Lecture 6: Reliable Transmission

CSE 123: Computer Networks
Alex C. Snoeren

HW 2 out Wednesday
Lecture 6 Overview

- Cyclic Remainder Check (CRC)
- Automatic Repeat Request (ARQ)
  - Acknowledgements (ACKs) and timeouts
- Stop-and-Wait
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
  
  \[
  \begin{array}{c}
  1101 \\
  110 \\
  \hline
  101110 \\
  110 \\
  \hline
  111 \\
  110 \\
  \hline
  011 \\
  000 \\
  \hline
  110
  \end{array}
  \]
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 $2^kD - r$ (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 $(2^kD - r)/g = 0$
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
  - Says any generator with two terms catches single bit errors
### CRC Example Encoding

Consider the following polynomials:

- \( x^3 + x^2 + 1 \)
- \( x^7 + x^4 + x^3 + x \)

We are given a message of 10011010 and a generator polynomial of 1101. The process of encoding involves dividing the message by the generator polynomial and appending the remainder to the original message to form a new message that can be transmitted reliably.

1. **Message:** 10011010
2. **Generator:** 1101
3. **Message plus k zeros:** 10011010000
4. **Division:**
   - Initial division: 1001 | 1011000  (Result: 1011000)
   - Continue dividing until the remainder is 0.

The final result is:

- **Result:** Transmit message followed by remainder: 10011010101

**Explanation:**
- **k + 1 bit check sequence:** This is the remainder of the division process, which is appended to the original message to form the transmitted message.
- **Message plus k zeros:** This is the original message followed by k zeros, which are appended to ensure the final message is a multiple of the generator polynomial.

The polynomial division shows how the remainder is obtained, and this remainder is what is appended to the message for error detection.
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

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

- \( k + 1 \) bit check sequence \( g \), equivalent to a degree-\( k \) polynomial
- Received message, no errors

Result:
CRC test is passed
CRC Example Failure

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

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

\[ \begin{array}{c}
  1101 \\
  0101 \\
  \vdots \\
  0101
\end{array} \]

\[ D \mod g \]

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 threw away corrupt frames last lecture
  - 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
  - HTTP
  - TCP
  - IP
  - Ethernet
  - interface

- Transport Layer
  - HTTP
  - TCP
  - IP
  - Ethernet
  - interface

- Network Layer
  - IP
  - router

- Link Layer
  - Ethernet interface
  - SONET interface

CSE 123 – Lecture 6: Reliable Transmission
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

CSE 123 – Lecture 6: Reliable Transmission
For Next Time

- Read 2.6 in P&D