Ethan Wang
CS 118
Dis 1B
12 Oct 2022

CS 118 HW 2

1a) The solution that uses the minimal amount of padding necessary to make sure we can use Alyssa's rules and yet decode correctly is to always add a single zero byte as padding regardless of whether the original frame ends with a zero byte. For example:

① You want to send "X0"
② Pad it so it becomes "X00"
③ Encode the padded message as "2X1"
④ Add the framing characters around it
↳ Note: This is ignored in the computation of overhead

1b) If the input data contains all 0's, then the length of the coded data would be the number of digits needed to represent the length of the input data. For example:

"000" → three 0's → 3. Coded data is one digit long, to represent '3'.

In general, the coded data may only be a couple bytes longer than the original input data. These extra bytes come from the length field at the beginning. For instance:

"abcdefghij" → 10abcdefghij

- The encoded data is 2 bytes longer due to the addition of the length field.
- A similar message of length 200 would have encoded data that is 3 bytes longer due to the 3-digit length field.

It depends on the length of the message and whether there are 0's in the message, but in general the coded data may only be 2 or 3 bytes longer than the input data.

1c) The question mentions "your scheme" - I'm assuming this is referring to my answer for 1a.

- $L << 254$

Say we have a very small message: m = "A"
We would then pad it with the zero byte: m = "A0"
Then the message gets encoded as: m = "1A"
Our final encoded message (2 bytes) is twice as long as our original input (1 byte). 1/2 of the bits in the encoded message are overhead (50%).

- $L >> 254$

Say we have a very large message: m = "A0A0A0..." [pretend this repeats many times]
The message would be split into many blocks: "A0"
Each of these blocks will then be padded: "A00"
Each of these blocks will be encoded: "2A1"
The final encoded message would be: "2A12A12A1..."
Our final encoded message will have 33% more bits as our original input. 1/3 bits in the encoded message are overhead (33%)

1d)

```
def decode(msg):
    output = ""
    while (msg && msg[0] is a number):
        x = msg[0]
        temp = the x characters following msg[0]
        output += temp
        delete msg[0:x]
    iterate through output:
        restore all occurences of 0
    return output
```

[I'm sorry if anything seems off - this question was incredibly ambiguous and when I asked my TA he gave even more convoluted responses. I tried my best to decipher the question but I'm not very confident.]

2. $x^4 + x + 1 \rightarrow 10011$

A) 100101

```
          10011
      ┌──────────
10011 │ 1001010000
      │ 10011↓
      │ 00001 1
      │ 00000↓
      │    110
      │    000↓
      │     1100
      │     0000↓
      │     11000
      │     10011↓
      │     01011 0
      │      1001 1
      │      0001 0 1   ← CRC
```

CRC is 101.
Message is sent as: 100101101

B) Message + CRC 100101101
Generator + 10011
100111110

The resulting message is: 100111110.

C) This generator does detect all 1-bit errors. It can detect any burst errors up to the highest degree of the generator (4-bit, 3-bit, 2-bit, 1-bit burst errors). A 1-bit burst error is the same as a 1-bit error, thus we know the generator can detect all 1-bit errors.

D)

```
      1 1 1 0 R1
   ┌──────────
11 │ 1 0 0 1 1
   │ 1 1↓
   │   1 0
   │   1 1↓
   │     1 1
   │     1 1↓
   │       0 0 1
```

There will be a remainder of 1. This means that this generator may not catch all odd-bit errors. If the generator was divisible by x+1 (11), then it would catch all odd-bit errors.

odd bit error - odd number of bits will be flipped in the result

[odd # of bits is never divisible by x+1]

[Even # of bits in generator - guarantees divisible by x+1, can detect odd bit errors]

E) $(x^5 + 1) \cdot (x^4 + x + 1)$

$x^9 + x^6 + x^5$

$x^4 + x + 1$

$x^9 + x^6 + x^5 + x^4 + x + 1$

There are 16 polynomials we could multiply our generator by in order to get an undetected burst error of length 10. Multiplying the generator by any of the polynomials on the right would yield a polynomial that starts with x^9 and ends with 1 and has optional terms for the remaining powers between 8 and 1. There are 16 polynomials for any degree 4 CRC. This is because the number of polynomials whose highest power is x^5 and whose lowest power is 1 is the same as the number of distinct combinations for the middle terms. The middle terms are x^4, x^3, x^2, and x. The number of combinations would be given by $2^4 = 16$. Every degree 4 CRC has 16 of these polynomials, and they are the same 16 listed above.

List of Polynomials

$x^5 + 1$

$x^5 + x + 1$

$x^5 + x^2 + 1$

$x^5 + x^2 + x + 1$

$x^5 + x^3 + 1$

$x^5 + x^3 + x + 1$

$x^5 + x^3 + x^2 + 1$

$x^5 + x^3 + x^2 + x + 1$

$x^5 + x^4 + 1$

$x^5 + x^4 + x + 1$

$x^5 + x^4 + x^2 + 1$

$x^5 + x^4 + x^3 + 1$

$x^5 + x^4 + x^2 + x + 1$

$x^5 + x^4 + x^3 + x + 1$

$x^5 + x^4 + x^3 + x^2 + 1$

$x^5 + x^4 + x^3 + x^2 + x + 1$

F) There are 2^{n-2} total possible burst errors. This comes from the number of distinct combinations that can be made where the lowest and highest terms are both set. There are 2^{n-k-2} undetected burst errors. This comes from the number of polynomials that can be multiplied by the generator to create a polynomial that has a lowest degree of 1 and highest degree of n. Having a generator with degree k and following our established logic, we get 2^{n-k-2}.

$P(\text{Not Detecting ERROR}) = \frac{\#\text{ undetected}}{\text{total errors}} = \frac{2^{n-k-2}}{2^{n-2}} = \frac{1}{2^k}$, than 32 $\frac{1}{2^{32}}$ times. This is very rare!

CRC-32 fails to detect burst errors greater

3. Error Recovery

A) The STATUS packet is necessary because there may be instances where the sender is unaware that an error has occurred. An example of this is if the final packet from the sender gets lost, then the sender will not receive a Nack in return. Even worse, every single packet from the sender could get lost, and the sender would never even see a Nack since the receiver never got any of the packets. It is also possible that the Nack itself may get lost from the receiver to the sender, which would be problematic. The STATUS packet is also useful because the sender can periodically check with the receiver to see if they are properly receiving messages.

B) The sender will periodically send STATUS packets as long as there are still unacknowledged data packets. This means that the sender will keep trying to send the data packets to the receiver over and over until the receiver acknowledges that they have received the packets. Thus, any data packet given to the sender will eventually reach the receiver (as long as the link delivers most packets without errors).

C) The timer should be started if there are packets remaining in the sender's queue. If there are no packers in the sender's queue, then the timer can be stopped.

D) In the worst-case scenario, the latency would be equivalent to the length of two status timers + the length of the round-trip time. See the sketch below for a visualization of what happens between the sender and the receiver. After the events shown in the sketch, the sender's queue is empty so the timer stops and the process is complete.