Lecture 5.2: Error Detection & Correction
Link Layer: Five Common Problems

• Basic problem: you can’t just send IP datagrams over the link!
• We first consider how to encode bits into the signal at the source and recover bits at the receiving node
• Once it is possible to transmit bits, we need to figure out how to package these bits into FRAME
• Assume each node is able to recognize the collections of bits making up a frame, the third problem is to determine if those bits are in error: Error Detection and Correction
• If frames arriving at destination contain errors, how to recover from such losses: ARQ
• Final problem related to multiple-access link: how mediate access to a shared link so that all nodes have a chance to transmit: We focus on Ethernet
Errors in Link

• Types of errors in frame
  • Isolated errors: Bit errors that do not affect other bits
  • Burst errors: A cluster of bits in which a number of errors occur

• Errors caused by: signal attenuation, noise, electrical interference etc.

• Burst errors increase with data rate
  • 1μs of impulse noise or fading effect will affect
    At most 2 bits when data rate is 1Mbps
    At most 101 bits when data rate is 100Mbps
How Often Isolated Errors Occur?

• BER = 10^{-7}, and a file is 10^4 long, then the probability of a **single error** is 10^4 \times 10^{-7} = 10^{-3}

• Probability of **exactly two errors**. For two bits $i, j$ the probability of error is 10^{-7} \times 10^{-7}

• But how many possible cases of two errors? What is the probability of two errors then?
How to Deal with Errors?

• Receiver must be aware that an error occurred in a frame
  • Need to have an ERROR DETECTION mechanism

• Receiver needs to receive the correct frame

• Two possible strategies to CORRECT ERRORS:
  • Add information redundancy to correct errors (error correcting codes)
  • Ask sender to re-send frame (retransmission strategies): Usually combined with Error Detection
• Basic Idea of Error Detection
  – To add redundant information to a frame that can be used to determine if errors have been introduced
  – Imagine (Extreme Case)
    • Transmitting two complete copies of data
      – Identical → No error
      – Differ → Error
      – Poor Scheme ???
        » n bit message, n bit redundant information
    • In general, we can provide strong error detection technique
      – k redundant bits, n bits message, k << n
      – In Ethernet, a frame carrying up to 12,000 bits of data requires only extra 32 bits
Error Detection

- Extra bits are redundant
  - They add *no new information* to the message
  - *Derived from the original message* using some algorithm
  - Both the sender and receiver know the algorithm

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>r</td>
<td>r</td>
</tr>
</tbody>
</table>

Receiver computes $r$ using $m$

If they match, no error
Error Detection – Single Parity Check

• Single parity checks
  • Append a single parity bit at end of frame. *Parity is 1 if # of ones is odd, and zero otherwise*
    
    • Example: 0 1 1 0 1 0 1 0 1 1 0 0 ← parity
  
  • Single parity check can detect any odd # of errors
  
  • Cannot tell where the error took place or how many occurred
  
  • Not useful for burst errors
Error Detection – Two-Dimensional Parity

• It is based on “simple” (one-dimensional) parity: adding one extra bit to a 7-bit code to balance the number of 1s in the byte. For example,
  – Odd parity sets the eighth bit to 1 if needed to give an odd number of 1s in the byte, and
  – Even parity sets the eighth bit to 1 if needed to give an even number of 1s in the byte

• Two-dimensional parity does a similar calculation for each bit position across each of the bytes contained in the frame
• This results in an extra parity byte for the entire frame, in addition to a parity bit for each byte
Two-Dimensional Parity: Example
How Good is Dimensional Parity?

- Error Detection?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
## How Good is Dimensional Parity?

### Error Correction?

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This row: 1 error

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This column: 1 error

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This row: 1 error
**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment.

Note: Not used at link level. In fact, at Transport Layer, e.g, *UDP*!!

**Sender:**
- treat segment contents as sequence of 16-bit integers
- Addition (*1’s complement sum*) of all segment contents
- Checksum: Previous result is complemented
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ....
1’s Complement Sum

What is 1’s complement addition?

1. Sum the numbers normally
2. The carry of the sum is whatever bits are to the left of the rightmost 16 bits.
3. If carry=0 then the normal sum is the 1’s complement sum.
   If carry!=0 then add the carry back into normal sum to get 1’s complement sum.

We will see some examples next!!
1’s Complement Sum

1 0 0 0 1 0 0 1   Word 1
1 0 1 1 1 0 0 0   Word 2

-------------------------------------
1 0 1 0 0 0 0 0 1   Normal Sum (note carry to remove)

1     Remove carry and add it back

-------------------------------------
0 1 0 0 0 0 1 0   1’s complement sum

0 0 1 0 1 0 0 1   Word 1
1 0 1 1 1 0 0 0   Word 2

-------------------------------------
1 1 1 0 0 0 0 1   1’s complement sum
Internet Checksum Example

• Note: when adding numbers, a carryout from the most significant bit needs to be added to the result: 1’s complement sum

• Example: add two 16-bit integers

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
\]

wraparound

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]

1’s compliment sum
checksum
Checksum: More than 2 Words/Sequences?

10001001  Input Word 1
11110000  Input Word 2
00111100  Input Word 3
10100001  Input Word 4

10001001  W1
11110000  W2

101111001  Normal sum
1  Carry

01111010  1’s Comp sum W1, W2
00111100  W3

10110110  1’s Comp sum W1, W2, W3
10100001  W4

101011001  Normal sum
1  Carry

01011010  1’s Comp sum W1, W2, W3, W4
10100101  complement

= checksum of W1-W4
• Usually, the word *checksum* is often used to *imprecisely* to mean *any form of error-detecting code*

• But as we just learnt *Internet checksum*: Just *use addition*

• To avoid confusion
  • When we talk about *checksum*: Only apply to codes using addition
  • Error-detecting codes: General class
Cyclic Redundancy Check (CRC)

- Add $r$ redundant bits on a $n$-bit message
  - Design goal $k \ll n$ so that overhead is low
  - Example: 32-bit CRC adequate for 12,000 bits (1,500) bytes

- In order to understand, let talk about Polynomial Arithmetic over Galois Field 2 first!!!!!!!

- What do you know about Polynomial Arithmetic?
Polynomial Arithmetic over GF(2)

- Represent \( n \)-bit messages as \((n-1)\) degree polynomials

- Example: 10001010 maps to \( x^7 + x^3 + x^1 \)

- The bits of the message to be transmitted become the coefficients of the polynomial: It can be either 0 or 1: Galois Field 2 (operations defined next).
Polynomial Arithmetic over GF(2)

• Basic operations:
  • Addition: logical XOR
  • Subtraction?
  • Multiplication: logical AND

• How to add two polynomials?
• How to subtract two polynomials?
• How to multiply two polynomials?
• How to divide two polynomials (using long division) to find quotient and remainder?
Polynomial Arithmetic

• Polynomial division: \( B(x)/C(x) \) - \( \text{deg}(B) \geq \text{deg}(C) \); \( C(x) \) is called the divisor

• If \( C(x) \) and \( B(x) \) are of the same degree, the remainder is obtained by subtracting \( C(x) \) from \( B(x) \)

• If the remainder is 0, we say \( B(x) \) is divisible by \( C(x) \).

• Modulo 2 arithmetic, subtraction is an XOR operation between coefficients

Example: \( B(x) = x^3 + 1 \), \( C(x) = x^3 + x^2 + 1 \)

Remainder: \( R(x) = x^2 \)

\( B(x) = 1001, \ C(x) =1101, \ R(x) = 0100 \) (XOR of \( B(x), \ C(x) \))
How CRC work?

• Let our information $M(x)$ be a frame with $n$ bits.
• Let $C(x)$ be a generator polynomial having less than $n$ bits say equal to $k+1$, known at both sender and receiver (degree of $k$).
• Based on $M(x)$ and $C(x)$, CRC will generate $P(x)$ of size $n+k$ bits: Divisible by $C(x)$. $P(x)$ will be transmitted instead of $M(x)$.
• At the receiver side, divide the received polynomial by $C(x)$.
  • If the remainder is 0, no error.
  • If not, error.
CRC Calculation

• Let $M(x)$ be a frame with $n$ bits and let the generator polynomial have less than $n$ bits say equal to $k$.
• Append $k$ zero bits to the low-order end of the frame, so it now contains $n+k$ bits and corresponds to the polynomial $x^kM(x)$. 
CRC Calculation

• Divide the bit string corresponding to $x^kM(x)$ by the bit string corresponding to $C(x)$ using modulo 2 division.

• Subtract the remainder (which is always $k$ or fewer bits) from the string corresponding to $x^kM(x)$ using modulo 2 subtraction (addition and subtraction are the same in modulo 2).

• The result is the checksummed frame to be transmitted. Call it polynomial $P(x)$. 
CRC Example

CRC Calculation using Polynomial Long Division
CRC Calculation using Polynomial Long Division
How to Select $C(x)$

- **Properties of Generator Polynomial**
  - Let $P(x)$ represent what the sender sent and $P(x) + E(x)$ is the received string. A 1 in $E(x)$ represents that in the corresponding position in $P(x)$ the message the bit is flipped.

  - We know that $P(x)/C(x)$ leaves a remainder of 0, but if $E(x)/C(x)$ leaves a remainder of 0, then either $E(x) = 0$ or $C(x)$ is factor of $E(x)$.

  - When $C(x)$ is a factor of $E(x)$, we have problem?  
    \[ \text{Errors go unnoticed.} \]

  - If there is a single bit error then $E(x) = x^i$, where $i$ determines the bit in error. If $C(x)$ contains non-zero first and last terms $x^k$ and $x^0$, it will never divide $E(x)$, so all single bit errors will be detected.
Properties of Generator Polynomial

- In general, it is possible to prove that the following types of errors can be detected by a $C(x)$ with the stated properties:
  - All single-bit errors, as long as the $x^k$ and $x^0$ terms have nonzero coefficients.
  - All double-bit errors, as long as $C(x)$ has a factor with at least three terms.
  - Any odd number of errors, as long as $C(x)$ contains the factor $(x+1)$.
  - Any “burst” error (i.e., sequence of consecutive error bits) for which the length of the burst is less than $k$ bits. (Most burst errors of larger than $k$ bits can also be detected.)
<table>
<thead>
<tr>
<th>CRC</th>
<th>$C(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-8</td>
<td>$x^8 + x^2 + x + 1$</td>
</tr>
<tr>
<td>CRC-10</td>
<td>$x^{10} + x^9 + x^5 + x^4 + x + 1$</td>
</tr>
<tr>
<td>CRC-12</td>
<td>$x^{12} + x^{11} + x^3 + x^2 + 1$</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>$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$</td>
</tr>
</tbody>
</table>

CCITT - Comité Consultatif International Téléphonique et Télégraphique

ITU (International Telecommunication Union) Telecommunication Standardization Sector (ITU-T)
• \( k \) bits CRC: \( k \) registers
• Label them from 0 to \((k-1)\), left to right
• Add XOR gate in front of register \( n \)th if \( C(x) \) contains \( x^n \)
• When all message (appended with \( k \) 0s) shifted, registers contain CRC bits
Error Correcting: Some Discussions

• Error Detecting can only detect errors
• We can perform more powerful function: Correcting Errors. How can we do it?
  • How do you back-up data at home?
  • Repetition Code
Error Correcting: Repetition Code

- Rate $1/n$, denoted as $R_n$, repetition code: Repeat each bit $(n - 1)$ times

- Encoding rule for $R_5$ code:
  - $0 \rightarrow 00000$
  - $1 \rightarrow 11111$

- Decoding rule:
  - Majority decoding rule: choose bit that occurs more frequently

- Example with $R_5$ code: We have information 10. After encoding, we have 1111100000. If 0110111000 is received (some bits are in error):
  - We first decode 01101 to 1
  - We then decode 11000 to 0
  - Decoded bits: 10.
How Good is Repetition Code?

- Without repetition code, assume the probability of error is \( f \).

- With \( R_n \) code, the probability of error is:

\[
P_b = \sum_{i=\frac{n+1}{2}}^{n} \binom{n}{i} f^i (1 - f)^{n-i}
\]

- Repetition is the simplest code: Is it a good code?
How Good is Repetition Code?

With $f = 10^{-1}$ and $R_3$ code, overall error $P_b$ is $2 \times 10^{-2}$:

\[ f = 10\% \]

How Good is Repetition Code?


- Not good if $n$ is small. If $n$ is large: Overhead burden
Current and Future Codes

• Well-known codes: *Hamming codes*; *Reed-Solomon codes*: Used in *DSL, CD, DVD, Blue-Ray Discs*
• Almost all digital comm. and storage systems: already upgraded or being upgraded to include very powerful error correcting codes:
  • *Turbo Code*, *Low-Density-Parity-Check (LDPC) Code*
• Deep space and satellite communications: *DVB-S2, DVB-RCS, Messenger to Mercury, Mars Reconnaissance Orbiter, Mars Telecomm Orbiter 2010*
• *10GBASE-T* or *IEEE 802.3an* (Ethernet)
• Wireless Communications: *WiFi, LTE, LTE-Advanced*
• Need to know more: *Take Error Control Coding course*
Error Detection or Error Correction?

• Look like Error Correction is always better?
  • Requires extra bits \textit{all the time}.
  • Re-transmission: \textit{when only needed}.
• Usually, error correction is preferred when errors are quite probable, wireless comm., or cost of transmission is too high
• Error Correction: \textit{Forward Error Correction}

• Can be used together: \textit{minor errors are corrected without retransmission}, and \textit{major errors are corrected via a request for retransmission}: this is called \textit{hybrid automatic repeat-request}