Point-to-Point Links

Outline
- Encoding (bits)
- Framing (frames)
- Error Detection
- Sliding Window Algorithm

Encoding

- Signals propagate over a physical medium
  - modulate electromagnetic waves
  - e.g., vary voltage
- Encode binary data onto signals
  - e.g., 0 as low signal and 1 as high signal
  - known as Non-Return to zero (NRZ)

Bits: 0 0 1 0 1 1 1 0 1 0 0 0 0 1 0

NRZ:  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |


Problem: Consecutive 1s or 0s

- Low signal (0) may be interpreted as no signal
- High signal (1) leads to baseline wander
  - Attenuation: the receiver uses the average to distinguish between low and high signals.
- Unable to recover clock
  - The clocks of the sender and the receiver must be synchronized
  - The receiver derives the signal from the signal transitions

Alternative Encodings

- Non-return to Zero Inverted (NRZI)
  - make a transition from current signal to encode a one; stay at current signal to encode a zero
  - solves the problem of consecutive ones

- Manchester
  - transmit XOR of the NRZ encoded data and the clock
  - only 50% efficient (bit rate = 1/2 baud rate)
    - Or requires higher bandwidth for higher baud rate
Encodings (cont)

- 4B/5B
  - every 4 bits of data encoded in a 5-bit code (p.79)
  - 5-bit codes selected to have no more than one leading 0 and no more than two trailing 0s
  - thus, never get more than three consecutive 0s
  - resulting 5-bit codes are transmitted using NRZI (for consecutive 1s)
  - achieves 80% efficiency
Framing

• Break sequence of bits into a frame
  – The beginning and the end of a frame?
• Typically implemented by network adaptor

![Diagram of Framing](image)

Approaches

• Santinell-based
• Bit-Oriented
  – e.g., HDLC
  – delineate frame with special pattern: 01111110
  – problem: special pattern appears in the payload
  – solution: *bit stuffing*
    • sender: insert 0 after five consecutive 1s
    • receiver: delete 0 that follows five consecutive 1s
• Byte-Oriented
  – e.g. PPP
  – Escape 01111110 by adding another 01111110
Approaches (cont)

- Clock-based
  - SONET (Synchronous Optical Network) STS-1frame: 90 bytes * 9
  - The first byte of each frame contain a special bit pattern
  - The receiver looks for the bit patterns that occurs every 810 bytes.

![Diagram showing overhead and payload with rows and columns]

Error Detection

- Sending two copies of data is inefficient
- Detect more errors with less overhead
- Two-Dimensional Parity
  - Even number of 1s
  - Catches all 1-2-3-bit errors
    - and most 4-bit errors

<table>
<thead>
<tr>
<th>Data</th>
<th>Parity bits</th>
<th>Parity byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>1</td>
<td>1111011</td>
</tr>
<tr>
<td>1101001</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011110</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0001110</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0110100</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1011111</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Internet Checksum Algorithm

- View message as a sequence of 16-bit integers; sum using 16-bit ones-complement arithmetic
- Simple to implement in software; Relies on complicated in layer CRC
  - E.g. word A LSB 1 to 0; Word B LSB 0 to 1

```c
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

- Add $k$ bits of redundant data to an $n$-bit message
  - want $k << n$
  - e.g., $k = 32$ and $n = 12,000$ (1500 bytes)
- Represent $n$-bit message as $n$-1 degree polynomial
  - e.g., $MSG=10011010$ as $M(x) = x^7 + x^4 + x^3 + x^1$
- Let $k$ be the degree of some divisor polynomial
  - e.g., $C(x) = x^3 + x^2 + 1$ (with degree $k$)
- Easy to implement in hardware
CRC (cont)

- Transmit polynomial $P(x)$ that is evenly divisible by $C(x)$
  - shift left $k$ bits, i.e., $M(x)x^k$
  - Remainder $E(x): M(x)x^k = C(x)\cdot ? + E(x)$
  - Transmit $P(x) = M(x)x^k + E(x)$

- Receiver receives $P'(x)$
  - $P'(x) = P(x) + \Delta(x) = C(x)\cdot ? + \Delta(x) = C(x)\cdot ?? + e(x)$
  - $e(x) = 0$ implies no errors, or $\Delta(x)$ happens to be divisible by $C(x)$
  - If no errors, $(P(x) - E(x))/x^k$ is the original message

CRC (cont)

- XOR division!
- Message 10011010
  - $\Rightarrow 10011010000$
- Divisor: 1101
- Reminder: 100
- Transmit: 10011010100
Selecting $C(x)$

- To detect all single-bit errors ($\Delta(x) = x^i$)
  - the $x^i$ and $x^0$ terms have non-zero coefficients.
- To detect all double-bit errors
  - $C(x)$ contains a factor with at least three terms
- To detect any odd number of errors
  - $C(x)$ contains the factor $(x + 1)$
- To detect any ‘burst’ error (i.e., sequence of consecutive error bits) with a length less than $k$ bits.
  - Most burst errors of larger than $k$ bits can also be detected
- See Table 2.6 on page 102 for common $C(x)$

Acknowledgements & Timeouts

Detect duplicate ACKs!
Stop-and-Wait

- Problem: keeping the pipe full
- Example
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB)
  - 1KB frames implies 1/8th link utilization

Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called window
SW: Sender

- Assign sequence number to each frame (SeqNum)
- Maintain three state variables:
  - send window size (SWS)
  - last acknowledgment received (LAR)
  - last frame sent (LFS)
- Maintain invariant: LFS - LAR <= SWS
- Advance LAR when ACK \geq LAR arrives
- Buffer up to SWS frames

SW: Receiver

- Maintain three state variables
  - receive window size (RWS)
  - largest frame acceptable (LFA)
  - last frame received i.e. received in order! (LFR)
- Maintain invariant: LFA - LFR <= RWS
- Frame SeqNum arrives:
  - if LFR < SeqNum \leq LFA accept
  - if SeqNum <= LFR or SeqNum > LFA discarded
- Send cumulative ACKs
- Advance LFR and deliver data to the application when LFR + 1 arrives
Sequence Number Space

- **SeqNum** field is finite; sequence numbers wrap around
- Sequence number space must be larger then number of outstanding frames
- **SWS ≤ MaxSeqNum - 1** is not sufficient
  - suppose 3-bit **SeqNum** field (0..7)
  - **SWS=RWS=7**
  - sender transmit frames 0..6
  - arrive successfully, but ACKs lost
  - sender retransmits 0..6
  - receiver expecting 7, 0..5, but receives second incarnation of 0..5
- **SWS < (MaxSeqNum+1)/2** ! (similar to Stop&Wait)
- Intuitively, **SeqNum** “slides” between two halves of sequence number space