ECS 152A: Computer Networks

Fall 2022

Project 3

(100 points)

Due Date: Monday, November 21st, 2022 by 5 PM

Reliable and Congestion Controlled Data Transfer over UDP

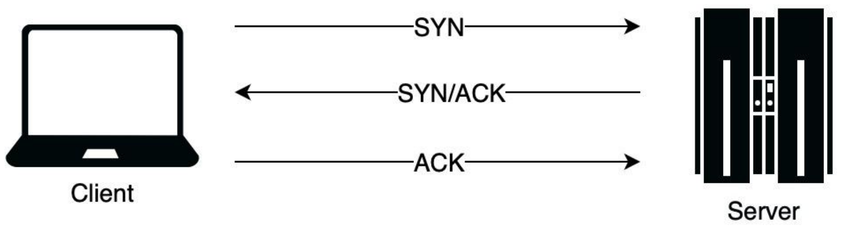
The goal of this project is to create a reliable version of UDP which is resistant to network congestion. The project is divided into three different parts to help you slowly build towards the reliable, connection oriented and congestion controlled UDP Berryessa.

Part 1: Adding Connection Support to UDP (UDP Putah) (20 points)

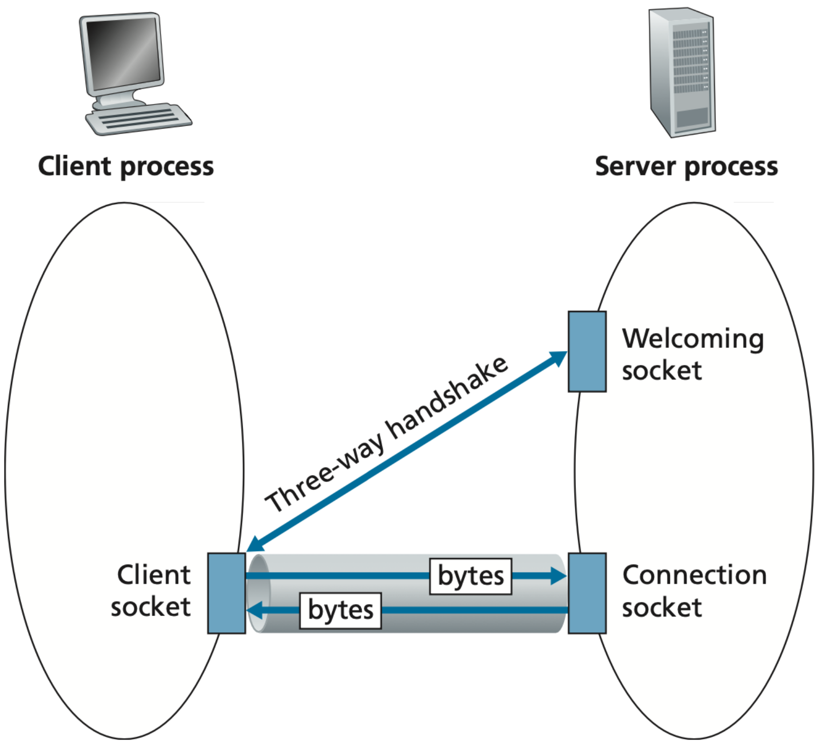
We have learned over the course of this quarter that UDP and TCP differ fundamentally in how they treat connections. While TCP requires a three-way handshake before any data can be transferred, a UDP socket can directly start sending data. While UDP's approach is certainly faster and easier to use, having persistent connections brings with itself benefits like a separate data socket for each connection, allowing servers to parallelize data transfer between multiple simultaneous users, while also ensuring reliability for each data stream.

In this part, you will add connection support to UDP using a three-way handshake (just like TCP!) and create a new version of UDP called UDP Putah.

When a client connects to a server, it will send a handshake message to the server. If the server is willing to connect, it will send the response back to the client telling it that a connection has been established on the server's end. Finally, the client also sets up a connection and sends an acknowledgement for it back to the server.



Now here comes the tricky part! If you refer back to section 2.7 of the book, TCP has two different types of sockets: a welcoming socket and a connection socket.



The welcoming socket is responsible for listening to incoming connection requests and performing three-way handshakes, while the connection socket is responsible for sharing the actual data. In TCP's own implementation, the client is abstracted from this whole process. It uses the same port and IP address for both connecting to the server and sharing the data. However, because we are building our reliable transport layer on top of UDP, we do not have the liberty of using the same IP and port for multiple different sockets (refer back to our discussion on multiplexing differences between TCP and UDP, discussed in detail in section 3.2). To overcome this issue, while still maintaining roughly the same process as TCP follows, you will implement the following:

1. A server will listen to the incoming requests using the "*accept*" function (so far we are following TCP to a tee)
2. A client will connect to a listening server using the "*connect*" function (so far so good)
3. A three-way handshake will take place between the client and the server (seems familiar)
4. Two new UDP sockets will be created (one on the client and one on the server) and their port numbers will be shared with each other (here we go)
5. All further communication will happen using these two new sockets (which will be running in their separate threads) while the main threads will go back to either accepting or connecting to new clients (we now have two brand new sockets to share data)
   Note: *Remember that UDP does not have accept and connect functions by default. You will be adding these functions to mimic and implement the functionality described above for UDP Putah.*

You are allowed flexibility in your implementation of this detail in your code, however, there are a few constraints to keep in mind:

1. Your code should have separate welcoming sockets and separate connection sockets.
2. You are not allowed to use more than three messages to create a new connection between the client and the server. All the information sharing about the new connection sockets should happen in the same three messages.
3. You should be able to keep running the welcoming socket in a thread to accept new connections while the connection socket should have the capacity to run in a separate thread to transfer data.
4. You are not allowed to add any fields to the custom TCP header that you will design which are not present in the actual TCP header. More discussion about the header is at the end of this part.

Evaluation

For this part, you have to do the following:

1. Run the server using the following command:
   ```
   python3 server_putah.py --ip XXXX.XXXX.XXXX.XXXX --port
   YYYY
   ```
   Note: *As most of you will be running this on a local computer, the IP will be "localhost" more often than not.*
2. The server will start listening for incoming connections. It should now be able to accept incoming connections
3. Run the first client using the following command:
   ```
   python3 client_putah.py --server_ip XXXX.XXXX.XXXX.XXXX
   --server_port YYYY
   ```
   Note: *Because we are less concerned about what port the client runs on (as long as it is a different port than the server), there is no need to provide a specific port to the client. It can rely on the operating system to provide it with a suitable, empty port number.*
4. Upon running the command, the client should initiate a three-way handshake with the server. The server will respond back to the handshake and establish a connection. ***Note:*** *Remember that SYN and SYN/ACK messages do not contain any message body, just the header.*
5. Upon the successful establishment of the connection, the client and the server should both print and log the IP address and Port number of the new connection sockets created (remember that the welcoming socket for a server will run on the port that we provide in the command, while the new connection sockets will run on completely different ports).
6. The client would now send data to the server (in this case, a simple "ping" message), while the server would respond with a simple "pong" message to the client. This ping pong should keep on happening indefinitely until we terminate the client using CTRL+C or CMD+C (a keyboard interrupt exception). Upon termination, the client or server being terminated will send a *FIN* message to the other host and wait for an acknowledgement before closing the connection socket. The other host will send back an acknowledgement and close the connection.
   a. There can be conditions where the ACK is lost (especially in later parts), for those cases, you will wait for the ACK to come before closing the connection, however, if the ACK never arrives, you will resend the *FIN* packet at most three times (including the first time it was sent), and if there's still no ACK, you will close the socket and assume that the other host is closed as well.

7. Now open another terminal window and run another client using the same command as step 3. Another client should spin up and start playing ping-pong with the server.
8. Log all the interactions and messages between the two clients and the server in the following format:

`Source | Destination | Message Type | Message Length | TimeStamp`

Note: *The source and destination here would be the port numbers. The message type tells us which kind of message is being sent. Is it an SYN, SYN/ACK, ACK, DATA, or a FIN message? Here DATA packet is any packet which contains actual data to be transferred and is not one of the other SYN/ACK/FIN packets.*

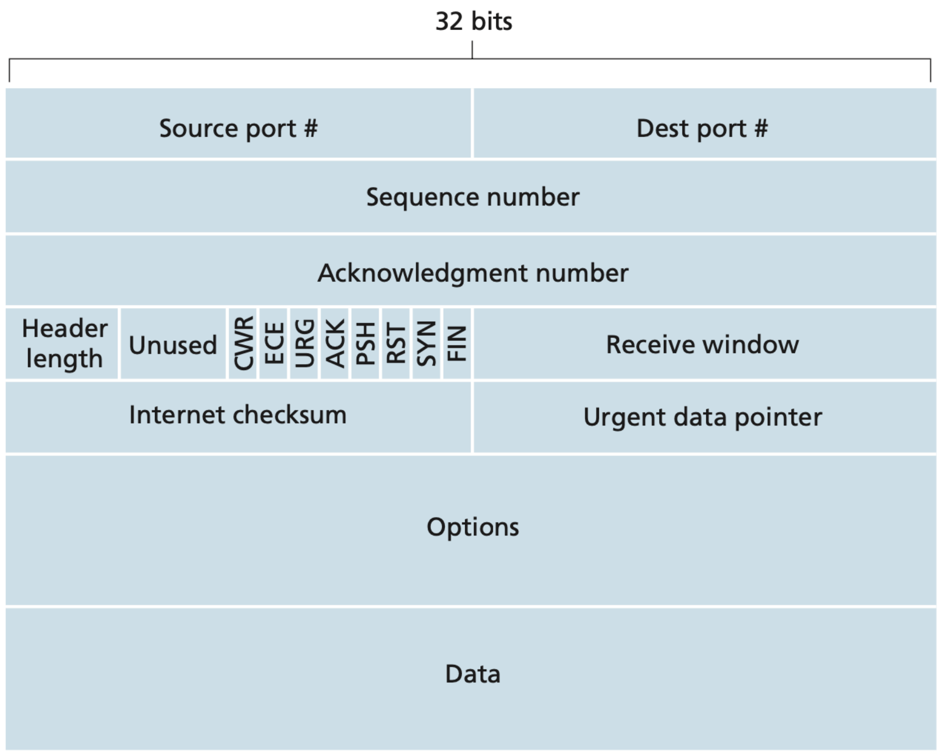
Answer the following questions as part of your report:

1. What are the differences between multiplexing done in a UDP socket vs a TCP socket? What would happen if we were to use the same IP and Port number for transferring data to different clients in our implementation of UDP Putah? (3 points)
2. Why does TCP require three messages to establish the connection? What would go wrong if you were to attempt building the connection with two messages? (3 points)
3. Explain how you implemented the sharing of connection socket port numbers between the client and the server in your implementation. Justify your choice of header and message content in sharing this information. (4 points)

Along with the answers to these questions, submit the python code for the server and client, along with the log files that you generated during your own testing (10 points).

Header for Custom UDP Implementation

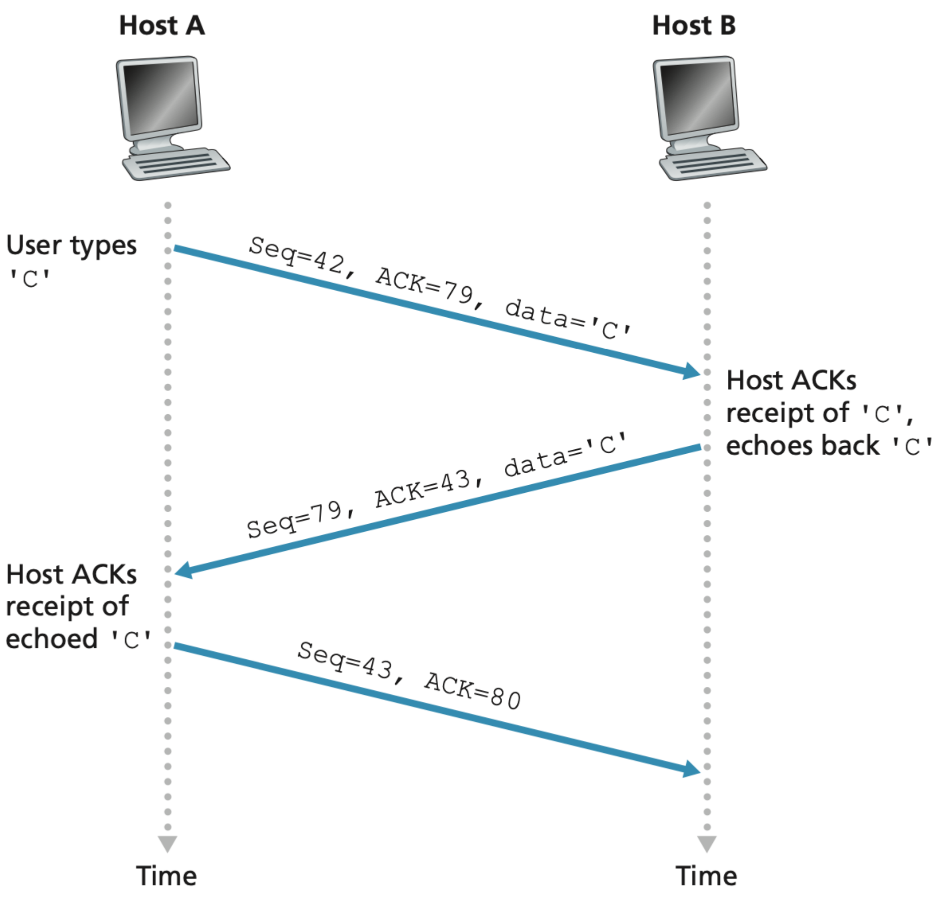
As part of each step in this project, you will create a separate header for your implementation. Here's the catch though. You can choose to not include fields from the original TCP header in your implementation, however, you are not allowed to add any new fields which are not originally present in the TCP header. And if you add any field, be sure to follow the exact size limit of that field. Be mindful and creative of your choices. You will also be judged on the number of fields you include in your header. If you include a header field which was not required, you will be penalised for it.



Part 2: Adding Reliability to our UDP Putah (UDP Solano) (20 points)

Now that we have our connection-oriented UDP Putah which can listen to and connect to multiple different clients, it's time to add some reliability to it. Remember from our discussions in class that TCP accomplishes reliability using acknowledgements. For each message that the sender sends to the receiver, the receiver must acknowledge it to let the sender know that the message was received successfully. If there's any packet loss, the acknowledgement will never arrive and the sender will have to resend the packet back to the receiver.

You will implement a similar acknowledgement process as TCP, where instead of acknowledging each individual packet, you will acknowledge the number of bytes that the receiver has successfully received so far. This is known as cumulative acknowledgement. If the receiver has received 1500 bytes so far, it will expect a packet starting with the sequence number 1501. Thus as an acknowledgement for the packet containing a sequence number of 1000, and 500 bytes, the receiver will send an acknowledgement back for 1500 bytes. You will be implementing a variation of the stop-and-wait algorithm for this part (you will be sending only one packet at a time and then waiting for its acknowledgement).



Evaluation

For this part, you have to do the following:

1. Because you are running this on a local server, and the information sharing is happening between two processes running on a local host, there will not be any major bandwidth constraint or packet loss. However, because we want to make things difficult for both you and your implementation of a UDP Solano, you will induce artificial packet loss and delay by doing the following:
    a. Your receiver (or what we were calling a server in part 1), will accept two additional arguments, namely, *packet loss percentage* and *round trip jitter*. The value of both of the *packet loss percentage* will be between 0 and 100, while the value of the *round trip jitter* will be a decimal number between 0 and 1.
    b. For each packet received, you will generate a random integer between 0 and 100. If the value of that random integer is equal to or less than the packet loss percentage, you will drop that packet and not send any acknowledgement back to the sender.
    c. Before sending each acknowledgement, you will generate a random decimal between 0 and 1. If the value of the random decimal is larger than the round trip jitter, you will sleep the receiver for that value in seconds before sending the acknowledgement back to the sender.
    d. Use the default value of packet loss percentage as 10 and round trip jitter of 0.5

2. You will run the receiver using the following command:
   ```
   python3 receiver_solano.py --ip XXXX.XXXX.XXXX.XXXX
   --port YYYY --packet_loss_percentage X
   --round_trip_jitter Y --output output.txt
   ```
3. You will run the sender (or what you called the client in the previous part) using the following command:
   ```
   python3 sender_solano.py --dest_ip XXXX.XXXX.XXXX.XXXX
   --dest_port YYYY --input input.txt
   ```
   Note: *You can assume for this part that the file will always be a text file, however, you should read and send the file as a regular binary file, which would make it agnostic to the file type used.*
4. You will use these two files to test your program:
   a. https://raw.githubusercontent.com/shaoormunir/ecs152a-fall-2022/main/alice29.txt
   b. https://raw.githubusercontent.com/shaoormunir/ecs152a-fall-2022/main/big.txt
5. The sender will connect to the receiver using the three-way handshake implemented in part 1. They will also share with each other the sequence numbers they will be using (it's important to remember here that for both reliability and security concerns, the sequence numbers used by the sender and receiver in TCP do not start from 0, rather they start from a randomly generated 32-bit number, read more about it here: Understanding TCP Sequence and Acknowledgment Numbers - PacketLife.net).
6. After establishing the connection and opening new sockets for data transfer, the sender will start sending the file to the receiver. The file will be transferred in packets of **size 1000 bytes** (including the additional header for your UDP Solano). Because the receiver is implementing packet loss and delay on its end, there will be dropped packets and the acknowledgements will arrive at a different time. Your sender should be able to:
   a. Resend correct packets which are lost
   b. Tweak the timeout at its end to account for the delay in packet sending (the timeout should not be too long as to increase the time that it takes for the whole file to be sent and it should not be too short as to introduce a lot more packet loss and timeouts). You can do this by implementing the following timeout interval formula from section 3.5.3:

   $$Timeout\ Interval = EstimatedRTT + 4 * DevRTT$$

   Where *EstimatedRTT* is calculated from the weighted average of previous RTTs and is calculated as:

   $$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

   DevRTT represents the variability in the round trip time and is calculated as:

   $$DevRTT = (1 - \beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT|$$

   Go through section 3.5.3 and this RFC carefully to learn more about default values suggested for each of these variables and when to update the timeout interval.
7. You will calculate the following statistics for the file transfer and output them at the sender's terminal:
   a. Time taken to transfer each file
   b. Total bandwidth achieved (amount of data sent/total time taken) for each file

   c. Packet loss observed (Packets lost/Packets Sent) for each file
8. You will also log all the interactions and messages between the two clients and the server in the following format:

`Source | Destination | Message Type | Message Length | TimeStamp`

   Note: *The source and destination here would be the port numbers. The message type tells us which kind of message is being sent. Is it an SYN, SYN/ACK, ACK, DATA, or a FIN message? Here DATA packet is any packet which contains actual data to be transferred and is not one of the other SYN/ACK/FIN packets.*
9. Because our UDP Solano can accept multiple clients; run the same command as step 3 on a different terminal to simultaneously start another file transfer. The receiver should be able to handle both file transfers at the same time.
   a. Note: As the receiver only gets one filename when running, if we save the data received from all clients under the same filename, it would result in the file being overwritten. To avoid this, instead of directly saving the file in the same directory as the code is running, you will save the file in the subdirectory represented by the port number of the sender. For example, if the sender is using port 5000 for its data socket, you will save the output file in *5000/{output.txt}*.
10. You will repeat this process once for the smaller file, and then again for the larger file.

Answer the following question as part of your report:

1. Explain your implementation of sequence and acknowledge numbers. Why do both sender and receiver maintain separate sequence and acknowledgement numbers? (3 points)
2. Explain and justify the additional header fields you needed to add compared to part 1 for this implementation. (3 points)
3. How would the performance of this file transfer have been affected if we did not use separate welcoming and connection sockets? (4 points)

Along with your report, submit the python code for both the sender and receiver, along with the log files generated (10 points).

Part 3: Adding Congestion Control to UDP Solano (UDP Berryessa) (60 points)

For this last part, we will be adding a flavour of congestion control to UDP Solano (making it a connection-oriented, reliable, congestion-controlled UDP Berryessa, quite a mouthful).

Remember from our lectures and discussions (along with your reading of sections 3.6 and 3.7), congestion control is another salient feature of TCP. In the previous part, we were sending only one packet at a time and then waited for its acknowledgement. But in this part, we will dynamically increase the number of packets that we are sending and adjust the different parameters based on how TCP Tahoe and TCP Reno implement congestion control.

Evaluation

For this part, you have to do the following:

1. Introduce another terminal argument to the receiver: *bdp* (bandwidth-delay product). This argument represents the maximum amount of data that can be sent on the network at a single time. The receiver should be aware of the rate at which the sender is sending data. If the rate at which the sender is sending data (number of packets sent in one window) exceeds the bdp, the receiver would enter into a congestion state. In a congestion state, you will multiply the packet loss percentage by three and triple the amount of time that the receiver sleeps when the round-trip jitter condition is triggered. Use the default value of bdp as 20,000. (***Hint:*** *remember that the congestion window is implemented on the sender's side, not the receiver's side. To send that congestion window size to the receiver, you can repurpose the receive window field in the TCP header specification.)*
2. You will use the following command to run the receiver.
   ```
   python3 receiver_berryessa.py --ip XXXX.XXXX.XXXX.XXXX
   --port YYYY --packet_loss_percentage X
   --round_trip_jitter Y --bdp Z --output output.txt
   ```
3. On the sender's side, you will implement both TCP Tahoe and Reno (You can read more about it in section 3.7 and by following this link: TCP Tahoe and TCP Reno - GeeksforGeeks). We should be able to switch the TCP implementation using a terminal switch *tcp_version*.
4. You will reuse the logic implemented in the previous part for setting a variable timeout interval based on network conditions.
5. You will use the following command to run the sender:
   ```
   python3 sender_berryessa.py --dest_ip XXXX.XXXX.XXXX.XXXX
   --dest_port YYYY --tcp_version tahoe/reno --input
   input.txt
   ```
6. You will log the following information for each message sent:

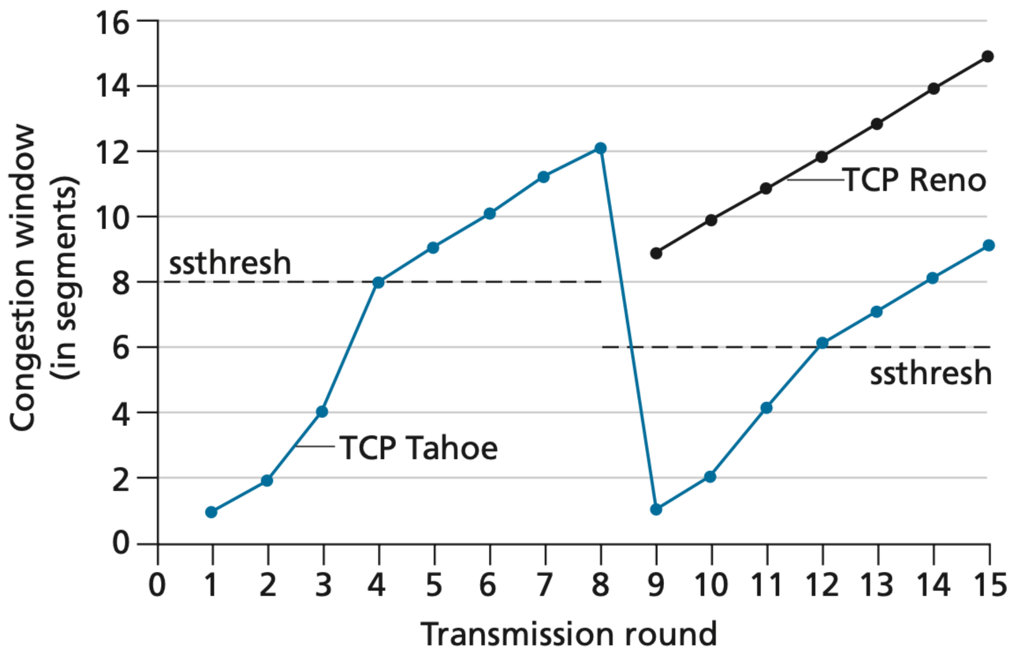
   Source | Destination | Message Type | Message Length | State | CWND | TimeStamp

   Note: *Here the state will tell the overall congestion control state the network is following. It can vary between congestion avoidance and slow start. The CWND will inform you about the window size currently being used by the sender.*
7. Similar to the previous part, run another client using the same command used in part 4. Flip the TCP version for this client. The receiver should be able to connect to both clients at the same time and successfully complete the transfer of files.
8. You will use a starting value of 16 for the Slow Start Threshold and a window size of 1.
9. Similar to the previous part, you will repeat this process once for the smaller file, and once for the larger file.

For this part you will calculate the following statistics at the sender's end:

1. Time taken to transfer each file
2. Total bandwidth achieved (amount of data sent/total time taken) for each file
3. Packet loss observed (Packets lost/Packets Sent) for each file

In addition to these statistics, you will also monitor the change in congestion window size after each transmission round (when all packets in a CWND have been sent and acknowledged) to plot graphs similar to the following for both Tahoe and Reno implementations (10 points):



Answer the following questions as part of your report:

1. Explain and justify the additions to your packet header to implement this part as compared to part 2. (3 points)
2. Based on the graphs that you generate, list down each time the state of your network changes (from a slow start to congestion avoidance and vice versa), explain the reason behind the change, and report the values of CWND and Slow Start Threshold (SSThreshold) before and after the state change. (12 points)
3. Compare the bandwidth difference between TCP Tahoe and Reno in your experiments. Compare it with the bandwidth that you measured in part 2. Do you notice any significant difference between these implementations? Explain why there is or why there is not a difference. (10 points)
4. Based on your experience and results in designing these different models, what else can be done to improve the bandwidth usage of your network? (5 points)

Along with the report, submit the python code for the receiver and sender along with the log files generated (20 points).

Testing Environment:

All submissions will be tested on Python 3+.

Late Submission Policy:

No late submissions are allowed. However, if you barely miss the deadline, you can get partial points up to 24 hours. The percentage of points you will lose is given by the equation below. This will give you partial points up to 24 hours after the due date and penalises you less if you narrowly miss the deadline.

Total Marks you get = (Actual Marks you would get if NOT late) 1 - hours late24

Late Submissions (later than 24 hours from the due date) will result in zero points, *unless you have our prior permission or documented accommodation*.

Best of luck

Submission Page

I certify that all submitted work is my own work. I have completed all of the assignments on my own without assistance from others except as indicated by appropriate citation. I have read and understand the university policy on plagiarism and academic dishonesty. I further understand that official sanctions will be imposed if there is any evidence of academic dishonesty in this work. I certify that the above statements are true.

Team Member 1:

Full Name (Printed)	Signature	Date

Team Member 2:

Full Name (Printed)	Signature	Date