Stop and Wait Data Communication Protocol

Devin Trejo
devin.trejo@temple.edu

April 15, 2016

1 Summary

Today we introduce the stop-and-wait sliding window transmission method and compare the performance for difference values of window size and sequence size. We find that a window size that is too small decreases performance but having a window size that is too big introduces the probability that errors will occur during transmission. From testing multiple parameters we find that a windows size of 04 produces the results with the fastest transmission for our 26 packet long test message.

2 Introduction

2.1 Sliding Window Background

For this lab we will analyze and implement a sliding-window transfer protocol between a PIC32 Server and Client application. Sliding window flow control is an effective transfer method in which the receiver will have a window of (**P**) frames that may be transmitted to a receiver. On the receiver, we send ACK packets back to the sender corresponding to a sequence number (**M**) seen in the header of received DataPacket. As ACKs come back into our reviver we move our window frame edge forward. If a DataPacket is lost in transmission or is corrupted once it reaches the receiver, the sender will attempt to re-send the DataPacket if no ACK is receiver within a timeout period.

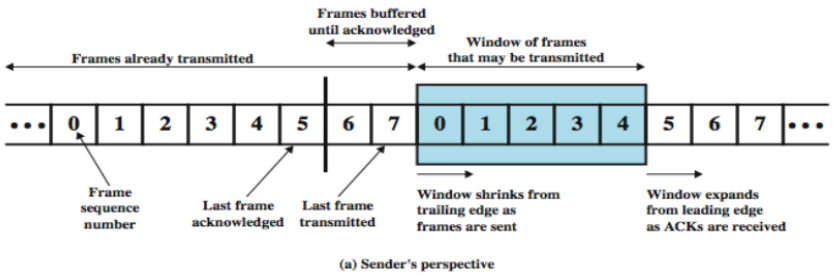

Figure 1: Sliding Window Flow Control Sender

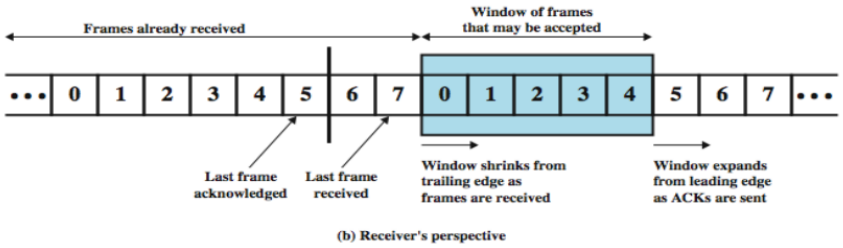

Figure 2: Sliding Window Flow Control Receiver

2.2 Sliding Window Implementation

For this lab we implement a stop-and-wait sliding window approach which means we won't send the next P window until all data-packets sent in the previous P window are acknowledged. Our data-packets are constructed to have a data portion size of 16 bytes and a header section containing a single field for the 1 byte sequence number as seen in figure 3. The sender will send back ACK packets which contain a 1 byte sequence number in the header and a 1 byte data portion containing the ASCII ACK character as seen in figure 4. We implement the protocol packet structures using structs in the C programming language as seen in code listing 1.

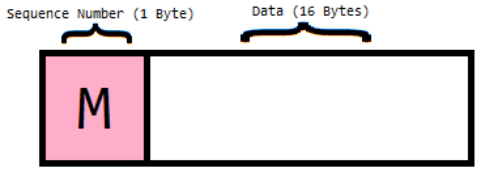

Figure 3: DataPacket Structure

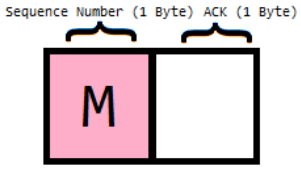

Figure 4: ACKPacketStructure

To test the protocol developed we utilize a Visual Basic Application which will buffer and received data from the PIC32 MCU, and echo back the buffered data after a period of X seconds. This application allows us to develop both client and server side message handling from the PIC32. To differentiate the received message from either a ACKPacket or a DataPacket we look at the received buffer size store in *rlen* variable. If the *rlen* of the received message is a multiple of our DataPacket size (17 bytes), we parse the message as if it were a DataPacket as seen in code listing 3. Otherwise we say the received message is a ACKPacket and parse it accordingly as seen in code listing 4.

The DataPacket message handler will mimic the role of the receiver in that it will respond with ACKPackets to all received DataPackets. To experiment further with the capabilities of sliding-window, we take the sequence numbers of all received DataPackets and shuffle them. We also will only send ACKPackets back for the corresponding received DataPackets with uniform probability of $PROBERR < 0.1$.

Our ACKPacket message handler will keep track of the number of ACKPackets it received back and compare it to the original number of DataPackets it sent out originally. As stated by the stop-and-wait sliding window protocol, we do not send the next P packets until all previously sent DataPackets have been Acknowledged. Once we have received all ACKPackets, we send the next P packets taking into account sequence number roll-over and hitting the end of overall message transmission.

To take into account the $PROBERR$ of not received an ACK we incorporate a timeout feature into our stop-and-wait sliding protocol implementation. For all sent DataPackets we expect a corresponding ACKPacket which is handled by the ACKPacket message handler. If after a period of $ACKTIMEOUT$ there still hasn't been a ACK for a subset of previously sent DataPackets, we re-transmit those DataPackets. This feature is handled by determining if the testStarted flag is set high and delayCount has reached the timeout limit. The process is seen in code listing 5.

There is a special case we also take into account for when we want to start our experiment, in which we detect if a global reset has been sent from the Visual Basic application. A global reset message starts with a character sequence of 0x0247. There would be a collision with detecting the global reset from a DataPacket if the sequence number of the DataPacket is 0x02 and first entry in the data field a 0x47 so we also set a *testStarted* flag high when the global reset has been set before. The global reset will solely begin transmission of our data sending the P packets as defined by the user.

3 Discussion

3.1 Sliding Window Testing Setup

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. We will test various values for independent parameters which are show in table 1. Also of note, due to limitations with our testing environment, (having to use a Visual Basic Application as a middleman repeating apparatus) we package our P DataPackets into one larger Packet transmission. In a more traditional client/server interaction, each one of DataPackets would be a separate transmission.

Parameter	Description
LENP	Window Size
LENM	Sequence Size
PROBERR	Uniform Probability of Transmission Error
ACKTIMEOUT	Timeout until Re-Transmission (in MSEC)

Table 1: Independent Experiment Parameters and their Descriptions

3.2 Sliding Window Test 1

The parameters for this first test as seen in table 2. For the first test we will demonstrate the effectiveness of our sliding window protocol implementation by using WireShark captures.

Parameter	Value
LENP	4
LENM	10
PROBERR	0.5
ACKTIMEOUT	5000

Table 2: Sliding Window Test 1

We begin by establishing connection between our PIC32 MCU and our Visual Basic application. We started the test by sending the Global Reset signal to our PIC32 which responds by sending the first 4 DataPackets to the the Visual Basic application totaling 122 bytes. We can see our packet data content contains the expect alphabet characters and our sequence number ranges from 1→4.

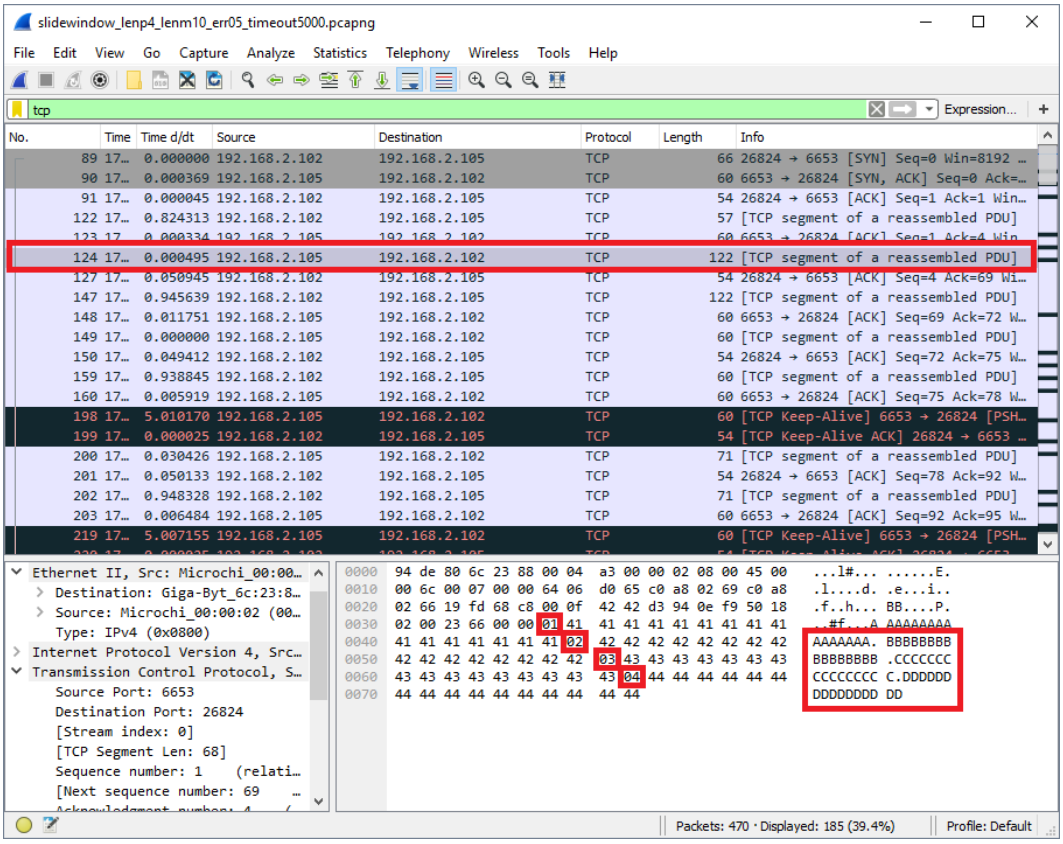

Figure 5: Test1 - First P-Frame DataPacket Transmission

After a period of 0.94 seconds, our Visual Basic echos back the message to our PIC32. At this point our DataPacket message handler takes in the four transmitted DataPackets and creates an ACK message for them in random order with a uniform probability of error of 0.5. In this test run we can see our receiver responds to three out of the four received DataPackets in a sequence number order of 02, 03, and 01. Missing is an ACKPacket for 04. Also, note the construction of the packet, containing the sequence number and the ASCII ACK characters corresponding to HEX value 0x06.

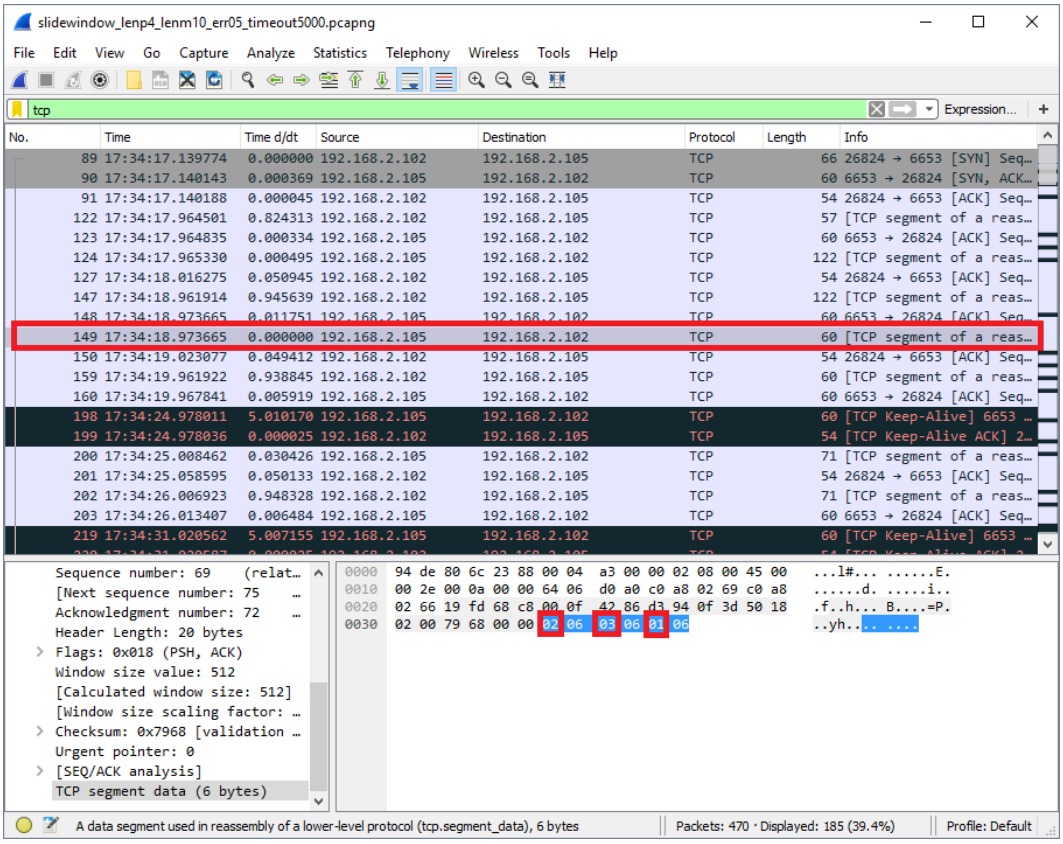

Figure 6: Test1 - First ACKPacket for P-Frame Transmission

The received ACKPackets are handled by the ACKPacketHandler which tracks the three acknowledged packets. Since not all four packets are acknowledge, we see no transmission of the next P packets. Instead after a time period of 7 seconds (not exactly 5 seconds due to processing overhead) we see the re-transmission of DataPacket containing sequence number 04.

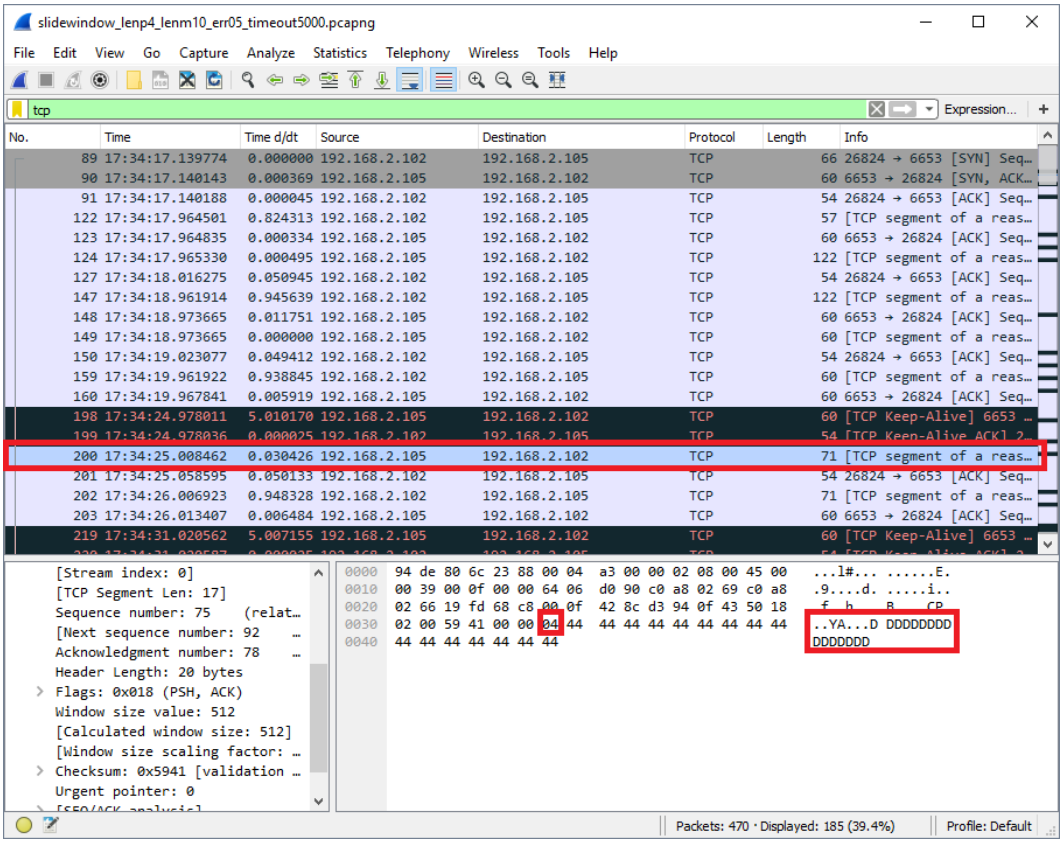

Figure 7: Test1 - First P-Frame DataPacket Re-transmission1

Due to the high probability of error for this test, we again do not receive an ACK for this DataPacket for another 2 attempts. The transmission line stays silent. After each re-transmission and a time-out period of around 5→7 seconds is hit, we see another attempt of re-transmission from our sender.

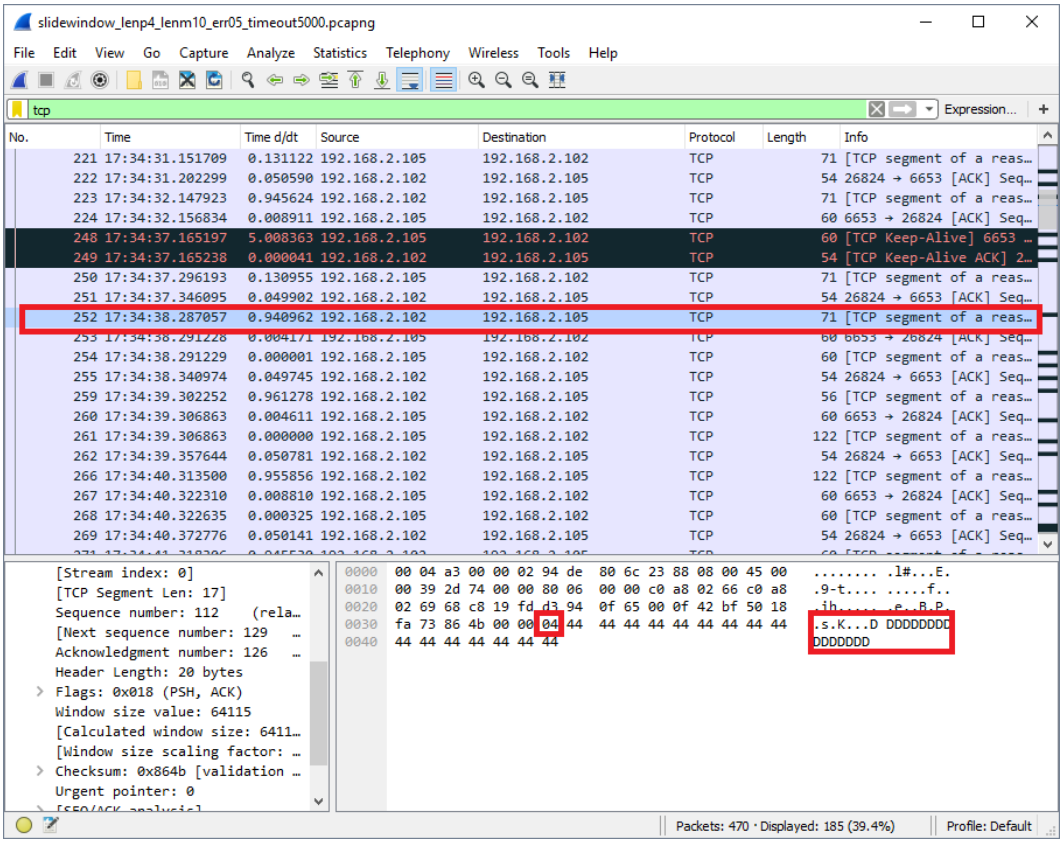

Figure 8: Test1 - First P-Frame DataPacket Re-transmission2

Finally, we see a ACKPacket for sequence number DataPacket 04.

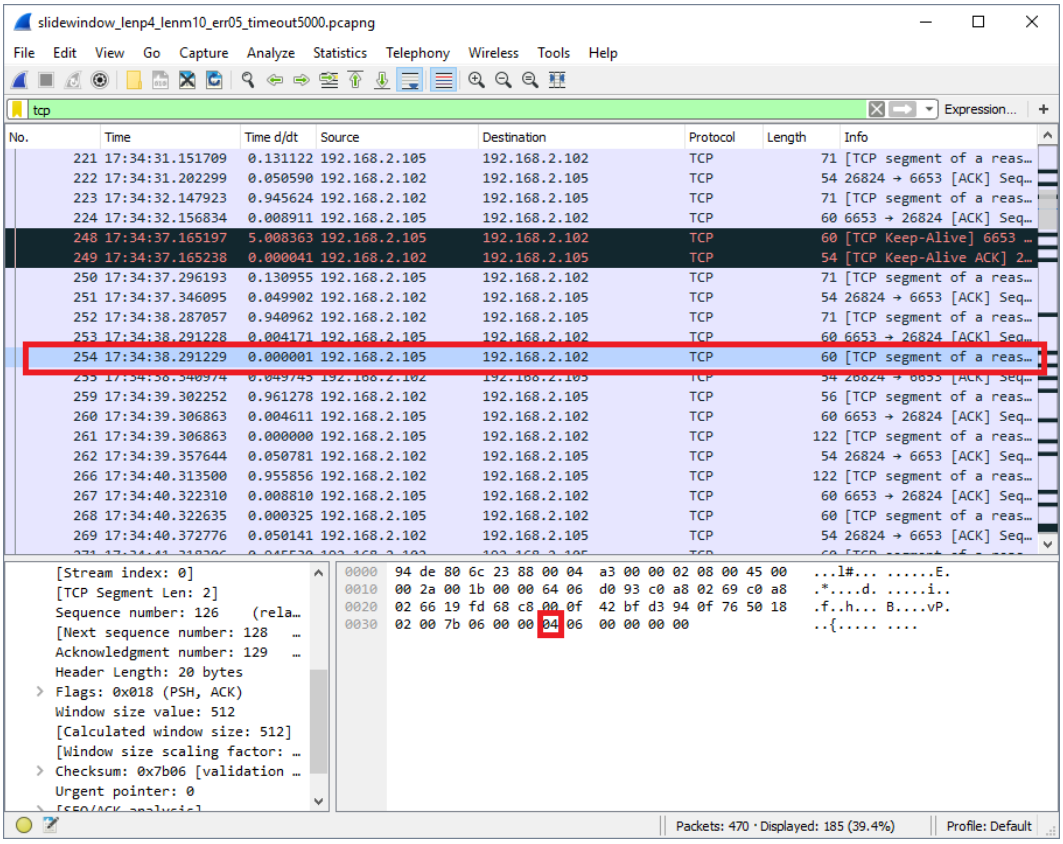

Figure 9: Test1 - ACKPacket for DataPacket 04 Transmission

Shortly after receiving acknowledgment for DataPacket sequence 04, we see transmission of the next P frame. Note the increasing sequence numbers since we still haven't reached our sequence number rollover.

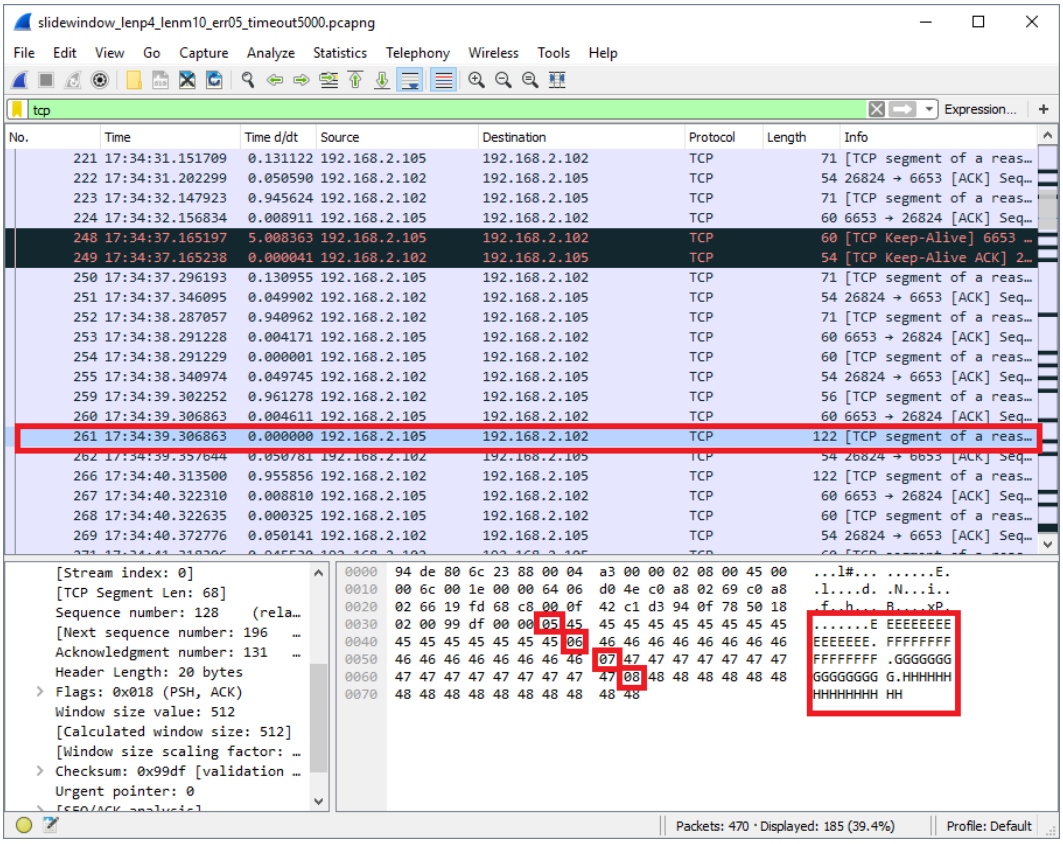

Figure 10: Test1 - Second P-Frame DataPacket Transmission

The process of ACKPacket send/receive and timeout re-transmission continues on for the full length of our 26 part message. It can be seen in figure 11 that our sequence number rolls over. We also see in figure 12 that the full 26 part message is eventually fully and successfully transfered.

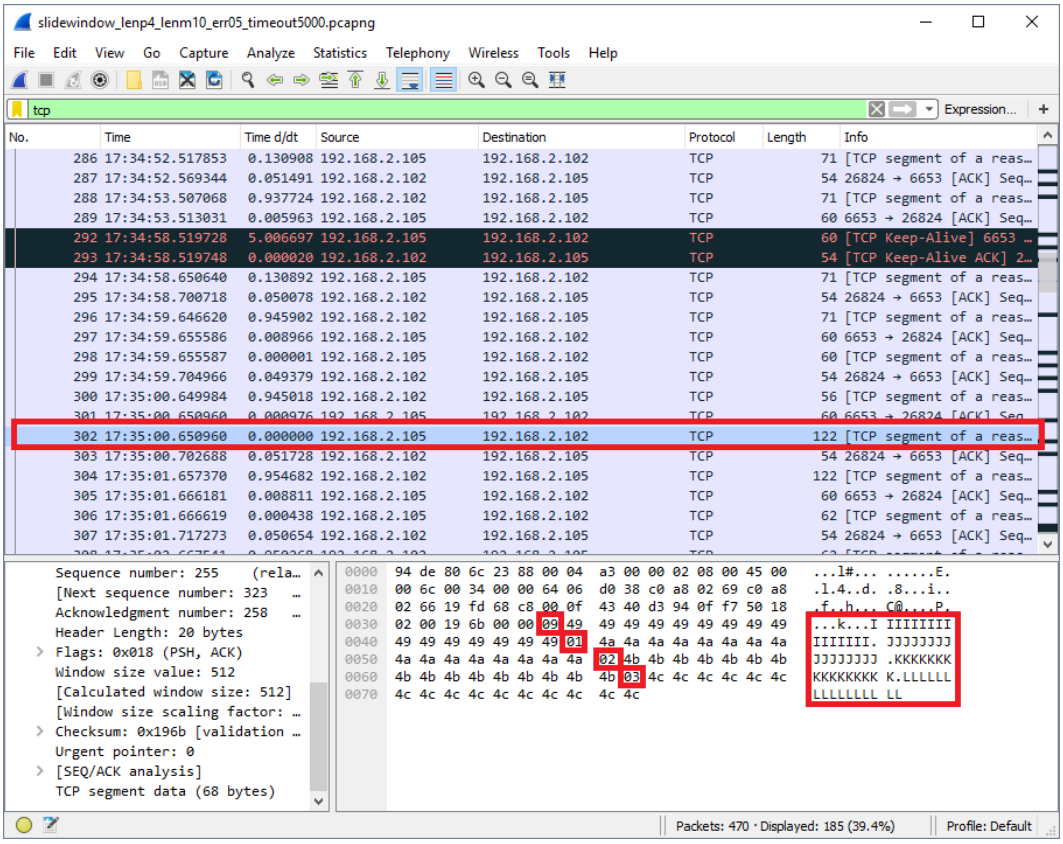

Figure 11: Test1 - Third P-Frame DataPacket Transmission

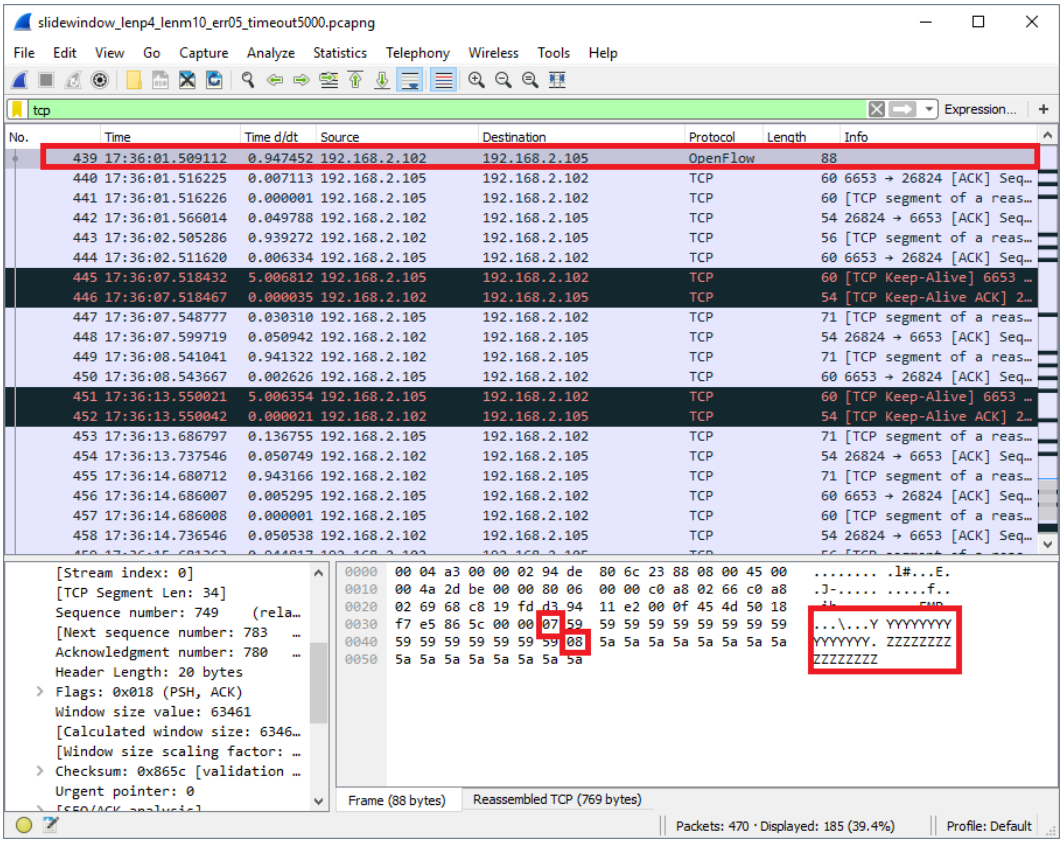

Figure 12: Test1 - Last P-Frame DataPacket Transmission

Tabulated results for runtime for the combination of input parameters are given table 3 averaged across 5 different runs.

Avg Runtime	1min 58secs
Avg Re-Transmission Count	21

Table 3: Sliding Window Test 1 - Performance

3.3 Sliding Window Test 2

Next we decrease our uniform probability of error and re-run our tests.

Parameter	Value
LENP	4
LENM	10
PROBERR	0.1
ACKTIMEOUT	5000

Table 4: Sliding Window Test 2

Tabulated results for runtime for the combination of input parameters are given table 5 averaged across 5 different runs.

Avg Runtime	28secs
Avg Re-Transmission Count	2

Table 5: Sliding Window Test 2 - Performance

3.4 Sliding Window Test 3

Next we increase our window size from the previous test and re-run the experiment.

Parameter	Value
LENP	7
LENM	10
PROBERR	0.1
ACKTIMEOUT	5000

Table 6: Sliding Window Test 3

Tabulated results for runtime for the combination of input parameters are given table 7 averaged across 5 different runs.

Avg Runtime	36secs
Avg Re-Transmission Count	4

Table 7: Sliding Window Test 3 - Performance

3.5 Sliding Window Test 4

The last test decreases our window size to 1.

Parameter	Value
LENP	1
LENM	10
PROBERR	0.1
ACKTIMEOUT	5000

Table 8: Sliding Window Test 4

Tabulated results for runtime for the combination of input parameters are given table 9 averaged across 5 different runs.

Avg Runtime	57secs
Avg Re-Transmission Count	7

Table 9: Sliding Window Test 4 - Performance

4 Conclusion

In conclusion, we found see that to best utilize the available bandwidth of your transmission line it is best to break your message into smaller message sizes. Doing so, ensures that you are utilizing the available bandwidth and minimizes the need for re-transmission since corruption of smaller packets is less likely. There is a sweet balance between window size that eliminates unnecessary re-transmissions but also reduces the probability of data corruption.

A better implementation of the sliding window approach would be selective reject. Selective reject is the process for the sender to request specific frames that it has failed to received or that have been received but corrupted. The process can ride along our stop-and-wait ACKPacket already implemented in this experiment but instead of only sending ASCII ACK characters we send ASCII NAK that tells the sender to retransmit a specific frame. It eliminates the need for the $ACKTIMEOUT$ to catch all transmission errors. We could also add error correction coding that would reduce need for re-transmission of all corrupted frames. Overall, we see that there is a sweet balance in transmission technique that is specific to the application at hand.

Appendix

Main program source code available on my GitHub:
`https://github.com/dtrejod/myece4532/blob/master/lab5`

Listing 1: DataPacket — ACK-Packet Data Structure

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

// WARNING IF myDataPacket length and myACKs are multiples of one another,
// the packet detection algo will fail.
struct myDataPacket
{
    uint8_t sequence;
    char data[DATALEN];
};
```

Listing 2: Sliding Window - 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 Frame Sent Variables
    tbfrDataTrackerI = 0;
    tbfrAckTrackerI = 0;

    // Reset Sequence Number
    tbfrSeqTracker = 1;

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

    for(tbfrDataTrackerI=0; tbfrDataTrackerI < LENP;
        tbfrDataTrackerI++)
    {
        // Check for seq rollover
        if(tbfrSeqTracker >= LENM)
        {
            tbfrSeqTracker = 1;
        }

        // Adjust offset
        if(tbfrDataTrackerI==0) offset=msgSent;
        // Populate sequence number
        tbfrData[tbfrDataTrackerI+offset].sequence =
            tbfrSeqTracker++;

        // We keep track of the seq numbers we do send
        tbfrDataTracker[tbfrDataTrackerI] =
            tbfrData[tbfrDataTrackerI+offset].sequence;

        // Copy tbfr over to
        tbfr[tbfrDataTrackerI] =
            tbfrData[tbfrDataTrackerI+offset];

        // Keep track of how much of the msg has been
        // sent
        msgSent++;

        if (msgSent > MSGLEN)
```

```
        {
            break;
        }
    }
    mPORTDClearBits(BIT_0);
    mPORTDSetBits(BIT_2); // LED3=1
    send(clientSock, tbfr,
        sizeof(struct myDataPacket)*tbfrDataTrackerI, 0);
    mPORTDClearBits(BIT_2); // LED3=0
    DelayMsec(100);
}
```

Listing 3: 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(struct myDataPacket)==0)
{
    i=0;
    // Reset data packets received
    rbfrDataTrackerI = 0;

    // Parse the receive buffer until end of buffer
    while (sizeof(struct myDataPacket)*i < rlen)
    {
        // Convert the received data into a dataPacket
        // struct
        rbfrData = (struct myDataPacket *) rbfrRaw+(i++);
        // Retrieve sequence number and store
        rbfrDataTracker[rbfrDataTrackerI++] =
            rbfrData->sequence;
    }
    // Shuffle order of rbfrDataTracker ACKs we will send
    // back
    shuffle(rbfrDataTracker, rbfrDataTrackerI);

    j=0;
    // Set sequence number P to zero and send LENP packets
    for(i=0; i < rbfrDataTrackerI; i++)
    {
        // Send ACK for Random number of packets
        if (randMToN(0.0,1.0) >= PROBERR)
        {
            rbfrAck[j].sequence = rbfrDataTracker[i];
            rbfrAck[j].ackChar = 0x06;
            j++;
        }
```

```
    }
    mPORTDClearBits(BIT_0);
    mPORTDSetBits(BIT_2); // LED3=1
    send(clientSock, rbfrAck,
        sizeof(struct myACK)*j, 0);
    mPORTDClearBits(BIT_2); // LED3=0
    DelayMsec(100);
}
```

Listing 4: Sliding Window - ACK Packet Detector and Handler

```
else if (rlen%sizeof(struct myACK)==0)
{
    // Randomize the
    // Parse the receive buffer for myACK until end of
    // buffer
    i=0;
    while (sizeof(struct myACK)*i < rlen)
    {
        // Convert the received data into a myAck struct
        tbfrAck = (struct myACK *) rbfrRaw+(i++);

        // Retrieve sequence number and store
        tbfrAckTracker[tbfrAckTrackerI++] =
            tbfrAck->sequence;
    }
    // Check if we recieved all the frames we orignally
    // sent. If yes transfer the next M frames
    if (tbfrAckTrackerI >= tbfrDataTrackerI)
    {
        if (msgSent > MSGLEN)
        {
            testStarted=0;
        }
        else
        {
            // Check if there are more frames to send
            // Reset Frame Sent Variables
            tbfrAckTrackerI = 0;

            for(tbfrDataTrackerI=0; tbfrDataTrackerI < LENP;
                tbfrDataTrackerI++)
            {
                // Check for seq rollover
                if(tbfrSeqTracker >= LENM)
                {
                    tbfrSeqTracker = 1;
                }
```

```
                // Adjust offset
                if(tbfrDataTrackerI==0) offset=msgSent;
                // Populate sequence number
                tbfrData[tbfrDataTrackerI+offset].sequence =
                    tbfrSeqTracker++;

                // We keep track of the seq numbers we do send
                tbfrDataTracker[tbfrDataTrackerI] =
                    tbfrData[tbfrDataTrackerI+offset].sequence;

                // Copy tbfr over to
                tbfr[tbfrDataTrackerI] =
                    tbfrData[tbfrDataTrackerI+offset];

                // Keep track of how much of the msg has been
                // sent
                msgSent++;

                if (msgSent > MSGLEN)
                {
                    break;
                }
            }
            mPORTDClearBits(BIT_0);
            mPORTDSetBits(BIT_2); // LED3=1
            send(clientSock, tbfr,
                sizeof(struct myDataPacket)*tbfrDataTrackerI, 0);
            mPORTDClearBits(BIT_2); // LED3=0

            DelayMsec(100);
        }
    }
}
```

Listing 5: Sliding Window - ACK Timeout Handler

```
// If rlen is zero length we increment delay count
else if (testStarted == 1)
{
    DelayMsec(10);
    delayCount++;

    // Check for ACK timeout
    if (delayCount*10 > ACKTIMEOUT)
    {
        // Retransmit any packets we haven't received ACKs back
        // for yet
```

```
        delayCount = 0;

        // Reset selective repeat
        selectiveRepeat = 0;

        // Set sequence number P to zero and send LENP
        // packets
        for(i=0; i < tbfrDataTrackerI; i++)
        {
            flag = 0;
            for(j=0; j< tbfrAckTrackerI; j++)
            {
                if(tbfrDataTracker[i]==tbfrAckTracker[j])
                {
                    flag = 1;
                    break;
                }
            }
            if (flag == 0)
            {
                tbfr[selectiveRepeat++] = tbfr[i];
            }
        }
        mPORTDClearBits(BIT_0);
        mPORTDSetBits(BIT_2); // LED3=1
        send(clientSock, tbfr,
            sizeof(struct myDataPacket)*selectiveRepeat, 0);
        mPORTDClearBits(BIT_2); // LED3=0
        DelayMsec(100);
    }
}
```