Lecture 5: Error Handling

CSE 123: Computer Networks
Alex C. Snoeren
Lecture 5 Overview

- Framing wrap-up
  - Byte stuffing
  - Clock-based framing

- Error handling through redundancy

- Hamming Distance
  - When we can detect
  - When we can correct

- Checksums
Byte Stuffing

- Same as bit stuffing, except at byte (character) level
  - Generally have two different flags, STX and ETX
  - Found in PPP, DDCMP, BISYNC, etc.
- Need to stuff if either appears in the payload
  - Prefix with another special character, DLE (data-link escape)
  - New problem: what if DLE appears in payload?
- Stuff DLE with DLE!
  - Could be as bad as 50% efficient to send all DLEs
Clock-Based Framing

- So far, we’ve based framing on what’s on the wire
  - Any bit errors may throw off our framing
  - What happens with missed flag? Spurious flag?

- An alternative is to base framing on external clock
  - Kind of like Phy-layer signaling: sample at specific intervals
  - This is what SONET does, among others

- Significant engineering tradeoffs
  - No extra bits needed in the data stream itself, but…
  - Need tight clock synchronization between sender and receiver
SONET

- Synchronous Optical NETwork
  - Engineering goal to reduce delay and buffering
- All frames take same amount of time
  - Independent of bit rate!
- Each frame starts with signal bits
  - Can synch clock just like PLL—look for periodic signal bits
  - No need to stuff; signal pattern is unlikely, so won’t be periodic in data
- Keep sync within frames with transitions
  -Encoded using NRZ, but
  - Data is XORed with special 127-bit pattern
  - Creates lots of transitions, makes signal pattern unlikely
When Things Go Wrong

- Clock drift may confuse frame boundaries
  - Read the end of one frame and beginning of the next

- What happens if there are **bit errors** on channel?
  - We might misinterpret sentinels as data or vice versa
  - What will the frames look like?

- In general, need some way to make sure we’re OK
  - Error detection—and perhaps correction
Error Detection

- Implemented at many layers
  - We’ll mainly focus on link-layer techniques today

CSE 123 – Lecture 5: Error Handling
Basic Idea

- The problem is data itself is not self-verifying
  - Every string of bits is potentially legitimate
  - Hence, any errors/changes in a set of bits are equally legit

- The solution is to reduce the set of potential bitstrings
  - Not every string of bits is allowable
  - Receipt of a disallowed string of bits means the original bits were garbled in transit

- Key question: which bitstrings are allowed?
Let’s start simple, and consider fixed-length bitstrings
- Reduce our discussion to $n$-bit substrings
- E.g., 7-bits at a time, or 4 bits at a time (4B/5B)
- Or even a frame at a time

We call an allowable sequence of $n$ bits a **codeword**
- Not all strings of $n$ bits are codewords!
- The remaining $n$-bit strings are “space” between codewords

Rephrasing previous question: how many codewords with how much space between them?
Hamming Distance

- Distance between legal codewords
  - Measured in terms of number of bit flips

- Efficient codes are of uniform Hamming Distance
  - All codewords are equidistant from their neighbors
2d+1 Hamming Distance

- Can **detect** up to $2d$ bit flips
  - The next codeword is always $2d+1$ bit flips away
  - Any fewer is guaranteed to land in the middle

- Can **correct** up to $d$ bit flips
  - We just move to the closest codeword
  - Unfortunately, no way to tell how many bit flips
    - E.g., 1, or $(2d+1)-1$?
Encoding

- We’re going to send only codewords
  - Non-codewords indicate errors to receiver

- But we want to send any set of strings
  - Need to embed arbitrary input into sequence of codewords

- We’ve seen this before: 4B/5B
  - We want more general schemes
Simple Embedding: Parity

- Code with Hamming Distance 2
  - Can detect one bit flip (no correction capability)
- Add extra bit to ensure odd(even) number of ones
  - Code is 66% efficient (need three bits to encode two)
  - Note: Even parity is simply XOR
Simple Correction: Voting

- Simply send each bit $n$ (3 in this example) times
  - Code with Hamming Distance 3 ($d=1$)
  - Can detect 2 bit flips and correct 1

- Straightforward duplication is extremely inefficient
  - We can be much smarter about this
Two-Dimensional Parity

- Start with normal parity
  - \( n \) data bits, 1 one parity bit
- Do the same across rows
  - \( m \) data bytes, 1 parity byte
- Can detect up to 3 bit errors
  - Even most 4-bit errors
- Can correct any 1 bit error
  - Why?
Per-Frame Detection Codes

- Want to add an error detection code per frame
  - Frame is unit of transmission; all or nothing.
  - Computed over the entire frame—including header! Why?
- Receiver checks EDC to make sure frame is valid
  - If frame fails check, throw it away
- We could use error-correcting codes
  - But they are less efficient, and we expect errors to be rare
  - Counter example: satellite communication
Checksums

- Simply sum up all of the data in the frame
  - Transmit that sum as the EDC

- Extremely lightweight
  - Easy to compute fast in hardware
  - Fragile: Hamming Distance of 2

- Also easy to modify if frame is modified in flight
  - Happens a lot to packets on the Internet

- IP packets include a 1’s compliment checksum
IP Checksum Example

- 1’s compliment of sum of words (not bytes)
  - Final 1’s compliment means all-zero frame is not valid

```c
u_short cksum(u_short *buf, int count) {
    register u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```
Checksum in Hardware

- Compute checksum in Modulo-2 Arithmetic
  - Addition/subtraction is simply XOR operation
  - Equivalent to vertical parity computation

- Need only a word-length shift register and XOR gate
  - Assuming data arrives serially
  - All registers are initially 0
Checksum Example

0101001111010010101111010001110101101001101111011110110
Checksum Example

0101001111010010101111010001110101101001101111011110110

000000000000000000

+ 0101...

Data

Parity Byte
Checksum Example

0101001111010010101111010001110110110100110111101110110

0 0 0 0 0 0 0 0 + 1010...

Data ↑ 0
Checksum Example

01010011110100101011110100011101011010011011111011110110

Data

01

0100…
Checksum Example

0101001111010010101111010001110101101001101111101110110

Data →↓ 010

0 0 0 0 0 0 1 0 + 1001...
Checksum Example

0101001111010010101111010001110101101001101111101110110

Data ↑ 0101
Checksum Example

0101001111010010101111010001110101101001101111101110110

\[ 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 1 \rightarrow + \rightarrow 1101... \]

Data \[ \uparrow \quad 01010011 \]
Checksum Example

01010111101001010111101000111010110100110111101110110

Data: 01010011
Parity Byte: 1

Parity Check:
1 0 1 0 0 1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 0

CSE 123 – Lecture 5: Error Handling
Checksum Example

010100111101001010111010001110101101001101111101110110

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

Data

Parity Byte

01010011
11
10

0100...
Checksum Example

0101001111010010010101111010001110101100110111101111011110110

1  0  0  0  0  0  0  1

Data

Parity Byte

01010011
11010010
10000001
Checksum Example

0101001111010010101111010011101110111101110110

0 0 0 0 0 1 0 + 0111...

Data

Parity Byte

01010011
11010010
1

0
Checksum Example

010100111101001010111010001110101101001101111011110110 +

1 1 1 1 0 1 1 0

01010011 11010010 10111101 00011101 01101001 10111110

Data

Parity Byte

01010011
11010010
10111101
00011101
01101001
11110110

CSE 123 – Lecture 5: Error Handling
For Next Class

- We’ll finish error detection and talk about reliable transport

- Read 2.5 in P&D