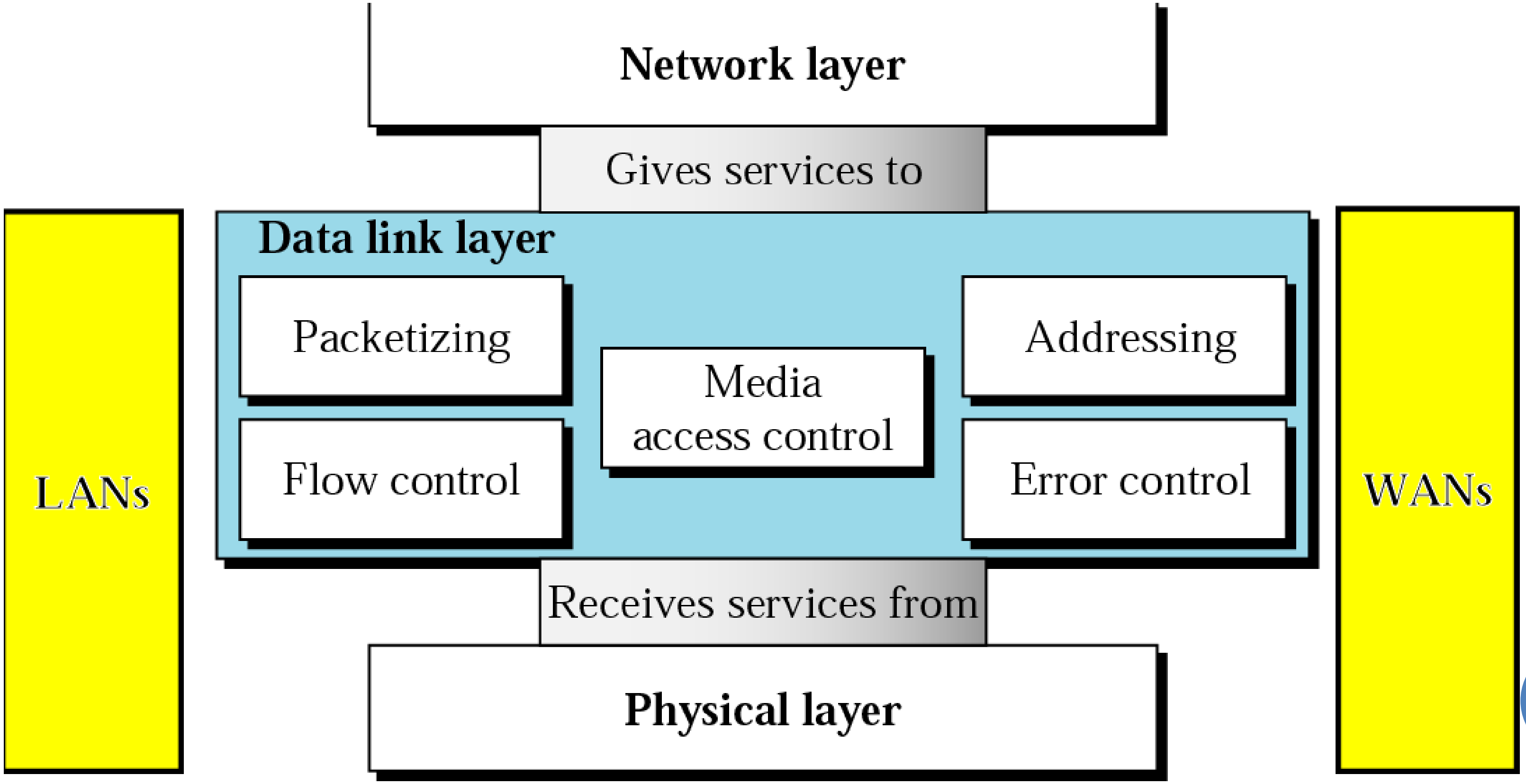
DATA LINK LAYER

MUKESH CHINTA
Asst Prof, CSE
VRSEC

Overview of DLL

The data link layer 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 include *framing*, *addressing*, *flow control*, *error control*, *and media access control.*



DLL Design Issues

❶ Services Provided to the Network Layer

- The network layer wants to be able to send packets to its neighbors without worrying about the details of getting it there in one piece.

❷ Framing

- Group the physical layer bit stream into units called **frames**. Frames are nothing more than "packets" or "messages". By convention, we use the term "frames" when discussing DLL.

❸ Error Control

- Sender **checksums** the frame and transmits checksum together with data. Receiver re-computes the **checksum** and compares it with the received value.

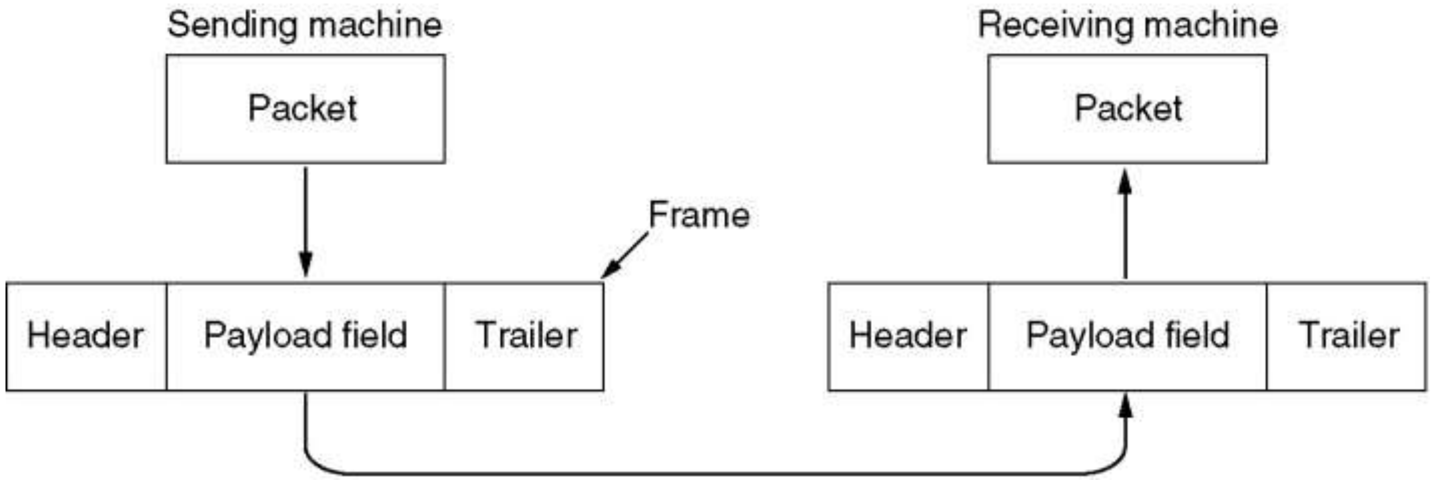
❹ Flow Control

- Prevent a **fast sender** from overwhelming a **slower receiver**.

Data Link Layer Design Issues

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

For this, the data link layer takes the packets it gets from the network layer and encapsulates them into frames for transmission. Each frame contains a frame header, a payload field for holding the packet, and a frame trailer



SERVICES PROVIDED TO THE NETWORK LAYER

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

- The data link layer can be designed to offer various services. The actual services offered can vary from system to system. Three reasonable possibilities that are commonly provided are

1) Unacknowledged Connectionless service

2) Acknowledged Connectionless service

3) Acknowledged Connection-Oriented service

Unacknowledged Connectionless service

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

- No logical connection is established beforehand or released afterward. If a frame is lost due to noise on the line, no attempt is made to detect the loss or recover from it in the data link layer.

- This class of service is appropriate when the error rate is very low so that recovery is left to higher layers. It is also appropriate for real-time traffic, such as voice, in which late data are worse than bad data. Most LANs use unacknowledged connectionless service in the data link layer.

Acknowledged Connectionless service

- When this service is offered, there are still no logical connections used, but each frame sent is individually acknowledged.

- In this way, the sender knows whether a frame has arrived correctly. If it has not arrived within a specified time interval, it can be sent again. This service is useful over unreliable channels, such as wireless systems.

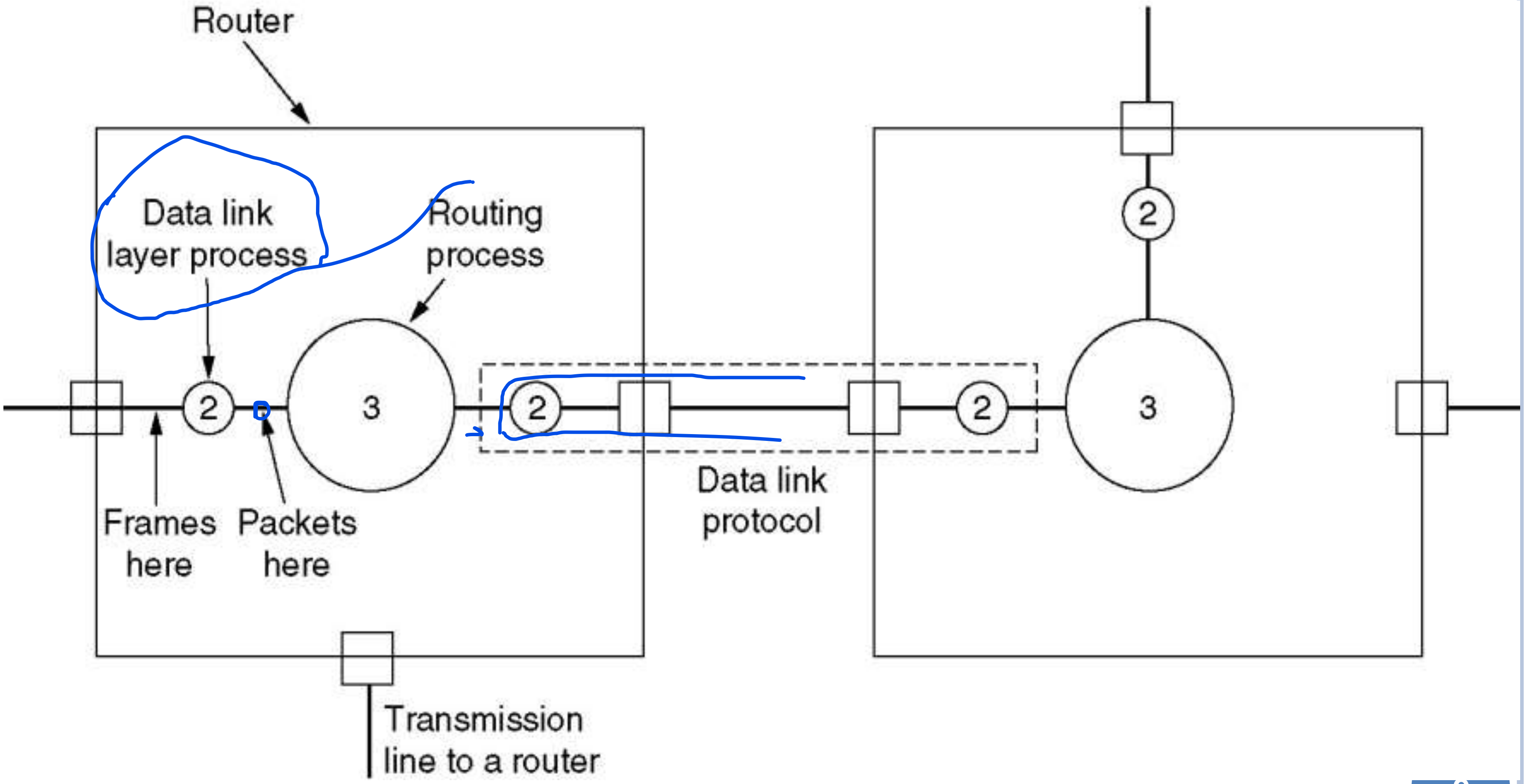
- Adding Ack in the DLL rather than in the Network Layer is just an optimization and not a requirement. If individual frames are acknowledged and retransmitted, entire packets get through much faster. On reliable channels, such as fiber, the overhead of a heavyweight data link protocol may be unnecessary, but on wireless channels, with their inherent unreliability, it is well worth the cost.

ACKNOWLEDGED CONNECTION-ORIENTED SERVICE

- Here, the source and destination machines establish a connection before any data are transferred. Each frame sent over the connection is numbered, and the data link layer guarantees that each frame sent is indeed received. Furthermore, it guarantees that each frame is received exactly once and that all frames are received in the right order.

- When connection-oriented service is used, transfers go through three distinct phases.
  - In the first phase, the connection is established by having both sides initialize variables and counters needed to keep track of which frames have been received and which ones have not.
  - In the second phase, one or more frames are actually transmitted.
  - In the third and final phase, the connection is released, freeing up the variables, buffers, and other resources used to maintain the connection

Placement of Data Link Protocol



Framing

- DLL translates the physical layer's raw bit stream into discrete units (messages) called frames.

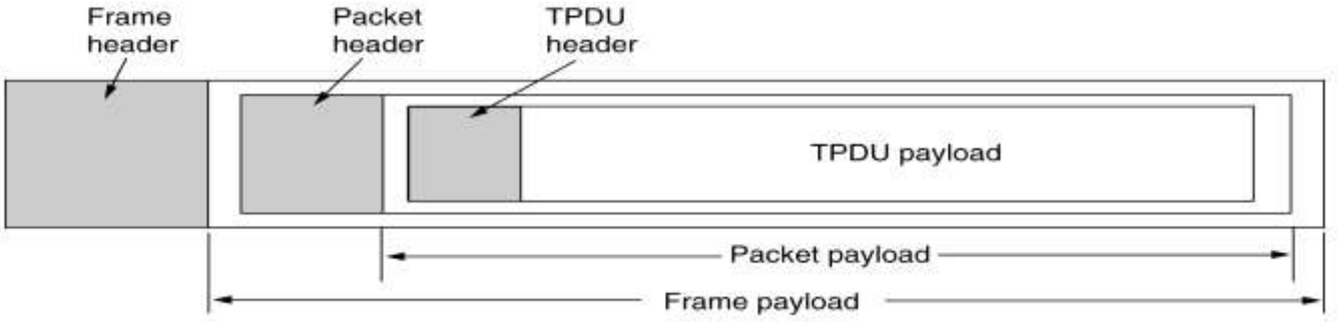
- How can frame be transmitted so the receiver can detect frame boundaries? That is, how can the receiver recognize the start and end of a frame?

  ❶ Character Count

  ❷ Flag byte with Byte Stuffing

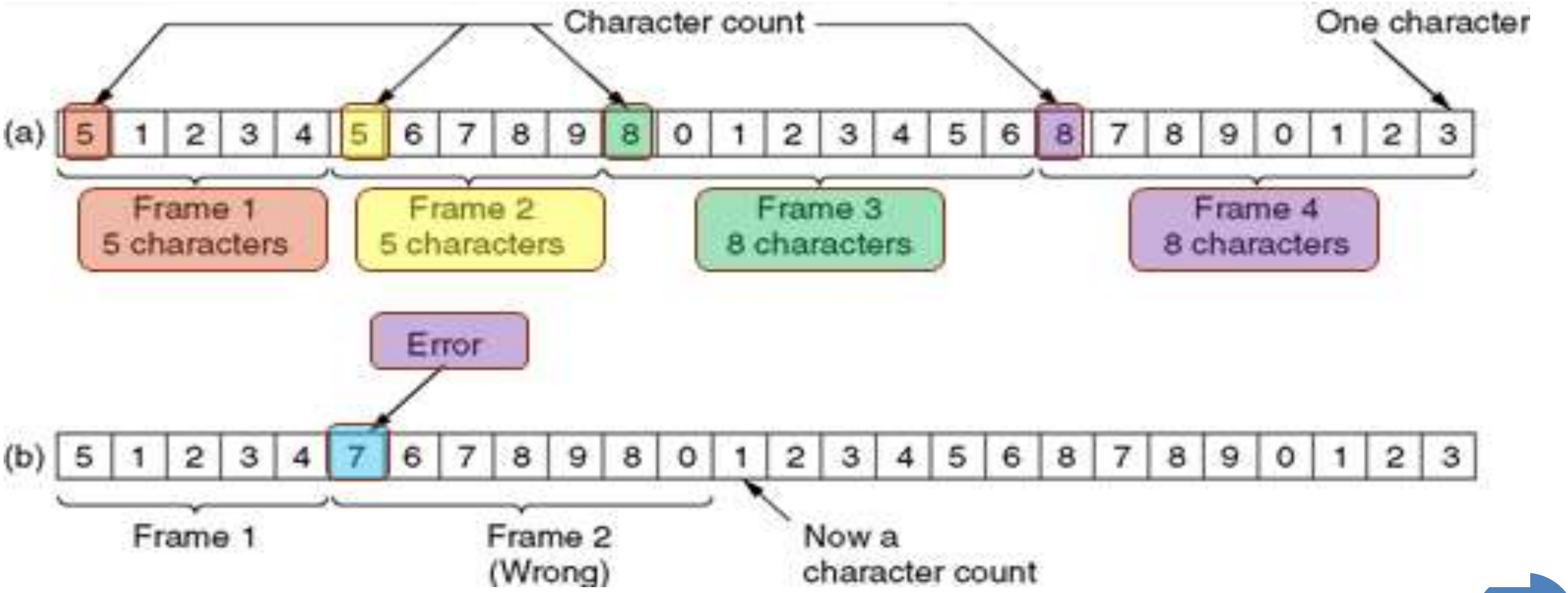
  ❸ Starting and ending flag with bite stuffing

  ❹ Encoding Violations



FRAMING – CHARACTER COUNT

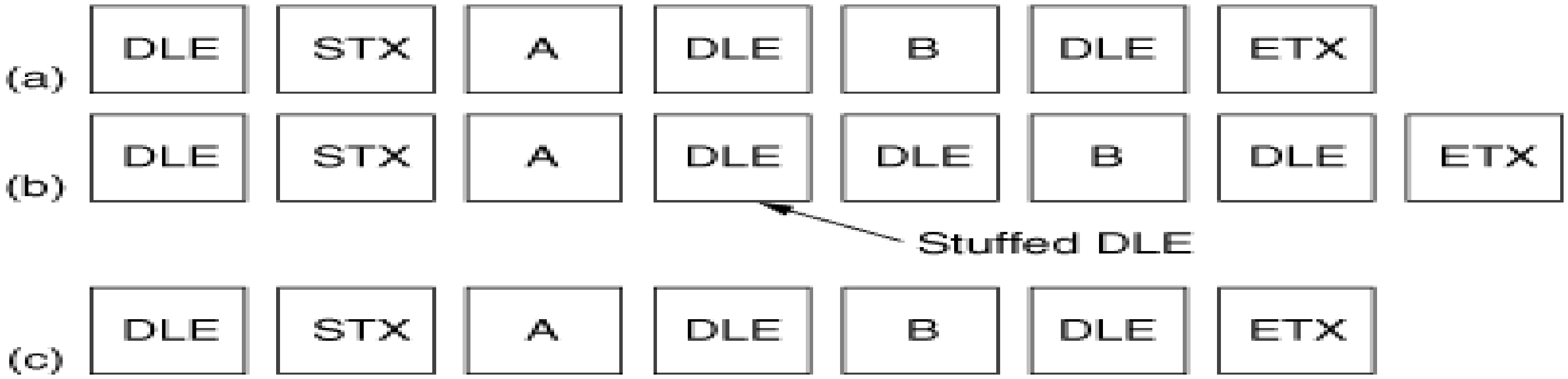
- The first framing method uses a field in the header to specify the number of characters in the frame. When the data link layer at the destination sees the character count, it knows how many characters follow and hence where the end of the frame is.



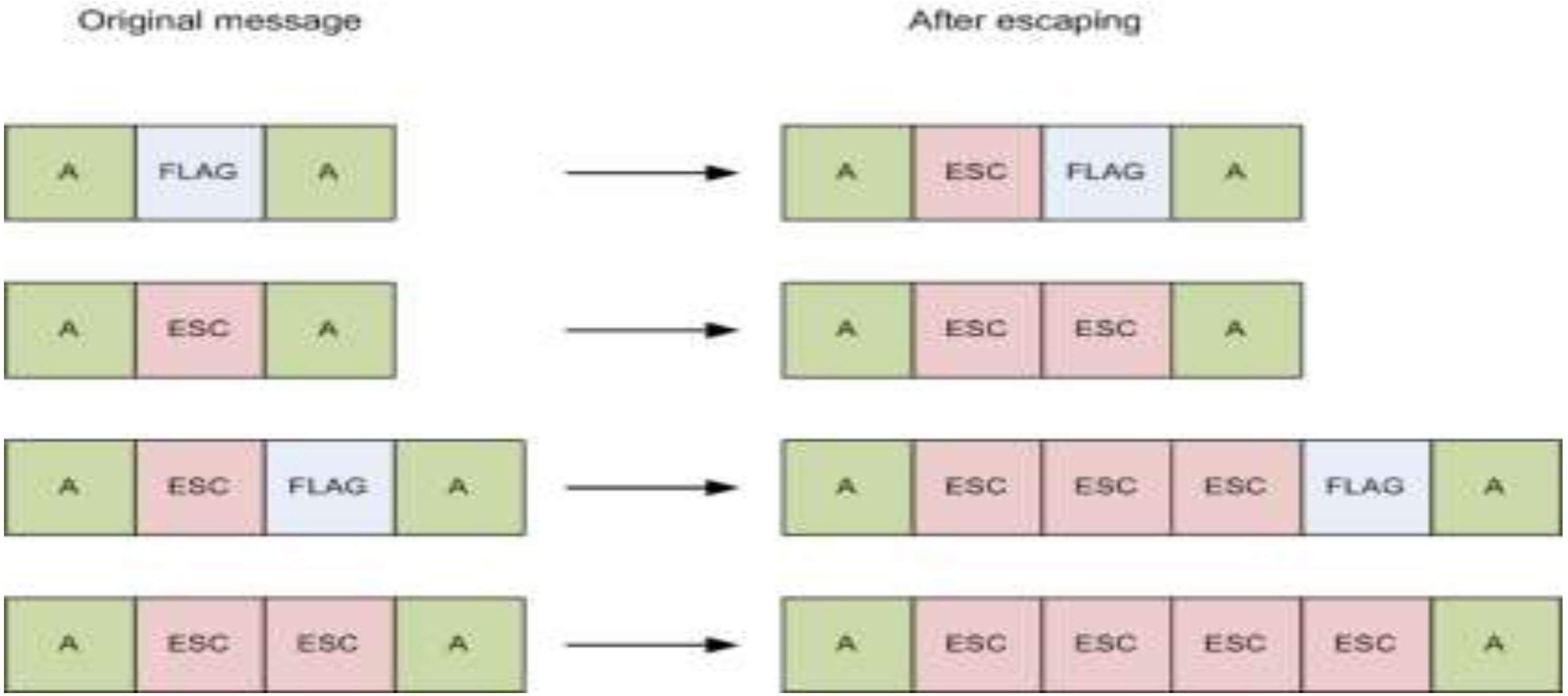
The trouble with this algorithm is that the count can be garbled by a transmission error.

FRAMING – BYTE STUFFING

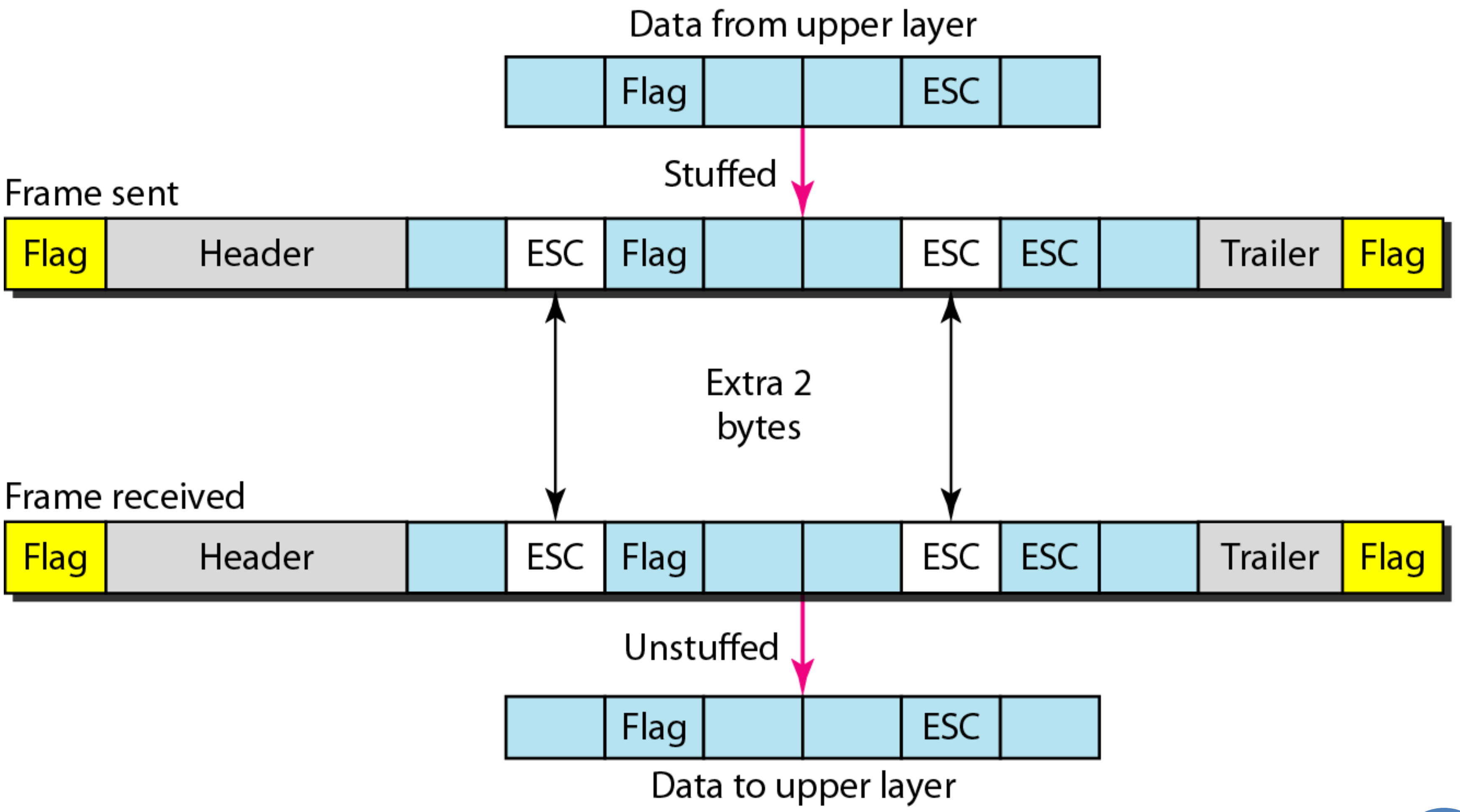
- Use reserved characters to indicate the start and end of a frame. For instance, use the two-character sequence DLE STX (Data-Link Escape, Start of TeXt) to signal the beginning of a frame, and the sequence DLE ETX (End of TeXt) to flag the frame's end.
- The second framing method, Starting and ending character stuffing, gets around the problem of resynchronization after an error by having each frame start with the ASCII character sequence DLE STX and end with the sequence DLE ETX.
- **Problem: What happens if the two-character sequence DLE ETX happens to appear in the frame itself?**
- **Solution**: Use *character stuffing*; within the frame, replace every occurrence of DLE with the two-character sequence DLE DLE. The receiver reverses the processes, replacing every occurrence of DLE DLE with a single DLE.
- Example: If the frame contained ``A B DLE D E DLE'', the characters transmitted over the channel would be ``DLE STX A B DLE DLE D E DLE DLE DLE ETX''.
- ***Disadvantage: character is the smallest unit that can be operated on; not all architectures are byte oriented.***



(a) Data sent by the network layer. (b) Data after being character stuffed by the data link layer. (c) Data passed to the network layer on the receiving side.

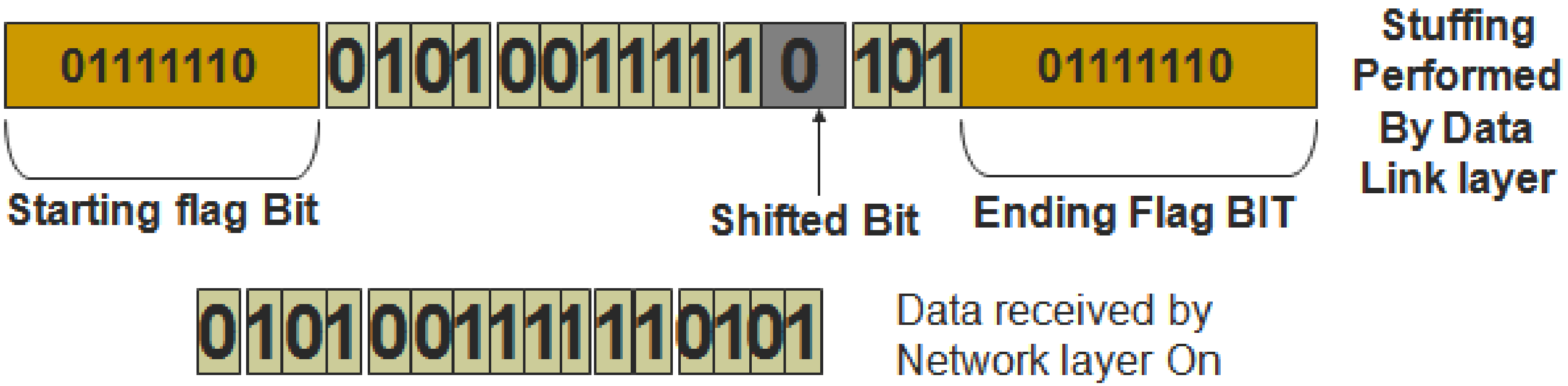


Byte stuffing and unstuffing



FRAMING – BIT STUFFING

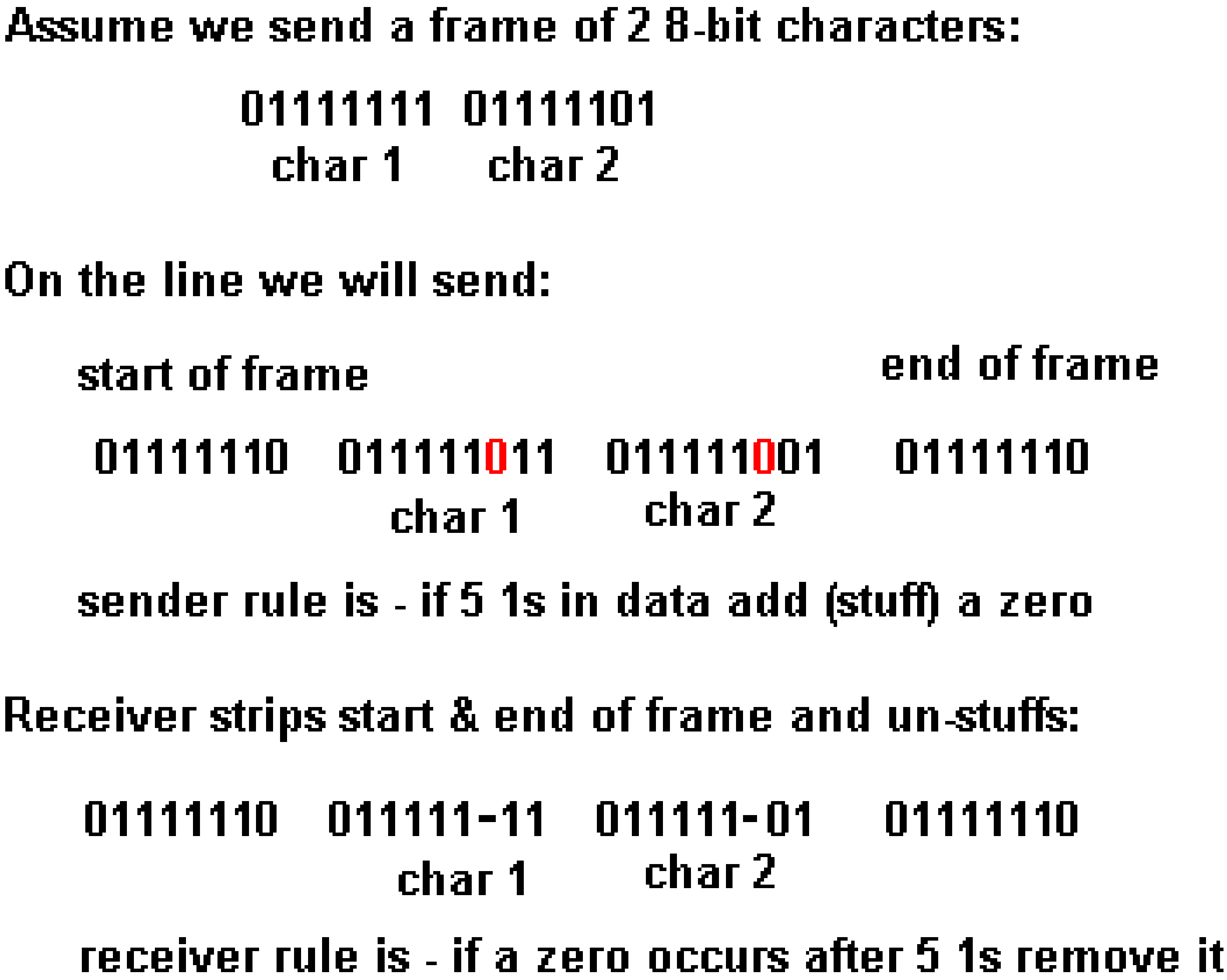
- This technique allows data frames to contain an arbitrary number of bits and allows character codes with an arbitrary number of bits per character. It works like this. Each frame begins and ends with a special bit pattern, 01111110 (in fact, a flag byte).
- Whenever the sender's data link layer encounters five consecutive 1s in the data, it automatically stuffs a 0 bit into the outgoing bit stream.
- This bit stuffing is analogous to byte stuffing, in which an escape byte is stuffed into the outgoing character stream before a flag byte in the data. When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e., deletes) the 0 bit



Data received by Network layer On reciever side After Performing Destuffing By data link Layer

BIT stuffing

BIT STUFFING EXAMPLE



Assume we send a frame of 2 8-bit characters:

01111111 01111101

char 1 char 2

On the line we will send:

start of frame end of frame

01111110 0111110011 0111110001 01111110

char 1 char 2

sender rule is - if 5 1s in data add (stuff) a zero

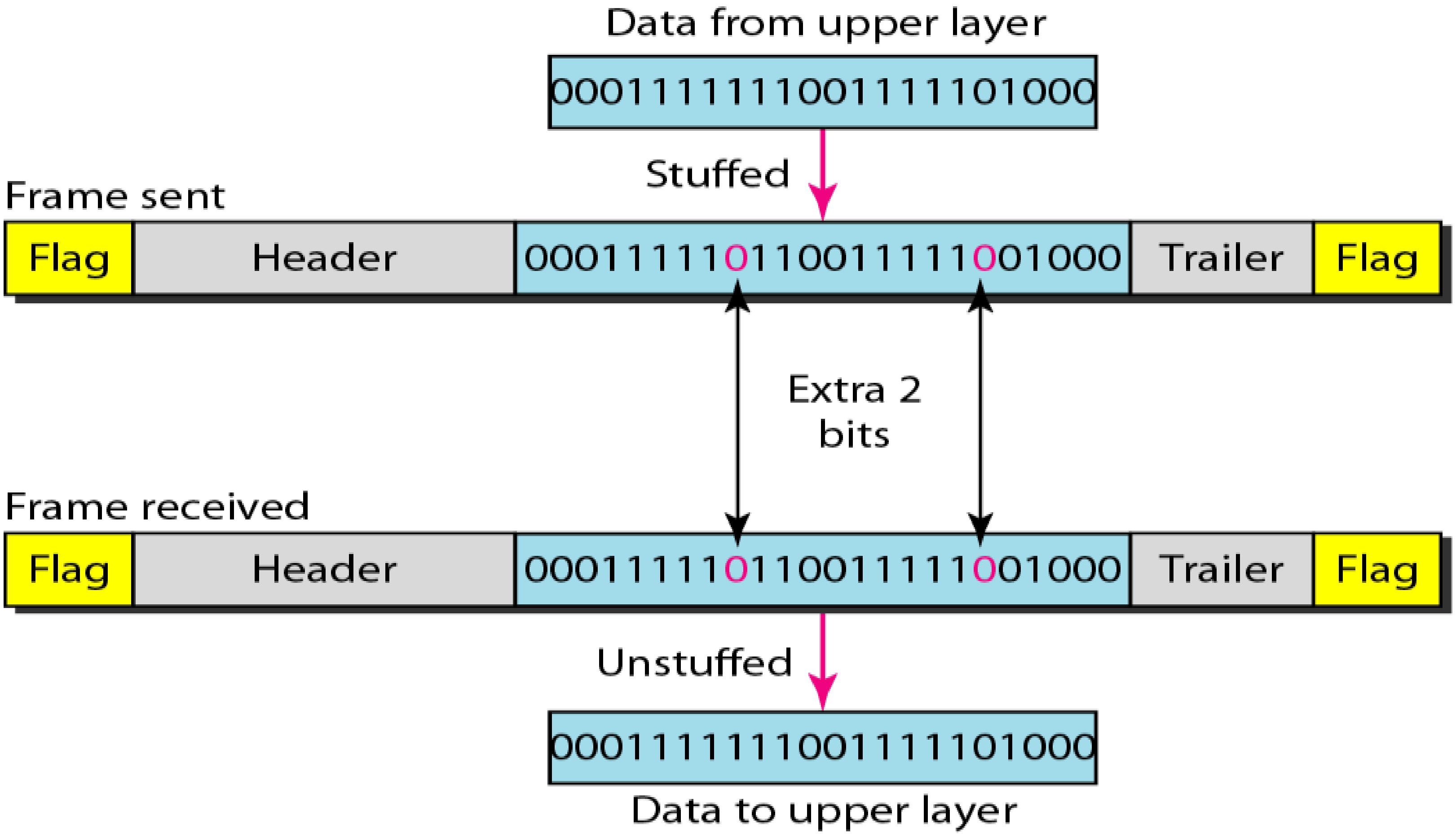
Receiver strips start & end of frame and un-stuffs:

01111110 011111-11 011111-01 01111110

char 1 char 2

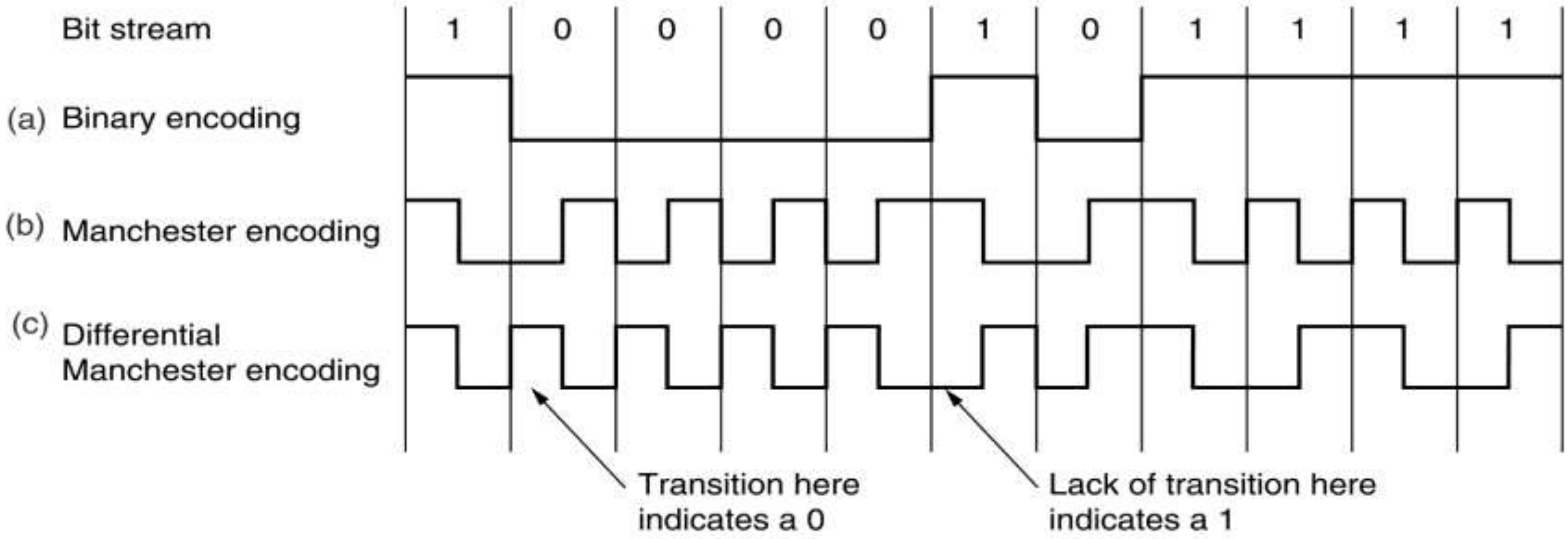
receiver rule is - if a zero occurs after 5 1s remove it

Byte stuffing and unstuffing



PHYSICAL LAYER CODING VIOLATIONS

- This Framing Method is used only in those networks in which Encoding on the Physical Medium contains some redundancy.
- Some **LANs** **e**ncode **e**ach bit of data by using two Physical Bits i.e. Manchester coding is Used. Here, Bit **1** is encoded into high-low(10) pair and Bit **0** is encoded into low-high(01) pair.
- The scheme means that every data bit has a transition in the middle, making it easy for the receiver to locate the bit boundaries. The combinations high-high and low-low are not used for data but are used for delimiting frames in some protocols.


Bit stream
1 0 0 0 0 1 0 1 1 1 1
(a) Binary encoding
(b) Manchester encoding
(c) Differential Manchester encoding
Transition here indicates a 0
Lack of transition here indicates a 1

ERROR CONTROL

- Error control is concerned with insuring that all frames are eventually delivered (possibly in order) to a destination. How? Three items are required.

- **Acknowledgements:** Typically, reliable delivery is achieved using the “acknowledgments with retransmission" paradigm, whereby the receiver returns a special acknowledgment (ACK) frame to the sender indicating the correct receipt of a frame.
  - In some systems, the receiver also returns a negative acknowledgment (NACK) for incorrectly-received frames. This is nothing more than a hint to the sender so that it can retransmit a frame right away without waiting for a timer to expire.

- **Timers:** One problem that simple ACK/NACK schemes fail to address is recovering from a frame that is lost, and as a result, fails to solicit an ACK or NACK. What happens if an ACK or NACK becomes lost?
  - Retransmission timers are used to resend frames that don't produce an ACK. When sending a frame, schedule a timer to expire at some time after the ACK should have been returned. If the timer goes o, retransmit the frame.

- **Sequence Numbers:** Retransmissions introduce the possibility of duplicate frames. To suppress duplicates, add sequence numbers to each frame, so that a receiver can distinguish between new frames and old copies.

Flow Control

- *Flow control* deals with throttling the speed of the sender to match that of the receiver.

- Two Approaches:
  - **feedback-based flow control**, the receiver sends back information to the sender giving it permission to send more data or at least telling the sender how the receiver is doing
  - **rate-based flow control**, the protocol has a built-in mechanism that limits the rate at which senders may transmit data, without using feedback from the receiver.

- Various Flow Control schemes uses a common protocol that contains well-defined rules about when a sender may transmit the next frame. These rules often prohibit frames from being sent until the receiver has granted permission, either implicitly or explicitly.

Error Correction and Detection

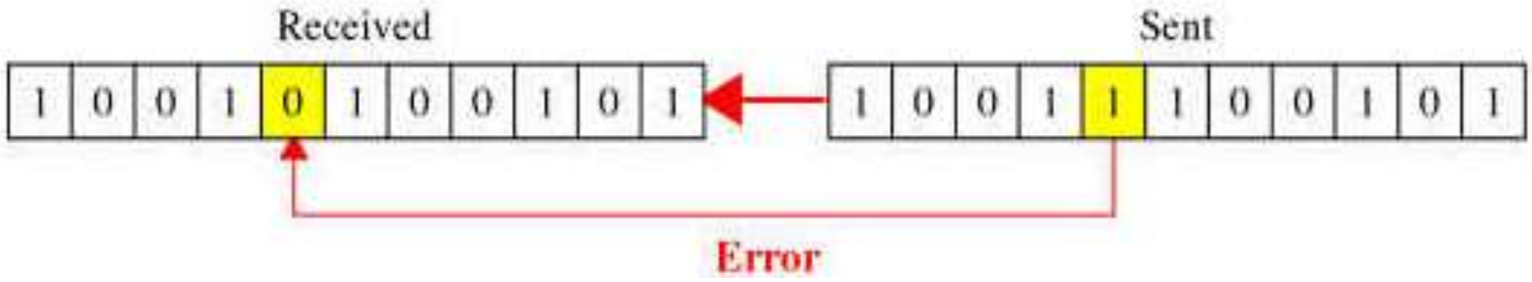
- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
  - In data communication, line noise is a fact of life (e.g., signal attenuation, natural phenomenon such as lightning, and the telephone repairman).
- As more bits are packed onto a square centimeter of disk storage, as communications transmission speeds increase, the likelihood of error increases-- sometimes geometrically.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.

- Detecting and correcting errors requires ***redundancy*** -- sending additional information along with the data.

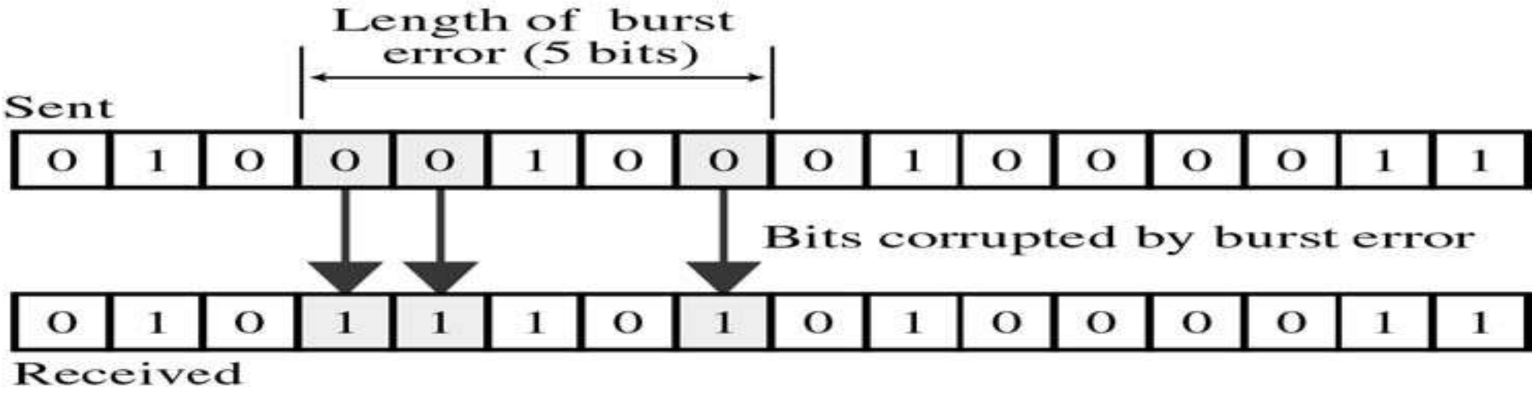
Types of Errors

- There are two main types of errors in transmissions:

1. **Single bit error** : It means only one bit of data unit is changed from 1 to 0 or from 0 to 1.

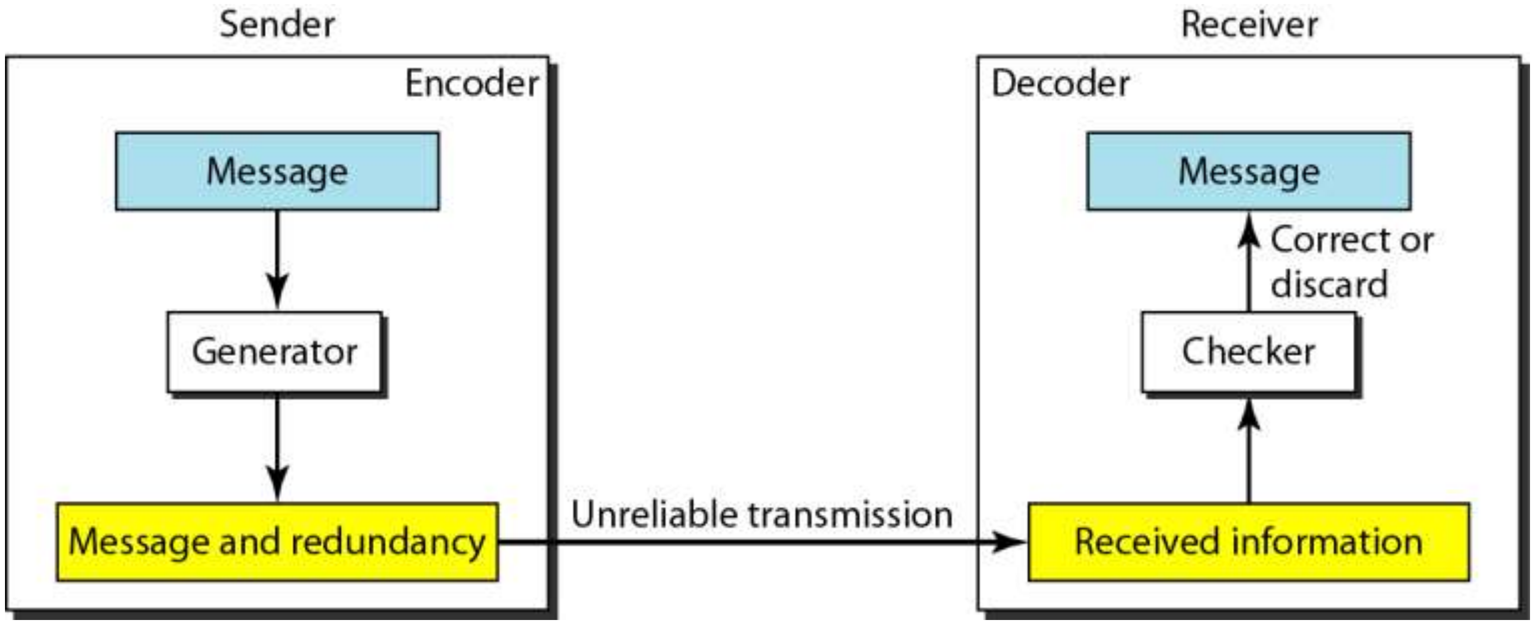


2. **Burst error** : It means two or more bits in data unit are changed from 1 to 0 from 0 to 1. In burst error, it is not necessary that only consecutive bits are changed. The length of burst error is measured from first changed bit to last changed bit



Error Detection Vs Error Correction

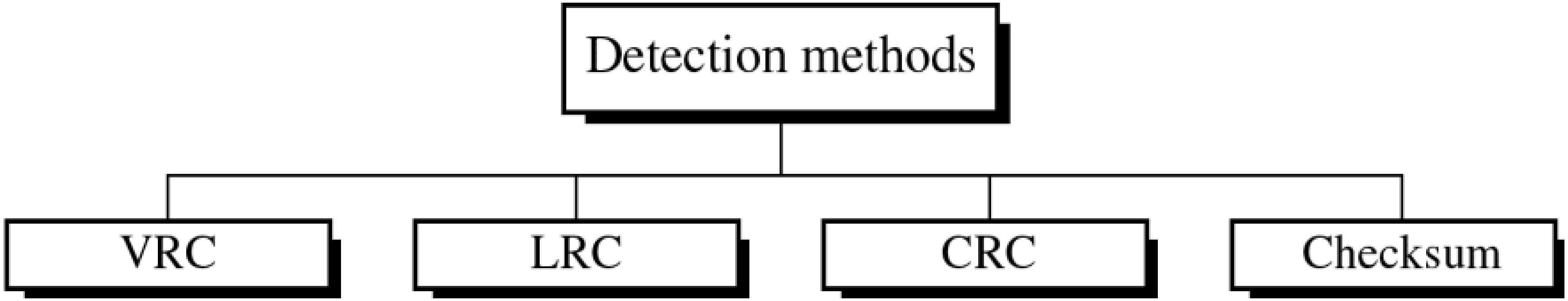
- There are two types of attacks against errors:
- **Error Detecting Codes:** Include enough redundancy bits to *detect* errors and use ACKs and retransmissions to recover from the errors.
- **Error Correcting Codes:** Include enough redundancy to detect *and* correct errors. The use of error-correcting codes is often referred to as forward error correction.



ERROR DETECTION

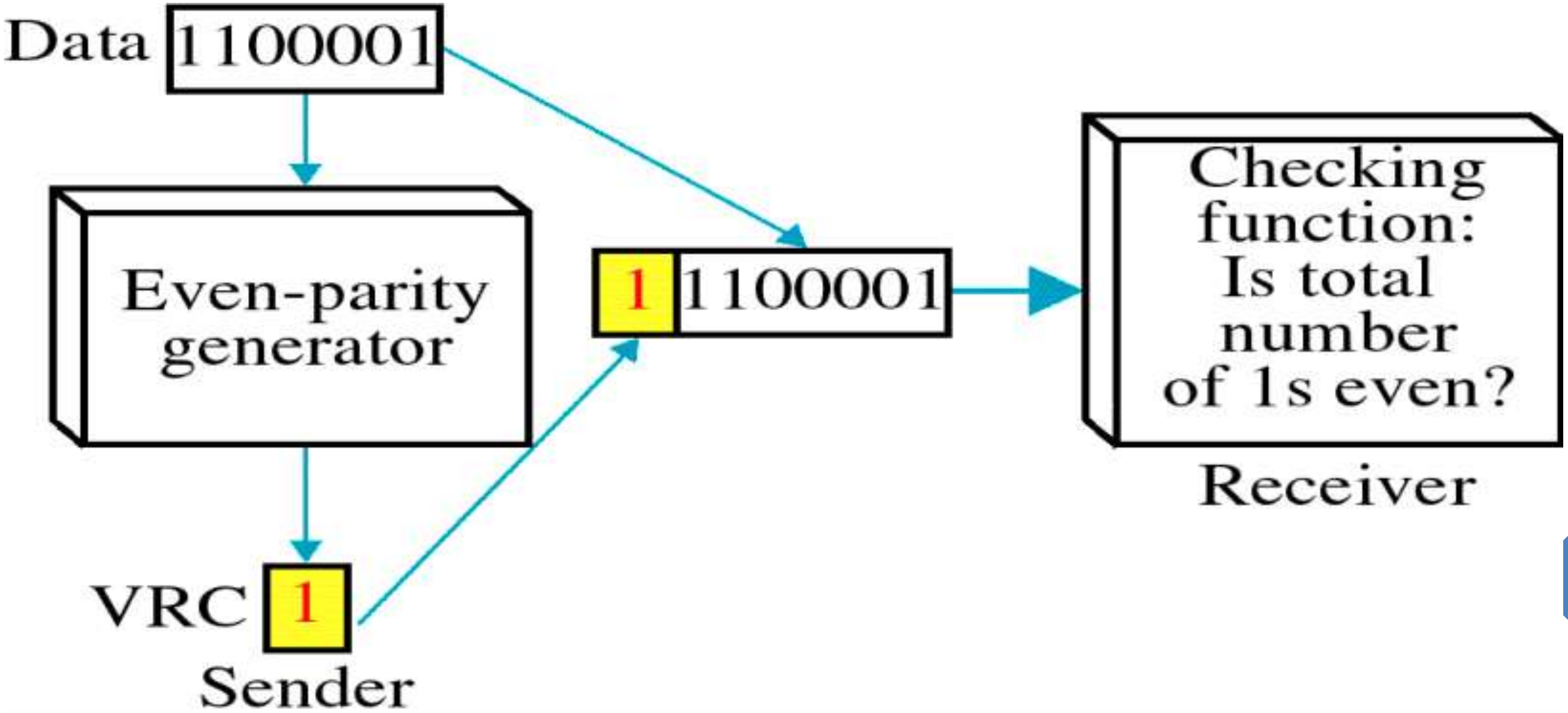
Error detection means to decide whether the received data is correct or not without having a copy of the original message.

Error detection **uses the concept of redundancy**, **which means** adding extra bits for detecting errors at the destination.



Vertical Redundancy Check (VRC)

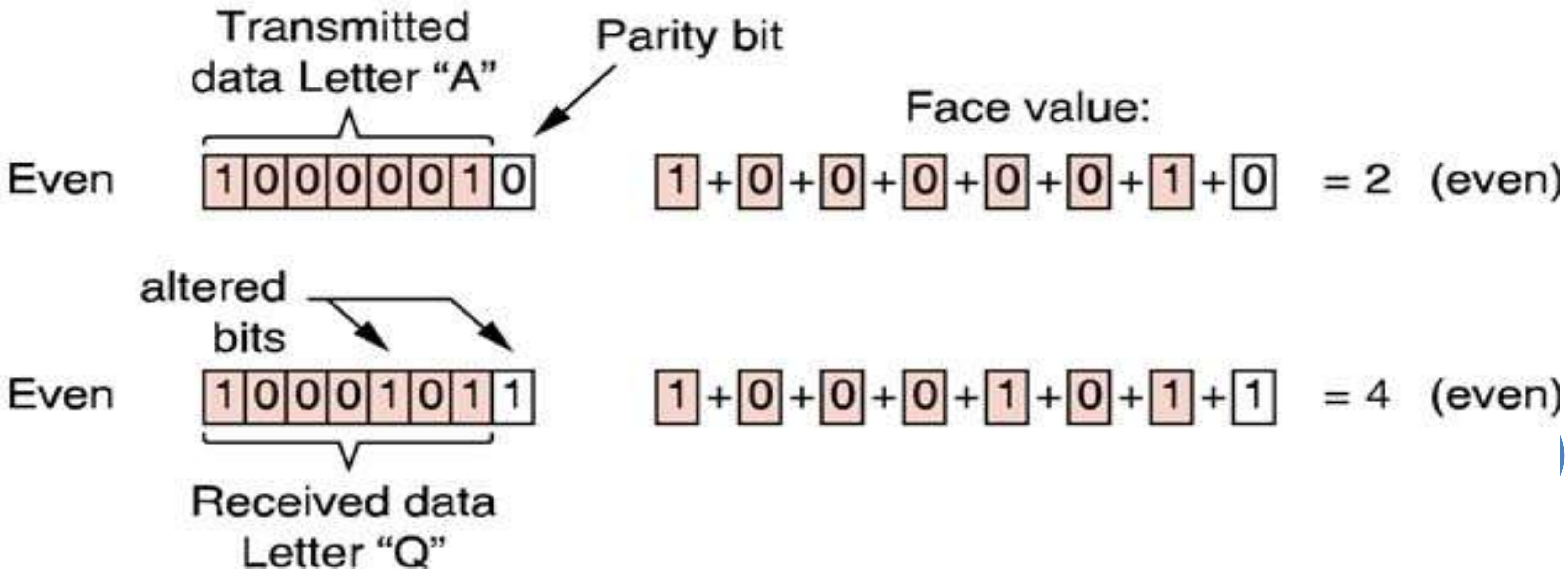
- Append a single bit at the end of data block such that the number of ones is even
  → Even Parity (odd parity is similar)
  0110011 → 01100110
  0110001 → 01100011
- VRC is also known as **Parity Check.** Detects **all odd-number errors** in a data block



EXAMPLE OF VRC

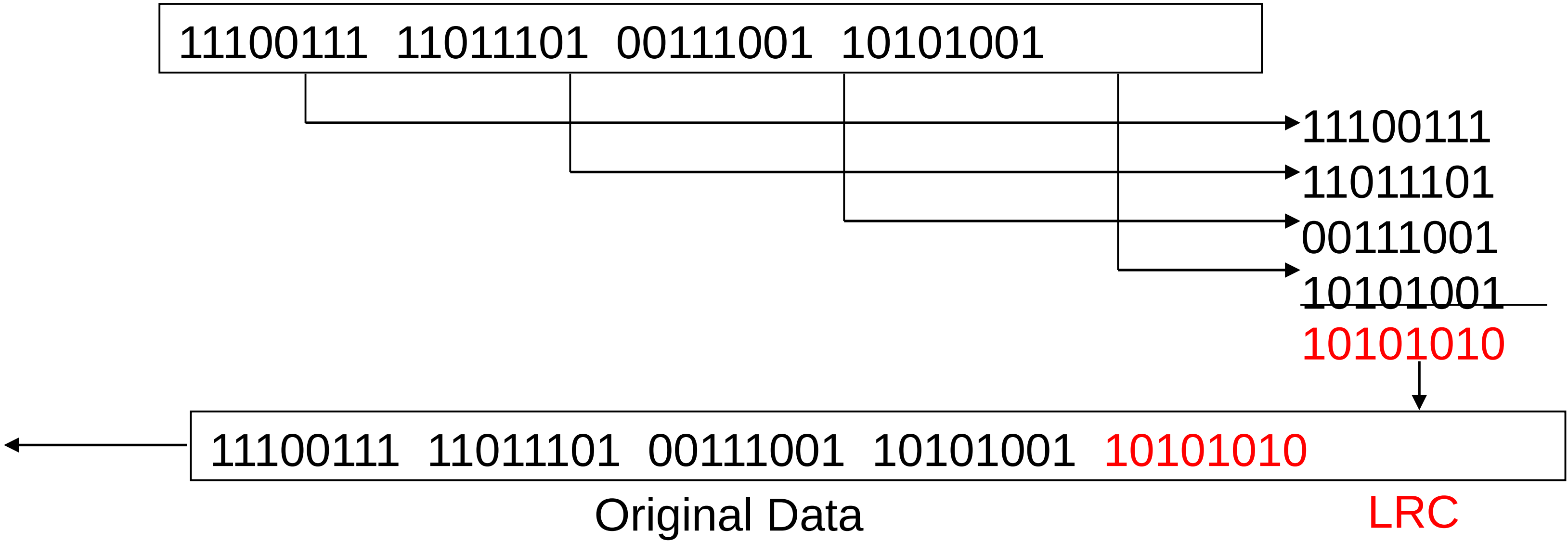
- The problem with parity is that it can only detect odd numbers of bit substitution errors, i.e. 1 bit, 3bit, 5, bit, etc. errors. If there two, four, six, etc. bits which are transmitted in error, using VRC will not be able to detect the error.

Multiple bit errors/even parity



LONGITUDINAL REDUNDANCY CHECK (LRC)

- Longitudinal Redundancy Checks (LRC) seek to overcome the weakness of simple, bit-oriented, one-directional parity checking.
- LRC adds a new character (instead of a bit) called the Block Check Character (BCC) to each block of data. Its determined like parity, but counted longitudinally through the message (also vertically)
- Its has better performance over VRC as it detects 98% of the burst errors (>10 errors) but less capable of detecting single errors
- If two bits in one data units are damaged and two bits in exactly the same positions in another data unit are also damaged, the LRC checker will not detect an error.



11100111 11011101 00111001 10101001

11100111
11011101
00111001
10101001
10101010

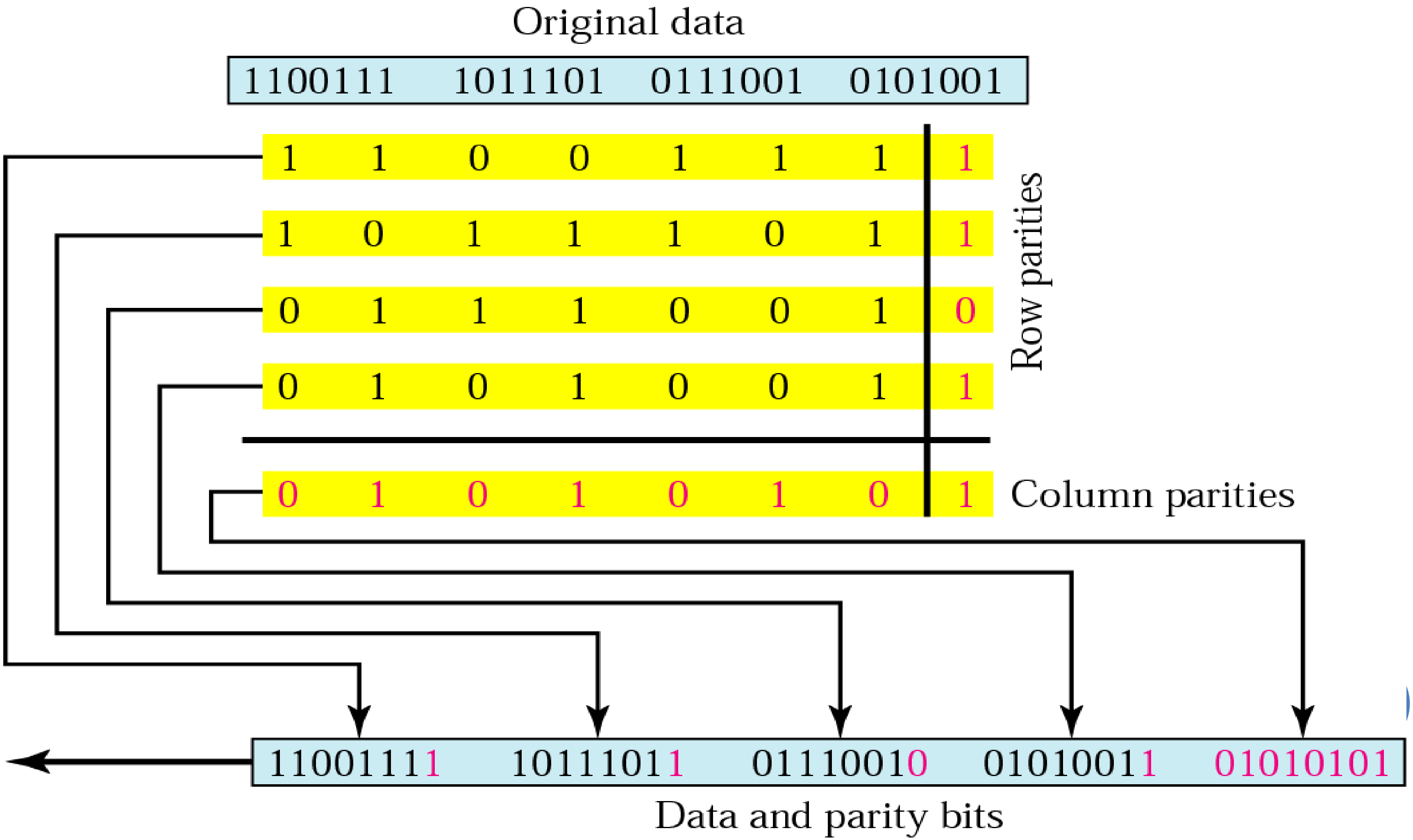
11100111 11011101 00111001 10101001 10101010

Original Data

LRC

TWO DIMENSIONAL PARITY CHECK

- Upon receipt, each character is checked according to its VRC parity value and then the entire block of characters is verified using the LRC block check character.



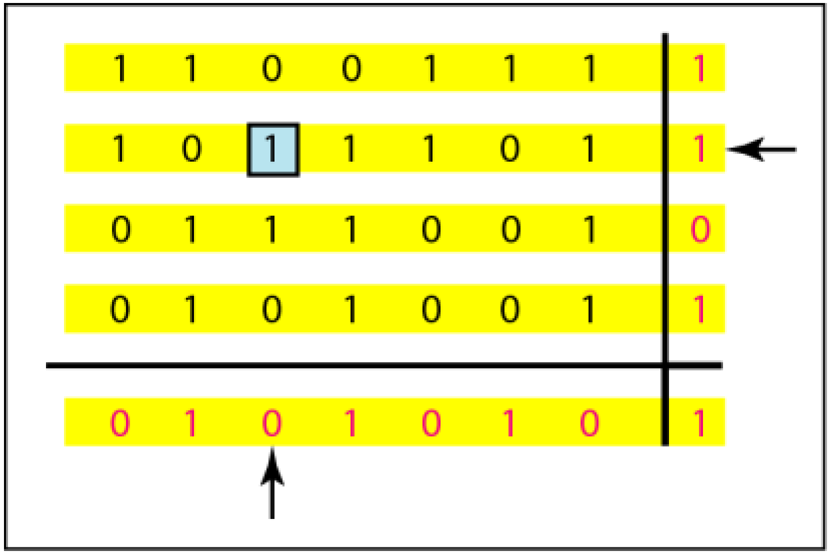
Original data

1100111 1011101 0111001 0101001

1	1	0	0	1	1	1	1	
1	0	1	1	1	0	1	1	Row parities
0	1	1	1	0	0	1	0	
0	1	0	1	0	0	1	1	
0	1	0	1	0	1	0	1	Column parities

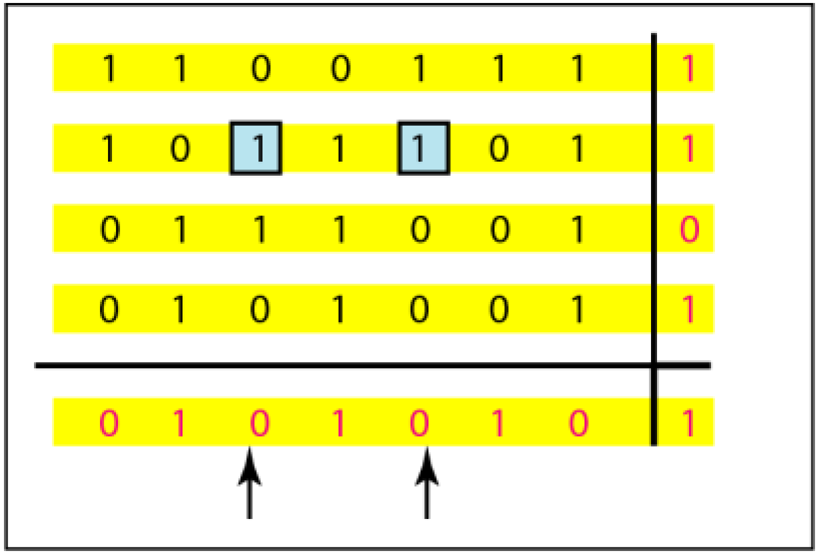
11001111 10111011 01110010 01010011 01010101

Data and parity bits



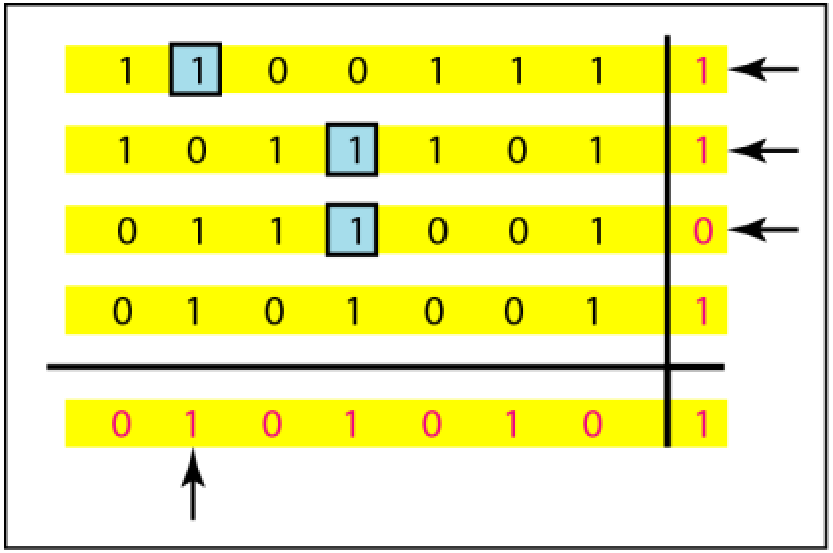
1	1	0	0	1	1	1	1
1	0	$\boxed{1}$	1	1	0	1	1
0	1	1	1	0	0	1	0
0	1	0	1	0	0	1	1
0	1	0	1	0	1	0	1

b. One error affects two parities



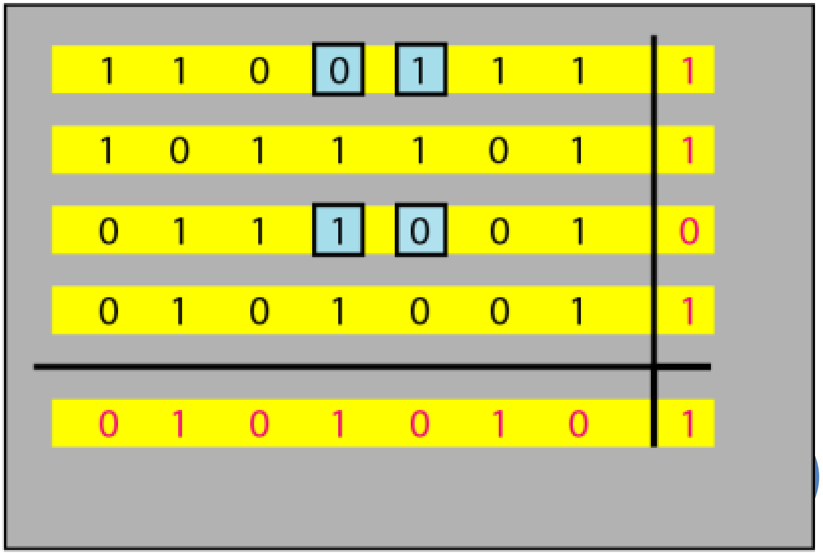
1	1	0	0	1	1	1	1
1	0	$\boxed{1}$	1	$\boxed{1}$	0	1	1
0	1	1	1	0	0	1	0
0	1	0	1	0	0	1	1
0	1	0	1	0	1	0	1

c. Two errors affect two parities



1	$\boxed{1}$	0	0	1	1	1	1
1	0	1	$\boxed{1}$	1	0	1	1
0	1	1	$\boxed{1}$	0	0	1	0
0	1	0	1	0	0	1	1
0	1	0	1	0	1	0	1

d. Three errors affect four parities

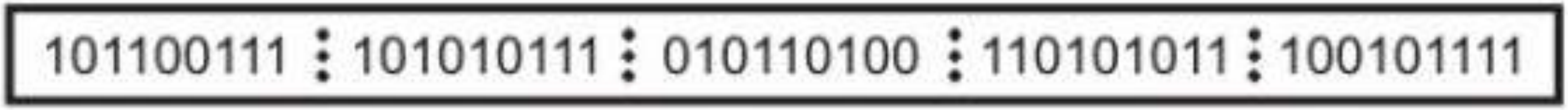


1	1	0	$\boxed{0}$	$\boxed{1}$	1	1	1
1	0	1	1	1	0	1	1
0	1	1	$\boxed{1}$	$\boxed{0}$	0	1	0
0	1	0	1	0	0	1	1
0	1	0	1	0	1	0	1

e. Four errors cannot be detected

Another example of two dimensional parity check

Consider the following bit stream that has been encoded using VRC, LRC and even parity. Locate the error if present



101100111 ⋮ 101010111 ⋮ 010110100 ⋮ 110101011 ⋮ 100101111

Cyclic Redundancy Check (CRC)

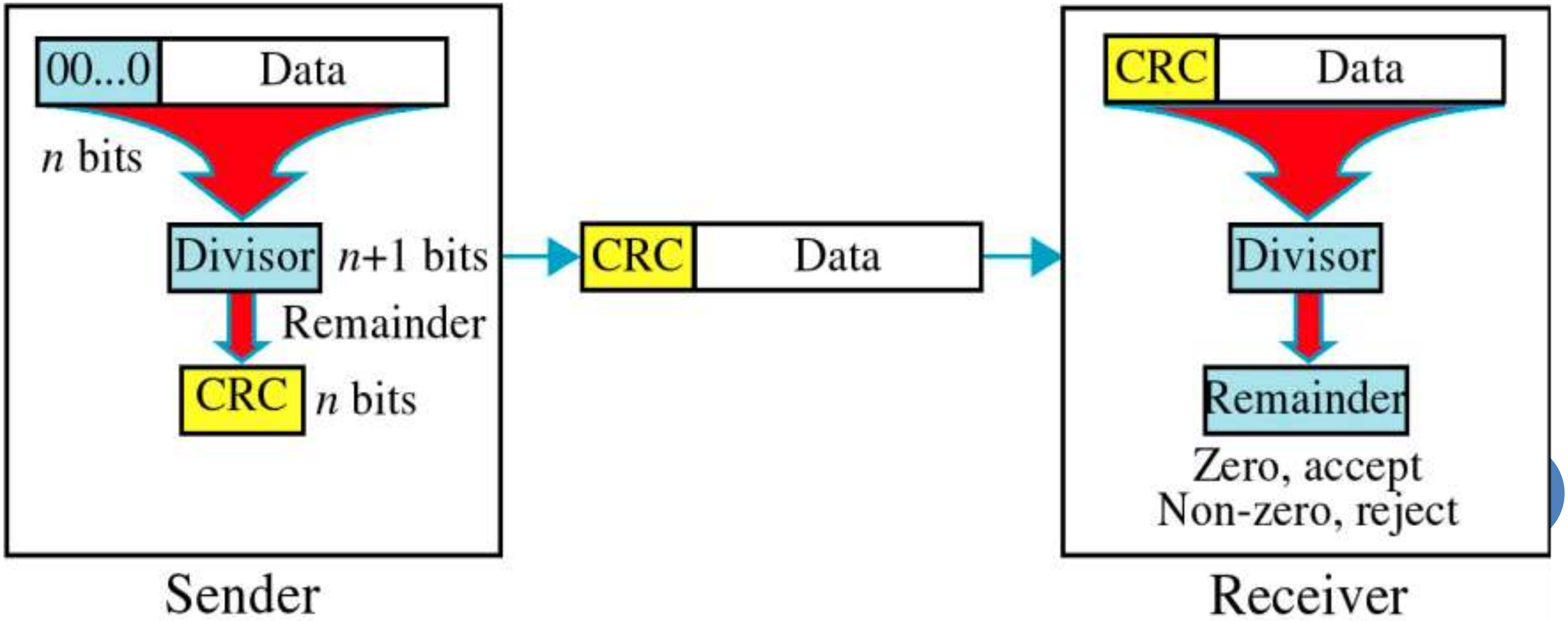
- The cyclic redundancy check, or CRC, is a technique for detecting errors in digital data, but not for making corrections when errors are detected. It is used primarily in data transmission
- In the CRC method, a certain number of check bits, often called a checksum, are appended to the message being transmitted. The receiver can determine whether or not the check bits agree with the data, to ascertain with a certain degree of probability whether or not an error occurred in transmission
- The CRC is based on polynomial arithmetic, in particular, on computing the remainder of dividing one polynomial in GF(2) (Galois field with two elements) by another.
- Can be easily implemented with small amount of hardware
  - Shift registers
  - XOR (for addition and subtraction)

Generator Polynomial

- A cyclic redundancy check (CRC) is a non-secure hash function designed to detect accidental changes to raw computer data, and is commonly used in digital networks and storage devices such as hard disk devices.
- CRCs are so called because the check (data verification) code is a redundancy (it adds zero information) and the algorithm is based on cyclic codes.
- The term CRC may refer to the check code or to the function that calculates it, which accepts data streams of any length as input but always outputs a fixed-length code
- The divisor in a cyclic code is normally called the **generator polynomial** or simply the generator. The proper
  - 1. It should have at least two terms.
  - 2. The coefficient of the term x^0 should be 1.
  - 3. It should not divide $x^t + 1$, for t between 2 and $n - 1$.
  - 4. It should have the factor $x + 1$.

CRC Calculation

- Given a k-bit frame or message, the transmitter generates an n-bit sequence, known as a *frame check sequence (FCS)*, so that the resulting frame, consisting of (k+n) bits, is exactly divisible by some predetermined number.



Cyclic Redundancy Check

- Let M(x) be the **message polynomial**
- Let P(x) be the **generator polynomial**
  - P(x) is fixed for a given CRC scheme
  - P(x) is known both by sender and receiver
- Create a block polynomial F(x) based on M(x) and P(x) such that F(x) is divisible by P(x)

$$\frac{F(x)}{P(x)} = Q(x) + \frac{0}{P(x)}$$

Cyclic Redundancy Check

- Sending
  1. Multiply M(x) by x^n
  2. Divide x^nM(x) by P(x)
  3. Ignore the quotient and keep the reminder C(x)
  4. Form and send F(x) = x^nM(x)+C(x)

- Receiving
  1. Receive F’(x)
  2. Divide F’(x) by P(x)
  3. Accept if remainder is 0, reject otherwise

Example of CRC

- Consider a message 110010 represented by the polynomial $M(x) = x^5 + x^4 + x$
  Consider a *generating polynomial* $G(x) = x^3 + x^2 + 1$ (1101)
  This is used to generate a 3 bit CRC = C(x) to be appended to M(x).

Steps:

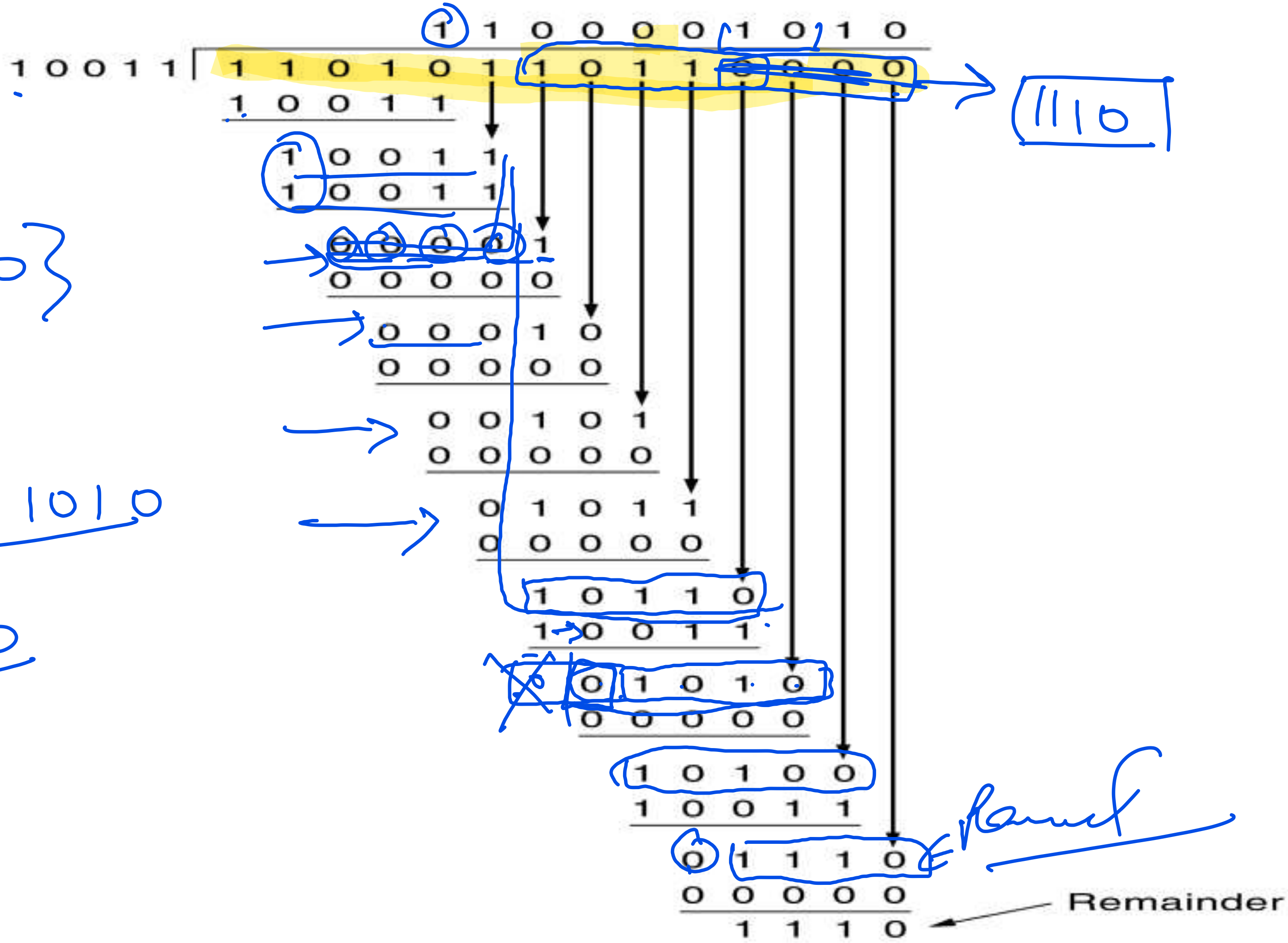
1. Multiply M(x) by x^3 (highest power in G(x)). i.e. Add 3 zeros. 110010000
2. Divide the result by G(x). The remainder = C(x).
   1101 long division into 110010000 (with subtraction mod 2)
   = 100100 remainder **100**
3. Transmit 110010000 + 100
   To be precise, transmit: $T(x) = x^3M(x) + C(x)$
   = 110010100
4. Receiver end: Receive T(x). Divide by G(x), should have remainder 0.

Note if G(x) has order n - highest power is x^n, then G(x) will cover (n+1) bits and the *remainder* will cover n bits. i.e. Add n bits to message.

Frame : 1 1 0 1 0 1 1 0 1 1

Generator: 1 0 0 1 1

Message after 4 zero bits are appended: 1 1 0 1 0 1 1 0 1 1 0 0 0 0



quotient

↓

{110000}

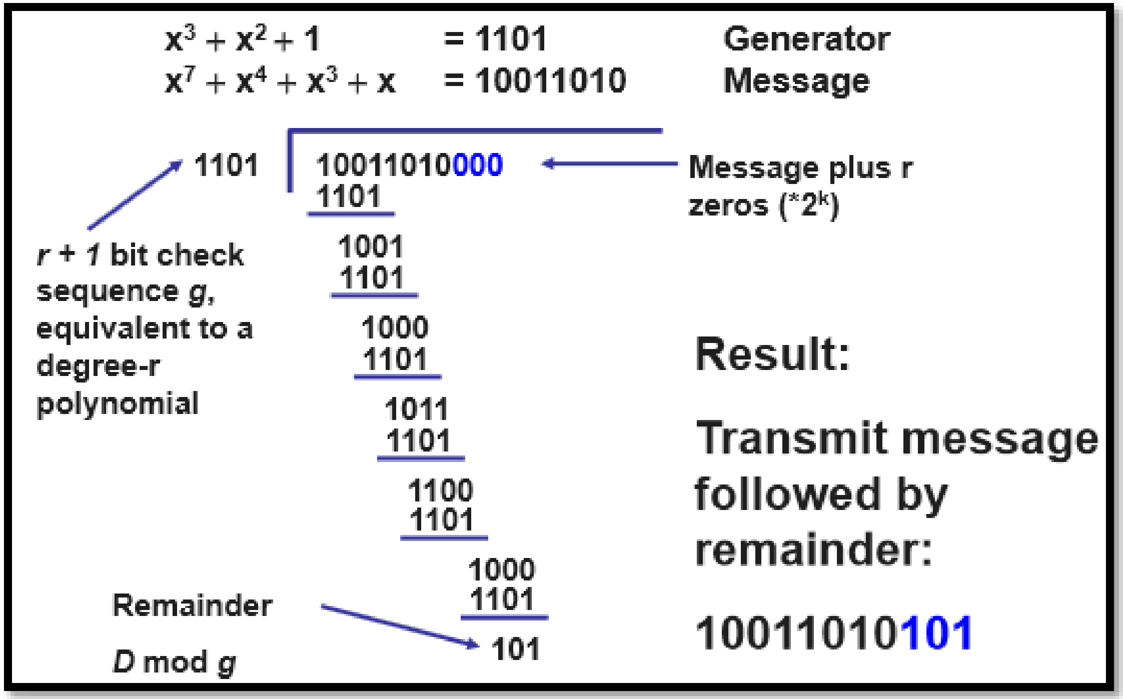
1010

q: 1100001010

R= 1110

Transmitted frame: 1 1 0 1 0 1 1 0 1 1 1 1 1 0

AT SENDER



$x^3 + x^2 + 1$ = 1101 Generator

$x^7 + x^4 + x^3 + x$ = 10011010 Message

```
1101 | 10011010000
       1101
        1001
        1101
         1000
         1101
          1011
          1101
           1100
           1101
            1000
            1101
             101
```

Message plus r zeros $(*2^k)$

r + 1 bit check sequence *g*, equivalent to a degree-r polynomial

Remainder

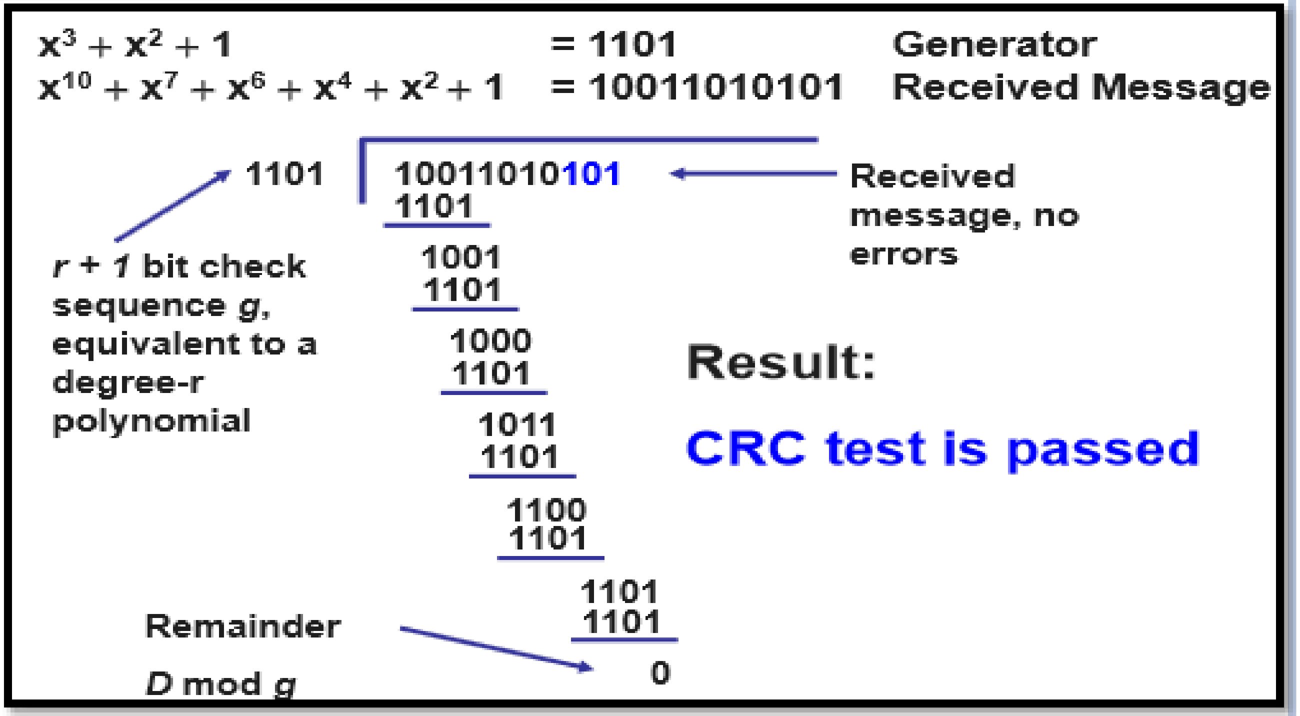
D mod *g*

Result:

Transmit message followed by remainder:

10011010101

AT RECEIVER



$x^3 + x^2 + 1$ = 1101 Generator

$x^{10} + x^7 + x^6 + x^4 + x^2 + 1$ = 10011010101 Received Message

```
1101 | 10011010101
       1101
        1001
        1101
         1000
         1101
          1011
          1101
           1100
           1101
            1101
            1101
               0
```

Received message, no errors

r + 1 bit check sequence *g*, equivalent to a degree-r polynomial

Remainder

D mod *g*

Result:

CRC test is passed

Another Example of CRC

- **Example: the polynomial R(X) (the appended bits)**

Message $[11100110]$ 8 bits

$$\mathbf{m}(X) = X^7 + X^6 + X^5 + X^2 + X$$

Given N-k=n=4, generator polynomial $\mathbf{g}(X) = X^4 + X^3 + 1 \rightarrow [11001]$

$$\frac{X^n\mathbf{m}(X)}{\mathbf{g}(X)} = \frac{X^{11} + X^{10} + X^9 + X^6 + X^5}{X^4 + X^3 + 1}$$

$$= X^7 + X^5 + X^4 + X^2 + X + \frac{X^2 + X}{X^4 + X^3 + 1}$$

$X^n m(X)$ is the polynomial corresponding to the message bit sequence to which a number n of 0's is appended. [111001100000]

The remainder **R**(X)=$X^2 + X$,

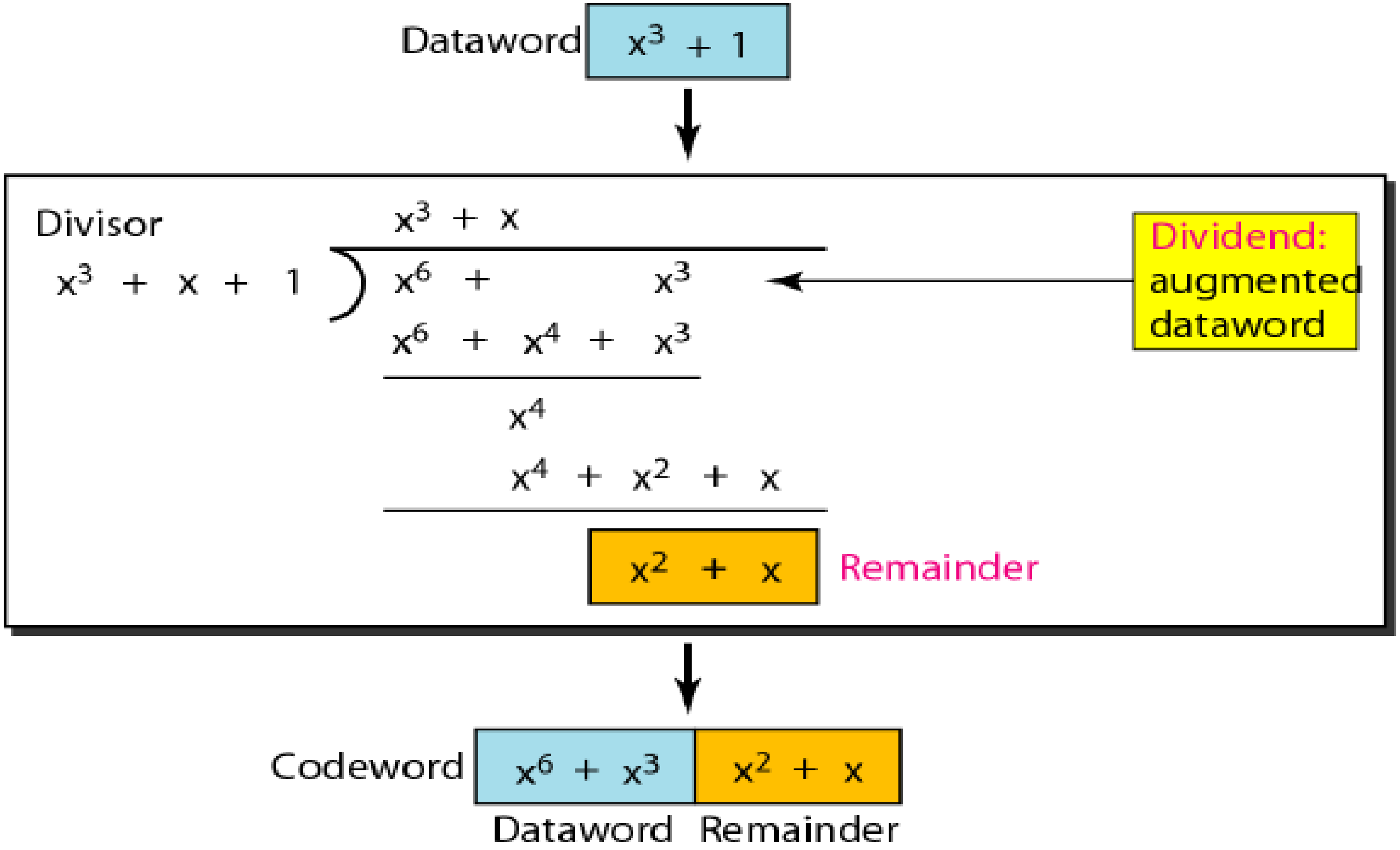
therefore the appended bits are $[0110]$ (since n=4).

The CRC code bits are $[111001100110]$

CRC Division in Polynomial Form

Example of CRC (7, 4) in Vector Form (4)

Division in the sender (Polynomial form):

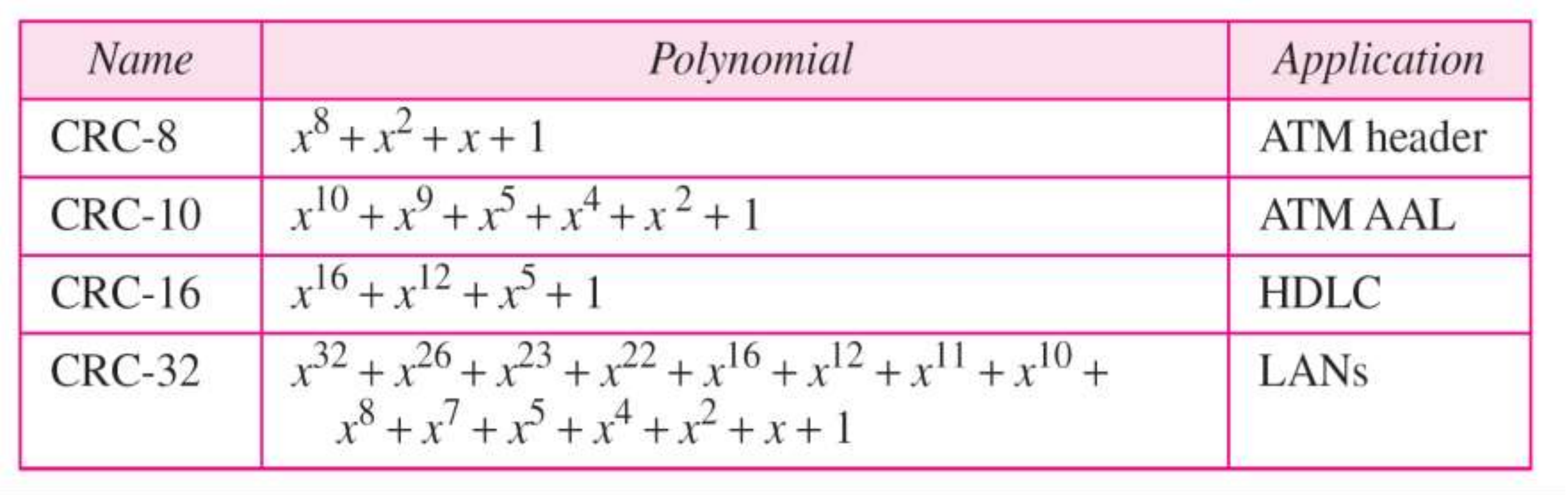


Example of CRC (7, 4) in Vector Form (1)

Generator polynomial $\mathbf{g}(X) = X^3 + X + 1 \rightarrow [1011]$

Dataword	*Codeword*	*Dataword*	*Codeword*
0000	0000000	1000	1000101
0001	0001011	1001	1001110
0010	0010110	1010	1010011
0011	0011101	1011	1011000
0100	0100111	1100	1100010
0101	0101100	1101	1101001
0110	0110001	1110	1110100
0111	0111010	1111	1111111

CRC Standard Polynomials



Name	*Polynomial*	*Application*
CRC-8	$x^8 + x^2 + x + 1$	ATM header
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$	ATM AAL
CRC-16	$x^{16} + x^{12} + x^5 + 1$	HDLC
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} +$ $x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	LANs

CRC Performance

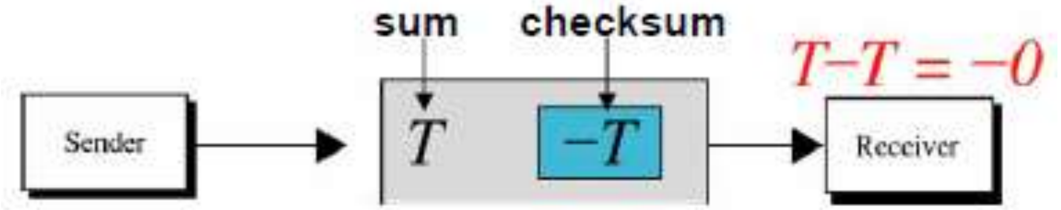
CRC is a very effective error detection technique. If the divisor is chosen according to the previously mentioned rules, its performance can be summarized as follows:

❖CRC can detect all single-bit errors
❖CRC can detect all double-bit errors (three 1’s)
❖CRC can detect any odd number of errors (X+1)
❖CRC can detect all burst errors of less than the degree of the polynomial.
❖CRC detects most of the larger burst errors with a high probability.

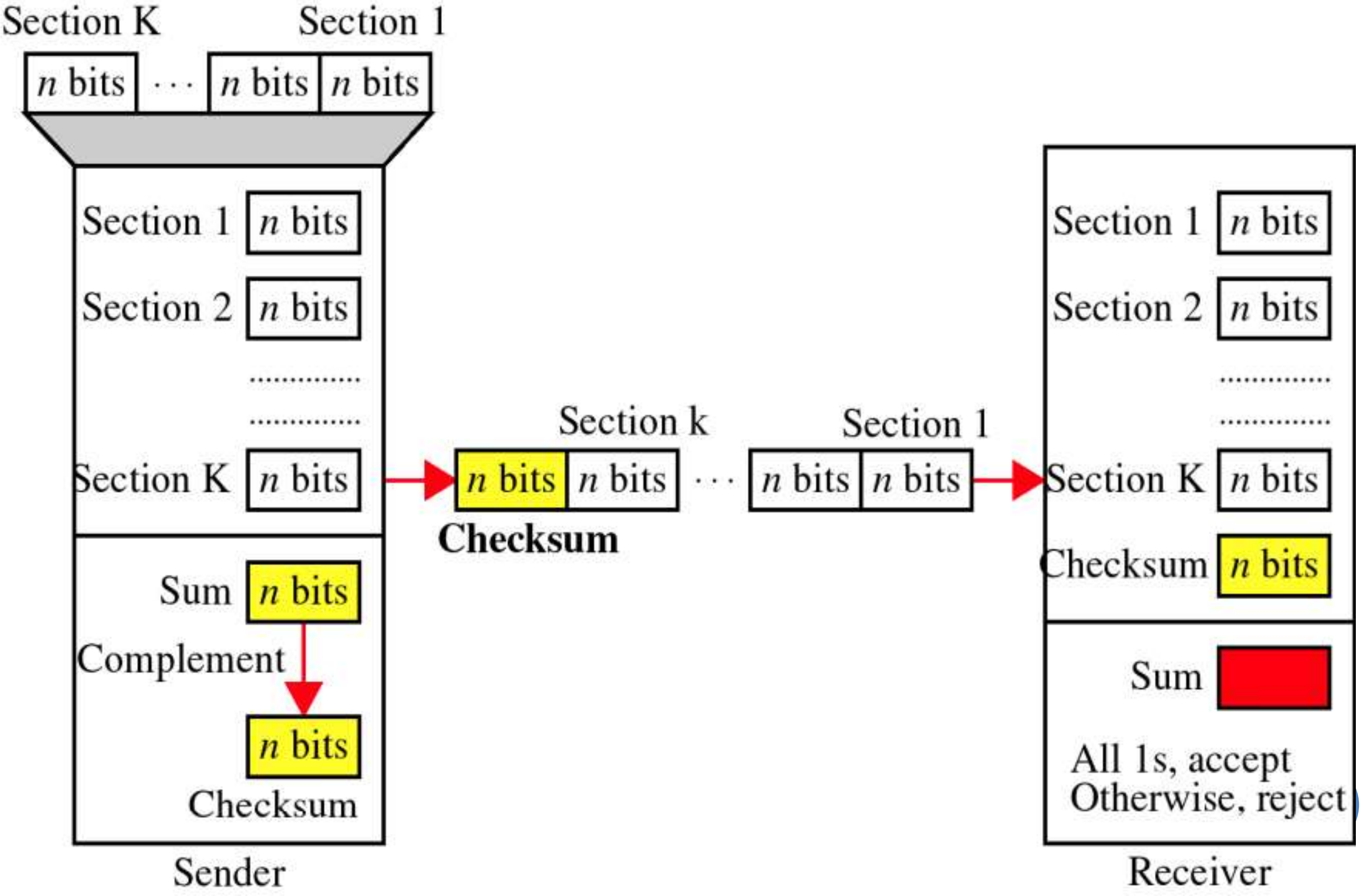
- For example CRC-12 detects 99.97% of errors with a length 12 or more.

Checksum

- Checksum is the error detection scheme used in IP, TCP & UDP.
- Here, 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
- The checksum detects all errors involving an odd number of bits. It also detects most errors involving even number of bits.



Checksum


Section K
Section 1
n bits
· · ·
n bits
n bits
Section 1
n bits
Section 2
n bits
....
....
Section K
n bits
Sum
n bits
Complement
n bits
Checksum
Sender
Section k
Section 1
n bits
n bits
· · ·
n bits
n bits
Checksum
Section 1
n bits
Section 2
n bits
....
....
Section K
n bits
Checksum
n bits
Sum
All 1s, accept
Otherwise, reject
Receiver

CHECKSUM EXAMPLE

Sender:

- data is divided into k sections each n bits long
- all sections are added using 1-s complement to get the sum
- the sum is bit-wise complemented and becomes the checksum
- the checksum is sent with the data

Receiver:

- data is divided into k sections each n bits long
- all sections are added using 1-s complement to get the sum
- the sum is bit-wise complemented
- if the result is zero, the data is accepted, otherwise it is rejected

Example [Internet Checksum]

Suppose the following block of 8 bits is to be sent using a checksum of 4 bits: 1100 1010. Find the checksum of the given bit sequence.

1100
1010
0000

sum: $\boxed{1}0110$

0110
1

1-s complement addition: 0111 (7)

1-s complement addition:
Perform standard binary addition.
If a carry-out (>nth) bit it produced, swing that bits around and add it back into the summation.

checksum: 1000 (-7)

Negative binary numbers:
Negative binary numbers are bit-wise complement of corresponding positive numbers.

Suppose the receiver receives the bit sequence and the checksum with no error.

1100
1010
1000

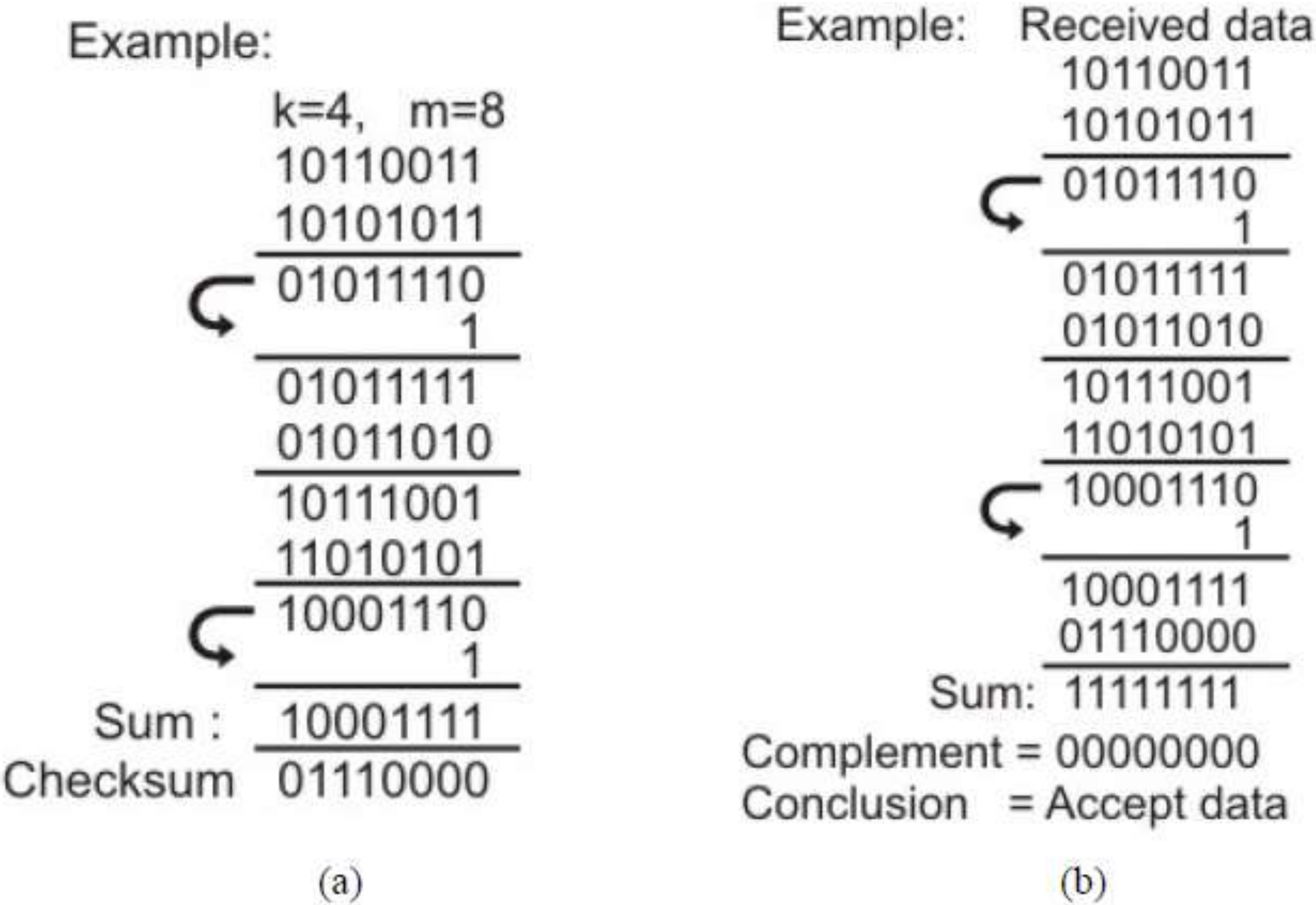
sum: $\boxed{1}1110$
1-s complement addition: 1111
bit-wise complement: 0000

When the receiver adds the three blocks, it will get all 1s, which, after complementing, is all 0s and shows that there is no error.

If one or more bits of a segment are damaged, <u>and the corresponding bit of opposite value in a second segment is also damaged,</u>
the sums of those columns will not change and the receiver will not detect the problem. ☹

Q) For a pattern of, **10101001 00111001 00011101** Find out whether any transmission errors have occurred or not

EXAMPLE OF CHECKSUM



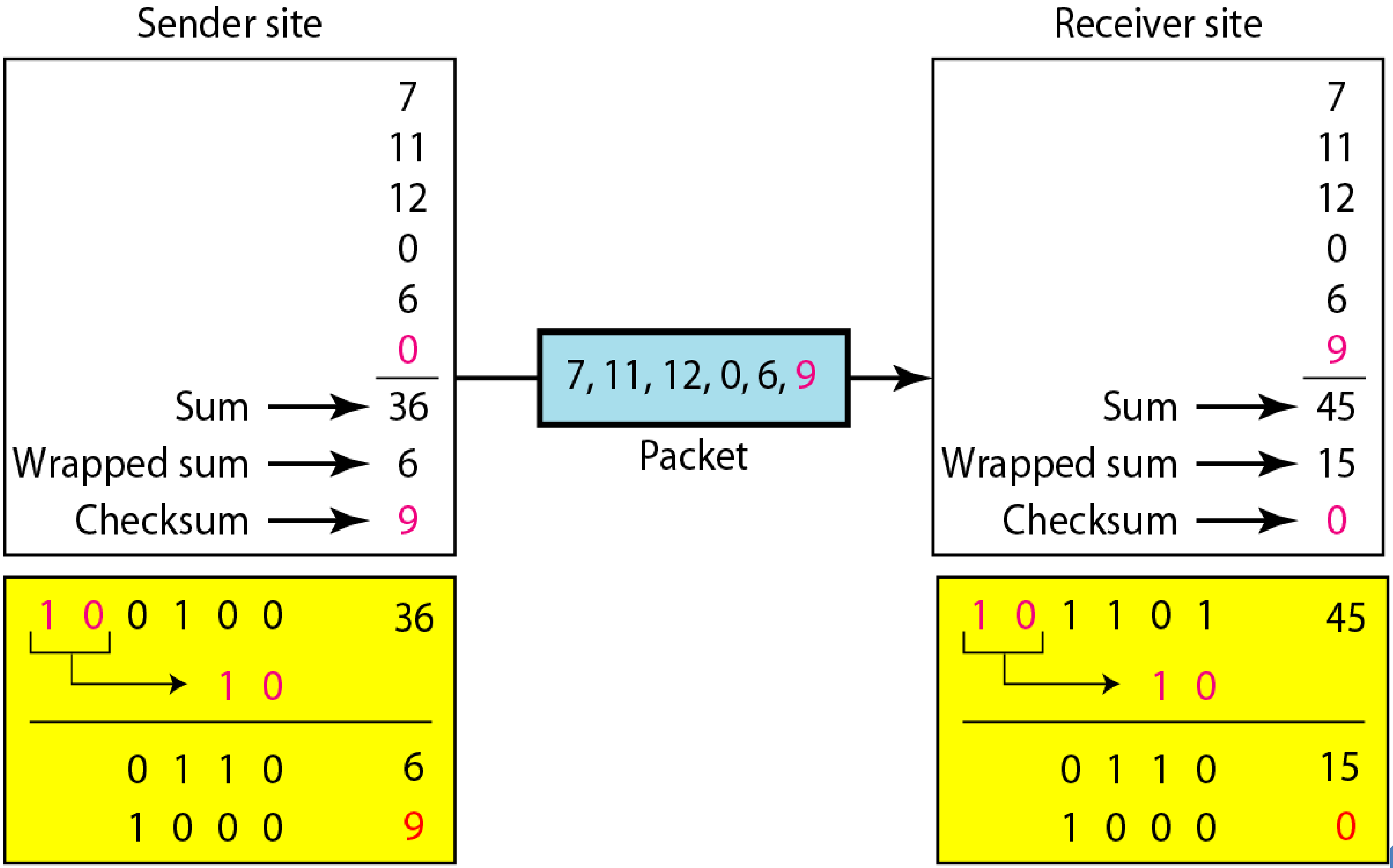
```
Example:
              k=4,   m=8
              10110011
              10101011
              --------
           ↩  01011110
                     1
              --------
              01011111
              01011010
              --------
              10111001
              11010101
              --------
           ↩  10001110
                     1
              --------
    Sum :     10001111
              --------
Checksum      01110000

                (a)
```

```
Example:   Received data
            10110011
            10101011
            --------
         ↩  01011110
                   1
            --------
            01011111
            01011010
            --------
            10111001
            11010101
            --------
         ↩  10001110
                   1
            --------
            10001111
            01110000
            --------
      Sum:  11111111
Complement = 00000000
Conclusion   = Accept data

                (b)
```

(a) Sender's end for the calculation of the checksum. (b) Receiving end for checking the checksum



Sender site

7
11
12
0
6
0

Sum → 36
Wrapped sum → 6
Checksum → 9

7, 11, 12, 0, 6, 9

Packet

Receiver site

7
11
12
0
6
9

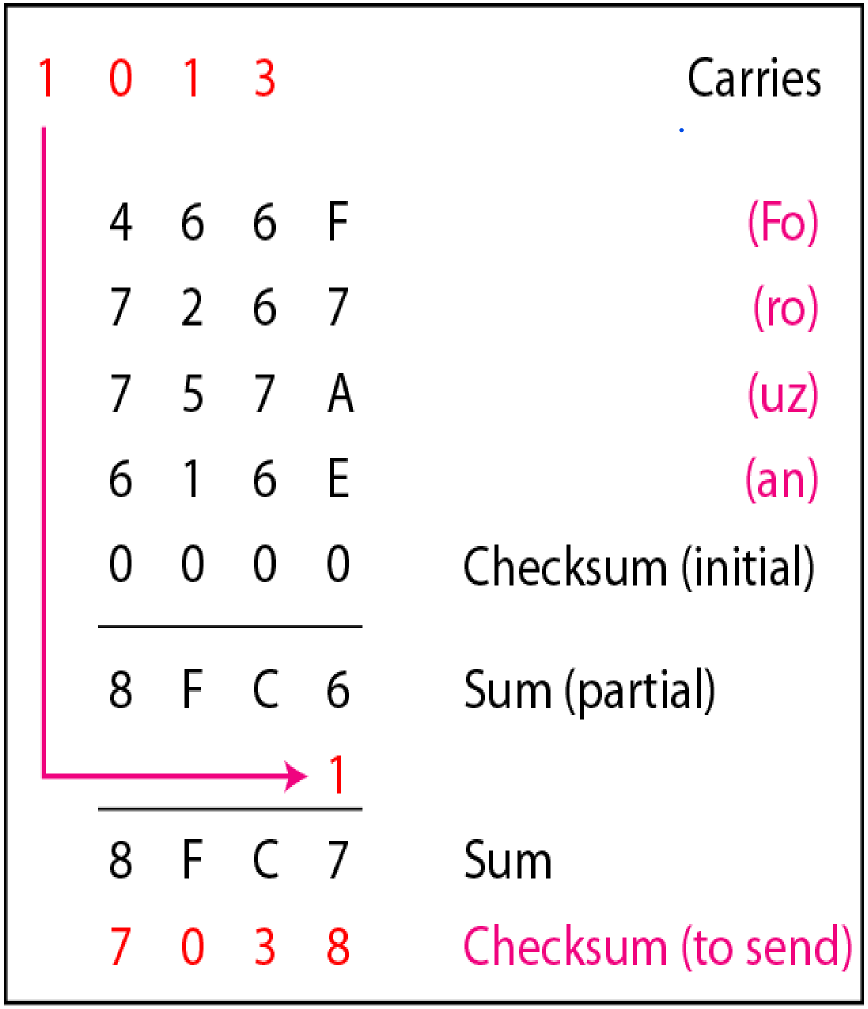
Sum → 45
Wrapped sum → 15
Checksum → 0

1 0 0 1 0 0 36
1 0
0 1 1 0 6
1 0 0 0 9

Details of wrapping and complementing

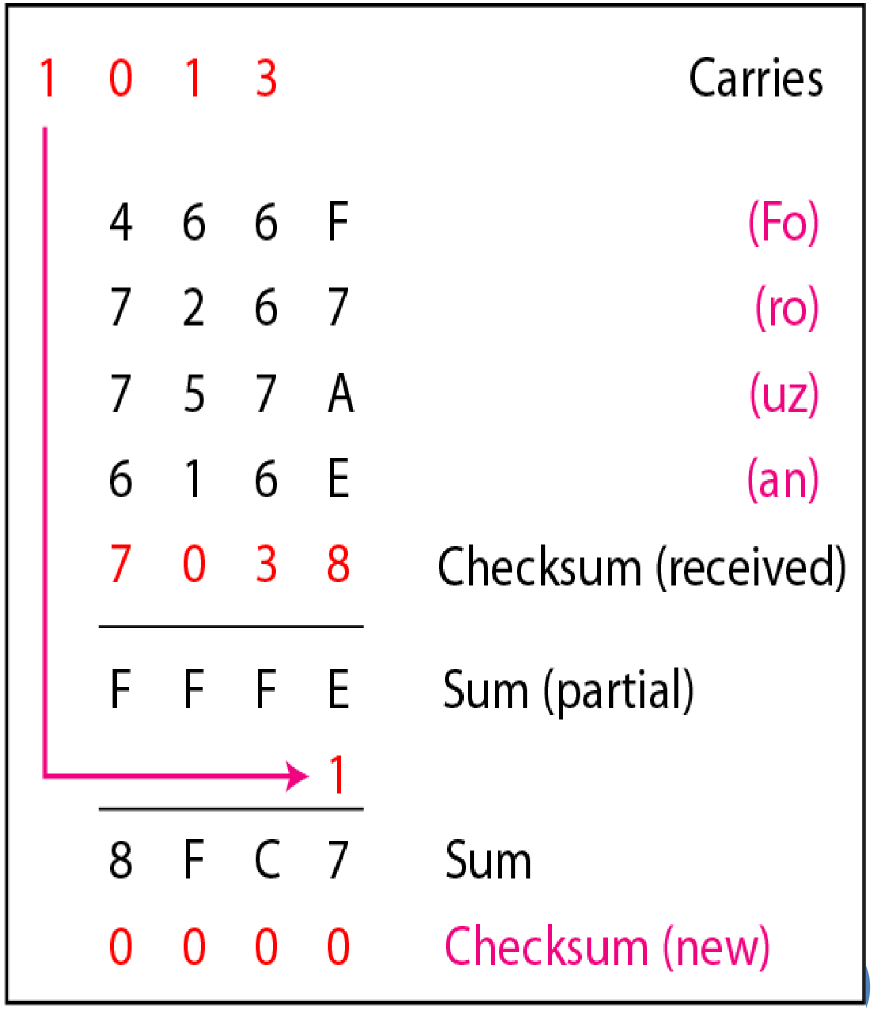
1 0 1 1 0 1 45
1 0
0 1 1 0 15
1 0 0 0 0

Details of wrapping and complementing



1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	7	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
0	0	0	0	Checksum (initial)
8	F	C	6	Sum (partial)
			1	
8	F	C	7	Sum
7	0	3	8	Checksum (to send)

a. Checksum at the sender site



1	0	1	3	Carries
4	6	6	F	(Fo)
7	2	6	7	(ro)
7	5	7	A	(uz)
6	1	6	E	(an)
7	0	3	8	Checksum (received)
F	F	F	E	Sum (partial)
			1	
8	F	C	7	Sum
0	0	0	0	Checksum (new)

a. Checksum at the receiver site

Checksum Vs CRC

- CRC is more thorough as opposed to Checksum in checking for errors and reporting.
- Checksum is the older of the two programs.
- CRC has a more complex computation as opposed to checksum.
- Checksum mainly detects single-bit changes in data while CRC can check and detect double-digit errors.
- CRC can detect more errors than checksum due to its more complex function.
- A checksum is mainly employed in data validation when implementing software.
- A CRC is mainly used for data evaluation in analogue data transmission.

ERROR CORRECTION

- Once detected, the errors must be corrected

- Two Techniques for error correction
  - **Retransmission (aka Backward error correction)**
    - Simplest, effective and most commonly used technique – involves correction by retransmission of data by the sender
    - Popularly called Automatic Repeat Request (ARQ)

  - **Forward Error Correction (FEC)**
    - Receiving device can correct the errors itself

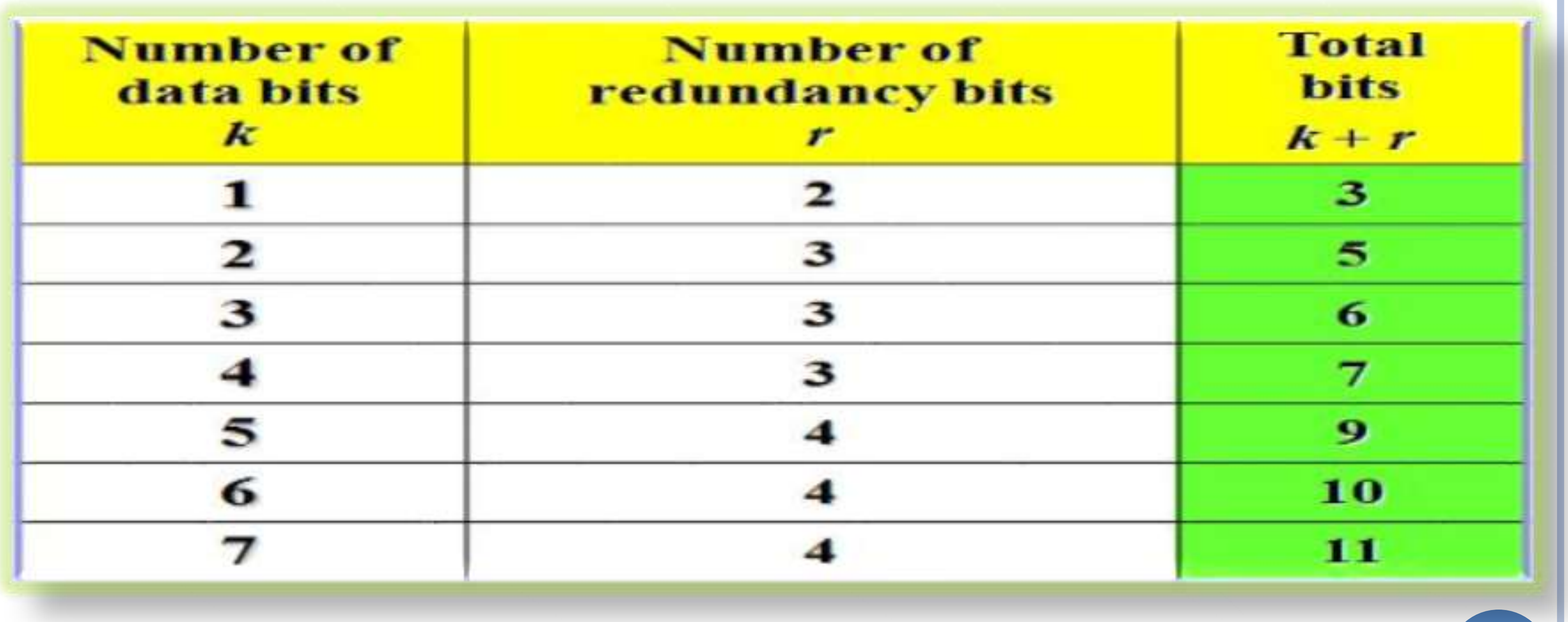
Error Correction

- Messages (frames) consist of *m* data (message) bits and *r* redundancy bits, yielding an *n* = (*m*+*r*)-bit *codeword*.
- *Hamming Distance*. Given any two codewords, we can determine how many of the bits differ. Simply exclusive or (XOR) the two words, and count the number of 1 bits in the result.
- Significance? If two codewords are *d* bits apart, *d* errors are required to convert one to the other.
- A code's *Hamming Distance* is defined as the ***minimum Hamming Distance*** between any two of its legal codewords (from all possible codewords).
- To detect *d* 1-bit errors requires having a Hamming Distance of at least ***d*+1 bits**.
- To correct *d* errors requires **2*d*+1 bits**. Intuitively, after *d* errors, the garbled messages is still closer to the original message than any other legal codeword.

Hamming Code

- Ex : If The Value of m is 7, the Relation will Satisfy if The Minimum Value of r is 4.

2^4 = 16 > 7+4+1

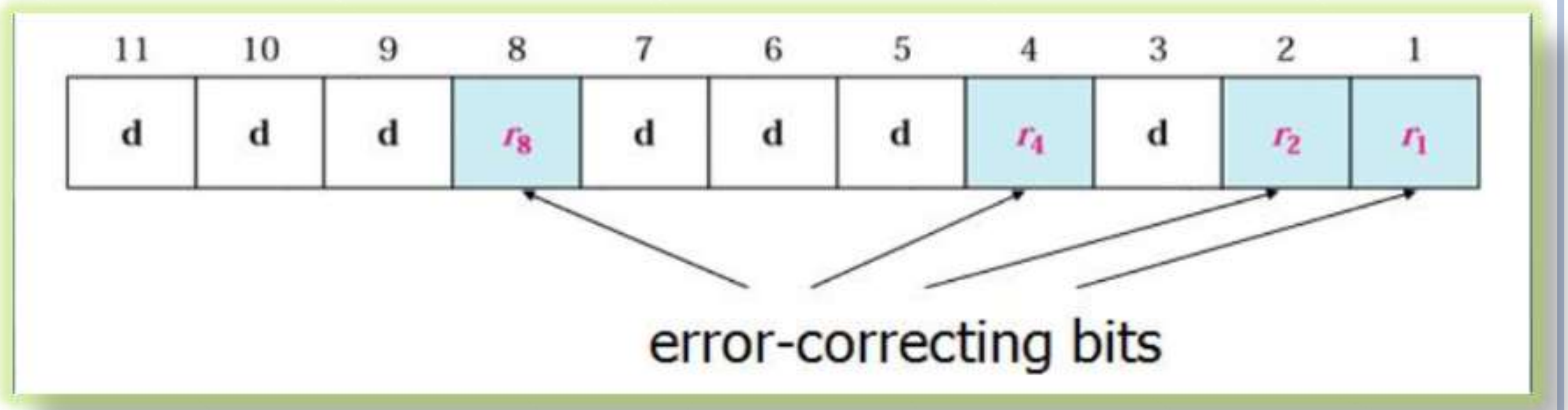


Number of data bits *k*	Number of redundancy bits *r*	Total bits *k* + *r*
1	2	3
2	3	5
3	3	6
4	3	7
5	4	9
6	4	10
7	4	11

Hamming Code Example

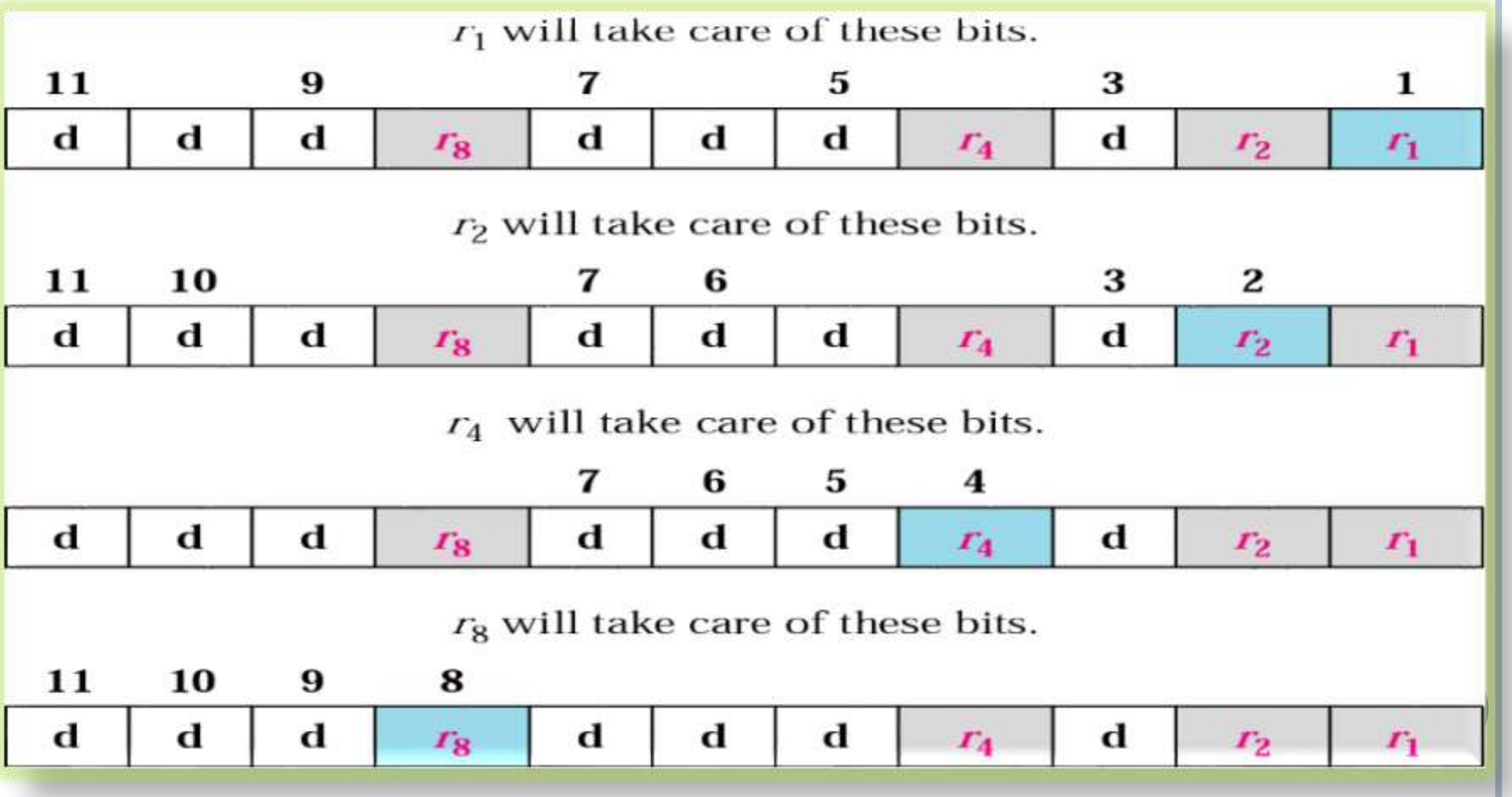
If The Number of Data bit is 7, Then The Position of Redundant bits Are :

2^0=1 2^1=2
2^2=4 2^3=8



11	10	9	8	7	6	5	4	3	2	1
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

Hamming Code



r_1 will take care of these bits.

11		9		7		5		3		1
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

r_2 will take care of these bits.

11	10			7	6			3	2	
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

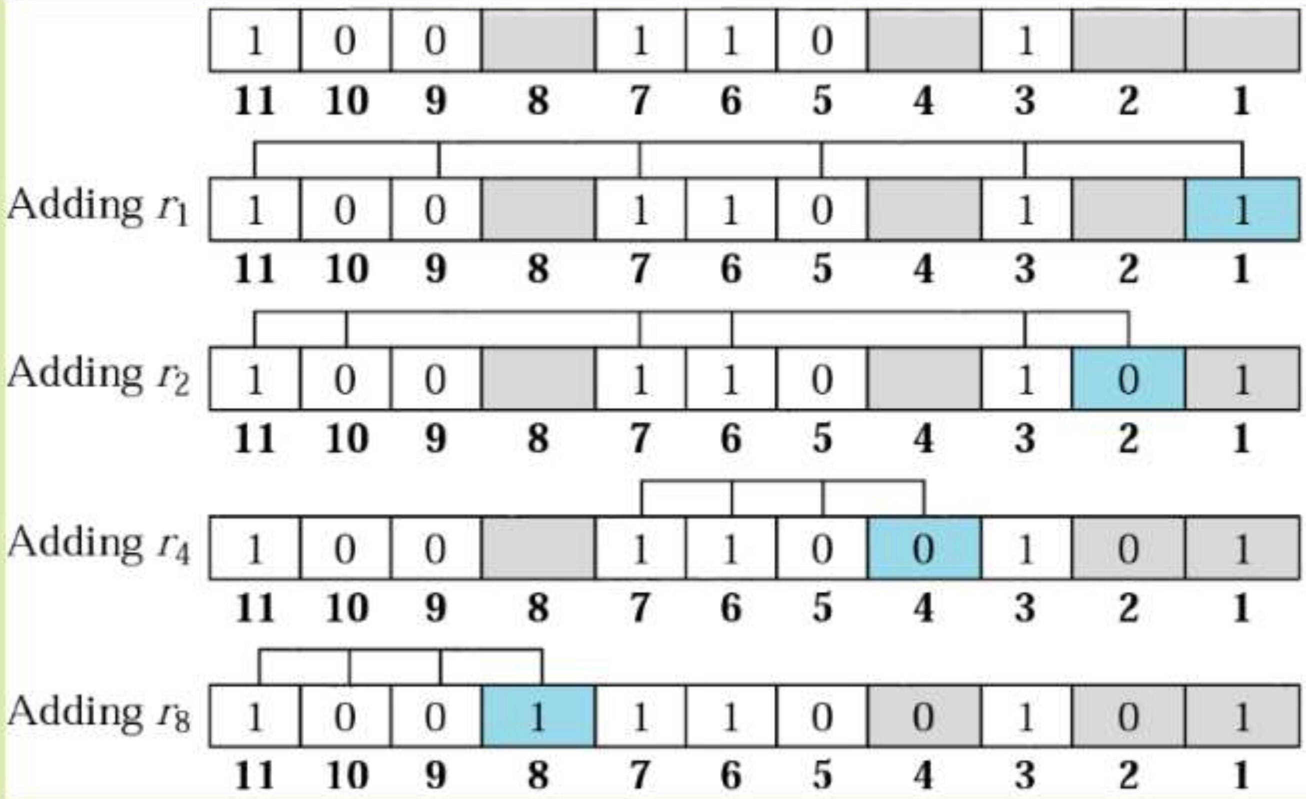
r_4 will take care of these bits.

				7	6	5	4			
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

r_8 will take care of these bits.

11	10	9	8							
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

Hamming Code



Data:
1 0 0 1 1 0 1

11	10	9	8	7	6	5	4	3	2	1
1	0	0		1	1	0		1		

Adding r_1

11	10	9	8	7	6	5	4	3	2	1
1	0	0		1	1	0		1		1

Adding r_2

11	10	9	8	7	6	5	4	3	2	1
1	0	0		1	1	0		1	0	1

Adding r_4

11	10	9	8	7	6	5	4	3	2	1
1	0	0		1	1	0	0	1	0	1

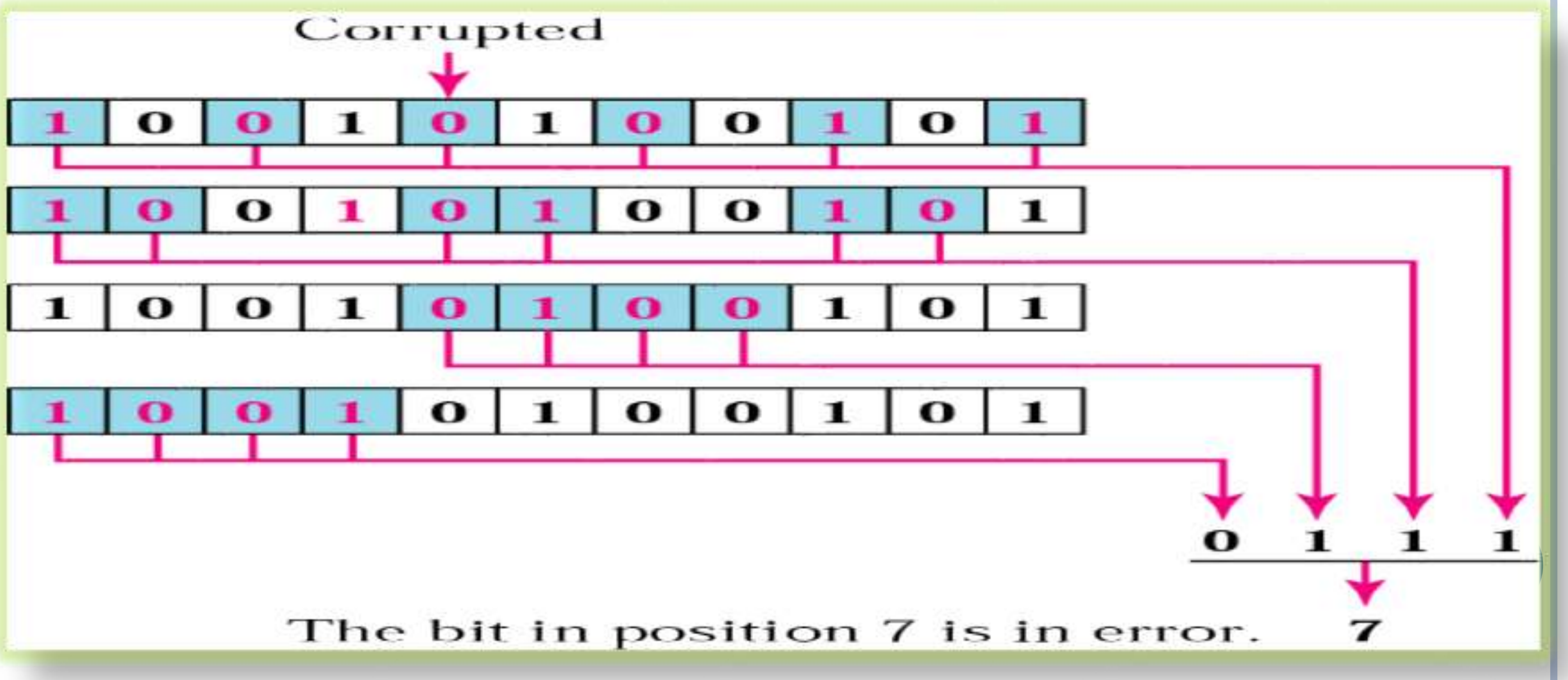
Adding r_8

11	10	9	8	7	6	5	4	3	2	1
1	0	0	1	1	1	0	0	1	0	1

Code:
1 0 0 1 1 1 0 0 1 0 1

Hamming Code

Let Receiver receives 10010100101



1 0 0 1 0 1 0 0 1 0 1

1 0 0 1 0 1 0 0 1 0 1

1 0 0 1 0 1 0 0 1 0 1

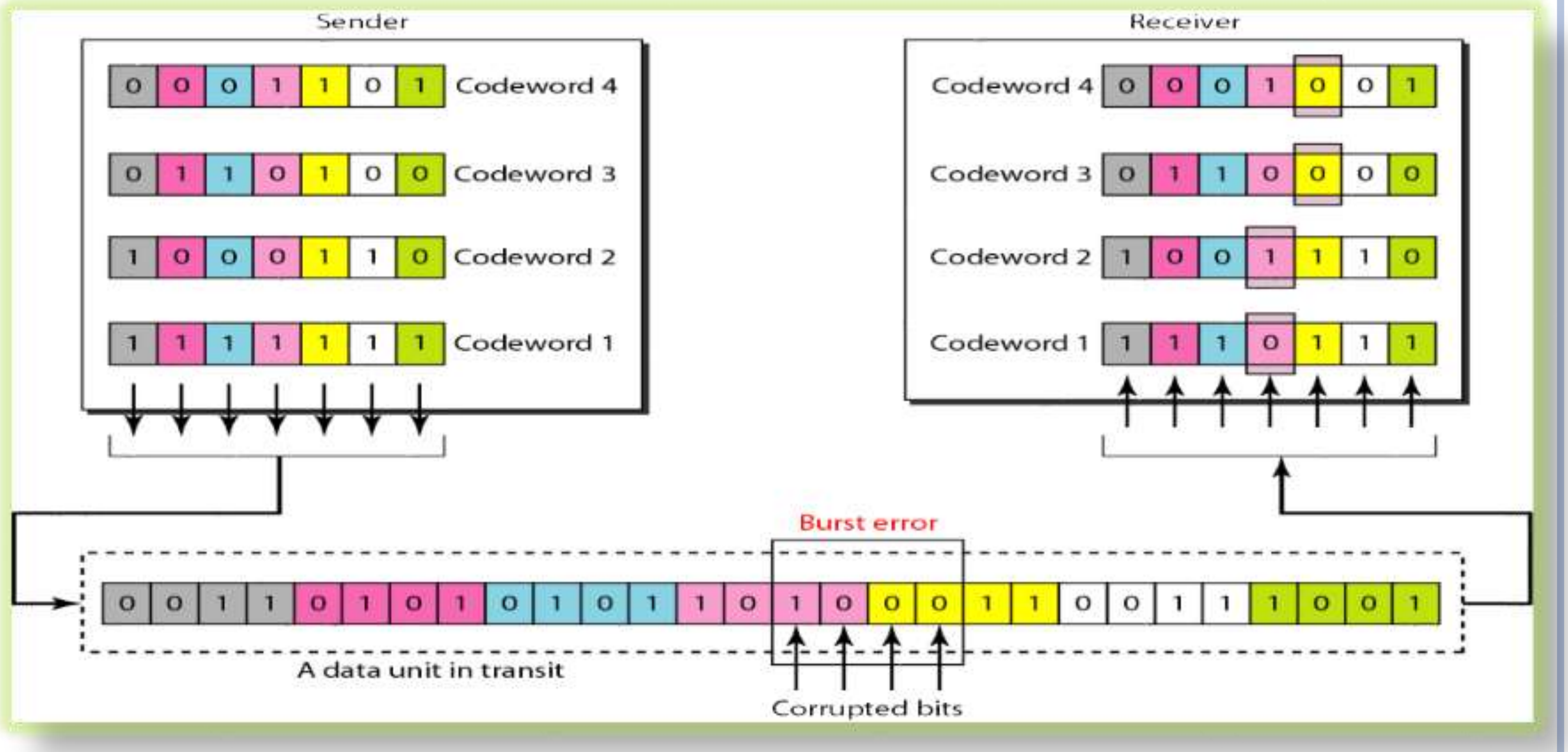
1 0 0 1 0 1 0 0 1 0 1

The bit in position 7 is in error.

Burst Error Correction

- Hamming Code Cannot Correct a burst Error Directly.
- it is Possible To Rearrange The Data and Then Apply The code.
- Instead of Sending All the bits in The data Unit Together, we can organize N units in a column.
- Send The First bits of Each Followed by The Second bit of each, and so on.
- In This Way, if a burst Error of M bit Occurs (M<N), Then The Error does not Corrupt M bit of Single Unit, it Corrupt Only 1 bit of Unit.
- Then We Can Correct it Using Hamming Code Scheme.

Hamming Code



Sender

0 0 0 1 1 0 1 Codeword 4

0 1 1 0 1 0 0 Codeword 3

1 0 0 0 1 1 0 Codeword 2

1 1 1 1 1 1 1 Codeword 1

Receiver

Codeword 4 0 0 0 1 0 0 1

Codeword 3 0 1 1 0 0 0 0

Codeword 2 1 0 0 1 1 1 0

Codeword 1 1 1 1 0 1 1 1

0 0 1 1 0 1 0 1 0 1 0 1 1 0 1 0 0 0 1 1 0 0 1 1 1 0 0 1

FUNCTIONS AND REQUIREMENTS OF THE DATA LINK PROTOCOLS

The basic function of the layer is to transmit frames over a physical communication link. Transmission may be *half duplex* or *full duplex*. To ensure that frames are delivered free of errors to the destination station (IMP) a number of requirements are placed on a data link protocol. The protocol (control mechanism) should be capable of performing:

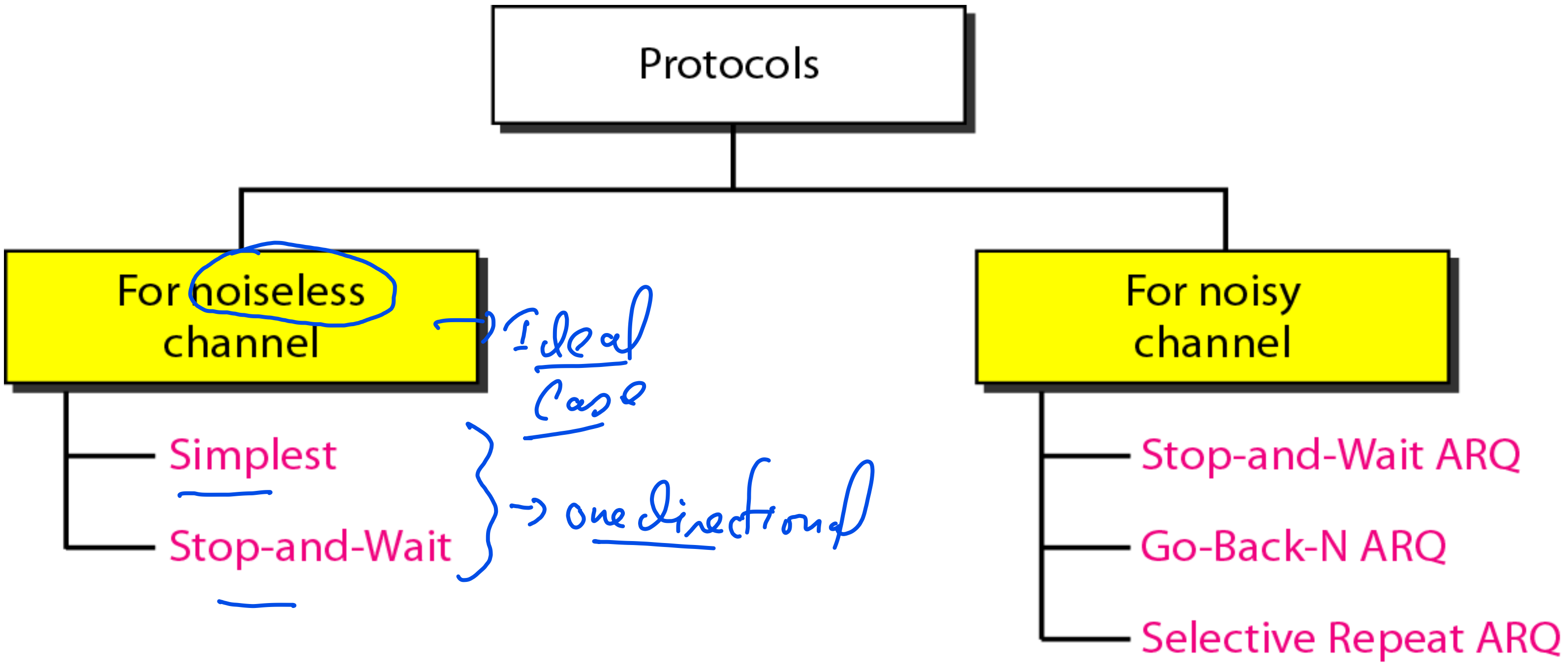
- The identification of a frame (i.e. recognise the first and last bits of a frame).
- The transmission of frames of any length up to a given maximum. Any bit pattern is permitted in a frame.
- The detection of transmission errors.
- The retransmission of frames which were damaged by errors.
- The assurance that no frames were lost.
- In a multidrop configuration -> Some mechanism must be used for preventing conflicts caused by simultaneous transmission by many stations.
- The detection of failure or abnormal situations for control and monitoring purposes.

It should be noted that as far as layer 2 is concerned a host message is pure data, every single bit of which is to be delivered to the other host. The frame header pertains to layer 2 and is never given to the host.

Elementary Data Link Protocols

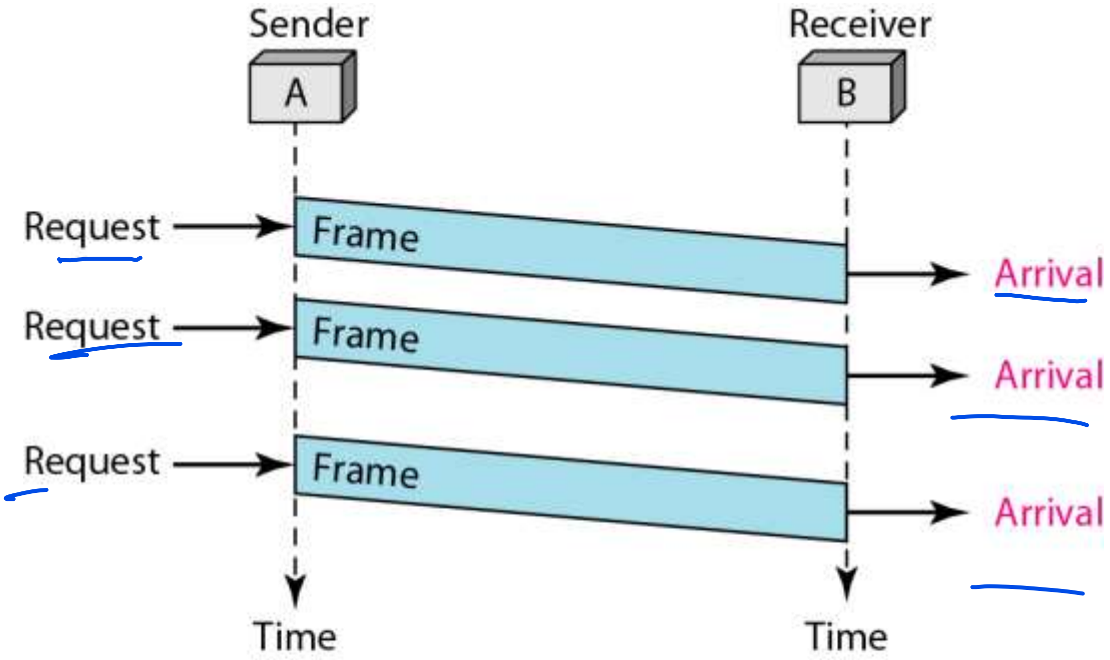
The protocols are normally implemented in software by using one of the common programming languages.

- An Unrestricted Simplex Protocol
- A Simplex Stop-and-Wait Protocol
- A Simplex Protocol for a Noisy Channel



An unrestricted simplex protocol

- In order to appreciate the step by step development of efficient and complex protocols we will begin with a simple but unrealistic protocol. In this protocol: Data are transmitted in one direction only
- The transmitting (Tx) and receiving (Rx) hosts are always ready
- Processing time can be ignored
- Infinite buffer space is available
- No errors occur; i.e. no damaged frames and no lost frames (perfect channel)



A SIMPLEX STOP-AND-WAIT PROTOCOL

- In this protocol we assume that Data are transmitted in one direction only
- No errors occur (perfect channel)
- The receiver can only process the received information at a finite rate
- These assumptions imply that the transmitter cannot send frames at a rate faster than the receiver can process them.

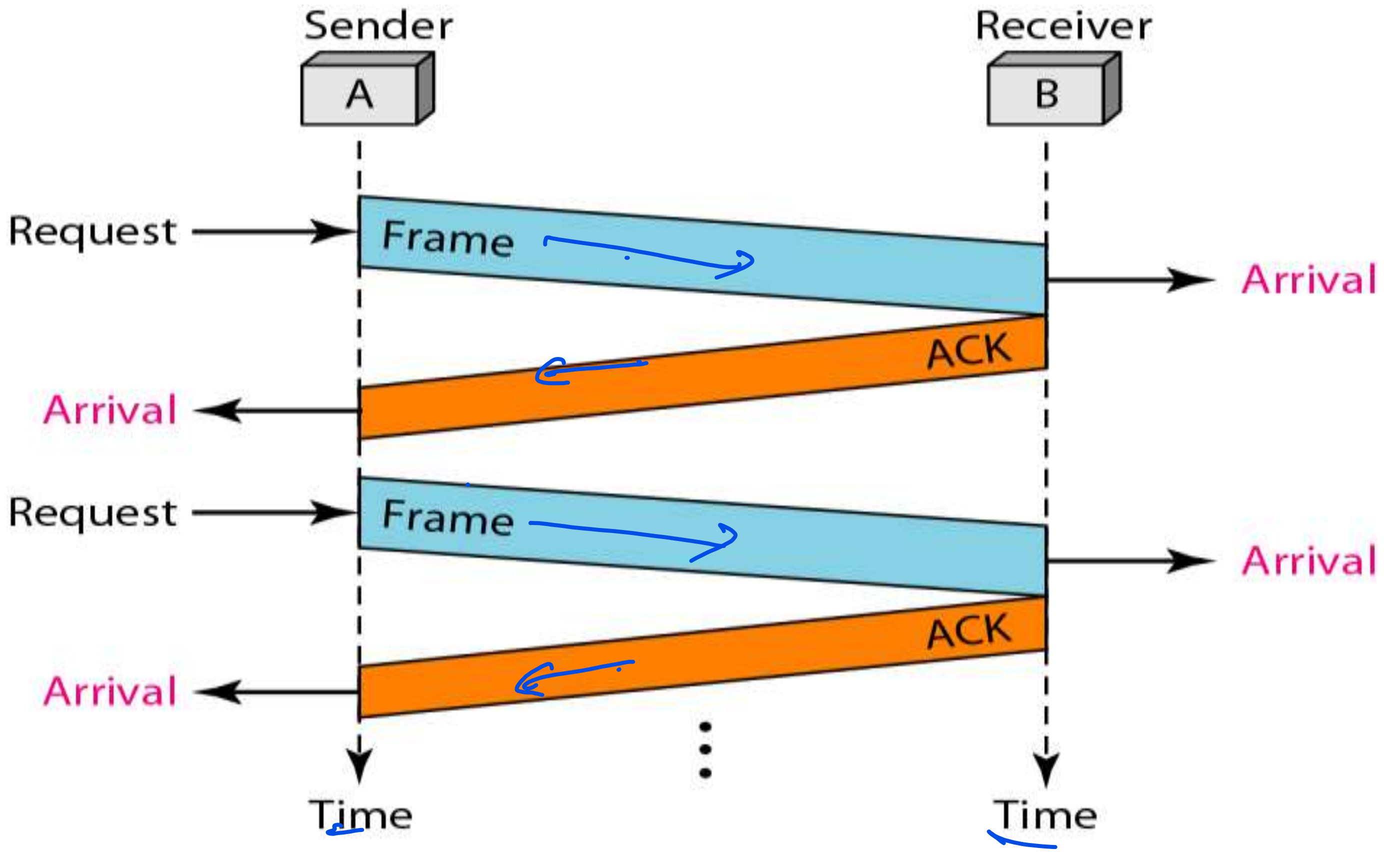
The problem here is how to prevent the sender from flooding the receiver.

- A general solution to this problem is to have the receiver provide some sort of feedback to the sender. The process could be as follows: The receiver send an acknowledge frame back to the sender telling the sender that the last received frame has been processed and passed to the host; permission to send the next frame is granted. The sender, after having sent a frame, must wait for the acknowledge frame from the receiver before sending another frame.

This protocol is known as *stop-and-wait.*

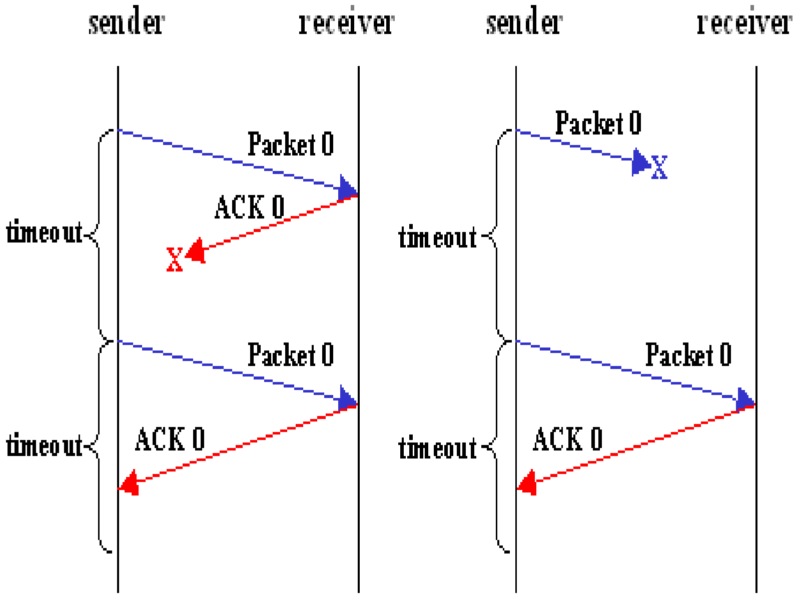
STOP & WAIT PROTOCOL

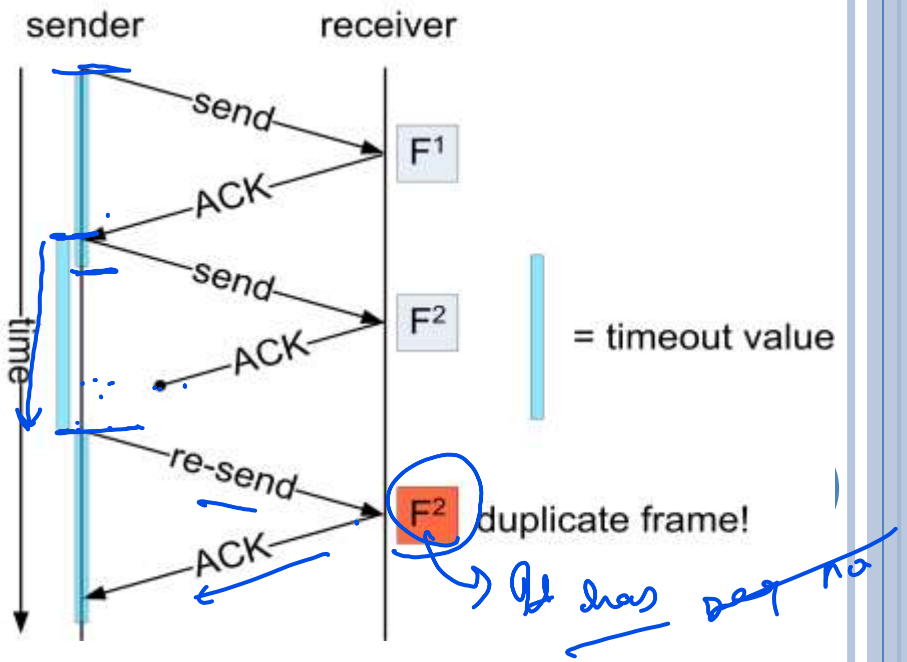
The sender sends one frame and waits for feedback from the receiver. When the ACK arrives, the sender sends the next frame



A SIMPLEX PROTOCOL FOR A NOISY CHANNEL

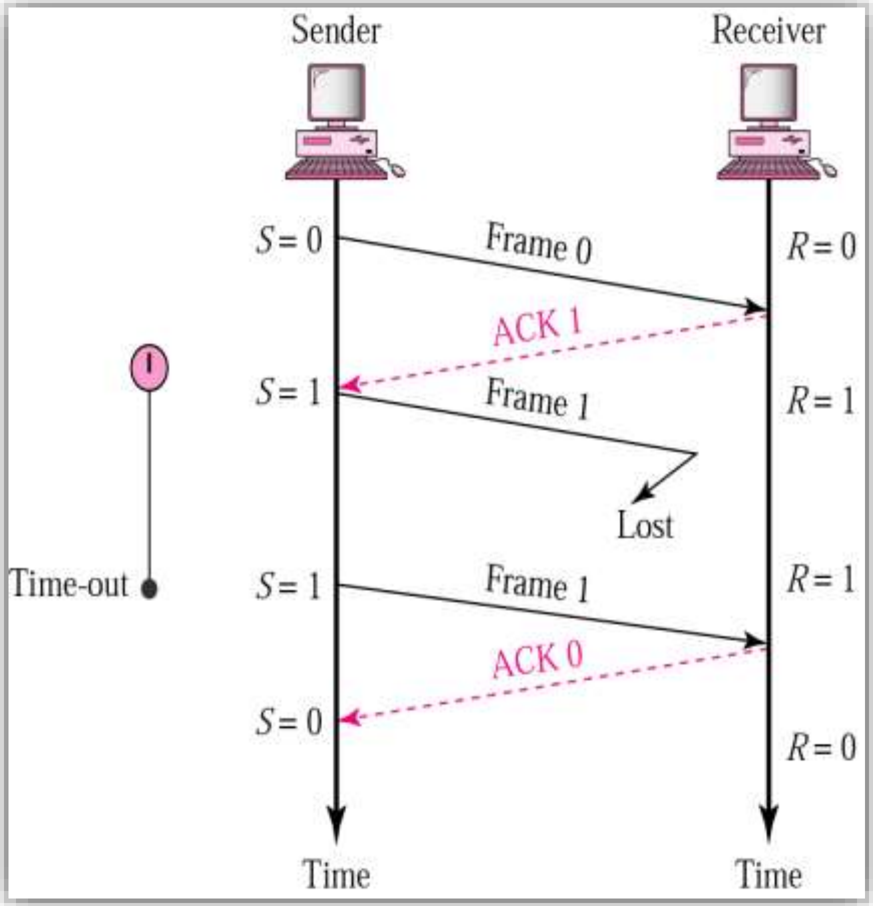
- In this protocol the unreal "error free" assumption in protocol 2 is dropped. Frames may be either damaged or lost completely. We assume that transmission errors in the frame are detected by the hardware checksum. One suggestion is that the sender would send a frame, the receiver would send an ACK frame only if the frame is received correctly. If the frame is in error the receiver simply ignores it; the transmitter would time out and would retransmit it.
- One fatal flaw with the above scheme is that if the ACK frame is lost or damaged, duplicate frames are accepted at the receiver without the receiver knowing it.



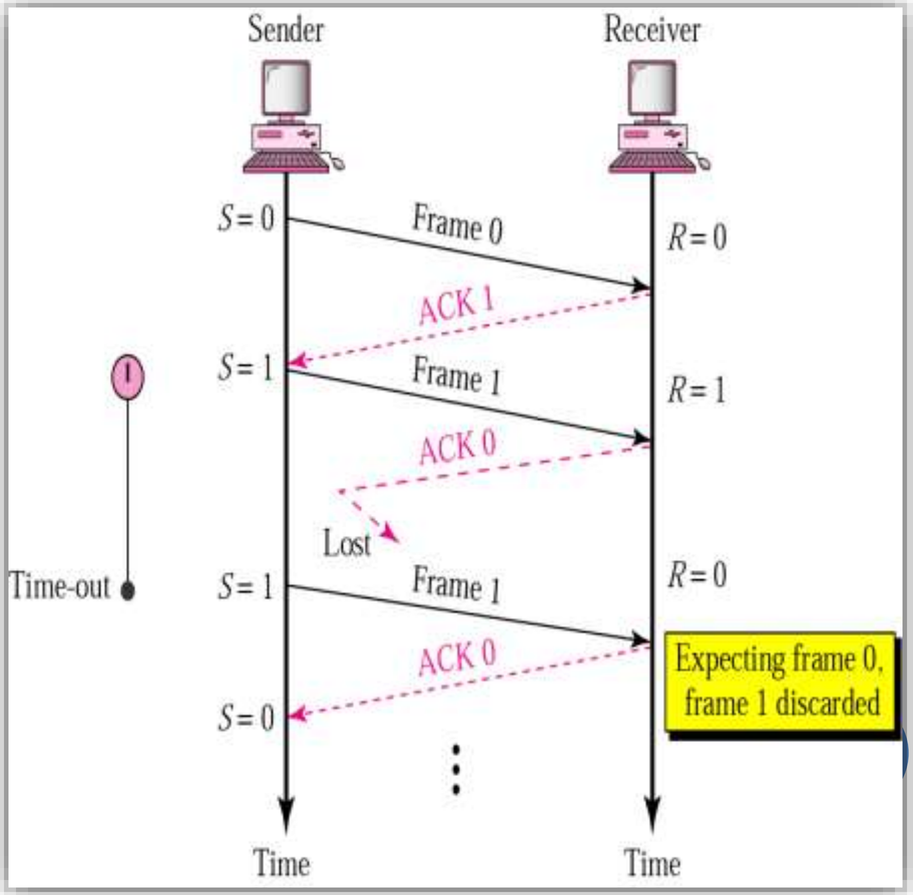


- Imagine a situation where the receiver has just sent an ACK frame back to the sender saying that it correctly received and already passed a frame to its host. However, the ACK frame gets lost completely, the sender times out and retransmits the frame. There is no way for the receiver to tell whether this frame is a retransmitted frame or a new frame, so the receiver accepts this duplicate happily and transfers it to the host. The protocol thus fails in this aspect.

STOP-AND-WAIT, LOST FRAME



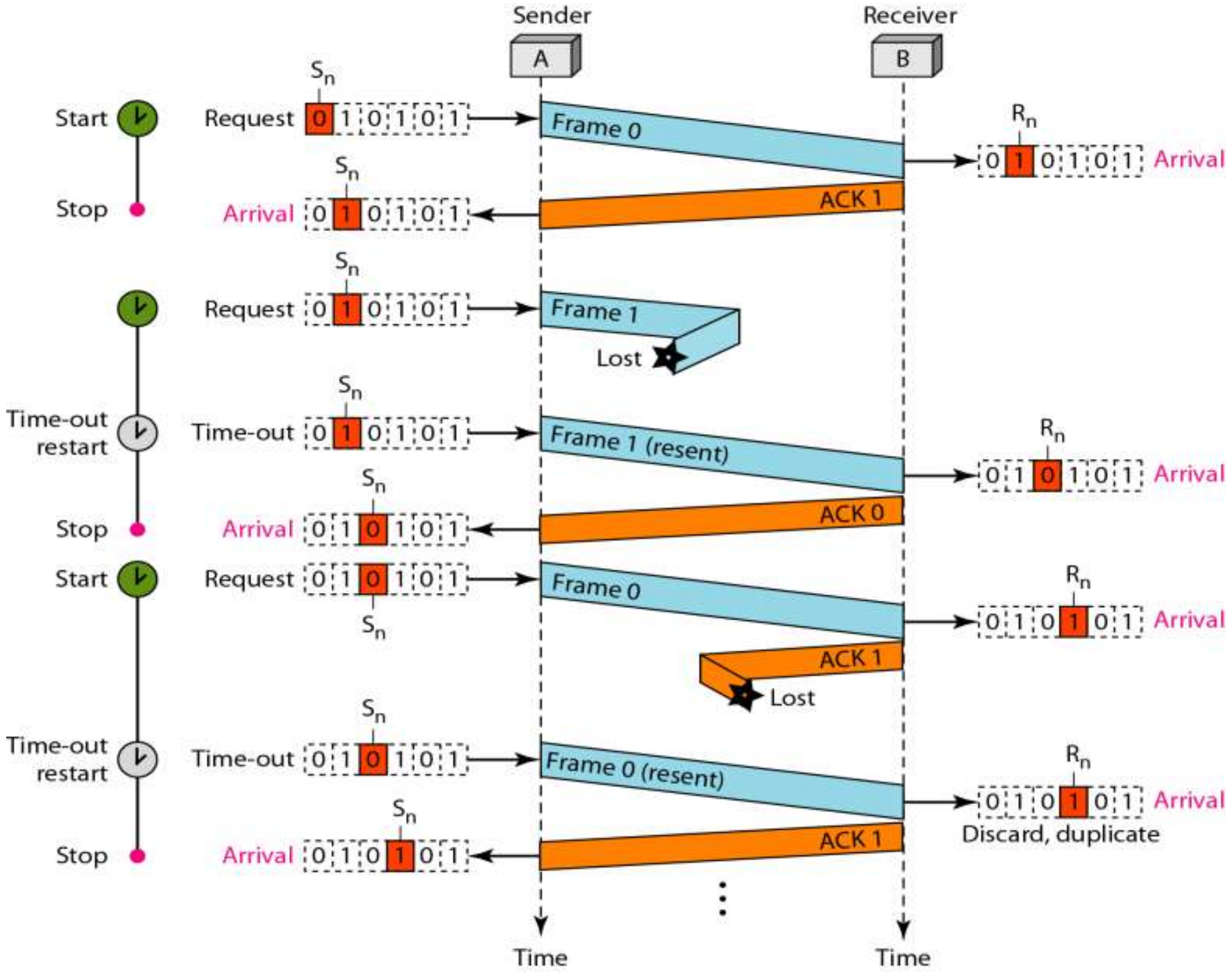
STOP-AND-WAIT, LOST ACK FRAME



- To overcome this problem it is required that the receiver be able to distinguish a frame that it is seeing for the first time from a retransmission. One way to achieve this is to have the sender put a **sequence number** in the header of each frame it sends. The receiver then can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.

- The receiver needs to distinguish only 2 possibilities: a new frame or a duplicate; a **1-bit sequence number** is sufficient. At any instant the receiver expects a particular sequence number. Any wrong sequence numbered frame arriving at the receiver is rejected as a duplicate. A correctly numbered frame arriving at the receiver is accepted, passed to the host, and the expected sequence number is incremented by 1 (modulo 2).

FLOW DIAGRAM OF A STOP & WAIT PROTOCOL



- After transmitting a frame and starting the timer, the sender waits for something exciting to happen.
  - Only three possibilities exist: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires.
- If a valid acknowledgement comes in, the sender fetches the next packet from its network layer and puts it in the buffer, overwriting the previous packet. It also advances the sequence number. If a damaged frame arrives or no frame at all arrives, neither the buffer nor the sequence number is changed so that a duplicate can be sent.
- When a valid frame arrives at the receiver, its sequence number is checked to see if it is a duplicate. If not, it is accepted, passed to the network layer, and an acknowledgement is generated. Duplicates and damaged frames are not passed to the network layer.

Sliding Window Protocols

DATA FRAME TRANSMISSION

- Unidirectional assumption in previous elementary protocols

  ⇒ Not general

- Full-duplex - approach 1
  - Two separate communication channels(physical circuits)
    - Forward channel for data
    - Reverse channel for acknowledgement

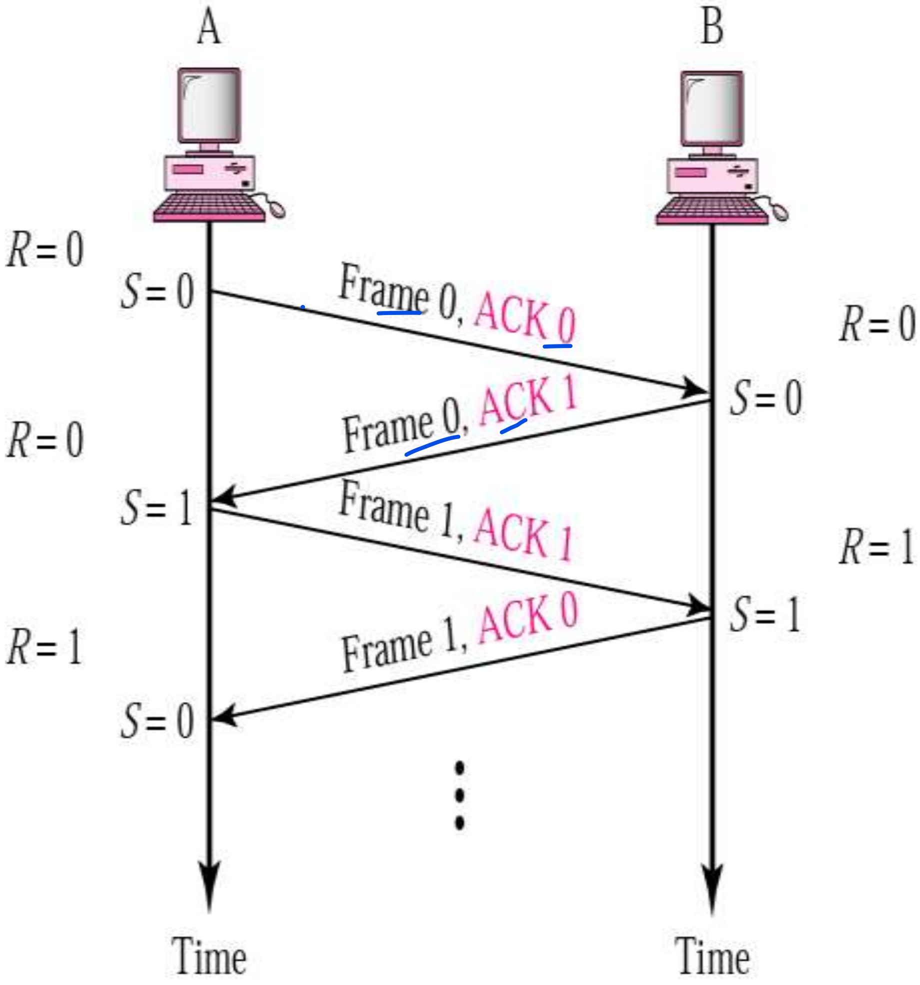
⇒ Problems: 1. reverse channel bandwidth wasted

2. cost

Data Frame Transmission

- Full-duplex - approach 2
  - Same circuit for both directions
  - Data and acknowledgement are intermixed
  - How do we tell acknowledgement from data?

    "*kind*" field telling data or acknowledgement
  - Can it be improved?



- Approach 3
  - Attaching acknowledgement to outgoing data frames

    ⇒ **PIGGYBACKING**

PIGGYBACKING

- Temporarily delaying transmission of outgoing acknowledgement so that they can be hooked onto the next outgoing data frame
- Advantage: higher channel bandwidth utilization
- Complication:
  - How long to wait for a packet to piggyback?
  - If longer than sender timeout period then sender retransmits

    ⇒ Purpose of acknowledgement is lost
- Solution for timing complexion
  - If a new packet arrives quickly

    ⇒ Piggybacking
  - If no new packet arrives after a receiver ack timeout

    ⇒ Sending a separate acknowledgement frame

Sliding Window Protocols

- The next three protocols are bidirectional protocols that belong to a class called sliding window protocols. (max sending window size, receiving window size)
  - One-bit sliding window protocol (1, 1)
  - Go back N (>1, 1)
  - Selective repeat (>1, >1)
- The three differ among themselves in terms of efficiency, complexity, and buffer requirements.
- Each outbound frame contains an *n*-bit sequence number
  - Range: 0 - MAX_SEQ (MAX_SEQ = 2^n - 1)
  - For stop-and-wait, *n* = 1 restricting the sequence numbers to 0 and 1 only

Sending & Receiving Windows

- At any instance of time
  - Sender maintains a set of sequence numbers of frames *permitted to send*
    - These frames fall within *sending window*
  - Receiver maintains a set of sequence numbers of frames *permitted to accept*
    - These frames fall within *receiving window*
- Lower limit, upper limit, and size of two windows *need not be the same* - Fixed or variable size
- **Senders Window** contains frames can be sent or have been sent but not yet acknowledged – *outstanding* frames
- When a packet arrives from network layer
  - Next highest sequence number assigned
  - Upper edge of window advanced by 1
- When an acknowledgement arrives
  - Lower edge of window advanced by 1

- If the maximum window size is n, the sender needs n buffers to hold the unacknowledged frames. If the window ever grows to its maximum size, the sending data link layer must forcibly shut off the network layer until another buffer becomes free.

- The receiving data link layer's window corresponds to the frames it may accept. Any frame falling outside the window is discarded without comment. When a frame whose sequence number is equal to the lower edge of the window is received, it is passed to the network layer, an acknowledgement is generated, and the window is rotated by one.

- Unlike the sender's window, the receiver's window always remains at its initial size.

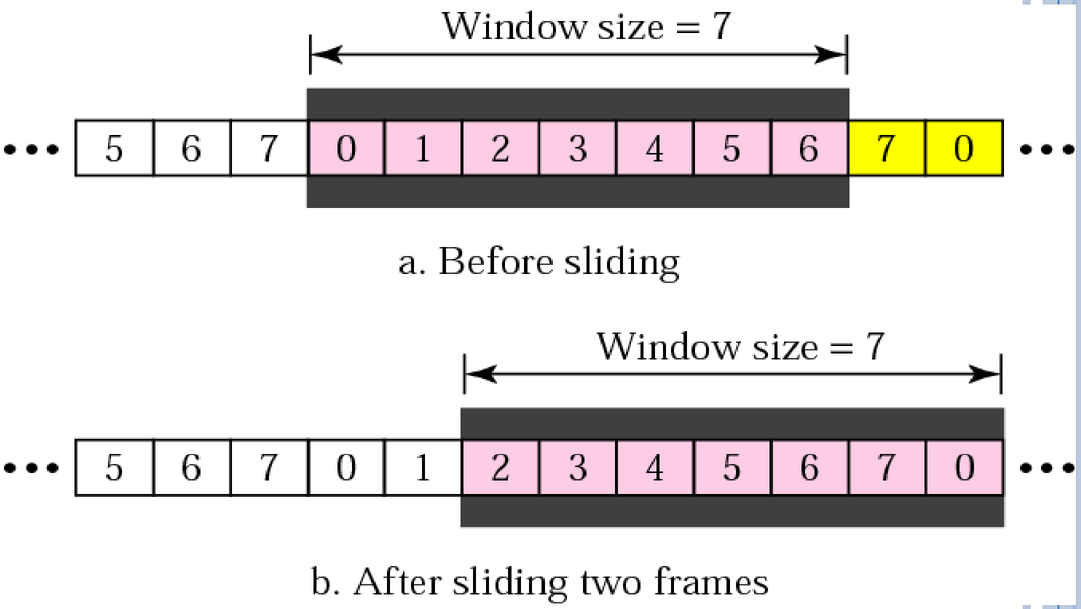
SENDER SLIDING WINDOW

- At the sending site, to hold the outstanding frames until they are acknowledged, we use the concept of a window.

- The size of the window is at most 2^m -1 where m is the number of bits for the sequence number.

- Size of the window can be variable, e.g. TCP.

- The window slides to include new unsent frames when the correct ACKs are received

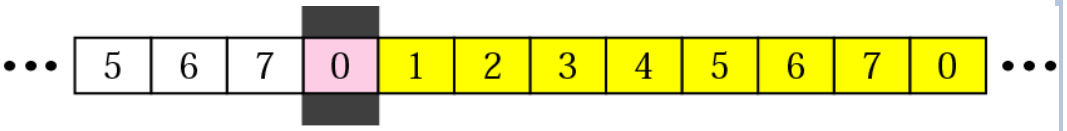


a. Before sliding

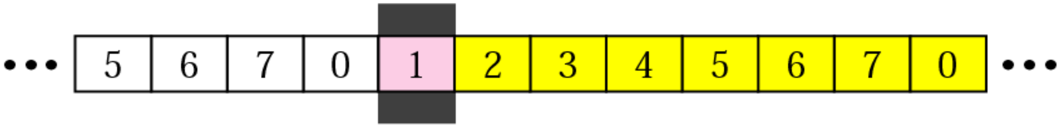
b. After sliding two frames

Receiver Sliding Window

- Size of the window at the receiving site is always 1 in this protocol.
- Receiver is always looking for a specific frame to arrive in a specific order.
- Any frame arriving out of order is discarded and needs to be resent.
- Receiver window slides as shown in fig.
  - Receiver is waiting for frame 0 in part a.



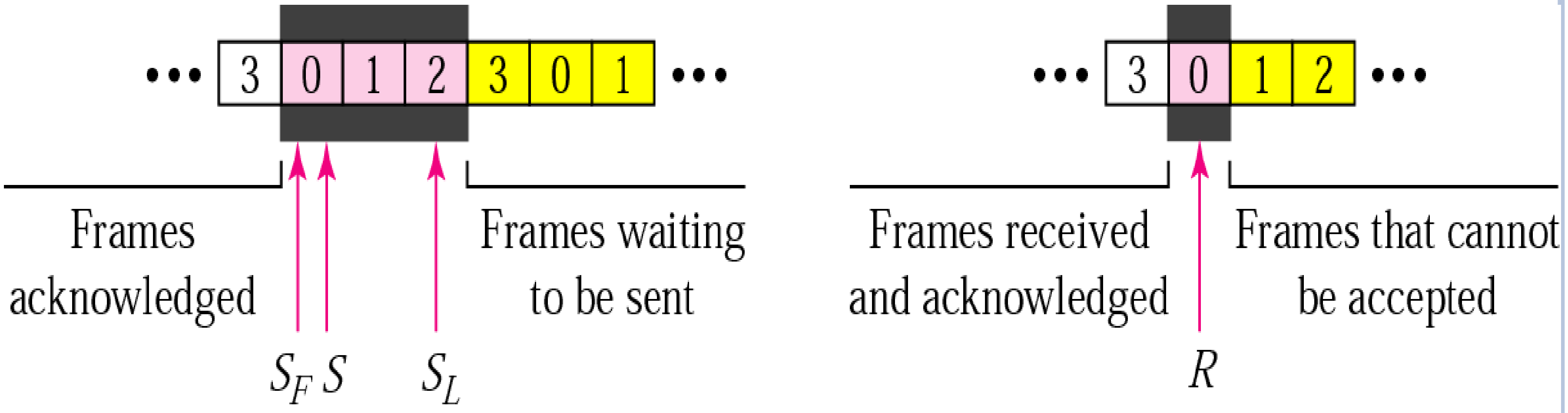
a. Before sliding



b. After sliding

Control Variables

- Sender has 3 variables: S, S_F, and S_L
- S holds the sequence number of recently sent frame
- S_F holds the sequence number of the first frame
- S_L holds the sequence number of the last frame
- Receiver only has the one variable, R, that holds the sequence number of the frame it expects to receive. If the seq. no. is the same as the value of R, the frame is accepted, otherwise rejected.



3
0
1
2
3
0
1
Frames acknowledged
Frames waiting to be sent
S_F
S
S_L
3
0
1
2
Frames received and acknowledged
Frames that cannot be accepted
R

a. Sender window

b. Receiver window

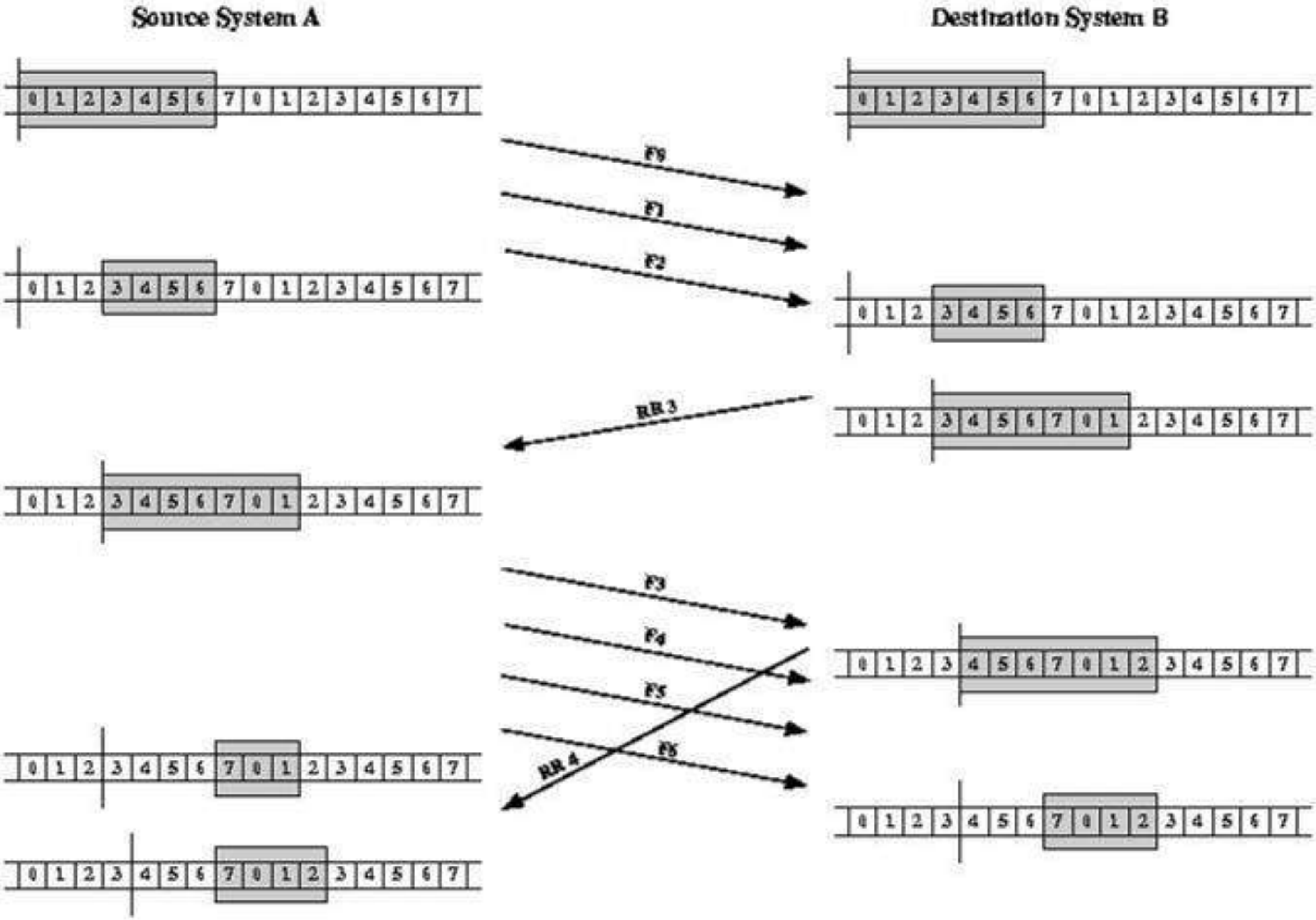
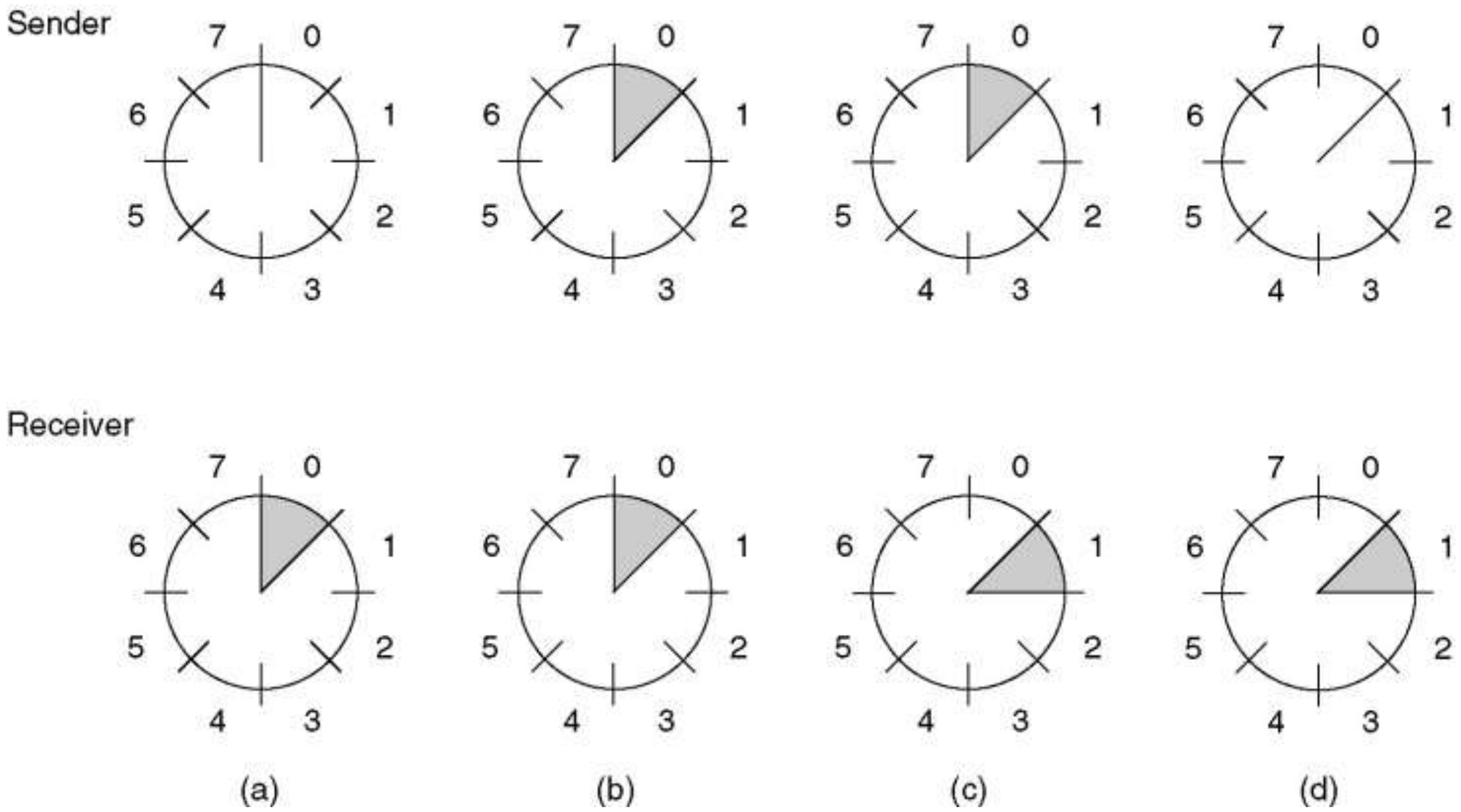


Figure 11.6 Example of a Sliding-Window Protocol

A One Bit Sliding Window Protocol



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

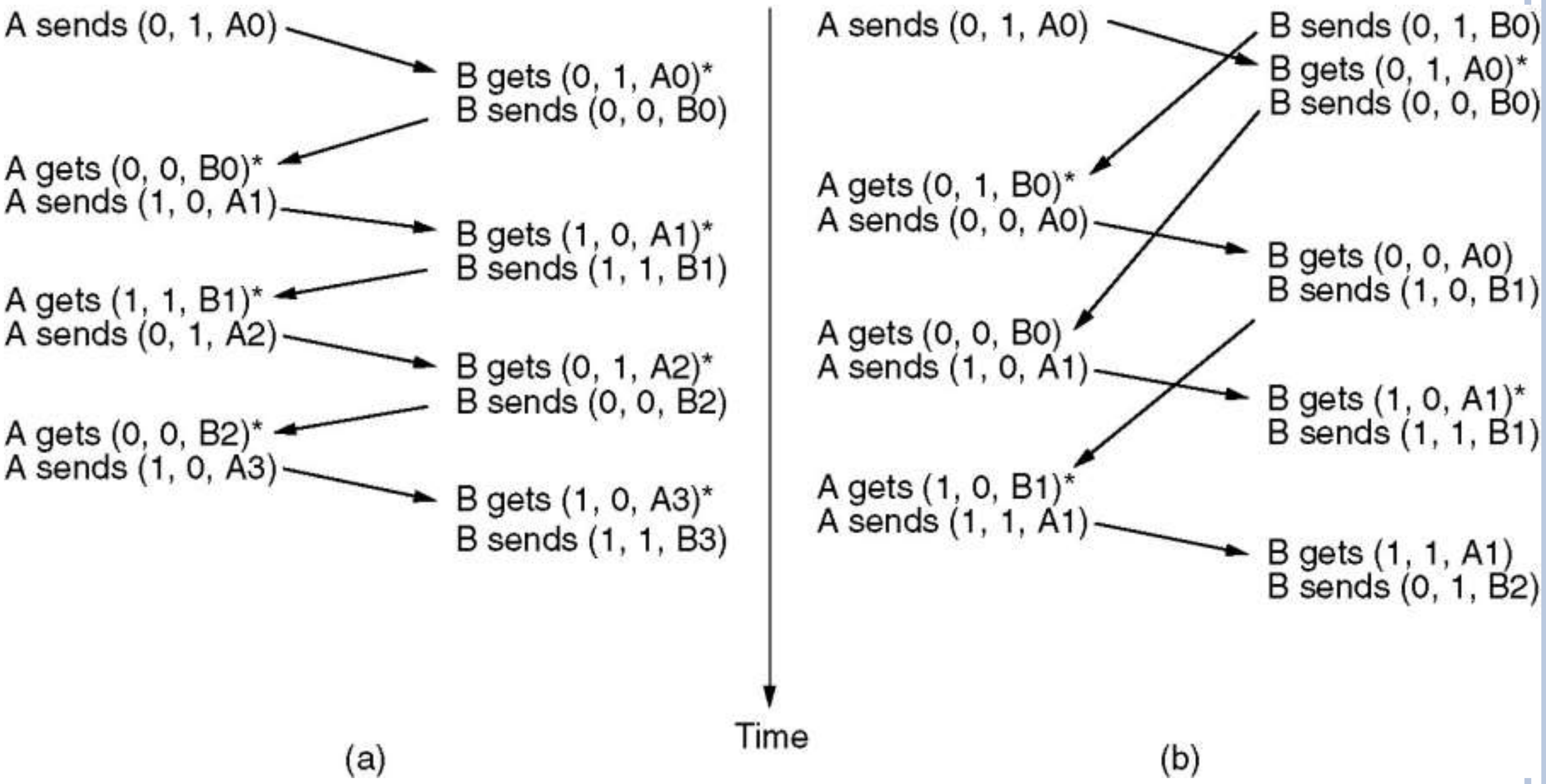
(a) Initially.

(b) After the first frame has been sent.

(c) After the first frame has been received.

(d) After the first acknowledgement has been received.

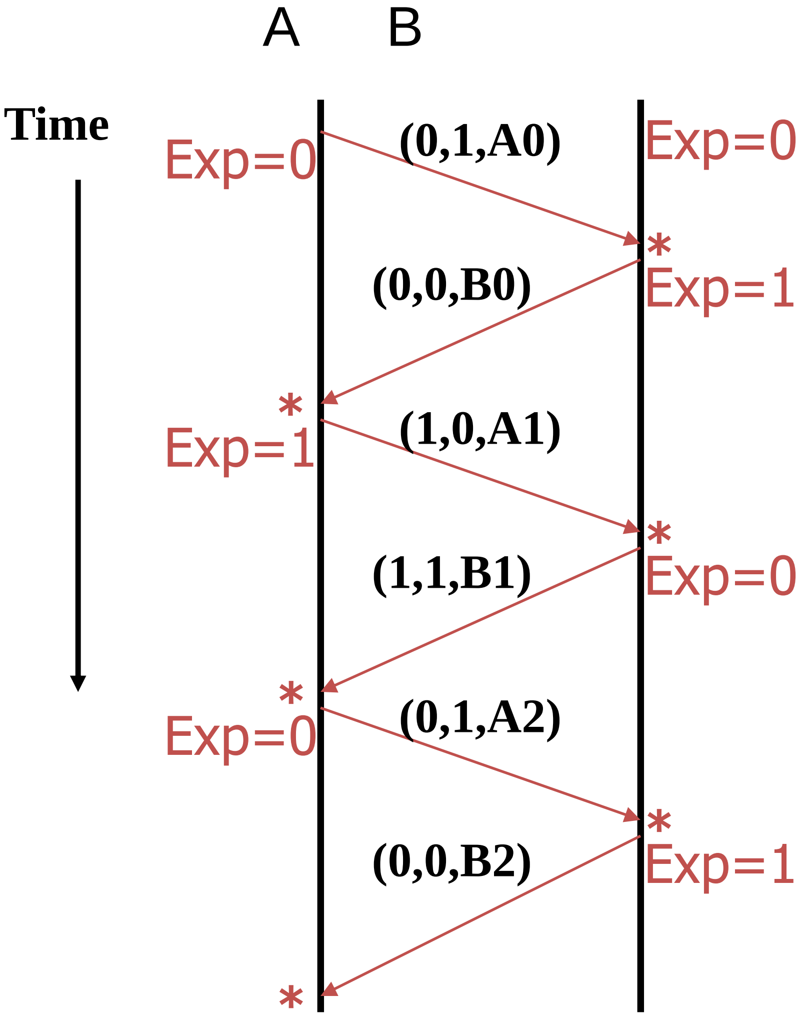
A ONE BIT SLIDING WINDOW PROTOCOL



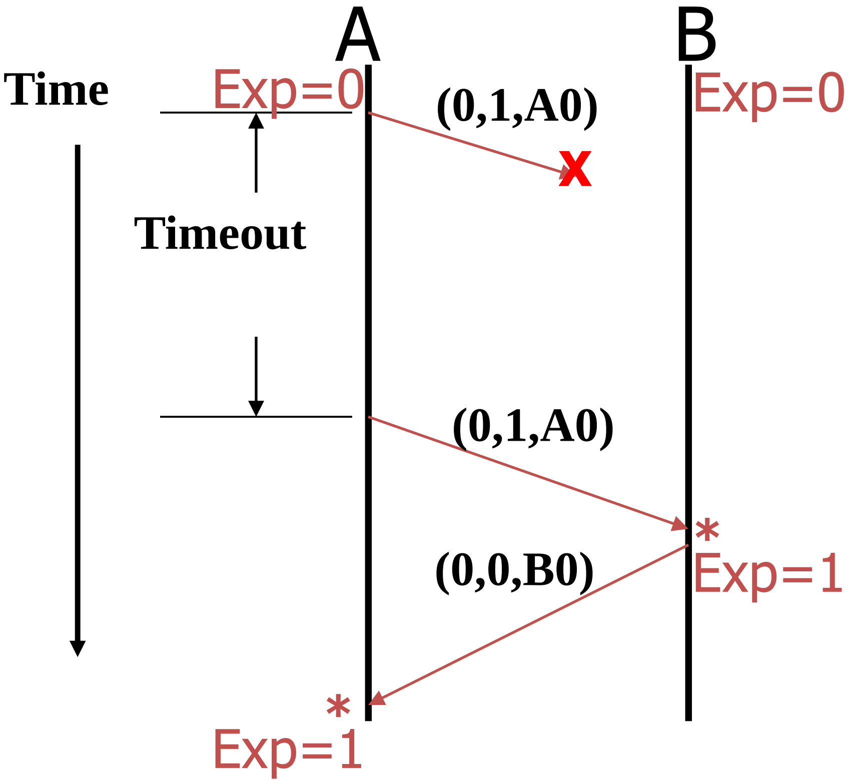
(a) Case 1: Normal case. (b) Case 7: Abnormal case.
The notation is **(seq, ack, packet number)**. An asterisk indicates where a network layer accepts a packet.

One Bit Sliding Window Protocol

- Case 1: no error

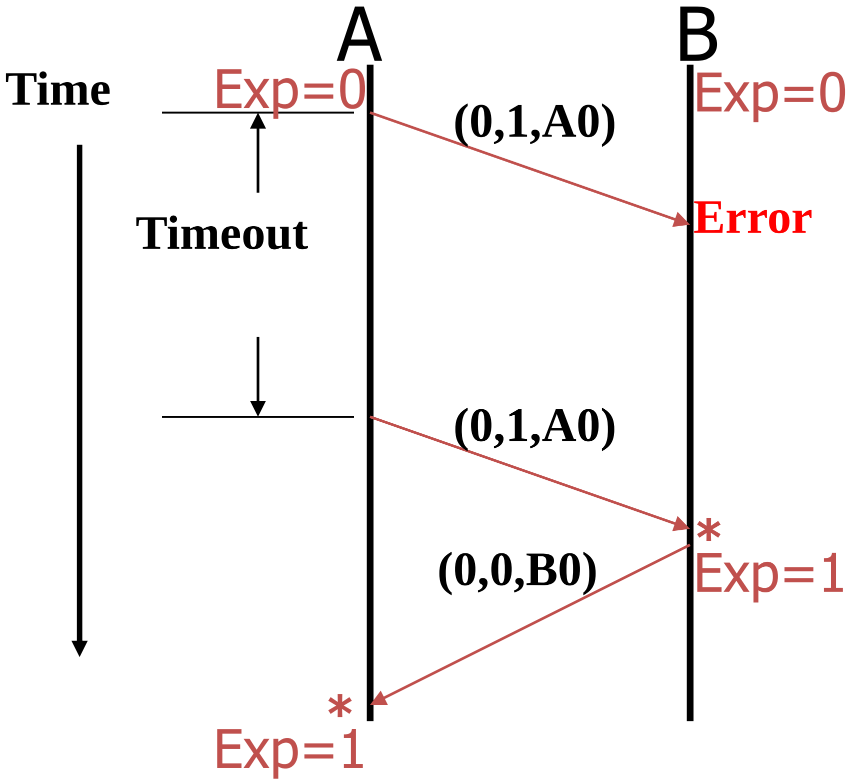


- Case 2: data lost

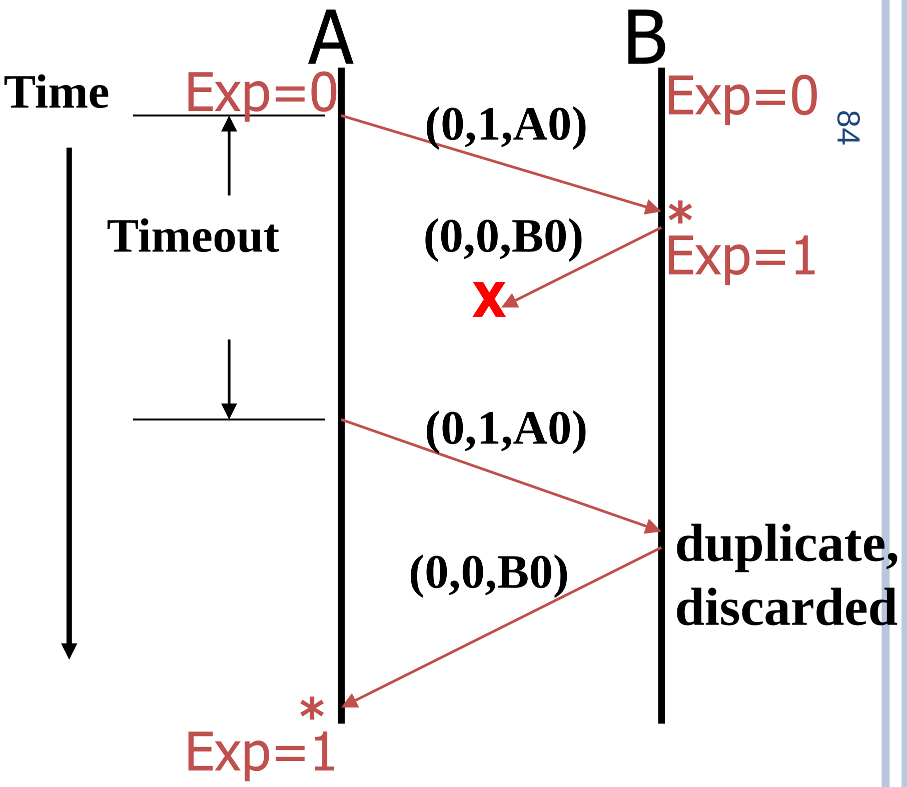


One Bit Sliding Window Protocol

- Case 3: data error

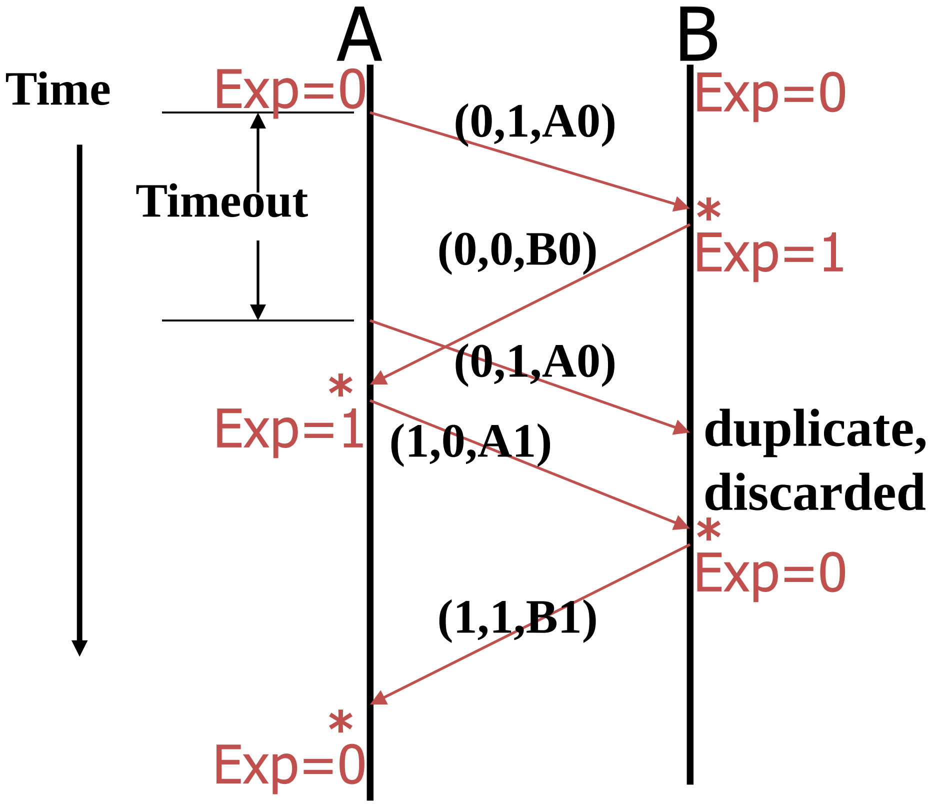


- Case 4: ack. lost

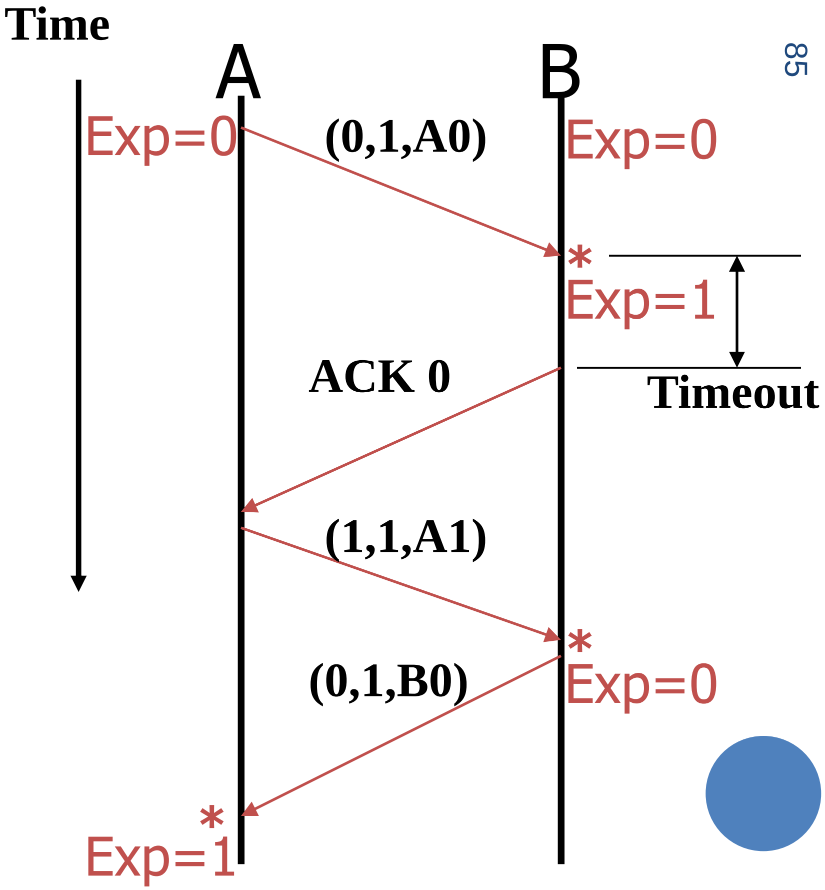


ONE BIT SLIDING WINDOW PROTOCOL

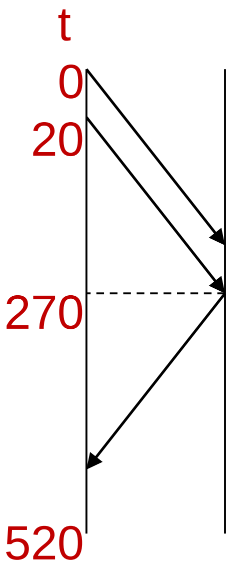
- Case 5: early timeout



- Case 6: outgoing frame timeout



Performance of Stop-and-Wait Protocol



- Assumption of previous protocols:
  - Transmission time is negligible
  - False, when transmission time is long
- Example - satellite communication
  - channel capacity: 50 kbps, frame size: 1kb
    round-trip propagation delay: 500 msec
  - Time:

t=0	start to send 1st bit in frame
t=20 msec	frame sent completely
t=270 msec	frame arrives
t=520 msec	best case of ack. Received

- Sender blocked 500/520 = 96% of time
  - Bandwidth utilization 20/520 = 4%

Conclusion:

Long transit time + high bandwidth + short frame length ⇒ **disaster**

Performance of Stop-and-Wait Protocol

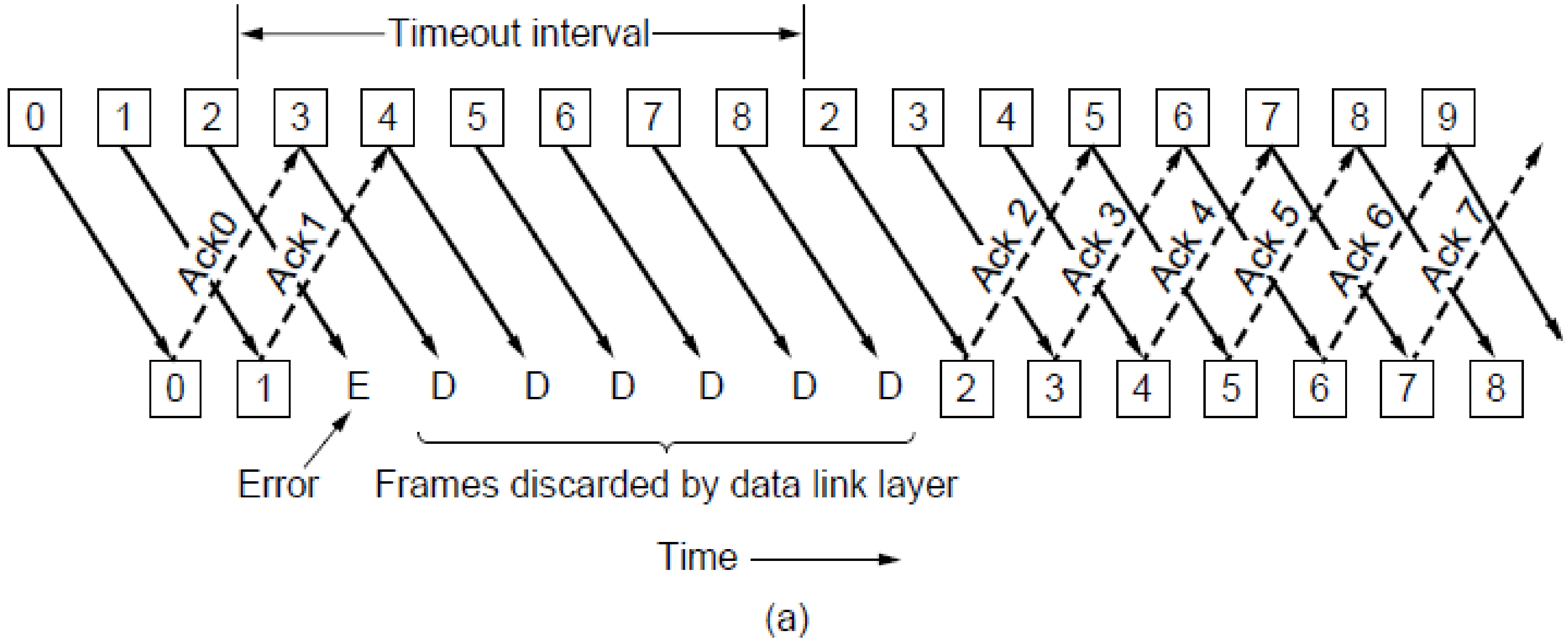
- In stop-and-wait, at any point in time, there is only one frame that is sent and waiting to be acknowledged.
- This is not a good use of transmission medium.
- To improve efficiency, multiple frames should be in transition while waiting for ACK.
- Solution: **PIPELINING**
  - Allowing *w* frames sent before blocking
- Problem: errors
- Solutions
  - **Go back *n* protocol (GNP)**
  - **Selective repeat protocol (SRP)**

Acknowledge *n* means frames *n*, *n*-1, *n*-2,… are acknowledged (i.e., received correctly)

GO BACK *N* PROTOCOL

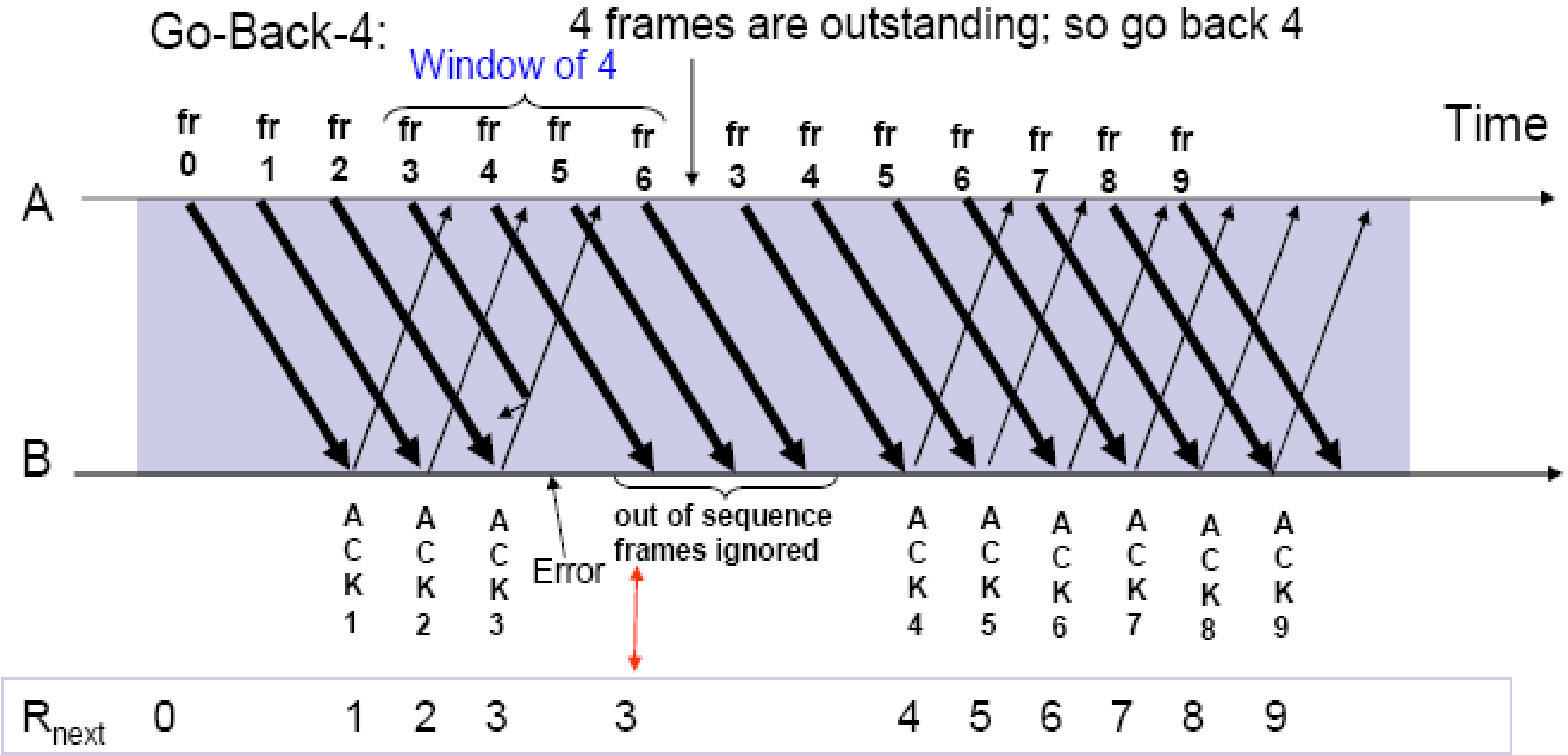
- Improves efficiency of Stop and Wait by not waiting
- Keep Channel busy by continuing to send frames
- Allow a window of upto W_s outstanding frames
- Use m-bit sequence numbering
- Receiver discards all subsequent frames following an error one, and send no acknowledgement for those discarded
- Receiving window size = 1 (i.e., frames must be accepted in the order they were sent)
- Sending window might get full
  - If so, re-transmitting unacknowledged frames
- Wasting a lot of bandwidth if error rate is high

Go Back *N* Protocol



Frames 0 and 1 are correctly received and acknowledged. Frame 2, however, is damaged or lost. The sender, unaware of this problem, continues to send frames until the timer for frame 2 expires. Then it backs up to frame 2 and starts all over with it, sending 2, 3, 4, etc. all over again.

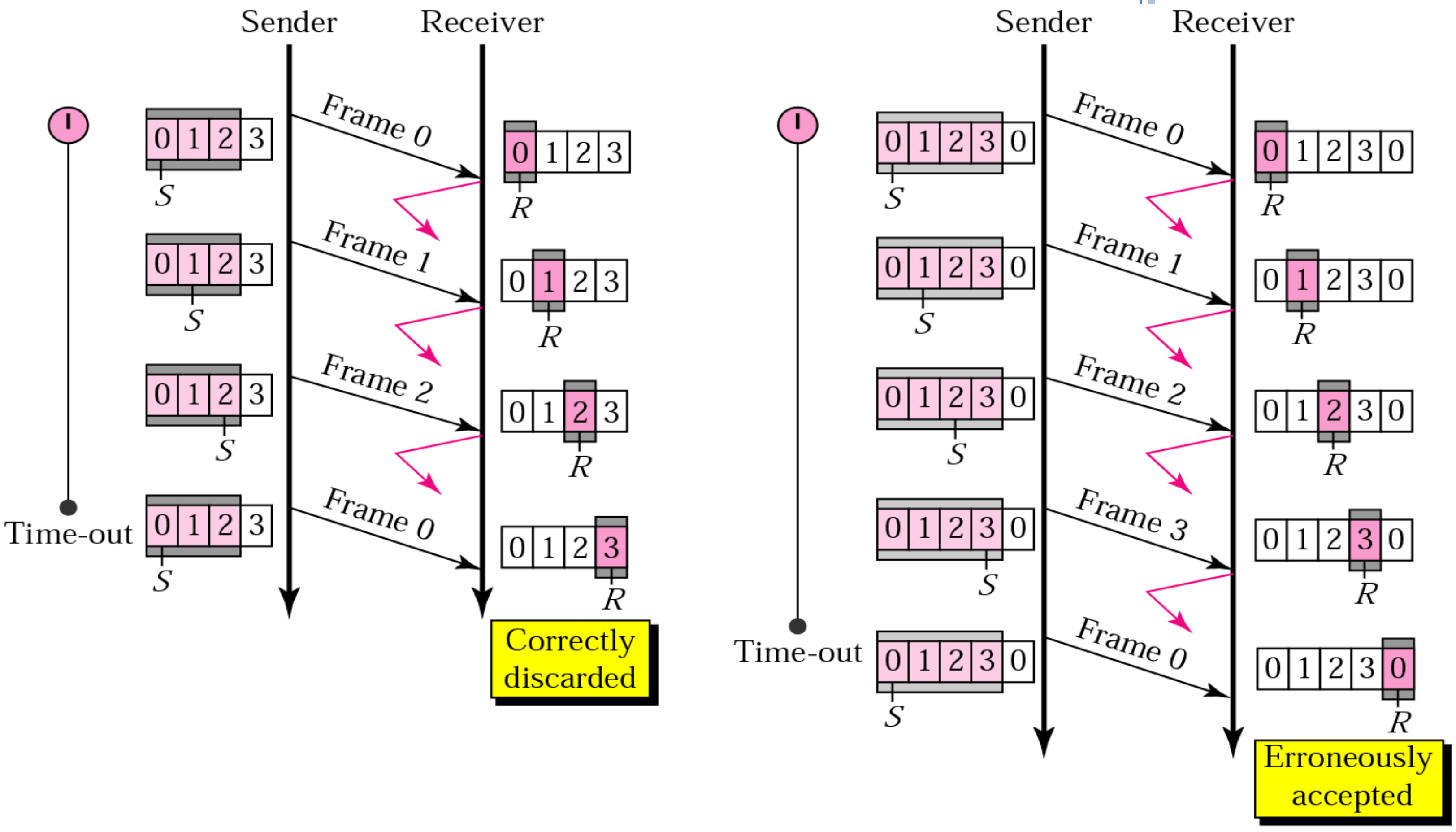
Go-Back-N ARQ with window=4



- □ Frame transmission are *pipelined* to keep the channel busy
- □ Frame with errors and subsequent out-of-sequence frames are ignored
- □ Transmitter is forced to go back when window of 4 is exhausted

GO-BACK-N ARQ, SENDER WINDOW SIZE

- Size of the sender window must be less than 2^m. Size of the receiver is always 1. If m = 2, window size = $2^m - 1 = 3$.
- Fig compares a window size of 3 and 4.



a. Window size < 2^m

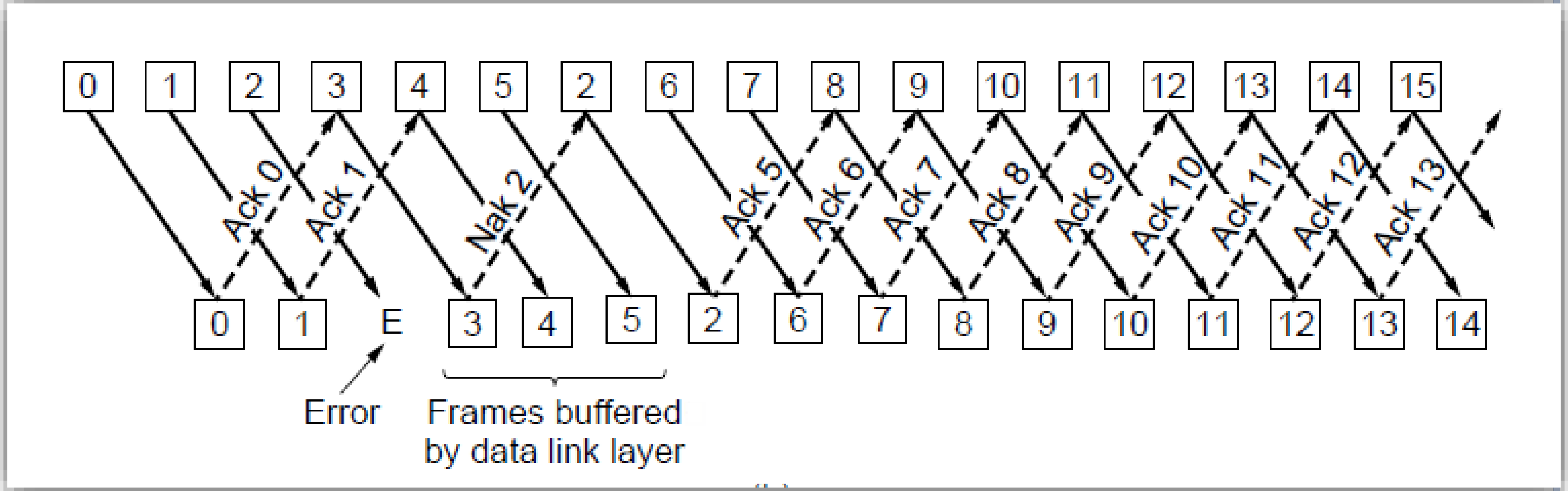
b. Window size = 2^m

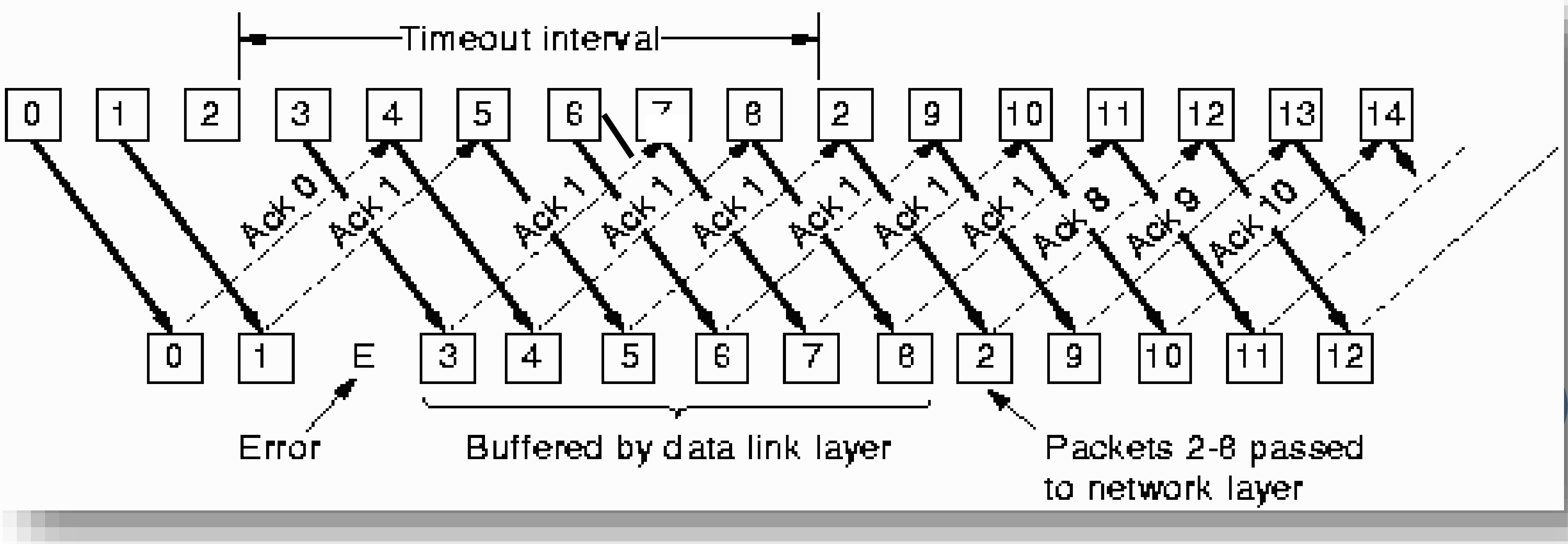
Accepts as the 1st frame in the next cycle-an **error**

Select Repeat Protocol

- Receiver stores correct frames following the bad one
- Sender retransmits the bad one after noticing
- Receiver passes data to network layer and acknowledge with the highest number
- Receiving window > 1 (i.e., any frame within the window may be accepted and buffered until all the preceding one passed to the network layer. Might need large memory
- ACK for frame *n* implicitly acknowledges all frames ≤ *n*
- SRP is often combined with NAK
- When error is *suspected* by receiver, receiver request retransmission of a frame
  - Arrival of a damaged frame
  - Arrival of a frame other than the expected\
- NAKs stimulate retransmission before the corresponding timer expires and thus improve performance.

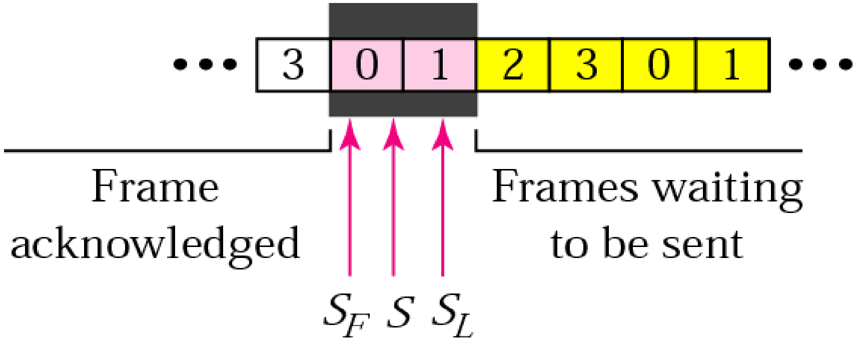
SELECTIVE REPEAT WITH NAK



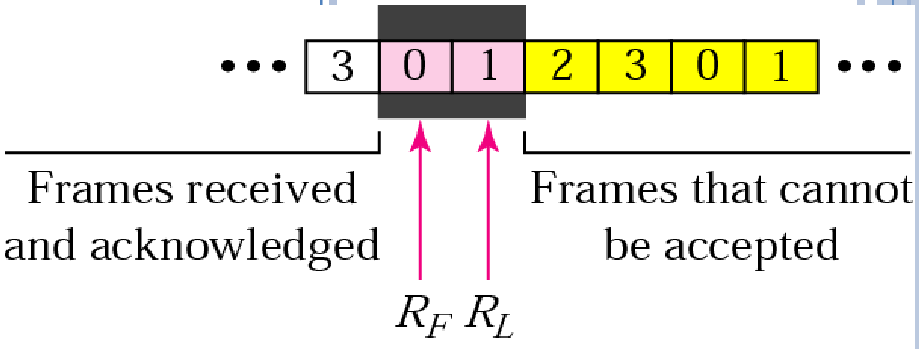


SELECTIVE REPEAT ARQ, SENDER AND RECEIVER WINDOWS.

- Go-Back-N ARQ simplifies the process at the receiver site. Receiver only keeps track of only one variable, and there is no need to buffer out-of-order frames, they are simply discarded.
- However, Go-Back-N ARQ protocol is inefficient for noisy link. It bandwidth inefficient and slows down the transmission.
- In Selective Repeat ARQ, only the damaged frame is resent. More bandwidth efficient but more complex processing at receiver.
- It defines a negative ACK (NAK) to report the sequence number of a damaged frame before the timer expires.



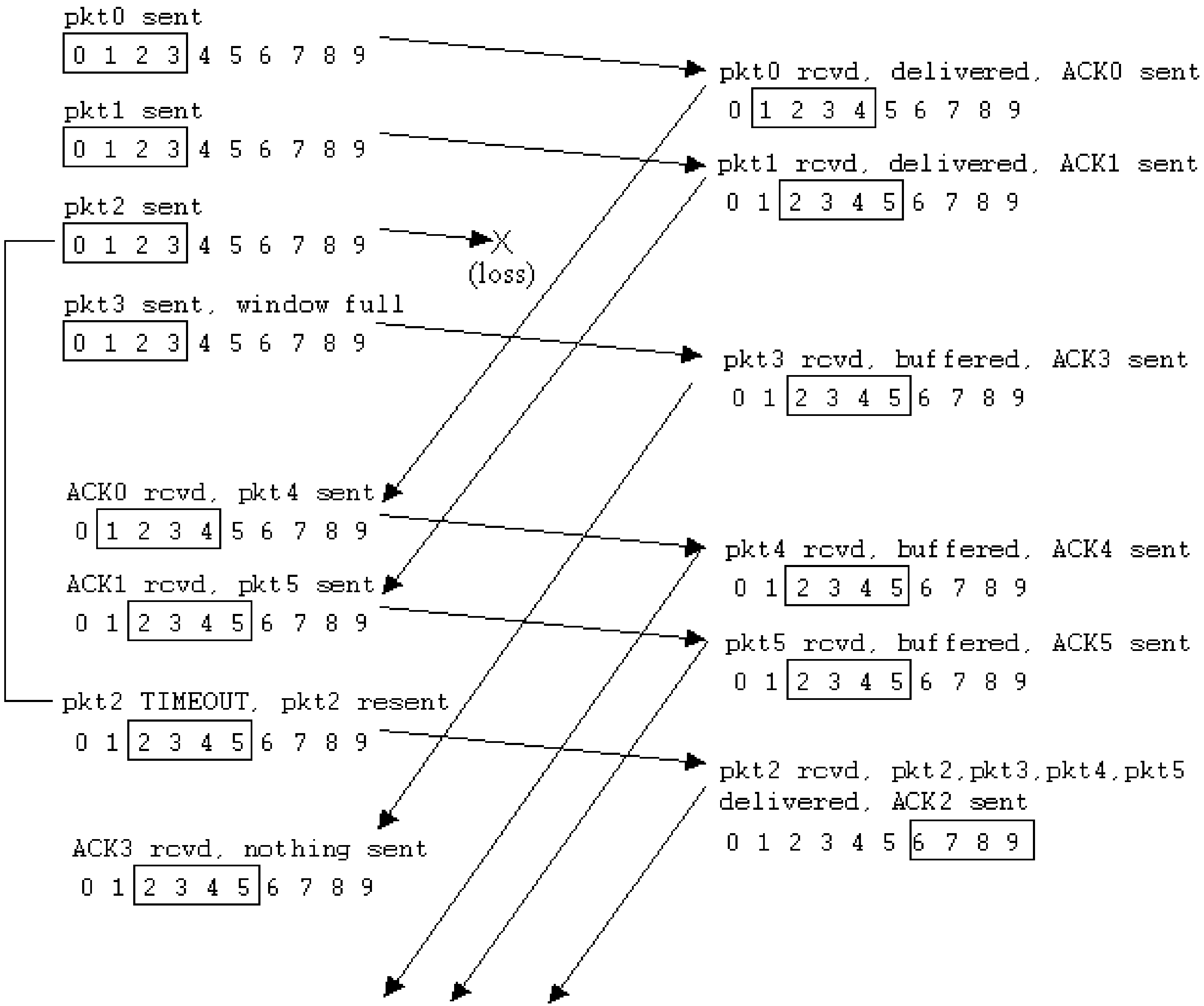
a. Sender window



•••
3
0
1
2
3
0
1
•••
Frames received and acknowledged
Frames that cannot be accepted
R_F
R_L

b. Receiver window

Selective Repeat in Action



pkt0 sent

0 1 2 3 4 5 6 7 8 9

pkt1 sent

0 1 2 3 4 5 6 7 8 9

pkt2 sent

0 1 2 3 4 5 6 7 8 9

X

(loss)

pkt3 sent, window full

0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent

0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent

0 1 2 3 4 5 6 7 8 9

pkt2 TIMEOUT, pkt2 resent

0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent

0 1 2 3 4 5 6 7 8 9

pkt0 rcvd, delivered, ACK0 sent

0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent

0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent

0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent

0 1 2 3 4 5 6 7 8 9

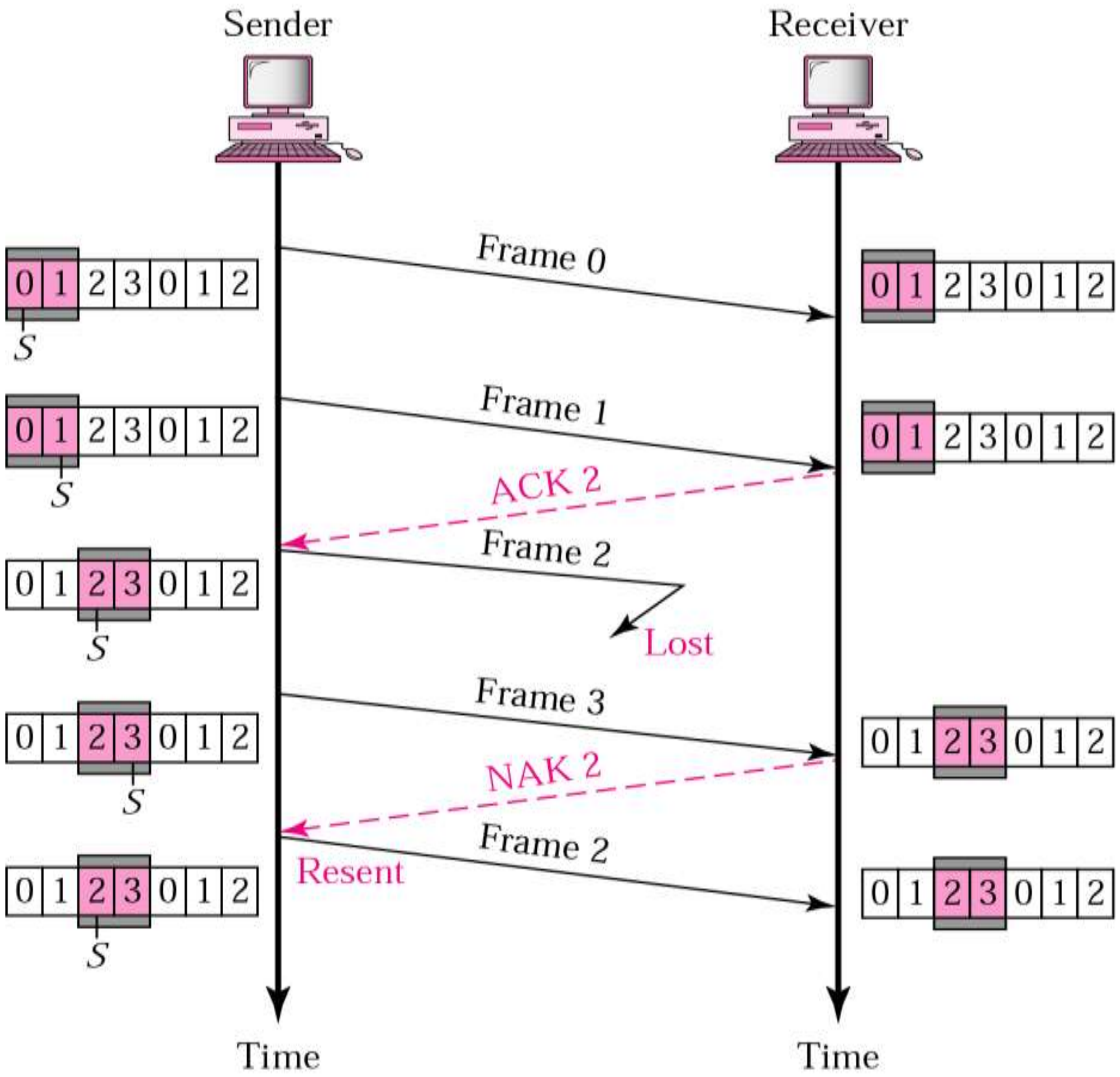
pkt5 rcvd, buffered, ACK5 sent

0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5
delivered, ACK2 sent

0 1 2 3 4 5 6 7 8 9

SELECTIVE REPEAT ARQ, LOST FRAME

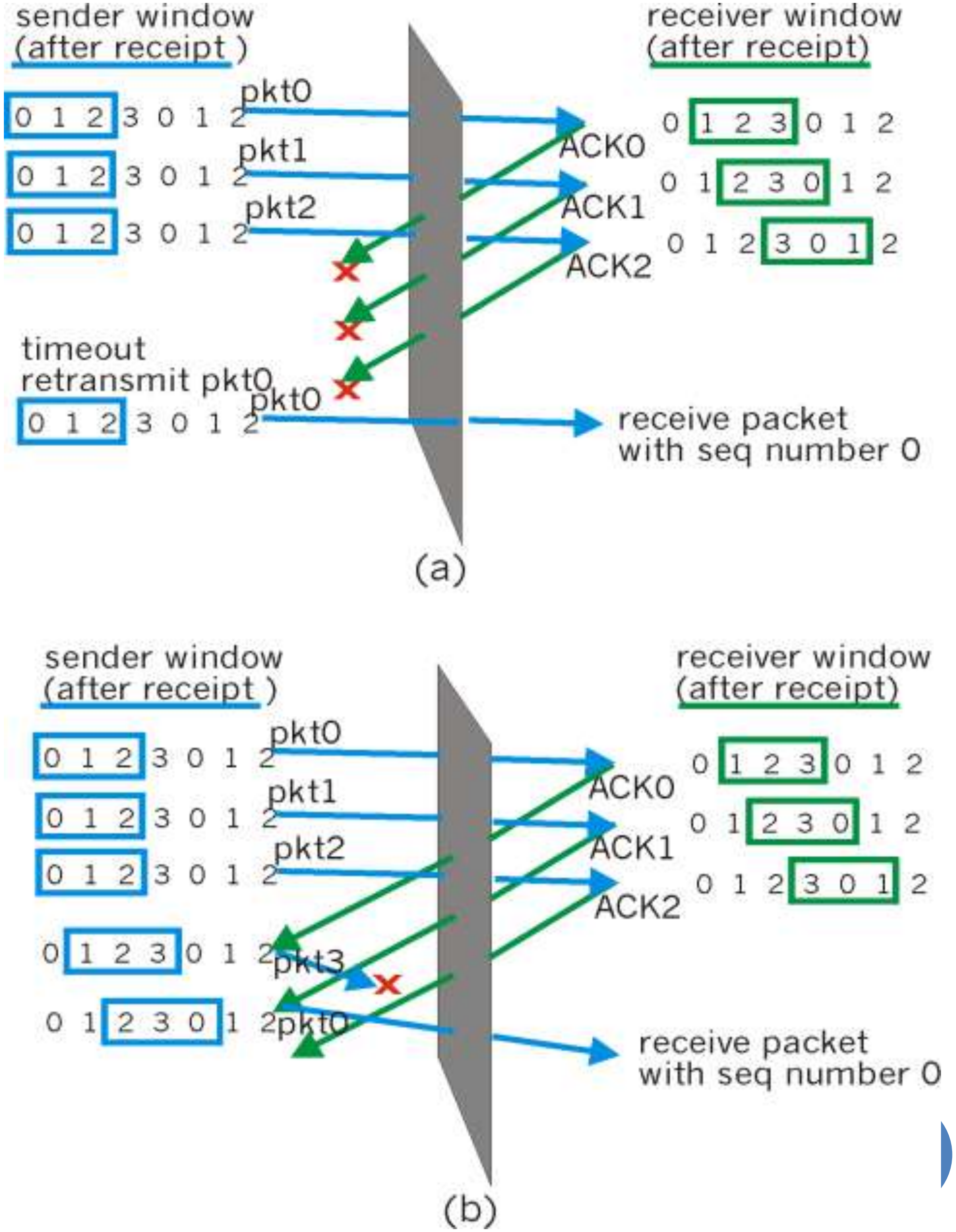


- Frames 0 and 1 are accepted when received because they are in the range specified by the receiver window. Same for frame 3.

- Receiver sends a NAK2 to show that frame 2 has not been received and then sender resends only frame 2 and it is accepted as it is in the range of the window.

Selective Repeat Dilemma

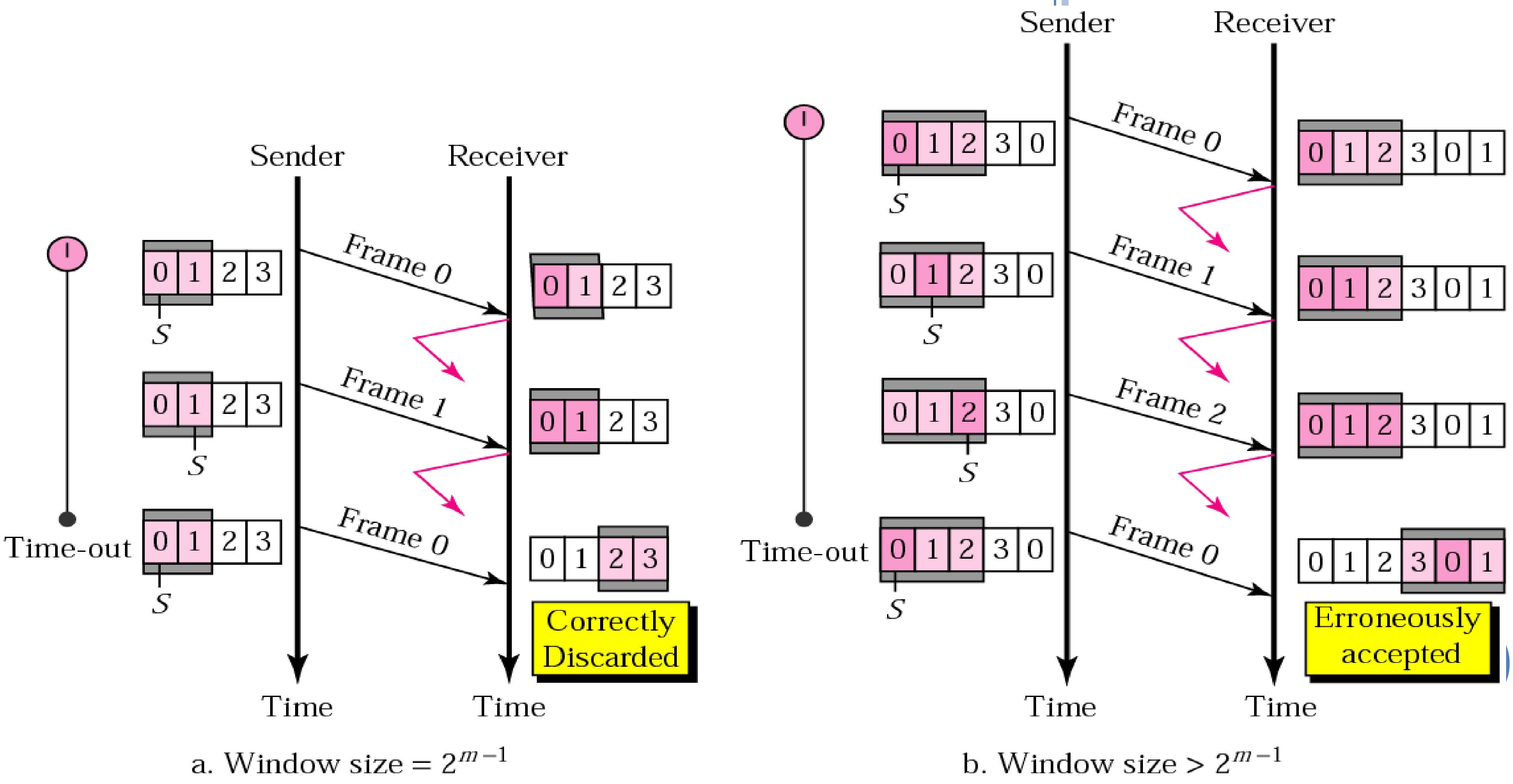
- Example:
- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)
- Q: what relationship between seq # size and window size?



sender window (after receipt)
receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
0 1 2 3 0 1 2
timeout retransmit pkt0
pkt0
receive packet with seq number 0
(a)
sender window (after receipt)
receiver window (after receipt)
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
pkt3
pkt0
receive packet with seq number 0
(b)

SELECTIVE REPEAT ARQ, SENDER WINDOW SIZE

- Size of the sender and receiver windows must be at most one-half of 2^m.
- If m = 2, window size should be $2^m/2 = 2$. Fig compares a window size of 2 with a window size of 3. Window size is 3 and all ACKs are lost, sender sends duplicate of frame 0, window of the receiver expect to receive frame 0 (part of the window), so accepts frame 0, as the 1st frame of the next cycle – an **error**.



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

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

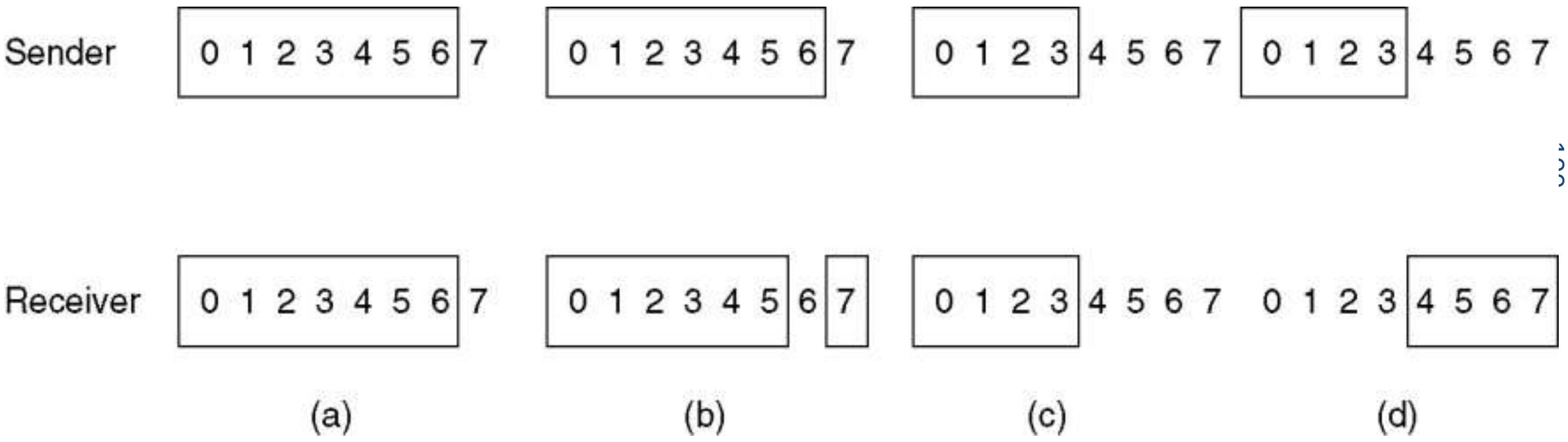
Select Repeat Protocol - Window Size

- Problem is caused by new and old windows overlapped
- Solution
  - Window size=(MAX_SEQ+1)/2
  - E.g., if 4-bit window is used, MAX_SEQ = 15

    ⇒ window size = (15+1)/2 = 8
- Number of buffers needed

  = window size

Select Repeat Protocol



(a) Initial situation with a window size seven.

(b) After seven frames sent and received, but not acknowledged.

(c) Initial situation with a window size of four.

(d) After four frames sent and received, but not acknowledged.

ACKNOWLEDGEMENT TIMER

- Problem
  - If the reverse traffic is light, effect?
  - If there is no reverse traffic, effect?
- Solution
  - Acknowledgement timer:

    If no reverse traffic before timeout

    send separate acknowledgement
  - Essential: ack timeout < data frame timeout Why?