Lecture 3 Overview

- Framing wrap-up
  - Sentinel-based framing
  - Clock-based framing

- Error handling through redundancy

- Hamming Distance
  - When we can detect
  - When we can correct

- Checksums
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
    » Impact on efficiency (can’t use symbol for data) of code
  - 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?

CSE 123 – Lecture 2: Layers & Framing
Byte Stuffing

- Same as bit stuffing, except at byte (character) level
  - Generally have two different flags, **STX** and **ETX**
  - Found in PPP, DDCMP, BISYNC, etc.

- Need to stuff if either appears in the payload
  - Prefix with another special character, **DLE** (data-link escape)
  - New problem: what if DLE appears in payload?

- Stuff DLE with DLE!
  - Could be as bad as 50% efficient to send all DLEs
Clock-Based Framing

- So far, we’ve based framing on what’s on the wire
  - Any bit errors may throw off our framing
  - What happens with missed delimiter? Spurious delimiter?

- An alternative is to base framing on external clock
  - Use some signal to indicate beginning of frame, and wait fixed time until frame ends
  - This is what SONET does, among others

- Significant engineering tradeoffs
  - No extra bits needed in the data stream itself, but…
  - Need tight clock synchronization between sender and receiver
SONET

- Synchronous Optical NETwork
  - Engineering goal to reduce delay and buffering

- All frames take same amount of time
  - Independent of how fast bits in frame are sent (bit rate)

- Each frame starts with special signal bits
  - Can sync clock—look for special periodic signal bits
  - No need to stuff; signal pattern is unlikely, so won’t be periodic in data
When Things Go Wrong

- Clock drift may confuse frame boundaries
  - Read the end of one frame and beginning of the next

- What happens if there are **bit errors** on channel?
  - We might misinterpret clock sync signals (sentinels) as data or vice versa

- In general, need some way to make sure we’re OK
  - Error detection—and perhaps correction
Error Detection

- Implemented at many layers
  - We’ll mainly focus on link-layer techniques today

---

CSE 123 – Lecture 3: Error Handling
Basic Idea

- The problem is data itself is not self-verifying
  - Every string of bits is potentially legitimate
  - Hence, any errors/changes in a set of bits are equally legit

- The solution is to reduce the set of potential bitstrings
  - Not every string of bits is allowable
  - Receipt of a disallowed string of bits means the original bits were garbled in transit

- Key question: which bitstrings are allowed?
Let’s start simple, and consider fixed-length bitstrings

- Reduce our discussion to $n$-bit substrings
- E.g., 7-bits at a time, or 4 bits at a time (4B/5B)
- Or even a frame at a time

We call an allowable sequence of $n$ bits a codeword

- Not all strings of $n$ bits are codewords!
- The remaining $n$-bit strings are “space” between codewords

Rephrasing previous question: how many codewords with how much space between them?
Hamming Distance

Distance between legal codewords
- Measured in terms of number of bit flips

Efficient codes are of uniform Hamming Distance
- All codewords are equidistant from their neighbors
2d+1 Hamming Distance

- Can **detect** up to 2d bit flips
  - The next codeword is always 2d+1 bit flips away
  - Any fewer is guaranteed to land in the middle

- Can **correct** up to d bit flips
  - We just move to the closest codeword
  - Unfortunately, no way to tell how many bit flips
    - E.g., 1, or (2d+1)-1?

CSE 123 – Lecture 3: Error Handling
Encoding

- We’re going to send only codewords
  - Non-codewords indicate errors to receiver

- But we *want* to send any set of strings
  - Need to embed arbitrary input into sequence of codewords
Simple Embedding: **Parity**

- **Code with Hamming Distance 2**
  - Can detect one bit flip (no correction capability)

- **Add extra bit to ensure odd(even) number of ones**
  - Code is 66% efficient (need three bits to encode two)
  - Note: Even parity is simply XOR
Simple Correction: Voting

- Simply send each bit $n$ (3 in this example) times
  - Code with Hamming Distance 3 ($d=1$)
  - Can detect 2 bit flips and correct 1
- Straightforward duplication is extremely inefficient
  - We can be much smarter about this
For Next Class

- We’ll finish error detection and talk about reliable transport on Wednesday

- Read 2.5 in P&D