Yeseul An
CSS 432
Assignment 3 Report
Professor: Brent Lagesse

# Documentation

In this assignment, I implemented stop-and-wait algorithm and sliding window algorithm and evaluated their performance in transferring 20,000 packets over 1Gbps network. When it comes to the connection between the server and the client, I have used UDP (connectionless unreliable transport layer protocol), which doesn't prevent loss or re-ordering of messages. It is given in UdpSocket.cpp UdpSocket.h files. In addition, when it comes to the measuring times in the program, Timer.cpp and Timer.h files are given, which allows me to measure performance in tv_sec and tv_usec. In the main program(hw2.cpp), the program instantiates UDP socket which allocates a 1460-byte message[], and evaluates the performance of UDP point-to-point communication using three different test cases:

### *Case 1: Unreliable test*

Unreliable test simply sends 20,000 UDP packets from the client to the server. The actual implementation is already implemented in hw2.cpp file. The client sends messages to the server through UDP socket object for 20,000 times. The server receives message from the client through UDP socket object. However, this is unreliable that means that the server never sends acknowledgement back to the client whether it receives the message or not. Therefore, the client can't find any way to retransmit lost packets back to the server. Also, the server may hang because UDP packet being sent by the client never received by the server.

### *Case 2: Stop-and-wait test*

Stop-and-wait algorithm is reliable in that when the client sends a message to the server, the server sends back an acknowledgement for that particular message back to the client. Also, if the message sent by the client is lost, the client retransmit messages again back to the server. The specific steps of the algorithm and the flow chart are explained below:

Client:

1) Initializes variables such as retransmission times to 0.
2) Starts looping from 0 to max-1 times, initialize sequence number in message[0] and sends the message through the UDP socket to the server.
3) Starts the timer in order to measure whether it can receive acknowledgement before TIME_OUT = 1500 usec.
4) While socket doesn't have any data to read, do timer.lap() in order to check whether it overs the TIME_OUT. If the timer says TIME_OUT, the client resends the message to the server, increments retransmission times, and restarts the timer.
5) If there is data to read in the socket and it is arrived before the TIME_OUT, the client receives acknowledgment from the server and go next of the for loop.
6) Returns number of retransmission times to the main program.

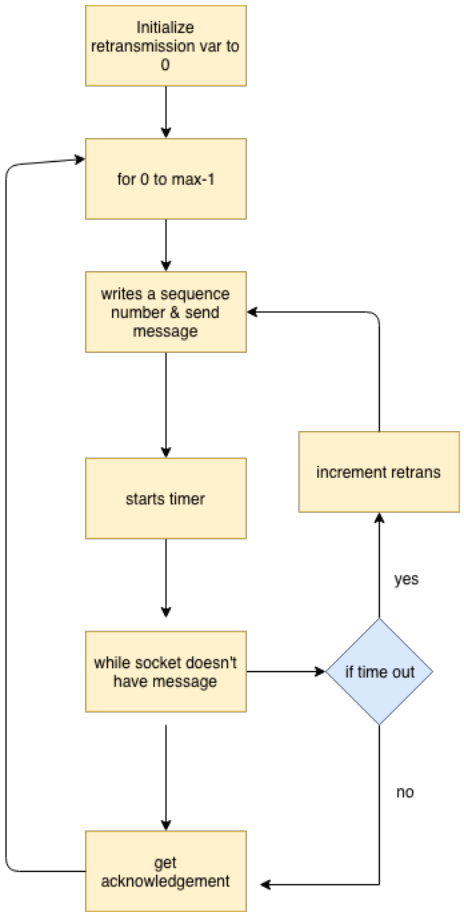



FIGURE 1 CLIENT SIDE STOP-AND-WAIT

Server:

1) Starts looping from 0 to max-1 times, do the while loop and check whether the socket has data to read.
2) If there is data to read, the server receives the message from the client, which has sequence number attached to it.
3) Checks whether the message's sequence number equals to the current i in the loop. If they are the same, it sends back the acknowledgement to the client.
4) Breaks the while loop and go next to the for loop.

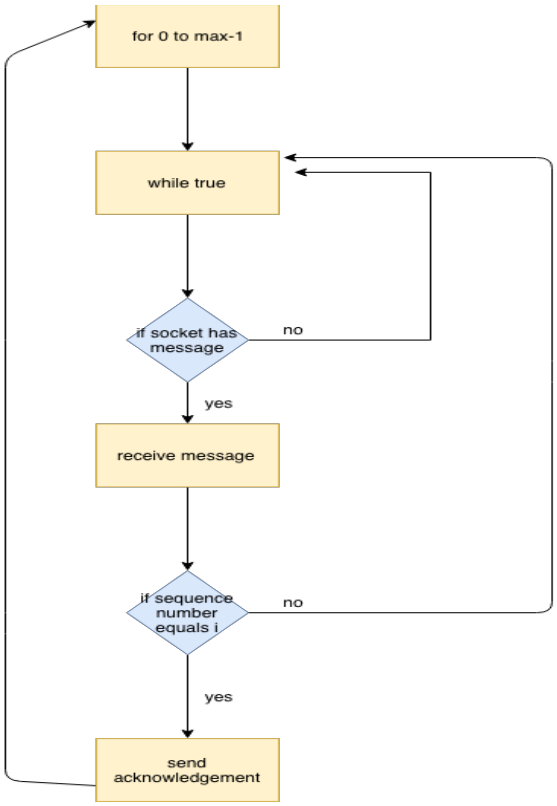



FIGURE 2 SERVER SIDE STOP-AND-WAIT

### *Case 3: Sliding Window*

Sliding window algorithm is more efficient than Stop-and-Wait algorithm in that the client can keep sending messages as long as the number of in-transit messages is less than a given window size. Sliding window algorithm sends messages as a pipelined fashion, which doesn't need to wait until it receives an integer acknowledgement of the sequence number for each message. The explanation of the algorithms is below for both client and server:

Client:

1) Initializes variables that are used throughout the function such as the number of retransmissions, sequence, acknowledged sequence, and last sequence received.
2) While sequence is less than max number of messages, if the number of in-transit messages are less than window size, it writes a message sequence number in message[0], send the message to the server, and increment sequence by 1.
3) If socket has any data, receives last sequence received from the server. If the value of last sequence received from the server equals acknowledged sequence, increment acknowledged sequence.
4) If socket has no data to read, starts the timer and see whether the timer passes TIME_OUT value.
5) If time out, it calculates the number of retransmissions by calculating number of retransmissions = current number of retransmissions + (sequence – acknowledgement).
6) Reinitializes sequence to the acknowledge sequence. This sets retransmission of the messages after the currently acknowledged message.
7) Returns the number of retransmissions to the main program.

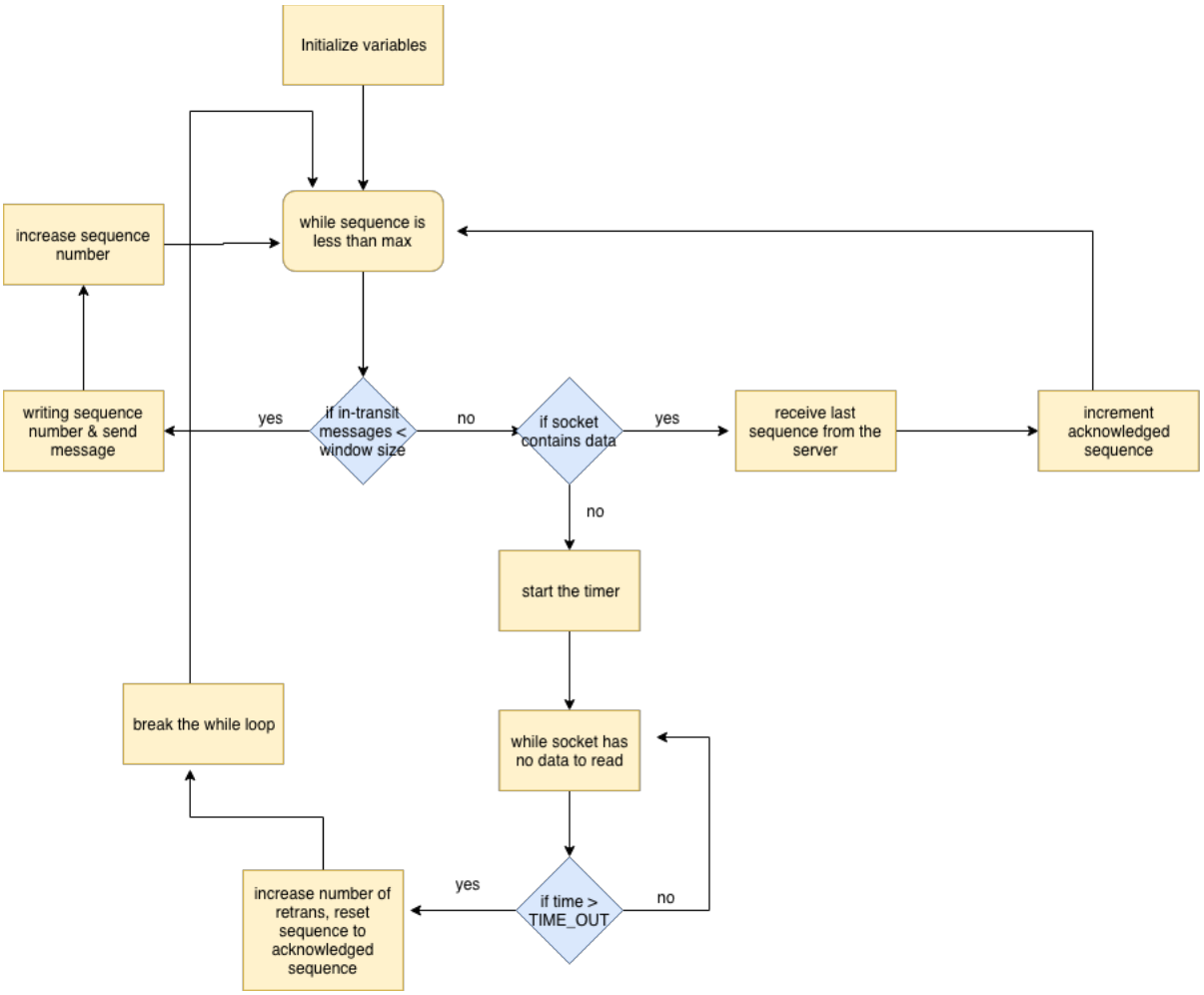



FIGURE 3 CLIENT SIDE SLIDING WINDOW

Server:

1) Initializes variables that are used throughout the function such as received message, base, and next sequence. Also, created the vector<bool>window, which is used to keep track of received messages and cumulative acknowledgements.
2) While base is less than maximum (20,000), if the socket contains the message, receives the message and assigns received message value to the message's sequence number in message[0].
3) If receivedMsg-base (value acknowledged) is greater than the size of window, drop the message because it can only handle the number of messages that are less or equal than the given window size.
4) If the value of receivedMsg is greater than the value of base, acknowledges messages by changing the value of window[receivedMsg]=true. Checks the sequence values stored in the vector whether they are continuous and assigns next sequence value to the base.
5) If value of receivedMsg == base, acknowledges the message by window[receivedMsg]=true. Assigns next sequence to the base.
6) Send that cumulative acknowledgement to the client.

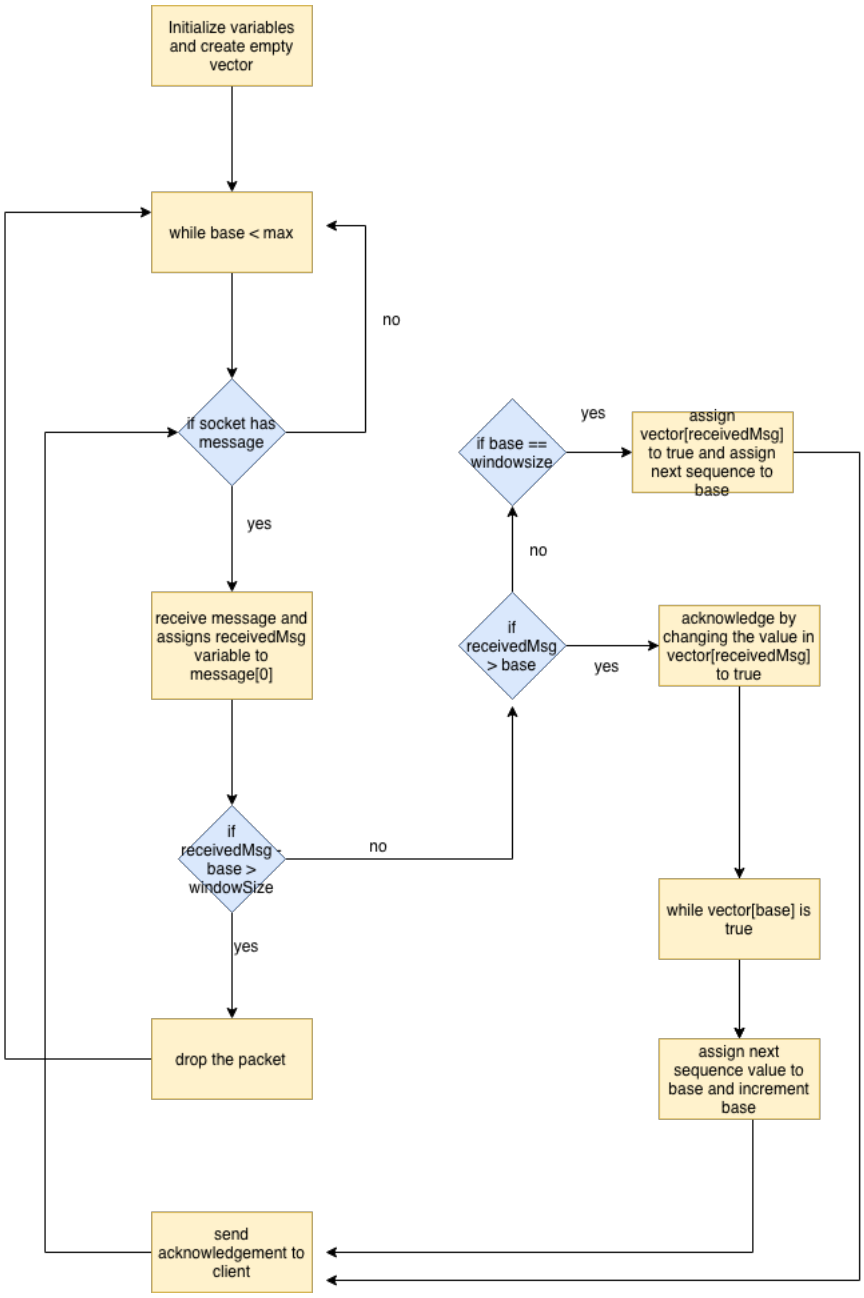



FIGURE 4SLIDING WINDOW SERVER

### *Case 4: Sliding Window with Drop Rates*

Client:

The client uses same sliding-window algorithm. Only difference is that the window size is only 1 and 30.

Server:

1) Initializes variables that are used throughout the function such as received message, base, and next sequence. Also, created the vector<bool>window, which is used to keep track of received messages and cumulative acknowledgements.
2) While base is less than maximum (20,000), if the socket contains the message, receives the message and assigns received message value to the message's sequence number in message[0].
3) Calculates randomly dropping rate through function and get dropFlag whether to decide the message should be dropped or not.
4) If the flag is ture, drop the message.
5) If receivedMsg-base (value acknowledged) is greater than the size of window, drop the message because it can only handle the number of messages that are less or equal than the given window size.
6) If the value of receivedMsg is greater than the value of base, acknowledges messages by changing the value of window[receivedMsg]=true. Checks the sequence values stored in the vector whether they are continuous and assigns next sequence value to the base.
7) If value of receivedMsg == base, acknowledges the message by window[receivedMsg]=true. Assigns next sequence to the base.
8) Send that cumulative acknowledgement to the client.

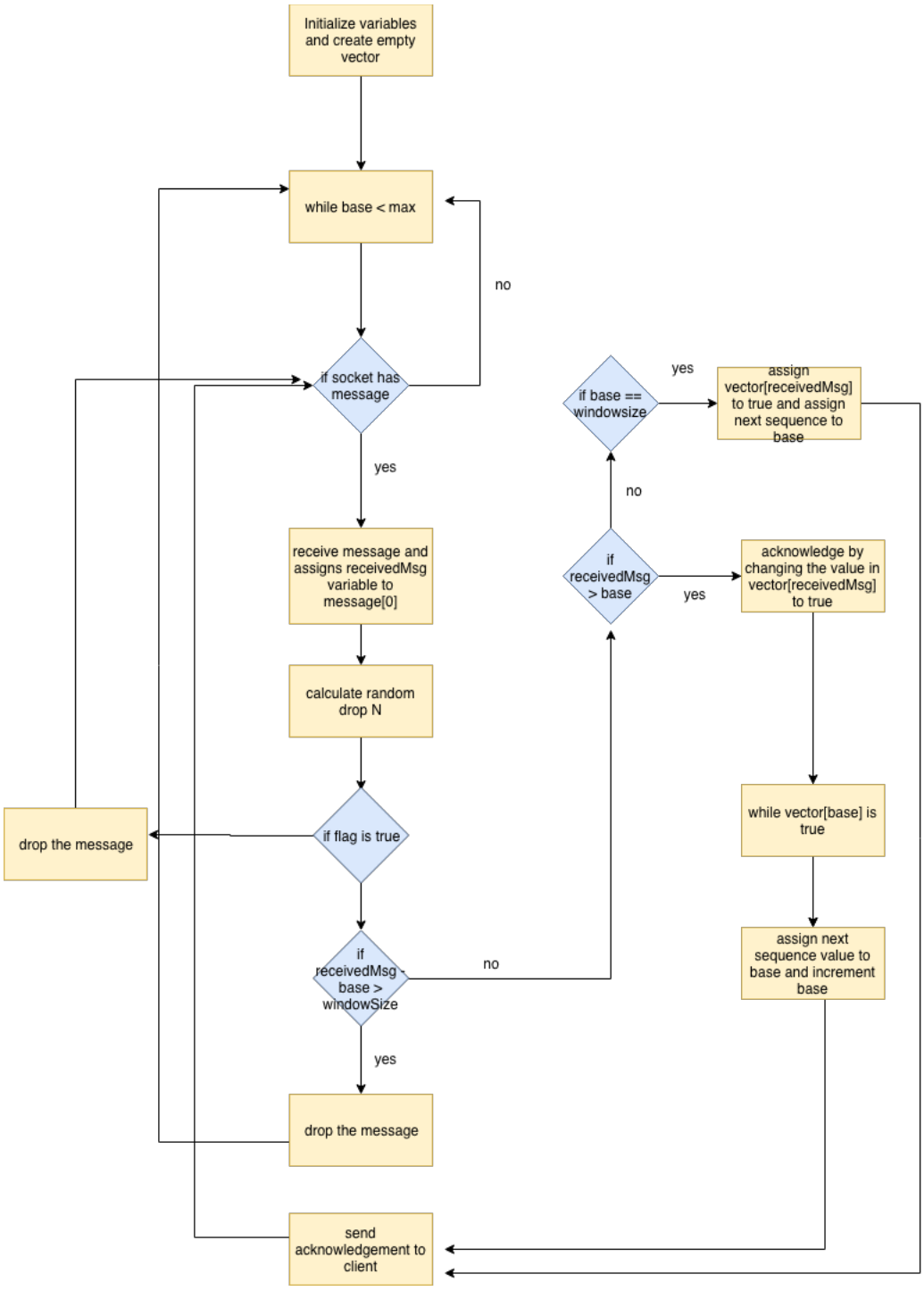



FIGURE 5 SLIDING WINDOW SERVER WITH DROP RATE

# Execution Output

### *How to Compile:*

These are files that are needed in order to compile:

| | |
|---|---|
| Timer.h | given timer program |
| Timer.cpp | given timer program |
| UdpSocket.h | given udp socket program |
| UdpSocket.cpp | given udp socket program |
| Hw2.cpp | given hw2 main program with function calls added |
| Hw3a.cpp | modified program of hw2 which is to run case 4 |
| Udp.cpp | program that implements functions that are used in hw2.cpp(case2,3) |
| Udpa.cpp | program that implements functions that are used in hw3a.cpp(case4) |
| res | Folder that contains plotted files and files that are needed for plotting |

### *To run case 1, 2, 3:*

**Compile:** Type $g++ UdpSocket.cpp Timer.cpp udp.cpp hw2.cpp -o hw2

**Running:**

Server: Type $./hw2

Client: Type $./hw2 uw1-320-0[number].uwb.edu

ex) uw1-320-07.uwb.edu

### *To run case 4:*

**Compile:** Type $g++ -std=c++14 UdpSocket.cpp Timer.cpp udpa.cpp udp.cpp hw3a.cpp -o hw3a

**Running:**

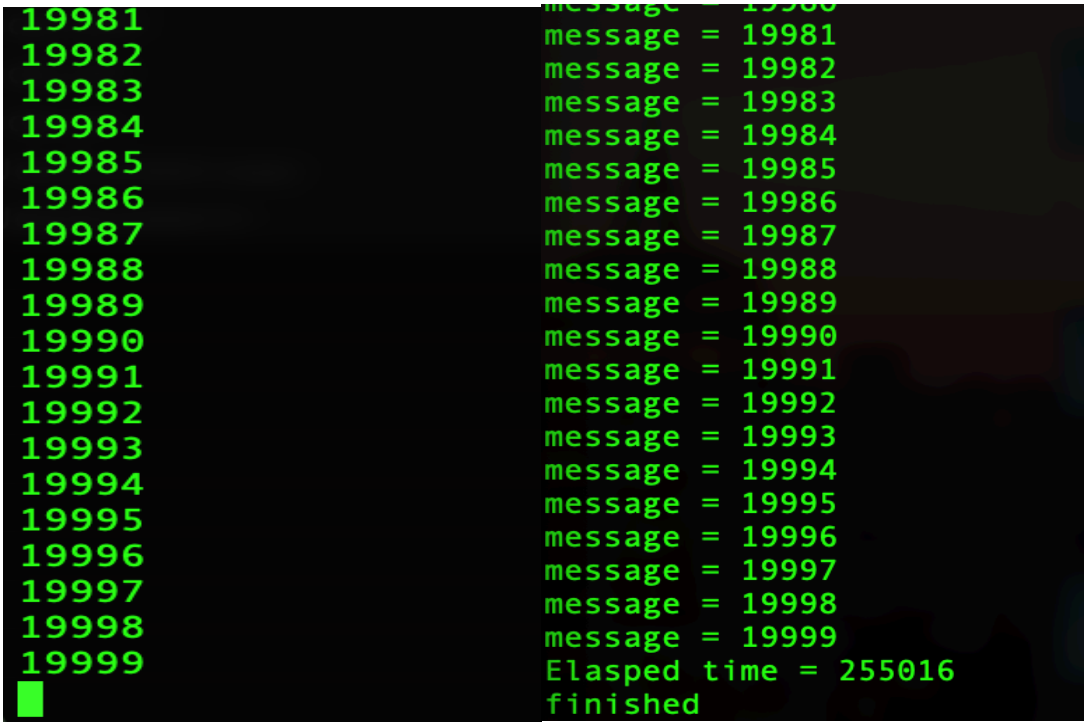
Server: Type $./hw3a

Client: Type $./hw3a uw1-320-0[number].uwb.edu

ex) uw1-320-07.uwb.edu

***Case 1: Unreliable test with Hanging***

Server Client




After running the program several times, I could observe that in some cases the program hangs due to the dropping off of UDP messages. As you can see in the above screenshot, it hangs in the server after it prints out the sequence number 19999 and never finishes its process. For unreliable case, the messages can be dropped so that sequence number might not be consecutive until 19999. However, when I run the program, I could only see the program hanging just before the server finishes its execution, the discrepancy between iteration number and the sequence number of the server is not different.

### *Case 1: Unreliable Test without Hanging*

Server

```
19978
19979
19980
19981
19982
19983
19984
19985
19986
19987
19988
19989
19990
19991
19992
19993
19994
19995
19996
19997
19998
19999
server ending...
finished
yeseul90@uw1-320-07:~/432/A4$
```

Client

```
message = 19977
message = 19978
message = 19979
message = 19980
message = 19981
message = 19982
message = 19983
message = 19984
message = 19985
message = 19986
message = 19987
message = 19988
message = 19989
message = 19990
message = 19991
message = 19992
message = 19993
message = 19994
message = 19995
message = 19996
message = 19997
message = 19998
message = 19999
Elasped time = 252510
finished
yeseul90@uw1-320-12:~/432/A4$
```

I could observe that in many cases the server completes its execution successfully without any hanging. I assume this means that the server completely received all packages from the client. In this case, sequence number is for sure not different from the iteration number from the client.

### *Case 2: Stop-and-wait test with 1500 usec TIME_OUT*

Server

```
Choose a testcase
   1: unreliable test
   2: stop-and-wait test
   3: sliding windows
[--> 2
server reliable test:
server ending...
finished
```

Client

```
Choose a testcase
   1: unreliable test
   2: stop-and-wait test
   3: sliding windows
[--> 2
client: stop and wait test:
Elasped time = 2957580
retransmits = 0
finished
```

The above screenshots show the program running with the stop-and-wait algorithm on each server and client side. The client retransmits the packet when it doesn't get acknowledgement from the server for TIME_OUT duration (1500 usec). In this case, no retransmission occurs that means all the packets are transmitted successfully from the client to the server. If retransmission occurs, it is counted up and saved to the number of retransmits. The performance may not reach the peak comparing that with sliding window algorithm because the client must wait for an acknowledgement every time it sends out a new message.

***Case 2: Stop-and-wait test with 400 usec TIME_OUT***

Server

```
Choose a testcase
   1: unreliable test
   2: stop-and-wait test
   3: sliding windows
[--> 2
server reliable test:
server ending...
finished
```

Client

```
Choose a testcase
   1: unreliable test
   2: stop-and-wait test
   3: sliding windows
[--> 2
client: stop and wait test:
Elasped time = 3096962
retransmits = 3
finished
```

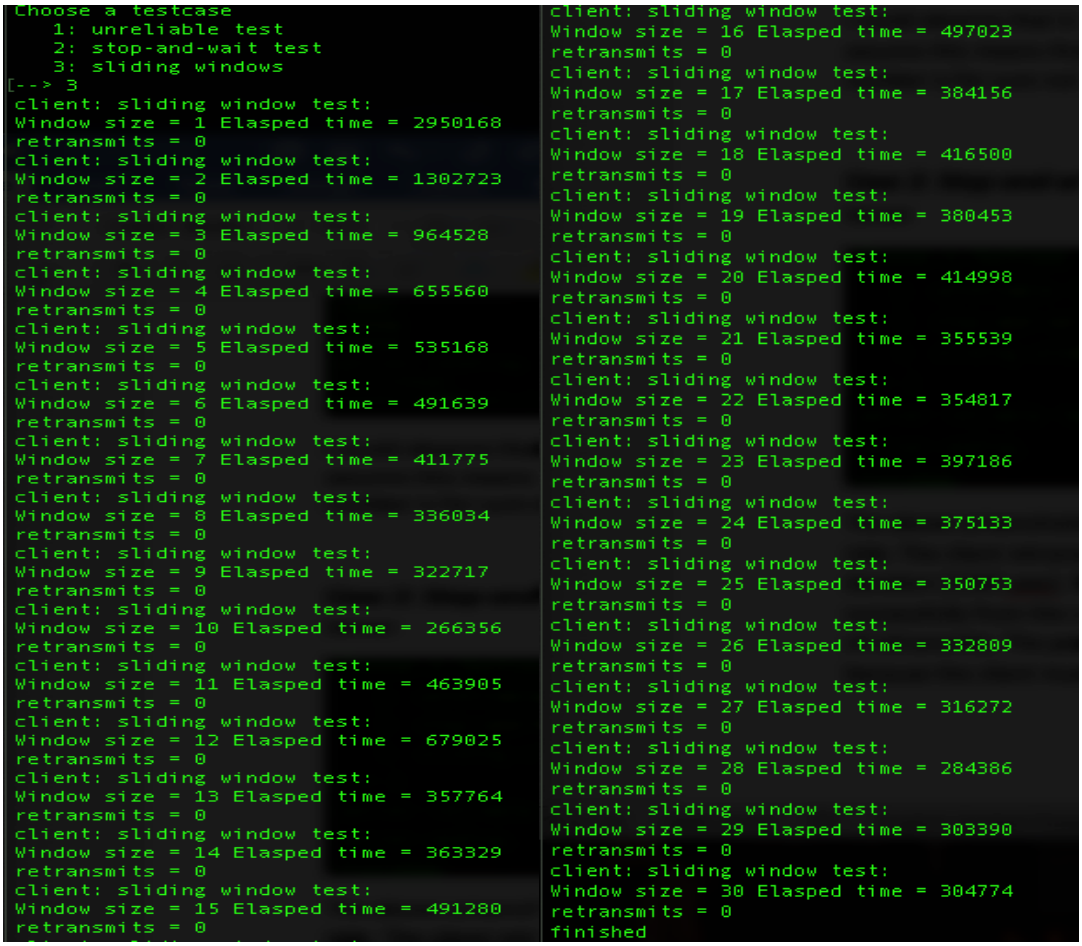
The above screenshots show the program with stop-and-wait algorithm with TIME_OUT duration (400 usec). I have set 400 usec in order to observe the number of retransmissions and compare with that of sliding window algorithm. For stop-and-wait algorithm, the client has to resend the packages that are lost. The program figures to see whether it lost packet or not by not receiving acknowledgement from the server for TIME_OUT duration. As you can see, the number of retransmissions is 3 and elapsed time performance takes slightly more than the one without retransmission. This might be because the number of retransmissions occurs only three times.

***Case 3: Sliding Window with 1500 usec TIME_OUT***

Server

```
Choose a testcase
   1: unreliable test
   2: stop-and-wait test
   3: sliding windows
--> 3
server early retrans test:
```

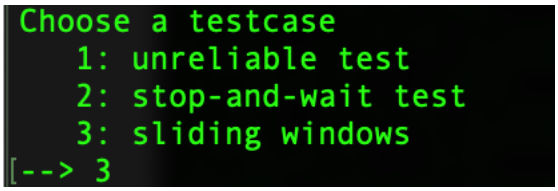
Client



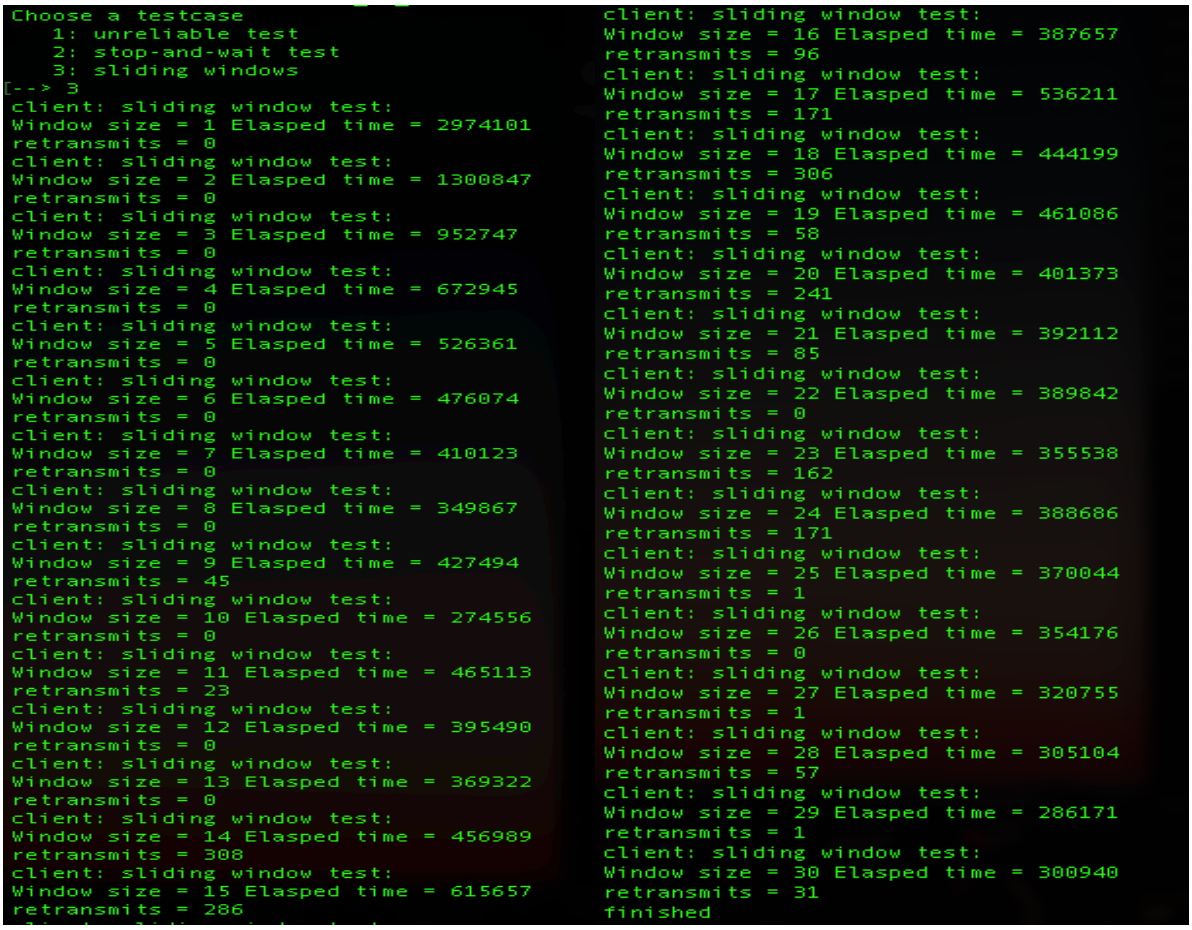

The above screenshots show the performance of the sliding window algorithm with 1500 usec TIME_OUT. Elapsed time performance, number of retransmissions are calculated based on the different sizes of windows. The client sends the messages until when the number of in_transit messages are not greater than the window size and waits for the cumulative acknowledgement from the server. The time performance of sliding window size=1 and stop-and-wait algorithms shows similar elapsed time. From the observation, I could find that the elapsed time performance enhances as window size increases.

### *Case 4: Sliding Window with 400 usec TIME_OUT*

Server



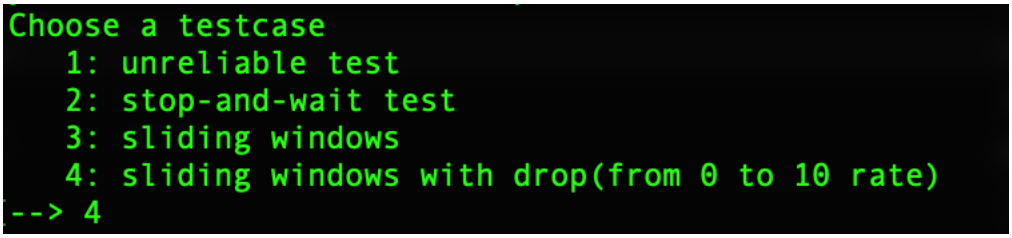

Client



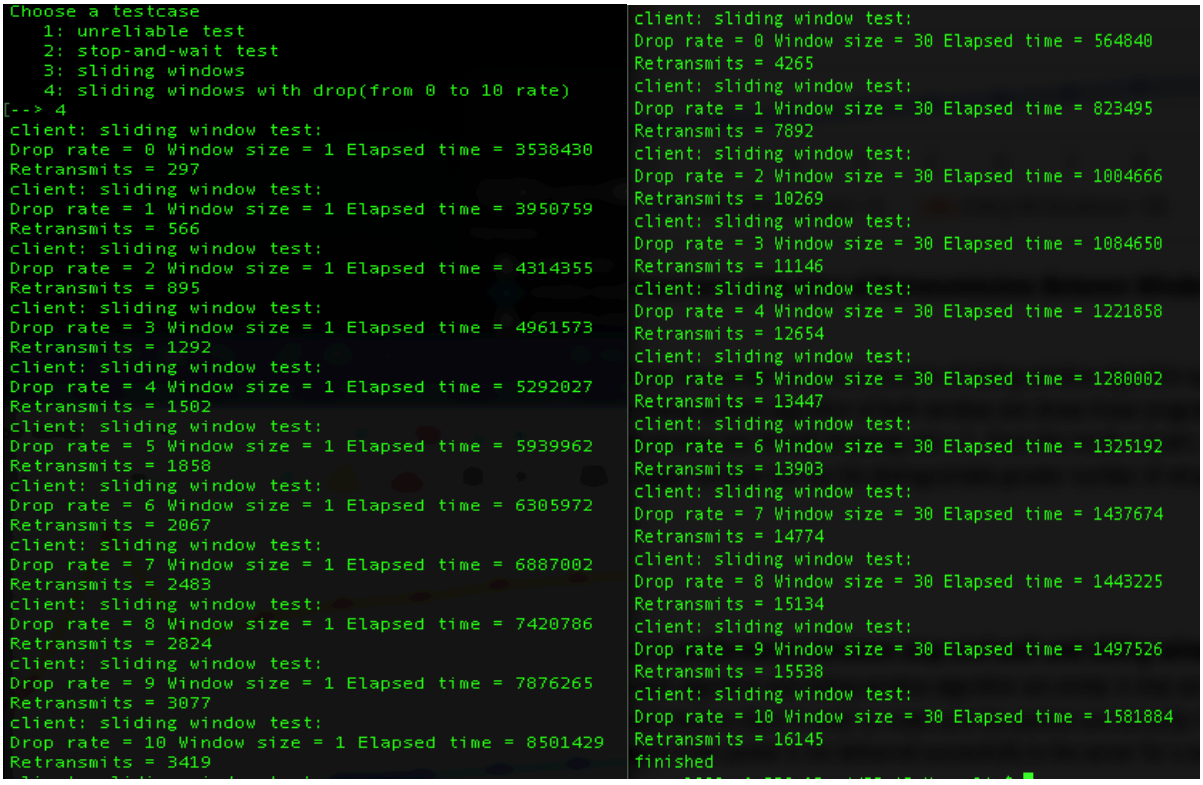

The above screenshots show the performance of the sliding window algorithm with 400 usec TIME_OUT. I have set 400 usec TIME_OUT in order to observe the number of retransmissions. From the result, I could assume that as window size increases, there are more retransmissions occur. I think this is because sliding window algorithm has to resend the messages that are the size of the window after last acknowledged message. Therefore, when a retransmission occur, the messages that are resent by the client include the messages that are already successfully received by the server.

### *Case 4: Sliding Window with Drop Rates, TIME_OUT 1500 usec*

Server




Client




The above screenshots show the program running with the sliding window algorithm with 0-10% of random dropping rates. I assumed that since we are purposely dropping the messages, there should be retransmissions occur with the 1500 usec. As the result shows, there are retransmission numbers which increases with the increment of the dropping rates. Also, as we observed before, the bigger window size with the most dropping percentage has the greatest number of retransmissions. This is because the bigger window has to retransmit more messages when retransmission occur comparing to that of window size 1 which individual message is resent when retransmission occurs.

## Performance Evaluation

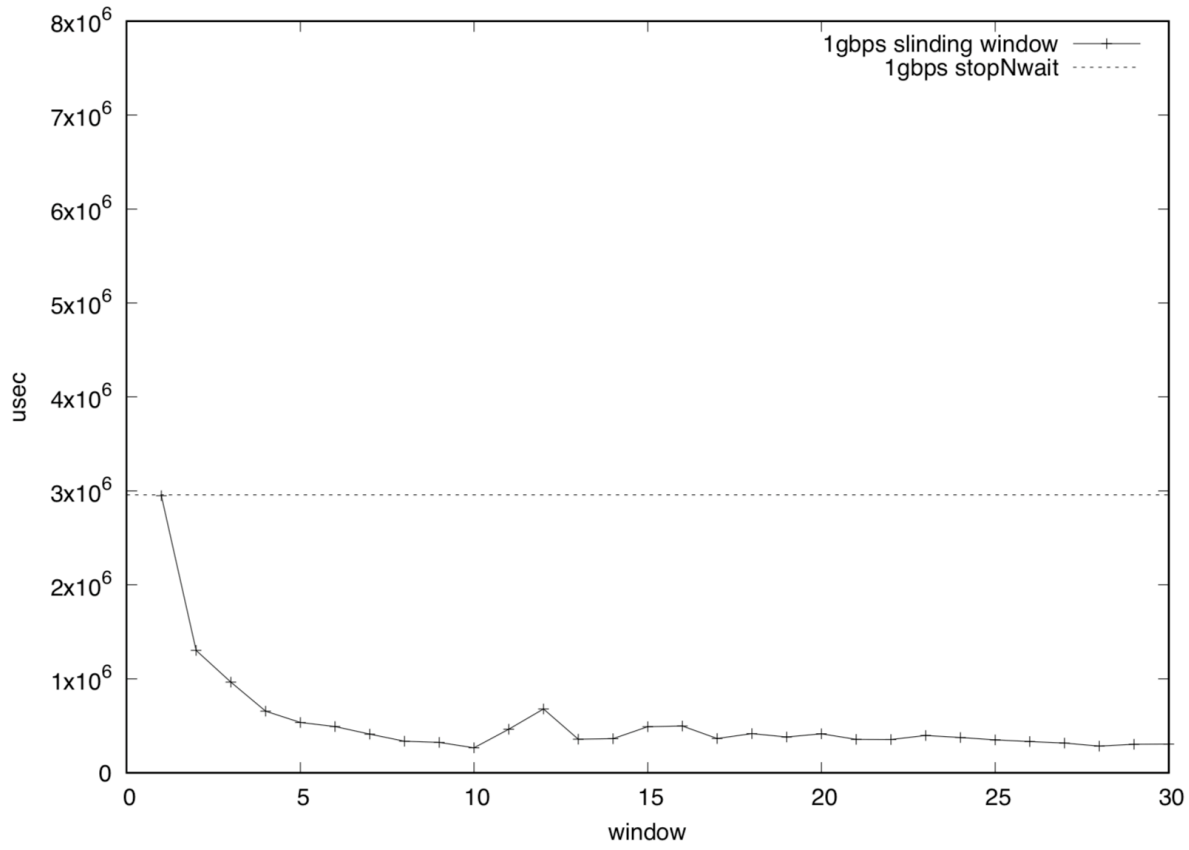



***Figure 6 Time Performance between Sliding Window and Stop and Wait***

The figure above shows the performance of stop-and-wait algorithm and sliding window algorithm. When window size is 0, the time performance point of stop-and-wait algorithm is very close to that of the sliding-window algorithm. As number of window size gets bigger, the performance of sliding window algorithm has much faster time performance than the stop-and-wait algorithm.

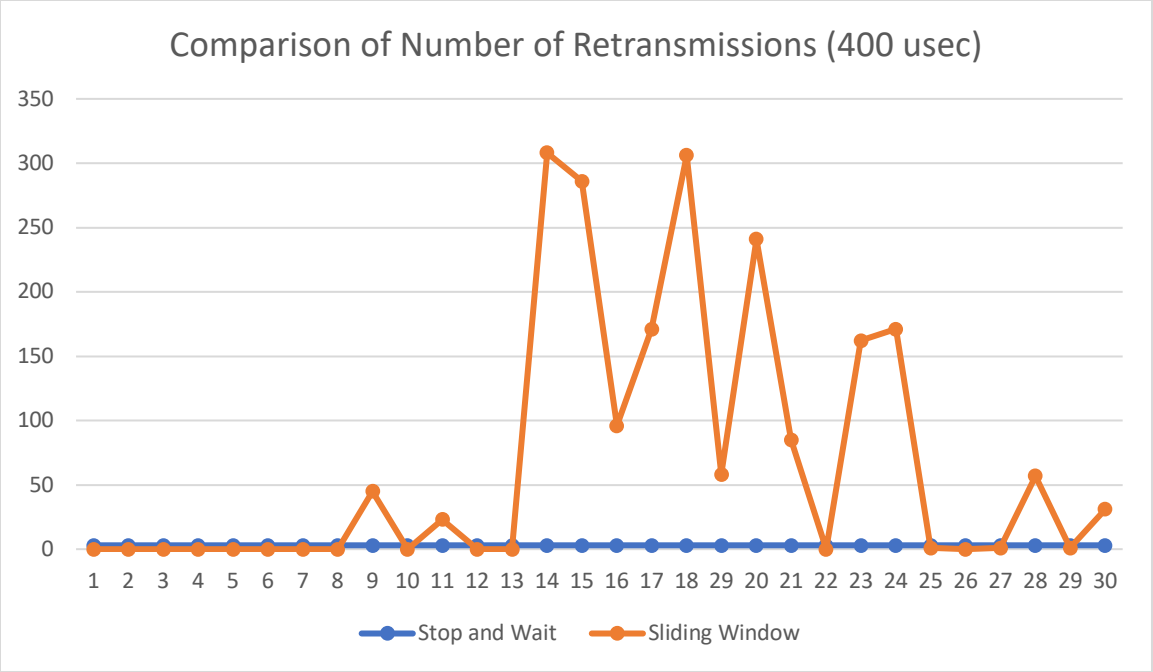



***Figure 7 Comparison of Number of Retransmissions***

The figure above shows the number of retransmissions of stop-and-wait algorithm and sliding window algorithm with 400 usec. I set 400 usec because with the 1500 usec I couldn't observe any number of retransmissions for both stop-and-wait and sliding window algorithm. As you can see, the gap of number of retransmissions between sliding window and stop-and-wait algorithms gets bigger after the window size 9. Especially, window size 14 has the biggest number of retransmissions. I think this is because when retransmission needs to be happened, the one with the bigger window size needs to transmit more messages that corresponds the window size. There are several peaks and drops in the graph because retransmissions don't always retransmission occurs in the sliding window algorithm.

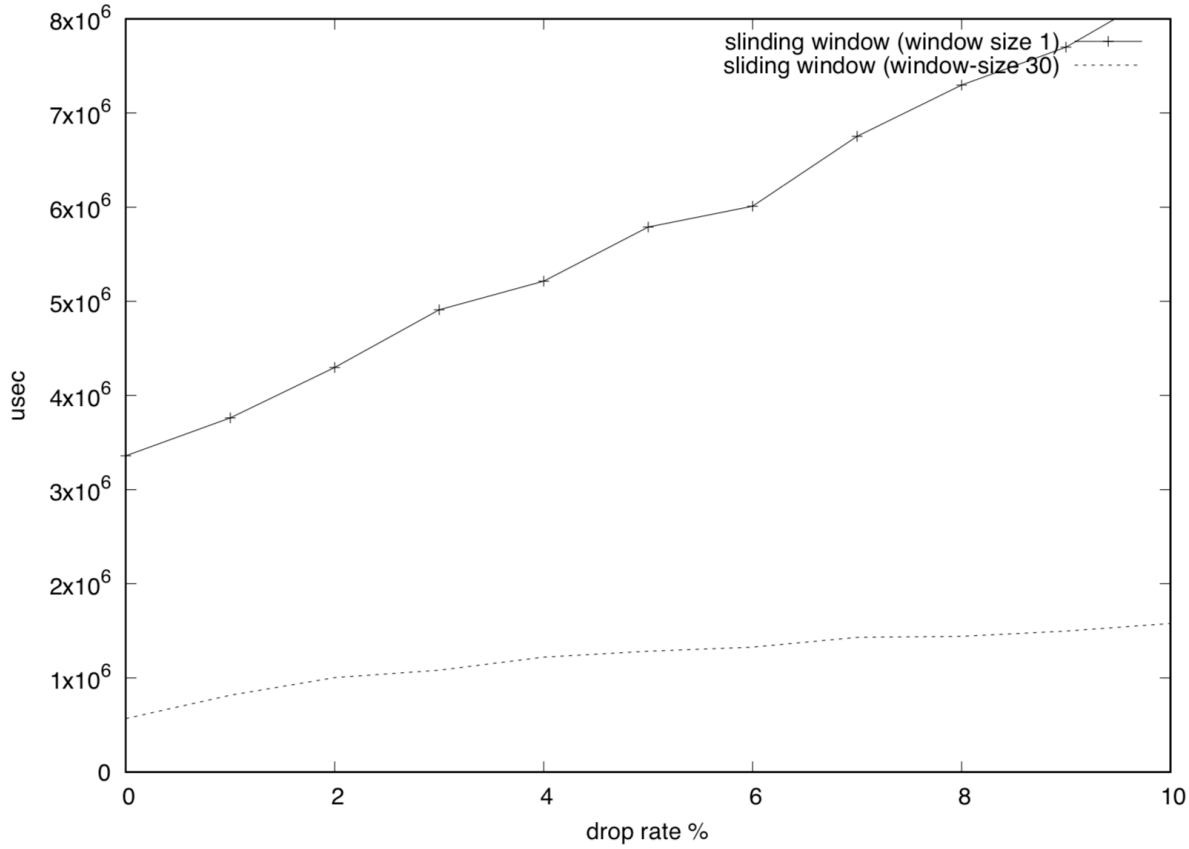



***Figure 8 Time Performance of Sliding Window Algorithm with Dropping Rate***

The above figure shows the time performance results of sliding window algorithm with window size 1 and 30 with the dropping rates between 0 to 10 percent. For each dropping rate, sliding window algorithm with bigger window size has better time performance from 0 – 10. In addition, the performance of sliding window algorithm with window size 1 shows linear progression; on the other hand, the performance of the sliding window algorithm with window size 30 shows logarithmic progression.

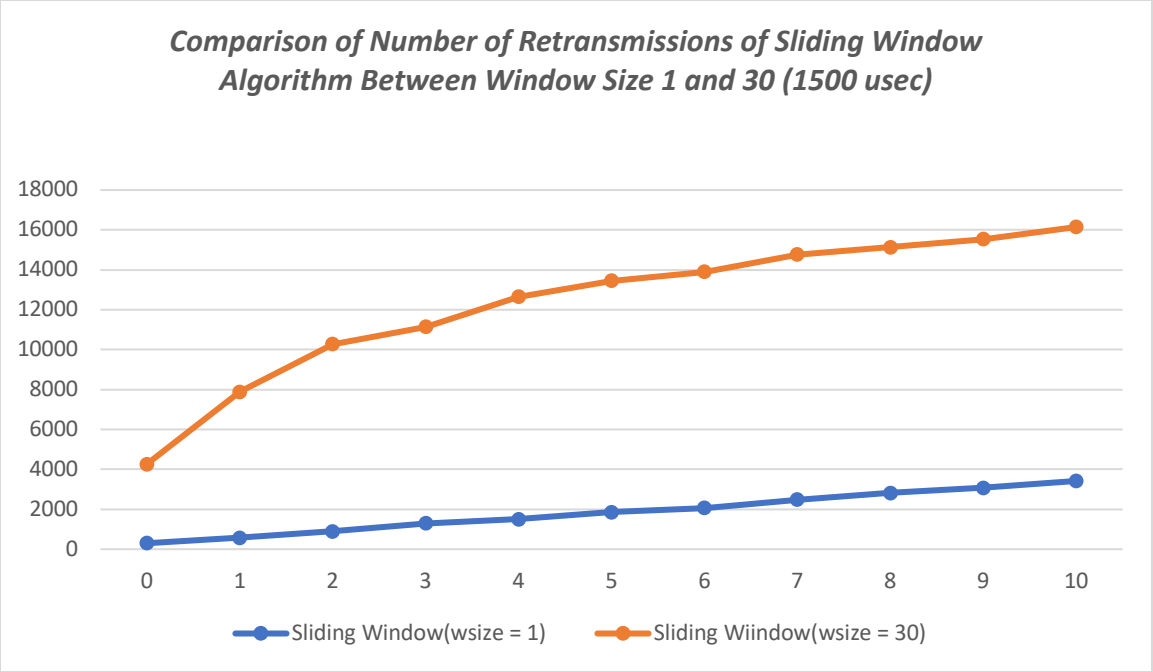



***Figure 9 Comparison of Number of Retransmissions of Sliding Window Algorithm Between Window Size 1 and 30***

The figure above shows the number of retransmissions of sliding window algorithm with different window sizes 1 and 30. The sliding window algorithm of both window size shows linear progression. From the result, I could see sliding window algorithm with the bigger window size has the distinguishably greater number of retransmissions for each dropping rate.

# Discussion

## (1) Difference in performance between stop and wait and sliding window

After comparing performances of stop-and-wait and sliding window algorithm, I could see that the time performance of stop-and-wait algorithm and that of sliding window with window size 1 are very close to each other. This is because the way stop-and-wait algorithm works of handling each individual message is similar to the sliding window algorithm with window size 1 in that sliding window can only move on to receive next message if the window size has available slot.

On the other hand, as window size gets bigger, I could vividly see the difference in the time performance between stop-and-wait algorithm and sliding window algorithm. Sliding window algorithm has distinctively better time performance as window size increases because it can continue sending messages whenever the in-transit messages are smaller than the size of the window. This is one of the advantage of the sliding window algorithm as it can send multiple messages and waits for the cumulative acknowledgement from the server.

### (2) Influence of window size on sliding window performance

As I mentioned before, as window size of sliding window algorithm gets bigger, the time performance of the algorithm become better than the stop-and-wait algorithm because sliding window algorithm can send multiple messages by not receiving each individual acknowledgement for a particular message. When it comes to the time performance, as Figure 6 shows, sliding window shows fairly similar performance over all the sizes after size 5.

On the other hand, when it comes to the number of retransmissions, sliding window algorithm with bigger window sizes have greater number of retransmissions than that of the smaller window sizes. I think this is because when a retransmission occurs, sliding window algorithm with bigger window sizes have to resend more packages that are the size of the window.

### (3) Difference in the number of messages retransmitted between stop-and-wait and sliding window algorithms

Between stop-and-wait algorithm and sliding window algorithm, sliding window algorithm has much greater number of messages retransmitted. Since the 1500 usec is not short enough to observe retransmission times between stop-and-wait and sliding window algorithms, I have set to 400 usec in order to measure retransmission times.

For the stop-and-wait algorithm with TIME_OUT=400 usec, I could observe that the number of retransmissions = 3. When I run stop-and-wait algorithm many times, sometimes stop-and-wait algorithm doesn't have any number of retransmissions at all. On the other hand, for sliding window algorithm, it generally has number of retransmissions with TIME_OUT=400 usec. For window size 14, it has the greatest number of retransmissions which is 308. I could conclude that sliding window algorithm with window size greater than 2 has generally greater number of retransmissions because when a retransmission occurs, it has to send the messages that are size of the window.

### (4) Include discussion of the effect of drop rates on both the window size of 1 and 30. You should also discuss about the difference in the number of messages retransmitted between stop-and-wait and sliding window algorithms.

For the time performance, window size 30 has much faster time performance comparing with that of window size 1. As Figure 9 shows, the graph of sliding window algorithm with window size 1 increases linearly; on the other hand, the graph of sliding window algorithm with window size 30 increases logarithmically. In addition, because of dropping rate, the time performance of both window sizes is not faster comparing to that of time performance without dropping rate. I assume that is because we set dropping of messages purposely, it will take more time for retransmitting messages which might not occur in the function without dropping of messages.

When it comes to the number of retransmissions, window size 30 has much larger retransmission times comparing with that of window size 1. This is similar to the case that the number of retransmissions is smaller for stop-and-wait than that of sliding window algorithm. Sliding window algorithm with window size 1 resend each message when retransmission occurs; on the other hand, sliding window algorithm with window size 30 resends multiple messages that are size of the window when retransmission occurs. As Figure 10 shows, there are big difference in the number of retransmissions between two different

algorithms and shows linear progression for both algorithms as dropping rate increases, which I have assumed before I tested the case.