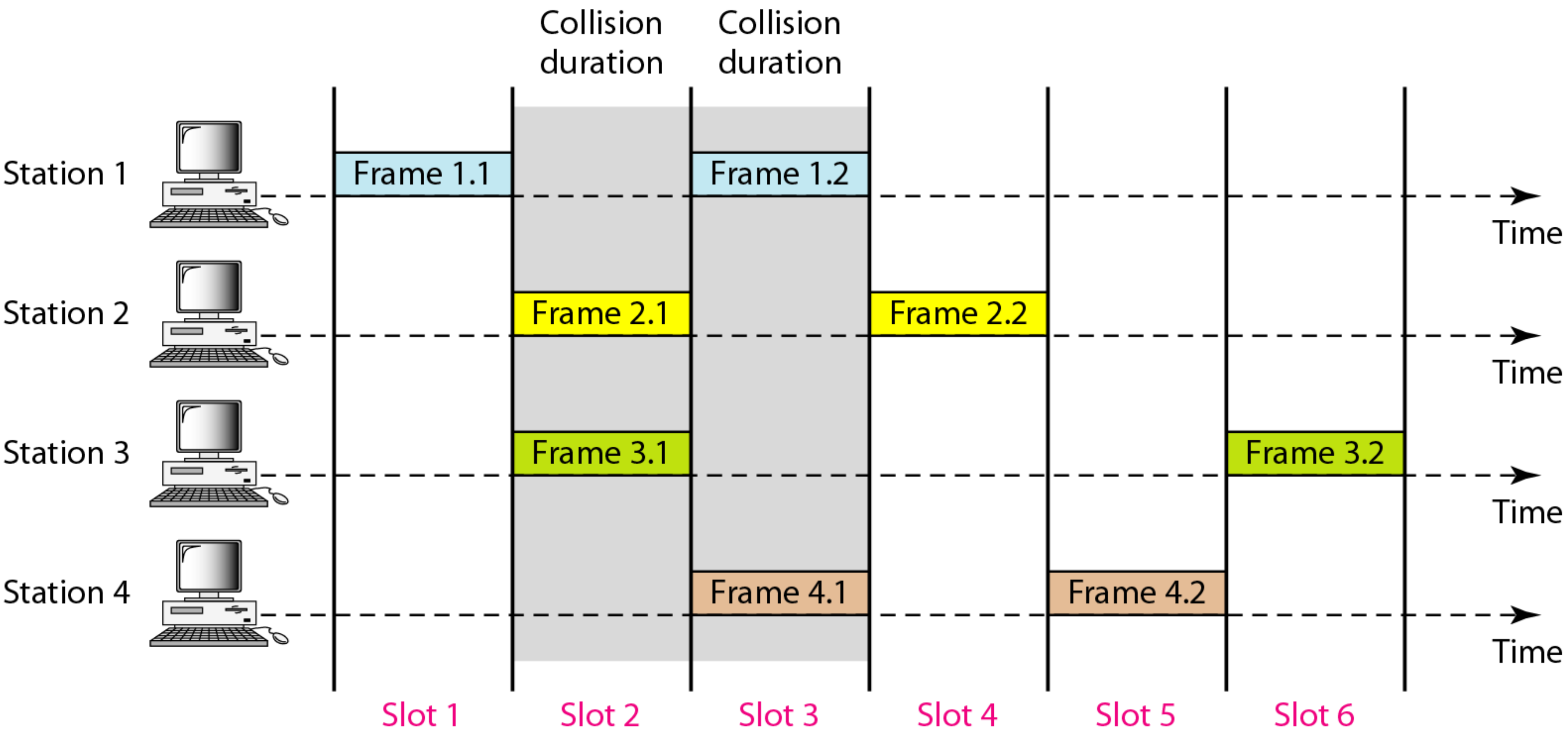
a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case $S = G \times e^{-2G}$ or $S = 0.135$ (13.5 percent). This means that the throughput is 1000 × 0.135 = 135 frames. Only 135 frames out of 1000 will probably survive.

Example 12.3 (continued)

***b.** If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case $S = G \times e^{-2G}$ or $S = 0.184$ (18.4 percent). This means that the throughput is 500 × 0.184 = 92 and that only 92 frames out of 500 will probably survive. Note that this is the maximum throughput case, percentagewise.*

***c.** If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case $S = G \times e^{-2G}$ or $S = 0.152$ (15.2 percent). This means that the throughput is 250 × 0.152 = 38. Only 38 frames out of 250 will probably survive.*

Figure 12.6 *Frames in a slotted ALOHA network*

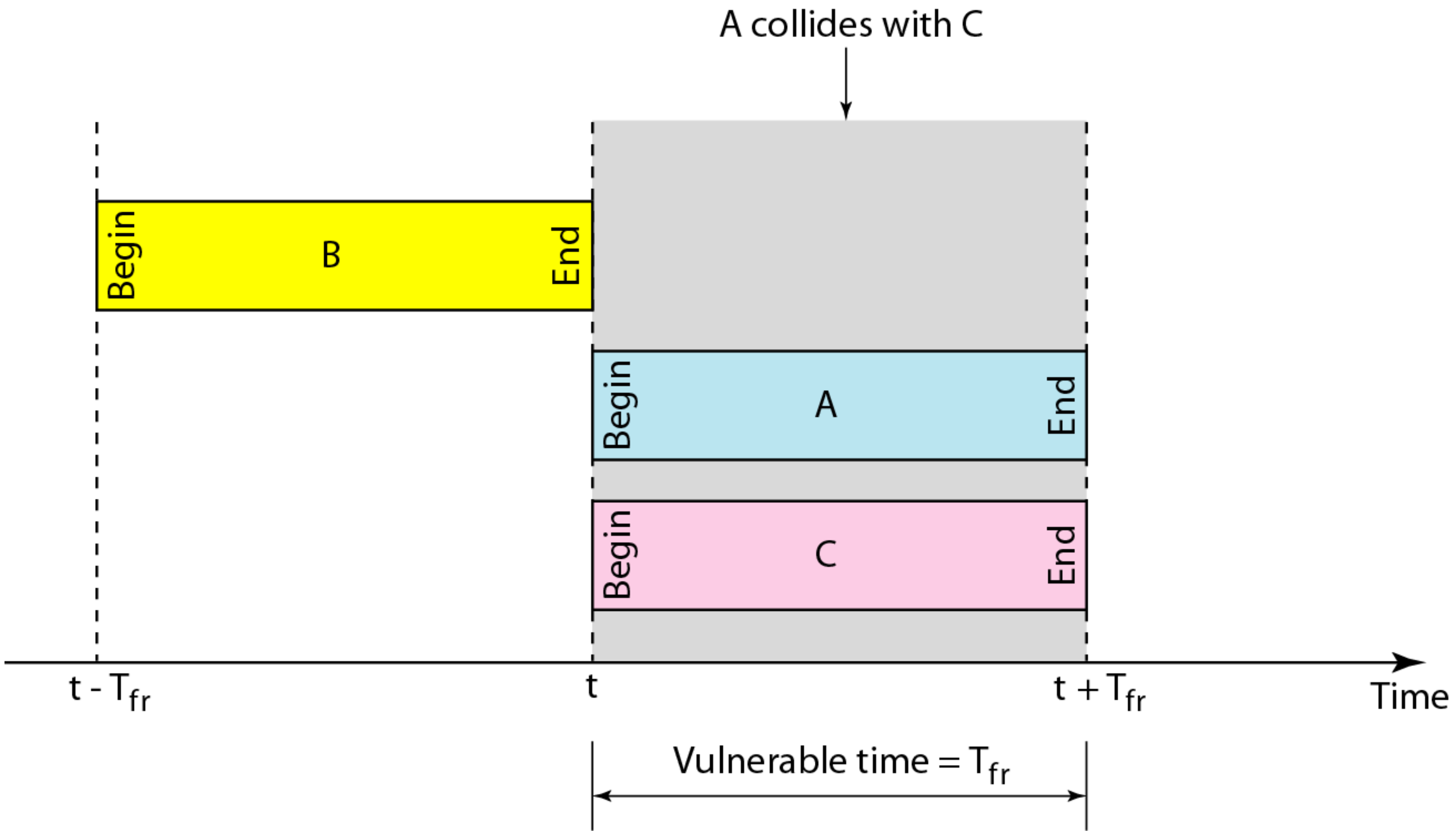


Note

The throughput for slotted ALOHA is $S = G \times e^{-G}$.

The maximum throughput $S_{max} = 0.368$ when G = 1.

Figure 12.7 *Vulnerable time for slotted ALOHA protocol*



Example 12.4

A slotted ALOHA network transmits 200-bit frames on a shared channel of 200 kbps. What is the throughput if the system (all stations together) produces

a. 1000 frames per second ***b. 500 frames per second***

c. 250 frames per second.

Solution

The frame transmission time is 200/200 kbps or 1 ms.

a. If the system creates 1000 frames per second, this is 1 frame per millisecond. The load is 1. In this case $S = G\times e^{-G}$ ***or*** $S = 0.368$ ***(36.8 percent). This means that the throughput is 1000 × 0.0368 = 368 frames. Only 386 frames out of 1000 will probably survive.***

Example 12.4 (continued)

b. If the system creates 500 frames per second, this is (1/2) frame per millisecond. The load is (1/2). In this case $S = G \times e^{-G}$ or $S = 0.303$ (30.3 percent). This means that the throughput is 500 × 0.0303 = 151. Only 151 frames out of 500 will probably survive.

c. If the system creates 250 frames per second, this is (1/4) frame per millisecond. The load is (1/4). In this case $S = G \times e^{-G}$ or $S = 0.195$ (19.5 percent). This means that the throughput is 250 × 0.195 = 49. Only 49 frames out of 250 will probably survive.

Figure 12.8 *Space/time model of the collision in CSMA*

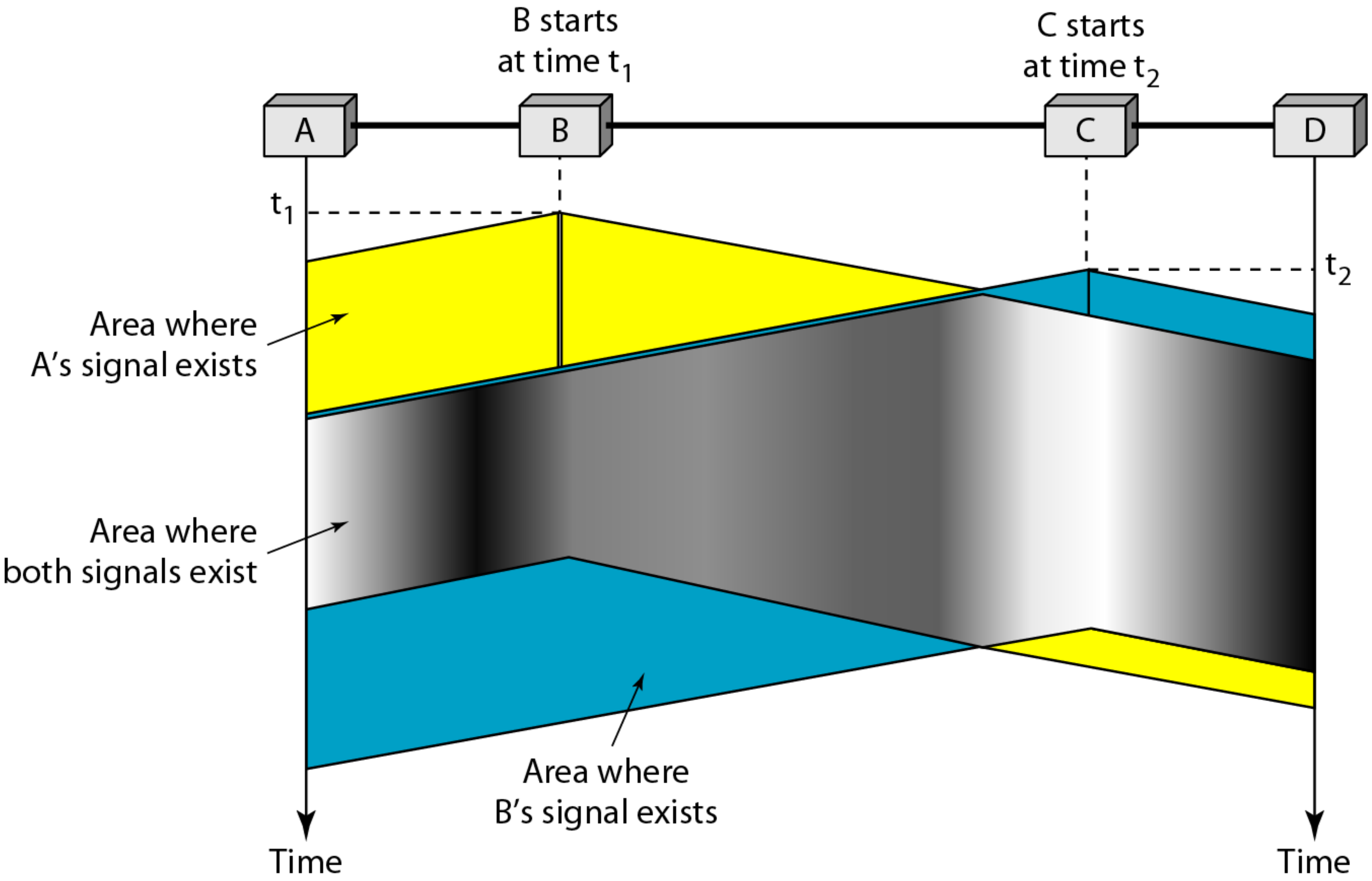


Figure 12.9 *Vulnerable time in CSMA*

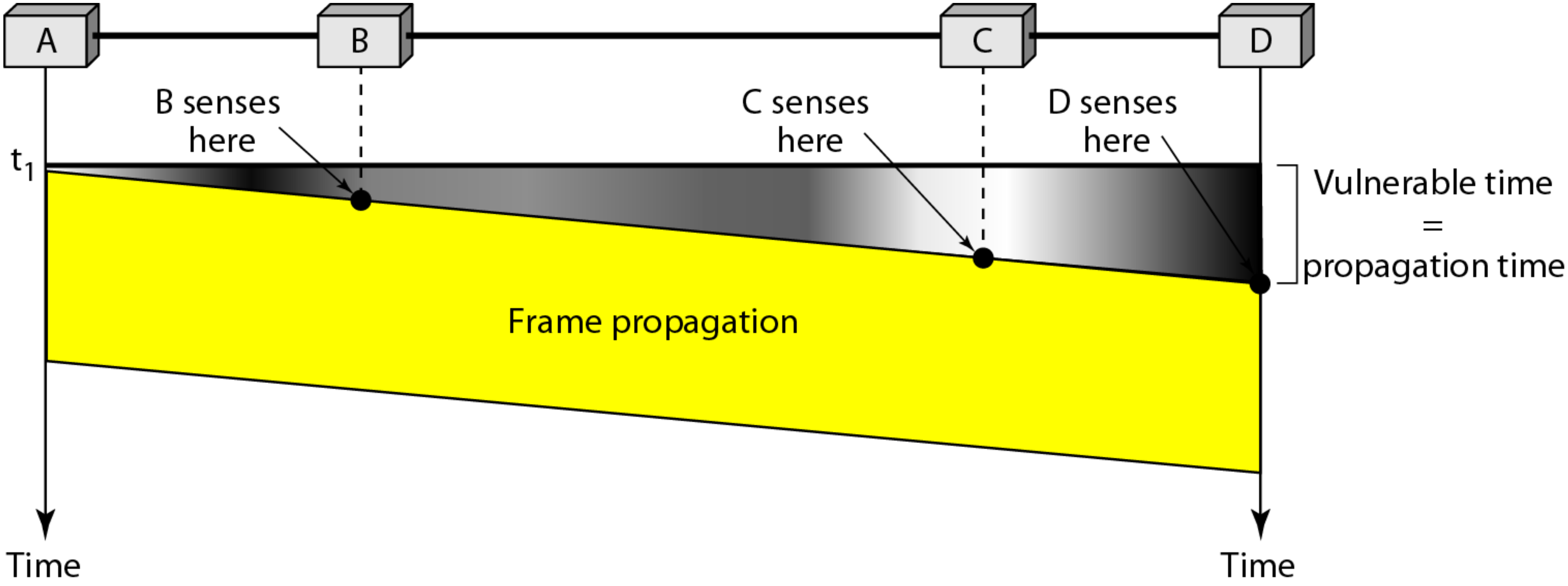
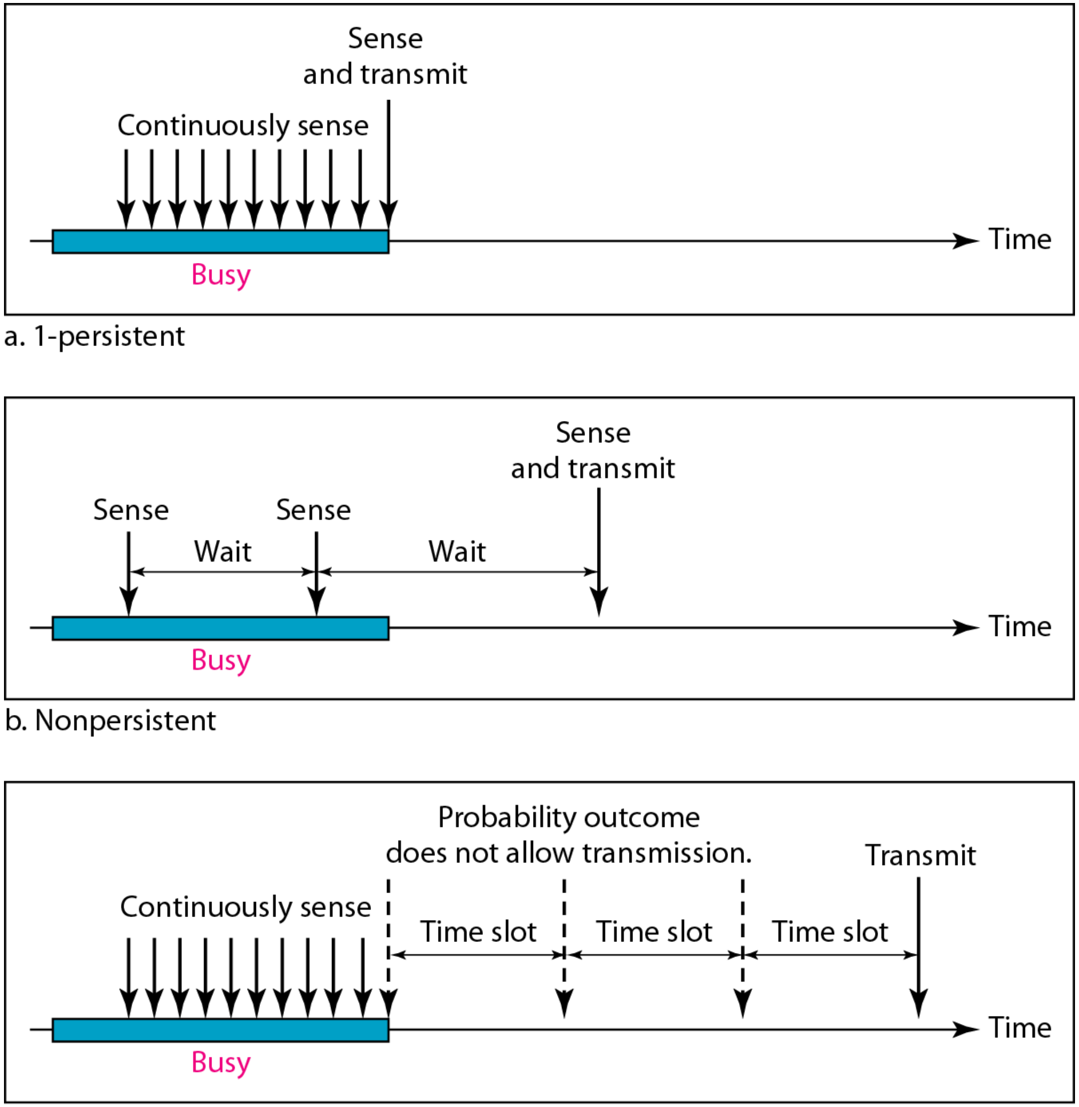


Figure 12.10 *Behavior of three persistence methods*

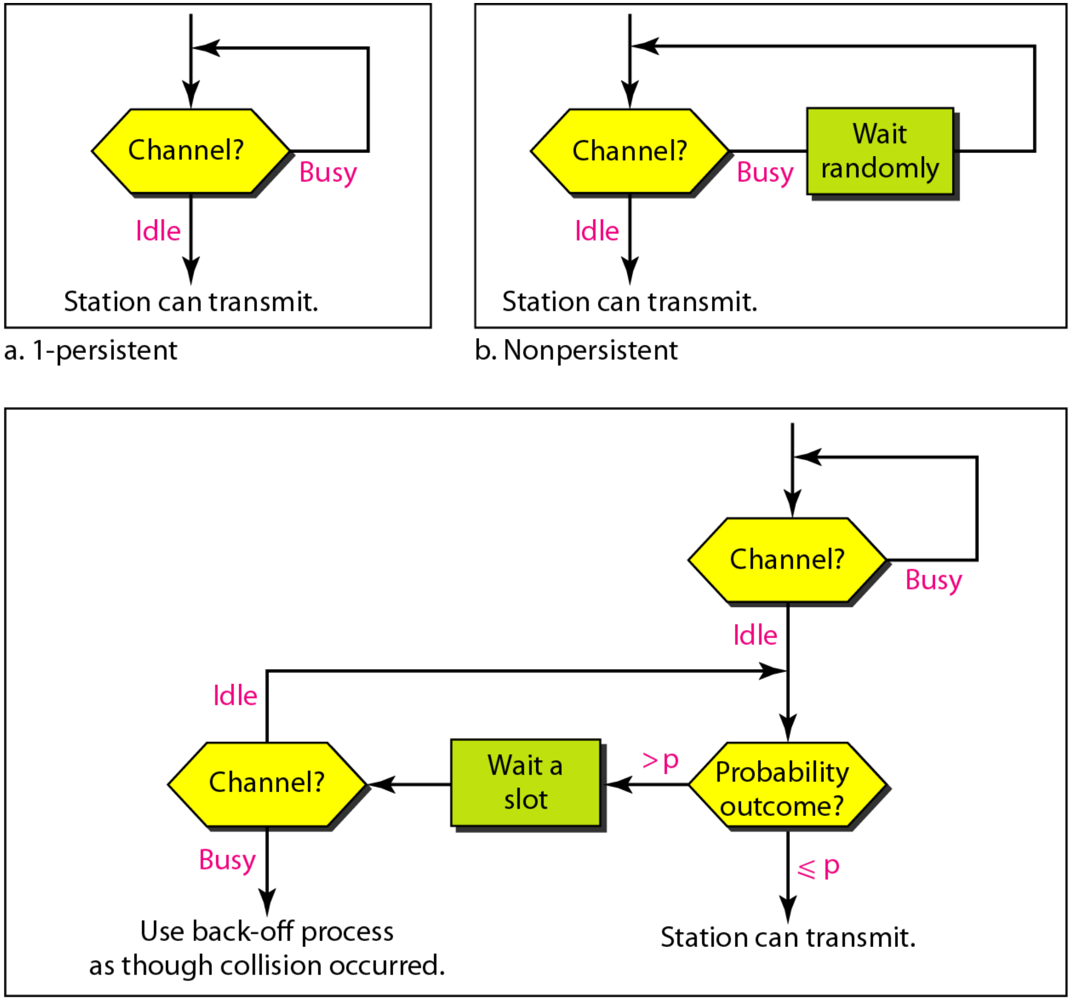


a. 1-persistent

b. Nonpersistent

c. p-persistent

Figure 12.11 *Flow diagram for three persistence methods*



a. 1-persistent

b. Nonpersistent

c. p-persistent

Figure 12.12 *Collision of the first bit in CSMA/CD*

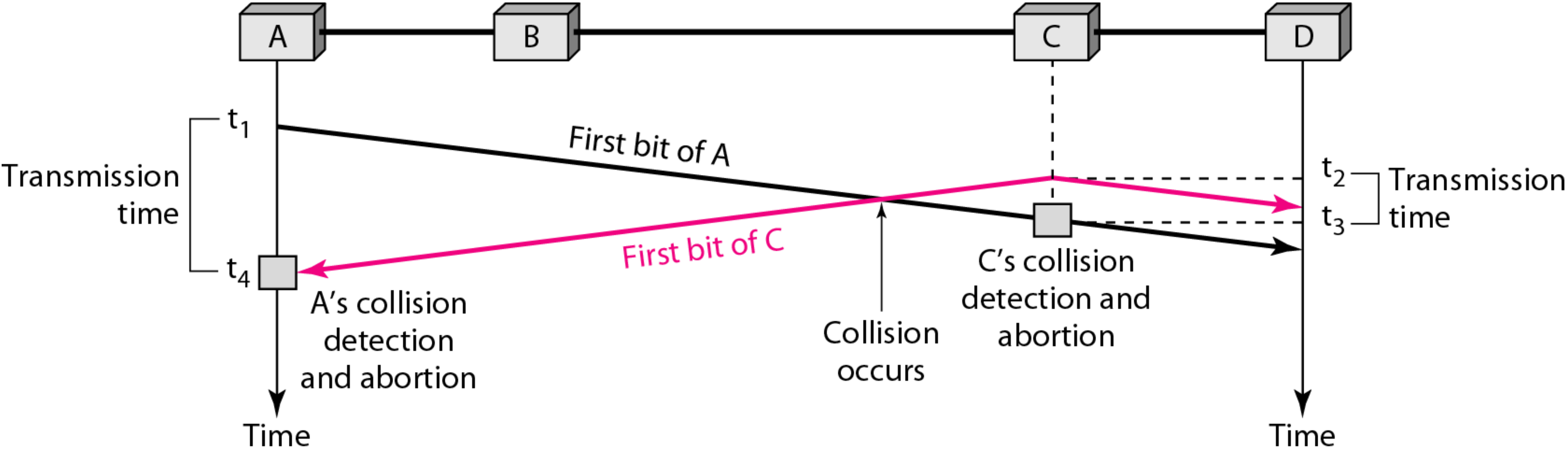
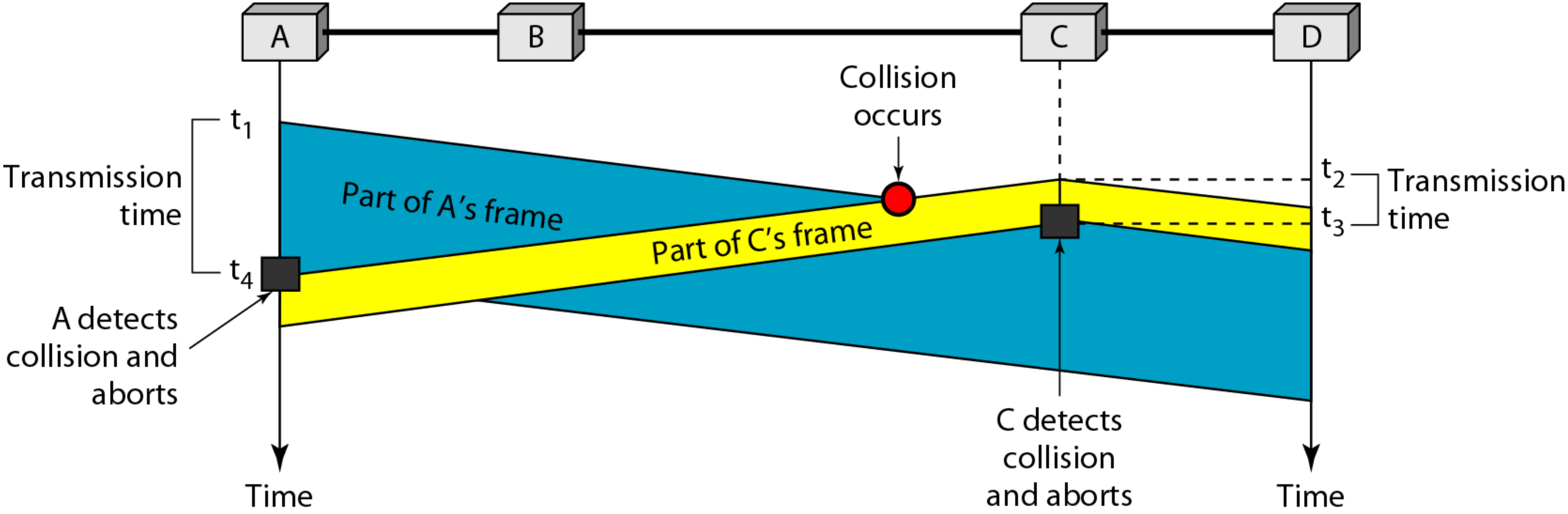


Figure 12.13 *Collision and abortion in CSMA/CD*



Example 12.5

A network using CSMA/CD has a bandwidth of 10 Mbps. If the maximum propagation time (including the delays in the devices and ignoring the time needed to send a jamming signal, as we see later) is 25.6 μs, what is the minimum size of the frame?

Solution

The frame transmission time is T_{fr} = 2 × T_p = 51.2 μs. This means, in the worst case, a station needs to transmit for a period of 51.2 μs to detect the collision. The minimum size of the frame is 10 Mbps × 51.2 μs = 512 bits or 64 bytes. This is actually the minimum size of the frame for Standard Ethernet.

Figure 12.14 *Flow diagram for the CSMA/CD*

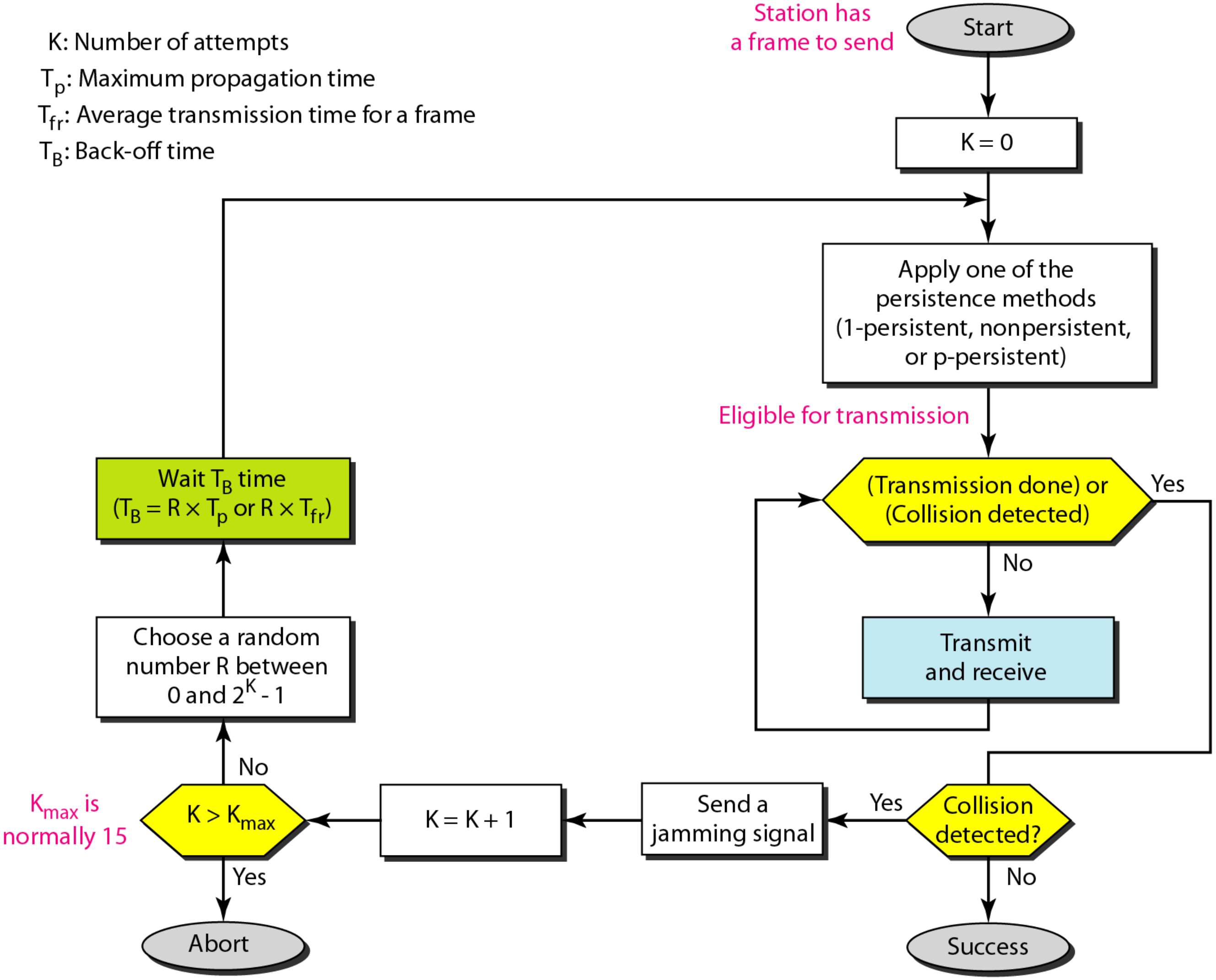

K: Number of attempts
T_p: Maximum propagation time
T_{fr}: Average transmission time for a frame
T_B: Back-off time
Station has a frame to send
Start
K = 0
Apply one of the persistence methods (1-persistent, nonpersistent, or p-persistent)
Eligible for transmission
(Transmission done) or (Collision detected)
Yes
No
Transmit and receive
Collision detected?
Yes
No
Success
Send a jamming signal
K = K + 1
$K > K_{max}$
K_{max} is normally 15
Yes
Abort
No
Choose a random number R between 0 and $2^K - 1$
Wait T_B time ($T_B = R \times T_p$ or $R \times T_{fr}$)

Figure 12.15 *Energy level during transmission, idleness, or collision*

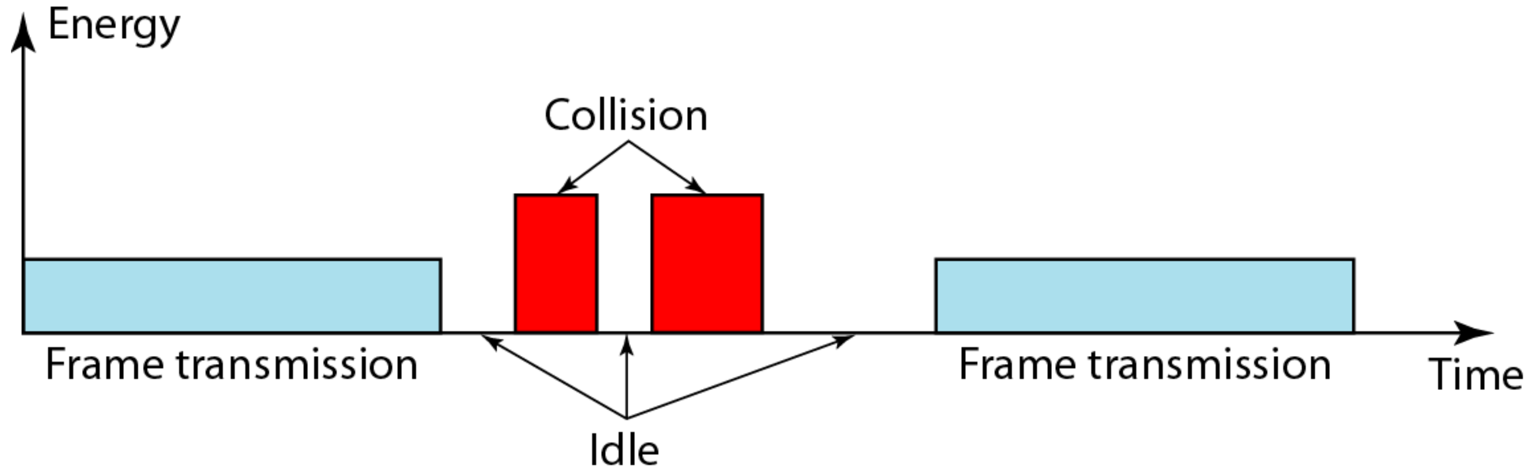
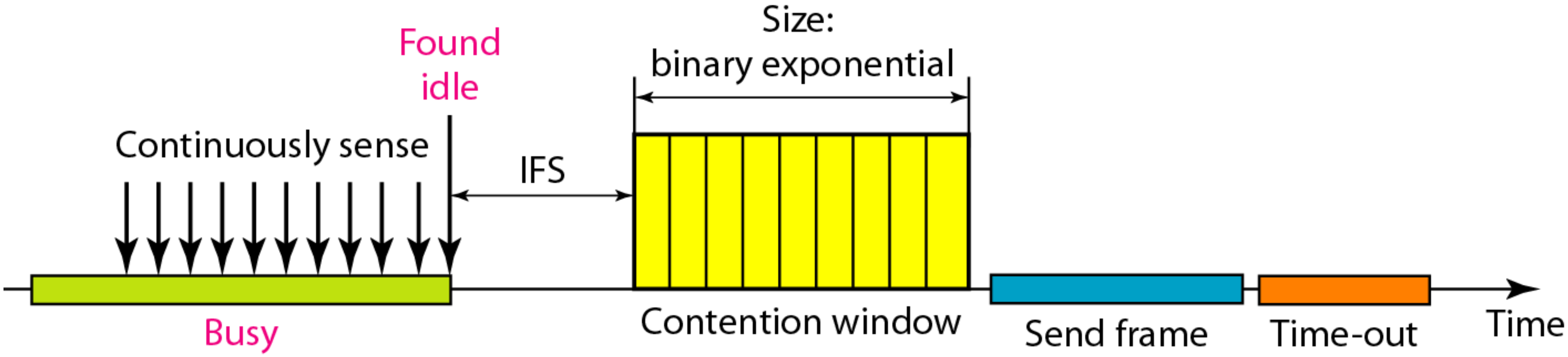


Figure 12.16 *Timing in CSMA/CA*



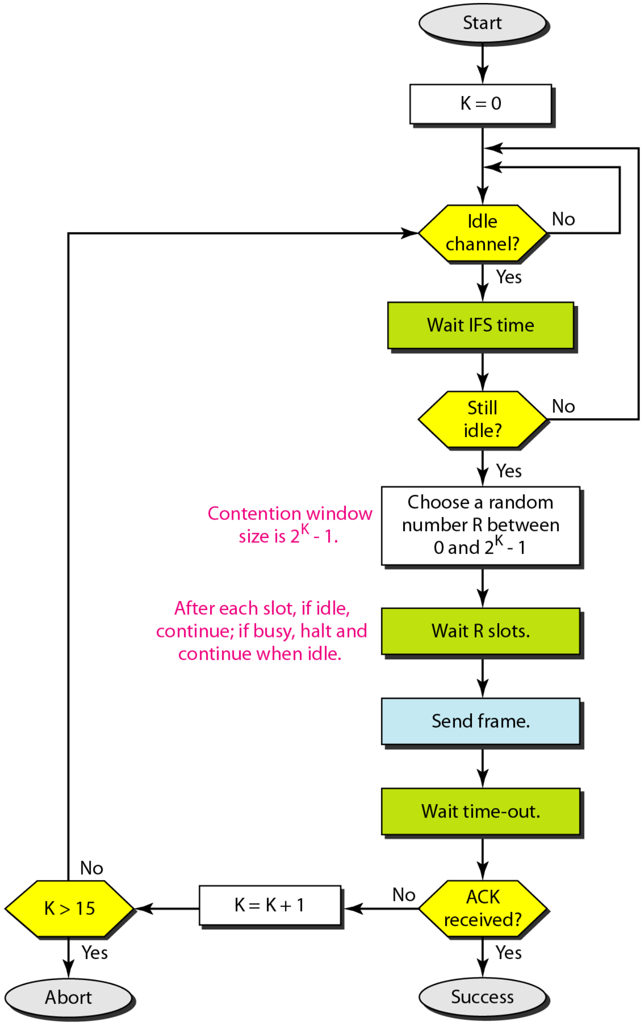
Note

In CSMA/CA, the IFS can also be used to define the priority of a station or a frame.

Note

In CSMA/CA, if the station finds the channel busy, it does not restart the timer of the contention window; it stops the timer and restarts it when the channel becomes idle.

Figure 12.17 *Flow diagram for CSMA/CA*


Start
K = 0
Idle channel?
No
Yes
Wait IFS time
Still idle?
No
Yes
Contention window size is 2^K - 1.
Choose a random number R between 0 and 2^K - 1
After each slot, if idle, continue; if busy, halt and continue when idle.
Wait R slots.
Send frame.
Wait time-out.
No
ACK received?
No
K = K + 1
K > 15
Yes
Yes
Abort
Success

12-2 CONTROLLED ACCESS

In controlled access, the stations consult one another to find which station has the right to send. A station cannot send unless it has been authorized by other stations. We discuss three popular controlled-access methods.

Topics discussed in this section:

Reservation
Polling
Token Passing

Figure 12.18 *Reservation access method*

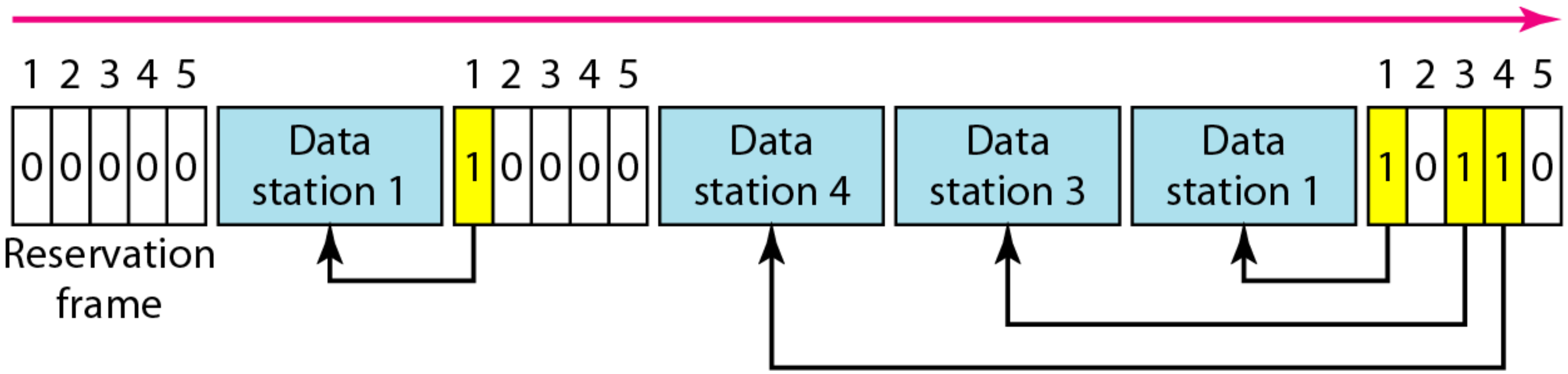


Figure 12.19 *Select and poll functions in polling access method*

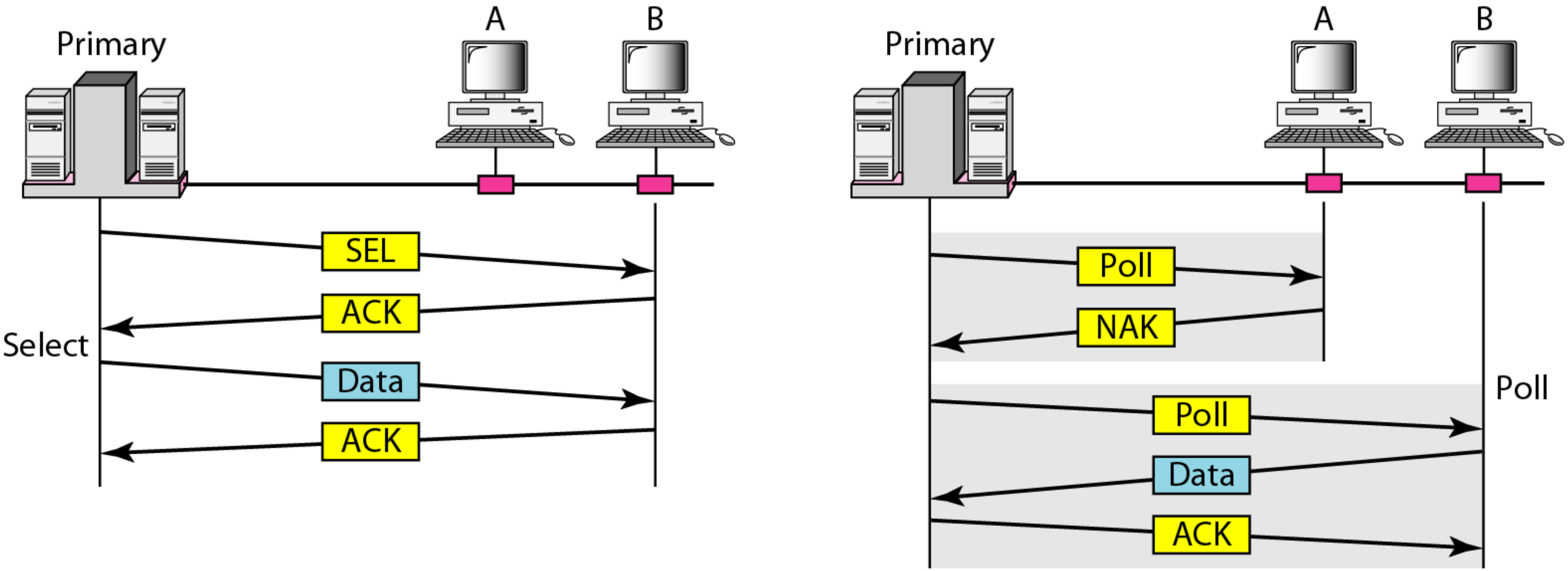
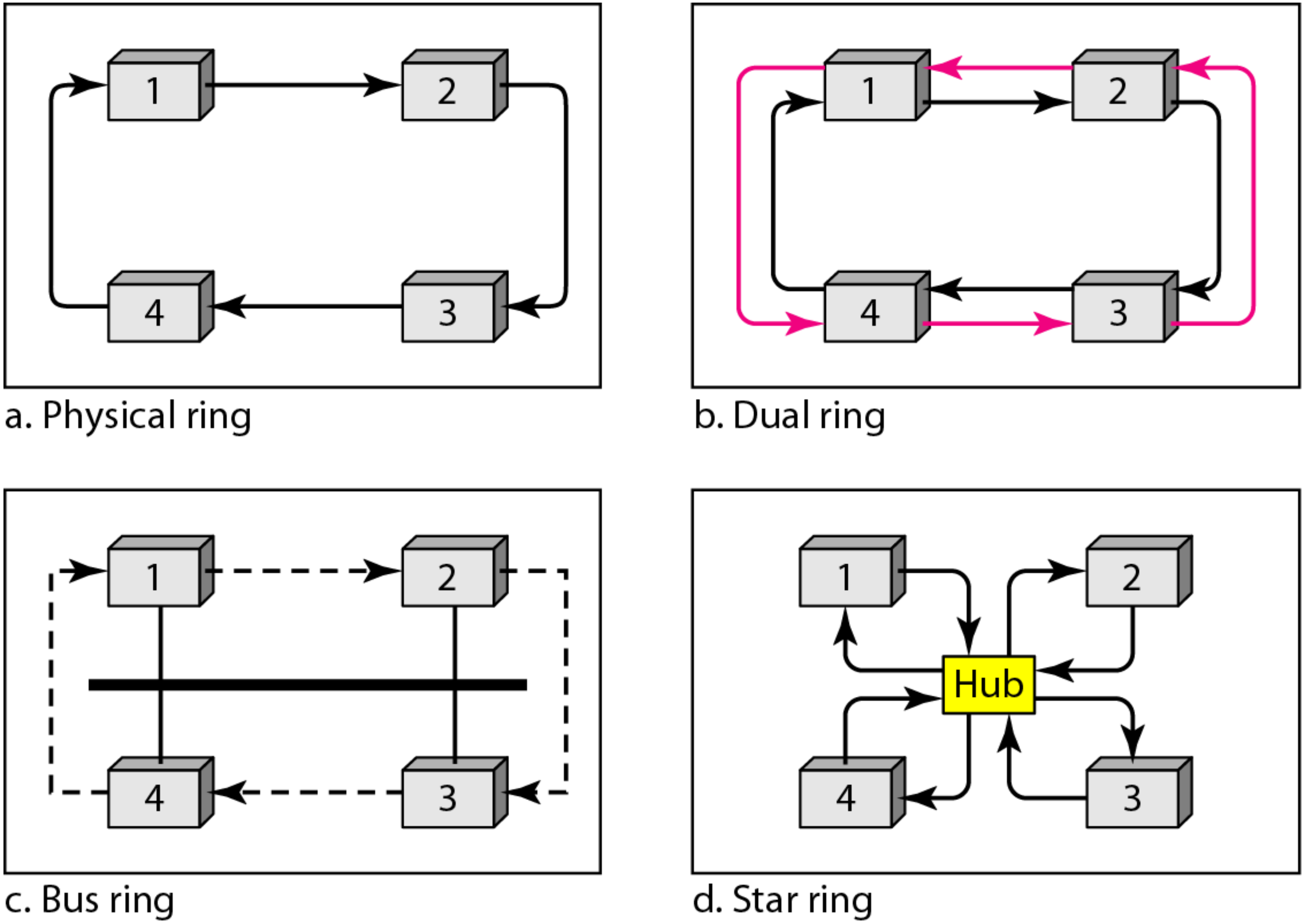


Figure 12.20 *Logical ring and physical topology in token-passing access method*



a. Physical ring

b. Dual ring

c. Bus ring

d. Star ring

12-3 CHANNELIZATION

***Channelization* is a multiple-access method in which the available bandwidth of a link is shared in time, frequency, or through code, between different stations. In this section, we discuss three channelization protocols.**

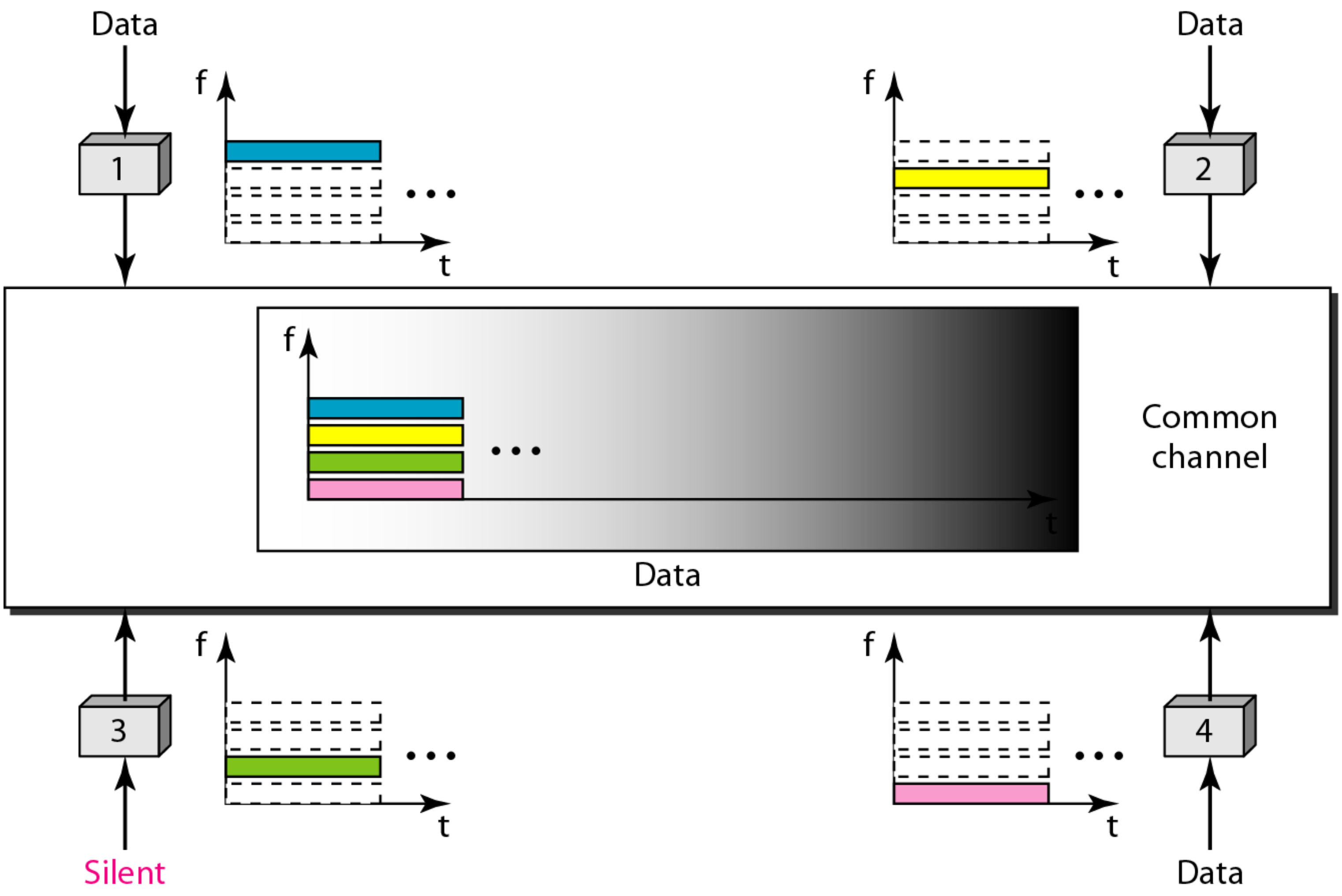
Topics discussed in this section:

Frequency-Division Multiple Access (FDMA)
Time-Division Multiple Access (TDMA)
Code-Division Multiple Access (CDMA)

Note

We see the application of all these methods in Chapter 16 when we discuss cellular phone systems.

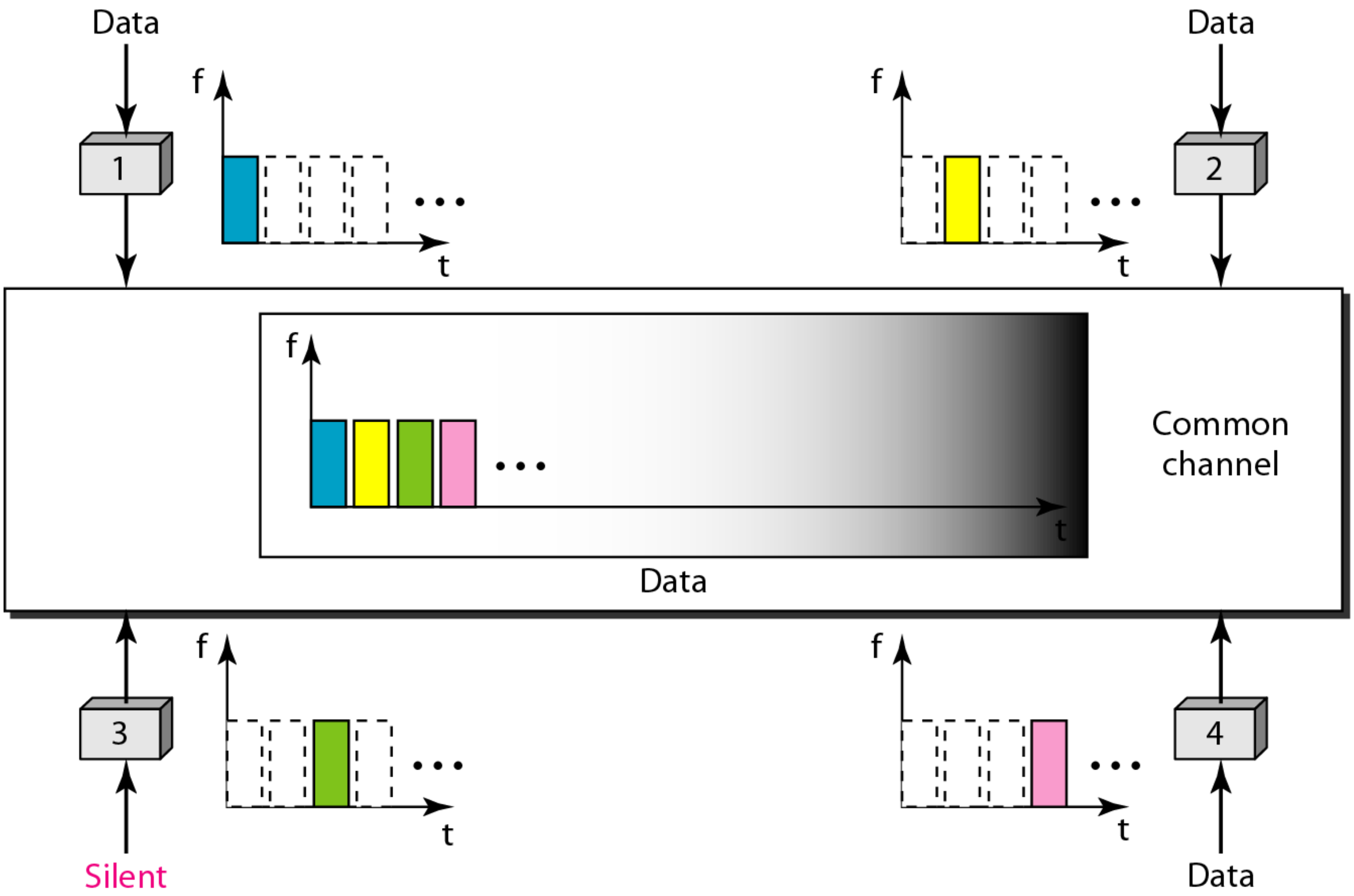
Figure 12.21 *Frequency-division multiple access (FDMA)*



Note

In FDMA, the available bandwidth of the common channel is divided into bands that are separated by guard bands.

Figure 12.22 *Time-division multiple access (TDMA)*



Note

In TDMA, the bandwidth is just one channel that is timeshared between different stations.

Note

In CDMA, one channel carries all transmissions simultaneously.

Figure 12.23 *Simple idea of communication with code*

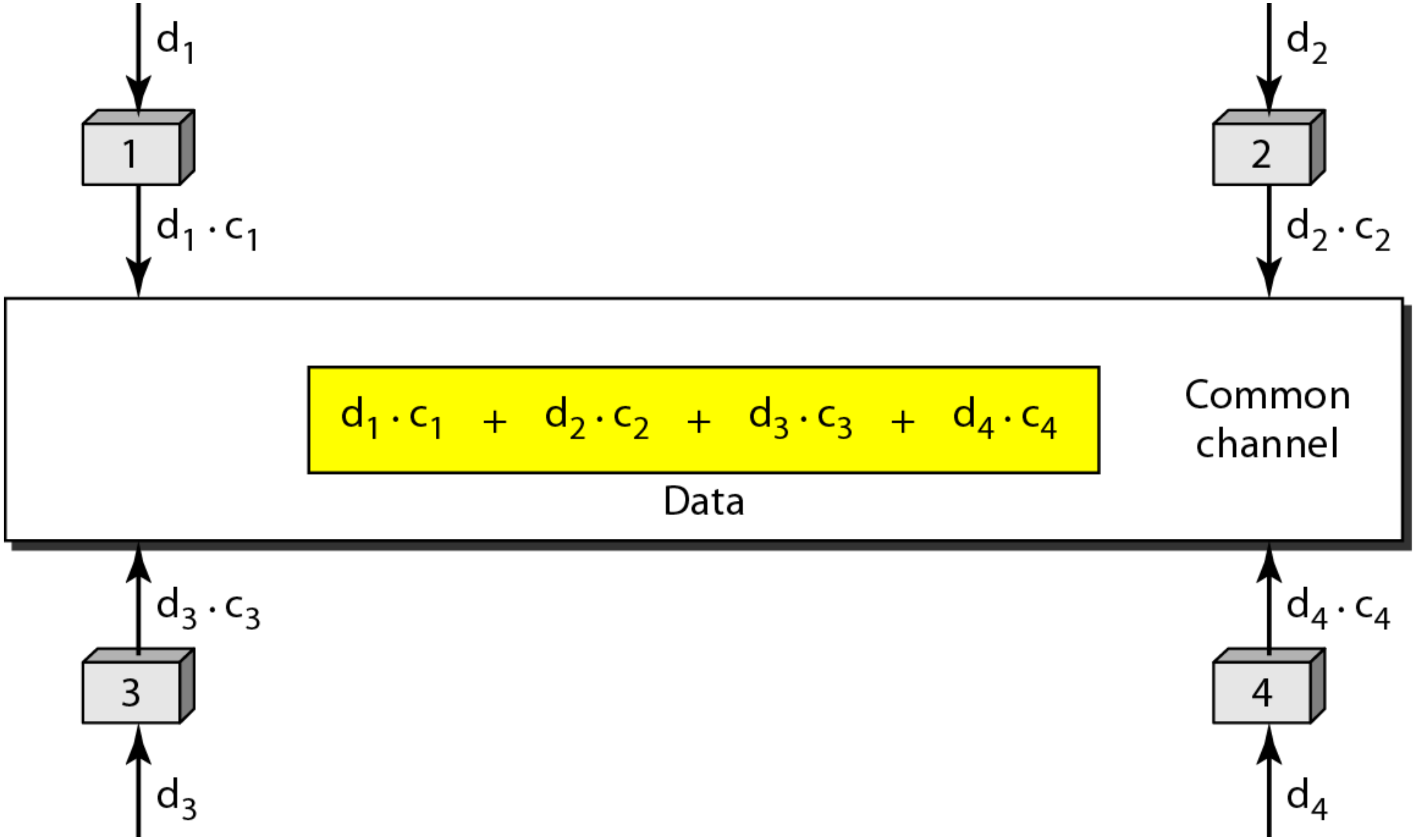
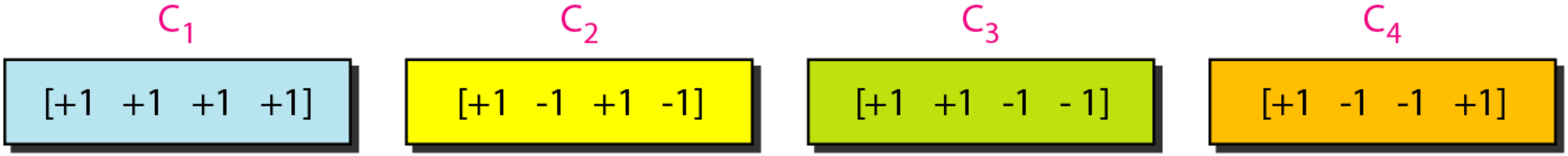


Figure 12.24 *Chip sequences*



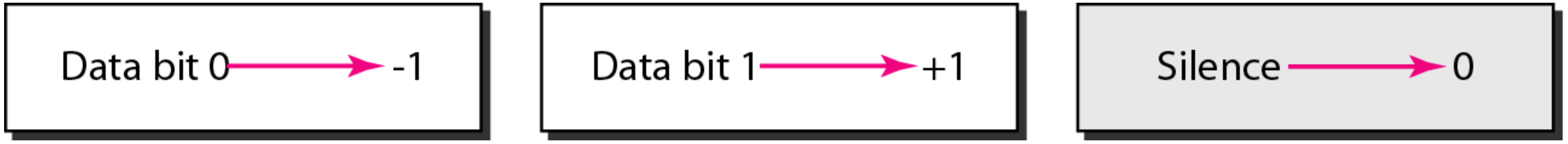
C_1 [+1 +1 +1 +1]

C_2 [+1 -1 +1 -1]

C_3 [+1 +1 -1 - 1]

C_4 [+1 -1 -1 +1]

Figure 12.25 *Data representation in CDMA*



Data bit 0 → -1

Data bit 1 → +1

Silence → 0

Figure 12.26 *Sharing channel in CDMA*

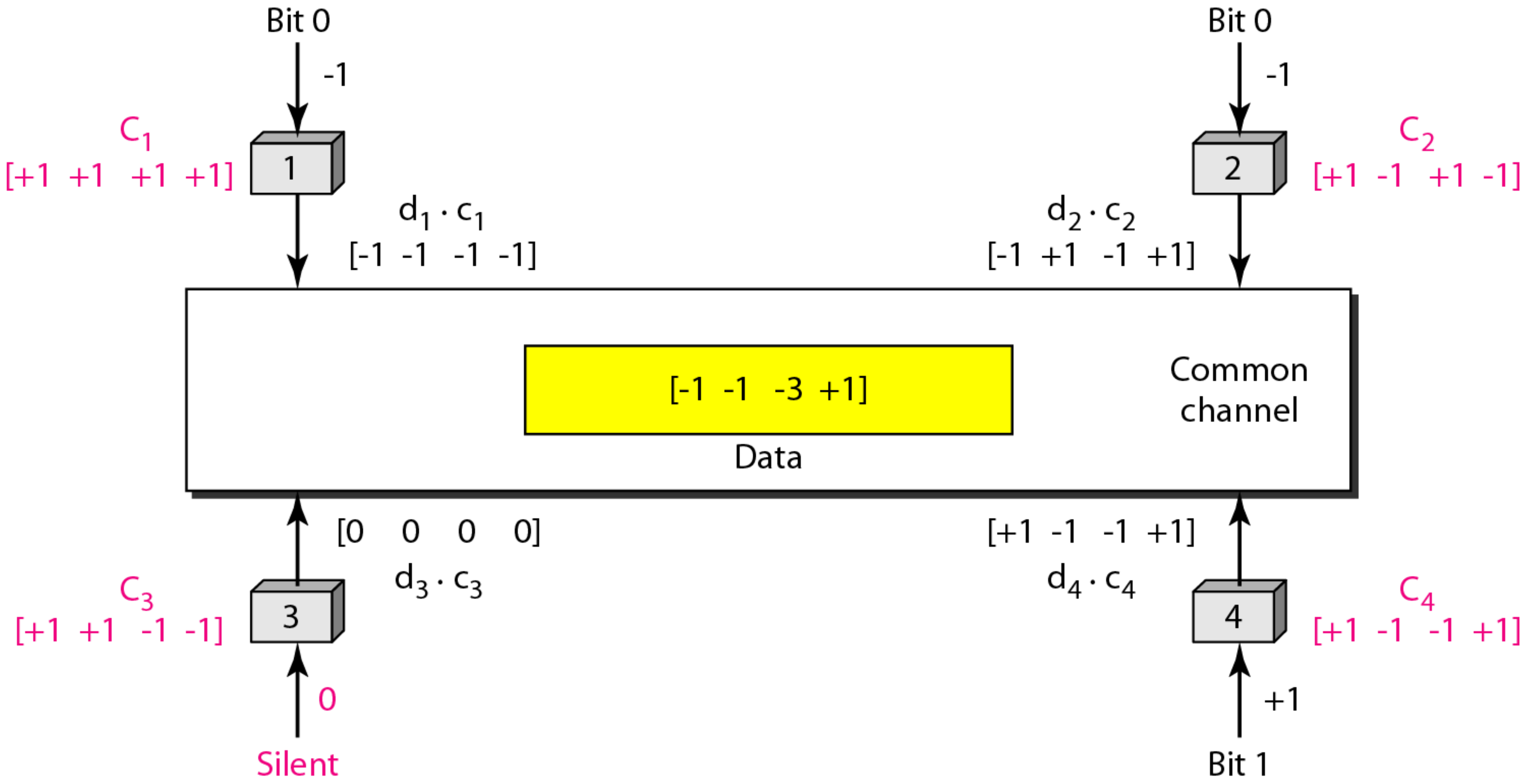


Figure 12.27 *Digital signal created by four stations in CDMA*

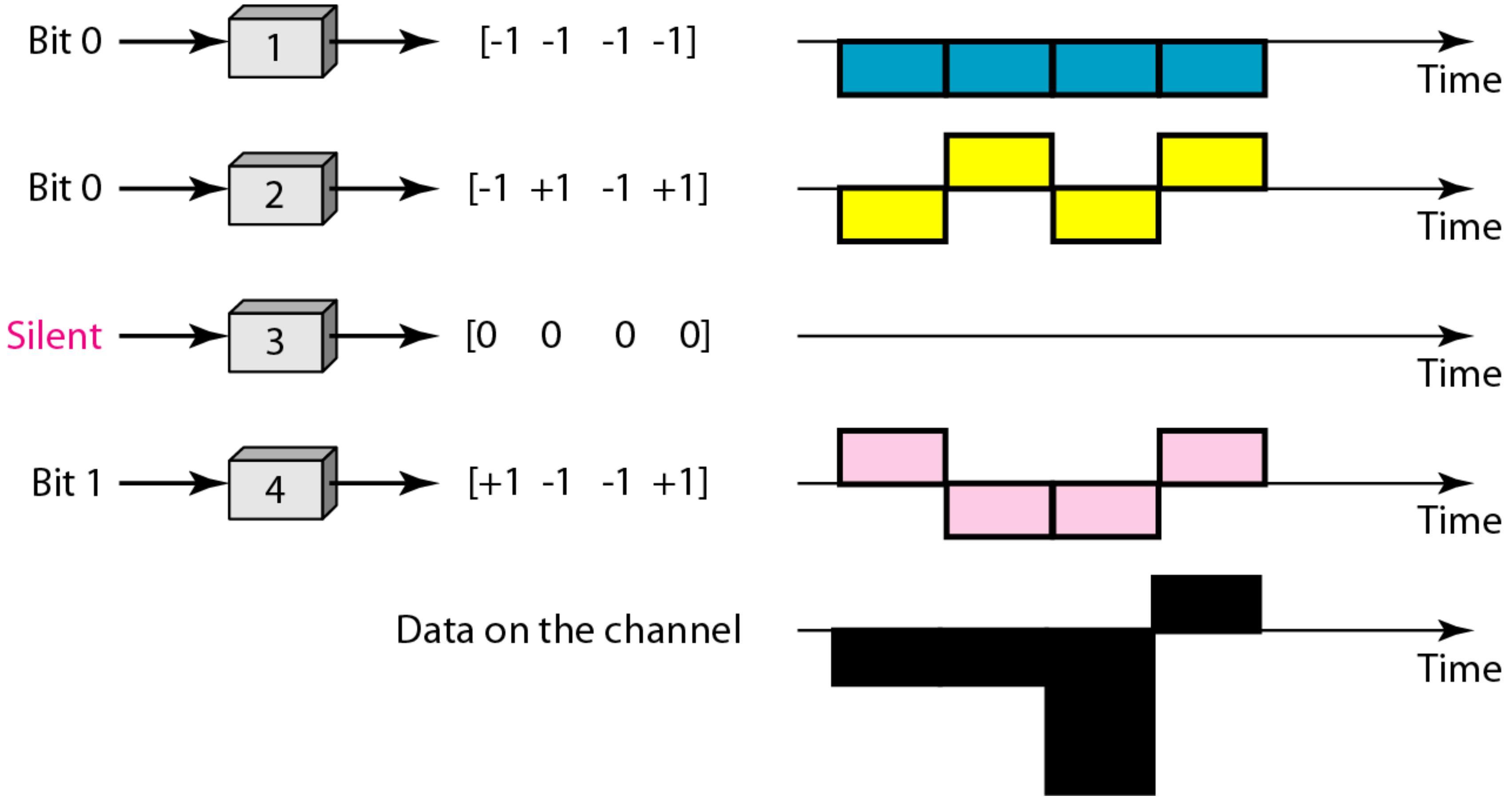


Figure 12.28 *Decoding of the composite signal for one in CDMA*

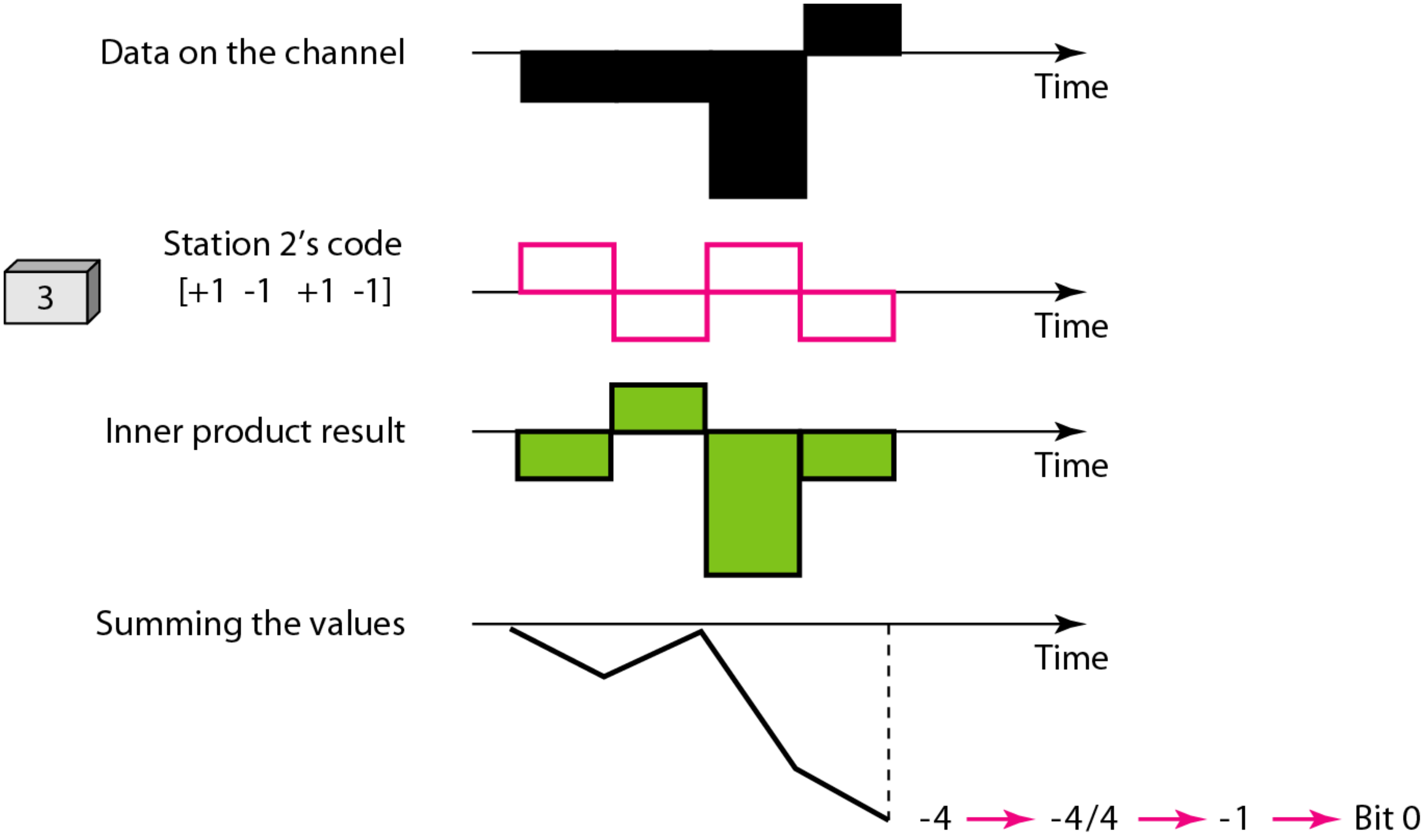


Figure 12.29 *General rule and examples of creating Walsh tables*

$$W_1 = \begin{bmatrix} +1 \end{bmatrix} \qquad\qquad W_{2N} = \begin{bmatrix} W_N & W_N \\ W_N & \overline{W_N} \end{bmatrix}$$

a. Two basic rules

$$W_1 = \begin{bmatrix} +1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \qquad W_4 = \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix}$$

b. Generation of W_1, W_2, and W_4

Note

The number of sequences in a Walsh table needs to be $N = 2^m$.

Example 12.6

Find the chips for a network with

a. Two stations ***b. Four stations***

Solution

We can use the rows of W_2 and W_4 in Figure 12.29:

a. For a two-station network, we have

[+1 +1] and [+1 −1].

b. For a four-station network we have

[+1 +1 +1 +1], [+1 −1 +1 −1],

[+1 +1 −1 −1], ***and*** **[+1 −1 −1 +1].**

Example 12.7

What is the number of sequences if we have 90 stations in our network?

Solution

The number of sequences needs to be 2^m. We need to choose m = 7 and N = 2^7 or 128. We can then use 90 of the sequences as the chips.

Example 12.8

Prove that a receiving station can get the data sent by a specific sender if it multiplies the entire data on the channel by the sender's chip code and then divides it by the number of stations.

Solution

Let us prove this for the first station, using our previous four-station example. We can say that the data on the channel

$$D = (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4).$$

The receiver which wants to get the data sent by station 1 multiplies these data by c_1.

Example 12.8 (continued)

$$\boxed{\begin{aligned} D \cdot c_1 &= (d_1 \cdot c_1 + d_2 \cdot c_2 + d_3 \cdot c_3 + d_4 \cdot c_4) \cdot c_1 \\ &= d_1 \cdot c_1 \cdot c_1 + d_2 \cdot c_2 \cdot c_1 + d_3 \cdot c_3 \cdot c_1 + d_4 \cdot c_4 \cdot c_1 \\ &= d_1 \times N + d_2 \times 0 + d_3 \times 0 + d_4 \times 0 \\ &= d_1 \times N \end{aligned}}$$

When we divide the result by N, we get d_1 .