Stop and Wait Data Communication Protocol

Devin Trejo
devin.trejo@temple.edu

April 29, 2016

1 Summary

The final lab of the semester goes over the Go Back N protocol implemented using a PIC32 MCU and a Windows desktop computer. The Go Back N protocol is efficient in that it keeps the transmission medium occupied with packets even though it has not received acknowledgments of all packets. We show an implementation of the Go Back N protocol in C and test the behavior for when the client fails to send back an acknowledgment.

2 Introduction

The Go Back N protocol behavior is shown in figure 1. In essence the protocol calls for a sever to send packets to a client non-stop. After the first packet is sent out, the server starts a timer which relates to a successfully sent packet. The timer is stopped if an ACKn is received from a client stating it successfully received the transmitted packet. If the timer expires, it interrupts the main sending program and restarts the transmission of all packets sequentially starting with the packet which the server failed to receive an ACK for.

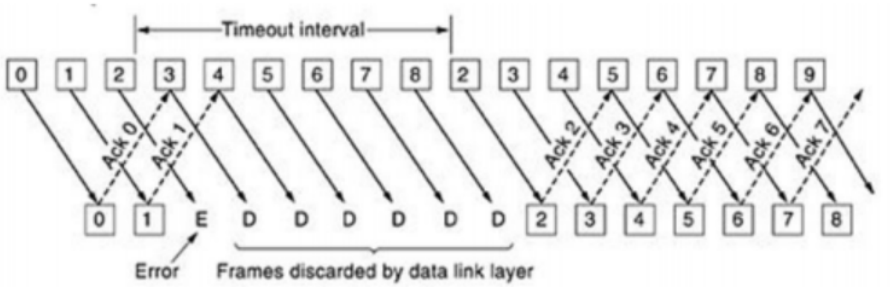

Figure 1: Go Back N Protocol

In this lab we will implement the Go Back N protocol using a PIC32 MCU. To effectively keep track of what ACKn we have sent or are waiting for we implement a Queue data structure in C. A Queue is a list of items which follow the first in first out (FIFO) behavior. We will use Queues to keep track of what sequence numbers we have received/sent already.

3 Discussion

For testing we construct a overall message spread across 26 DataPackets. Each packet will contain 16 bytes of which will be a corresponding letter of the alphabet. For example, the first DataPacket will contain a sequence number of 1 and contain 16 'A' characters within the data portion of the DataPacket.

3.1 Test 1: Go Back N Demonstration

To begin we test our system where there is no probability of error in the original data frame being sent nor the ACKn being sent. We expect the system to send the first $FRAMEDELAY$ packets before we receive our first response. For all these tests $FRAMEDELAY = 3$.

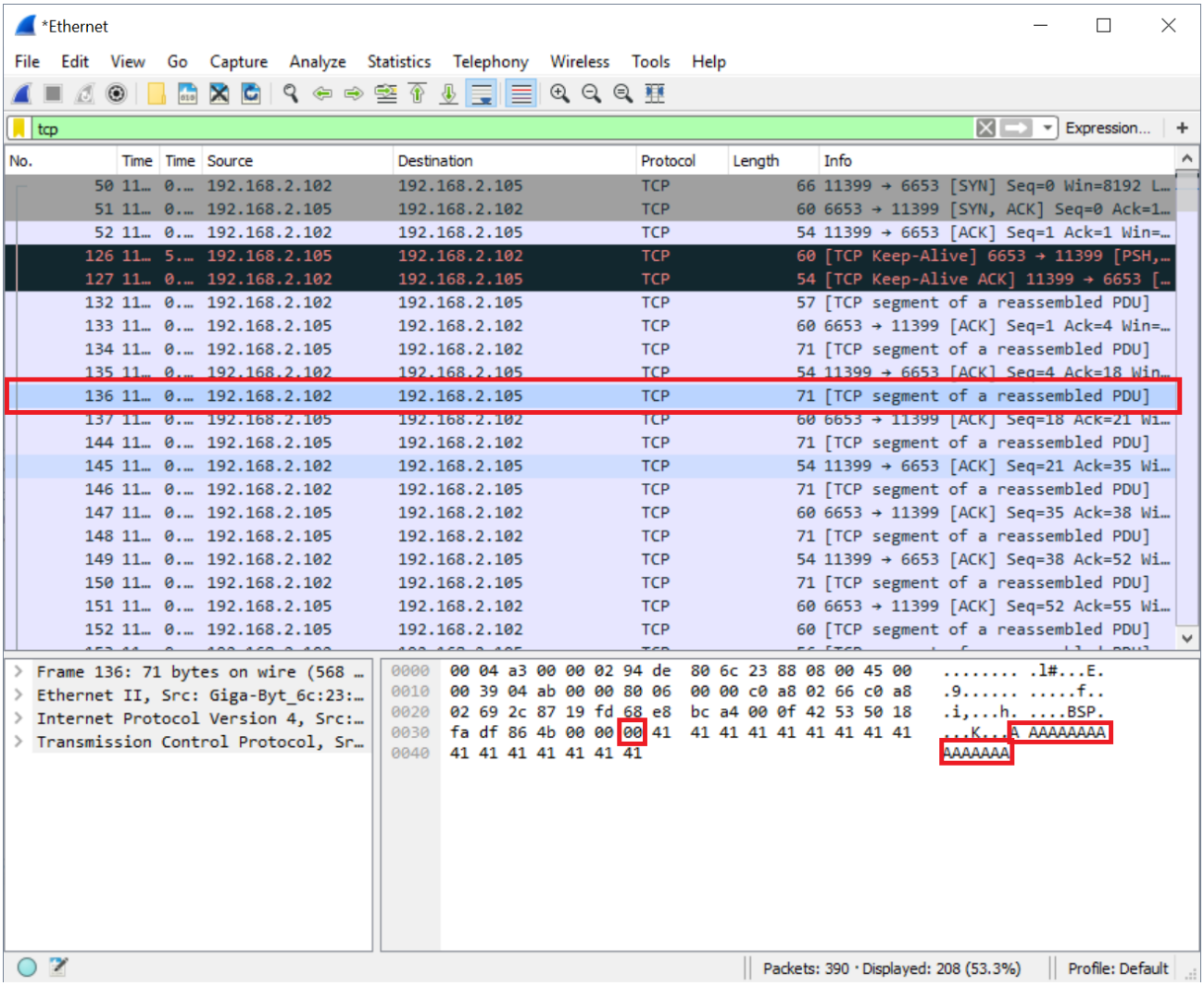

Figure 2: Test1 - First DataPacket Sent

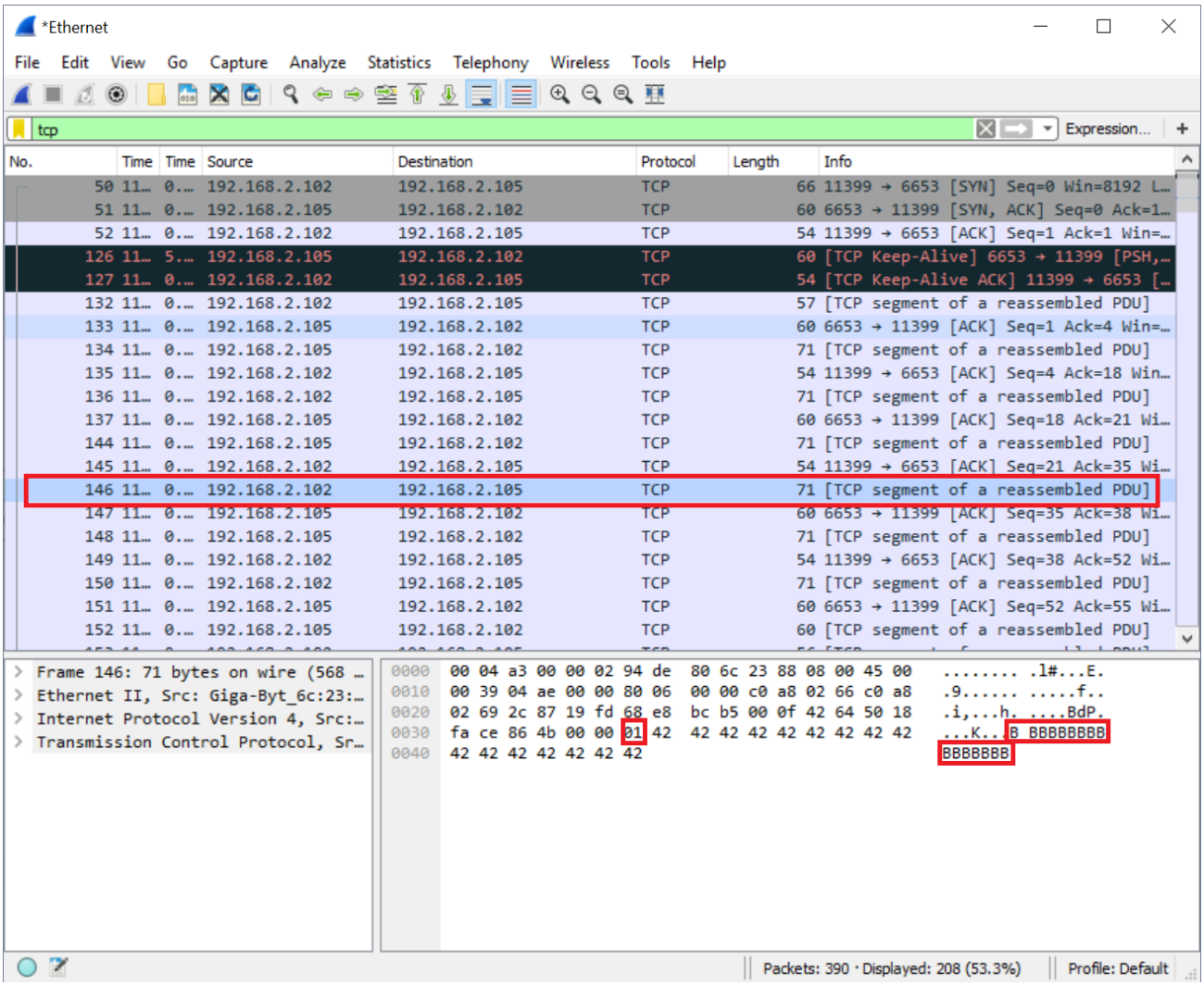

Figure 3: Test1 - Second DataPacket Sent

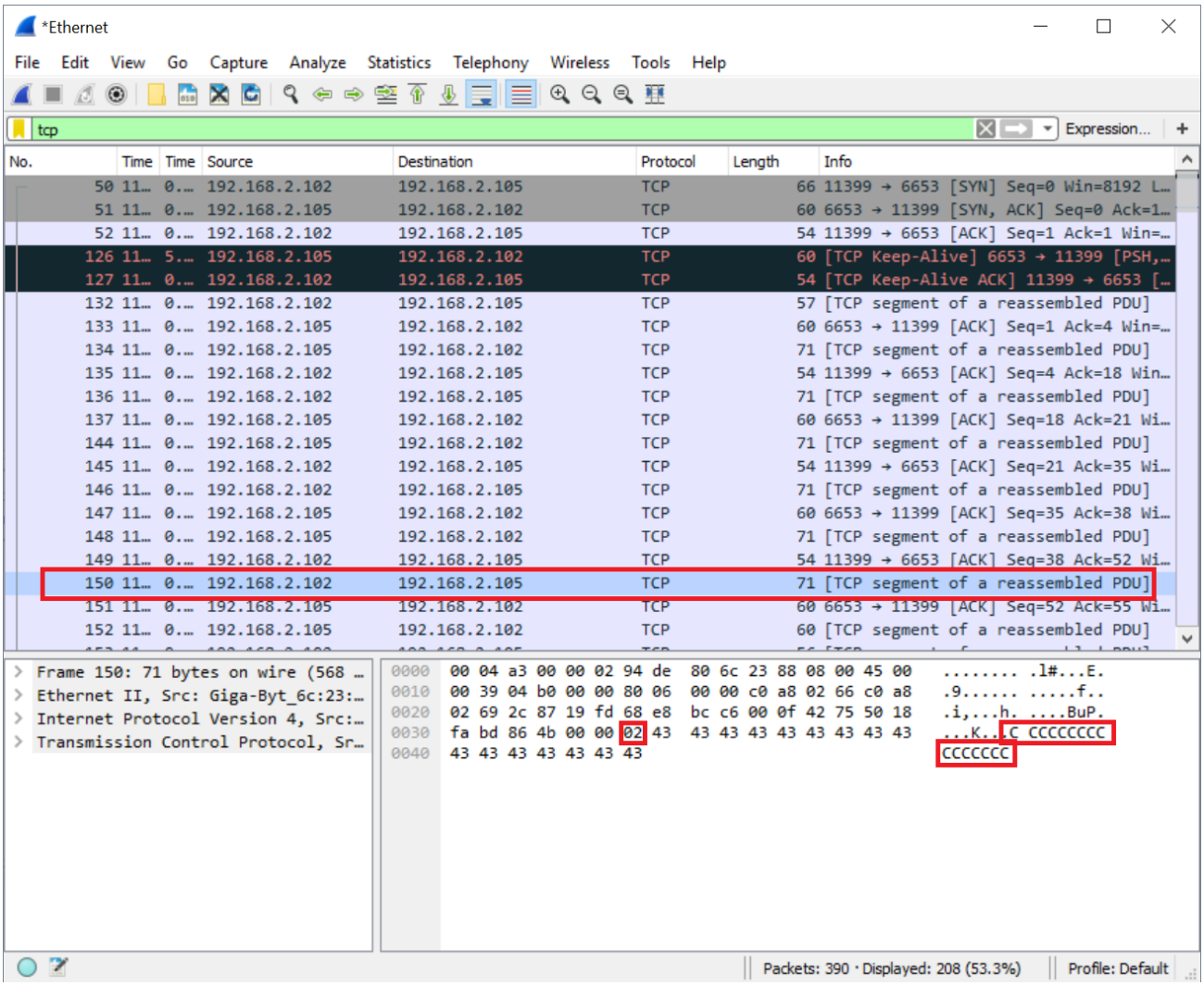

Figure 4: Test1 - Third DataPacket Sent

As can be seen in figures 2,3, and 4, our first three packets are sent out by our server. Meanwhile the client is receiving this packets and filling up a receive buffer queue. Once the receive buffer queue has reach minimum size, the first ACKn packet is sent back for the sequence number corresponding to the first data packet sent out. In this case the ACKn will acknowledge sequence number 0 as seen in figure 5.

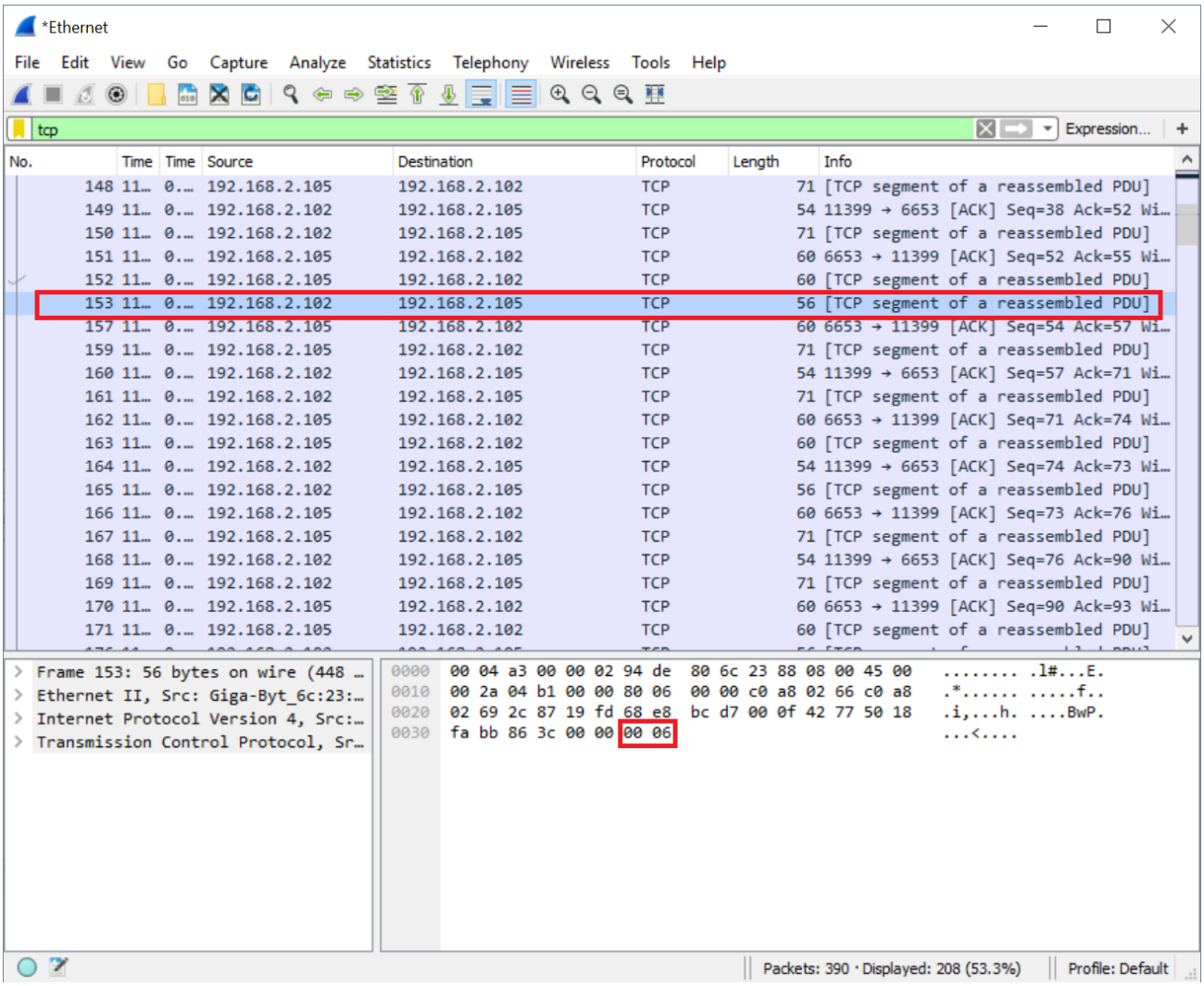

Figure 5: Test1 - Ackn Sent Back for Seq 0

Next we will see a ping-pong effect where we send one DataPacket and receive a ACKn for the last $SequenceNumber - FRAMEDELAY = 0$ packet.

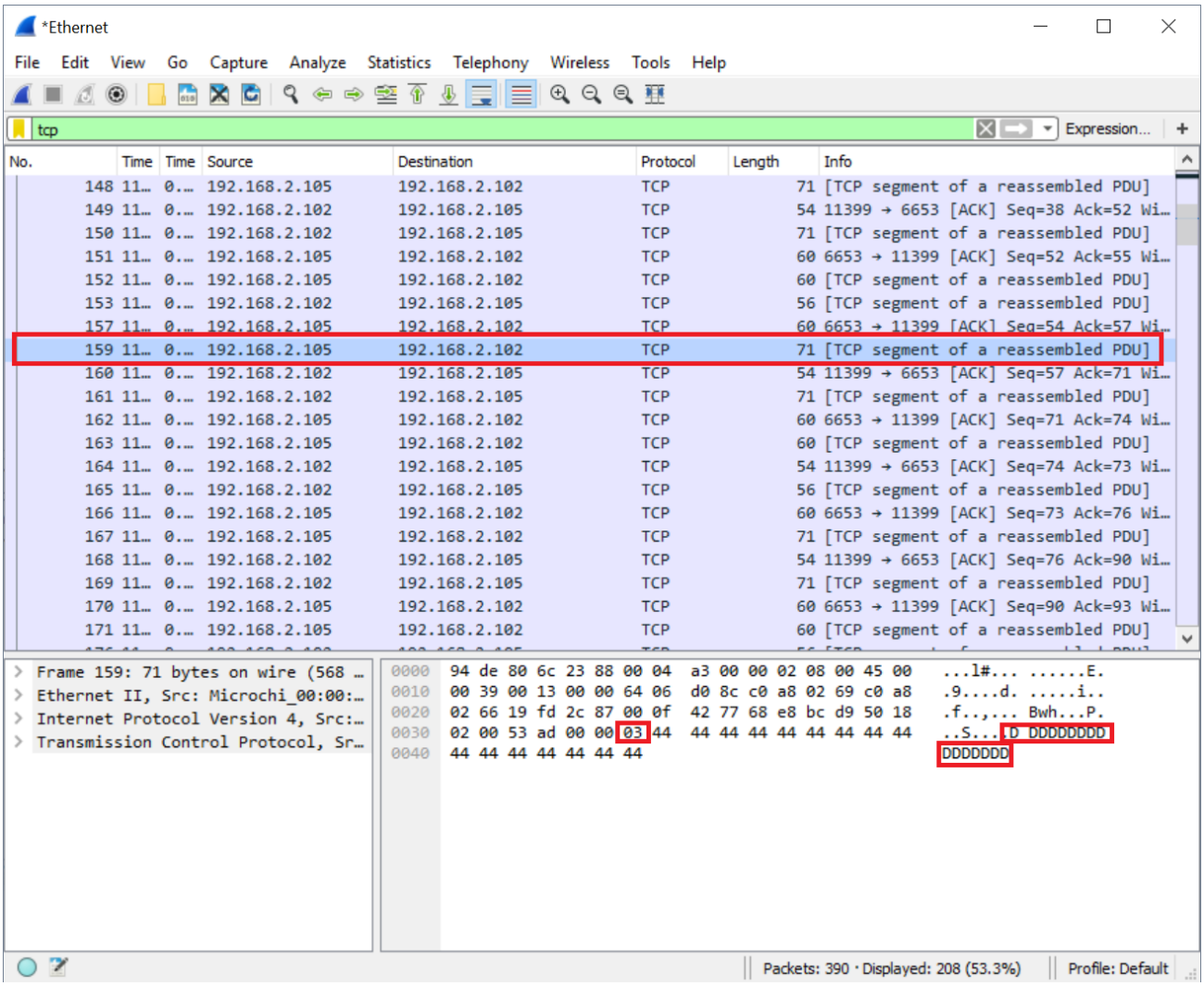

Figure 6: Test1 - Forth Data Packet Sent

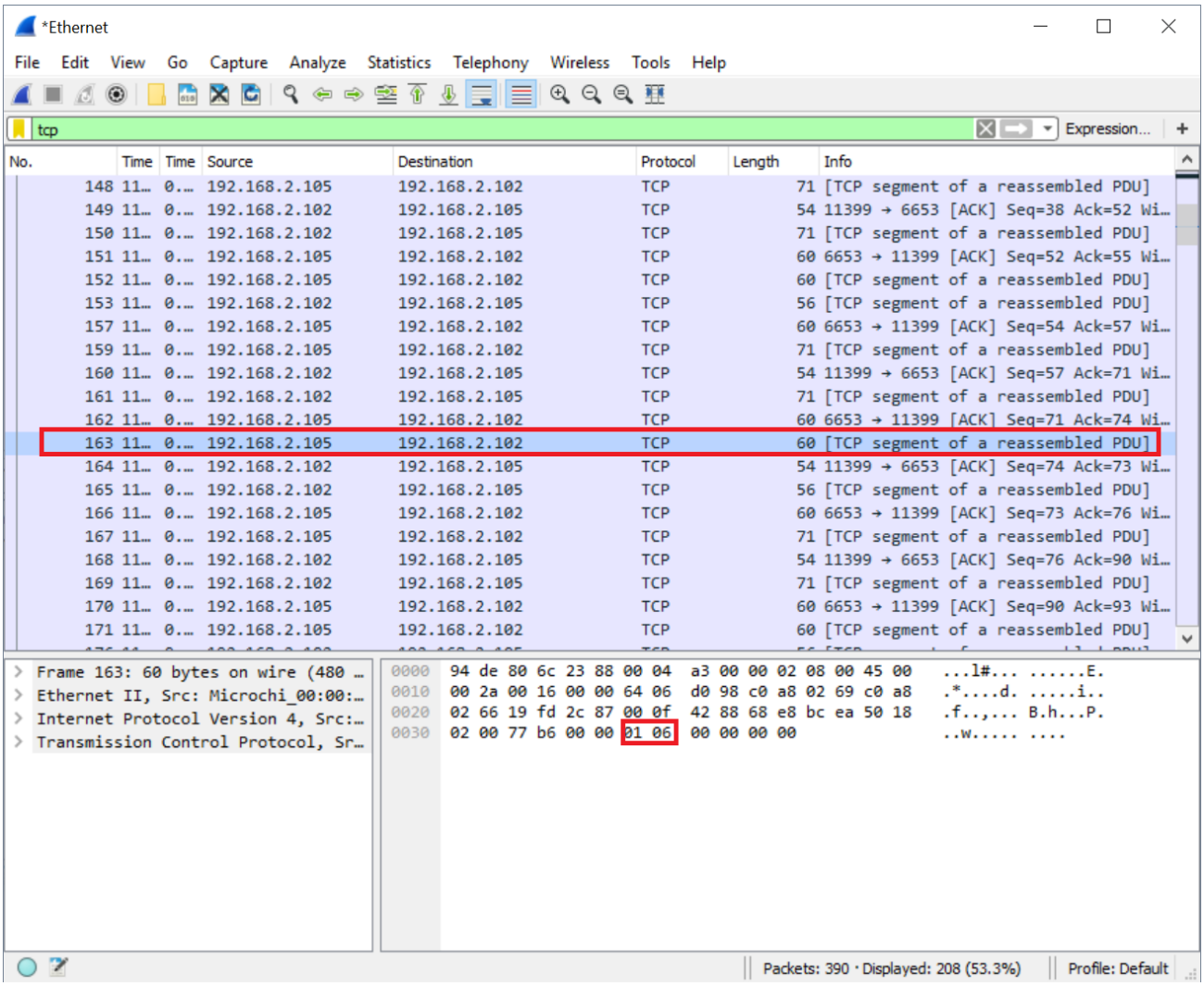

Figure 7: Test1 - Ackn Sent Back for Seq 1

At some point we expect a sequence number roll over. The design criteria calls for a sequence roll over at $Seq = 15 = 0x0f$. The following WireShark capture demonstrates a successful sequence number roll-over.

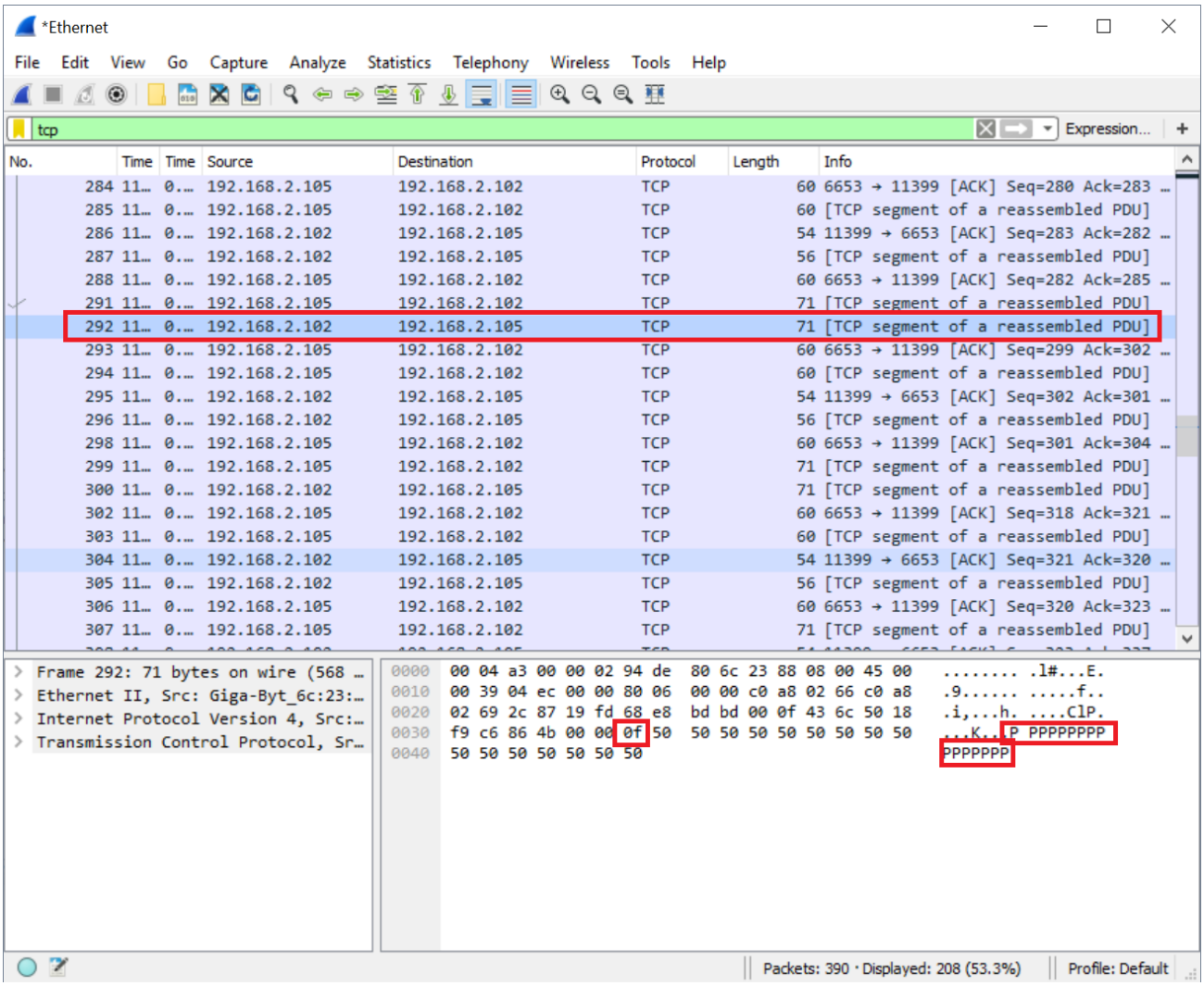

Figure 8: Test1 - Data Packet with Sequence 15

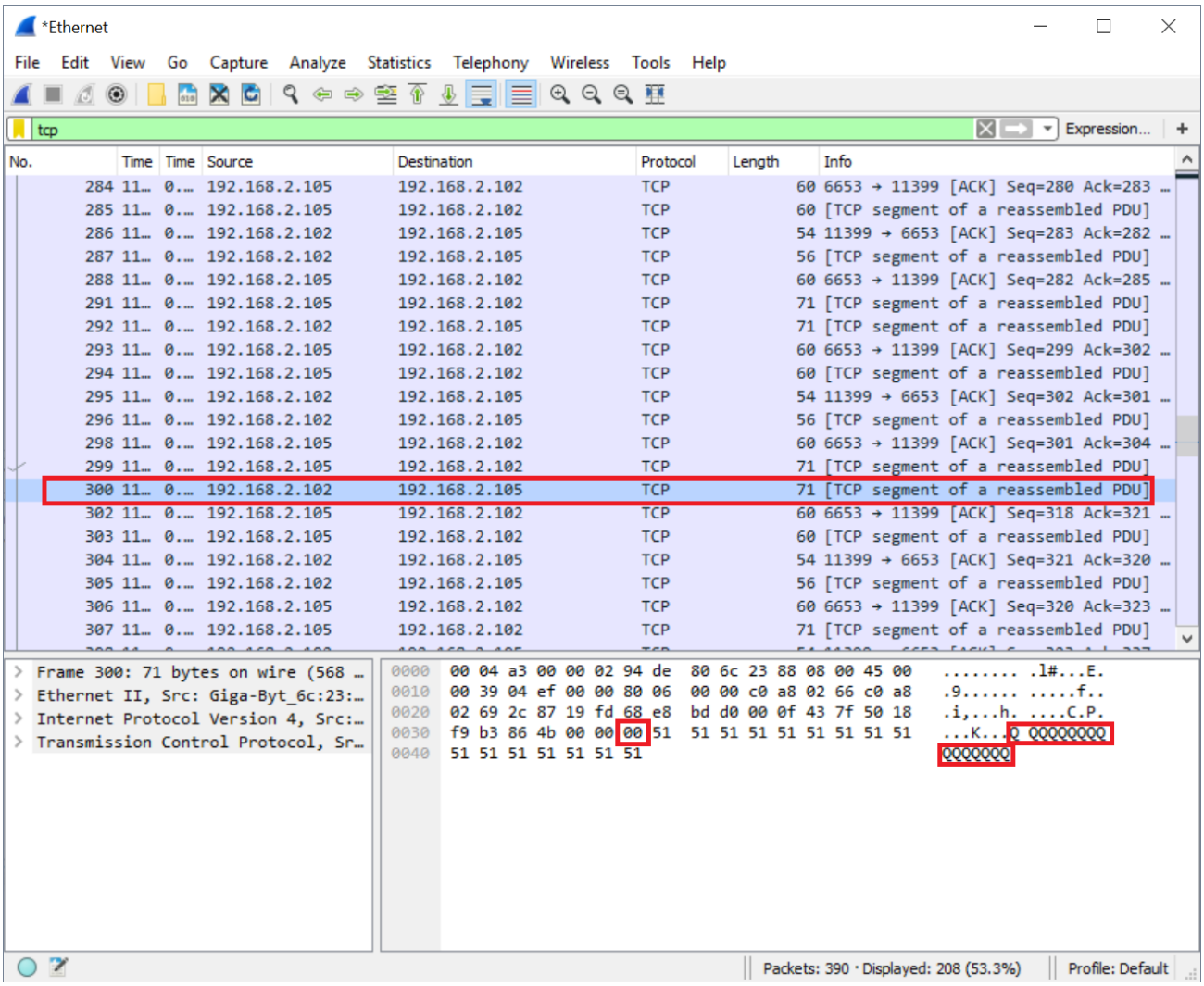

Figure 9: Test1 - Data Packet with Sequence 00 (Rollover)

3.2 Test 2: Go Back N Frame Corruption

What happens if the frame sent is never received by the client? This test demonstrates the capabilities of Go Back N by retransmitting a lost frame after a timer has expired. Since the test platform is a perfect system with no possibility of transmission error we introduce some random probability that our test will fail to send a frame packet. Also note for this test $FRAMDELAY = 5$, $LENM(Sequence\#Max) = 15 = 0x0f$, $PROBSENTERR = 0.5$

As in section 3.1, we expect that the first $FRAMEDELAY$ packets to be sent before the first ACKn. Since we introduce the probability of error that the frame is never originally sent, we see DataPacket containing sequence number 02 or all Bs is never sent.

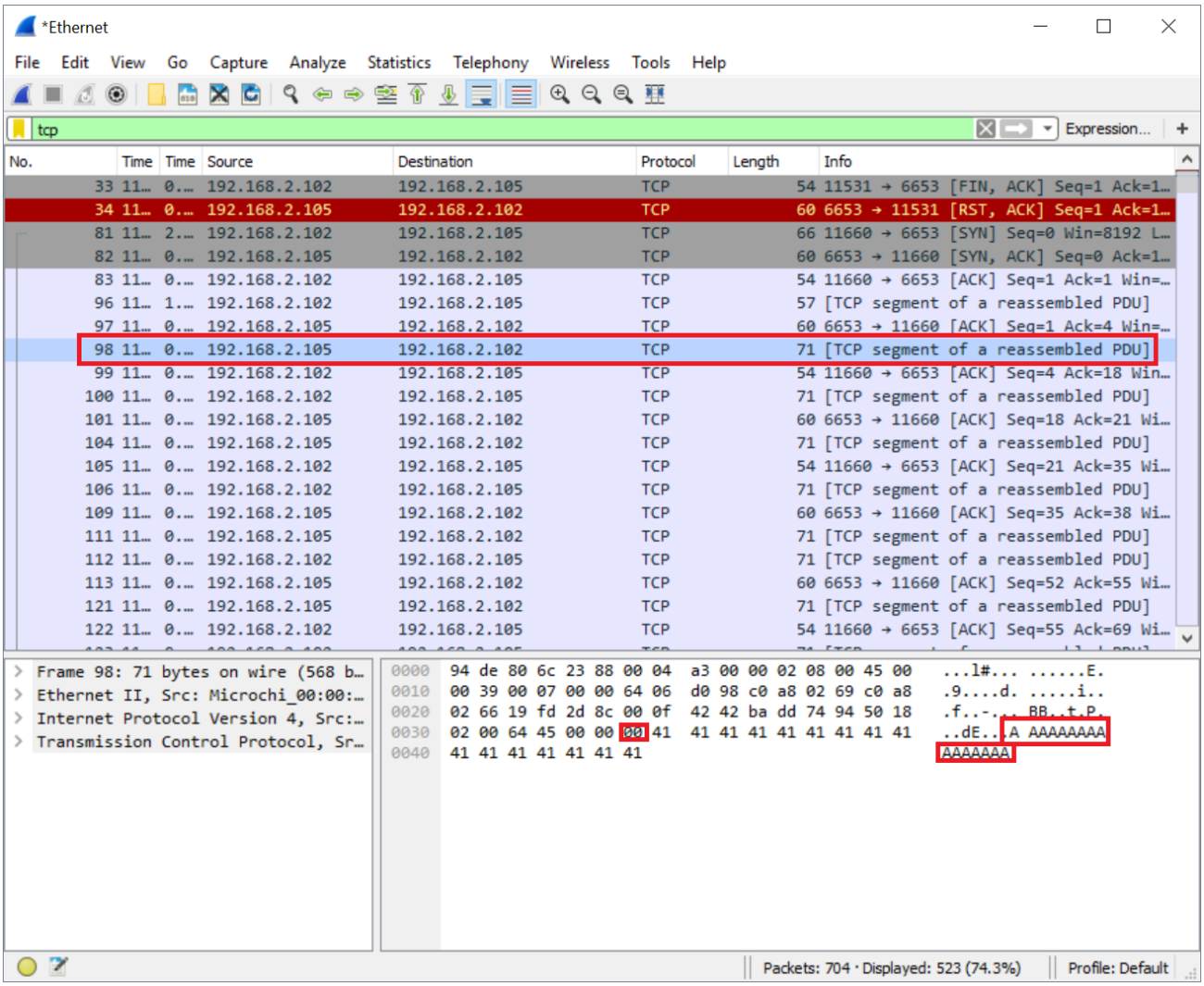

Figure 10: Test2 - Data Packet with Sequence 00 Sent First

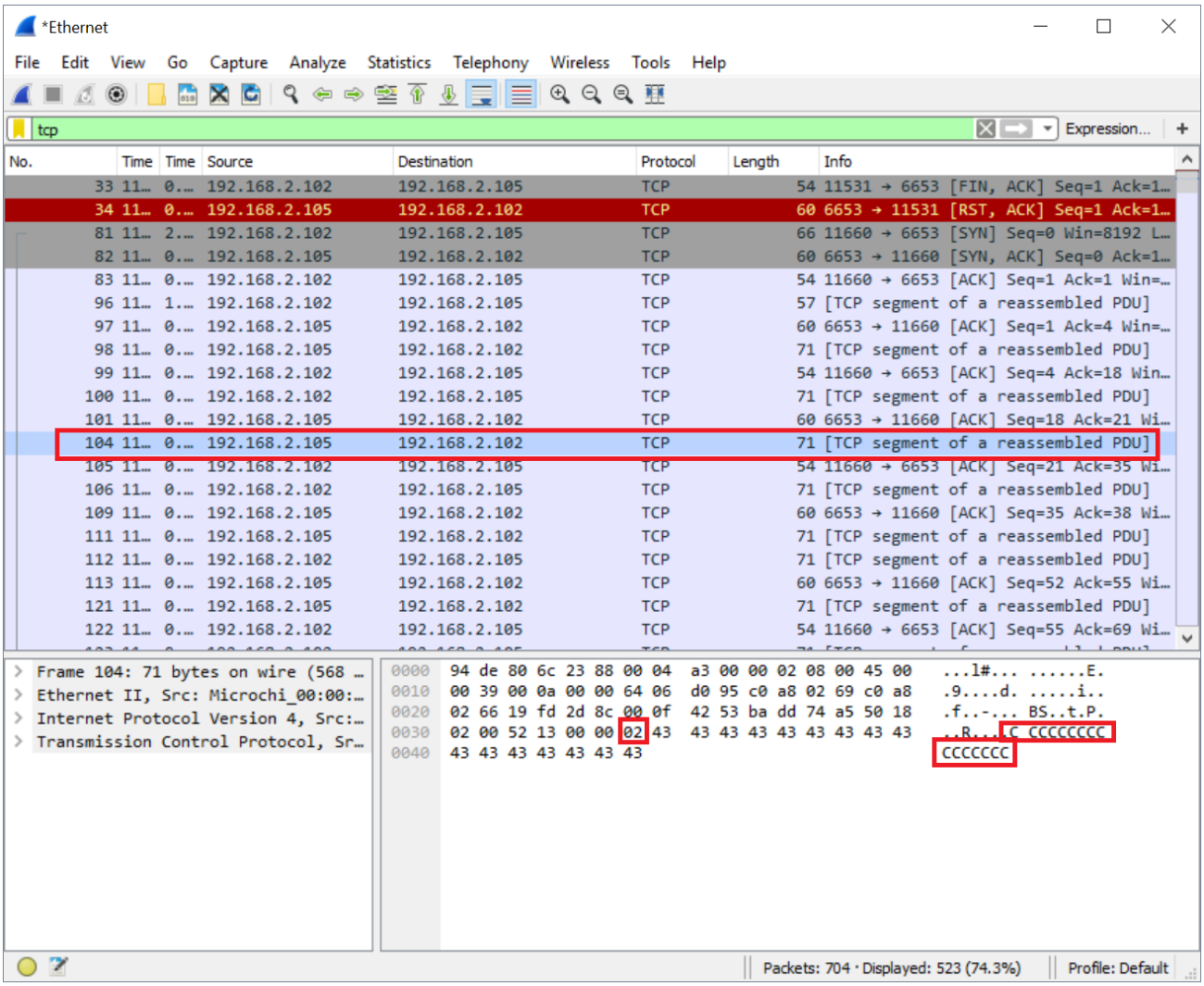

Figure 11: Test2 - Data Packet with Sequence 03 Sent Second

We can see that our server continues to send data packets up to sequence number 11 or 0x0b before it realizes it never received an ACKn for any sent packets. It then goes back N frames and resends all frames it hasn't received a ACKn for. It starts with DataPacket containing all As.

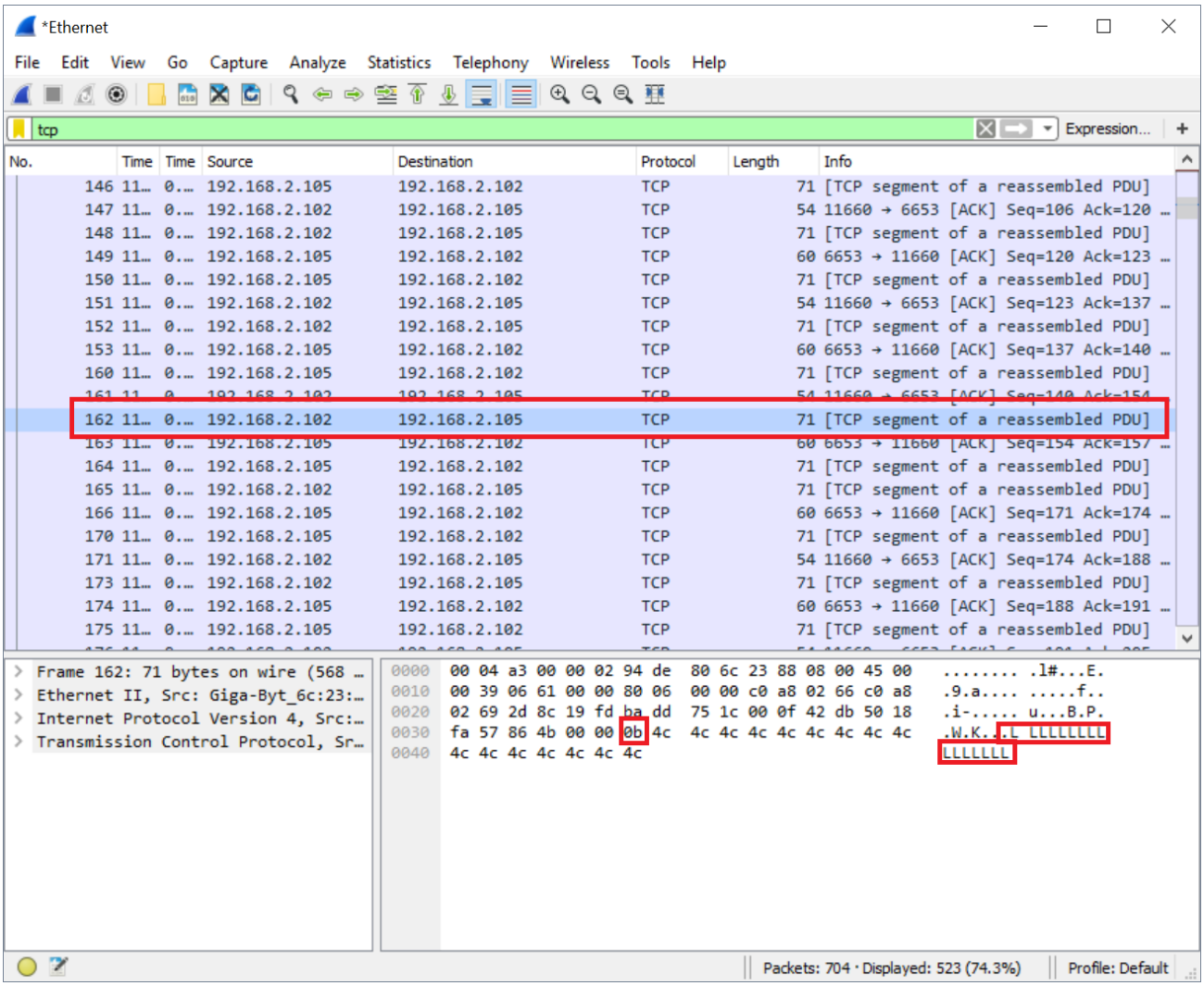

Figure 12: Test2 - Data Packet with Sequence 11 Sent

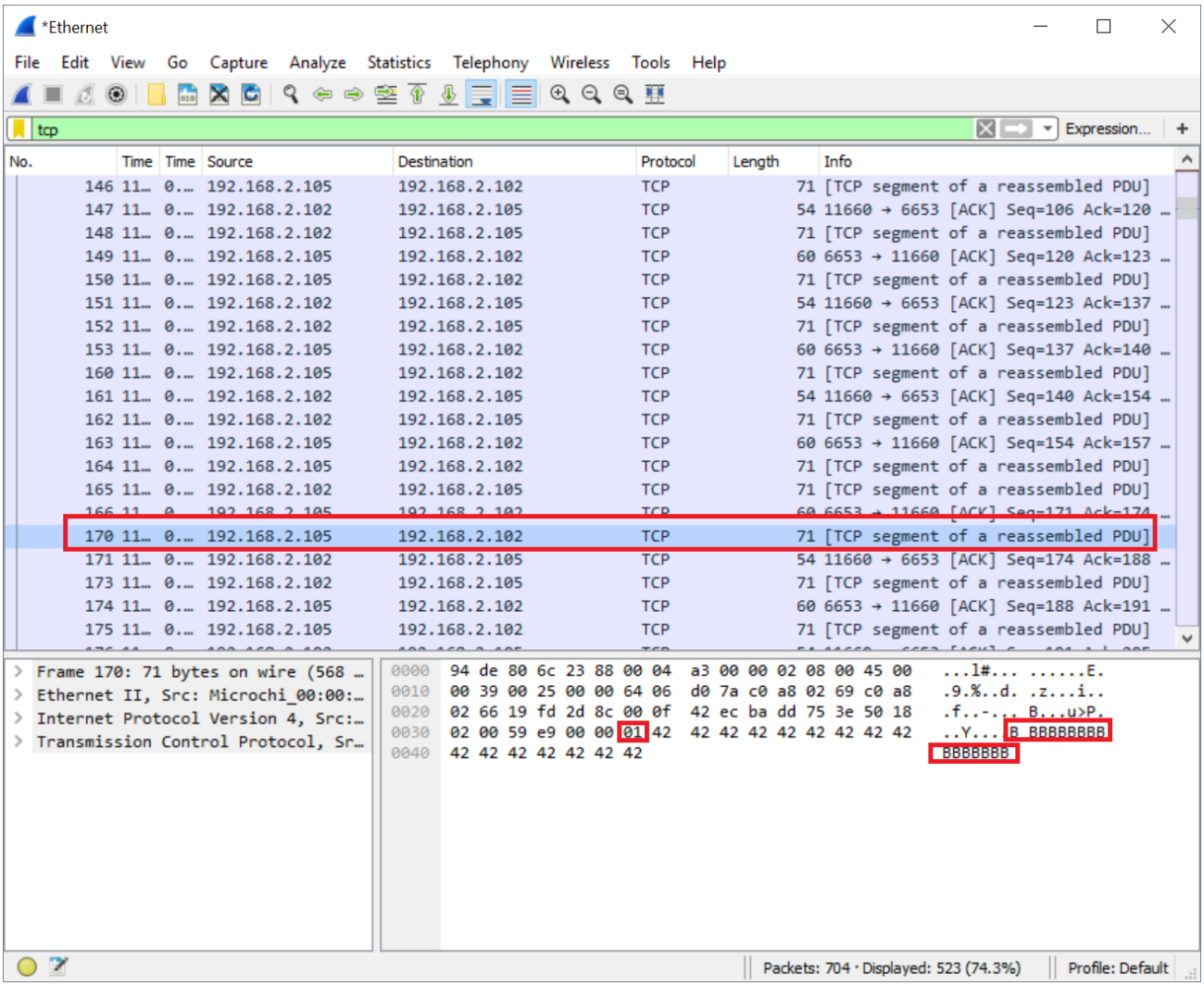

Figure 13: Test2 - Re-Transmission of Data Packet with Sequence 00

Finally, we see the client side acknowledge it received DataPacket with sequence 00 in figure 14.

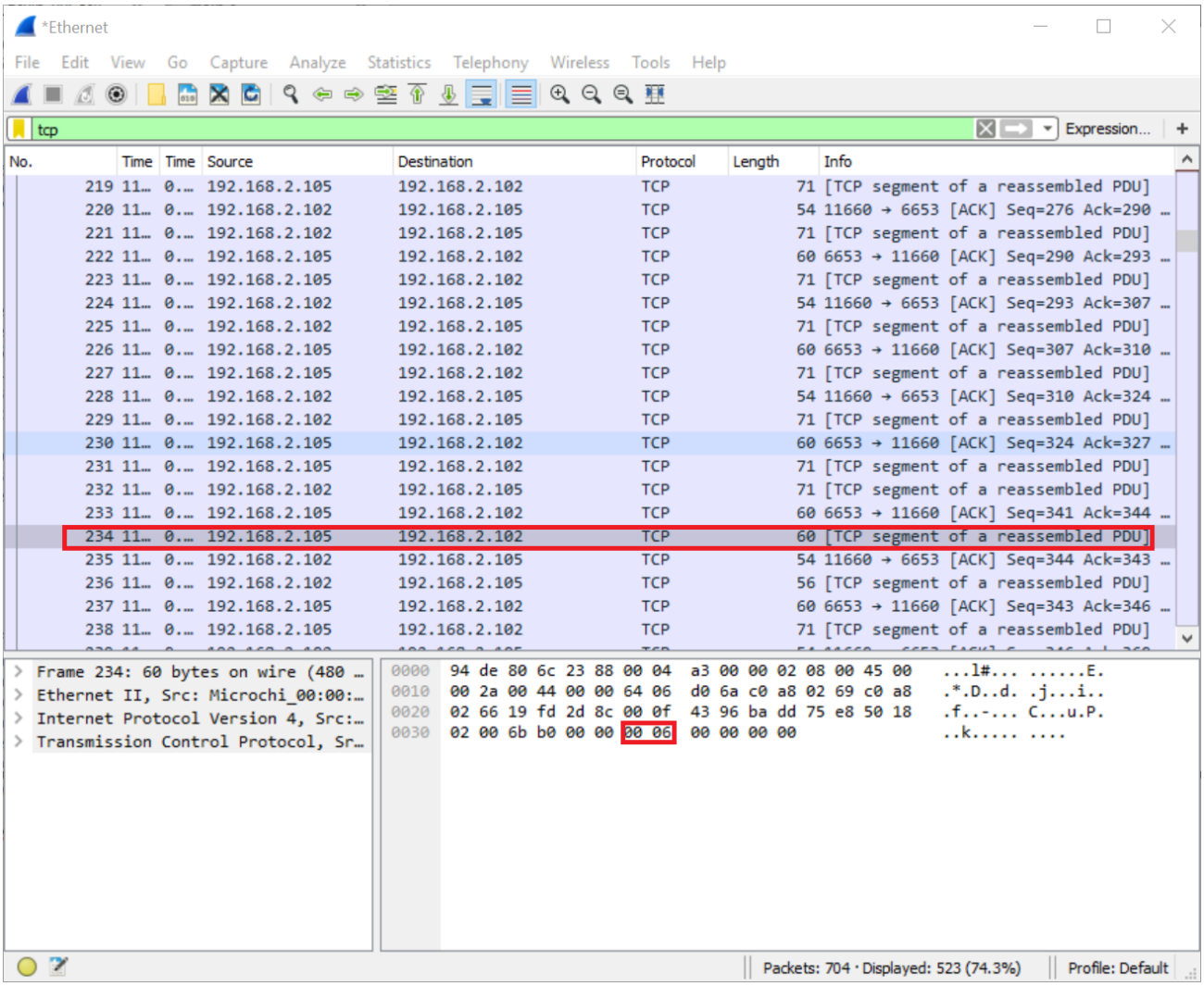

Figure 14: Test2 - Acknowledgment of DataPacket with Sequence 00

4 Conclusion

In conclusion we show the efficiency of Go Back N in how it is always keeping the transmission line occupied with new data. The Go Back N technique shares similar properties shown in the last lab's sliding window, except instead of waiting for a group of ACKn to be received before sending the next P packets we only wait for an ACKn from the oldest packet sent before sending the next frame. Both implementations work off a timer that interrupts normal operation when it detects some ACKn has not been received for some time.

Overall, both techniques are worth-wild with the Go Back N being slight more efficient. In our implementation we discarded any frames we did not receive in sequential order. A more effective approach would be to selectively request a frame from the client side that has no been received.

Appendix

Main program source code available on my GitHub:
`https://github.com/dtrejod/myece4532/blob/master/lab6`

Listing 1: DataPacket — ACK-Packet Data Structure & Queue Structure

```
#pragma pack(1)

// ACK Struct
typedef struct myACK
{
    uint8_t sequence;
    char ackChar;
} myACK;

// WARNING IF myDataPacket length and myACKs are multiples of one another,
// the packet detection algo will fail.
typedef struct myDataPacket
{
    uint8_t sequence;
    char data[DATALEN];
} myDataPacket;
#pragma pack(0) // turn packing off

typedef struct Queue
{
        int capacity;
        int size;
        int front;
        int rear;
        int *elements;
} Queue;
```

Listing 2: Queue Functions

```
Queue * createQueue(int maxElements); // Create a new queue data struct
void Dequeue(Queue *Q); // Remove the element at the front of the queue
int front(Queue *Q); // Peak at what is at the front of the queue
int rear(Queue *Q); // Peak at what is at the end of the queue
void Enqueue(Queue *Q, int element); // Queue object to end of queue
void clearQueue(Queue *Q); // Remove all elements from Queue
```

Listing 3: Go Back N - Global Reset Handler

```
// Check to see if message begins with
// '0271' signifying message is a global reset
// We use this as a signal to start the lab
// experiment.
```

```
if ((testStarted == 0) && (rbfrRaw[0] == 02) &&
        (rbfrRaw[1] == 71))
{
    // Reset Sequence Number
    tbfrSeqTracker = 0;

    // Reset total msg sent counter
    msgSent = 0;
    endMsg=0;
    testStarted = 1;

    // reset delay counts
    transDelayCount = 0;
    ackDelayCount = 0;

    // Check for seq rollover
    if(tbfrSeqTracker > LENM)
    {
        tbfrSeqTracker = 0;
    }

    // Populate sequence number
    tbfrData[msgSent].sequence =
        tbfrSeqTracker++;

    // Copy tbfrData over to tbfr and keep track
    // of how much msg has been sent thus far
    tbfr = tbfrData[msgSent++];

    mPORTDClearBits(BIT_0);
    mPORTDSetBits(BIT_2); // LED3=1
    send(clientSock, &tbfr, sizeof(myDataPacket), 0);
    mPORTDClearBits(BIT_2); // LED3=0
    DelayMsec(100);

    // Mark frame as sent by queuing up sequence in ACK
    // awaiting response.
    Enqueue(tbfrAckQueue, tbfr.sequence);
```

Listing 4: Sliding Window - Data Packet Detector and Handler

```
// Check what time of message was revived based on its
// size
// Check if received is an myDataPacket
if (rlen%sizeof(myDataPacket)==0)
{
    // Convert the received data into a dataPacket
    // struct
```

```
    rbfrData = (myDataPacket *) rbfrRaw;

    // Store sequence in receive Queue if it is
    // next in Queue

    // Check for start of tranmission OR
    // Check for sequential sequence number OR
    // Check for sequential sequence rollover
    if(rbfrSeqTracker == (rbfrData->sequence))
    {
        rbfrSeqTracker++;
        // Check for seq rollover
        if(rbfrSeqTracker > LENM)
        {
            rbfrSeqTracker = 0;
        }
        Enqueue(rbfrDataQueue, rbfrData->sequence);
    }

    // If FRAMEDELAY Equal to size
    if(rbfrDataQueue->size == FRAMEDELAY || endMsg == 1)
    {
        // Send ACK for Random number of packets
        if (randMToN(0.0,1.0) >= PROBACKERR)
        {
            rbfrAck.sequence = front(rbfrDataQueue);
            rbfrAck.ackChar = 0x06;
            mPORTDClearBits(BIT_0);
            mPORTDSetBits(BIT_2); // LED3=1
            send(clientSock, &rbfrAck,
                sizeof(myACK), 0);
            mPORTDClearBits(BIT_2); // LED3=0
            DelayMsec(100);
        }

        // Remove the sent ACK from receive Q
        Dequeue(rbfrDataQueue);
    }
}
```

Listing 5: Go Back N - ACK Packet Detector and Handler

```
// Check if received is an myACK
else if (rlen%sizeof(myACK)==0)
{
    // Convert the received data into a myAck struct
    tbfrAck = (myACK *) rbfrRaw;
```

```
    // Check if ACK
    if(tbfrAck->ackChar == 0x06)
    {
        if(front(tbfrAckQueue) == (tbfrAck->sequence))
        {
            Dequeue(tbfrAckQueue);
            ackDelayCount = 0;
        }
        // Check if end of expirment
        if (tbfrAckQueue->size == 0 && endMsg == 1)
        {
            testStarted = 0;
        }
    }
}
```

Listing 6: Go Back N - Send Next Frame (On Timer)

```
// Check if time to send another DataPacket and if
// we have more msg to send.
if (transDelayCount*10 > TRANSMISSIONDELAY && msgSent < MSGLEN)
{
    // reset delay counts
    transDelayCount = 0;

    // Check for seq rollover
    if(tbfrSeqTracker > LENM)
    {
        tbfrSeqTracker = 0;
    }

    // Populate sequence number
    tbfrData[msgSent].sequence =
        tbfrSeqTracker++;

    // Copy tbfrData over to tbfr and keep track
    // of how much msg has been sent thus far
    tbfr = tbfrData[msgSent++];

    // Send FRAME with random error change
    if (randMToN(0.0,1.0) >= PROBSENTERR)
    {
        mPORTDClearBits(BIT_0);
        mPORTDSetBits(BIT_2); // LED3=1
        send(clientSock, &tbfr, sizeof(myDataPacket), 0);
        mPORTDClearBits(BIT_2); // LED3=0
        DelayMsec(100);
    }
```

```
    // Mark frame as sent by queuing up sequence in ACK
    // awaiting response.
    Enqueue(tbfrAckQueue, tbfr.sequence);

    // Check to see if we hit FRAMEDELAY Limit. If so we
    // start the ackDelay Count (or in other words don't reset)
    if (tbfrAckQueue->size <= FRAMEDELAY)
    {
        ackDelayCount = 0;
    }

    // Check to see if end of tranmission
    if (msgSent == MSGLEN) endMsg = 1;
}
```

Listing 7: Go Back N - ACK Timeout

```
// Check for ACK timeout
else if (ackDelayCount*10 > ACKTIMEOUT)
{
    // reset delay counts
    transDelayCount = 0;
    ackDelayCount = 0;

    // Reset msgSent tracker
    msgSent = msgSent - tbfrAckQueue->size;

    // Clear sent queue
    clearQueue(tbfrAckQueue);

    // Copy tbfrData over to tbfr and keep track
    // of how much msg has been sent thus far
    tbfr = tbfrData[msgSent++];

    // Reset seq tracker
    tbfrSeqTracker = tbfr.sequence+1;

    // Send FRAME with random error change
    if (randMToN(0.0,1.0) >= PROBSENTERR)
    {
        mPORTDClearBits(BIT_0);
        mPORTDSetBits(BIT_2); // LED3=1
        send(clientSock, &tbfr, sizeof(myDataPacket), 0);
        mPORTDClearBits(BIT_2); // LED3=0
        DelayMsec(100);
    }
```

```
    // Mark frame as sent by queuing up sequence in ACK
    // awaiting response.
    Enqueue(tbfrAckQueue, tbfr.sequence);
}
```