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.

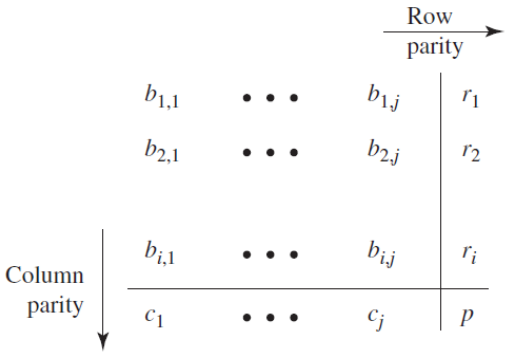

Figure 1: Parity Calculation [1, p. 192]

0	1	1	1	0	1
0	1	1	1	0	1
0	1	0	0	0	1
0	1	0	1	1	1
0	0	0	1	1	0

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.

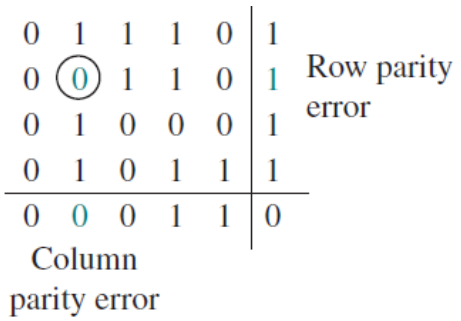

Figure 3: Correctable Even-Parity Error [1, p. 192]

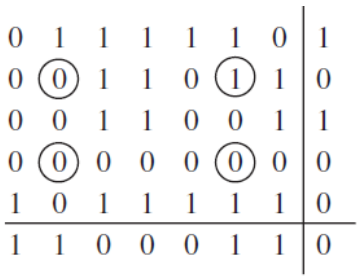

Figure 4: Undetectable Even-Parity Encoded Error [1, p. 192]

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 has an odd number of bits we make it even by ORing 0x01 with our shifted char. The parity bits are highlighted in yellow below.

Table 1: Char Even Row Parity Calculation (First 5 Bytes)

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
e	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

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.

Table 2: Even Column Parity Calculation

ASCII	ASCII BINARY (Shifted & Parity)								ASCII Hex (Shifted & Parity)
D	1	0	0	0	1	0	0	0	0x88
e	1	1	0	0	1	0	1	0	0xca
v	1	1	1	0	1	1	0	1	0xed
i	1	1	0	1	0	0	1	0	0xd2
n	1	1	0	1	1	1	0	1	0xdd
("SPACE")	0	1	0	0	0	0	0	1	0x41
Even Column Parity	1	1	1	0	0	0	0	1	0xe1

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.

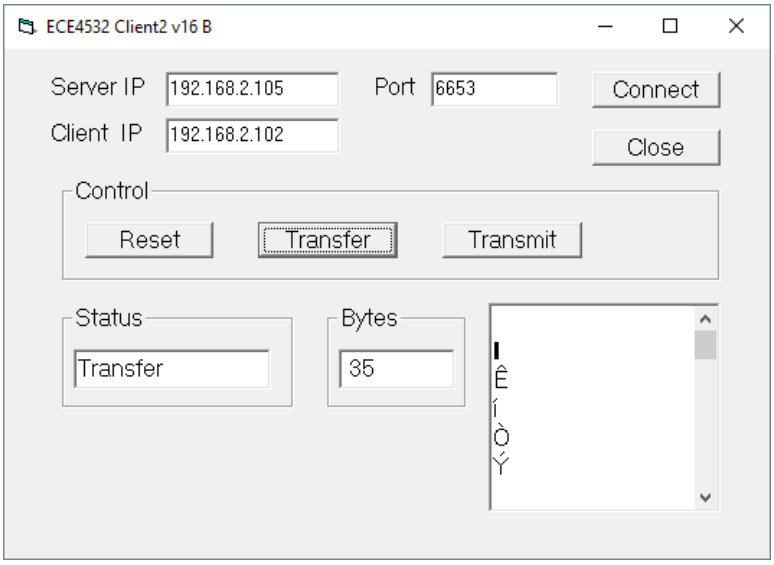

Figure 5: Visual Basic Even Parity Application

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.

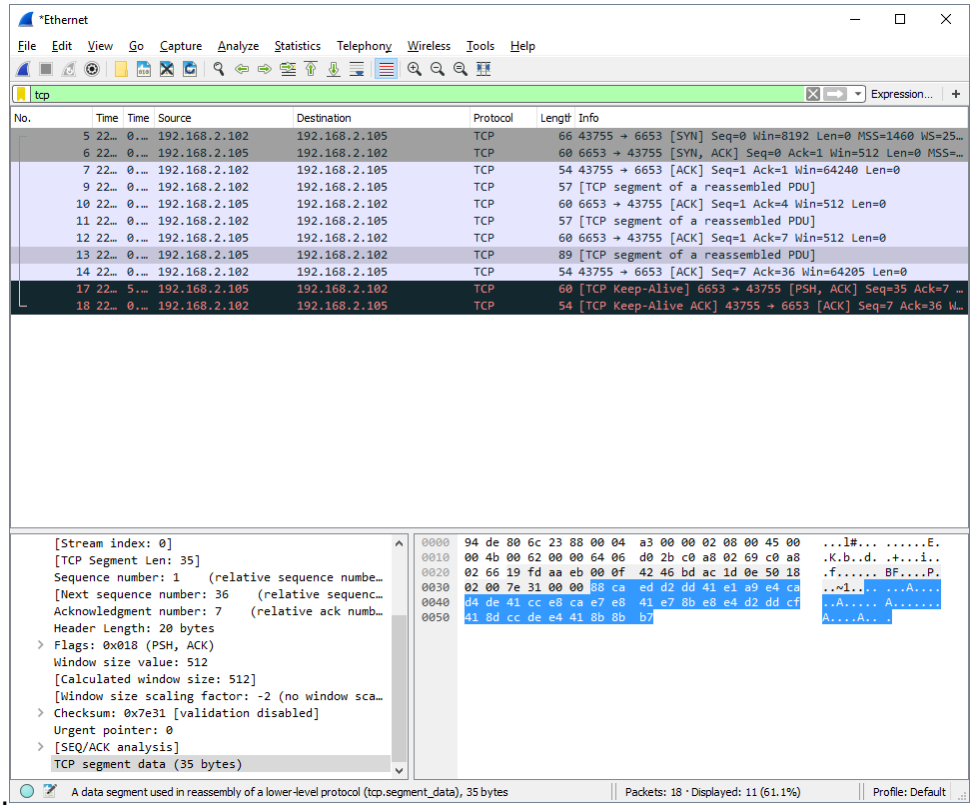

Figure 6: Wireshark Encoded Payload Analysis

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".

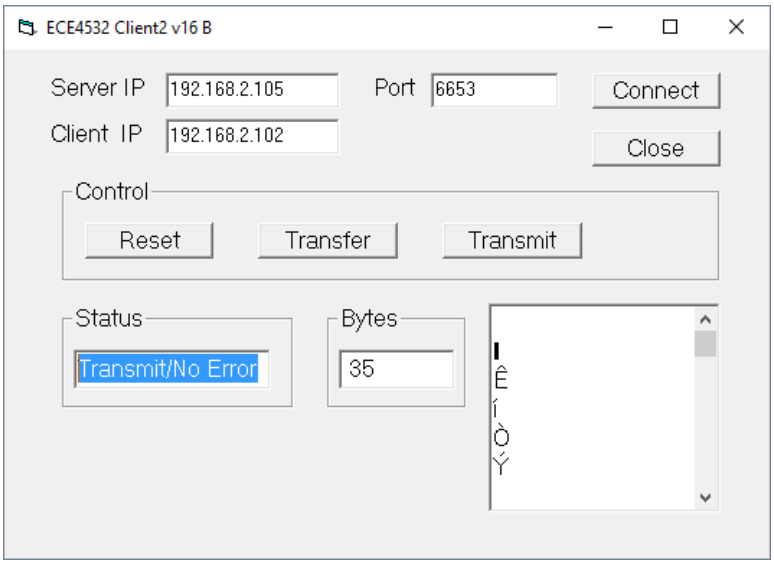

Figure 7: VB Application when Transmit Enters no Error

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

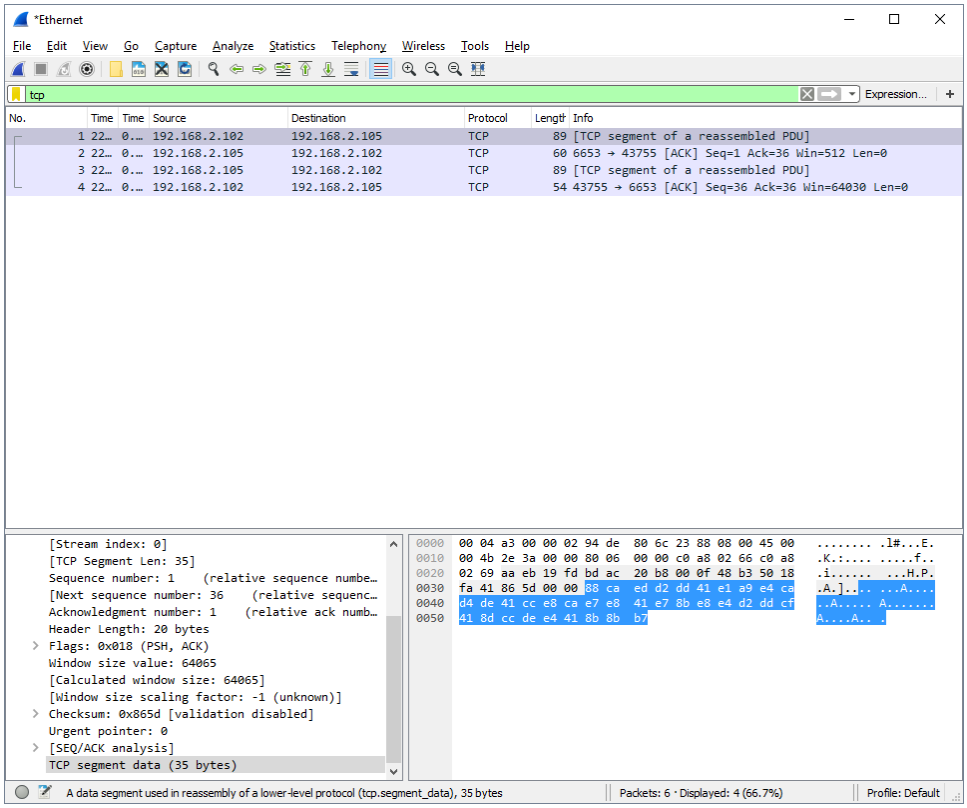

Figure 8: Client Transmission to Server with No Error

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.

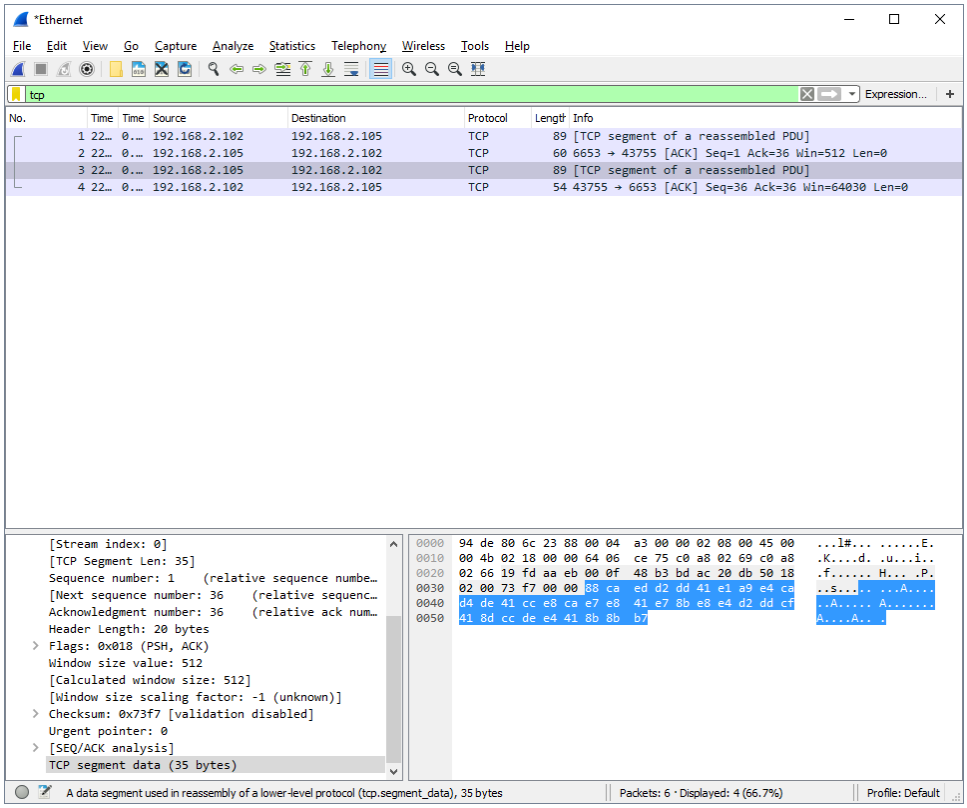

Figure 9: Server Transmission echoed back to Client

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”.

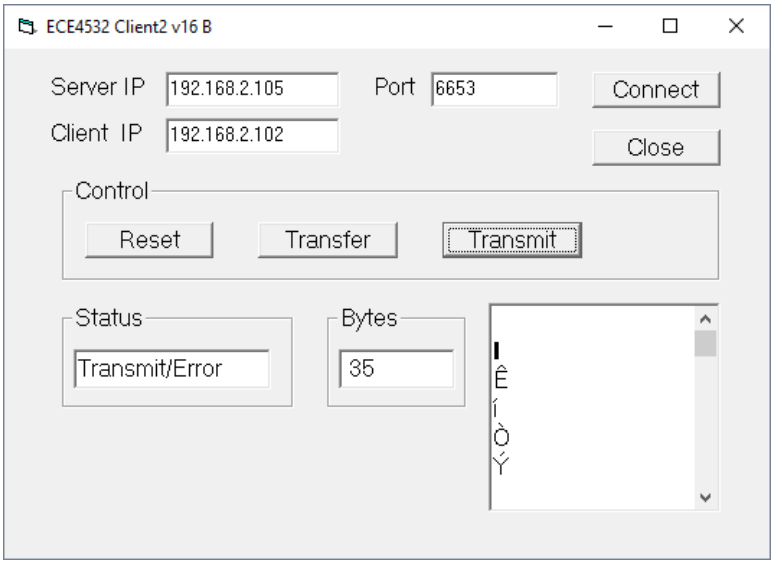

Figure 10: VB Application when Transmit Enters an Error

We can look at the received message from the client:

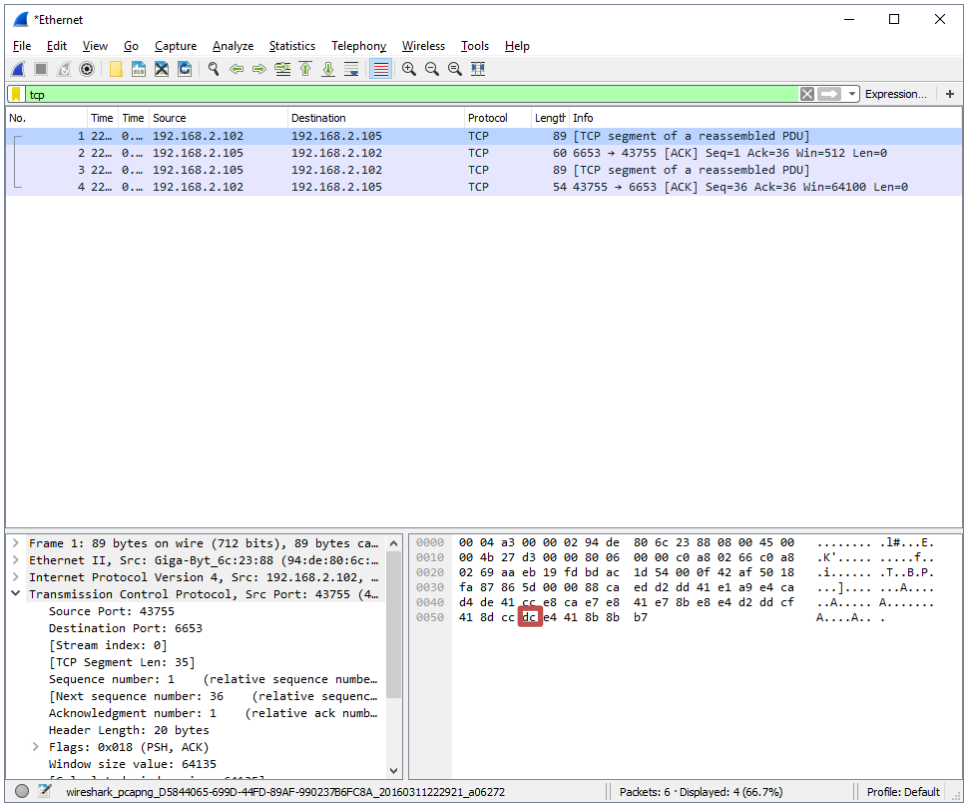

Figure 11: Client Transmission to Server with Error

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.

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

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 dc e4 41 8b 8b b7

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.

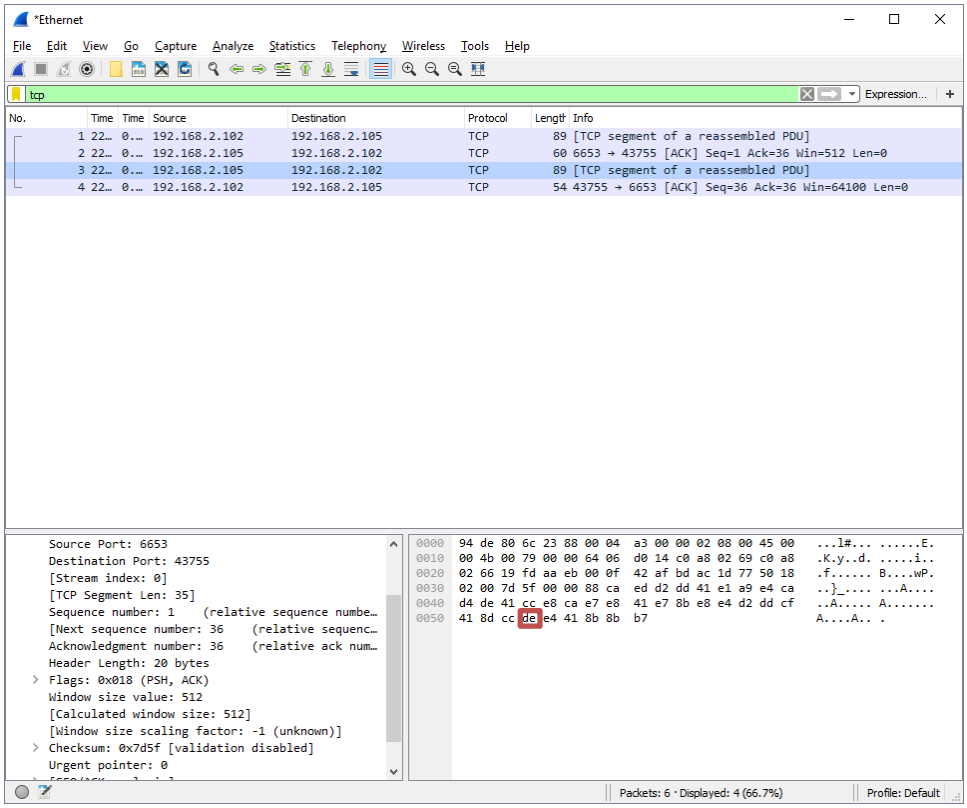

Figure 12: Server Transmission to Client with Error Corrected

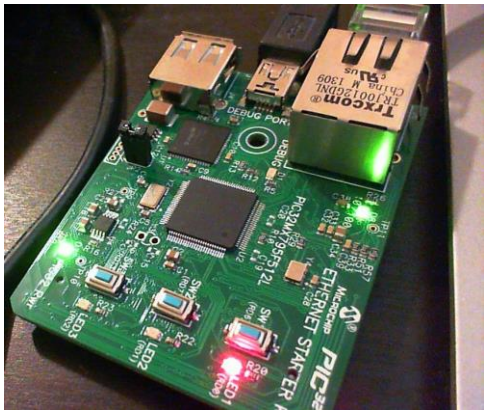

Figure 13: PIC32 LED1 High when Error is Detected

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.

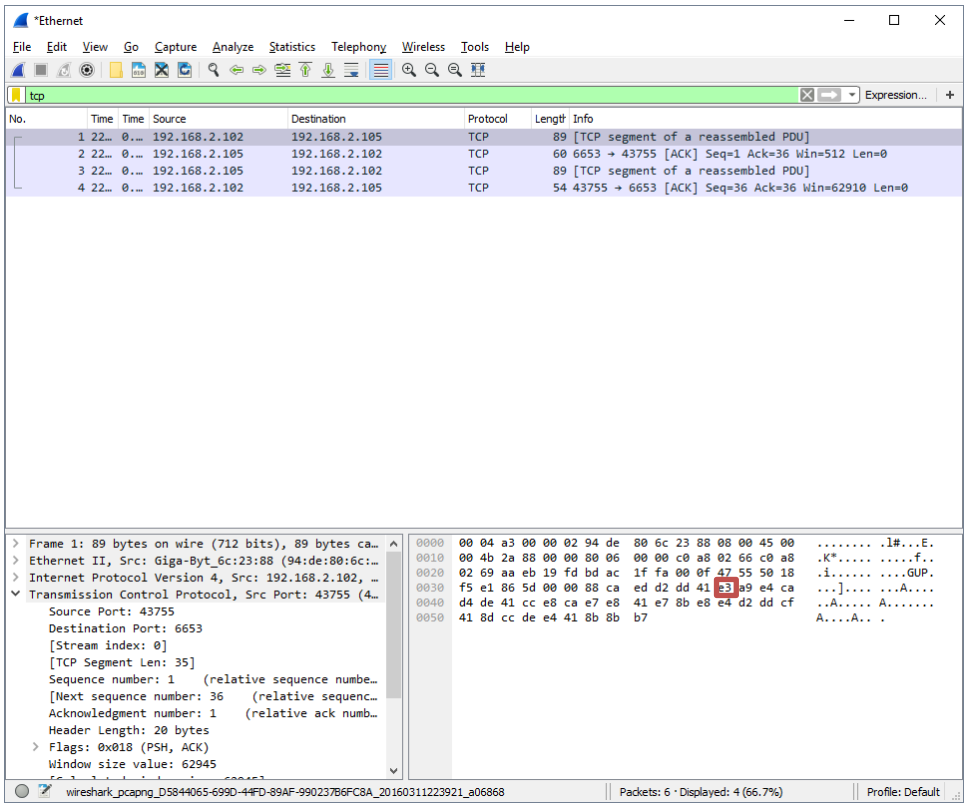

Figure 14: Client Transmission to Server with Error in Row Parity Character

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.

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

Group	7 Byte Group
1	88 ca ed d2 dd 41 e3
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

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.

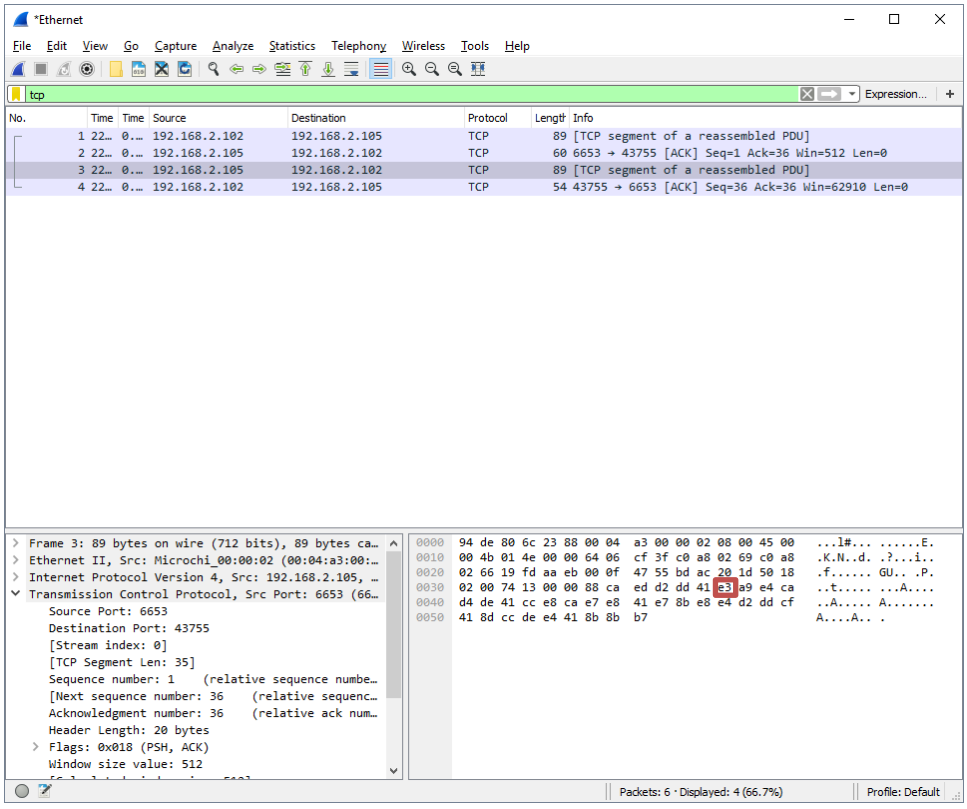

Figure 15: Server Transmission to Client with ROW Parity Error Not Corrected

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++)
    {
        // Check if we are calculating col parity
        if(i!=0 && (j==6))
        {
            setColParity(rowBuffer, transmitBuffer, i);
            j = 0;
            continue;
        }
        // Copy one bit shifted string contents into transmit buffer
        transmitBuffer[i] = myStr[m++] << 1;

        // See if we need to set row parity bit high
        if (hasEvenParity(transmitBuffer[i])==0)
        {
            transmitBuffer[i] ^= 1;
        }
        rowBuffer[j++] = transmitBuffer[i];
    }
}
```

Code Snippet 1

```
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
        //
        for (j = 0; j < rowBufferLen; j++)
        {
            colBitBuffer |= ((rowBuffer[j] & (0x01 << i)) >> i) << j;
        }
        if (hasPartialEvenParity(colBitBuffer, rowBufferLen)==0)
        {
            transmitBuffer[colIndex] |= (1 << i);
        }
    }
}
```

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)
    {
        // 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++)
            {
                if (hasEvenParity(rowBuffer[j])==0)
                {
                    // flip the problem bit
                    rowBuffer[j] ^= (0x01 << colError);

                    // 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
        //
        for (j = 0; j < rowBufferLen; j++)
        {
            colBitBuffer |= ((rowBuffer[j] & (0x01 << i)) >> i) << j;
        }
        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;
}
```

Code Snippet 5

Full raw source code available upon request. Email devin.trejo@temple.edu.

V. References

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