Lecture 4:
Reliable Transmission

HW 1 due next FRIDAY
Lecture 4 Overview

- Finishing Error Detection
  - Checksums
  - Cyclic Remainder Check (CRC)

- Handling errors
  - Automatic Repeat Request (ARQ)
  - Acknowledgements (ACKs) and timeouts
  - Stop-and-Wait
Checksums

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

● Extremely lightweight

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

● IP packets include a 1’s complement checksum

What is the Hamming Distance of a checksum-based code?

A. 1
B. 2
C. It depends
D. I don’t know
IP Checksum Example

- 1’s complement of sum of 16-bit words (not bytes)

```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);
}
```

What is the IP checksum of an all-zero packet?

A. 0x0000  
B. 0xFFFF  
C. 0x1111  
D. I don’t know
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
Modulo-2 Arithmetic

- Addition & subtraction are XOR
  - $1 + 1 = 0$; $0 - 1 = 1$ (no carries!)

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

- Division
  
  \[
  \begin{array}{c}
  \underline{1101} \\
  110 \\
  \hline \\
  101110 \\
  110 \\
  \hline \\
  111 \\
  110 \\
  \hline \\
  011 \\
  000 \\
  \hline \\
  110 \\
  \end{array}
  \]

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011111010010111010001101011011001101111011110110

Data

Parity Byte

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011111010010101111010001110101101101101111101101111011110110

+ 0101...

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011110100110111101001101110111111011110110+1010…

Data 0
Checksum Example

010100111101001010111101000111010110100110111101111011011110

Data 01

0100...
Checksum Example

0101001110100101011101000111010110100110111101110110

0 0 0 0 0 1 0 + 1001...

Data 010
Checksum Example

0101001111010010101111010001110101101001101111011110110

Data  0101
Checksum Example

Data: 0101001111010111101000111101101100111011101111011110110

01010011101010111101000111101101100111011101111011110110
Checksum Example

01010011110100101011110100011101011010011011111011110110

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

Data

Parity Byte

1

1
Checksum Example

0101001111010010101111010001110101101001101111110111101110

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

+ 0100...

Data 01010011
Parity Byte 11

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

Data: 01010011
Parity Byte: 11010010

101100001 + 1011...
Checksum Example

0101001111010010101111010001110101101001101111101110110

\[\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & + \\
\end{array}\]

0111...

Data

Parity Byte

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

010100111101001010111101000111010110100110111110111011101110

Data
- 01010011
- 11010010
- 10111101
- 00011101
- 01101001
- 01111101

Parity Byte
- 11101110
From Sums to Remainders

- 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
  - All we need are 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…
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 $2^kD - r$ (i.e., $2^kD$ XOR $r$)
  - Note $2^kD$ is just $D$ shifted left $k$ bits
  - Remainder must be at most $k$ bits
- Receiver checks that $(2^kD-r)/g = 0$
Error Detection – CRC

- View data bits, D, as a binary number
- Choose r+1 bit pattern (generator), G
- Goal: choose r CRC bits, R, such that
  - \(<D \mid R> \) exactly divisible by G (modulo 2)
  - Receiver knows G, divides \(<D \mid R> \) by G. If non-zero remainder: error detected!
  - Can detect all burst errors less than r+1 bits
- Widely used in practice (Ethernet, FDDI, ATM)
CRC: Rooted in Polynomials

- We’re actually doing polynomial arithmetic
  - Each bit is actually a coefficient of corresponding term in a $k^{th}$-degree polynomial

$$1101 = (1 \times X^3) + (1 \times X^2) + (0 \times X^1) + (1 \times X^0)$$

- 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

\[ x^3 + x^2 + 1 \]
\[ x^7 + x^4 + x^3 + x \]

= 1101
= 10011010

Generator \((k+1)\) bits
Message

Message plus \(k\) zeros \((^2k)\)

\(k + 1\) bit check sequence \(g\), equivalent to a degree-\(k\) polynomial

Result:
Transmit message followed by remainder:

10011010101
CRC Example Decoding

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

Generator
Received Message

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

\[ \text{Remainder } D \mod g \]
\[ = 1101 \]
\[ = 1101 \]
\[ = 0 \]

Result:
CRC test is passed
### CRC Example Failure

#### Will this be caught?

<table>
<thead>
<tr>
<th>Option</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| A. Yes, because 2 bits is less than four | x + 1 bit check sequence g, equivalent to a degree-k polynomial  
| B. Yes, because CRC always works |                                             |
| C. No, CRC can’t catch even numbers of bit errors |                                 |
| D. No, the errors are in the message, not the remainder |                    |

#### Result:

CRC test failed
## Common Generators

<table>
<thead>
<tr>
<th>Generator</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>$x^{32} + x^{26} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$</td>
</tr>
</tbody>
</table>
Error Handling Summary

- 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
Picking up the Pieces

- Link layer is lossy
  - We deliberately throw away corrupt frames!
  - Infrequent bit errors still lead to occasional frame errors
    - 10,000+ bits in each frame

- Things get even harder if we consider multiple links
  - In a few lectures, we’ll start sending frames on long trips
  - Each intermediate stop might lose, corrupt, *reorder*, etc.
  - Regardless of cause, we’ll call loss events **drops**

- We want to provide reliable, in-order delivery
  - Can—and will—do this at multiple layers
For Next Time

- Homework due next Friday: 10/11
- Read 2.5 in P&D
- Have a great weekend!