Error Control  (1)

Required reading:
Garcia  3.9.1, 3.9.2, 3.9.3

CSE 3213,  Fall 2010
Instructor: N. Vlajic
Modulo-2 Arithmetic

Modulo 2 arithmetic is performed digit by digit on binary numbers. Each digit is considered independently from its neighbours. Numbers are not carried or borrowed.

\[
\begin{array}{c}
0 \oplus 0 = 0 \\
1 \oplus 1 = 0 \\
\end{array}
\]

a. Two bits are the same, the result is 0.

\[
\begin{array}{c}
0 \oplus 1 = 1 \\
1 \oplus 0 = 1 \\
\end{array}
\]

b. Two bits are different, the result is 1.

c. Result of XORing two patterns

\[
\begin{array}{cccccc}
1 & 0 & 1 & 1 & 1 & 0 \\
\oplus & 1 & 1 & 1 & 0 & 0 \\
\hline
0 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Data Link Layer

Network layer
Gives services to

Data link layer
Packetizing
Flow control
Media access control
Addressing
Error control

Receives services from

Physical layer

LANs

WANs

Duties of data link layer
Packetizing
Addressing
Error control
Flow control
Access control
Why Error Control? – data sent from one computer to another should be transferred reliably – unfortunately, the physical link cannot guarantee that all bits, in each frame, will be transferred without errors

- error control techniques are aimed at improving the error-rate performance offered to upper layer(s), i.e. end-application

Probability of Single-Bit Error – aka bit error rate (BER):

- wireless medium: \( p_b = 10^{-3} \)
- copper-wire: \( p_b = 10^{-6} \)
- fibre optics: \( p_b = 10^{-9} \)

Approaches to Error Control

1. **Error Detection + Automatic Retrans. Request (ARQ)**
   - fewer overhead bits 😊
   - return channel required 😞
   - longer error-correction process and waste of bandwidth when errors are detected 😞

2. **Forward Error Correction (FEC)**
   - **error detection** + error correction
Error Control (cont.)

Types of Errors

(1) **Single Bit Errors**
- only one bit in a given data unit (byte, packet, etc.) gets corrupted

(2) **Burst Errors**
- two or more bits in the data unit have been corrupted
- errors do not have to occur in consecutive bits
- burst errors are typically caused by external noise (environmental noise)
- burst errors are more difficult to detect / correct
Error Control (cont.)

Key Idea of Error Control

- redundancy!!! – add enough extra information (bits) for detection / correction of errors at the destination

  - redundant bits = ‘compressed’ version of original data bits
  - error correction requires more redundant bits than error detection
  - more redundancy bits ⇒ better error control 😊 ⇒ more overhead 😞

![Diagram of Error Control Process]

- Calculation of check bits
- Channel
- Recalculation of check bits
- Comparison
- Information accepted if check bits match

Example:

\[ f(S) \]  
\[ 1 0 0 1 1 0 1 0 \]  
\[ \rightarrow 1 0 \]  
\[ f(S') \]  
\[ 1 0 0 1 1 0 1 0 1 0 \]  
\[ \rightarrow 1 0 \text{ comp} 0 0 \]
Hamming Distance

Hamming Distance
between 2 Codes

– number of differences between corresponding bits
  - can be found by applying XOR on two codewords and counting number of 1s in the result

Minimum Hamming Distance \( (d_{\text{min}}) \) in a Code

– minimum Hamming distance between all possible pairs in a set of codewords
  - \( d_{\text{min}} \) bit errors will make one codeword look like another
  - larger \( d_{\text{min}} \) – better robustness to errors

Example [ \( k=2, n=5 \) code ]

Code that adds 3 redundant bits to every 2 information bits, thus resulting in 5-bit long codewords.

<table>
<thead>
<tr>
<th>Dataword</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000</td>
</tr>
<tr>
<td>01</td>
<td>01011</td>
</tr>
<tr>
<td>10</td>
<td>10101</td>
</tr>
<tr>
<td>11</td>
<td>11110</td>
</tr>
</tbody>
</table>

Any 1-bit error will not be detected.

Any 1-bit and 2-bit errors will be detected.
Minimum Hamming Distance for Error Detection

– to guarantee detection of up to $s$ errors in all cases, the minimum Hamming distance must be

$$d_{\text{min}} = s + 1$$

Example  [ code with $d_{\text{min}}=2$ is able to detect $s=1$ bit-errors ]

<table>
<thead>
<tr>
<th>Datawords</th>
<th>Codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>011</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>
Minimum Hamming Distance for Error Correction

To guarantee correction of up to $t$ errors in all cases, the minimum Hamming distance must be

$$d_{\text{min}} = 2t + 1$$

**Example** [Hamming distance]

A code scheme has a Hamming distance $d_{\text{min}}=4$. What is the error detection and error correction capability of this scheme?

The code guarantees the detection of up to three errors ($s=3$), but it can correct only 1-bit errors!
**Error Detection: Single Parity Check**

### Error Detection Techniques

- **Parity check**
- **Checksum**
- **Cyclic redundancy check**

---

**Single Parity Check (Even Parity)**

- take *k* information bits and append a single check bit so that **overall number of 1s is even**!

- **Info Bits:** \( b_1, b_2, b_3, \ldots, b_k \)

- **Check Bit:** \( b_{k+1} = b_1 + b_2 + b_3 + \ldots + b_k \mod 2 \)

- **Codeword:** \( [b_1, b_2, b_3, \ldots, b_k, b_{k+1}] \)

---

- receiver checks if number of 1s is even
  - receiver **CAN DETECT** all single-bit errors & burst errors with odd number of corrupted bits
  - single-bit errors **CANNOT** be CORRECTED – position of corrupted bit remains unknown
  - all even-number burst errors are **undetectable** !!!
Example [encoder and decoder for single parity check code]
Example [ single parity check ]

- Information (7 bits): \[ [0, 1, 0, 1, 1, 0, 0] \]
- Parity Bit: \[ b_8 = 0 + 1 + 0 + 1 + 1 + 0 \mod 2 = 1 \]
- Codeword (8 bits): \[ [0, 1, 0, 1, 1, 0, 0, 1] \]

- If single error in bit 3: \[ [0, 1, 1, 1, 1, 0, 0, 1] \]
  - \# of 1’s = 5, odd
  - Error detected 😊 !

- If errors in bits 3 and 5: \[ [0, 1, 1, 1, 0, 0, 0, 1] \]
  - \# of 1’s = 4, even
  - Error not detected 😞 !!!

- If errors in bit 3, 5, 6: \[ [0, 1, 1, 1, 0, 1, 0, 1] \]
  - \# of 1’s = 5, odd
  - Error detected 😊 !
Example  [ single parity check code C(5,4) ]

<table>
<thead>
<tr>
<th>Datawords</th>
<th>Codewords</th>
<th>Datawords</th>
<th>Codewords</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

Single Parity Check Codes and Minimum Hamming Distance ($d_{\text{min}}$) – for ALL parity check codes, $d_{\text{min}} = 2$
Effectiveness of Single Parity Check

original codeword: \( b = [b_1 \ b_2 \ b_3 \ ... \ b_n] \)

received codeword: \( b' = [b'_1 \ b'_2 \ b'_3 \ ... \ b'_n] \)

error vector: \( e = [e_1 \ e_2 \ e_3 \ ... \ e_n] \)

\[
e_k = \begin{cases} 
1, & \text{if } b'_k \neq b_k \\ 
0, & \text{if } b'_k = b_k 
\end{cases}
\]

(1) Random Error Vector

Channel Model — there are \( 2^n \) possible error (e) vectors — all error are equally likely

- e.g. \( e=[0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0] \) and \( e=[1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1] \) are equally likely
- 50% of error vectors have an even # of 1s, 50% of error vectors have an odd # of 1s
- probability of error detection failure = 0.5
- not very realistic channel model !!!
(2) **Random Bit Error Channel Model** — bit errors occur independently of each other — \( p_b = \text{prob. of error in a single-bit transmission} \)

(2.1) **probability of single bit error** \( w(e) = 1 \) — where \( w(e) \) represents the number of 1s in \( e \)
- bit-error occurs at an arbitrary (but particular) position

\[
\begin{align*}
\text{e}_1 &= 0 & \text{e}_2 &= 0 & \text{e}_3 &= 1 & \text{e}_{n-2} &= 0 & \text{e}_{n-1} &= 0 & \text{e}_n &= 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\end{align*}
\]

\[
P(w(e) = 1) = (1 - p_b) \cdot (1 - p_b) \cdot p_b \cdot \ldots \cdot (1 - p_b) \cdot (1 - p_b) \cdot (1 - p_b)
\]

**probability of correctly transmitted bit**

\[
P(w(e) = 1) = (1 - p_b)^{n-1} \cdot p_b
\]
(2.2) **probability of two bit errors:** \( w(e)=2 \)

\[
P(w(e) = 2) = (1-p_b)^{n-2} \cdot (p_b)^2 = (1-p_b)^{n-1} \cdot p_b \cdot \frac{p_b}{1-p_b} < 1, \text{ since } p_b < 0.5
\]

\[
P(w(e) = 2) = P(w(e) = 1) \cdot \left( \frac{p_b}{1-p_b} \right) < P(w(e) = 1)
\]

(2.3) **probability of \( w(e) = k \) bit errors:** \( w(e)=k \)

\[
P(w(e) = k) = (1-p_b)^{n-k} \cdot (p_b)^k = (1-p_b)^{n-1} \cdot p_b \cdot \left( \frac{p_b}{1-p_b} \right)^{k-1} = P(w(e) = 1) \cdot (a)^{k-1}
\]

\[
P(w(e) = k) < ... < P(w(e) = 2) < P(w(e) = 1)
\]

1-bit errors are more likely 2-bit errors, and so forth!
(2.4) probability that single parity check fails?!

\[ P(\text{error detection failure}) = P(\text{error patterns with even number of 1s}) = \]
\[ = P(\text{any 2 bit error}) + P(\text{any 4 bit error}) + P(\text{any 6 bit error}) + \ldots = \]
\[ = (\text{# of 2-bit errors}) \cdot P(w(e) = 2) + \]
\[ + (\text{# of 4-bit errors}) \cdot P(w(e) = 4) + \]
\[ + (\text{# of 6-bit errors}) \cdot P(w(e) = 6) + \ldots \]

number of combinations ‘\( n \) choose \( k \)’:

\[ (\text{# of } k \text{-bit errors}) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \]

progressively smaller components …
Example  [ probability of error detection failure ]

Assume there are \( n = 32 \) bits in a codeword (packet).
Probability of error in a single bit transmission \( p_b = 10^{-3} \).
Find the probability of error-detection failure.

\[
P(error \, detection \, failure) = \binom{32}{2} p_b^2 (1-p_b)^{30} + \binom{32}{4} p_b^4 (1-p_b)^{28} + \binom{32}{6} p_b^6 (1-p_b)^{26} + \ldots
\]

\[
\binom{32}{2} p_b^2 (1-p_b)^{30} \approx \frac{32 \times 31}{2} (10^{-3})^2 = 496 \times 10^{-6}
\]

\[
\binom{32}{4} p_b^4 (1-p_b)^{28} \approx \frac{32 \times 31 \times 30 \times 29}{2 \times 3 \times 4} (10^{-3})^4 = 35960 \times 10^{-12}
\]

\[
P(error \, detection \, failure) = 496 \times 10^{-6} = 4.96 \times 10^{-4} \approx \frac{1}{2000}
\]

Approximately, 1 in every 2000 transmitted 32-bit long codewords is corrupted with an error pattern that cannot be detected with single-bit parity check.
Error Detection: 2-D Parity Check

Two Dimensional Parity Check – a block of bits is organized in a table (rows & columns)
a parity bit is calculated for each row and column

- 2-D parity check increases the likelihood of detecting burst errors
  - all 1-bit errors CAN BE DETECTED and CORRECTED
  - all 2-, 3- bit errors can be DETECTED
  - 4- and more bit errors can be detected in some cases

- **drawback**: too many check bits !!!
Example  [ effectiveness of 2-D parity check ]

a. Design of row and column parities

b. One error affects two parities

c. Two errors affect two parities

d. Three errors affect four parities

e. Four errors cannot be detected
Example  [ 2-D parity check ]

Suppose the following block of data, error-protected with 2-D parity check, is sent: 10101001 00111001 11011101 11100111 10101010.

However, the block is hit by a burst noise of length 8, and some bits are corrupted. 10100011 10001001 11011101 11100111 10101010.

Will the receiver be able to detect the burst error in the sent data?

1010100 1 1010001 1
0011100 1 1000100 1
1101110 1 1101110 1
1110011 1 1110011 1
1010101 0 1010101 0
### Signed Number Representation

http://en.wikipedia.org/wiki/Signed_number_representations

#### 8 bit signed magnitude

<table>
<thead>
<tr>
<th>Binary</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01111111</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>10000000</td>
<td>−0</td>
<td>128</td>
</tr>
<tr>
<td>10000001</td>
<td>−1</td>
<td>129</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111111</td>
<td>−127</td>
<td>255</td>
</tr>
</tbody>
</table>

#### 8 bit ones’ complement

<table>
<thead>
<tr>
<th>Binary value</th>
<th>Ones’ complement interpretation</th>
<th>Unsigned interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>+0</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01111101</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>01111110</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>01111111</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>10000000</td>
<td>−127</td>
<td>128</td>
</tr>
<tr>
<td>10000001</td>
<td>−126</td>
<td>129</td>
</tr>
<tr>
<td>10000010</td>
<td>−125</td>
<td>130</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111110</td>
<td>−1</td>
<td>254</td>
</tr>
<tr>
<td>11111111</td>
<td>−0</td>
<td>255</td>
</tr>
</tbody>
</table>
Error Detection: Internet Checksum

(Internet) Checksum – error detection method used by IP, TCP, UDP

- checksum calculation:
  - IP/TCP/UDP packet is divided into n-bit sections
  - n-bit sections are added using “1-s complement arithmetic” – the sum is also n-bits long!
  - the sum is complemented to produce checksum (complement of a number in 1-s arithmetic is the negative of the number)

- advantages:
  - relatively small packet overhead is required – n bits added regardless of packet size
  - easy / fast to implement in software

- disadvantages:
  - weak protection compared to CRC – e.g. will NOT detect misordered bytes/words
  - detects all errors involving an odd number of bits and most errors involving an even number of bits

\[ T - \overline{T} = 0 \]
### IPv4 Header Structure

- **Total length**: 16 bits
- **Identification**: 16 bits
- **Flags**: 3 bits
- **Fragmentation offset**: 13 bits
- **Time to live**: 8 bits
- **Protocol**: 8 bits
- **Header checksum**: 16 bits
- **Source IP address**
- **Destination IP address**

#### Bit Allocations

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VER</td>
<td>4 bits</td>
</tr>
<tr>
<td>HLEN</td>
<td>4 bits</td>
</tr>
<tr>
<td>Service type</td>
<td>8 bits</td>
</tr>
<tr>
<td>Total length</td>
<td>16 bits</td>
</tr>
<tr>
<td>Identification</td>
<td>16 bits</td>
</tr>
<tr>
<td>Flags</td>
<td>3 bits</td>
</tr>
<tr>
<td>Fragmentation offset</td>
<td>13 bits</td>
</tr>
<tr>
<td>Time to live</td>
<td>8 bits</td>
</tr>
<tr>
<td>Protocol</td>
<td>8 bits</td>
</tr>
<tr>
<td>Header checksum</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

### Data Region

- **Length**: 20-65536 bytes
- **Header**: 20-60 bytes

### Option

- **Size**: Variable
Sender:
- data is divided into k sections each n bits long
- all sections are added using 1-s complement to get the sum
- the sum is bit-wise complemented and becomes the checksum
- the checksum is sent with the data

Receiver:
- data is divided into k sections each n bits long
- all sections are added using 1-s complement to get the sum
- the sum is bit-wise complemented
- if the result is zero, the data is accepted, otherwise it is rejected
Example [ Internet Checksum ]
Suppose the following block of 8 bits is to be sent using a checksum of 4 bits: 1100 1010. Find the checksum of the given bit sequence.

\[
\begin{array}{c}
1100 \\
1010 \\
0000 \\
\hline
\text{sum: } 10110 \\
0110 \\
1 \\
\hline
\text{1-s complement addition: } 0111 (7)
\end{array}
\]

1-s complement addition: Perform standard binary addition. If a carry-out (>nth) bit it produced, swing that bits around and add it back into the summation.

\[
\begin{array}{c}
\text{checksum: } 1000 (-7)
\end{array}
\]

Negative binary numbers: Negative binary numbers are bit-wise complement of corresponding positive numbers.
Suppose the receiver receives the bit sequence and the checksum with no error.

\[
\begin{array}{c}
1100 \\
1010 \\
1000 \\
\end{array}
\]

\[
\text{sum: } \overline{11110}
\]

\[
\text{1-s complement addition: } 1111
\]

\[
\text{bit-wise complement: } 0000
\]

When the receiver adds the three blocks, it will get all 1s, which, after complementing, is all 0s and shows that there is no error.

If one or more bits of a segment are damaged, and the corresponding bit of opposite value in a second segment is also damaged, the sums of those columns will not change and the receiver will not detect the problem. 😞
Example  [ Internet Checksum ]

Suppose the following block of 16 bits is to be sent using a checksum of 8 bits. 10101001  00111001. The numbers are added using one’s complement:

\[
\begin{align*}
10101001 \\
00111001 \\
00000000
\end{align*}
\]

\[
\begin{align*}
\text{Sum} & \quad 11100010 \\
\text{Checksum} & \quad 00011101
\end{align*}
\]

The pattern sent is 10101001 00111001 00011101.

Now suppose the receiver receives the pattern with no error. 10101001 00111001 00011101

When the receiver adds the three blocks, it will get all 1s, which, after complementing, is all 0s and shows that there is no error.

\[
\begin{align*}
10101001 \\
00111001 \\
00011101
\end{align*}
\]

\[
\begin{align*}
\text{Sum} & \quad 11111111 \\
\text{Complement} & \quad 00000000
\end{align*}
\]

means that the pattern is OK.
Example [Internet Checksum]

Now suppose that in the previous example, there was a burst error of length 5 that affected 4 bits.

\[
\begin{align*}
10101 & 111\ 1111001\ 00011101 \\
\text{Partial Sum} & 111000101 \\
\text{Checksum} & 11000110 \\
\text{Complement} & 00111001 \quad \text{the pattern is corrupted.}
\end{align*}
\]