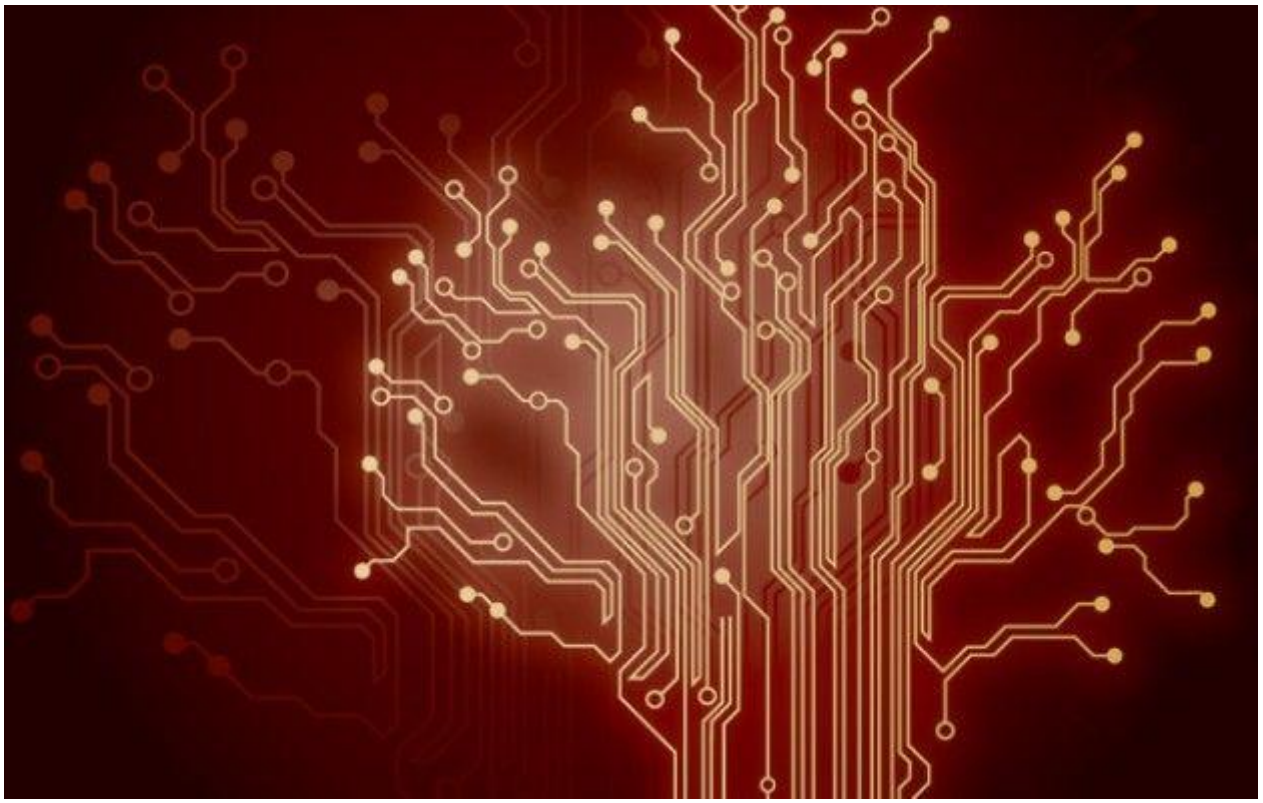
NETWORK LAB REPORT

CO1: Design and implement error detection techniques within a simulated network environment



Atanu Ghosh

BCSE-III (2019-2023) 5th sem, Section: A-1,

Roll: 001910501005, Date: 05/08/2021

ASSIGNMENT-1

Design and implement an error detection module

PROBLEM STATEMENT

Design and implement an error detection module which has four schemes namely LRC, VRC, Checksum and CRC. Please note that you may need to use these schemes separately for other applications (assignments). You can write the program in any language. The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the data frame (decide the size of the frame) from the input. Based on the schemes, codewords will be prepared. Sender will send the codeword to the Receiver. Receiver will extract the dataword from the codeword and show if there is any error detected. Test the same program to produce a PASS/FAIL result for the following cases.

(a) Error is detected by all four schemes. Use a suitable CRC polynomial (list is given).

(b) Error is detected by checksum but not by CRC.

(c) Error is detected by VRC but not by CRC.

[Note: Inject error in random positions in the input data frame. Write a separate method for that.]

DESIGN

This project is implemented in the Python 3 programming language (Tested with CPython 3.8.10). A mixture of procedural and multi-threaded techniques along with inter-process communication using TCP sockets (provided by the runtime system) has been used.

There are 3 principal components of the System, as shown in Fig. 1 :

1. SenderProcess: Generates untainted data by reading content from the input file and slices it into packets of data of a given size and then by 4 different encoding schemes (VRC, LRC, CheckSum, and CRC) produces code-words (which are tainted later by **ChannelProcess** to generate tainted codewords) and sends them to **ReceiverProcess** via `socket` connection for error checking. The process of dataword generation, encoding schemes, codeword generation, data mutation (error-injection by Channel), and displaying error-detection results are all managed by **SenderProcess**.

2. ChannelProcess: Solely responsible for injecting errors in the untainted codewords generated by **SenderProcess**. It does NOT establish the connection between Sender and Receiver, rather the untainted packets of codewords are passed through it which gets tainted (error injection is done by flipping a random number of bits ranging from 1 to length-of-codeword, at random positions of the codewords). Since the number of errors is randomly decided, bit and burst errors are introduced randomly in the data-streams / codewords.

N.B. The working procedure of **ChannelProcess** is integrated into the **SenderProcess** itself, i.e. both of these processes work as an integrated system to produce datawords and codewords, inject errors and display results (generated by **ReceiverProcess**) of error detection, and generate report files (.txt files).

3. ReceiverProcess: Receives the tainted codewords through the Channel. Checks for errors against the required error checking algorithms and sends the confirmation message(s) back to **SenderProcess** through a `socket` connection. The receiver behaves like a multi-threaded server to which more than one sender can connect and send codewords for error-checking.

SCHEMATIC DIAGRAM (as a part of DESIGN)

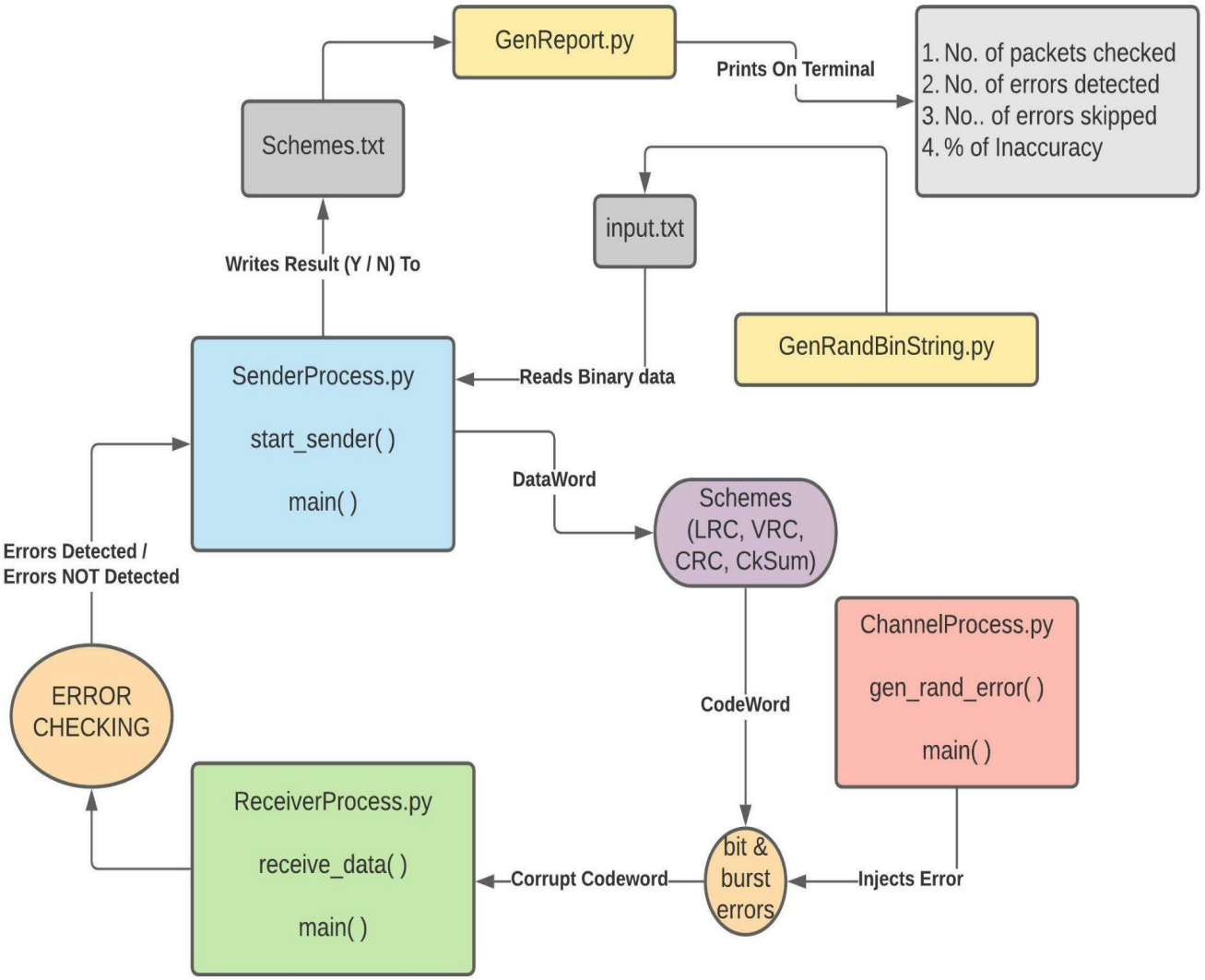


Fig. 1: Schematic Diagram to demonstrate flow-control of the whole error-detection process

IMPLEMENTATION

A. Packet Structure

CodeWord for VRC Encoding :

DataWord (32-bits) VRC (1-bit)

CodeWord Size = 33-bits

CodeWord for LRC Encoding :

DataWord (32-bits) LRC (8-bits)

CodeWord Size = 40-bits

CodeWord for CheckSum Encoding :

DataWord (32-bits) CheckSum (8-bits)

CodeWord Size = 40-bits

CodeWord for CRC Encoding :

DataWord (32-bits) CRC (8-bits or 16-bits)

CodeWord Size = 40-bits (CRC-8) or 48-bits (CRC-16)

- Length of actual data = 32,00,000 bits = 3.2 Megabits
- Number of packets (each packet containing one dataword) = 1,00,000
- Dataword size for each scheme is = **32-bits**
- **VRC is 1-bit** and is calculated from the cumulative even parity of 32-bits.
- **LRC is 8-bits** and is the column-wise even parity of 32 bits divided into 4 octets.
- **CheckSum is 8-bits** and is the 1's complement of the least significant 8 bits of the sum of 4 octets.
- **CRC is 8-bits for CRC-8 and 16-bits for CRC-16** and is the modulo-two-division of the data by a given polynomial.

B. Algorithm for SenderProcess

```
connection = socket.socket()                                    # create socket
connection.connect((host, port))                               # connect via socket

with open(input_file) as text_file:
      data = text_file.read()                          # read data from input file

number_of_datawords = length_of_data / length_of_dataword
for i in range(number_of_datawords):
      dataword = generate_dataword(data)                       # create dataword
      codeword = generate_codeword(dataword)                   # create codeword
      connection.sendall(codeword)                                # send request

receiver_response = connection.recv()                          # receive response

with open(output_file) as result:
      result.write(receiver_response)          # write response to output_file

connection.close()                                      # terminate the connection
```

C. Algorithm for ChannelProcess

```
channel.import()                                          # allocate the channel

channel.catch_each(codeword)                                  # read codeword

for each codeword caught:
      corruption = inject_error(codeword)                          # push error
      channel.add_to_datastream(corruption)                     # add to stream
```

D. Algorithm for ReceiverProcess

```
connection = socket.socket()                                   # create socket

connection.bind((host, port))                          # bind to unique address

connection.listen(1)                                      # listen to sender

threading.__init__()                                   # initialise threads

while number_of_threads <= max_limit:
      sender_request = connection.recv()                       # receive requests
      receiver_response = sender_request.has_error() # check if corrupt
      connection.sendall(receiver_response)          # send response to sender

connection.close()                                    # terminate the connection
```

E. SourceCode Structure

The SourceCode Folder contains three sub-folders - **./packages**, **./main** and **./textfiles**.

./textfiles folder contains .txt files of input data and output results.

./packages contains -

1. **VRC.py** contains -
    - **gen_VRC(data: str) -> str :** Takes string of 1-s and 0-s as input and returns even parity based on number of 1-s.
    - **main() :** To test the functionality of **gen_VRC(data)**.
2. **LRC.py** contains -
    - **gen_VRC(data: str) -> str:** Takes a string of 1-s and 0-s as input and returns even parity based on the number of 1-s.
    - **gen_LRC(data: str) -> str:** Takes a string of 1-s and 0-s as input, slices it into packets of equal length, and returns 2D even parity of digits at same indices of packets.
    - **main() :** To test the functionality of **gen_LRC(data)**.
3. **CheckSum.py** contains -
    - **binary_sum() :** Takes strings of 1-s and 0-s as input and returns binary sum of inputs read as binary numbers.
    - **gen_CheckSum(data: str, k: int) -> str :** Takes string of 1-s and 0-s as input and returns checksum as a string.
    - **main() :** To test the functionality of **gen_CheckSum(data)**.
4. **CRC.py** contains -
    - **xor(x: str, y: str) -> str :** Returns XOR of x and Y.
    - **mod2div(dividend: str, divisor: str) -> str :** Returns modulo-two-division of dividend and divisor.
    - **gen_CRC(data: str, key: str) -> str :** Returns CRC of data based on the key (CRC-Polynomial).
5. **__init__.py** contains -

    - Nothing.

    -Kept for indicating that the **./package** folder is a module for import.

./main contains -

1. **GenRandBinString.py** contains -
    - **gen_rand_string(size:** `int`**) ->** `str` **:** Returns a random string of 1-s and 0-s of given size.
    - **main() :** To test the functionality of **gen_rand_string(size)**.
2. **SenderProcess.py** contains -
    - **start_sender() :** Reads data, creates datawords, creates codewords, mixes erroneous strings into data stream, sends erroneous codewords to the receiver, captures receiver response and prints them into specific files.
    - **main() :** To test the functionality of **start_sender()**.
3. **ChannelProcess.py** contains -
    - **gen_rand_error(data:** `str`**, count:** `int`**) ->** `str` **:** Takes strings of 1-s and 0-s as input, injects specified number of errors at random positions of the string and returns erroneous string.
    - **main() :** To test the functionality of **gen_rand_error(data, count)**.
4. **ReceiverProcess.py** contains -
    - **receive_data():** Receives erroneous codewords from the receiver, tries to detect whether the error is present or not based on 4 different schemes, and sends the report to the sender.
    - **sender_thread(connection:** `socket.socket`**) :** Enables support for multiple users to connect at the same time.
    - **main() :** To test the functionality of **receive_data()**.
5. **GenReport.py** -

    Parses through the file specified by the user and keeps track of the count of errors detected and calculates %-accuracy of detection of errors.

TEST CASES

Erroneous codewords received by the Receiver and the error detection results are included in the textfiles (VRC.txt, LRC.txt, CheckSum.txt, CRC.txt).

A few test cases where :

(a) Error is detected by all four schemes :

```
   (original 32 bit codeword | corrupt 32 bit codeword)

1) 01000011100101101010001001100100 | 01000001010011101110011001001001
2) 00111011011111100000111101010111 | 00010011011011100000111010010110
```

(b) Error is detected by checksum but not by CRC :

```
   (original 32 bit codeword | corrupt 32 bit codeword)

1) 10000011001100001001010101011111 | 10000011001100010001010101011010
2) 00011110111011011010100000111001 | 00011110111011000010100000111100
```

(c) Error is detected by VRC but not by CRC :

```
   (original 32 bit codeword | corrupt 32 bit codeword)

1) 11000001111000000110100011110100 | 00000111011000000110100011110100
2) 10011111101001110110110000001000 | 10011111101001110110110110000101
```

A test cases list (testcases.txt) is provided along with the source code.

RESULTS

RUN - 1(a) (Size of each data packet = **32 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	100000	48671	48.671 %
LRC	100000	397	0.397 %
CheckSum	100000	6820	6.820 %
CRC-8	100000	336	0.336 %

RUN - 1(b) (Size of each data packet = **32 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	100000	48637	48.637 %
LRC	100000	358	0.358 %
CheckSum	100000	6805	6.805 %
CRC-16	100000	4	0.004 %

RUN - 2(a) (Size of each data packet = **64 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	50000	24822	49.644 %
LRC	50000	192	0.384 %
CheckSum	50000	814	1.628 %
CRC-8	50000	193	0.386 %

RUN - 2(b) (Size of each data packet = **64 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	50000	24532	49.064 %
LRC	50000	172	0.344 %
CheckSum	50000	799	1.598 %
CRC-16	50000	2	0.004 %

RUN - 3(a) (Size of each data packet = **128 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	25000	12329	49.316 %
LRC	25000	112	0.448 %
CheckSum	25000	176	0.704 %
CRC-8	25000	78	0.312 %

RUN - 3(b) (Size of each data packet = **128 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	25000	12371	49.484 %
LRC	25000	105	0.420 %
CheckSum	25000	183	0.732 %
CRC-16	25000	1	0.004 %

RUN - 4(a) (Size of each data packet = **256 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	12500	6255	50.040 %
LRC	12500	52	0.416 %
CheckSum	12500	46	0.368 %
CRC-8	12500	52	0.416 %

RUN - 4(b) (Size of each data packet = **256 bits**)

Detecting Scheme	Total No. of Packets	No. of Packet Missed	% of Inaccuracy
VRC	12500	6245	49.96 %
LRC	12500	68	0.544 %
CheckSum	12500	43	0.344 %
CRC-8	12500	0	0.000 %

ANALYSIS

% -inaccuracy is calculated by the following formula =

```
(NumOfErrorNotDetected) / (NumOfErrorneousPacketsSent) * 100
```

PERCENTAGE OF INACCURACY / UNDETECTED ERRORS

RUNS	VRC	LRC	CheckSum	CRC-8	CRC-16
1(a)	48.671 %	0.397 %	6.820 %	0.336 %	-
2(a)	49.644 %	0.384 %	1.628 %	0.386 %	-
3(a)	49.316 %	0.448 %	0.704 %	0.312 %	-
4(a)	50.040 %	0.416 %	0.368 %	0.416 %	-
1(b)	48.637 %	0.358 %	6.805 %	-	0.004 %
2(b)	49.064 %	0.344 %	1.598 %	-	0.004 %
3(b)	49.484 %	0.420 %	0.732 %	-	0.004 %
4(b)	49.96 %	0.544 %	0.344 %	-	0.000 %
Average : -	**49.349 %**	**0.414 %**	**2.370 %**	**0.363 %**	**0.003 %**

CRC-32 showed no errors over Random Mutations.

- Average error percentage for VRC = **49.349** %
- Average error percentage for LRC = **0.414 %**
- Average error percentage for CheckSum (8 bit) = **2.370 %**
- Average error percentage for CRC (8 bit) = **0.363 %**
- Average error percentage for CRC (16 bit) = **0.003 %**

PLOT showing % Inaccuracy of different detection algorithms/schemes

CONCLUSION

From the %-inaccuracy table of 4 schemes (5 considering CRC-16 also) we can see that **VRC is the least accurate** scheme of all of them, skipping almost 49.349 out of 100 erroneous codewords. It proves that VRC is extremely inefficient to burst errors caused while transferring data packets. Whereas **CRC-16 is the most sensitive and accurate against erroneous data**, extremely efficient at detecting both bits and burst errors, skipping a mere 0.003 out of 100 erroneous data packets.

COMMENTS

There are different ways to detect errors in the data link layer. But not all methods of error detection can detect error accurately and effectively. Every method has its own specialty, advantage, and mechanism to detect the error. Parity check (VRC) is simple and can detect all single-bit errors. CRC has a very good performance in detecting single-bit errors, double errors, an odd number of errors, and burst errors while CheckSum is not efficient as the CRC in error detection when the two words are incremented with the same amount, the two errors cannot be detected because the sum and checksum remain the same.