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*
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
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

- **Frame sent**
  - Data from upper layer: 000111111001111101000
  - Stuffed: 0001111110110011111001000
  - Extra 2 bits

- **Frame received**
  - Header: 000111110110011111001000
  - Unstuffed: 000111111001111101000

- **Data to upper layer**
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
Flow control refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment.
Error control in the data link layer is based on automatic repeat request, which is the retransmission of data.
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*

- **Protocols**
  - For noiseless channel
    - Simplest
    - Stop-and-Wait
  - For noisy channel
    - Stop-and-Wait ARQ
    - Go-Back-N ARQ
    - Selective Repeat ARQ
Let us first assume we have an ideal channel in which no frames are lost, duplicated, or corrupted. We introduce two protocols for this type of channel.

Topics discussed in this section:
Simplest Protocol
Stop-and-Wait Protocol
Figure 11.6 The design of the simplest protocol with no flow or error control
Algorithm 11.1  Sender-site algorithm for the simplest protocol

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

11.16
Algorithm 11.2  Receiver-site algorithm for the simplest protocol

```c
1 while(true)  // Repeat forever
2 {
3     WaitForEvent();  // Sleep until an event occurs
4     if(Event(ArrivalNotification)) // Data frame arrived
5         {
6             ReceiveFrame();
7             ExtractData();
8             DeliverData();  // Deliver data to network layer
9         }
10    }
```
Example 11.1

Figure 11.7 shows an example of communication using this protocol. It is very simple. The sender sends a sequence of frames without even thinking about the receiver. To send three frames, three events occur at the sender site and three events at the receiver site. Note that the data frames are shown by tilted boxes; the height of the box defines the transmission time difference between the first bit and the last bit in the frame.
Figure 11.7  Flow diagram for Example 11.1
Figure 11.8  **Design of Stop-and-Wait Protocol**
Algorithm 11.3  Sender-site algorithm for Stop-and-Wait Protocol

1: while(true)       //Repeat forever
2:   canSend = true  //Allow the first frame to go
3:   
4:   WaitForEvent(); // Sleep until an event occurs
5:   if(Event(RequestToSend) AND canSend)
6:     
7:     GetData();
8:     MakeFrame();
9:     SendFrame();    //Send the data frame
10:    canSend = false;  //Cannot send until ACK arrives
11:  }
12:  WaitForEvent(); // Sleep until an event occurs
13:  if(Event(ArrivalNotification) // An ACK has arrived
14:    
15:    ReceiveFrame();  //Receive the ACK frame
16:    canSend = true;
17:  }
18: }
Algorithm 11.4  \textit{Receiver-site algorithm for Stop-and-Wait Protocol}

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

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

(continued)
Algorithm 11.5  \textit{Sender-site algorithm for Stop-and-Wait ARQ}  \textit{(continued)}

\begin{tabular}{|l|}
\hline
17 & if(Event(ArrivalNotification)) \hspace{1em} // An ACK has arrived \\
18 & \hspace{1em} \\
19 & \hspace{1em} ReceiveFrame(ackNo); \hspace{3em} // Receive the ACK frame \\
20 & \hspace{1em} if(not corrupted AND ackNo == S_n) \hspace{1em} // Valid ACK \\
21 & \hspace{1em} \hspace{1em} \\
22 & \hspace{1em} \hspace{1em} Stoptimer(); \\
23 & \hspace{1em} \hspace{1em} PurgeFrame(S_{n-1}); \hspace{3em} // Copy is not needed \\
24 & \hspace{1em} \hspace{1em} canSend = true; \\
25 & \hspace{1em} \hspace{1em} \hspace{1em} \\
26 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
27 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
28 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
29 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
30 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
31 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
32 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
33 & \hspace{1em} \hspace{1em} \hspace{1em} \} \\
\hline
\end{tabular}
Algorithm 11.6  Receiver-site algorithm for Stop-and-Wait ARQ Protocol

```java
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

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

<table>
<thead>
<tr>
<th>Frames already acknowledged</th>
<th>Frames sent, but not acknowledged (outstanding)</th>
<th>Frames that can be sent, but not received from upper layer</th>
<th>Frames that cannot be sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6</td>
<td>7 8 9 10 11 12 13 14</td>
<td>15 0 1</td>
<td>13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1</td>
</tr>
</tbody>
</table>

Send window, size $S_{size} = 2^m - 1$

a. Send window before sliding

b. Send window after sliding
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*

a. Receive window

b. Window after sliding
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

The figure illustrates the design of Go-Back-N Automatic Repeat reQuest (ARQ) in a network. The process involves the sender and receiver interacting via the network, data link, and physical layers.

- **Sender**:
  - Network layer requests data.
  - Data link layer processes the request and sends data frames.
  - Physical layer receives and sends frames.

- **Receiver**:
  - Network layer receives data.
  - Data link layer processes the data.
  - Physical layer receives and sends frames.

**Key Elements**:
- **$S_f$**: First outstanding frame.
- **$S_n$**: Next to send.
- **$R_n$**: Next to receive.
- **Data frame**: Sequenced and acknowledged.
- **ACK frame**: Acknowledges received frames.

The diagram shows the flow of data and frames between the layers, indicating how the ARQ protocol manages the transmission of data packets in a network.
Figure 11.15  Window size for Go-Back-N ARQ

a. Window size $< 2^m$

b. Window size $= 2^m$
In Go-Back-N ARQ, the size of the send window must be less than $2^m$; the size of the receiver window is always 1.
Algorithm 11.7  Go-Back-N sender algorithm

1 \( S_w = 2^m - 1; \)
2 \( S_f = 0; \)
3 \( S_n = 0; \)
4 
5 \text{while (true)} 
6 { 
7 \hspace{1em} \text{//Repeat forever} 
8 \hspace{1em} \text{WaitForEvent();} 
9 \hspace{1em} \text{//A packet to send} 
10 \hspace{1em} \text{if(Event(RequestToSend()))} 
11 \hspace{2em} \text{//If window is full} 
12 \hspace{2em} \{ 
13 \hspace{3em} \text{if}(S_n - S_f \geq S_w) 
14 \hspace{4em} \text{Sleep();} 
15 \hspace{4em} \text{GetData();} 
16 \hspace{4em} \text{MakeFrame}(S_n); 
17 \hspace{4em} \text{StoreFrame}(S_n); 
18 \hspace{4em} \text{SendFrame}(S_n); 
19 \hspace{3em} \} 
20 \hspace{2em} \text{if}(\text{timer not running}) 
21 \hspace{3em} \text{StartTimer();} 
22 \}

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

21 if(Event(ArrivalNotification)) //ACK arrives
22 {
23     Receive(ACK);
24     if(corrupted(ACK))
25         Sleep();
26     if((ackNo>S_f) && (ackNo<=S_n)) //If a valid ACK
27         While(S_f <= ackNo)
28             {
29                 PurgeFrame(S_f);
30                 S_f = S_f + 1;
31             }
32         StopTimer();
33 }
34
35 if(Event(TimeOut)) //The timer expires
36 {
37     StartTimer();
38     Temp = S_f;
39     while(Temp < S_n);
40         {
41             SendFrame(S_f);
42             S_f = S_f + 1;
43         }
44 }
45 }
Algorithm 11.8  Go-Back-N receiver algorithm

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

- $S_f$: Send window, first outstanding frame
- $S_n$: Send window, next frame to send

- Frames already acknowledged
- Frames sent, but not acknowledged
- Frames that can be sent
- Frames that cannot be sent

$S_{\text{size}} = 2^{m-1}$
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 > $2^{m-1}$
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*

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S_w = 2^{m-1}$ ;</td>
</tr>
<tr>
<td>2</td>
<td>$S_f = 0$ ;</td>
</tr>
<tr>
<td>3</td>
<td>$S_n = 0$ ;</td>
</tr>
<tr>
<td>4</td>
<td>[ \text{(continued)} ]</td>
</tr>
<tr>
<td>5</td>
<td>while (true) \quad //Repeat forever</td>
</tr>
<tr>
<td>6</td>
<td>{</td>
</tr>
<tr>
<td>7</td>
<td>\quad WaitForEvent() ;</td>
</tr>
<tr>
<td>8</td>
<td>\quad if(Event(RequestToSent)) \quad //There is a packet to send</td>
</tr>
<tr>
<td>9</td>
<td>\quad {</td>
</tr>
<tr>
<td>10</td>
<td>\quad \quad if(S_n-S_f &gt;= S_w) \quad //If window is full</td>
</tr>
<tr>
<td>11</td>
<td>\quad \quad \quad Sleep() ;</td>
</tr>
<tr>
<td>12</td>
<td>\quad \quad GetData() ;</td>
</tr>
<tr>
<td>13</td>
<td>\quad \quad MakeFrame(S_n) ;</td>
</tr>
<tr>
<td>14</td>
<td>\quad \quad StoreFrame(S_n) ;</td>
</tr>
<tr>
<td>15</td>
<td>\quad \quad SendFrame(S_n) ;</td>
</tr>
<tr>
<td>16</td>
<td>\quad \quad S_n = S_n + 1 ;</td>
</tr>
<tr>
<td>17</td>
<td>\quad \quad StartTimer(S_n) ;</td>
</tr>
<tr>
<td>18</td>
<td>\quad }</td>
</tr>
<tr>
<td>19</td>
<td>[ \text{(continued)} ]</td>
</tr>
</tbody>
</table>
Algorithm 11.9  *Sender-site Selective Repeat algorithm*  

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

1. \( R_n = 0; \)
2. \( \text{NakSent} = \text{false}; \)
3. \( \text{AckNeeded} = \text{false}; \)
4. Repeat(for all slots)
   5. \( \text{Marked(slot)} = \text{false}; \)
   
6. while (true)  //Repeat forever
7. {
8.   \( \text{WaitForEvent}(); \)
9. }
10. if(Event(ArrivalNotification))  //Data frame arrives
11. {
12.   \( \text{Receive(Frame)}; \)
13.   if(corrupted(Frame)) && (NOT NakSent)
14.     {
15.       \( \text{SendNACK}(R_n); \)
16.       \( \text{NakSent} = \text{true}; \)
17.       \( \text{Sleep}(); \)
18.     }
19.   if(seqNo <> R_n) && (NOT NakSent)
20.     {
21.       \( \text{SendNACK}(R_n); \)
22.     }
Algorithm 11.10  Receiver-site Selective Repeat algorithm

23 NakSent = true;
24 if ((seqNo in window) && (!Marked(seqNo))
25 {
26    StoreFrame(seqNo)
27    Marked(seqNo) = true;
28    while (Marked(Rn))
29    {
30        DeliverData(Rn);
31        Purge(Rn);
32        Rn = Rn + 1;
33        AckNeeded = true;
34    }
35    if (AckNeeded);
36    {
37        SendAck(Rn);
38        AckNeeded = false;
39        NakSent = false;
40    }
41  }
42}
43}
44}
Figure 11.22  Delivery of data in Selective Repeat ARQ

a. Before delivery

b. After delivery

ackNo sent: 3
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.
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.
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
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

I-frame

- Flag
- Address
- Control
- User information
- FCS
- Flag

S-frame

- Flag
- Address
- Control
- FCS
- Flag

U-frame

- Flag
- Address
- Control
- Management information
- FCS
- Flag
Figure 11.28 Control field format for the different frame types

- **I-frame**: 0, \( \text{N(S)} \), \( \text{P/F} \), \( \text{N(R)} \)
- **S-frame**: 1, 0, \( \text{P/F} \), \( \text{N(R)} \)
- **U-frame**: 1, 1, \( \text{P/F} \), \( \text{Code} \), \( \text{Code} \)
### Table 11.1  U-frame control command and response

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

- **Dead**
  - Carrier dropped
  - Failed
  - Carrier detected
  - Establish
    - Options agreed by both sides
    - Authenticate
      - Authentication successful
      - Network
        - Network layer configuration
        - Open
          - Done
          - Terminate
            - Failed
            - If authentication not needed
Figure 11.34 *Multiplexing in PPP*

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

LCP: 0xC021
AP: 0xC023 and 0xC223
NCP: 0x8021 and ....
Data: 0x0021 and ....
Figure 11.35  *LCP packet encapsulated in a frame*
Table 11.2  *LCP packets*

<table>
<thead>
<tr>
<th>Code</th>
<th>Packet Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Configure-request</td>
<td>Contains the list of proposed options and their values</td>
</tr>
<tr>
<td>0x02</td>
<td>Configure-ack</td>
<td>Accepts all options proposed</td>
</tr>
<tr>
<td>0x03</td>
<td>Configure-nak</td>
<td>Announces that some options are not acceptable</td>
</tr>
<tr>
<td>0x04</td>
<td>Configure-reject</td>
<td>Announces that some options are not recognized</td>
</tr>
<tr>
<td>0x05</td>
<td>Terminate-request</td>
<td>Request to shut down the line</td>
</tr>
<tr>
<td>0x06</td>
<td>Terminate-ack</td>
<td>Accept the shutdown request</td>
</tr>
<tr>
<td>0x07</td>
<td>Code-reject</td>
<td>Announces an unknown code</td>
</tr>
<tr>
<td>0x08</td>
<td>Protocol-reject</td>
<td>Announces an unknown protocol</td>
</tr>
<tr>
<td>0x09</td>
<td>Echo-request</td>
<td>A type of hello message to check if the other end is alive</td>
</tr>
<tr>
<td>0x0A</td>
<td>Echo-reply</td>
<td>The response to the echo-request message</td>
</tr>
<tr>
<td>0x0B</td>
<td>Discard-request</td>
<td>A request to discard the packet</td>
</tr>
</tbody>
</table>
Table 11.3  *Common options*

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum receive unit (payload field size)</td>
<td>1500</td>
</tr>
<tr>
<td>Authentication protocol</td>
<td>None</td>
</tr>
<tr>
<td>Protocol field compression</td>
<td>Off</td>
</tr>
<tr>
<td>Address and control field compression</td>
<td>Off</td>
</tr>
</tbody>
</table>
Figure 11.36  PAP packets encapsulated in a PPP frame
Figure 11.37 CHAP packets encapsulated in a PPP frame

CHAP packets

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>ID</th>
<th>Length</th>
<th>Challenge/Response length</th>
<th>Challenge/Response value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Challenge</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Response</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Success</td>
<td>3</td>
<td></td>
<td></td>
<td>Message</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>4</td>
<td></td>
<td></td>
<td>Message</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Code</th>
<th>IPCP Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Configure-request</td>
</tr>
<tr>
<td>0x02</td>
<td>Configure-ack</td>
</tr>
<tr>
<td>0x03</td>
<td>Configure-nak</td>
</tr>
<tr>
<td>0x04</td>
<td>Configure-reject</td>
</tr>
<tr>
<td>0x05</td>
<td>Terminate-request</td>
</tr>
<tr>
<td>0x06</td>
<td>Terminate-ack</td>
</tr>
<tr>
<td>0x07</td>
<td>Code-reject</td>
</tr>
</tbody>
</table>
Figure 11.39  *IP datagram encapsulated in a PPP frame*
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.
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)*