L2: Data Link Layer

Transmits *frames* of data across a link

L2 Functions:

- Accepts data from higher layers and forms it into standard frames
- Synchronizes frames across the link
- Controls the rate of frame flow so that receiver is not overwhelmed
- Recognizes and deals with frame errors
- Manages the link

The LLC Sublayer

The data link layer is usually thought of (and often implemented) as two distinct sublayers performing different functions

**Logical Link Control (LLC) Sublayer**
Performs Data Link functions that are independent of the type of medium used

**Media Access Control (MAC) Sublayer**
Performs Data Link functions that depend on the type of medium used
The LLC sublayer of L2

Frames

First bit sent

Typical data frame structure:

<table>
<thead>
<tr>
<th>Header</th>
<th>Body</th>
<th>Trailer</th>
</tr>
</thead>
</table>

- Synch bytes
- Protocol options
- Frame identifier
- Body pointer
- Data from higher layers
- Error codes

Considerations in frame design:
- Need a dependable way to identify the start and end of frame
- If we have a variable-length body, we need a way to find the end
- We need a way to distinguish frame control patterns from the same patterns in data
- On multidrop links, we need a way to identify source and destination addresses
- We need to send both Data and Link Control messages (usually two separate formats)
Types of frame formats

- Ways to identify start/end of frames
  - Byte counts used to determine body length
    - Example: DDCMP - Byte-oriented data
  - Special flags to mark start/end of body
    - Examples: BISYNC / BSC - Byte-oriented data, HDLC - Bit-oriented data

- Frames and fields identified by timing
  - Tn - PSTN / Copper
  - SONET - PSTN / Optical

A byte-oriented protocol using counts
DEC Digital Data Comm Msg Protocol (DDCMP)

- Byte-oriented
- # bytes in body is sent as part of the frame

<table>
<thead>
<tr>
<th>bits:</th>
<th>8</th>
<th>8</th>
<th>8</th>
<th>14</th>
<th>42</th>
<th>var</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SYN</td>
<td>SYN Class</td>
<td>Count</td>
<td>Header</td>
<td>Body</td>
<td>CRC</td>
<td></td>
</tr>
</tbody>
</table>

Synch flags # bytes in body error code

A problem:

Sent:
- Frame 1:
  - SYN
  - SYN Class: 32
- Frame 2:
  - CRC
  - SYN
  - SYN Class

Rcvd:
- Frame 1:
  - SYN
  - SYN Class: 34
- Frame 2:
  - CRC
  - SYN
  - SYN Class

Rcvr thinks Frame 1 ends here

The error will cause a bad CRC (error code). Rcvr will recover at the next SYN SYN.
But 2 frames are lost due to one bit error - a big penalty.
A byte-oriented protocol using sentinals
IBM Binary Synchronous Comm (BISYNC/BSC)

- Byte-oriented
- Fields marked by sentinal characters.
- Byte stuffing

```
Bytes: 1 1 1 1 1 2
SYN SYN SOH Header STX Body ETX CRC
```

"Transparency Mode": Any occurrence of one of the special characters in body is preceded by the character "DLE" (including DLE). This is called "byte stuffing"

**Example:**
Data = a ETX b STX c DLE d
Frame = SYN SYN SOH Header STX a DLE ETX b DLE STX c DLE d ETX CRC

---

A bit-oriented protocol
ISO High-level Data Link Control (HDLC)

- Bit-oriented
- Start and end of frame marked with special flags
- Bit stuffing needed to protect the flags

```
bits: 8 16 var 16 8
01111110 Header Body CRC 01111110
```

**Problem:** Body data may contain "01111110" -- receiver will read this as the end of the frame

**The fix: Bit stuffing**

**SENDER LOGIC:** After every sequence of five ones in data, stuff a zero

**RECEIVER LOGIC:**
IF five ones in a row have been received, THEN
{Read the next bit
  IF bit = zero THEN (this is a stuffed zero, delete it)
  ELSE
    {Read the next bit
      IF bit = zero THEN (this is the flag marking the end of the frame)
      ELSE (there are 7 ones in a row - ERROR)
    }
  }
}
Clock-based framing protocols - T1

- Generally have less overhead than other types
- Require a higher level of clock synchronization

T1 / DS1 Frame format:

<table>
<thead>
<tr>
<th>Bits</th>
<th>1</th>
<th>8</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sample from Voice chan 1</td>
<td>Sample from Voice chan 2</td>
<td>Sample from Voice chan 24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 data bits + 1 control</td>
<td>7 data bits + 1 control</td>
<td>7 data bits + 1 control</td>
<td></td>
</tr>
</tbody>
</table>

5.2 usec

193 bits, 125 usec

The receiver determines where fields start based on timing.

Note: 193 bits / 125 usec = 1.544 Mbps (this is the T1 bandwidth)

How bandwidth requirements increase

Sample
- 1 voice sample from a single circuit (8 bits)
- Sample must be sent every 125 usec
- Bandwidth required = 8 bits / 125 usec = 64Kbps

T1 frame
- 1 sample from 24 conversations plus control bits (193 bits)
- Frame must be sent every 125 usec
- Bandwidth required = 193 bits / 125 usec = 1.544 Mbps

T3 frame
- 28 T1 links + control bits
- Frame must be sent every 125 usec
- Bandwidth required = 44.7 Mbps
Synchronous Optical Network (SONET)

SONET STS-1 (OC-1) Frame format

- A SONET STS-1 Frame = 9 x 90 = 810 bytes (= samples from 810 voice circuits)
- A new frame must be sent each 125 usec
- Bandwidth required = 8 x 810 bytes / 125 usec = 51.84 Mbps

A higher-rate “STS-n” SONET link is formed by interleaving n STS-1 frames:

<table>
<thead>
<tr>
<th>STS</th>
<th>BW</th>
<th>STS</th>
<th>BW</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS-1</td>
<td>51.84 Mbps</td>
<td>STS-48</td>
<td>2.49 Gbps</td>
</tr>
<tr>
<td>STS-3</td>
<td>148.6 Mbps</td>
<td>STS-192</td>
<td>9.95 Gbps</td>
</tr>
</tbody>
</table>

Error Control
Error Control

• How do we detect that a frame/packet we have received contains an error? (*Error Detection*)

• And if we do, what do we do about it? (*Error Correction*)

Review: Types of binary errors

• Random errors (single-bit errors)
  – Inverted/lost bit

• Burst Errors
Error Detection

- Usually done by sending redundant bits
- We’d like to minimize the number of bits we add
- We think about detection methods in terms of:
  - Numbers/types of errors detected
  - Efficiency (1-fraction of bits that are added for error control)

Example: Send two copies of data
Detects any odd number of bits in error
Efficiency = 1-.5 = 50%

Simple parity

\[ b_{n-1} b_{n-2} \ldots b_1 b_0 p \]

The value of the parity bit is chosen so that the total number of 1’s is:
- Even (called “even parity”), or
- Odd (called “odd parity”)

Simple, fast, detects any odd number of bit errors

Example: Odd parity, n=7
Data = 0110111  \hspace{1cm} \text{parity} = 0
Data = 1110010 \hspace{1cm} \text{parity} = 1
Efficiency = 1 - 1/8 = 87.5%

Example: Even parity, n=3
Data = 011 \hspace{1cm} \text{parity} = 0
Data = 111 \hspace{1cm} \text{parity} = 1
Efficiency = 75%
Checksum

- Checksum is calculated by adding the data bytes
- Simple, fast for large data blocks (files, etc.)
- Detects all single errors, many others
- High efficiency (example: 1KB file, 16-bit checksum $\rightarrow$ 99.8% efficiency)

Cyclic Redundancy Code (CRC)

- Very high efficiency
- Detects single, double, all odd errors
- Adding "r" CRC bits allows detection of burst errors of r-1 bits
- Complex-looking arithmetic, but easily implemented in hardware (see text)
- Used in many networking applications
Error Correction

- Error correcting codes allow receiver not only to determine that there was an error but also to determine which bit(s) are incorrect.
- Requires more added bits than detection alone.

Example: Send three copies of data. Detects any number of bit errors as long as two copies of the bit are correct. Efficiency = 1 - .67 = 33%

Example: 2-D parity error correction method

Corrects any single bit error – byte parity error identifies the byte, bit parity error identifies the bit.

<table>
<thead>
<tr>
<th>Transmitted</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 1 0</td>
<td>1 0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 1 1 1 1</td>
<td>0 0 1 1 1</td>
</tr>
<tr>
<td>1 0 0 0 0 1 1 0</td>
<td>1 0 0 0 0 1 1 0</td>
</tr>
<tr>
<td>1 1 0 1 0 0 0 1</td>
<td>1 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>
Implementation of bit error control in networks

• Most reliable network protocols:
  – Use CRC codes (or equiv) to detect errors
  – Re-transmit to correct errors

• Error correction codes are used in some highly-specialized applications (e.g, Mars lander comm links) where complexity can be justified