Lecture 4: CRC & Reliable Transmission

CSE 123: Computer Networks
Chris Kanich

Quiz 1: Tuesday July 5th

Lecture 4: CRC & Reliable Transmission

Overview

- CRC – toward a better EDC
- Reliable Transmission
  - How do we ensure that a message was received?
  - Automatic Repeat Request (ARQ)
  - Acknowledgements (ACKs) and timeouts
  - Stop-and-Wait
  - Sliding Window
  - Forward Error Correction

Checksum review

- Sum up all data in frame. Check when receiving
  - 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 compliment checksum
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
  
  \[
  \begin{array}{c}
  1101 \\
  110 \\
  \hline
  0000 \\
  11010 \\
  110100 \\
  101110
  \end{array}
  \]

- Division
  
  \[
  \begin{array}{c}
  1101 \\
  110 \\
  \hline
  110110 \\
  110 \\
  111 \\
  110 \\
  011 \\
  000 \\
  110
  \end{array}
  \]

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 \( 2^k D \) by \( g \) to get remainder, \( r \)
  - Remainder is called frame check sequence

- Send \( 2^k D + r \)
  - Note \( 2^k D \) is just \( D \) shifted left \( k \) bits
  - Remainder must be at most \( k \) bits

- Receiver checks that \( (2^k D + r)/g = 0 \)
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 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

- Generator
  
  Message

- Message plus $k$ zeros ($2^k$)

Result:
Transmit message followed by remainder:

10011010101

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**

\[
x^9 + x^8 + 1 = 1101 \\
x^{10} + x^9 + x^4 + x^2 + 1 = 10011010101
\]

\[\text{Generator} \quad 10011010101 \]

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

Result:

**CRC test is passed**

---

**CRC Example Failure**

\[
x^9 + x^8 + 1 = 1101 \\
x^{10} + x^9 + x^4 + x^2 + 1 = 10011010101
\]

\[\text{Generator} \quad 10011010101 \]

\[\text{Received message} \]

\[\text{Two bit errors} \]

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^7 + x^1 + 1 )</td>
</tr>
<tr>
<td>CRC-10</td>
<td>( x^{10} + x^9 + x^8 + x^7 + 1 )</td>
</tr>
<tr>
<td>CRC-12</td>
<td>( x^{12} + x^9 + x^8 + x^4 + x^1 + 1 )</td>
</tr>
<tr>
<td>CRC-16</td>
<td>( x^{16} + x^8 + x^3 + 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^{30} + x^{27} + x^{26} + x^{25} + x^{24} + x^{23} + x^{22} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^{9} + x^{8} + x^{7} + x^{6} + x^{5} + x^{4} + x^{3} + x^{2} + x^{1} + 1 )</td>
</tr>
</tbody>
</table>
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

Moving up the Stack

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? Remember 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
- Duplicate acknowledgements signal loss
  - ACK the highest consecutive frame received
  - Ignore (for now) non-sequential frames

Go-Back-N

- Retransmit from point of loss upon duplicate ACK
  - Packets between loss event and retransmission are ignored
  - Also "go-back-N" if a timeout event occurs
- ACKs are cumulative
  - Acknowledge current frame and all previous ones

Send Window

- Bound on number of outstanding packets
  - Window "opens" upon receipt of new ACK
  - Window resets entirely upon a timeout
- Limits amount of waste
  - Still lots of duplicates
  - We can do better with selective retransmission
Sliding Window

- Single mechanism that supports:
  - Multiple outstanding packets
  - Reliable delivery
  - In-order delivery
  - Flow control

- At the core of all modern ARQ protocols

- Go-Back-N is a special case
  - Receive window size of one

---

Sliding Window – Sender

- Window bounds outstanding unACKed data
  - Implies need for buffering at sender
  - “Last” ACK applies to in-order data

- What to do on a timeout?
  - Go-Back-N: send all unacknowledged data on timeout
  - Selective Repeat: timer per packet, resend as needed

---

Sliding Window – Receiver

- Receiver buffers too:
  - data may arrive out-of-order
  - or faster than can be consumed—flow control

- Receiver ACK choices:
  - Cumulative, Selective (exempt missing frames), Negative
Deciding When to Retransmit

- How do you know when a packet has been lost?
  - Ultimately sender uses timers to decide when to retransmit

- But how long should the timer be?
  - Too long: inefficient (large delays, poor use of bandwidth)
  - Too short: may retransmit unnecessarily (causing extra traffic)

- Right timer is based on the **round-trip time (RTT)**
  - Which can vary greatly for reasons well see later

Can we shortcut the timeout?

- Timeout is long in practice

- If packets are usually in order then out-of-order ACKs imply that a packet was lost
  - Negative ACK
  - Receiver requests missing packet
  - Fast retransmit
    - When sender receives multiple duplicate acknowledgements resends missing packet

Fast retransmit

- Don’t bother waiting
  - Receipt of duplicate acknowledgement (dupACK) indicates loss
  - Retransmit immediately

- Used in TCP
  - Need to be careful if frames can be reordered
Is ARQ the Only Way?

- No. We could use redundancy
  - Send additional data to compensate for lost packets

- Why not use retransmission?
  - Broadcast media with lots of receivers
    - If each one ACK/NAK then hard to scale
    - Lots of messages
    - Lots of state
  - Heterogeneous receivers
    - E.g., variable quality wireless reception
  - Highly lossy or very long delay channels (e.g., satellite)

Forward Error Correction

- Use erasure codes to redundantly encode $k$ data frames into $m > k$ encoded frames
  - Why do it at the frame level?
    - E.g., Reed Solomon Codes, Tornado codes

- Multicast/broadcast encoded frames speculatively

- A receiver can reconstruct message from any $k$ frames in the set of $m$ encoded frames

A “Digital Fountain”

File

User 1

User 2

Transmission

- 0 hours
- 1 hour
- 2 hours
- 3 hours
- 4 hours
- 5 hours
For Next Time

- Read 2.6 in P&D – Media Access
  - Now that we can reliably transmit from point A to point B, how do we share the transmission medium nicely?
- Have a great holiday weekend!