Data Link Layer: Functionality

- The data link layer must:
  - Detect errors (using redundancy bits)
  - Request retransmission if data is lost (using automatic repeat request -- ARQ)
  - Perform framing (detect packet start and end)
  - Support initialization and disconnection operations

Data Link Layer

- These functions require that extra bits be added to the packet to be transmitted
  - Header bits are added to the front of each packet
  - Trailer bits are added to the rear of each packet
  - The header, packet from upper layer (service data unit), and trailer form a frame

Frame Format

- The packet from the upper layer is the service data unit (SDU)
- The frame (header, network layer packet, and trailer) is the protocol data unit (PDU)
- Note that the data link layer does not care what is in the network layer packet and the physical layer does not care what is in the frame generated by the data link layer

Error Detection

- Two types
  - Error Detection Codes (e.g. CRC, Parity, Checksums)
  - Error Correction Codes (e.g. Hamming, Reed Solomon)
- Basic Idea
  - All bit combinations in a packet are valid
  - Add redundant information to determine if errors have been introduced
    - Why redundant?

Error Detection Codes

- Naïve scheme
  - Send a duplicate copy of the message
- Problems
  - Takes up too much space
  - Poor performance.
    - Can’t even detect 2 bit errors
Single Parity Checks
- Technically used for 1 bit error detection. Can also detect any odd number of bit errors.
- Involves adding an extra “parity” bit to the bit string
- Two varieties:
  - Even Parity
  - Odd Parity
- Basic Idea:
  - For even parity, make the total number of 1’s in the bit string an even number. This mechanism decides the value of the parity bit. Odd parity makes the number of 1’s odd instead of even.
- Single Parity cannot detect burst errors
  - Burst errors cause errors in a sub-string of arbitrary length
  - A burst error is as likely to cause an even number of errors as an odd number of errors

Two Dimensional Parity Checks
- Arrange a string of bits as a two-dimensional array and compute parity over each row and each column of the array
- Can detect
  - Any number of errors in a single row (detect even number of errors with column parity)
  - Any number of errors in a single column (detect even number of errors with row parity)
- Does it protect against everything?
  - Answer is no. Why? Hint: Read between the lines. Single row or column
  - What about burst errors
  - Need something stronger. CRC codes

Two-Dimensional Parity

<table>
<thead>
<tr>
<th>Data</th>
<th>Parity byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>1</td>
</tr>
<tr>
<td>1101001</td>
<td>0</td>
</tr>
<tr>
<td>1011110</td>
<td>1</td>
</tr>
<tr>
<td>0001110</td>
<td>1</td>
</tr>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011111</td>
<td>0</td>
</tr>
</tbody>
</table>

- Each byte is protected by a parity bit
- The entire frame is protected by a parity byte

Cyclic Redundancy Check
- Treat the (n+1) bit message as a polynomial of degree n. The bits are the coefficients of the polynomial.
  - 1101 = 1*x^3 + 1*x^2 + 0*x^1 + 1*x^0
- Calculating CRC
  - Sender and transmitter choose a divisor polynomial of degree k, e.g. x^k + x^2 + 1. Call this C(x)
  - Add k bits to the (n+1) bit message such that the n+1+k bit message is exactly divisible by the divisor
- Choice of divisor is very important.
  - It determines the kind of errors that the CRC can guard against.

CRC Codes
- Burst errors are hard to model -- three parameters are typically used to measure the effectiveness of a code for error detection
  1. Minimum distance of the code
  2. Burst-detecting capability
  3. Probability that a random string is accepted as being error-free

CRC Computation
- Given:
  - Message: M(x)
  - Divisor: C(x)
- Multiply M(x) by x^k, i.e. add k zeroes to the end of the message. Call this T(x)
- Divide T(x) by C(x).
- Subtract the remainder from T(x)
- The result is the message including the CRC
CRC Computation

- \( C(x) = x^3 + x^2 + 1 \)
- \( M(x) = x^7 + x^4 + x^3 + x \)
- Subtraction: logical XOR operation

Generator 1101
11111001
10011010000 Message
1101
1001
1101
1000
1101
1011
1101
1100
1101
1000
1101
101 Remainder

CRC Codes

- Note: The CRC is computed over the entire message, not a byte or a row/column.
- When a message+CRC arrives at the destination, divide it by the generator polynomial. If the remainder is 0, the message is intact, else it has been corrupted.
  - The bit pattern that can cause a CRC code to fail is not a regular pattern such as random error or burst errors. That's why CRCs are strong.
  - Try out an example, where you try corrupting the CRC in the previous slide

Real World Example: Internet Checksum

- What's a checksum?
  - Take a guess, check sum!
  - Another error detection scheme
- Treat message as a sequence of 16-bit integers
- Add these integers together using 16-bit one’s-complement arithmetic
- Take the one’s complement of the result
- Resulting 16-bit number is the checksum

Example: Internet Checksum

```c
u_short cksum(u_short *buf, int count)
{
    register u_long sum = 0;
    while (count--)
    {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            /* carry, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```