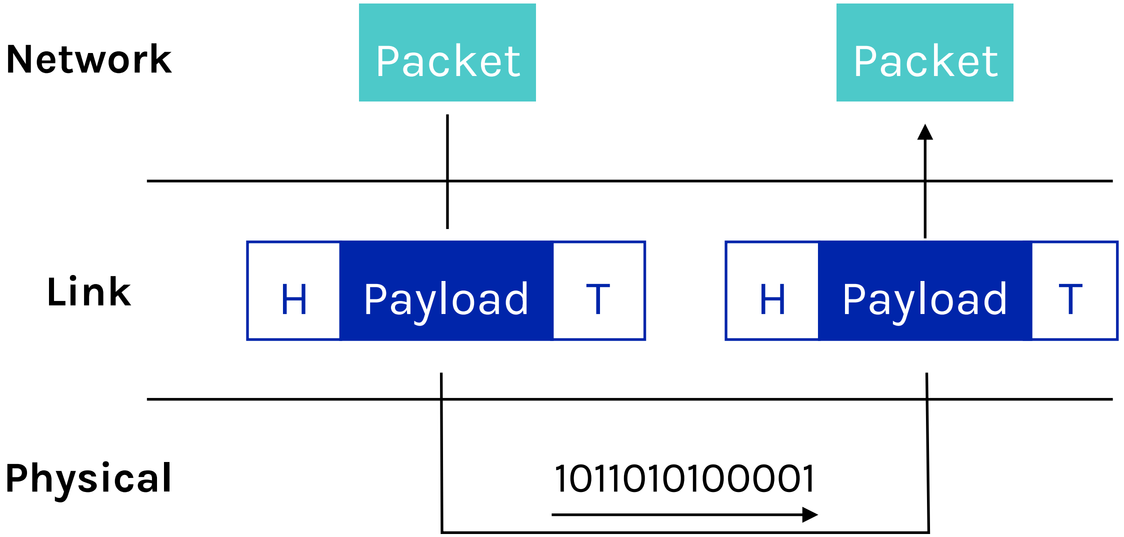
Computer Networks (WS23/24)

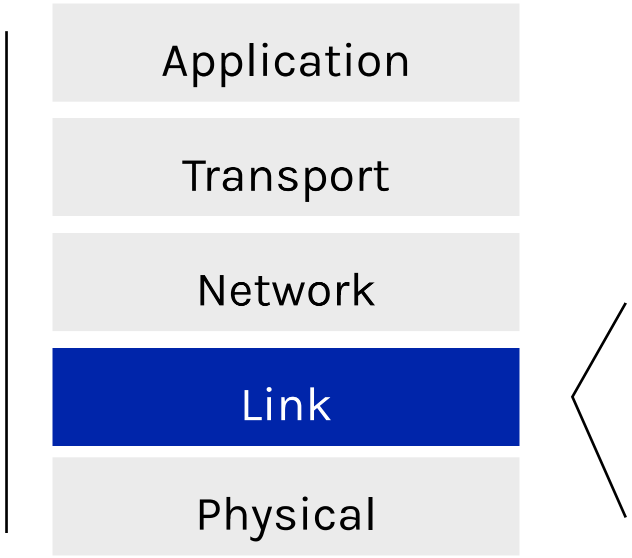
L3: The Link Layer - Part 1

Prof. Dr. Lin Wang
Computer Networks Group (PBNet)
Department of Computer Science
Paderborn University

Materials inspired by David J. Wetherall and Shyam Gollakota

Learning objectives





Part 1

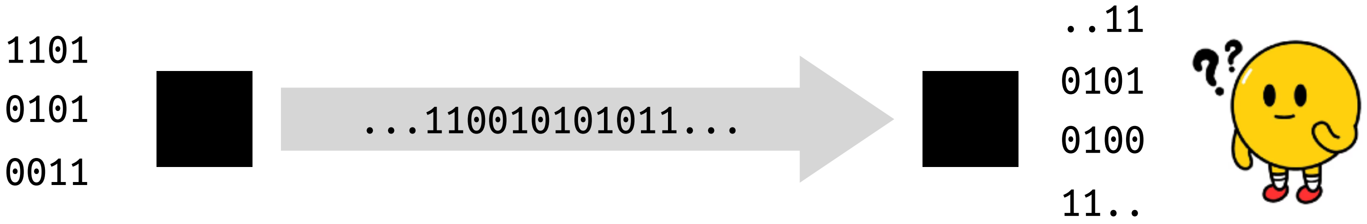
- Framing
- Error detection and correction
- Reliability: retransmission

Part 2

- Multi-access: 802.11
- Ethernet: 802.3
- Switching

Framing

From stream of bits to sequence of frames

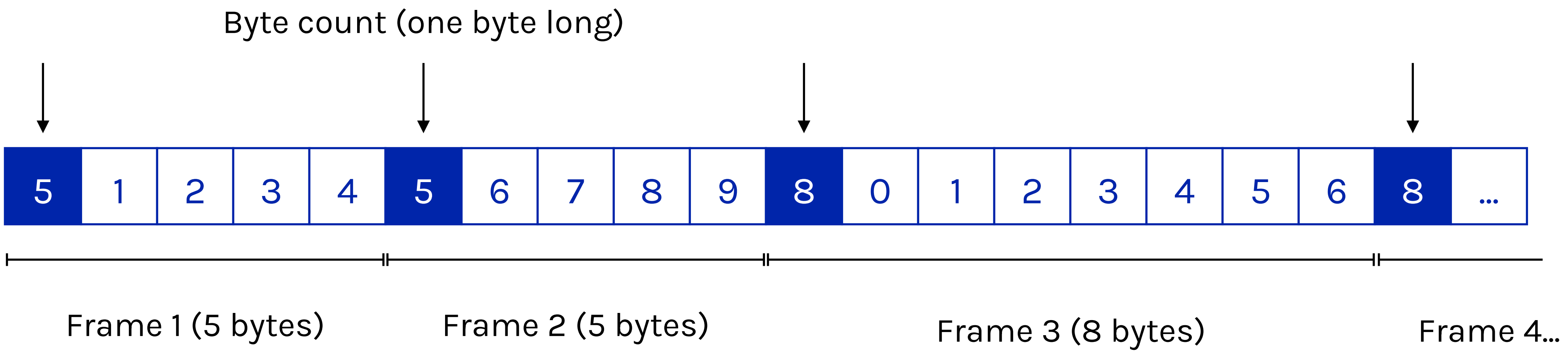


Sequence of frames Stream of bits Sequence of frames

The receiver has to know how to seperate the stream of bits into frames

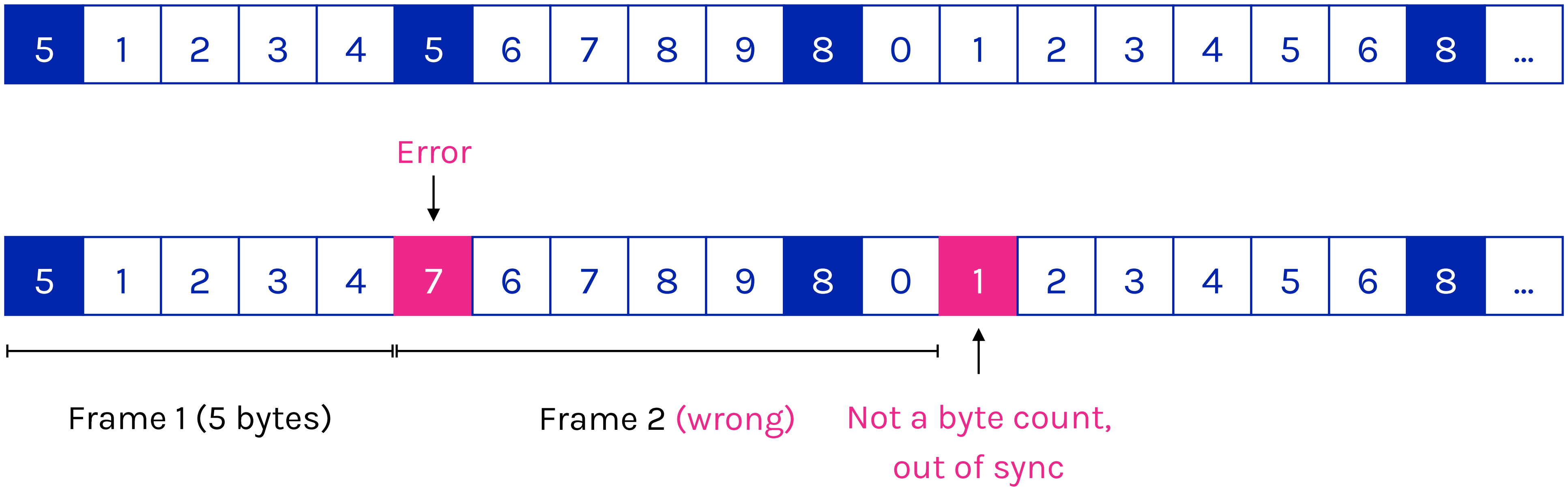
Here, we assume the bits are delivered in exactly the same order in which they are sent, which is typically true for a single link (a wire-like channel).

Simple idea: byte count



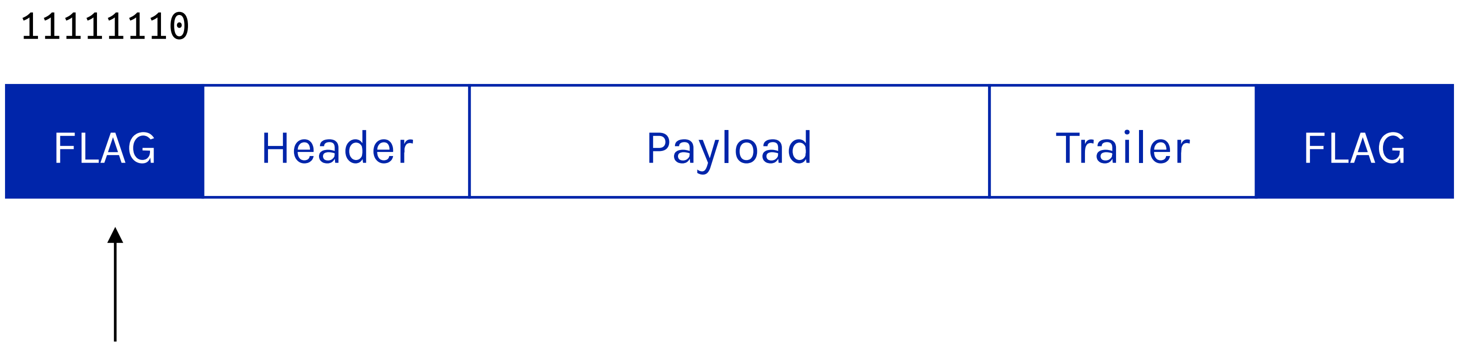
What are the problems?

Byte count issue



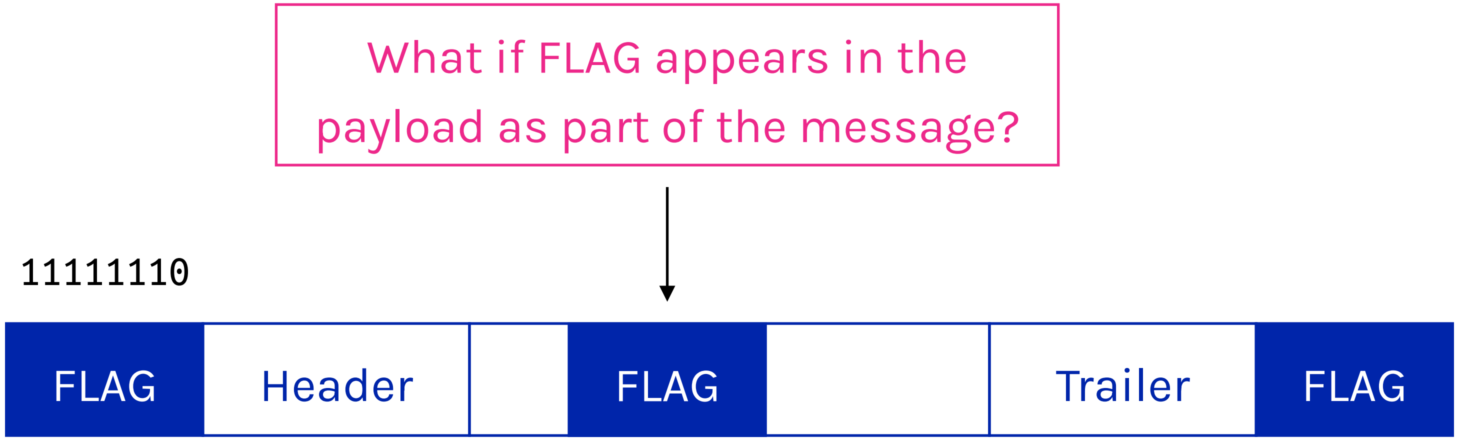
Difficult to **resync** after framing error; needs a way to scan for the start of a frame

Framing through byte stuffing

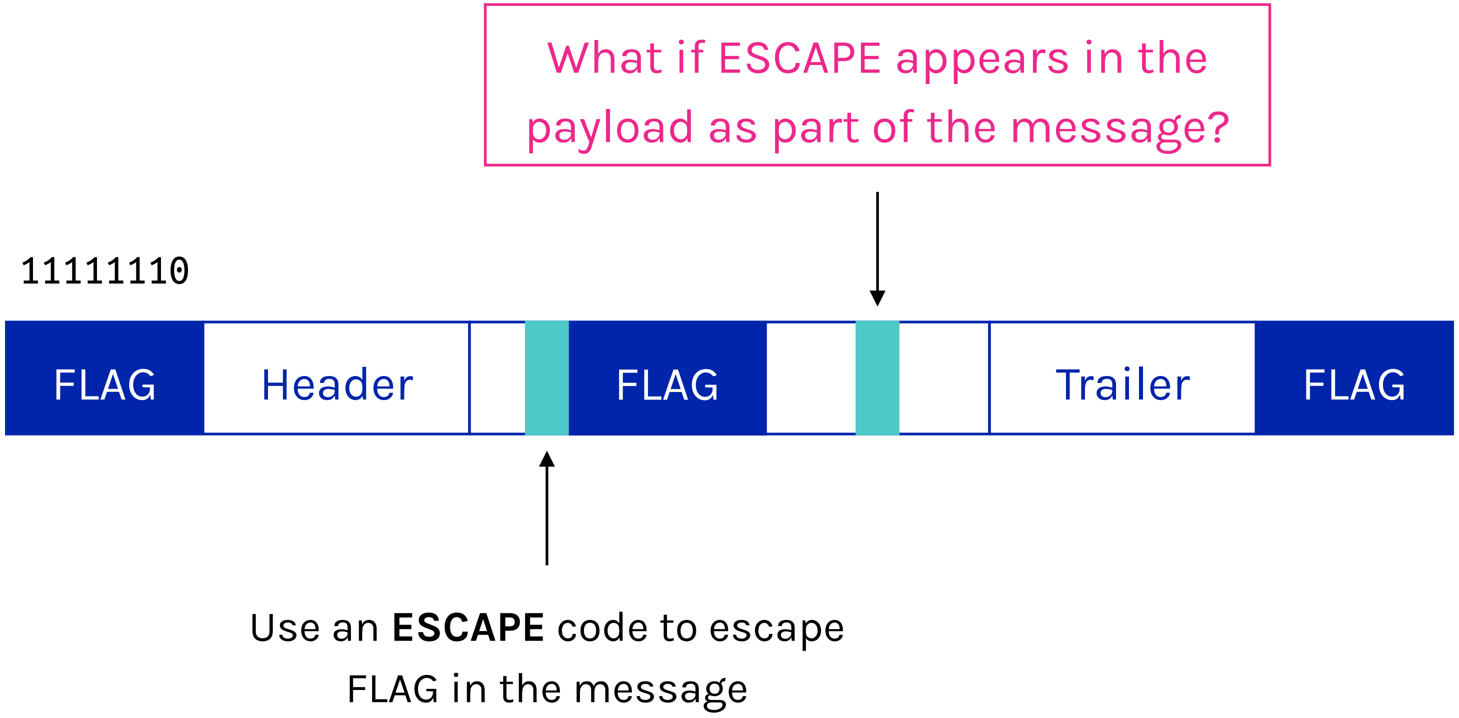


Use a special FLAG byte to
indicate the start/end of a frame

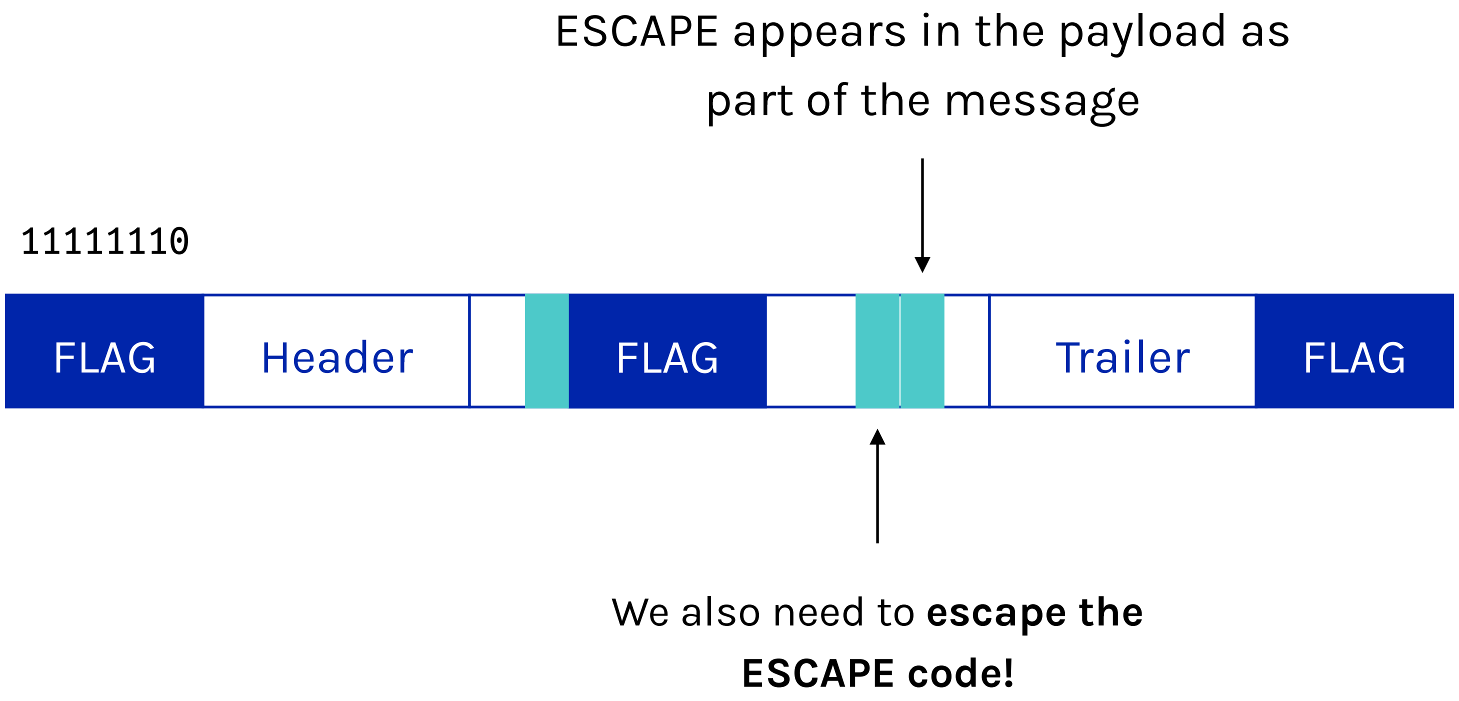
Byte stuffing: issues



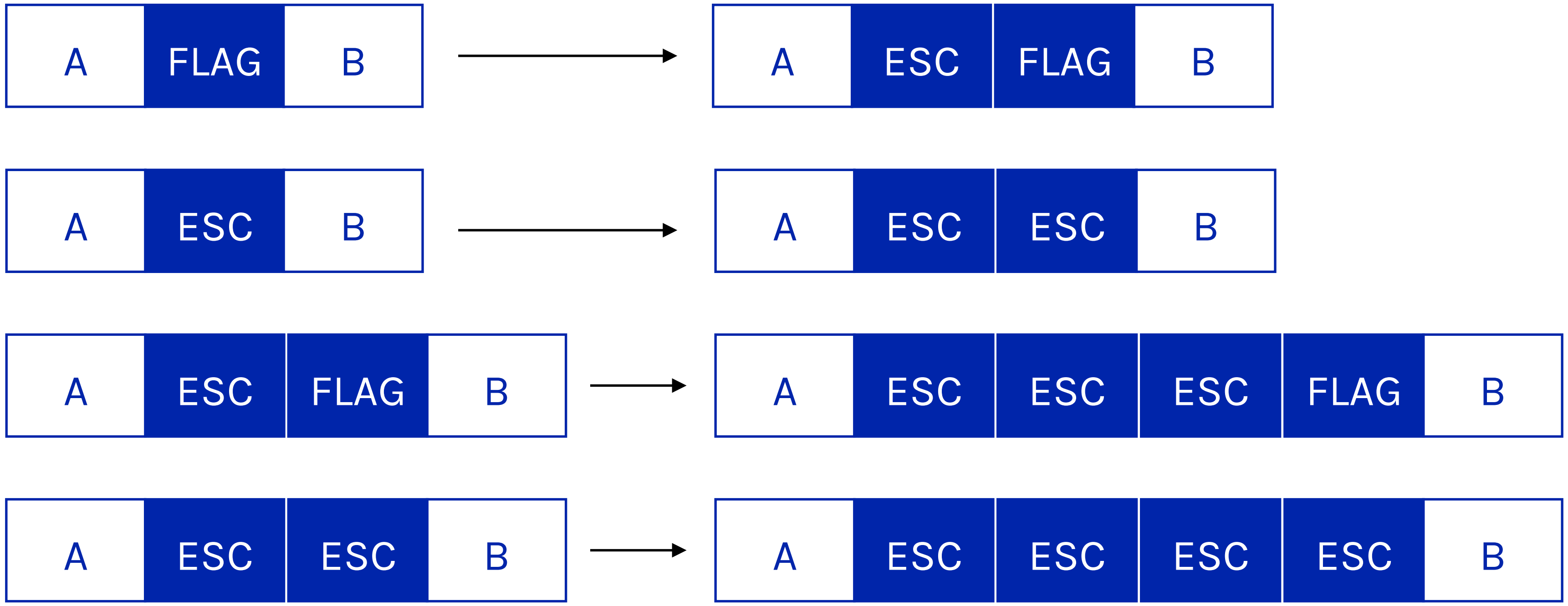
Byte stuffing: escape



Byte stuffing: escaping escape



Byte stuffing: example



Byte stuffing: rules

When you see

- Solitary FLAG: start or end of a frame
- Solitary ESC: something went wrong
- ESC FLAG: remove ESC and pass FLAG through
- ESC ESC FLAG: remove ESC, pass ESC through, and then start or end of a frame
- ESC ESC ESC FLAG: pass ESC FLAG through

Framing through bit stuffing

Stuffing at the bit level

- Call a FLAG six consecutive 1s
- On transmit, after five consecutive 1s in the message, insert a 0
- On receive, a 0 after five 1s is deleted

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

Framing through coding violations

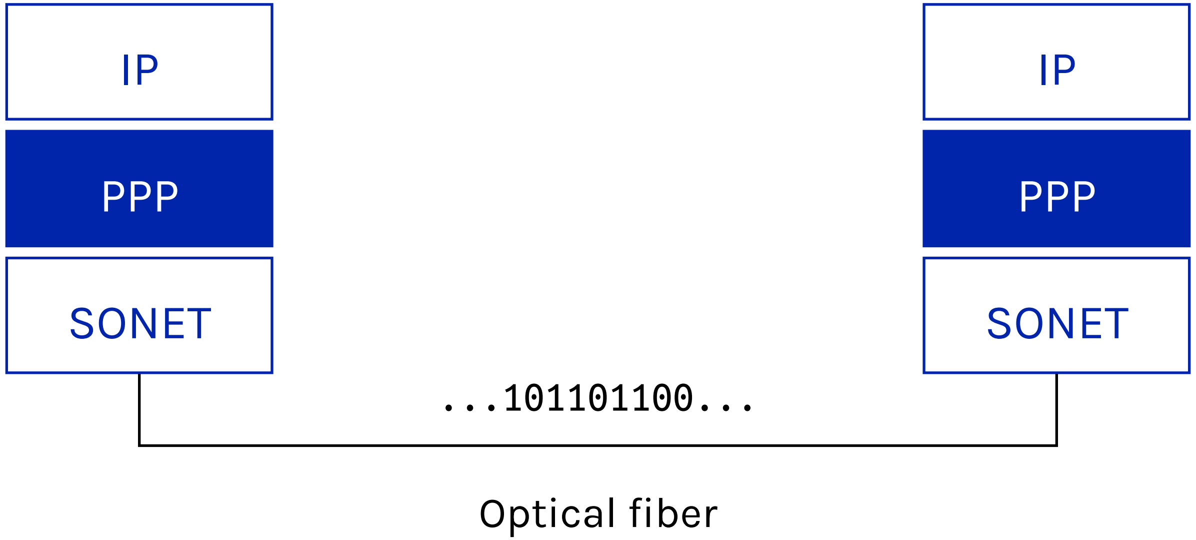
Recall 4B/5B encoding

- Map every 4 data bits into 5 code bits without long runs of zeros
- 16 out of 32 possibilities are unused (not in regular data)

Use some of reserved signals to delimit the frame

- Easy to find the start and end of the frame
- No need to stuff the data

Link example: point-to point protocol (PPP)



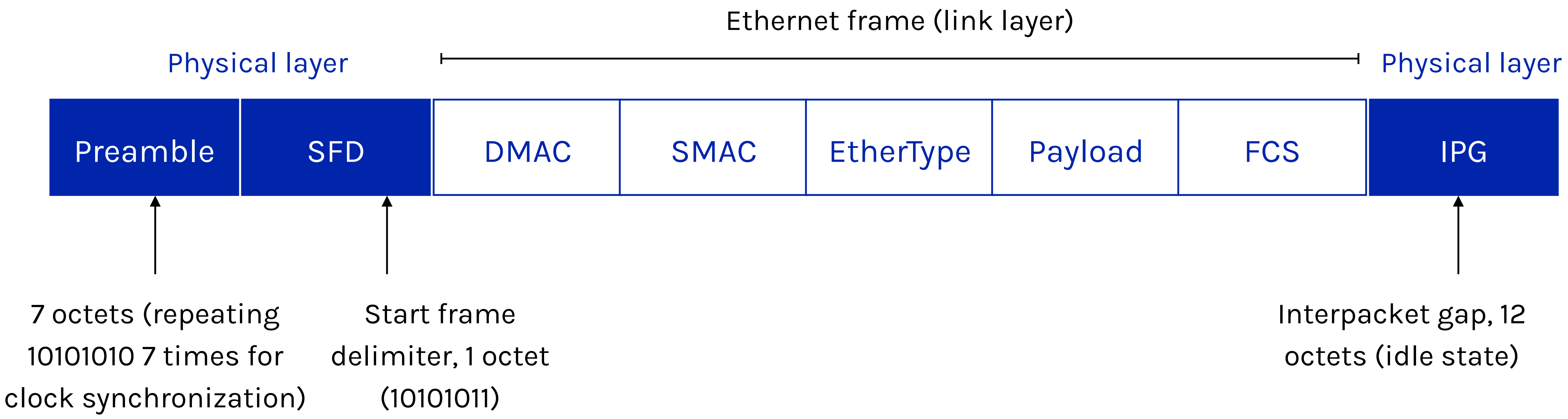
Byte stuffing in PPP

FLAG: 0x7E (01111110), ESC: 0x7D (01111101)

	FLAG 0111110	Address 11111111	Control 00000011	Protocol	Payload	Checksum	FLAG 01111110
Size (in bytes)	1	1	1	1 or 2	Variable	2 or 4	1

Link example: Ethernet

The physical layer helps with the detection of frame boundaries



Error Detection and Correction

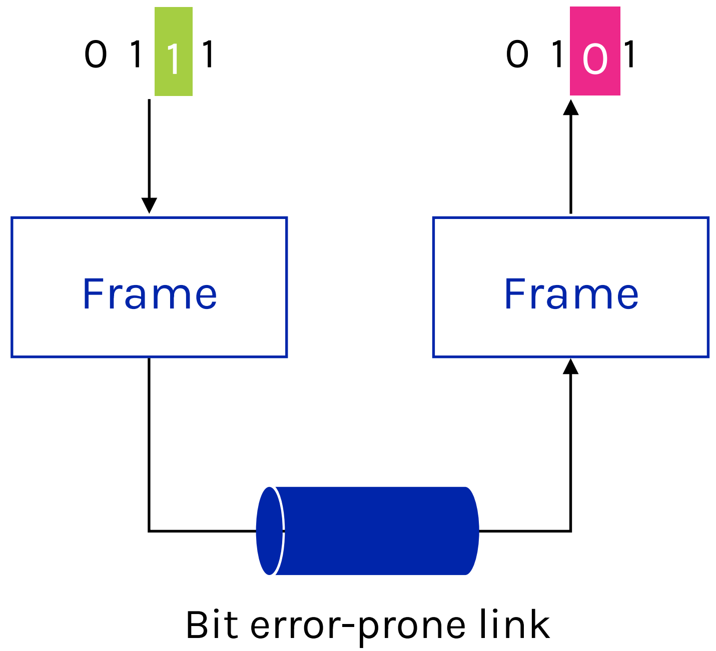
Errors

Some bits may be received in error (due to noise)

- Detect errors with codes: drop the frame and let the higher layers to take corrective measures
- Correct errors with codes: correct as many errors as possible

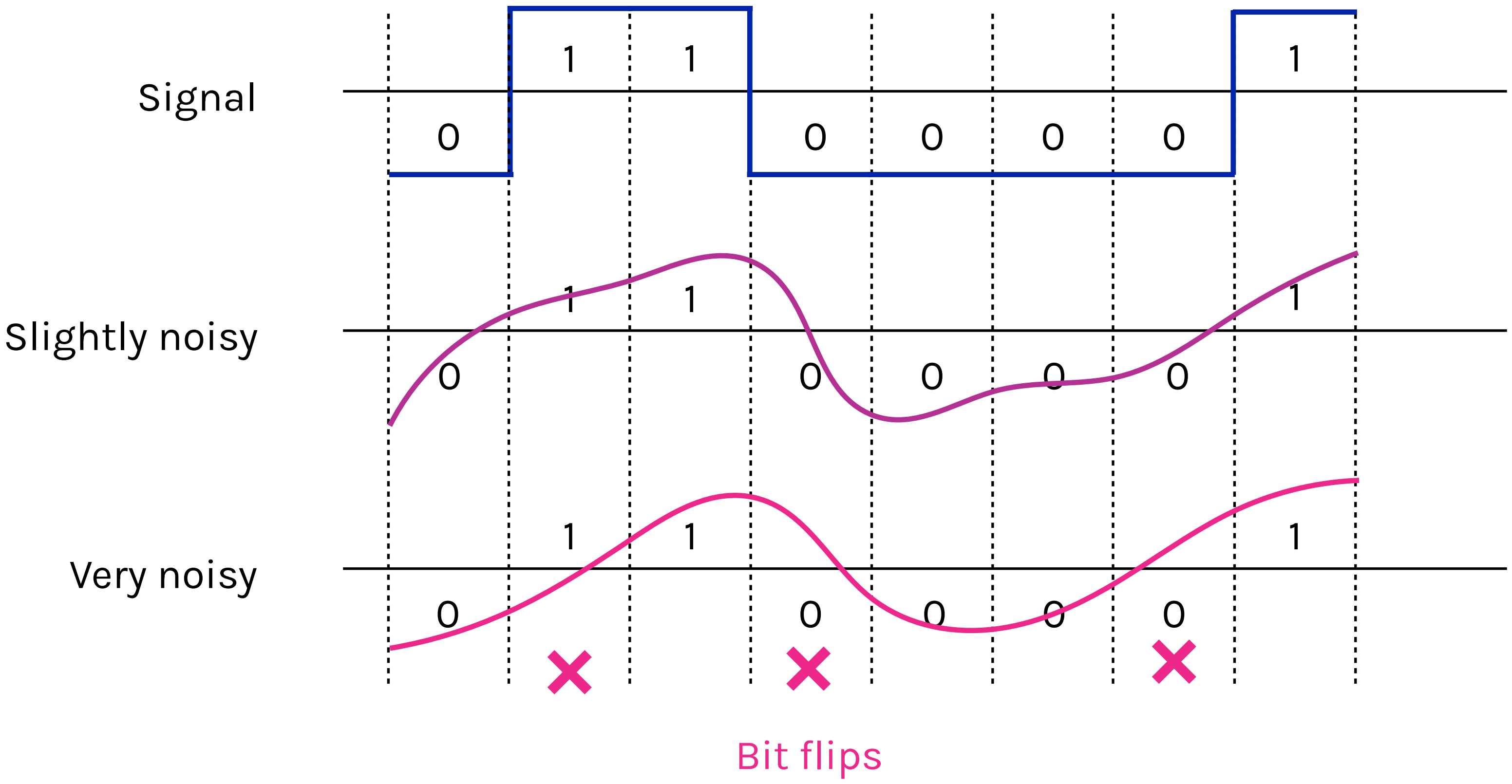
Reliability is a concern that cuts across the layers

- We will cover more in the upper layers



Problem

Noise may flip the received bits



Approach: adding redundancy

Error detection codes

- Add **check bits** to the message bits to let some errors be detected at the receiver

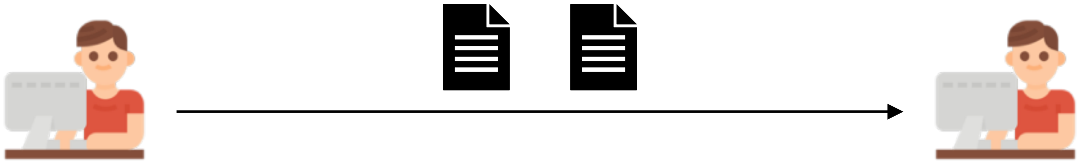
Error correction codes

- Add more **check bits** to let some errors be corrected

Key issue: How to construct the check bits?

- Detect **many** errors with **few** check bits
- **Resonable computation** at both the sender and receiver

Motivating example



Send two copies of the same message; error if different

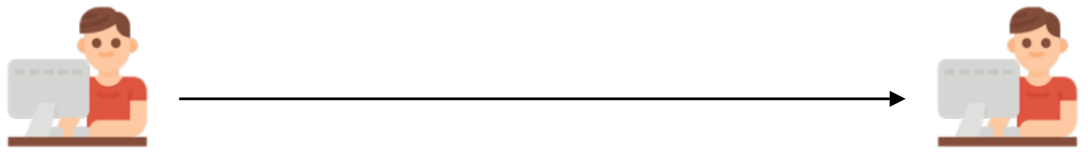
How many errors can it detect/correct?

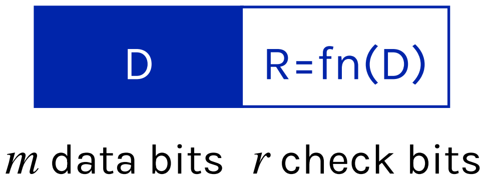
How many errors will make it fail?

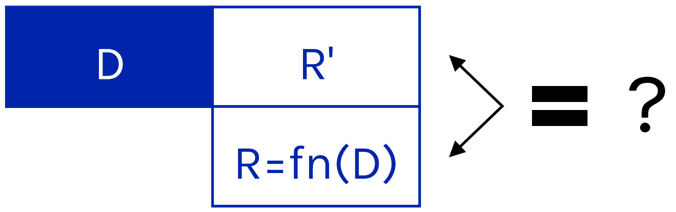
Error codes

Computer r check bits based on the m data bits; send the codeword of $m + r$ bits

Receive $m + r$ bits with unknown errors; **recompute** r check bits based on the m data bits: error if R does not match R'

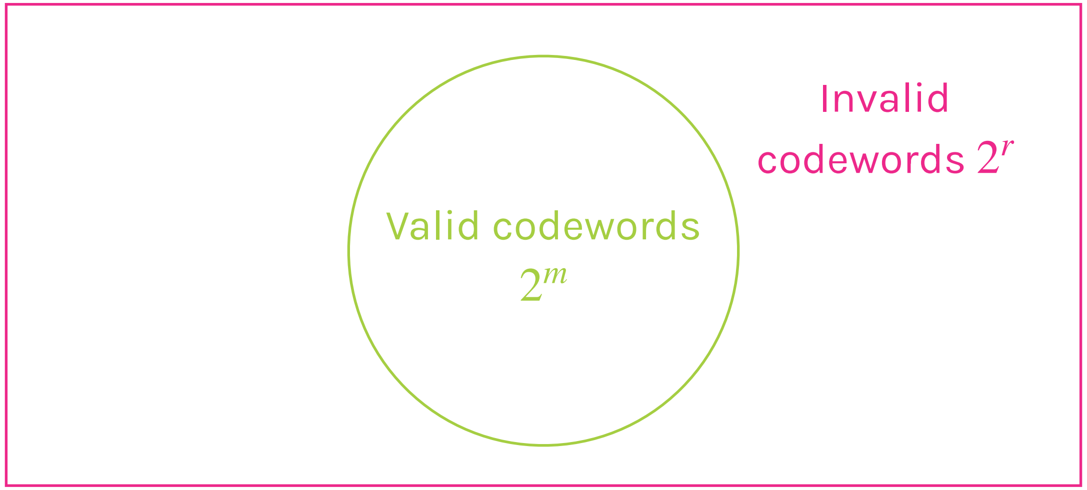






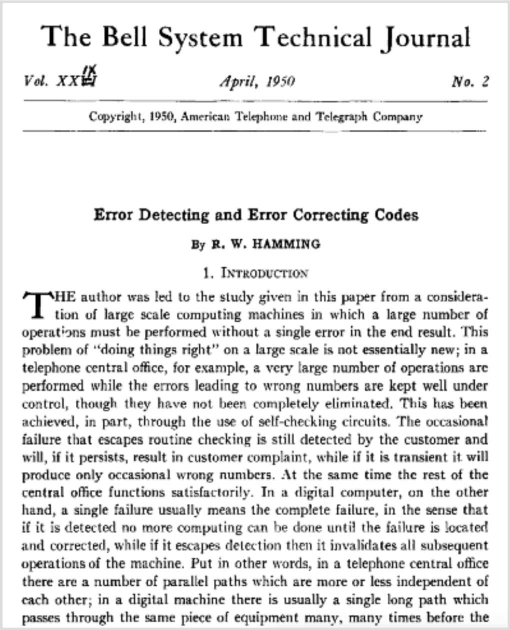
Intuition for error codes

Codewords 2^n ($= 2^m 2^r$)



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

R. W. Hamming (1915 - 1998)



The Bell System Technical Journal

Vol. XXIX | April, 1950 | No. 2

Error Detecting and Error Correcting Codes

By R. W. HAMMING

1. INTRODUCTION

THE author was led to the study given in this paper from a consideration of large scale computing machines in which a large number of operations must be performed without a single error in the end result. This problem of "doing things right" on a large scale is not essentially new; in a telephone central office, for example, a very large number of operations are performed while the errors leading to wrong numbers are kept well under control, though they have not been completely eliminated. This has been achieved, in part, through the use of self-checking circuits. The occasional failure that escapes routine checking is still detected by the customer and will, if it persists, result in customer complaint, while if it is transient it will produce only occasional wrong numbers. At the same time the rest of the central office functions satisfactorily. In a digital computer, on the other hand, a single failure usually means the complete failure, in the sense that if it is detected no more computing can be done until the failure is located and corrected, while if it escapes detection then it invalidates all subsequent operations of the machine. Put in other words, in a telephone central office there are a number of parallel paths which are more or less independent of each other; in a digital machine there is usually a single long path which passes through the same piece of equipment many, many times before the

"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."

https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

Hamming distance (HD)

Definition

- The number of positions at which the corresponding symbols are different for two strings of equal length

Hamming distance between two codewords (D1 and D2) is the number of bit flips needed to change D1 to D2

karolin → kathrin $HD = 3$

kathrin → kerstin $HD = 4$

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

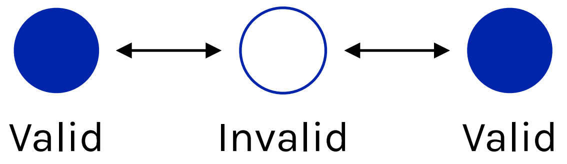
Hamming distance requirements

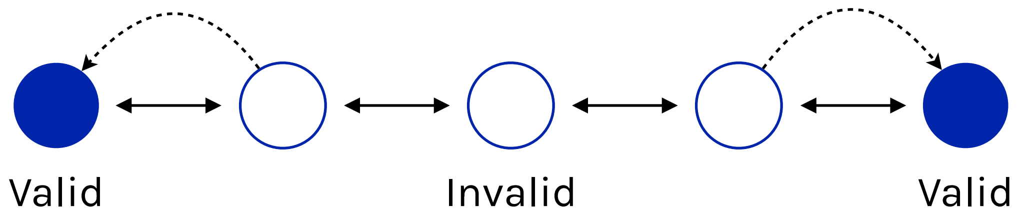
Error detection

For a coding of distance $d + 1$, up to d errors will always be detected

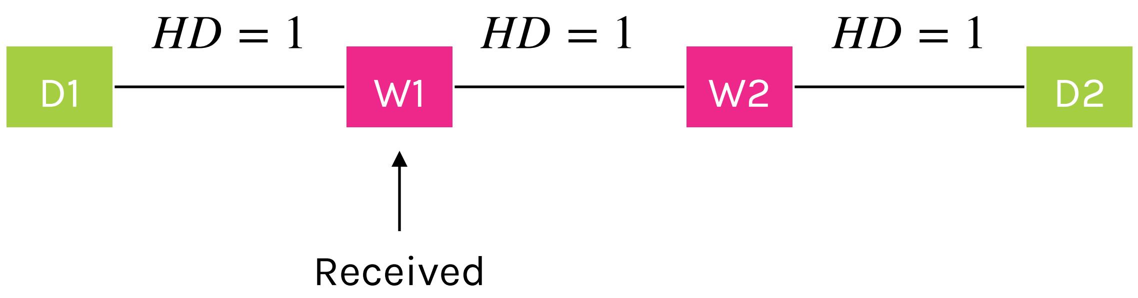
Error correction

For a coding of distance $2d + 1$, up to d errors can always be corrected by mapping to the closest valid codeword





Hamming distance requirements example



Case 1 Originally, D1 had been sent, but 1 bit error occured

Case 2 Originally, D2 had been sent, but 2 bit errors occured

Case 3 Originally, some other data had been sent, but at least 2 bit errors occured

Assuming fewer errors have happened, a received frame W1 is **presumed** to have been caused by sending D1!

Simple error detection: parity bit

Take D data bits, add one check bit that is the sum of the D bits

- Sum is modulo 2 or XOR

How well does parity work?

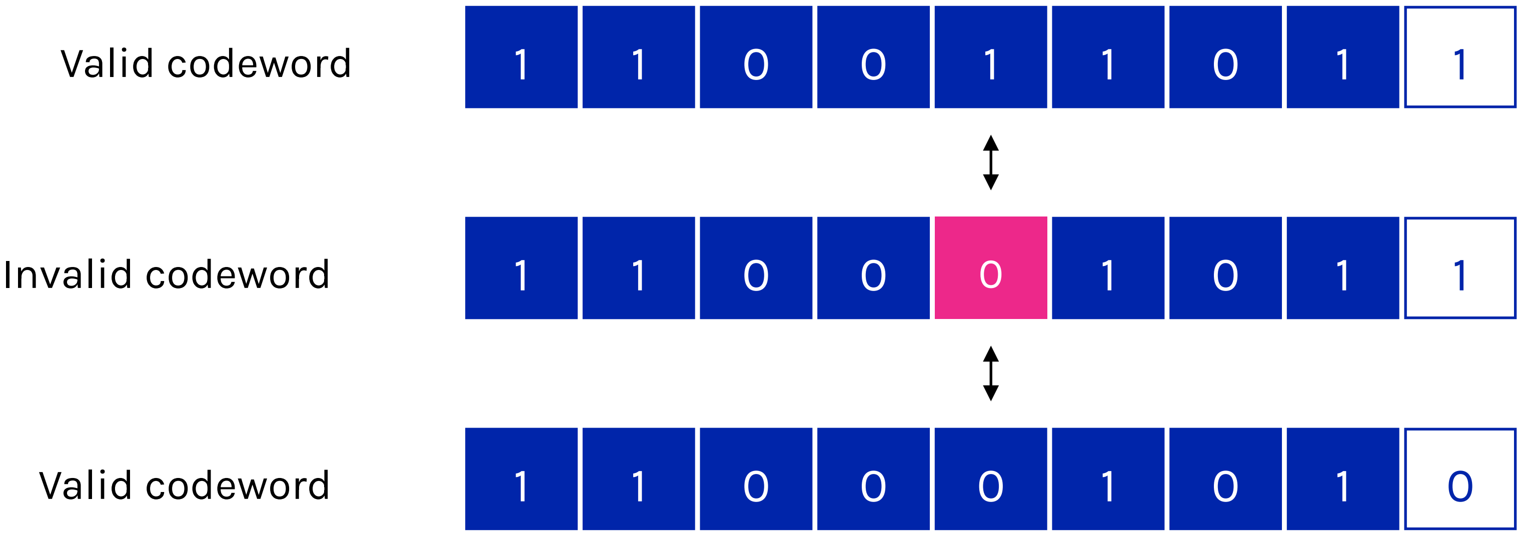
- What is the HD of the code?
- How many errors will it detect/correct?

What about larger errors?

1	1	0	0	1	1	0	1	1

Data bits Parity bit

Parity bit analysis



Parity bits have $HD = 2$ → Detect $d = 1$ bit error

Checksums

Idea: sum up data in k bit words

- Widely used in network protocols like TCP/UDP and IP
- Stronger protection than parity

Internet checksum

- Sum is defined in 1s compliment arithmetic (must add back carries)

> "The checksum field is the 16-bit one's compliment of the one's complement sum of all 16-bit words." - RFC 791

https://datatracker.ietf.org/doc/html/rfc791

Internet checksum example

Sending

- Arrange data in 16-bit word
- Put zero in the checksum position, add
- Add any carryover back to get 16 bits
- Negate (complement) to get sum

```
     0000000000000001
     1111001000000100
     1111010011110101
     1111011011110111
+    0000000000000000
     ----------------
    21101110111110001
   +                2   <----  Add carry back
     ----------------
     1101110111110011
(n)  0010001000001100   <----  Negade to get
                               1's complement
```

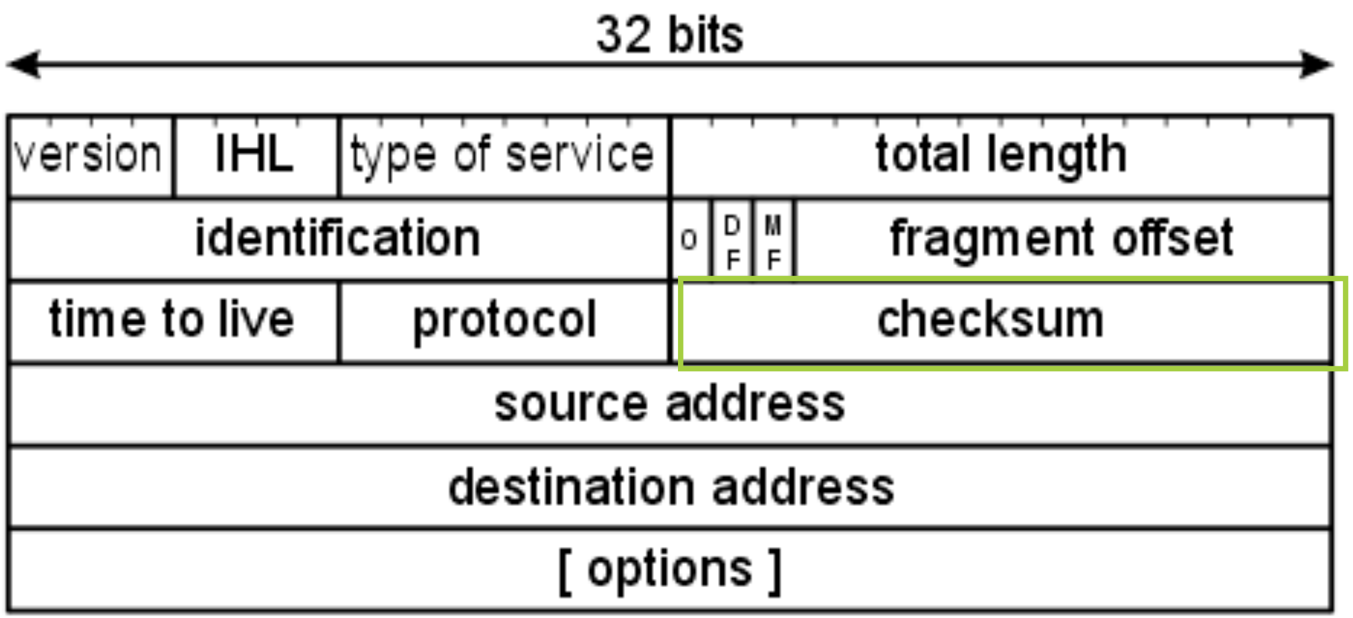
Internet checksum example

Receiving

- Arrange data in 16-bit word
- Checksum will be non-zero, add
- Add any carryover back to get 16 bits
- Negate (complement) and check if it is 0

```
      0000000000000001
      1111001000000100
      1111010011110101
      1111011011110111
    + 0010001000001100
      ----------------
     21111111111111101
    +                2  <---- Add carry back
      ----------------
      1111111111111111
  (n) 0000000000000000  <---- Negade to get 1's complement
```

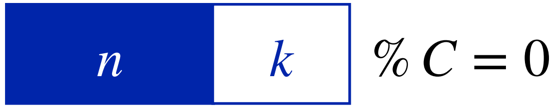
Internet checksum



Cyclic redundancy check (CRC)

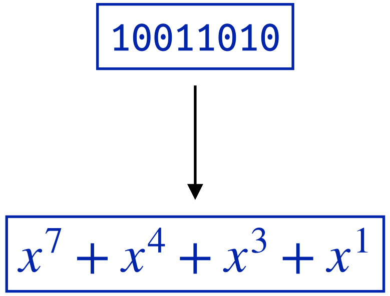
How does it work?

- Given n data bits, generate k check bits such that the $n + k$ bits are evenly divisible by a generator C
- Example: $n = 302, k = 1, C = 3$



The catch

- It is based on mathematics of finite fields, in which "numbers" represent polynomials
- This means we work with binary values and operate using modulo 2 arithmetic (XOR operations)



CRC procedures

Sending

- Extend the n data bits with k zero bits
- Divide by the generator value C (highest order is k, hence $k + 1$ terms)
- Keep remainder, ignore quotient
- Adjust k check bits by remainder

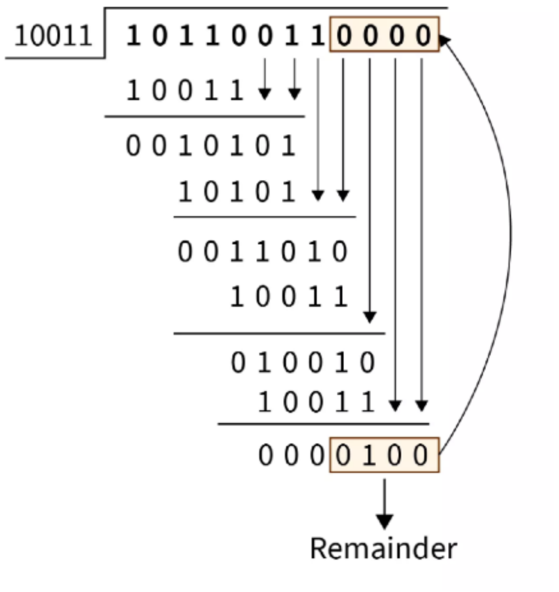
Receiving

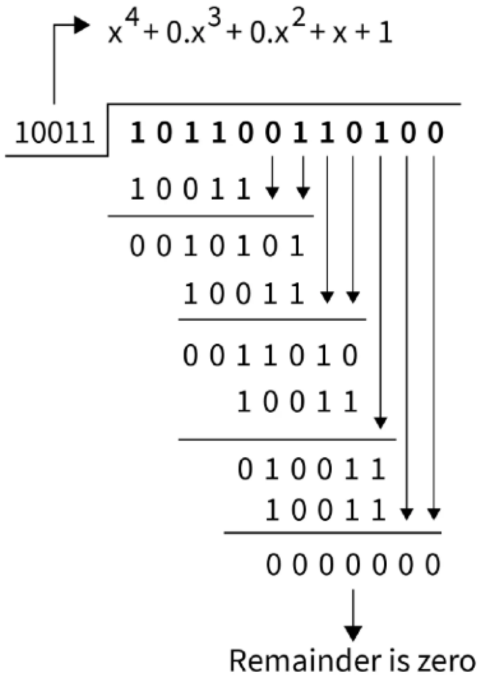
- Divide and check for zero remainder

Data bits
10110011

Check bits
$C(x) = x^4 + x^1 + 1$ $C = 10011$ $k = 4$

CRC example





Transmitted bits: 101100110100

CRC properties

Error protection depends on the generator

- For standard CRC-32 it is `10000010 01100000 10001110 110110111`

Properties

- $HD = 4$, detects up to triple bit errors
- Odd number of errors
- Bursts of up to k bits in error
- Not vulnerable to systematic errors like checksums

Why error correction is hard

If we had reliable check bits we could use them to narror down the position of the error

- The correction can then be easily done

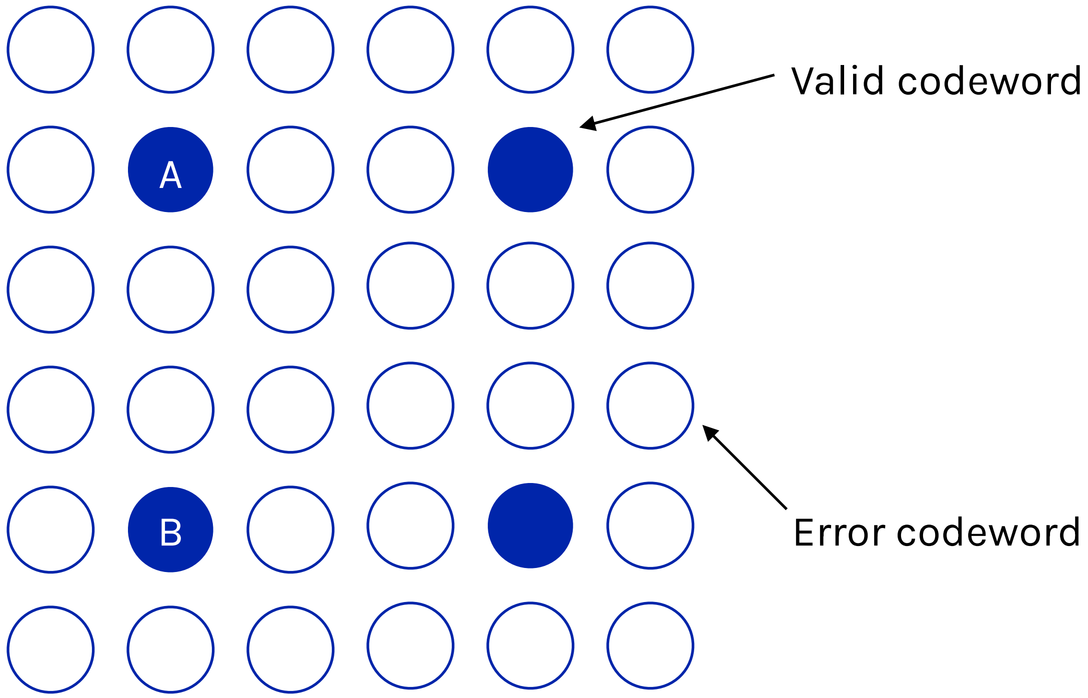
Errors can be in the check bits as well as the data bits!

- Data might even be correct, but not the check bits

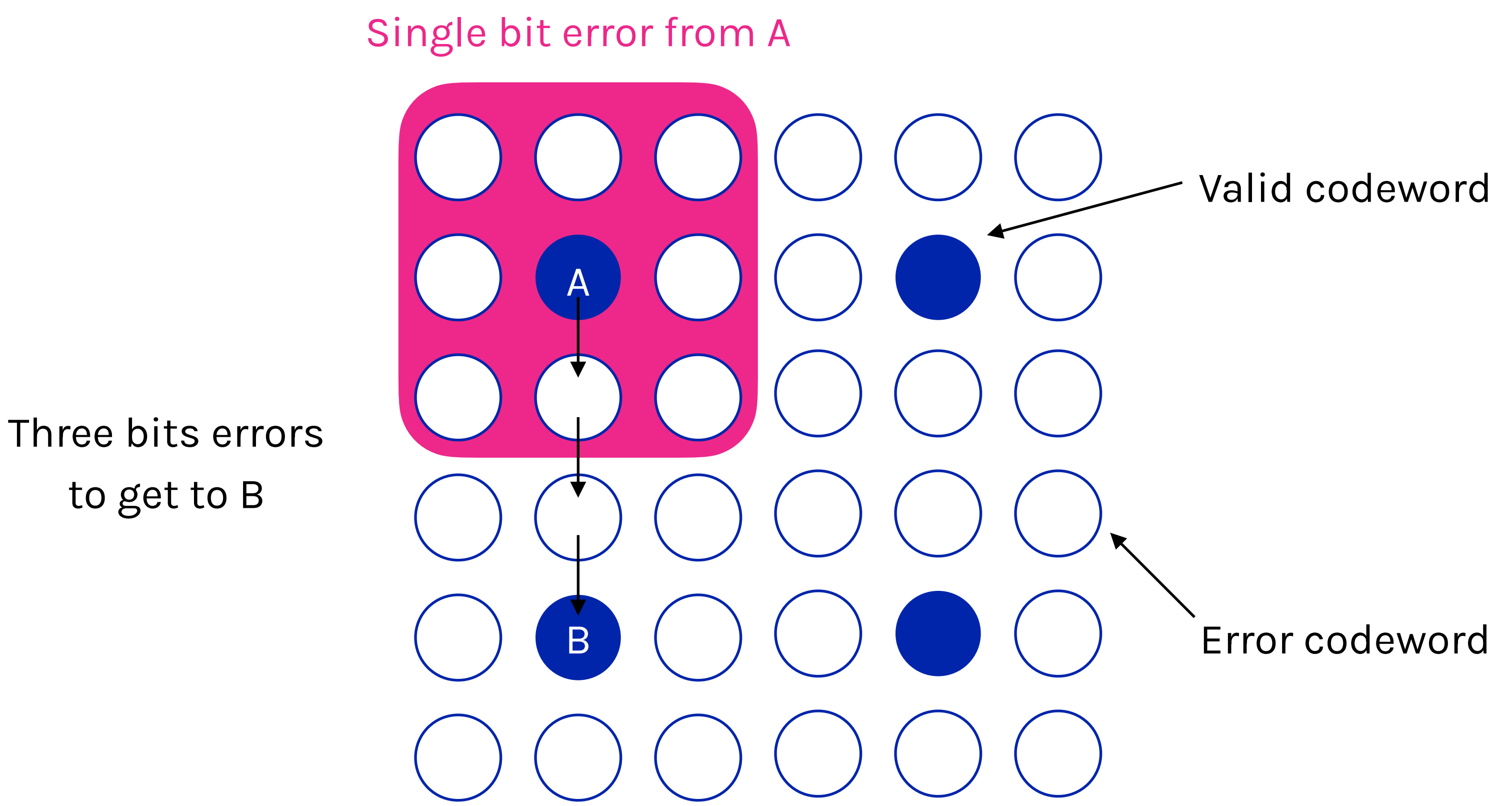
Intuition for error correcting code

- Suppose we construct a code with a Hamming distance of at least 3
- If we assume errors are only one bit, we can correct them by mapping an error to the closest valid codeword: works for d errors if the hamming distance $\geq 2d + 1$

Intuition



Intuition


Single bit error from A
A
B
Valid codeword
Three bits errors
to get to B
Error codeword

Required redundancy bits

Assume we have m data bits and r check bits, i.e., $n = m + r$

- For each of the 2^m valid codewords, there are n invalid codewords at a distance of one from it (formed by systematically inverting each of the n bits)
- Each of the 2^m valid codewords requires $n + 1$ bit patterns dedicated to it

The number of check bits required must satisfy

$$(n+1)2^m \leq 2^n \Rightarrow (m+r+1) \leq 2^r$$

Hamming code

A method for constructing a code with $HD = 3$

- Uses $m = 2^r - r - 1$, e.g., $m = 4, r = 3$
- Put the 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-th term in their values

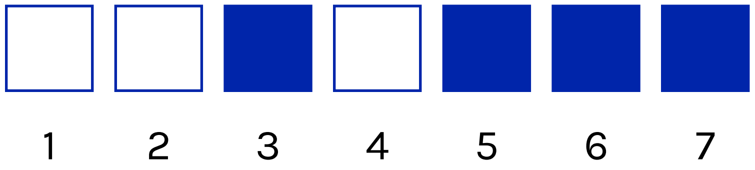
001	010	011	100	101	110	111
1	2	3	4	5	6	7

There is an easy way to correct errors

Hamming code example

Data: 0101, 3 check bits

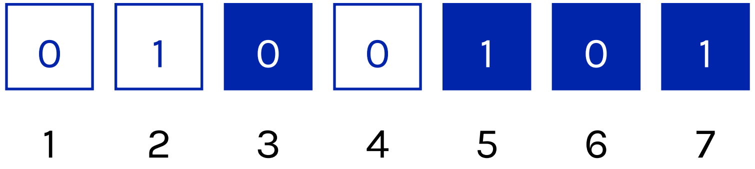
- 7-bit code, check bit positions 1, 2, and 4
- Check 1 covers positions 1, 3, 5, and 7
- Check 2 covers positions 2, 3, 6, 7
- Check 4 covers positions 4, 5, 6, 7



Hamming code example

Data: 0101, 3 check bits

- 7-bit code, check bit positions 1, 2, and 4
- Check 1 covers positions 1, 3, 5, and 7
- Check 2 covers positions 2, 3, 6, 7
- Check 4 covers positions 4, 5, 6, 7



Check bits calculation

- $p_1 = 0 + 1 + 1 = 0$
- $p_2 = 0 + 0 + 1 = 1$
- $p_4 = 1 + 0 + 1 = 0$

Hamming code example

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; flip bit to correct the error otherwise

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

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

$$p_2 = 1 + 0 + 0 + 1 = 0$$

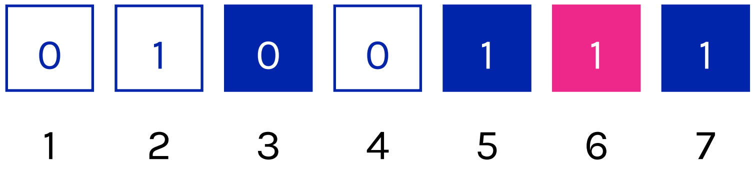
$$p_4 = 0 + 1 + 0 + 1 = 0$$

syndrome $= 0 \Rightarrow$ no error

Hamming code example

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; flip bit to correct the error otherwise



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

$$p_2 = 1 + 0 + 1 + 1 = 1$$

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

syndrome $= 110 \Rightarrow$ flip position 6

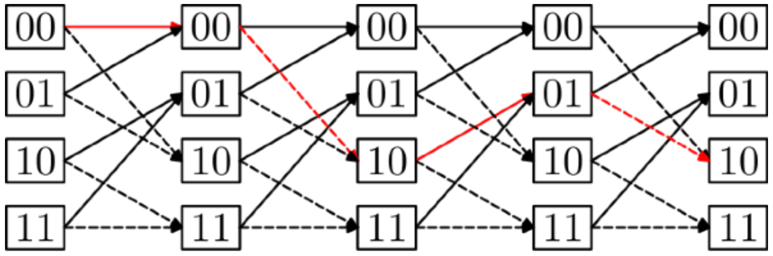
Data is `0101`, correct after the flip!

Other error correction codes

Real-world codes are more involved than Hamming code

Example: convolutional codes

- Take a stream of data and output a mix of the input bits
- Spreads the impact of errors across multiple bits in the encoded sequence; makes each output bit less fragile
- Decode using Viterbi algorithm (which can use bit confidence values)



https://en.wikipedia.org/wiki/Convolutional_code

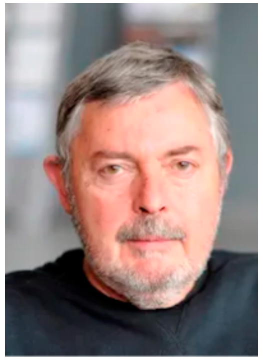
Other codes: Turbo codes

Turbo codes

- Evolution of convolutional codes
- Sends multiple sets of parity bits with payload, decodes sets togehter (e.g., Sudoku)
- Used in 3G and 4G cellular technologies

Low Density Parity Check (LDPC) codes

- Based on sparse matrices
- Decodes iteratively using a belief propagation algorithm



Claude Berrou

Robert Gallager

Error detection vs. correction

Which is better?

- Depends on the error pattern
- Example: 1000-bit messages with a bit error rate (BER) of 1 in 10.000

Which has less overhead?

- It still depends. We need to know more about the errors!

Error detection vs. correction

Assume bit errors are random

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

Error correction

- Needs ~10 check bits per message

Error detection

- Needs ~1 check bits per message plus 1000 bits retransmission 1/10 of the time

Error detection vs. correction

Assume errors come in bursts of 100

- Only 1 or 2 messages in 1.000 have significant (multi-bit) errors

Error correction

- Needs >> 100 check bits per message

Error detection

- Needs 32 check bits per message plus 1000 bits resend 2/1000 of the time

Error detection vs. correction

Error correction

- Needed when errors are expected
- When no time for retransmission
- Example: wireless networks (physical), real-time video streaming (application)

Error detection

- More efficient when errors are not expected
- When errors are large when they do occur
- Example: TCP, UDP, IP

Error correction in practice

Heavily used in the physical layer

- LDPC is the future, used for demanding links like 802.11
- 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

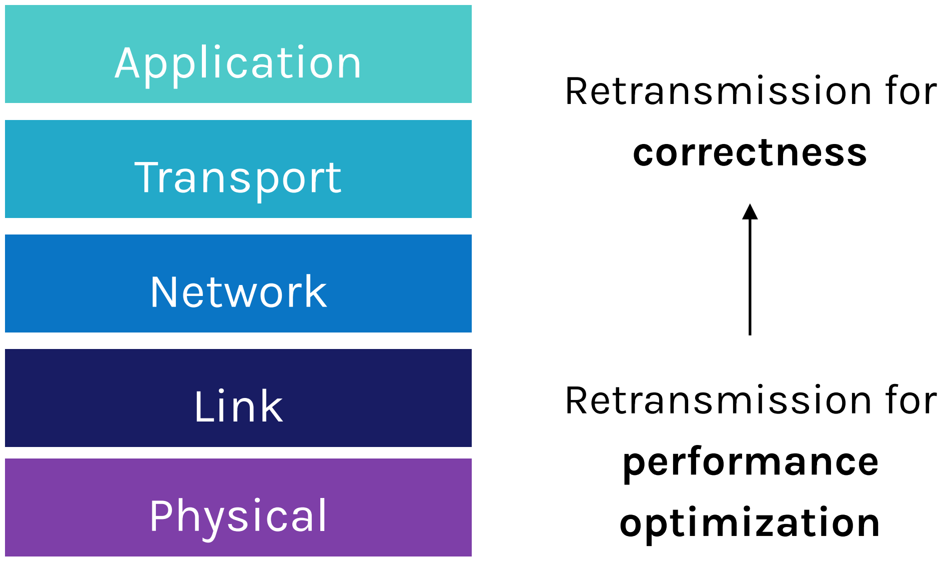
- Forward Error Correction (FEC)
- Normally with an erasure error model, e.g., Reed-Solumon (CDs, DVDs…)

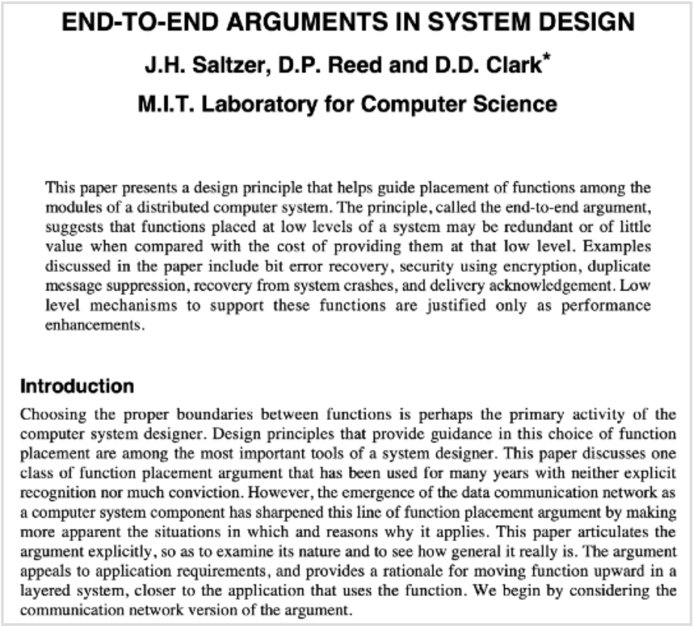
Reliability via Retransmission

Where should we place reliability functions?

- Application
- Transport
- Network
- Link
- Physical

End-to-end principle





END-TO-END ARGUMENTS IN SYSTEM DESIGN

J.H. Saltzer, D.P. Reed and D.D. Clark*

M.I.T. Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgement. Low level mechanisms to support these functions are justified only as performance enhancements.

Introduction

Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer. Design principles that provide guidance in this choice of function placement are among the most important tools of a system designer. This paper discusses one class of function placement argument that has been used for many years with neither explicit recognition nor much conviction. However, the emergence of the data communication network as a computer system component has sharpened this line of function placement argument by making more apparent the situations in which and reasons why it applies. This paper articulates the argument explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements, and provides a rationale for moving function upward in a layered system, closer to the application that uses the function. We begin by considering the communication network version of the argument.

[TOCS'84]

ARQ (automatic repeat request)

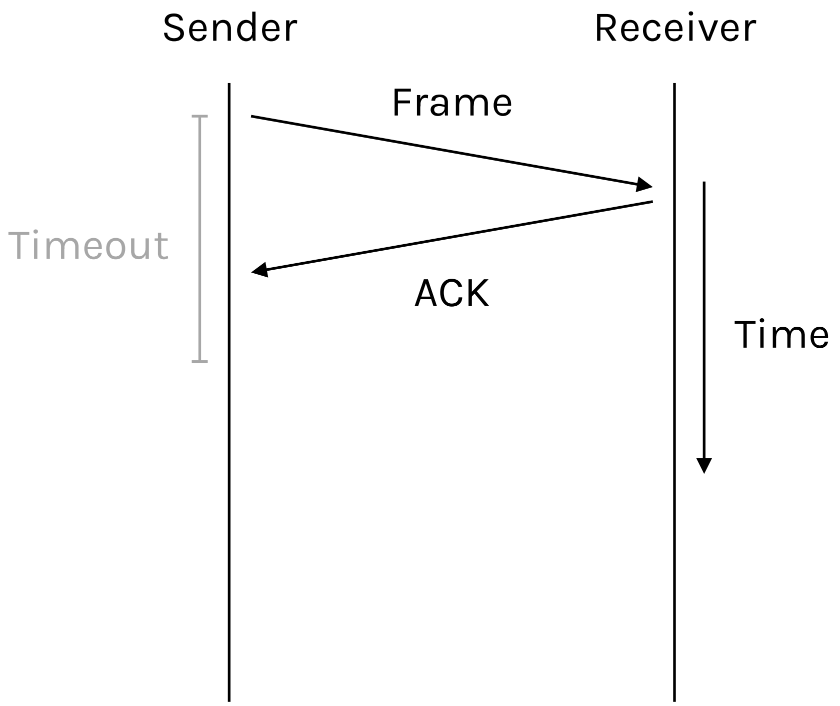
ARQ is often used when errors are common or must be corrected

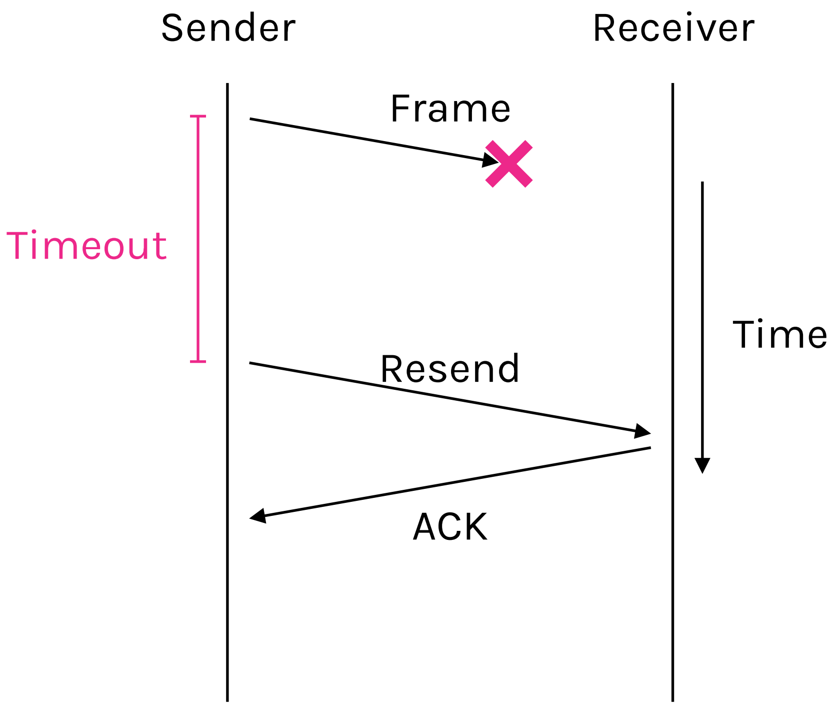
- For example: WiFi (at the link layer), and TCP (at the transport layer)

Rules at the sender and receiver

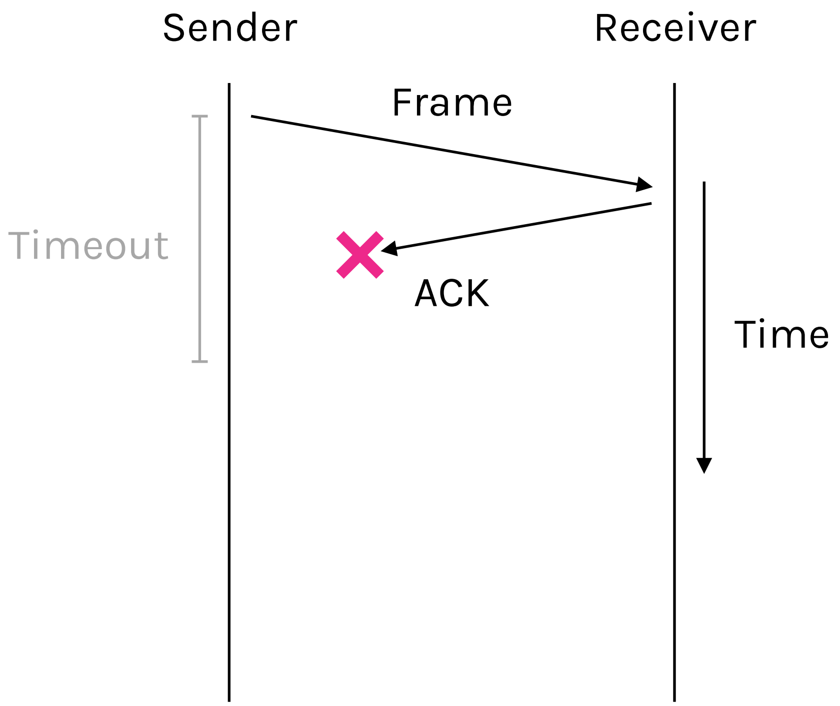
- Receiver automatically acknowledges correct frames with an acknowledgement (ACK)
- Sender automatically resends after a timeout, until an ACK is received

ARQ operations

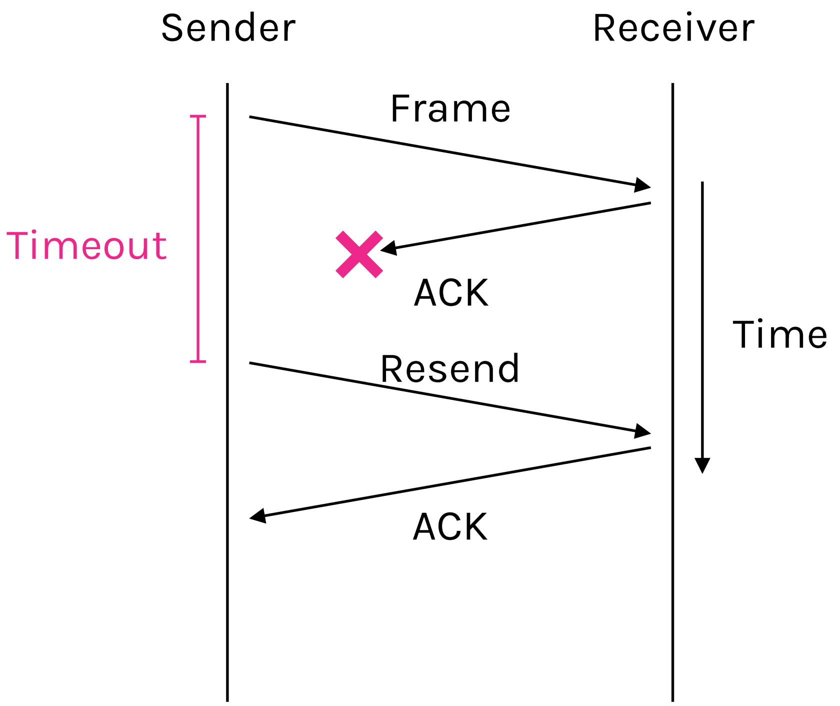




Problems with ARQ

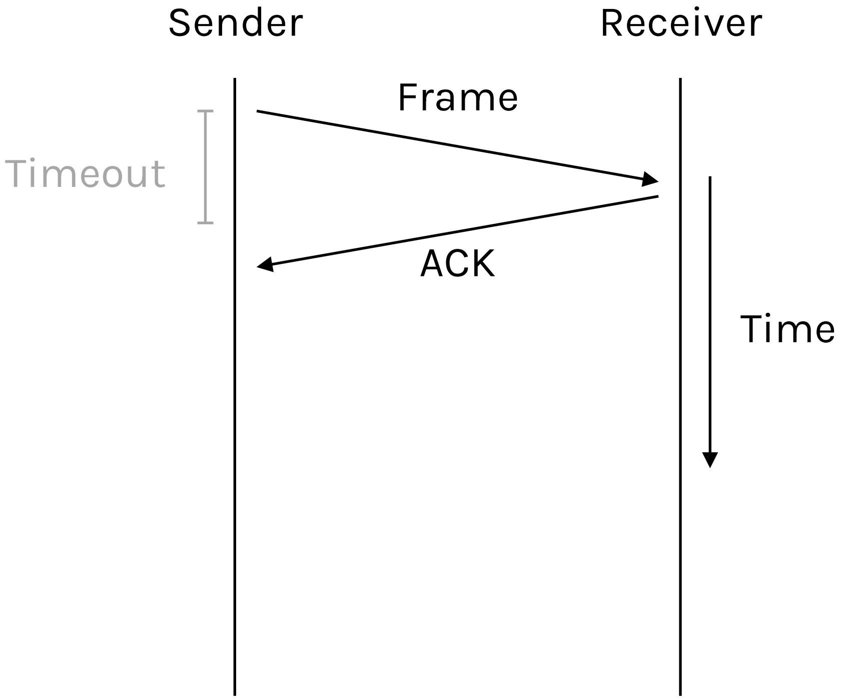


ACK can be lost

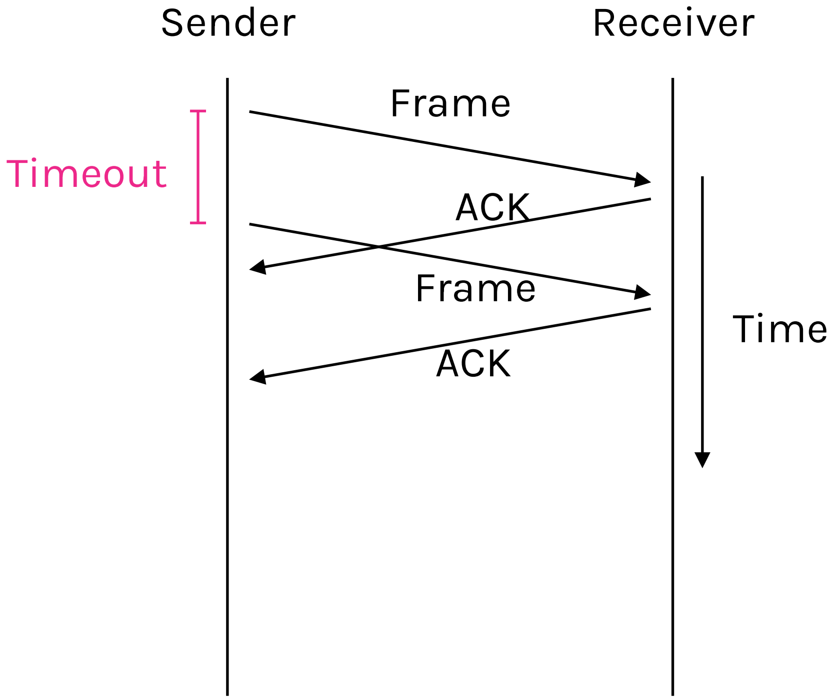


Was it a new frame or a repeated frame?

Problems with ARQ



Timeout maybe too early



Was it a new frame or a repeated frame?

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

Any ideas?

Timeouts

Timeouts should be

- Not too big: link goes idle, low efficiency
- Not too small: spurious resend, low efficiency

Fairly easy on a local area network (LAN)

- Clear worst case, little variation

Fairly difficult over the Internet

- Much variation, no obvious bound
- We will revisit this problem with TCP later

Sequence numbers

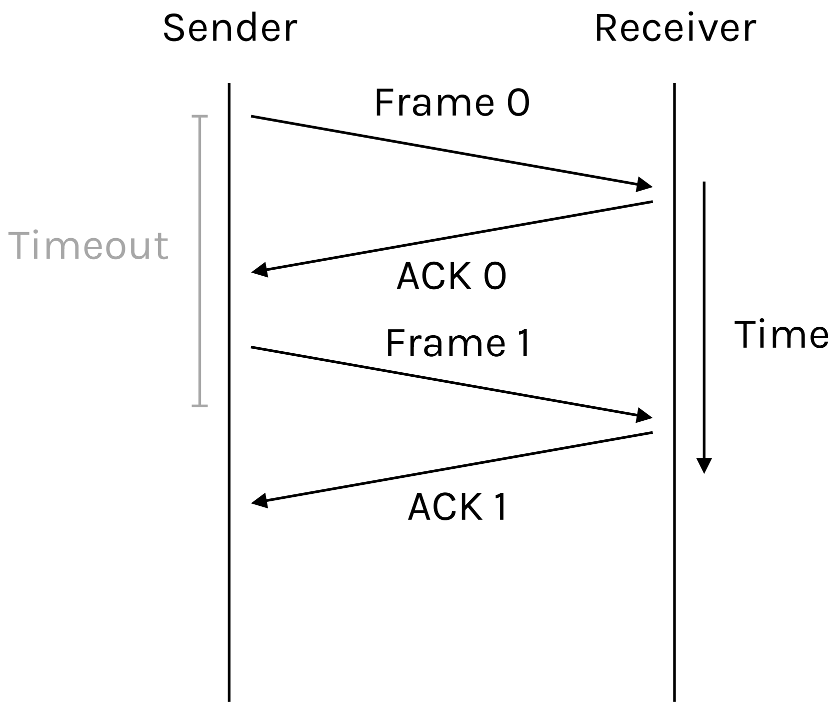
Frames and ACKs must both carry sequence numbers

- Sender and receiver agree on the status of each frame to ensure correctness

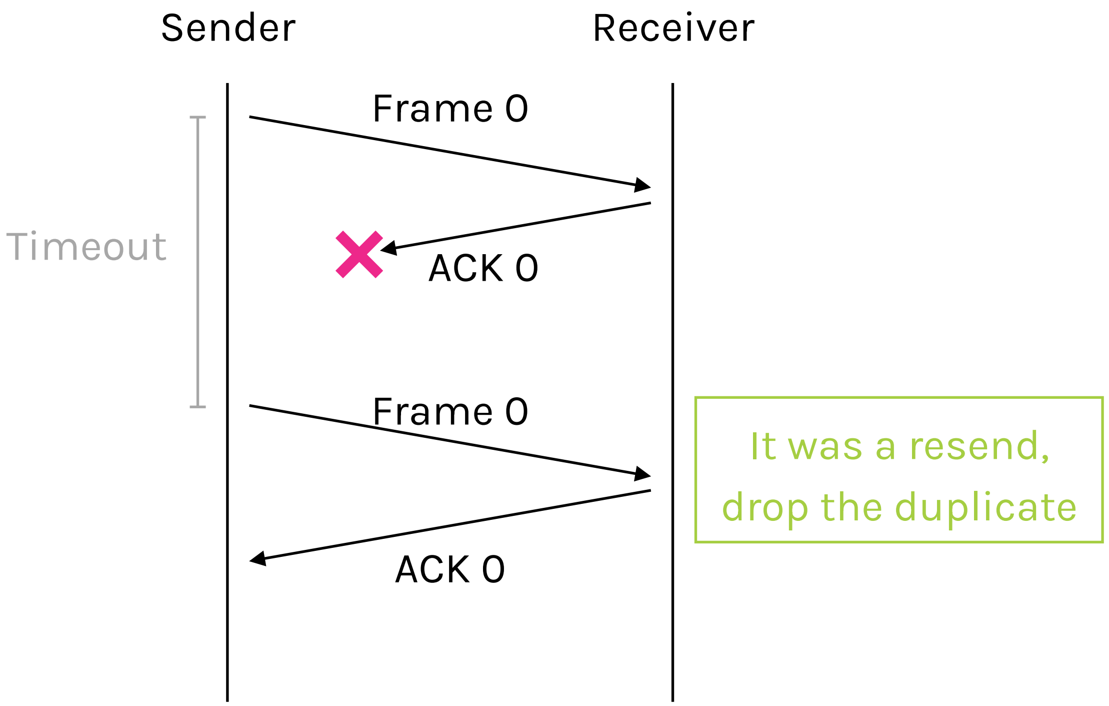
To distinguish the current frame from the next one, a single bit (two numbers) is sufficient

- So called stop-and-wait

Stop-and-wait

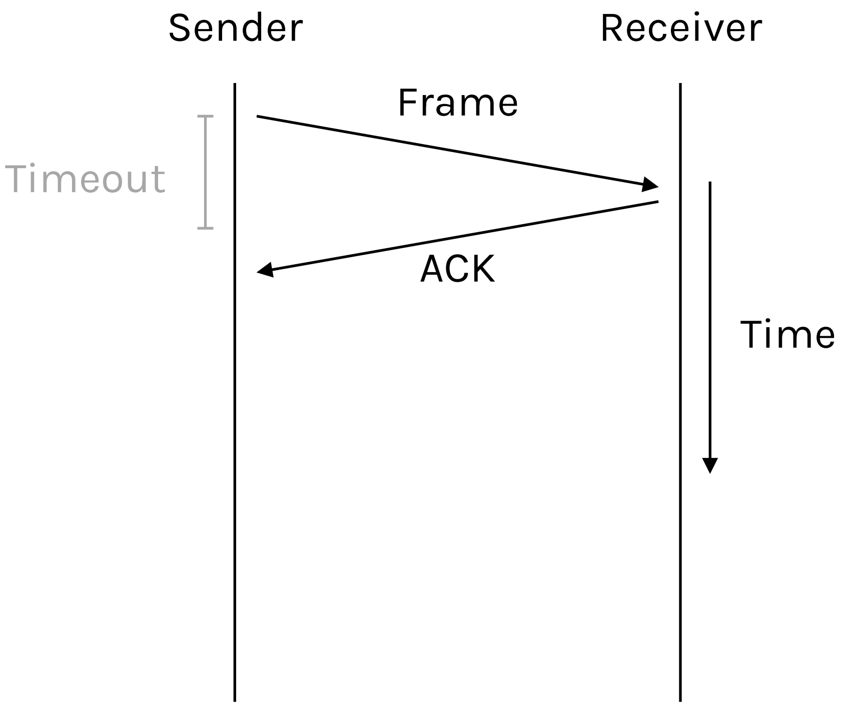


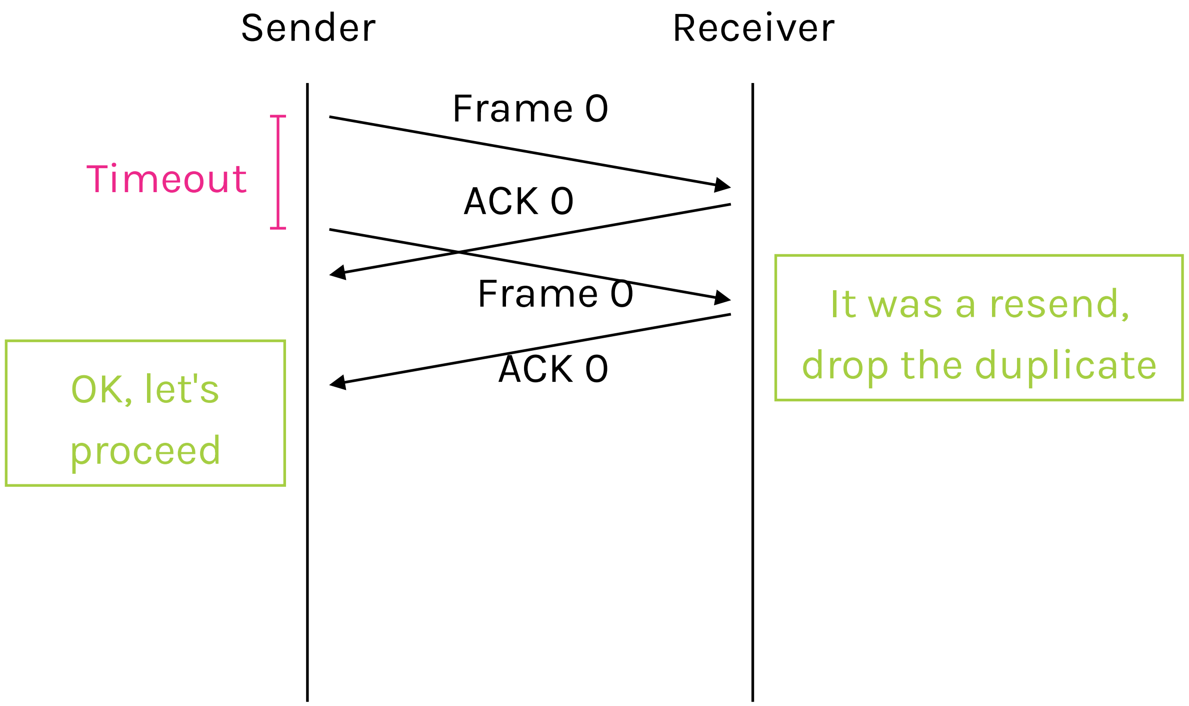
Normal case



With ACK lost

Stop-and-wait





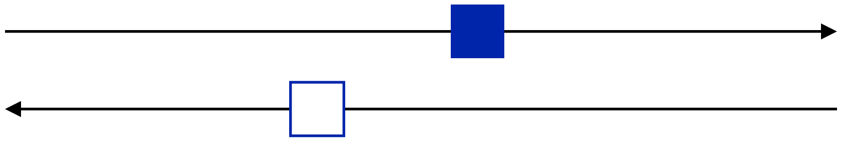
Limitations of stop-and-wait

Allows only a single frame to be outstanding from the sender

- Good for LAN, not efficient for high BDP

Example: $R = 1$ **Mbps,** $D = 50$ **ms**

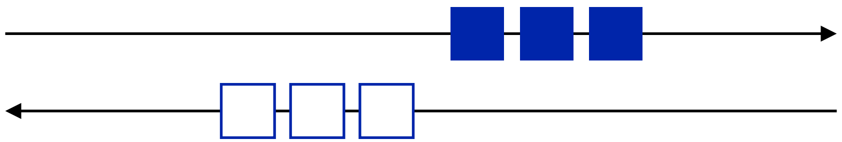
- Per-frame latency: 100 ms → 10 frames/sec
- Link utilization: $(10 \times 1500 \times 8)/(1 \times 10^6) = 12\,\%$
- What if $R = 10$ Mbps?



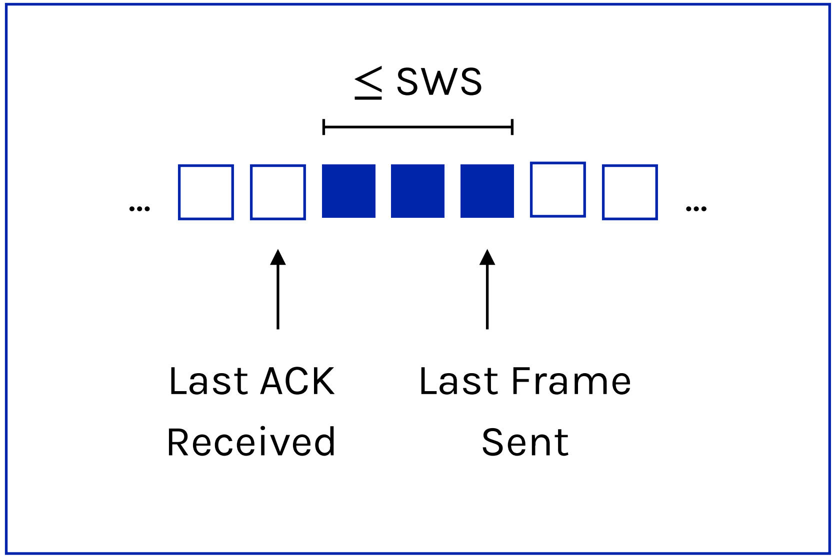
Sliding window

Generalization of stop-and-wait

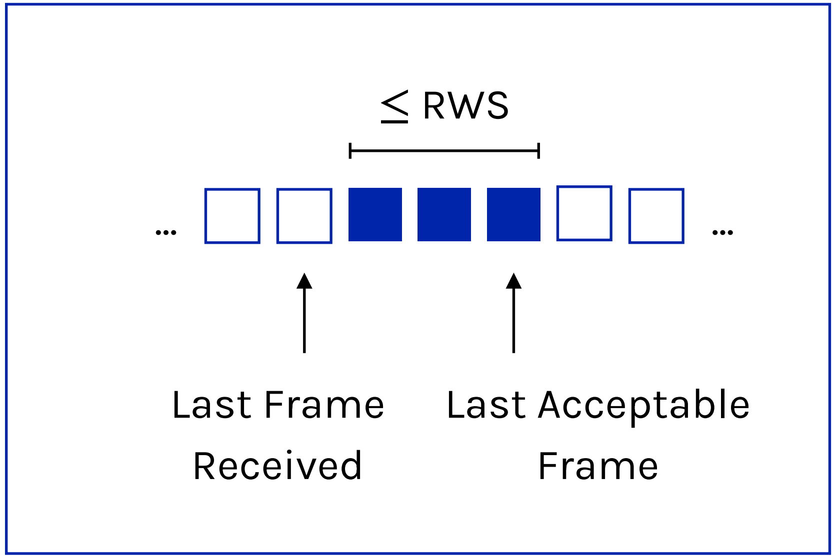
- Allows W frames to be outstanding
- Can send W frames per RTT (=$2D$)
- Various options for numbering frames/ACKs and handling loss



Sliding window



Sender



Receiver

We will discuss more about it in the transport layer

Summary

Framing

- Byte stuffing
- Bit stuffing
- Coding violations

Error detection and correction

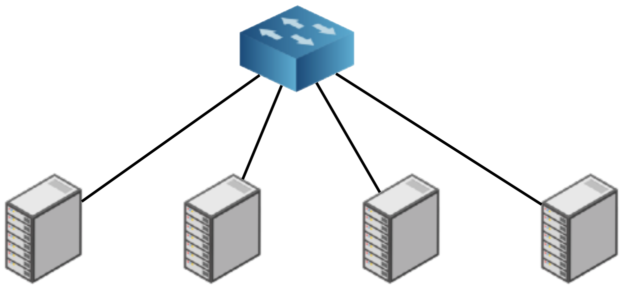
- Hamming distance and requirements
- Error detection codes (parity, checksum, CRC)
- Error correction codes (Hamming code)

Reliability via retransmission

- Automatic repeat request (ARQ)
- Stop-and-wait
- Sliding windows

Next time: data link layer

Hub vs. switch



Hosts (servers, laptops)

How to interconnect **more than two end-devices** on a network?

Further reading material

Andrew S. Tanenbaum, David J. Wetherall. Computer Networks (5th edition).

- Chapter 3: The Data Link Layer

Larry Peterson, Bruce Davie. Computer Networks: A Systems Approach.

- Chapters 2.3: Framing
- Chapter 2.4: Error Detection
- Chapter 2.5: Reliable Transmission