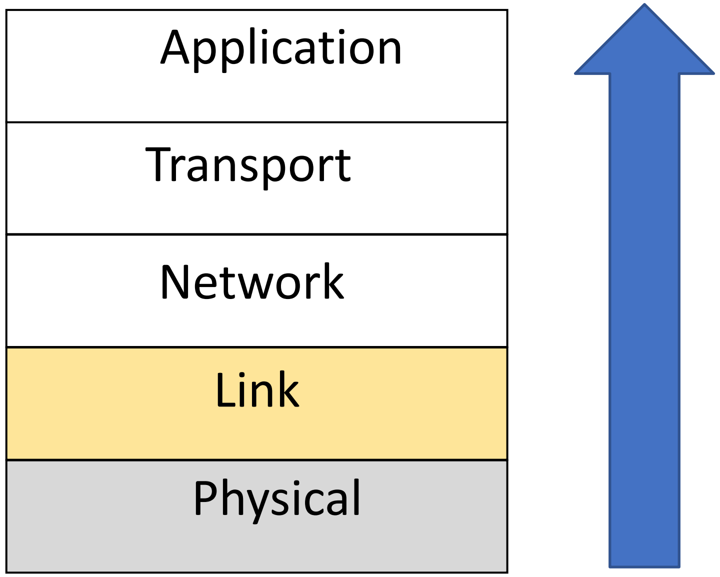
Where we left off…

- Deep in the physical layer
  - Encoding bits onto a physical medium in a way that allows for clock recovery and baseline recovery
  - Limits to how much data we can actually communicate within phy constraints of *bandwidth (Hz)* and *SNR (dB)*

While we're waiting– what were the key implications of Shannon capacity from last class?

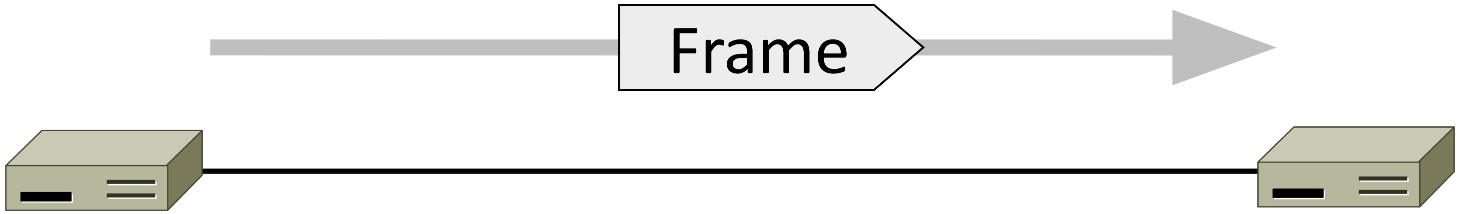
Where we are in the Course

- Today: moving on up to the Link Layer!

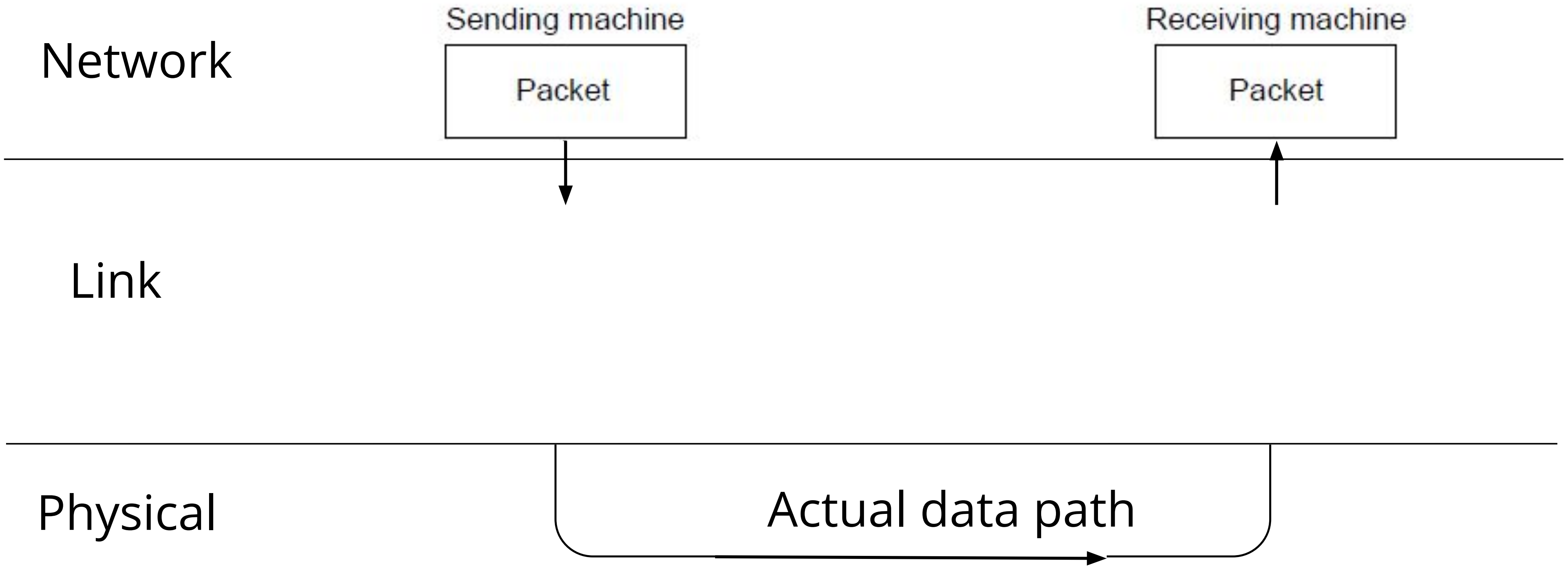


Scope of the Link Layer

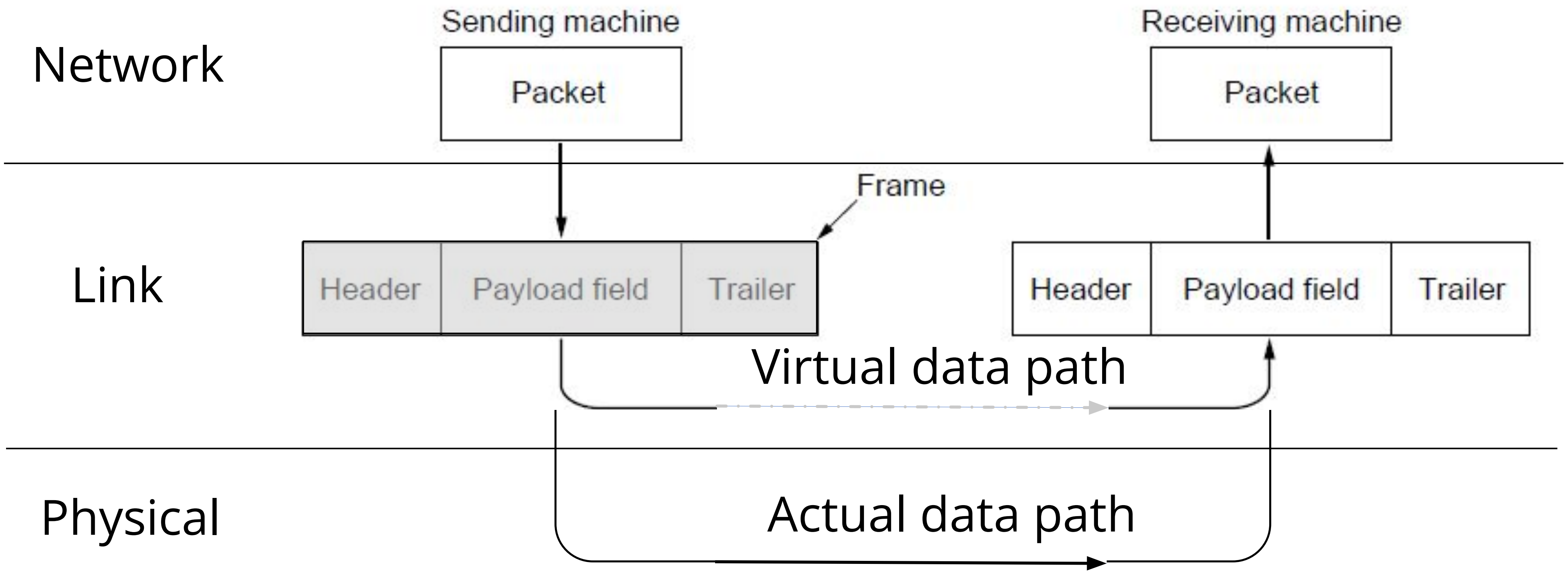
- Concerns how to transfer messages over one or more connected links
  - Messages are frames, of limited size
  - Builds on the “bits on a wire” abstraction provided by the phy!



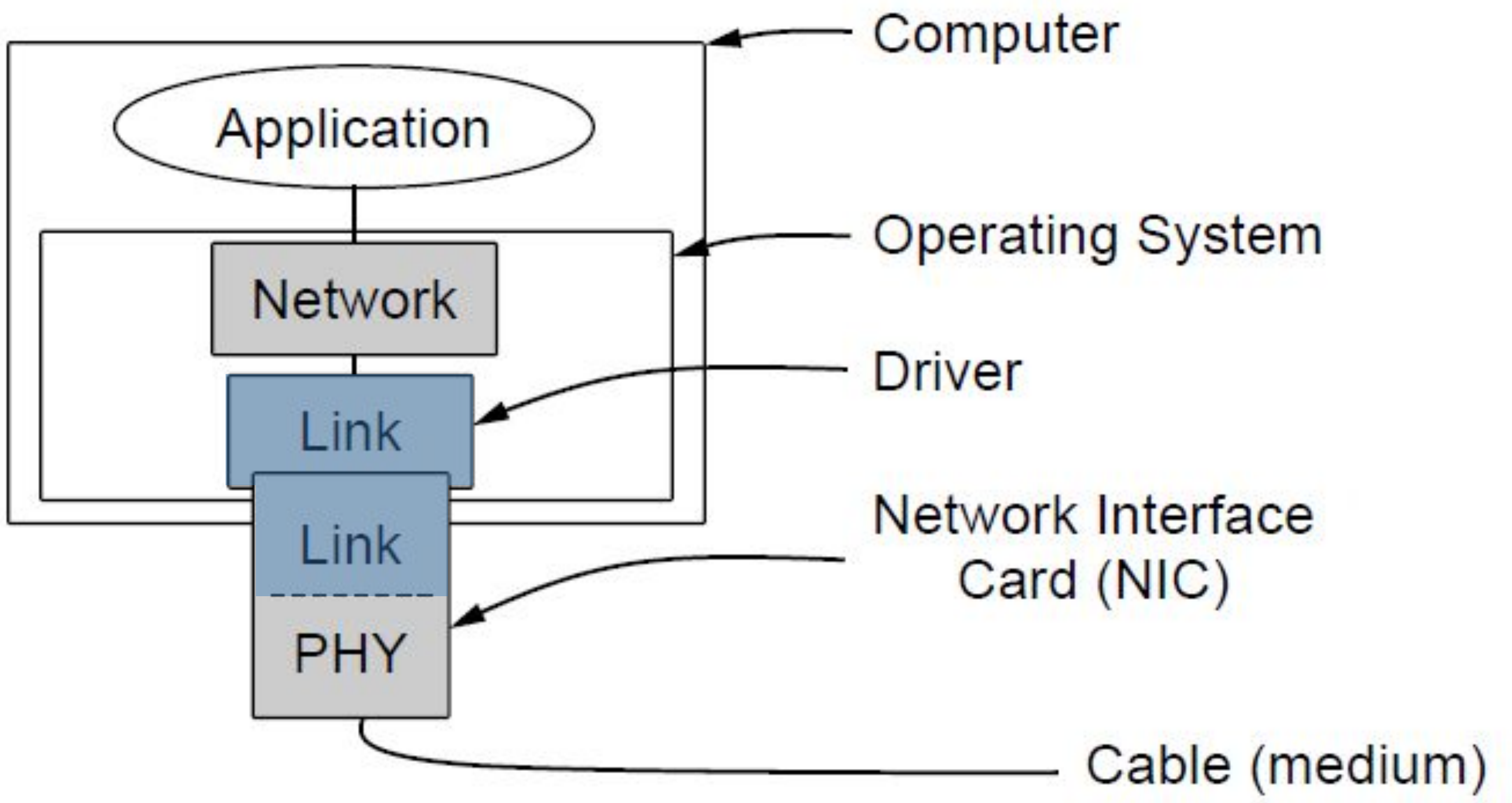
In terms of layers …



In terms of layers (2)



Typical Implementation of L2



L2 Topics

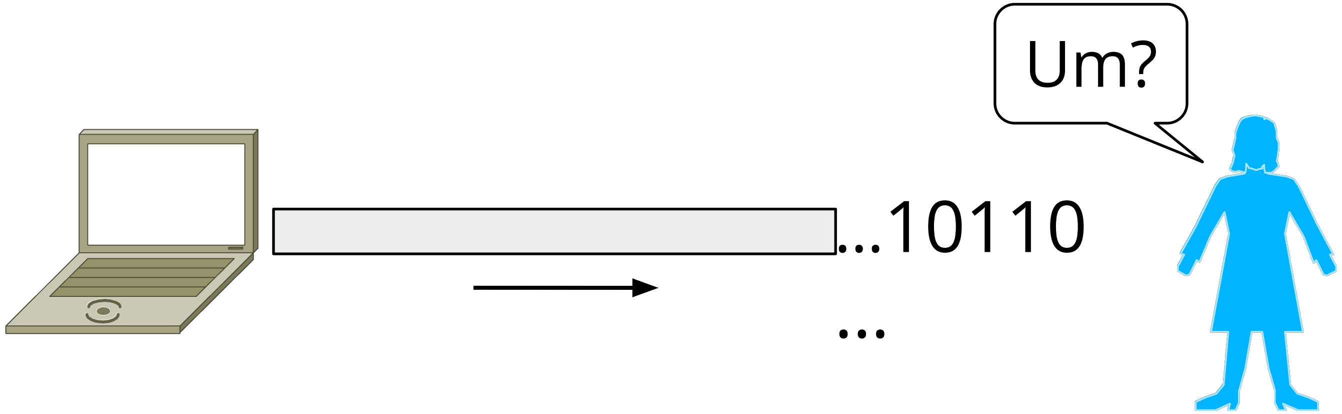
1. Framing
   - Delimiting start/end of frames
2. Error detection and correction
   - Handling errors
3. Retransmissions
   - Handling loss
4. Multiple Access
   - 802.11, classic Ethernet
5. Switching
   - Modern Ethernet

Framing

Delimiting start/end of frames

Topic

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



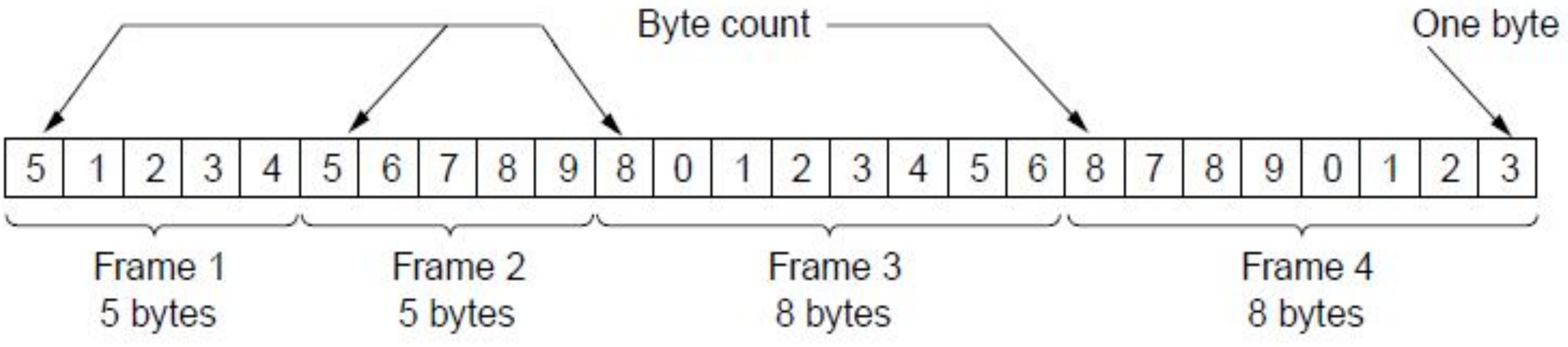
Framing Methods

- We'll look at:
  - Byte count (motivation)
  - Byte stuffing
  - Bit stuffing
- The book also discusses clock-based framing (2.3.3)
  - Happy to discuss on Ed or office hours if of interest
- Note: in practice, the physical layer often helps to identify and/or confirm frame boundaries
  - E.g., Ethernet, 802.11
  - Detect "gaps" in the analog signal, clock, etc.

Byte Count

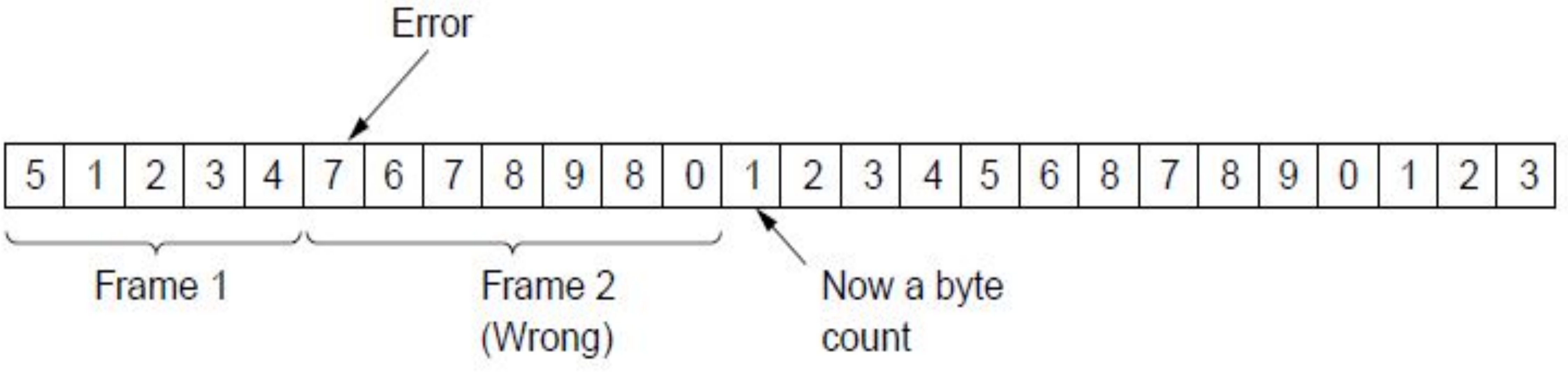
- First try:
  - Let’s start each frame with a length field!
  - It’s simple, and hopefully good enough …

Byte Count (2)


Byte count
One byte
5 1 2 3 4 5 6 7 8 9 8 0 1 2 3 4 5 6 8 7 8 9 0 1 2 3
Frame 1
5 bytes
Frame 2
5 bytes
Frame 3
8 bytes
Frame 4
8 bytes

Byte Count (3)

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



Byte Stuffing (1)

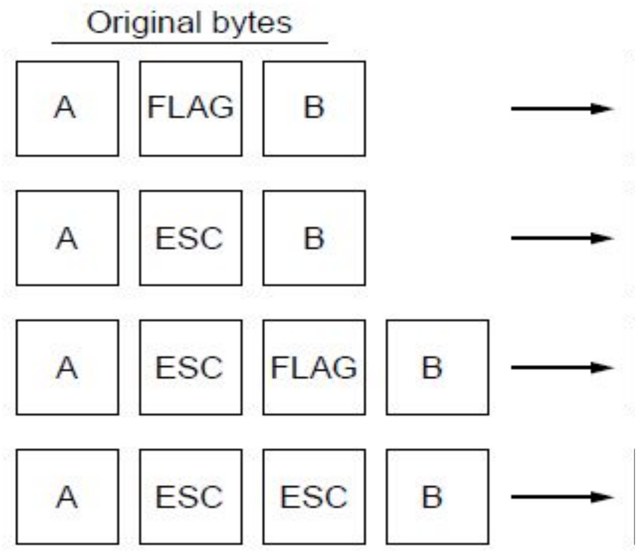
- Different idea:
  - Have a special flag byte value for start/end of frame
  - Replace (“stuff”) the flag with an escape code
  - Problem?

Byte Stuffing (2)

- Different idea:
  - Have a special flag byte value for start/end of frame
  - Replace ("stuff") the flag with an escape code
- Complication: have to escape the escape code too!

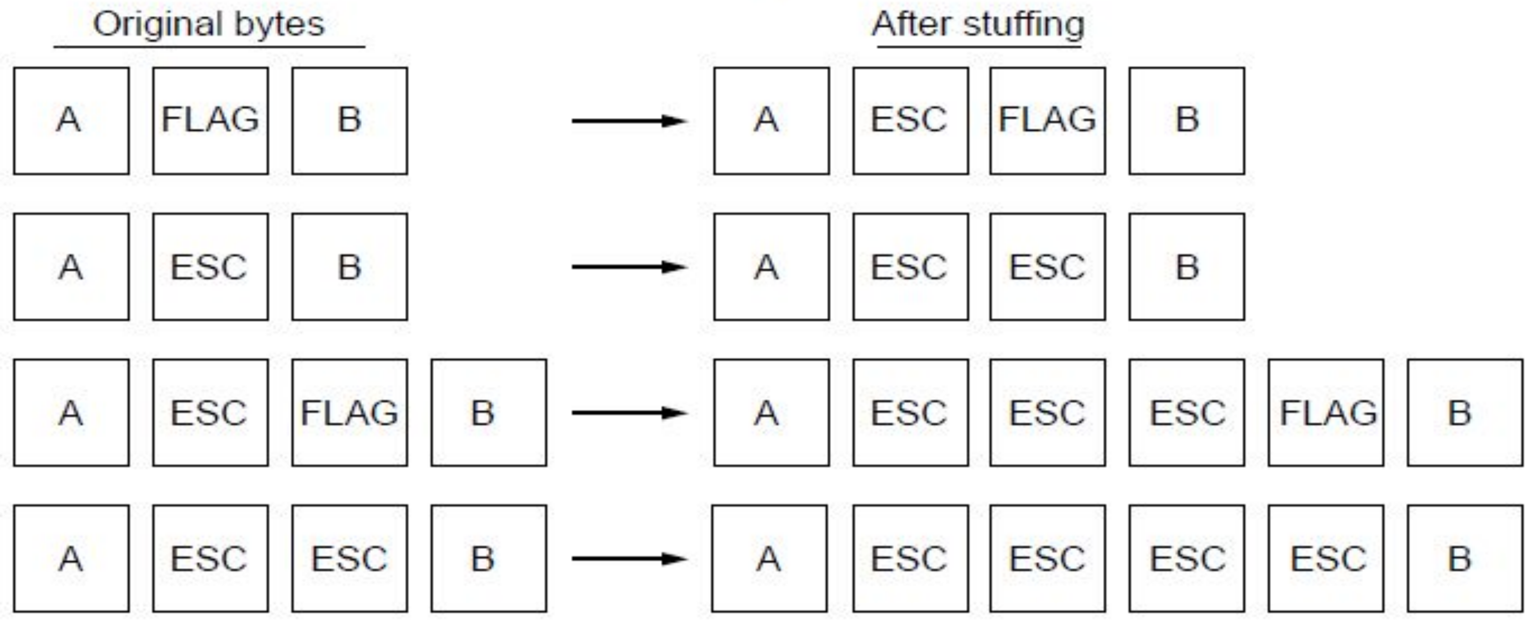
Byte Stuffing (3)

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



Byte Stuffing (4)

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



Unstuffing

You see:

1. Solitary FLAG?
2. Solitary ESC?
3. ESC FLAG?
4. ESC ESC FLAG?
5. ESC ESC ESC FLAG?
6. ESC FLAG FLAG?

Unstuffing

You see:

1. Solitary FLAG? -> Start or end of frame
2. Solitary ESC? -> Bad frame!
3. ESC FLAG? -> pass FLAG through
4. ESC ESC FLAG? -> pass ESC through, then start or end of frame
5. ESC ESC ESC FLAG? -> pass ESC FLAG through
6. ESC FLAG FLAG? -> pass FLAG through then start or end of 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 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 Example (2)

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

How does it compare to byte stuffing???

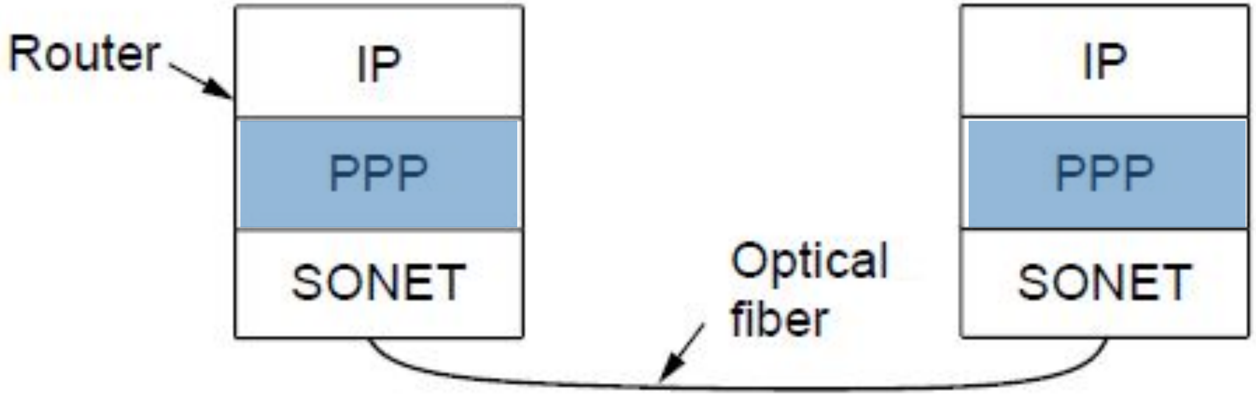
Possible small gain in efficiency, at the expense of byte alignment : (

Link Example: PPP over SONET

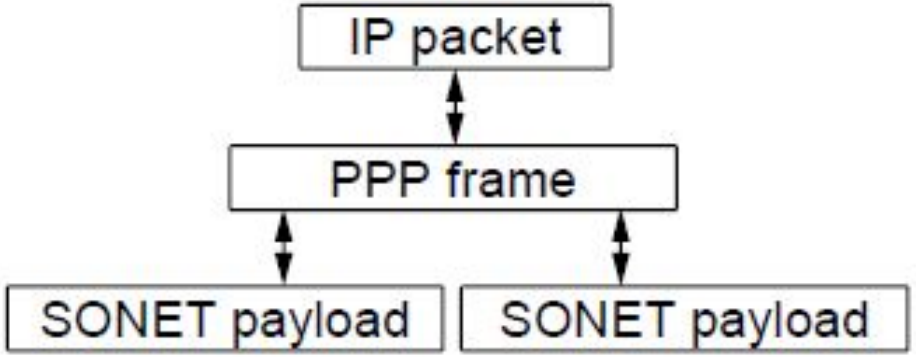
- PPP is Point-to-Point Protocol
  - Widely used for link framing
  - E.g., it is used to frame variable-length IP packets that are sent over SONET optical links (which have a fixed frame size!)

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



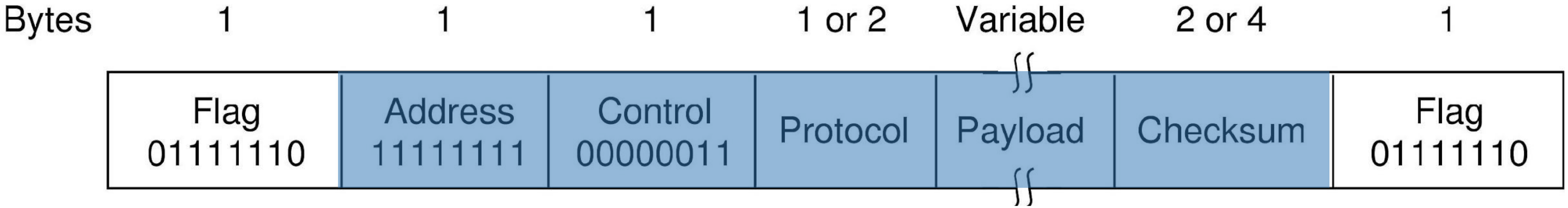
Protocol stacks



PPP frames may be split over SONET payloads

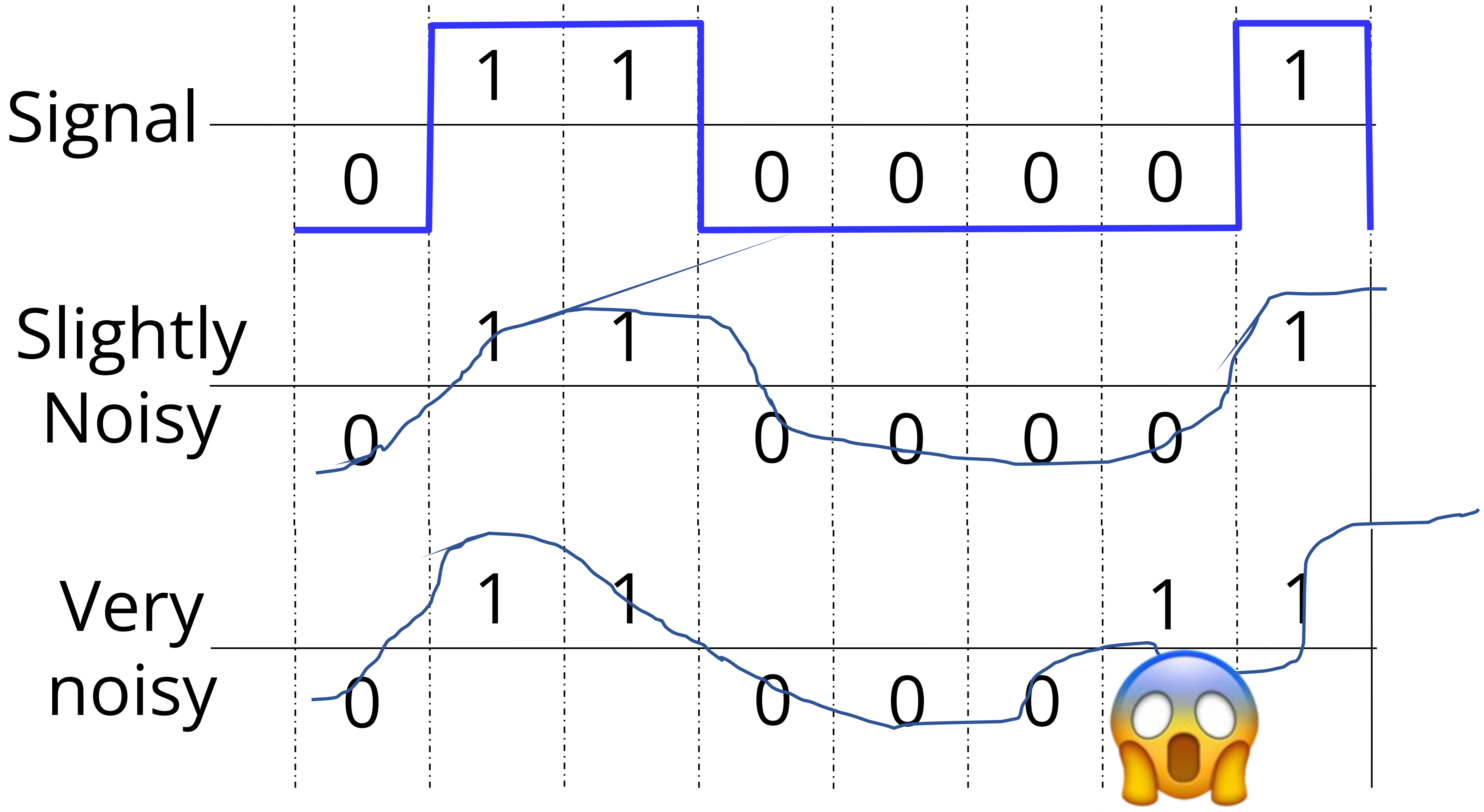
Link Example: PPP over SONET (3)

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



Link Layer: Error detection and correction

Problem – Noise may flip received bits



UW CSE-461
27

Topic

- Some bits will be received in error due to noise. 😱
- What can we do?
  - Detect errors with codes
  - Retransmit lost frames
  - Correct errors with codes

Later

- Reliability is a concern that cuts across the layers!

Approach – Add Redundancy

- Error detection codes
  - Add <u>check bits</u> to the message bits to let some errors be detected
- Error correction codes
  - Add even more <u>check bits</u> 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

Ideas?

Motivating Example

- A simple code to handle errors:
  - Send two copies! Error if different.

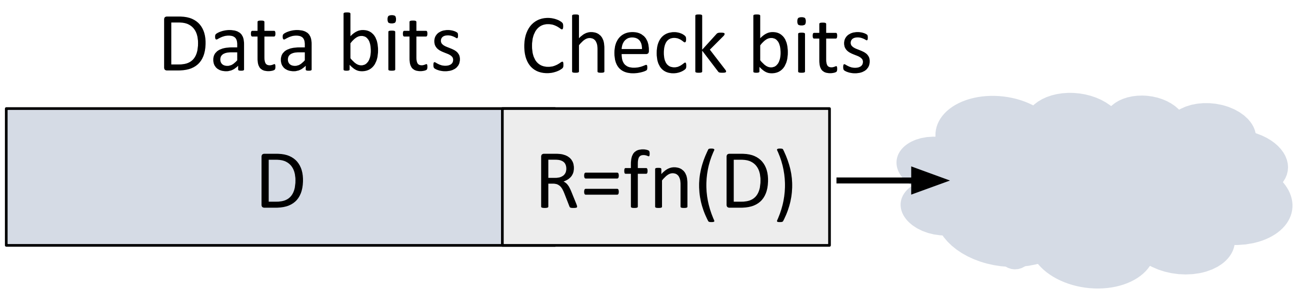
- How good is this code?
  - How many errors can it detect/correct?
  - How many errors will make it fail?

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 (non-malicious) errors
    - Will look at secure hashes later

Using Error Codes

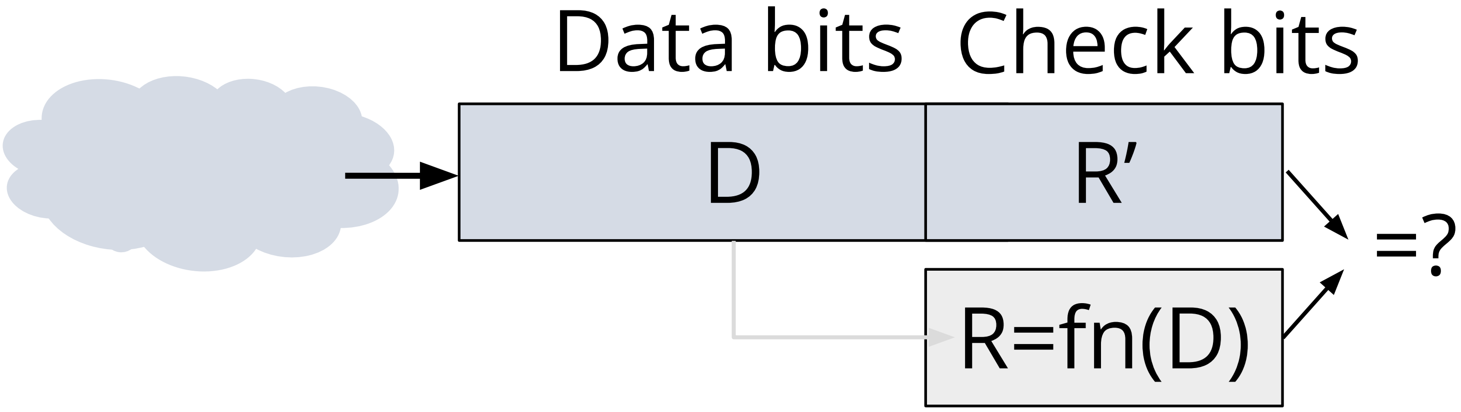
- Codeword consists of D data plus R check bits
  - =systematic block code



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

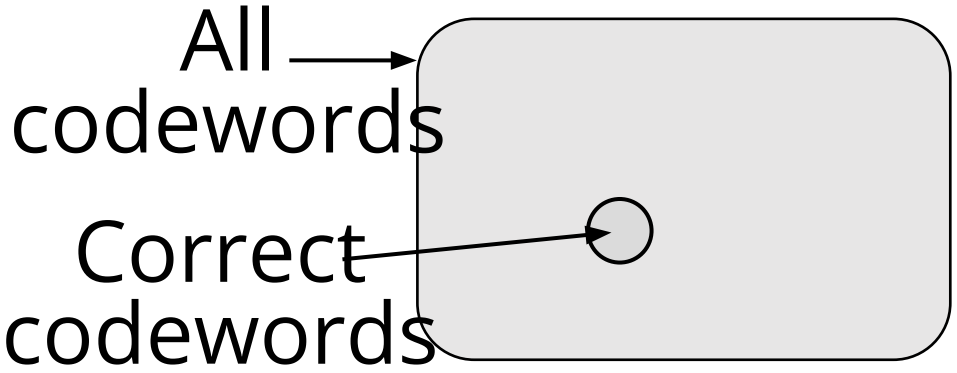
Using Error Codes (2)

- 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

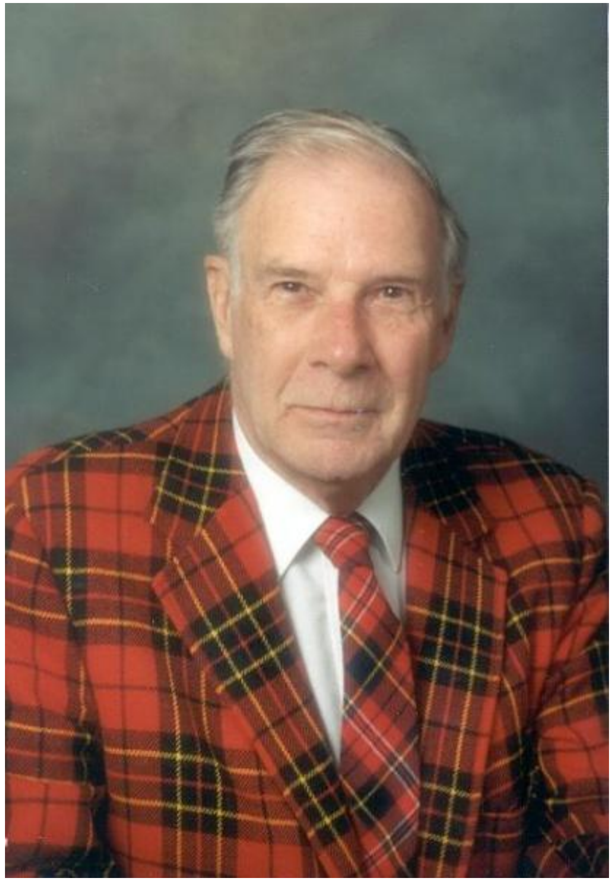
- For D data bits, R check bits:



- 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
- *“If the computer can tell when an error has occurred, surely there is a way of telling where the error is so the computer can correct the error itself”*
  - Hamming



Source: IEEE GHN, © 2009 IEEE

Fun Fact:
Shared an office with Claude Shannon (from last class) at Bell Labs!

Hamming Distance

- Hamming distance **between two codes** (D_1 D_2) is the number of bit flips needed to change D_1 to D_2

Alternatively (and confusingly)…

- Hamming distance of a **coding** is the minimum error distance between any pair of codewords (bit-strings) that cannot be detected

Hamming Distance (2)

- Error detection:
  - For a coding of distance d+1, up to d errors will always be *detected*

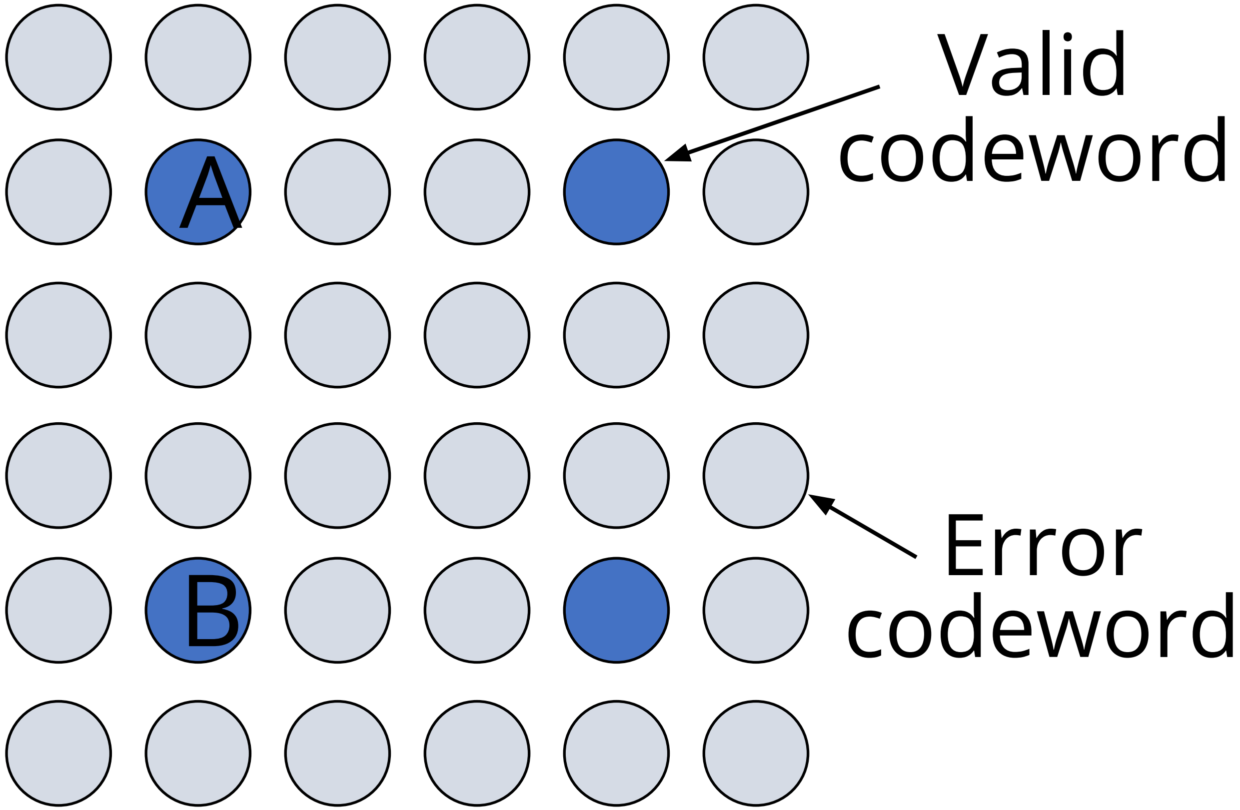
- Error correction:
  - For a coding of distance 2d+1, up to d errors can always be *corrected* by mapping to the closest valid codeword

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

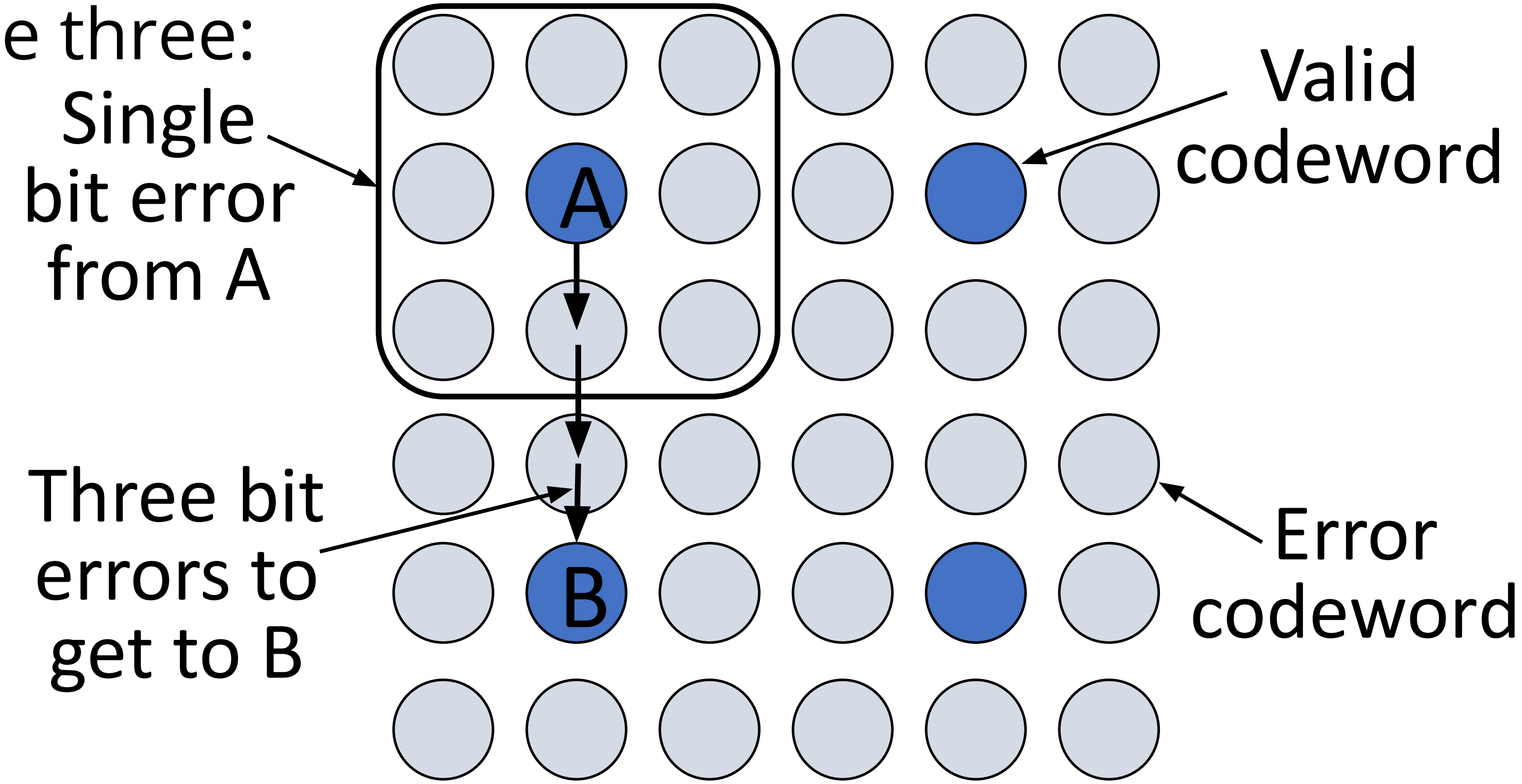
Intuition (2)

- Visualization of code w/ Hamming distance three:



Intuition (3)

- Visualization of code w/ Hamming distance three:



You cannot have it both ways… receiver needs to pick if receiving in the detection or correction mode : /

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 Hamming distance of the code?
  - How many errors will it reliably detect/correct?

2

Detect 1, correct 0

- What about larger errors?

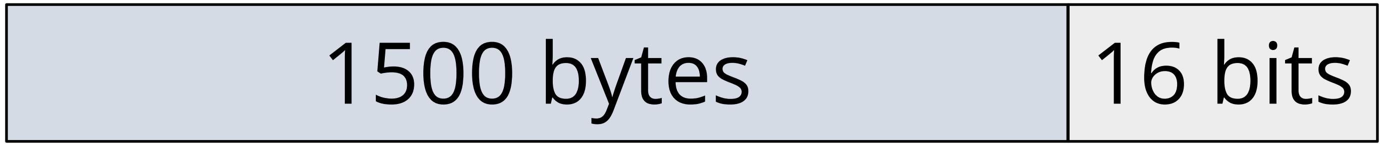
Can detect all odd number of errors!

Check your understanding…

- What is the Hamming distance of the duplicate message code we used as a motivating example?

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: `0x0001f204f4f5f6f7`

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
   f204
   f4f5
   f6f7
+(0000)
 ------
  2ddf1
   ↓
   ddf1
 +    2
 ------
   ddf3
   ↓
   220c
```

Internet Checksum (3)

Receiving: `0x0001f204f4f5f6f7220c`

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
  f204
  f4f5
  f6f7
+ 220c
------
 2fffd
   ↓
  fffd
+    2
------
  ffff
   ↓
  0000 ✅
```

Internet Checksum (4)

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

- What about larger errors?

Internet Checksum (5)

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

- What about larger errors?

Hamming distance of 2 : (Fooled by two errors in certain positions! Same as humble parity bit

But does handle bursts of up to 16 bits, and large random errors have lower probability of passing (1/2^16)

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 : (

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
- AND provides an algorithm to determine where to correct!

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	1
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	_	0	_	1	0	1
1	2	3	4	5	6	7

→ $p_1 = 0+1+1 = 0$

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 _ 1 0 1
1 2 3 4 5 6 7

→

$p_1 = 0+1+1 = 0$

$p_2 = 0+0+1 = 1$

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

→

$p_1 = 0+1+1 = 0$

$p_2 = 0+0+1 = 1$

$p_4 = 1+0+1 = 0$

Hamming Code (4)

- To decode:

1. Recompute check bits
   - (with parity sum *including* the check bit)
2. Arrange as a binary number
3. Value (syndrome) tells error position
4. Value of zero means no error
5. Otherwise, flip bit to correct

Hamming Code (5)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

$\underline{0}$ $\underline{1}$ 0 $\underline{0}$ 1 0 1
1 2 3 4 5 6 7

p_1=

p_2=

p_4=

Syndrome =

Data = 0101

Hamming Code (8)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

<u>0</u> <u>1</u> 0 <u>0</u> 1 0 1
1 2 3 4 5 6 7

p_1=

p_2=

p_4= 0+1+0+1 = 0

Syndrome = 0

Data = 0101

Hamming Code (8)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

<u>0</u> <u>1</u> 0 <u>0</u> 1 0 1
1 2 3 4 5 6 7

p_1=

p_2= 1+0+0+1 = 0

p_4= 0+1+0+1 = 0

Syndrome = 00

Data = 0101

Hamming Code (8)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

$\underline{0}$ $\underline{1}$ 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 = 0101

Hamming Code (9)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

<u>0</u> <u>1</u> 0 <u>0</u> 1 **1** 1
1 2 3 4 5 6 7

p_1=

p_2=

p_4=

Syndrome =

Data =

Hamming Code (10)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

💥
0 1 0 0 1 **1** 1
1 2 3 4 5 6 7

p_1=

p_2=

p_4=

Syndrome =

Data = 0 1 1 1

Hamming Code (11)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

<u>0</u> <u>1</u> 0 <u>0</u> 1 **1** 1
1 2 3 4 5 6 7

p_1=

p_2=

p_4= 0+1+**1**+1 = **1**

Syndrome = **1**

Data = 0 1 1 1

Hamming Code (12)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

$\underline{0}$ $\underline{1}$ 0 $\underline{0}$ 1 **1** 1
1 2 3 4 5 6 7

p_1=

p_2= 1+0+**1**+1 = **1**

p_4= 0+1+**1**+1 = **1**

Syndrome = **1 1**

Data = 0 1 1 1

Hamming Code (13)

C_1 = 1,3,5,7
C_2 = 2,3,6,7
C_3 = 4,5,6,7

💥

<u>0</u> <u>1</u> 0 <u>0</u> 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 = **1 1** 0, flip position 6

Data = 0 1 0 1 (correct after flip 🤯)

Hamming Code (14)

No magic, the parity bits are just cleverly constructed to address the failing common bit!

- 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	**1**	1
1	2	3	4	5	6	7

→

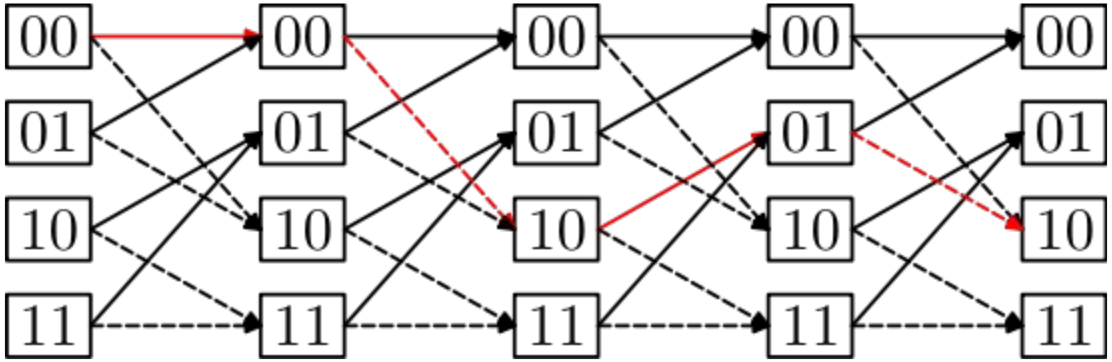
$p_1 = 0+0+1+1 = 0$

$p_2 = 1+0+\mathbf{1}+1 = \mathbf{1}$

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

Other Error Correction Codes

- Many commonly used codes are more involved than Hamming
  - (Hamming still used for ECC ram since very easy to implement in HW)
- E.g., Convolutional codes
  - 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)



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:
  - n = 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:

5. Divide by C
6. Check for zero remainder

CRCs (4)

Data bits:

1101011111

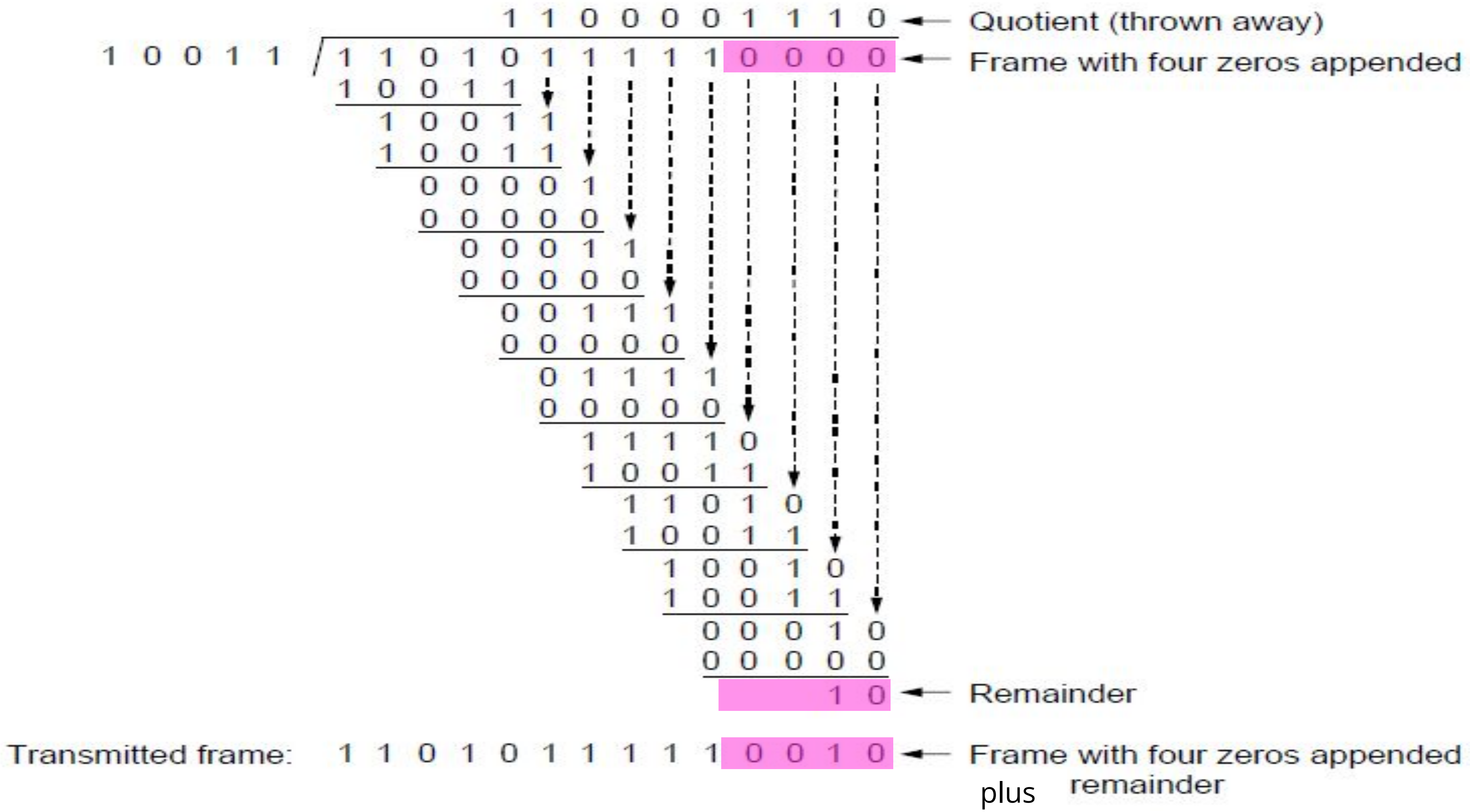
Check bits:

$C(x)=x^4+x^1+1$

C = 10011

k = 4

CRCs (5)



1 1 0 0 0 0 1 1 1 0
Quotient (thrown away)
1 0 0 1 1
1 1 0 1 0 1 1 1 1 1 0 0 0 0
Frame with four zeros appended
1 0 0 1 1
1 0 0 1 1
1 0 0 1 1
0 0 0 0 1
0 0 0 0 0
0 0 0 1 1
0 0 0 0 0
0 0 1 1 1
0 0 0 0 0
0 1 1 1 1
0 0 0 0 0
1 1 1 1 0
1 0 0 1 1
1 1 0 1 0
1 0 0 1 1
1 0 0 1 0
1 0 0 1 1
0 0 0 1 0
0 0 0 0 0
1 0
Remainder
Transmitted frame: 1 1 0 1 0 1 1 1 1 1 0 0 1 0
Frame with four zeros appended plus remainder

CRCs (6)

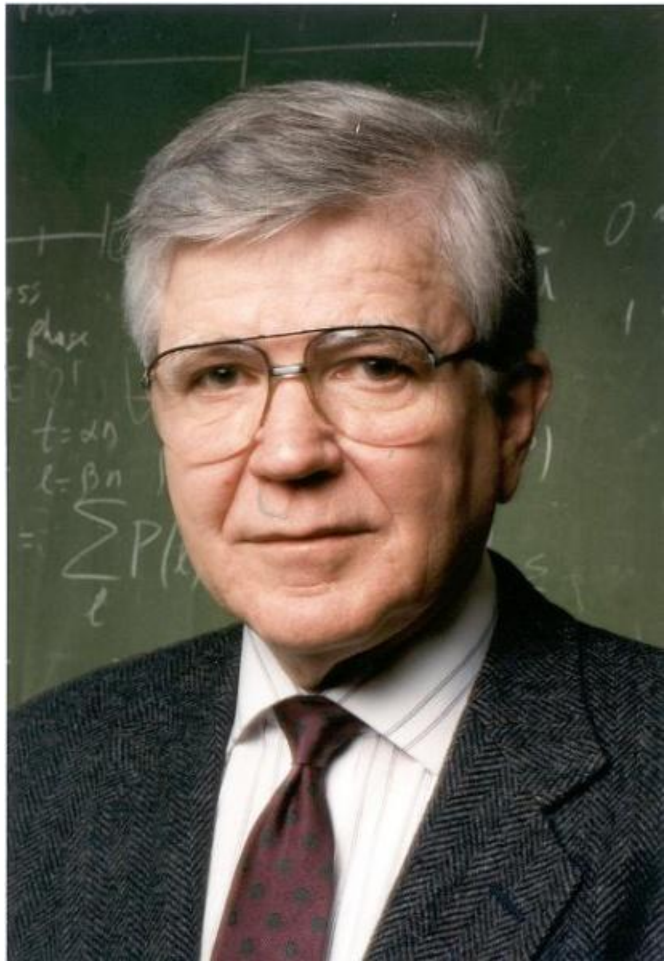
- Protection depends on generator
  - Standard CRC-32 is 10000010 01100000 10001110 110110111
- Properties:
  - Hamming Distance=4, detects up to triple bit errors!
  - Also odd number of errors
  - And bursts of up to k bits in error
  - Not vulnerable to systematic errors like checksums

Other Codes (2) – Turbo Codes

- Turbo Codes
  - Evolution of convolutional codes
  - Sends multiple sets of parity bits with payload
  - Decodes sets together (e.g. Sudoku)
  - Used in 3G and 4G cellular technologies
  - Empirically approach Shannon capacity!
- Invented and patented by Claude Berrou
  - Professor at École Nationale Supérieure des Télécommunications de Bretagne

Other Codes (3) – LDPC

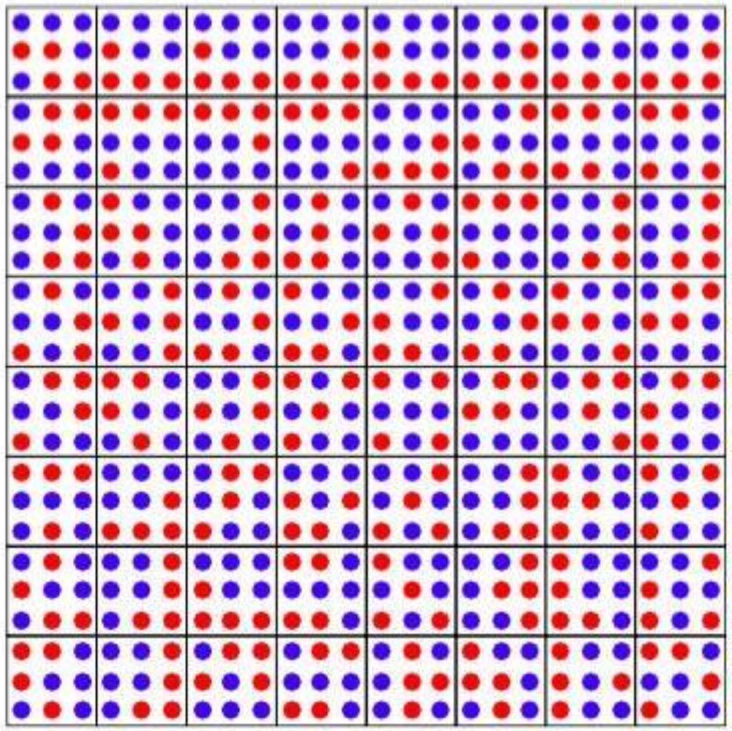
- Low Density Parity Check
  - LDPC based on sparse matrices
  - Decoded iteratively using a belief propagation algorithm
  - Empirically approach Shannon capacity!
- Invented by Robert Gallager in 1963 as part of his PhD thesis
  - Promptly forgotten until 1996 …
  - Now used for the 5G dataplane, WiFi



Source: IEEE GHN, © 2009 IEEE

Other Codes (3) – Polar codes

- New kid on the block
  - Invented 2008 by Erdal Arıkan, a Turkish professor
- *Provably achieve Shannon capacity!!!*
- Don’t have high throughput implementations yet
  - Require iteration in decode that makes it hard to parallelize
- Used in the 5G control plane



More coding theory

- This is a **huge** field.
- See EE 505, 514, 515 for more info
  - These are graduate classes
- Key points:
  - Coding allows us to detect and correct bit errors received from the PHY
  - It can get complicated…
    - Abstract away with Hamming Distance :)

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?

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 still depends! We need to know more about the errors

Detection vs. Correction (2)

Numbers here are approximate, specifics depend on specific code and implementation

Assume bit errors are random

- Messages have 0 or maybe 1 error (1/10 of the time)

Error correction:

- Need ~10 check bits per message
- Overhead: ?

Error detection:

- Need ~1 check bits per message plus 1000 bit retransmission
- Overhead: ?

Detection vs. Correction (3)

Numbers here are approximate, specifics depend on specific code and implementation

Assume bit errors are random

- Messages have 0 or maybe 1 error (1/10 of the time)

Error correction:

- Need ~10 check bits per message
- Overhead: 10b/m

Error detection:

- Need ~1 check bits per message plus 1000 bit retransmission
- Overhead: 1b/m + 1000/10 = 101bpm

Detection vs. Correction (4)

Numbers here are approximate, specifics depend on specific code and implementation

Assume errors come in bursts of 100

- Only 1 or 2 messages in 1000 have significant (multi-bit) errors

Error correction:

- Need >>100 check bits per message
- Overhead: ?

Error detection:

- Need 32 check bits per message plus 1000 bit resend 2/1000 of the time
- Overhead: ?

Detection vs. Correction (5)

Numbers here are approximate, specifics depend on specific code and implementation

Assume errors come in bursts of 100

- Only 1 or 2 messages in 1000 have significant (multi-bit) errors

Error correction:

- Need >>100 check bits per message
- Overhead: >>100 b/m

Error detection:

- Need 32 check bits per message plus 1000 bit resend 2/1000 of the time
- Overhead: 32b/m + 1000* .002= 34b/m

Detection vs. Correction (6)

- Error correction:
  - Needed when errors are expected
  - Or when no time for retransmission

- Error detection:
  - More efficient when errors are not expected
  - And when errors are large when they do occur

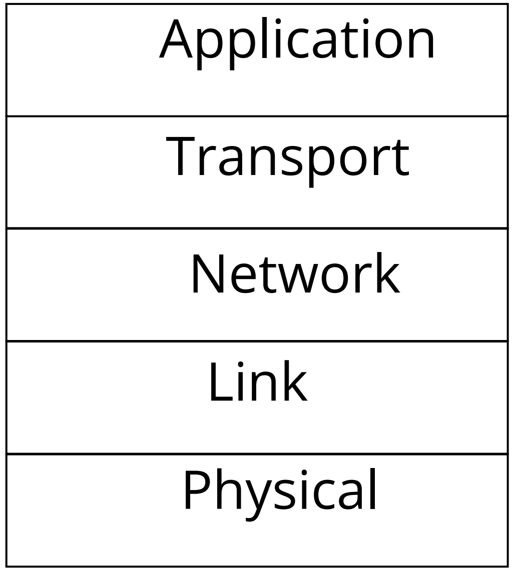
Error Correction in Practice

- Heavily used when the phy is error prone
  - LDPC + polar is the future, used for demanding links like 802.11, DVB, 5G, power-line…
  - Convolutional codes widely used in practice
- Error detection (w/ retransmission) is used in the link layer and above for residual errors
- Correction can also used in the application layer
  - Called Forward Error Correction (FEC)
  - Normally with an “erasure” error model (bits lost instead of flipped)
  - E.g., Reed-Solomon (CDs, DVDs, etc.)

Link Layer: Retransmissions

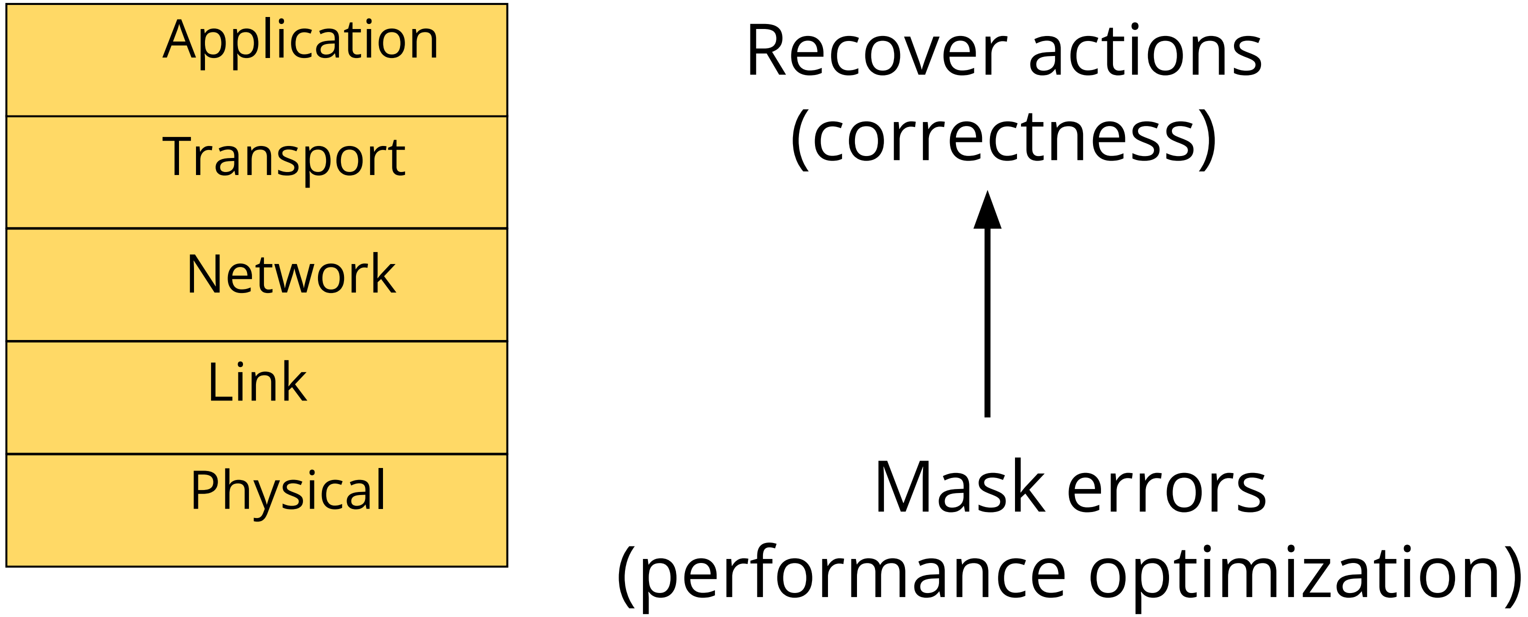
Context on Reliability

- Where in the stack should we place reliability functions?



Context on Reliability

- Where in the stack should we place reliability functions?
- Everywhere! It is a key issue
  - Different layers contribute differently



What do we do if a frame is corrupted?

- From sender?
- From receiver?

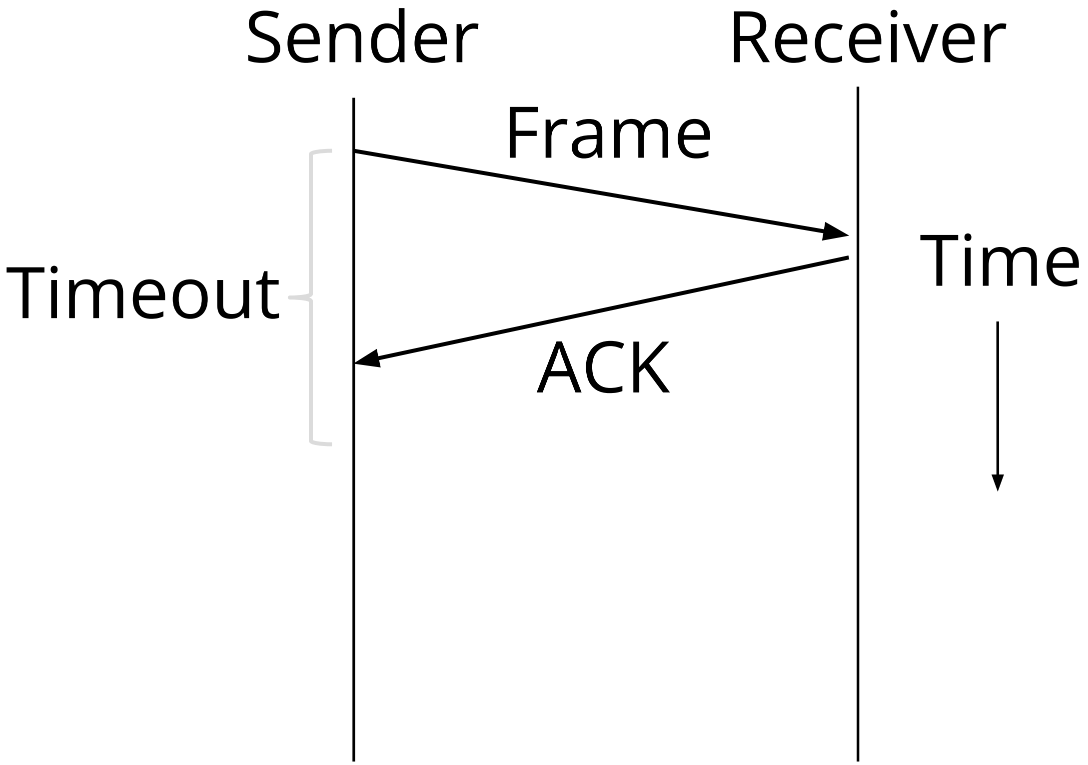
ARQ (Automatic Repeat reQuest)

- ARQ often used when errors are common or must be corrected
  - E.g., WiFi, and TCP (later)

- Rules at sender and receiver:
  - Receiver automatically acknowledges correct frames with an ACK
  - Sender automatically resends after a timeout, until an ACK is received

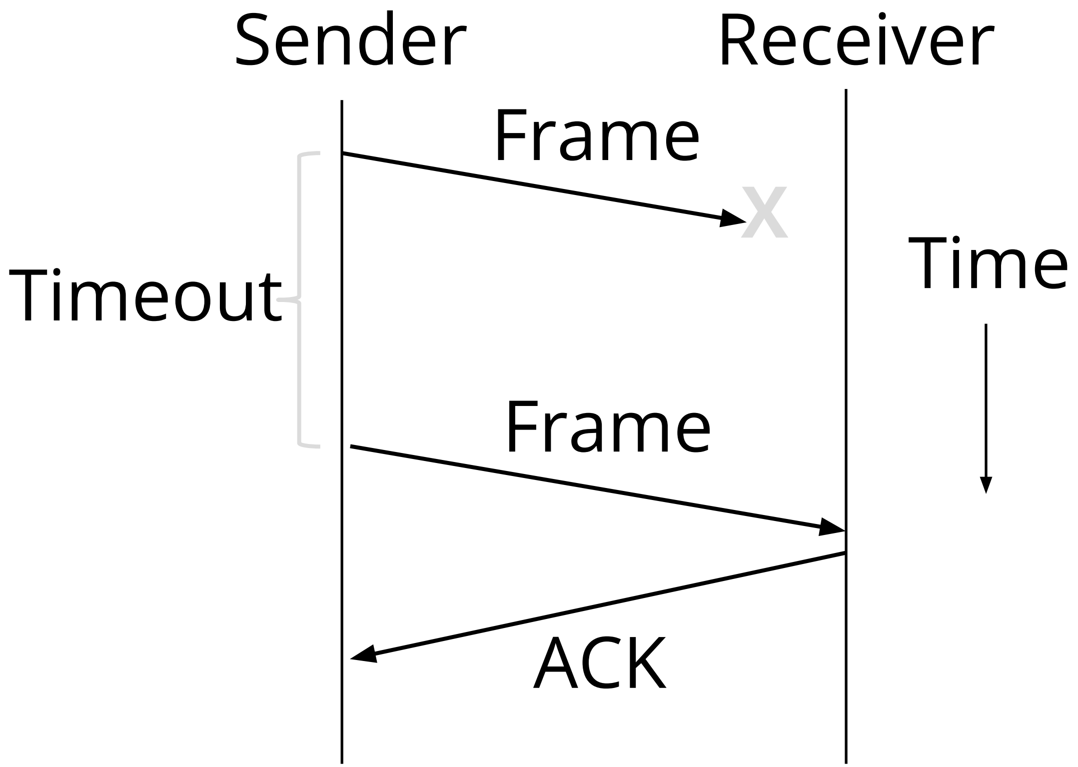
ARQ (2)

- Normal operation (no loss)



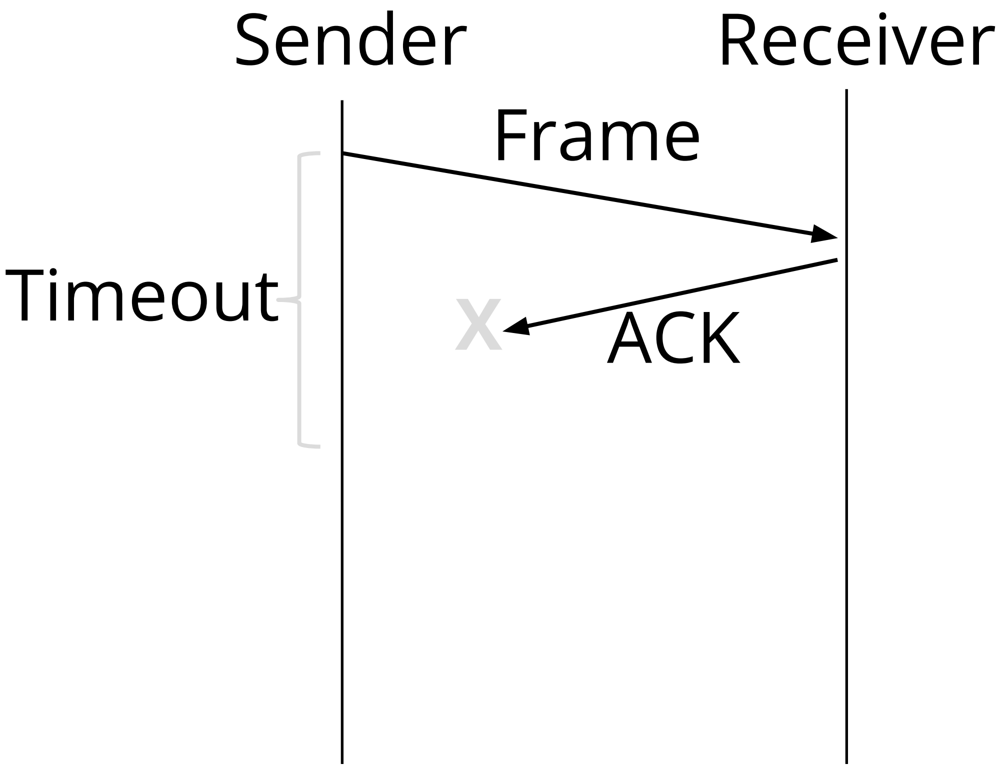
ARQ (3)

- Loss and retransmission



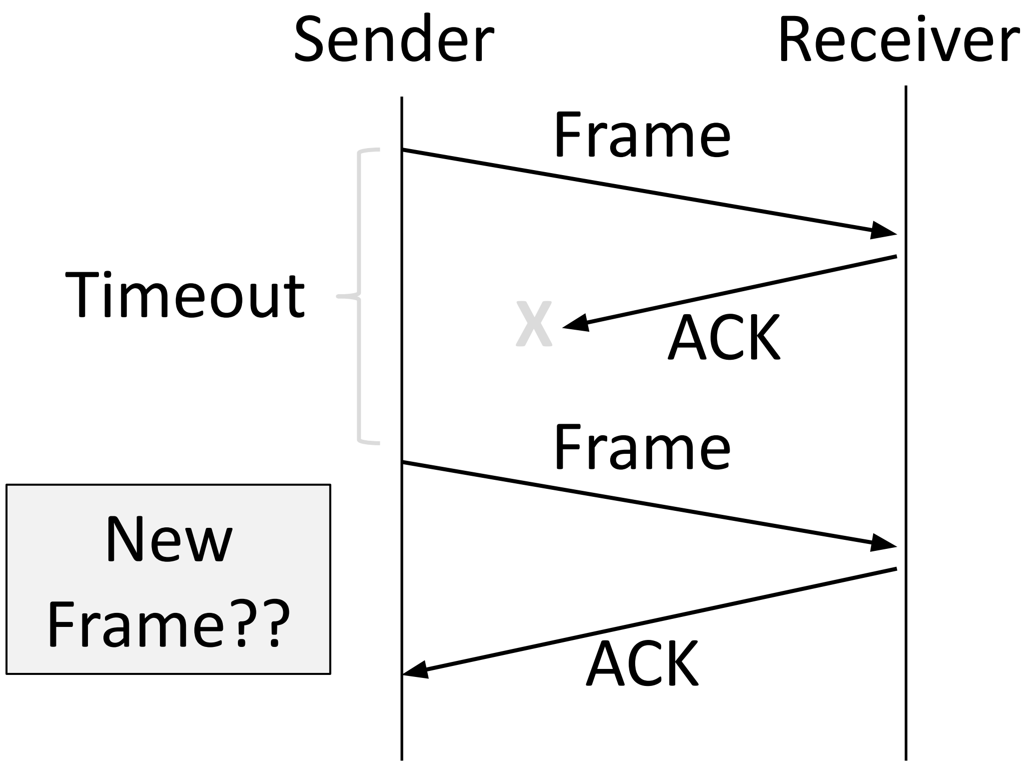
Duplicates

- What happens if an ACK is lost?



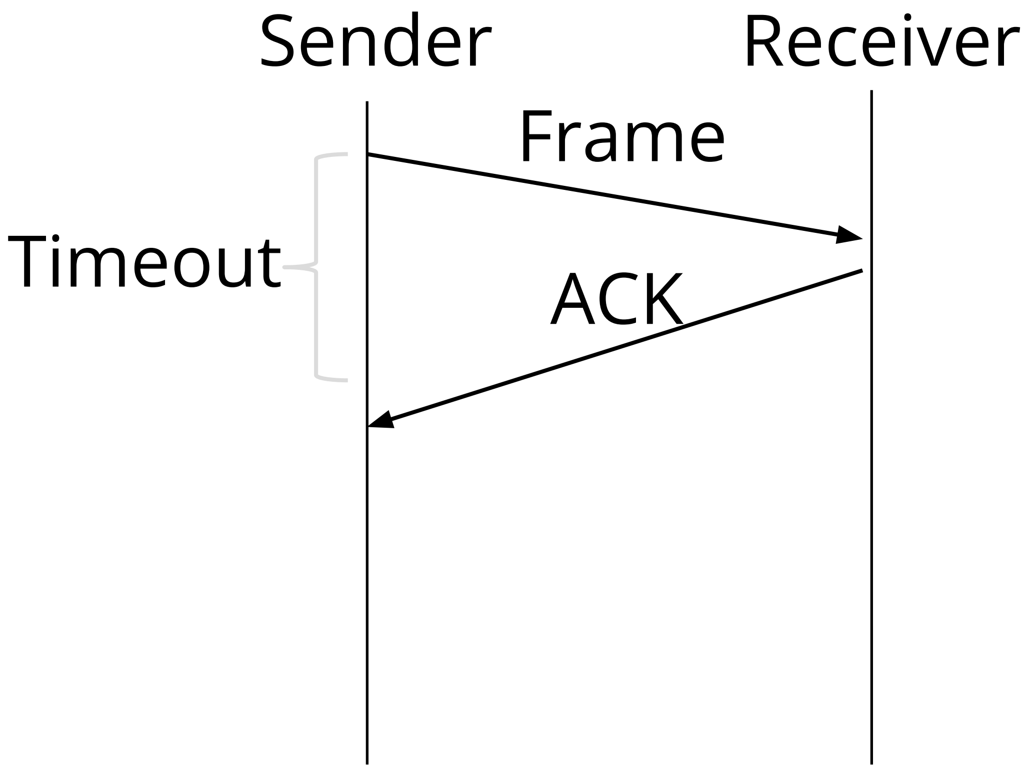
Duplicates (2)

- What happens if an ACK is lost?



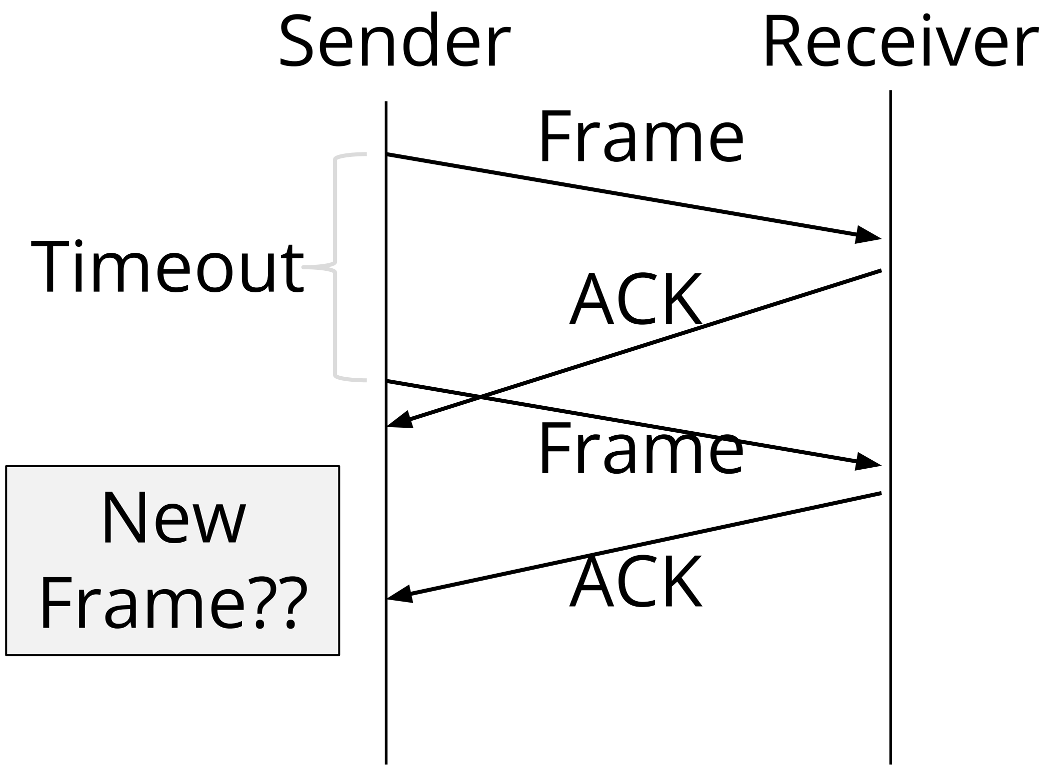
Duplicates (3)

- Or the timeout is early?



Duplicates (4)

- Or the timeout is early?



So What's Tricky About ARQ?

- Two non-trivial issues:
  - How long to set the timeout?
  - How to avoid accepting duplicate frames as new frames
- Want performance in the common case and correctness always…
- Ideas?

Timeouts

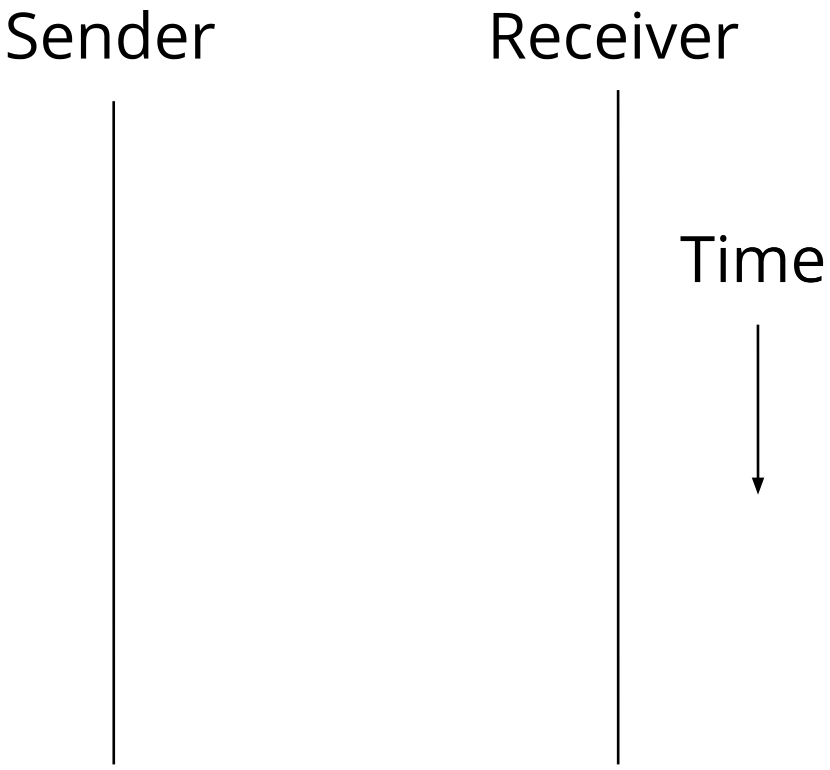
- Timeout should be:
  - Not too big (link goes idle)
  - Not too small (spurious resend)
- Fairly easy on a LAN
  - Clear worst case, little variation
- Fairly difficult over the Internet : (
  - Much variation, no obvious bound
  - We'll revisit this with TCP (later)

Sequence Numbers

- Frames and ACKs must both carry sequence numbers for correctness
- To distinguish the current frame from the next one, a single bit (two numbers) is sufficient
  - Called Stop-and-Wait

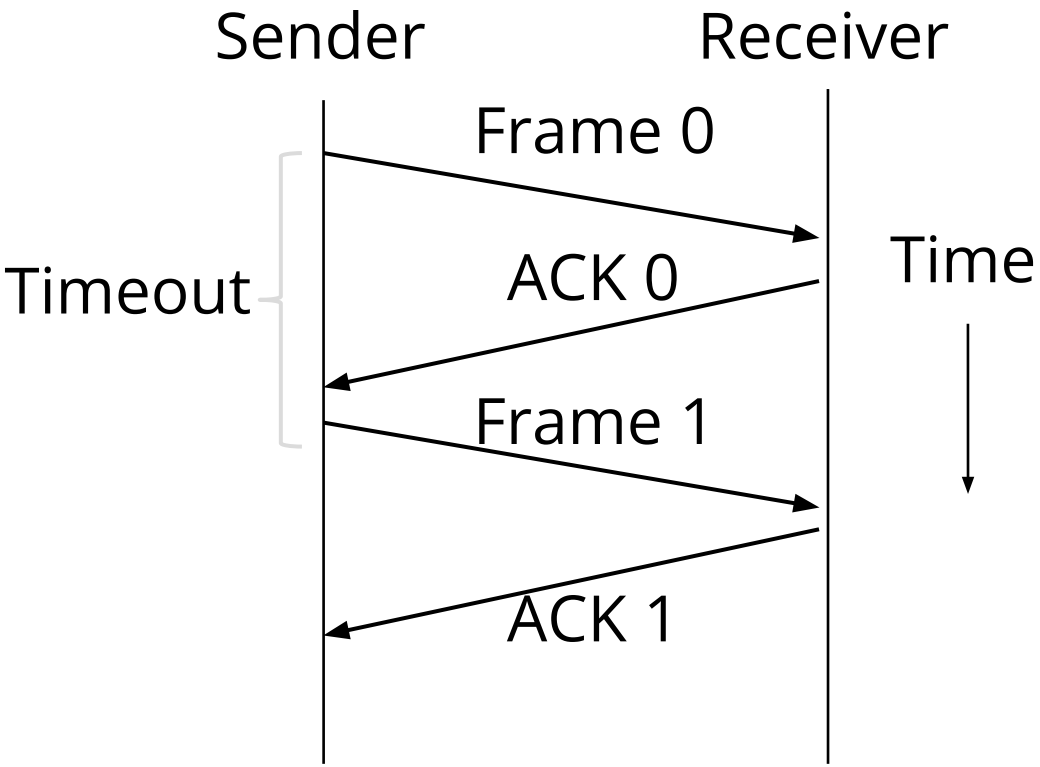
Stop-and-Wait

- In the normal case:



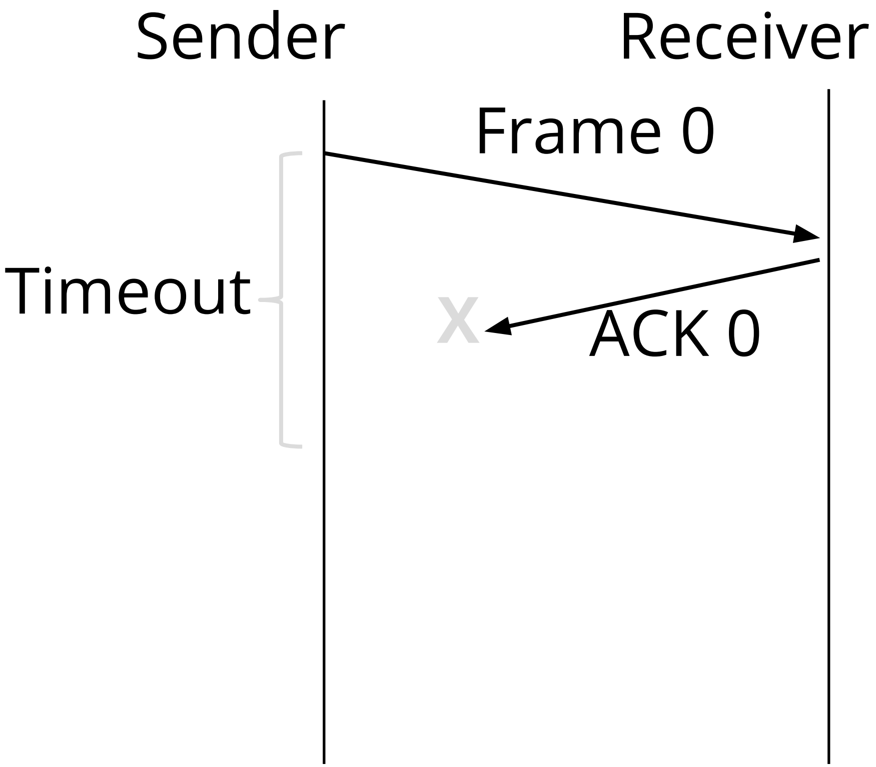
Stop-and-Wait (2)

- In the normal case:



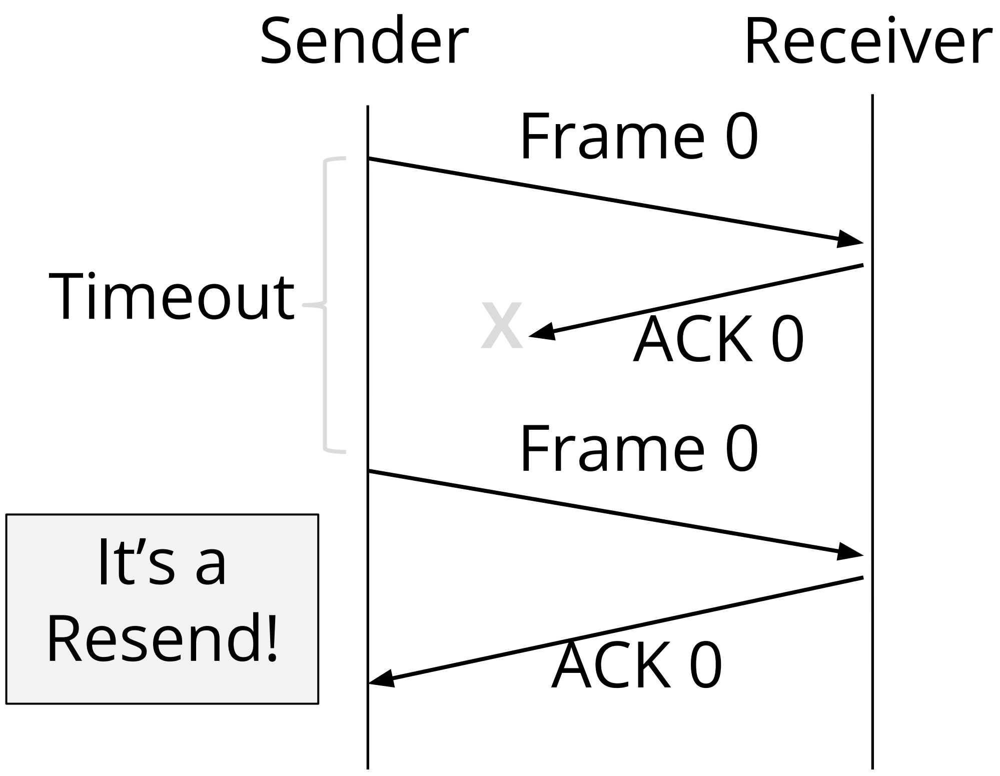
Stop-and-Wait (3)

- With ACK loss:



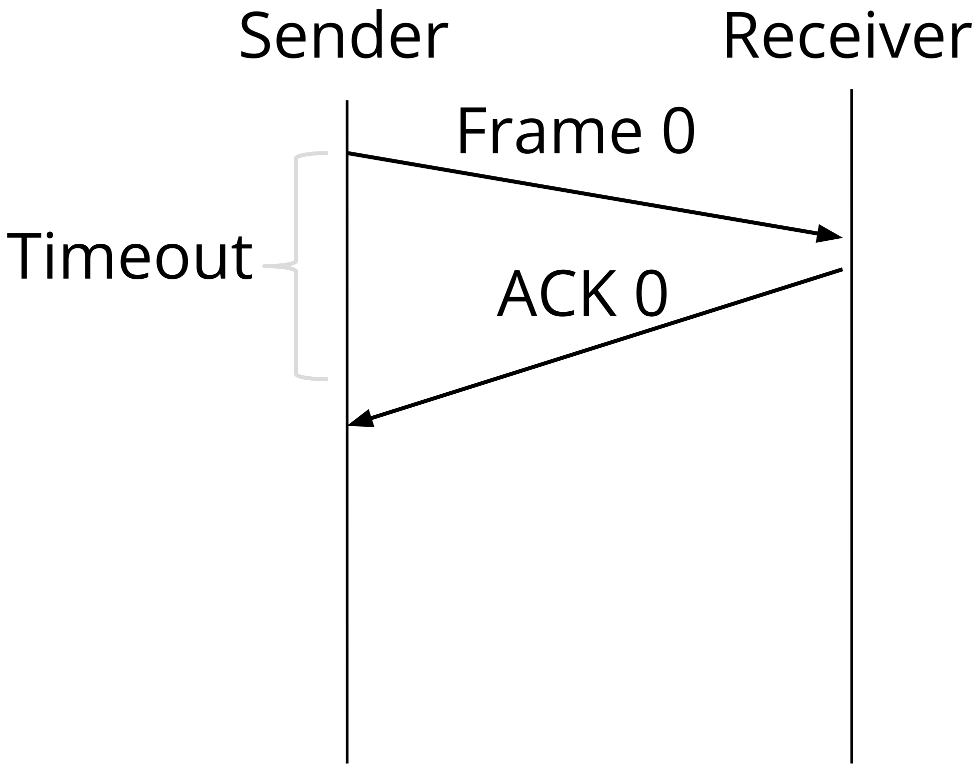
Stop-and-Wait (4)

- With ACK loss:



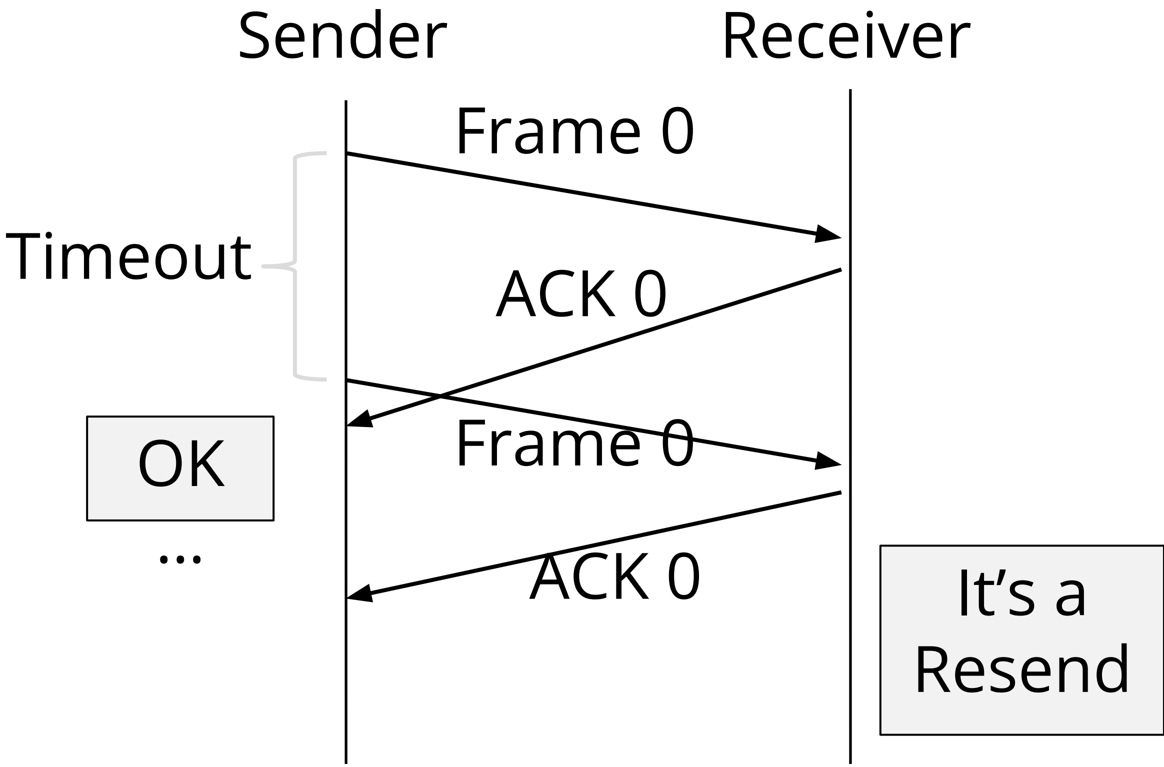
Stop-and-Wait (5)

- With early timeout:



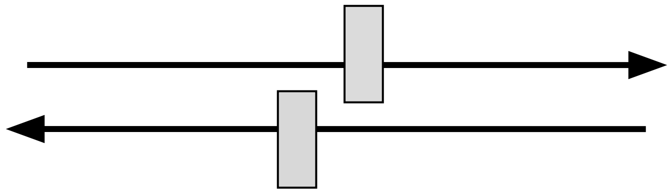
Stop-and-Wait (6)

- With early timeout:



Limitation of Stop-and-Wait

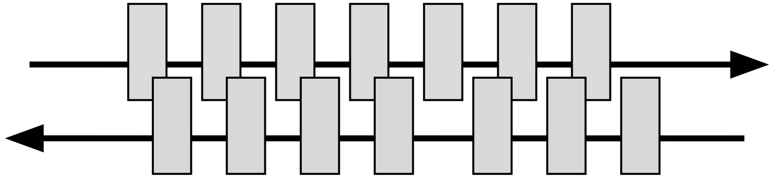
- It allows only a single frame to be outstanding from the sender:
  - Good for LAN, not efficient for high Bandwidth x Delay Product



- Ex: R=1 Mbps, D = 50 ms
- Approximately how many frames/sec? If R=10 Mbps?

Sliding Window

- Generalization of stop-and-wait
  - Allows W frames to be outstanding
  - Can send W frames per RTT (=2D)



- Various options for numbering frames/ACKs and handling loss
  - Will look at along with TCP (later)