The Data Link Layer

Data Link Layer Design Issues

- Network layer services
- Framing
- Error control
- Flow control

Data Link Layer

- Algorithms for achieving:
  - Reliable,
  - Efficient, communication of a whole units – frames (as opposed to bits – Physical Layer) between two machines.
- Two machines are connected by a communication channel that acts conceptually like a wire (e.g., telephone line, coaxial cable, or wireless channel).
- Essential property of a channel that makes it “wire-like” connection is that the bits are delivered in exactly the same order in which they are sent.

Data Link Layer

- For ideal channel (no distortion, unlimited bandwidth and no delay) the job of data link layer would be trivial.
- However, limited bandwidth, distortions and delay makes this job very difficult.

Data Link Layer Design Issues

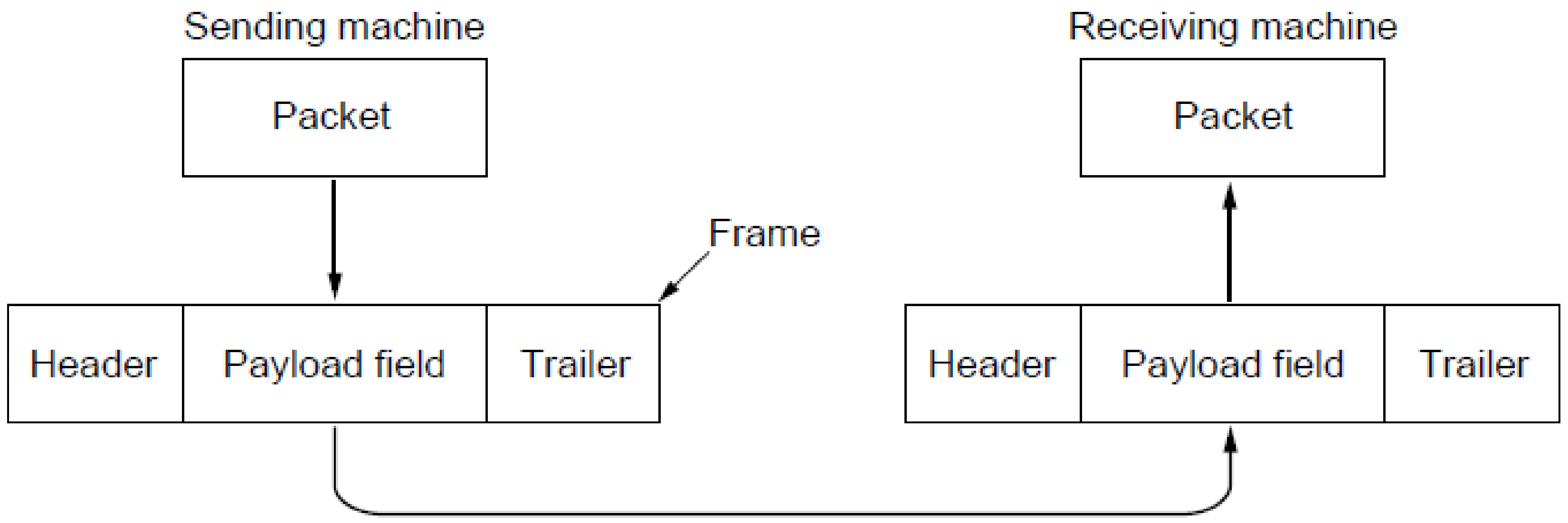
- Physical layer delivers bits of information to and from data link layer. The functions of Data Link Layer are:
  1. Providing a well-defined service interface to the network layer.
  2. Dealing with transmission errors.
  3. Regulating the flow of data so that slow receivers are not swamped by fast senders.
- Data Link layer
  – Takes the packets from Physical layer, and
  – Encapsulates them into **frames**

Data Link Layer Design Issues

- Each frame has a
  - frame header – a field for holding the packet, and
  - frame trailer.
- Frame Management is what Data Link Layer does.

- See figure in the next slide:

Packets and Frames

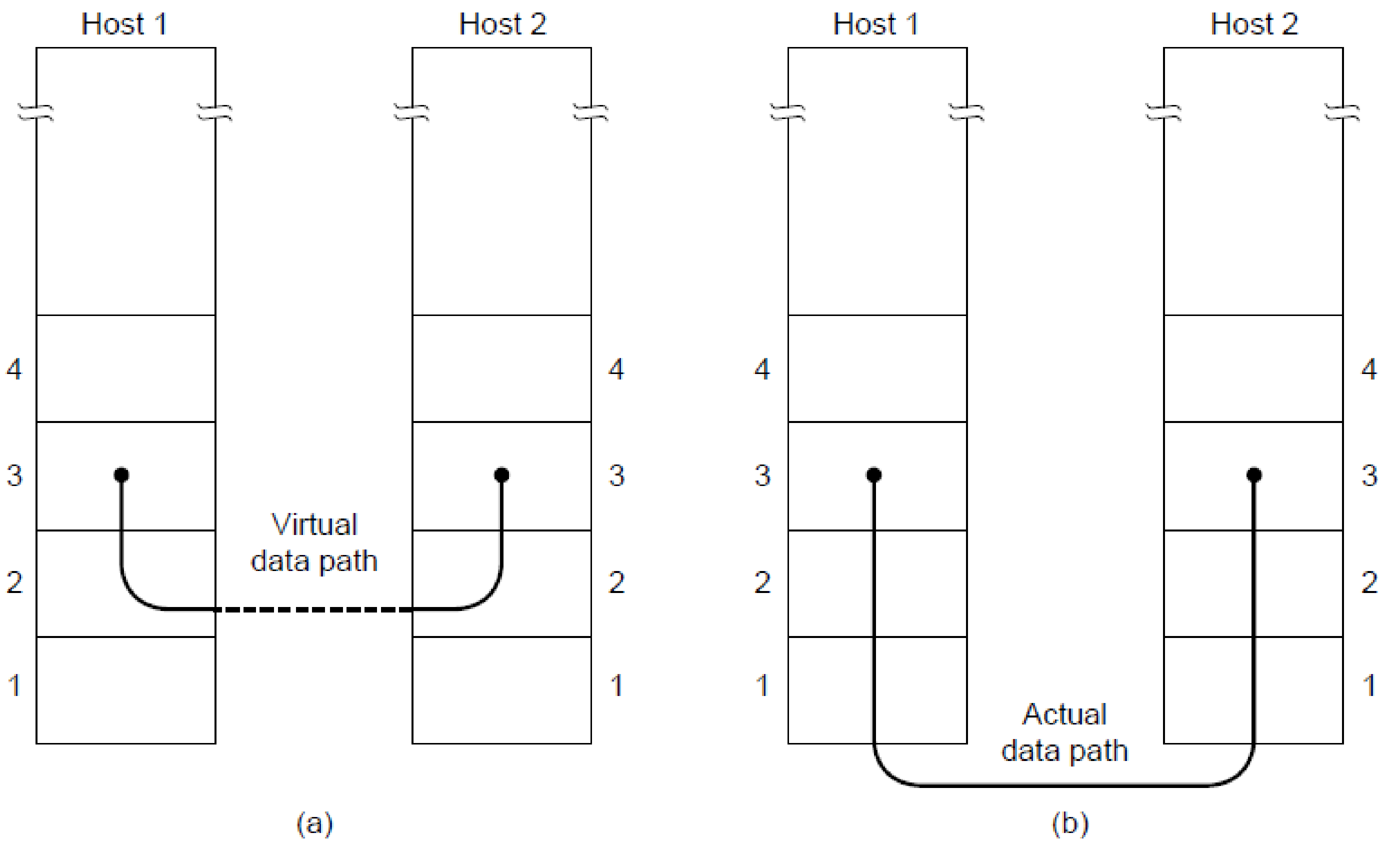


Relationship between packets and frames.

Services Provided to the Network Layer

- Principal Service Function of the data link layer is to transfer the data from the network layer on the source machine to the network layer on the destination machine.
  - Process in the network layer that hands some bits to the data link layer for transmission.
  - Job of data link layer is to transmit the bits to the destination machine so they can be handed over to the network layer there (see figure in the next slide).

Network Layer Services



(a) Virtual communication. (b) Actual communication.

Possible Services Offered

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

Unacknowledged Connectionless Service

- It consists of having the source machine send independent frames to the destination machine without having the destination machine acknowledge them.
- Example: Ethernet, Voice over IP, etc. in all the communication channel were real time operation is more important that quality of transmission.

Acknowledged Connectionless Service

- Each frame send by the Data Link layer is acknowledged and the sender knows if a specific frame has been received or lost.
- Typically the protocol uses a specific time period that if has passed without getting acknowledgment it will re-send the frame.
- This service is useful for commutation when an unreliable channel is being utilized (e.g., 802.11 WiFi).

- Network layer does not know frame size of the packets and other restriction of the data link layer. Hence it becomes necessary for data link layer to have some mechanism to optimize the transmission.

Acknowledged Connection Oriented Service

- Source and Destination establish a connection first.
- Each frame sent is numbered
  - Data link layer guarantees that each frame sent is indeed received.
  - It guarantees that each frame is received only once and that all frames are received in the correct order.
- Examples:
  - Satellite channel communication,
  - Long-distance telephone communication, etc.

Acknowledged Connection Oriented Service

- Three distinct phases:
  1. Connection is established by having both side initialize variables and counters needed to keep track of which frames have been received and which ones have not.
  2. One or more frames are transmitted.
  3. Finally, the connection is released – freeing up the variables, buffers, and other resources used to maintain the connection.

Framing

- To provide service to the network layer the data link layer must use the service provided to it by physical layer.
- Stream of data bits provided to data link layer is not guaranteed to be without errors.
- Errors could be:
  - Number of received bits does not match number of transmitted bits (deletion or insertion)
  - Bit Value
- It is up to data link layer to correct the errors if necessary.

Framing

- Transmission of the data link layer starts with breaking up the bit stream
  - into discrete frames
  - Computation of a checksum for each frame, and
  - Include the checksum into the frame before it is transmitted.
- Receiver computes its checksum error for a receiving frame and if it is different from the checksum that is being transmitted will have to deal with the error.
- Framing is more difficult than one could think!

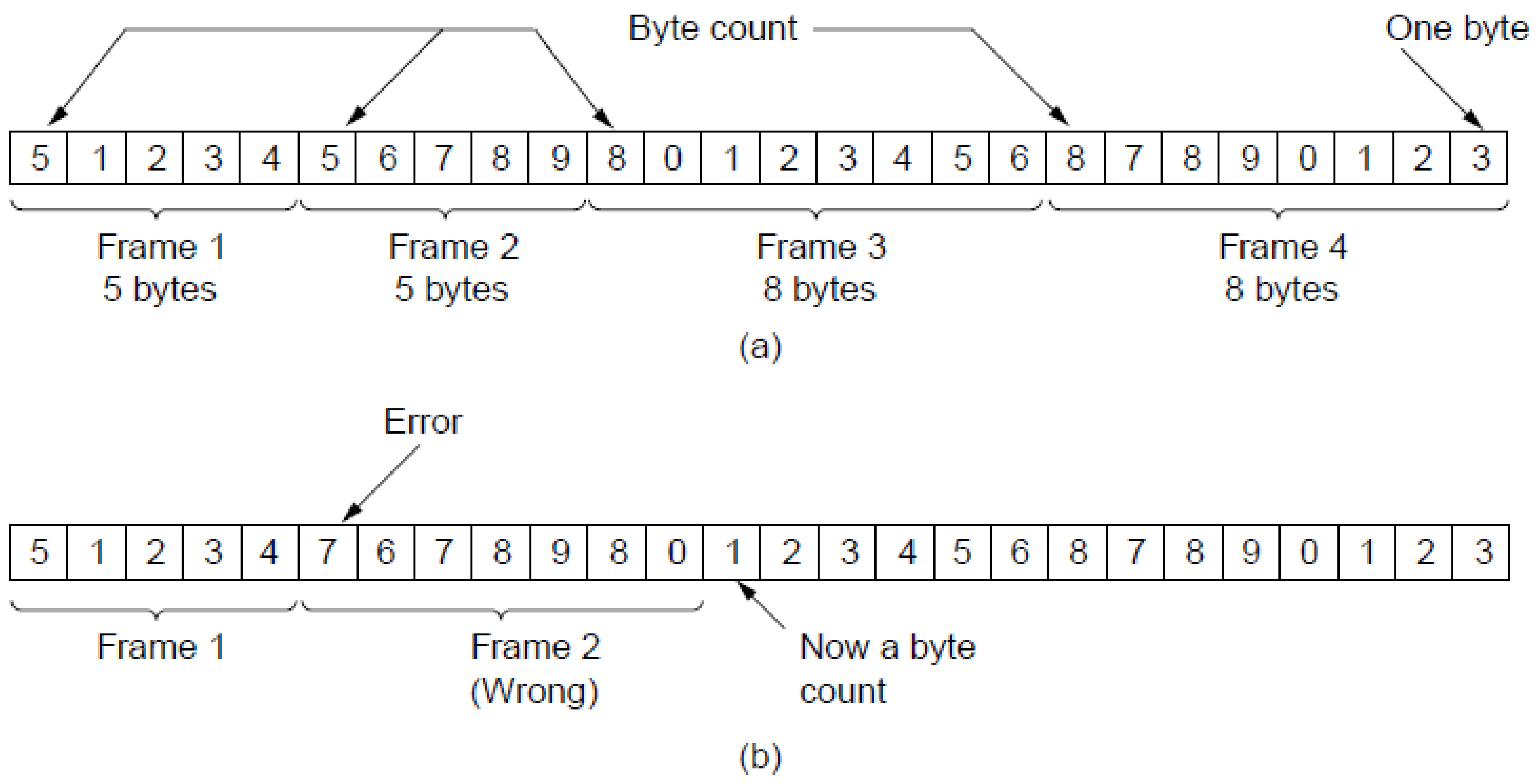
Framing Methods

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

Byte Count Framing Method

- It uses a field in the header to specify the number of bytes in the frame.
- Once the header information is being received it will be used to determine end of the frame.
- See figure in the next slide:
- Trouble with this algorithm is that when the count is incorrectly received the destination will get out of synch with transmission.
  - Destination may be able to detect that the frame is in error but it does not have a means (in this algorithm) how to correct it.

Framing (1)

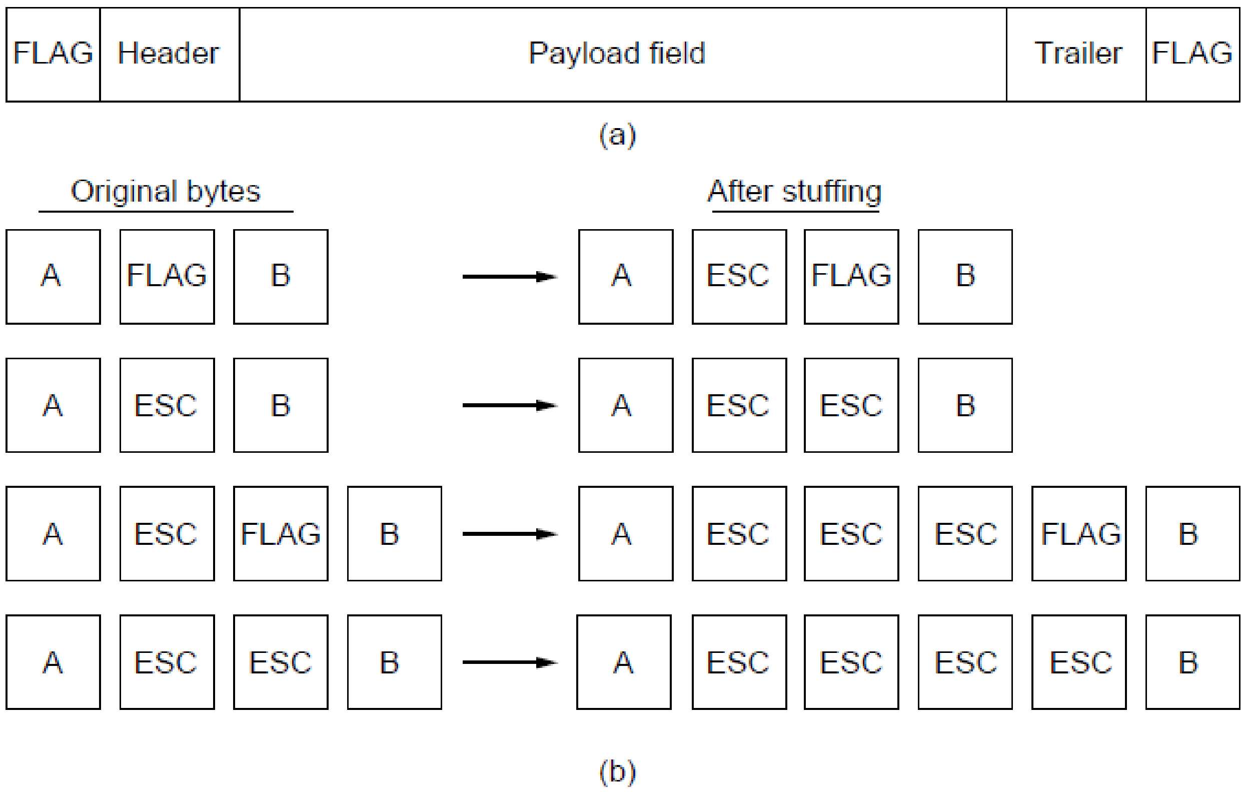


A byte stream. (a) Without errors. (b) With one error.

Flag Bytes with Byte Stuffing Framing Method

- This methods gets around the boundary detection of the frame by having each appended by the frame start and frame end special bytes.
- If they are the same (beginning and ending byte in the frame) they are called **flag byte**.
- In the next slide figure this byte is shown as FLAG.
- If the actual data contains a byte that is identical to the FLAG byte (e.g., picture, data stream, etc.) the convention that can be used is to have escape character inserted just before the “FLAG” character.

Framing (2)

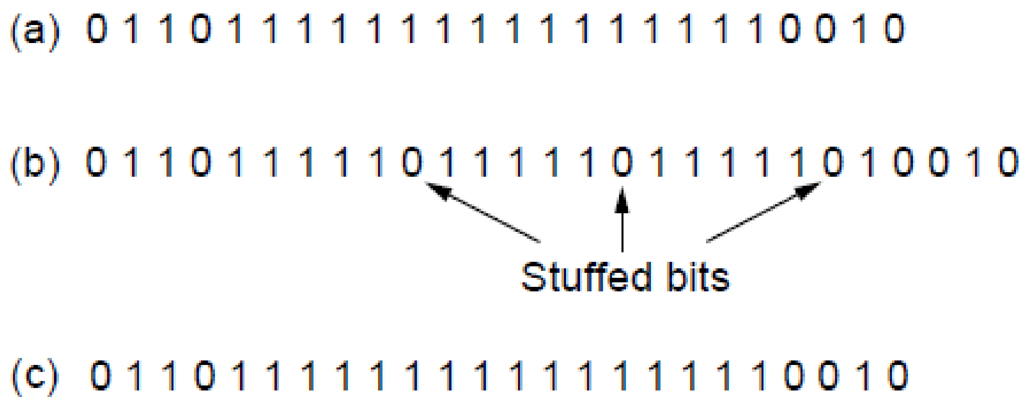


a) A frame delimited by flag bytes.
b) Four examples of byte sequences before and after byte stuffing.

Flag Bits with Bit Stuffing Framing Method

- This methods achieves the same thing as Byte Stuffing method by using Bits (1) instead of Bytes (8 Bits).
- It was developed for High-level Data Link Control (HDLC) protocol.
- Each frames begins and ends with a special bit patter:
  - 01111110 or 0x7E <- Flag Byte
  - Whenever the sender's data link layer encounters five consecutive 1s in the data it automatically stuffs a 0 bit into the outgoing bit stream.
  - USB uses bit stuffing.

Framing (3)


(a) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0
(b) 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0
Stuffed bits
(c) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

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.

Framing

- Many data link protocols use a combination of presented methods for safety. For example in Ethernet and 802.11 each frame begin with a well-defined pattern called a preamble.
- Preamble is typically 72 bits long.
- It is then followed by a length fileld.

Error Control

- After solving the marking of the frame with start and end the data link layer has to handle eventual errors in transmission or detection.
  - Ensuring that all frames are delivered to the network layer at the destination and in proper order.
- Unacknowledged connectionless service: it is OK for the sender to output frames regardless of its reception.
- Reliable connection-oriented service: it is NOT OK.

Error Control

- Reliable connection-oriented service usually will provide a sender with some feedback about what is happening at the other end of the line.
  - Receiver Sends Back Special Control Frames.
  - If the Sender Receives positive Acknowledgment it will know that the frame has arrived safely.
- Timer and Frame Sequence Number for the Sender is Necessary to handle the case when there is no response (positive or negative) from the Receiver .

Flow Control

- Important Design issue for the cases when the sender is running on a fast powerful computer and receiver is running on a slow low-end machine.
- Two approaches:
  1. Feedback-based flow control
  2. Rate-based flow control

Feedback-based Flow Control

- Receiver sends back information to the sender giving it permission to send more data, or
- Telling sender how receiver is doing.

Rate-based Flow Control

- Built in mechanism that limits the rate at which sender may transmit data, without the need for feedback from the receiver.

Error Detection and Correction

- Two basic strategies to deal with errors:
  1. Include enough redundant information to enable the receiver to deduce what the transmitted data must have been.

     Error correcting codes.
  2. Include only enough redundancy to allow the receiver to deduce that an error has occurred (but not which error).

     Error detecting codes.

Error Detection and Correction

- Error codes are examined in Link Layer because this is the first place that we have run up against the problem of reliability transmitting groups of bits.
  - Codes are reused because reliability is an overall concern.
  - The error correcting code are also seen in the physical layer for noise channels.
  - Commonly they are used in link, network and transport layer.

Error Detection and Correction

- Error codes have been developed after long fundamental research conducted in mathematics.
- Many protocol standards get codes from the large field in mathematics.

Error Detection & Correction Code (1)

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

Error Detection & Correction Code

- All the codes presented in previous slide add redundancy to the information that is being sent.
- A frame consists of
  - m data bits (message) and
  - r redundant bits (check).
- ***Block code*** - the r check bits are computed solely as function of the m data bits with which they are associated.

- ***Systemic code*** – the m data bits are send directly along with the check bits.
- ***Linear code*** – the r check bits are computed as a linear function of the m data bits.

Error Detection & Correction Code

- n – total length of a block (i.e., $n = m + r$)
- (n, m) – code
- n – bit ***codeword*** containing n bits.
- m/n – code rate (range ½ for noisy channel and close to 1 for high-quality channel).

Error Detection & Correction Code

Example

- Transmitted: 10001001
- Received: 10110001

XOR operation gives number of bits that are different.

- XOR: 00111000

- Number of bit positions in which two codewords differ is called ***Hamming Distance***. It shows that two codes are ***d*** distance apart, and it will require ***d*** errors to convert one into the other.

Error Detection & Correction Code

- All 2^m possible data messages are legal, but due to the way the check bits are computers not all 2^n possible code words are used.
- Only small fraction of $2^m/2^n=1/2^r$ ***are possible will be legal codewords.***
- The error-detecting and error-correcting codes of the block code depend on this Hamming distance.
- To reliably detect d error, one would need a distance $d+1$ code.
- To correct d error, one would need a distance $2d+1$ code.

Error Detection & Correction Code

- All 2^m possible data messages are legal, but due to the way the check bits are computers not all 2^n possible code words are used.
- Only small fraction of $2^m/2^n=1/2^r$ ***are possible will be legal codewords.***
- The error-detecting and error-correcting codes of the block code depend on this Hamming distance.
- To reliably detect $\boldsymbol{d}$ error, one would need a distance $\boldsymbol{d+1}$ code.
- To correct $\boldsymbol{d}$ error, one would need a distance $2\boldsymbol{d+1}$ code.

Error Detection & Correction Code

Example:

- 4 valid codes:
  - 0000000000
  - 0000011111
  - 1111100000
  - 1111111111
- Minimal Distance of this code is 5 => can correct and double errors and it detect quadruple errors.
- 0000000111 => single or double – bit error. Hence the receiving end must assume the original transmission was 0000011111.
- 0000000000 => had triple error that was received as 0000000111 it would be detected in error.

Error Detection & Correction Code

- One cannot perform double errors and at the same time detect quadruple errors.
- Error correction requires evaluation of each candidate codeword which may be time consuming search.
- Through design this search time can be minimized.
- In theory if n = m + r, this requirement becomes:
  - $(m + r + 1) \leq 2^r$

Hamming Code

- Codeword: b1 b2 b3 b4 ….
- Check bits: The bits that are powers of 2 (p1, p2, p4, p8, p16, …).
- The rest of bits (m3, m5, m6, m7, m9, …) are filled with m data bits.
- Example of the Hamming code with $m = 7$ data bits and $r = 4$ check bits is given in the next slide.

The Hamming Code

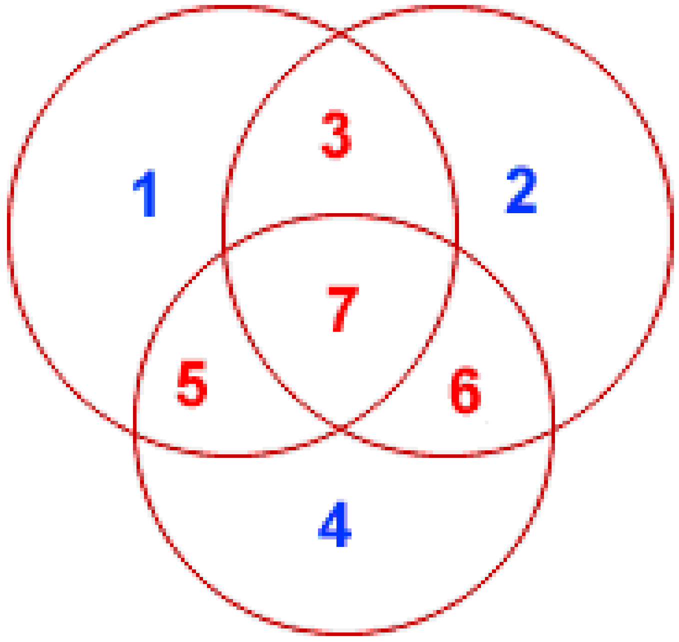
Consider a message having four data bits (D) which is to be transmitted as a 7-bit codeword by adding three error control bits. This would be called a (7,4) code. The three bits to be added are three EVEN Parity bits (P), where the parity of each is computed on different subsets of the message bits as shown below.

7	6	5	4	3	2	1	
D	D	D	P	D	P	P	7-BIT CODEWORD
D	-	D	-	D	-	P	(EVEN PARITY)
D	D	-	-	D	P	-	(EVEN PARITY)
D	D	D	P	-	-	-	(EVEN PARITY)

Hamming Code

- **Why Those Bits?** - The three parity bits **(1,2,4)** are related to the data bits **(3,5,6,7)** as shown at right. In this diagram, each overlapping circle corresponds to one parity bit and defines the four bits contributing to that parity computation. For example, data bit **3** contributes to parity bits **1** and **2**. Each circle (parity bit) encompasses a total of four bits, and each circle must have EVEN parity. Given four data bits, the three parity bits can easily be chosen to ensure this condition.
- It can be observed that changing any one bit numbered 1..7 uniquely affects the three parity bits. Changing bit **7** affects all three parity bits, while an error in bit **6** affects only parity bits **2** and **4**, and an error in a parity bit affects only that bit. The location of any single bit error is determined directly upon checking the three parity circles.

Hamming Code



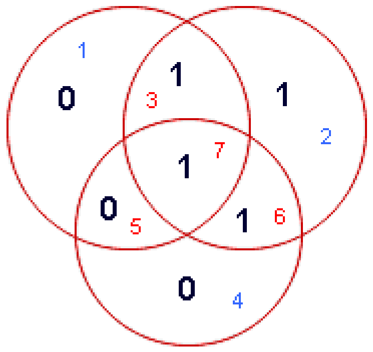
Hamming Code

- For example, the message 1101 would be sent as 1100110, since:

7	6	5	4	3	2	1	
1	1	0	0	1	1	0	7-BIT CODEWORD
1	-	0	-	1	-	**0**	(EVEN PARITY)
1	1	-	-	1	**1**	-	(EVEN PARITY)
1	1	0	**0**	-	-	-	(EVEN PARITY)

Hamming Codes

- When these seven bits are entered into the parity circles, it can be confirmed that the choice of these three parity bits ensures that the parity within each circle is EVEN, as shown here.



Hamming Code

- It may now be observed that if an error occurs in any of the seven bits, that error will affect different combinations of the three parity bits depending on the bit position.

- For example, suppose the above message 1100110 is sent and a single bit error occurs such that the codeword 1110110 is received:

```
transmitted message                              received message
      1 1 0 0 1 1 0        ------------>               1 1 1 0 1 1 0
 BIT: 7 6 5 4 3 2 1                              BIT: 7 6 5 4 3 2 1
```

The above error (in bit 5) can be corrected by examining which of the three parity bits was affected by the bad bit:

Hamming Code

7	6	5	4	3	2	1			
1	1	1	**0**	1	**1**	**0**	7-BIT CODEWORD		
1	-	1	-	1	-	0	(EVEN PARITY)	**NOT!**	**1**
1	1	-	-	1	1	-	(EVEN PARITY)	**OK!**	**0**
1	1	1	0	-	-	-	(EVEN PARITY)	**NOT!**	**1**

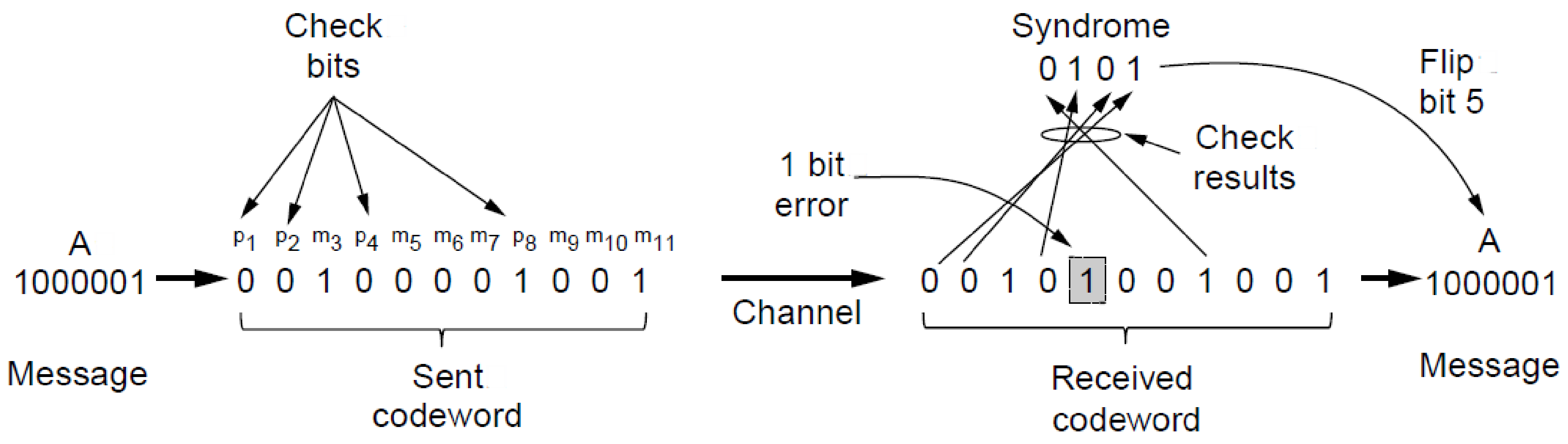
Hamming Code

- *In fact, the bad parity bits labeled **101** point directly to the bad bit since **101** binary equals **5**.* Examination of the 'parity circles' confirms that any single bit error could be corrected in this way.
- The value of the Hamming code can be summarized:

1. Detection of 2 bit errors (assuming no correction is attempted);
2. Correction of single bit errors;
3. Cost of 3 bits added to a 4-bit message.

- The ability to correct single bit errors comes at a cost which is less than sending the entire message twice. (Recall that simply sending a message twice accomplishes no error correction.)

Error Detection Codes (2)

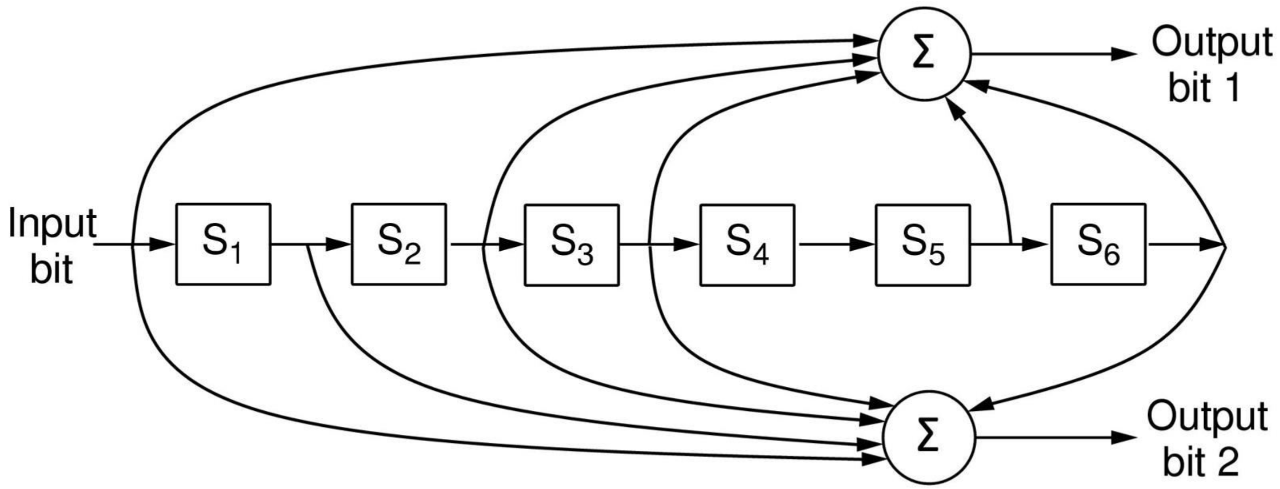


Example of an (11, 7) Hamming code correcting a single-bit error.

Convolutional Codes

- Not a block code
- There is no natural message size or encoding boundary as in a block code.
- The output depends on the current and previous input bits. Encoder has memory.
- The number of previous bits on which the output depends is called the **constraint length** of the code.
- They are deployed as part of the
  - GSM mobile phone system
  - Satellite Communications, and
  - 802.11 (see example in the previous slide).

Error Detection Codes (3)



The NASA binary convolutional code used in 802.11.

Convolutional Encoders

- Like any error-correcting code, a convolutional code works by adding some structured redundant information to the user's data and then correcting errors using this information.
- A convolutional encoder is a *linear system*.
- A binary convolutional encoder can be represented as a *shift register*. The outputs of the encoder are modulo 2 sums of the values in the certain register's cells. The input to the encoder is either the unencoded sequence (for *non-recursive codes*) or the unencoded sequence added with the values of some register's cells (for *recursive codes*).
- Convolutional codes can be *systematic* and *non-systematic*. Systematic codes are those where an unencoded sequence is a part of the output sequence. Systematic codes are almost always recursive, conversely, non-recursive codes are almost always non-systematic.

Convolutional Encoders

- A combination of register's cells that forms one of the output streams (or that is added with the input stream for recursive codes) is defined by a *polynomial*. Let *m* be the maximum degree of the polynomials constituting a code, then *K*=*m*+1 is a *constraint length* of the code.

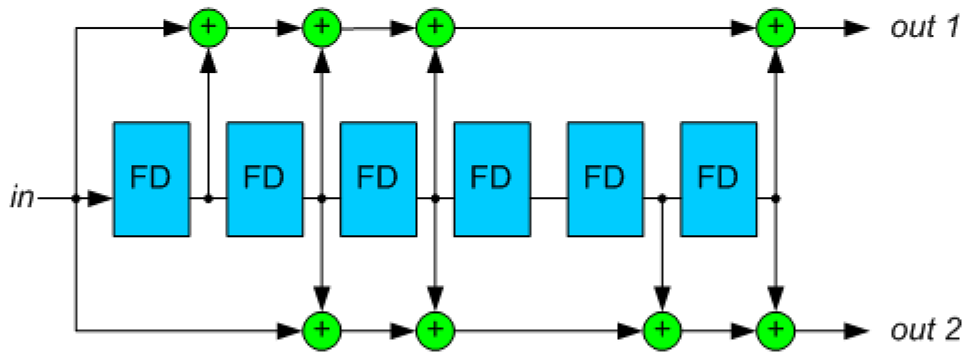


Figure 1. A standard NASA convolutional encoder with polynomials (171,133).

Convolutional Encoders

- For example, for the decoder on the Figure 1, the polynomials are:

$$g_1(z)=1+z+z^2+z^3+z^6$$

$$g_2(z)=1+z^2+z^3+z^5+z^6$$

- A code rate is an inverse number of output polynomials.
- For the sake of clarity, in this article we will restrict ourselves to the codes with rate R=1/2. Decoding procedure for other codes is similar.
- Encoder polynomials are usually denoted in the octal notation. For the above example, these designations are “1111001” = 171 and “1011011” = 133.
- The constraint length of this code is 7.
- An example of a recursive convolutional encoder is on the Figure 2.

Example of the Convolutional Encoder

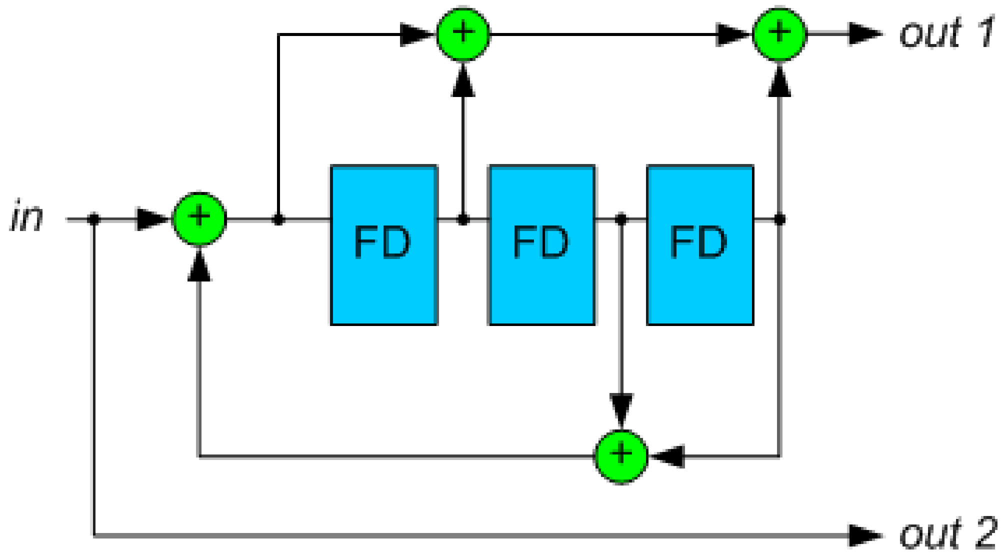


Figure 2. A recursive convolutional encoder.

Trellis Diagram

- A convolutional encoder is often seen as a *finite state machine*. Each state corresponds to some value of the encoder's register. Given the input bit value, from a certain state the encoder can move to two other states. These state transitions constitute a diagram which is called a *trellis diagram*.
- A trellis diagram for the code on the Figure 2 is depicted on the Figure 3. A solid line corresponds to input 0, a dotted line – to input 1 (note that encoder states are designated in such a way that the rightmost bit is the newest one).
- Each path on the trellis diagram corresponds to a valid sequence from the encoder's output. Conversely, any valid sequence from the encoder's output can be represented as a path on the trellis diagram. One of the possible paths is denoted as red (as an example).

Trellis Diagram

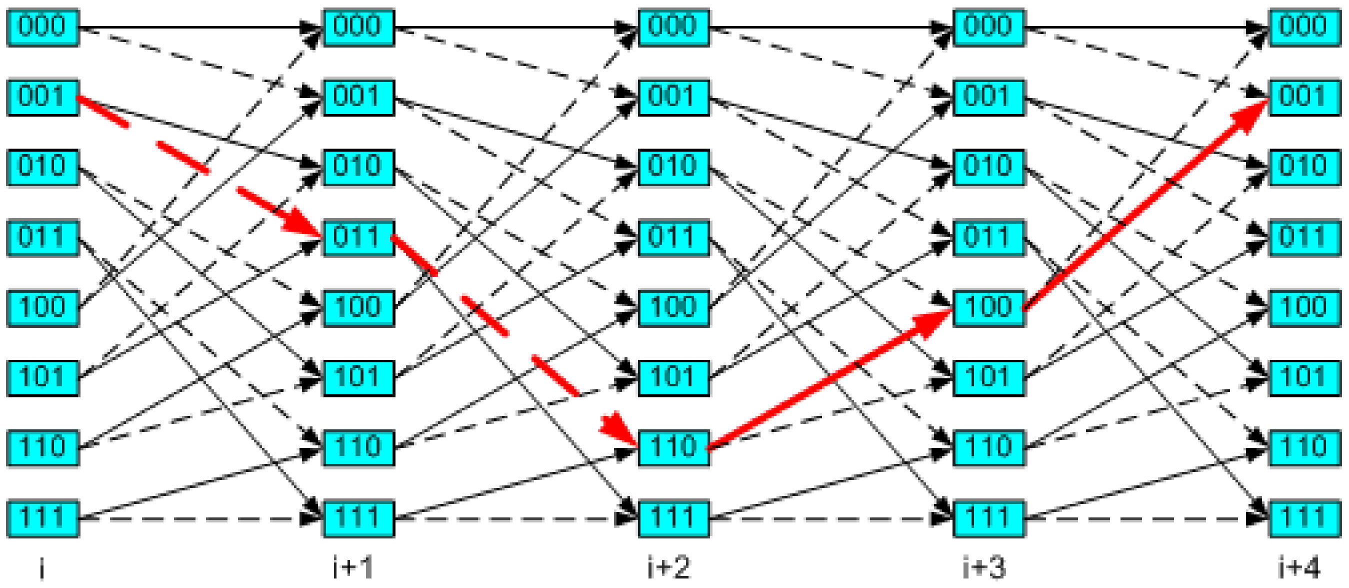


Figure 3. A trellis diagram corresponding to the encoder on the Figure 2.

Trellis Diagram

- Note that each state transition on the diagram corresponds to a pair of output bits. There are only two allowed transitions for every state, so there are two allowed pairs of output bits, and the two other pairs are forbidden. If an error occurs, it is very likely that the receiver will get a set of forbidden pairs, which don't constitute a path on the trellis diagram. So, the task of the decoder is to find a path on the trellis diagram which is the closest match to the received sequence.

Trellis Diagram

- Let's define a *free distance d_f* as a minimal Hamming distance between two different allowed binary sequences (a Hamming distance is defined as a number of differing bits).
- A free distance is an important property of the convolutional code. It influences a number of closely located errors the decoder is able to correct.

Viterbi Algorithm

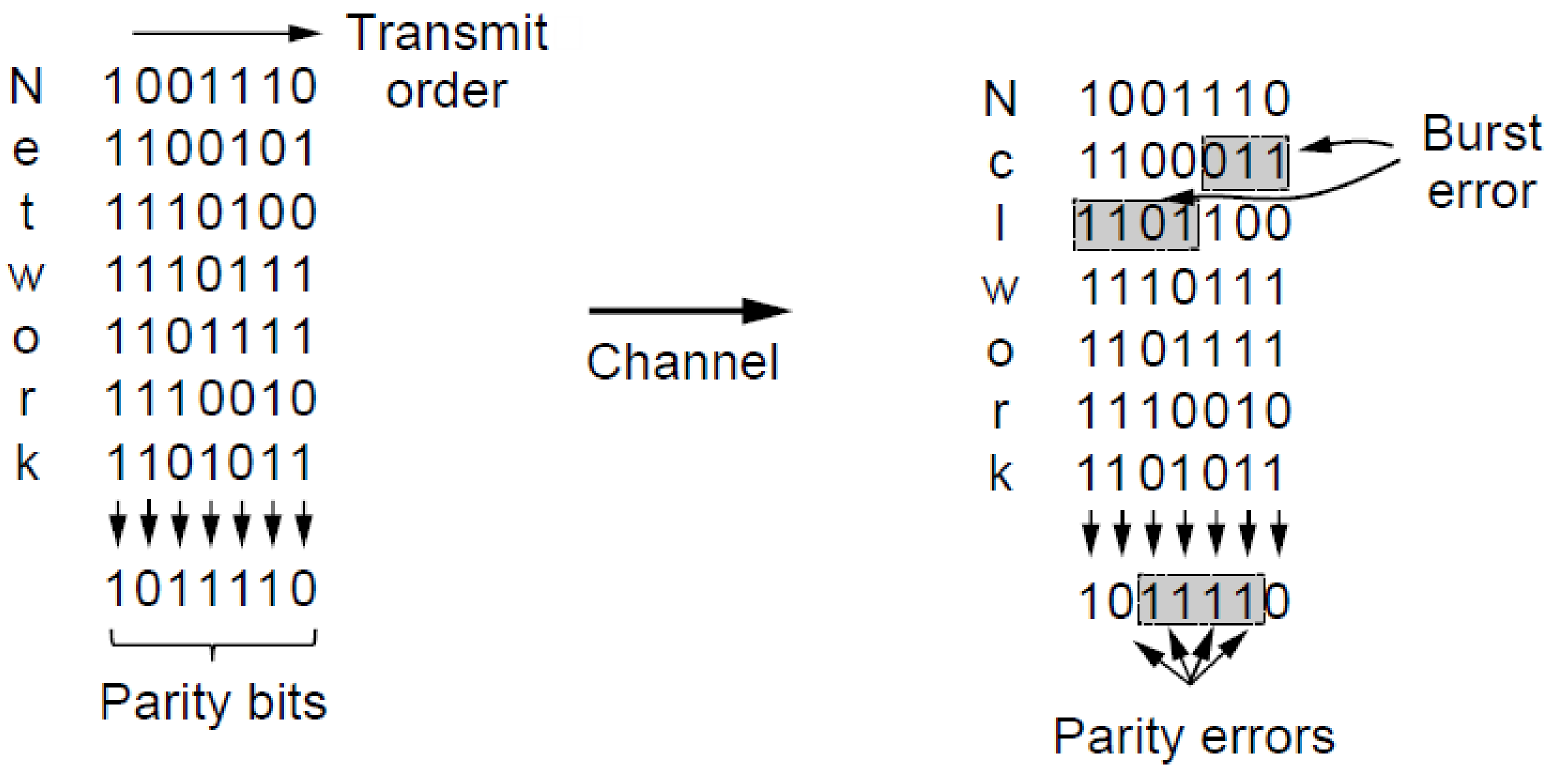
- Viterbi algorithm reconstructs the maximum-likelihood path for a given input sequence.

Error-Detecting Codes (1)

Linear, systematic block codes

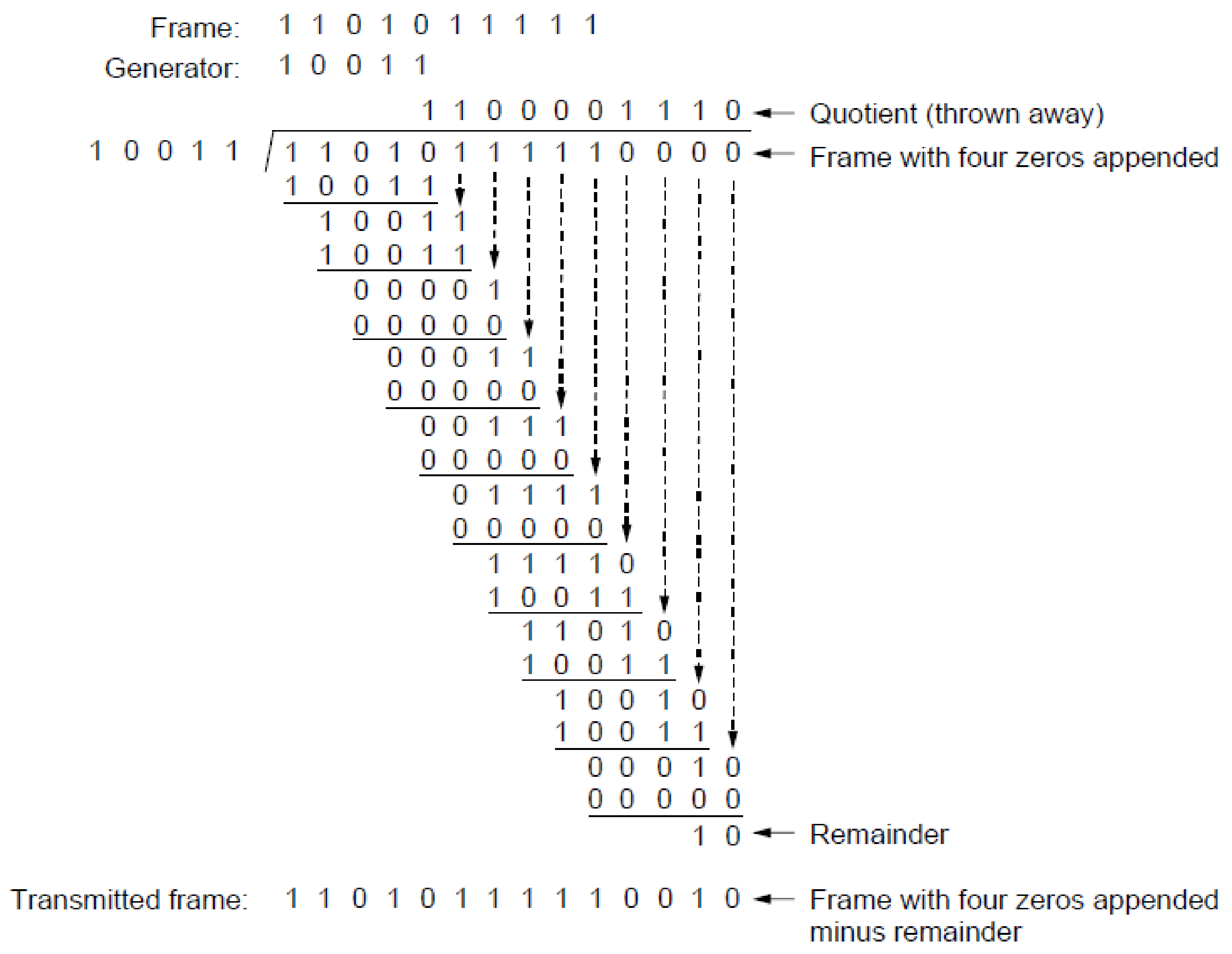
1. Parity.
2. Checksums.
3. Cyclic Redundancy Checks (CRCs).

Error-Detecting Codes (2)



Interleaving of parity bits to detect a burst error.

Error-Detecting Codes (3)



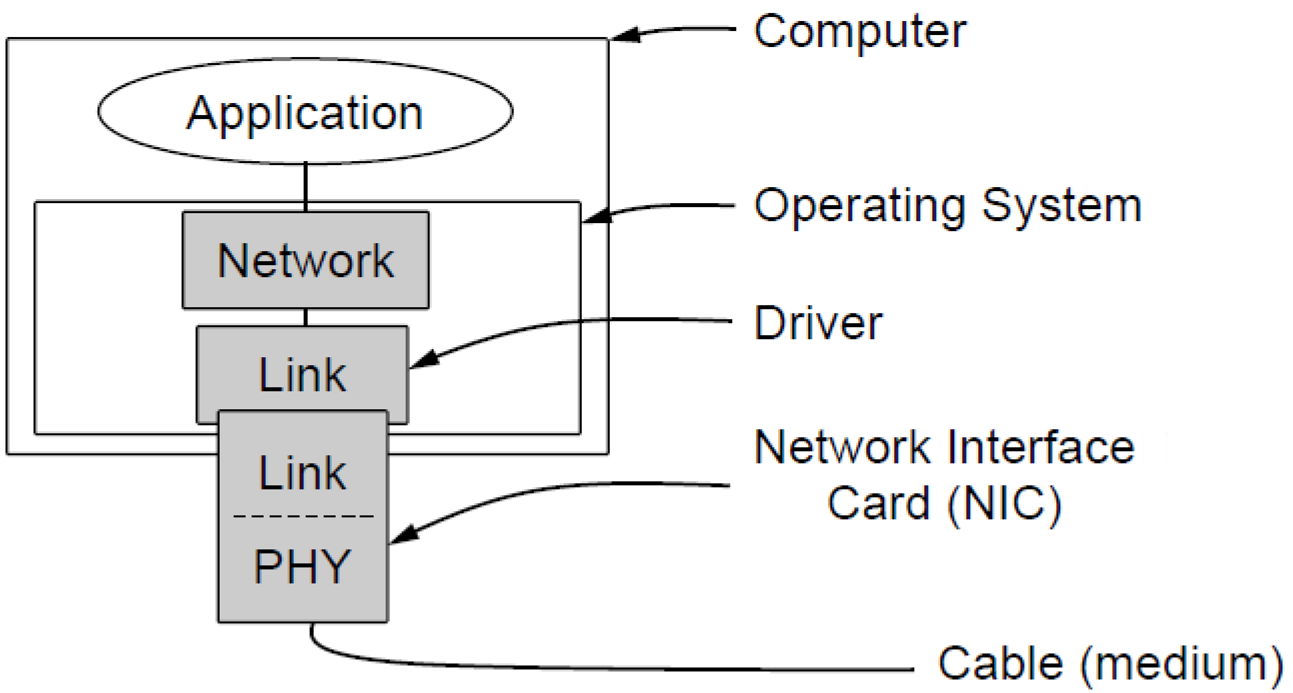
```
              Frame:  1 1 0 1 0 1 1 1 1 1
          Generator:  1 0 0 1 1

                              1 1 0 0 0 0 1 1 1 0  ← Quotient (thrown away)
          1 0 0 1 1 / 1 1 0 1 0 1 1 1 1 1 0 0 0 0  ← Frame with four zeros appended
                      1 0 0 1 1
                        1 0 0 1 1
                        1 0 0 1 1
                          0 0 0 0 1
                          0 0 0 0 0
                            0 0 0 1 1
                            0 0 0 0 0
                              0 0 1 1 1
                              0 0 0 0 0
                                0 1 1 1 1
                                0 0 0 0 0
                                  1 1 1 1 0
                                  1 0 0 1 1
                                    1 1 0 1 0
                                    1 0 0 1 1
                                      1 0 0 1 0
                                      1 0 0 1 1
                                        0 0 0 1 0
                                        0 0 0 0 0
                                              1 0  ← Remainder

 Transmitted frame:   1 1 0 1 0 1 1 1 1 1 0 0 1 0  ← Frame with four zeros appended
                                                      minus remainder
```

Example calculation of the CRC

Elementary Data Link Protocols (1)

- Utopian Simplex Protocol
- Simplex Stop-and-Wait Protocol
    - Error-Free Channel
- Simplex Stop-and-Wait Protocol
    - Noisy Channel

Elementary Data Link Protocols (2)



Implementation of the physical, data link, and network layers.

Elementary Data Link Protocols (3)

```
#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;
          . . .
```

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

Elementary Data Link Protocols (4)

```
/* 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); . . .
```

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

Elementary Data Link Protocols (5)

```
/* 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
```

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

Utopian Simplex Protocol (1)

```
/* 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 */
}
. . .
```

A utopian simplex protocol.

Utopian Simplex Protocol (2)

```
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 */
  }
}
```

A utopian simplex protocol.

Simplex Stop-and-Wait Protocol for a Noisy Channel (1)

```
/* Protocol 2 (Stop-and-wait) also provides for a one-directional flow of data from
   sender to receiver. The communication channel is once again assumed to be error
   free, as in protocol 1. However, this time the receiver has only a finite buffer
   capacity and a finite processing speed, so the protocol must explicitly prevent
   the sender from flooding the receiver with data faster than it can be handled. */

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 */
  }
}   . . .
```

A simplex stop-and-wait protocol.

Simplex Stop-and-Wait Protocol for a Noisy Channel (2)

```
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 */
  }
}
```

A simplex stop-and-wait protocol.

Sliding Window Protocols (1)

```
/* 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;


    . . .
```

A positive acknowledgement with retransmission protocol.

Sliding Window Protocols (2)

```
  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 */
          }
     }
  }
}
```

. . .

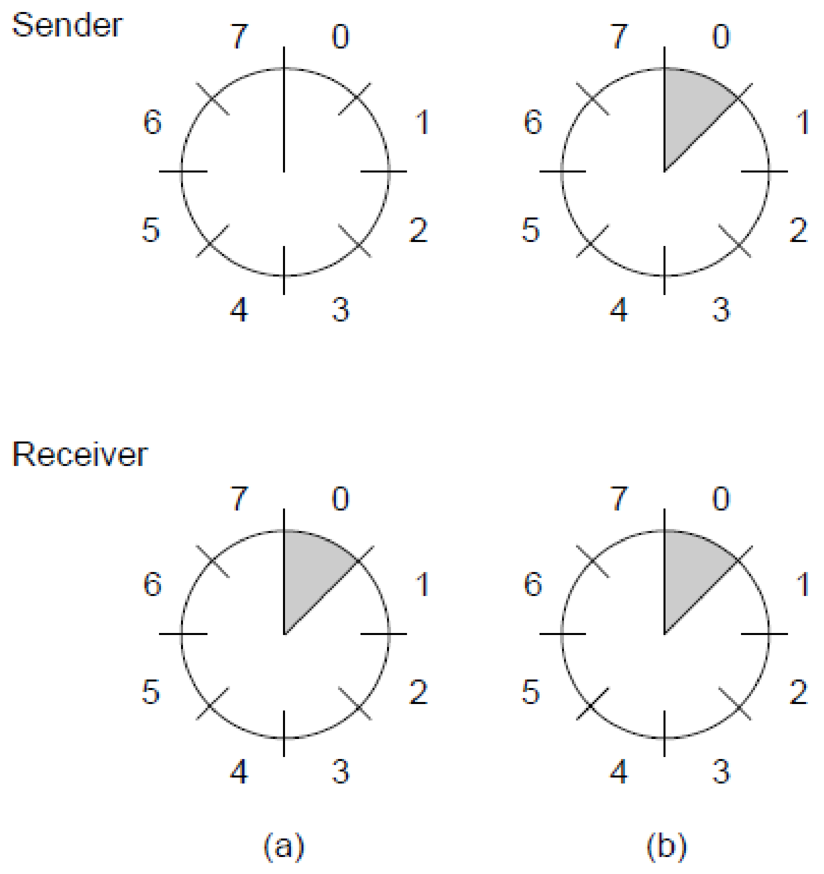
A positive acknowledgement with retransmission protocol.

Sliding Window Protocols (3)

```
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 */
      }
  }
}
```

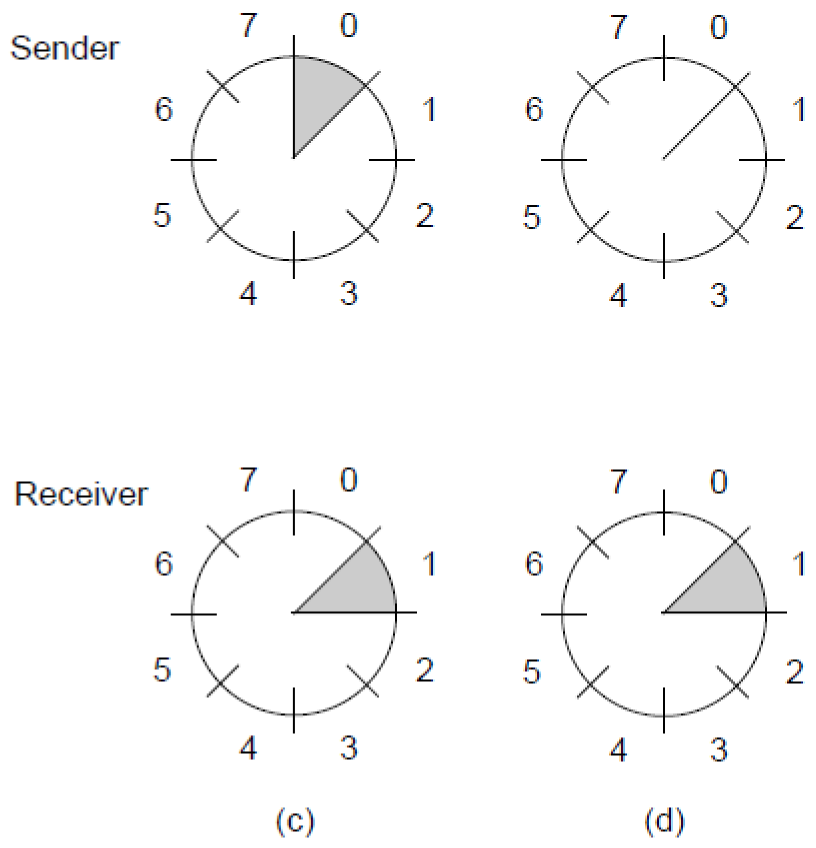
A positive acknowledgement with retransmission protocol.

Sliding Window Protocols (4)



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

Sliding Window Protocols (5)



A sliding window of size 1, with a 3-bit sequence number

(c) After the first frame has been received. (d) After the first acknowledgement has been received.

One-Bit Sliding Window Protocol (1)

```
/* 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 */
. . .
```

A 1-bit sliding window protocol.

One-Bit Sliding Window Protocol (2)

```
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 */
        }
    }

 . . .
```

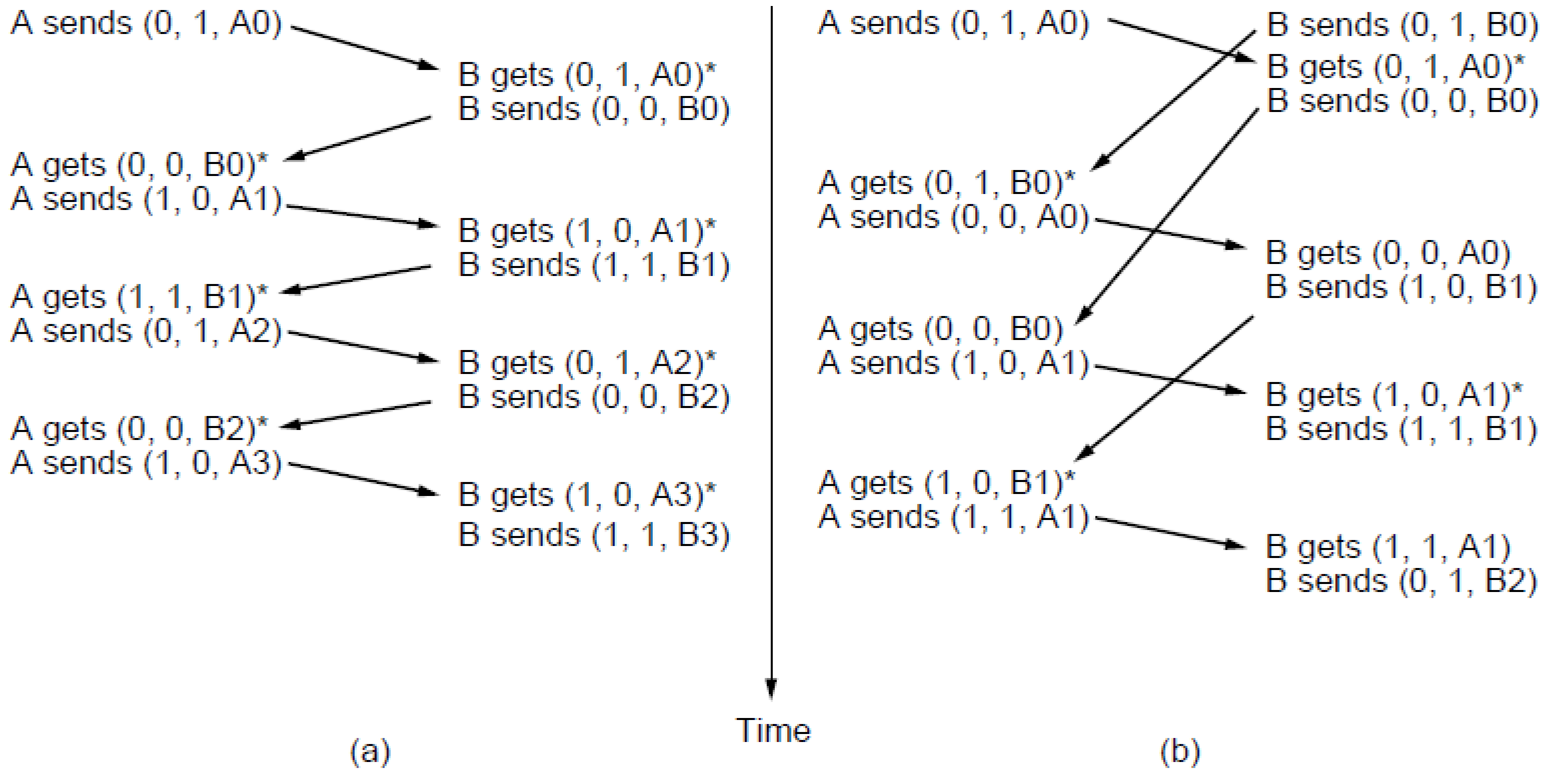
A 1-bit sliding window protocol.

One-Bit Sliding Window Protocol (3)

```
        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 */
   }
}
```

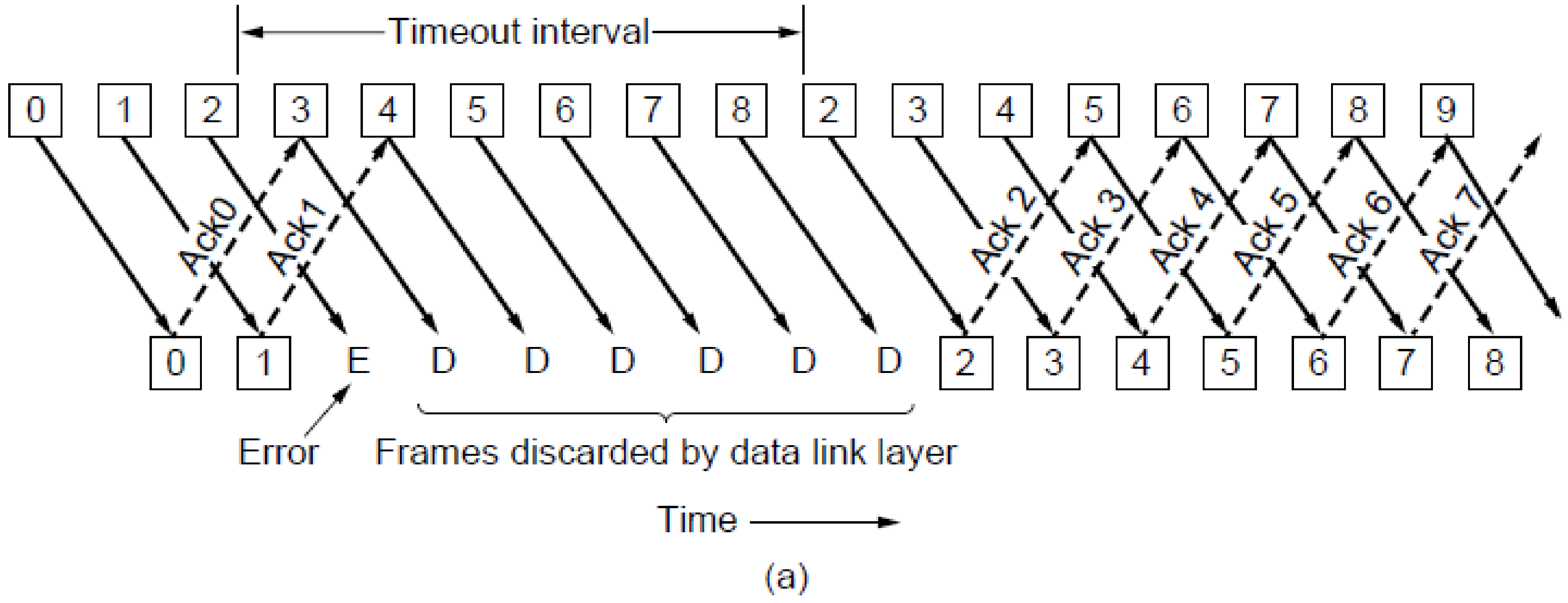
A 1-bit sliding window protocol.

One-Bit Sliding Window Protocol (4)



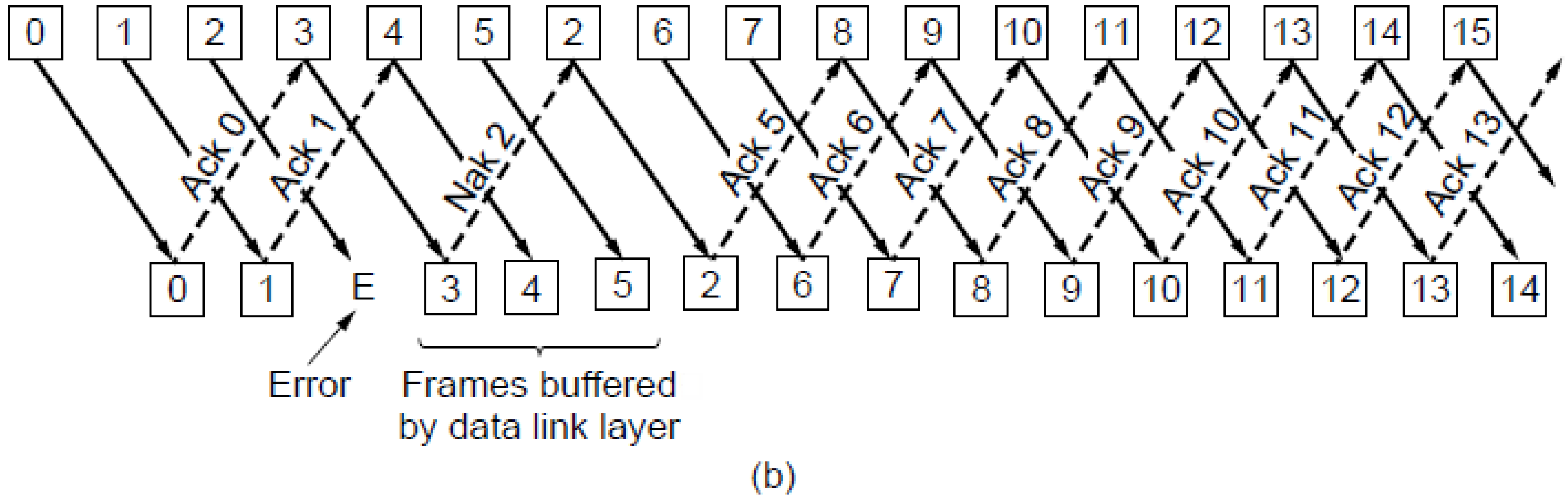
Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet

Protocol Using Go-Back-N (1)



Pipelining and error recovery. Effect of an error when (a) receiver's window size is 1

Protocol Using Go-Back-N (2)



Pipelining and error recovery. Effect of an error when (b) receiver's window size is large.

Protocol Using Go-Back-N (3)

```
/* Protocol 5 (Go-back-n) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike in the previous
   protocols, the network layer is not assumed to have a new packet all the time. Instead,
   the network layer causes a network_layer_ready event when there is a packet to send. */

#define MAX_SEQ 7
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Return true if a <= b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
        return(true);
    else
        return(false);
}
  . . .
```

A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (4)

```
static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data frame. */
  frame s;                                        /* scratch variable */

  s.info = buffer[frame_nr];                      /* insert packet into frame */
  s.seq = frame_nr;                               /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);/* piggyback ack */
  to_physical_layer(&s);                          /* transmit the frame */
  start_timer(frame_nr);                          /* start the timer running */
}
```

. . .

A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (5)

```
void protocol5(void)
{
  seq_nr next_frame_to_send;                 /* MAX_SEQ > 1; used for outbound stream */
  seq_nr ack_expected;                       /* oldest frame as yet unacknowledged */
  seq_nr frame_expected;                     /* next frame expected on inbound stream */
  frame r;                                   /* scratch variable */
  packet buffer[MAX_SEQ + 1];                /* buffers for the outbound stream */
  seq_nr nbuffered;                          /* number of output buffers currently in use */
  seq_nr i;                                  /* used to index into the buffer array */
  event_type event;


  . . .
```

A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (6)

```
enable_network_layer();                /* allow network_layer_ready events */
ack_expected = 0;                      /* next ack expected inbound */
next_frame_to_send = 0;                /* next frame going out */
frame_expected = 0;                    /* number of frame expected inbound */
nbuffered = 0;                         /* initially no packets are buffered */

while (true) {
   wait_for_event(&event);             /* four possibilities: see event_type above */

. . .
```

A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (7)

```
switch(event) {
  case network_layer_ready:                    /* the network layer has a packet to send */
       /* Accept, save, and transmit a new frame. */
       from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
       nbuffered = nbuffered + 1;              /* expand the sender's window */
       send_data(next_frame_to_send, frame_expected, buffer);/* transmit the frame */
       inc(next_frame_to_send);                /* advance sender's upper window edge */
       break;

  case frame_arrival:                          /* a data or control frame has arrived */
       from_physical_layer(&r);                /* get incoming frame from physical layer */

       if (r.seq == frame_expected) {
            /* Frames are accepted only in order. */
            to_network_layer(&r.info);         /* pass packet to network layer */
            inc(frame_expected);               /* advance lower edge of receiver's window */
        }

. . .
```

A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (8)

```
            /* Ack n implies n – 1, n – 2, etc.  Check for this. */
            while (between(ack_expected, r.ack, next_frame_to_send)) {
                 /* Handle piggybacked ack. */
                 nbuffered = nbuffered – 1;        /* one frame fewer buffered */
                 stop_timer(ack_expected);         /* frame arrived intact; stop timer */
                 inc(ack_expected);                /* contract sender's window */
            }
            break;

       case cksum_err: break;                      /* just ignore bad frames */

       case timeout:                               /* trouble; retransmit all outstanding frames */
            next_frame_to_send = ack_expected;         /* start retransmitting here */
            for (i = 1; i <= nbuffered; i++) {
                 send_data(next_frame_to_send, frame_expected, buffer);/* resend frame */
                 inc(next_frame_to_send);          /* prepare to send the next one */
            }

   }
. . .
```

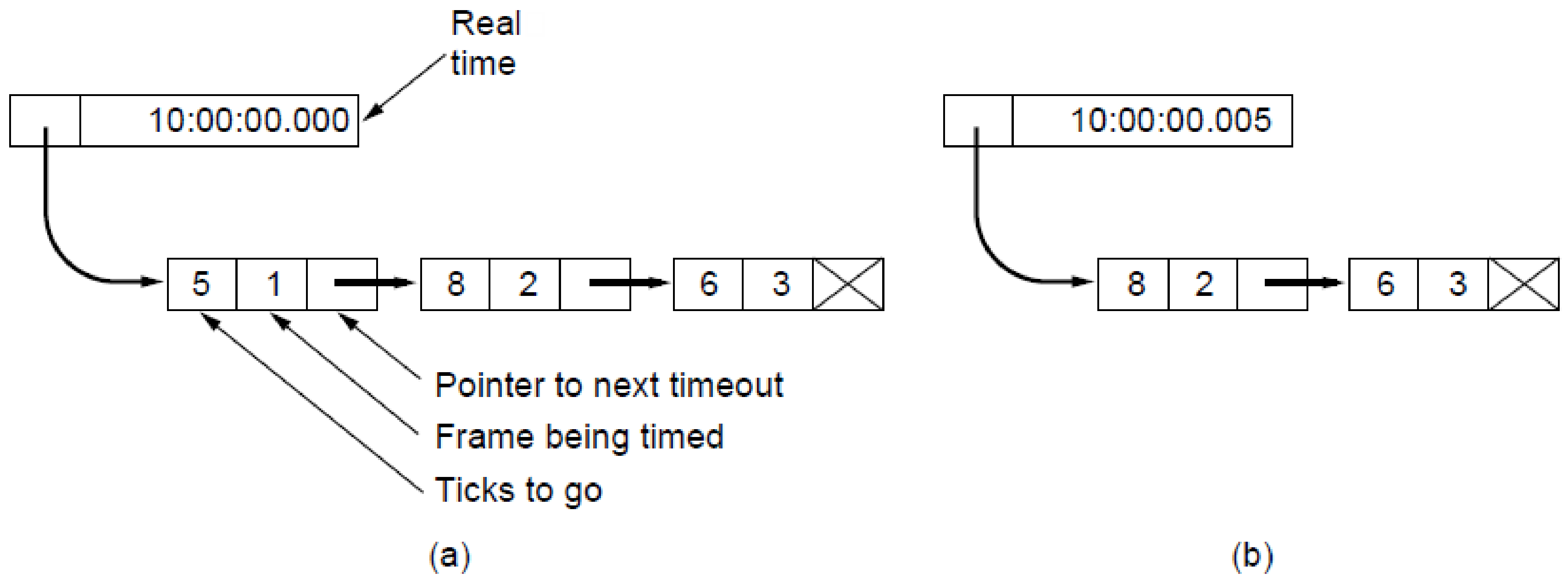
A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (9)

```
      if (nbuffered < MAX_SEQ)
              enable_network_layer();
      else
              disable_network_layer();
   }
 }
```

A sliding window protocol using go-back-n.

Protocol Using Go-Back-N (10)



Simulation of multiple timers in software. (a) The queued timeouts (b) The situation after the first timeout has expired.

Protocol Using Selective Repeat (1)

```
/* Protocol 6 (Selective repeat) accepts frames out of order but passes packets to the
   network layer in order. Associated with each outstanding frame is a timer. When the timer
   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

#define MAX_SEQ 7                                  /* should be 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                             /* no nak has been sent yet */
seq_nr oldest_frame = MAX_SEQ + 1;                 /* initial value is only for the simulator */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Same as between in protocol 5, but shorter and more obscure. */
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}
```

. . .

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (2)

```
static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data, ack, or nak frame. */
  frame s;                                       /* scratch variable */

  s.kind = fk;                                   /* kind == data, ack, or nak */
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;                              /* only meaningful for data frames */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;                 /* one nak per frame, please */
  to_physical_layer(&s);                         /* transmit the frame */
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();                              /* no need for separate ack frame */
}
```

. . .

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (3)

```
void protocol6(void)
{
  seq_nr ack_expected;                        /* lower edge of sender's window */
  seq_nr next_frame_to_send;                  /* upper edge of sender's window + 1 */
  seq_nr frame_expected;                      /* lower edge of receiver's window */
  seq_nr too_far;                             /* upper edge of receiver's window + 1 */
  int i;                                      /* index into buffer pool */
  frame r;                                    /* scratch variable */
  packet out_buf[NR_BUFS];                    /* buffers for the outbound stream */
  packet in_buf[NR_BUFS];                     /* buffers for the inbound stream */
  boolean arrived[NR_BUFS];                   /* inbound bit map */
  seq_nr nbuffered;                           /* how many output buffers currently used */
  event_type event;

. . .
```

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (4)

```
enable_network_layer();                         /* initialize */
ack_expected = 0;                               /* next ack expected on the inbound stream */
next_frame_to_send = 0;                         /* number of next outgoing frame */
frame_expected = 0;
too_far = NR_BUFS;
nbuffered = 0;                                  /* initially no packets are buffered */
for (i = 0; i < NR_BUFS; i++) arrived[i] = false;

. . .
```

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (5)

```
while (true) {
  wait_for_event(&event);                          /* five possibilities: see event_type above */
  switch(event) {
    case network_layer_ready:                      /* accept, save, and transmit a new frame */
         nbuffered = nbuffered + 1;                /* expand the window */
         from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
         send_frame(data, next_frame_to_send, frame_expected, out_buf);/* transmit the frame */
         inc(next_frame_to_send);                  /* advance upper window edge */
         break;
```

. . .

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (6)

```
case frame_arrival:                            /* a data or control frame has arrived */
     from_physical_layer(&r);                  /* fetch incoming frame from physical layer */
     if (r.kind == data) {
          /* An undamaged frame has arrived. */
          if ((r.seq != frame_expected) && no_nak)
            send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
          if (between(frame_expected,r.seq,too_far) && (arrived[r.seq%NR_BUFS]==false)) {
               /* Frames may be accepted in any order. */
               arrived[r.seq % NR_BUFS] = true;     /* mark buffer as full */
               in_buf[r.seq % NR_BUFS] = r.info;    /* insert data into buffer */
. . .
```

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (7)

```
            while (arrived[frame_expected % NR_BUFS]) {
                  /* Pass frames and advance window. */
                  to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                  no_nak = true;
                  arrived[frame_expected % NR_BUFS] = false;
                  inc(frame_expected);      /* advance lower edge of receiver's window */
                  inc(too_far);             /* advance upper edge of receiver's window */
                  start_ack_timer();        /* to see if a separate ack is needed */
            }
      }
}
  . . .
```

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (8)

```
        if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
              send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

        while (between(ack_expected, r.ack, next_frame_to_send)) {
              nbuffered = nbuffered – 1;          /* handle piggybacked ack */
              stop_timer(ack_expected % NR_BUFS);      /* frame arrived intact */
              inc(ack_expected);                  /* advance lower edge of sender's window */
        }
        break;
  case cksum_err:
        if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* damaged frame */
        break;
   . . .
```

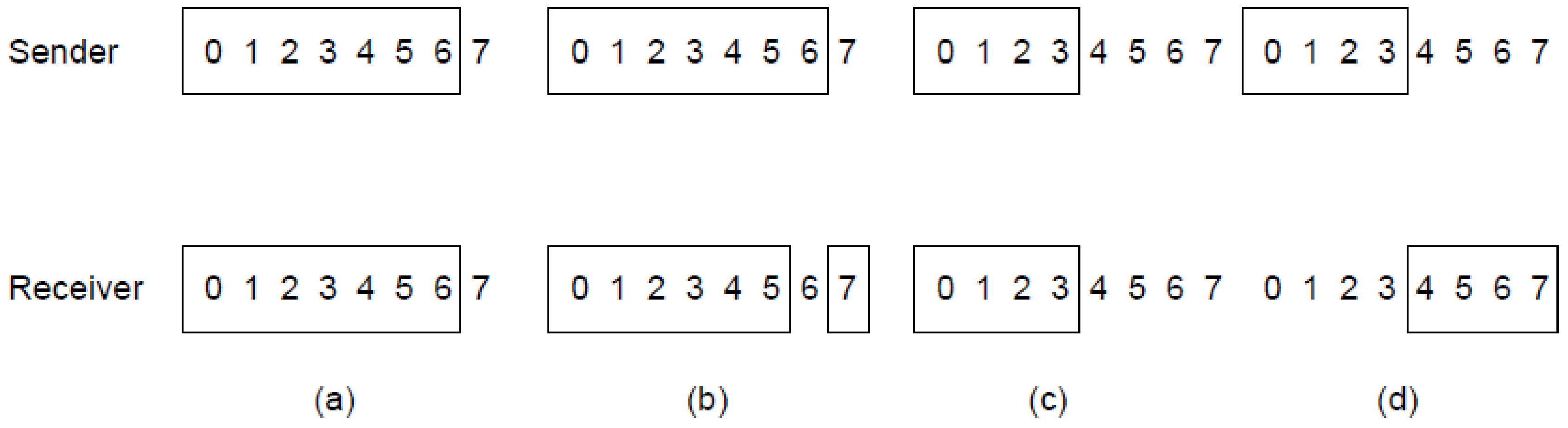
A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (9)

```
       case timeout:
            send_frame(data, oldest_frame, frame_expected, out_buf); /* we timed out */
            break;
       case ack_timeout:
            send_frame(ack,0,frame_expected, out_buf);      /* ack timer expired; send ack */
     }
     if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
   }
 }
```

A sliding window protocol using selective repeat.

Protocol Using Selective Repeat (10)

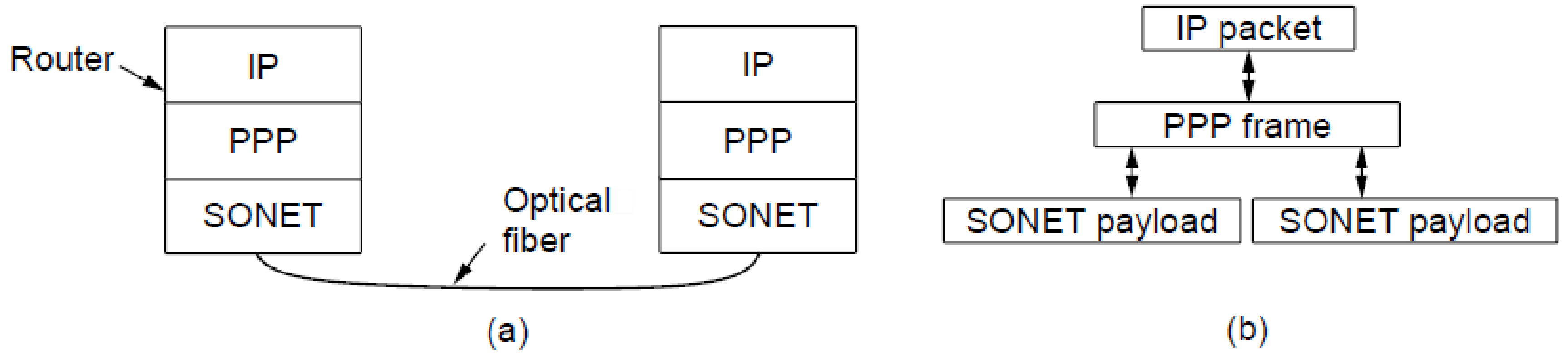


a) Initial situation with a window of size7
b) After 7 frames sent and received but not acknowledged.
c) Initial situation with a window size of 4.
d) After 4 frames sent and received but not acknowledged.

Example Data Link Protocols

1. Packet over SONET
2. ADSL (Asymmetric Digital Subscriber Loop)

Packet over SONET (1)



Packet over SONET. (a) A protocol stack. (b) Frame relationships

Packet over SONET (2)

PPP Features

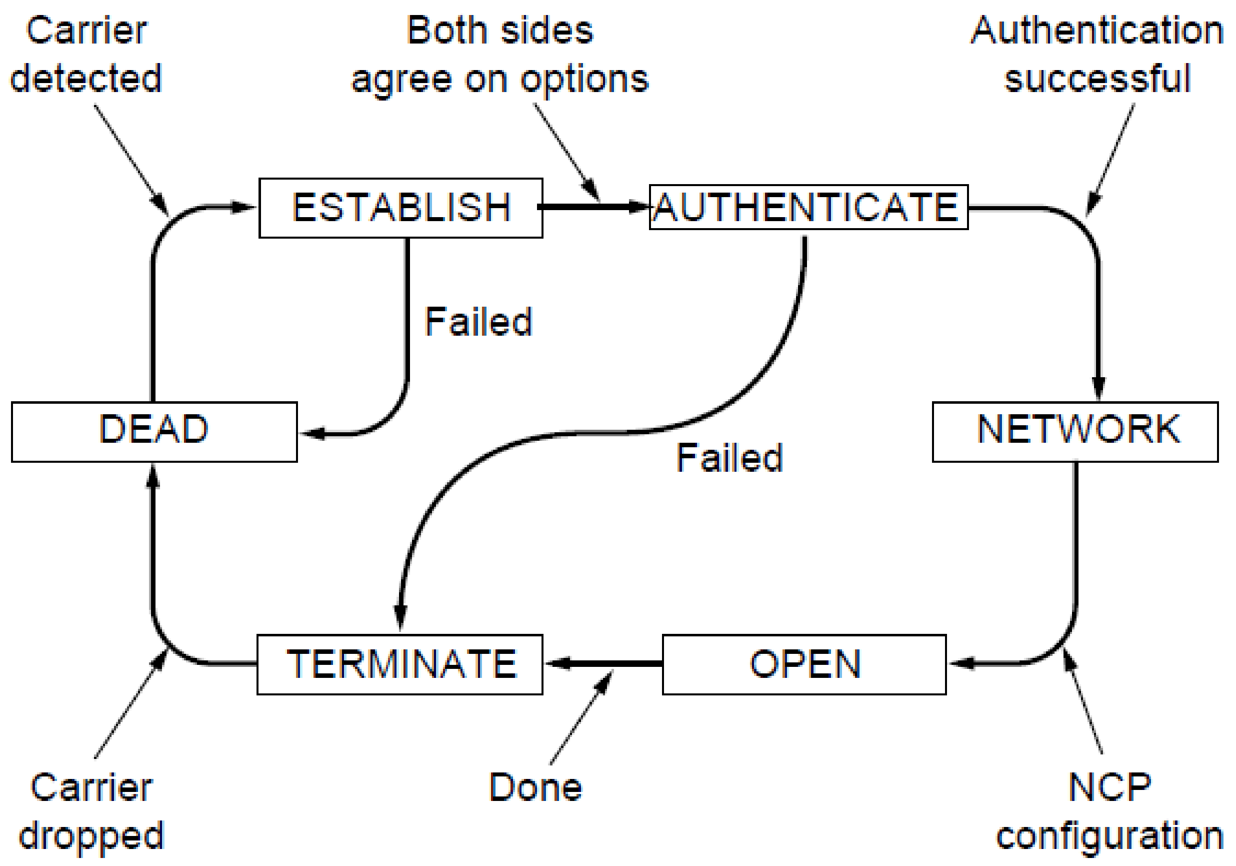
1. Separate packets, error detection
2. Link Control Protocol
3. Network Control Protocol

Packet over SONET (3)

Bytes	1	1	1	1 or 2	Variable	2 or 4	1
	Flag 01111110	Address 11111111	Control 00000011	Protocol	Payload	Checksum	Flag 01111110

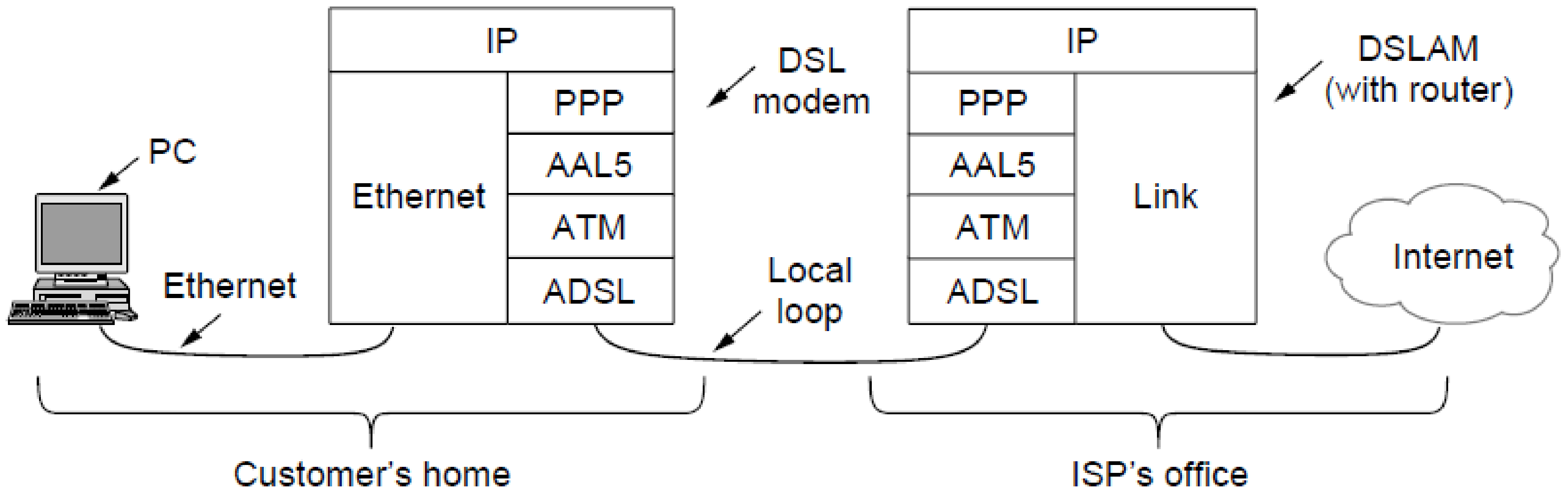
The PPP full frame format for unnumbered mode operation

Packet over SONET (4)



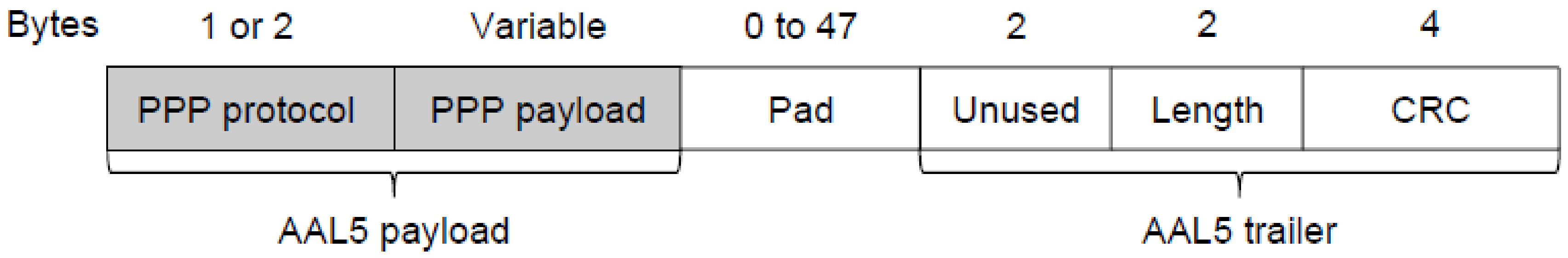
State diagram for bringing a PPP link up and down

ADSL (Asymmetric Digital Subscriber Loop) (1)



ADSL protocol stacks.

ADSL (Asymmetric Digital Subscriber Loop) (1)



AAL5 frame carrying PPP data

End

Chapter 3