Lecture 4: Reliable Transmission

Project 1a due two weeks from today
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
  
  \[ \begin{array}{r}
  \text{1101} \\
  \text{110} \\
  \underline{\text{0000}} \\
  \text{11010} \\
  \underline{\text{110100}} \\
  \text{101110}
  \end{array} \]

- Division
  
  \[ \begin{array}{r}
  \text{1101} \\
  \underline{\text{110}} \\
  \underline{\text{111}} \\
  \underline{\text{110}} \\
  \underline{\text{011}} \\
  \underline{\text{000}} \\
  \text{110}
  \end{array} \]

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011110100101011110101000111011010101101101111011110110

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

010100111010010101110100011101011010011011111011110110

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

010100111101001011110100011101011010011011111011110110

Data ↓ 0

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

0101001111010010101111010001110101101001101111011110110

Data 01

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011101001010111010001110101101001101111101111011011110110110

Data 010

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

\[0101001110100101011101011010011011111011110110\]

Data \(\downarrow\) \[0101\]

\[0011\ldots\]
Checksum Example

01010011110100101101101000111010100100110111110110

Data  
01010011

1101...
Checksum Example

010100111101001010111010001110101110100110111101110110

\[ 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad + \quad 1010 \ldots \]

Data

\[ 01010011 \]

Parity Byte

\[ 1 \]
Checksum Example

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

0101001111010010101111010001110101101001101111101110110
1011...

Data
Parity Byte

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

0101001111010010101111010001111011011011101111101110110
0101001111010010101111010001111011011011101111101110110

Data
Parity Byte

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

010100111101001010111101000111010110100110111110111110110

1 1 1 0 1 1 0

Data

01010011 11010010 10111101 00011101 01101001 10111110

Parity Byte

11101110

CSE 123 – Lecture 4: Reliable Transmission
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)

$D: \text{data bits to be sent} \quad | \quad R: \text{CRC bits}$

$D \times 2^r \quad \text{XOR} \quad R$

Mathematical formula
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

\[ 1101 \text{ is } (1 \times X^3) + (1 \times X^2) + (0 \times X^1) + (1 \times X^0) \]

- Why do we care?
  - Can use the properties of finite fields to analyze effectiveness
  - E.g., 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  
Generator (\(k+1\) bits)

= 10011010  
Message

1101  
\(k+1\) bit check sequence \(g\), equivalent to a degree-\(k\) polynomial

10011010000  
Message plus \(k\) zeros (\(2^k\))

1001  
1101

1000  
1101

1011  
1101

1100  
1101

1000  
1101

Remainder \(D \mod g\)

101

Result:

Transmit message followed by remainder:

10011010101
CRC Example Decoding

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

Result:
CRC test is passed
CRC Example Failure

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

\[ = 1101 \]
\[ = 10010110101 \]

Generator
Received Message

Will this be caught?
A. Yes, because 2 bits is less than four
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

Will this be caught?
\[ k + 1 \text{ bit check sequence } g, \approx \text{ equivalent to a degree-}k \text{ polynomial} \]

Remainder
\[ D \mod g \]
### Common Generators

<table>
<thead>
<tr>
<th>Generator</th>
<th>Generator 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 harrier 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
Moving up the Stack

CSE 123 – Lecture 4: Reliable Transmission
Reliable Transmission

- The data networking version of the same problem
  - How do we reliably send a message when packets can be lost/corrupted in the network?

- Two options
  - Detect a loss/corruption and retransmit
  - Send data redundantly to tolerate loss/corruption
Simple Idea: ARQ

- Receiver sends **acknowledgments** (ACKs)
  - Sender “times out” and retransmits if it doesn’t receive them
- Basic approach is generically referred to as **Automatic Repeat Request** (ARQ)
Not So Fast…

- Loss can occur on ACK channel as well
  - Sender cannot distinguish data loss from ACK loss
  - Sender will retransmit the data frame
- ACK loss—or early timeout—results in duplication
  - The receiver thinks the retransmission is new data
For Next Time

- Project 1a due two weeks from today: 1/25
- Read 2.5 in P&D
- Discussion starts in 10 minutes!