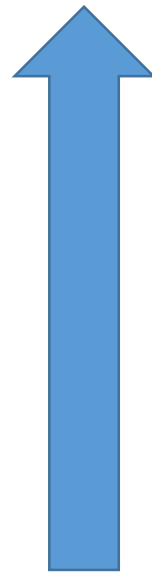
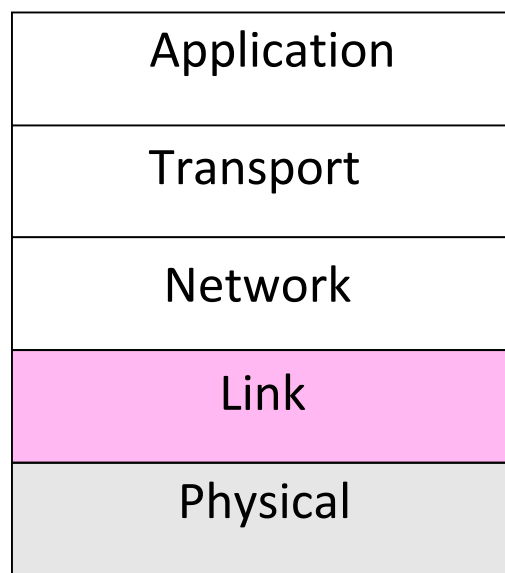


Link Layer

(continued)

Where we are in the Course

- Moving on up to the Link Layer!

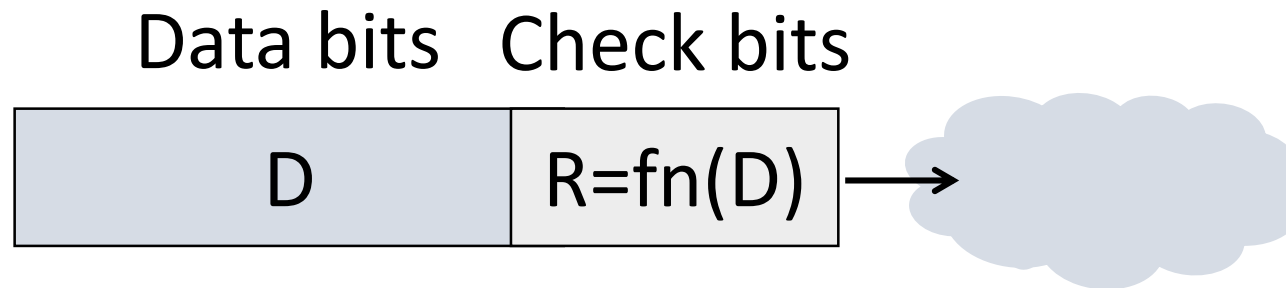


Topics

1. Framing
 - Delimiting start/end of frames
2. Error detection and correction
 - Handling errors
3. Retransmissions
 - Handling loss
4. Multiple Access
 - 802.11, classic Ethernet
5. Switching
 - Modern Ethernet

Using Error Codes

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

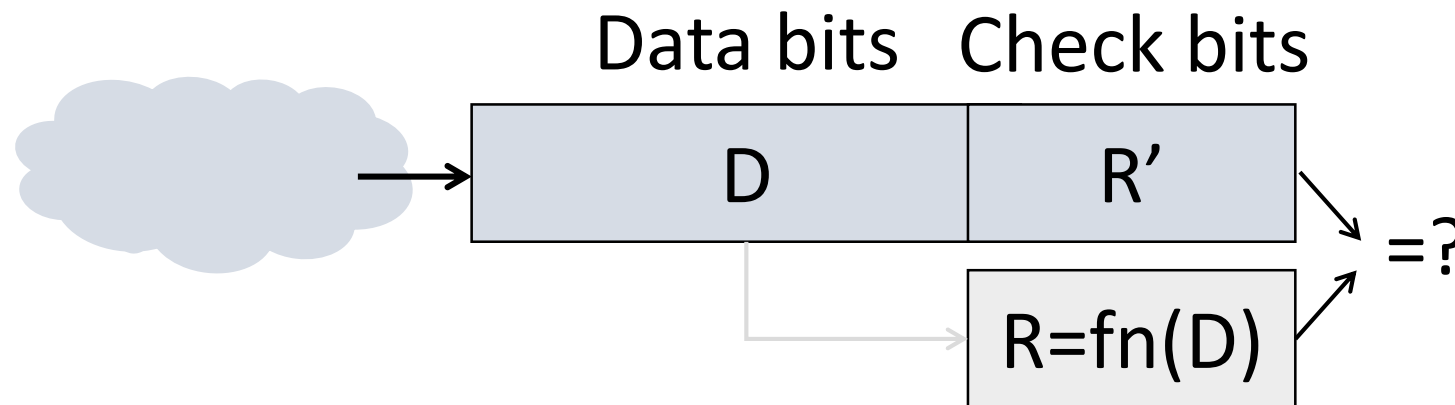


- Sender:
 - Compute R check bits based on the D data bits; send the codeword of $D+R$ bits

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'



Internet Checksum

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

$$\begin{array}{r} 0001 \\ \text{f}204 \\ \text{f}4\text{f}5 \\ \text{f}6\text{f}7 \\ + (0000) \\ \hline 2\text{d}\text{d}\text{f}1 \\ \downarrow \\ \text{d}\text{d}\text{f}1 \\ + \quad 2 \\ \hline \text{d}\text{d}\text{f}3 \\ \downarrow \\ 220\text{c} \end{array}$$

Why Error Correction is Hard

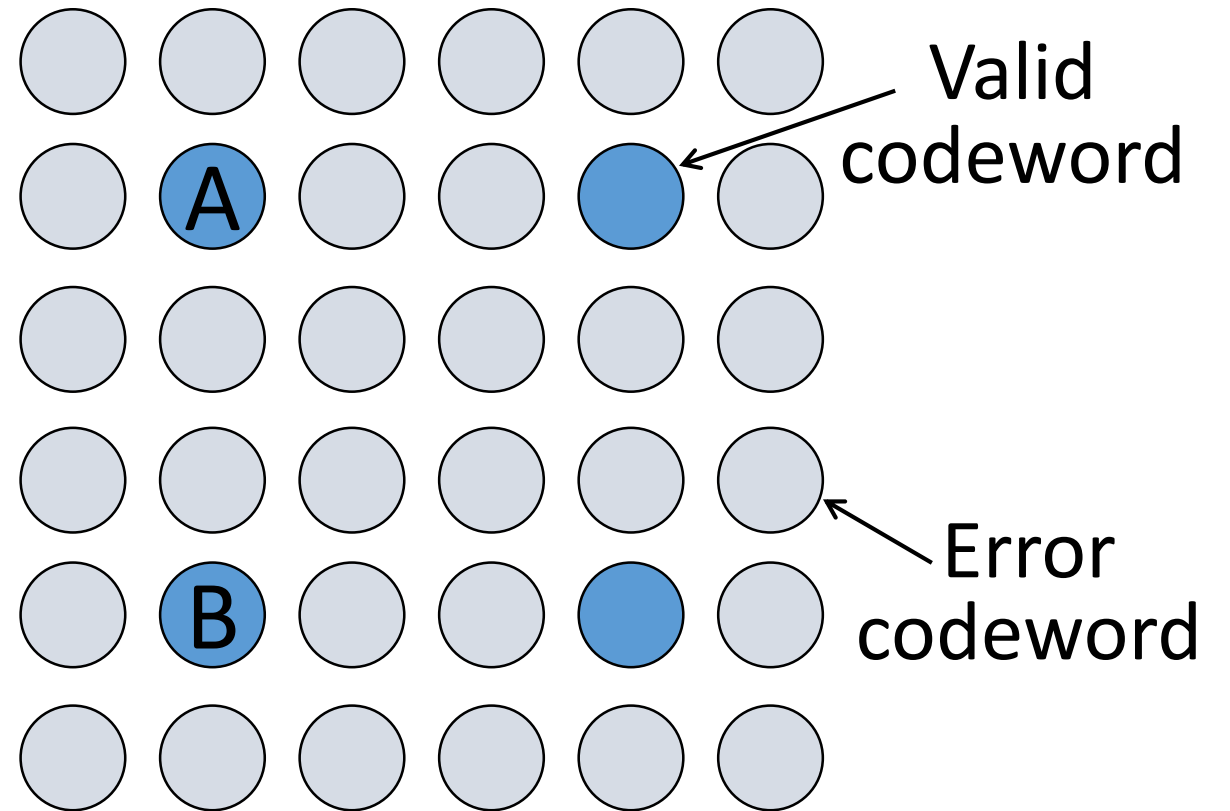
- If we had reliable check bits we could use them to narrow down the position of the error
 - Then correction would be easy
- But error could be in the check bits as well as the data bits!
 - Data might even be correct

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 $HD \geq 2d + 1$

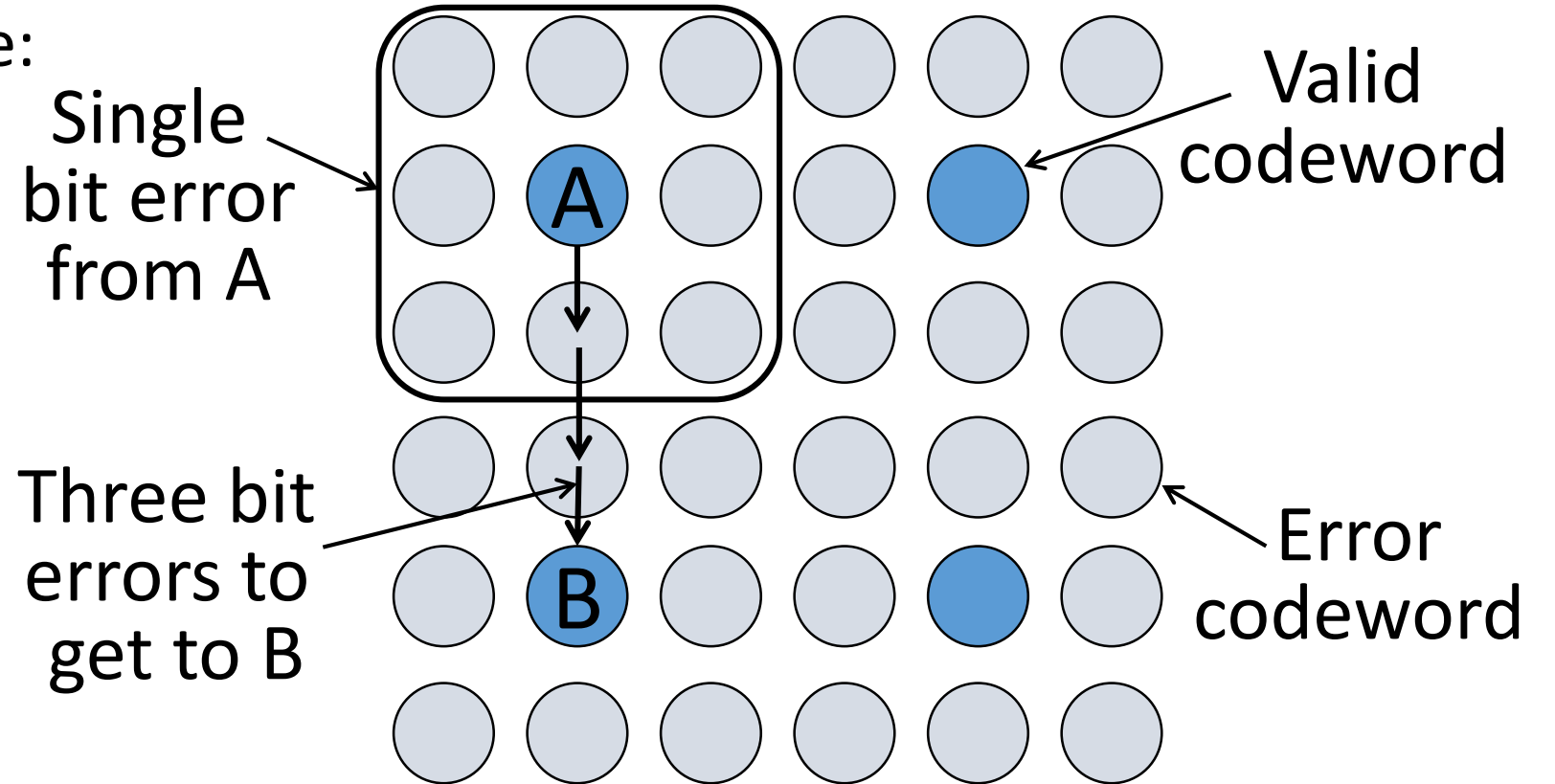
Intuition (2)

- 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 bit 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

1 2 3 4 5 6 7

Hamming Code (3)

- Example: data=0101, 3 check bits
 - 7 bit 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

0 1 0 0 1 0 1 \longrightarrow
1 2 3 4 5 6 7

$$p_1 = 0 + 1 + 1 = 0, \quad p_2 = 0 + 0 + 1 = 1, \quad p_4 = 1 + 0 + 1 = 0$$

Hamming Code (4)

- 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 (5)

- Example, continued

→ 0 1 0 0 1 0 1
1 2 3 4 5 6 7

$p_1 =$

$p_2 =$

$p_4 =$

Syndrome =

Data =

Hamming Code (6)

- Example, continued

→ 0 1 0 0 1 0 1
1 2 3 4 5 6 7

$$p_1 = 0 + 0 + 1 + 1 = 0, \quad p_2 = 1 + 0 + 0 + 1 = 0,$$

$$p_4 = 0 + 1 + 0 + 1 = 0$$

Syndrome = 000, no error

Data = 0 1 0 1

Hamming Code (7)

- Example, continued

→ 0 1 0 0 1 **1** 1
1 2 3 4 5 6 7

$p_1 =$

$p_2 =$

$p_4 =$

Syndrome =

Data =

Hamming Code (8)

- Example, continued

→ 0 1 0 0 1 **1** 1
1 2 3 4 5 6 7

$$p_1 = 0+0+1+1 = 0, \quad p_2 = 1+0+\mathbf{1}+1 = \mathbf{1},$$

$$p_4 = 0+1+\mathbf{1}+1 = \mathbf{1}$$

Syndrome = **1 1** 0, flip position 6

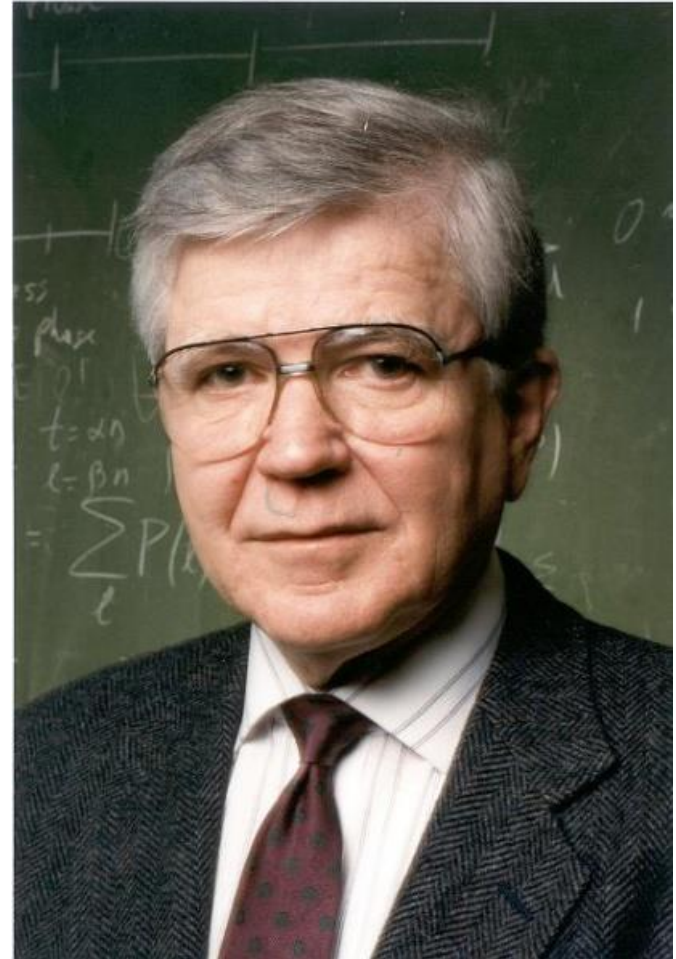
Data = 0 1 0 1 (correct after flip!)

Other Error Correction Codes

- Codes used in practice are more involved than Hamming
- Convolutional codes (§3.2.3)
 - Take a stream of data and output a mix of the input bits
 - Makes each output bit less fragile
 - Decode using Viterbi algorithm (which can use bit confidence values)

Other Codes (2) – LDPC

- Low Density Parity Check (§3.2.3)
 - 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, © 2009 IEEE