CSE160: Computer Networks

Lecture #04 – Framing, Error Detection and Correction

2020-09-08

Professor
Alberto E. Cerpa

Last Time

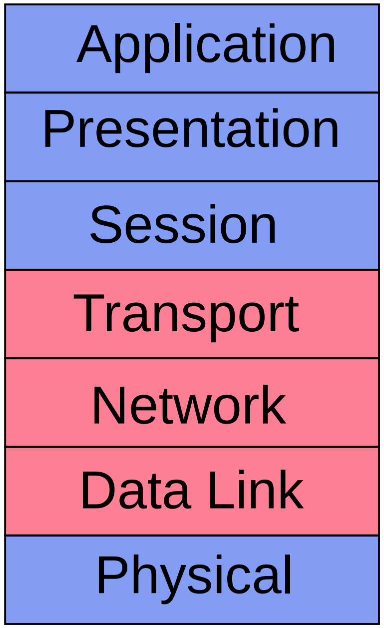
- We typically model links in terms of bandwidth and delay, from which we can calculate message latency
- Different media have different properties that affect their performance as links
- We need to encode bits into signals so that we can recover them at the other end of the channel
- Passband modulation allows us to transmit baseline signals in other higher frequencies for fiber and wireless
- Shannon showed us the limit of what is possible to transmit on any channel

This Lecture

- Framing and Error detection and correction

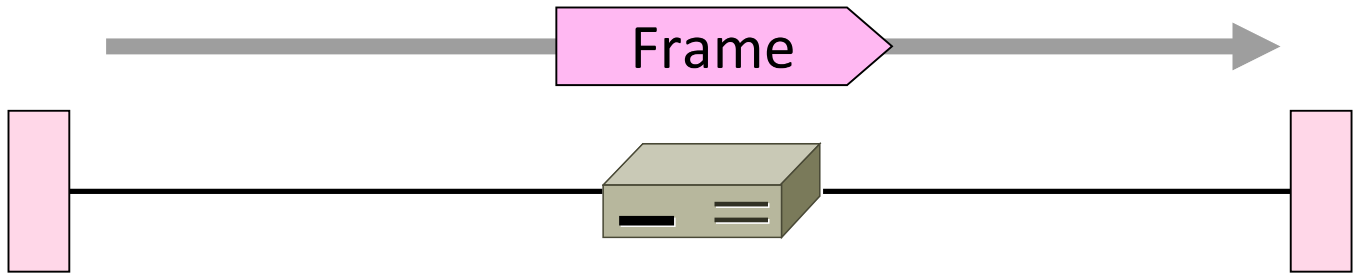
- Focus: How do we send full messages? How do we detect and correct messages that are garbled during transmission?

- The responsibility for doing this cuts across the different layers

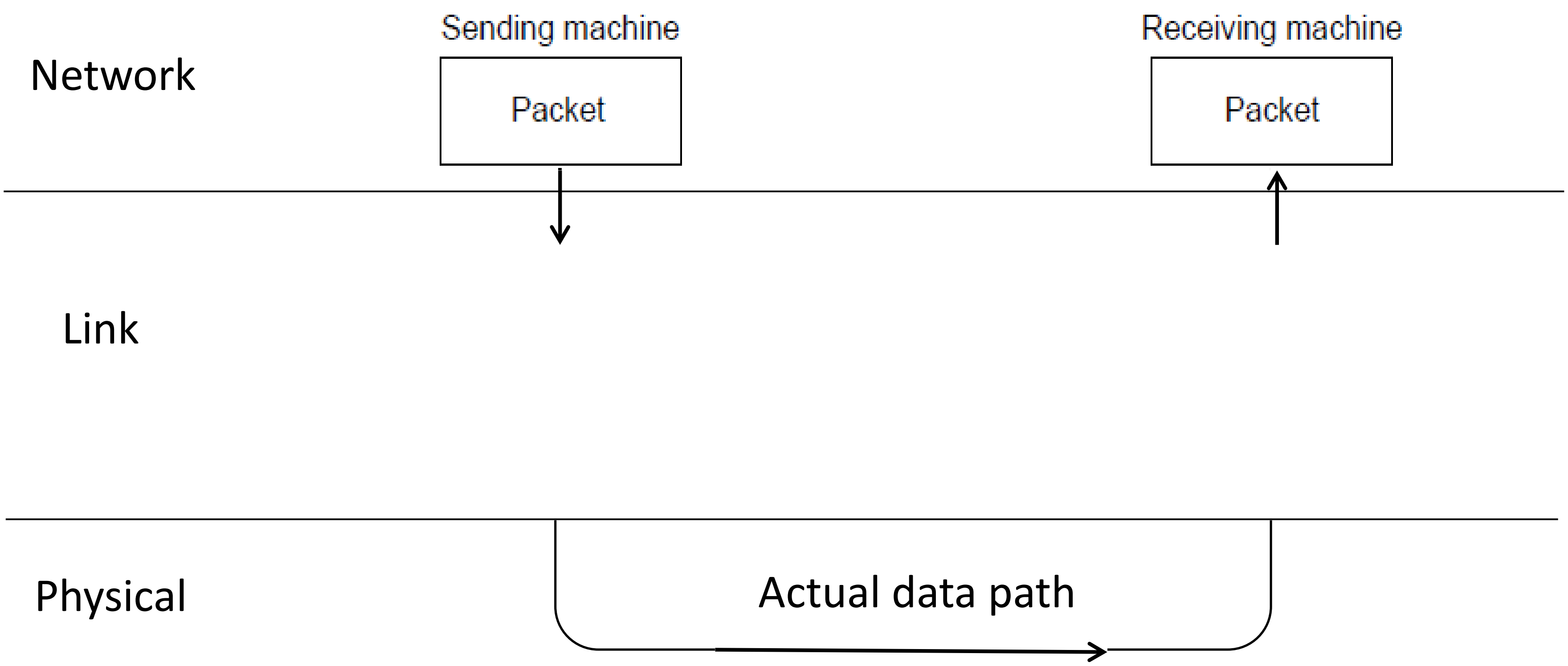


Scope of the Link Layer

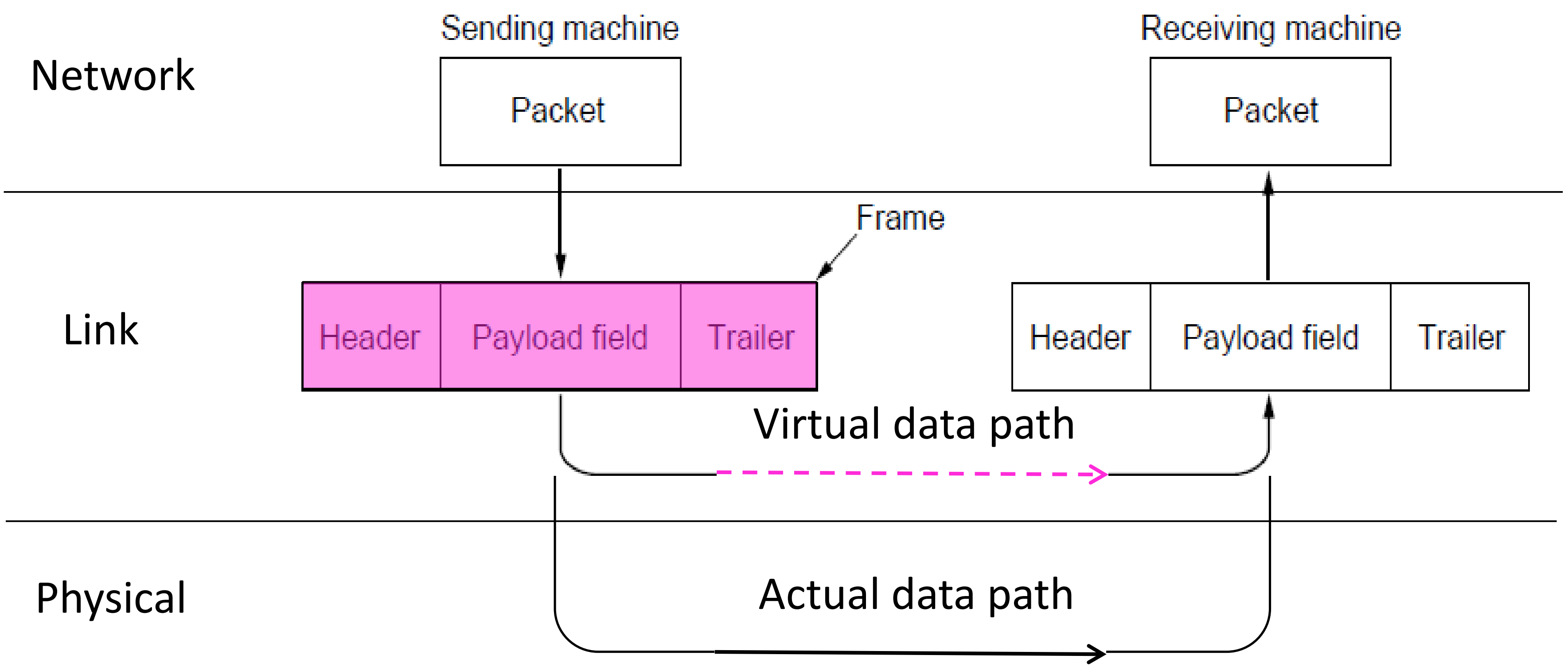
- Concerns how to transfer messages over one or more connected links
  - Messages are frames, of limited size
  - Buildings on the physical layer



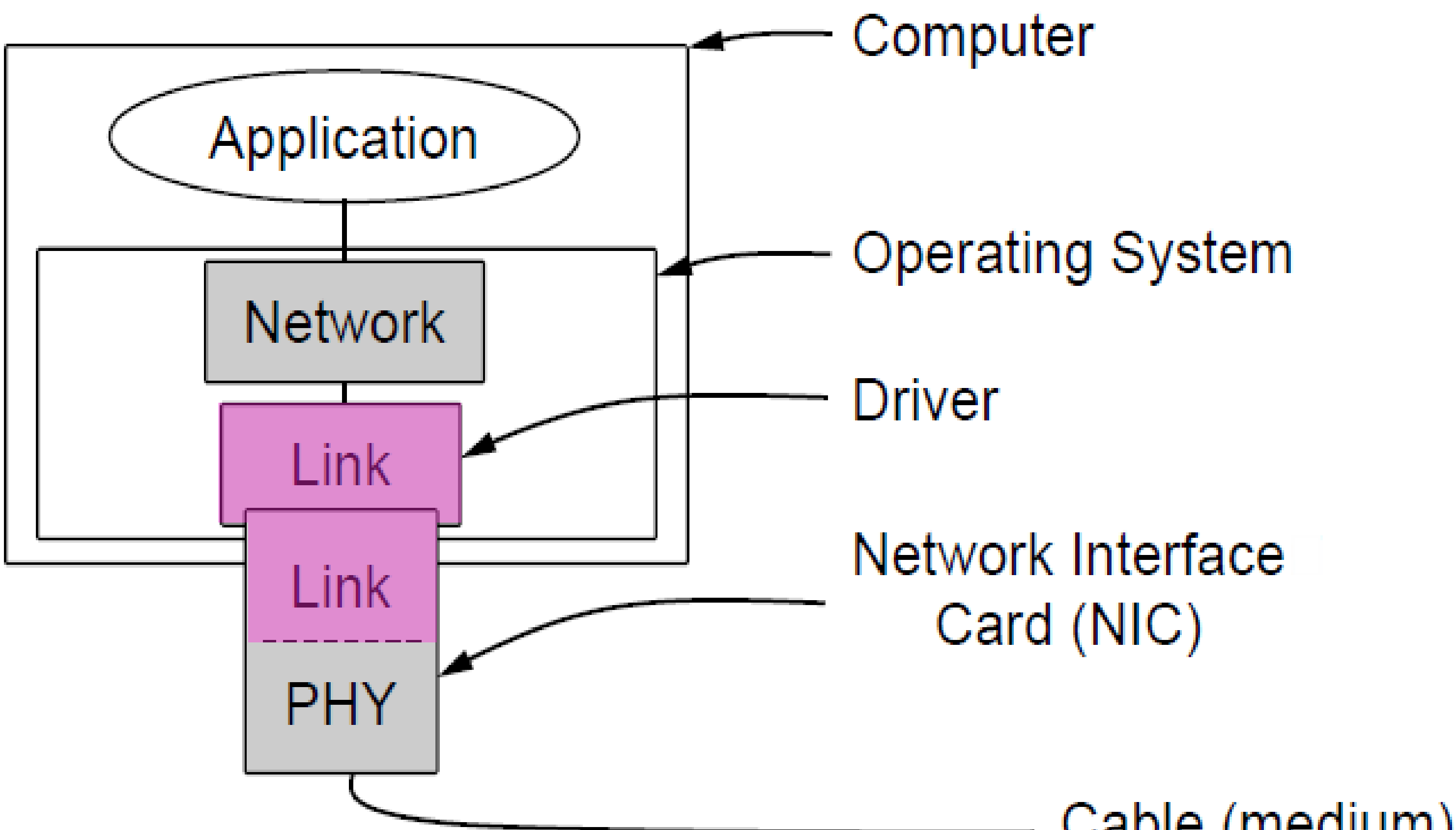
In terms of layers



In terms of layers

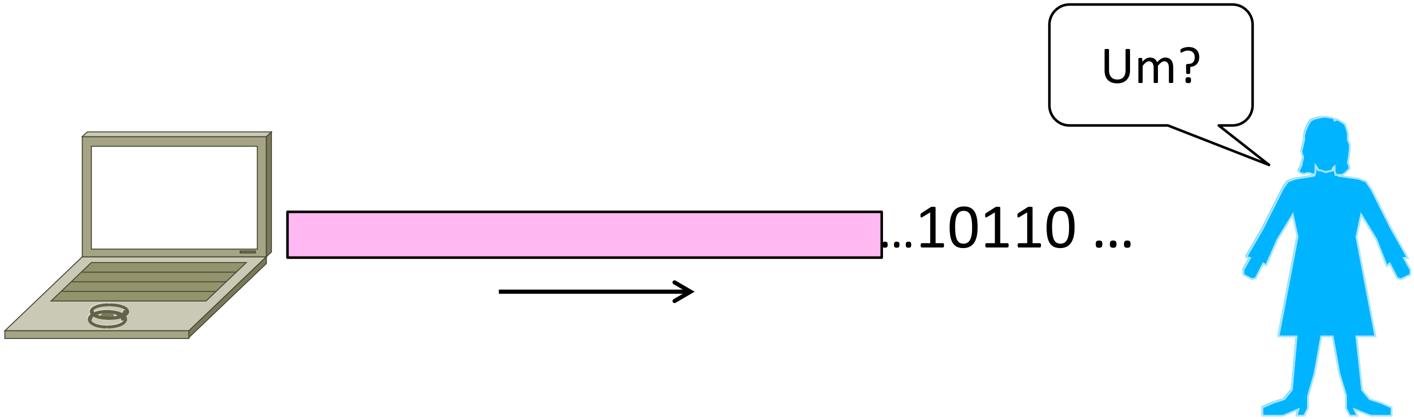


Typical Implementation of Layers



Framing

- Need to send message, not just bits
  - The physical layers gives us a stream of bits. How do we interpret it as a sequence of frames?
  - Requires that we synchronize on the start of message reception at the far end of the link



Framing Methods

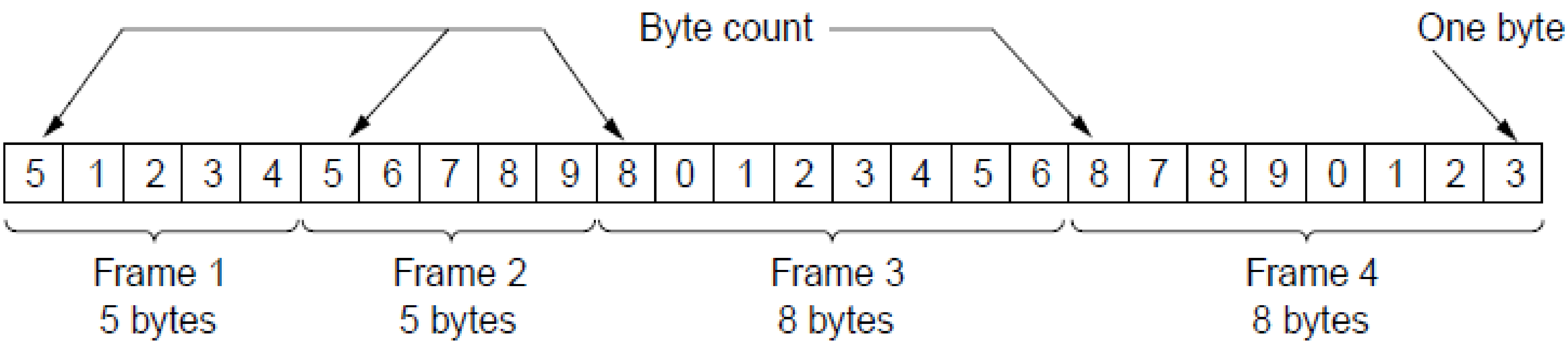
- We'll look at:
  - Byte count (motivation)
  - Byte stuffing
  - Bit stuffing
- In practice, the physical layer often helps to identify frame boundaries
  - E.g., Ethernet, 802.11

Byte Count

- First try:
  - Let's start each frame with a length field!
  - It's simple, and hopefully good enough …

Byte Count (2)

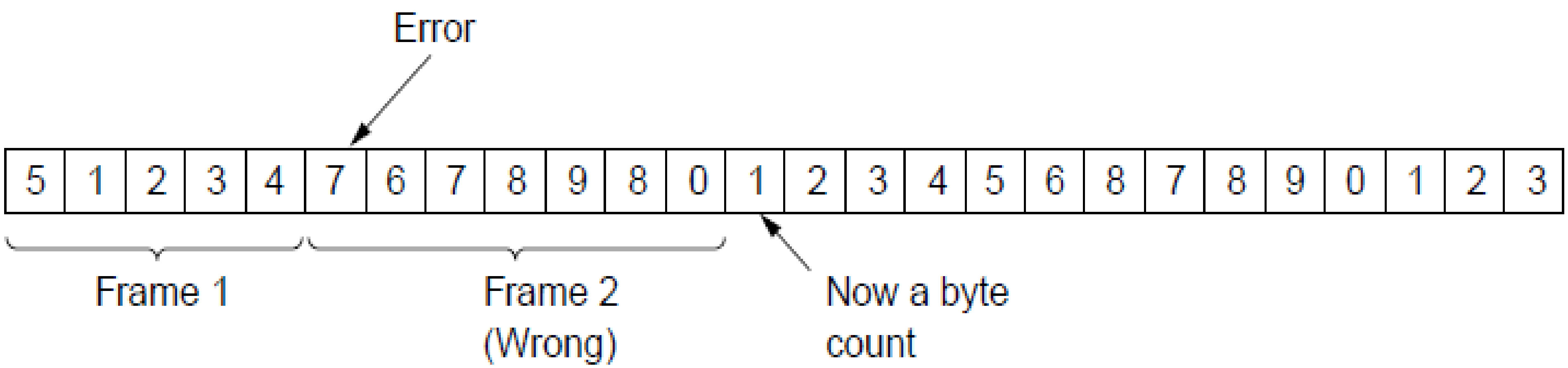
- First byte determines the total frame size



- How well do you think it works?

Byte Count (3)

- Difficult to re-synchronize after framing error
  - Want a way to scan for a start of a frame



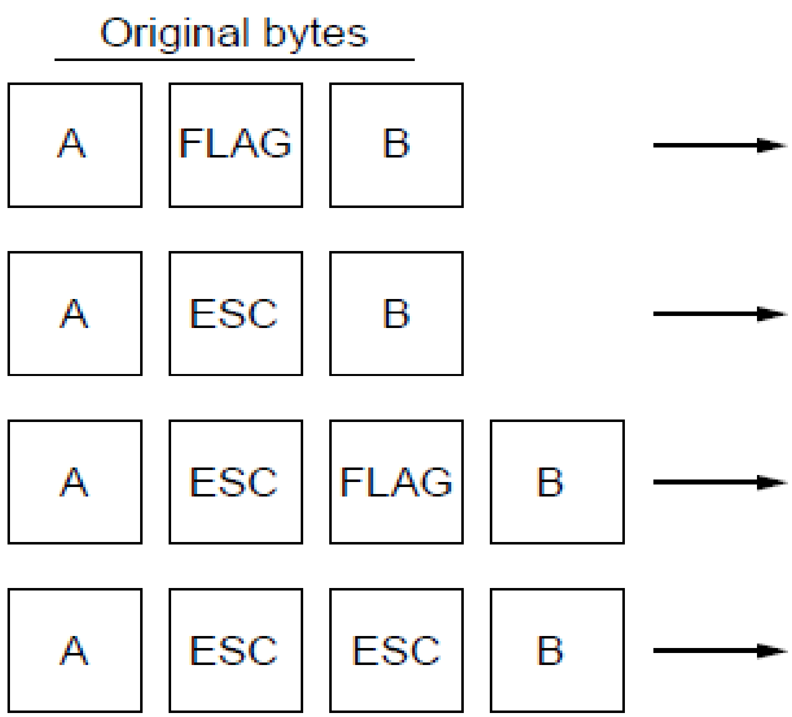
Byte Stuffing

- Better idea:
  - Have a special flag byte value that means start/end of frame
  - Replace (“stuff”) the flag inside the frame with an escape code
  - Complication: have to escape the escape code too!

FLAG	Header	Payload field	Trailer	FLAG

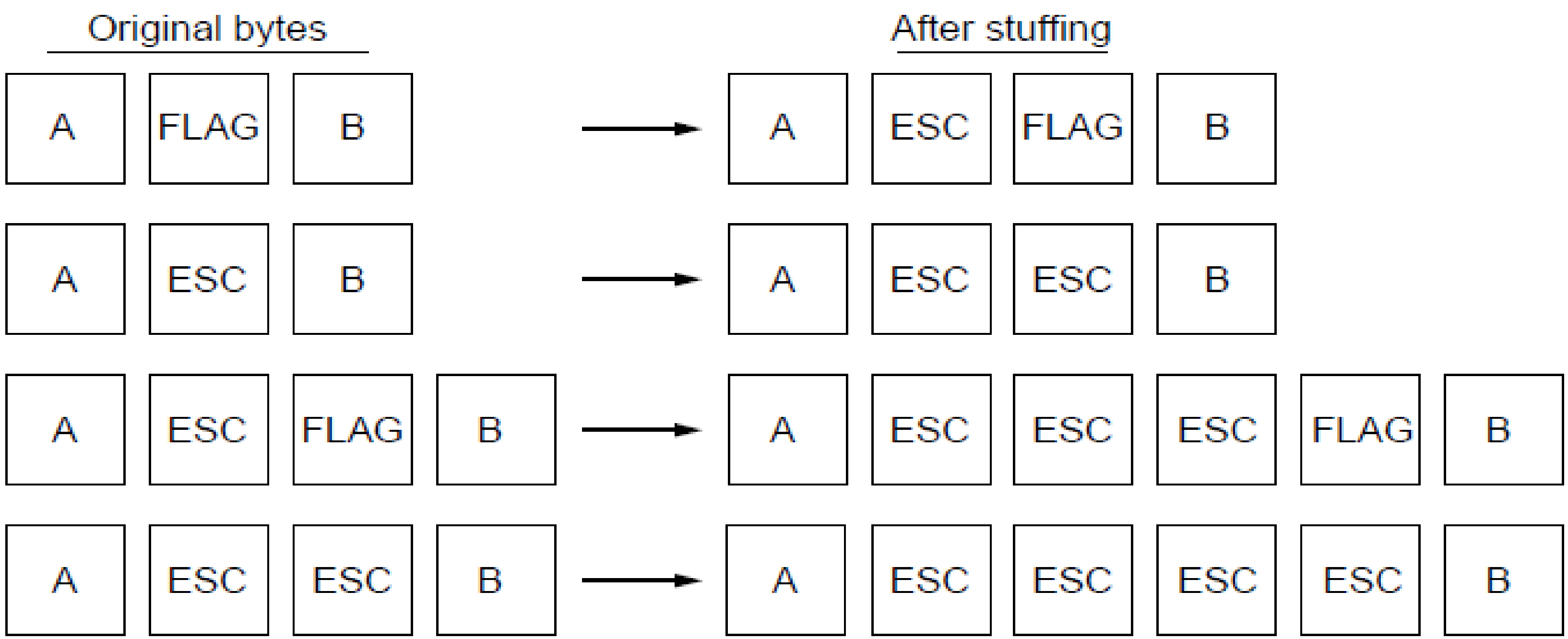
Byte Stuffing (2)

- Rules:
  – Replace each FLAG in data with ESC FLAG
  – Replace each ESC in data with ESC ESC



Byte Stuffing (3)

- Now any unescaped FLAG is the start/end of a frame



Bit Stuffing

- Can stuff at the bit level too
  - Call a flag six consecutive 1s
  - On transmit, after five 1s in the data, insert a 0
  - On receive, a 0 after five 1s is deleted

Bit Stuffing (2)

- Example:

Data bits 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

Transmitted bits
with stuffing

Bit Stuffing (2)

- So how does it compare with byte stuffing?

Data bits 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

Transmitted bits with stuffing 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

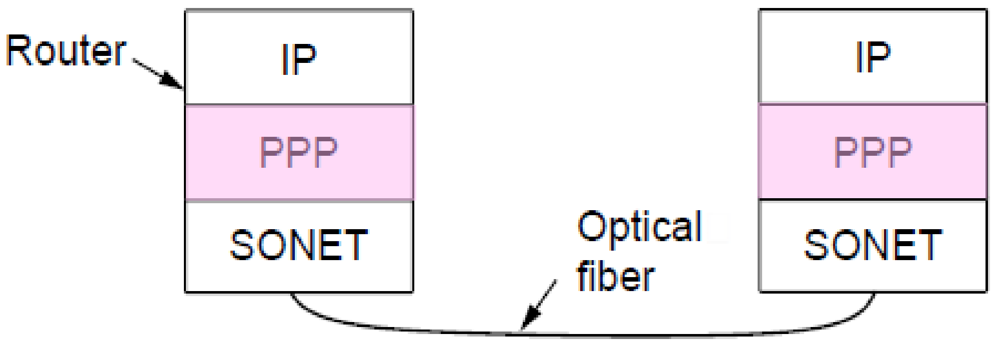
- It is slightly more efficient but more complicated in practice

Link Example: PPP over SONET

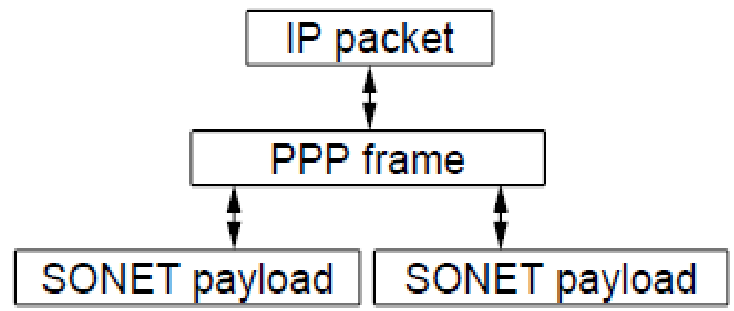
- PPP is Point-to-Point Protocol
- Widely used for link framing
  - E.g., it is used to frame IP packets that are sent over SONET optical links

Link Example: PPP over SONET (2)

- Think of SONET as a bit stream, and PPP as the framing the carries an IP packet over the link



Protocol stacks



PPP frames may be split over SONET payloads

Link Example: PPP over SONET (3)

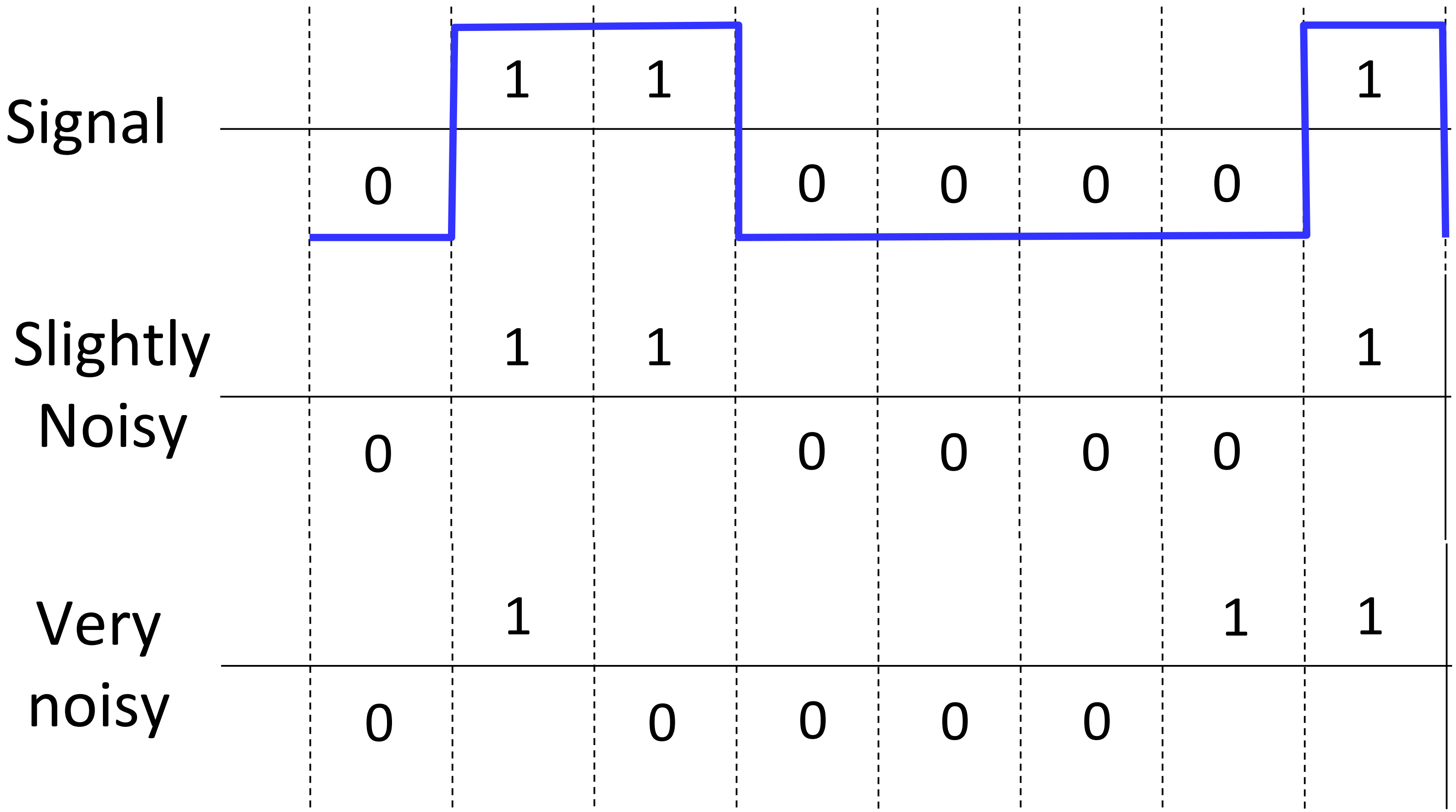
- Framing uses Byte stuffing method:
  - FLAG is 0x7E and ESC is 0x7D
  - To stuff (unstuff) a byte, add (remove) ESC (0x7D), and XOR byte with 0x20
  - Removes FLAG from the contents of the frames

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

Error Coding

- Noise can flip some of the bits we receive
  - We must be able to detect when this occurs!
  - Why?
  - Who needs to detect it? (links, routers, OSs, or apps?)
- What can we do?
  - Detect errors with codes
  - Correct errors with codes
  - Retransmit lost frames ← Later
- Reliability is a concern that cuts across the layers – we'll see it again

Problem – Noise may flip received bits



Approach – Add Redundancy

- Error detection codes allow errors to be recognized
  - Add check bits to the message bits to let some errors be detected
- Error correction codes allow errors to be repaired
  - Add more check bits to let some errors be corrected
- Key issue is now to structure the code to detect many errors with few check bits and modest computation

Motivating Example

- What would be the simplest error detection scheme?
- A simple error detection scheme:
  - Just send two copies. Differences imply errors.

- How good is this code?
  - How many errors can it detect/correct?
  - How many errors will make it fail?

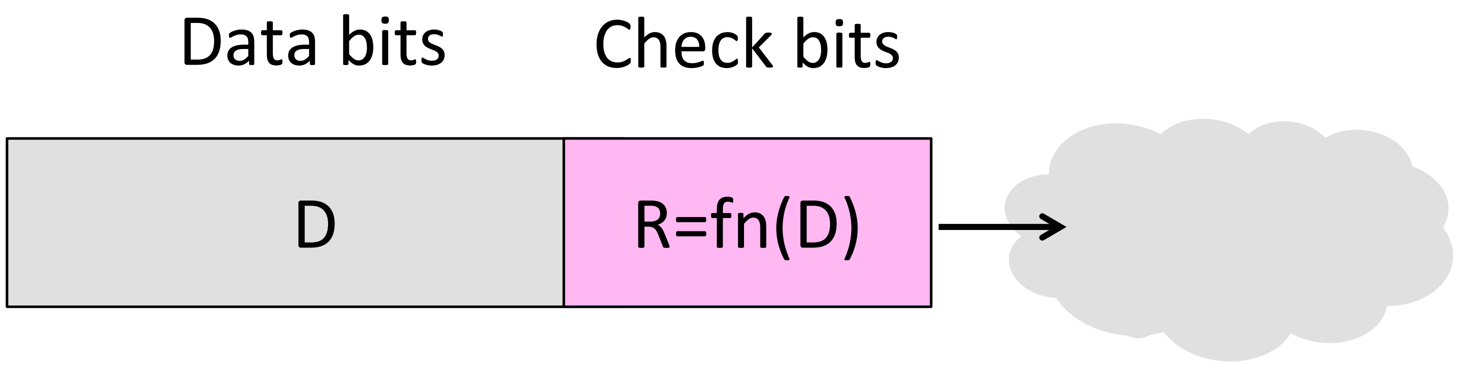
Motivating Example (2)

- We want to handle more errors with less overhead
  - Will look at better codes; they are applied mathematics
  - But, they can't handle all errors
  - And they focus on accidental errors (will look at secure hashes later)

- We will look at basic block codes
  - K bits in, N bits out is a (N,K) code
  - Simple, memoryless mapping

Using Error Codes

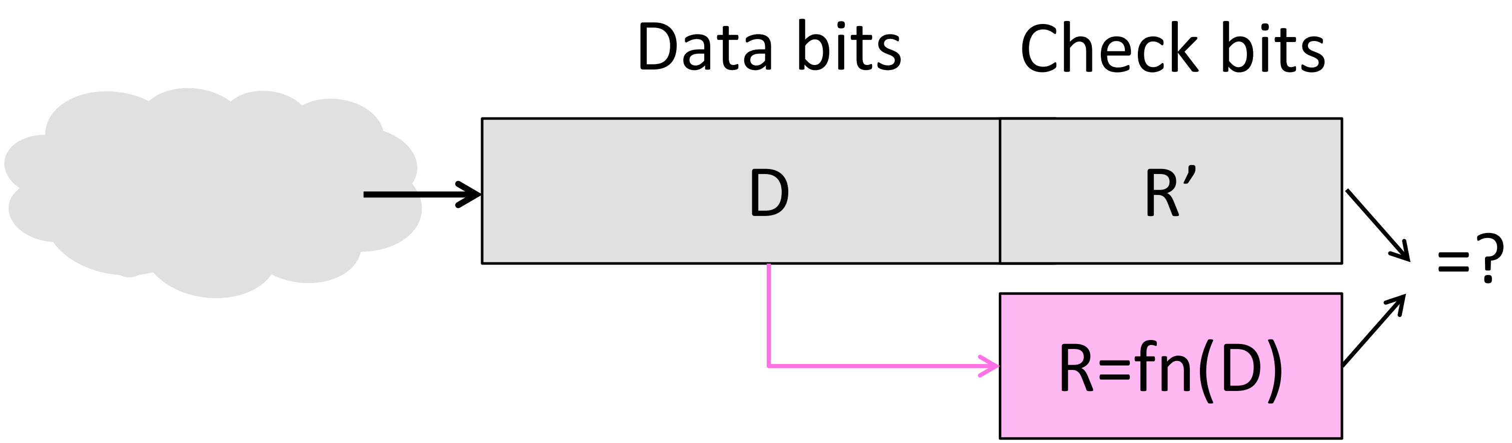
- Codeword consists of D data plus R check bits (=systematic block code)



- Sender:
  - Compute R check bits based on the D data bits; send the codeword of D+R bits

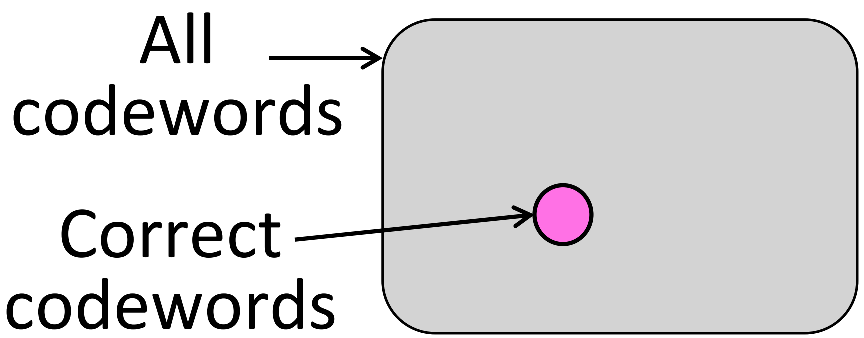
Using Error Codes (2)

- Receiver:
  - Receive D+R bits with unknown errors
  - Recompute R check bits based on the D data bits; error if R doesn't match R'
  - Note the error can be in D, R or both



Intuition for Error Codes

- For D data bits, R check bits:



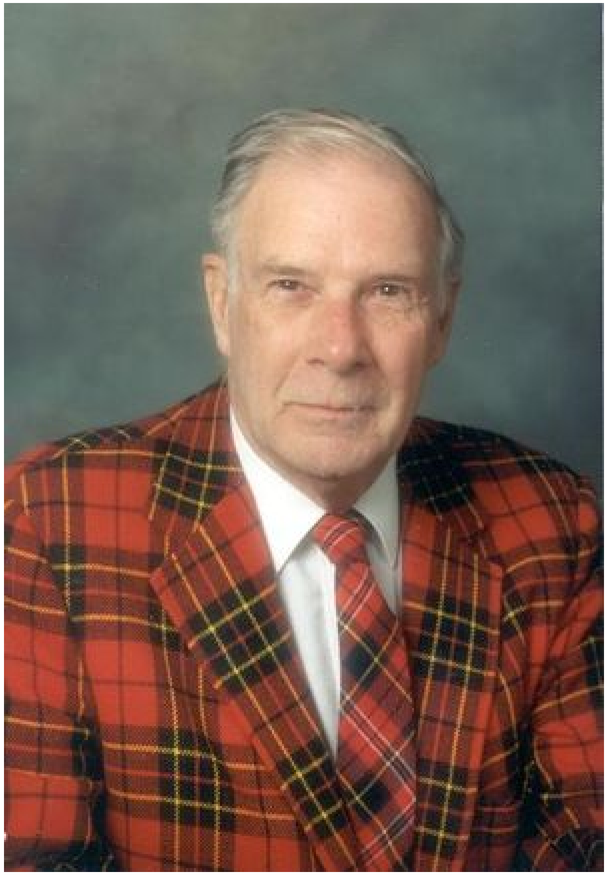
$2^{(D+R)}$ that could be received

$2^{(D)}$ that could be correct

- Randomly chosen codeword is <u>unlikely</u> to be correct; overhead is low
  – How unlikely?
  – Prob = $1/2^R$

Richard W. Hamming (1915-1998)

- Much early work on codes:
  - “Error Detecting and Error Correcting Codes”, BSTJ, 1950
  - Very easy to read, and very elegant way to explain hamming codes
- See also:
  - “You and Your Research”, 1986



Source: IEEE GHN, © 2019 IEEE

The Hamming Distance

- Errors must not turn one valid codeword into another valid codeword, or we cannot detect/correct them
- Hamming distance of a code is the smallest number of bit differences that turn any one codeword into another (i.e. $(D+R)_1$ to $(D+R)_2$)
  - e.g, code 000 for 0, 111 for 1, Hamming distance is 3
- For code with distance d = D+1:
  - D errors can be detected, e.g, 001, 010, 100, 110, 101, 011
- For code with distance d = 2C+1:
  - C errors can be corrected, e.g., 001 → 000

Error Detection

- Some bits may be received in error due to noise How do we detect this?
  - Parity
  - Checksums
  - CRCs

- Detection will let us fix the error, for example, by retransmission (later)

Simple Error Detection – Parity Bit

- Have you ever used a modem or a serial connection?   Ever heard of parity?
- Start with D bits and add 1 check bit that is the sum of the D bits
  - Sum is modulo 2 or XOR
  - e.g. 0110010 → 0110010$\boxed{1}$
  - Easy to compute as XOR of all input bits
- Will detect an odd number of bit errors
  - But not an even number
- Does not correct any errors
- Hamming distance?

2D Parity

- Add parity row/column to array of bits
- Detects all 1, 2, 3 bit errors, and many errors with >3 bits.
- Corrects all 1 bit errors

```
0101001  1
1101001  0
1011110  1
0001110  1
0110100  1
1011111  0

1111011  0
```

Checksums

- Idea: sum up the data in N-bit words and send the data along with the sum
  - Widely used in, e.g., TCP/IP/UDP
  - Stronger protection than parity

- Algorithm:
  - checksum is the 1s complement of the 1s complement sum of the data interpreted 16 bits at a time (for 16-bit TCP/UDP checksum)
- 1s complement: flip all bits to make number negative
  - Adding requires carryout to be added back

Internet Checksum Example

- Sending:
  1. Arrange data in 16-bit words
  2. Put zero in checksum position, add
  3. Add any carryover back to get 16 bits
  4. Negate (complement) to get sum

```
   0001
   f203
   f4f5
   f6f7
+(0000)
------
  2ddf0
    ↓
   ddf0
+     2
------
   ddf2
    ↓
   220d
```

Internet Checksum Example (2)

- Receiving:
  1. Arrange data in 16-bit words
  2. Checksum will be non-zero, add
  3. Add any carryover back to get 16 bits
  4. Negate (complement) the result and check it is 0

```
  0001
  f203
  f4f5
  f6f7
+ 220d
------
 2fffd
   ↓
  fffd
+    2
------
  ffff
   ↓
  0000
```

Internet Checksum

- How well does the checksum work?
  - What is the distance of the code?
  - How many errors can detect/correct?
    - Detect: 1, Correct: 0
    - Hamming distance = 2

- What about larger errors?
  - All burst errors up to 16 in length
  - Random errors (random data), with probability $1/2^{16}$

Checksum Code

```
uint16_t
cksum(uint16_t *buf, int count) {
   register uint32_t sum = 0;
   while (count--) {
          sum += *buf++;
          if (sum & 0xFFFF0000) {
                   sum &= 0xFFFF;
                   sum++;
          }
   }
   return ~(sum & 0xFFFF);
}
```

CRCs (Cyclic Redundancy Check)

- Stronger protection than checksums
  - Used widely in practice, e.g., Ethernet CRC-32
  - Implemented in hardware (XORs and shifts)

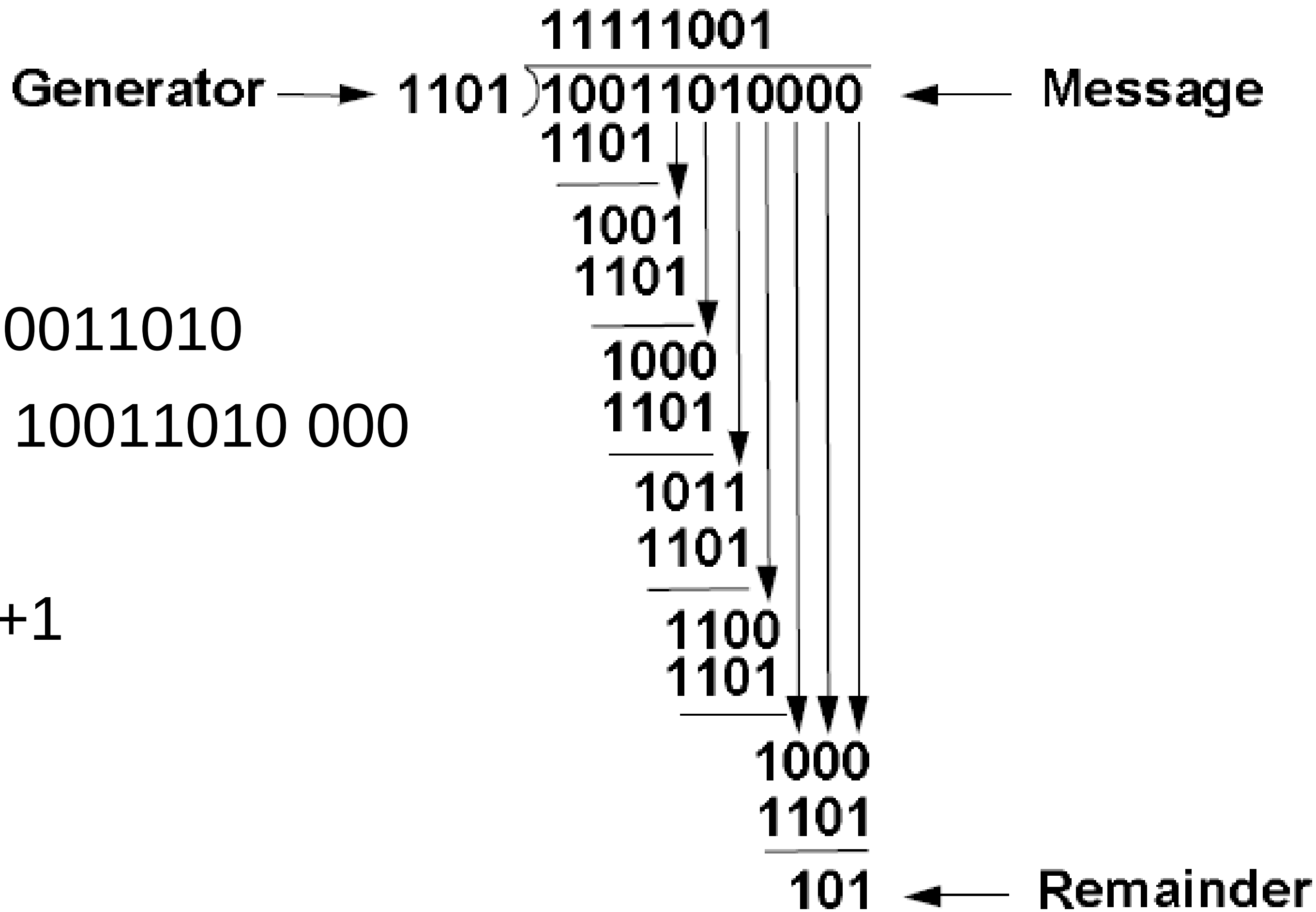
- Algorithm: Given n bits of data, generate a k check bits such that the n+k bits that are evenly divisible by a chosen divisor C(x)

- Based on mathematics (arithmetic) of finite fields
  - “numbers” correspond to polynomials, use modulo arithmetic
  - e.g, interpret 10011010 as $x^7 + x^4 + x^3 + x^1$

CRC Example

- How do we generate the check sequence?
  - Have our message, e.g., 10011010 (m=8)
  - Have the CRC as a divisor polynomial e.g., C(x) = 1110 ($x^3 + x^2 + x^1$; k=3)
  - Want to make m + k bits divisible by this divisor…
- Send Procedure:
  1. Extend the n data bits with k zeros
  2. Divide by C(x) to find the reminder (ignore quotient)
  3. Adjust k check bits by remainder
- Receive Procedure:
  1. Divide and check for zero remainder

Example – Polynomial Division



Data bits: 10011010

Mesg. bits: 10011010 000

Check bits:

$C(x)=x^3+x^2+1$

C = 1101

k = 3

Example – Remainder to CRC

- So we see the remainder is 101
- Thus the zero extended message – 101 must be evenly divisible by C(x)!
- So perform the subtraction to discover the check bits
  - Subtraction/addition is XOR in module 2 arithmetic
  - E.g., we get 10011010000 – 101 = 10011010101
  - The check bits are 101
- Finally, message we send is 10011010101

How is C(x) Chosen?

- Mathematical properties:
  - All 1-bit errors if non-zero x^k and x^0 terms
  - All 2-bit errors if C(x) has a factor with at least three terms
  - Any odd number of errors if C(x) has (x + 1) as a factor
  - Any burst error < k bits
  - For random errors, Prob(corruption undetected) = 1/(2^n) (e.g. CRC-n with n being 8, 12, 16, 32, etc.)
- There are standardized polynomials of different degree that are known to catch many errors
  - Ethernet CRC-32: 100000100110000010001110110110111
  - Corruption undetected (random) = 2.3*10^(-10), less than one in a billion

Standard CRC Polynomials

- CRC-8 100000111
  - ITU-T I.432.1 (02/99); ATM HEC, ISDN HEC and others
- CRC-10 11000110011
  - ATM; ITU-T I.610, mobile networks
- CRC-12 1100000000111
  - Telecom systems
- CRC-16 10001000000100000
  - X.25, V.41, HDLC *FCS*, XMODEM, Bluetooth, PACTOR, SD, DigRF, many others
- CRC-32 100000100110000010001110110110111
  - ISO 3309 (HDLC), ISO/IEC/IEEE 802-3 (Ethernet), SATA, MPEG2, PKZIP, Gzip, Bzip2, POSIX cksum, PNG, ZMODEM, many others

Error Detection in Practice

- CRC are widely used on links
  - Ethernet, 802.11, ADSL, Cable…
- Checksum used in Internet
  - IP, TCP, UDP, … but it is weak
- Parity
  - Is little used

Error Correction

- Some bits may be received in error due to noise. How do we fix them?
  – Hamming code
  – Other codes

- And why should we use detection when we can use correction?

Why Error Correction is Hard?

- If we had reliable check bits we could use them to narrow down the position of the error
  – Then correction would be easy

- But error could be in the check bits as well as the data bits!
  – Data might even be correct!

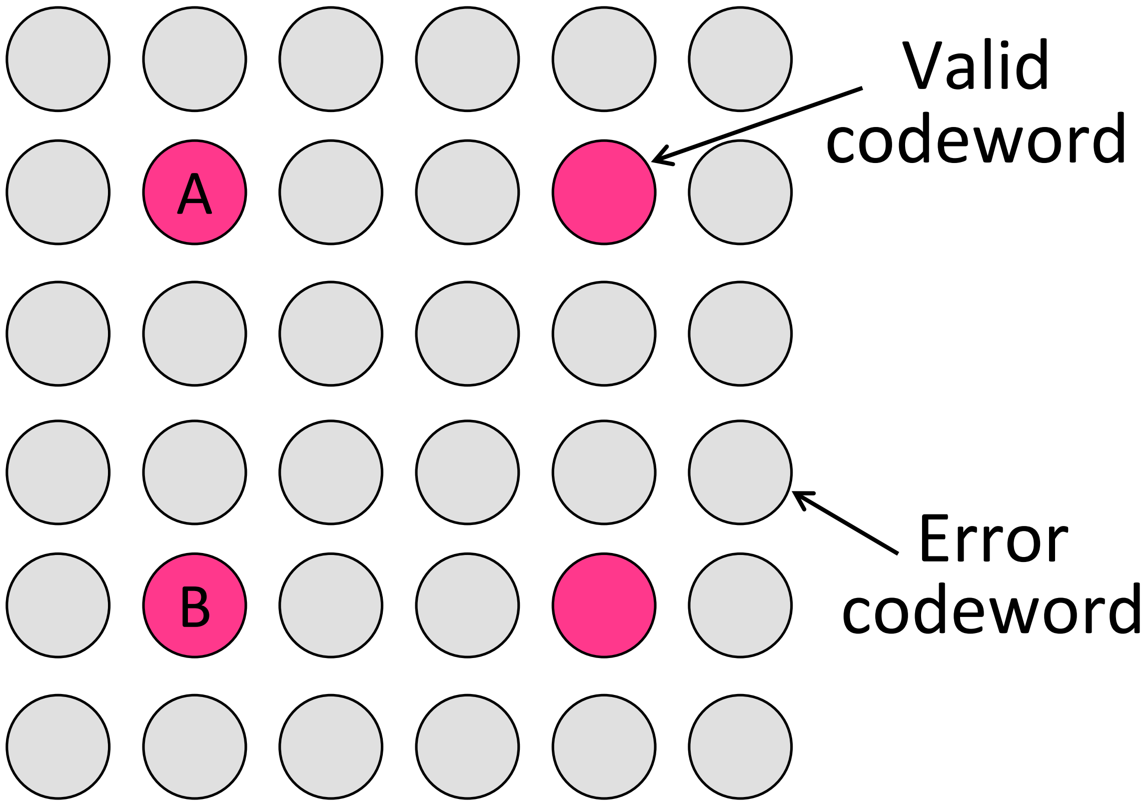
Intuition for Error Correcting Code

- Suppose we construct a code with a Hamming distance of at least 3
  - Need ≥3 bit errors to change one valid codeword into another
  - Single bit errors will be closest to a unique valid codeword

- If we assume errors are only 1 bit, we can correct them by mapping an error to the closest valid codeword
  - Works for d errors if HD ≥ 2d + 1

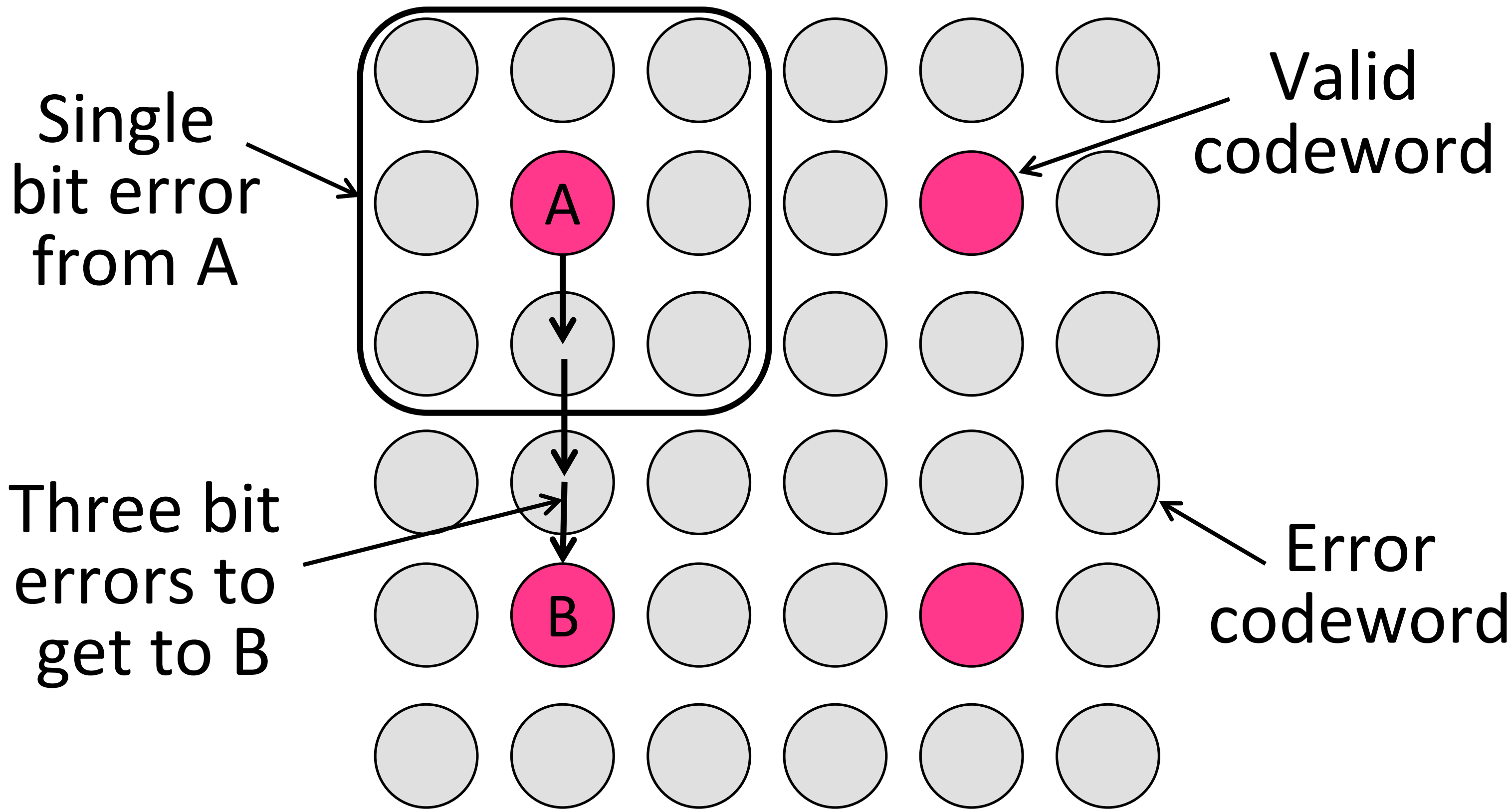
Intuition (2)

- Visualization of code:



Intuition (3)

- Visualization of code:



Hamming Code

- Gives a method for constructing a code with a distance of 3
  - Uses $n = 2^k - k - 1$, e.g., n=4 k=3
  - Put check bits in positions p that are powers of 2, starting with position 1
  - Check bit in position p is parity of positions with a p term in their values
- Plus an easy way to correct [soon]

Hamming Code (2)

- Example: data = 0101, 3 check bits
  - 7 bits code, check bit positions 1, 2, 4
  - Check 1 covers positions 1, 3, 5, 7
  - Check 2 covers positions 2, 3, 6, 7
  - Check 4 covers positions 4, 5, 6, 7

$\underline{0}\ \underline{1}\ 0\ \underline{0}\ 1\ 0\ 1$

1 2 3 4 5 6 7

  - $P_1 = 0+1+1 = 0$
  - $P_2 = 0+0+1 = 1$
  - $P_4 = 1+0+1 = 0$

Hamming Code (3)

- To decode:
  - Recompute check bits (with parity sum including the check bit)
  - Arrange as a binary number
  - Value (syndrome) tells error position
  - Value of zero means no error
  - Otherwise, flip bit to correct

Hamming Code (4)

- Example, continued

$$\underline{0}\ \underline{1}\ 0\ \underline{0}\ 1\ 0\ 1$$

1 2 3 4 5 6 7

  - $P_1 = 0+0+1+1 = 0$
  - $P_2 = 1+0+0+1 = 0$
  - $P_4 = 0+1+0+1 = 0$
  - Syndrome = 000 (no error)
  - Data = 0 1 0 1

Hamming Code (5)

- Example, continued

$$\underline{0}\ \underline{1}\ 0\ \underline{0}\ 1\ 1\ 1$$

1 2 3 4 5 6 7

  - $P_1 = 0+0+1+1 = 0$
  - $P_2 = 1+0+1+1 = 1$
  - $P_4 = 0+1+1+1 = 1$
  - Syndrome = 110 $\rightarrow$ 6 (6th position), flip position 6
  - Data = 0 1 0 1 (corrected after flip!)

Other Error Correction Codes

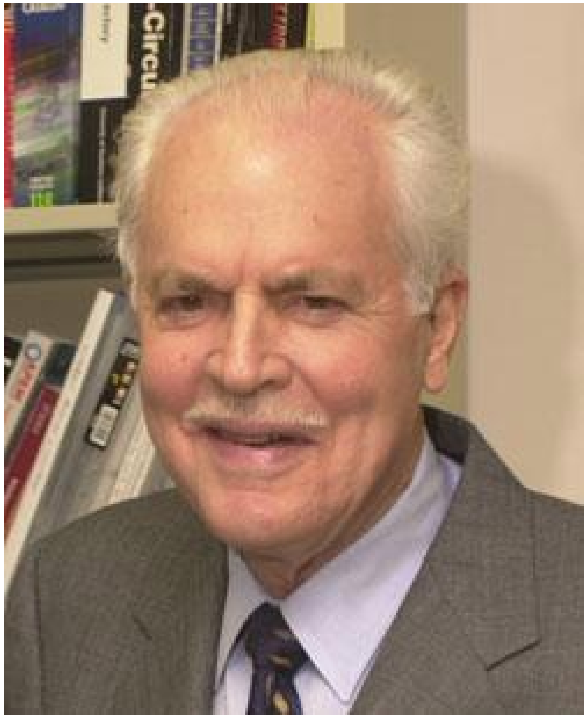
- Codes used in practice are much more involved than Hamming

- Convolutional codes
  - Take a stream of data and output a mix of the recent input bits
  - Makes each output bit less fragile
  - Decode using Viterbi algorithm (which can use bit confidence values)

Reed-Solomon Code

- Developed to protect data on magnetic disks
- Used for CDs, Blue-Rays and cable modems too
- Property: 2t redundant bits can correct <= t errors
- Slowly being replaced by low-density parity-check (LDPC) or turbo codes
- Mathematics somewhat more involved …

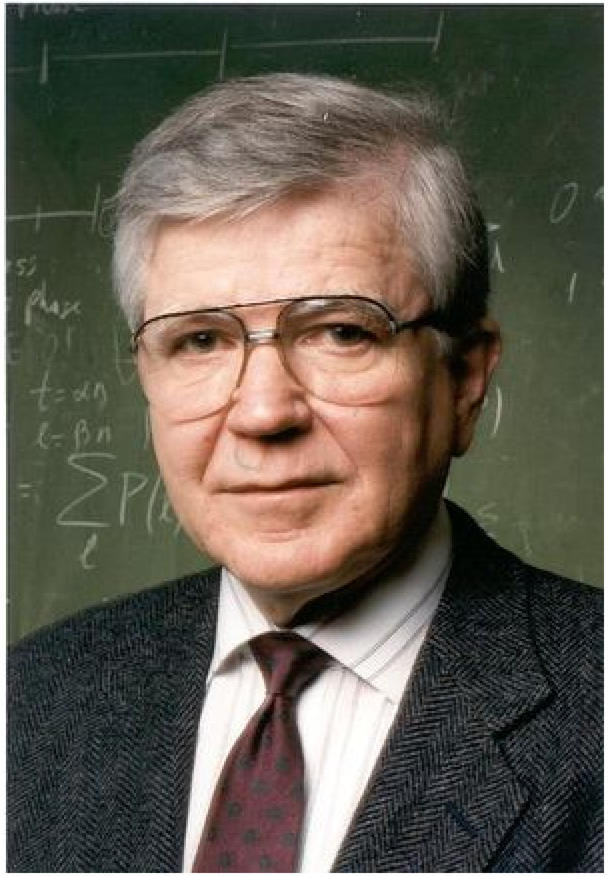
Irving S. Reed



Source: IEEE GHN, © 2019 IEEE

Other Codes -- LDPC

- Low Density Parity Check
  - LDPC based on sparse matrices
  - Decoded iteratively using a belief propagation algorithm
  - State of the art today
- Invented by Robert Gallager in 1963 as part of his PhD thesis
  - Promptly forgotten until 1996…


Source: IEEE GHN, © 2019 IEEE

Detection vs. Correction

- Two strategies to correct errors:
  - Detect and retransmit, or Automatic Repeat reQuest. (ARQ)
  - Error correcting codes, or Forward Error Correction (FEC)
- Satellites, real-time media tend to use error correction
- Retransmissions typically at higher levels (Network+)
- Question: Which should we choose?

Retransmissions vs. FEC

- The better option depends on the kind of errors and the cost of recovery
- Example: Message with 1000 bits, with a bit error rate (BER) of 1 in 1000, i.e., prob(bit error) = 0.001
  – Case 1: random errors
  – Case 2: bursts of 1000 errors
  – Case 3: real-time application (teleconference)

Case 1 – Random Errors

BER: 1 in 1000

1. Assume bit errors are random
   - Messages have 0 or maybe 1 error
- Error correction:
  - Need ~10 check bits per message
  - Overhead: 10 bits
- Error detection:
  - Need ~1 check bits per message plus 1000 bit retransmission ~63.2% of the time
  - Overhead: 1 + 1000*0.632 = 633 bits

Case 2 – Bursty Errors

BER: 1 in 1000

2. Assume errors come in <u>bursts of 1000</u>
   - Only 1 or 2 messages in 1000 have errors

- Error correction:
  - Need >>1000 check bits per message
  - Overhead: >1000 bits
- Error detection:
  - Need 32? check bits per message plus 1000 bit resend 2/1000 of the time
  - Overhead: 32 + 1000/1000*2 = 34 bits

Case 3 – Teleconference App

BER: 1 in 1000

3. Assume bit errors are <u>random</u>

- Error correction:
  - Need ~10 check bits per message; Overhead: 10 bits
- Error detection:
  - Need ~1 check bits per message plus 1000 bit retransmission 63.2% of the time; Overhead: 1 + 1000*0.632 = 633 bits
- Solution: Neither!
  - No error detection/correction.
  - Reduction of latency is critical for the app.
  - If a 32 x 32 pixel image (~1000 bits)… can you afford having an error in a single pixel?
    - YES!

Error Correction in Practice

- Heavily used in physical layer
  - LDPC is the future, used for demanding links like 802.11, DVB, WiMAX, LTE, power-line, …
  - Convolutional codes widely used in practice
- Error detection (w/retransmission) is used in the link layer and above residual errors
- Correction also used in the application layer
  - Called Forward Error Correction (FEC)
  - Normally with an erasure error model
  - E.g., Reed-Solomon (CDs, DVDs, etc.)

Key Concepts

- Framing allows us to send full messages instead of bits
- Redundant bits are added to messages to protect against transmission errors
- Two recovery strategies are retransmissions (ARQ) and error correcting codes (FEC)
- The Hamming distance tells us how much error can safely be tolerated
- The optimal recovery strategy depends on the error expected in the channel and the app requirements