Lecture 4: Layers & Framing
Lecture 4 Overview

- Finish encoding schemes
  - Manchester, 4B/5B, etc.

- Layering
  - Focus on the Link Layer

- Framing
  - Stuffing
Synchronous Coding

- Asynchronous receiver phase locks each symbol
  - Takes time, limiting transmission rates

- So, start symbols need to be extra slow
  - Need to fire up the clock, which takes time

- Instead, let’s do this training once, then just keep sync
  - Need to continually adjust clock as signal arrives
  - Ever hear of Phase Lock Loops (PLLs)?

- Basic idea is to use transitions to lock in
Non-Return to Zero (NRZ)

- **Signal to Data**
  - High $\Rightarrow$ 1
  - Low $\Rightarrow$ 0

- **Comments**
  - Transitions maintain clock synchronization
  - Long strings of 0s confused with no signal
  - Long strings of 1s causes *baseline wander*
    - We use average signal level to infer high vs low
  - Both inhibit clock recovery
Non-Return to Zero Inverted (NRZI)

- **Signal to Data**
  - Transition \( \Rightarrow \) 1
  - Maintain \( \Rightarrow \) 0

- **Comments**
  - Solves series of 1s, but not 0s

![NRZ and NRZI waveforms](image-url)
Manchester Encoding
(10Mbps Ethernet)

- **Signal to Data**
  - XOR NRZ data with senders clock signal
  - High to low transition $\Rightarrow 1$
  - Low to high transition $\Rightarrow 0$

- **Comments**
  - Solves clock recovery problem
  - Only 50% efficient (½ bit per transition)
  - Still need preamble (typically 0101010101… trailing 11 in Ethernet)
4B/5B (100Mbps Ethernet)

- Goal: address inefficiency of Manchester encoding, while avoiding long periods of low signals
- Solution:
  - Use five bits to encode every sequence of four bits
  - No 5 bit code has more than one leading 0 and two trailing 0’s
  - Use NRZI to encode the 5 bit codes
  - Efficiency is 80%

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11110</td>
</tr>
<tr>
<td>0001</td>
<td>01001</td>
</tr>
<tr>
<td>0010</td>
<td>10100</td>
</tr>
<tr>
<td>0011</td>
<td>10101</td>
</tr>
<tr>
<td>0100</td>
<td>01010</td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
</tr>
<tr>
<td>0110</td>
<td>01110</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10010</td>
</tr>
<tr>
<td>1001</td>
<td>10011</td>
</tr>
<tr>
<td>1010</td>
<td>10110</td>
</tr>
<tr>
<td>1011</td>
<td>10111</td>
</tr>
<tr>
<td>1100</td>
<td>11010</td>
</tr>
<tr>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>1110</td>
<td>11100</td>
</tr>
<tr>
<td>1111</td>
<td>11101</td>
</tr>
</tbody>
</table>
Manchester Encoding (10Mbps Ethernet)

- Signal to Data
  - XOR NRZ data with senders clock signal
  - High to low transition $\Rightarrow$ 1
  - Low to high transition $\Rightarrow$ 0

- Comments
  - Solves clock recovery problem
  - Only 50% efficient (½ bit per transition)
  - Still need preamble (typically 01010101... trailing 11 in Ethernet)

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

NRZ

Clock

Manchester
4B/5B (100Mbps Ethernet)

- Goal: address inefficiency of Manchester encoding, while avoiding long periods of low signals

- Solution:
  - Use five bits to encode every sequence of four bits
  - No 5 bit code has more than one leading 0 and two trailing 0’s
  - Use NRZI to encode the 5 bit codes
  - Efficiency is 80%

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11110</td>
</tr>
<tr>
<td>0001</td>
<td>01001</td>
</tr>
<tr>
<td>0010</td>
<td>10100</td>
</tr>
<tr>
<td>0011</td>
<td>10101</td>
</tr>
<tr>
<td>0100</td>
<td>01010</td>
</tr>
<tr>
<td>0101</td>
<td>01011</td>
</tr>
<tr>
<td>0110</td>
<td>01110</td>
</tr>
<tr>
<td>0111</td>
<td>01111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4-bit</th>
<th>5-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10010</td>
</tr>
<tr>
<td>1001</td>
<td>10011</td>
</tr>
<tr>
<td>1010</td>
<td>10110</td>
</tr>
<tr>
<td>1011</td>
<td>10111</td>
</tr>
<tr>
<td>1100</td>
<td>11010</td>
</tr>
<tr>
<td>1101</td>
<td>11011</td>
</tr>
<tr>
<td>1110</td>
<td>11100</td>
</tr>
<tr>
<td>1111</td>
<td>11101</td>
</tr>
</tbody>
</table>
Encoding Summary

- **Signaling & Modulation**
  - Transforming digital signal to and from analog representation
  - Fundamental limits (Shannon)
  - Lots of ways to encode signal (modulation) onto a given medium

- **Clock recovery**
  - Receiver needs to adjust its sampling times to best extract signal from channel
  - Sender can code signal to make it far easier to do this
Communications is complicated
  - Modulation and encoding bits
  - Splitting sequences of bits into packets
  - Fixing errors
  - Controlling access to network
  - Routing
  - Recovering from lost messages
  - Etc….

Really hard to think about all of this and get it right

Not all applications need all of it

How to achieve interoperability?
Layering: A Modular Approach

- Sub-divide the problem
  - Each layer relies on services from layer below
  - Each layer exports services to layer above

- Interface between layers defines interaction
  - Hides implementation details (encapsulation)
  - Layers can change without disturbing other layers (modularity)

- Interface among peers in a layer is a protocol
  - If peers speak same protocol, they can interoperate
Key Design Decision

- How do you divide functionality across the layers?

- **End-to-end argument [Saltzer84]**
  - Functionality should be implemented at a lower layer iff it can be **correctly** and **completely** implemented there.
  - Incomplete versions of a function can be used as a performance enhancement, but not for correctness.

- Early, and still relevant, example
  - ARPAnet provided reliable link transfers between switches.
  - Was this enough for reliable communication?
  - No, packets could still get corrupted on host-switch link, or inside of the switches.
  - Hence, still need reliability at higher layers.
Protocol Standardization

- Communicating hosts speaking the same protocol
  - Standardization to enable multiple implementations
  - Or, the same folks have to write all the software

- Internet Engineering Task Force
  - Based on working groups that focus on specific issues
  - Produces “Request For Comments” (RFCs)
    » Rough consensus and running code
    » After enough time passes, promoted to Internet Standards

- Other standards bodies exist
  - ISO, ITU, IEEE, etc.
Encapsulation via Packet Headers

- Typical Web packet

- Notice that layers add overhead
  - Space (headers), effective bandwidth
  - Time (processing headers, “peeling the onion”), latency
The Hourglass Model

The Hourglass Model

Applications
Transport
Data Link
Physical

FTP
HTTP
NV
TFTP
TCP
UDP
IP
NET_1
NET_2
...
NET_n

“Thin Waist”
So Far: Physical layer

- **Tasks**
  - Encode binary data from source node into signals that physical links carry
  - Signal is decoded back into binary data at receiving node
  - Work performed by network adapter at sender and receiver

- **Synchronous encoding algorithms**
  - NRZ, NRZI, Manchester, 4B/5B, etc
Moving on: (Data) Link Layer

- **Framing**
  - Break stream of bits up into discrete chunks

- **Error handling**
  - Detect and/or correct errors in received frames

- **Media access**
  - Arbitrate which nodes can send frames at any point in time
  - Not always necessary; e.g. point-to-point duplex links

- **Multiplexing**
  - Determine appropriate destination for a given frame
  - Also not always required; again, point-to-point
Today’s Focus: Framing

- Break down a stream of bits into smaller, digestible chunks called **frames**

- Allows the physical media to be shared
  - Multiple senders and/or receivers can **time multiplex** the link
  - Each frame can be separately addressed

- Provides manageable unit for error handling
  - Easy to determine whether something went wrong
  - And perhaps even to fix it if desired
What’s a Frame?

- Wraps payload up with some additional information
  - Header usually contains addressing information
  - Maybe includes a trailer (w/checksum—next lecture)
- Basic unit of reception
  - Link either delivers entire frame payload, or none of it
  - Typically some maximum transmission unit (MTU)
- Some link layers require absence of frames as well
  - I.e., minimum gaps between frames
Identifying Frames

- First task is to delineate frames
  - Receiver needs to know when a frame **starts** and **ends**
  - Otherwise, errors from misinterpretation of data stream

- Several different alternatives
  - Fixed length (bits) frames
  - Explicitly delimited frames
    - Length-based framing
    - Sentinel-based framing
  - Fixed duration (seconds) frames
Fixed-Length Frames

- Easy to manage for receiver
  - Well understood buffering requirements

- Introduces inefficiencies for variable length payloads
  - May waste space (padding) for small payloads
  - Larger payloads need to be fragmented across many frames
  - Very common inside switches

- Requires explicit design tradeoff
  - ATM uses 53-byte frames (cells)
To avoid overhead, we’d like variable length frames
  - Each frame declares how long it is
  - E.g. DECNet DDCMP

What’s the issue with explicit length field?
  - Must correctly read the length field (bad if corrupted)
    » Need to decode *while* receiving
  - Still need to identify the beginning…
Sentinel-based Framing

- Allow for variable length frames
- Idea: mark start/end of frame with special “marker”
  - Byte pattern, bit pattern, signal pattern
- But… must make sure marker doesn’t appear in data

- Two solutions
  - Special non-data physical-layer symbol (e.g., 00000 in 4B/5B)
    » Impact on efficiency (can’t use symbol for data) and utility of code (now can have some strings of repeated 000’s)
  - Stuffing
    » Dynamically remove marker bit patterns from data stream
    » Receiver “unstuffs” data stream to reconstruct original data
Stuffing

- Insert bytes/bits into data stream to make sure that sentinel (flag) does not appear in payload
Bit-level Stuffing

- Avoid sentinel bit pattern in payload data
  - Commonly, sentinel is bit pattern **01111110** (0x7E)
  - Invented for SDLC/HDLC, now standard pattern

- Sender: any time **five** ones appear in outgoing data, insert a zero, resulting in **01111110**

- Receiver: any time five ones appear, removes next zero
  - If there is no zero, there will either be six ones (sentinel) or
  - It declares an error condition!
  - Note bit pattern that cannot appear is **01111111** (0x7F)

- What’s the worst case for efficiency?
For next class

- Read 2.4
- Get going on Homework 1
- We’re moving classrooms starting MONDAY