Data Link Layer (1)

Networked Systems 3
Lecture 6
Purpose of Data Link Layer

- Arbitrate access to the physical layer
  - Structure and frame the raw bits
  - Provide flow control
  - Detect and correct bit errors
  - Perform media access control

- Turn the raw bit stream into a structured communications channel
Framing and Synchronisation

• Physical layer provides unreliable raw bit stream
  • Bits might be corrupted
  • Timing can be disrupted

• Data link layer must correct these problems
  • Break the raw bit stream into *frames*
  • Transmit and repair individual frames
  • Limit scope of any transmission errors
Frame Structure

- Synchronisation
- Timing recovery
- Frame identification
- Network layer protocol data

- Start Code
- Header
- Data (~kbytes)
- Error Detection
Example: PPP Frame

| 01111110 | Address | Control | Protocol | Payload (≤ 1500 bytes) | Checksum (2 bytes) | 01111110 |

Start code

Headers

Data (IP packet)

Error detection

Maximum Transmission Unit (MTU)
Synchronisation (1)

• How to detect the start of a message?
  • Leave gaps between frames
    • Problem – physical layer typically doesn’t guarantee timing (clock skew, etc.)
  • Precede each frame with a length field
    • What if that length is corrupted? How to find next frame?
  • Add a special *start code* to beginning of frame
    • A unique bit pattern that *only* occurs at the start of each frame
    • Enables synchronisation after error – wait for next start code, begin reading frame headers
Synchronisation (2)

- What makes a good start code?
  - Must not appear in the frame headers, data, or error detecting code
  - Must allow timing recovery

Start code should generate a regular pattern after physical layer coding.

Manchester Encoding

Receiver measures timing
Synchronisation (3)

• What if the start code appears in the data?

• Use \textit{bit stuffing} to give a transparent channel
  
  • $11111 \rightarrow 111110 \rightarrow 11111$

• Can also use byte stuffing – double up the start code byte if it appears in the data

\[
\begin{align*}
01101111111111111110010 \\
\downarrow & \quad \text{Prepare for transmission} \\
01101111101111011111010010 \\
\downarrow & \quad \text{Stuffed bits} \\
011011111111111111110010 \\
\downarrow & \quad \text{Destuff at receiver}
\end{align*}
\]
Error Detection

• Noise and interference at the physical layer can cause bit errors
  • Rare in wired links, common in wireless systems

• Add *error detecting code* to each packet
Parity Codes

• Simplest error detecting code
• Calculate *parity* of the data
  • How many 1 bits are in the data?
  • An odd number → parity 1
  • An even number → parity 0
  • Parity bit is the XOR ("⊕") of data bits
• Transmit parity with the data, check at receiver
  • Detects all single bit errors
#include <stdint.h>

// Internet checksum algorithm. Assumes
// data is padded to a 16-bit boundary.
uint16_t
internet_cksum(uint16_t *buf, int buflen)
{
    uint32_t sum = 0;

    while (buflen--) {
        sum += *(buf++);
        if (sum & 0xffff0000) {
            // Carry occurred, wrap around
            sum &= 0x0000ffff;
            sum++;
        }
    }
    return ~(sum & 0x0000ffff);
}

• Sum data values, send as a **checksum** in each frame
  • Internet protocol uses a 16 bit ones complement checksum

• Receiver recalculates, mismatch → bit error

• Better error detection than parity code
  • Detects many multiple bit errors
Other Error Detecting Codes

• Parity codes and checksums relatively weak
  • Simple to implement
  • Undetected errors reasonably likely

• More powerful error detecting codes exist
  • *Cyclic redundancy code* (CRC)
  • More complex → fewer undetected errors
  • (see recommended reading for details)
Error Correction

• How to correct bit errors?
  • **Forward error correction (FEC)**
    • Sender includes additional information in the initial transmission, allowing receiver to correct the error itself
  • **Automatic repeat request (ARQ)**
    • Receiver contacts sender to request a retransmission of the incorrect data
Forward Error Correction

- Extend error detecting codes to *correct* errors
  - Sender transmits error correcting code
    - Additional data within each frame
    - Additional frames
  - Allows receiver to correct (some) errors without contacting sender
FEC: Within a Frame

• **Example: Hamming code**
  
  • Send $n$ data bits and $k$ check bits every word
  
  • **Check bits** are sent as bits 1, 2, 4, 8, 16, ...
  
  • Each check bit codes parity for some data bits
    
    • $b_1 = b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \ldots$
    
    • $b_2 = b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \oplus b_{14} \oplus b_{15} \ldots$
    
    • $b_4 = b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} \ldots$
    
    • i.e. starting at check bit $i$, check $i$ bits, skip $i$ bits, repeat

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII</th>
<th>Hamming Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1001000</td>
<td>00110010000</td>
</tr>
<tr>
<td>a</td>
<td>1100001</td>
<td>10111001001</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11101010101</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11101010101</td>
</tr>
<tr>
<td>i</td>
<td>1101001</td>
<td>01101011001</td>
</tr>
<tr>
<td>n</td>
<td>1101110</td>
<td>01101010110</td>
</tr>
<tr>
<td>g</td>
<td>1100111</td>
<td>11111001111</td>
</tr>
<tr>
<td></td>
<td>0100000</td>
<td>10011000000</td>
</tr>
<tr>
<td>c</td>
<td>1100011</td>
<td>11111000011</td>
</tr>
<tr>
<td>o</td>
<td>1101111</td>
<td>00101011111</td>
</tr>
<tr>
<td>d</td>
<td>1100100</td>
<td>11111001100</td>
</tr>
<tr>
<td>e</td>
<td>1100101</td>
<td>00111000101</td>
</tr>
</tbody>
</table>
FEC: Within a Frame

- On reception:
  - Set `counter = 0`
  - Recalculate each check bit, `k`, in turn (`k = 1, 2, 4, 8, ...`); if incorrect, `counter += k`
  - If (`counter == 0`) {
      - no errors
  } else {
      - bit `counter` is incorrect
  }

- Allows correction of all single bit errors

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII</th>
<th>Hamming Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1001000</td>
<td>00110010000</td>
</tr>
<tr>
<td>a</td>
<td>1100001</td>
<td>10111001001</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11010101011</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11010101011</td>
</tr>
<tr>
<td>i</td>
<td>1101001</td>
<td>01101011001</td>
</tr>
<tr>
<td>n</td>
<td>1101110</td>
<td>01101010110</td>
</tr>
<tr>
<td>g</td>
<td>1100111</td>
<td>11111001111</td>
</tr>
<tr>
<td></td>
<td>0100000</td>
<td>10011000000</td>
</tr>
<tr>
<td>c</td>
<td>1100011</td>
<td>11111000111</td>
</tr>
<tr>
<td>o</td>
<td>1101111</td>
<td>00101011111</td>
</tr>
<tr>
<td>d</td>
<td>1100100</td>
<td>11111001100</td>
</tr>
<tr>
<td>e</td>
<td>1100101</td>
<td>00111000101</td>
</tr>
</tbody>
</table>
FEC: Error Correcting Frames

• Alternative: send extra error correcting frames

• Example: packet level parity
  • Send \( k \) parity packets every \( n \) data packets
  • Each parity packet is \( \oplus \) of the data packets
  • Correct loss or error since \( \oplus \) commutative
  • E.g. if B in error, repair B = A \( \oplus \) C \( \oplus \) D \( \oplus \) P

\[
\begin{align*}
A & : 0100001000011011 \\
B & : 1101011011110001 \\
C & : 0111001011010010 \\
D & : 1001001010110110 \\
P & : 1100110100101011 \\
P & = A \oplus B \oplus C \oplus D
\end{align*}
\]
Automatic Repeat Request

• Each frame includes a sequence number
• Receiver sends *acknowledgements* as it receives data frames
  • Can be sent as dedicated acknowledgement frames, or piggybacked onto returning data frames
  • Can be a positive acknowledgement (“I got frame $n$”) or a negative acknowledgement (“frame $n$ is missing”)

Stop and Wait

- Simplest ARQ scheme:
  - Transmit a frame
  - Await positive acknowledgement from receiver
  - If no acknowledgement after some time out, retransmit frame

- Limitation:
  - One frame outstanding on link → limited performance on links with high bandwidth × delay product
Bandwidth × Delay Product

- Signal has limited speed:
  - ≈ 2.3x10^8 m/s in electrical cable,
  - ≈ 2.0x10^8 m/s in optical fibre
- Determines propagation delay for the link
  - Baseline value – queuing will lead to higher delays
- Example link capacity:
  - Glasgow – London (~670km) → 3ms propagation delay

• Assume a 10 gigabit per second link speed
• 0.003 seconds x 10000000000 bits/second = 30000000 bits link capacity (~3.5 Mbytes of data in flight)

Bandwidth x delay = link capacity
Link Utilisation

- Assume it takes time, $t_s$, to serialise a frame onto link
  - $t_s = \text{frame size} / \text{link bandwidth}$
- Acknowledgement returns $t_{RTT}$ seconds later
- Utilisation, $U = t_s / t_{RTT}$
  - Desire link fully utilised: $U \sim 1.0$
  - But $U \ll 1.0$ for stop-and-wait
Sliding Window Protocol

Improve link utilisation in ARQ by allowing several frames to be outstanding on link.

Window size = bandwidth x delay for full link utilisation.
Sliding Window Protocol

- Stop-and-wait acceptable in LAN
  - Bandwidth delay product small, since RTT tiny
  - Reasonably efficient

- Variants on sliding window protocol required for wide area ARQ
  - How to choose window size? What is acknowledged?
  - Example: TCP congestion control
Questions?