Link Layer: Error detection and correction
Problem: Noise may Flip Received Bits

• Link layers provides some protection
  • Detect errors with codes
  • Correct errors with codes
  • Retransmit lost frames
    Later

• Reliability concern cuts across the layers
  • E.g., TCP in the transport layer, DNS in the app layer
Problem: Noise may Flip Received Bits

<table>
<thead>
<tr>
<th>Signal</th>
<th>Slightly Noisy</th>
<th>Very Noisy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1</td>
<td>0 0 0 0 1</td>
<td>0 0 0 0 1</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

Ideas?
Approach – Add Redundancy

• Error detection codes: Add check bits to the message bits to let some errors be detected

• Error correction codes: Add more check bits to let some errors be corrected

• Key issue: Structure the code such that
  • Need few check bits to detect/correct many errors
  • Modest computation
Motivating Example

• A simple code to handle errors:
  • Send two copies! Error detected if different.

• How good is this code?
  • How many errors can it detect/correct?
  • How many errors will make it fail?
Want to Handle More Errors w/ Fewer Bits

• We’ll look at better codes (applied mathematics)
  • But, they can’t handle all errors
  • And they focus on accidental (random) errors
Using Error Codes

• Codeword consists of D data plus R check bits (=systematic block code)

<table>
<thead>
<tr>
<th>Data bits</th>
<th>Check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>$R=fn(D)$</td>
</tr>
</tbody>
</table>

• Sender:
  • Compute R check bits based on the D data bits; send the codeword of D+R bits
Using Error Codes (2)

• Receiver:
  • Receive D+R bits with unknown errors
  • Recompute R check bits based on the D data bits
  • Error detected if R doesn’t match R’

\[
\begin{array}{c|c}
\text{Data bits} & \text{Check bits} \\
\hline
D & R' \\
R' = fn(D) & \\
\end{array}
\]
Intuition for Error Codes

• For D data bits, R check bits:

  All possible D+R bits

  Correct codewords

• Randomly chosen D+R bits is unlikely to be correct
  • Low, controllable overhead
R.W. Hamming (1915-1998)

• Much early work on codes:
  • “Error Detecting and Error Correcting Codes”, BSTJ, 1950

• See also:
  • “You and Your Research”, 1986
Hamming Distance

• Distance is the number of bit flips needed to change $D_1$ to $D_2$

• Hamming distance of a coding is the minimum error distance between any pair of codewords (bit-strings) that cannot be detected
Hamming Distance (2)

• Error detection:
  • For a coding of distance $d+1$, up to $d$ errors will always be detected

• Error correction:
  • For a coding of distance $2d+1$, up to $d$ errors can always be corrected by mapping to the closest valid codeword
Simple Error Detection – Parity Bit

• Take D data bits, add 1 check bit that is the sum of the D bits
  • Sum is modulo 2 or XOR
Parity Bit (2)

• How well does parity work?
  • What is the distance of the code?
  • How many errors will it detect/correct?

• What about larger errors?
Checksums

• Idea: sum up data in N-bit words
  • Widely used in, e.g., TCP/IP/UDP

| 1500 bytes | 16 bits |

• Stronger protection than parity
Internet Checksum

• Sum is defined in 1s complement arithmetic (must add back carries)
  • And it’s the negative sum

• “The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words ...” – RFC 791
Internet Checksum (2)

Sending:
1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

0001
f204
f4f5
f6f7
Internet Checksum (3)

Sending:
1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum
Internet Checksum (4)

Receiving:
1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0
Internet Checksum (5)

Receiving:
1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0
Internet Checksum (6)

• How well does the checksum work?
  • What is the distance of the code?
  • How many errors will it detect/correct?
Why Error Correction is Hard

• If we had reliable check bits we could use them to narrow down the position of the error
  • Then correction would be easy
• But error could be in the check bits as well as the data bits!
  • Data might even be correct
Intuition for Error Correcting Code

- Suppose we construct a code with a Hamming distance of at least 3
  - Need $\geq 3$ bit errors to change one valid codeword into another
  - Single bit errors will be closest to a unique valid codeword
- If we assume errors are only 1 bit, we can correct mapping an error to the closest valid codeword
  - Works for $d$ errors if $HD \geq 2d + 1$
Intuition (2)

• Visualization of code:
Intuition (3)

- Visualization of code:

  - Single bit error from A
  - Three bit errors to get to B
  - Valid codeword
  - Error codeword
Hamming Code

• Gives a method for constructing a code with a distance of 3
  • Uses $n = 2^k - k - 1$, e.g., $n=4, k=3$
  • Put check bits in positions $p$ that are powers of 2, starting with position 1
  • Check bit in position $p$ is parity of positions whose $p$-th LSBit is same as $p$’s

• Plus an easy way to correct [soon]
Hamming Code (2)

• Example: data=0101, 3 check bits
  • 7 bit code, check bit positions 1, 2, 4
  • Check 1 covers positions 1, 3, 5, 7 (LSB is 1)
  • Check 2 covers positions 2, 3, 6, 7 (2nd LSB is 1)
  • Check 4 covers positions 4, 5, 6, 7 (3rd LSB is 1)
Hamming Code (3)

• Example: data=0101, 3 check bits
  • 7 bit code, check bit positions 1, 2, 4
  • Check 1 covers positions 1, 3, 5, 7
  • Check 2 covers positions 2, 3, 6, 7
  • Check 4 covers positions 4, 5, 6, 7

\[
\begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]

\[ p_1 = 0+1+1 = 0, \quad p_2 = 0+0+1 = 1, \quad p_4 = 1+0+1 = 0 \]
Hamming Code (4)

• To decode:
  • Recompute check bits (with parity sum including the check bit)
  • Arrange as a binary number
  • Value (syndrome) tells error position
  • Value of zero means no error
  • Otherwise, flip bit to correct
Hamming Code (5)

• Example, continued

\[
\begin{array}{ccccccc}
\text{p}_1 &=& 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\text{p}_2 &=& 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]

\[
\text{Syndrome } = \\
\text{Data } =
\]

\[
\text{p}_1 = \\
\text{p}_2 = \\
\text{p}_4 =
\]
Hamming Code (6)

• Example, continued

\[ \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\end{array} \]

\[ p_1 = 0 + 0 + 1 + 1 = 0, \quad p_2 = 1 + 0 + 0 + 1 = 0, \]
\[ p_4 = 0 + 1 + 0 + 1 = 0 \]

Syndrome = 000, no error
Data = 0 1 0 1
Hamming Code (7)

- Example, continued

\[ \begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array} \]

\[ p_1 = \quad \quad p_2 = \quad \quad p_4 = \]

Syndrome =
Data =
Hamming Code (8)

• Example, continued

\[ \rightarrow 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \]

\[ \begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ p_1 = 0 + 0 + 1 + 1 = 0, \quad p_2 = 1 + 0 + 1 + 1 = 1, \]
\[ p_4 = 0 + 1 + 1 + 1 = 1 \]

Syndrome = 1 1 0, flip position 6
Data = 0 1 0 1 (correct after flip!)
Hamming Code (3)

- Example: bad message 0100111
  - 7 bit code, check bit positions 1, 2, 4
  - Check 1 covers positions 1, 3, 5, 7
  - Check 2 covers positions 2, 3, 6, 7
  - Check 4 covers positions 4, 5, 6, 7

\[
\begin{align*}
0 & \quad 1 & \quad 0 & \quad 0 & \quad 1 & \quad 1 & \quad 1 \\
1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 \\
\end{align*}
\]

- \( p_1 = 0 + 0 + 1 + 1 = 0 \)
- \( p_2 = 1 + 0 + 1 + 1 = 1 \)
- \( p_4 = 0 + 1 + 1 + 1 = 1 \)
Hamming Code (3)

- Example: bad message 0100111
  - 7 bit code, check bit positions 1, 2, 4
  - Check 1 covers positions 1, 3, 5, 7
  - Check 2 covers positions 2, 3, 6, 7
  - Check 4 covers positions 4, 5, 6, 7

\[ \begin{array}{ccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ p_1 = 0 + 0 + 1 + 1 = 0, \quad p_2 = 1 + 0 + 1 + 1 = 1, \quad p_4 = 0 + 1 + 1 + 1 = 1 \]
Detection vs. Correction

• Which is better will depend on the pattern of errors. For example:
  • 1000 bit messages with a bit error rate (BER) of 1 in 10000

• Which has less overhead?
Detection vs. Correction

• Which is better will depend on the pattern of errors. For example:
  • 1000 bit messages with a **bit error rate (BER)** of 1 in 10000

• Which has less overhead?
  • It still depends! We need to know more about the errors
Detection vs. Correction (2)

Assume bit errors are random
• Messages have 0 or maybe 1 error (1/10 of the time)

Error correction:
• Need ~10 check bits per message
• Overhead:
  • 10 bits per message

Error detection:
• Need ~1 check bits per message plus 1000 bit retransmission
• Overhead:
  • 101 bits per message
Detection vs. Correction (3)

Assume errors come in bursts of 100
  • Only 1 or 2 messages in 1000 have significant (multi-bit) errors

Error correction:
  • Need >>100 check bits per message
  • Overhead:
    • >> 100 bpm

Error detection:
  • Need 32 check bits per message plus 1000 bit resend 2/1000 of the time
  • Overhead:
    • 34 bits per message
Detection vs. Correction (4)

• Error correction:
  • Needed when errors are expected
  • Or when no time for retransmission

• Error detection:
  • More efficient when errors are not expected
  • And when errors are large when they do occur
Error Correction in Practice

• Heavily used in physical layer
  • LDPC is the future, used for demanding links like 802.11, DVB, WiMAX, power-line, ...
  • Convolutional codes widely used in practice

• Error detection (w/ retransmission) is used in the link layer and above for residual errors

• Correction also used in the application layer
  • Called Forward Error Correction (FEC)
  • Normally with an erasure error model
  • E.g., Reed-Solomon (CDs, DVDs, etc.)