Lecture 4: Error Handling

CSE 123: Computer Networks
Alex C. Snoeren

HW 1 due Thursday
Lecture 4 Overview

- Error handling through redundancy
  - Adding extra bits to the frame

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

- Checksum

- Cyclic Remainder Check (CRC)
Error Detection

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

Simple Correction: Voting

- 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?
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*
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);
}
```
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

01010011110100101011110100011101011011011011111011110110

Data

Parity Byte

01010011
11010010
10111101
00011101
01101001
10111110
11110110
Checksum Example

0101001111010010101111010011101101100111011101101110

0 0 0 0 0 0 0 0 + 0101...

Data

Parity Byte
Checksum Example

01010011110100101011110100011101011010011011111011110110

0 0 0 0 0 0 0 + 01010...

Data 0
Checksum Example

0101001111010010101111010011101011010011011111011110110

\[ 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 1 \] + \quad 0100\ldots

Data \uparrow \downarrow 01
Checksum Example

Data 010

01010011110100101011110100011101011010011011111011110110

0 0 0 0 0 0 1 0

+ 1001...
Checksum Example

Data

01010011110100101011110100011101011010011011111011110110
Checksum Example

Data: 01010011

 checksum: 010101101001010111101001101101001101111011110110

Checksum: 1101...
Checksum Example

010100111101001010101110100011101011010011011111011110110
+ 1010...

Data

Parity Byte

CSE 123 – Lecture 4: Error Handling
Checksum Example

Data
Parity Byte

01010011110100101011011001101001101101111011101110110

0 1 0 0 1 0 1 0 1 1 1 1 0 1 0 1 1 0 0 1 1 0 1 0 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0

01010011

11

Parity Byte

10

0100...
Checksum Example

01010011110100101011110100011101011010011011111011110110

1  0  0  0  0  0  0  1

Parity Byte

Data

01010011
11010010

Parity Byte

10000001

1011...
Checksum Example

010100111101001010111101111010011110011011110100110111101001100011011110111101011110110

0 \[ \rightarrow \] 0 \[ \rightarrow \] 0 \[ \rightarrow \] 0 \[ \rightarrow \] 0 \[ \rightarrow \] 0 \[ \rightarrow \] 0 \[ \rightarrow \] 1 \[ \rightarrow \] 0 \[ \rightarrow \] + \[ \rightarrow \] 0111...
Checksum Example

0101001111010010101111010001111010110010011011111011110110

1 1 1 1 0 1 1 0

Data

Parity Byte

01010011
11010010
10111011
00011101
01101001
01101001
10111110

CSE 123 – Lecture 4: Error Handling
Checksums are easy to compute, but very fragile
- In particular, **burst** errors are frequently undetected
- We’d rather have a scheme that “smears” parity

Need to remain easy to implement in hardware
- So far just shift registers and an XOR gate

We’ll stick to Modulo-2 arithmetic
- Multiplication and division are XOR-based as well
- Let’s do some examples…
Modulo-2 Arithmetic

- **Multiplication**
  
  \[
  \begin{array}{c}
  1101 \\
  110 \\
  \hline \\
  0000 \\
  11010 \\
  110100 \\
  \hline \\
  101110
  \end{array}
  \]

- **Division**
  
  \[
  \begin{array}{c}
  1101 \\
  110 \underline{101110} \\
  110 \\
  110 \\
  110 \\
  110 \\
  \hline \\
  011 \\
  000 \\
  \hline \\
  110
  \end{array}
  \]
Cyclic Remainder Check

- Idea is to divide the incoming data, $D$, rather than add
  - The divisor is called the generator, $g$
- We can make a CRC resilient to $k$-bit burst errors
  - Need a generator of $k+1$ bits
- Divide $2^kD$ by $g$ to get remainder, $r$
  - Remainder is called frame check sequence
- Send $2^kD+r$
  - Note $2^kD$ is just $D$ shifted left $k$ bits
  - Remainder must be at most $k$ bits
- Receiver checks that $(2^kD+r)/g = 0$
We’re actually doing polynomial arithmetic
  - Each bit is actually a coefficient of corresponding term in a $k^{th}$-degree polynomial

$$1101 \text{ is } (1 \cdot X^3) + (1 \cdot X^2) + (0 \cdot X^1) + (1 \cdot X^0)$$

Why do we care?
  - Can use the properties of finite fields to analyze effectiveness
  - Says any generator with two terms catches single bit errors
CRC Example Encoding

\[
\begin{align*}
x^3 + x^2 + 1 &= 1101 & \text{Generator} \\
x^7 + x^4 + x^3 + x &= 10011010 & \text{Message}
\end{align*}
\]

\[1101 \quad 10011010000
\]

\[1101 \quad 1001 \quad 1101
\]

\[1101 \quad 1000 \quad 1101
\]

\[1101 \quad 1011
\]

\[1100 \quad 1101
\]

\[101 \quad 1000 \quad 1101
\]

\[D \mod g \]

Result:
Transmit message followed by remainder:
\[10011010101\]
CRC in Hardware

- Key observation is only subtract when MSB is one
  - Recall that subtraction is XOR
  - No explicit check for leading one by using as input to XOR

- Hardware cost very similar to checksum
  - We’re only interested in remainder at the end
  - Only need \( k \) registers as remainder is only \( k \) bits
CRC Example Decoding

$x^3 + x^2 + 1 = 1101$  
$\ x^{10} + x^7 + x^6 + x^4 + x^2 + 1 = 10011010101$  

$\text{Generator}$  
$\text{Received Message}$

$k + 1$ bit check sequence $g$, equivalent to a degree-$k$ polynomial

$1101$  
$10011010101$  
$1001$  
$1101$  
$1000$  
$1101$  
$1011$  
$1101$  
$1100$  
$1101$

$D \mod g$

Remainder

$1101$

$1101$

$0$

Result:

CRC test is passed
CSE 123 – Lecture 4: Error Handling

CRC Example Failure

\[ x^3 + x^2 + 1 = 1101 \]
\[ x^{10} + x^7 + x^5 + x^4 + x^2 + 1 = 10010110101 \]

\[ k + 1 \text{ bit check sequence } g, \text{ equivalent to a degree-}k \text{ polynomial} \]

\[ \text{Received message} \]

\[ \text{Two bit errors} \]

\[ \text{Result: CRC test failed} \]
## Common Generators

<table>
<thead>
<tr>
<th></th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-8</td>
<td>(x^8 + x^2 + x^1 + 1)</td>
</tr>
<tr>
<td>CRC-10</td>
<td>(x^{10} + x^9 + x^5 + x^4 + x^1 + 1)</td>
</tr>
<tr>
<td>CRC-12</td>
<td>(x^{12} + x^{11} + x^3 + x^2 + x^1 + 1)</td>
</tr>
<tr>
<td>CRC-16</td>
<td>(x^{16} + x^{15} + x^2 + 1)</td>
</tr>
<tr>
<td>CRC-CCITT</td>
<td>(x^{16} + x^{12} + x^5 + 1)</td>
</tr>
<tr>
<td>CRC-32</td>
<td>(x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1)</td>
</tr>
</tbody>
</table>
Add redundant bits to detect if frame has errors
- A few bits can detect errors
- Need more to correct errors

Strength of code depends on Hamming Distance
- Number of bitflips between codewords

Checksums and CRCs are typical methods
- Both cheap and easy to implement in hardware
- CRC much more robust against burst errors
For Next Class

- Read 2.5, 5.1 in P&D

- Homework 1 due at the beginning of class