COMPUTER NETWORKS

Author of Textbook: ANDREW S. TANENBAUM

3 THE DATA LINK LAYER

4 THE MEDIUM ACCESS CONTROL SUBLAYER

DATA LINK LAYER

3.1 DATA LINK LAYER DESIGN ISSUES

The data link layer uses the services of the physical layer to send and receive bits over communication channels.

It has a number of functions, including:

1. Providing a well-defined service interface to the network layer.
2. Dealing with transmission errors.
3. Regulating the flow of data so that slow receivers are not swamped by fast senders.

To accomplish these goals, the data link layer takes the packets it gets from the network layer and encapsulates them into frames for transmission.

- Each frame contains a frame header, a payload field for holding the packet, and a frame trailer, as illustrated in Fig. 3-1.

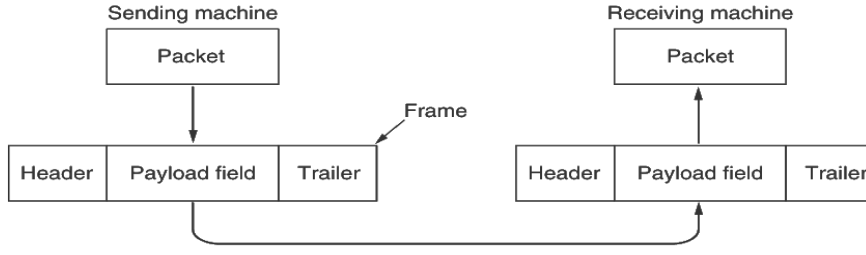


Figure 3-1. Relationship between packets and frames.

- Frame management forms the heart of the data link layer.

3.1.1 Services Provided to the Network Layer

The function of the data link layer is to provide services to the network layer. The principal service is transferring data from the network layer on the source machine to the network layer on the destination machine.

On the source machine is an entity, call it a process, in the network layer that hands some bits to the data link layer for transmission to the destination. The job of the data link layer is to transmit the bits to the destination machine so they can be handed over to the network layer there, as shown in Fig. 3-2(a). The actual transmission follows the path of Fig. 3-2(b), but it is easier to think in terms of two data link layer processes communicating using a data link protocol.

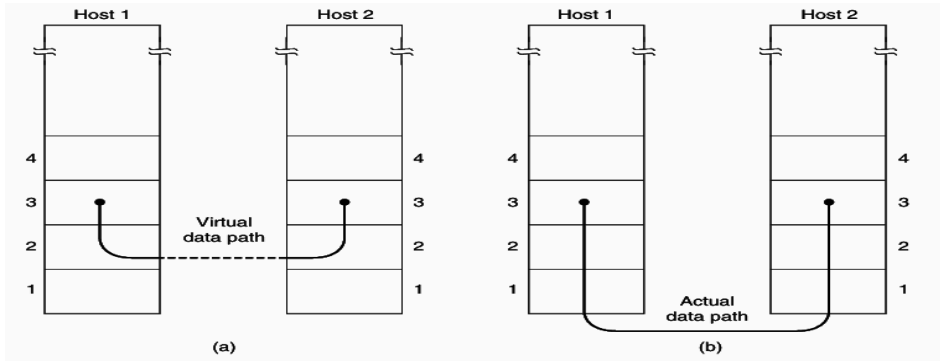


Figure 3-2. (a) Virtual communication. (b) Actual communication.

The data link layer can be designed to offer various services. The actual services that are offered vary from protocol to protocol.

Three reasonable possibilities that are considered are:

1. **Unacknowledged connectionless service.**
2. **Acknowledged connectionless service.**
3. **Acknowledged connection-oriented service.**

Unacknowledged connectionless service consists of having the source machine send independent frames to the destination machine *without having the destination machine acknowledge them.*

- Ethernet is a good example of a data link layer that provides this class of service.
- No logical connection is established beforehand or released afterward. If a frame is lost due to noise on the line, no attempt is made to detect the loss or recover from it in the data link layer.
- This class of service is appropriate when the error rate is very low, so recovery is left to higher layers.

Acknowledged connectionless service

To achieve reliability, acknowledged connectionless service is used. When this service is offered, there are no logical connections used, but each frame sent is individually acknowledged.

- In this way, the sender knows whether a frame has arrived correctly or been lost. If it has not arrived within a specified time interval, it can be sent again.

- This service is useful over unreliable channels, such as wireless systems. 802.11 (WiFi) is a good example of this class of service.

Connection-oriented service

With this service, the source and destination machines establish a connection before any data are transferred.

- Each frame sent over the connection is numbered, and the data link layer guarantees that each frame sent is indeed received.
- Furthermore, it guarantees that each frame is received exactly once and that all frames are received in the right order.
- When connection-oriented service is used, transfers go through three distinct phases.
  1 First the connection is established by having both sides initialize variables and counters needed to keep track of which frames have been received and which ones have not.
  2 In this phase, one or more frames are actually transmitted.
  3 Finally the connection is released, freeing up the variables, buffers, and other resources used to maintain the connection.

3.1.2 Framing

The data link layer breaks the bit stream into discrete frames and computes a short token called a checksum for each frame, and include the checksum in the frame when it is transmitted.

When a frame arrives at the destination, the checksum is recomputed. If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has occurred and takes steps to deal with it.

- Breaking up the bit stream into frames is more difficult.
- A good design must make it easy for a receiver to find the start of new frames while using little of the channel bandwidth.

There are four methods:

1. Byte count.
2. Flag bytes with byte stuffing.
3. Flag bits with bit stuffing.

4. Physical layer coding violations.

Byte count.

This framing method uses a field in the header to specify the number of bytes in the frame.

- When the data link layer at the destination sees the byte count, it knows how many bytes follow and hence where the end of the frame is.
- This technique is shown in Fig. 3-3(a) for four small example frames of sizes 5, 5, 8, and 8 bytes, respectively.

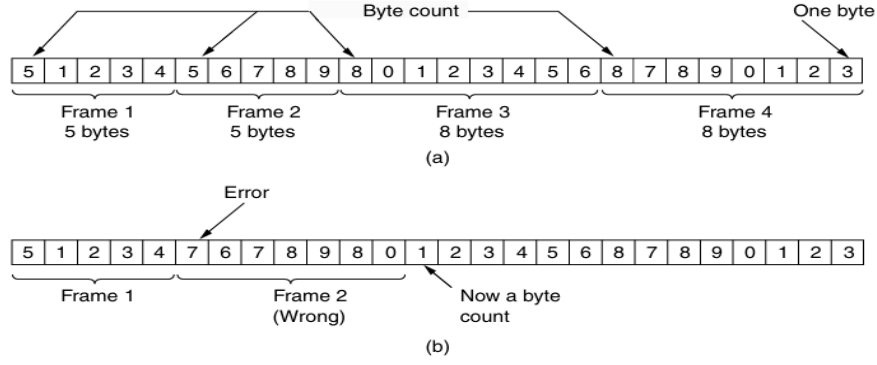
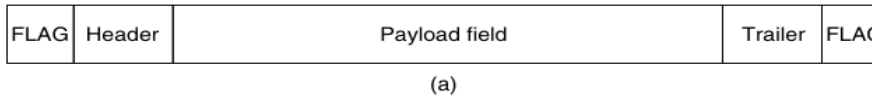


Figure 3-3. A byte stream. (a) Without errors. (b) With one error.

Flag bytes with byte stuffing

The second framing method gets around the problem of resynchronization after an error by having *each frame start and end with special bytes.*

- Often the same byte, called a flag byte, is used as both the starting and ending delimiter. This byte is shown in Fig. 3-4(a) as FLAG.
- Two consecutive flag bytes indicate the end of one frame and the start of the next. Thus, if the receiver ever loses synchronization it can just search for two flag bytes to find the end of the current frame and the start of the next frame.



(a)

Figure 3-4. (a) A frame delimited by flag bytes.

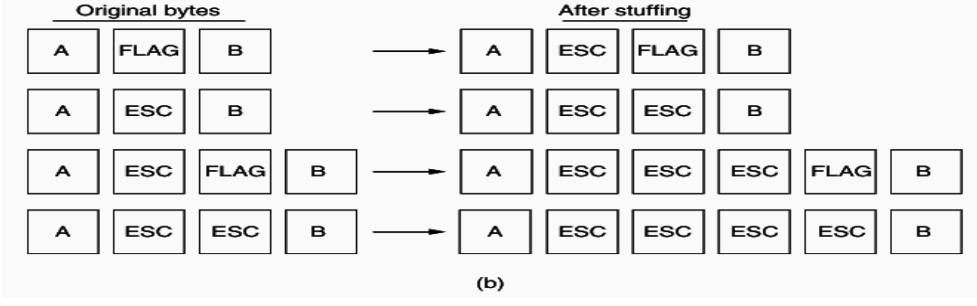
Sometimes there may happen that the flag byte occurs in the data, especially when binary data such as photographs or songs are being transmitted. This situation would interfere with the framing.

- One way to solve this problem is to have the sender's data link layer insert a special escape byte (ESC) just before each ''accidental'' flag byte in the data. Thus, a framing flag byte can be distinguished from one in the data by the absence or presence of an escape byte before it.
- The data link layer on the receiving end removes the escape bytes before giving the data to the network layer. This technique is called byte stuffing.

But what if an escape byte occurs in the middle of the data?

- The answer is that it, too, is stuffed with an escape byte.
- At the receiver, the first escape byte is removed, leaving the data byte that follows it (which might be another escape byte or the flag byte).
- Some examples are shown in Fig. 3-4(b).

- In all cases, the byte sequence delivered after destuffing is exactly the same as the original byte sequence.



(b) Four examples of byte sequences before and after byte stuffing

Flag bits with bit stuffing.

The third method of *delimiting the bit stream* gets around a disadvantage of byte stuffing, which is that it is tied to the use of 8-bit bytes. Framing can be done at the bit level, so frames can contain an arbitrary number of

bits made up of units of any size. Each frame begins and ends with a special bit pattern, 01111110. This pattern is a flag byte.

- Whenever the sender's data link layer encounters five consecutive1 in the data, it automatically stuffs a 0 bit into the outgoing bit stream.
- When the receiver *sees five consecutive incoming 1 bits, followed by a 0 bit*, it automatically destuffs (i.e., deletes) the 0 bit.

- If the user data contain the flag pattern, 01111110, this flag is transmitted as 011111010 but stored in the receiver's memory as 01111110.

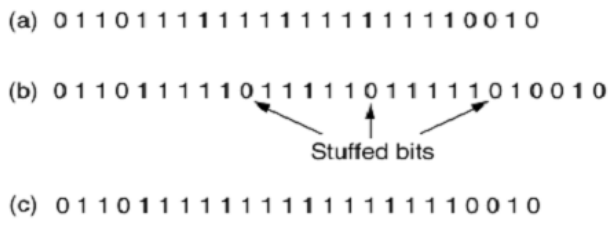


Figure 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuffing.

3.1.3 Error Control

The usual way to ensure reliable delivery is to provide the sender with some feedback about what is happening at the other end of the line. Typically, the protocol calls for the receiver to send back special control frames bearing positive or negative acknowledgements about the incoming frames.

If the sender receives a positive acknowledgement about a frame, it knows the frame has arrived safely. On the other hand, a negative acknowledgement means that something has gone wrong and the frame must be transmitted again.

An additional complication comes from the possibility that hardware troubles may cause a frame to vanish completely (e.g., in a noise burst). In this case, the receiver will not react at all, since it has no reason to react. Similarly, if the acknowledgement frame is lost, the sender will not know how to proceed. It should be clear that a protocol in which the

sender transmits a frame and then waits for an acknowledgement, positive or negative, will hang forever if a frame is ever lost due to, for example, malfunctioning hardware or a faulty communication channel.

This possibility is dealt with by introducing timers into the data link layer. When the sender transmits a frame, it generally also starts a timer. The timer is set to expire after an interval long enough for the frame to reach the destination, be processed there, and have the acknowledgement propagate back to the sender. Normally, the frame will be correctly received and the acknowledgement will get back before the timer runs out, in which case the timer will be canceled.

However, if either the frame or the acknowledgement is lost, the timer will go off, alerting the sender to a potential problem. The obvious solution is to just transmit the frame again. However, when frames may be transmitted multiple times there is a danger that the receiver will accept the same frame two or more times and pass it to the network layer more than once.

To prevent this from happening, it is generally necessary to assign sequence numbers to outgoing frames, so that the receiver can distinguish retransmissions from originals.

3.1.4 Flow Control

Another important design issue that occurs is what to do with a sender that systematically wants to transmit frames faster than the receiver can accept them. This situation can occur when the sender is running on a fast, powerful computer and the receiver is running on a slow, low-end machine.

A common situation is when a smart phone requests a Web page from a far more powerful server, which then turns on the fire hose and blasts the data at the poor helpless phone until it is completely swamped. Even if the transmission is error free, the receiver may be unable to handle the frames as fast as they arrive and will lose some.

3.2 ERROR DETECTION AND CORRECTION

Network designers have developed two basic strategies for dealing with errors. Both add redundant information to the data that is sent.

- One strategy is to include enough redundant information to enable the receiver to deduce what the transmitted data must have been.
- The other is to include only enough redundancy to allow the receiver to deduce that an error has occurred (but not which error) and have it request a retransmission.

The former strategy uses error-correcting codes and the latter uses error-detecting codes. The use of error-correcting codes is often referred to as FEC (Forward Error Correction)

On channels that are highly reliable, such as fiber, it is cheaper to use an error-detecting code and just retransmit the occasional block found to be faulty. On channels such as wireless links that make many errors, it is better to add redundancy to each block so that the receiver is able to figure out what the originally transmitted block was.

FEC is used on noisy channels because retransmissions are just as likely to be in error as the first transmission. A key consideration for these codes is the type of errors that are likely to occur. Neither error-correcting codes nor error-detecting codes can handle all possible errors since the redundant bits that offer protection are as likely to be received in error as the data bits (which can compromise their protection).

3.2.1 Error-Correcting Codes

Four different error-correcting codes:

1. Hamming codes.
2. Binary convolutional codes.
3. Reed-Solomon codes.
4. Low-Density Parity Check codes.

All of these codes add redundancy to the information that is sent. A frame consists of m data (i.e., message) bits and r redundant (i.e. check) bits.

- In a block code, the r check bits are computed solely as a function of the m data bits with which they are associated, as though the m

bits were looked up in a large table to find their corresponding r check bits.

- In a systematic code, the m data bits are sent directly, along with the check bits, rather than being encoded themselves before they are sent.
- In a linear code, the r check bits are computed as a linear function of the m data bits. Exclusive OR (XOR) or modulo 2 addition is a popular choice. This means that encoding can be done with operations such as matrix multiplications or simple logic circuits.

Let the total length of a block be n (i.e., n = m + r). We will describe this as an (n,m) code.

An n-bit unit containing data and check bits is referred to as an n bit codeword. The code rate is the fraction of the codeword that carries information that is not redundant, or m/n. The rates used in practice vary widely. They might be 1/2 for a noisy channel, in which case half of the received information is redundant, or close to 1 for a high-quality channel, with only a small number of check bits added to a large message.

To understand how errors can be handled, it is necessary to first look closely at what an error really is.

- Given any two codewords that may be transmitted or received—say, 10001001 and 10110001—it is possible to determine how many corresponding bits differ.
- In this case, 3 bits differ. To determine how many bits differ, just XOR the two codewords and count the number of 1 bits in the result.

For example:

```
10001001
10110001
00111000
```

The number of bit positions in which two codewords differ is called the Hamming distance.

Given the algorithm for computing the check bits, it is possible to construct a complete list of the legal codewords, and from this list to find

the two codewords with the smallest Hamming distance. This Distance is the Hamming distance of the complete code.

The Error-detecting and error-correcting properties of a block code depend on its Hamming distance.

- To reliably detect d errors, you need a distance d+1 code because with such a code there is no way that single-bit errors can change a valid codeword into another valid codeword. When the receiver sees an illegal codeword, it can tell that a transmission error has occurred.
- Similarly, to correct d errors, you need a distance 2d+1 code because that way the legal codewords are so far apart that even with d changes the original codeword stil closer than any other codeword.

As a simple example of an error-correcting code, consider a code with only four valid codewords:

0000000000, 0000011111, 1111100000, and 1111111111

This code has a distance of 5, which means that it can correct double errors or detect quadruple errors.

In our example, the task of decoding by finding the legal codeword that is closest to the received codeword can be done by inspection. Unfortunately, in the most general case where all codewords need to be evaluated as candidates, this task can be a time-consuming search.

Imagine that we want to design a code with m message bits and r check bits that will allow all single errors to be corrected. Each of the 2^m legal messages has n illegal codewords at a distance of 1 from it. These are formed by systematically inverting each of the n bits in the n-bit codeword formed from it. Thus, each of the 2^m legal messages requires n + 1 bit patterns dedicated to it.

- Since the total number of bit patterns is 2^n, we must have $(n + 1)2^m \leq 2^n$. Using n = m + r, this requirement becomes $(m + r + 1) \leq 2^r$

- Given m, this puts a lower limit on the number of check bits needed to correct single errors.

In Hamming codes the bits of the codeword are numbered consecutively, starting with bit 1 at the left end, bit 2 to its immediate right, and so on. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the m data bits. This pattern is shown for an (11,7) Hamming code with 7 data bits and 4 check bits in Fig. 3-6.

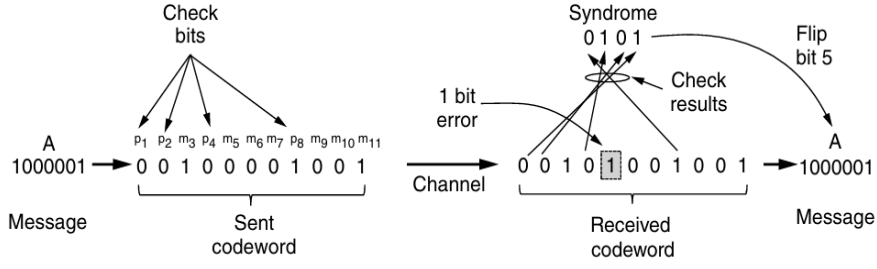


Figure 3-6. Example of an (11, 7) Hamming code correcting a single-bit error.

Each check bit forces the modulo 2 sum, or parity, of some collection of bits, including itself, to be even (or odd).

A bit may be included in several check bit computations. To see which check bits the data bit in position k contributes to, rewrite k as a sum of powers of 2. For example, 11 = 1 + 2 + 8 and 29 = 1 + 4 + 8 + 16. A bit is checked by just those check bits occurring in its expansion (e.g., bit 11 is checked by bits 1, 2, and 8).

In the example, the check bits are computed for even parity sums for a message that is the ASCII letter ‘‘A’’. This construction gives a code with a Hamming distance of 3, which means that it can correct single errors (or detect double errors). When a codeword arrives, the receiver redoes the check bit computations including the values of the received check bits. We call these the check results.

- If the check bits are correct then, for even parity sums, each check result should be zero. In this case the codeword is accepted as valid.
- If the check results are not all zero, however, an error has been detected. The set of check results forms the error syndrome that is used to pinpoint and correct the error.

In Fig. 3-6, a single-bit error occurred on the channel so the check results are 0, 1, 0, and 1 for k = 8, 4, 2, and 1, respectively. This gives a syndrome of 0101 or 4 + 1=5. By the design of the scheme, this means that the fifth bit is in error. Flipping the incorrect bit (which might be a check bit or a data bit) and discarding the check bits gives the correct message of an ASCII ‘‘A.’’

3.2.2 Error-Detecting Codes

Error-correcting codes are widely used on wireless links, which are notoriously noisy and error prone when compared to optical fibers.

There are three different error-detecting codes. They are all linear, systematic block codes:

1. **Parity.**
2. **Checksums.**
3. **Cyclic Redundancy Checks (CRCs).**

To see how they can be more efficient than error-correcting codes, consider the first error-detecting code, in which a single parity bit is appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd). Doing this is equivalent to computing the (even) parity bit as the modulo 2 sum or XOR of the data bits.

For example, when 1011010 is sent in even parity, a bit is added to the end to make it 10110100. With odd parity 1011010 becomes 10110101.

A code with a single parity bit has a distance of 2, since any single-bit error produces a codeword with the wrong parity. This means that it can detect single-bit errors.

Consider a channel on which errors are isolated and the error rate is 10^{-6} per bit. This is a tiny error rate, but it is at best a fair rate for a long wired cable that is challenging for error detection. Typical LAN links provide bit error rates of $10^{-10.}$ Let the block size be 1000 bits. To provide error correction for 1000-bit blocks, that 10 check bits are needed. Thus, a megabit of data would require 10,000 check bits.

To detect a block with a single 1-bit error, one parity bit per block will suffice. Once every 1000 blocks, a block will be found to be in error and an extra block (1001 bits) will have to be transmitted to repair the error. The total overhead for the error detection and re transmission method is

only 2001 bits per megabit of data, versus 10,000 bits for a Hamming code.One difficulty with this scheme is that a single parity bit can only reliably detect a single-bit error in the block. If the block is badly garbled by a long burst error, the probability that the error will be detected is only 0.5, which is hardly acceptable. The odds can be improved considerably if each block to be sent is regarded as a rectangular matrix n bits wide and k bits high. Now, if we compute and send one parity bit for each row, up to k bit errors will be reliably detected as long as there is at most one error per row.

To get the better protection against burst errors: we can compute the parity bits over the data in a different order than the order in which the data bits are transmitted. This is called interleaving. In this case, we will compute a parity bit for each of the n columns and send all the data bits as k rows, sending the rows from top to bottom and the bits in each row from left to right in the usual manner. At the last row, we send the n parity bits. This transmission order is shown in Fig. 3-8 for n = 7 and k = 7.

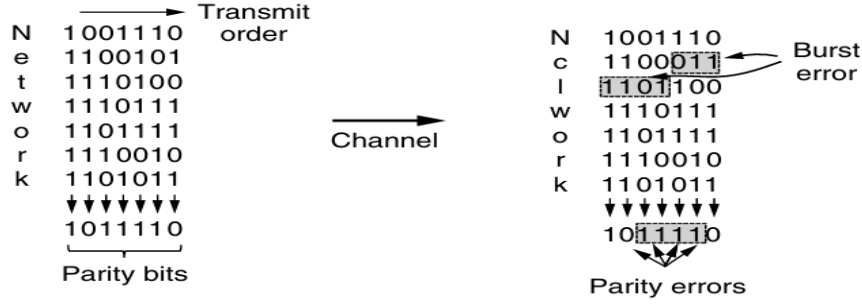


Figure 3-8. Interleaving of parity bits to detect a burst error.

Interleaving is a general technique to convert a code that detects (or corrects) isolated errors into a code that detects (or corrects) burst errors.

In Fig.3-8,when a burst error of length n=7 occurs, the bits that are in error are spread across different columns.

(A burst error does not imply that all the bits are wrong; it just implies that at least the first and last are wrong. In Fig.3-8, 4 bits were flipped over a range of 7 bits.) Atmost1bit in each of the n columns will be affected, so the parity bits on those columns will detect the error. This

method uses n parity bits on blocks of kn data bits to detect a single burst error of length or less.

Checksum

The second kind of error-detecting code, the checksum, is closely related to groups of parity bits. The word ‘‘checksum’’ is often used to mean a group of check bits associated with a message, regardless of how are calculated.

A group of parity bits is one example of checksum.However, there are other, stronger checksums based on a running sum of the data bits of the message. The checksum is usually placed at the end of the message, as the complement of the sum function. This way, errors may be detected by summing the entire received codeword, both data bits and checksum. If the result comes out to be zero, no error has been detected.

3.3 ELEMENTARY DATA LINK PROTOCOLS

To start with, we assume that the physical layer, data link layer, and network layer are independent processes that communicate by passing messages back and forth. A common implementation is shown in Fig. 3-10.

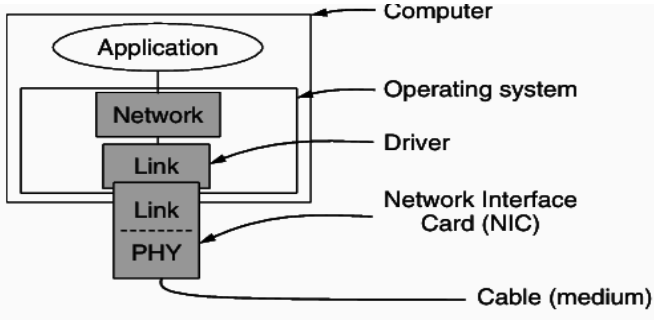


Figure 3-10. Implementation of the physical, data link, and network layers.

The physical layer process and some of the data link layer process run on dedicated hardware called a NIC (Network Interface Card). The rest of the link layer process and the network layer process run on the main CPU

as part of the operating system, with the software for the link layer process often taking the form of a device driver

Some definitions needed in the protocols to follow. These definitions are located in the file protocol.h

Five data structures are defined there**:**

boolean, seq_nr, packet, frame_kind, and frame.

1 A boolean is an enumerated type and can take values as true or false.
2 A seq nr is a small integer used to number the frames so that we can tell them apart.

```
#define MAX_PKT 1024                                    /* determines packet size in bytes */

typedef enum {false, true} boolean;                     /* boolean type */
typedef unsigned int seq_nr;                            /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet;   /* packet definition */
typedef enum {data, ack, nak} frame_kind;               /* frame_kind definition */

typedef struct {                                        /* frames are transported in this layer */
  frame_kind kind;                                      /* what kind of frame is it? */
  seq_nr seq;                                           /* sequence number */
  seq_nr ack;                                           /* acknowledgement number */
  packet info;                                          /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);
```

```
/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

These sequence numbers run from 0 up to and including MAX SEQ, which is defined in each protocol needing it.

3 A packet is the unit of information exchanged between the network layer and the data link layer on the same machine, or between network layer peers. In our model it always contains MAX PKT bytes, but more realistically it would be of variable length.

4 A frame is composed of four fields: kind, seq, ack, and info, the first three of which contain control information and the last of which may contain actual data to be transferred. These control fields are collectively called the frame header. The kind field tells whether there are any data in the frame, because some of the protocols distinguish frames containing only control information from those containing data as well. The seq and ack fields are used for sequence numbers and acknowledgements, respectively; The info field of a data frame contains a single packet.

3.3.1 A Utopian Simplex Protocol

Consider a simple protocol that does not worry about the possibility of anything going wrong.

- Data are transmitted in one direction only. Both the transmitting and receiving network layers are always ready.
- Processing time can be ignored. Infinite buffer space is available. The communication channel between the data link layers never damages or loses frames.

- It's implementation is shown in Fig. 3-12.

The protocol consists of two distinct procedures, a sender and a receiver. The sender runs in the data link layer of the source machine, and the receiver runs in the data link layer of the destination machine. No sequence numbers or acknowledgements are used here, so MAX SEQ is not needed. The only event type possible is frame arrival (i.e., the arrival of an undamaged frame).

The sender is in an infinite while loop just pumping data out onto the line as fast as it can. The body of the loop consists of three actions:

1. Go fetch a packet from the network layer,
2. Construct an outbound frame using the variable s, and
3. Send the frame on its way.

```
/* Protocol 1 (Utopia) provides for data transmission in one direction only, from
   sender to receiver. The communication channel is assumed to be error free
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
  frame s;                              /* buffer for an outbound frame */
  packet buffer;                        /* buffer for an outbound packet */

  while (true) {
      from_network_layer(&buffer);      /* go get something to send */
      s.info = buffer;                  /* copy it into s for transmission */
      to_physical_layer(&s);            /* send it on its way */
  }                                     /* Tomorrow, and tomorrow, and tomorrow,
                                           Creeps in this petty pace from day to day
                                           To the last syllable of recorded time.
                                               – Macbeth, V, v */
}

 void receiver1(void)
 {
   frame r;
   event_type event;                    /* filled in by wait, but not used here */

   while (true) {
       wait_for_event(&event);          /* only possibility is frame_arrival */
       from_physical_layer(&r);         /* go get the inbound frame */
       to_network_layer(&r.info);       /* pass the data to the network layer */
   }
 }
```

Figure 3-12. A utopian simplex protocol.

Only the info field of the frame is used by this protocol, because the other fields have to do with error and flow control and there are no errors or flow control restrictions here. The receiver is equally simple. Initially, it waits for something to happen, the only possibility being the arrival of an undamaged frame. Eventually, the frame arrives and the procedure wait for event returns, with event set to frame arrival (which is ignored anyway). The call to from physical layer removes the newly arrived frame from the hardware buffer and puts it in the variable r, where the receiver code can get at it. Finally, the data portion is passed on to the network layer, and the data link layer settles back to wait for the next frame, effectively suspending itself until the frame arrives.

3.3.2 A Simplex Stop-and-Wait Protocol for an Error-Free Channel

To tackle the problem of preventing the sender from flooding the receiver with frames faster than the latter is able to process them.

One solution is to build the receiver to be powerful enough to process a continuous stream of back-to-back frames. It must have sufficient buffering and processing abilities to run at the line rate and must be able to pass the frames that are received to the network layer quickly enough. However, this is a worst-case solution. It requires dedicated hardware and can be wasteful of resources if the utilization of the link is mostly low. Moreover, it just shifts the problem of dealing with a sender that is too fast elsewhere; in this case to the network layer.

A more general solution to this problem is to have the receiver provide feedback to the sender. After having passed a packet to its network layer, the receiver sends a little dummy frame back to the sender which, in effect, gives the sender permission to transmit the next frame. After having sent a frame, the sender is required by the protocol to bide its time until the little dummy (i.e., acknowledgement) frame arrives. This delay is a simple example of a flow control protocol. Protocols in which the sender sends one frame and then waits for an acknowledgement before proceeding are called stop-and-wait. Figure 3-13 gives an example of a simplex stop-and-wait protocol.

```
typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender2(void)
{
  frame s;                              /* buffer for an outbound frame */
  packet buffer;                        /* buffer for an outbound packet */
  event_type event;                     /* frame_arrival is the only possibility */

  while (true) {
      from_network_layer(&buffer);      /* go get something to send */
      s.info = buffer;                  /* copy it into s for transmission */
      to_physical_layer(&s);            /* bye-bye little frame */
      wait_for_event(&event);           /* do not proceed until given the go ahead */
  }
}

void receiver2(void)
{
  frame r, s;                           /* buffers for frames */
  event_type event;                     /* frame_arrival is the only possibility */
  while (true) {
      wait_for_event(&event);           /* only possibility is frame_arrival */
      from_physical_layer(&r);          /* go get the inbound frame */
      to_network_layer(&r.info);        /* pass the data to the network layer */
      to_physical_layer(&s);            /* send a dummy frame to awaken sender */
  }
}
```

Figure 3-13. A simplex stop-and-wait protocol.

Consider the following scenario:

1. The network layer on A gives packet 1 to its data link layer. The packet is correctly received at B and passed to the network layer on B. B sends an acknowledgement frame back to A.
2. The acknowledgement frame gets lost completely. It just never arrives at all.
3. The data link layer on A eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frame was lost or damaged and sends the frame containing packet 1 again.
4. The duplicate frame also arrives intact at the data link layer on B and is unwittingly passed to the network layer there. If A is sending a file to B, part of the file will be duplicated. In other words, the protocol will fail.

Clearly, what is needed is some way for the receiver to be able to distinguish a frame that it is seeing for the first time from a retransmission. The obvious way to achieve this is to have the sender put a sequence number in the header of each frame it sends. Then the receiver can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.

What is the minimum number of bits needed for the sequence number? The header might provide 1 bit, a few bits, 1 byte, or multiple bytes for a sequence number depen

ding on the protocol.

A 1-bit sequence number (0 or 1) is used. At each instant of time, the receiver expects a particular sequence number next. When a frame containing the correct sequence number arrives, it is accepted and passed to the network layer, then acknowledged. Then the expected sequence number is incremented modulo 2 (i.e., 0 becomes 1 and 1 becomes 0).

Any arriving frame containing the wrong sequence number is rejected as a duplicate. However, the last valid acknowledgement is repeated so that the sender can eventually discover that the frame has been received.

An example of this kind of protocol is shown in Fig. 3-14. Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called ARQ (Automatic Repeat reQuest) or PAR (Positive Acknowledgement with Retransmission).

Protocol 3 differs from its predecessors in that both sender and receiver have a variable whose value is remembered while the data link layer is in the wait state. The sender remembers the sequence number of the next frame to send in next frame to send; the receiver remembers the sequence number of the next frame expected in frame expected. Each protocol has a short initialization phase before entering the infinite loop. After transmitting a frame, the sender starts the timer running. If it was already running, it will be reset to allow another full timer interval.

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */
#define MAX_SEQ 1                                   /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;                        /* seq number of next outgoing frame */
  frame s;                                          /* scratch variable */
  packet buffer;                                    /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;                           /* initialize outbound sequence numbers */
  from_network_layer(&buffer);                      /* fetch first packet */
  while (true) {
      s.info = buffer;                              /* construct a frame for transmission */
      s.seq = next_frame_to_send;                   /* insert sequence number in frame */
      to_physical_layer(&s);                        /* send it on its way */
      start_timer(s.seq);                           /* if answer takes too long, time out */
      wait_for_event(&event);                       /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);                  /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
              stop_timer(s.ack);                    /* turn the timer off */
              from_network_layer(&buffer);          /* get the next one to send */
              inc(next_frame_to_send);              /* invert next_frame_to_send */
          }
      }
  }
}
```

```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);                       /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {                 /* a valid frame has arrived */
          from_physical_layer(&r);                  /* go get the newly arrived frame */
          if (r.seq == frame_expected) {            /* this is what we have been waiting for */
              to_network_layer(&r.info);            /* pass the data to the network layer */
              inc(frame_expected);                  /* next time expect the other sequence nr */
          }
          s.ack = 1 – frame_expected;               /* tell which frame is being acked */
          to_physical_layer(&s);                    /* send acknowledgement */
      }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

The interval should be chosen to allow enough time for the frame to get to the receiver, for the receiver to process it in the worst case, and for the acknowledgement frame to propagate back to the sender. Only when that

interval has elapsed is it safe to assume that either the transmitted frame or its acknowledgement has been lost, and to send a duplicate.

If the timeout interval is set too short, the sender will transmit unnecessary frames. While these extra frames will not affect the correctness of the protocol, they will hurt performance. After transmitting a frame and starting the timer, the sender waits for something exciting to happen.

Only three possibilities exist: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires.

- If a valid acknowledgement comes in, the sender fetches the next packet from its network layer and puts it in the buffer, overwriting the previous packet. It also advances the sequence number.
- If a damaged frame arrives or the timer expires, neither the buffer nor the sequence number is changed so that a duplicate can be sent.

In all cases, the contents of the buffer (either the next pack et or a duplicate) are then sent. When a valid frame arrives at the receiver, its sequence number is checked to see if it is a duplicate. If not, it is accepted, passed to the network layer, and an acknowledgement is generated. Duplicates and damaged frames are not passed to the network layer, but they do cause the last correctly received frame to be acknowledged to signal the sender to advance to the next frame or retransmit a damaged frame.

3.3.3 A Simplex Stop-and-Wait Protocol for a Noisy Channel

Now let us consider the normal situation of a communication channel that makes errors. Frames may be either damaged or lost completely. However, we assume that if a frame is damaged in transit, the receiver hardware will detect this when it computes the checksum. If the frame is damaged in such a way that the checksum is nevertheless correct—an unlikely occurrence—this protocol (and all other protocols) can fail (i.e., deliver an incorrect packet to the network layer).

The goal of the data link layer is to provide error-free, transparent communication between network layer processes. The network layer on

machine A gives a series of packets to its data link layer, which must ensure that an identical series of packets is delivered to the network layer on machine B by its data link layer. In particular, the network layer on B has no way of knowing that a packet has been lost or duplicated, so the data link layer must guarantee that no combination of transmission errors, however unlikely, can cause a duplicate packet to be delivered to a network layer. Consider the following scenario:

1. The network layer on A gives packet 1 to its data link layer. The packet is correctly received at B and passed to the network layer on B. B sends an acknowledgement frame back to A.
2. The acknowledgement frame gets lost completely. It just never arrives at all.
3. The data link layer on A eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frame was lost or damaged and sends the frame containing packet 1 again.
4. The duplicate frame also arrives intact at the data link layer on B and is unwittingly passed to the network layer there. In other words, the protocol will fail

Clearly, what is needed is some way for the receiver to be able to distinguish a frame that it is seeing for the first time from a retransmission.

The obvious way to achieve this is to have the sender put a sequence number in the header of each frame it sends. Then the receiver can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.Since the protocol must be correct and the sequence number field in the header is small to use the link efficiently, the question arises: what is the minimum number of bits needed for the sequence number? The header might provide 1 bit, a few bits, 1 byte, or multiple bytes for a sequence number depending on the protocol.

The important point is that it must carry sequence numbers that are large enough for the protocol to work correctly, or it is not much of a protocol.

The only ambiguity in this protocol is between a frame, m, and it's direct successor, m + 1. If frame m is lost or damaged, the receiver will not acknowledge it, so the sender will keep trying to send it. Once it has been

correctly received, the receiver will send an acknowledgement to the sender. It is here that the potential trouble crops up. Depending upon whether the acknowledgement frame gets back to the sender correctly or not, the sender may try to send m or m + 1.

At the sender, the event that triggers the transmission of frame m + 1 is the arrival of an acknowledgement for frame m. But this situation implies that m − 1 has been correctly received, and furthermore that its acknowledgement has also been correctly received by the sender. Otherwise, the sender would not have begun with m, let alone have been considering m + 1. As a consequence, the only ambiguity is between a frame and its immediate predecessor or successor, not between the predecessor and successor themselves.

A 1-bit sequence number (0 or 1) is therefore sufficient. At each instant of time, the receiver expects a particular sequence number next. When a frame containing the correct sequence number arrives, it is accepted and passed to the network layer, then acknowledged. Then the expected sequence number is incremented modulo 2 (i.e., 0 becomes 1 and 1 becomes 0). Any arriving frame containing the wrong sequence number is rejected as a duplicate. However, the last valid acknowledgement is repeated so that the sender can eventually discover that the frame has been received.

An example of this kind of protocol is shown in Fig. 3-14. Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called ARQ (Automatic Repeat reQuest) or PAR (Positive Acknowledgement with Retransmission). Like protocol 2, this one also transmits data only in one direction.

Protocol 3 differs from its predecessors in that both sender and receiver have a variable whose value is remembered while the data link layer is in the wait state. The sender remembers the sequence number of the next frame to send in next frame to send; the receiver remembers the sequence number of the next frame expected in frame expected. Only three possibilities exist: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires.

If a valid acknowledgement comes in, the sender fetches the next packet from its network layer and puts it in the buffer, overwriting the previous packet. It also advances the sequence number. If a damaged frame arrives or the timer expires, neither the buffer nor the sequence number is changed so that a duplicate can be sent. In all cases, the contents of the buffer (either the next pack et or a duplicate) are then sent.

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */
#define MAX_SEQ 1                                   /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;                        /* seq number of next outgoing frame */
  frame s;                                          /* scratch variable */
  packet buffer;                                    /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;                           /* initialize outbound sequence numbers */
  from_network_layer(&buffer);                      /* fetch first packet */
  while (true) {
      s.info = buffer;                              /* construct a frame for transmission */
      s.seq = next_frame_to_send;                   /* insert sequence number in frame */
      to_physical_layer(&s);                        /* send it on its way */
      start_timer(s.seq);                           /* if answer takes too long, time out */
      wait_for_event(&event);                       /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);                  /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
              stop_timer(s.ack);                    /* turn the timer off */
              from_network_layer(&buffer);          /* get the next one to send */
              inc(next_frame_to_send);              /* invert next_frame_to_send */
          }
      }
  }
}
```

```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);                       /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {                 /* a valid frame has arrived */
          from_physical_layer(&r);                  /* go get the newly arrived frame */
          if (r.seq == frame_expected) {            /* this is what we have been waiting for */
              to_network_layer(&r.info);            /* pass the data to the network layer */
              inc(frame_expected);                  /* next time expect the other sequence nr */
          }
          s.ack = 1 – frame_expected;               /* tell which frame is being acked */
          to_physical_layer(&s);                    /* send acknowledgement */
      }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

Each protocol has a short initialization phase before entering the infinite loop. After transmitting a frame and starting the timer, the sender waits for something exciting to happen. When a valid frame arrives at the receiver, its sequence number is checked to see if it is a duplicate. If not, it is accepted, passed to the network layer, and an acknowledgement is generated. Duplicates and damaged frames are not passed to the network layer, but they do cause the last correctly received frame to be acknowledged to signal the sender to advance to the next frame or retransmit a damaged frame.

3.4 SLIDING WINDOW PROTOCOLS

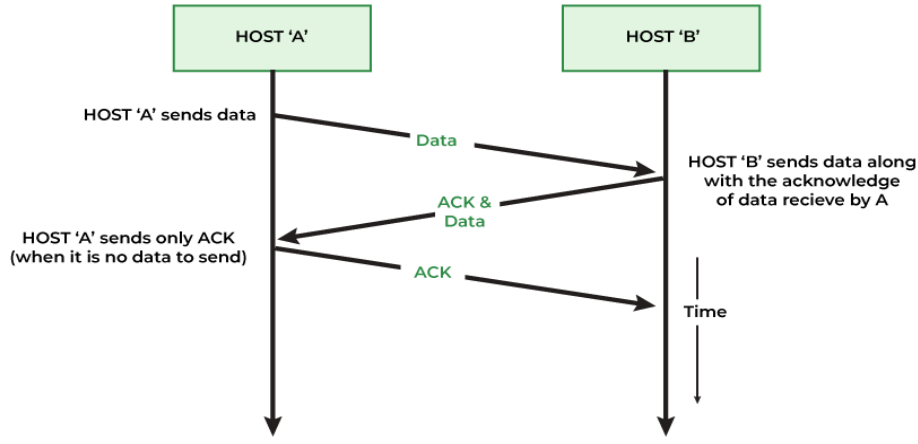
In the previous protocols, data frames were transmitted in **one direction only**. In most practical situations, there is a need to **transmit data in both directions.**

One way of achieving full-duplex data transmission is to **run two instances** of one of the previous protocols, each **using a separate link** for simplex data traffic (in different directions). Each link is then comprised of a **"forward" channel** (for data) and a **"reverse" channel** (for acknowledgements). In both cases the capacity of the reverse channel is almost **entirely wasted.**

A better idea is to **use the same link** for data in both directions. After all, in protocols 2 and 3 it was already being used to transmit frames both ways, and the reverse channel normally has the same capacity as the forward channel. In this model the data frames from A to B are **intermixed** with the acknowledgement frames from A to B.

By looking at the **kind field** in the header of an incoming frame, the receiver can tell whether the frame is data or an acknowledgement. When a data frame arrives, instead of **immediately sending a separate control frame,** the receiver **restrains itself and waits** until the network layer passes it the next packet. The acknowledgement is attached to the outgoing data frame (using the ack field in the frame header). In effect, the acknowledgement gets a **free ride** on the next outgoing data frame. The technique of **temporarily delaying** outgoing acknowledgements so

that they can be hooked onto the next outgoing data frame is known as **piggybacking.**



The principal advantage of using piggybacking over having distinct acknowledgement frames is a **better use of the available channel bandwidth**. The **ack field** in the frame header costs only a few bits, whereas a separate frame would need a header, the acknowledgement, and a checksum. In addition, fewer frames sent generally means a lighter processing load at the receiver.

In the next protocol to be examined, the piggyback field costs only 1 bit in the frame header. It rarely costs more than a few bits. The next three protocols are **bidirectional protocols** that belong to a class called **sliding window protocols.** The three differ among themselves in terms of **efficiency, complexity, and buffer requirements,** as discussed later.

In these, as in all sliding window protocols, each outbound frame contains a sequence number, ranging from 0 up to some maximum. The maximum is usually $2n - 1$ so the sequence number fits exactly in an n-bit field. The stop-and-wait sliding window protocol uses $n = 1$, restricting the sequence numbers to 0 and 1, but more sophisticated versions can use an arbitrary n.

The sequence numbers within the sender's window represent frames that have been sent or can be sent but are as **yet not acknowledged.** Whenever a new packet arrives from the network layer, it is given the next highest sequence number, and the **upper edge of the window is**

advanced by one. When an acknowledgement comes in, **the lower edge is advanced by one**. In this way the window **continuously maintains a list of unacknowledged frames.**

- Figure 3-15 shows an example.

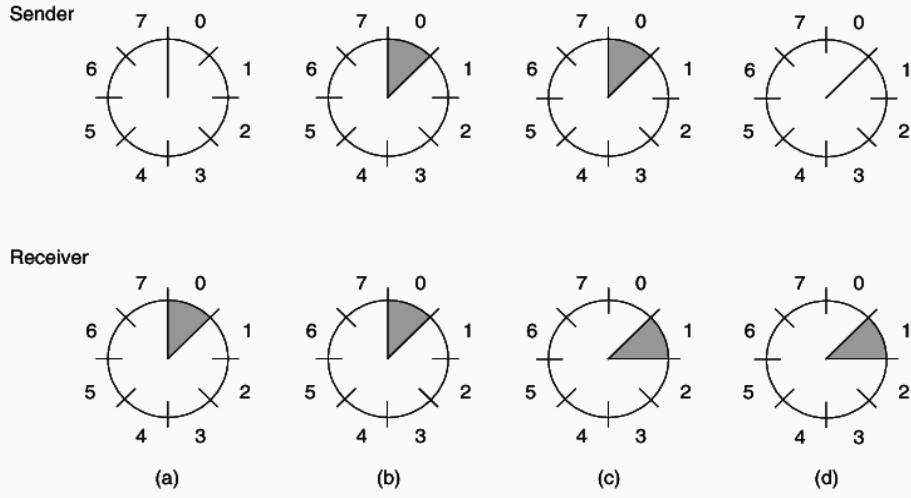


Figure 3-15. A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

3.4.1 A One-Bit Sliding Window Protocol

Figure 3-16 depicts such a protocol. Like the others, it starts out by defining some variables.

Next_frame_to_send tells which frame the sender is trying to send. Similarly, frame_expected tells which frame the receiver is expecting. In both cases, 0 and 1 are the only possibilitiesUnder normal circumstances, one of the **two data link layers goes first** and transmits the first frame. The starting machine **fetches the first packet** from its network layer, builds a frame from it, and sends it. When this frame arrives, the receiving data link layer **checks** to see if it is a duplicate. If the frame is the one expected, it is passed to the network layer and the receiver's window is **slid up.**

The **acknowledgement field** contains the ***number of the last frame received*** without error. If this number agrees with the sequence number of the frame the sender is trying to send, the sender knows it is done with

the frame stored in buffer and can fetch the next packet from its network layer. If the sequence number **disagrees,** it must continue trying to send the same frame. Whenever a frame is received, a frame is also sent back.

```
/* Protocol 4 (Sliding window) is bidirectional. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;                        /* 0 or 1 only */
  seq_nr frame_expected;                            /* 0 or 1 only */
  frame r, s;                                       /* scratch variables */
  packet buffer;                                    /* current packet being sent */
  event_type event;
  next_frame_to_send = 0;                           /* next frame on the outbound stream */
  frame_expected = 0;                               /* frame expected next */
  from_network_layer(&buffer);                      /* fetch a packet from the network layer */
  s.info = buffer;                                  /* prepare to send the initial frame */
  s.seq = next_frame_to_send;                       /* insert sequence number into frame */
  s.ack = 1 - frame_expected;                       /* piggybacked ack */
  to_physical_layer(&s);                            /* transmit the frame */
  start_timer(s.seq);                               /* start the timer running */
  while (true) {
      wait_for_event(&event);                       /* frame_arrival, cksum_err, or timeout */
      if (event == frame_arrival) {                 /* a frame has arrived undamaged */
          from_physical_layer(&r);                  /* go get it */
          if (r.seq == frame_expected) {            /* handle inbound frame stream */
              to_network_layer(&r.info);            /* pass packet to network layer */
              inc(frame_expected);                  /* invert seq number expected next */
          }
          if (r.ack == next_frame_to_send) {        /* handle outbound frame stream */
              stop_timer(r.ack);                    /* turn the timer off */
              from_network_layer(&buffer);          /* fetch new pkt from network layer */
              inc(next_frame_to_send);              /* invert sender's sequence number */
          }
      }
      s.info = buffer;                              /* construct outbound frame */
      s.seq = next_frame_to_send;                   /* insert sequence number into it */
      s.ack = 1 - frame_expected;                   /* seq number of last received frame */
      to_physical_layer(&s);                        /* transmit a frame */
      start_timer(s.seq);                           /* start the timer running */
  }
}
```

Figure 3-16. A 1-bit sliding window protocol.

Now let us examine protocol 4

Assume that computer A is trying to send its frame 0 to computer B and that B is trying to send its frame 0 to A. Suppose that A sends a frame to B, but A's timeout interval is a little too short. Consequently, A may time out repeatedly, sending a series of identical frames, all with **seq = 0 and ack = 1.**When the first valid frame **arrives** at computer B, it will be **accepted** and frame expected will be set to a value of 1. All the subsequent frames received will be rejected because B is now expecting frames with sequence number **1**, not 0.

Furthermore, **since all the duplicates will have ack = 1** and B is still waiting for an acknowledgement of 0, ***B will not go and fetch a new packet from its network layer.***After every rejected duplicate comes in, B will send A a frame containing **seq = 0 and ack = 0.** Eventually, one of these will arrive correctly at A, causing A to begin sending the next packet. No combination of lost frames or premature timeouts can cause

the protocol to deliver duplicate packets to either network layer, to skip a packet, or to deadlock. The protocol is correct.

To show how subtle protocol interactions can be, we note that a peculiar situation arises if both sides **simultaneously send an initial packet.** This synchronization difficulty is illustrated by Fig. 3-17. In part (a), the normal operation of the protocol is shown. In (b) the **peculiarity** is illustrated. If B waits for A's first frame before sending one of its own, the sequence is as shown in (a), and every frame is accepted.

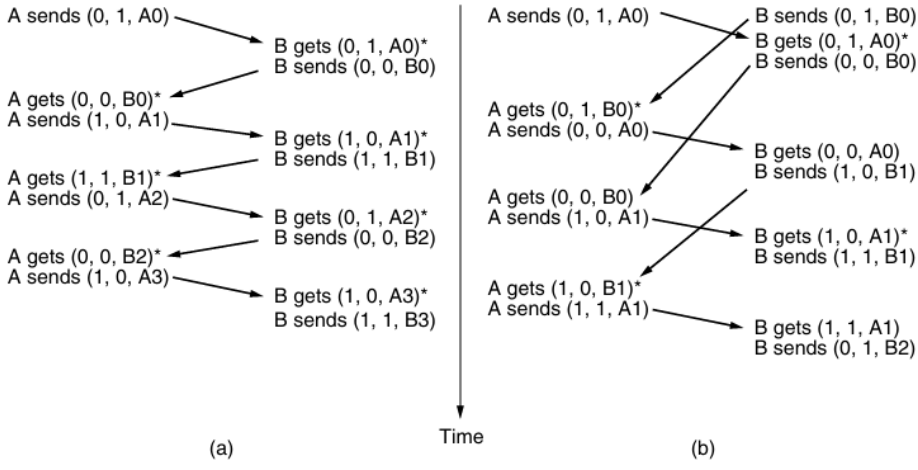


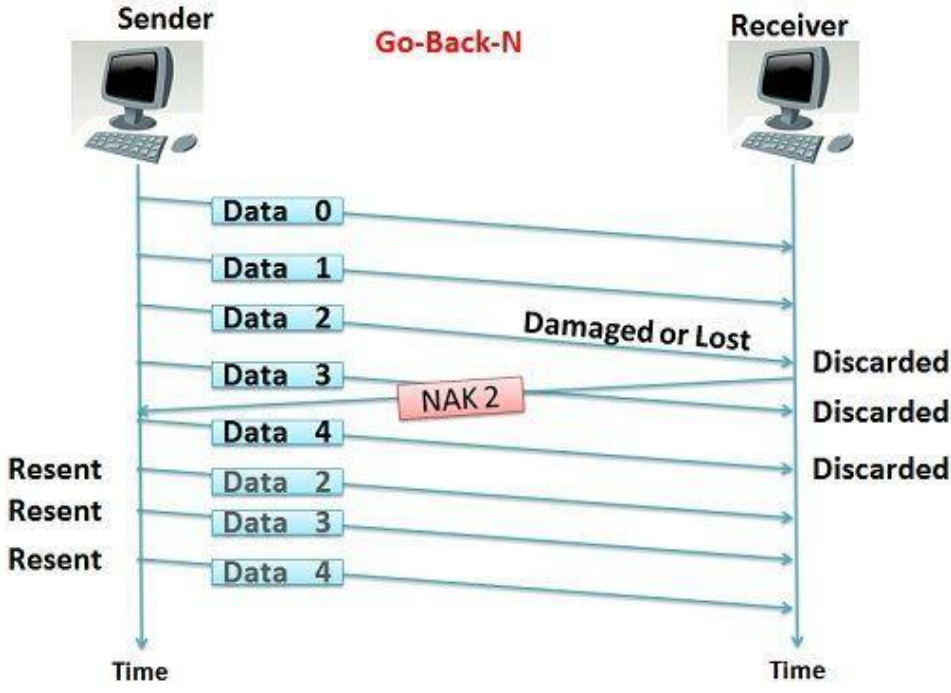
Figure 3-17. Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.

However, if A and B simultaneously initiate communication, their first frames cross, and the data link layers then get into situation (b). In (a) each frame arrival brings a new packet for the network layer; **there are no duplicates.** In (b) half of the frames contain duplicates, even though there are **no transmission errors.** Similar situations can occur as a **result of premature timeouts**, even when one side clearly starts first. In fact, if multiple premature timeouts occur, frames may be sent three or more times, **wasting valuable bandwidth.**

3.4.2 A Protocol Using Go-Back-N Repeat

Go-Back-N protocol is a sliding window protocol. It is a mechanism to detect and control the error in datalink layer. During transmission of frames between sender and receiver, if a frame is damaged, lost, or

an acknowledgement is lost then the action performed by sender and receiver is explained in the following content.



Damaged Frame

If a receiver receives a damaged frame or if an error occurs while receiving a frame then, the receiver sends the NAK (negative acknowledgement) for that frame along with that frame number, that it expects to be retransmitted. After sending NAK, the receiver discards all the frames that it receives, after a damaged frame.

The receiver does not send any ACK (acknowledgement) for the discarded frames. After the sender receives the NAK for the damaged frame, it retransmits all the frames onwards the frame number referred by NAK.

Lost frame

The receiver checks the number on each frame, it receives. If a frame number is skipped in a sequence, then the receiver easily detects the loss of a frame as the newly received frame is received out of sequence. The

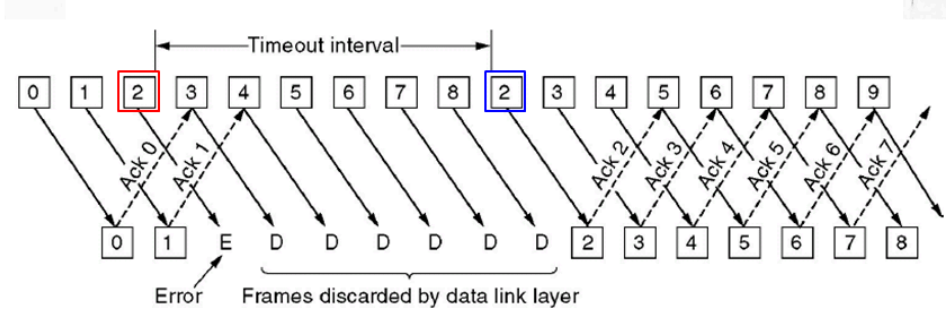
receiver sends the NAK for the lost frame and then the receiver discards all the frames received after a lost frame.

The receiver does not send any ACK (acknowledgement) for that discarded frames. After the sender receives the NAK for the lost frame, it retransmits the lost frame referred by NAK and also retransmits all the frames which it has sent after the lost frame.

Lost Acknowledgement

If the sender does not receive any ACK or if the ACK is lost or damaged in between the transmission. The sender waits for the time to run out and as the time run-outs, the sender retransmits all the frames for which it has not received the ACK. The sender identifies the loss of ACK with the help of a timer.

The ACK number, like NAK (negative acknowledgement) number, shows the number of the frame, that receiver expects to be the next in sequence. The window size of the receiver is 1 as the data link layer only require the frame which it has to send next to the network layer. The sender window size is equal to 'w'. If the error rate is high, a lot of bandwidth is lost wasted.



(maximal number of unacknowledged frame
= maximal window size = maximal sequence number-1)
= *MAX_SEQ*

Definition of Selective Repeat

The go-back-n protocol works well if errors are less, but if the line is poor it wastes a lot of bandwidth on retransmitted frames.

Selective Repeat

An alternative strategy, the selective repeat protocol, is to allow the receiver to accept and buffer the frames following a damaged or lost one. Selective Repeat attempts to retransmit only those packets that are actually lost (due to errors) :

- Receiver must be able to accept packets out of order.
- Since receiver must release packets to higher layer in order, the receiver must be able to buffer some packets.

Selective repeat is also the sliding window protocol which detects or corrects the error occurred in the datalink layer. The selective repeat protocol retransmits only that frame which is damaged or lost. In selective repeat protocol, the retransmitted framed is received out of sequence. The selective repeat protocol can perform the following actions

The receiver is capable of sorting the frame in a proper sequence, as it receives the retransmitted frame whose sequence is out of order of the receiving frame.

The sender must be capable of searching the frame for which the NAK has been received.

The receiver must contain the buffer to store all the previously received frame on hold till the retransmitted frame is sorted and placed in a proper sequence.

The ACK number, like NAK number, refers to the frame which is lost or damaged.

It requires the less window size as compared to go-back-n protocol.

Selective repeat

Damaged frames

If a receiver receives a damaged frame, it sends the NAK for the frame in which error or damage is detected. The NAK number, like in go-back-n also indicates the acknowledgement of the previously received frames and error in the current frame.

The receiver keeps receiving the new frames while waiting for the damaged frame to be replaced. The frames that are received after the

damaged frame are not be acknowledged until the damaged frame has been replaced.

Lost Frame

As in a selective repeat protocol, a frame can be received out of order and further they are sorted to maintain a proper sequence of the frames. While sorting, if a frame number is skipped, the receiver recognizes that a frame is lost and it sends NAK for that frame to the sender.

After receiving NAK for the lost frame the sender searches that frame in its window and retransmits that frame. If the last transmitted frame is lost then the receiver does not respond and this silence is a negative acknowledgement for the sender.

Lost Acknowledgement

If the sender does not receive any ACK or the ACK is lost or damaged in between the transmission. The sender waits for the time to run out and as the time run-outs, the sender retransmits all the frames for which it has not received the ACK. The sender identifies the loss of ACK with the help of a timer.

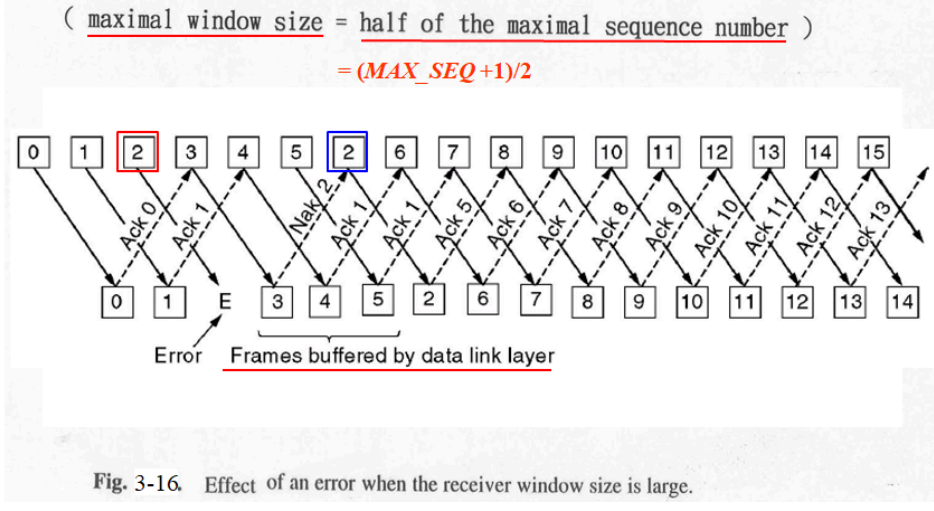
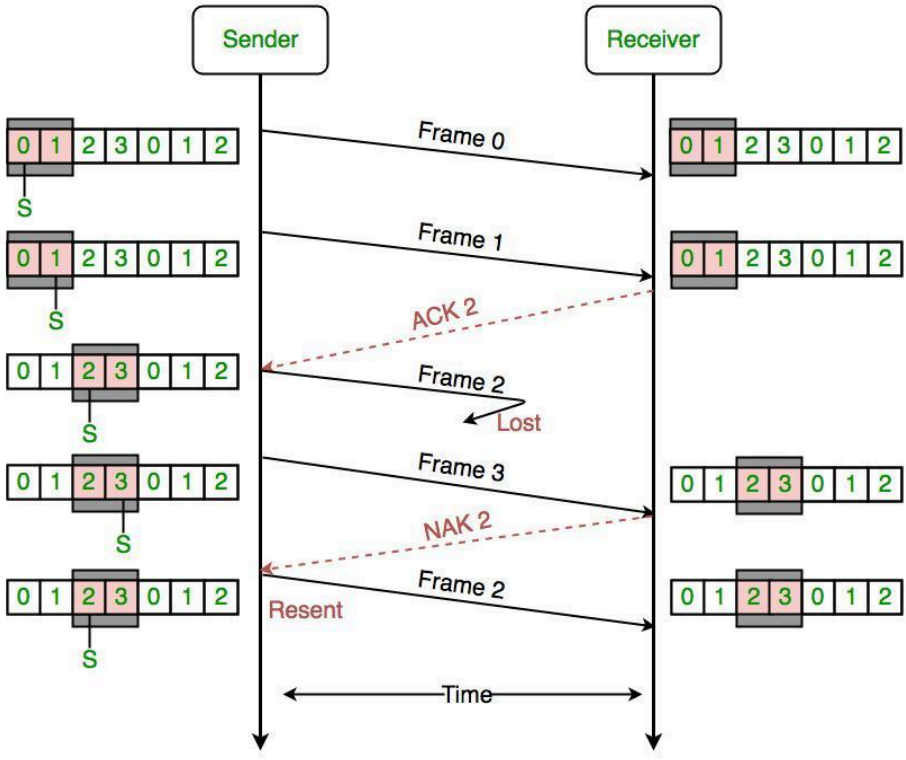


Fig. 3-16. Effect of an error when the receiver window size is large.



Key Differences Between Go-Back-N and Selective Repeat

Go-Back-N protocol is design to retransmit all the frames that are arrived after the damaged or a lost frame. On the other hand, Selective Repeat protocol retransmits only that frame that is damaged or lost.

If the error rate is high i.e. more frames are being damaged and then retransmitting all the frames that arrived after a damaged frame waste the lots of bandwidth. On the other hand, selective repeat protocol re-transmits only damaged frame hence, minimum bandwidth is wasted.

All the frames after the damaged frame are discarded and the retransmitted frames arrive in a sequence from a damaged frame onwards, so, there is less headache of sorting the frames hence it is less complex. On the other hand, only damaged or suspected frame is retransmitted so, extra logic has to be applied for sorting hence, it is more complicated.

Go-Back-N has a window size of N-1 and selective repeat have a window size <=(N+1)/2.

Neither sender nor receiver needs the sorting algorithm in Go-Back-N whereas, the receiver must be able to sort the as it has to maintain the sequence.

In Go-Back-N receiver discards all the frames after the damaged frame hence, it doesn't need to store any frames. Selective repeat protocol does not discard the frames arrived after the damaged frame instead it stores those frames till the damaged frame arrives successfully and is sorted in a proper sequence.

In selective repeat, NAK frame refers to the damaged frame number and in Go-Back-N, NAK frame refers to the next frame expected.

Generally, the Go-Back-N is more in use due to its less complex nature instead of Selective Repeat protocol.

Conclusion

The selective repeat is a more efficient protocol as it does not waste bandwidth for the frames which are properly received but, its complexity and expense favours the use of the go-back-n protocol