**Design Document**

Title and Authors

Phase 5: Implementation of Go-Back-N Protocol over an unreliable UDP Channel

Team Member(s): Michael Burton, Jocelyn Frechette, Jesse Hayes Lewis

Introduction:

In this project phase, the purpose was to implement the transfer protocol known as Go-Back-N. In this phase, a jpeg image is sent over an unreliable UDP channel. In the Go-Back-N protocol, packets are sent in a pipelined fashion. In pipelining, packets are sent in burst in a defined window. For the Go-Back-N protocol to be reliable, the sender will send a burst of packets in the given window. The receiver will then send acknowledgements for the packets that it receives. Once an acknowledgement for the oldest packet is sent, the defined window will slide to the right and send the newest packet. From the previous project phases, the UDP channel is considered unreliable with the implementation of both packet corruption and loss, where packets are chosen at random on both sending and receiving side of the UDP channel. Meaning that both data packets and acknowledgement packets are susceptible to loss and corruption. In order to combat the unreliable UDP channel in the Go-Back-N Protocol, the sending side has a timer for the oldest in flight packet, if an acknowledgement is not received for that specific packet, a timeout will occur causing for all packets in the defined window to be resent to the receiver. On the receiving side, the receiver will continue to send the for last received packet. This process will continue until the receiver receives all packets in order. Once all packets are received in order, the data will be then pushed up to the application layer and the delivered jpeg image should show no errors.

Flowchart for sender:

A diagram of a diagram

Description automatically generated

Flowchart for receiver:

A diagram of a workflow

Description automatically generated

Code Description:

The description of the code will be split by the sending (UDPClient.py) and the receiving side (UDPServer.py) each broken down into their respective code blocks.

UDPClient.py

In this code block, the python modules necessary to execute the code are imported and all constants are defined including, the states and overall packet structure.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

The MakePacket() function takes in filename and a packet size. The file is open using the open() function. We begin assembling the packet by appending the ACK field, the sequence number, and the data for the desired packet size. Then, the checksum is calculated using the checksum() function which is then appended into the individual packet. This initial step is for the first packet of data to be sent out. In the while() loop, this process is repeated until the end of the file is reached. The main difference change in the MakePacket() function is that sequence numbers are set from 0 to 255, due to the sliding window of the Go-Back-N sender.

A black rectangle with white text

Description automatically generated

In this code block, a new function was added in order to assign sequence number from 0 to 255, this function increments the sequence number based on the previous set sequence number.

A screenshot of a computer screen

Description automatically generated

This block of code is a block functions used to implement the state machine. The split\_ack\_packet() function chops up the received packet and returns the packet in its respective parts which includes the checksum, the ACK response, and the sequence number. The is\_ACK() function takes the split packet and confirms that ACK sequence is what we are expecting. The is\_corrupt() function checks to if the received checksum and the calculated checksum on are not the same. If they are not same, then the packet is corrupt.

A computer screen with text

Description automatically generated

The corruptor() function takes in a byte object, which in the case can either be data packet or an ACK packet, performs a bit-wise NOT which is AND with 0xFF (255 in decimal) bit mask. The corrupted packet is returned to the main script.

A screen shot of a computer program

Description automatically generated

The checksum() function takes the data portion of the packet and calculates a custom checksum. The custom checksum takes a sum of all the bytes plus the second byte, and then takes 2 bytes from the end, and returns that as a checksum.

A computer screen shot of a program

Description automatically generated

The code block above are the send() and receive() methods within the UDP Client class. The send() method uses the socket.sendto() function to send packets from the sender to the defined receiver port and IP address. The receive() method is where the timer for the RDT 3.0 protocol is defined. The socket.settimeout() function takes in the timeout window that is set by the user upon initialization of the UDPClient code. If there is no timeout present, the message will fall under the “try” case which will send out over the UDP channel. In the case of error/loss in the channel, the recvfrom() has a blocking call which not allow the code to move forward without the timeout present. So when the timer timeouts, a “Timeout Error” will be returned by the “except” case. Further explanation of this “Timeout Error” is explained in the state machine code block which is to follow.

A computer screen shot of a program

Description automatically generated

Within this code block is the constructor of the UDPClient. Parameters that are inherent to the class defined here included the timeout, window size, the length of the packet list, initialization of sequence numbers, initialization of the base pointer, and the indices that access of the array related to the sequence number and base are also initialized. Additionally, the determination of lost and corrupted packets are also performed here.

A computer screen shot of a code

Description automatically generated

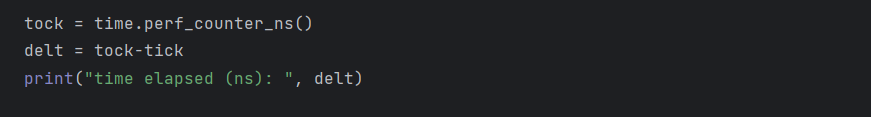
The code block above is a newly added function from previous phases, the corr\_loss\_send() function takes in the class and the index that is to be sent next in the packet array. The corruption and loss levels are determined at runtime of the code by user input. From the user input, the percentage is divided by 100 to help determine the probability of corruption or loss. Once in this function, the corruption and loss is determined prior to sending by the np.random.choice() python function. This function follows a Bernoulli distribution based on the probability of corruption or loss. This Bernoulli distribution can be equated to a weight coin flip, where 1 denotes a corrupted or loss packet and a 0 denotes a packet with no corruption or loss. If the packet is slated for corruption and no loss, the packet will be put through the corruptor function prior to being sent. If the packet is slated for no loss and no corruption, the packet will simply by sent and its correct form. Lastly, if the packet is slated for loss and no corruption, the packet will simply not be sent.

A screen shot of a computer program

Description automatically generated

The function above known as next\_state() is the implementation of state machine logic. The function takes in the instantiation of the client, and the size of the packet to be sent. The state machine starts with the sending portion of the state machine the checks to see if next sequence number is less than base pointer plus the window size as well as checking if the index of the next sequence number is still within the packet list. If these conditions are met, the index of the next sequence number is checked to see if it slated for corruption or loss and then is sent. The sequence number is then incremented as well as the index for the next sequence number.

The else condition is the receiving portion of the sending side. If a message is successfully received, the packet is split up into its respective components, and checked for corruption. If it is not corrupt, the base pointer and index of the base is incremented. If there is a timeout due to errors in the UDP channel, the packets are then resent for the given window size.

A screenshot of a computer program

Description automatically generated

The code block above is where sender is run, the port number, address, timeout (in millseconds), corruption percentages, lost percentages, and window sizes are defined here. Packets are packetized for the given file is executed here as well as the overall state machine is called. Additionally, performance timers are implemented for performance metrics which are shown in the plots in a subsequent section.

UDPServer.py

A screenshot of a computer program

Description automatically generated

The first block of code is definition of constants on the server side, this where the states are defined as certain numbers in order to implement logic in the code. The defined constants for sequence numbers for ACKs are no longer used in this phase as ACKS are now assigned on a 0 to 255 basis.

A computer screen shot of a program code

Description automatically generated

Deliver\_data writes the packet\_array (data at this point) to a file\_name in the local directory.

Corruptor – see UDPClient.py explanation

A screen shot of a computer program

Description automatically generated

checksum-see UDPClient.py explanation

A screen shot of a computer program

Description automatically generated

split\_packet-works similarly to split\_ack\_packet from UDPClient.py but gets the data as well and returns it.

A screen shot of a computer

Description automatically generated

is\_ack-see UDPClient.py explanation

is\_corrupt-see UDPClient.py explanation

A black background with numbers and symbols

Description automatically generated

inc\_seq\_num()-see UDPClient.py explanation

A computer screen shot of a black background

Description automatically generated

The receive() function within the UDPServer class simply takes the ACK packet and sends to the UDPClient.

**A computer screen shot of a program

Description automatically generated**

The code block above is the initialization of the UDPServer class, which is the receiving side for data. In this code block all necessary parameters inherent to the UDPServer class is initialized here. Parameters include the initial state, port, address, socket, expected sequence number, calculation of the checksum, and assembling of the ACK packets.

**A screenshot of a computer program

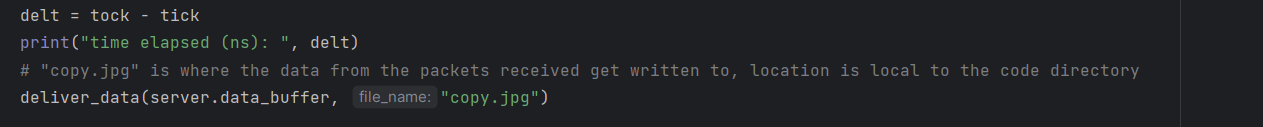
Description automatically generated**

The code block above is the beginning of the next\_state() function, which is the implementation of the state machine logic for the server. First the server receives the first packet, the packet is unpacked and checked for corruption, if the packet is not corrupt, the packet is put into buffer for later delivery to the application layer. The ACK packet is then assembled and is ready to be transported. The expected sequence number is then incremented.

A computer code on a black background

Description automatically generated

The code block above is the continuation of the next\_state() function, at this point, corruption and loss of the ACK packet is determined. This is done by using a random number generator that follows a Bernoulli distribution for both corruption and loss, If the packet is slated corruption, the packet is run through the corruptor function and sent. Similarly, packet loss is determined, if the packet is not selected to be lost, the ACK packet is then sent to the client.

**A computer screen with many colorful text

Description automatically generated**

The code block above is where the UDPServer is executed. The address and port number is defined, loss and corruption percentages are defined, and the UDPServer class is instantiated. At the first while loop, the server is waiting for the first packet to be sent over correctly. Following the first successfully received packet, the next while loop can then be executed, the loop will execute for as long as the packet index is less than number of packets received. If the previous sequence number is not equal to the expected sequence number, the packet index is then incremented. Once all packets are received, the data is then delivered to the application layer for the received data to be reassembled. The received data should result in a copy of the original jpeg file. Additionally, there are performance timers implemented for assessment of the performance of the UDPServer. The performance of the UDPServer with respect to loss/corruption percentages, window sizes, and timeout are shown in a subsequent section.

**Execution Screenshots:**

Scenario #1: No bit errors or loss

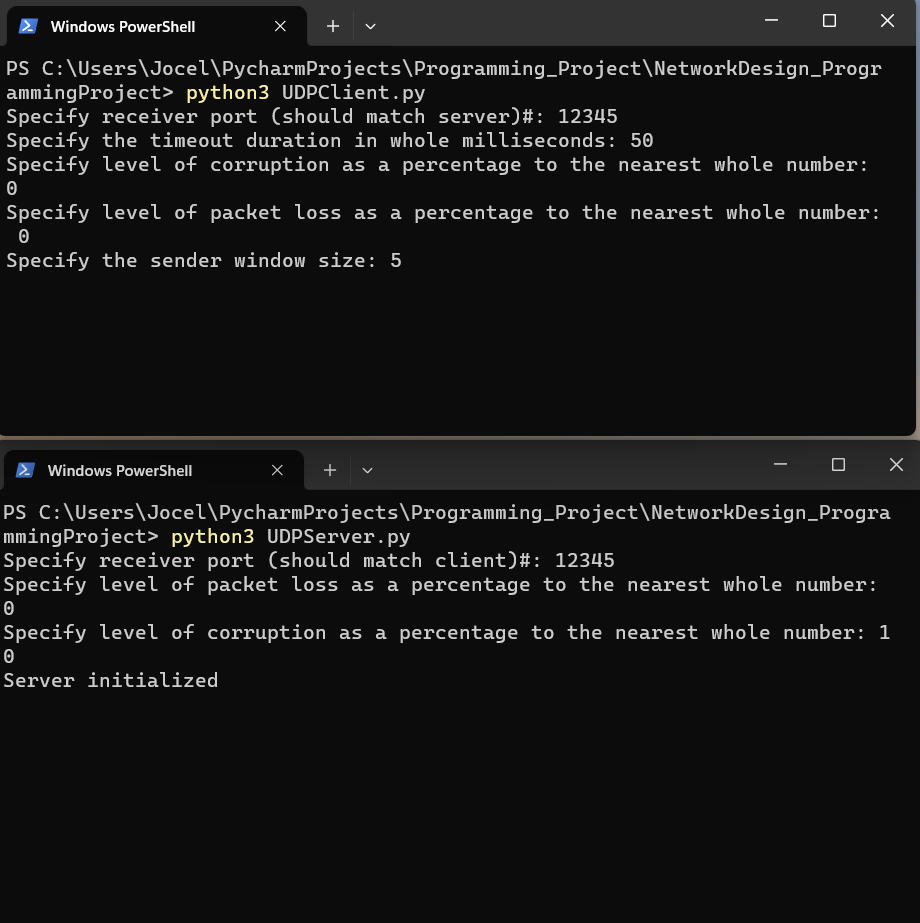
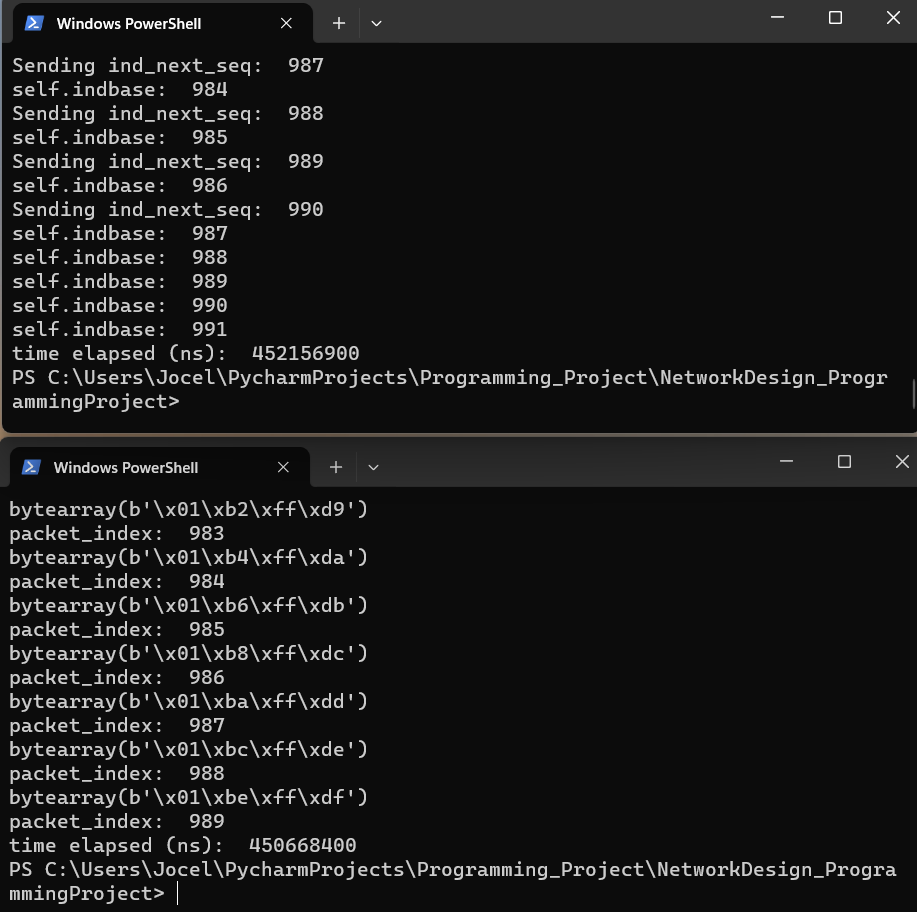
A screenshot of a computer program

Description automatically generatedThe screenshot below on the left is the at the initialization stage, both the sender and the receiver both have 0% packet loss and 0% corruption. A window size of 5 is set and a timeout of 50 ms is set on the sending side. The screen below of the right shows completion stage of code execution.

A screenshot of a computer program

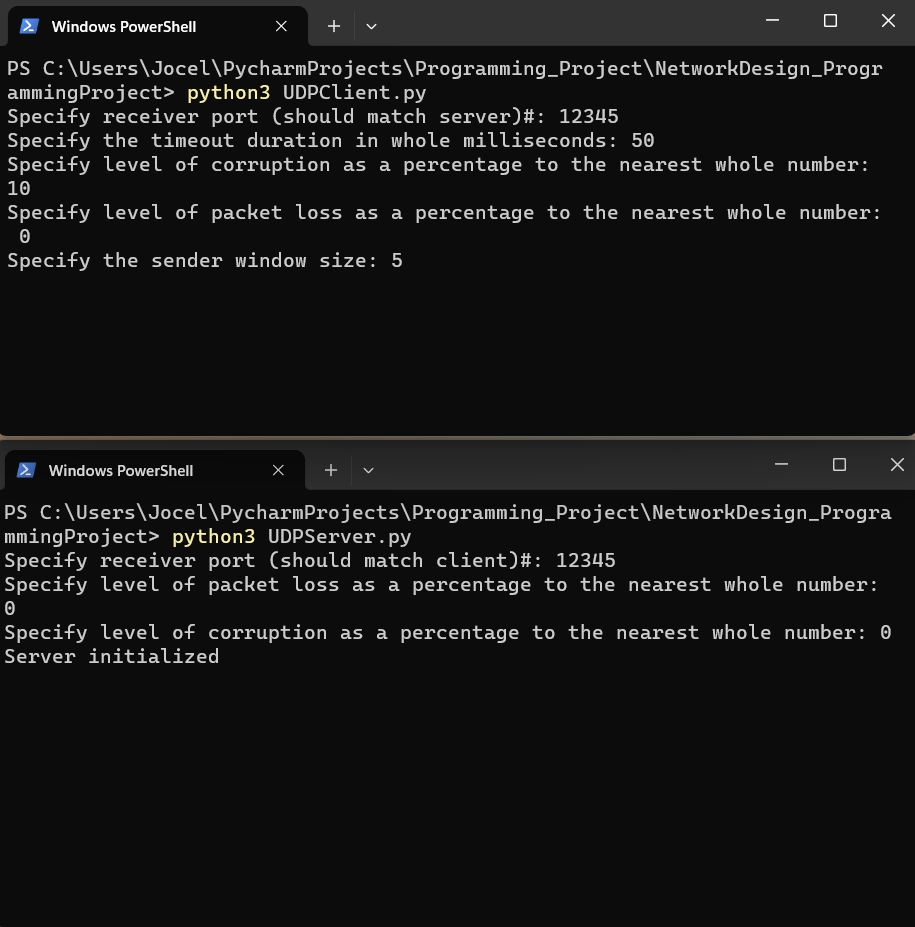
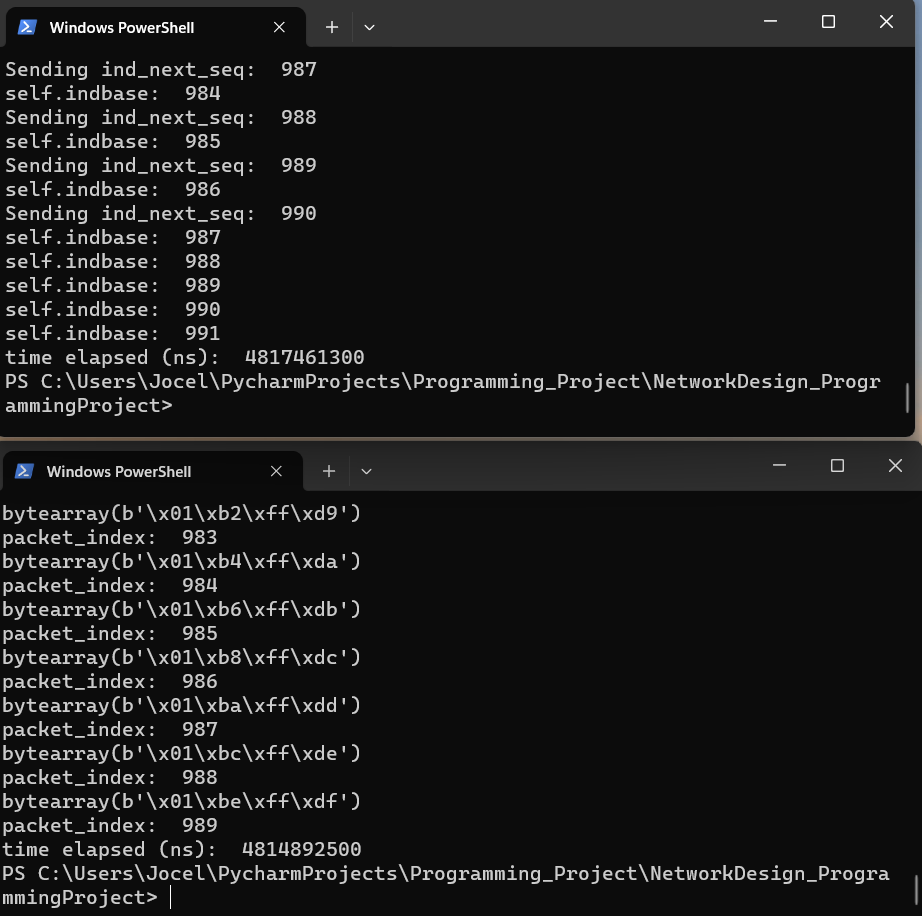
Description automatically generated

Scenario #2: ACK Packet Bit Error

The screenshot below on the left shows the initialization step of scenario 2 where acknowledgment bit errors are present. The bottom screen shows that 10% acknowledgment packet corruption, all other packet corruption is shown to be 0%. Additionally, the sending side has a window size of 5 and a timeout of 50 milliseconds. The screenshot below on the right shows the post-execution of the code with successful data transfer.

Scenario #3: Data Packet Bit Error

The screenshot on the left shows the initialization stage of scenario 3, on the top screen, the corruption percentage has been set to 10%, all other corruption and loss percentages have been set 0%. The sending side has a timeout of 50 milliseconds and a window size of 5. The screenshot on the right shows the execution after completion of the data transfer, showing that the data transfer was successful.



Scenario #4: ACK Packet Loss

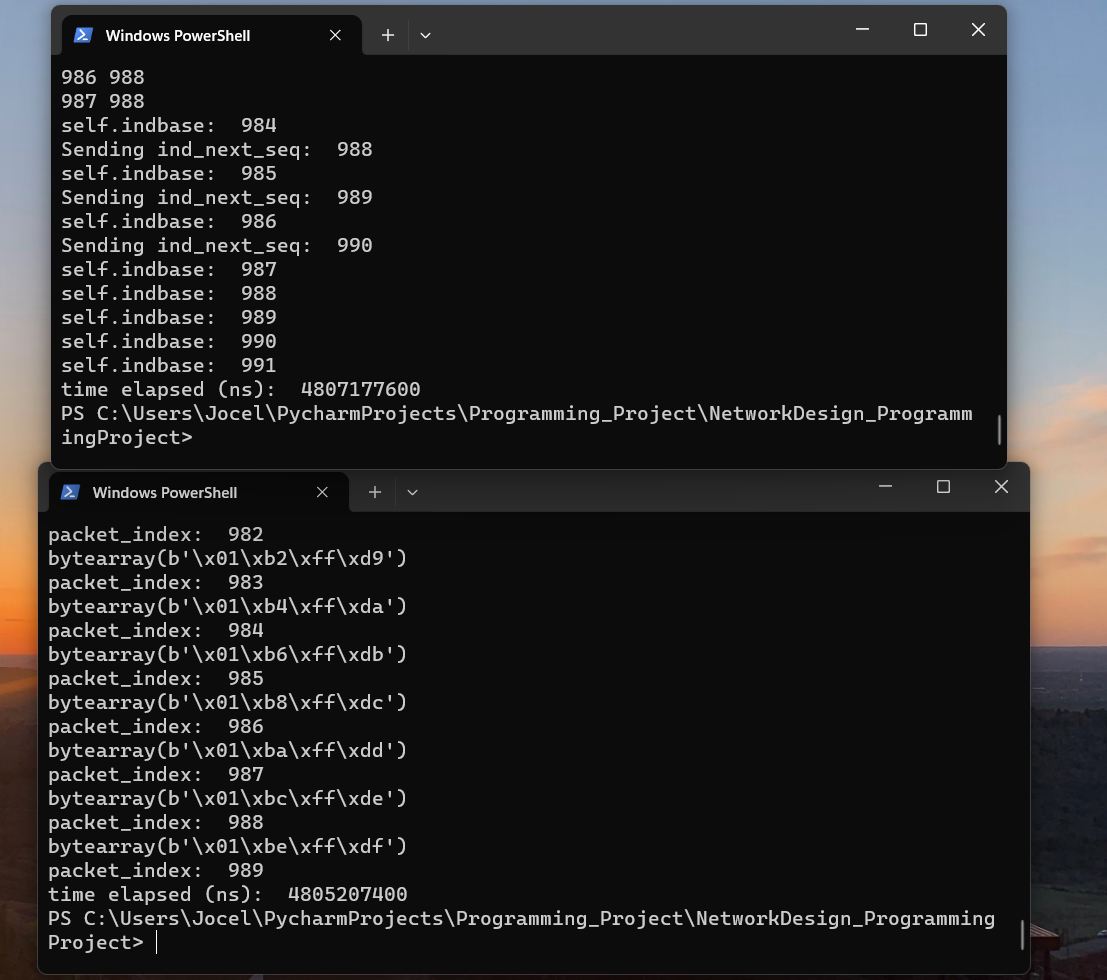
A screenshot of a computer program

Description automatically generatedThe screenshot below on the left shows the initialization step of scenario 4, it can been on the bottom screen that the ACK packet loss is set to 10%, all other corruption and loss percentages are shown to be set to 0%. The sending side timeout and window size to set similar to all other scenarios. The screenshot on the right shows the completion of the execution, showing a successful data transfer.

A screenshot of a computer program

Description automatically generated

Scenario #5: Data Packet Loss

A screenshot of a computer program

Description automatically generatedThe screenshot on the left shows the initialization stage for scenario 5, on the top screen, the packet loss percentage is set to 10%, all other packet loss and corruption percentages are set to 0%. The sending side timeout and window side is set similarly to all other scenarios. The screenshot on the right shows the completion of the execution indicating a successful data transfer.

**Performance Plots:**

**Client-Side Packet Corruption:**

The following data points were captured by sweeping the random-corruption rate setting on the Client-Side from a value of 0% to 60%, in steps of +5%. At each Client-Side corruption rate, the Elapsed Time shown is the average across 10 separate runs at that specific Client-Side Corruption setting.

Constants for Client-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Server-Side Corruption set to 0%.
* Client-Side Loss set to 0%
* Server-Side Loss set to 0%
* Timeout = 40ms

From the collected data we can see both the client completion time and server completion time increase linearly together as the Client-side corruption percentage is increased, as seen by the extrapolated trend line for each respective completion time.

**Client-Side Packet Loss:**

The following data points were captured by sweeping the random-packet loss rate setting on the Client-Side from a value of 0% to 60%, in steps of +5%. At each Client-Side loss rate, the Elapsed Time shown is the average across 10 separate runs at that specific Client-Side loss setting.

Constants for Client-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Server-Side Corruption set to 0%.
* Client-Side Corruption set to 0%
* Server-Side Loss set to 0%
* Timeout = 40ms

From the collected data we can see both the client completion time and server completion time increase linearly together as the Client-side loss percentage is increased, as seen by the extrapolated trend line for each respective completion time.

**Server-Side Packet Corruption:**

The following data points were captured by sweeping the random-corruption rate setting on the Server-Side from a value of 0% to 60%, in steps of +5%. At each Server-Side corruption rate, the Elapsed Time shown is the average across 10 separate runs at that specific Server-Side Corruption setting.

Constants for Server-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Client-Side Corruption set to 0%
* Client-Side Loss set to 0%
* Server-Side Loss set to 0%
* Timeout = 40ms

From the collected data we see both the Client and Server completion times increase relatively close to one another as the Server-side corruption increases. Once again, we see behavioral differences between sweeping the client-side corruption setting and the server-side corruption setting. Instead of seeing a linear increase in total completion time we see an **exponential** increase as the server-side corruption is increased. This is most likely due to the way our client-side code handles invalid or missing Ack messages from the server.

**Server-Side Packet Loss:**

The following data points were captured by sweeping the random-Packet loss rate setting on the Server-Side from a value of 0% to 60%, in steps of +5%. At each Server-Side loss rate, the Elapsed Time shown is the average across 10 separate runs at that specific Server-Side loss setting.

Constants for Server-Side Corruption vs Completion Time:

* Port 4005 used for all tests.
* Client-Side Corruption set to 0%
* Client-Side Loss set to 0%
* Server-Side Corruption set to 0%
* Timeout = 40ms

From the collected data we see both the Client and Server completion times increase relatively close to one another as the Server-side loss increases. Once again, we see behavioral differences between sweeping the client-side corruption setting and the server-side loss setting. Instead of seeing a linear increase in total completion time we see an **exponential** increase as the server-side loss is increased. This is most likely due to the way our client-side code handles invalid or missing Ack messages from the server. Another item of note when comparing phase 3 results to phase 4 results, we see a much larger value for Elapsed Time as the corruption/loss parameters are increased towards 60%. We see delays of up to 70 seconds to transfer the file using only timeout’s and no invalid Ack ID’s like we did in Phase 3.