Homework 2: Bit Stuffing, Error Recovery, CRCs

Due on Tuesday, October 22nd, 2024 at 2:00 pm (Week 4)

Note: Questions 4 and 5 are merely for exam practice; do not turn them in. We will also just pick one of the three mandatory questions to grade.

Question 1: PPP Byte Stuffing versus HDLC Bit Stuffing

While HDLC used bit stuffing, a much more commonly used protocol today is called PPP and uses *byte* stuffing. PPP uses the same flag **F** that HDLC does (**`01111110`**). To prevent the occurrence of the flag in the data, PPP uses an escape byte **E** (**`01111101`**) and a mask byte **M** (**`00100000`**).

When a flag byte **F** is encountered in the data, the sender replaces it with two bytes: first, the escape byte **E** followed by a second byte, **`XOR(F, M)`**. (**`XOR(A, B)`** denotes the Exclusive OR of A and B.)

Similarly, if the data contains an escape byte **E**, then the sender replaces it with two bytes: **E** followed by **`XOR(E, M)`**.

As an example, if the data is:
`01111110 00010001 01111101`

the stuffed output data is:
`01111101 01011110 00010001 01111101 01011101`

Questions

1. **Resulting Frame** (7 points)
   Suppose the sender has the following data to send:
   `01111101` **`01111101`** **`01111110`** **`01111110`**

   Show the resulting frame after stuffing and adding flags.
   ANS. The resulting frame would be: **`01111101 01011101 01111101 01011101 01111101 01011110 01111101 01011110`**

2. **Byte versus Bit Stuffing** (3 points)
   Why is byte stuffing easier for software implementations than bit stuffing?
   ANS. Byte stuffing is easier to implement at the software level as it is easier to manipulate data in bytes(supported by char and uint_8 data types), while bit stuffing would require additional operations that may not be supported without more support. Along with that, in bit stuffing, you need to ensure that the bits are correctly synchronized and aligned with the data stream. Managing bits across byte boundaries adds

complexity, especially when inserting or removing bits(requiring tracking the positions etc) while for byte stuffing the operations are more straightforward because they occur at fixed byte boundaries.

Overhead (5 points)
In HDLC, assuming all bit patterns are equally likely, there is roughly a 1 in 32 chance that the 5-bit pattern **11111** occurs in random data. This leads to roughly **3%** overhead for bit stuffing in random data. Assuming that user data is random and that each byte value is equally likely, what is the chance that byte stuffing will be done in a PPP frame?

ANS. There are 256 byte combinations, and there are two patterns that are required to be modified(E and F). Thus, there is a 2 in 256 chance or 1 in 128 chance that the bit pattern for E or F occurs in this random data. This equates to a 0.78% overhead for byte stuffing in PPP when the given sequence is completely random.

3. **Worst Case** (5 points)
What is the worst case overhead for HDLC? In other words, pick a sequence of data bits that causes HDLC to add the most stuffed bits and find the overhead. Define overhead as stuffed bits divided by total bits (of the original sequence).
ANS. The worst case scenario would be for the entire sequence to have consecutive ones-> 11111111 11111111 …. Thus, after every 5 bits, there would be a zero bit that would be added. As overhead is stuffed bits divided by total bits, the overhead would be 1/5 = 20% overhead.

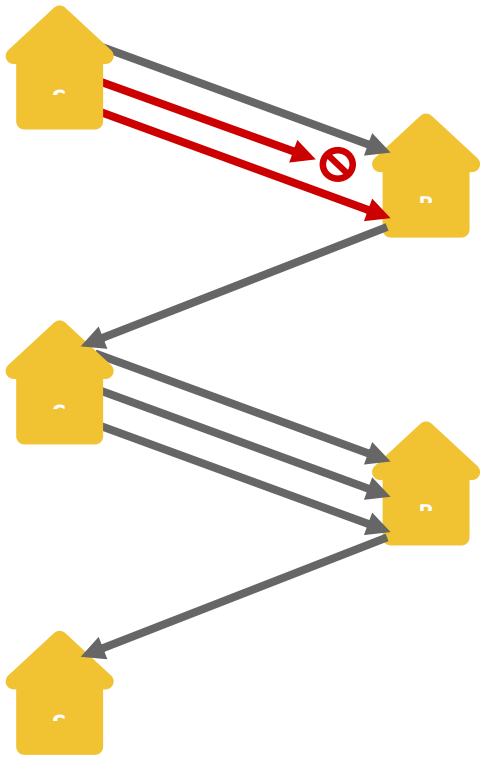
What is the worst case overhead for PPP? In each case, give the data that causes the worst case and also the worst case overhead percentage.
This would occur when the entire frame would be E's, F's or a combination of both. This would lead to a byte added after each byte encountered. An example of this would be `01111110 01111110 01111110` …… Thus, the overhead would be 1 stuffed byte/1 original byte = 100% overhead.

Question 2: Error Recovery

Peter Protocol has been consulting with an Internet Service Provider and finds that they use an unusual error recovery protocol shown in the figure below.

The protocol is very similar to Go-Back-N with numbered data packets; the key difference is that the sender does not normally send ACKs. The receiver sends a message called a **NACK** only if it detects an error in the received sequence or if it receives a so-called **STATUS** packet.

In the figure below, the sender sends off the first three data packets. The second one is lost; thus when the receiver gets the third packet, it detects an error and sends a **NACK** which contains the highest number the receiver has received in sequence. When **NACK 1** gets to the sender, the sender retransmits data packets **2** and **3**. Periodically, based on a timer, the sender transmits a **STATUS** packet. The receiver always replies to a **STATUS** packet using a **NACK**.



Questions

1. **STATUS Packets** (3 points)
   Why is the **STATUS** packet needed? What can go wrong if the sender does not send **STATUS** packets?

   Ans. The STATUS packet is required to make sure that there is still a line of communication between the two devices, and to make sure that if packets were lost during transmission and were not detected by the receiver, then the sender would not simply assume that all packets were received correctly.

Since the receiver only sends a NACK upon detecting an error or upon receiving a STATUS packet, if neither condition is met (i.e., no further packets are lost after the error or STATUS is triggered), the sender might remain unaware of the need to retransmit the lost packets. This could result in lost data going unnoticed. Also, if a NACK itself is lost, without the periodic STATUS packets-> the sender won't retransmit the missing data, leading to a stale communication where both sender and receiver think the transmission is complete but both have inconsistent data.

2. **Timers** (2 points)
   A **STATUS** packet is sent when a **STATUS** timer expires. The sender maintains the following property:
   While there remains unacknowledged data, the STATUS timer is running.

   Why does this property guarantee that any data packet given to the sender will eventually reach the receiver (as long as the link delivers most packets without errors)?
   ANS. If the data packet contains any errors, the reciever will only notify the sender if it received a STATUS packet or identifies an error in receiving the packets By sending STATUS packets periodically, the sender regularly prompts the receiver to check for missing packets. This mechanism will guarantee that any unacknowledged packet will eventually be identified, which prompts the required retransmission.

3. **Timer Conditions** (5 points)
   Under what conditions must the timer be stopped and started to maintain the property

   ANS. While there remains unacknowledged data, the STATUS timer should be running. For starting the timer, specifically, this should be done when the sender transmits a new data packet and it enters the unacknowledged window, or after receiving a NACK from the receiver, indicating that retransmission is needed. The STATUS timer should stop when all outstanding data has been acknowledged, which happens when either the receiver acknowledges (implicitly or explicitly) all packets that were sent or a NACK indicates that the receiver has received all required data.

4. **Loss Latency** (10 points)
   Consider sending a single data packet **D** that is lost. After that, no packets get lost. What is the worst-case latency before the receiver receives **D** and the sender knows the receiver has received **D**?
   ANS. The worst-case latency for receiving a lost data packet D occurs when the sender doesn't realize the packet is lost until the STATUS timer expires, which prompts a STATUS packet to the receiver. Upon receiving the STATUS packet, the receiver detects the missing packet D and sends a NACK back to the sender. The sender then retransmits D, which the receiver acknowledges, completing the process. The total latency consists of the full STATUS timer interval plus the round-trip time (RTT) for the NACK and the retransmission, making the worst-case latency approximately equal to one STATUS timer interval + 2×RTT.

Question 3: CRCs

Imagine you ask ChatGPT (or any other LLM) the following:

"Can you generate a Python script for computing the CRC of a given message using a provided generator polynomial? The function should take in 2 inputs:
1) message - string message to compute the CRC for, and
2) generator - integer representation of the generator polynomial."

ChatGPT then provides the following code as its response:

```
def str_to_bin(message):
    """Converts a message string to its binary representation."""
    return ''.join(format(ord(char), '08b') for char in message)


def bin_div(dividend, divisor):
    """Performs binary division and returns the remainder."""
    divisor_length = len(divisor)

    temp = dividend[:divisor_length]

    while len(dividend) >= divisor_length:
        temp = bin(int(temp, 2) ^ int(divisor, 2))[2:].zfill(divisor_length)
        temp = temp[1:]
        if len(dividend) > divisor_length:
            temp += dividend[divisor_length]
            dividend = dividend[1:]
        else:
            break

    return temp


def compute_crc(message, generator):
    """
    Computes the CRC of a given message using the provided generator polynomial.

    Parameters:
    message (str): The input message to calculate the CRC for.
    generator (int): The generator polynomial as an integer.

    Returns:
    str: The binary representation of the CRC remainder.
    """
    message_bin = str_to_bin(message)

    generator_bin = bin(generator)[2:]
    generator_length = len(generator_bin)

    padded_message = message_bin + '0' * (generator_length)

    crc_remainder = bin_div(padded_message, generator_bin)
```

```
    return crc_remainder


# Example usage:
message = "H"
generator = 0b1101  # Example generator polynomial (in binary)
crc = compute_crc(message, generator)
print(f"CRC Remainder: {crc}")
```

Unfortunately, ChatGPT's answer is incorrect and does not properly implement the mod 2 CRC operation described in class. IMPORTANT: We will be using the mod 2 CRC operation described in class as the correct reference implementation!

Questions

1. **Debugging** (10 points)
   Identify the problem(s) and explain how you would fix it/them.
   ANS. The issue in the bin_div function stems from incorrect handling of the mod 2 division logic. The code has problems with bit manipulation and dividend management. There's a shifting error where the line temp = temp[1:] removes the most significant bit after each XOR operation. In mod 2 division, instead of discarding bits, we should be shifting in the next bit from the dividend. Additionally, modifying the dividend by slicing (dividend = dividend[1:]) messes with the overall length, leading to incorrect loop conditions. The while loop also has an issue since it checks len(dividend) >= divisor_length, but this condition becomes unreliable as the dividend is altered within the loop, which may cause premature termination or infinite looping.

   To fix these issues, we need to implement the correct mod 2 division logic. We should write a separate function to handle bitwise XOR between two binary strings of equal length. In each iteration, if the most significant bit of the current remainder (temp) is 1, we should XOR it with the divisor; otherwise, XOR it with a string of zeros of the same length. After each XOR operation, we need to shift in the next bit from the dividend without altering the original dividend. Loop management must also be adjusted to process all bits of the dividend and update the remainder (temp) correctly in each iteration. Finally, after the loop finishes, the remainder should be the correct length, specifically divisor_length - 1 bits, as required for CRC calculations.

2. **Incorrect CRC** (5 points)
   For the example at the bottom, what incorrect CRC is currently printed (before your fixes)? Be sure to show your work.
   ANS. To determine the incorrect CRC produced by the original code, we step through the compute_crc function using the example message "H" and the generator polynomial 0b1101. First, we convert the message "H" to binary, resulting in 01001000 . Then, the binary representation of the generator polynomial 0b1101 (which is 13 in decimal) is 1101, and its length is 4. We pad the binary message with four zeros, resulting in 010010000000 . Using the bin_div function to perform binary division, we begin by extracting the first four bits of the dividend (0100) and XOR it with the divisor (1101). The XOR result is 1001, and we discard the most significant bit (MSB) to get 001 . We append the next bit of the dividend to form 0011 and continue the process. In the following iterations, we repeatedly XOR the temporary value with the divisor,

discard the MSB, and append the next bit from the dividend. However, due to the incorrect implementation, the dividend is manipulated improperly, leading to cycling through similar values and we get an incorrect remainder of 011 . Therefore, the incorrect CRC remainder produced by the original code is 011, resulting from errors in how the loop handles the dividend and discards bits incorrectly.

3. **Correct CRC** (5 points)
   For the example at the bottom, what should the correct CRC be (after your fixes)? Be sure to show your work.

   Ans. To compute the correct CRC for the given example, we first convert the message "H" into its binary form: 01001000. The generator polynomial is 1101, a 4-bit binary number, so we pad the binary message with three zeros, resulting in 01001000000 . We then perform bitwise mod-2 division of this padded message by the generator polynomial. The process involves taking the first four bits, and if the leftmost bit is 0, we shift in the next bit. If it's 1, we XOR it with the generator. This continues until the entire padded message is processed.

   Starting with 0100, since the first bit is 0, we XOR with 0000 and shift in the next bit to get 1001 . We then XOR with the generator 1101, resulting in 0100, and continue the process until all bits have been divided. After the last bit is processed, the remainder, which is the final CRC, is 0010 . This value is the correct 3-bit CRC for the message "H" using the generator polynomial 1101, ensuring error detection during transmission.

Question 4: Data Link Protocols on Synchronous Links *(Optional, Exam Practice Only)*

So far in all our Data Link protocols, we have assumed the links to be asynchronous; the delay of a frame or an **ACK** could be arbitrary.

Now we consider the case that the time taken for a message or an **ACK** is **0.5 time units**. Further senders send frames only at integer times like **0**, **1**, and **2**. When a receiver gets an error-free frame (sent at time **n**) at time **n + 0.5**, the receiver sends an **ACK** back that arrives (if successful) just before time **n + 1**. Suppose we use the standard alternating bit protocol—except that the sender also waits to send at integer times.

Questions

1. **Sequenced Sender** (10 points)
   Does the sender need to number the data frames? If your answer is yes, give a counterexample to show what goes wrong when it does not.

2. **Sequenced Receiver** (10 points)
   Does the receiver need to number the **ACK** frames? If your answer is yes, give a counterexample to show what goes wrong when it does not.

3. **Protocol Design** (5 points)
   Describe a simple protocol for the sender to initialize the receiver state after a crash.

Question 5: HDLC Framing *(Optional, Exam Practice Only)*

The HDLC protocol uses a flag **`01111110`** at the start and end of frames. In order to prevent data bits from being confused with flags, the sender stuffs a **`0`** after every 5 consecutive ones in the data. We want to understand that not all flags work, but perhaps some others do work. Maybe the HDLC flag is not the only possible one…

Questions

1. **All Ones** (5 points)
   Consider the flag **`11111111`** and a similar stuffing rule to HDLC (stuff a **`0`** after **`5`** consecutive **`1`**s). Show a counterexample to show this does not work.

2. **Correct All Ones** (5 points)
   Consider the flag **`11111111`**. Find a stuffing rule that works and argue that it is correct. What is the worst case efficiency of this rule? (Recall HDLC had a worst case efficiency of 1 in 5 bits, or 20%.)

3. **New Flag** (8 points)
   Hugh Hopeful has invented another new flag for HDLC (Hopeful Data Link Control) protocol. He uses the flag **`00111100`**. In order to prevent data bits from being confused with flags, the sender stuffs a **`1`** after receiving **`001111`**. Does this work? Justify your answer with a short proof or counterexample.

4. **New Overhead** (7 points)
   To reduce the overhead, Hugh tries to stuff a **`1`** after receiving **`0011110`**. Will this work? Justify your answer with a short proof or counterexample.