

Operating Systems and Networks

Network Lecture 3: Link Layer (1)

Adrian Perrig

Network Security Group

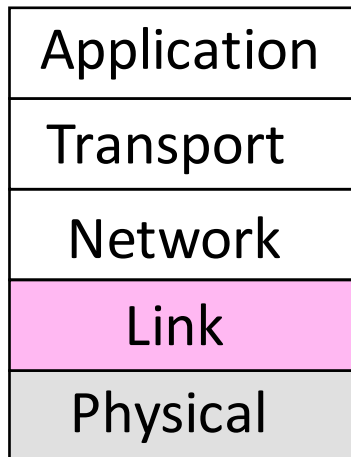
ETH Zürich

Pending Issues

- Project 1 is out
- Exercise sessions starting next week
 - Tuesday and Friday only for next week
 - Project 1 and homework will be discussed

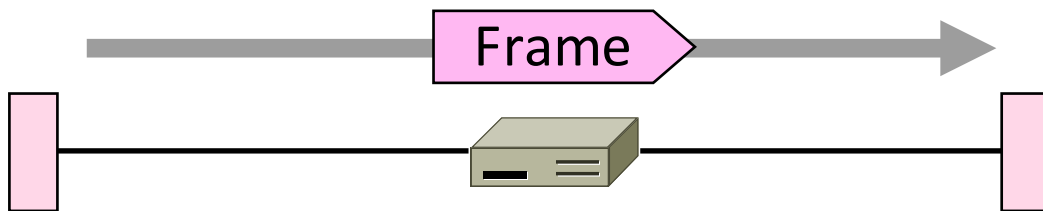
Where we are in the Course

- Moving on to the Link Layer!

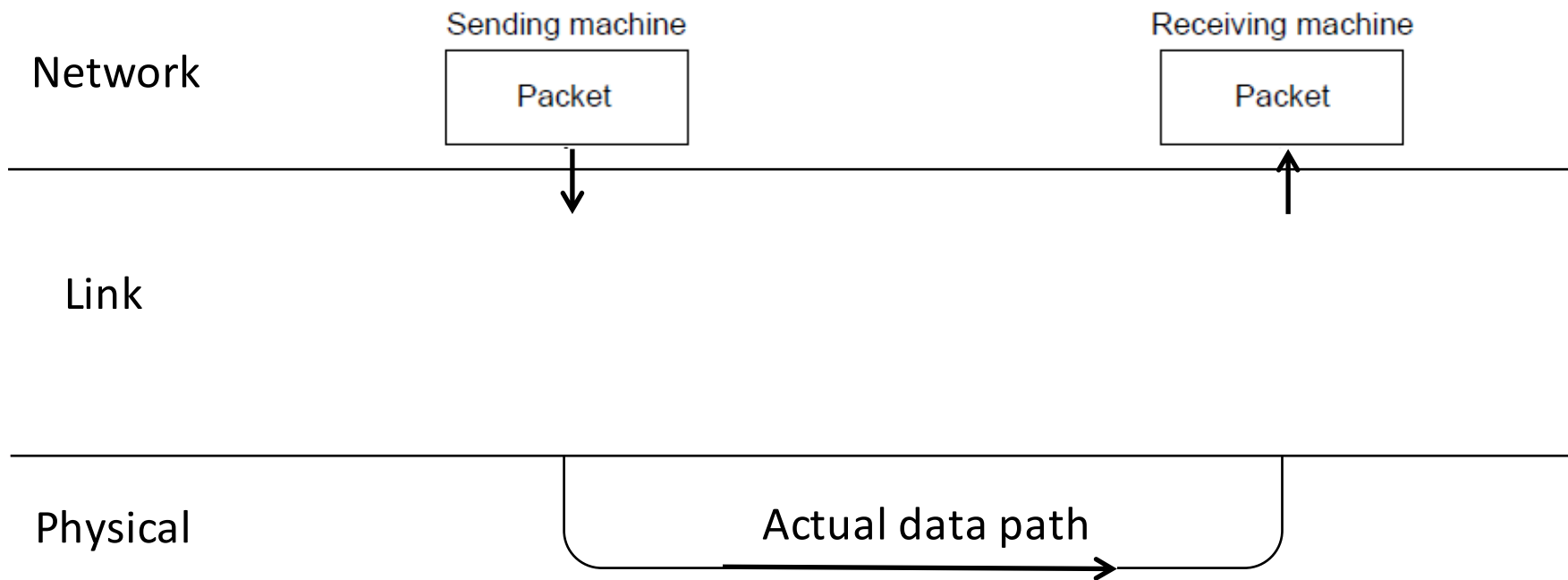


Scope of the Link Layer

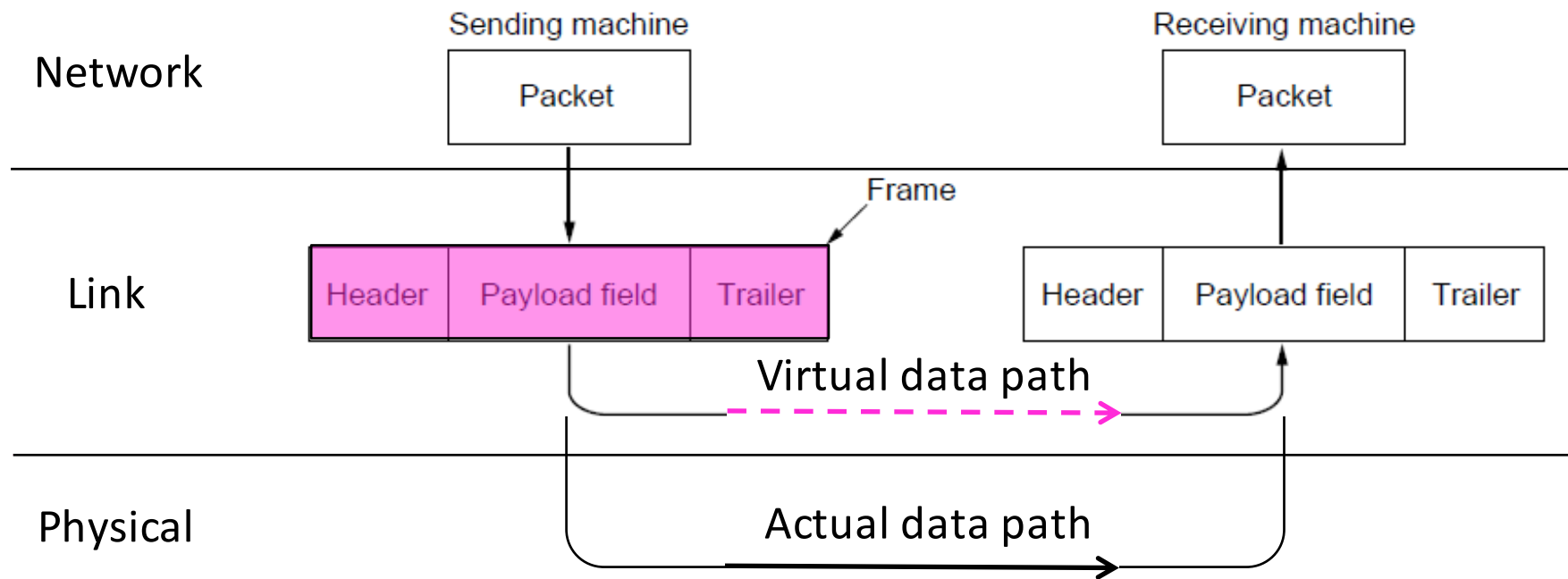
- Concerns how to transfer messages over one or more connected links
 - Messages are frames, of limited size
 - Builds on the physical layer



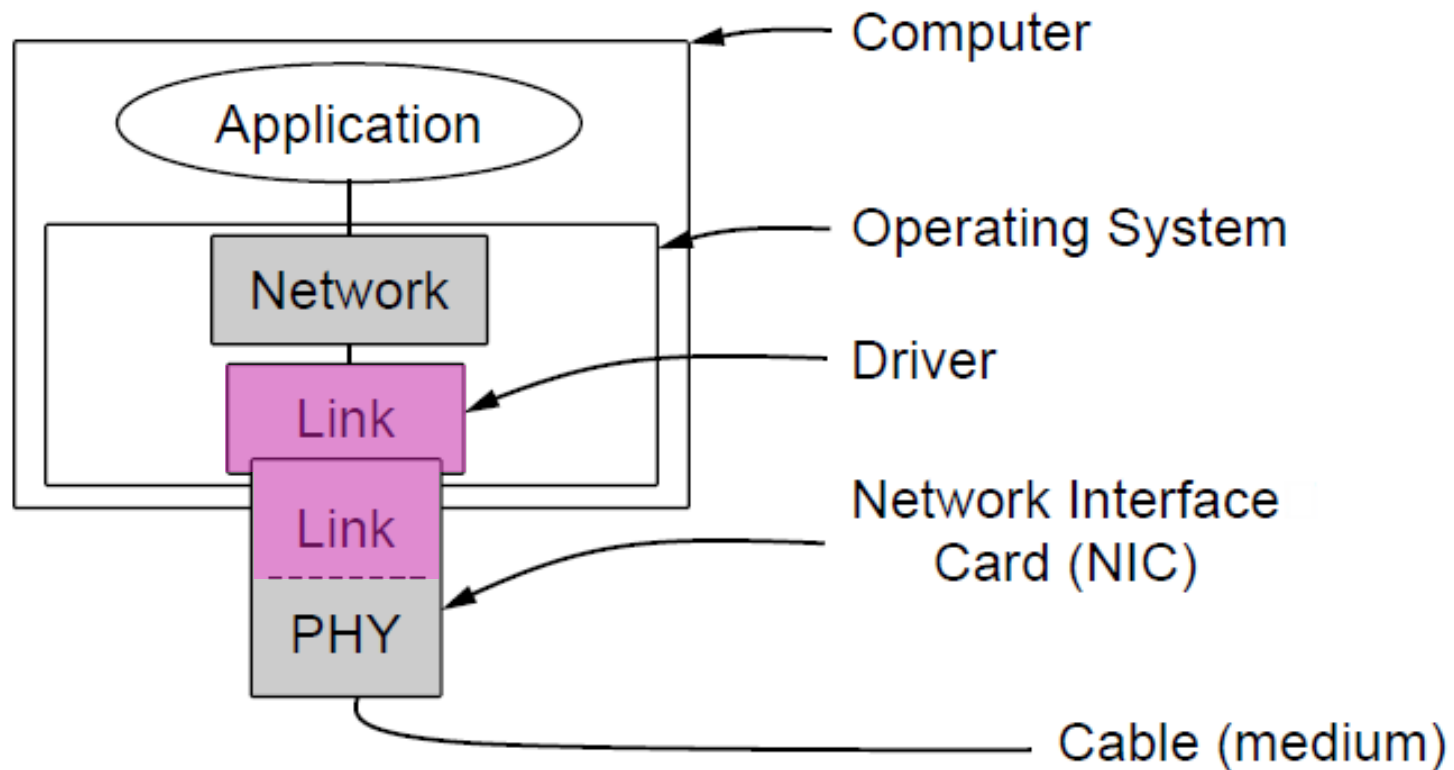
In terms of layers ...



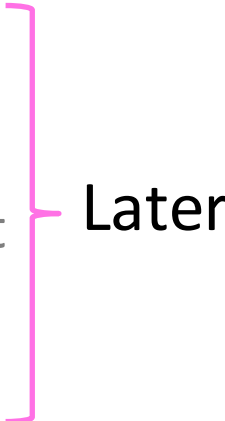
In terms of layers (2)



Typical Implementation of Layers

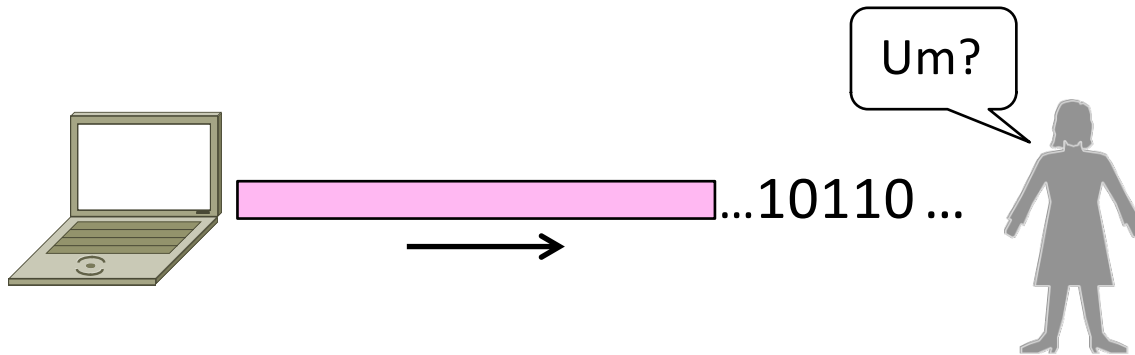


Topics

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
- 
- Later

Framing (§3.1.2)

- The Physical layer gives us a stream of bits. How do we interpret it as a sequence of frames?



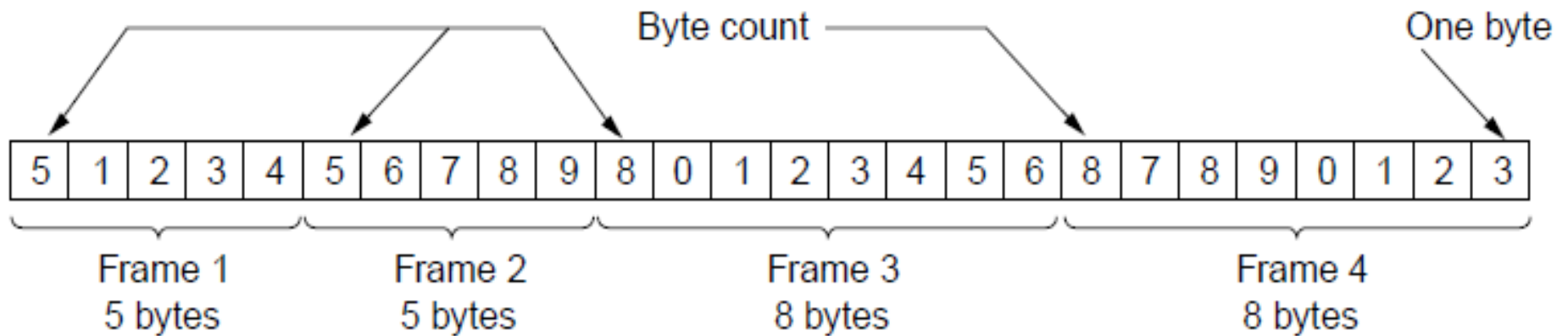
Framing Methods

- 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 ...

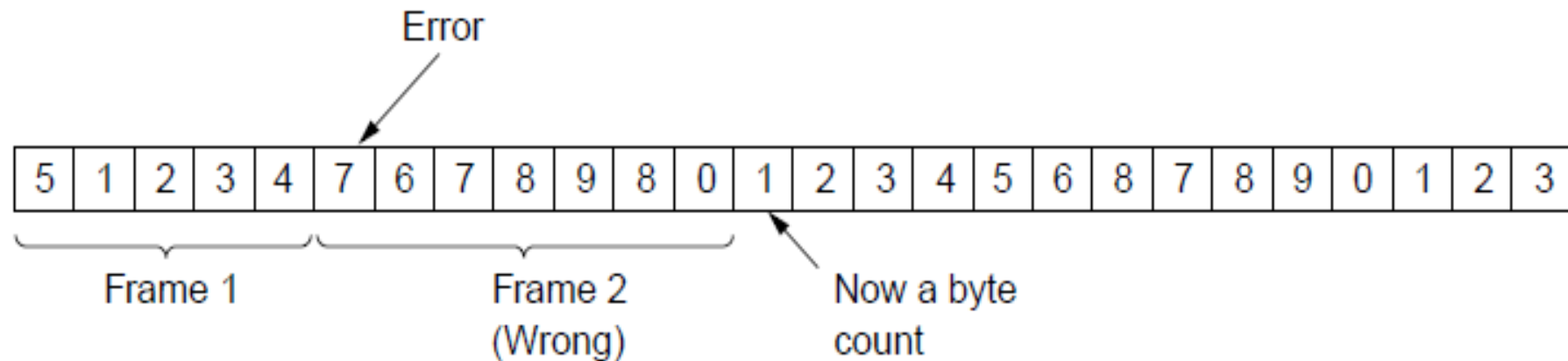
Byte Count (2)



- How well do you think it works?

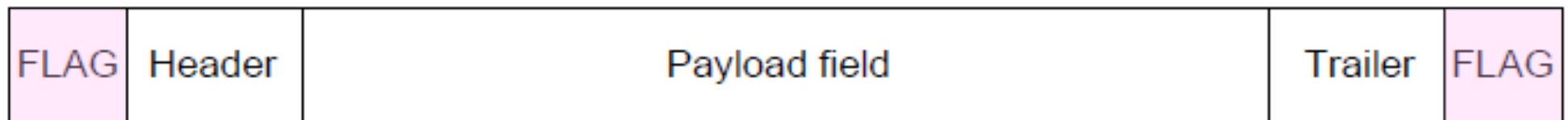
Byte Count (3)

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



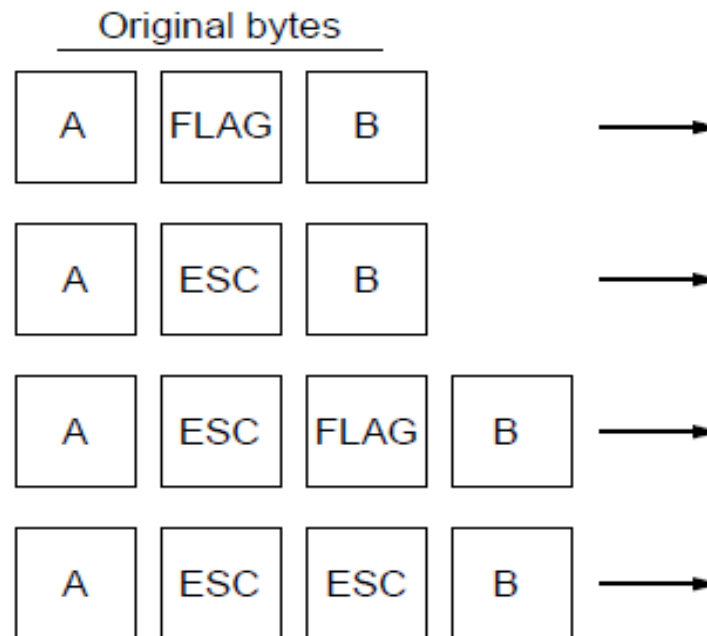
Byte Stuffing

- 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!



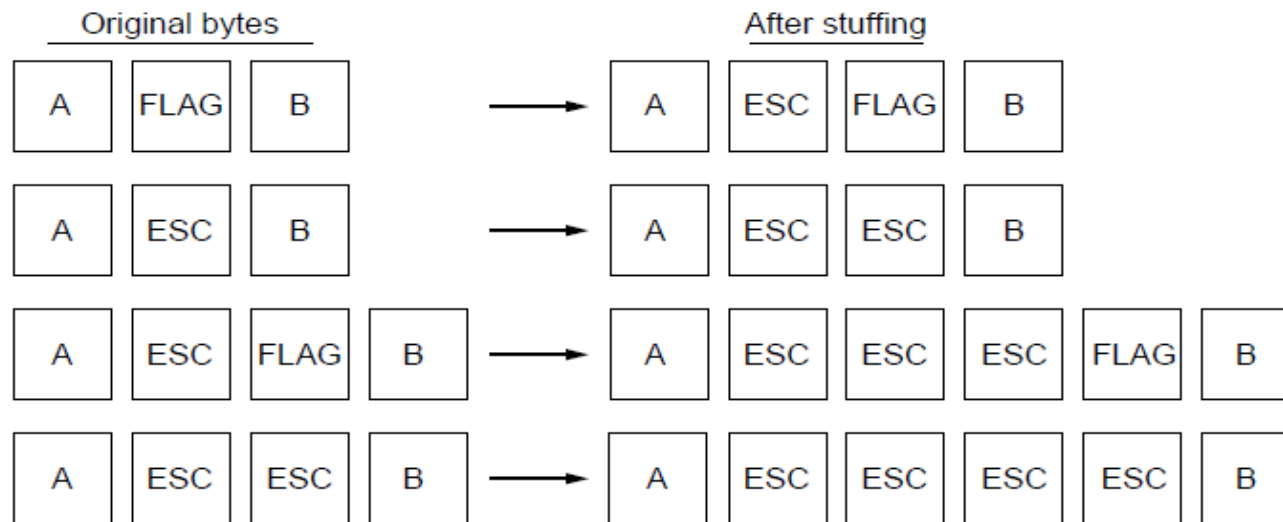
Byte Stuffing (2)

- 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

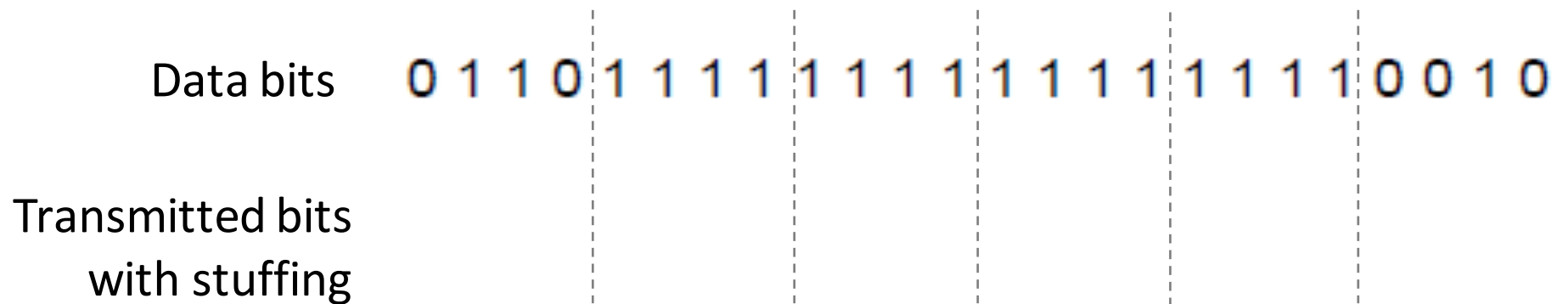


Bit Stuffing

- 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

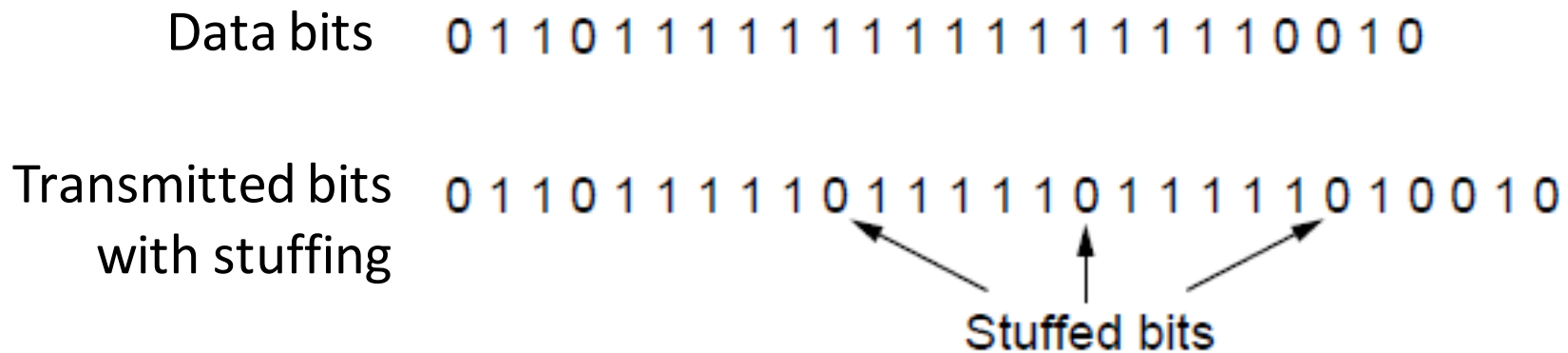
Bit Stuffing (2)

- Example:



Bit Stuffing (3)

- So how does it compare with byte stuffing?

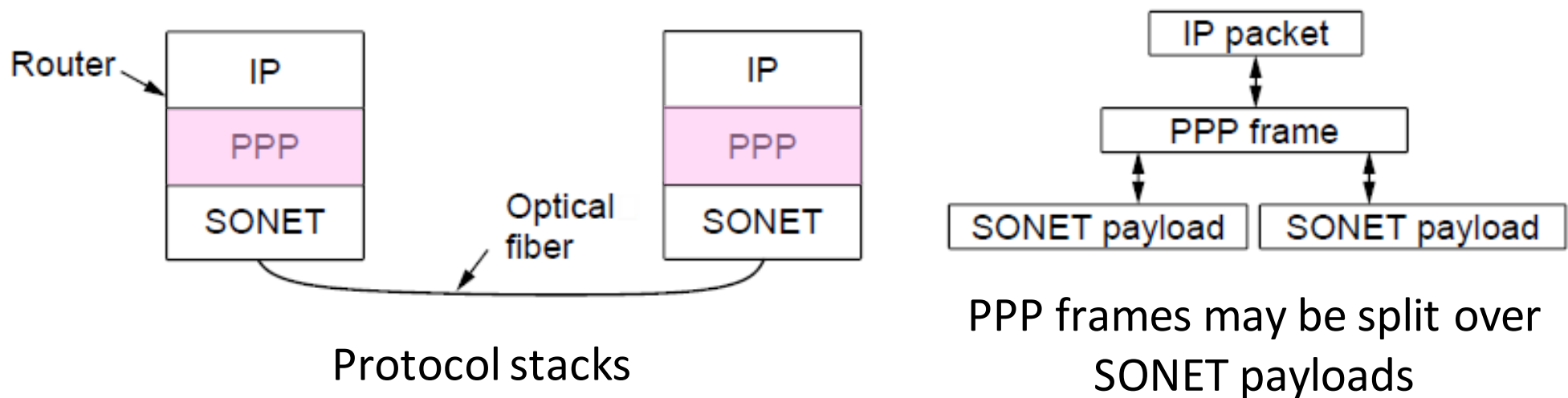


Link Example: PPP over SONET

- 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

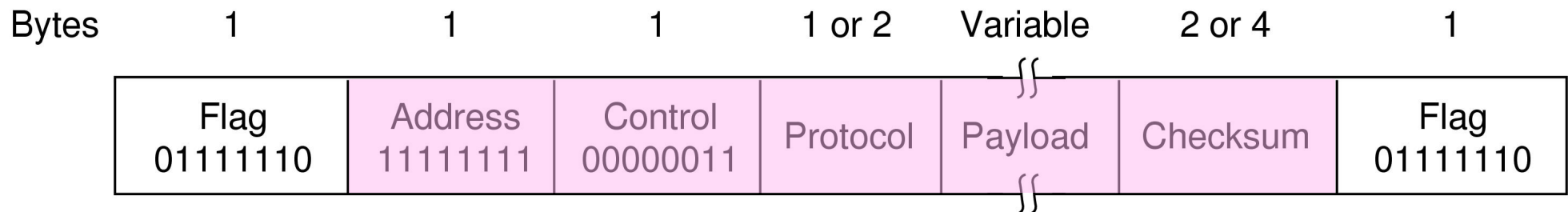
Link Example: PPP over SONET (2)

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



Link Example: PPP over SONET (3)

- Framing uses byte stuffing
 - FLAG is 0x7E and ESC is 0x7D



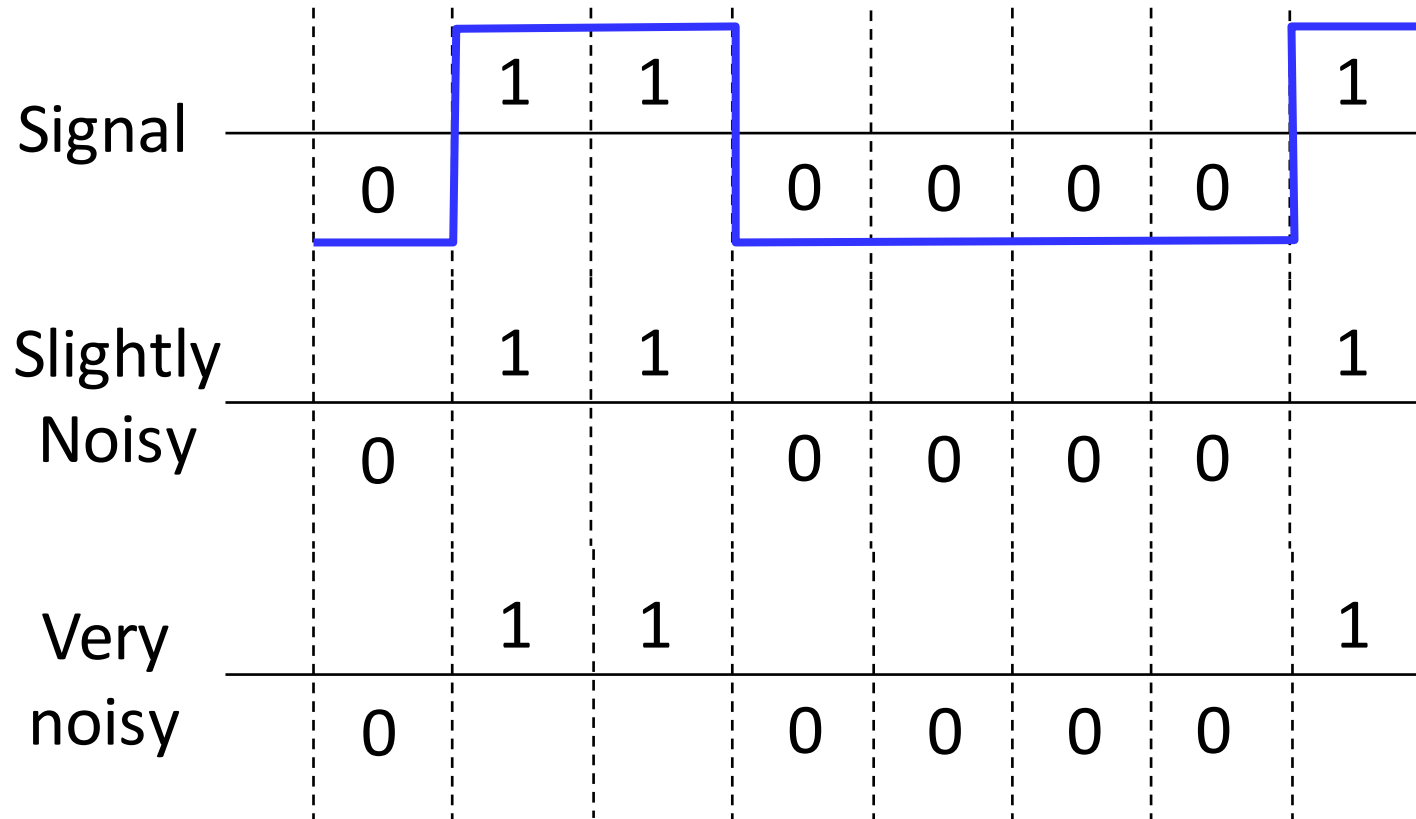
Link Example: PPP over SONET (4)

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

Error Coding Overview (§3.2)

- Some bits will be received in error due to noise. 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
 - Add check bits to the message bits to let some errors be detected
- Error correction codes
 - 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

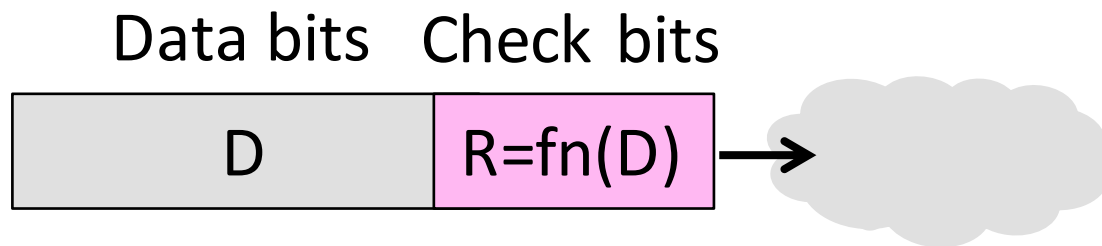
- A simple code to handle errors:
 - Send two copies! Error if different.
- How good is this code?
 - How many errors can it detect/correct?
 - How many errors will make it fail?

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

Using Error Codes

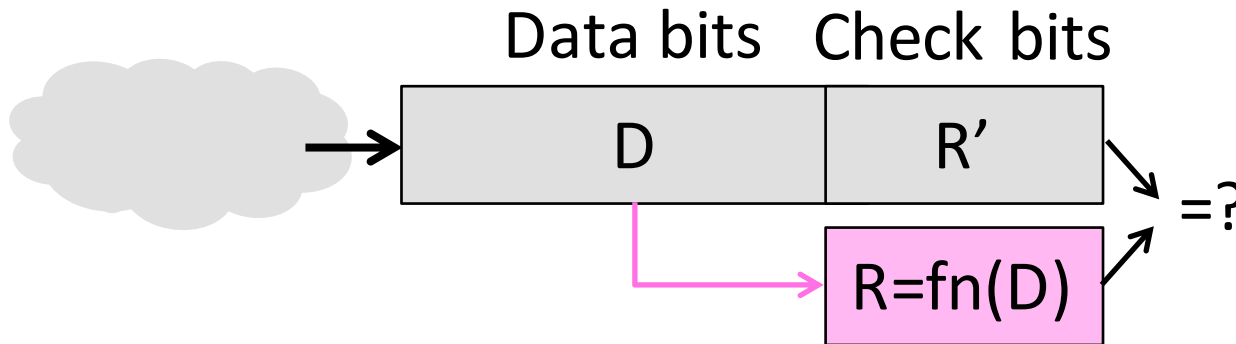
- Codeword consists of D data plus R check bits (=systematic block code)



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

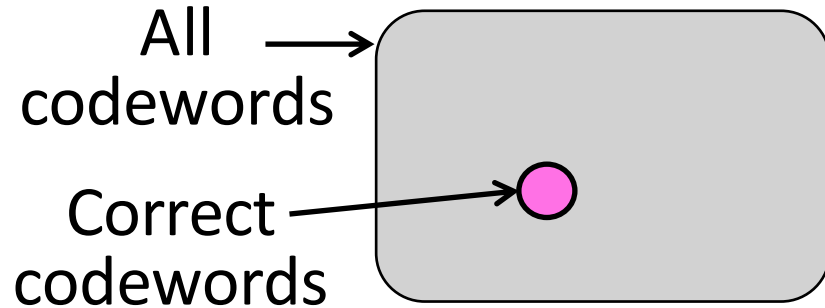
Using Error Codes (2)

- 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'



Intuition for Error Codes

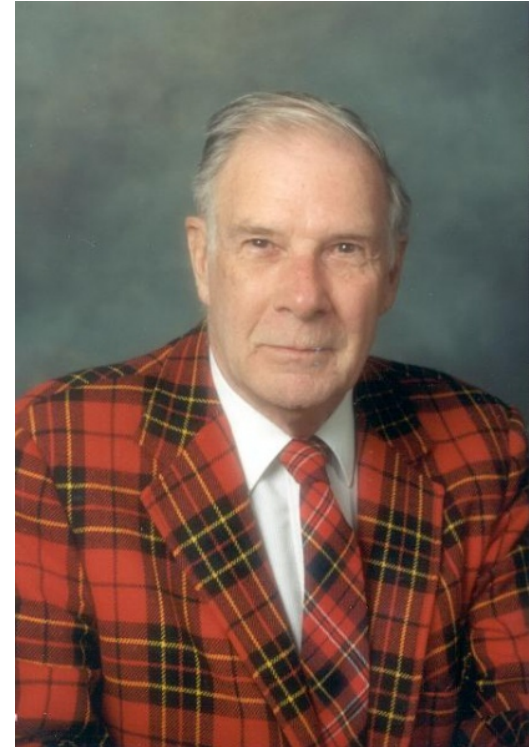
- For D data bits, R check bits:



- Randomly chosen codeword is unlikely to be correct; overhead is low

R.W. Hamming (1915-1998)

- Much early work on codes:
 - “Error Detecting and Error Correcting Codes”, BSTJ, 1950
- See also:
 - “You and Your Research”, 1986



Source: IEEE GHN, © 2009 IEEE

Hamming Distance

- Distance is the number of bit flips needed to change $D+R_1$ to $D+R_2$
- Hamming distance of a code is the minimum distance between any pair of codewords

Hamming Distance (2)

- Error detection:
 - For a code of Hamming distance $d+1$, up to d errors will always be detected

Hamming Distance (3)

- Error correction:
 - For a code of Hamming distance $2d+1$, up to d errors can always be corrected by mapping to the closest codeword

Error Detection (§3.2.2)

- 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

- Take D data bits, add 1 check bit that is the sum of the D bits
 - Sum is modulo 2 or XOR

Parity Bit (2)

- How well does parity work?
 - What is the distance of the code?
 - How many errors will it detect/correct?
- What about larger errors?

Checksums

- Idea: sum up data in N-bit words
 - Widely used in, e.g., TCP/IP/UDP



- Stronger protection than parity

Internet Checksum

- Sum is defined in 1s complement arithmetic (must add back carries)
 - And it's the negative sum
- *“The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words ...”* – RFC 791

Internet Checksum (2)

Sending:

0001
f203
f4f5
f6f7

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

Internet Checksum (3)

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 (4)

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 the result and check it is 0

```
0001
f203
f4f5
f6f7
+ 220d
-----
```

Internet Checksum (5)

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 the result and check it is 0

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

Internet Checksum (6)

- How well does the checksum work?
 - What is the distance of the code?
 - How many errors will it detect/correct?
- What about larger errors?

Cyclic Redundancy Check (CRC)

- Even stronger protection
 - Given n data bits, generate k check bits such that the $n+k$ bits are evenly divisible by a generator C
- Example with numbers:
 - Message = 302, k = one digit, $C = 3$

CRCs (2)

- The catch:
 - It's based on mathematics of finite fields, in which “numbers” represent polynomials
 - e.g., 10011010 is $x^7 + x^4 + x^3 + x^1$
- What this means:
 - We work with binary values and operate using modulo 2 arithmetic

CRCs (3)

- Send Procedure:
 1. Extend the n data bits with k zeros
 2. Divide by the generator value C
 3. Keep remainder, ignore quotient
 4. Adjust k check bits by remainder
- Receive Procedure:
 1. Divide and check for zero remainder

CRCs (4)

Data bits:
1101011111

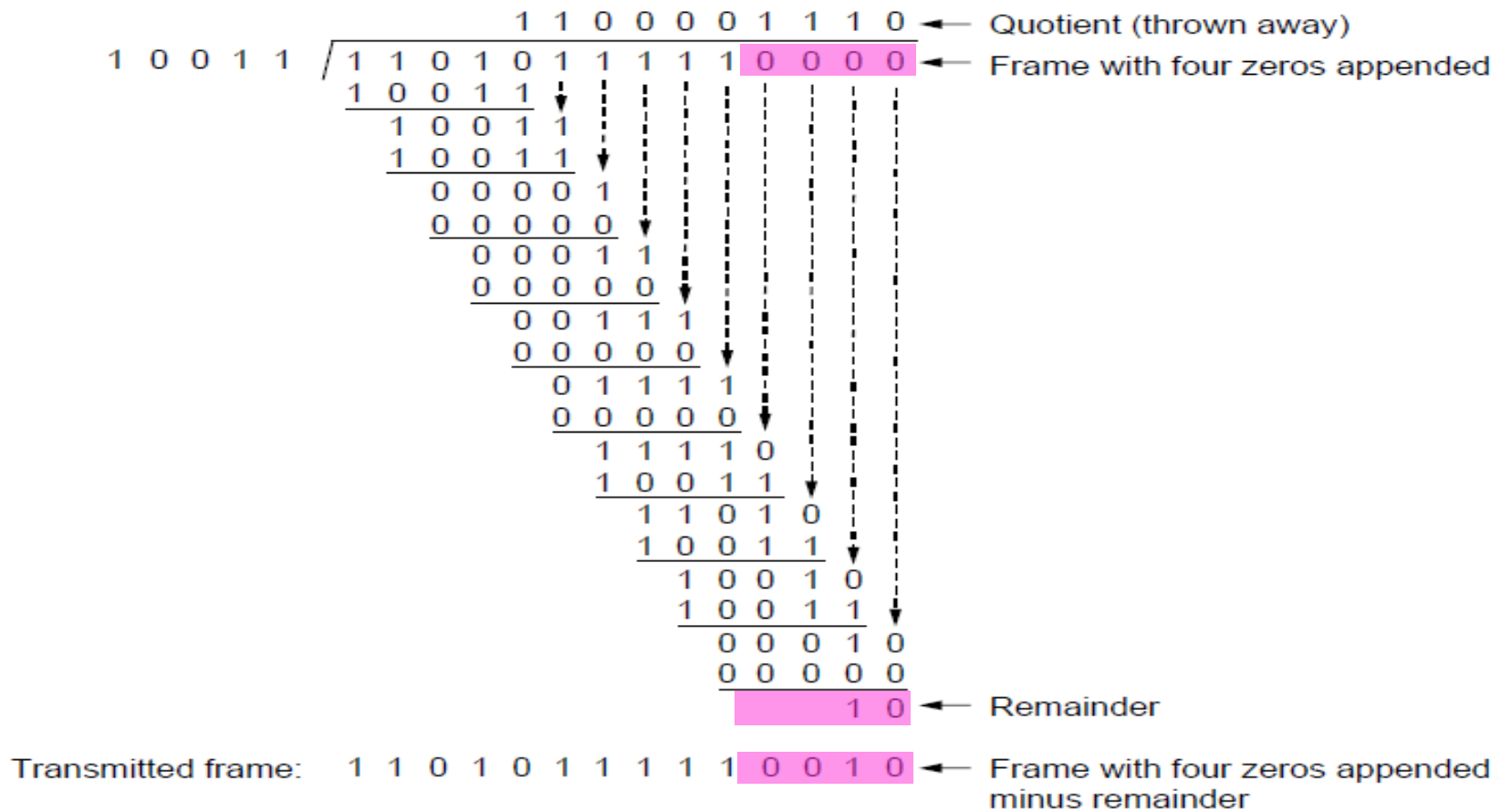
1 0 0 1 1 | 1 1 0 1 0 1 1 1 1 1

Check bits:
 $C(x) = x^4 + x^1 + 1$

$C = 10011$

$k = 4$

CRCs (5)



CRCs (6)

- Protection depend on generator
 - Standard CRC-32 is 1 0000 0100 1100 0001 0001 1101 1011 0111
- Properties:
 - HD=4, detects up to triple bit errors
 - Also odd number of errors
 - And bursts of up to k bits in error
 - Not vulnerable to systematic errors (i.e., moving data around) like checksums

Error Detection in Practice

- CRCs 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 (§3.2.1)

- 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

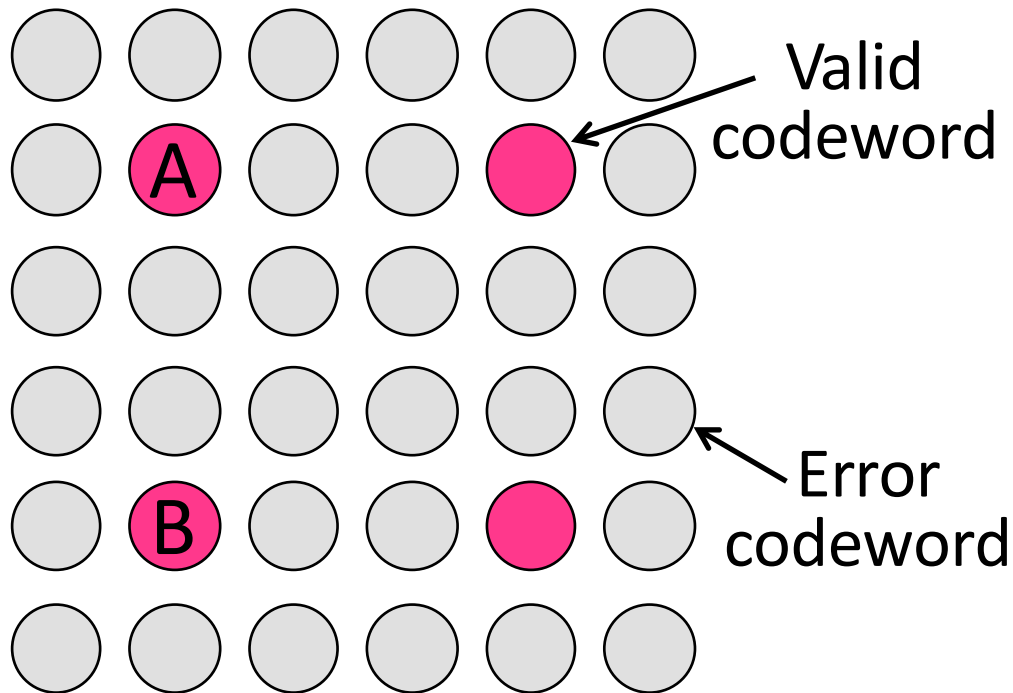
- If we had reliable check bits we could use them to narrow down the position of the error
 - Then correction would be easy
- But error could be in the check bits as well as the data bits!
 - Data might even be correct

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 $HD \geq 2d + 1$

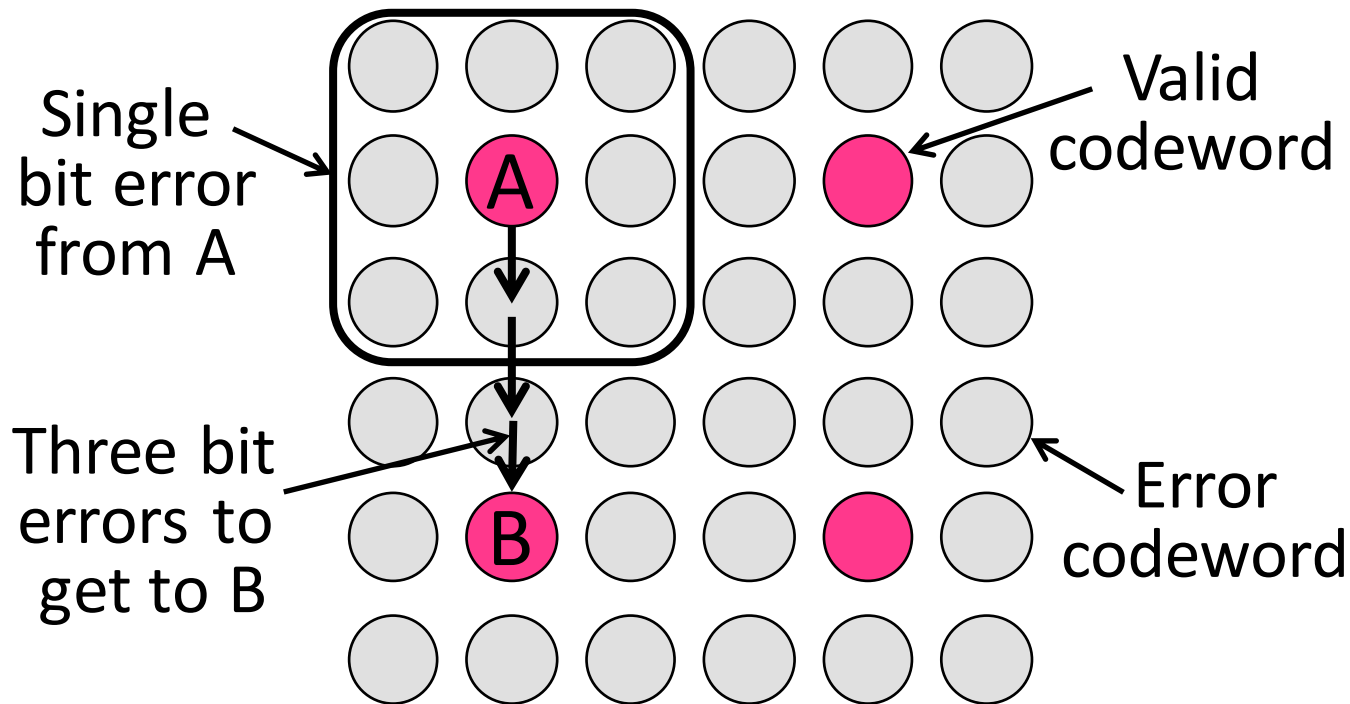
Intuition (2)

- 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 →
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

→ $\underline{0} \ \underline{1} \ 0 \ \underline{0} \ 1 \ \mathbf{1} \ 1$
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 = $\mathbf{1} \ \mathbf{1} \ 0$, flip position 6

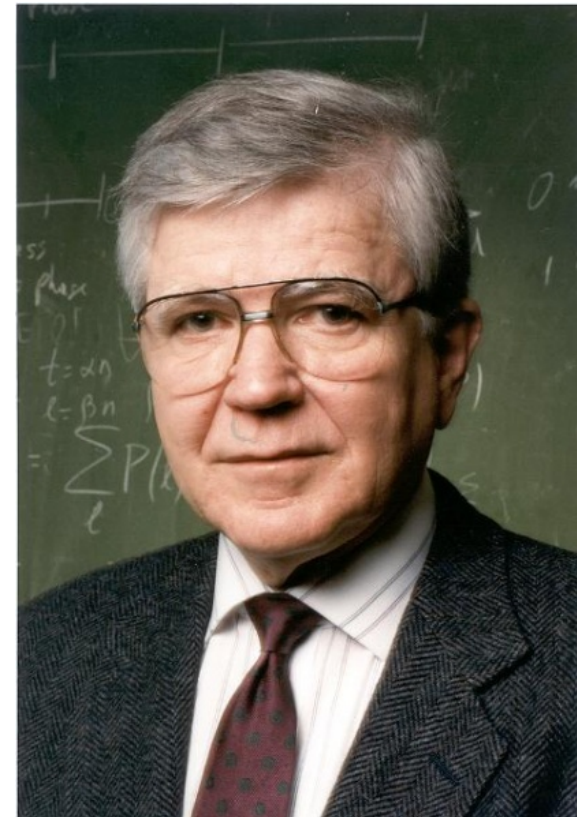
Data = 0 1 0 1 (correct after flip!)

Other Error Correction Codes

- Codes used in practice are much more involved than Hamming
- Convolutional codes (§3.2.3)
 - 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)

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

Detection vs. Correction

- Which is better will depend on the pattern of errors. For example:
 - 1000 bit messages with a bit error rate (BER) of 1 in 10000
- Which has less overhead?
 - It depends! We need to know more about the errors

Detection vs. Correction (2)

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

Detection vs. Correction (3)

2. Assume errors come in bursts of 100 consecutively garbled bits
 - Only 1 or 2 messages in 1000 have errors
- Error correction:
 - Need $\gg 100$ check bits per message
 - Overhead:
- Error detection:
 - Can use 32 check bits per message plus 1000 bit resend 2/1000 of the time
 - Overhead:

Detection vs. Correction (4)

- Error correction:
 - Needed when errors are expected
 - Small number of errors are correctable
 - Or when no time for retransmission
- Error detection:
 - More efficient when errors are not expected
 - And when errors are large when they do occur

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 (with retransmission) is used in the link layer and above for residual errors
- Correction also used in the application layer
 - Called Forward Error Correction (FEC)
 - Normally with an erasure error model (entire packets are lost)
 - E.g., Reed-Solomon (CDs, DVDs, etc.)