Lecture 4: Reliable Transmission

CSE 123: Computer Networks

HW 1 out 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
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: wireless communication
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?
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
    So can only detect how many bit errors (poll)?

- 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 16bit 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);
}
```

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

Data:
- 01010011110100101011110100011101011010011011111011110110

Parity Byte:
- 01010011
- 11010010
- 10111101
- 00011101
- 01101001
- 10111110

Parity Byte:
- 11110110
Checksum Example

```
01010011110100101011110100011101011010011011111011110110
```

Data

Parity Byte

CSE 123 – Lecture 4: Error Handling
Checksum Example

01010011110100101011110100011101011010011011111011110110
+ 1010...

Data 0
Checksum Example

Data \[\uparrow \quad 01\]

01010011110100101011110100011101011010011011111011110110 + 0100…
Checksum Example

0101001111010010101111010000111010110100010110111101111011011110110

Data ▲ 010
Checksum Example

01010011110100101011110100011101011010010011011111011110110

Data 0101

0011...
Checksum Example

01010011110100101011110100011101011010010011011111011110110

Data 0101011

+ 1101...
Checksum Example

0101001110100101011101000111010110100110111101110110

Data

01010011
1

Parity Byte

1

+ 1010...
Checksum Example

010100111101001010111101000111010110100110110111101111011010

0101 0011
Data

01
Parity Byte

10

0100...
Checksum Example

Data: 01010011
Parity Byte: 11010010
Checksum: 1011...

CSE 123 – Lecture 4: Error Handling
Checksum Example

Prior parity result

Data

Parity Byte

CSE 123 – Lecture 4: Error Handling
Checksum Example

0101000111101001010111101000111010110100110111110111101100011111110111101110

Parity Byte

Data

Parity Byte

CSE 123 – Lecture 4: Error Handling
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
  - 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
  
  \[
  \begin{array}{c}
  1101 \\
  \quad 110 \\
  \hline
  0000 \\
  11010 \\
  110100 \\
  \hline
  101110
  \end{array}
  \]

- Division
  
  \[
  \begin{array}{c}
  1101 \\
  \quad 110 \\
  \hline
  101110 \\
  110 \\
  111 \\
  110 \\
  011 \\
  000 \\
  \hline
  110
  \end{array}
  \]
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 $2^kD - r$ (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
  - $<D, R>$ 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)

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

$D$: data bits to be sent
$R$: CRC bits

bit pattern

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

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

- 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

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

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

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

Generator

= 10011010101

Received Message

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

\[ 10011010101 \]

Received message, no errors

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

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

1101

10010110101

Received message

Two bit errors

Remainder

\( D \mod g \)

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>$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$</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

host

host

CSE 123 – Lecture 4: Reliable Transmission
Reliable Transmission

- The **packet**-based version of the same problem
  - How do we reliably send a message when **packets** (not just bits) 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
Sequence Numbers

- Sequence numbers solve this problem
  - Receiver can simply ignore duplicate data
  - But must still send an ACK! (Why?)
- Simplest ARQ: **Stop-and-wait**
  - Only one outstanding frame at a time
Stop-and-Wait Performance

- Lousy performance if xmit 1 pkt << prop. delay
  - How bad?

- Want to utilize all available bandwidth
  - Need to keep more data “in flight”
  - How much? Called the bandwidth-delay product

- Also limited by quality of timeout (how long?)
Pipelined Transmission

- Keep multiple packets “in flight”
  - Allows sender to make efficient use of the link
  - Sequence numbers ensure receiver can distinguish frames

- Sender buffers outstanding un-acked packets
  - Receiver ACKs the highest *consecutive* frame received
  - ACKs are **cumulative** (covers current frame and all previous)
For Next Time

- Homework 1 is out Friday due in one week