Data Communication (2/2023) Class Project

Instructor: Asst. Prof. Boonsit Yimwadsana, Assoc. Prof. Vasaka Visoottiviseth

Submission deadline: Fri 12 Apr 2024, 11:59 pm

This assignment can be done in a group of one or two students. (less number of students may receive higher score due to increased amount of workload distribution)

Scoring: 100 points (each problem is 25 points)

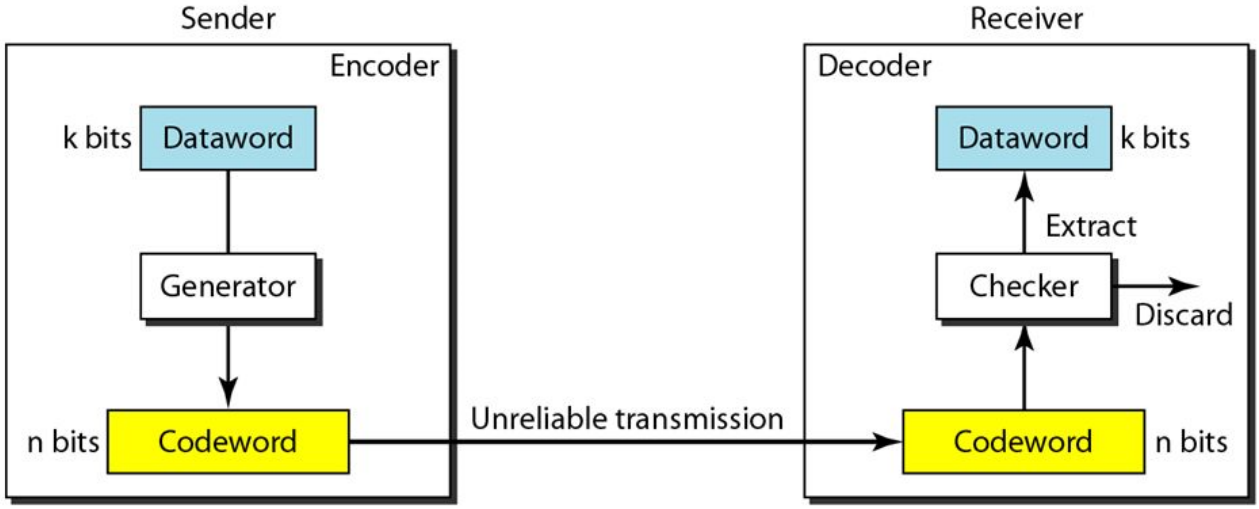
Criteria: Correctness of output (35%), Correctness of methods (35%), Presentation (30%)

Programming Language: C, Java, Python

- **Project Presentation**
- Create a 10-minute video presentation that includes each presenter's face
  - Show how to run the program
  - Explain the code
  - You may create a team and use MS teams to record the video and open the camera.
  - The deadline of video submission is Sat 20 April 2024, 23:59
  - Video Size must be less than 50 MB; otherwise upload the Video Link in your **Shared** drive; and make sure that you give us the view access rights.
  - Name your file as Video-SecX-StudentID-StudentID.mp4
    - For example, Video-Sec1-6588001-6588002.mp4

Problem:

In data communication, error can happen during the transmission of data from sender to receiver over unreliable medium.



In order for the receiver to check for error, redundancy bits are added to the dataword to help the receiver verifies the validity of the received data (codeword).

Use the knowledge you learnt in class, implement the following programs or functions and write a demo code showing how all these functions can be used for error control.

1. **Error detection using Parity Checking**:

   Generator: codeword = **2d*_parity_gen***(datawords, word_size, num_words)

   Implement a function that generates two-dimensional even parity codeword to an array of input dataword.

   Input:
   1. Datawords: A two-dimensional array of bits. Each bit-string (dataword) has size equal to word_size (word_size = 8 bits). The 2D datawords array must consists of num_words datawords (num_words = 4).
   2. word_size: Each bit-string (dataword) has size equal to word_size (word_size = 8 bits).
   3. num_words: The 2D datawords array must consists of num_words datawords (num_words = 4).

   Output:
   1. An array of codeword using 2D even parity bit checking

   Behavior:

   This function calculates the redundancy bit(s) for the array of datawords and output an array of codeword.

   - The input must be an array of datawords. The size of the two-dimensional block for dataword must be word_size x num_words (number of rows = num_words and number of columns = word_size). If some dataword has size < word_size, the dataword must be appended (padded) with bit '0' until the size the dataword is word_size.

   Testing:

   Check if this function encodes the dataword correctly for at least 10 trials for correct case and 10 trials for incorrect case using random two-dimensional arrays.

   Checker: (syndrome, error_loc) = 2d*_parity_check*(codeword)
   Input:
   1. codeword to check

   Output:
   1. Syndrome that shows the validity of codeword
   2. The location of error

   Behavior:

   This function check whether the codeword received has error or not.

If an error is detected, assume a single bit error, determine the location of the error in the received codeword.

Testing:

Check if this function checks whether the codeword is received correctly for at least 10 trials for correct case and 10 trials for incorrect case using random two-dimensional arrays. For the incorrect case, make sure that the errors are single-bit error so that the error location can be determined.

2. CRC

Generator: codeword = *CRC_gen*(dataword, word_size, CRC-type)

Implement a function that generates CRC codeword for an input dataword.

Input:

1. Dataword of size up to word_size
2. Maximum size of each dataword (word_size) where word_size = 48
3. CRC-type (string e.g, CRC-8, CRC-16, CRC-32, …)

Output:

1. A codeword based on CRC

Behavior:

This function calculates the redundancy bit(s) for the dataword and output the codeword for CRC. Hint: you have to create a modulo-2 division function. Use the CRC divisor as shown in the table below.

Testing:

Check if this function encodes the dataword correctly for at least 10 samples.

CRC Method	Generator Polynomial	Number of Bits
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	32
CRC-24	$x^{24}+x^{23}+x^{14}+x^{12}+x^8+1$	24
CRC-16	$x^{16}+x^{15}+x^2+1$	16
Reversed CRC-16	$x^{16}+x^{14}+x+1$	16
CRC-8	$x^8+x^7+x^6+x^4+x^2+1$	8
CRC-4	$x^4+x^3+x^2+x+1$	4

Checker: syndrome = *CRC_check*(codeword, CRC-type)

Input:

1. codeword
2. CRC-type (string e.g, CRC-4, CRC-8, CRC-16, CRC-24, CRC-32, …)

Output:

1. Syndrome of codeword

Behavior

This function verifies the CRC codeword.

Testing:

1) Check if this function verifies the codeword correctly for at least 10 samples.
2) It is well known that if the degree of the generator is k, it can detect all burst errors up to length k and it can detect errors with length > k with high probability. Show at least two erroneous codewords that CRC-4 fails to detect error.

3. Checksum

Generator: codeword = *Checksum_gen*(file1_name)

Implement a function that generates a checksum codeword for an input file.
Assume Maximum size of each dataword (word_size) is equal to 8 bits.

Input:

1. Filename of the 1st file with the ASCII text of any length.

Example of contents in File1.txt

Checksum is an error-detecting technique.

Output:

1. A codeword based on Checksum (could be an array of data)

Behavior:

This function will open the file from the given filename, read the data from the file, chop them into the word_size (8 bits; 1 alphabet) and calculate Checksum.

Checker: validity = *Checksum_check*(file2_name, checksum_value)
Input:

1. The 2nd file name (this file is a copied version of the 1st file but modified some text)
2. The value of checksum got from the 1st file

Example of contents in File2.txt

Checksum are an error-detecting technique.

Output:

1. Validity of codeword (PASS=0 or FAIL= -1)

Behavior

This function verifies the Checksum codeword by calculate the sum of the input data similar to the generator, but add the checksum at the end. If the total sum is equal to 11111111 (ZERO after 1-complement), then the data is correct.

4. Hamming code

Generator: codeword = *Hamming_gen*(input_alphabet)

Implement a function that generates a codeword for an input alphabet (dataword), which is an alphabet letter from a-z and A-Z.

Input:

1. Dataword, which is an alphabet letter

Output:

1. A codeword based on Hamming code

Behavior

This function translates the input dataword from ASCII text to 8 bits of binary. After removing the most significant bit (most left), there will be 7 bits. From these 7 bits, calculates the redundancy bit(s) for the dataword and output the 11-bit codeword for Hamming code.

Checker: error_pos = *Hamming_check*(codeword)
Input:

1. A 11-bit codeword as such 1100100110.

For example,
Correct Codeword (11 bits) of 'a' = 1100000110
Then, test with a wrong Codeword (11 bits) = 1100100110

Output:

1. Position of error (in case of a single bit error, but return 0 if there is no error found)

Behavior

This function verifies the Hamming codeword and report the location of error for the case of single bit error.

Deliverables:

1. Names and IDs of group member
2. Programming source codes
   2.1. Clean and easy-to-understand source code with proper indentation and comments with get better scores.
   2.2. The code must work.
3. Documents (PDF) containing the following content
   3.1. How to run your programs
   3.2. Screenshots of outputs from running your programs (testing results)
4. Penalty
   4.1. Copy codes from others and the Internet

Please zip your source codes and PDF report in one file and name it as ITCS323-2023-Sec*?*-*StudentID1-StudentID2*.zip
For example, ITCS323-2023-Sec1-6588001-6588002.zip