TCP/IP Buffered Data Transmission

Devin Trejo

devin.trejo@temple.edu

Date: 2/22/2016

I. Summary

In the TCP/IP Buffered Data Transmission lab we look into sending a large message across several smaller datagrams. Blocking data up into smaller sizes is commonly done on the server side when a client requests a large message to be sent across the network. In this experiment we use our PIC32 MCU to provide a 1060 byte message and parse it in smaller blocks of data. We then send each block of data to the client and analyze the transmission in Wireshark to see if the server does send the data successfully and if the client receives the intended full message. We show success in transmission between server and client when our larger message is parsed into two and three separate datagrams with arbitrary delays put in between. The delays are put in to simulate transmission propagation time. Upon trying to send our larger message in 50 byte blocks we receive server re-transmission behavior that is expected when using TCP as a transfer protocol, as discuss in the previous lab.

I. Introduction

Most of web traffic today is to either Google owned YouTube or Netflix [1]. Both these sites provided video entertainment to their end users. Sending video over the backbone of the internet is a bandwidth intensive operation that requires the frames that compose the resulting video to be split across multiple internet frames.

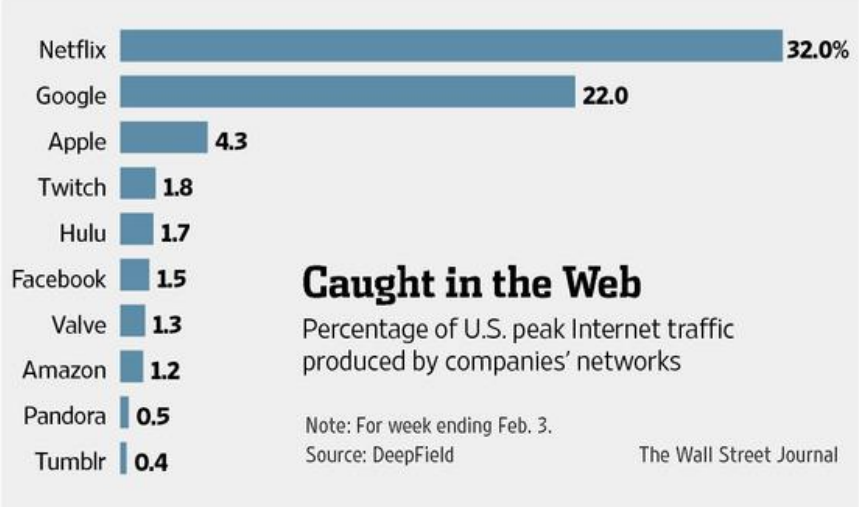

Figure 1: Top Websites by Bandwidth Usage [1]

Today most file transfers exceeded the limit of the 1500 bytes put on by TCP/IP frames [2]. Therefore, a coordination between server and client is necessary for parsing data into smaller data sizes on the server side and assembly of the full larger message on the client side. This lab was a demonstration of such coordination is indeed possible.

We will experiment with sending a 1060 byte message across two buffered frames, three buffered frames, and finally numerous frames 50 bytes in size. We also experiment with putting in delays between each transmission to simulate transmission propagation time. What we uncover in this lab shows how larger data transfer occur when using TCP/IP.

II. Discussion

Two Buffered Message Transmission

The first experiment is to send our message spread across two send buffers. In between each send command we put in an arbitrary delay. The resulting server send code is seen in Code Snippet 1.

We now boot up our PIC32 board and Wireshark to analyze the packets in transit. In the last lab we analyzed the packets sent when the users *'connects'* and *'resets'* the connection via the Visual Basic GUI. The full Wireshark output for our two buffered message send is as follows:

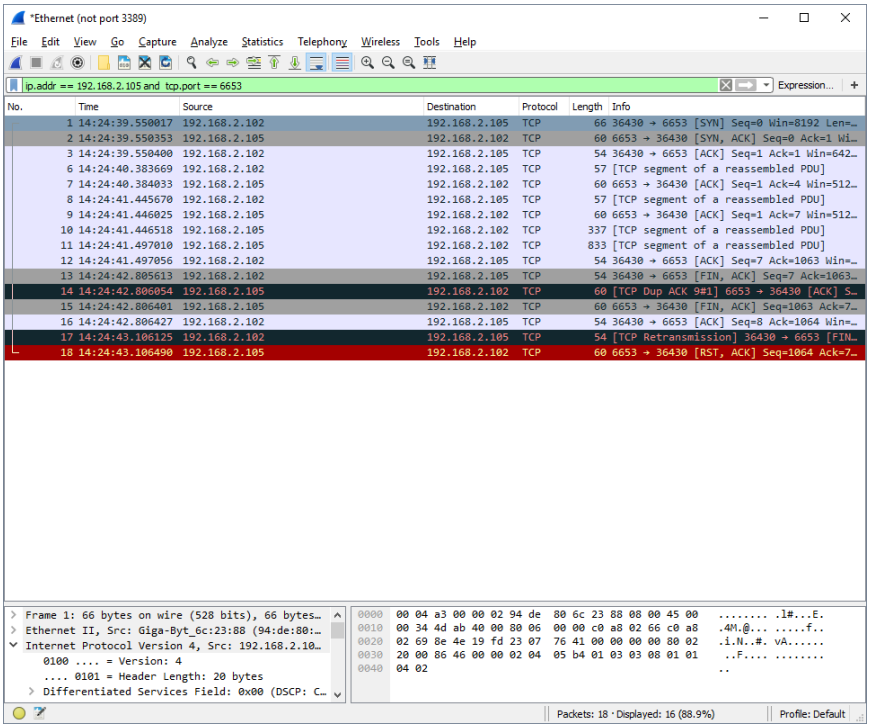

Figure 2: Two buffer transfer with 50 msec delay

We filter the Wireshark output to only monitor traffic that contains the IPv4 address of our PIC32 board server "192.168.2.105", and with a destination port of 6653. The first two packets sent originate from our client computer when it attempts to connect to the PIC32 board. The clients attempts a connection and the server responds with an ACK stating that connection is established. The client computer

also responds with an ACK as well to confirm the connection. Following the successful connection we see packets 6 and 7 in the image shown above show the packets sent when attempting a *'reset'* of the connection. Again the client computer sends a reset command to the server and the server responds with an ACK response notifying it received the full message correctly.

Packet 9 is the request from the command client to send the message. Recall before we split our full 1061 byte message into two pieces dependent on my birth month (April). The code for splitting the message up into two buffers is shown Code Snippet 2. We expect our two messages to be split into the following lengths:

$$tlen1 = tlen * \left(\frac{BM}{15}\right) = 1061 * \left(\frac{4}{15}\right) = 282 \quad (1)$$

$$tlen2 = tlen - tlen1 = 1061 - 282 = 779 \quad (2)$$

The client sends a message to the server prefixed with the first four bytes as: "02 84". This sequence of numbers signifies to the server to start our full message transfer spanning two datagrams. Looking at our Wireshark capture we can see the indeed that the response from the server is split into two datagrams. The first datagram seen in packet 10 is 337 bytes in length and the second datagram is 833 bytes in length. The lengths are longer than expect since we also have to account for the IPv4 and TCP overhead headers. Analyzing the contents of these messages confirm our message is being sent across two datagrams.

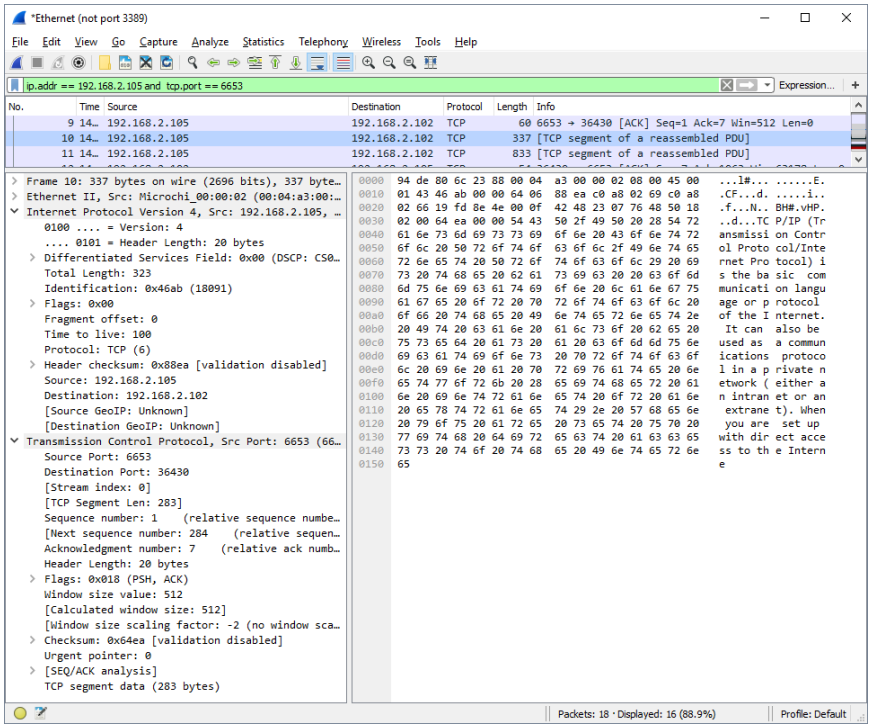

Figure 3: Two buffer transfer Packet 10

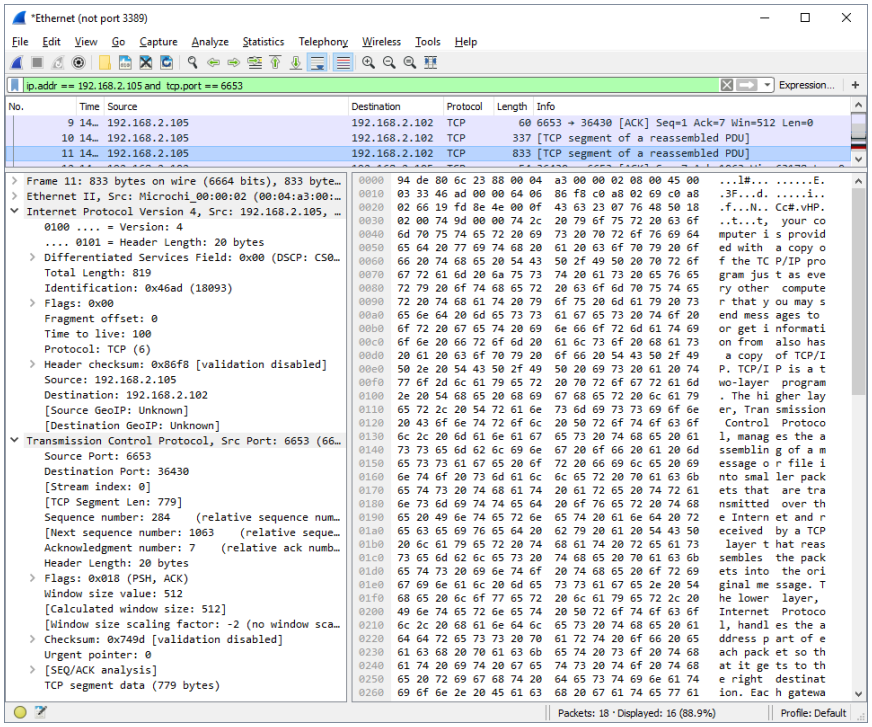

Figure 4: Two buffer transfer Packet 11

Refering back to Figure 2 we can see the timestamps between packet 10 and 11 equate to a 50.492 msec delay between the two messages. What happens when

we decrease that time length to 0 msecs? Referring back to Code Snippet 1 we simply set the *DelayMsec* function parameter to 0.

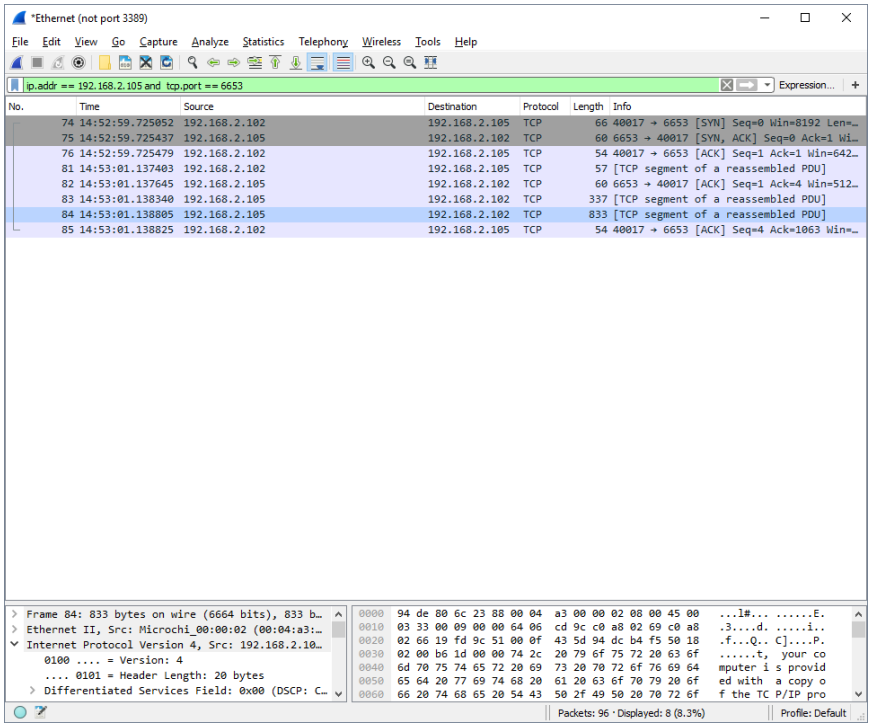

Figure 5: Two buffer transfer with 0 msec delay

The figure above shows the Wireshark capture when the delay between the two message is set to zero. In this capture packet 83 and 84 show the transmission of our message. We see no ill effects to decreasing the time between messages to zero. The time delay as seen in the Wireshark capture shows the sever sent the message with a difference of 0.0465 msec delay.

Now we experiment with increasing the delay to outside the millisecond range to see if transmission will still complete. We start by increasing the delay to 5 secs.

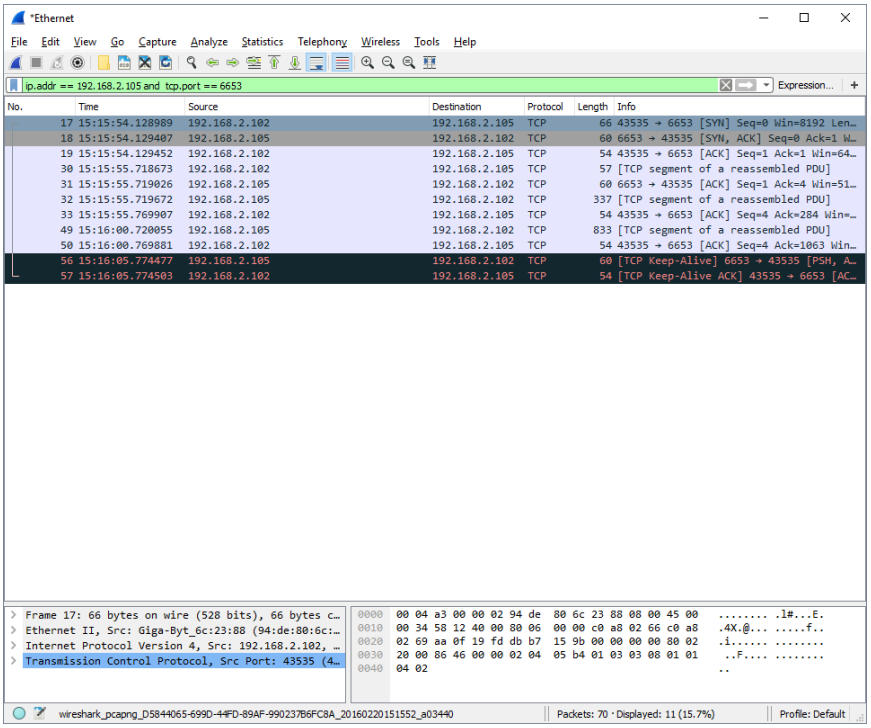

Figure 6: Two buffer transfer with 5 sec delay

In this transmission we see the second message being sent in the Wireshark capture with the expected 5 second delay. It appears that after the first message buffer was send (Packet 9) the client eventually timed out and send an ACK message back to the server. The server was still waiting to send the second message however so after 5 seconds the server finally sends the second half the message. A separate ACK message is seen being sent back from the client after this second buffers is received. A tabulated result of the successful two buffer message transfers with arbitrary delay lengths is provided below.

Table 1: Two buffer transfer with Arbitrary Delays

Coded Server Delay Between Buffers	Wireshark Observed Delay	Two Buffers Successfully Sent?
0 msec	0.0465 msec	Yes
25 msec	25.636 msec	Yes
50 msec	50.492 msec	Yes
500 msec	500.333 msec	Yes
1000 msec	1000.174 msec	Yes
2500 msec	2500.508 msec	Yes
5000 msec	5000.383 msec	Yes

From the results observe red it does not appear there is a limit to the delay between message buffers. Transmission propagation time needs to be account for however to ensure the client does not request a re-transfer of a message that is still in

transmission. For example, on a Gigabit Ethernet physical link between a server and client the propagation time can be expected to be in the millisecond range. However in satellite communications propagation times should be expected to be in the seconds range. In the former scenario a client should be programed so it waits for a longer time between transmissions.

Three Buffered Message Transmission

Next we repeat the experiment complete before but with our message split into three separate buffers. As before our message lengths are parsed based on my birth month/day of April 6th.

$$tlen1 = tlen * \left(\frac{BM}{25}\right) = 1061 * \left(\frac{4}{25}\right) = 169 \quad (3)$$

$$tlen2 = tlen * \left(\frac{BD}{100}\right) = 1061 * \left(\frac{6}{100}\right) = 63 \quad (4)$$

$$tlen3 = tlen - tlen1 - tlen2 = 1061 - 169 - 63 = 829 \quad (5)$$

Parsing code is seen in Code Snippet 3. Sending the message is the same as two message buffer send but with an extra delay and send command added (see Code Snippet 2).

Again we start with a delay of a default of 50 msecs.

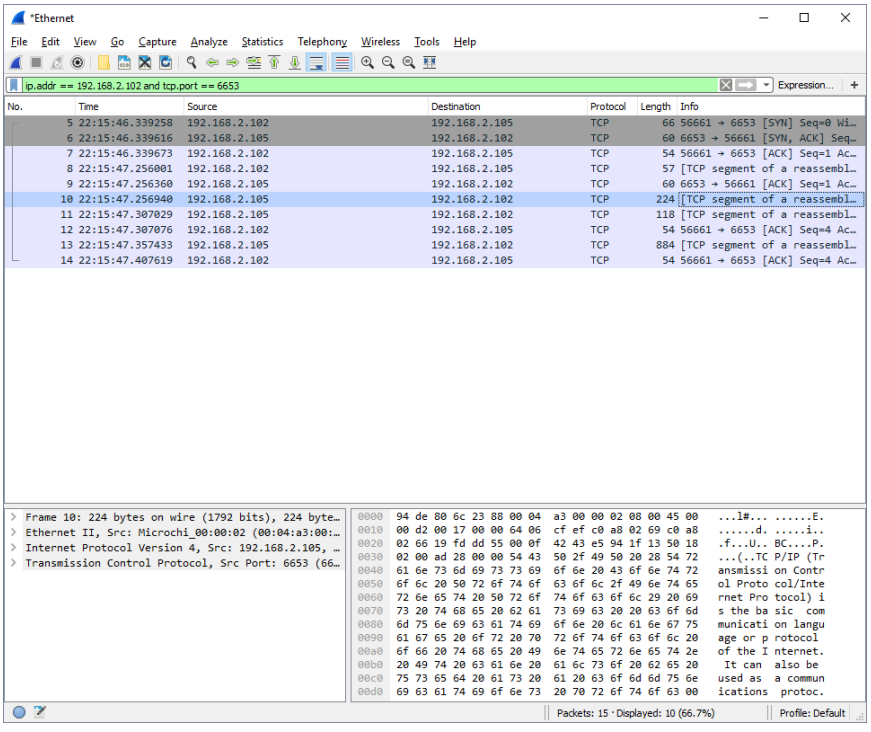

Figure 7: Three buffer transfer with 50 msec delay

Our first message is 224 bytes in length which again includes the overhead of the TCP and IPv4 headers. We can see that our actual message is contained in the packet if we view the ASCII print out at the bottom of the Wireshark output. The second byte occurs approx. 50 msecs after a length of 118 bytes. Last we see the reminder of our message sent in Packet 13. The Visual Basic application (as seen in Figure 8) confirms our last message was sent with a length of 830 bytes (829+1 with '\0' termination).

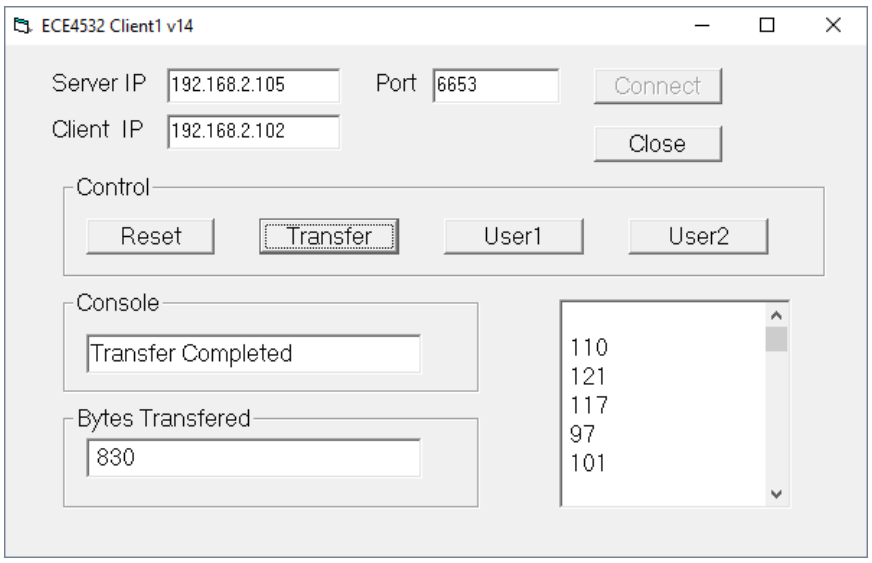

Figure 8: VB out: Three buffer transfer with 50 msec delay

Interestingly enough the client sends an ACK (Packet 12) back to the server after the transmission of the second message. An ACK message was not sent back after the transmission of Packets 10 and 11. We now investigate the effect of delaying the time between messages to see if we can determine why this behavior occurs.

Table 2: Three buffer transfer with Arbitrary Delays

Coded Server Delay Between Buffers	**Wireshark Observed Delay**		**Two Buffers Successfully Sent?**
	Packet 1-2	Packet 2-3	
0 msec	0.482 msec	0.335 msec	Yes
25 msec	25.149 msec	25.573 msec	Yes
50 msec	50.106 msec	50.508 msec	Yes
500 msec	449.746 msec	500.502 msec	Yes
1000 msec	1000.372 msec	1000.411 msec	Yes
2500 msec	2500.126 msec	2500.497 msec	Yes
5000 msec	5000.118 msec	5000.605 msec	Yes

Again at any delay interval the message will still eventually reach the client. The Visual Basic Application updates itself to reflect the latest message that has been

received. Also of note is that at values greater than 50 msecs we always see an ACK message between packet 1, 2 and 3. The Wireshark capture below shows the a experiment with a delay of 55 msecs. Note the ACKs between messages.

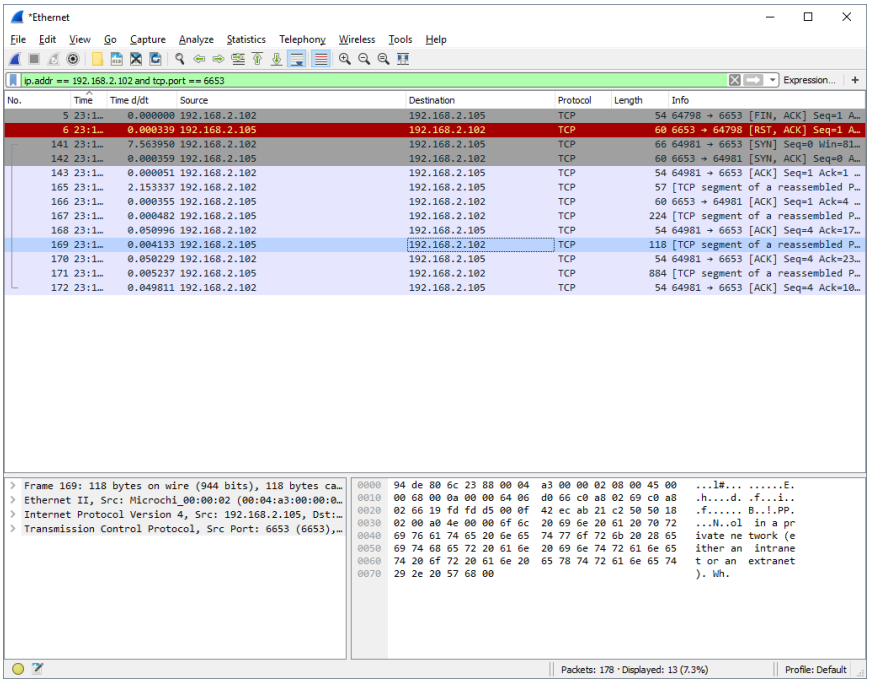

Figure 9: Three buffer transfer with 55 msec delay

Multiple 50 Byte Buffered Message Transmission

For our final experiment we run our server so that it sends our message parsed up into 50 byte chunks. The code for parsing the data into 50 byte sizes is seen in Code Snippet 4. Note we congregate the data upon running sending it instead of declaring 20 char arrays to store our message into the 50 bytes chunks. We use one 50 byte chunk *'tbfr1'* and transfer data into it using *'memcpy'*. We keep track of how many bytes we have sent thus far in a new int variable declared as *'bytesSent'*. The copying of data into different buffers induces a overhead between each transmission.

For our baseline experiment we run our server with a 50 msec delay between each message sent. The resulting Wireshark capture is shown below.

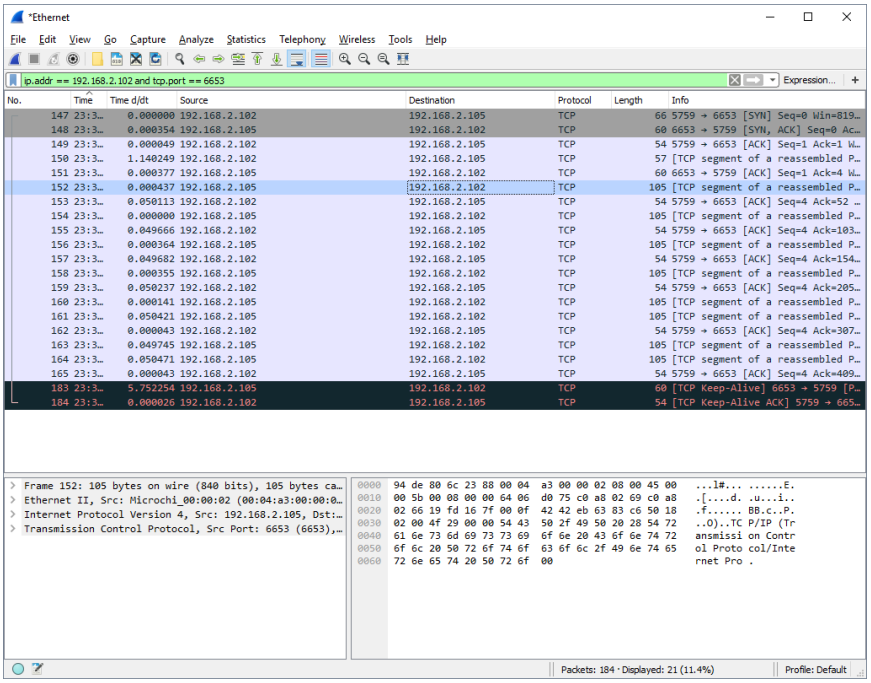

Figure 10: Multiple buffer transfer with 50 msec delay

The completion time between messages still stays around the expected 50 msecs. The overhead induced by copying the message during the send operation is negligible.

By parsing up the message into 50 byte blocks we expect to see around 22 messages being sent to the client. However we seem to have reached a limitation in the amount of packets a client will accept. After 8 consecutive message sends from the server we see no more messages in transmission. The limitation is not due to the delay in between message sends either. We experimented with sending messages with varying delays from 0 msecs to 5000 msecs and the same behavior is observed.

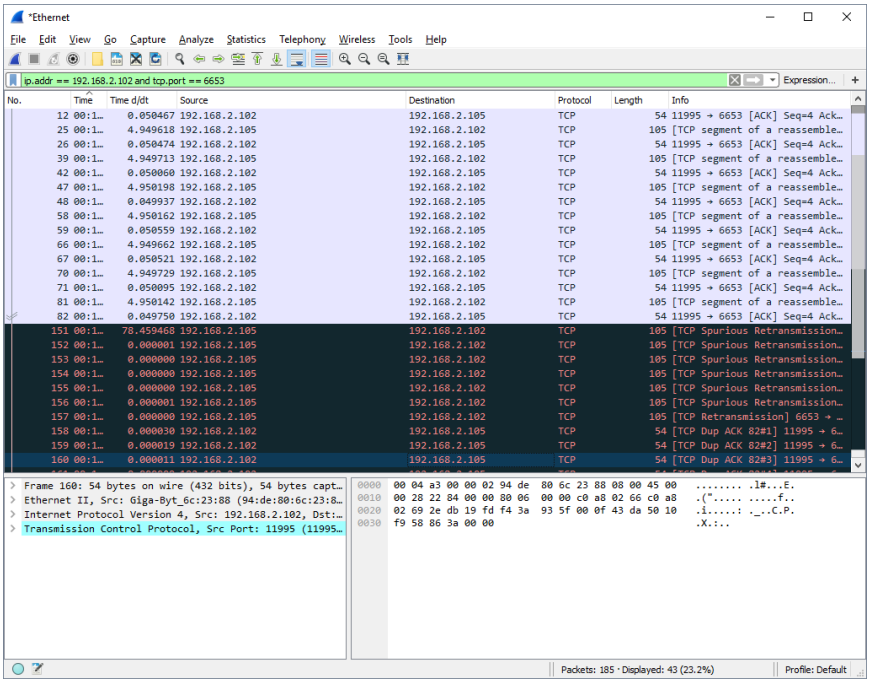

Figure 11: Multiple buffer transfer with 5000 msec delay

Behavior during the 5000 msec experiment showed the server responded to the client sending a “TCP Spurious Retransmission” or re-transmission of a previously sent frame [2]. From the Wireshark capture shown above, transmission packet 25 was sent and but then retransmitted in packet 151. The client responds to the server in packet 158 that what it received in packet 151 was a duplicate frame of packet 25.

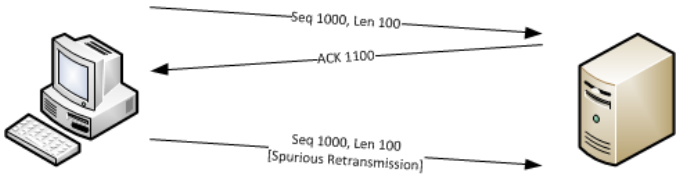

Figure 12: Demonstration of Spurious Retransmission [2].

The server believes that packet 25 was lost in transmission so it re-transmits the original message to the client. The client later tells the server the TCP message is a duplicate. After this conversation between client and server the server terminates transmission of the rest of the message.

III. Conclusion

What we learned in this lab experiment was the process of sending data across multiple message buffers and the limitations that come with it. In our first experiment we looked at sending our ~1060 byte message across two buffers which worked without hiccup. The second experiment expanded our message

across three buffers and again the message was received by the client without error. Even when introducing long delays between each buffer we saw that the client was able to decipher the message. With the results from these experiments we can ensure that even if there is a long transmission propagation time we can expect each frame to be acknowledged by the client correctly.

Behavior during the multiple 50 bytes transfer was a bit more concerning. First it was important to have your socket options set so that the PIC32 board would not wait for a full message buffer to send your message. Specifying *'TCP_NODELAY'* in the socket options function ensured our message was not buffered. However still we were receiving transmission errors since our entire never was successful in reaching the client. The effect was not dependent on delays between message transmissions either. The server appears to be confused about what message the client received and attempted re-transmission of the packets 1-8 of our 50 byte blocked data. Further work needs to be done to understand why transmission did not complete successfully between client and server. After modifying various parameters on the server side socket operations behavior of this problem could not be fixed.

IV. Appendix

```
    // If the received message starts with a second byte is
    // '84' it signifies a initiate transfer
    if(rbfr[1]==84){
        mPORTDSetBits(BIT_2);   // LED3=1
        // Send full message
        //
        send(clientSock, tbfr1, tlen1+1, 0);
        DelayMsec(50);
        send(clientSock, tbfr2, tlen2+1, 0);
        mPORTDClearBits(BIT_2);  // LED3=0
    }
```

Code Snippet 1

```
    // We store our desired transfer paragraph
    //
    char myStr[] = "TCP/IP (Transmission Control Protocol/Internet Protocol) is "
        "the basic communication language or protocol of the Internet. "
        "It can also be used as a communications protocol in a private "
        "network (either an intranet or an extranet). When you are set up "
        "with direct access to the Internet, your computer is provided "
        "with a copy of the TCP/IP program just as every other computer "
        "that you may send messages to or get information from also has "
        "a copy of TCP/IP. TCP/IP is a two-layer program. The higher "
        "layer, Transmission Control Protocol, manages the assembling "
        "of a message or file into smaller packets that are transmitted "
        "over the Internet and received by a TCP layer that reassembles "
        "the packets into the original message. The lower layer, "
        "Internet Protocol, handles the address part of each packet so "
        "that it gets to the right destination. Each gateway computer on "
        "the network checks this address to see where to forward the "
        "message. Even though some packets from the same message are "
        "routed differently than others, they'll be reassembled at the "
        "destination.\0";

    // Store my birthmonth
    //
    int BM = 4;

    // Chunk up our data
    //
    tlen = strlen(myStr) + 1;
    tlen1 = tlen*BM/15;
    tlen2 = tlen - tlen1;
    char tbfr1[tlen1 + 1];
    char tbfr2[tlen2 + 1];

    // Chunk up data into desired lengths
    //
    memcpy(tbfr1, myStr, tlen1);
    memcpy(tbfr2, myStr+tlen1, tlen2);

    // Null terminate the string array
    //
    tbfr1[tlen1] = '\0';
    tbfr2[tlen2] = '\0';
```

Code Snippet 2

```
// Store my birthmonth
//
int BM = 4;
int BD = 6;

// Chunk up our data
//
tlen = strlen(myStr) + 1;
tlen1 = tlen*BM/25;
tlen2 = tlen*BD/100;
tlen3 = tlen - tlen2 - tlen1;
char tbfr1[tlen1 + 1];
char tbfr2[tlen2 + 1];
char tbfr3[tlen3 + 1];

// Chunk up data into desired lengths
//
memcpy(tbfr1, myStr, tlen1);
memcpy(tbfr2, myStr+tlen1, tlen2);
memcpy(tbfr3, myStr+tlen1+tlen2, tlen3);

// Null terminate the string array
//
tbfr1[tlen1] = '\0';
tbfr2[tlen2] = '\0';
tbfr3[tlen3] = '\0';
```

Code Snippet 3

```
if(rbfr[1]==84){
    mPORTDSetBits(BIT_2);   // LED3=1
    bytesSent = 0;
    //sent = 0;
    while (bytesSent < tlen){
        memcpy(tbfr1, myStr+bytesSent, tlen1);
        if (bytesSent > 1049){
            tbfr1[tlen-bytesSent+1] = '\0';
            send(clientSock, tbfr1, tlen-bytesSent+1, 0);
        }
        else{
            tbfr1[tlen1] = '\0';
            // Loop until we send the full message
            //
            send(clientSock, tbfr1, tlen1+1, 0);
       }
        bytesSent += tlen1;
        DelayMsec(100);
    }
    mPORTDClearBits(BIT_2);  // LED3=0
}
mPORTDClearBits(BIT_0); // LED1=0
```

Code Snippet 4

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

V. References

[1] D. FITZGERALD and D. WAKABAYASHI, "Apple Quietly Builds New Networks," Wall Street Journal, 3 February 2014. [Online]. Available: http://www.wsj.com/news/articles/SB10001424052702304851104579361201655365302. [Accessed 21 February 2016].

[2] J. BONGERTZ, "Spurious Retransmissions," 6 June 2013. [Online]. Available: https://blog.packet-foo.com/2013/06/spurious-retransmissions/comment-page-1/. [Accessed 21 February 2016].

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