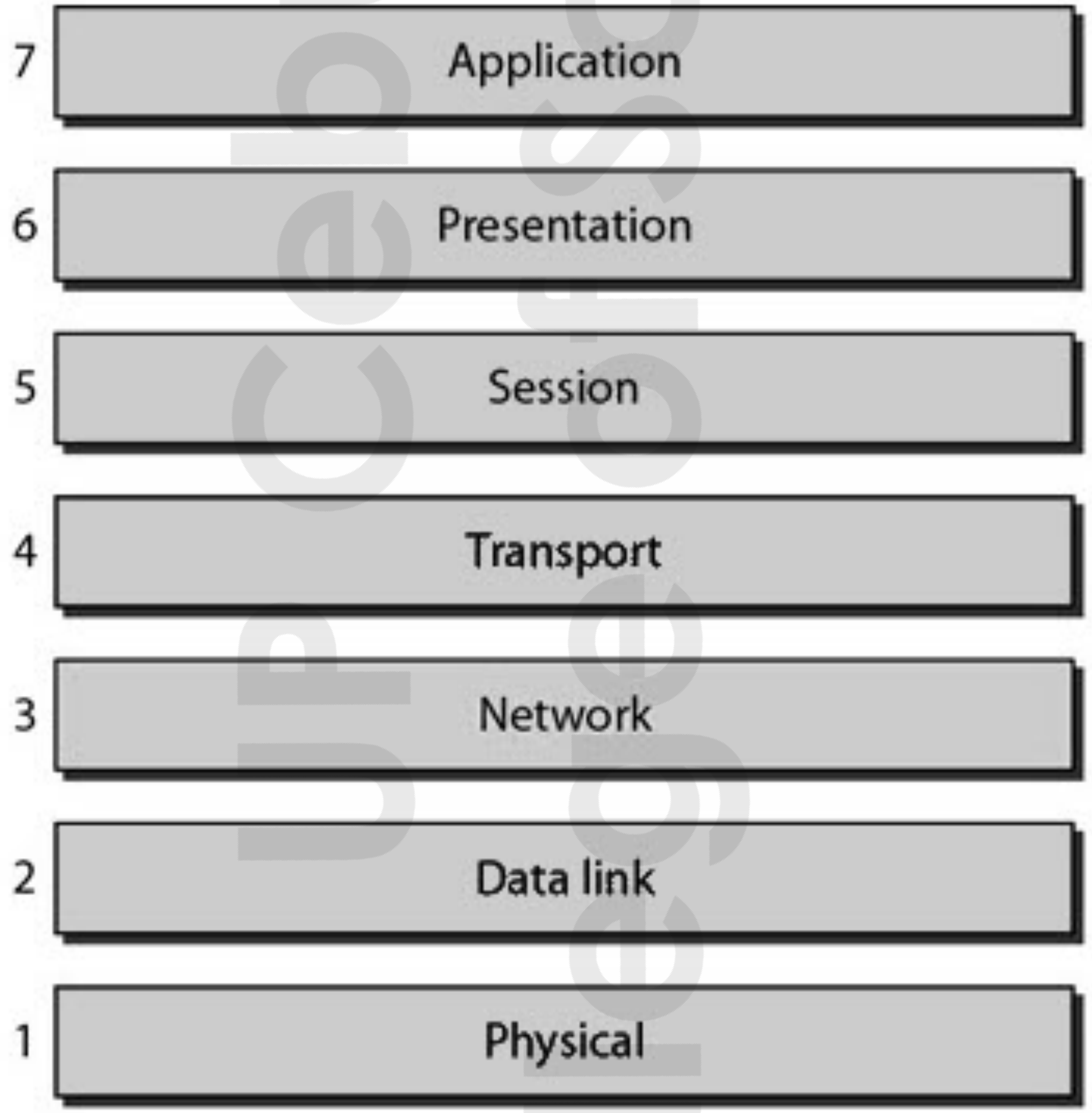
The Data Link Layer

Prof. Dhong Fhel K. Gom-os

Contents

- Data Link Protocols
- Data Link Layer Example

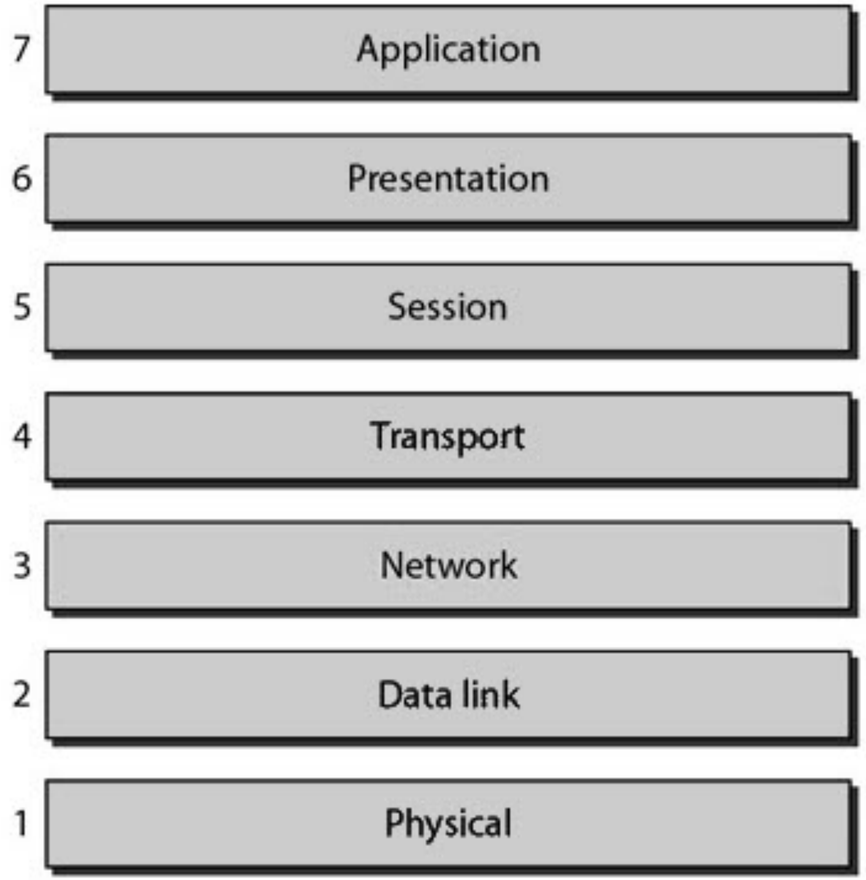


The Physical Layer is concerned of…

- Physical characteristics of interfaces and medium
- Representation of bits for transmission
- Data rate
- Synchronisation of bits
- Line configuration
- Physical topology
- Transmission mode

Data Link Layer

- Layer 2 in the OSI Model
- Transforms the physical layer, a raw transmission facility, to a link responsible for node-to-node (hop-to-hop) communication



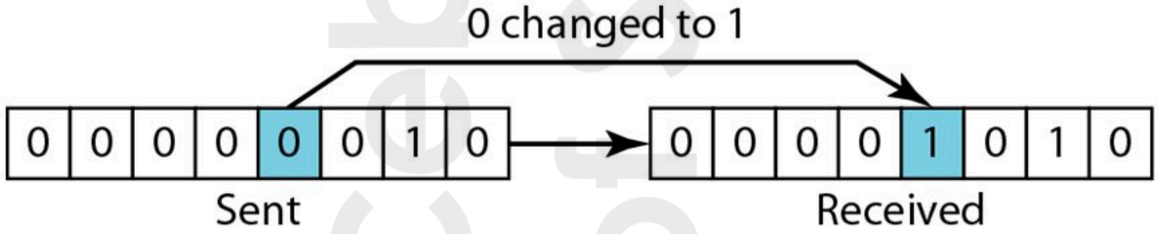
Specific Responsibilities of the Data Link Layer

1. **Framing** - It divides the stream of bits received from the network layer into manageable data units called *frames*.
2. **Addressing** - It adds a header to the frame to define the addresses of the sender and receiver of the frame.
3. **Flow Control** - If the rate at which the data are absorbed by the receiver is less than the rate at which data are produced in the sender, the data link layer imposes a flow control mechanism to avoid overwhelming the receiver.
4. **Error Control** - It adds reliability to the physical layer by adding mechanisms to detect and retransmit damaged, duplicate, or lost frames.
5. **Media Access Control** - When two or more devices are connected to the same link, data link layer protocols are necessary to determine which device has control over the link at any given time.

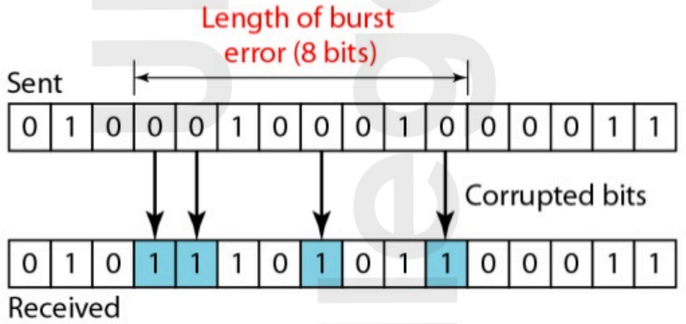
Data can be corrupted during transmission. Some applications can tolerate a small level of error. Give examples of these applications.

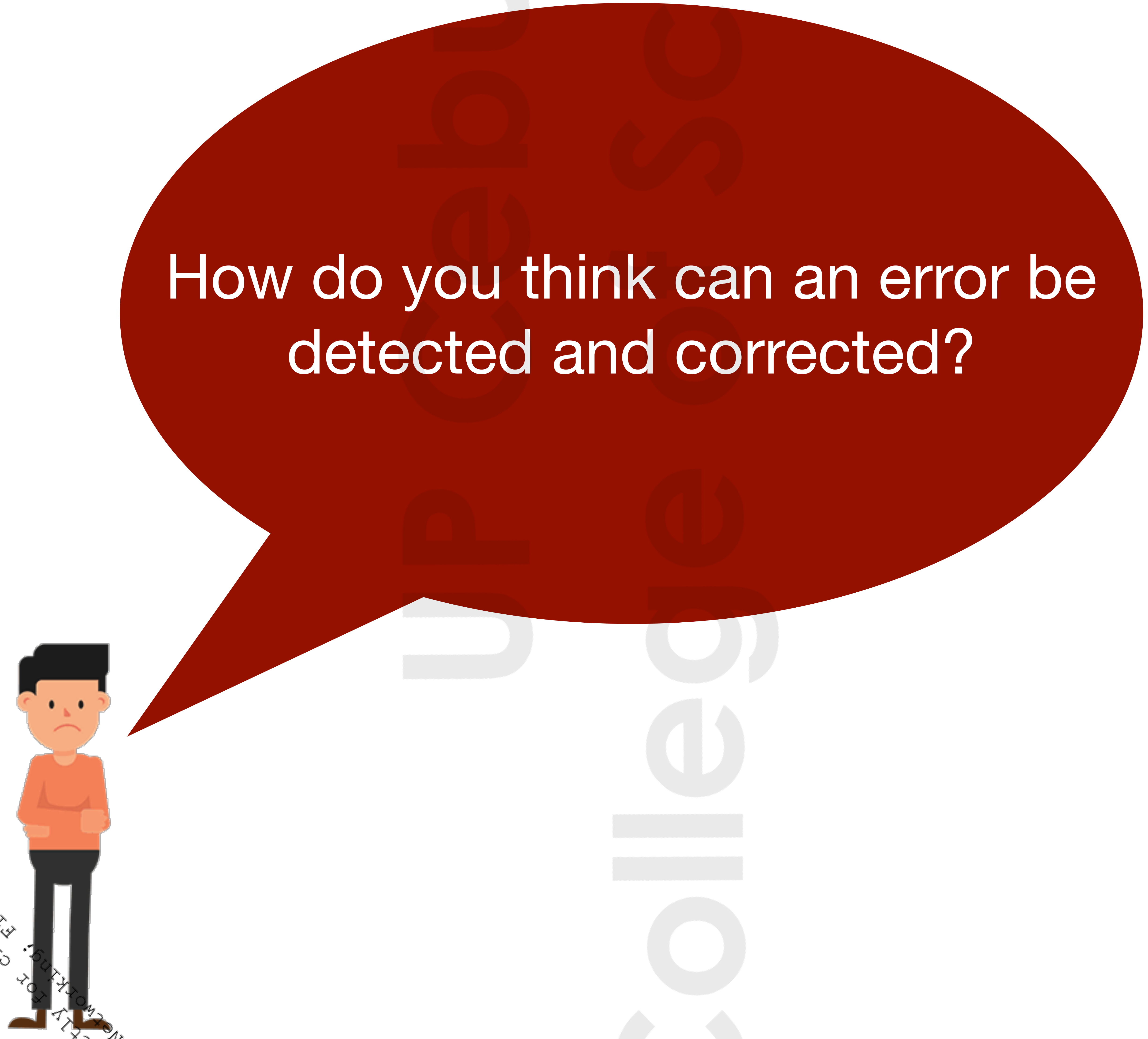
Types of Errors

1. **Single-Bit Error** - only 1 bit of a given data unit (byte, character or packet) is changed from 1 to 0 or from 0 to 1. Impact on transmitted data.



2. **Burst Error** - two or more bits in the data unit have changed. Not necessarily consecutive bits.





Redundancy

- The central concept in detecting or correcting errors.
- Need to send some extra (redundant) bits with data.
- Redundant bits are added by the sender and removed by the receiver allowing receiver to detect and correct corrupted bits.

Detection versus Correction

- **Error Detection** - looking only to see if any error has occurred. Not interested in the number of errors. Single-bit error is the same as a burst error.
- **Error Correction** - need to detect and correct. Need to know the number and location of errors, and the size of the data unit. Example, to correct one single error in an 8-bit data unit, need to consider 8 possible error locations; to correct 2 errors of same data unit size, need to consider 28 possibilities.

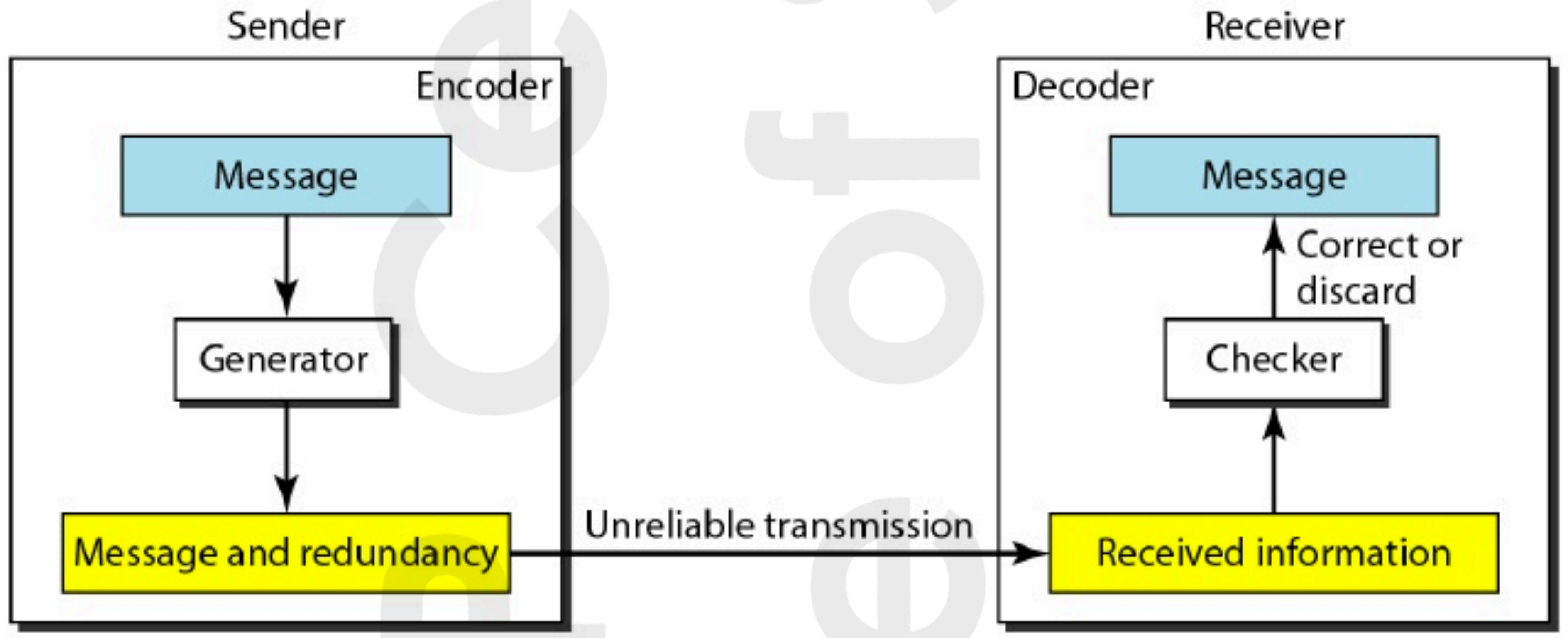
$$^{n}C_{r} = \frac{n!}{r!(n-r)!}$$

Two Methods of Error Correction

1. **Forward Error Correction** - the process in which the receiver tries to guess the message by using redundant bits. Ideal if the number of errors is small.
2. **Correction By Retransmission** - a technique in which the receiver detects the occurrence of an error and asks the sender to resend the message. Repeated until receiver believes message is error-free.

Coding

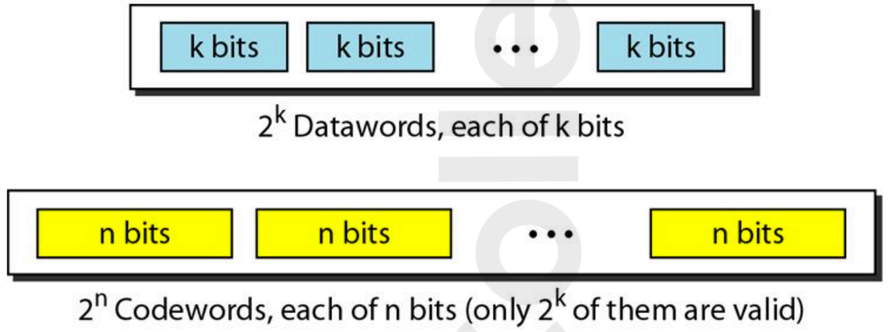
- Sender adds redundant bits through a process that creates a relationship between the redundant bits and the actual data bits.
- The receiver checks this relationship.



- Two schemes:
  1. Block Coding
  2. *Convolution Coding*

Block Coding

- How it is done:
  1. Divide message into blocks, each of *k* bits, called *datawords*
  2. Add redundant bits to each block to make the length $n = k + r$ (*codeword*)
- With *k* bits, can create a combination of 2^k data words and with *n* bits, can create 2^n codewords
- Since $n > k$, so number of possible codewords > datawords
- The block coding process is one-to-one; the same dataword is encoded as codeword.
- $2^n - 2^k$ codewords are not used (invalid or illegal)

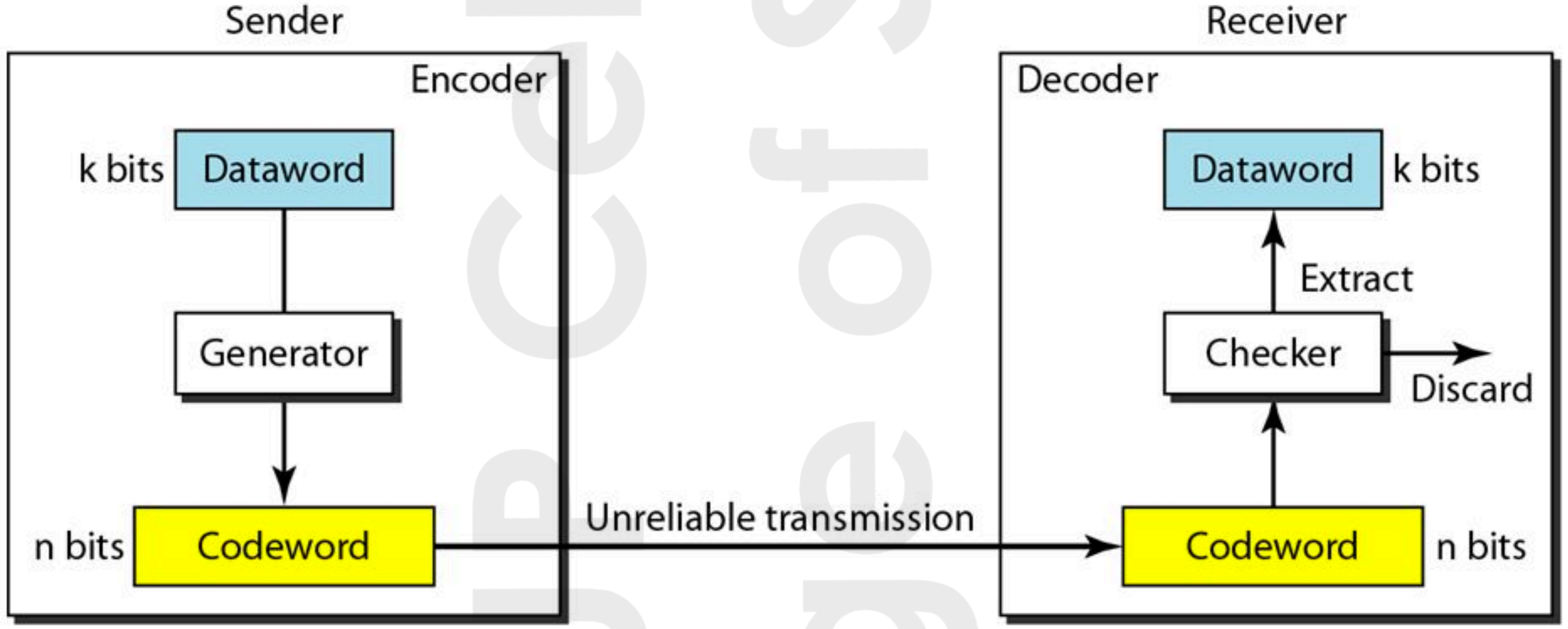


Example 10.1

The 4B/5B block coding discussed in Chapter 4 is a good example of this type of coding. In this coding scheme, $k = 4$ and $n = 5$. As we saw, we have $2^k = 16$ datawords and $2^n = 32$ codewords. We saw that 16 out of 32 codewords are used for message transfer and the rest are either used for other purposes or unused.

How Errors are Detected Using Block Coding?

1. The receiver has (or can find) a list of valid codewords.
2. The original codeword has changed to an invalid one.



- This coding can detect only single errors.
- Two or more errors may remain undetected.

Example 10.2

Let us assume that $k = 2$ and $n = 3$. Table 10.1 shows the list of datawords and codewords. Later, we will see how to derive a codeword from a dataword.

Table **10.1** *A code for error detection (Example 10.2)*

Datawords	*Codewords*
00	000
01	011
10	101
11	110

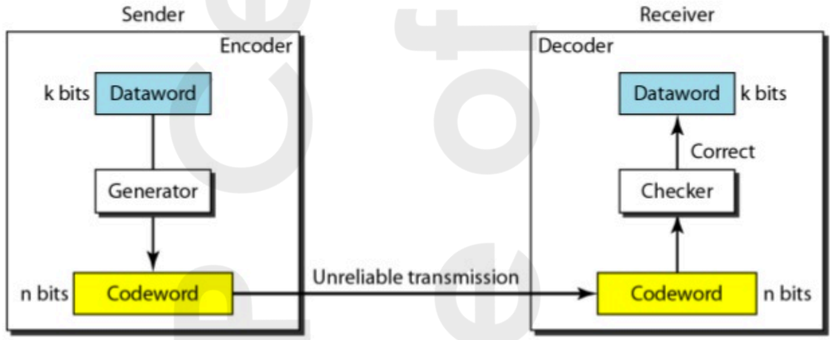
Assume the sender encodes the dataword 01 as 011 and sends it to the receiver. Consider the following cases:

1. The receiver receives 011. It is a valid codeword. The receiver extracts the dataword 01 from it.
2. The codeword is corrupted during transmission, and 111 is received (the leftmost bit is corrupted). This is not a valid codeword and is discarded.
3. The codeword is corrupted during transmission, and 000 is received (the right two bits are corrupted). This is a valid codeword. The receiver incorrectly extracts the dataword 00. Two corrupted bits have made the error undetectable.

An error-detecting code can detect only the types of errors for which it is designed; other types of errors may remain undetected.

How Errors are Corrected Using Block Coding?

- Receiver needs to find (or guess) the original codeword sent
- There's a need of more redundant bits for error correction and for detection



Example 10.3

Let us add more redundant bits to Example 10.2 to see if the receiver can correct an error without knowing what was actually sent. We add 3 redundant bits to the 2-bit dataword to make 5-bit codewords. Again, later we will show how we chose the redundant bits. For the moment let us concentrate on the error correction concept. Table 10.2 shows the datawords and codewords.

Assume the dataword is 01. The sender consults the table (or uses an algorithm) to create the codeword 01011. The codeword is corrupted during transmission, and 01001 is received (error in the second bit from the right). First, the receiver finds that the received codeword is not in the table. This means an error has occurred. (Detection must come before correction.) The receiver, assuming that there is only 1 bit corrupted, uses the following strategy to guess the correct dataword.

Table 10.2 *A code for error correction (Example 10.3)*

Dataword	*Codeword*
00	00000
01	01011
10	10101
11	11110

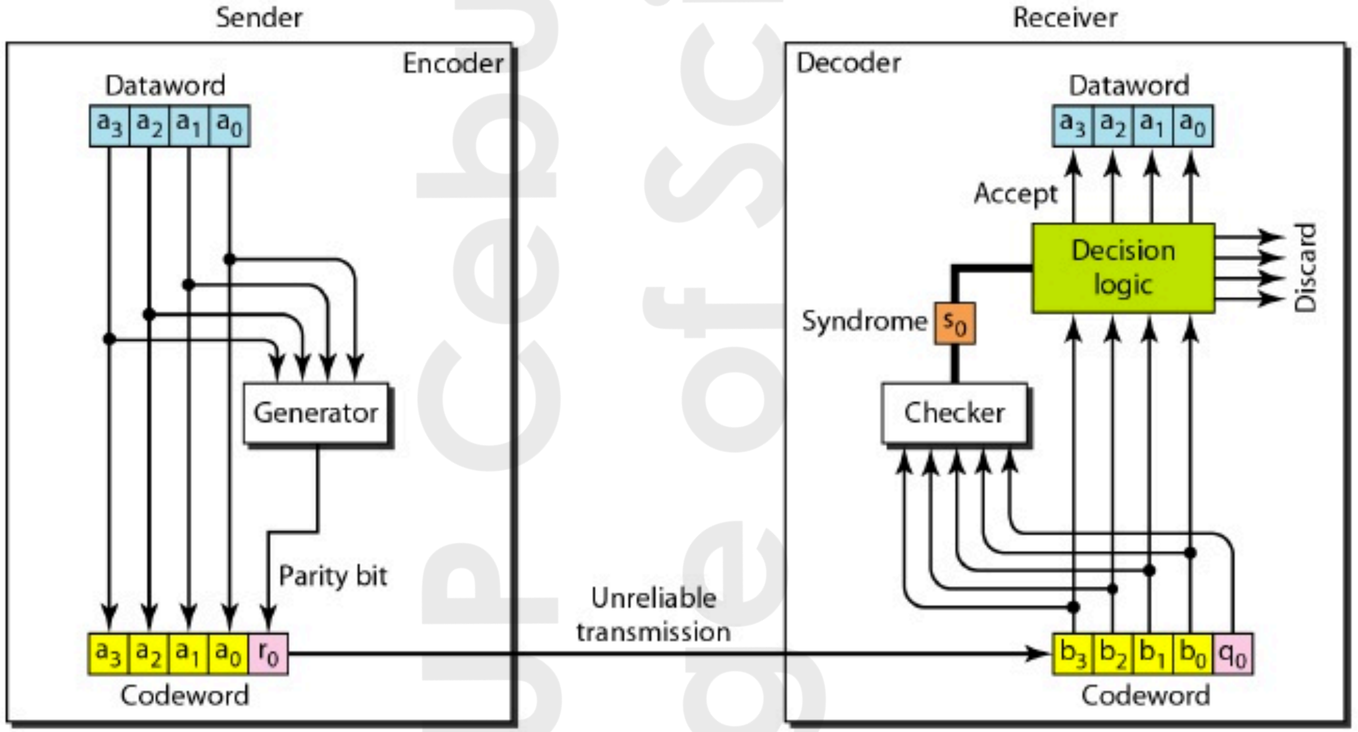
1. Comparing the received codeword with the first codeword in the table (01001 versus 00000), the receiver decides that the first codeword is not the one that was sent because there are two different bits.
2. By the same reasoning, the original codeword cannot be the third or fourth one in the table.
3. The original codeword must be the second one in the table because this is the only one that differs from the received codeword by 1 bit. The receiver replaces 01001 with 01011 and consults the table to find the dataword 01.

Simple Parity-Check Code

- A *k-bit* dataword is changed to an n-bit codeword where $n = k + 1$
- The extra bit, called the parity bit, is selected to make the total number of 1s in the codeword even (even parity)
- A simple parity-check code can detect an odd number of errors

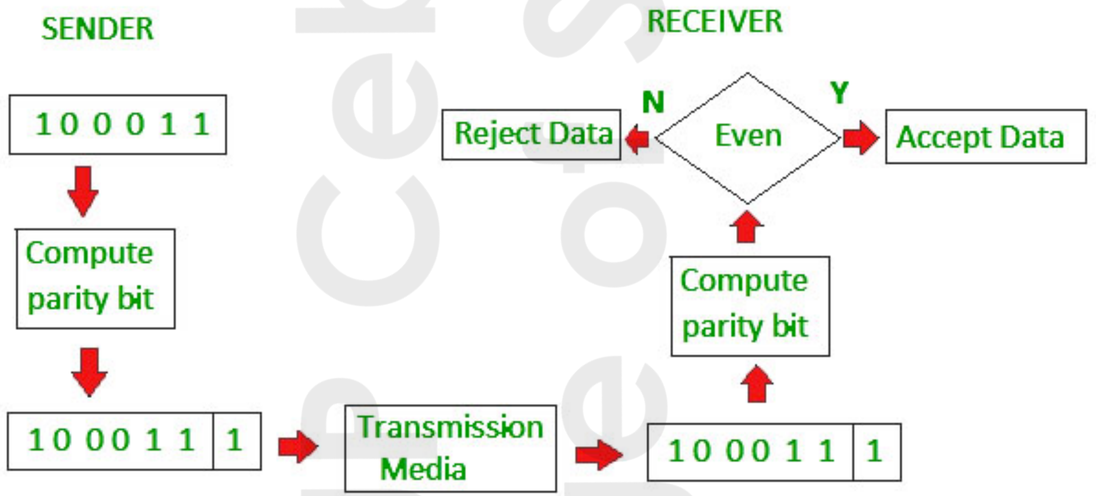
Table 10.3 *Simple parity-check code C(5, 4)*

Datawords	Codewords	Datawords	Codewords
0000	00000	1000	10001
0001	00011	1001	10010
0010	00101	1010	10100
0011	00110	1011	10111
0100	01001	1100	11000
0101	01010	1101	11011
0110	01100	1110	11101
0111	01111	1111	11110



$$r_0 = a_3 + a_2 + a_1 + a_0$$

$$s_0 = b_3 + b_2 + b_1 + b_0 + q_0$$

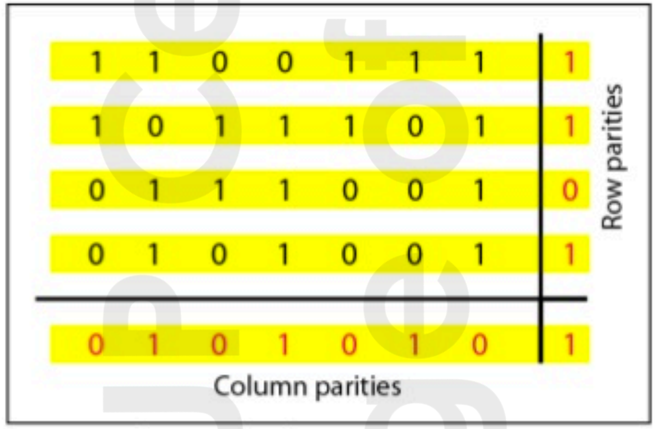


Example 10.7

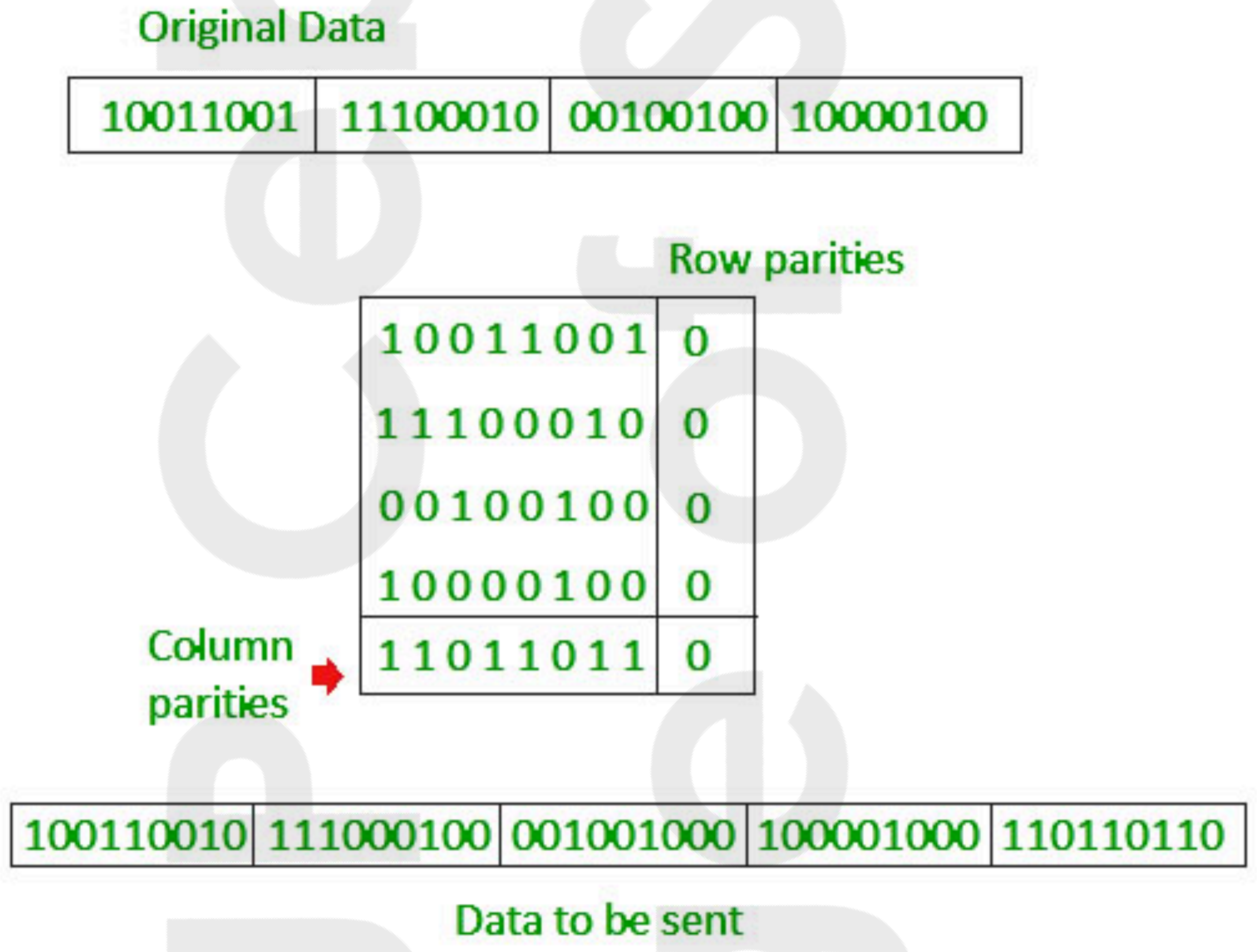
Let us look at some transmission scenarios. Assume the sender sends the dataword 10110. The codeword created from this dataword is 10111, which is sent to the receiver. We examine five cases:

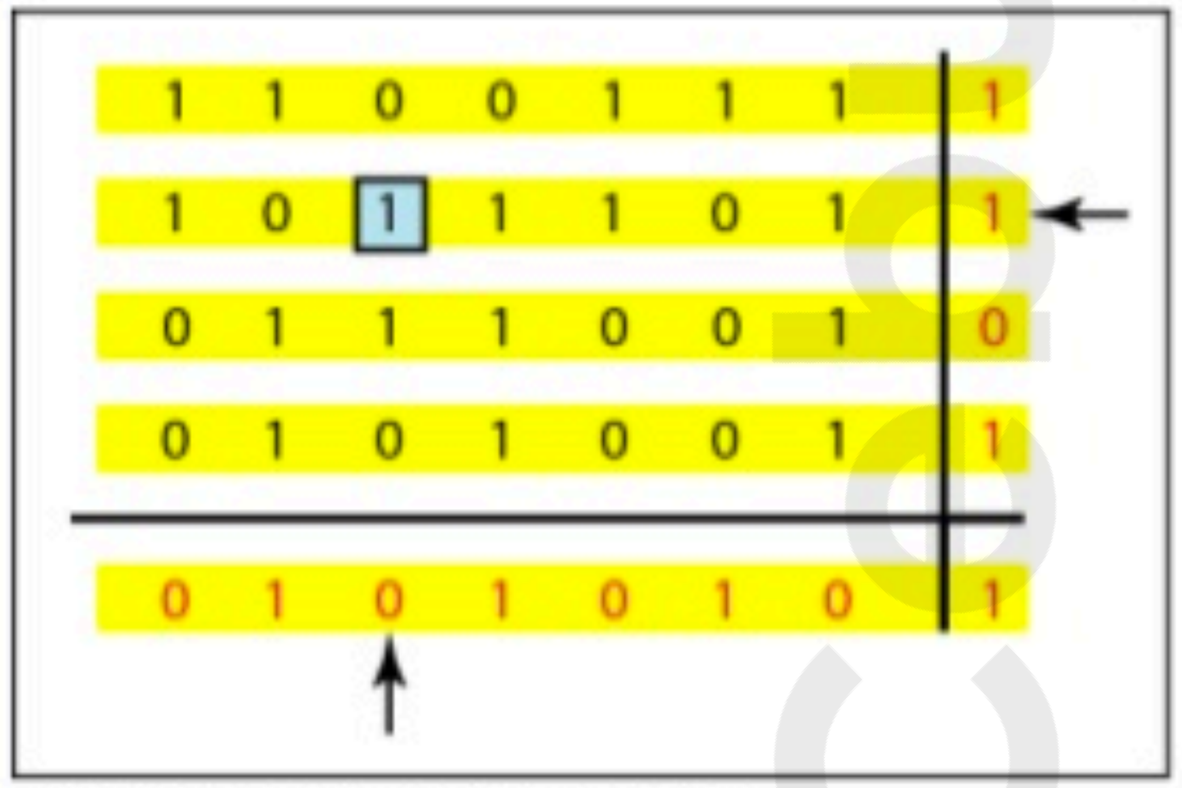
1. No error occurs; the received codeword is 10111. The syndrome is 0. The dataword 1011 is created.
2. One single-bit error changes a_1. The received codeword is 10011. The syndrome is 1. No dataword is created.
3. One single-bit error changes r_0. The received codeword is 10110. The syndrome is 1. No dataword is created. Note that although none of the dataword bits are corrupted, no dataword is created because the code is not sophisticated enough to show the position of the corrupted bit.
4. An error changes r_0 and a second error changes a_3. The received codeword is 00110. The syndrome is 0. The dataword 0011 is created at the receiver. Note that here the dataword is wrongly created due to the syndrome value. The simple parity-check decoder cannot detect an even number of errors. The errors cancel each other out and give the syndrome a value of 0.
5. Three bits—a_3, a_2, and a_1—are changed by errors. The received codeword is 01011. The syndrome is 1. The dataword is not created. This shows that the simple parity check, guaranteed to detect one single error, can also find any odd number of errors.

A better approach is the two dimensional parity check...

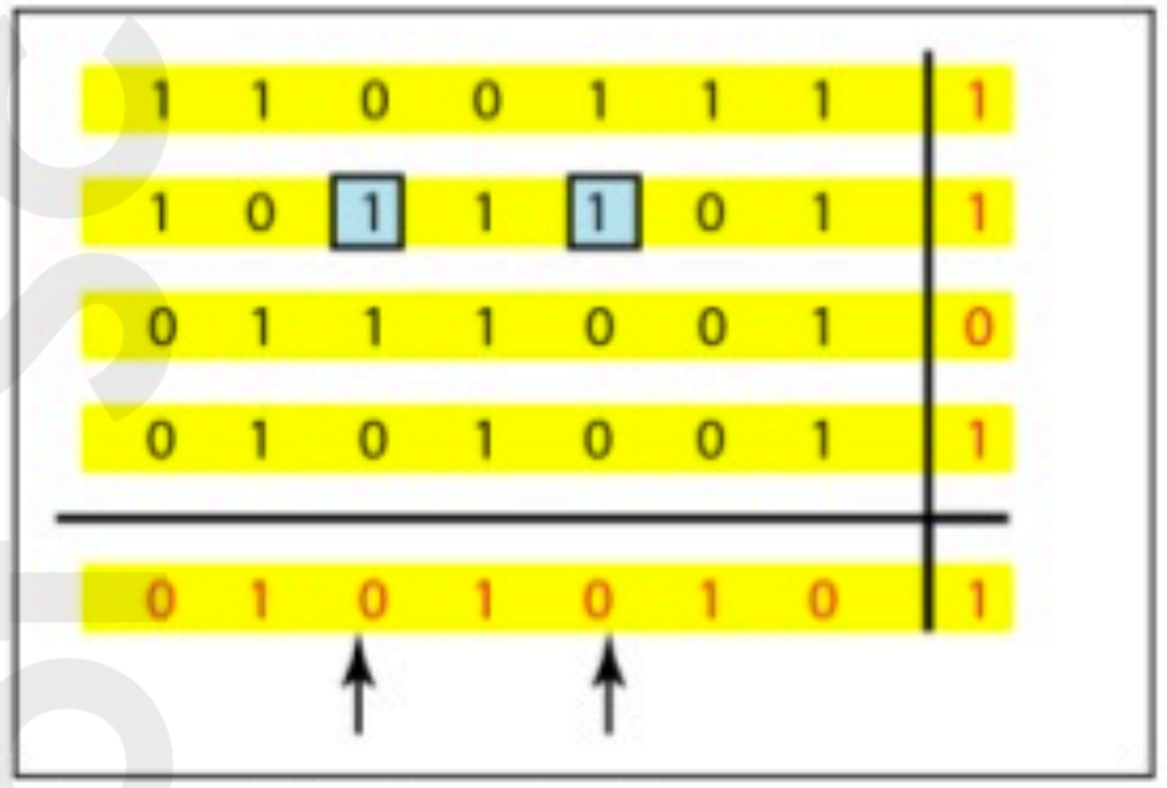


a. Design of row and column parities

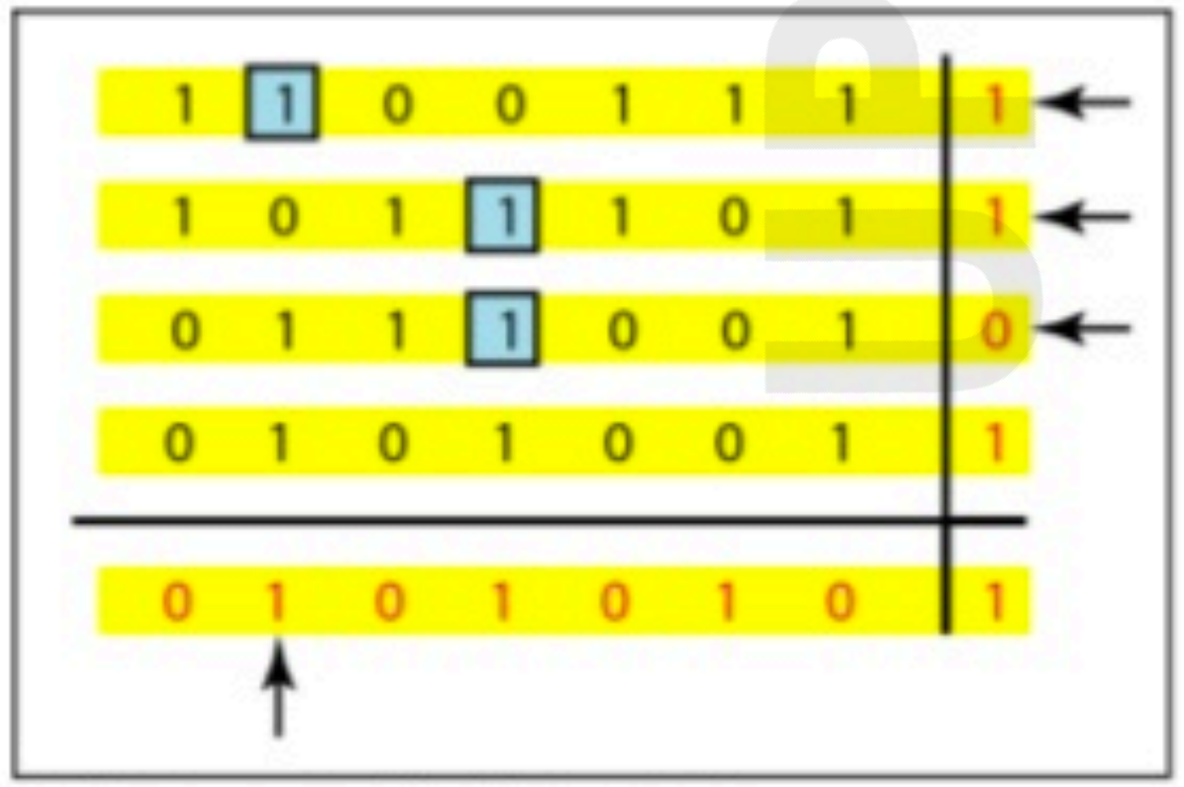




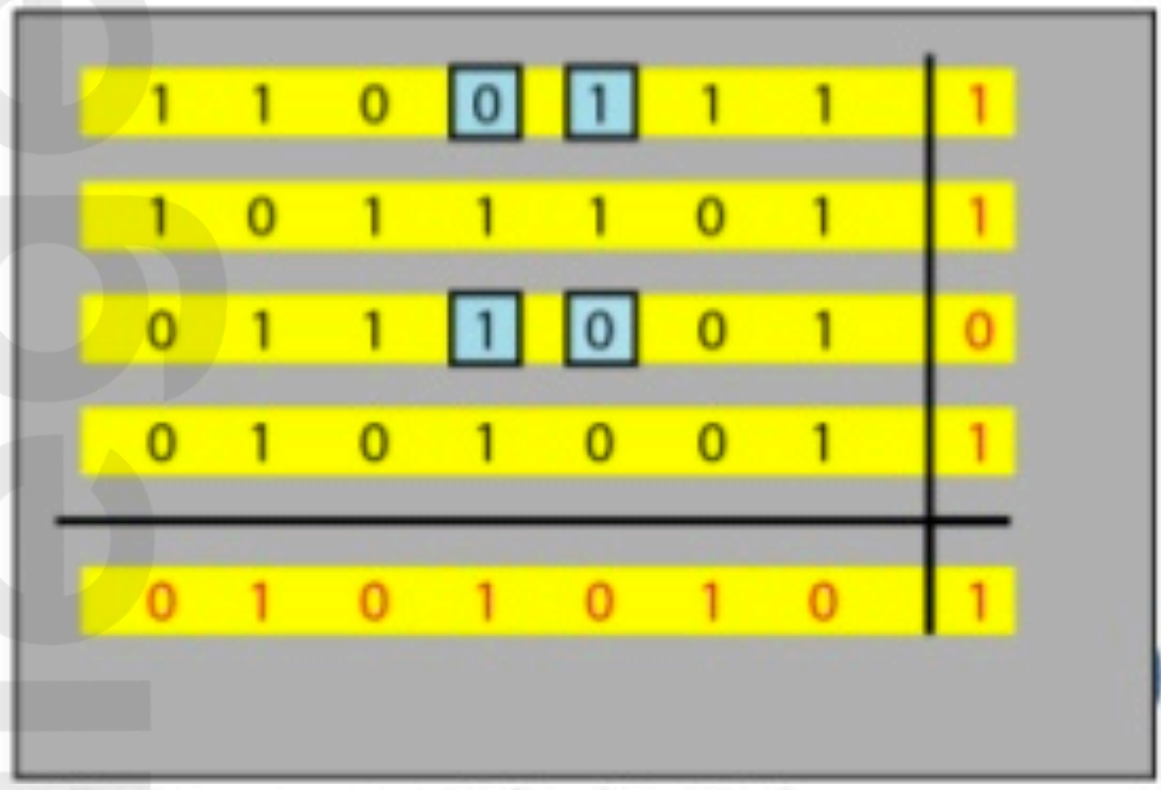
b. One error affects two parities



c. Two errors affect two parities



d. Three errors affect four parities



e. Four errors cannot be detected

Checksum

- Also known as hash sum.
- Created by calculating the binary values in a packet or other block of data using some algorithm (based on addition) and storing the results with the data.
- When the data is received at the other end, a new checksum is calculated and compared with the existing checksum.
- A non-match indicates an error; a match does not necessarily mean the absence of errors, but only that the simple algorithm was not able to detect any.
- Among the types of errors that cannot be detected by simple checksum algorithms are reordering of the bytes and multiple errors that cancel each other out.

Example 10.18

Suppose our data is a list of five 4-bit numbers that we want to send to a destination. In addition to sending these numbers, we send the sum of the numbers. For example, if the set of numbers is (7, 11, 12, 0, 6), we send (7, 11, 12,0,6,36), where 36 is the sum of the original numbers. The receiver adds the five numbers and compares the result with the sum. If the two are the same, the receiver assumes no error, accepts the five numbers, and discards the sum. Otherwise, there is an error somewhere and the data are not accepted.

- Scheme:
  - The data is divided into k segments each of m bits.
  - In the sender's end the segments are added using 1's complement arithmetic to get the sum. The sum is complemented to get the checksum.
  - The checksum segment is sent along with the data segments.
  - At the receiver's end, all received segments are added using 1's complement arithmetic to get the sum. The sum is complemented.
  - If the result is zero, the received data is accepted; otherwise discarded.

Original Data

10011001	11100010	00100100	10000100
1	2	3	4

k=4, m=8

Sender

```
1  10011001
2  11100010
  (1)01111011
        1
   01111100
3  00100100
   10100000
4  10000100
  (1)00100100
        1
Sum: 00100101
CheckSum: 11011010
```

Reciever

```
1  10011001
2  11100010
  (1)01111011
        1
   01111100
3  00100100
   10100000
4  10000100
  (1)00100100
        1
   00100101
   11011010
Sum: 11111111
Complement: 00000000
```

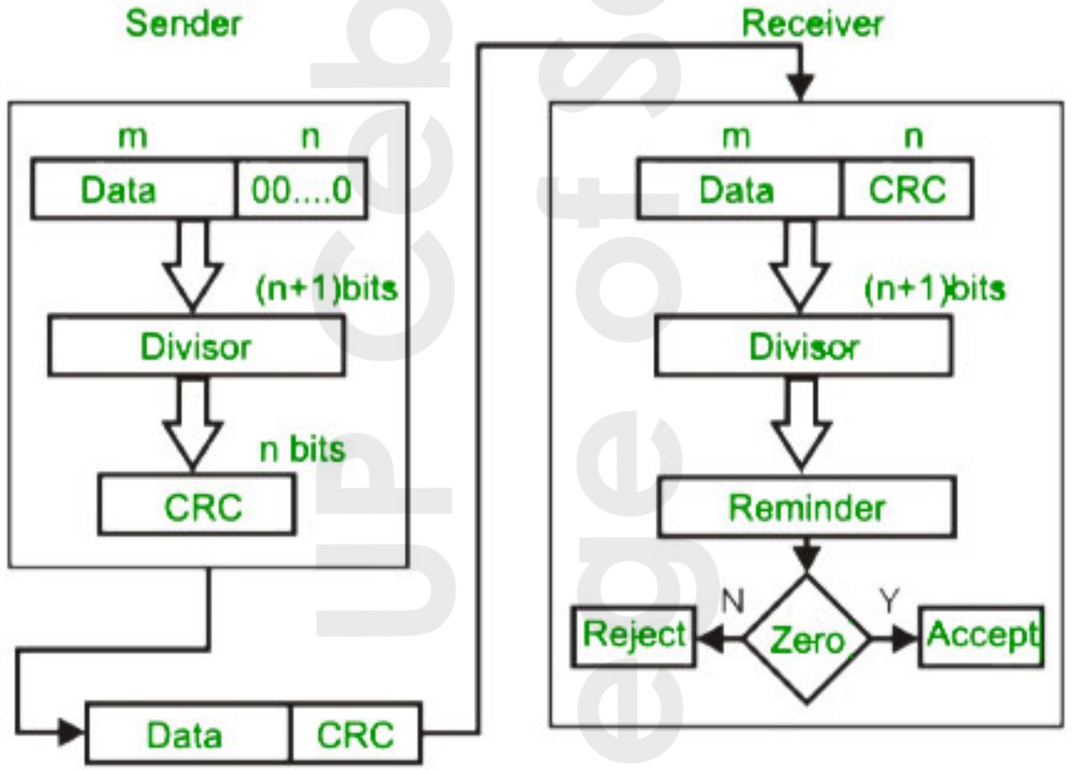
Conclusion: Accept Data

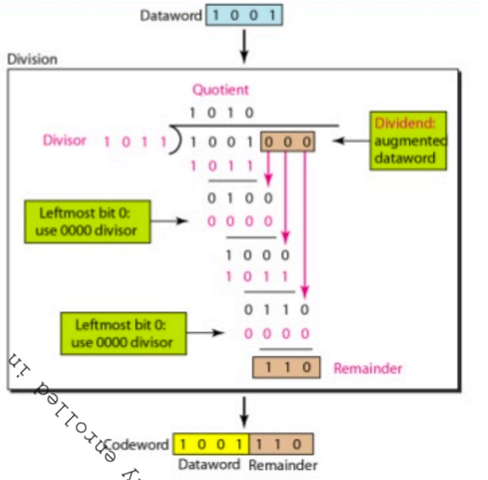
What do you think are the limitations of checksum?

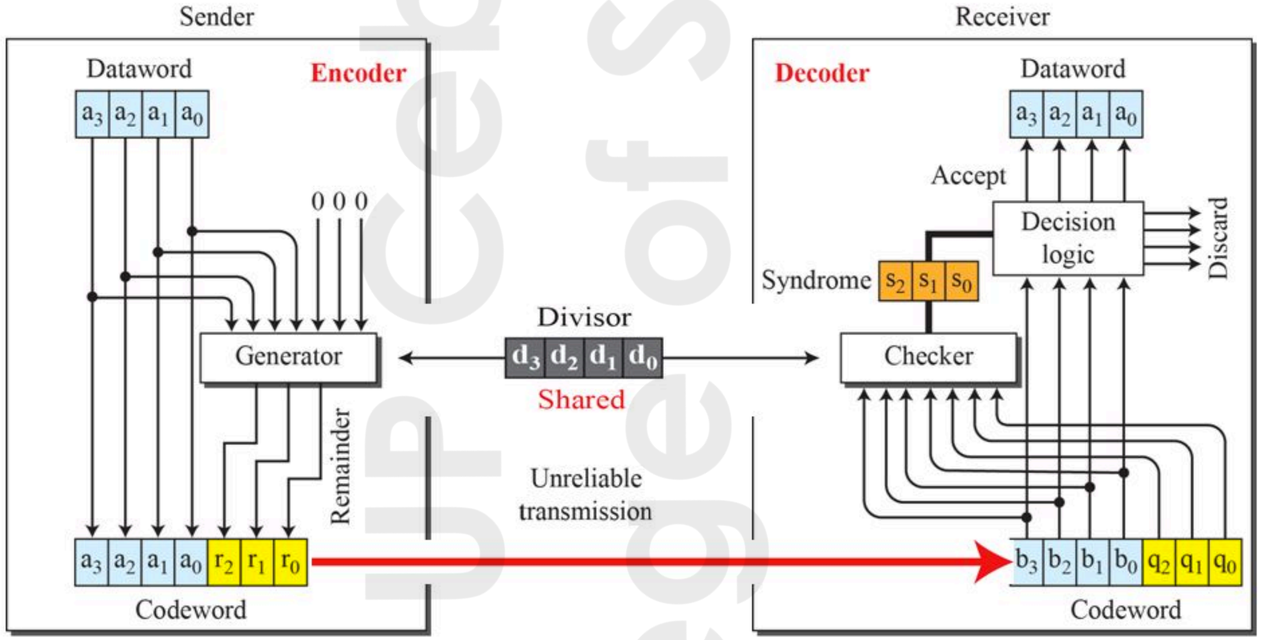
- Among the types of errors that cannot be detected by simple checksum algorithms are reordering of the bytes and multiple errors that cancel each other out.

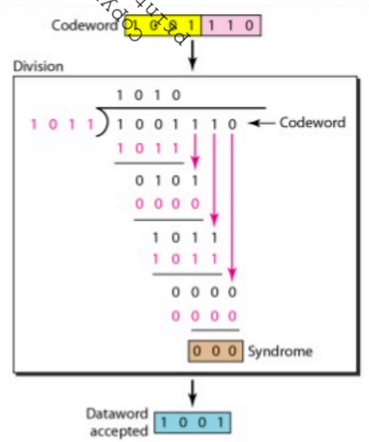
Cyclic Redundancy Check

- A technique based on division.
- The basic idea is to treat the message string as a single binary word *M*, and divide it by a key word *k* that is known to both the transmitter and the receiver.
- The remainder *r* left after dividing *M* by *k* constitutes the "check word" for the given message.
- The transmitter sends both the message string *M* and the check word *r*, and the receiver can then check the data by repeating the calculation, dividing *M* by the key word *k*, and verifying that the remainder is *r*.
- Other algorithms divide *M* and *r* with *k* and check if remainder is 0.









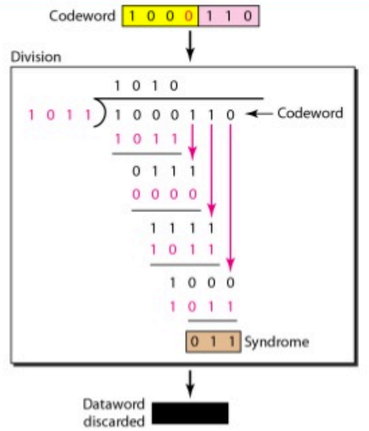


Figure 10.15 ***Division in CRC encoder***

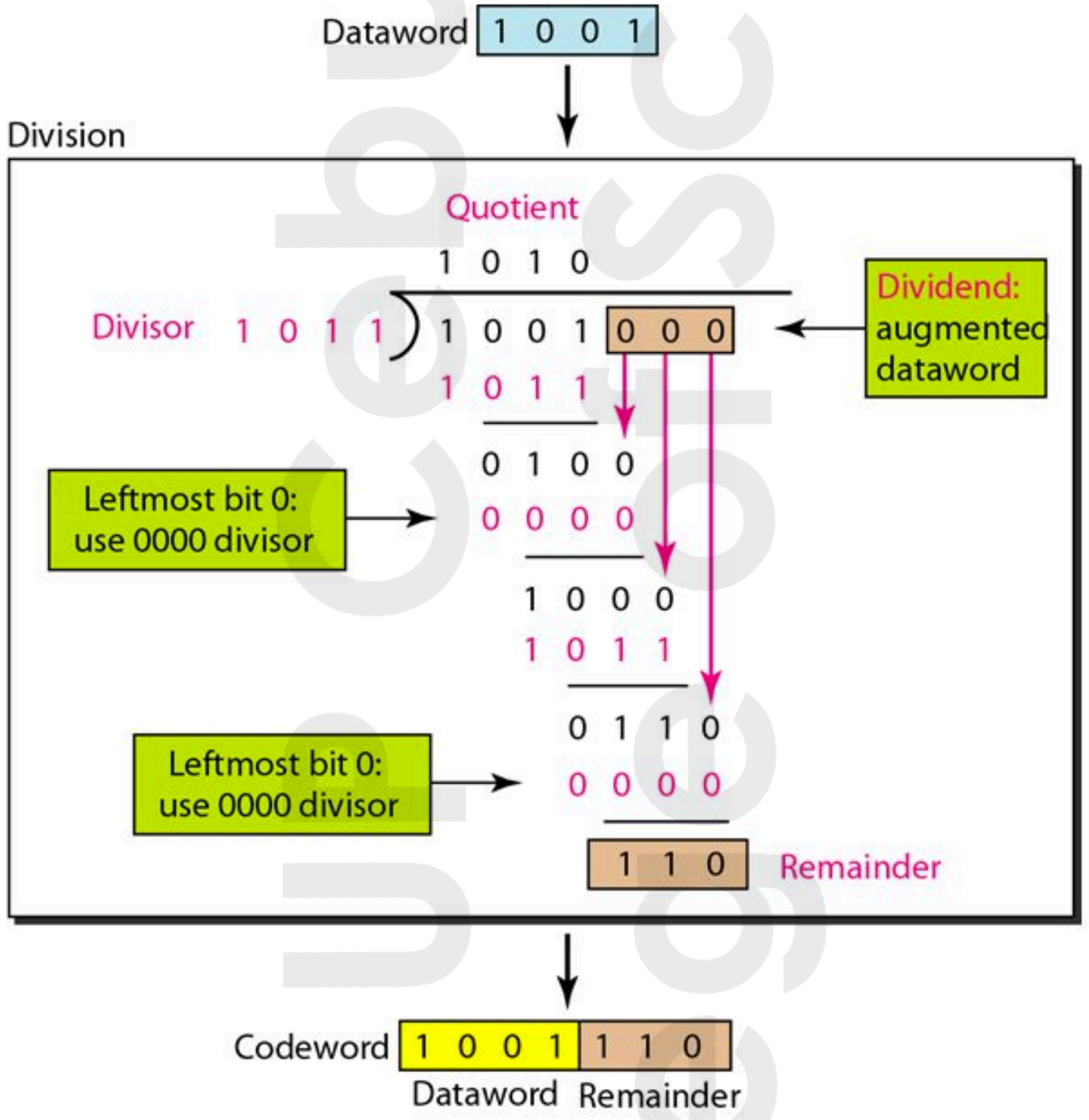
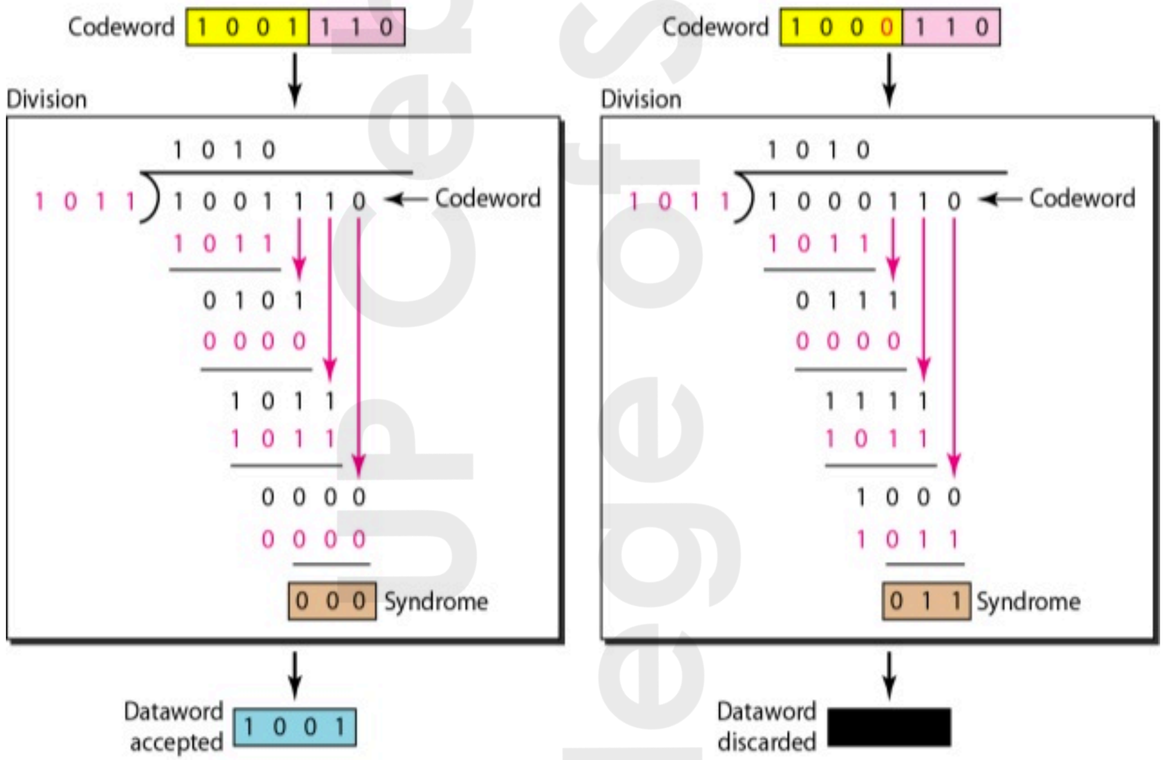


Figure 10.16 *Division in the CRC decoder for two cases*

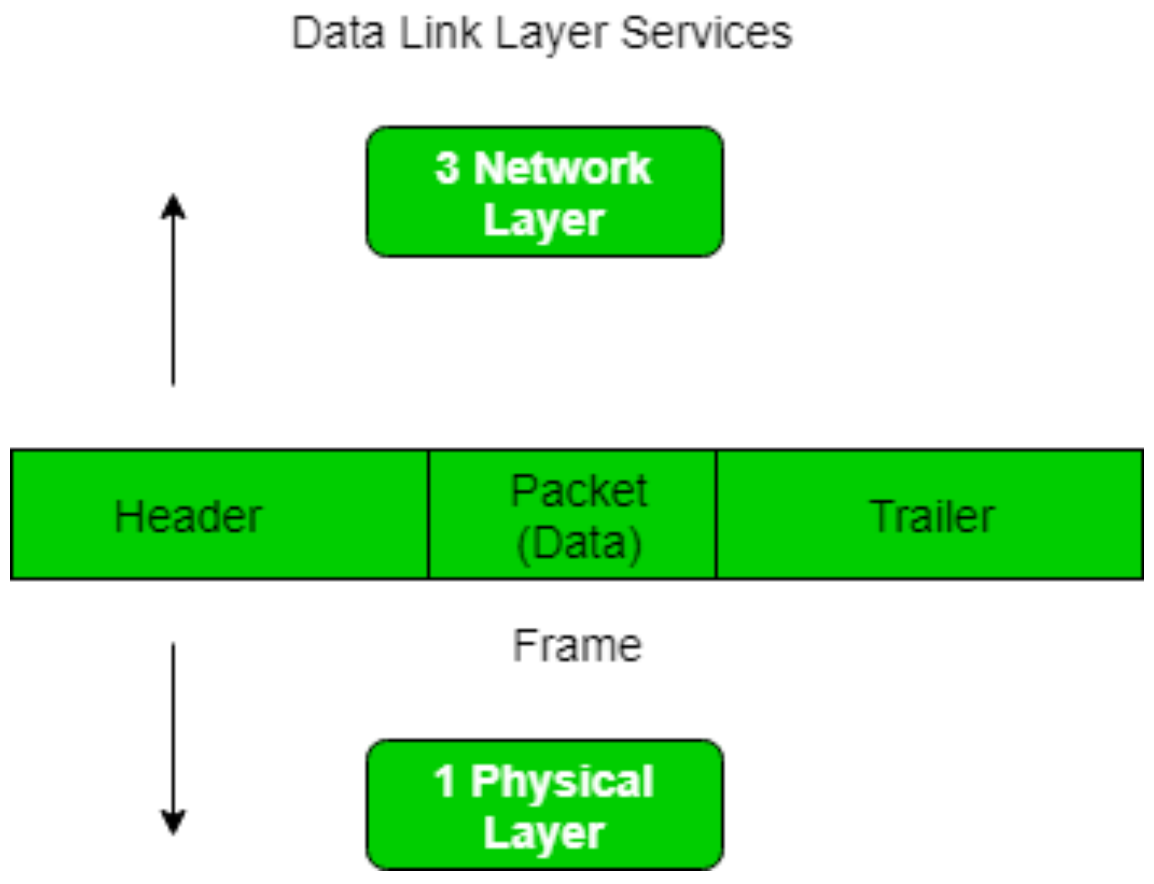


Popular Techniques for Error Detection

- Simple Parity Check
- Two-dimensional Parity Check
- Checksum
- CRC

Framing

- A process of separating a message from one source to a destination, or from other messages to other destinations, by adding a sender address and a destination address.
- Goal: Determine the start and end of packets, and make each frame distinguishable from another.
- The data link layer frame includes:
  1. Data - the packet from the network layer
  2. Header - contains information such as addressing (source and destination)
  3. Trailer - contains information such as error detection or error correction redundant bits



Ethernet Protocol

A Common Data Link Layer Protocol for LANs

Frame

Field name	Preamble	Destination	Source	Type	Data	Frame Check Sequence
Size	8 bytes	6 bytes	6 bytes	2 bytes	46 - 1500 bytes	4 bytes

Preamble - used for synchronization; also contains a delimiter to mark the end of the timing information.

Destination Address - 48 bit MAC address for the destination node.

Source Address - 48 bit MAC address for the source node.

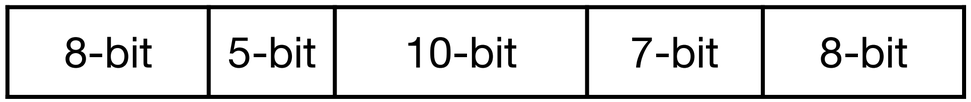
Type - value to indicate which upper layer protocol will receive the data after the Ethernet process is complete.

Data or payload - this is the PDU, typically an IPv4 packet, that is to be transported over the media.

Frame Check Sequence (FCS) - A value used to check for damaged frames.

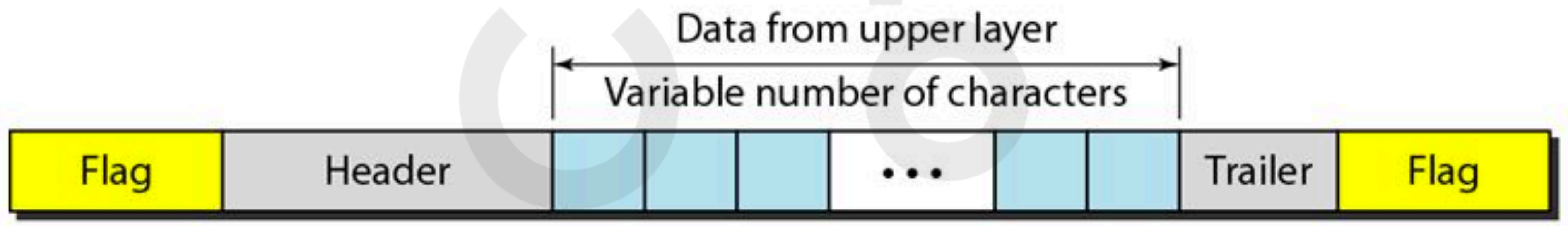
Types of Framing

1. Fixed-Size Framing
   - Frame size is fixed.
   - No need to provide boundaries to the frame.
   - Length of the frame itself acts as delimiter.
2. Variable-Size Framing
   - Prevalent in LANS.
   - End of frame and beginning of next need to be defined.
   - Two approaches:
     1. Character-Oriented
     2. Bit-Oriented

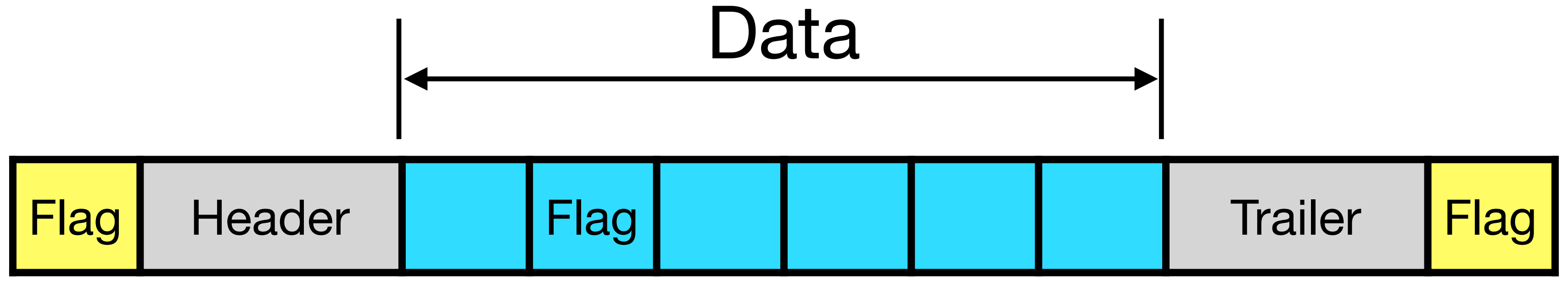


1. Character-Oriented

- Byte-oriented.
- Data to be carried are 8-bit characters from a coding such as ASCII.
- Header and trailer are also multiple of 8 bits.
- An 8-bit flag is added at the beginning and end of frame to separate a frame from the next.

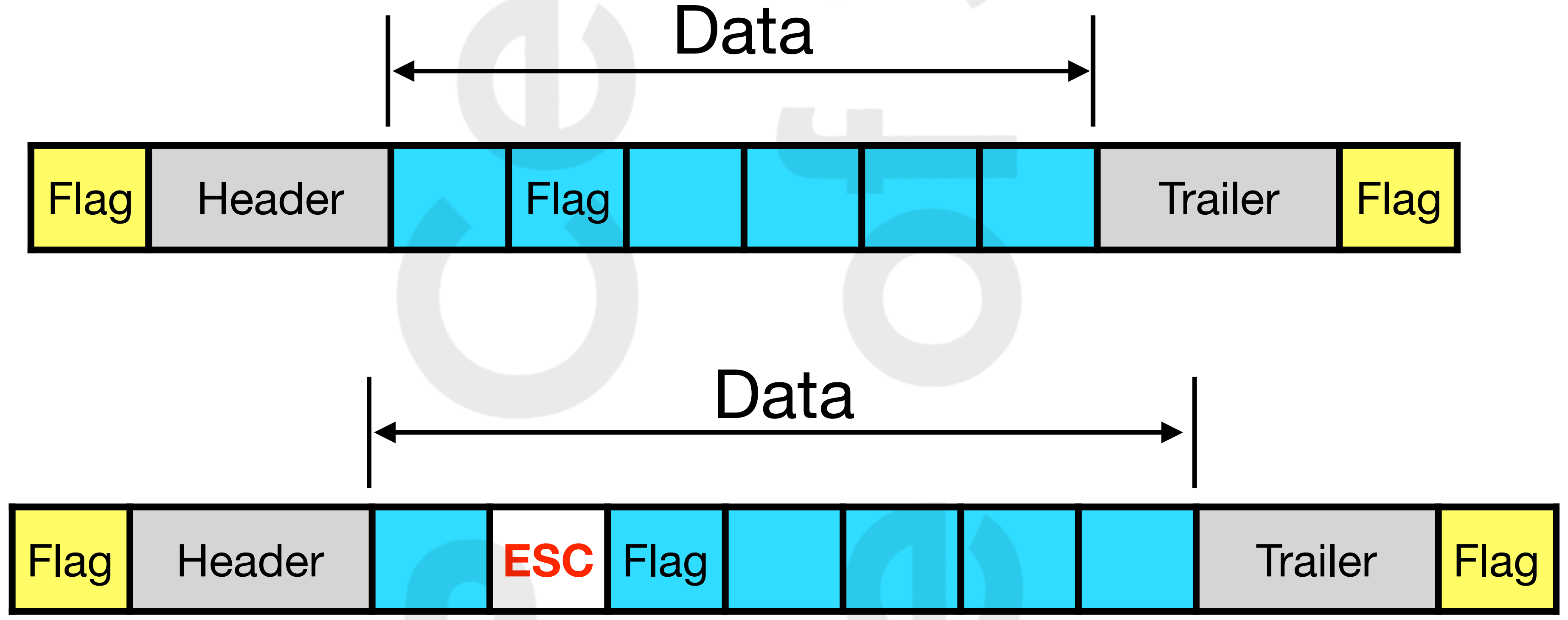


- Popular when only text was exchanged.
- Flag must be any character not used for text communication.
- Problem: Sending other types of information: graphs, audio and video. *Pattern used for flag could also be part of the information.*
- Solution: Byte stuffing or character stuffing.

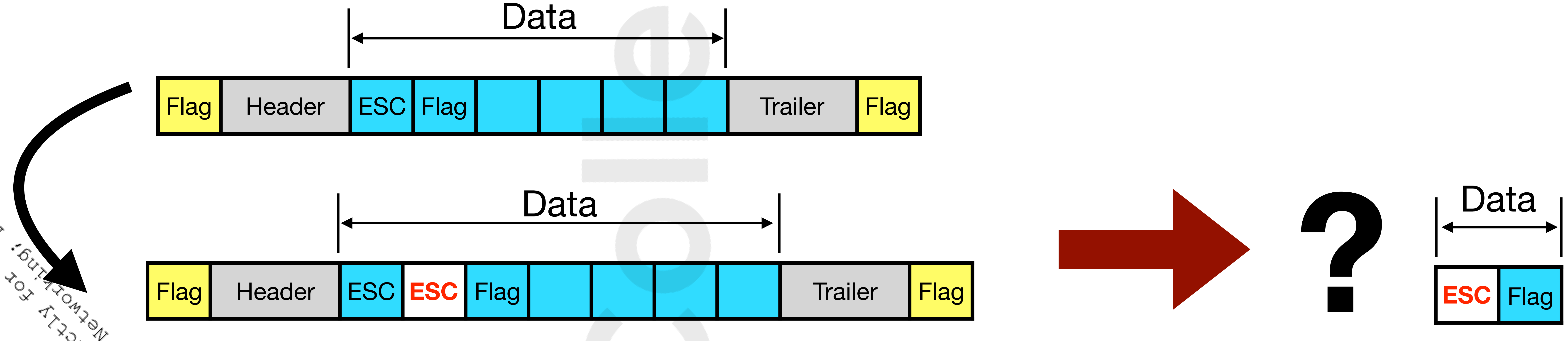


Byte Stuffing

- Special byte is added (stuffed) to the data section of the frame when there is a character with same pattern as the flag.
- The extra byte is usually called the escape character (ESC) with a predefined bit pattern.

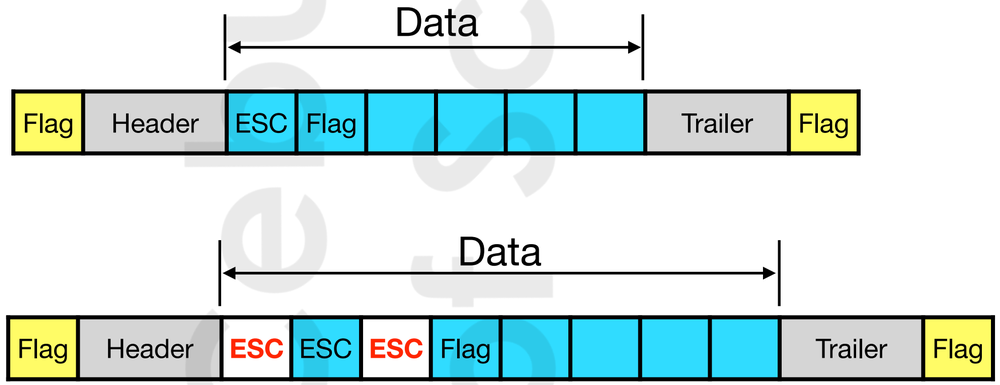


- When receiver encounters the ESC character, it removes it from the data section and treats the next character as data, not a delimiting flag.
- Problem???



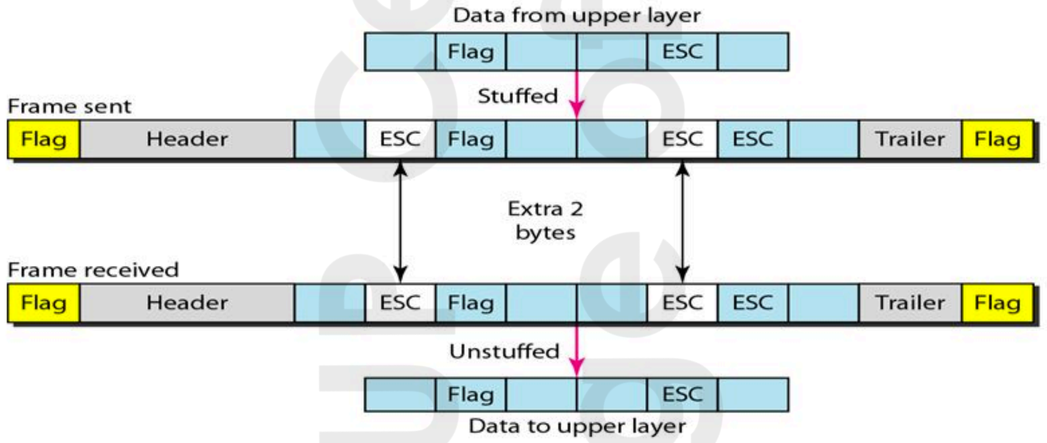
Solution…

- Escape characters (ESC) must also be marked by another escape character.



Another example…

Byte stuffing and unstuffing

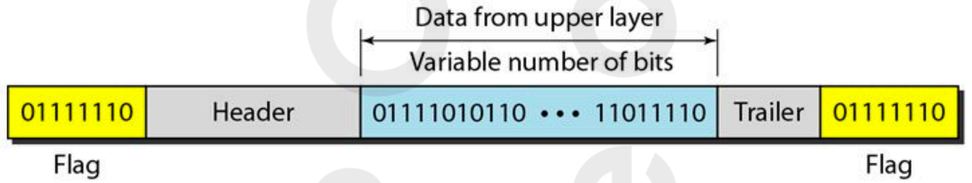


Yet another problem…

- The universal coding systems in use today, such as Unicode, have 16-bit and 32-bit characters that conflict with 8-bit characters.

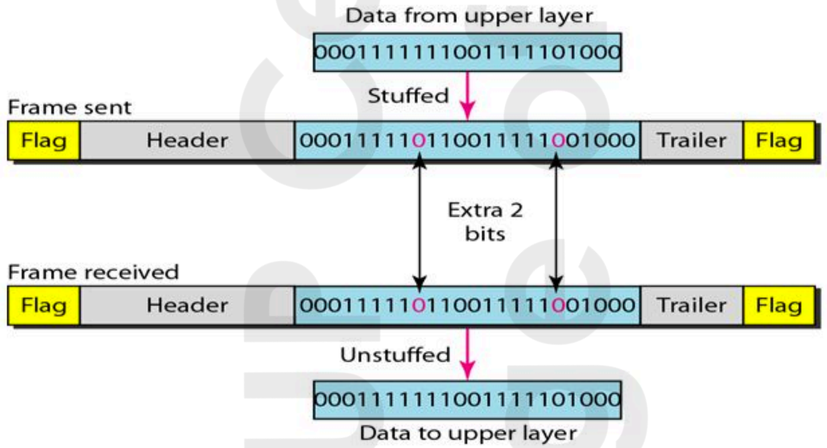
2. Bit-Oriented

- Data section is sequence of bits and not grouped according to bytes.
- In addition to headers (and possible trailers), still needs a delimiter to separate one frame from the other.
- Most protocols use a special 8-bit pattern flag "*0111 1110*" as delimiter.



- Problem???

- Solution: Adding one extra *0* whenever five consecutive *1s* follow a *0* in the data regardless of the value of next bit.
- Extra stuffed bit is eventually removed from the data by the receiver.

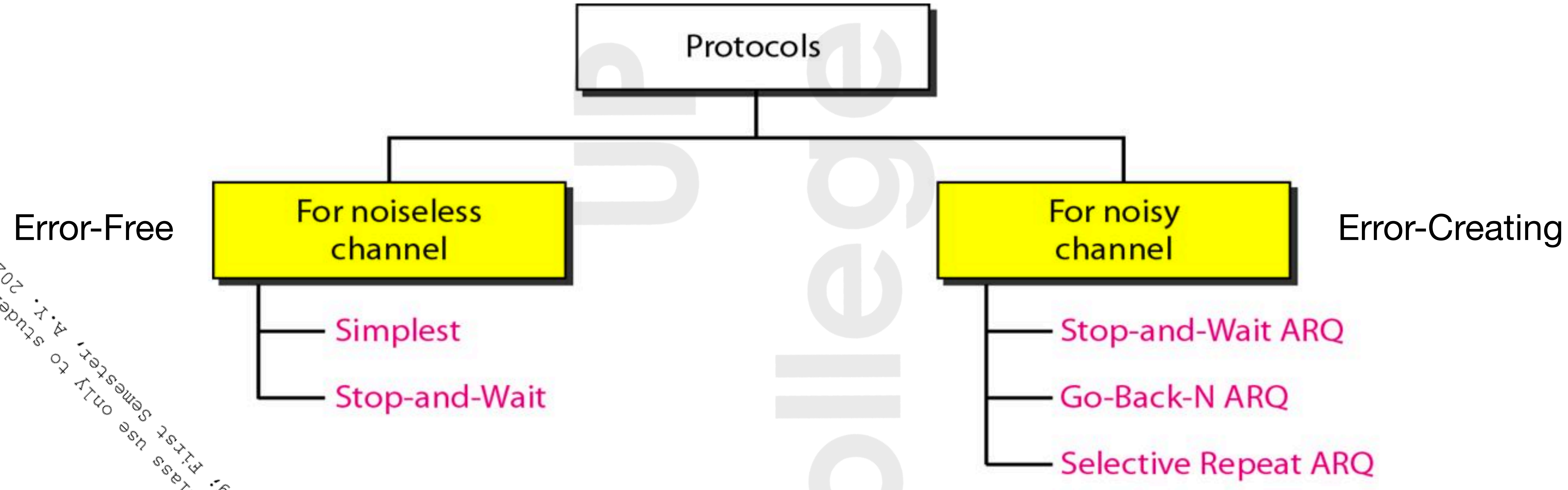


Flow and Error Control

- Collectively known as data link control.
- *Flow Control*
  - Refers to a set of procedures used to restrict the amount of data that the sender can send before waiting for acknowledgment.
  - Each receiving device has a block memory, called a *buffer*, reserved for storing incoming data until they are processed. Notify if full.
- Error Control
  - Refers primarily to methods of error detection and retransmission.
  - Allows receiver to inform sender of any frames lost or damaged in transmission and coordinates retransmission of those frames by the sender (*automatic repeat request* or *ARQ*).

Protocols

- A set of rules in the data link layer in implementing and governing an orderly exchange of data between devices.
- Normally implemented in software by using one of the common programming languages.
- Unidirectional versus bidirectional protocols.
- Taxonomy of protocols:

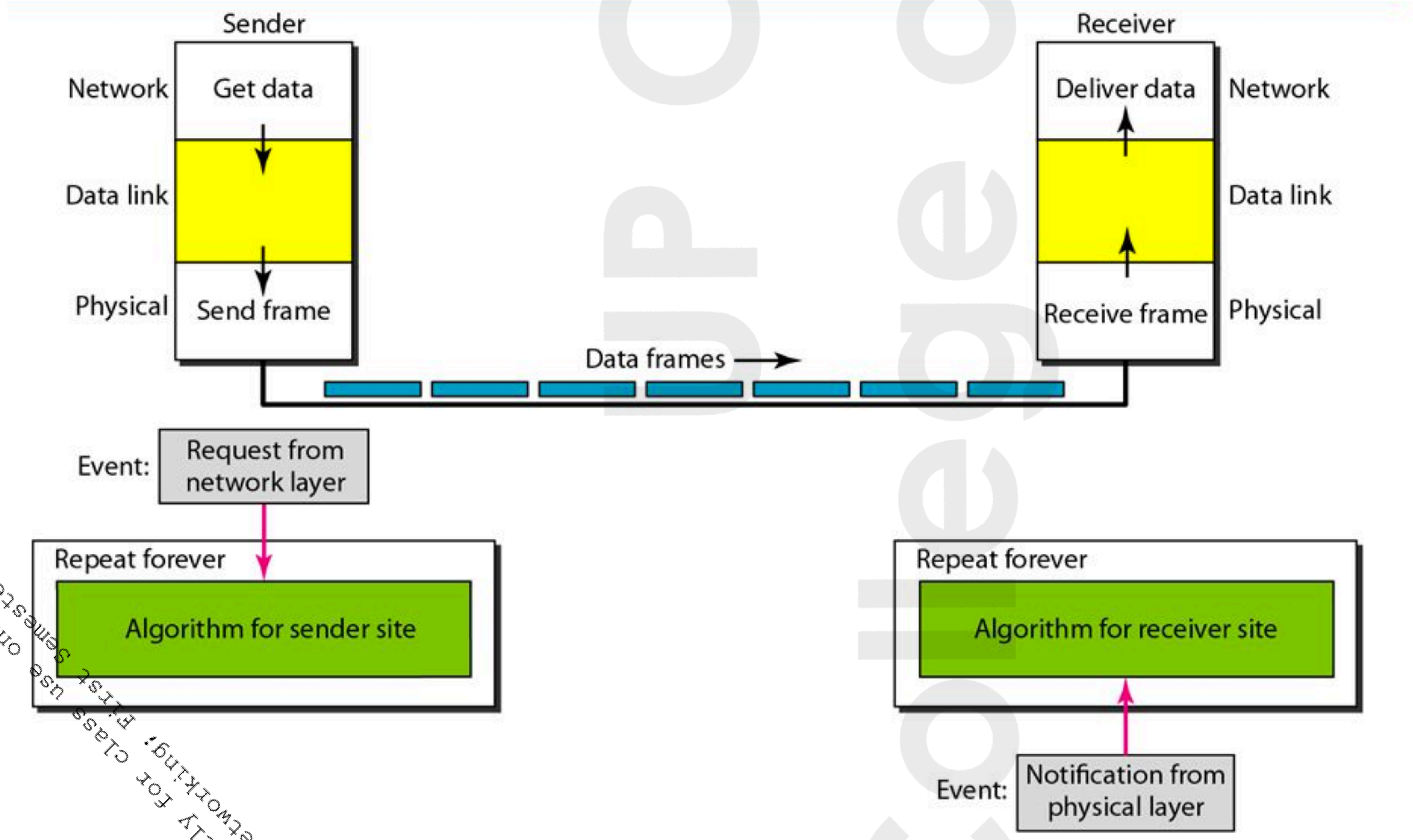


Noiseless Channels

- Assumption: Ideal channel in which no frames are lost, duplicated, or corrupted.
- Two protocols:
  1. Simplest Protocol
  2. Stop-and-Wait Protocol

1. Simplest Protocol

- No flow or error control.
- Assumption: Receiver can immediately handle any frame it receives. Layer 2 of receiver immediately removes header from frame and hands data packet to Layer 3. In short, receiver can never be overwhelmed with incoming frames.
- Design:



Algorithms

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

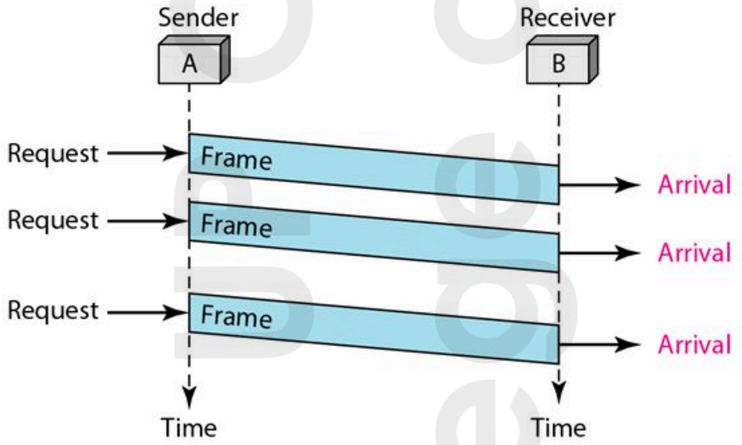
```
while(true)                          // Repeat forever
{
  WaitForEvent();                    // Sleep until an event occurs
  if(Event(RequestToSend))           //There is a packet to send
  {
     GetData();
     MakeFrame();
     SendFrame();                    //Send the frame
  }
}
```

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

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

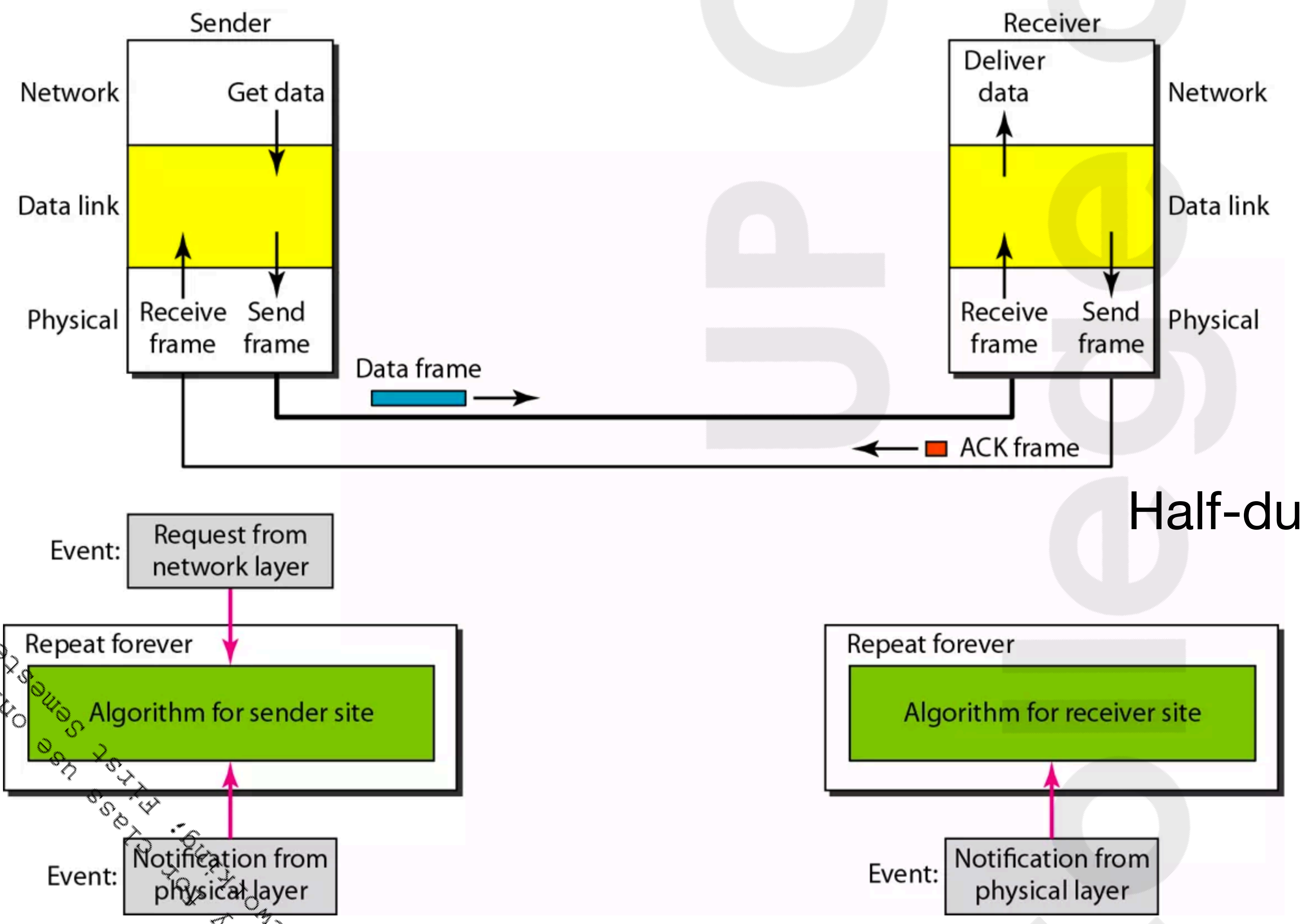
Example 11.1

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



2. Stop-and-Wait Protocol

- The sender sends one frame, stops until it receives confirmation from the receiver (okay to go ahead), and then sends the next frame.
- Although unidirectional communication for data frames, auxiliary ACK frames (simple tokens of acknowledgment) travel from the other direction.
- Feature: Added flow control from previous protocol to prevent overwhelming.
- Design:



Half-duplex Link

Algorithms

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

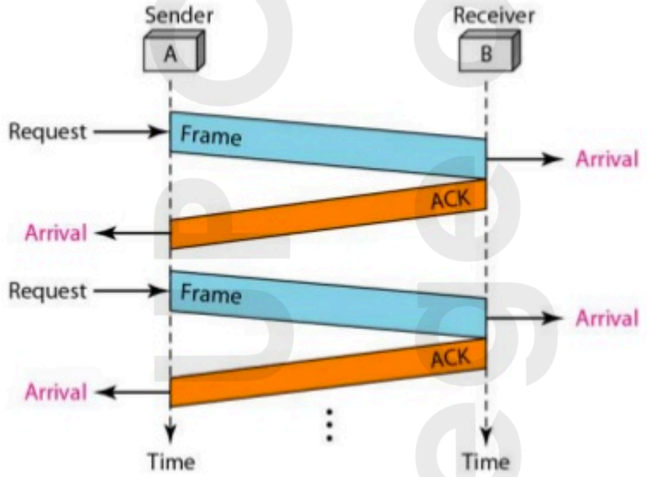
Algorithm 3 *Sender-site algorithm for Stop-and-Wait Protocol*

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

Algorithm 4 *Receiver-site algorithm for Stop-and-Wait Protocol*

Example 11.2

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



Noisy Channels

- Realistic as opposed to non-existent noiseless.
- Addition of error control.
- Three protocols:
  1. Stop-and-Wait Automatic Repeat Request
  2. Go-Back-N Automatic Repeat Request
  3. Selective Repeat Automatic Repeat Request

1. Stop-and-Wait Automatic Repeat Request

- Short: Stop-and-Wait ARQ.
- **Feature 1:** Addition of a simple error control mechanism.
  - When frame arrives at receiver, it is checked and if corrupted, it is silently discarded.
  - Detection of errors is manifested by the silence of the receiver.
  - During sending, sender keeps copy of sent frame and starts a timer.
  - If timer expires and there is no ACK for the sent frame, the frame is resent and timer is restarted.

- Problem with previous protocols: Incorrect frame, duplicate frame or frame out of order.
- **Feature 2:** Frames are numbered.
  - When receiver receives frame that is out of order, this means that frames were either lost or duplicated.
- Problem: ACK frame can also be corrupted and lost.
- **Feature 3:** Addition of redundancy bits and sequence number to ACK.
  - The sender simply discards a corrupted ACK frame or ignores an out-of-order one.

Sequence Numbers

- Range: *0* to *2^m-1*, *m* is number of bits
- Example scenarios:

Assume we have used *x* as a sequence number; we need to use *x + 1* after that. Assume that sender has sent the frame numbered *x*. Three things can happen.

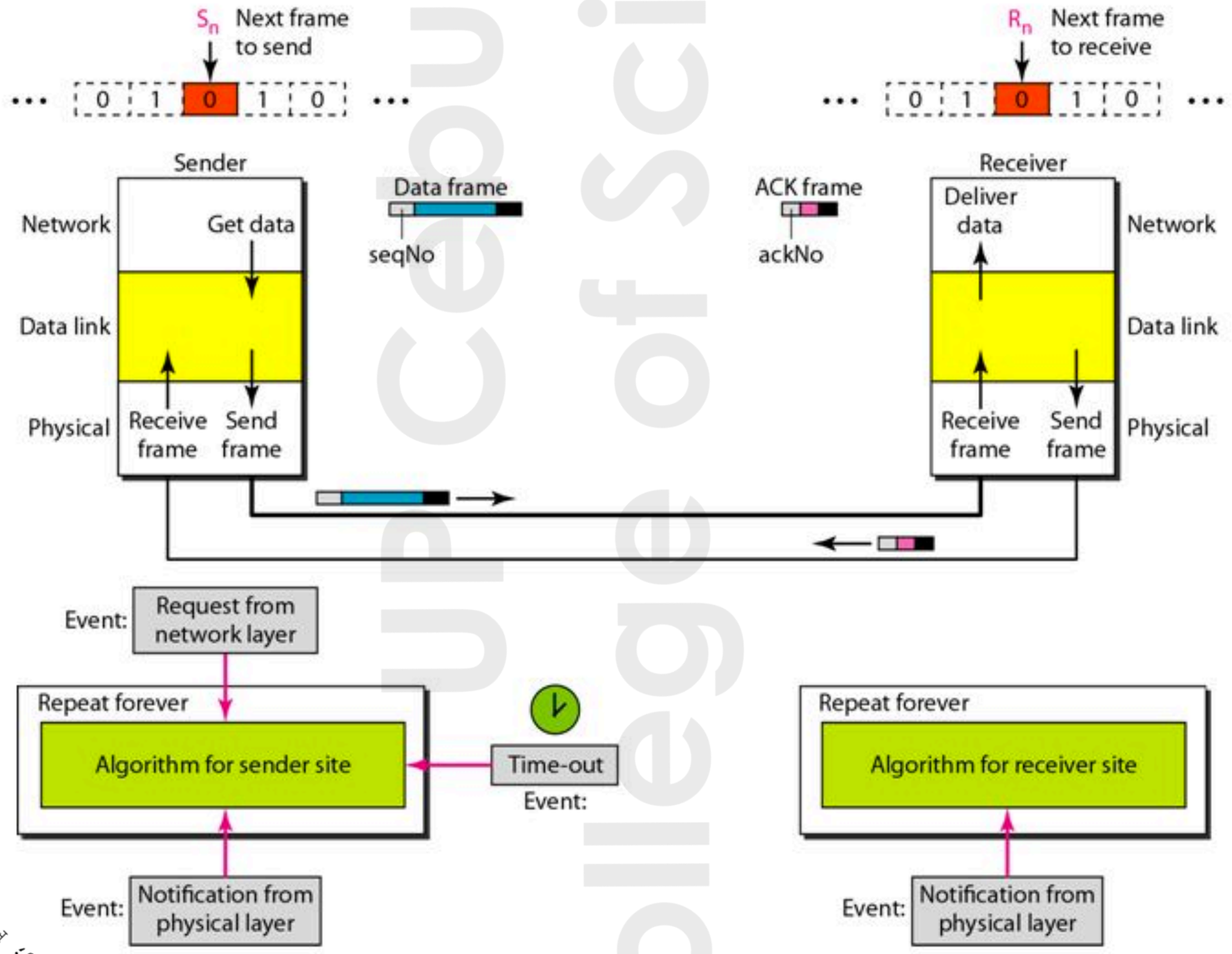
1. The frame arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next frame numbered $x + 1$.
2. The frame arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost. The sender resends the frame (numbered *x*) after the time-out. Note that the frame here is a duplicate. The receiver can recognize this fact because it expects frame $x + 1$ but frame x was received.
3. The frame is corrupted or never arrives at the receiver site; the sender resends the frame (numbered *x*) after the time-out.

Sequence number is: 0, 1, 0, 1, 0, and so on… Only x and x + 1.

Acknowledgment Numbers

- Convention: Acknowledgment Numbers always announce the sequence number of the next frame expected by the receiver.
- For example, if frame 0 has arrived safe and sound, the receiver sends an ACK frame with acknowledgment 1 and vice versa.

Design



Algorithm (Sender)

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

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

(continued)

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

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

Algorithm (Receiver)

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

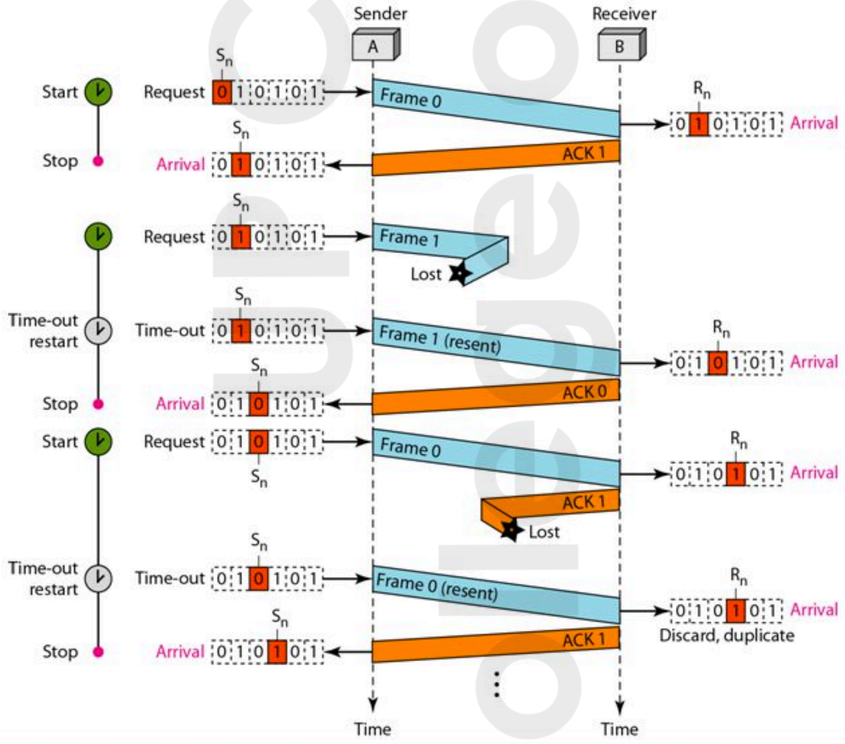
```
Rn = 0;                          // Frame 0 expected to arrive first
while(true)
{
  WaitForEvent();                // Sleep until an event occurs
  if(Event(ArrivalNotification)) //Data frame arrives
  {
     ReceiveFrame();
     if(corrupted(frame));
        sleep();
     if(seqNo == Rn)      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

Http://netwk.hannam.ac.kr HANNAM UNIVERSITY

Example 11.3

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

Figure 11.11 *Flow diagram for Example 11.3*



Efficiency

- Stop-and-Wait ARQ is very inefficient if channel is thick and long.
  - Thick - Channel has large bandwidth.
  - Long - Round-trip delay is long.
- Product of these two is called the *bandwidth delay product*.
  - If channel is considered as a pipe, then bandwidth delay product is volume of the pipe in bits.
  - Pipe is always there; if not used, then not efficient.
  - Bandwidth delay product is a measure of the number of bits we can send out of the system while waiting for news from the receiver.

Example 11.4

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

Solution

The bandwidth-delay product is $(1 \times 10^6) \times (20 \times 10^{-3}) = 20{,}000$ bits

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

Example 11.5

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

Solution

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

Pipelining

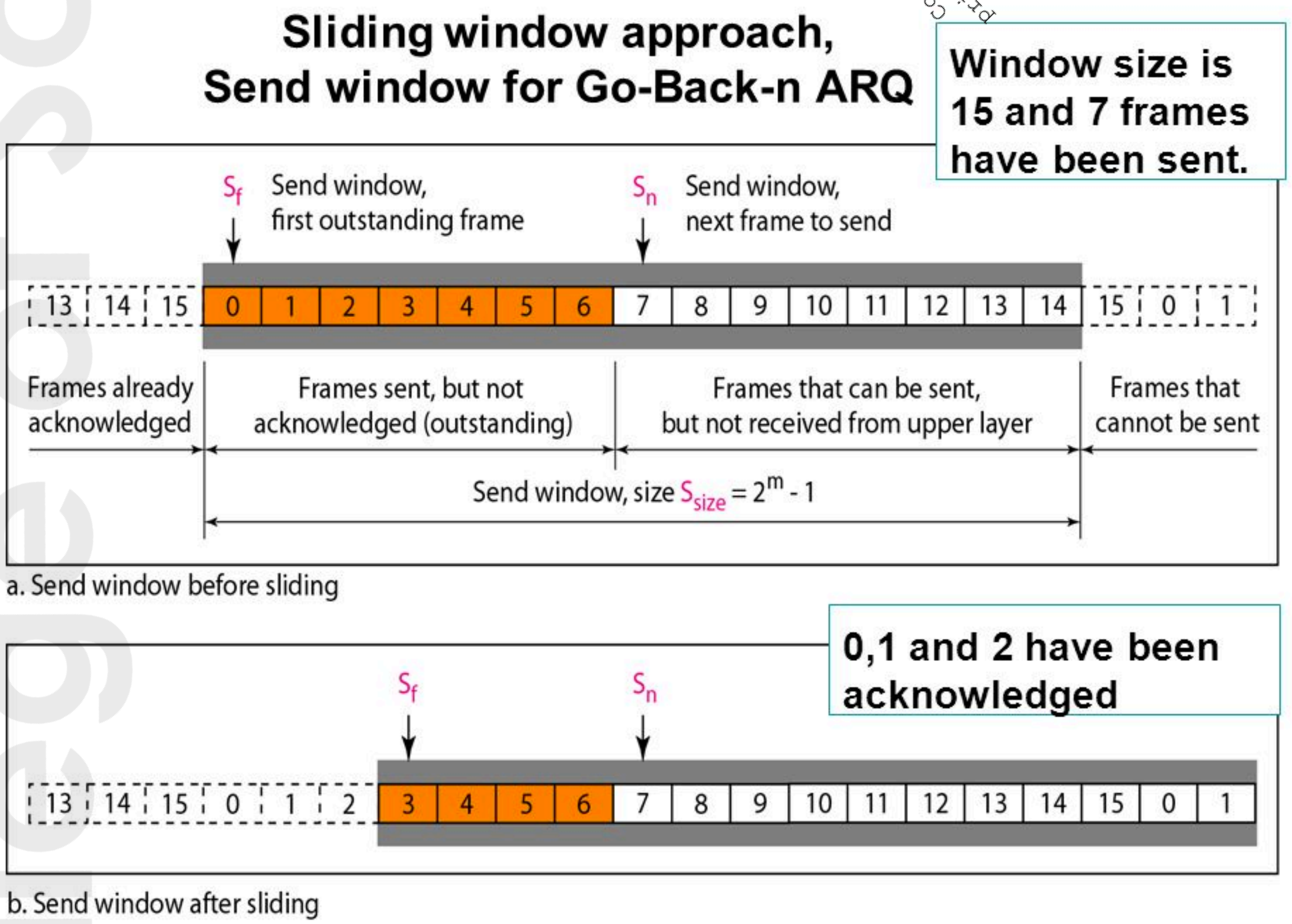
- In networking and in other areas, a task is often begun before the previous task has ended (pipelining).
- If this is the case, efficiency of the transmission improves if number of bits in transition is large with respect to the bandwidth-delay product.
- Stop-and-Wait ARQ has no pipelining because there is a need to wait for a frame to reach destination and be acknowledged before next frame is sent.
- However, next two protocols can apply pipelining.

2. Go-Back-N Automatic Repeat Request

- Short: Go-Back-N ARQ.
- Several frames can be sent before receiving acknowledgments; it's important to keep a copy of frames until the acknowledgments arrive.
- Sequence Numbers.
    - Frames from a sending station are numbered sequentially.
    - Range: 0 to $2^m - 1$.
    - Can repeat if exhausted.
- Sliding Window.
    - An abstract concept that defines the range of sequence numbers that is the concern of the sender and receiver per transaction.
    - Sender and receiver need to deal with only part of the possible sequence numbers (send sliding window and receive sliding window).

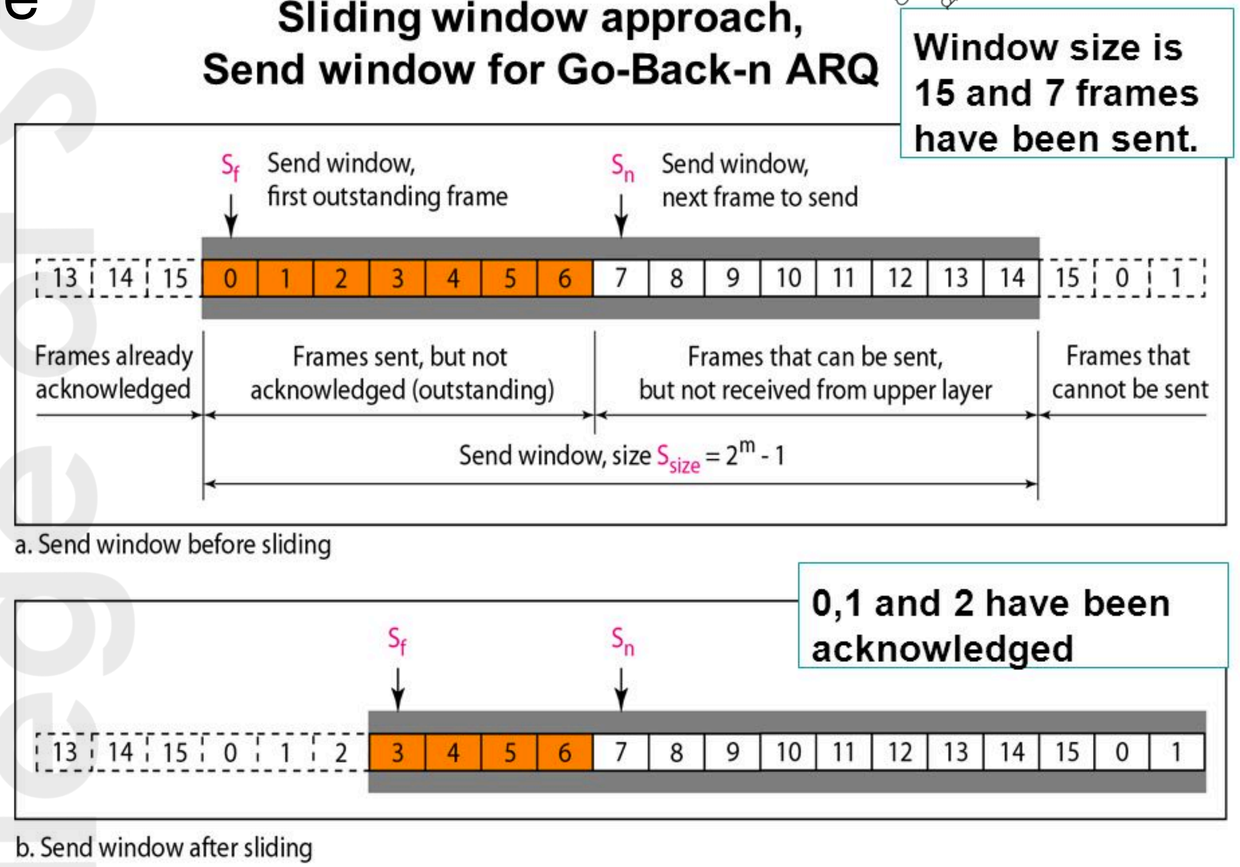
Send Sliding Window

- The send window is an imaginary box covering the sequence numbers of the data frames which can be transmitted (both sent and can be sent).
- Max size is $2^m - 1$; *m* being the number of bits. *Reason to be discussed later.* Fixed and set to max but advanced protocols may have variable window size.
- Entire sequence divided into 4 regions: (1) sequence already acknowledged, (2) sent and unknown status (outstanding frames), (3) can be sent, and (4) cannot be used.



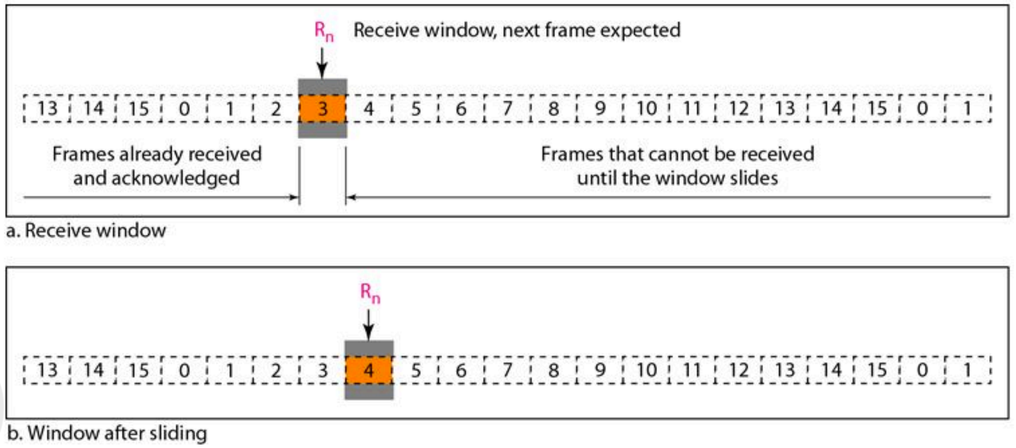
More on Send Sliding Window…

- Three relevant variables:
  1. S_f - first (oldest) outstanding frame
  2. S_n - next frame to be sent
  3. S_{size} - size of send window
- The acknowledgments in this protocol are cumulative: more than one frame can be acknowledged by an ACK frame.
- Send window can slide on or more slots when a valid acknowledgment arrives.



Receive Sliding Window

- Makes sure that the correct data frames are received and correct acknowledgments are sent.
- Size of window is always 1.
- Frame arriving out of order is discarded and needs retransmission.
- The window slides when the correct frame has arrived; slide one slot at a time.
- R_n - next frame expected.



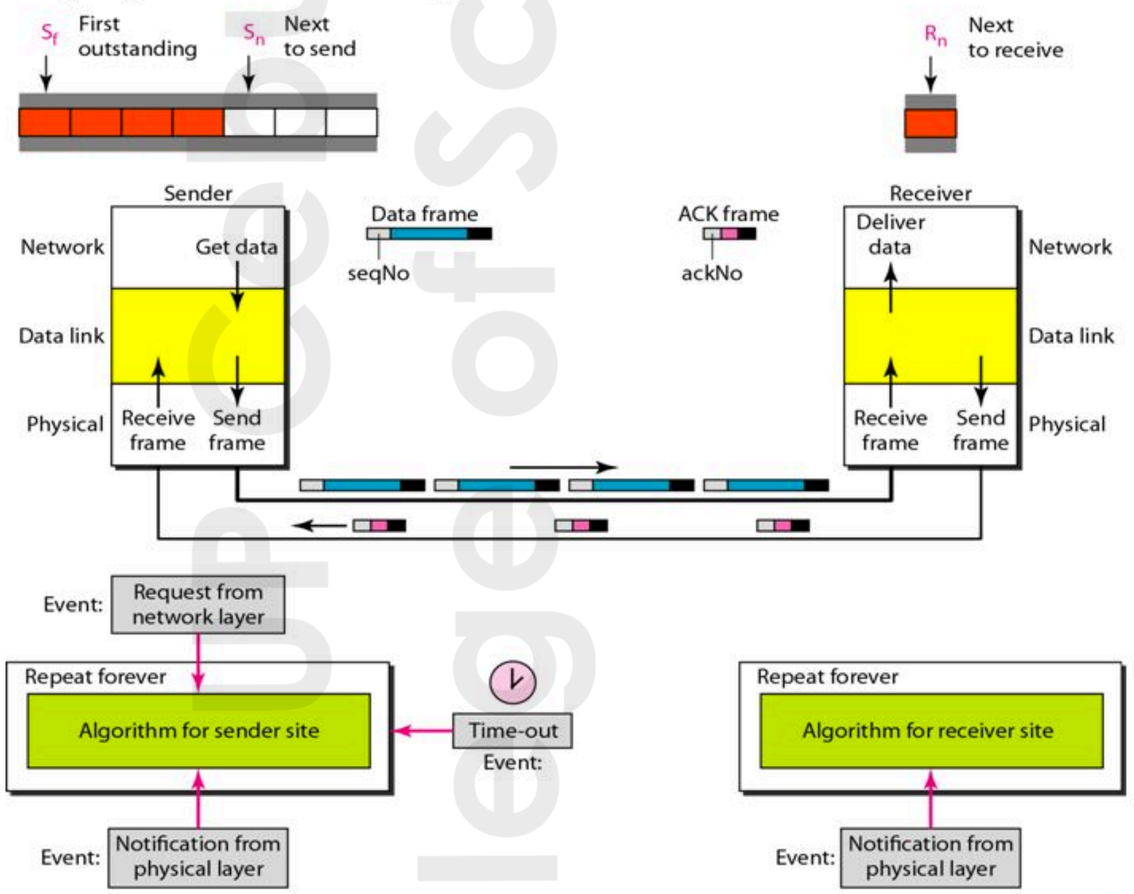
Timers and Acknowledgment

- Timers
  - Although can have timer for each frame sent, this protocol uses only one.
  - Reason: Timer for the first outstanding frame always expires first; if so, resend all outstanding frames.
- Acknowledgment
  - Receiver sends positive acknowledgment if frame has arrived safe and sound and in order.
  - If frame is damaged or received out of order, receiver is silent and discards subsequent frames until correct one is received.
  - No need to acknowledge each frame received; can send one cumulative acknowledgment for several frames.

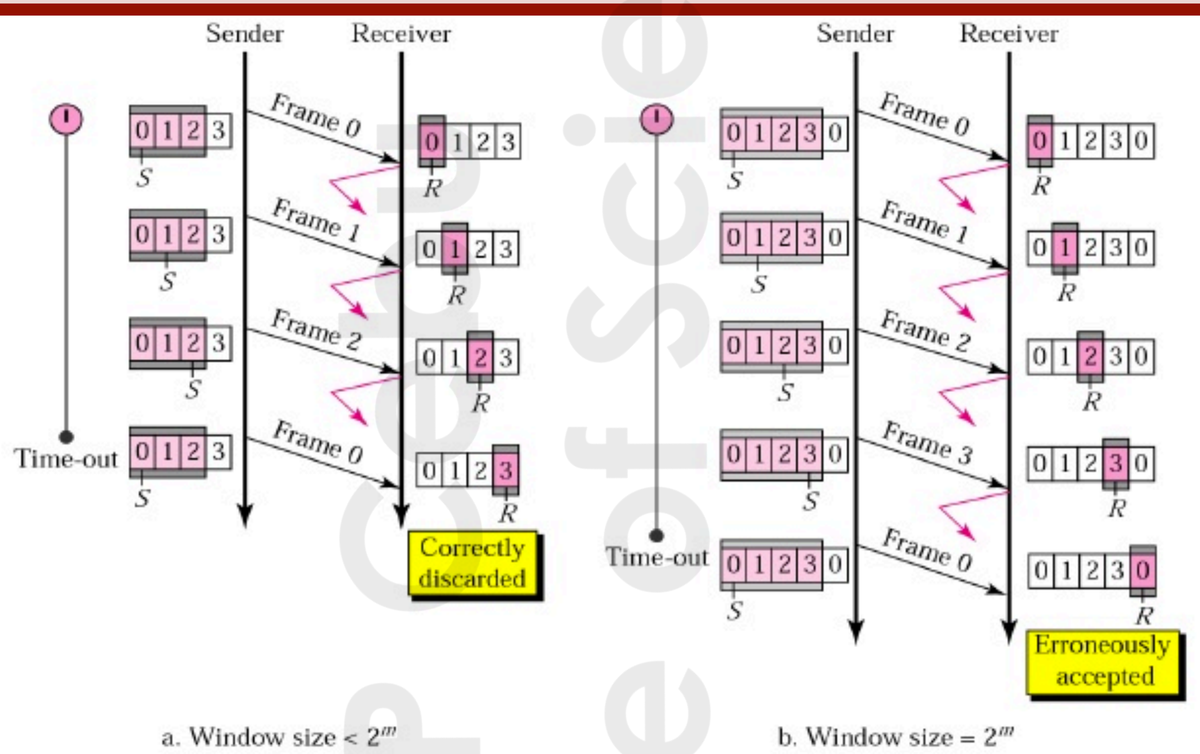
Resending a Frame

- When timer expires, sender resends all outstanding frames.
- Example: Suppose sender already sent frame 6, but timer for frame 3 expires. This means that frame 3 was not acknowledged; the sender goes back and resends frames 3, 4, 5 and 6 again; hence, the name Go-Back-N ARQ.

Design of Go-Back-N ARQ



Reason why send window size must be 2^m-1…



a. Window size < 2^m b. Window size = 2^m

We can now show why the size of the send window must be less than 2^m. As an example, we choose $m = 2$, which means the size of the window can be 2^m - 1, or 3. Figure 11.15 compares a window size of 3 against a window size of 4. If the size of the window is 3 (less than 2^2) and all three acknowledgments are lost, the frame timer expires and all three frames are resent. The receiver is now expecting frame 3, not frame 0, so the duplicate frame is correctly discarded. On the other hand, if the size of the window is 4 (equal to 2^2) and all acknowledgments are lost, the sender will send a duplicate of frame 0. However, this time the window of the receiver expects to receive frame 0, so it accepts frame 0, not as a duplicate, but as the first frame in the next cycle. This is an error.

Algorithm (Sender)

Algorithm 11.7 *Go-Back-N sender algorithm*

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

(continued)

11.47

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

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

Notice any mistake?

11.48

Algorithm (Receiver)

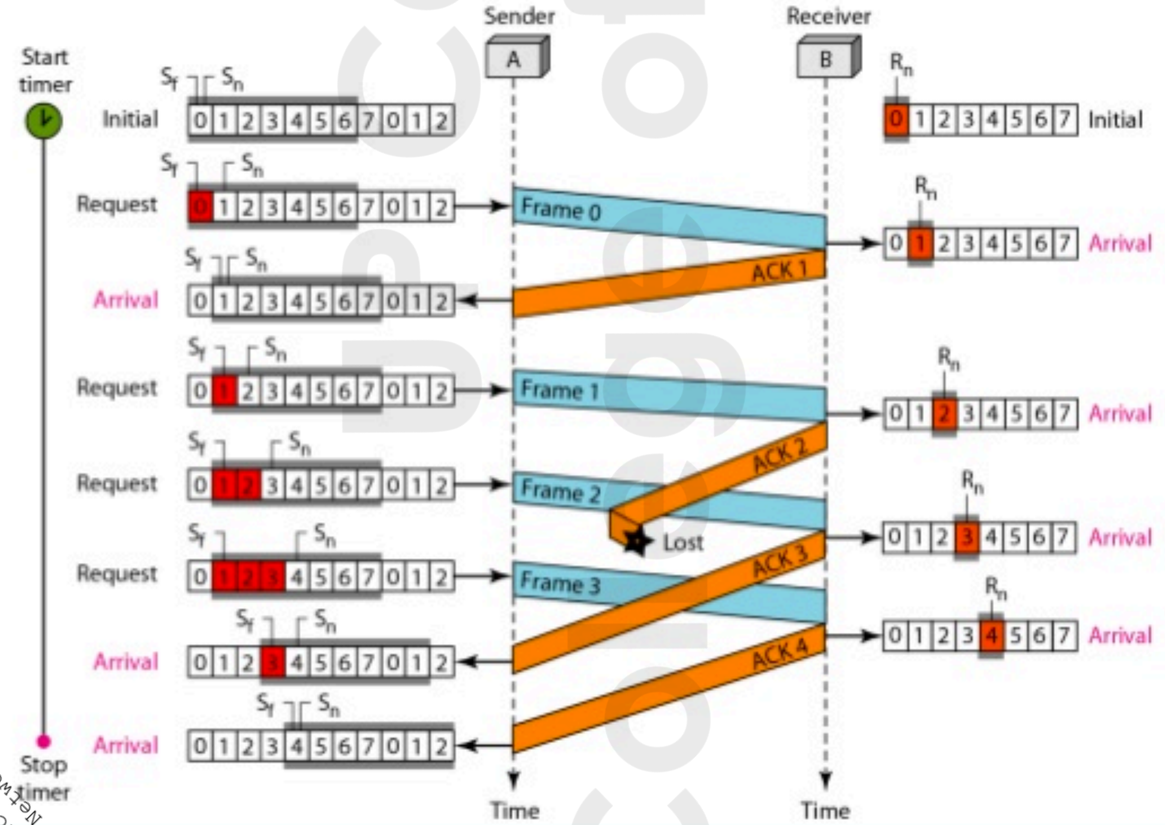

```
1  Rn = 0;
2
3  while (true)                          //Repeat forever
4  {
5    WaitForEvent();
6
7    if(Event(ArrivalNotification)) /Data frame arrives
8    {
9       Receive(Frame);
10      if(corrupted(Frame))
11           Sleep();
12      if(seqNo == Rn)                  //If expected frame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

Example 11.6

Figure 11.16 shows an example of Go-Back-*N*. This is an example of a case where the forward channel is reliable, but the reverse is not. No data frames are lost, but some ACKs are delayed and one is lost. The example also shows how cumulative acknowledgments can help if acknowledgments are delayed or lost.

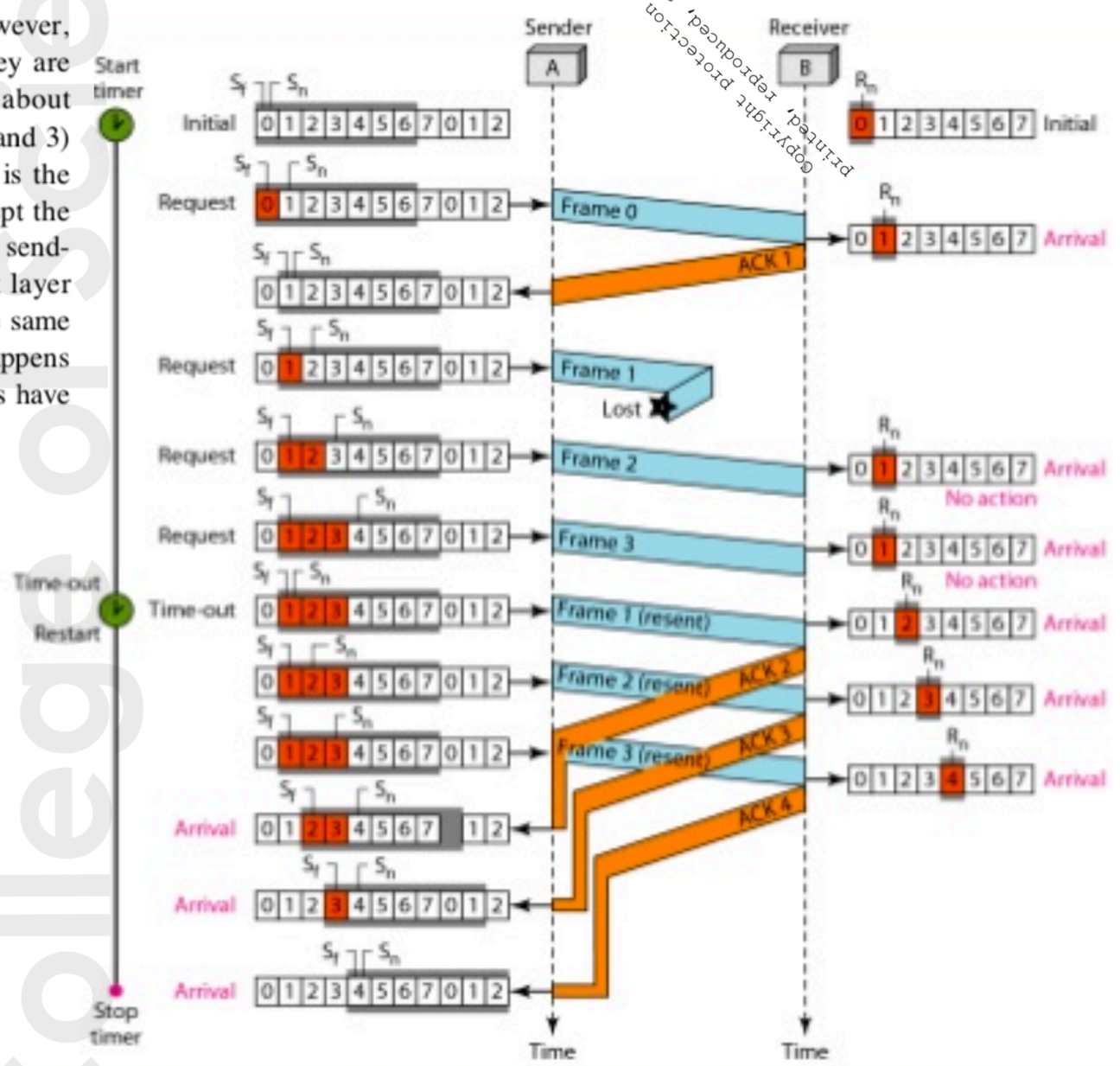
After initialization, there are seven sender events. Request events are triggered by data from the network layer; arrival events are triggered by acknowledgments from the physical layer. There is no time-out event here because all outstanding frames are acknowledged before the timer expires. Note that although ACK 2 is lost, ACK 3 serves as both ACK 2 and ACK3.

There are four receiver events, all triggered by the arrival of frames from the physical layer



Example 11.7

Figure 11.17 shows what happens when a frame is lost. Frames 0, 1, 2, and 3 are sent. However, frame 1 is lost. The receiver receives frames 2 and 3, but they are discarded because they are received out of order (frame 1 is expected). The sender receives no acknowledgment about frames 1, 2, or 3. Its timer finally expires. The sender sends all outstanding frames (1, 2, and 3) because it does not know what is wrong. Note that the resending of frames 1, 2, and 3 is the response to one single event. When the sender is responding to this event, it cannot accept the triggering of other events. This means that when ACK 2 arrives, the sender is still busy with sending frame 3. The physical layer must wait until this event is completed and the data link layer goes back to its sleeping state. We have shown a vertical line to indicate the delay. It is the same story with ACK 3; but when ACK 3 arrives, the sender is busy responding to ACK 2. It happens again when ACK 4 arrives. Note that before the second timer expires, all outstanding frames have been sent and the timer is stopped.



Go-Back-N ARQ Versus Stop-and-Wait ARQ

- Similarity can be found between Go-Back-N ARQ and Stop-and-Wait ARQ.
- We can say that Stop-and-Wait ARQ Protocol is actually a Go-Back-N ARQ in which there are only *two sequence numbers* and the *send window size is 1*.
- In other words, $m = 1$, $2^m - 1 = 1$.

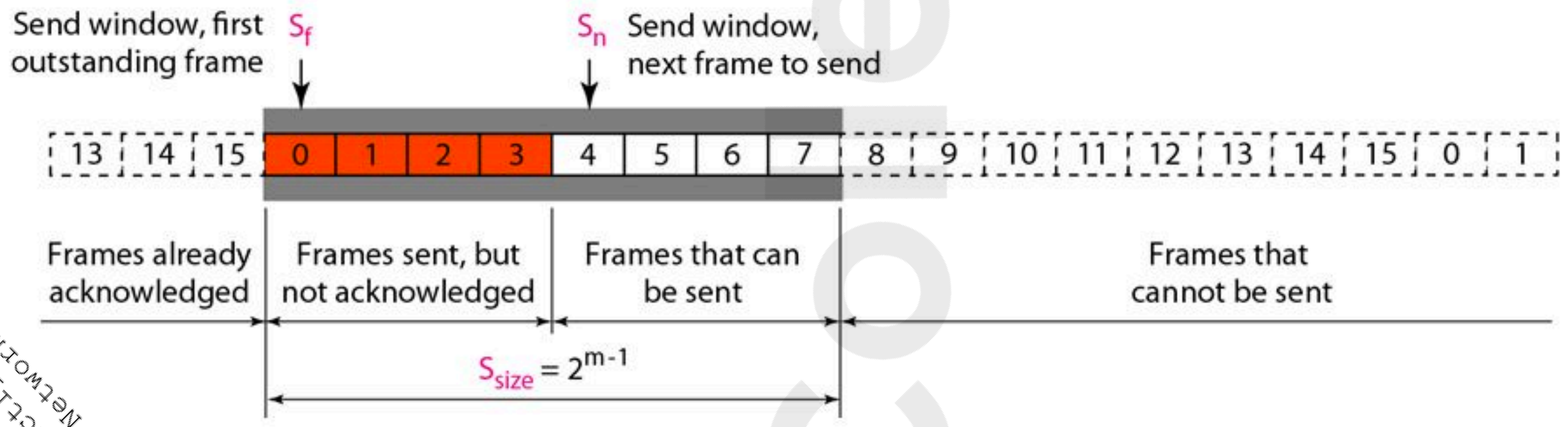
Can we improve on Go-Back-N ARQ?

Summary

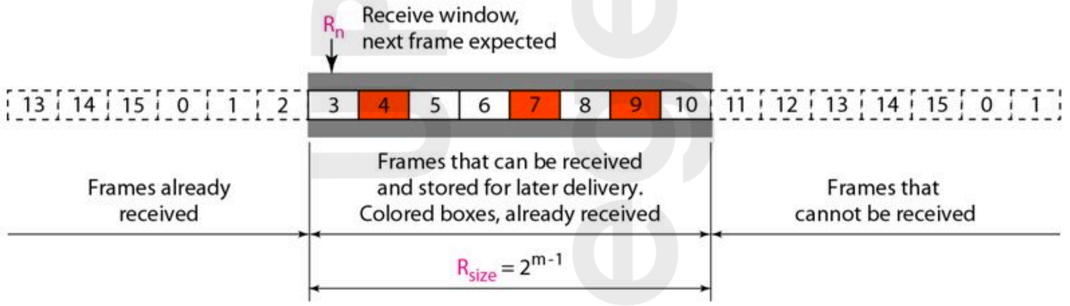
- Go-Back-N ARQ simplifies the process at the receiver site.
  - Receiver keeps track of only one variable.
  - No need to buffer out-of-order frames; just discarded.
  - This is inefficient for a noisy link with high probability of frames being damaged, which means resending of multiple frames.
- This resending uses up the bandwidth and slows down the transmission.
- For noisy links, there is another mechanism that does not resend *N* frames when just one frame is damaged; only the damaged frame is resent.
- This mechanism is called *Selective Repeat ARQ*.
- More efficient for noisy links, but processing at the receiver is more complex.

3. Selective Repeat Automatic Repeat Request

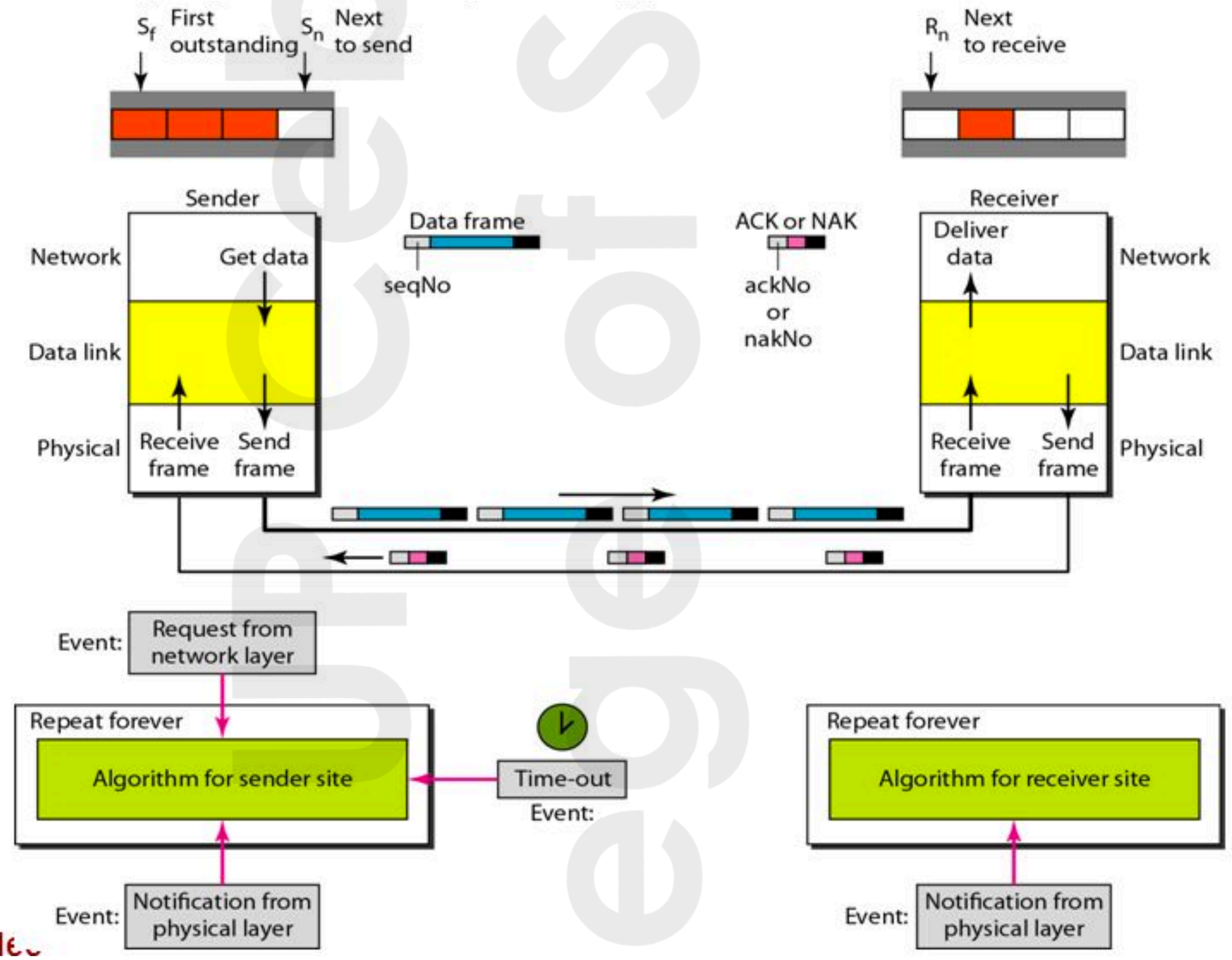
- Short: Selective Repeat ARQ.
- Uses two windows: (1) send window and (2) receive window.
- Difference from Go-Back-N in terms of send window:
  - Size of send window is much smaller; it's 2^{m-1}. *(To be discussed later.)*
  - The receive window is the same size as the send window.
  - Similar variables are used.
  - Each frame has its own timer.
- Smaller window size means less efficiency in filling pipe, but the fact that there are fewer duplicate frames can compensate for this.



- In terms of receive window:
  - Size of receive window is same with send window; it's 2^{m-1}.
  - Allows as many frames as the size of the receive window to arrive out of order and be kept until there is a set of in-order frames to be delivered to upper layer.
- However, receiver never delivers frames out of order to the upper layer.



Design of Selective Repeat ARQ

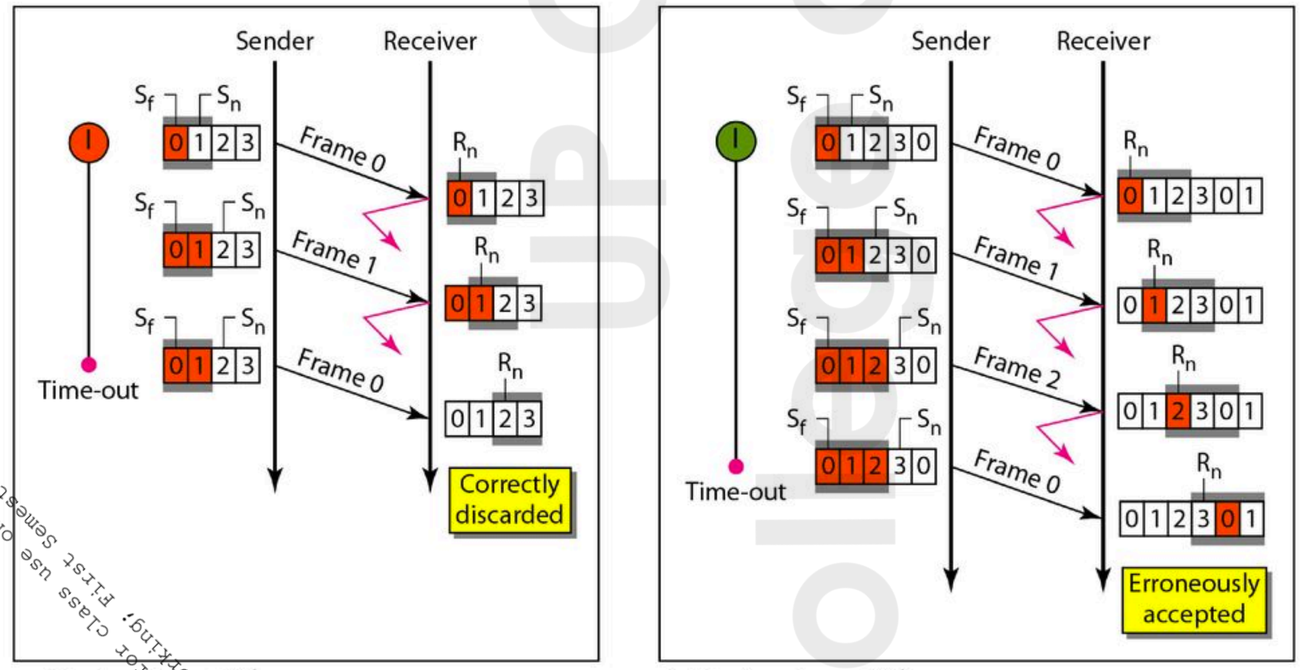


Kyung Hee

Reason why window sizes must be at most one half of 2^m…

We can now show why the size of the sender and receiver windows must be at most one-half of 2^m. For an example, we choose $m = 2$, which means the size of the window is $2^m/2$, or 2. Figure 11.21 compares a window size of 2 with a window size of 3.

If the size of the window is 2 and all acknowledgments are lost, the timer for frame 0 expires and frame 0 is resent. However, the window of the receiver is now expecting frame 2, not frame 0, so this duplicate frame is correctly discarded. When the size of the window is 3 and all acknowledgments are lost, the sender sends a duplicate of frame 0. However, this time, the window of the receiver expects to receive frame 0 (0 is part of the window), so it accepts frame 0, not as a duplicate, but as the first frame in the next cycle. This is clearly an error.



a. Window size 2^{m-1}

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

Algorithm (Sender)

Algorithm 11.9 *Sender-site Selective Repeat algorithm*

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

(continued)

11.61

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

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

11.62

(continued)

Algorithm (Receiver)

Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

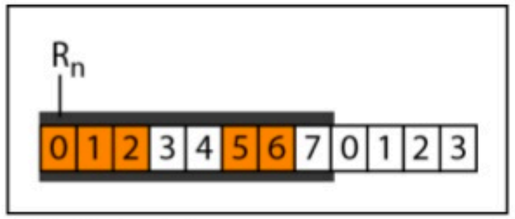
```
1  Rn = 0;
2  NakSent = false;
3  AckNeeded = false;
4  Repeat(for all slots)
5      Marked(slot) = false;
6
7  while (true)                              //Repeat forever
8  {
9    WaitForEvent();
10
11   if(Event(ArrivalNotification))          /Data frame arrives
12   {
13      Receive(Frame);
14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

11.64

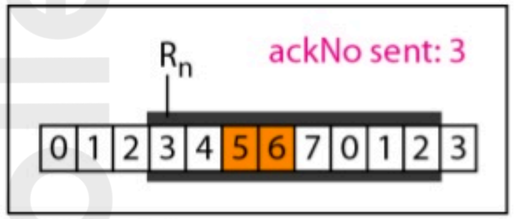
Algorithm 11.10 *Receiver-site Selective Repeat algorithm*

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

11.65



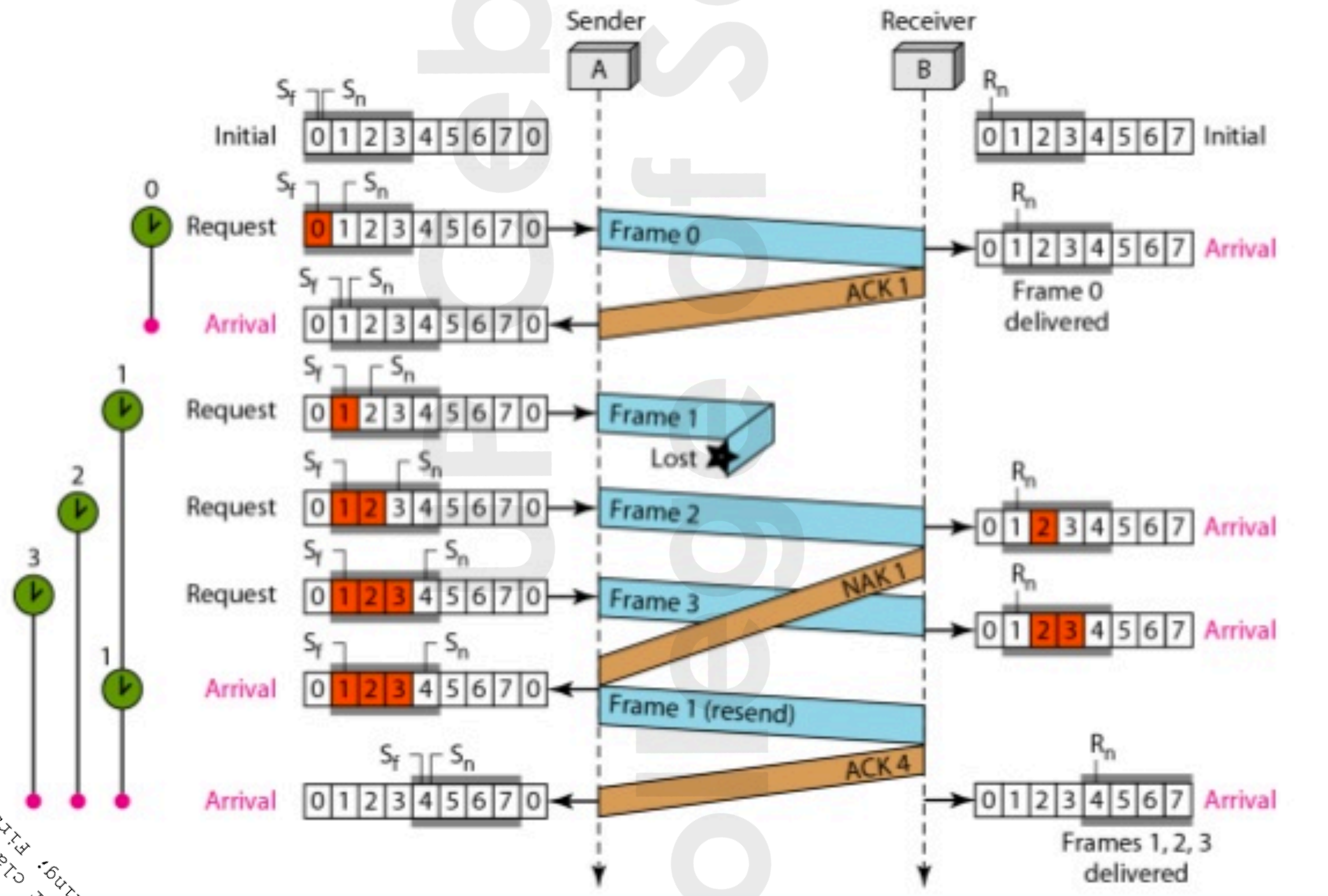
a. Before delivery



b. After delivery

Example 11.8

This example is similar to Example 11.3 in which frame 1 is lost. We show how Selective Repeat behaves in this case. Figure 11.23 shows the situation.



Piggybacking

- A technique in which control information (ACK/NAK) about arrived (or lost) frames can be carried in both directions in order to facilitate bidirectional communication.

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