

NAME: MBUA SEDRICK GOBINA KOFI

COURSE: CEF 663(NETWORK PROTOCOL DESIGN)

HOMEWORK 2

Exercise 4-1

Analysis: The code rate here is inefficient, given that we're sending 5 bytes to transmit 1 byte of actual data, so the code rate is $1/5 = 0.2$ or 20%. That's using only 20% of our bandwidth for actual data. For burst error protection, this scheme has a significant weakness. If a burst error affects multiple consecutive transmissions of the same byte, the majority vote could pick the wrong byte. For example, if bytes 2, 3, and 4 in a set of five get corrupted by a burst, the majority (3 out of 5) would be wrong, and we'd accept corrupted data. The scheme works well for random isolated errors but struggles with burst errors that span more than 2 of the 5 transmissions.

Exercise 4-2

Protocol Design:

Sender side:

1. *Send message M with sequence number*
2. *Wait for echo from receiver*
3. *Compare echoed message with original*
4. *If match: message accepted, proceed to next*
5. *If mismatch or timeout: retransmit M*

Receiver side:

1. *Receive message M*
2. *Immediately echo back exactly what was received*
3. *Wait for next message*

Exercise 4-3

Improvement Analysis: This is much more efficient than echoing the entire message. Instead of sending back the whole message, we only send back a small CRC checksum (typically 16-32 bits vs. potentially thousands of bits for the message).

The sender computes what the CRC *should* be for the message it sent, then compares it with the received CRC. If they match, the receiver got the message correctly. This drastically reduces acknowledgment overhead while maintaining error detection capability. The code rate improves significantly because we're not sending redundant copies of the entire message on the return path.

Exercise 4-4

Answer: Forward error control

Reasoning: English grammar rules let us detect errors without needing to "ask" for retransmission. When we see "the dogs runs," we immediately know it's wrong based on subject-verb agreement rules built into the language itself. We don't need feedback from the listener to know we made an error, hence Forward error control.

Exercise 4-5

The polynomial in question is $x^6 + x^4 + x + 1$, which is of degree 6.

(a) The checksum is 6 bits (The degree of the polynomial)

(b) *Original data = message – checksum.*

The message is 101011000110 (12 bits total)

Original data = first 6 bits = 101011

Checksum = last 6 bits = 000110

(c) Yes, there were transmission errors because when we divide the received message (101011000110) by the generator polynomial (1010011), we don't get a remainder of zero. In a correctly transmitted message protected by CRC, the entire received string should be divisible by the generator polynomial with zero remainder.

Exercise 4-6

Circumstances favoring FEC with code rate 0.1:

1. Very high round-trip delays - satellite links, deep space communication where RTT might be seconds or minutes
2. High error rates - if errors are frequent, constant retransmissions become costly
3. Real-time applications - voice, video streaming where retransmission isn't acceptable

4. Simplex channels - where return path isn't available or is very expensive
5. Multicast scenarios - one sender, many receivers (feedback becomes impractical)

Even though we're only getting 10% throughput, if each retransmission takes seconds and errors are common, the effective throughput could be even lower.

Exercise 4-7

Method: Take n messages and arrange them in a matrix where each row is a message. Apply error correction code to each *column* instead of each row. This spreads bits from one codeword across n different messages.

Example: Say we have 4 messages ($n=4$), each 8 bits:

M1: 10110101

M2: 11001100

M3: 10101010

M4: 01110011

Read column-wise and add error correction to each column. Now if a burst error hits M2 completely, it only corrupts 1 bit from each of the 8 codewords. If each code corrects 1 error, all data recovers.

This protects against bursts up to k bits by ensuring bits from the same codeword are separated by at least k positions in the transmitted stream.

Problem 4-8: Catching Even-Numbered Errors

Observation: Polynomials with $(x+1)$ catch odd numbers of errors because they detect when an odd number of bits flip.

Proposed Method: Transmit twice - once normal, once with a deliberate single bit flipped. If even errors occur in first transmission, it becomes odd errors. If odd errors occur, the second transmission (which started with odd) experiences even total errors.

Problem: This doubles transmission overhead (code rate becomes 0.5 at best), and the logic gets complicated. A better approach is simply using a polynomial that detects both odd and even errors, or using a longer CRC that catches virtually all error patterns regardless of parity.