Link Layer
Link Layer

• Transfer **frames** over one or more connected links
  • Frames are messages of limited size
  • Builds on the physical layer which moves stream of **bits**
In terms of layers ...

Network

Sending machine
Packet

Receiving machine
Packet

Link

Physical Actual data path
In terms of layers ...

Network

Sending machine
Packet

Receiving machine
Packet

Link

Header Payload field Trailer

Virtual data path

Physical

Actual data path
Typical Implementation of Layers (2)
Topics we’ll cover

1. Framing
   • Delimiting start/end of frames
2. Error detection and correction
   • Handling errors
3. Multiple Access
   • 802.11, classic Ethernet
4. Switching
   • Modern Ethernet
Framing

Delimiting start/end of frames
Framing: Problem

• How do we interpret a stream of bits as a sequence of frames?
Framing Methods

1. Fixed-size frames (motivation)
2. Byte count (motivation)
3. Byte stuffing
4. Bit stuffing

• In practice, the physical layer often helps to identify frame boundaries
  • E.g., Ethernet, 802.11
1. Fixed-size frames

• Make every frame a fixed number of bits
  • Pad smaller frames

• Problems?
  • Wasted transmissions for small frames
2. Byte Count

• Start each frame with a length field

• Problems?
2. Byte Count: Problem

• Difficult to re-synchronize after framing error
  • Want a way to scan for a start of frame
3. Byte Stuffing

- A special **flag** byte value for start/end of frame
  - Replace ("stuff") the flag with an escape code

- Problems?
3. Byte Stuffing: Problem

- Must escape the escape code too! Rules:
  - Replace each FLAG in data with ESC FLAG
  - Replace each ESC in data with ESC ESC

- Now any unescaped FLAG denotes frame start/end
# Unstuffing

You see:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>What it means</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Solitary FLAG?</td>
<td>-&gt; Start or end of packet</td>
</tr>
<tr>
<td>2.</td>
<td>Solitary ESC?</td>
<td>-&gt; Bad packet!</td>
</tr>
<tr>
<td>3.</td>
<td>ESC FLAG?</td>
<td>-&gt; remove ESC and pass FLAG through</td>
</tr>
<tr>
<td>4.</td>
<td>ESC ESC FLAG?</td>
<td>-&gt; removed ESC and then start of end of packet</td>
</tr>
<tr>
<td>5.</td>
<td>ESC ESC ESC FLAG?</td>
<td>-&gt; pass ESC FLAG through</td>
</tr>
<tr>
<td>6.</td>
<td>ESC FLAG FLAG?</td>
<td>-&gt; pass FLAG through then start of end of packet</td>
</tr>
</tbody>
</table>
4. Bit Stuffing

• Can stuff at the bit level too
  • Call a flag six consecutive 1s
  • On transmit, after five 1s in the data, insert a 0
  • On receive, a 0 after five 1s is deleted

Data bits: 0110111111111111111111111110010

Transmitted bits with stuffing: 01101111101111111011111110110010

Stuffed bits
Link Example: PPP over SONET

- PPP is Point-to-Point Protocol
- Widely used for link framing
  - E.g., it is used to frame IP packets that are sent over SONET optical links
Link Example: PPP over SONET (2)

- Think of SONET as a bit stream, and PPP as the framing that carries an IP packet over the link

Protocol stacks

PPP frames may be split over SONET payloads
Link Example: PPP over SONET (3)

• Framing uses byte stuffing
  • **FLAG** is 0x7E and **ESC** is 0x7D

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1 or 2</th>
<th>Variable</th>
<th>2 or 4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>01111110</td>
<td>Address</td>
<td>11111111</td>
<td>Control</td>
<td>00000011</td>
<td>Protocol</td>
<td>Payload</td>
</tr>
</tbody>
</table>

CSE 461 University of Washington
Link Example: PPP over SONET (4)

• Byte stuffing method:
  • To stuff (unstuff) a byte
    • add (remove) ESC (0x7D)
    • and XOR byte with 0x20
  • Removes \textbf{FLAG} from the contents of the frame
Link Layer: Error detection and correction
Problem: Noise may Flip Received Bits

• Link layers provides some protection
  • Detect errors with codes
  • Correct errors with codes
  • Retransmit lost frames → Later

• Reliability concern cuts across the layers
  • E.g, TCP in the transport layer, DNS in the app layer
Problem: Noise may Flip Received Bits

<table>
<thead>
<tr>
<th>Signal</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slightly Noisy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Very noisy</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Ideas?
Approach – Add Redundancy

• Error detection codes: Add check bits to the message bits to let some errors be detected

• Error correction codes: Add more check bits to let some errors be corrected

• Key issue: Structure the code such that
  • Need few check bits to detect/correct many errors
  • Modest computation
Motivating Example

• A simple code to handle errors:
  • Send two copies! Error detected if different.

• How good is this code?
  • How many errors can it detect/correct?
  • How many errors will make it fail?
Want to Handle More Errors w/ Fewer Bits

• We’ll look at better codes (applied mathematics)
  • But, they can’t handle all errors
  • And they focus on accidental (random) errors
Using Error Codes

- Codeword consists of D data plus R check bits (=systematic block code)

\[
\begin{array}{c|c}
\text{Data bits} & \text{Check bits} \\
\hline
D & R = fn(D) \\
\end{array}
\]

- **Sender:**
  - Compute R check bits based on the D data bits; send the codeword of D+R bits
Using Error Codes (2)

• Receiver:
  • Receive D+R bits with unknown errors
  • Recompute R check bits based on the D data bits
  • Error detected if R doesn’t match R’

\[
\begin{align*}
\text{Data bits} & \quad \text{Check bits} \\
D & \quad R' \\
R = fn(D) & = ?
\end{align*}
\]
Intuition for Error Codes

• For D data bits, R check bits:
  - Randomly chosen D+R bits is unlikely to be correct
  - Low, controlllable overhead
R.W. Hamming (1915-1998)

• Much early work on codes:
  • “Error Detecting and Error Correcting Codes”, BSTJ, 1950

• See also:
  • “You and Your Research”, 1986
Hamming Distance

• Distance is the number of bit flips needed to change $D_1$ to $D_2$

• **Hamming distance** of a coding is the minimum error distance between any pair of codewords (bit-strings) that cannot be detected
Hamming Distance (2)

• **Error detection:**
  • For a coding of distance $d+1$, up to $d$ errors will always be detected

• **Error correction:**
  • For a coding of distance $2d+1$, up to $d$ errors can always be corrected by mapping to the closest valid codeword
Simple Error Detection – Parity Bit

• Take D data bits, add 1 check bit
  • Check bit could be sum modulo 2 or XOR
Parity Bit (2)

• How well does parity work?
  • What is the distance of the code?
  • How many errors will it detect/correct?

• What about larger errors?
Checksums

• Idea: sum up data in N-bit words
  • Widely used in, e.g., TCP/IP/UDP

| 1500 bytes | 16 bits |

• Stronger protection than parity
Internet Checksum

• Sum is defined in 1s complement arithmetic (must add back carries)
  • And it’s the negative sum

• “The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words ...” – RFC 791
Internet Checksum (2)

Sending:
1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

\[
\begin{align*}
0001 & \quad f204 \\
& \quad f4f5 \\
& \quad f6f7 \\
\end{align*}
\]
Internet Checksum (3)

Sending:
1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum
Internet Checksum (4)

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

```
0001
f204
f4f5
f6f7
+ 220c
-----
```
Internet Checksum (5)

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

\[
\begin{array}{c}
0001 \\
f204 \\
f4f5 \\
f6f7 \\
+ 220c \\
\hline
2fffd \\
\hline
fffd \\
+ 2 \\
\hline
ffff \\
\hline
0000
\end{array}
\]
How well does the checksum work?
  - What is the distance of the code?
  - How many errors will it detect/correct?