Two Dimension Parity Check for Error Detection

Devin Trejo

devin.trejo@temple.edu

Date: 3/11/2016

I. Summary

Lab 3 introduces two-dimensional parity error checking for an original 30 byte transmission between a client computer and a PIC32 server. The two-dimensional parity checking code allows us to encode our original message with redundant bits to detect errors in a received message. The original message is arranged in a grid and the parity bit is calculated across the rows and down the columns. Our original 30 byte string increases in size to a 35 byte string that contains the parity information. We demonstrate that our two-dimensional decoder is capable of detecting and correcting one bit errors from the client received message.

I. Introduction

Two-dimensional parity checkers are capable of correcting errors up to one bit in a received encoded message. To construct our encoded message we lay our information into a two dimensional gird. We ensure that the bits across our rows and down the column of our message bit stream contain an even number of bits.





Figure 2: Even-Parity Encoded Message [1, p. 192]

If a single error is introduced by our transmission medium we can correct it by detecting what column the errors is contained in, then cross reference that column with a row that also does not contain even parity. We now have the row and column of the infringing bit and flip it to maintain the even encoding scheme. After we have correct the errors we can flatten and retrieve the original message from the encoded message.

There are limitations to the two-dimensional error correction. We say that at best we can correct up to one error in our received message. We can correct a higher number of bits as long as the errors do no occur in the same row and column within our encoded message; creating a square. If they do occur in this shape then it is not possible to detect these errors and thus we cannot correct them.



II. Discussion

2D Even Parity Implementation

The string 30 byte string I chose to encode using the even parity encoder is stored into myStr.

Char *mystr="Devin Trejo test string for EE";

Next we pass the string into our "evenParityEncoder()" which will take our 30 byte input string and encode it into a transmit buffer of length 35 bytes. The extra 5 bytes are the parity bits for our 30 byte string.

Our encoder works by breaking our input string into 5 byte chunks. Since our string consists of the standard ASCII characters, our input chars MSB will always be zero. To take advantage of all bits in the sent message we will encode a parity bit into our input chars. To do so we shift our input chars left one bit. We now check our char for even bit parity. If the shifted char has an even number of bits we leave the shifted LSB as zero. If the shifted char. The parity bits are highlighted in yellow below.

| ASCII | | ASCII BINARY (No Shift) | | | | | | | ASCII BINARY (Shifted & Parity) | | | | | | | |
|-----------|---|-------------------------|---|---|---|---|---|---|---------------------------------|---|---|---|---|---|---|---|
| D | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| е | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| v | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| i | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| n | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| ("SPACE") | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

| Tabla | 4. | Char | E | David | Deviter | Calavilation | (First | E D. | 400) |
|---------------|----|------|------|-------|---------|--------------|---------|------|------|
| <i>i able</i> | 1. | Char | Even | ROW | Parity | Calculation | (First) | э ву | nes) |

After finding the parity bit for each shifted char we now need to find the row even parity bit. The row even parity bit is calculated by looking at our 5 byte chunk of shifted chars and computing even parity down the columns. The Even Column Parity for our first 5 bytes is shown below and the corresponding code for this process can be seen in Code Snippet 2.

| ASCII | AS | CII B | INA | RY (S | hifte | ed & | ASCII Hex (Shifted & Parity) | | |
|--------------------|----|-------|-----|-------|-------|------|------------------------------|---|------|
| D | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0x88 |
| е | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Охса |
| v | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | Oxed |
| i | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0xd2 |
| n | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | Oxdd |
| ("SPACE") | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0x41 |
| Even Column Parity | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0xe1 |

Table 2: Even Column Parity Calculation

We repeat this process for the rest of the 25 bytes in our input string. Our resulting encoded transmit buffer for our original string becomes:

Table 3: 2D Even Parity Encoded Transmit Buffer

| Group | 7 Byte Group |
|-------|----------------------|
| 1 | 88 ca ed d2 dd 41 e1 |
| 2 | a9 e4 ca d4 de 41 cc |
| 3 | e8 ca e7 e8 41 e7 8b |
| 4 | e8 e4 d2 dd cf 41 8d |
| 5 | cc de e4 41 8b 8b b7 |

We also created a corresponding "evenParityDecoder()". The decoder reverses the steps of the encoder by first looking at the parity down the columns of our 7 byte groups. Any column that has even parity we conclude to be correct. If all columns have even parity we can check our next group. If one column does not have even parity we loop back across the chars in our group (excluding the even column parity char) to look for the char that does not have even parity. When we find the infringing row we now know the row and column of the problem bit. We flip the incorrect bit by XORing it with 0x01.

2D Even Parity Demonstration - Wireshark Analysis

To test our implementation we use the provided Visual Basic application which will receive the transmit buffer, insert a random error in the payload, and send it back to our PIC32 server.

| ECE4532 Client2 v16 B | _ | | × |
|---|-----|-------|---|
| Server IP 192.168.2.105 Port 6653 | Cor | nnect | |
| Control Reset Transfer Transmit | | lose | |
| Status Transfer 35 Í Ó Ý | | ~ | |
| | | | |

Figure 5: Visual Basic Even Parity Application

We also use Wireshark to analyze the payload as it is sent between the client and server. To begin we simply want to ensure our server is sending the correct encoded message to our client.

| *Ethernet | | | | | | - | | × |
|--|--------------------------------------|------------------------------|---------------|-------------------|---------------------------------------|--------------|------------------|---------|
| <u>F</u> ile <u>E</u> dit <u>V</u> iew <u>G</u> o <u>C</u> apture <u>A</u> nalyze <u>S</u> | tatistics Telephon <u>y W</u> ireles | is <u>T</u> ools <u>H</u> el | p | | | | | |
| | 👳 🗿 👢 🗮 📕 🕀 😡 | Q. III | | | | | | |
| | | | | | 5 | Evn | ression | + |
| Na Tina Tina Causa | Destination | Destand | Longth Tafa | | <u> </u> | | | |
| No. Time Time Source | 100 169 0 105 | TCD | | S CCER FOUNT | 1 Soc-0 Win-9102 | Lon-Q MEE-1 | 1460 LI | 5-25 |
| 6 22 0 192.168.2.102 | 192.168.2.102 | TCP | 60 6653 | → 43755 [SYN | , ACK] Sea=0 Ack: | =1 Win=512 | en=0 | MSS= |
| 7 22 0 192.168.2.102 | 192.168.2.105 | TCP | 54 43755 | → 6653 [ACK | Seq=1 Ack=1 Wi | n=64240 Len: | =0 | 1.55 11 |
| 9 22 0 192.168.2.102 | 192.168.2.105 | TCP | 57 [TCP | segment of a | reassembled PDU |] | | |
| 10 22 0 192.168.2.105 | 192.168.2.102 | TCP | 60 6653 | → 43755 [ACK] |] Seq=1 Ack=4 Wi | n=512 Len=0 | | |
| 11 22 0 192.168.2.102 | 192.168.2.105 | TCP | 57 [TCP | segment of a | reassembled PDU |] | | |
| 12 22 0 192.168.2.105 | 192.168.2.102 | TCP | 60 6653 | → 43755 [ACK] |] Seq=1 Ack=7 Wi | n=512 Len=0 | | |
| 13 22 0 192.168.2.105 | 192.168.2.102 | TCP | 89 [TCP | segment of a | reassembled PDU |] | | |
| | 192.168.2.105 | TCP | 54 43/55 | → 6653 [ACK | Seq=7 Ack=36 W | in=64205 Ler | 1=0 | 1 |
| | 192.168.2.102 | тср | 50 [TCP | Keep-Alive A | 2003 → 43755 [PSI -V1 43755 → 6653 | ACK] Seq | =35 AC 7 Ack= | K=7 |
| | | | | | | | | |
| [Stream index: 0] | ▲ 000 | 0 94 de 80 | 6c 23 88 00 0 | 04 a3 00 00 | 02 08 00 45 00 | 1# | I | Е. |
| [TCP Segment Len: 35] | 001 | 00 4b 00 | 62 00 00 64 0 | 06 d0 2b c0 | a8 02 69 c0 a8 | .K.bd | +i | |
| Sequence number: 1 (relative | sequence numbe 002 003 | 0 02 00 7e | 31 00 00 88 | ca ed d2 dd | 41 e1 a9 e4 ca | ~1 | | |
| [Next sequence number: 36 (re. | lative sequenc 004 | 0 d4 de 41 | cc e8 ca e7 e | e8 41 <u>e78b</u> | e8 e4 d2 dd cf | | · · · · · · | |
| Header Length: 20 bytes | 005 ack numb | 0 41 8d cc | de e4 41 8b 8 | 8b b7 | | AA | | |
| > Flags: 0x018 (PSH, ACK) | | | | | | | | |
| Window size value: 512 | | | | | | | | |
| [Calculated window size: 512] | | | | | | | | |
| [Window size scaling factor: -2 | (no window sca… | | | | | | | |
| > Checksum: 0x7e31 [validation dis | abled] | | | | | | | |
| Urgent pointer: 0 | | | | | | | | |
| > [SEQ/ACK analysis] | | | | | | | | |
| ICP segment data (35 bytes) | ~ | | | | | | | |
| A data segment used in reassembly of a low | er-level protocol (tcp.segment_da | ita), 35 bytes | | Packets: 18 · Di | splayed: 11 (61.1%) | Pro | file: Defi | ault |

Figure 6: Wireshark Encoded Payload Analysis

We first see the three-way TCP handshake between client and server as connection is establish. To ensure our PIC32 board is setup correctly we use the VB application to RESET the server. Next we TRANSFER our transmitbuffer to our client. Analyzing the payload we can see the correct encoded message is being sent when compared to the computed expected encoded message shown in Table 3.

Next, we can TRANSMIT a message (with possible errors) back to our server and use the 2D even parity checks to look for errors. If an error is detected we set the PIC32's LED1 (red LED) high. We will look at the Wireshark payload of three cases.

- 1. No Error
- 2. Error in Original String
- 3. Error in Parity Location

Case 1 – No Error

The control case is where there are no errors introduced by the VB client. For this scenario, the even parity decoder should not flip any bits since the message was never corrupted. When the VB application transmits a message back to the server with no errors it has a status of "Transmit/No Error".

| 5. ECE4532 Client2 v16 B | _ | | × |
|-----------------------------------|-----|------|---|
| Server IP 192.168.2.105 Port 6653 | Con | nect | |
| Client IP 192.168.2.102 | Clo | se | |
| Control | | | 1 |
| Reset Transfer Transmit | | | |
| Status Bytes | | ^ | |
| | | | |
| Ý | | ~ | |
| | | | |

Figure 7: VB Application when Transmit Enters no Error

Using Wireshark we can see the message our VB client sends across the socket when no errors are introduced.

| *Ethernet | | | - 🗆 X |
|--|---|-------------------------------------|---------------------|
| <u>File Edit View Go Capture Analyze Statistics Telephor</u> | y <u>W</u> ireless <u>T</u> ools <u>H</u> elp | | |
| ▲ ■ ∅ ⊛ | | | |
| | | | - Everagion + |
| [,, μφ | | | - Expression + |
| No. Time Time Source Destination | Protocol Lengtr Info | | |
| | ICP 89 [ICP | segment of a reassembled PDU | |
| 2 22 0 192.168.2.105 192.168.2.10 | TCP 60 6653 | → 43/55 [ALK] Seq=1 ACK=36 W1n=51. | 2 Len=0 |
| A 22 0 102 158 2 102 102 102 108 2 102 | TCP 54 4375 | 5 a 6653 [ACK] Seg-36 Ack-36 Win-6/ | 1030 Len-0 |
| - 4 22 0 192.100.2.102 192.100.2.10 | 10- 34 4373 | 3 4 0033 [ACK] 364-30 ACK-30 WIN-04 | +050 Len=0 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| [Stream index: 0] | ▲ 0000 00 04 a3 00 00 02 94 | de 80 6c 23 88 08 00 45 00 | E. |
| [TCP Segment Len: 35] | 0010 00 4b 2e 3a 00 00 80 | 06 00 00 c0 a8 02 66 c0 a8 .K.: | f |
| Sequence number: 1 (relative sequence numbe | 0020 02 69 aa eb 19 fd bd | ac 20 b8 00 0f 48 b3 50 18 .i | н.р. |
| [Next sequence number: 36 (relative sequenc | 0030 ta 41 86 5d 00 00 88 | ca ed d2 dd 41 e1 a9 e4 ca .A. | ••••••A···· |
| Acknowledgment number: 1 (relative ack numb | 0040 04 0e 41 cc e8 ca e/ | eo 41 e/ 60 e8 e4 d2 dd cfA. | Δ |
| Header Length: 20 bytes | 41 80 CC de e4 41 80 | 80 D7 A | ······ |
| > Flags: 0x018 (PSH, ACK) | | | |
| Window size value: 64065 | | | |
| [Calculated window size: 64065] | | | |
| [Window size scaling factor: -1 (unknown)] | | | |
| > Checksum: 0x865d [validation disabled] | | | |
| Urgent pointer: 0 | | | |
| > [SEQ/ACK analysis] | | | |
| TCP segment data (35 bytes) | | | |
| A data segment used in reassembly of a lower-level protocol (tcp | segment_data), 35 bytes | Packets: 6 · Displayed: 4 (66.7%) | Profile: Default .: |

Figure 8: Client Transmission to Server with No Error

If we compare the payload of this message to our original transmitted message shown in Table 3 we see there are no errors introduced.

For this lab demonstration we have coded our server to echo back a corrected version of the message received from the client after is has been processed by our even parity decoder function. To make sure our decoder does not introduce any unintended bit flips when a message is sent without error we analyze this echoed message the server sends back to the client.

| *Ethernet | – 🗆 X |
|--|--|
| <u>File Edit View Go Capture Analyze Statistics Telephony</u> | <u>W</u> ireless <u>T</u> ools <u>H</u> elp |
| | |
| | |
| tcp | Expression + |
| No. Time Time Source Destination | Protocol Lengt' Info |
| 1 22 0 192.168.2.102 192.168.2.105 | TCP 89 [TCP segment of a reassembled PDU] |
| 2 22 0 192.168.2.105 192.168.2.102 | TCP 60 6653 → 43755 [ACK] Seq=1 Ack=36 Win=512 Len=0 |
| 3 22 0 192.168.2.105 192.168.2.102 | TCP 89 [TCP segment of a reassembled PDU] |
| 4 22 0 192.168.2.102 192.168.2.105 | ICP 54 43/55 → 6653 [ACK] Seq=36 Ack=36 Win=64030 Len=0 |
| | |
| [Stream index: 0] | ▲ 0000 94 de 80 6c 23 88 00 04 a3 00 00 02 08 00 45 001#E. |
| [TCP Segment Len: 35] | 0010 00 4b 02 18 00 00 64 06 ce 75 c0 a8 02 69 c0 a8 .Kdui |
| Sequence number: 1 (relative sequence numbe | 0030 02 00 73 f7 00 00 88 ca ed d2 dd 41 e1 a9 e4 ca |
| [Next sequence number: 36 (relative sequenc | 0040 d4 de 41 cc e8 ca e7 e8 41 e7 8b e8 e4 d2 dd cf A |
| Header Length: 20 bytes | 0050 41 8d cc de e4 41 8b 8b b7 AA |
| > Flags: 0x018 (PSH, ACK) | |
| Window size value: 512 | |
| [Calculated window size: 512] | |
| [Window size scaling factor: -1 (unknown)] | |
| > Checksum: 0x73f7 [validation disabled] | |
| Urgent pointer: 0 | |
| > [SEQ/ACK analysis] | |
| TCP segment data (35 bytes) | × |
| A data segment used in reassembly of a lower-level protocol (tcp.seg | gment_data), 35 bytes Packets: 6 · Displayed: 4 (66.7%) Profile: Default |

Figure 9: Server Transmission echoed back to Client

The echoed message from our PIC32 server demonstrates that no bit flips were introduced in this transmission since there were no errors. The control experiment where no errors are introduced is proven to work.

Case 2 – Error in Original String

When the VB application transmits a message back to the server with an error it has a status of "Transmit/Error".

| 5. ECE4532 Client2 v16 B | _ | | × |
|-------------------------------------|------|------|---|
| Server IP 192.168.2.105 Port 6653 | Conr | nect |] |
| Client IP 192.168.2.102 | Clo | se | |
| Control | | | 1 |
| Reset Transfer Transmit | | | |
| Status Bytes I Transmit/Error 35 | | ^ | |
| Í Ý | | | |
| | | ~ | |
| | | | |

Figure 10: VB Application when Transmit Enters an Error

We can look at the received message from the client:

| 📕 *Ethe | rnet | | | | | | | | | | - | | × |
|------------------------|----------------------------|----------------|-----------------|----------------------|----------|-----------------|--------------|----------------------|------------------|------------------|--------------|--------------|-------|
| <u>File</u> <u>E</u> d | it <u>V</u> iew <u>G</u> o | <u>Capture</u> | <u>A</u> nalyze | Statistics Telephony | Wireles | s <u>T</u> ools | <u>H</u> elp | | | | | | |
| | 1 🙆 📃 🔚 | | Q (= = | - | ΘΘ | 9 11 | | | | | | | |
| | | | •••• | | | - | | | | | | | L at |
| | | | | | | | | | | | | Expression. | |
| No. | Time Time | Source | | Destination | | Protoc | ol Length | Info | | | | | |
| Г | 1 22 0 | 192.168. | .2.102 | 192.168.2.105 | | TCP | 89 | [TCP se | gment of a | reassembled | PDU J | 0 | |
| | 2 22 0 | 192.168 | 2.105 | 192.168.2.102 | | тср | 50 | 6653 → [TCD co | 43755 [ACK | J Seq=1 ACK=: | 36 W1N=512 L | en=0 | |
| | 4 22 0 | 192.168 | .2.102 | 192.168.2.105 | | TCP | 54 | 43755 → | 6653 [ACK | 1 Sea=36 Ack: | =36 Win=6410 | 0 Len=0 | |
| | | 2021200 | | 1021200121105 | | 1.61 | 54 | | LINCK | 1 and an work | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| > Frame | : 1: 89 bytes | on wire | (712 bit | ts), 89 bytes ca… | ▲ 0000 | 00 04 | a3 00 00 (| 02 94 de | 80 6c 23 | 88 08 00 45 | 00 | ··· .1# | Е. |
| > Ether | net II, Src: | Giga-By | t_6c:23:8 | 38 (94:de:80:6c:… | 0010 | 00 4b | 27 d3 00 (| 30 80 06 Ed bd oc | 00 00 c0 | a8 02 66 c0 | a8 .K' | ··· ····† | |
| > Inter | rnet Protocol | Version | 4, Src: | 192.168.2.102, | 0020 | fa 87 | 86 5d 00 0 | 10 DU AC 30 88 ca | bh Ch ha | 41 e1 a9 e4 | ca 1 | Δ | F . |
| Y Trans | mission Contr | rol Prot | ocol, Sro | : Port: 43755 (4 | 0040 | d4 de | 41 cc_e8 | ca e7 e8 | 41 e7 8b | e8 e4 d2 dd | cf | A | |
| So | urce Port: 43 | 3755 | | | 0050 | 41 8d | cc dc e4 4 | 41 8b 8b | b7 | | AA | | |
| De | stination Por | rt: 6653 | | | | | _ | | | | | | |
| [[2 | tream index: | 0] | | | | | | | | | | | |
| <u>[</u> | CP Segment Le | en: 35] | | | | | | | | | | | |
| Se | quence number | . 1 | (relative | sequence numbe | | | | | | | | | |
| | ext sequence | number: | 20 (r | eracive sequenc | | | | | | | | | |
| | Knowledgment | number: | ⊥ (re | racive ack numb | | | | | | | | | |
| | auer Length: | 20 Dyces | \$ | | | | | | | | | | |
| | ags: 0x010 (P | - 311, ACK | / | | | | | | | | | | |
| W1 | nuow size Val | .ue: 041: | | | × | | | | | | | | |
| 0 7 | wireshark_pcapng | _D584406 | 5-699D-44FC | -89AF-990237B6FC8A_2 | 16031122 | 2921_a0627 | 2 | | Packets: 6 · Dis | played: 4 (66.7% | | Profile: Def | fault |

Figure 11: Client Transmission to Server with Error

The error is in the fifth group and is the third character. Compare the errored payload with the original transmitted payload (seen in Table 3) and we see 0xde was switched to 0xdc.

| Group | 7 Byte Group | | | | | | |
|-------|-----------------------------------|--|--|--|--|--|--|
| 1 | 88 ca ed d2 dd 41 e1 | | | | | | |
| 2 | a9 e4 ca d4 de 41 cc | | | | | | |
| 3 | e8 ca e7 e8 41 e7 8b | | | | | | |
| 4 | e8 e4 d2 dd cf 41 8d | | | | | | |
| 5 | cc <mark>dc</mark> e4 41 8b 8b b7 | | | | | | |

Table 4: 2D Even Parity Encoded Receive Buffer (With Error Highlighted)

We now look at the corrected echoed message server sends back to client and see the 0xdc was corrected back to 0xde. The error was clearly detected since we coded our PIC32 red led (LED1) to go high when an error was detected.

| *Ethernet | | | | | | | - | | × |
|--|----------------|------------------------------|---------|--------------------|----------------------|----------------------------------|-------------|--------------|-------|
| <u>File Edit View Go Capture Analyze Statistics Telephor</u> | <u>y W</u> ire | eless <u>T</u> ools <u>H</u> | elp | | | | | | |
| 🛋 🔳 🔬 💿 📙 🛅 🗙 🖻 I 🍳 🗢 🕾 🕾 💆 🧮 | Ð, | ର୍ ବ୍ 🎹 | | | | | | | |
| 📕 tcp | | | | | | 6 | Х 🔿 🔹 Б | pression | . + |
| No. Time Time Source Destination | | Protocol | Length | Info | | | | | |
| 1 22 0 192.168.2.102 192.168.2.105 | | TCP | 89 | [TCP se | gment of a | reassembled PDU |] | | |
| 2 22 0 192.168.2.105 192.168.2.103 | | TCP | 60 | 6653 → | 43755 [ACK] |] Seq=1 Ack=36 W | in=512 Len | =0 | |
| 3 22 0 192.168.2.105 192.168.2.10 | | TCP | 89 | [TCP se | gment of a | reassembled PDU |] | | |
| 4 22 0 192.168.2.102 192.168.2.10 | | TCP | 54 | 43755 → | 6653 [ACK |] Seq=36 Ack=36 | Win=64100 | Len=0 | |
| | | | | | | | | | |
| Source Port: 6653 | A 0 | 000 94 de 80 | 6c 23 8 | 8 00 04 | a3 00 00 | 02 08 00 45 00 | 1# | | |
| Destination Port: 43755 | 0 | 010 00 4b 00 |)79000 | 0 64 06 h 00 0f | d0 14 c0 42 af bd | a8 02 69 c0 a8 ac 1d 77 50 18 | .K.yd. f | i. B w | . |
| [Stream index: 0] | 0 | 030 02 00 7d | 5f 00 0 | 0 88 ca | ed d2 dd | 41 e1 a9 e4 ca | | A | . |
| Sequence number: 1 (relative sequence numbe | 0 | 040 d4 de 41 | cc e8 c | a e7 e8 | 41 e7 8b | e8 e4 d2 dd cf | Ă | Α | |
| [Next sequence number: 36 (relative sequence | 0 | 050 41 8d cc | de e4 4 | 1 8b 8b | b7 | | AA | • | |
| Acknowledgment number: 36 (relative ack num | | | | | | | | | |
| Header Length: 20 bytes | | | | | | | | | |
| > Flags: 0x018 (PSH, ACK) Window ping values 512 | | | | | | | | | |
| Minuow Size Value: 512 [Calculated window size: 512] | | | | | | | | | |
| [Window size scaling factor: -1 (unknown)] | | | | | | | | | |
| > Checksum: 0x7d5f [validation disabled] | | | | | | | | | |
| Urgent pointer: 0 | ~ | | | | | | | | |
| | | | | | Packets: 6 · Dis | played: 4 (66.7%) | Pi | rofile: Defa | ault |

Figure 12: Server Transmission to Client with Error Corrected



Figure 13: PIC32 LED1 High when Error is Detected

Case 3 – Error in Parity Location

A special case where we investigate an error in the column parity character will conclude our results. If an error occurs in the column parity character our program will still say there are no errors. The calculation is correct since a bit flip in the row parity character does no compromise our original message.

To demonstrate, we loop the client sending information to the server until we introduce an error in the row parity character.

| *Ethernet | | | | | - 🗆 X |
|---|---------------------------|----------------------------|-------------|-----------------------------------|------------------|
| <u>File Edit View Go Capture Analyze Statistics Telephony</u> | <u>y</u> <u>W</u> ireless | <u>T</u> ools <u>H</u> elp | | | |
| ◢ ■ ∅ ◎ 📙 🖿 🗙 🖬 ۹ ⇔ 🕾 🗿 📃 🚍 | . ⊕, ⊖, | Q. 🎹 | | | |
| 📙 tcp | | | | | Expression + |
| No. Time Time Source Destination | | Protocol | Length Info | | |
| 1 22 0 192.168.2.102 192.168.2.105 | | TCP | 89 [TCP | segment of a reassembled PDU | n] |
| 2 22 0 192.168.2.105 192.168.2.102 | | TCP | 60 6653 | → 43755 [ACK] Seq=1 Ack=36 N | Win=512 Len=0 |
| 3 22 0 192.168.2.105 192.168.2.102 | | TCP | 89 [TCP | segment of a reassembled PDU | n] |
| 4 22 0 192.168.2.102 192.168.2.105 | | TCP | 54 43755 | 5 → 6653 [ACK] Seq=36 Ack=36 | Win=62910 Len=0 |
| | | | | | |
| > Frame 1: 89 bytes on wire (712 bits), 89 bytes ca | ▲ 0000 0010 | 00 04 a3 0 | 00 02 94 | de 80 6c 23 88 08 00 45 00 | E. ** f |
| Ethernet 11, SrC: Giga-Byt_6C:23:88 (94:de:80:6C:) Internet Protocol Version 4, Src: 192 168 2 162 | 0020 | 02 69 aa el | 5 19 fd bd | ac 1f fa 00 0f <u>47</u> 55 50 18 | .iGUP. |
| Transmission Control Protocol. Src Port: 43755 (4 | 0030 | f5 e1 86 5 | 88 00 00 88 | ca ed d2 dd 41 e3 a9 e4 ca |]A |
| Source Port: 43755 | 0040 | d4 de 41 c | : e8 ca e7 | e8 41 e7 8b e8 e4 d2 dd cf | A A |
| Destination Port: 6653 | 0050 | 41 80 CC 0 | e e4 41 6D | 8D D7 | AA |
| [Stream index: 0] | | | | | |
| [TCP Segment Len: 35] | | | | | |
| Sequence number: 1 (relative sequence numbe | | | | | |
| [Next sequence number: 36 (relative sequenc | | | | | |
| Acknowledgment number: 1 (relative ack numb | | | | | |
| > Flags: 0x018 (PSH_ACK) | | | | | |
| Window size value: 62945 | | | | | |
| wirechark pranne D5844065-6000-44ED-80AE-00073786EC8A / | 201603112230 | 21 206868 | | Parkete: 6 - Dicolaved: 4 (66 7%) | Profile: Default |

Figure 14: Client Transmission to Server with Error in Row Parity Character

In the example shown above we can see there is an error in the row parity character for the first group. What was a row parity character of 0xe1 was flipped to 0xe3.

| Group | 7 Byte Group |
|-------|-----------------------------------|
| 1 | 88 ca ed d2 dd 41 <mark>e3</mark> |
| 2 | a9 e4 ca d4 de 41 cc |
| 3 | e8 ca e7 e8 41 e7 8b |
| 4 | e8 e4 d2 dd cf 41 8d |
| 5 | cc de e4 41 8b 8b b7 |

Table 5: 2D Even Parity Encoded Receive Buffer (With Row Parity Error Highlighted)

Our PIC32 server does not detect errors in this transmission since the error occurred in the even row parity character. The message echo back from the server to the client thus does not have this bit flip corrected. In fact, we cannot correct this error since the even row parity character does not necessarily need to contain an even number of bits. We therefore cannot detect which bit was flipped. In the echoed message back to the client we see the error is still present.

| 📕 *Ether | net | | | | | | | | | | | | | | _ | | × |
|---------------------------|-------------------------------|----------------------|-----------------|-------------------------|-----------|-------------------|----------------|----------------|----------------|--------------|------------------|----------------|--------------------|--------------------|----------------------|-------------|----------|
| <u>F</u> ile <u>E</u> dit | <u>V</u> iew <u>G</u> o | <u>C</u> apture | <u>A</u> nalyze | <u>S</u> tatistics | Telephony | <u>W</u> ireless | Tools | <u>H</u> elp | | | | | | | | | |
| 🛋 🔳 🖉 | 💿 📘 🛅 | 🗙 🖻 | ۹ 👄 ه | 😫 🗧 | 🕹 📃 🔳 | \oplus Θ | Q, 🎹 | | | | | | | | | | |
| 📙 tcp | | | | | | | | | | | | | | | $\times \rightarrow$ | Expression | n + |
| No. | Time Time | Source | | Destir | ation | | Protoc | ol L | ength I | Info | | | | | | | |
| Г | 1 22 0 | 192.168 | .2.102 | 192. | 168.2.105 | | TCP | | 89 | [TCP s | segment | of a | reasser | nbled PC | U] | | |
| | 2 22 0 | 192.168 | .2.105 | 192. | 168.2.102 | | TCP | | 60 (| 6653 - | → 43755 | [ACK] | Seq=1 | Ack=36 | Win=512 | Len=0 | |
| | 3 22 0 | 192.168 | .2.105 | 192. | 168.2.102 | | TCP | | 89 | TCP : | segment | of a | reasser | nbled PC | 00] | 10 10 | |
| | 4 22 0 | 192.168 | .2.102 | 192. | 168.2.105 | | TCP | | 54 4 | 43755 | → 0053 | [ACK] | Seq=30 | ACK=36 | W1N=629 | 10 Len=0 | |
| | | | | | | | | | | | | | | | | | |
| | | | (= | | | | | | | | | | | | 2.0 | | - |
| > Frame | 3: 89 bytes net II. Sect | On wire Microch | i 00:00 | .ts), 89 b 02 (00∙04 | ytes ca | 0010 | 94 de 00 4b | о0 bc 01 4e | 23 88 | 0000 0640 | 94 a3 0 06 cf | 00 00 3f c0 | o2 08 0 a8 02 6 | w 45 00 9 c0 a8 | 1# .K.N. | .d? | .c. i |
| > Interr | net Protocol | Version | 4, Src: | 192.168. | 2.105, | 0020 | 02 66 | 19 fd | aa eb | 00 0 | of 47 | 55 bd | ac 20 1 | d 50 18 | .f | GU | .Р. |
| Y Transm | nission Cont | rol Prot | ocol, Sr | rc Port: 6 | 653 (66 | 0030 | 02 00 d4 de | 74 13 41 cc | 00 00 e8 ca |) 88 c | a ed 8 41 | d2 dd e7 8h | 41 e3 a e8 e4 d | 9 e4 ca 2 dd cf | t | A. | ••• |
| Sou | irce Port: 6 | 653 | | | | 0050 | 41 8d | cc de | e4 41 | L 8b 8 | 3b b7 | ., | | 2 00 01 | A | A | |
| Des | tination Po | rt: 4375 | 5 | | | | | | | | | | | | | | |
| | P Segment | en: 351 | | | | | | | | | | | | | | | |
| Seq | uence numbe | r: 1 | (relativ | e sequenc | e numbe | | | | | | | | | | | | |
| [Ne | ext sequence | number: | 36 (| relative | sequenc | | | | | | | | | | | | |
| Ack | nowledgment | number: | 36 (| relative | ack num | | | | | | | | | | | | |
| Hea | der Length: | 20 byte | s N | | | | | | | | | | | | | | |
| Win | ags: exerte (ndow size va | гэп, АСК lue: 512 | , | | | | | | | | | | | | | | |
| | 1 1 | | 54.03 | | | <u> </u> | | | | | Darkete | - 6 - Dim | alaved: 44 | (66.7%) | | Profile: Dr | fault |

Figure 15: Server Transmission to Client with ROW Parity Error Not Corrected

III. Conclusion

In conclusion, we have shown that a message can be encoded to contain redundant information so if an error is introduced during transmission we can correct it on the receiving side. The process is efficient since it does not require the server to retransmit the entire original message if an error is introduced. The receiver has the means to correct the error itself. 2D even parity encoding is an efficient error correction algorithm since it does not require the original message to be sent twice repeatedly. For example if in this experiment our original message was 30 bytes in length. With the 2D even parity encoding the message length only increased by 5 bytes instead of doubling our message by two to 60 bytes.

There are limitations to the 2D parity encoder. For example, in this lab we restricted errors to only occur to one bit in our original message. If there were two errors we wouldn't be able to correct the message since there is probability the errors will occur in the same row or column. If there are two bit flips to one row and column the even parity encoding check will hold true and pass through our decoder undetected. For more than two bits an error in a transmission we need even more redundant parity bits. We will investigate this process in the next lab.

IV. Appendix

```
void evenParityEncoder(const char *myStr, char *transmitBuffer, int tlen)
{
    int i;
    int j = 0;
    int m = 0;
    // Create last row buffer container
    char rowBuffer[bufferCols];
    for(i=0; i < tlen; i++)</pre>
    ł
        // Check if we are calculating col parity
        if(i!=0 && (j==6))
        {
            setColParity(rowBuffer, transmitBuffer, i);
            i = 0;
            continue;
        }
        // Copy one bit shifted string contents into transmit buffer
        transmitBuffer[i] = myStr[m++] << 1;</pre>
        // See if we need to set row parity bit high
        if (hasEvenParity(transmitBuffer[i])==0)
        {
            transmitBuffer[i] ^= 1;
        }
        rowBuffer[j++] = transmitBuffer[i];
   }
}
```

```
void setColParity(char *rowBuffer, char *transmitBuffer, int colIndex)
{
   int i, j;
int rowBufferLen = strlen(rowBuffer);
char colBitBuffer;
   transmitBuffer[colIndex] = 0;
   for (i = 0; i < 8; i++)
    {
        colBitBuffer = 0;
        // Last Row is reserved for parity
        11
        for (j = 0; j < rowBufferLen; j++)</pre>
        {
             colBitBuffer |= ((rowBuffer[j] & (0x01 << i)) >> i) << j;</pre>
        }
        if (hasPartialEvenParity(colBitBuffer, rowBufferLen)==0)
        {
             transmitBuffer[colIndex] |= (1 << i);</pre>
        }
   }
}
```

Code Snippet 2

```
void evenParityDecoder(char *recieveBuffer, int rlen)
{
    int i;
    int j = 0;
   int colError;
    // Plus one for the col parity bit character
    char rowBuffer[bufferCols+1];
    // First we check column parity chars
    for(i=0; i < rlen; i+=bufferCols+1)</pre>
    {
        // create a buffer to hold our row and col parity bit character
        memcpy(rowBuffer, recieveBuffer+i, bufferCols+1);
        // check if array of characters has even parity along the column
        colError = hasEvenParityArray(rowBuffer, bufferCols+1);
        // if a column char doesn't have even parity then we know
        // there is an error
        if (colError >= 0)
        {
            // Set led 1 (red) high
            mPORTDSetBits(BIT_0);
            for(j=0; j < bufferCols; j++)</pre>
            {
                if (hasEvenParity(rowBuffer[j])==0)
                {
                    // flip the problem bit
                    rowBuffer[j] ^= (0x01 << colError);</pre>
                    // Correct the received data.
                    recieveBuffer[i+j] = rowBuffer[j];
                    return;
                }
            }
            // We detected an error but is was a in the parity col. We blink
            // led 1 (red))
            DelayMsec(100);
            mPORTDClearBits(BIT_0);
            DelayMsec(100);
            mPORTDSetBits(BIT_0);
            DelayMsec(100);
            mPORTDClearBits(BIT_0);
        }
   }
}
```

Code Snippet 3

```
int hasEvenParityArray(char *rowBuffer, int rowBufferLen)
{
   int i, j;
char colBitBuffer;
    for (i = 0; i < 8; i++)
    {
        colBitBuffer = 0;
        // Last Row is reserved for parity
        11
        for (j = 0; j < rowBufferLen; j++)</pre>
        {
            colBitBuffer |= ((rowBuffer[j] & (0x01 << i)) >> i) << j;</pre>
        }
        if (hasPartialEvenParity(colBitBuffer, rowBufferLen)==0)
        {
            return i;
        }
    }
    return -1;
}
```

Code Snippet 4

```
int hasPartialEvenParity(char x, int colCount) {
    int i;
    int count = 0;
    // Loop through each bit of the car
    //
    for (i = 0; i < colCount; i++)
    {
        if (x & (0x01 << i))
        {
            count++;
        }
    }
    if (count % 2) return 0;
    return 1;
}</pre>
```

Code Snippet 5

Full raw source code available upon request. Email <u>devin.trejo@temple.edu</u>.

V. References

[1] W. Stallings, Data and Computer Communications, Person Education Inc., 2014.