Introduction to LAN/WAN

Data Link Layer

Topics

- Introduction
- Errors
- Protocols
- Modeling
- Examples

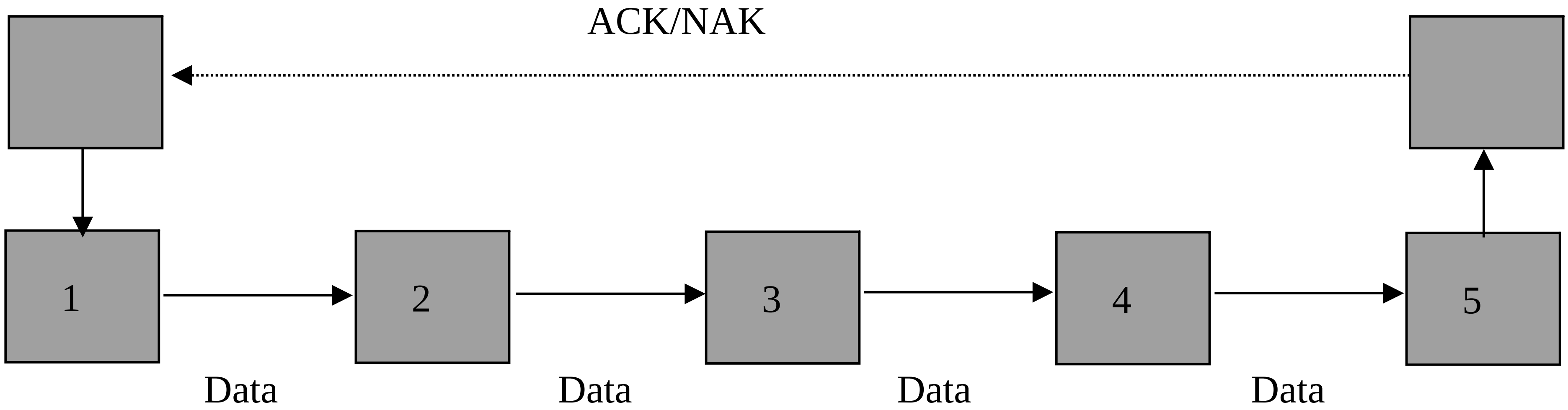
Introduction

- ☞ Reliable, efficient communication between two adjacent machines
- ☞ Machine A puts bits on wire, B takes them off. Trivial, right? Wrong!
- ☞ Challenges:
  - – Circuits make errors
  - – Finite data rate
  - – Propagation delay
- ☞ Protocols must deal!

Data Link Layer Functions

- Provides a *well-defined service interface* to the network layer.
- Determines how the bits of the physical layer are grouped into frames *(framing)*.
- Deals with transmission errors *(CRC and ARQ)*.
- Flow control: regulates the flow of frames.
- Performs general link layer management. (seq #, protocols, etc)

End to End



Hop by Hop

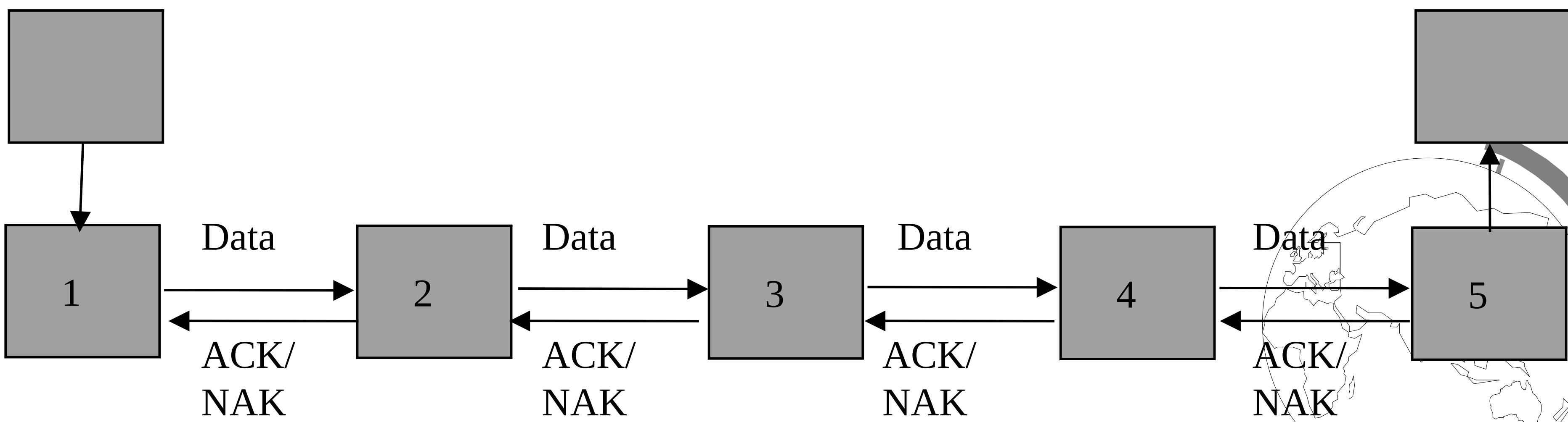


Figure 5.7

Data Link Services

- Network layer has bits
- Says to data link layer:
  - "send these to this other network layer"
- Data link layer sends bits to other data link layer
- Other data link layer passes them up to network layer

Data Link Services

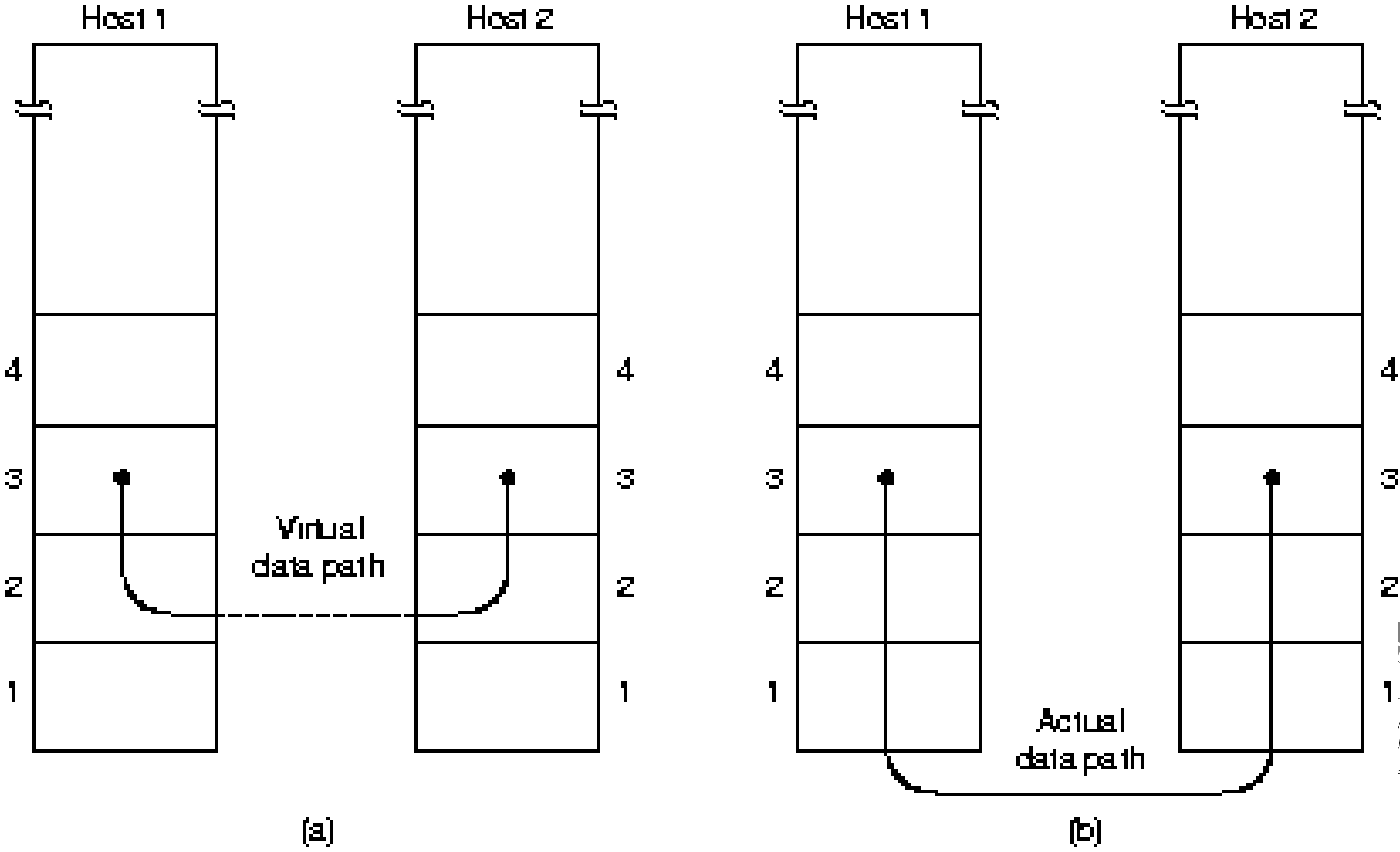
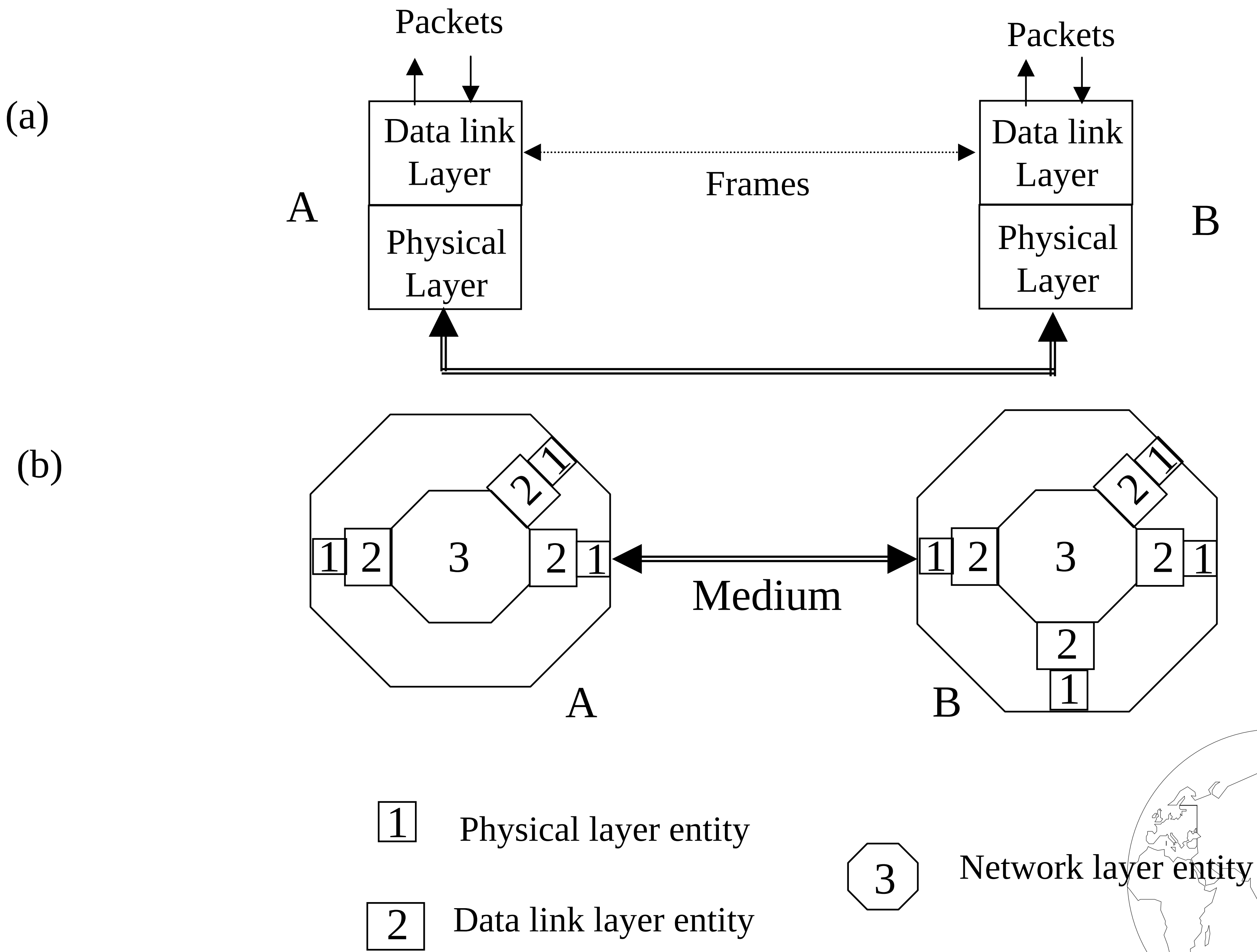


Fig. 3-1. (a) Virtual communication. (b) Actual communication.



(a)
Packets
Packets
A
Data link Layer
Physical Layer
Frames
B
Data link Layer
Physical Layer
(b)
3
2
1
2
1
2
1
Medium
3
2
1
2
1
2
1
2
1
A
B
1 Physical layer entity
2 Data link layer entity
3 Network layer entity

Figure 5.2

Types of Services Possible

- Unacknowledged connectionless (best effort)
  - No acknowledgements
  - No logical connection beforehand
  - Frame lost, no detection or recovery
  - Why would you want this service?
    - When loss infrequent, easy for upper layer to recover
    - "Better never than late" (real-time traffic)
- Acknowledged connectionless service
  - Still no connection
  - Packets acknowledged
  - Why would you want this service?
    - Unreliable channel (wireless)

Types of Services Possible

- Acknowledged connection-oriented service
  - – Connection is set up
  - – All frames are numbered
  - – Data link guarantees:
    - ◆ All frames sent are received
    - ◆ No duplicates
    - ◆ Frames received in order
    - ◆ Network layers sees equivalent of reliable bit stream

Framing

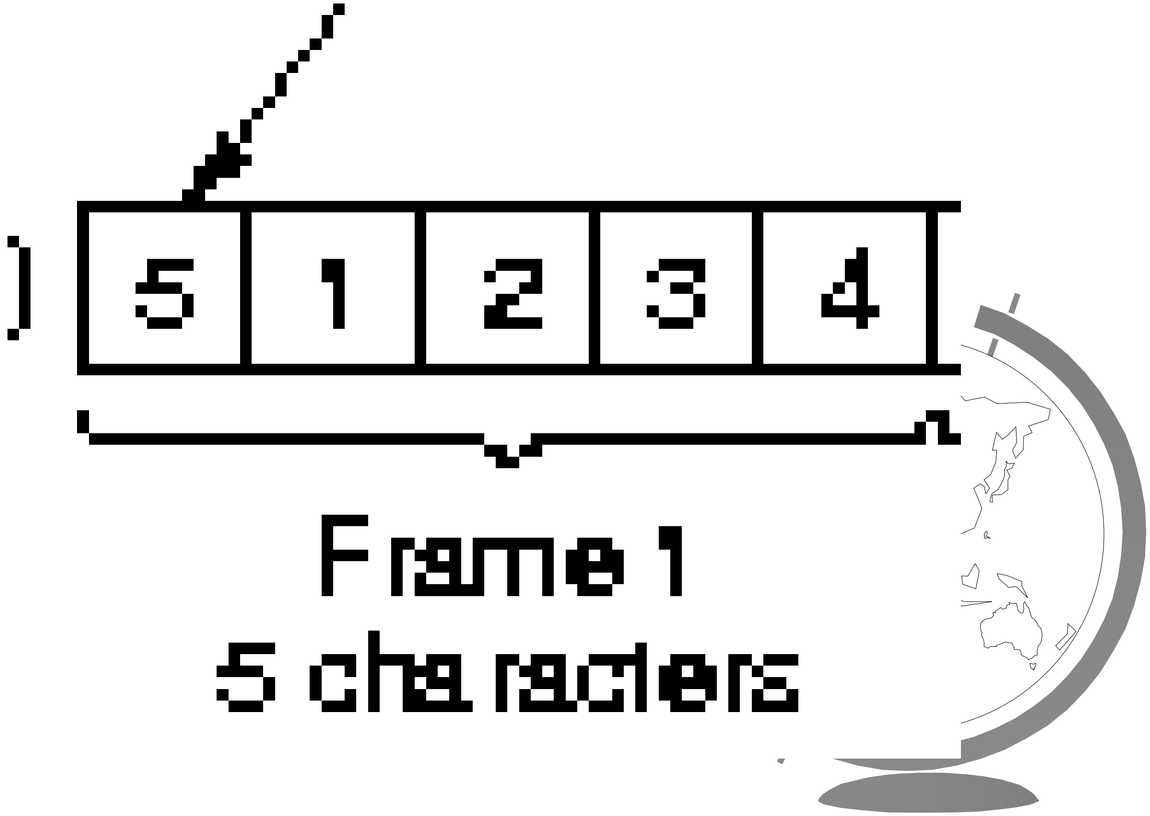
- ☞ Data link breaks physical layer stream of bits into *frames*

      ...0101101001010011010100 10...

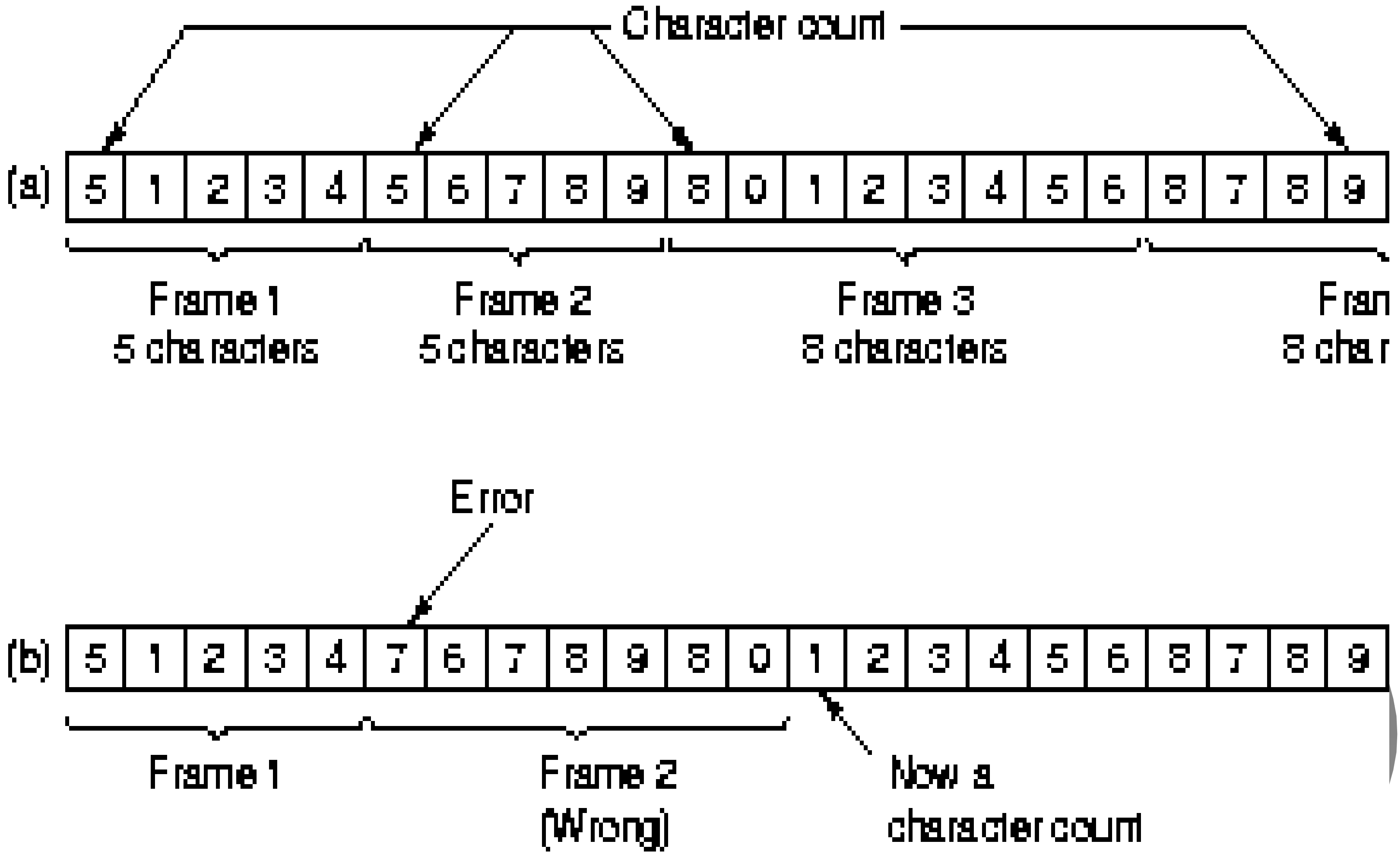
- ☞ Varying propagation delays: can't count on timing
- ☞ How does receiver detect boundaries?
  - – Length count
  - – Byte stuffing: special flag characters
  - – Bit stuffing
  - – Special physical layer encoding

Length count

- First field is length of frame
- Count until end
- Then, look for next frame
- Problems?

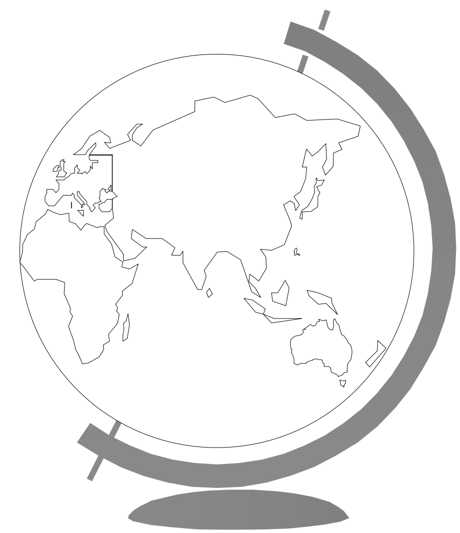


Length Count Problems



Byte Stuffing: Special Characters

- ☞ Reserved ASCII characters for framing delimiters (beginning and end)
- ☞ HDLC Example:
  - Beginning: DLE STX (Data-Link Escape, Start of TeXt)
  - End: DLE ETX (Data-Link Escape, End of TeXt)
- ☞ Problems?
- ☞ Solution?



Byte Stuffing

[HDLC Example]

- Prob 1: reserved character patterns occur within the "transparent" data.
- Prob. 1 Soln:
  - sender stuffs an extra DLE into the data stream just before each occurrence of an "accidental" DLE in the data stream.
  - The data link layer on the receiving end unstuffs the DLE before giving the data to the network layer.

HDLC Byte Stuffing

DLE	STX	*Transparent Data*	DLE	ETX

Before

DLE	STX	A	B	DLE	H	W	DLE	ETX

Stuffed

DLE	STX	A	B	DLE	DLE	H	W	DLE	ETX

Unstuffed

DLE	STX	A	B	DLE	H	W	DLE	ETX

Bit Stuffing

- Prob. 2: Not all architectures are character oriented: arbitrary-sized characters?
- Soln:
  - stuff at bit level (bit stuffing)
  - Each frame begins and ends with a special bit pattern called a flag byte [01111110].
  - What if flag bit pattern [01111110] occurs in data?
  - Soln: Whenever sender data link layer encounters 5 *consecutive 1's* in the data stream, it automatically stuffs a 0 bit into the outgoing stream.
  - When the receiver sees *5 consecutive incoming 1's followed by a 0 bit*, it automatically destuffs the 0 bit before sending the data to the network layer.
  - Problem? Wasted bandwidth/processing

Bit Stuffing

Input Stream

[redacted]

Stuffed Stream

[redacted]

[redacted]

Unstuffed Stream

[redacted]

Special PHY-Layer Encoding

- Send a signal that does not have legal representation
  - low to high means a 1
  - high to low means a 0
  - high to high means frame end
  - IEEE 802.4 (token bus)

- Lastly, 2 or more delimiting methods used
- Combination of above:
  - length plus frame boundary
  - IEEE 802.3 (ethernet)

Topics

- ☞ Introduction
- ☞ Framing
- ☞ Errors ←
  - why
  - detecting
  - correction
- ☞ Protocols
- ☞ Modeling **?**
- ☞ Examples **?**

Errors

- Trends
- Lines becoming digital
  - errors rare
- Copper the “last mile”
  - errors infrequent
- Wireless
  - errors common
- Errors are here for a while
- Plus, consecutive errors
  - bursts

Handling Errors

- Add redundancy to data
- Example:
  - “hello, world” is the data
  - “hzllo, world” received (detect? correct?)
  - “xello, world” received (detect? correct?)
  - “jello, world” received (detect? correct?)
  - what about similar analysis with “caterpillar”?
- Some: *error detection*
- More: *error correction (Forward Error Correction)*

What is an Error?

- Frame has *m* data bits, *r* redundancy bits
  - n = (m+r) bit *codeword*
- Given two codewords, compute distance:
  - `10001001`
  - `10110001`
  - `00111000`
  - XOR, 3 bits difference
  - *Hamming Distance*
- "So what?"

Code Hamming Distance

- ☞ Two codewords are *d* bits apart,
  - – then *d* errors are required to convert one to other
- ☞ *Code Hamming Distance* min distance between any two legal codewords

Error Detection using Parity Bit

- ☞ Single bit is appended to each data chunk
  - makes the total number of 1 bits even/odd
- ☞ Example: for even parity
  - `1000000(1)`
  - `1111101(0)`
  - `0000000(1)`
- ☞ What is the *Hamming distance*?
- ☞ How many bit errors can it detect?
- ☞ How many bit errors can it correct?

Hamming Distance Example

- Consider 8-bit code with 4 valid codewords:

  00000000 00001111 11110000 11111111

- What is the *Hamming distance*?
- What is the min bits needed to encode?
  - What are n, m, and r?
- What if 00001110 arrives?
- What if 00001100 arrives?

Ham On

- Consider a 10-bit code with 4 codewords:

  00000 00000 00000 11111 11111 00000 11111 11111
- *Hamming distance*?
- Correct how many bit errors?
  - 10111 00010 received, becomes 11111 00000 corrected
  - 11111 00000 sent, 00011 00000 received
- Might do better
  - 00111 00111 received, 11111 11111 corrected
  - and contains 4 single-bit errors

Fried Ham

- All possible data words are legal
- Choosing careful redundant bits can results in large Hamming distance
  - to be better able to detect/correct errors
- To detect *d* 1-bit errors requires having a Hamming Distance of at least *d+1* bits
  - Why?
- To correct *d* errors requires *2d+1* bits.
  - Why?

Designing Codewords

- Fewest number of bits needed for 1-bit errors?
  - `n=m+r` bits to correct all 1-bit errors
- Each message has *n* illegal codewords a distance of 1 from it
  - form codeword (n-bits)
  - invert each bit, one at a time
- Need *n+1* bits for each message
  - *n* that are one bit away and 1 for the message

Designing Codewords (cont)

- The total number of bit patterns = 2^n
  - So, $(n+1)\ 2^m \le 2^n$
  - So, $(m+r+1) \le (2^{m+r}) / 2^m$
  - Or, $(m+r+1) \le 2^r$
- Given *m*, have lower limit on the number of check bits required to detect (and correct!) 1-bit errors

Example

- 8 data bits, $m = 8$
- How many check bits required to detect and correct 1-bit errors?
- $(8 + r + 1) < 2^r$
  - Is 3 bits enough?
  - Is 5 bits enough?
- Use *Hamming code* to achieve lower limit

Hamming Code

- Bits are numbered left-to-right starting at 1
- Powers of two (1, 2, 4 ...) are check bits
- Check bits are parity bits for previous set
- Bit checked by only those check bits in the expansion
  - example: bit 19 expansion = 1 + 2 + 16
- Examine parity of each check bit, *k*
  - If not, add *k* to a *counter*
- If 0, no errors else *counter* gives bit to correct

Ham It Up

- Examples:
  - Check bit 1 covers bits 1, 3, 5 ...
  - Check bit 2 covers bits 2, 3, 6, 7, 10, 11 ...

Hamming Code and Burst Errors

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	11111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	00101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

Error Correction

- Expensive
  - example: 1000 bit message
  - Correct single errors? (10 check bits)
  - Detect single errors? (1 parity bit)
- Useful mostly:
  - simplex links (one-way)
  - long delay links (say, satellite)
  - links with very high error rates
    - would get garbled every time resent

Error Detection

- Most popular use *Polynomial Codes* or *Cyclic Redundancy Codes* (CRCs)
  - checksums
- Acknowledge correctly received frames
- Discard incorrect ones
  - may ask for retransmission
- Error correction Vs. detection, tradeoff between:
  - Number of redundant bits added
  - Packet retransmission overhead
  - Natural ecological niche for each technique depending on error rate

Polynomial Codes

- Bit string as polynomial w/0 and 1 coeffs
  - ex: k bit frame, then x^{k-1} to x^0
  - ex: `10001` is $1x^4+0x^3+0x^2+0x^1+1x^0 = x^4+x^0$
- Polynomial arithmetic mod 2

```
 10011011    11110000    00110011
+11001010   -10100110   +11001101
 --------    --------    --------
 01010001    01010110    11111110
```

- Long division same, except subtract as above
- “Ok, so how do I use this information?”

Doing CRC

- Sender + receiver agree *generator polynomial*
  - *G*(x), ahead of time, part of protocol
  - with low and high bits a ‘1’, say `1001`
- Compute checksum to frame (*m* bits)
  - *M*(x) + checksum to be evenly divisible by *G*(x)
- Receiver will divide by *G*(x)
  - If no remainder, frame is ok
  - If remainder then frame has error, so discard
- “But how do we compute the checksum?”

Computing Checksums

- Let r be *degree* of G(x)
  - *If* $G(x) = x^2+x^0 =$ `101`, then r is 2
- Append r zero bits to frame M(x)
  - get $x^rM(x)$
  - ex: `1001` + *`00`* = `100100`
- Divide $x^rM(x)$ by G(x) using mod 2 division
  - ex: `100100 / 101`
- Care about *remainder*
- "Huh? Do you have an example?"

Dividing $x^r M(x)$ by $G(x)$

```
          ____1011__
101 |  100100
       101
       ---
        011
        000
        ---
         110
         101
         ---
          110
          101
          ---
           11
```

← Remainder

“Ok, now what?”

Computing Checksum Frame

- ☞ Subtract (mod 2) remainder from x^r*M*(x)

```
100100
    11
------
100111
```

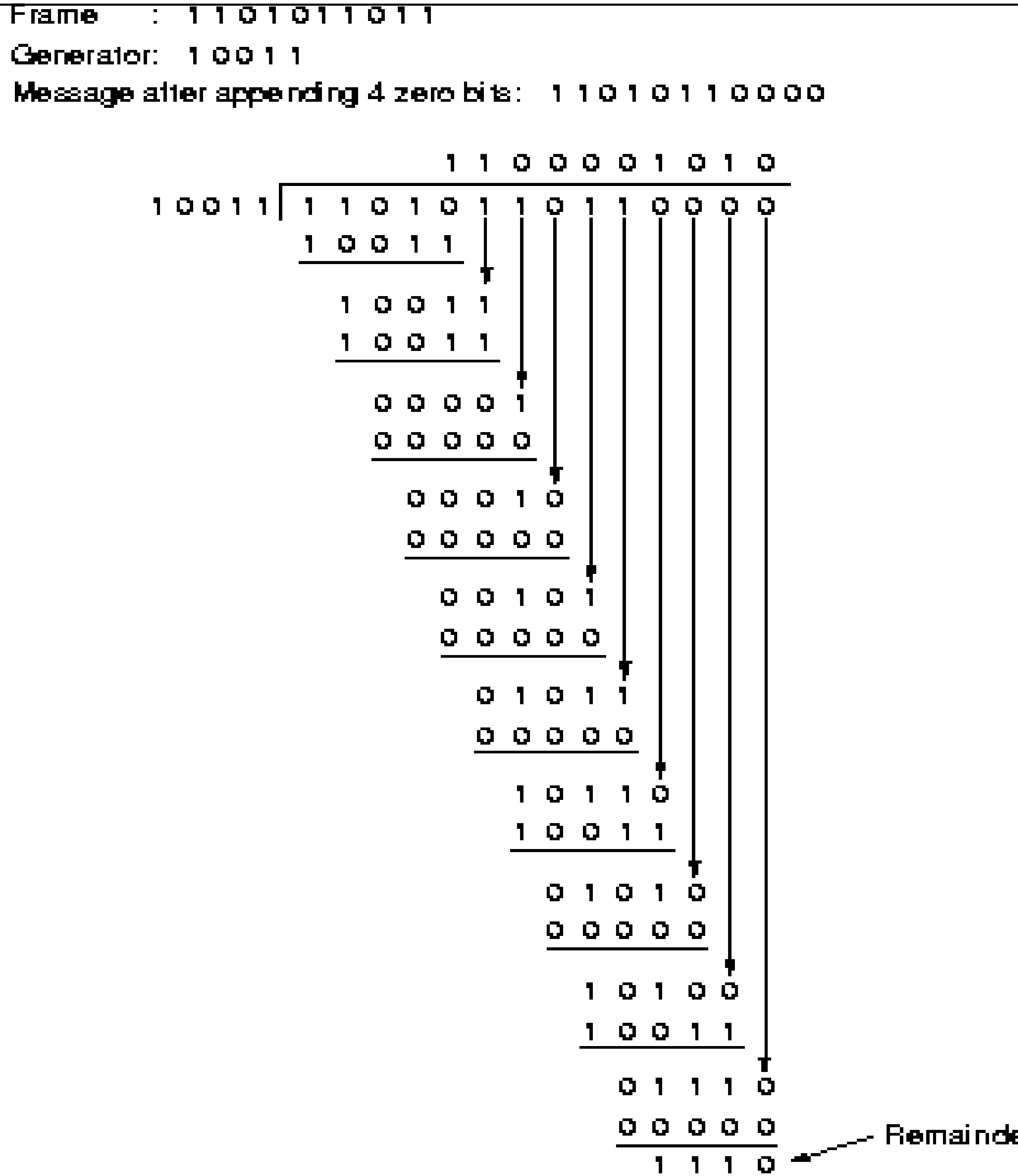
- ☞ Result is checksum frame to be transmitted
  - – *T*(x) `= 100111`
- ☞ What if we divide *T*(x) by *G*(x)?
  - – Comes out evenly, with no remainder
  - – Ex: 210,278 / 10,941 remainder 2399
  - – 210,279 - 2399 is divisible by 10,941
- ☞ "Cool!"

Let’s See if it Worked

```
          ____1011__
101 | 100111
      101
      ---
       011
       000
       ---
        111
        101
        ---
         101
         101
         ---
           0    ← yeah!
```

Another Example

(Figure 3-8)



Power of CRC?

- Assume an error, T(x) + E(x) arrives
- Each 1 bit in E(x) is an inverted bit
- Receiver does [T(x) + E(x)] / G(x)
- Since T(x) / G(x) = 0, result is E(x) / G(x)
- If E(x) factor of G(x), then error slips by
  - all other errors are caught

Power of CRC!!

- IEEE 802 Standard:
  - $x^{32}+x^{26}+x^{22}+x^{16}+x^{12}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$
  - Detects burst errors of length 32 or less
- Final words:
  - Checksum calculation seems complex
  - Only need a simple shift register circuit to compute and verify
  - Virtually all LANs and point-to-point lines use it
  - Previous assumption: bits in frame are random
  - Correlation between bits make errors more common