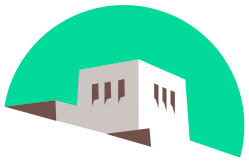
Chapter 3: The Data Link Layer



Data Link Layer

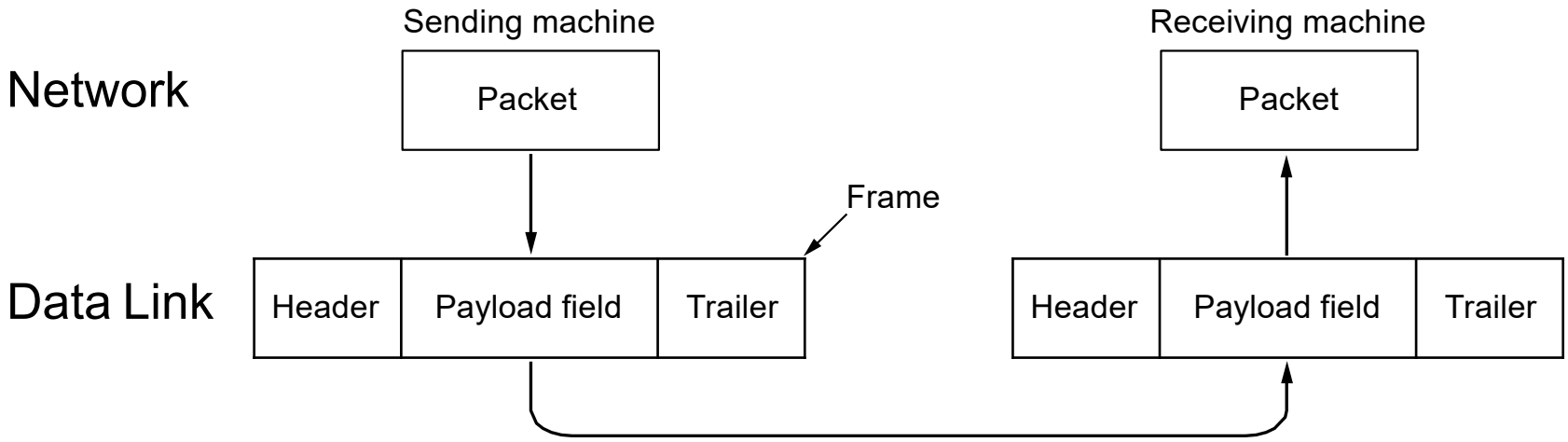
Reliable bitstream between adjacent computers

- Design Issues
- Error Detection and Correction
- Elementary Protocols
- Sliding Window Protocols Protocol Verification
- Example Data Link Protocols

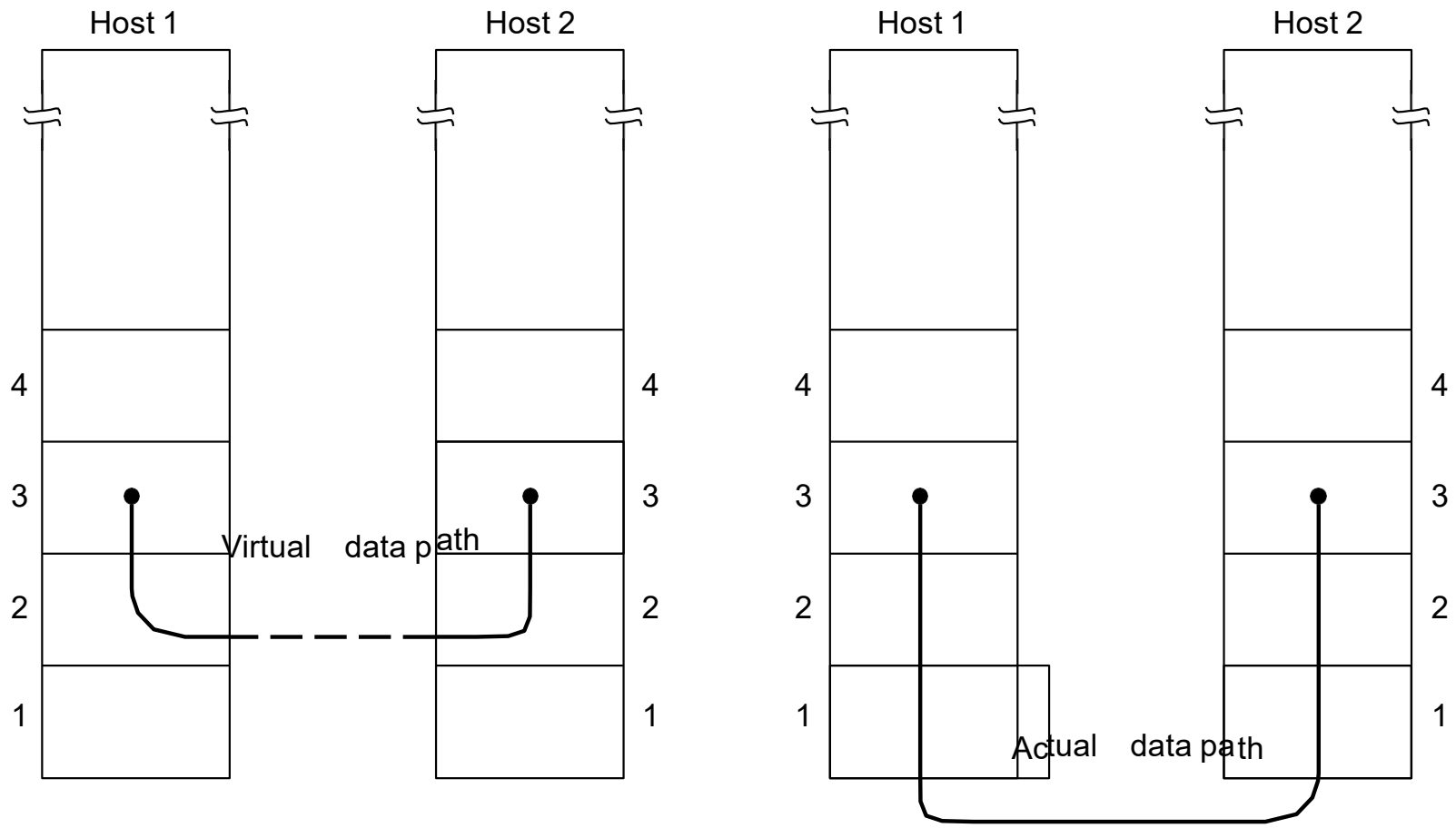
Design Issues

- Provides an interface to network layer
- Deals with transmission errors
- Regulates flow of data so slow receivers are not swamped
- Topics
  - Services
  - Framing
  - Error control
  - Flow control

Terminology: Packets and Frames



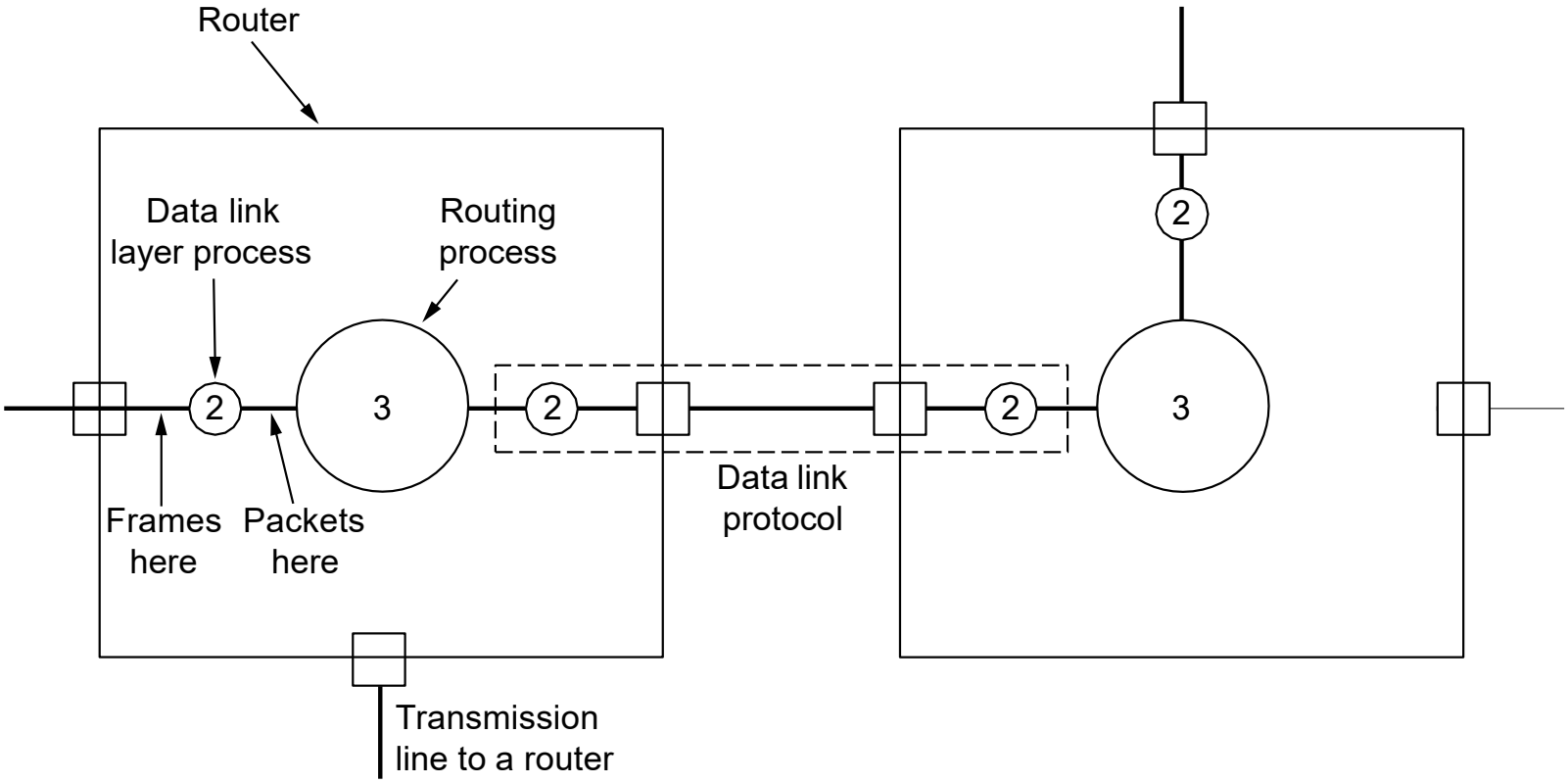
Layered Communication



Kinds of Services

- Unacknowledged connectionless service
  - no error detection or recovery
  - appropriate when error rate is very low
  - real-time – late data are worse than bad data
- Acknowledged connectionless service
  - each packet is acknowledged
  - retransmit after timeout expires
  - optimization when errors are frequent
- Acknowledged connection-oriented service
  - number frames to detect duplicates
  - connection setup, transmission, connection release

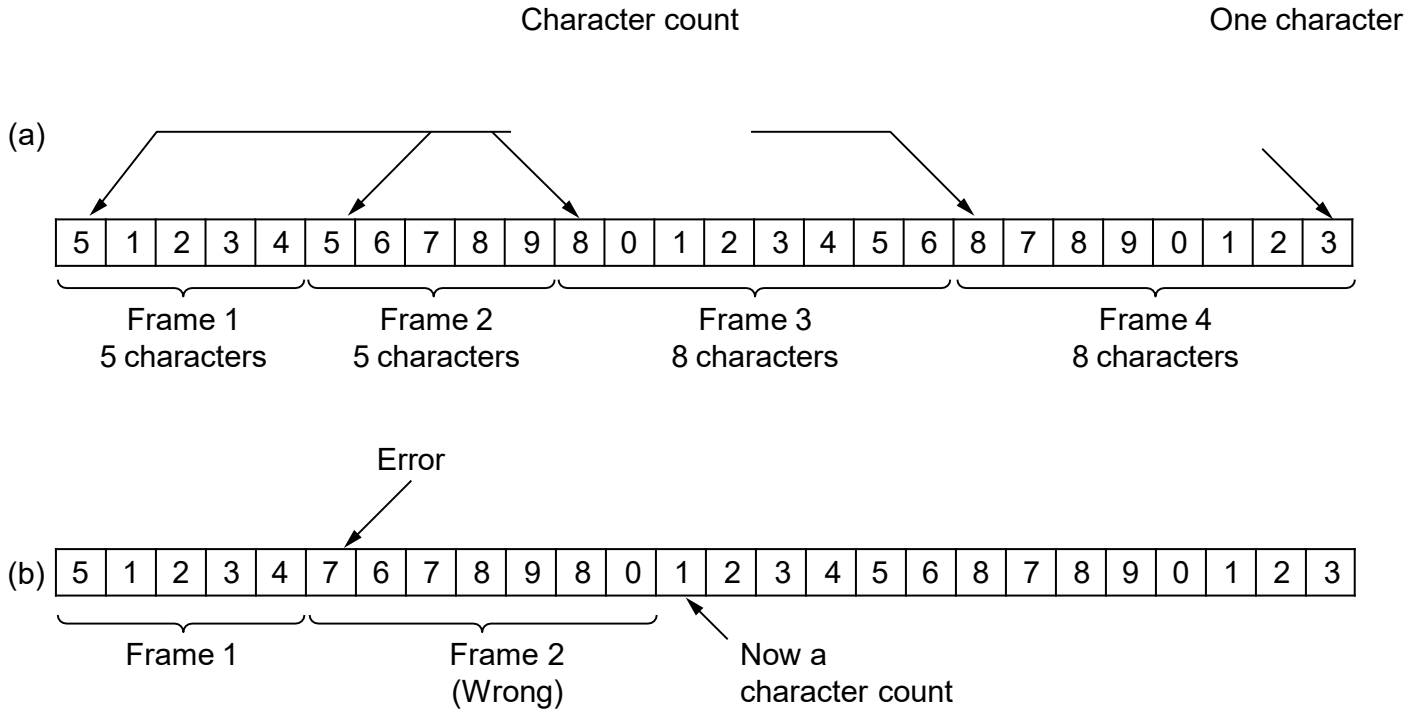
Example: The Data Link Layer in Context



Framing (1 of 3)

How to mark the start and end of each frame?

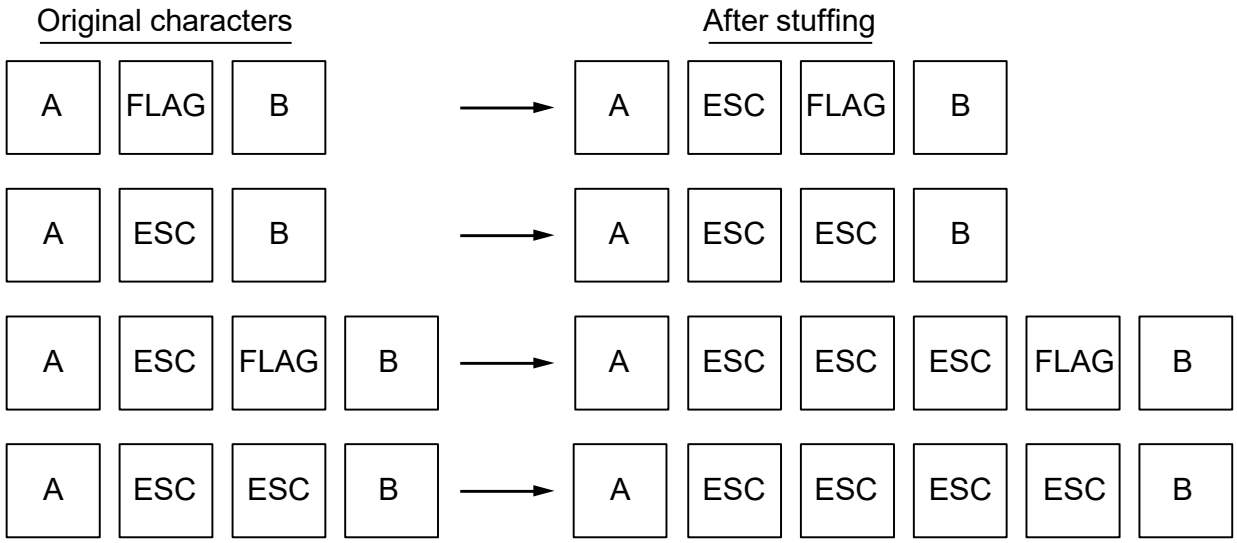
- 1. Character count: field in header for frame size
- transmission errors are problematic



Framing (2 of 3)

- 2. Flag bytes: two consecutive flags indicates end and start of frame
  - flag bytes

  - escape conventions (stuffing)



Framing (Byte Stuffing)

Problem 3. The following data fragment occurs in the middle of a data stream for which the bytestuffing algorithm described in the text is used:

A B ESC C ESC FLAG FLAG D.

What is the output after stuffing?

Solution: After stuffing, we get

A B ESC ESC C ESC ESC ESC FLAG ESC FLAG D.

Framing (3 of 3)

- 3. Flags with bit stuffing

  Flag is 01111110

  (a) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

  (b) 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0

  Stuffed bits

  (c) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

- 4. Physical layer coding violations

  physical layer provides a “frame” service

Error Control

- Receiver provides feedback in the form of positive or negative acknowledgements (ACKs)
- Dropped frames are handled using timeouts
- Dropped transmissions or ACKs are handled by retransmitting
- Duplicate frames are handled using sequence numbers

Flow Control

->Slowing down a fast sender

- feedback-based flow control
- rate-based flow control

Data Link Layer

Reliable bitstream between adjacent computers

- Design Issues
- Error Detection and Correction
- Elementary Protocols
- Sliding Window Protocols Protocol Verification
- Example Data Link Protocols

3.2 Error Detection and Correction

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

- One strategy is to deduce what the transmitted data must have been: **Error Correction**
- The other is to only deduce that an error has occurred (but not which error); **Error Detection**

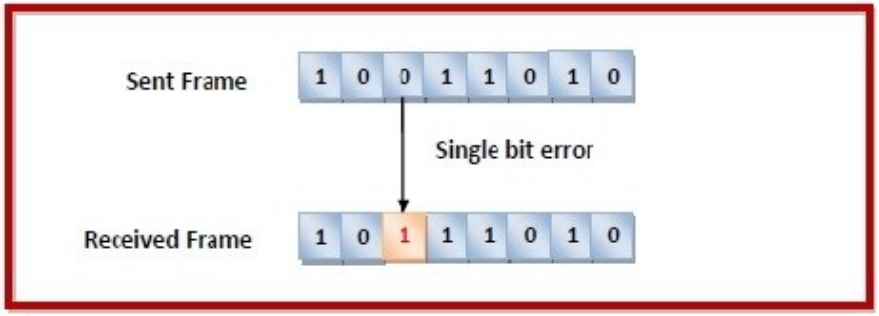
Neither error-correcting codes nor error-detecting codes can handle **all possible errors**.

- Error correcting is used on unreliable links
- Error detecting (enough to know something went wrong) is used on reliable links. Just retransmission.

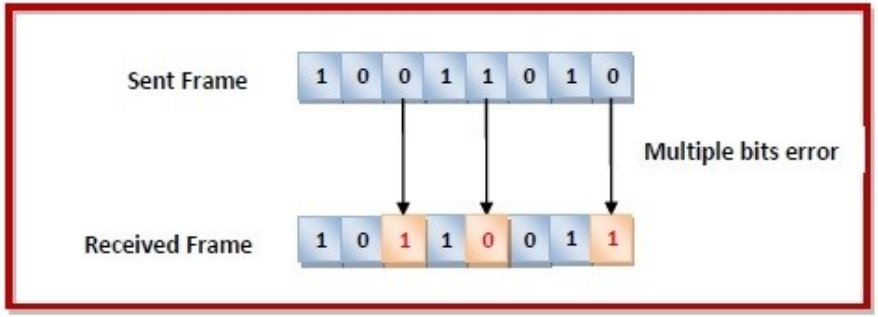
3.2 Error Detection and Correction

Types of Errors: Errors can be of three types, namely single bit errors, multiple bit errors, and burst errors.

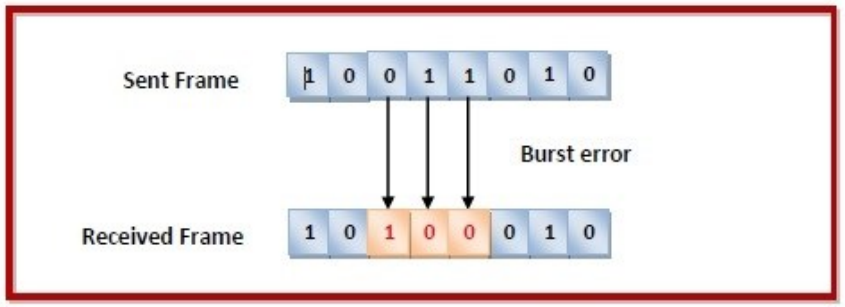
•**Single bit error** − In the received frame, only one bit has been corrupted, i.e. either changed from 0 to 1 or from 1 to 0.



•**Multiple bits error** − In the received frame, more than one bits are corrupted.



•**Burst error** − In the received frame, more than one consecutive bits are corrupted.



3.2 Error Detection and Correction

We will examine three different error-detecting codes. They are all linear, systematic block codes:

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

Parity Check

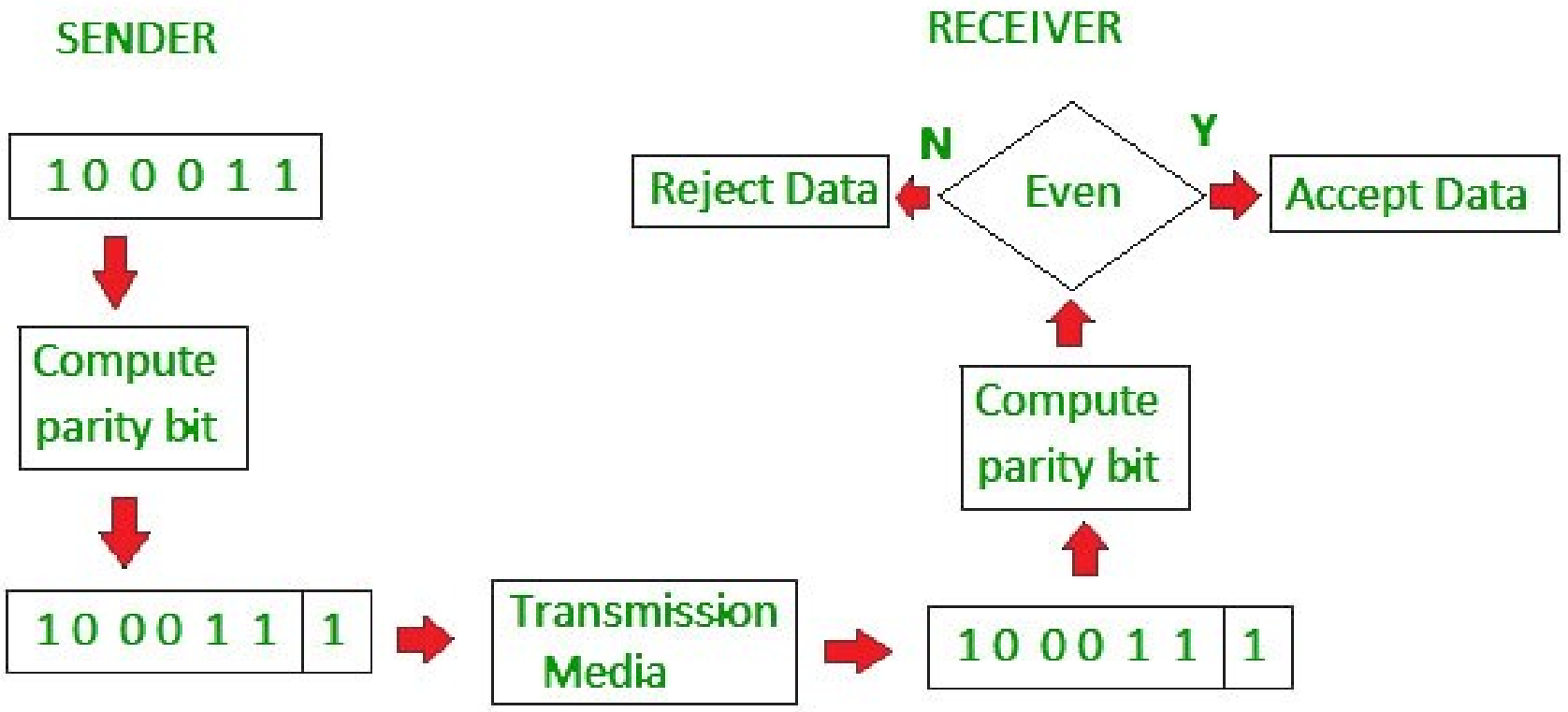
The parity check is done by adding an extra bit, called parity bit to the data to make a number of 1s either even in case of even parity or odd in case of odd parity.

- **In case of even parity:** If a number of 1s is even then parity bit value is 0. If the number of 1s is odd then parity bit value is 1.
- **In case of odd parity:** If a number of 1s is odd then parity bit value is 0. If a number of 1s is even then parity bit value is 1.

The parity check is suitable for **single bit error detection** only.

3.2 Error Detection and Correction

Even parity



Checksum

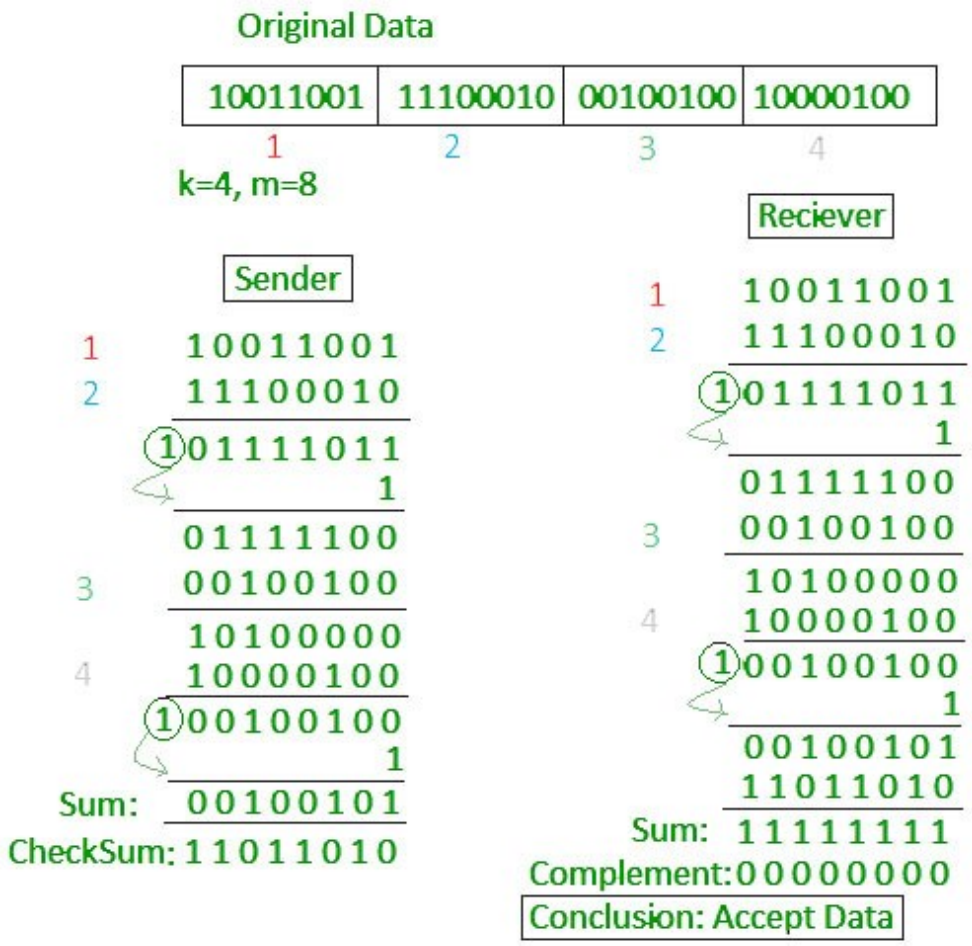
•The data is **divided into k segments** each of m bits.

•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.



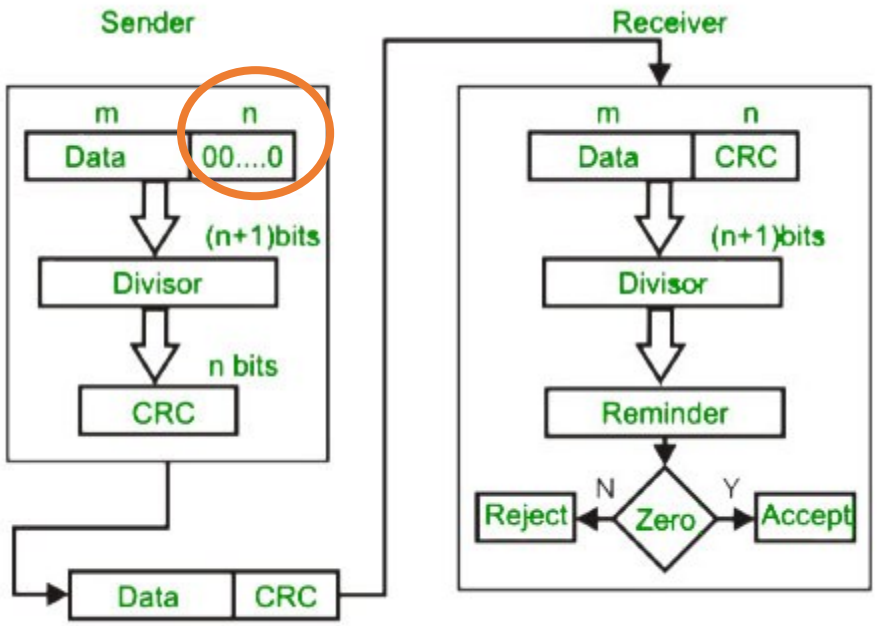
Cyclic redundancy check (CRC)

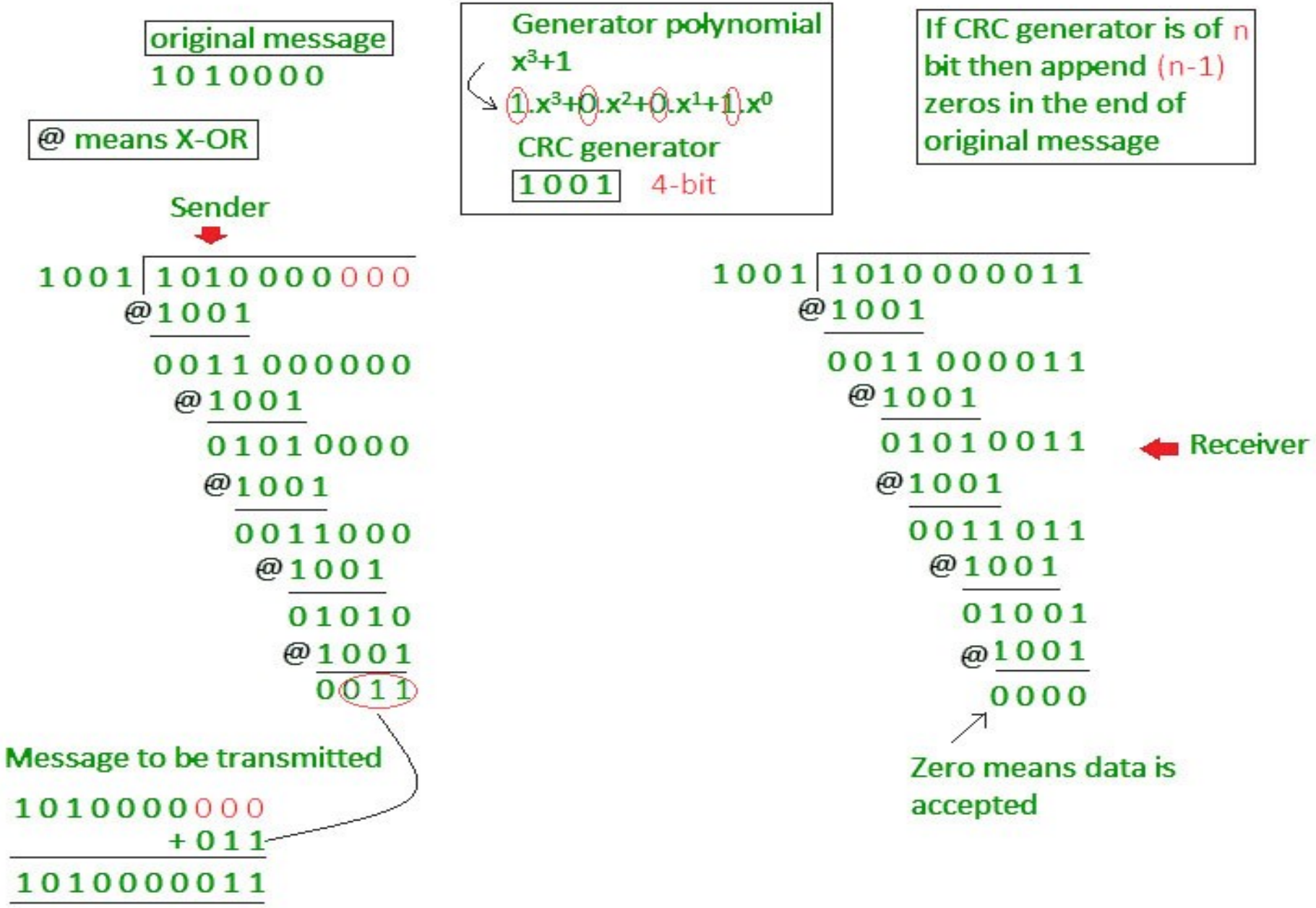
•Unlike checksum scheme, which is based on addition, **CRC is based on binary division**.

•In CRC, **a sequence of redundant bits**, called cyclic redundancy check bits, are **appended** to the end of data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number.

•At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted.

•A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.




original message
1 0 1 0 0 0 0
@ means X-OR
Generator polynomial
x^3+1
$1.x^3+0.x^2+0.x^1+1.x^0$
CRC generator
1 0 0 1 4-bit
If CRC generator is of n bit then append (n-1) zeros in the end of original message
Sender
1 0 0 1 | 1 0 1 0 0 0 0 0 0 0
@1 0 0 1
0 0 1 1 0 0 0 0 0 0
@1 0 0 1
0 1 0 1 0 0 0 0
@1 0 0 1
0 0 1 1 0 0 0
@1 0 0 1
0 1 0 1 0
@1 0 0 1
0 0 1 1
Message to be transmitted
1 0 1 0 0 0 0 0 0 0
+ 0 1 1
1 0 1 0 0 0 0 0 1 1
1 0 0 1 | 1 0 1 0 0 0 0 0 1 1
@1 0 0 1
0 0 1 1 0 0 0 0 0 1 1
@1 0 0 1
0 1 0 1 0 0 1 1
@1 0 0 1
0 0 1 1 0 1 1
@1 0 0 1
0 1 0 0 1
@1 0 0 1
0 0 0 0
Receiver
Zero means data is accepted

3.2 Error Detection and Correction

Hamming Codes, 1950

- **Codeword**, n bits = m message (data) bits + r redundant (check) bits
- A ***code*** is a set of codewords; consider a code with only four valid codewords:
  0000000000, 0000011111, 1111100000, and 1111111111
- **Hamming distance**

  count of corresponding bits that differ

  e.g., 10001001 and 10110001 have distance 3

```
    10001001
Xor 10110001
------------
    00111000
```

  if distance is d, d single bit errors are needed to convert one to the other

Hamming Codes Definitions

- The Hamming distance for a set of code words is the ***minimum* Hamming distance** for any two codewords in the code. For these 4 codewords, what is the hamming distance?

  0000000000, 0000011111, 1111100000, and 1111111111

- To detect d single bit errors, the code must have distance $d+1$

- To correct d single bit errors, the code must have distance $2d+1$, to ensure that the original codeword is closer than any other

Hamming Codes

- To correct d single bit errors, the code must have distance $2d+1$, to ensure that the original codeword is closer than any other

0000000000, 0000011111, 1111100000, and 1111111111

But the arrived codeword is 0000000111

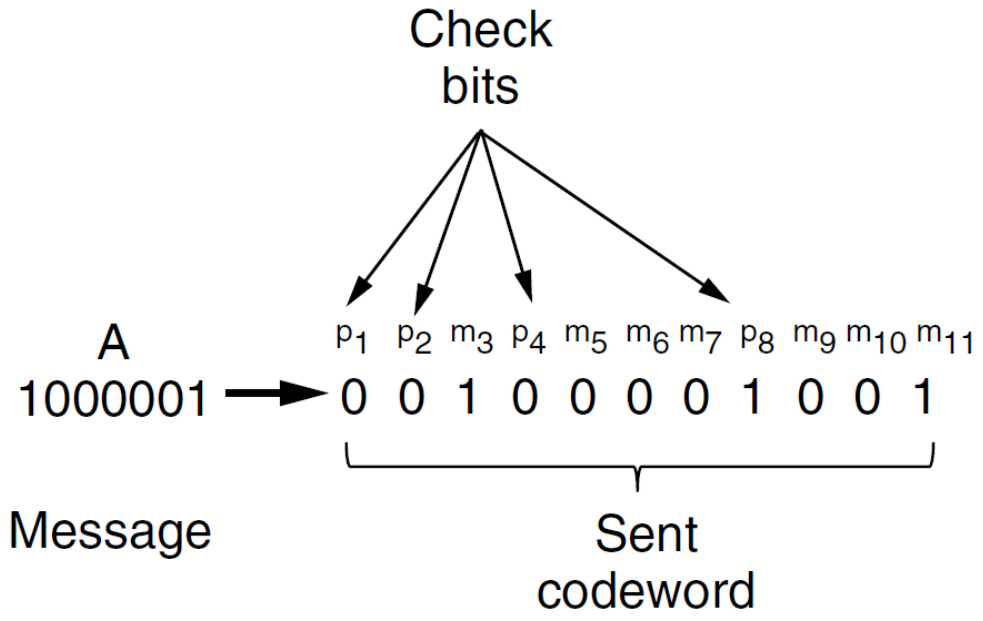
Here, Hamming distance: 5 (value of d=2)
So double bit error can be corrected. So, the original must have been 0000011111

Hamming Codes

Attaining a code with minimal n

- Bits 1, 2, 4, 8, 16, . . . are used as parity bits
- Bits 3, 5, 6, 7, 9, 10, 11, . . . are used for the message
- For $k \quad \sum c_i 2^i$, bit is position k is checked by bit in positions 2^i where $c_i \neq 0$

Hamming Codes (5 of 8)



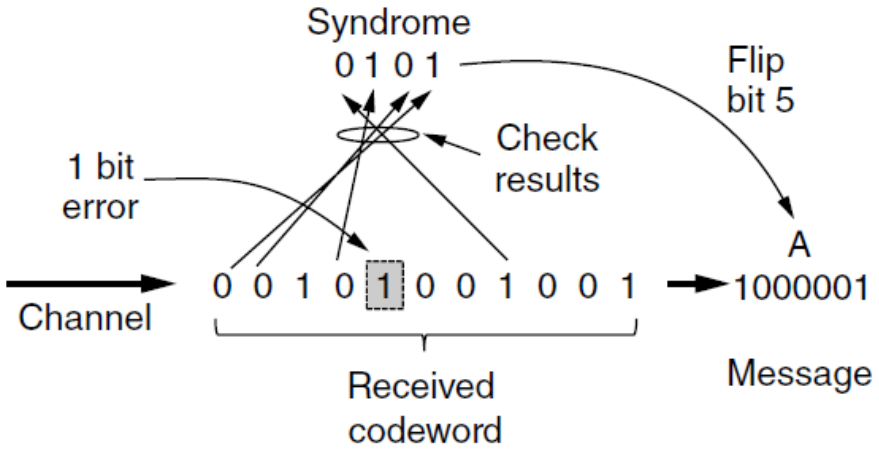
1	2	3	4	5	6	7	8	9	10	11
		1		0	0	0		0	0	1

Position	R4	R3	R2	R1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

P1: 10001(p1 will be 0 to make it even)
P2: 10001(p2 will be 0 to make it even parity)
P4: 000(p4 =0)
P8: 001(p8=1)

R1 -> 1,3,5,7,9,11
R2 -> 2,3,6,7,10,11
R3 -> 4,5,6,7
R4 -> 8,9,10,11

Hamming Codes



Position	R4	R3	R2	R1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1

p1	p2	m3	p4	**m5**	m6	m7	p8	m9	m10	m11
0	0	1	0	1	0	0	1	0	0	1

Similar Even parity calculation to **find the position:**
R1: 011001(R1 should be1 to make it even)
R2: 10001(R2 should be 0 to make it even)
R3: 0100(R3 should be 1)
R4: 1001(R4 should be 0)

R1 -> 1,3,5,7,9,11
R2 -> 2,3,6,7,10,11
R3 -> 4,5,6,7
R4 -> 8,9,10,11

Error found in 0101 position

**R1,R2,R3 represent the redundant bits, it can also be known as parity bit

Sender and Receiver have agreed upon Hamming code for error correction. Now at a certain time receiver has received the following message.

101101010110

Find out the position in which the error occurred if any error is detected.

P1	p2	m3	p4	**m5**	m6	m7	p8	m9	m10	m11	m12
1	0	1	1	0	1	0	1	0	1	1	0

Position	R4	R3	R2	R1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0

Similar Even parity calculation to **find if any error has been occurred or not:**

P1: 110001(P_1 will be 1 to make it even, Error detected)

P2: 011011(P_2 will be 0 to make it even parity, No error)

P4: 1010(P_4 =0, No error)

P8: 1011(P_8 =1, Error detected)

P1/R1→ 1,3,5,7,9,11

P2/R2→ 2,3,6,7,10,11

P4/R3→ 4,5,6,7,12

P8/R4→ 8,9,10,11,12

If the Calculated Parity or check bit is '1', error is detected.

Now lets find out, where the error is. Error found at $P_8\ P_4\ P_2\ P_1 = 1001 = 9^{th}\ position$

Sender and Receiver have agreed upon Hamming code for error correction. Now at a certain time receiver has received the following message.

101101010110

Find out the position in which the error occurred if any error is detected.

1	2	3	4	**5**	6	7	8	9	10	11	12
1	0	1	1	0	1	0	1	0	1	1	0

Similar Even parity calculation to **find the position:**

R1: $R_1$10001(R1 will be 0 to make it even)

R2: R_2 11011(R2 will be 0 to make it even parity)

R4: R_4 010(R_4 =1)

R8: 011(R_8 =0)

P1/R1→ 1,3,5,7,9,11
P2/R2→ 2,3,6,7,10,11
P4/R3→ 4,5,6,7,12
P8/R4→ 8,9,10,11,12

Error found in 0100 position

Similar Even parity calculation to **find the position:**

R1: $R_1$10001(R1 will be 0 to make it even)

R2: R_2 11011(R2 will be 0 to make it even parity)

R4: R_4 010(R_4 =1)

R8: 011(R_8 =0)

3.3 Elementary Protocols

- Utopia
- Simplex Stop-and-Wait
- Positive Acknowledgement with Retransmission

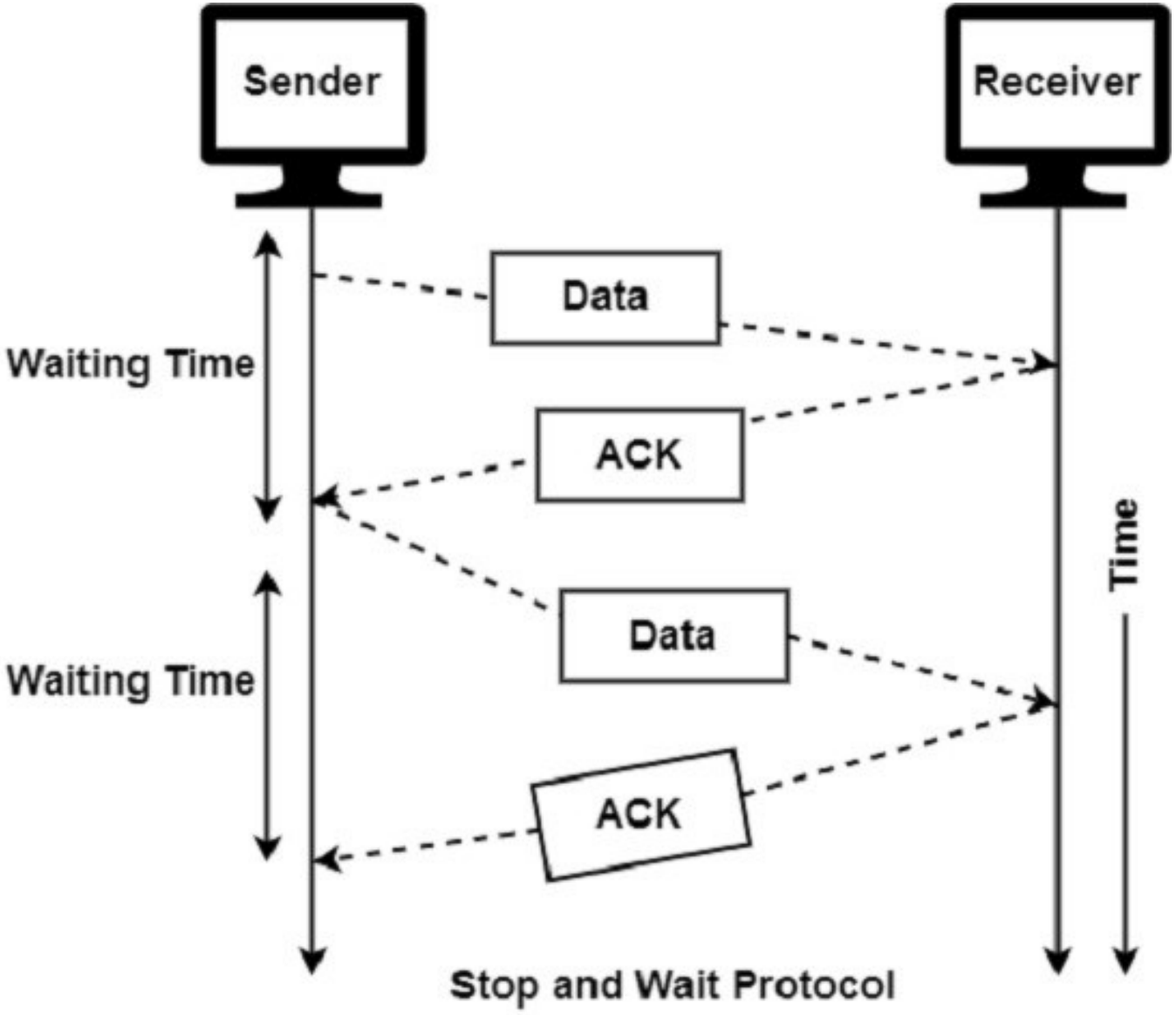
Utopia

```
void sender1( void )
{
  frame s;
 packet buffer;

   while( true ) {
   from_net_layer( &buffer );
   s.info = buffer;
   to_phy_layer( &s );
 }
}
```

```
• void receiver1( void )
• {
 • frame r;
   event_type event;

   • while( true ) {
     wait_for_event( &event );
     from_phys_layer( &r );
     to_net_layer( &r.info );
 • }
}
```

1 1 1 0

Stop-and-Wait



1 1 1 0

Stop-and-Wait

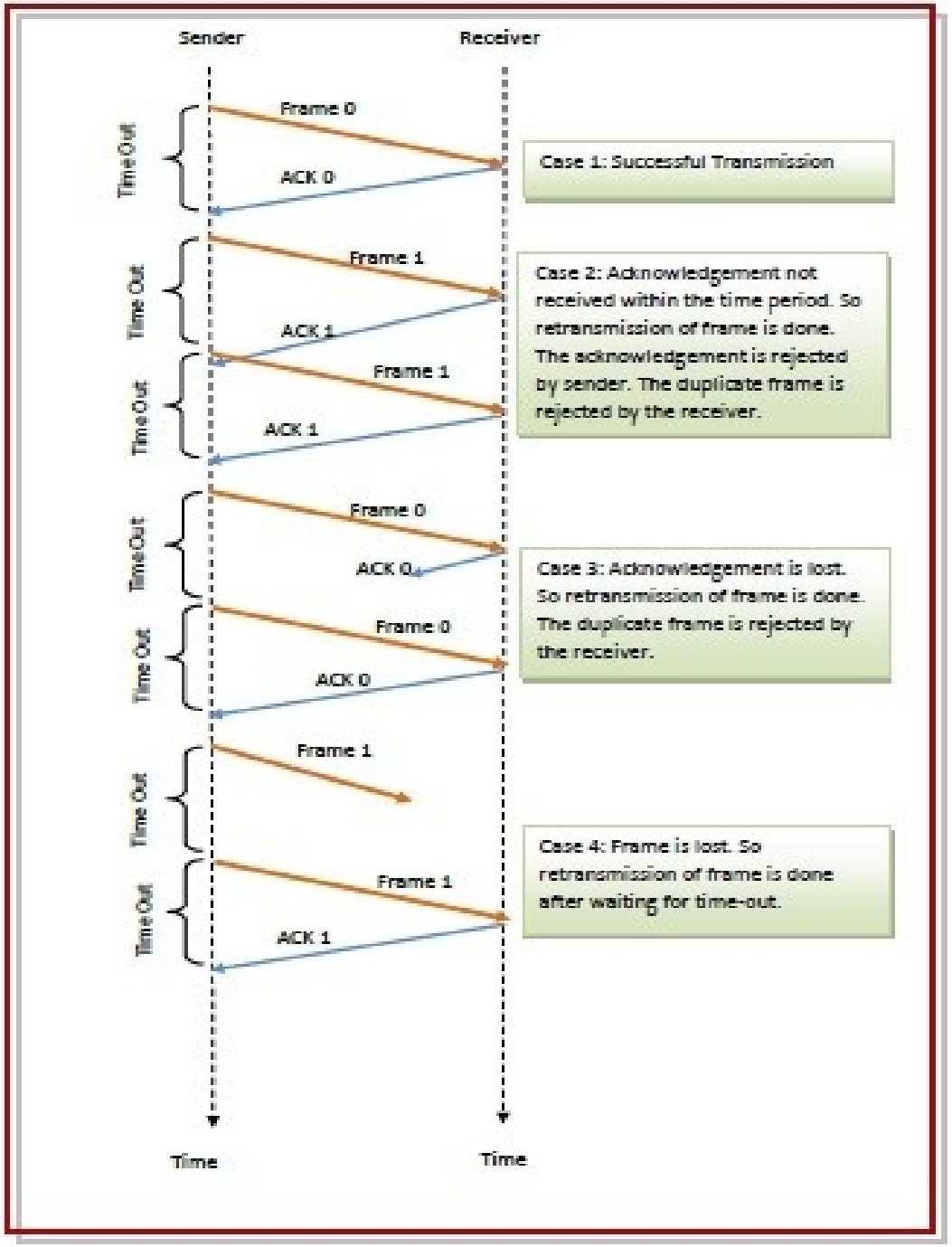
- **void** sender2(**void**)
- {
  - frame s;
    packet buffer;
  - event_type event;
    - **while**(**true**) {
      from_net_layer(&buffer);
      s.info = buffer;
      to_phy_layer(&s);
      wait_for_event(&event);
  - }
- }

- **void** receiver2(**void**)
- {
  - frame r,s;
    event_type event;
    - **while**(**true**) {
      wait_for_event(&event);
      from_phys_layer(&r);
      to_net_layer(&r.info);
      to_phy_layer(&s);

}• }

1 1 1 0

Stop-and-Wait Noisy Channel



Positive Acknowledgement with Retransmission

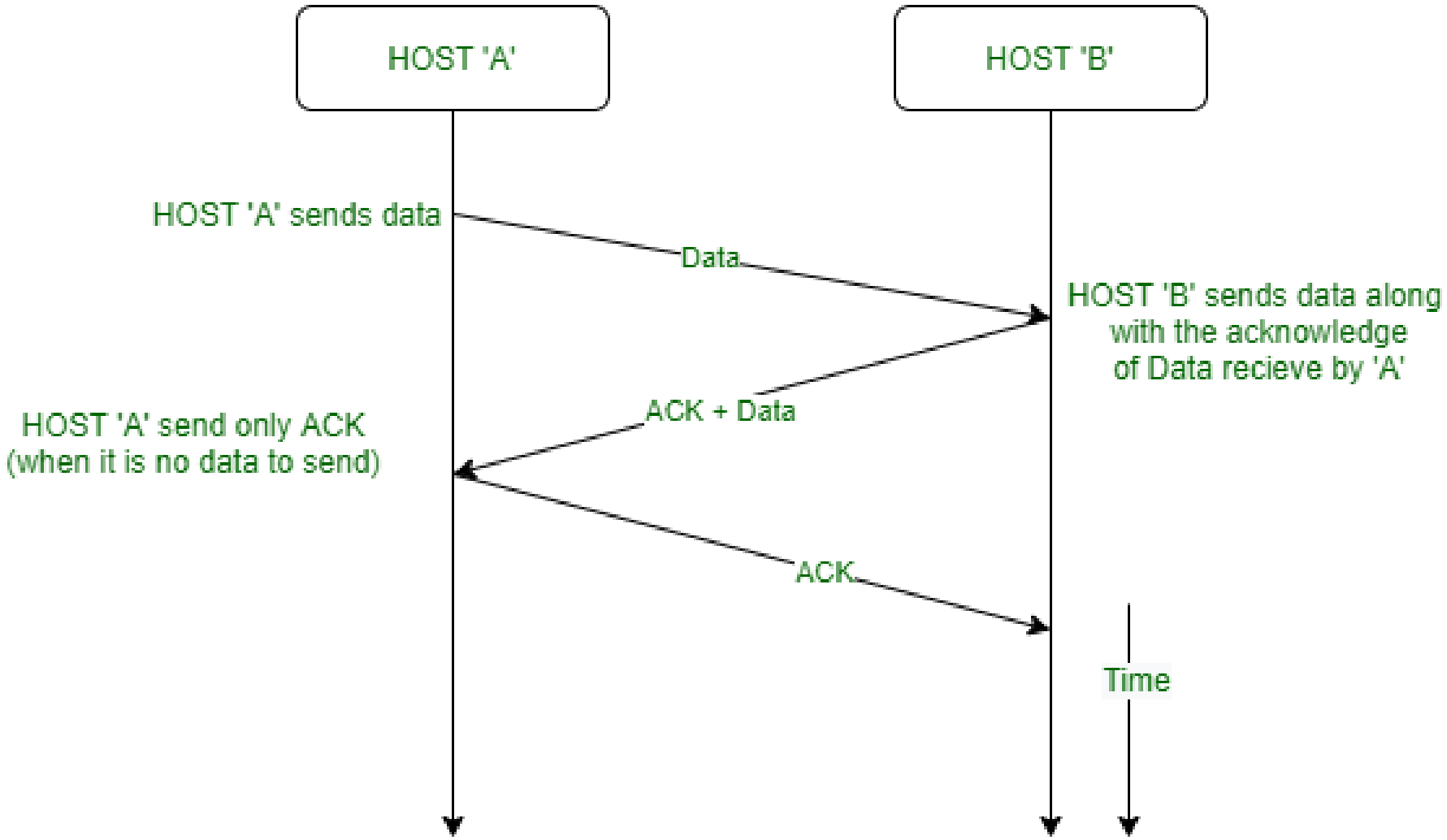
```
void sender3( void )
{
   seq_nr next_frame_to_send;
   frame s;
   packet buffer;
   event_type event;

   next_frame_to_send = 0;
   from_net_layer( &buffer );
   while( true ) {
      s.info = buffer;
      s.seq = next_frame_to_send;
      to_phy_layer( &s );
      set_timer( s.seq );
      wait_for_event( &event );
      if( event == frame_arrival ) {
         from_phys_layer( &s );
         if( s.ack == next_frame_to_send ) {
            stop_timer( s.ack );
            from_net_layer( &buffer );
            inc( next_frame_to_send );
         }
      }
   }
}
```

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

   frame_expected = 0;
   while( true ) {
      wait_for_event( &event );
      if( event == frame_arrival ) {
         from_phys_layer( &r );
         if( r.seq == frame_expected ) {
            to_net_layer( &r.info );
            inc( frame_expected );
         }
         s.ack = 1 – frame_expected;
         to_phy_layer( &s );
      }
   }
}
```

**3.4 Sliding Window Protocols

- Piggybacking
- One-Bit Sliding Window
- Go Back N
- Selective Repeat

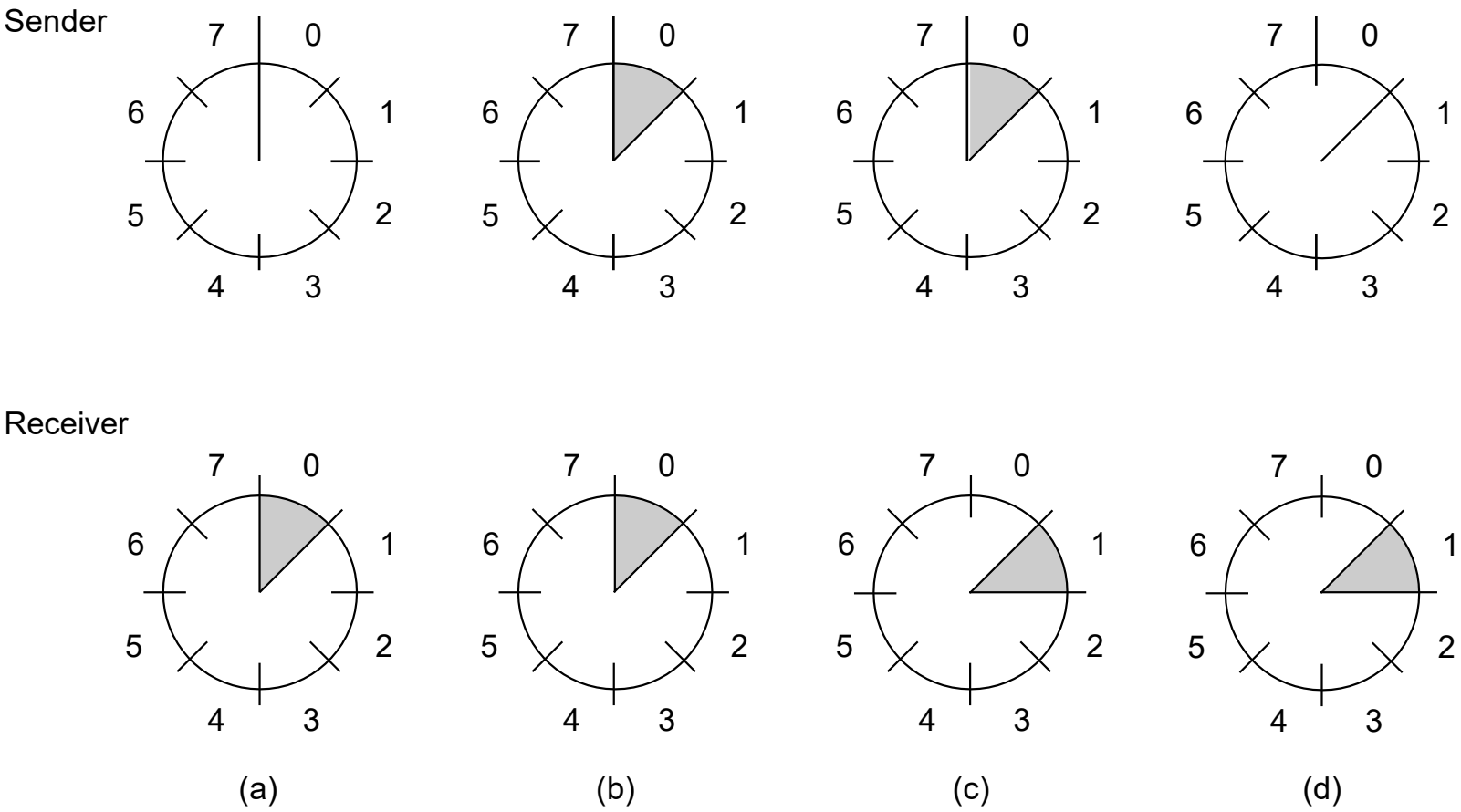
Piggybacking



Piggybacking

- Wasted bandwidth on ACK channel
- Interleave ACKs in reverse data channel (it's probably there)
- Piggybacking–add ACKs in the header of outgoing data packets
- Issue: how long to wait for an outgoing data packet

Sliding Window of Size 1



(a) initial; (b) after first frame sent; (c) first frame received; (d) first frame acknowledged

One-Bit Sliding Window Initialization

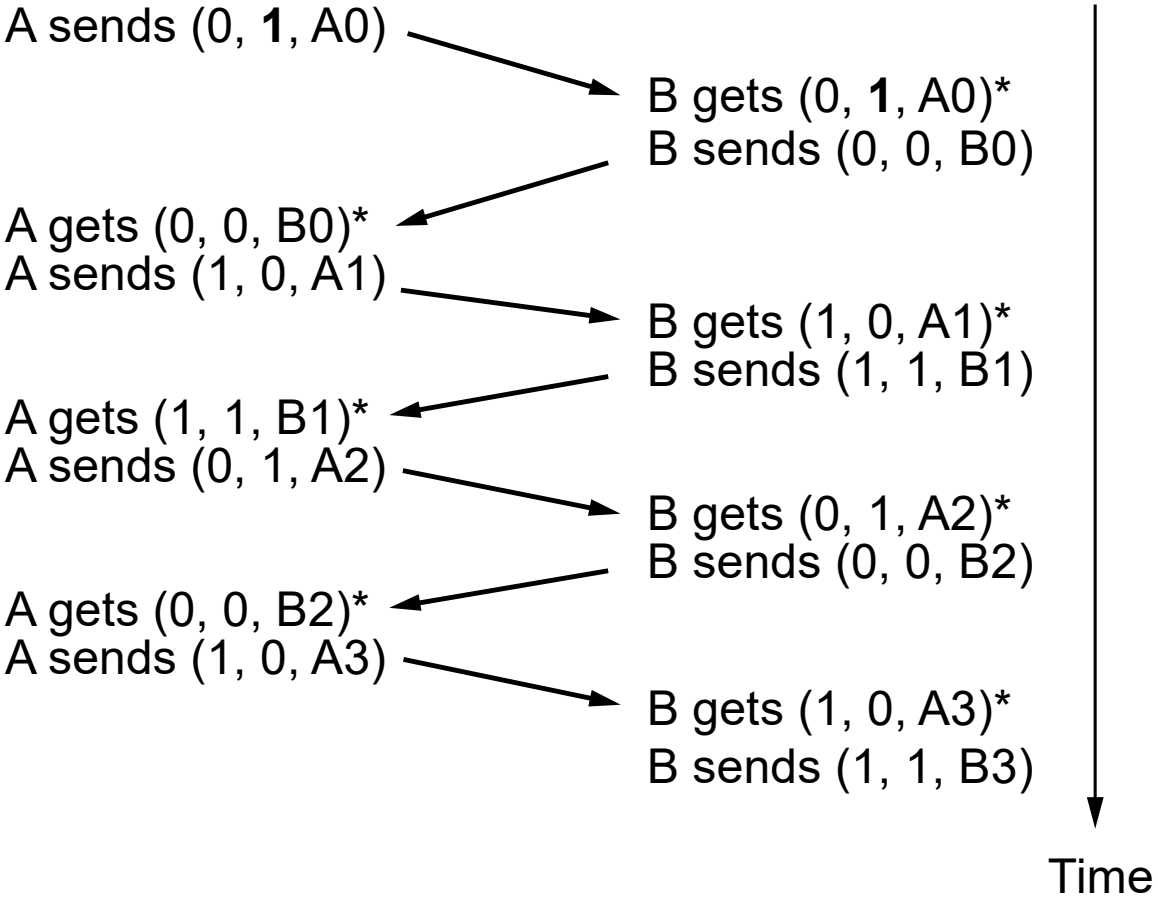
```
void protocol4( void )
{
   seq_nr next_frame_to_send, frame_expected;
   frame r, s;
   packet buffer;
   event_type event;

   next_frame_to_send = 0;
   frame_expected = 0;
   from_net_layer( &buffer );
   s.info = buffer;
   s.seq = next_frame_to_send;
   s.ack = 1 - frame_expected;
   to_phys_layer( &s );
   start_timer( s.seq );
```

One-Bit Sliding Window Main Loop

```
while( true ) {
   wait_for_event( &event );
   if( event == frame_arrival ) {
      from_physical_layer( &r );
      if( r.seq == frame_expected ) {
         to_net_layer( &r.info ); inc( frame_expected );
      }
      if( r.ack == next_frame_to_send ) {
         stop_timer( r.ack );
         from_net_layer( &buffer ); inc( next_frame_to_send );
      }
   }
   s.info = buffer;
   s.seq = next_frame_to_send;
   s.ack = 1 - frame_expected;
   to_phys_layer( &s ); start_timer( s.seq );
}
```

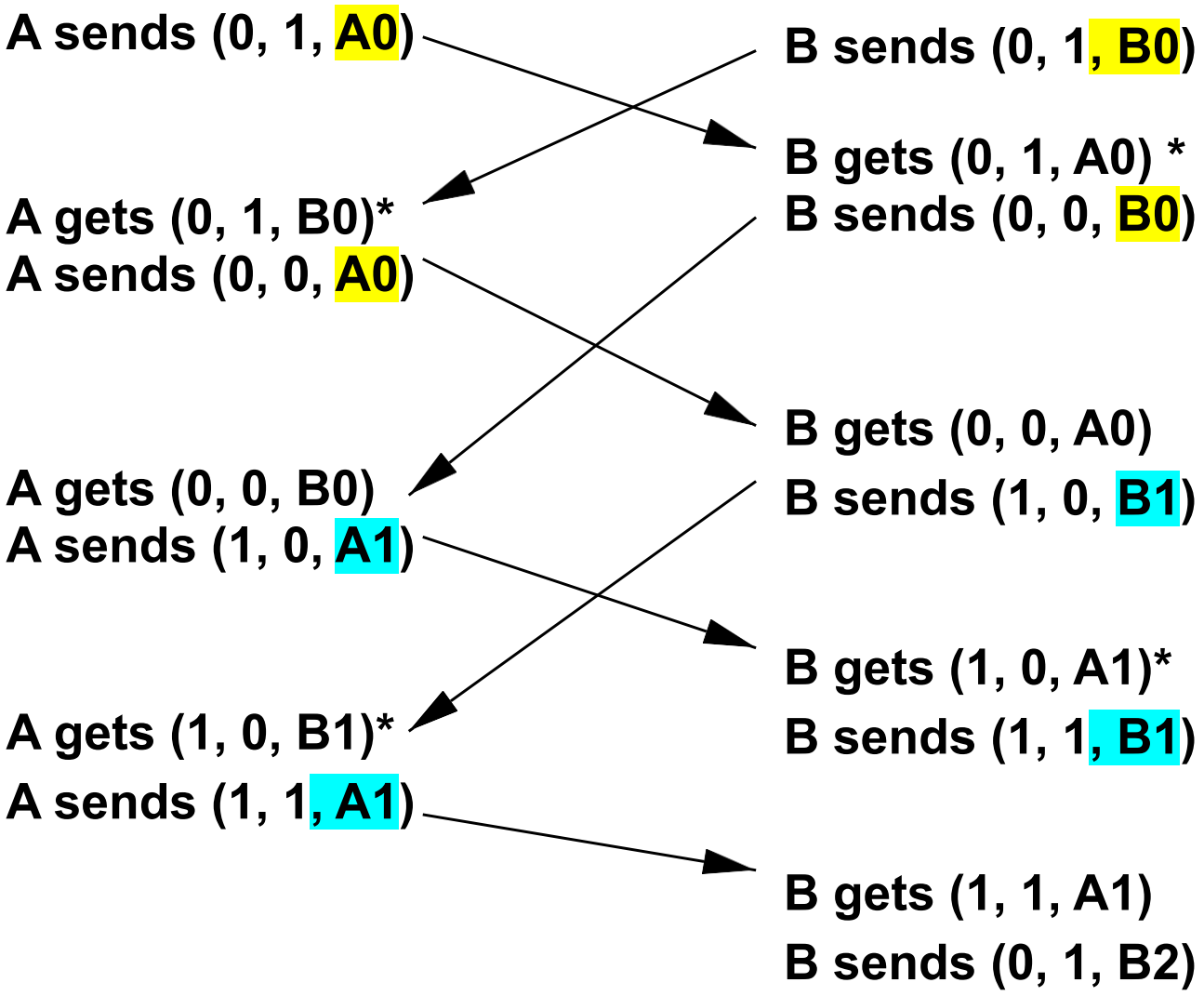
One-Bit Sliding Window Normal Case

Notation: (sequence, **ack**, packet)



An asterisk indicates where a network layer accepts a packet.

One-Bit Sliding Window Degenerate Case



Half of the frames contain duplicates, even though there are no transmission errors.

Pipelining

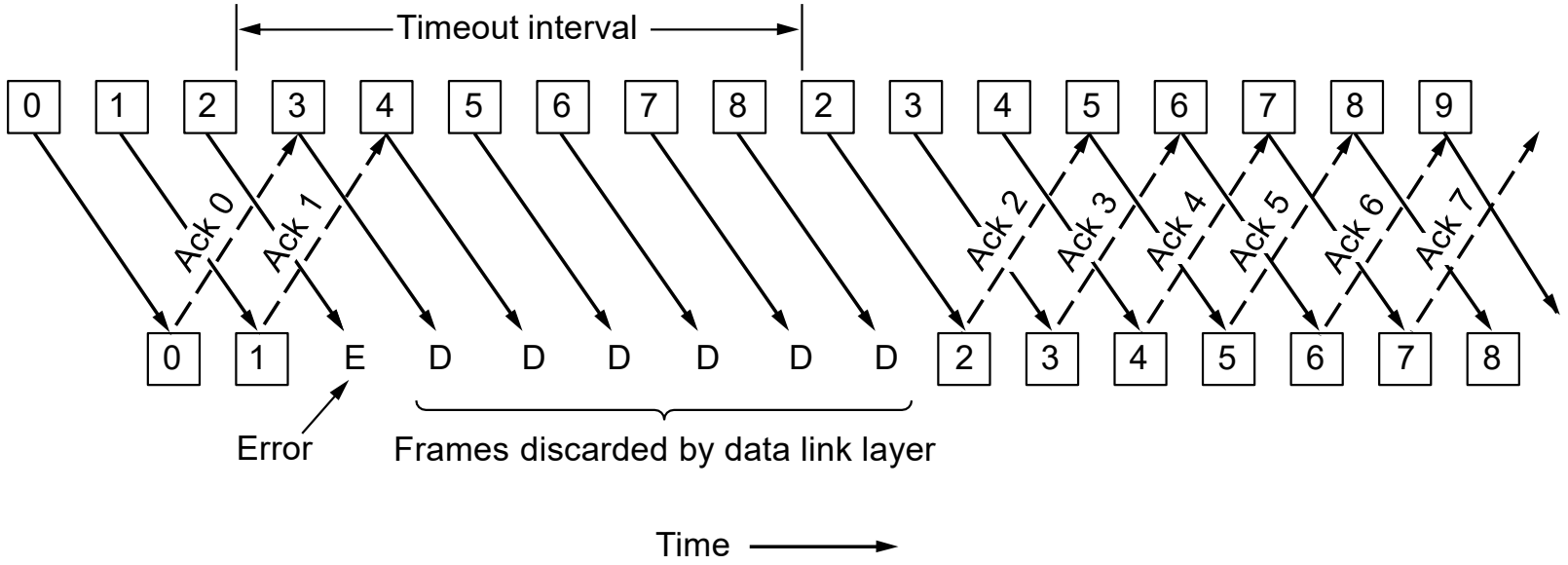
The technique of keeping multiple frames in flight is an example of **pipelining**.

- Want to avoid stalls in transmission
  - in one-bit sliding window, we wait a full round trip between packet sends
- Make sender window large when:
  - bandwidth is high or
  - the round-trip time is high
  - bandwidth x round-trip time
- line utilization = $l/(l + bR)$
  - l is the frame size (in bits)
  - b is the channel capacity (in bits/sec)
  - R is the round-trip time (in sec)
- If $l<bR$ the efficiency is less than 50%

Pipelining frames over an unreliable communication channel raises some serious issues.

- First, what happens if a frame in the middle of a long stream is damaged or lost?
  - Large numbers of succeeding frames will arrive at the receiver before the sender even finds out that anything is wrong.

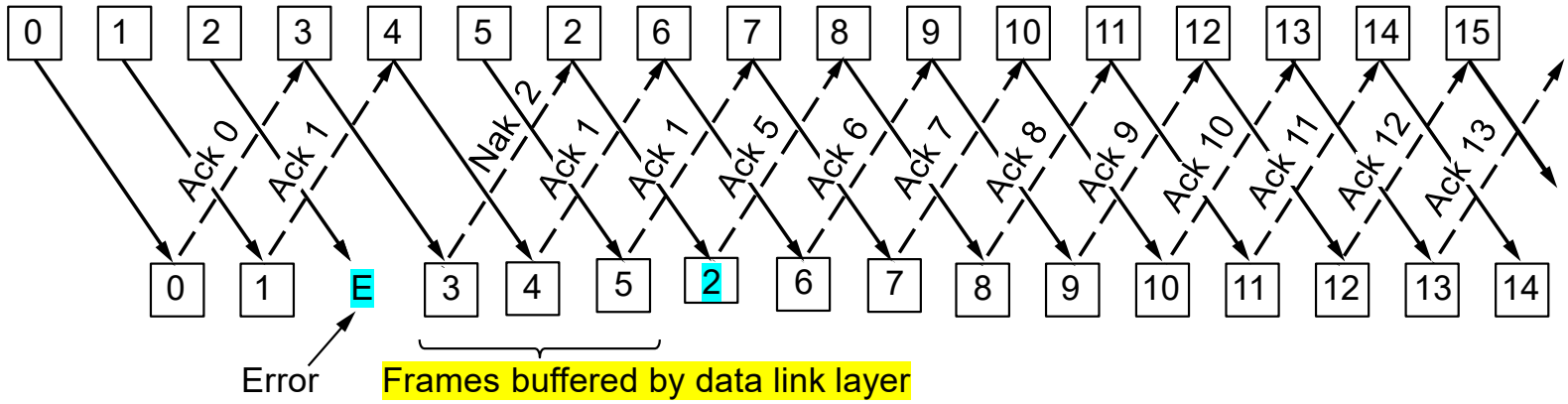
Receiver Window Size is One



Leads to Go-back *n*

Receiver's Window is Large

Dealing with Error



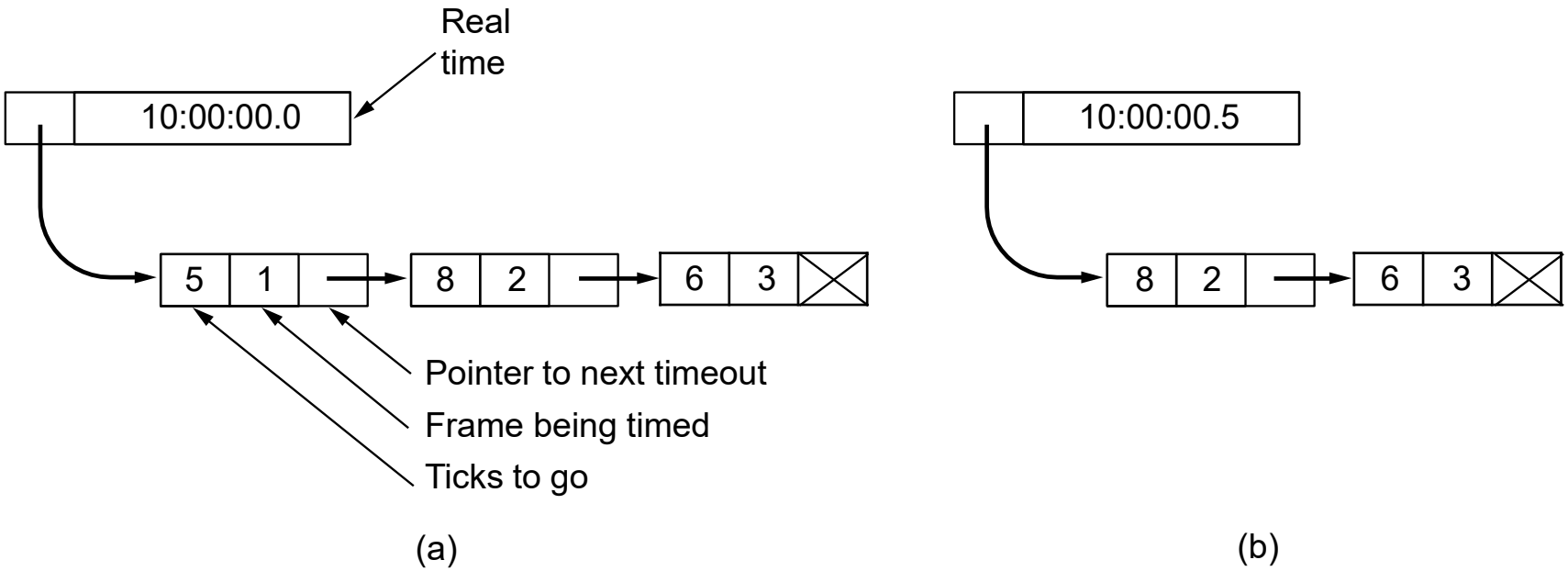
Leads to selective repeat (So in Selective Repeat Receiver window is larger than one)

- tradeoff bandwidth and buffer capacity

Go Back n Off by one

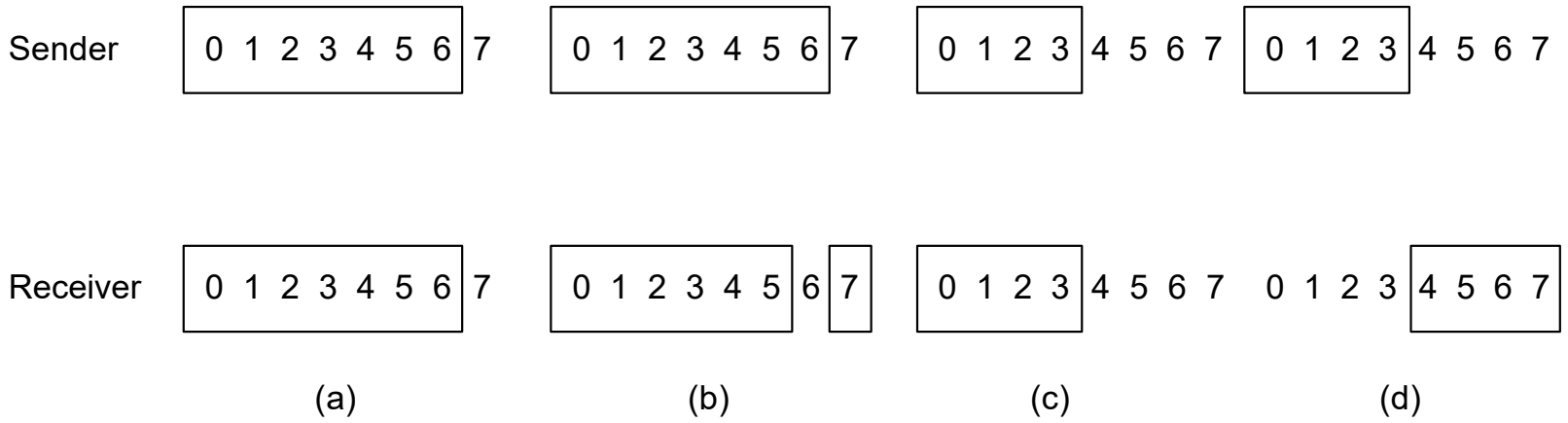
- Can only have MAX_SEQ Frames in flight (not MAX_SEQ + 1)
- Consider MAX_SEQ = 7
  - Sender sends frames 0 through 7
  - Receiver acknowledges frame 7
  - Sender sends the next 8 frames with sequence numbers 0 though 7
  - Another acknowledgement for frame 7 arrives
  - Is this an acknowledgement for the first group or the second?

Managing Multiple Timers



Problem for Selective Repeat

Do not allow sender to overlap sequence numbers in consecutive windows



(a) initial situation window size is 7; (b) 7 frames sent, received, but not acknowledged;
(c) initial situation, widow size is 4; (d) all frames sent, received, but not acknowledged