Administrative updates

- I’m out all next week – no lectures, but...
- You will have work
  - HW #1: will be posted by 10pm tonight, will be due next Wed (Frank will collect)
  - Project #1: will also be posted by 10pm tonight, will be due the Monday after next (electronic turn-in before class)
- We’re stuck with this room
Last time...

- How to send a signal from here to there
Today: Data-link layer

- Framing
- Error detection/correction
- Media Access
- Bridging/Switching
Framing

- Goal: separate bitstream into distinct units of transfer (a frame)
- Why?
  - Synchronization recovery
  - Link multiplexing
  - Efficient error detection
- Challenges
  - How can we determine exactly what set of bits constitute a frame?
  - How do we determine the beginning and end of a frame?
Framing

- **Approaches**
  - Sentinel (like C strings)
  - Length-based (like Pascal strings)
  - Clock based

- **Characteristics**
  - Bit- or byte-oriented
  - Fixed or variable length
  - Data-dependent or data-independent length
Sentinel-Based Framing

- Basic idea: identify start/end of frame with special “marker”
  - Byte pattern, bit pattern, signal pattern

- Challenge: what if marker is in data stream

- Solution: “stuffing” recode data to prevent marker from occurring
Byte-oriented Sentinels

- STX – start of text
- ETX – end of text
- Problem: what if ETX appears in the data portion of the frame?
- Solution
  - If ETX appears in the data, introduce a special character DLE (Data Link Escape) before it
  - If DLE appears in the text, introduce another DLE character before it
  - Efficiency can be only 50%
- Protocol examples
  - BISYNC, PPP, DDCMP
Consistent-Overhead Byte Stuffing (COBS)

- Run length encoding applied to byte stuffing
  - Add implied 0 to end of frame
  - Each 0 is replaced with (number of bytes to next 0) + 1
  - What if no 0 within 255 bytes? – 255 value indicates 254 bytes followed by no zero
  - Worst case – no 0’s in packet – 1/254 overhead
- Appropriate for very low-bandwidth links

<table>
<thead>
<tr>
<th>Code</th>
<th>Followed by</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>(not applicable)</td>
<td>(not allowed)</td>
</tr>
<tr>
<td>0x01</td>
<td>No data bytes</td>
<td>A single zero byte</td>
</tr>
<tr>
<td>$n$</td>
<td>$(n-1)$ data bytes</td>
<td>Data followed by 0</td>
</tr>
<tr>
<td>0xFF</td>
<td>254 data bytes</td>
<td>Data, no following 0</td>
</tr>
</tbody>
</table>
Length-Based Framing

- End of frame
  - Calculated from length sent at start of frame
  - Challenge: Corrupt length markers
- Examples
  - DECNET’s DDCMP:
    » Byte-oriented, variable-length
  - RS-232 framing:
    » Bit-oriented, implicit fixed-length
Clock-Based Framing

- Continuous stream of fixed-length frames
- Clocks must remain synchronized
  - No bit or byte stuffing

- Example:
  - Synchronous Optical Network (SONET)

- Problems:
  - Frame synchronization
  - Clock synchronization
SONET

- All frames (STS formats) are 125 μsec long
- Problem: how to recover frame synchronization
  - 2-byte synchronization pattern starts each frame (unlikely in data)
  - Wait until pattern appears in same place repeatedly
- Problem: how to maintain clock synchronization
  - NRZ encoding, data scrambled (XOR’d) with 127-bit pattern
  - Creates transitions
  - Also reduces chance of finding false sync. pattern
SONET Multiplexing

- STS-3 has the payloads of three STS-1’s byte-wise interleaved.
- For STS-N, frame size is always 125 microsec
  - STS-1 frame is 810 bytes
  - STS-3 frame is 810x3 = 2430 bytes
SONET

- STS-1 merged bytewise round-robin into STS-3
- Unmerged (single-source) format called STS-3c
- Problem: simultaneous synchronization of many distributed clocks

not too difficult to synchronize clocks such that first byte of all incoming flows arrives just before sending first 3 bytes of outgoing flow
SONET

... but now try to synchronize this network’s clocks
Error Detection

- Goal: validate “correctness” of frame
- Idea: send additional redundant data with frame to check if it has been damaged

- Checked at many layers
  - Physical (e.g. modulation)
  - Datalink (e.g. cyclic redundancy check)
  - Network/Transport (e.g. IP Checksum)
  - Application (e.g. MD5 hash)

- Today: simple parity, redundancy w/voting, 2 dimensional parity, IP checksum, CRCs
Error Detection from 10,000 feet

- EDC = Error Detection bits (redundancy)
- D = Data protected by error checking, may include header fields
- Error detection not 100% reliable!
  - Protocol may miss some errors, but rarely
  - Larger EDC field yields better detection
Parity

- 1-bit error detection with parity
  - Add an extra bit to a code to ensure an even (odd) number of 1s
  - Every code word has an even (odd) number of 1s

Valid code words

Parity Encoding:
White – invalid (error)
Voting

- 1-bit error correction with voting
  - Every codeword is transmitted \( n \) times

Valid code words:

- White – correct to 1
- Blue - correct to 0

Voting:

```
011
010
000
001
111
110
100
101
```
**Hamming Distance**

- The Hamming distance between two code words is the minimum number of bit flips to move from one to the other
  - Example:
    - 00101 and 00010
    - Hamming distance of 3

- The minimum Hamming distance of a code is the minimum distance over all pairs of codewords
  - Minimum Hamming Distance for parity = 2
  - Minimum Hamming Distance for voting = 3

- N-bit error detection
  - No code word changed into another code word
  - Requires Minimum Hamming Distance of N+1
Two-Dimensional Parity

- **Use 1-dimensional parity**
  - Add one bit to a 7-bit code to ensure an even/odd number of 1s

- **Add 2nd dimension**
  - Add an extra byte to frame
    - Bits are set to ensure even/odd number of 1s in that position across all bytes in frame

- **Comments**
  - Catches all 1-, 2- and 3-bit and most 4-bit errors
Internet Checksum

- Idea
  - Add up all the words
  - Transmit the sum
- Internet Checksum
  - 1’s complement of the 1’s complement sum on 16bit codewords
  - Example
    » Message: e34f 2396 4427 99f3
    » 2s complement sum is 1e4ff
    » 1s complement sum is e4ff + 1 = e500
    » Checksum is 1aff

- Comments
  - VERY easy to implement, fast incremental updates
  - Not very robust
u_short cksum(u_short *buf, int count) {
    register u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
Cyclic Redundancy Check (CRC)

- Polynomial code
  - Treat packet bits a coefficients of n-bit polynomial
    » Message = 10011010
    » Polynomial
      \[ = 1 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 0 \]
      \[ = x^7 + x^4 + x^3 + x \]
  - Choose r+1 bit generator polynomial (well known – chosen in advance)
  - Add r bits to packet such that message is divisible by generator polynomial
  - Note: easy way to think of polynomial arithmetic mod 2
    » Multiplication: binary addition without carries
    » Division: binary subtraction without carries

- Better loss detection properties than checksums
Error Detection – CRC

- View data bits, \( D \), as a binary number
- Choose \( r+1 \) bit pattern (generator), \( G \)
- Goal: choose \( r \) CRC bits, \( R \), such that
  - \( \langle D, R \rangle \) exactly divisible by \( G \) (modulo 2)
  - Receiver knows \( G \), divides \( \langle D, R \rangle \) by \( G \). If non-zero remainder: error detected!
  - Can detect all burst errors less than \( r+1 \) bits
- Widely used in practice (Ethernet, FDDI, ATM)

\[
D \cdot 2^r \text{ XOR } R
\]
### Common Generator Polynomials

<table>
<thead>
<tr>
<th>Generator</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-8</td>
<td>$x^8 + x^2 + x^1 + 1$</td>
</tr>
<tr>
<td>CRC-10</td>
<td>$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$</td>
</tr>
<tr>
<td>CRC-12</td>
<td>$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$</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>$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$</td>
</tr>
</tbody>
</table>
**CRC – Example Encoding**

\[
x^3 + x^2 + 1 = 1101 \quad \text{Generator}
\]
\[
x^7 + x^4 + x^3 + x = 10011010 \quad \text{Message}
\]

1101

\[
\begin{array}{c}
1101 \\
10011010000
\end{array}
\]

Message plus k zeros

Result:
Transmit message followed by remainder:

\[
10011010101
\]
CRC – Example Decoding – No Errors

\[ x^3 + x^2 + 1 \]
\[ x^{10} + x^7 + x^6 + x^4 + x^2 + 1 = 1101 \]
\[ x^{10} + x^7 + x^6 + x^4 + x^2 + 1 = 10011010101 \]

\[ 1101 \]
\[ 10011010101 \]

k + 1 bit check sequence, equivalent to a degree-k polynomial

\[ 0m \mod c \]

Result:
CRC test is passed
CRC – Example Decoding – with Errors

\[ x^3 + x^2 + 1 = 1101 \]
\[ x^{10} + x^7 + x^5 + x^4 + x^2 + 1 = 10010110101 \]

Generator

Received Message

\[ x^{10} + x^7 + x^5 + x^4 + x^2 + 1 \]

\[ 10010110101 \]

Received message

\[ x^3 + x^2 + 1 \]

\[ 1101 \]

k + 1 bit check sequence c, equivalent to a degree-k polynomial

Two bit errors

Result:
CRC test failed

\[ m \mod c \]

Remainder

0101
Summary

- Framing
  - Bunching bits into distinct messages (frames)
  - Challenge is in finding where one frame starts and another begins

- Error detection
  - Determine if frame is corrupted by checking it against redundant data

- Next time (one week from next Monday): more on the datalink layer
  - Media access
  - Read 2.6, 2.7, 2.8