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.

2. **Byte versus Bit Stuffing** (3 points)
   Why is byte stuffing easier for software implementations than bit stuffing?

3. **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?

4. **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).

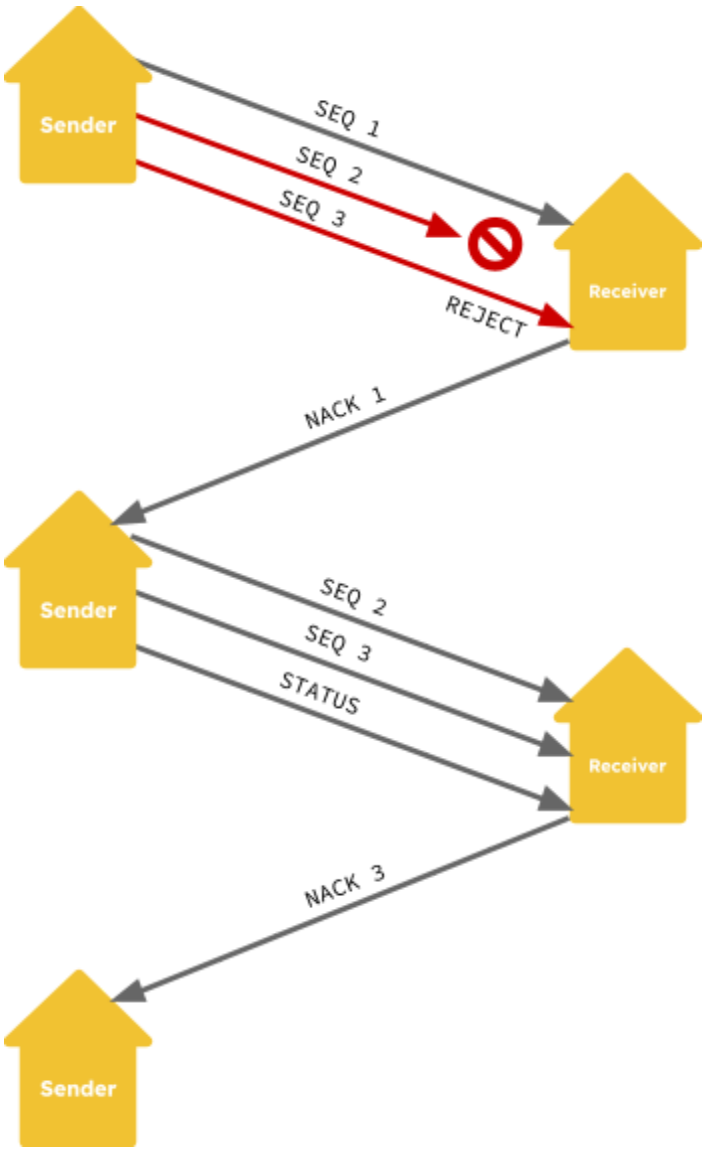
   What is the worst case overhead for PPP? In each case, give the data that causes the worst case and also the worst case overhead percentage.

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?

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)?

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

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**?

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.

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.

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.

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.