Project 1 Report

Authors: Corey Nguyen (ctsnguyen@ucdavis.edu), Chotrawit Benko (csomsri@ucdavis.edu)

I. Implementation

A. Stop and Wait Protocol

In the implementation of this protocol, we decided to put the data in a list from the mp3 files and then binded the sender port with local host and a random arbitrary port number.

Then we start sending packets until we run out of packets, but before we send the next packet we wait for an acknowledgement (ACK) sent from the receiver.py. We put this in a try and except block so that if we fail to receive an ACK, the exception will trigger and we will go into timeout.

Also before we send the packet we format it by adding the header based on the sequence ID and turning it into bytes at the size of the expected Sequence ID size. Then adding the payload with it the prefix and payload, which is just the data.

How we check whether we are able to see if we received a valid ACK is if the acknowledgement sequence is the same as the base acknowledgement sequence. Basically the base acknowledgement sequence shows the previous and by incrementing it by 1, we can know what the expected acknowledgement sequence ID would be. Therefore by checking whether the received ACK is the same as the next base acknowledgement (Expected, we shall know if it is valid or not.

Finally we finally close off the sender that we sent b'==FINACK==', which tells the receiver that we have sent all of our packages and both sockets can close.

B. Fixed Sliding Window Protocol

For our implementation of the sliding window sender protocol, we opted to use our previous Stop and Wait protocol as a base to build upon, as they were both essentially identical protocols aside from their sender window size. This difference in sender window size was the crux of our implementation for the sliding window protocol.

Unlike the Stop and Go protocol, we made use of a while loop to put 100 packets into flight, and to keep 100 packets in flight at any given time. Within this while loop, we would keep track of the most recently acknowledged packet, which we defined as the lower bound of the data among file.mp3 we wished to send, and that bound + the window size being the upper bound of the data we wished to send. In other words, while we were still receiving acknowledgements, we would ensure 100 packets were in flight by taking the most recently acknowledged packet and sending the next 100 data payloads after it.

In conjunction with our sliding window adjustment technique for sending data, we also changed our receiver logic to adapt to this. We made it such that if a timeout occurred before an ACK was received, the program would resend all 100 packets starting from the byte after the most recently acknowledged byte.

C. TCP Reno

For TCP Reno, we implemented a TCP Reno class to which its core function involves a state machine to which will go to the next state based on output strings of each class method.

The tcp_reno class constructor defines all the macros/constants that we need, this includes the initial state being Slow Start, initializing our Base ACK sequence ID, Next ACK sequence ID, and the congestion window being initially 1 and the slow start threshold being 64.

The class starts with tcp_reno.run() where we create sockets based on UDP and create a list of valid States that includes Slow Start, AIMD, Recovery, and Retransmit. Each State is defined by its own class method and each of them will return the string to which the next class is supposed to be.

Again we start in the Slow Start state to which we put our data in flight by sending until the next sequence ID - the base sequence ID is less than the congestion window. After putting our packets in flight. For each Acknowledgement (ACK) that we receive, we will increase the congest window by 1. Doing this until the congestion window >= the slow start threshold (ssthresh).

When the congestion window becomes equal or greater than the slow start threshold, the class switches states from Slow Start to AIMD, effectively slowing down the growth of number of packets in flight from exponential to additive size increase of the congestion window size increase. By using slow start up until ssthresh and switching to AIMD afterward, this allows us to maximize the amount of round-trip time we spend in a domain of window sizes that are close to the highest we can achieve without causing congestion collapse, effectively optimizing our throughput.

Once we surpass the congestion window past the cliff and cause a congestion collapse indicated by three ACKs received, we switch to the Fast Retransmit state to identify this and swap to the Fast Recovery state. In this state, we quickly divide the congestion window size by two and redefine this as the new ssthresh in which we use as the range to separate between usage of Slow Start and AIMD after the congestion collapse.

II. Results

```
Sent!
Entering Loop
Sending seq=5319688, data='170'
ACK received: 5319689
Delay Per Packet Time: 0.0000898
Sent!
Entering Loop
Sending seq=5319689, data='170'
ACK received: 5319690
Delay Per Packet Time: 0.0001282
Sent!
Entering Loop
Sending seq=5319690, data='170'
ACK received: 5319691
Delay Per Packet Time: 0.0001046
Sent!
Entering Loop
Sending seq=5319691, data='170'
ACK received: 5319692
Delay Per Packet Time: 0.0001018
Sent!
Entering Loop
Sending seq=5319692, data='170'
ACK received: 5319693
Delay Per Packet Time: 0.0001062
Sent!
All data sent. Closing stop and go socket.
Throughput Time: 1167.7144456
Average Delay per Packet Time: 0.0001468
```

Sliding Window

```
Window moved forward!
Entering Loop
ACK received: 5319688
Delay Per Packet Time: 0.0000015
Window moved forward!
Entering Loop
ACK received: 5319689
Delay Per Packet Time: 0.0000018
Window moved forward!
Entering Loop
ACK received: 5319690
Delay Per Packet Time: 0.0000036
Window moved forward!
Entering Loop
ACK received: 5319691
Delay Per Packet Time: 0.0000019
Window moved forward!
Entering Loop
ACK received: 5319692
Delay Per Packet Time: 0.0000018
Window moved forward!
Entering Loop
ACK received: 5319693
Delay Per Packet Time: 0.0000015
Window moved forward!
All data sent. Closing sliding window socket.
Throughput Time: 234.7666142
Average Delay per Packet Time: 0.0000031
```

TCP Reno

```
Sending seq=2159988, data='255'
Sending seq=2159989, data='229'
Sending seq=2159990, data='158'
Sending seq=2159991, data='105'
Sending seq=2159992, data='191'
Sending seq=2159993, data='253'
Sending seq=2159994, data='239'
Sending seq=2159995, data='243'
Sending seq=2159996, data='249'
Sending seq=2159997, data='63'
Sending seq=2159998, data='151'
Sending seq=2159999, data='249'
Sending seq=2160000, data='101'
Sending seq=2160001, data='0'
Sending seq=2160002, data='144'
Sending seq=2160003, data='3'
Sending seq=2160004, data='56'
Sending seq=2160005, data='238'
Sending seq=2160006, data='16'
Sending seq=2160007, data='211'
Sending seq=2160008, data='72'
Sending seq=2160009, data='131'
Sending seq=2160010, data='17'
Sending seq=2160011, data='83'
Sending seq=2160012, data='43'
Fast retransmit seq=2159876
All data sent. Closing TCP Reno socket.
Throughput Time: 163.5129322
Average Delay per Packet Time: 0.0000025
```

Stop and Wait	Throughput	Average Delay per Packet	Metric
Trial #1	976.2289256	0.0001163	6019.209464
Trial #2	898.4546034	0.0001065	6573.039489
Trial #3	897.3800841	0.000106	6604.042799

Trial #4	2989.560544	0.0004062	1724.185888
Tiral #5	3415.927619	0.0004694	1492.290224
Trial #6	3079.467466	0.0004308	1625.807777
Trial #7	999.2038117	0.0001226	5709.924557
Trial #8	1363.929947	0.000173	4046.651954
Tirial #9	1269.981379	0.0001592	4397.365919
Trial #10	1167.714446	0.0001468	4768.742685
Average	1705.784882	0.00022368	4296.126076
Standard Deviation	1021.588657	0.0001485315515	2038.013241
Fixed Sliding Window Protocol			
Trial #1	243.5333316	0.0000032	218750.0731
Trial #2	246.3061623	0.0000034	205882.4268
Trial #3	263.3162957	0.000004	175000.079
Trial #4	255.9410661	0.0000036	194444.5212
Tiral #5	247.8576806	0.0000033	212121.2865
Trial #6	249.3534233	0.0000033	212121.2869
Trial #7	249.990431	0.0000033	212121.2871
Trial #8	240.5629946	0.0000032	218750.0722
Tirial #9	234.4636782	0.0000031	225806.522
Trial #10	234.7666142	0.0000031	225806.522
Average	246.6091678	0.00000335	210080.4077
Standard Deviation	8.948015896	0.0000002718251072	15456.66536
TCP Reno			
Trial #1	155.48142	0.0000025	280000.0466
Trial #2	151.9500517	0.0000025	280000.0456
Trial #3	153.1534956	0.0000025	280000.0459
Trial #4	180.4113273	0.0000023	304347.8802
Trial #5	167.0626662	0.0000023	304347.8762
Trial #6	151.4944313	0.0000022	318181.8636
Trial #7	185.4618062	0.0000022	318181.8738
Trial #8	151.2768493	0.0000026	269230.8146
Tirial #9	151.3503	0.0000026	269230.8146
Trial #10	163.5129322	0.0000025	280000.0491
Average	161.115528	0.00000242	290352.131
Standard Deviation	12.78042339	0.0000242	19036.903

```
Stop and Go 2:
All data sent. Closing stop and go socket.
Throughput Time: 898.4546034
Average Delay per Packet Time: 0.0001065

Stop and Go 3:
All data sent. Closing stop and go socket.
Throughput Time: 897.3800841
Average Delay per Packet Time: 0.0001060

Stop and Go 4:
All data sent. Closing stop and go socket.
Throughput Time: 2989.5605436
Average Delay per Packet Time: 0.0004062

Stop and Go 5:
All data sent. Closing stop and go socket.
Throughput Time: 3415.9276185
Average Delay per Packet Time: 0.0004694

Stop and Go 6: At Size FULL
All data sent. Closing stop and go socket.
Throughput Time: 3079.4674663
Average Delay per Packet Time: 0.0004308

Stop and Go 7: At Size FULL
All data sent. Closing stop and go socket.
Throughput Time: 999.2038117
Average Delay per Packet Time: 0.0001226

Stop and Go 8: At Size FULL
All data sent. Closing stop and go socket.
Throughput Time: 1363.9299468
Average Delay per Packet Time: 0.0001730

Stopa nd Go 9: At Size FULL
All data sent. Closing stop and go socket.
Throughput Time: 1269.9813789
Average Delay per Packet Time: 0.0001592

Stop and Go 10: At Size FULL
All data sent. Closing stop and go socket.
Throughput Time: 1167.7144456
Average Delay per Packet Time: 0.0001468
```

```
Sliding Window 1: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 243.5333316
Average Delay per Packet Time: 0.0000032

Sliding Window 2: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 246.3061623
Average Delay per Packet Time: 0.0000034

Sliding Window 3: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 263.3162957
Average Delay per Packet Time: 0.0000040

Sliding Window 4: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 255.9410661
Average Delay per Packet Time: 0.0000036

Sliding Window 5: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 247.8576806
Average Delay per Packet Time: 0.0000033

SLiding Window 6: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 249.3534233
Average Delay per Packet Time: 0.0000033

Sliding Window 7: At Size FULL
All data sent. Closing sliding window socket.
Throughput Time: 249.9904310
Average Delay per Packet Time: 0.0000033

Sliding Window 8:
All data sent. Closing sliding window socket.
Throughput Time: 240.5629946
Average Delay per Packet Time: 0.0000032

Sliding Window 9:
All data sent. Closing sliding window socket.
Throughput Time: 234.4636782
Average Delay per Packet Time: 0.0000031

Sliding Window 10:
All data sent. Closing sliding window socket.
Throughput Time: 234.7666142
Average Delay per Packet Time: 0.0000031
```

```
TCP Reno 1:
All data sent. Closing TCP Reno socket.
Throughput Time: 155.4814200
Average Delay per Packet Time: 0.0000025

TCP Reno 2:
All data sent. Closing TCP Reno socket.
Throughput Time: 151.9500517
Average Delay per Packet Time: 0.0000025

TCP Reno 3:
All data sent. Closing TCP Reno socket.
Throughput Time: 153.1534956
Average Delay per Packet Time: 0.0000025

TCP Reno 4:
All data sent. Closing TCP Reno socket.
Throughput Time: 180.4113273
Average Delay per Packet Time: 0.0000023

TCP Reno 5:
All data sent. Closing TCP Reno socket.
Throughput Time: 167.0626662
Average Delay per Packet Time: 0.0000023

TCP Reno 6:
All data sent. Closing TCP Reno socket.
Throughput Time: 151.4944313
Average Delay per Packet Time: 0.0000022

TCP Reno 7:
All data sent. Closing TCP Reno socket.
Throughput Time: 185.4618062
Average Delay per Packet Time: 0.0000022

TCP Reno 8:
All data sent. Closing TCP Reno socket.
Throughput Time: 151.2768493
Average Delay per Packet Time: 0.0000026

TCP Reno 9:
All data sent. Closing TCP Reno socket.
Throughput Time: 151.3503000
Average Delay per Packet Time: 0.0000026

TCP Reno 10:
All data sent. Closing TCP Reno socket.
Throughput Time: 163.5129322
```

III. Citations

Citations for Stop and Wait Protocol:

Citations for Fixed Sliding Window Protocol:

Citations for Fixed Sliding Window Protocol:
Used ChatGPT for Algorithm Explanation and Debugging
Used GeeksforGeeks for explanation of TCP Reno:
https://www.geeksforgeeks.org/computer-networks/tcp-reno-with-example/

IV. Submission Page

Include this signed page with your submission
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.

For Group Submissions,
Team Member 1's contributions to this assignment:
Short Summary: I implemented TCP Reno, pair programmed Stop and Wait, and added Metrics for both Stop and Wait and Sliding Window. I also made a few helper scripts to check up on the progress of the project. I also wrote the Stop and Wait technical Report.
Full Name: Chotrawit Benko
Signature Date: 2/13/26

Team Member 2's contributions to this assignment:
Short Summary: I pair programmed the Stop and Wait protocol with Chotrawit. I also programmed the Fixed Sliding Window protocol and added metrics for the TCP Reno protocol. I also recorded some of the results for the Fixed Sliding Window protocol, but as the time differed greatly from Chotrawit's readings due to differing hardware, we opted to use Chotrawit's readings instead. For the report, I wrote the implementation details for the Fixed Sliding Window and wrote half of the TCP Reno implementation details.
Full Name: Corey Nguyen
Signature Date: 2/13/26