Lecture 4: Reliable Transmission

CSE 123: Computer Networks
Alex C. Snoeren

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
  - 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 complement checksum
IP Checksum Example

- 1’s complement of sum of *words* (not bytes)
  - Final 1’s complement 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
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
  
  $\begin{array}{c}
  1101 \\
  \hline
  110 \quad 101110 \\
  \hline
  110 \\
  111 \\
  110 \\
  011 \\
  000 \\
  \hline
  110
  \end{array}$
Checksum Example

01010011110100101011110101110100111111101110110

Data

Parity Byte

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011110100101011110100011101011010011011111011101110110

01010011110100101011110100011101011010011011111011101110110

0101...
Checksum Example

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

Data ↓ 0

010100111101001010111101000111010110100110111110111101101110110

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

01010011110100101011110100011101011010011011111011110110

0 0 0 0 0 0 1

Data ↓ 01

0100...
Checksum Example

010100110100101011101000110101101001101111011101110110

Data

010

+ 1001…
Checksum Example

Data 0101

01010011110100101011110100011101011010011111011110110

+ 0011
Checksum Example

010100111101001010111101000111010110100111011111011110110

Data 01010011

010100011

1101…
Checksum Example

01010011110100101011110100011101011010011011111011110110

1 0 1 0 0 1 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

Data 01010011
Parity Byte 1

1010...
Checksum Example

Data: 01010011
Parity Byte: 10

Calculating the checksum for the data 01010011:

01010011 + 10 = 01010011110100101011110100011101011010011011111011110110

The parity byte is calculated to ensure the checksum is correct.
Checksum Example

0101001111010010101111010001110101101001101111011101101110

1 0 0 0 0 0 0 1

+ 1011...

Data
Parity Byte

01010011
11010010
10000001

CSE 123 – Lecture 4: Reliable Transmission
Checksum Example

0101001111010010101111010001110101101001101111110111101101111…

0 0 0 0 0 1 0 + 0111…

Data

Parity Byte

01010011
11010010
1

0
Checksum Example

01010011101101010111101100011101011010011011110111101110110

1 1 1 1 0 1 1 0 +

01010011
11010010
10111101
00011101
01101001
10111110
11110110

Parity Byte

Data

CSE 123 – Lecture 4: Reliable Transmission
Checkssums 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 \( 2^k D \) by \( g \) to get remainder, \( r \)
  - Remainder is called *frame check sequence*
- Send \( 2^k D - r \) (i.e., \( 2^k D \) XOR \( r \))
  - Note \( 2^k D \) is just \( D \) shifted left \( k \) bits
  - Remainder must be at most \( k \) bits
- Receiver checks that \( (2^k D - r)/g = 0 \)
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,R> exactly divisible by G (modulo 2)
  - Receiver knows G, divides <D,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)

\[
D \times 2^r \text{ XOR } R
\]

\(D: \text{data bits to be sent} \quad R: \text{CRC bits}\)
We’re actually doing polynomial arithmetic
- Each bit is actually a coefficient of corresponding term in a $k^{th}$-degree polynomial

1101 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

\[ x^3 + x^2 + 1 = 1101 \]
\[ x^7 + x^4 + x^3 + x = 10011010 \]

Generator

Message

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

\[ 1101 \]

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

Result:

Transmit message followed by remainder:

\[ 10011010101 \]
CRC Example Decoding

\[
x^3 + x^2 + 1 = 1101 \quad \text{Generator}
\]
\[
x^{10} + x^7 + x^6 + x^4 + x^2 + 1 = 10011010101 \quad \text{Received Message}
\]

\[D \mod g\]

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

\[1101\]

\[10011010101 \quad \text{Received message, no errors}\]

\[1101\]

Result:

CRC test is passed
CRC Example Failure

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

\[ = 1101 \quad \text{Generator} \]
\[ = 10010110101 \quad \text{Received Message} \]

\[
\begin{array}{c}
1101 \\
\text{k + 1 bit check sequence } g, \newline \text{equivalent to a degree-k polynomial}
\end{array}
\]

\[
\begin{array}{c}
10010110101 \\
\text{Received message}
\end{array}
\]

\[
\begin{array}{c}
1000 \\
1101
\text{Two bit errors}
\end{array}
\]

\[
\begin{array}{c}
1011 \\
1101
\end{array}
\]

\[
\begin{array}{c}
1101 \\
1101
\end{array}
\]

\[
\begin{array}{c}
0101
\end{array}
\]

Result:

\text{CRC test failed}

\[ D \mod g \]
## 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>$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x^{1} + 1$</td>
</tr>
</tbody>
</table>
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

Application Layer

Transport Layer

Network Layer

Link Layer

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

CSE 123 – Lecture 4: Reliable Transmission
For Next Time

- Homework due next Friday: 10/12
- Read 2.5 in P&D
- Have a great weekend!