Alex Kwiatkowski
Birju Patel
Dr. Amy Babay
September 26, 2022

Project 1: Submission Document

Final Design, Packet Types

We utilized two types of packets for two different functions of the protocol. Data transmission from the sender to the receiver was done via a UPKT, while cumulative acknowledgements and negative acknowledgements were transferred from receiver to sender via a RCV_MSG.

UPKT:

- Flags
    - IS_RCV_MSG: Identifies if the packet is a UPKT or a RCV_MSG. Set as false.
    - INIT: Used by client to ask server to initiate new connection. Used by the server to ask the client to begin data transfer.
    - SLEEP: Used by the server to signal to the client that it is busy, and the client should wait.
    - FIN: Used by the client to mark the end of data transmission. Used by the server to confirm the termination of the connection.
- SEQ: The sequence number of the packet.
- PAYLOAD_LEN: The amount of data sent in this packet.
- PAYLOAD: Data transmitted as an array of characters.

RCV_MSG:

- Flags
    - IS_RCV_MSG: Identifies if the packet is a UPKT or a RCV_MSG. Set as true.
- ARU: All received up to. The server has received all packets up to this packet. The client should slide its window up to this.
- NUM_GAPS: Number of gaps included in this message
- GAPS: Ordered array of all the gaps currently known by the receiver. A gap is a series of packets in a row inside the window that have been dropped that the receiver knows about.
    - Gap Struct :
        - Start: start of a gap (inclusive)
        - End: end of a gap (not inclusive)

Final Design, Data Structures

The window for the sender is an array of UPKTs. This allows for efficient indexing when we have to resend packets as we can use the sequence number % size of the window. The buffer of packets for the receiver is a singly-linked list that is ordered by sequence numbers and also contains the size of the packet payload and the payload itself. The gap list maintained by the

receiver is a doubly-linked list whose nodes point to gap structs. Since both the buffer and the gap list see frequent manipulation in the middle of the list, a linked list was the structure that made the most sense to use.

We created a structure conn to hold the data for each connection. When a client attempts to connect to the server, the server creates a new conn structure to track the state of this connection. This structure stores the state of the connection as either INITIALIZING, SLEEPING, ACTIVE, or CLOSING. In order to handle multiple senders, all conn structures are put into a queue as they are created. This allows the server to handle clients in a FIFO manner.

Final Design, Client Side

Initialization Sequence

The client begins by setting up a UDP socket and parsing the target IP address and target file name. It then constructs an INIT packet whose payload consists of the target file name. The client sends this packet to the server in a loop every 250ms. The client begins in the INITIALIZING state.

If the client receives a response packet, it may be an INIT packet or a SLEEP packet. A SLEEP packet indicates that the server is busy, so the client transitions to the SLEEPING state. An INIT packet indicates that the server is ready for the client, so the client transitions to the ACTIVE state and should begin transmission.

Sleeping Sequence

If the client is in the SLEEPING state, it will enter this sequence. The client simply sleeps until it receives an INIT packet from the server. This indicates that it should wake up, transition to the ACTIVE state, and begin transmission. If no INIT packet is received in 10 minutes, we assume the server is down and close down.

Closing Sequence

After the final packet has been acknowledged, the client knows the server has received the entire file, so it closes the connection. It sends FIN packets repeatedly to the server every 250ms, and stops only when it receives a FIN in response.

Final Design, Server Side

Initialization Sequence

When the server receives an INIT packet from the client, there are four distinct possibilities. The server could be busy or not busy, and the packet could be a duplicate or a new request.

If the server is not busy, we can create and handle this new connection immediately. We create a new conn structure, set it as the current connection, place this connection in the active state, and send the client an INIT packet to ask the client to begin transmission.

If the server is busy, we first check to see if this INIT packet is a duplicate resent because the client has not received our response. If the INIT packet is a retransmission from the client we are currently serving, we resend an INIT packet. If it is a retransmission from a client that we

have placed in the queue, we resend a SLEEP packet. Each client is identified uniquely by its port and IP address, and this is compared to the port and IP from which we received our INIT. If, after checking the current connection and the queue, we find that this is a new client, we create a new connection and place this client at the back of the connection queue.

Closing Sequence

When we receive a FIN packet, we first verify that the sender of this FIN packet is the current client. If not, we disregard it. If it is, we send a FIN packet in response and transition this connection to the CLOSING state. We now set a timer for 3 seconds. If the client sends another FIN, we assume our FIN was dropped, resend it, and reset the timer. If we hear nothing when the timer expires, we assume the client has received the FIN. We close down the connection, close the file, reset sequence numbers and tracking data, and calculate summary statistics for the run. Then, we poll the next client from the connection queue, and send an INIT packet to it. If there is no next client, we enter an idle state and wait for the next INIT request from a new client.

Final Design, Transmission Sequence

After the initialization procedure has finished, the sender can start reading the file and creating packets as long as they fit in the window (number of packets in flight does not exceed window size). As the packets are created, the number of packets in flight is incremented, they are assigned a sequence number, the file data is read into them, they are sent to the receiver as well as saved to the sender's window in case they need to be retransmitted. Once the receiver receives a packet, two things may happen. If the packet is the first or its sequence number directly follows the previous packet's, the information in the packet is written to the file; additionally, an integer representing the start of the window and an integer representing the highest sequence number received are both set to the just received sequence number. Finally, any packets currently in the buffer that directly follow the just written packet's sequence number are also written to file and the integers are likewise updated accordingly. If the packet's sequence number is not directly after the previous packet's, this packet is added to the receiver's buffer ordered by sequence number. If this sequence number is greater than the highest received packet plus one, we have a new gap, starting at the highest received packet plus one and ending at the just received sequence number, that we must keep track of and add this gap to the end of the list. Otherwise, this packet fills part of a preexisting gap, and this gap either needs to be deleted if it no longer exists or split in two using this sequence number.

After a new gap has been detected, we have not sent a message for a while, or a large number of packets have been written to file since the last message (i.e. half the window size), the receiver will send the sender the RCV_MSG detailed above. The ARU is set to the integer representing the start of their window plus one to indicate all was received up to (but not including) this number. The size of the gap list is maintained in the struct and is used to set the NUM_GAPS, Finally, the gaps are copied from the receiver maintained gap list into the messages array GAPS.

After each packet is sent, the sender checks to see if it has received any messages from the receiver, and, if so, they process them accordingly. Using the cumulative acknowledgement from the receiver, the sender is able to subtract the difference between the ARU and the start of their window from the number of packets in flight and then set the start of their window to the ARU. Before sending the next packet, the sender resends all the packets indicated as lost in the GAPS array included in the message. However, if the sender has not moved their window recently, they resend the first packet in the window plus an additional few packets depending on an adjustable parameter. To avoid excessive resending of packets from this mechanism, the time before the window timeout is always greater than the receiver's message sending timeout.

Performance Results

LAN Results

File Size: 104.463782MB

Final Results - Run 1

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.103572	11.475032	X
0% loss - ncp	18.277288	5.715497	120.135853
1% loss - ncp	22.061841	4.735044	124.242053
5% loss - ncp	29.923872	3.490985	133.630453
10% loss - ncp	39.421555	2.649915	147.102653
20% loss - ncp	38.592588	2.706835	148.022453
30% loss - ncp	116.206282	0.898951	228.849106
Same time - t_ncp	20.371318	5.127983	X
Same time - ncp	26.040211	4.011633	120.139106

Final Results - Run 2

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.177662	11.382396	X
0% loss - ncp	15.668344	6.667187	112.298653
1% loss - ncp	23.457828	4.453259	122.977853

5% loss - ncp	31.321347	3.335226	135.0880306
10% loss - ncp	36.218876	2.884236	147.062053
20% loss - ncp	61.894655	1.687767	192.125253
30% loss - ncp	116.267868	0.898475	229.984506
Same time - t_ncp	21.854644	4.779935	X
Same time - ncp	26.054637	4.009412	129.828053

Final Results - Run 3

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.178383	11.381502	X
0% loss - ncp	13.353379	7.823022	107.523253
1% loss - ncp	26.201003	3.987015	124.790853
5% loss - ncp	30.570024	3.417197	134.477453
10% loss - ncp	37.960355	2.751918	148.766306
20% loss - ncp	62.149431	1.680849	192.108906
30% loss - ncp	119.577239	0.873609	229.548653
Same time - t_ncp	20.164231	5.180648	X
Same time - ncp	28.166007	3.708860	129.242853

Final Results - Run 4

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.227009	11.321522	X
0% loss - ncp	13.674864	7.639109	108.623653
1% loss - ncp	21.558207	4.845662	132.789053
5% loss - ncp	32.438366	3.220377	137.063253
10% loss - ncp	35.747326	2.922282	147.913253

20% loss - ncp	62.750816	1.664740	189.423253
30% loss - ncp	106.872126	0.977465	231.802653
Same time - t_ncp	18.453862	5.660809	X
Same time - ncp	29.284632	3.567188	128.075253

Final Results - Run 5

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.021290	11.579694	X
0% loss - ncp	15.113015	6.912174	107.913853
1% loss - ncp	23.035468	4.534910	122.128053
5% loss - ncp	29.901281	3.493622	133.452653
10% loss - ncp	35.621842	2.932577	148.785453
20% loss - ncp	60.776313	1.718824	187.699853
30% loss - ncp	120.807263	0.864714	229.816506
Same time - t_ncp	17.748962	5.885628	X
Same time - ncp	26.145925	3.995414	121.435053

Final Results - Average

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.1415832	11.4280292	X
0% loss - ncp	15.217378	6.9513978	111.299053
1% loss - ncp	23.2628694	4.511178	125.385573
5% loss - ncp	30.830978	3.3914814	134.7423685
10% loss - ncp	36.9939908	2.8281856	147.9259436
20% loss - ncp	57.2327606	1.891803	181.8759436

30% loss - ncp	115.9461556	0.9026428	230.0002848
Same time - t_ncp	19.7186034	5.3270006	X
Same time - ncp	27.1382824	3.8585014	125.7440636

Artificial WAN Results
File Size 100.000012 MB

Final Results - Run 1

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	10.490203	9.532705	X
0% loss - ncp	70.984804	1.408752	100.074292
1% loss - ncp	94.438740	1.058888	102.784692
5% loss - ncp	128.931086	0.775608	112.842292
10% loss - ncp	149.142335	0.670501	130.438892
20% loss - ncp	187.682292	0.532815	174.821184
30% loss - ncp	226.847090	0.440826	212.304784
Same time - t_ncp	12.641325	7.910564	X
Same time - ncp	71.939793	1.390051	100.103692

Final Results - Run 2

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.499860	10.526472	X
0% loss - ncp	71.189472	1.404702	100.082692
1% loss - ncp	93.618300	1.068167	102.571892
5% loss - ncp	127.677160	0.783226	113.015892
10% loss - ncp	150.929816	0.662560	129.398692
20% loss - ncp	186.532566	0.536099	174.030692

30% loss - ncp	227.640233	0.439290	213.884368
Same time - t_ncp	14.074893	7.104851	X
Same time - ncp	72.015258	1.388595	100.233892

Final Results - Run 3

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.615678	10.399684	X
0% loss - ncp	71.092040	1.406627	100.078492
1% loss - ncp	94.190359	1.061680	100.000012
5% loss - ncp	128.418506	0.778704	112.587492
10% loss - ncp	150.178430	0.665875	130.564892
20% loss - ncp	183.384141	0.545303	174.398384
30% loss - ncp	230.077342	0.434637	214.742692
Same time - t_ncp	14.092797	7.095824	X
Same time - ncp	71.773981	1.393263	100.162492

Final Results - Run 4

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.604079	10.412244	X
0% loss - ncp	71.566931	1.397294	100.114892
1% loss - ncp	93.794543	1.066160	102.417892
5% loss - ncp	129.563156	0.771824	113.071892
10% loss - ncp	153.460020	0.651636	130.172384
20% loss - ncp	188.973544	0.529175	173.048784
30% loss - ncp	228.893484	0.436884	213.688492
Same time - t_ncp	14.602895	6.847958	X

Same time - ncp	72.166690	1.385681	100.267492

Final Results - Run 5

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.008723	11.100354	X
0% loss - ncp	71.125495	1.405966	100.098092
1% loss - ncp	93.158004	1.073445	102.300292
5% loss - ncp	126.370126	0.791326	112.104492
10% loss - ncp	149.165389	0.670397	129.839692
20% loss - ncp	185.292582	0.539687	174.908492
30% loss - ncp	229.698099	0.435354	212.923584
Same time - t_ncp	15.275178	6.546569	X
Same time - ncp	72.147567	1.386048	100.225492

Final Results - Average

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.6437086	10.3942918	X
0% loss - ncp	71.1917484	1.4046682	100.089692
1% loss - ncp	93.8399892	1.065668	102.014956
5% loss - ncp	128.1920068	0.7801376	112.724412
10% loss - ncp	150.575198	0.6641938	130.0829104
20% loss - ncp	186.373025	0.5366158	174.2415072
30% loss - ncp	228.6312496	0.4373982	213.508784
Same time - t_ncp	14.1374176	7.1011532	X
Same time - ncp	72.0086578	1.3887276	100.198612

WAN-EC Results
Locations: Virginia Tech and University of Vermont
RTT = 26.3 ms = 0.0263 sec
File Size 104.463782 MB

Final Results - Run 1

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.064166	11.524919	X
0% loss - ncp	51.176038	2.041264	115.683853
1% loss - ncp	56.091725	1.862374	119.564653
5% loss - ncp	68.458812	1.525936	136.024453
10% loss - ncp	76.495045	1.365628	157.247053
20% loss - ncp	86.393072	1.209169	216.366253
30% loss - ncp	96.336321	1.084365	277.087053
Same time - t_ncp	30.046702 sec	3.476714	X
Same time - ncp	64.028275	1.631526	118.100253

Final Results - Run 2

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	8.695804	12.013125	X
0% loss - ncp	51.225952	2.039275	114.394453
1% loss - ncp	55.980375	1.866079	119.269253
5% loss - ncp	68.812726	1.518088	136.623653
10% loss - ncp	75.889982	1.376516	156.680053
20% loss - ncp	85.656263	1.219570	218.582453
30% loss - ncp	96.559084	1.081864	276.601253
Same time - t_ncp	52.027321	2.007864	X

Same time - ncp	63.882504	1.635249	118.934653

Final Results - Run 3

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	8.669388	12.049730	X
0% loss - ncp	50.730737	2.059181	113.905853
1% loss - ncp	57.634147	1.812533	122.578853
5% loss - ncp	68.058432	1.534913	136.951253
10% loss - ncp	76.260615	1.369826	157.156053
20% loss - ncp	86.108672	1.213162	217.686453
30% loss - ncp	96.584873	1.081575	275.416318
Same time - t_ncp	8.924588	11.705166	X
Same time - ncp	63.673957	1.640605	116.995653

Final Results - Run 4

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
0% loss - ncp	8.669388	12.049730	X
1% loss - ncp	50.730737	2.059181	113.905853
5% loss - ncp	57.634147	1.812533	122.578853
10% loss - ncp	68.058432	1.534913	136.951253
20% loss - ncp	76.260615	1.369826	157.156053
30% loss - ncp	96.158744	1.086368	273.521706
Same time - t_ncp	56.172854	1.859684	X
Same time - ncp	66.118777	1.579941	120.236653

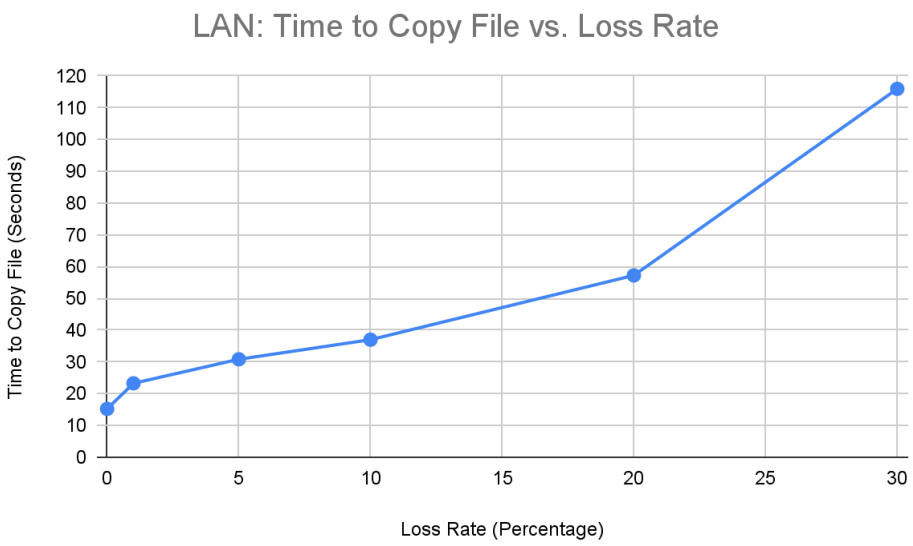
Final Results - Run 5

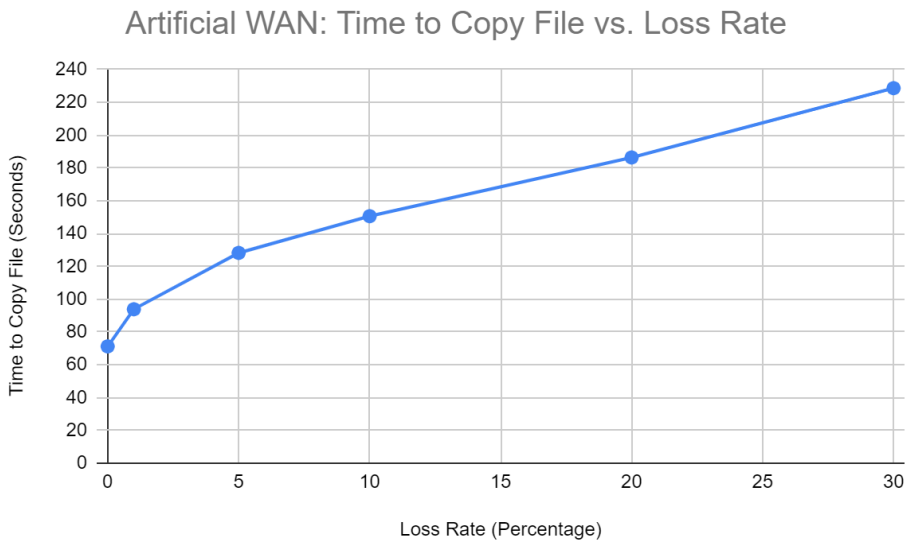
	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	9.098790	11.481063	X
0% loss - ncp	50.516295	2.067922	111.441853
1% loss - ncp	57.576045	1.814362	124.081053
5% loss - ncp	68.468189	1.525727	135.892853
10% loss - ncp	76.121865	1.372323	157.311453
20% loss - ncp	86.083362	1.213519	215.278906
30% loss - ncp	96.743316	1.079804	274.604853
Same time - t_ncp	42.740744	2.444126	X
Same time - ncp	62.835432	1.662498	117.298053

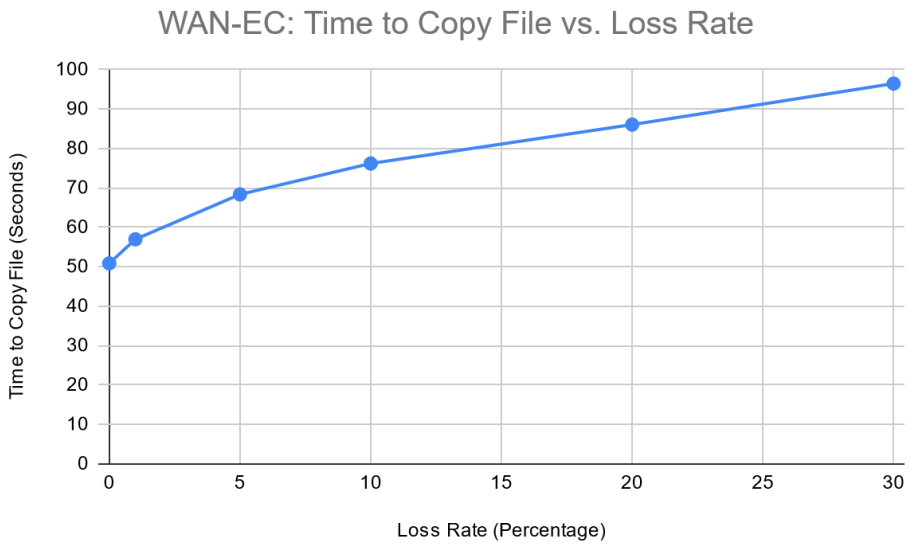
Final Results - Average

	Transfer Time (S)	Average Transfer Rate (MB/S)	Total Data Sent (MB)
t_ncp	8.8395072	11.8237134	X
0% loss - ncp	50.8759518	2.0533646	113.866373
1% loss - ncp	56.9832878	1.8335762	121.614533
5% loss - ncp	68.3713182	1.5279154	136.488693
10% loss - ncp	76.2056244	1.3708238	157.110133
20% loss - ncp	86.0700082	1.2137164	217.1201036
30% loss - ncp	96.4764676	1.0827952	275.4462366
Same time - t_ncp	39.96637675	4.2987108	X
Same time - ncp	64.107789	1.6299638	118.313053

Graphs


LAN: Time to Copy File vs. Loss Rate
Time to Copy File (Seconds)
120
110
100
90
80
70
60
50
40
30
20
10
0
0
5
10
15
20
25
30
Loss Rate (Percentage)


Artificial WAN: Time to Copy File vs. Loss Rate
Time to Copy File (Seconds)
240
220
200
180
160
140
120
100
80
60
40
20
0
0
5
10
15
20
25
30
Loss Rate (Percentage)



Discussion

Parameter Tuning

The parameters that we are able to tune are: window size, window timeout, the number of packets to resend on timeout, and a receiver message timeout. In order to tune the aforementioned variables, a slightly smaller file was used; since some of the larger loss rates take quite a while longer to process, using a smaller file helped make tuning more efficient. Additionally, using a smaller file meant that the new parameter values could be more quickly tested multiple times to find an average to get a better estimate of performance; similar to the final results, the parameter values were tested a total of 5 separate times at different loss levels.

LAN Tuning

The first parameter that was tuned was the window size. Starting with the optimal window size found using the calculation below:

throughput = window_size*packet_size/RTT

(100 Megabits/s * 1000000 bit/Megabit) = window_size * (1400 Bytes * 8 bits/byte)/(0.00055 s)

window_size = (100 Megabits/s * 1000000 bit/Megabit) * (0.00055 s)/(1400 Bytes * 8 bits/byte) = 4.9107

With 0% loss, a window size of 5 finished quickly and had an average rate around 9 MB/S. However, with a 30% loss rate, the file transfer took several magnitudes longer to finish and we were lucky if the transfer rate was even 0.2 MB/S. Using a 30% loss rate, the window size was then increased until the time and send rate approached a much more tolerable level. Once a good window size was found for 30% loss, it was tested on a 0% loss rate, but over-tuning to the high loss meant that loss was occurring in this scenario as more packets were being sent to the receiver than they could actually process. Thus, tuning was done to find a window size that struck a balance between these two which ended up being a window size of 45.

The timeouts were tuned by starting with a relatively high timeout (1 second for the receiver timeout and 2 seconds for the window timeout) and gradually decreasing them until no more improvements were seen. The receiver timeout was tuned first, and the window timeout was tuned after. This ended up being a timeout of 0.125 second for the receiver timeout and 1 second for the window timeout.

Finally, the number of packets to resend on timeout did not really affect performance all that much; it slightly increased the total amount of data transferred. This parameter needed to be at least one, and there was an occasional time benefit for using 2. Thus, this parameter ended up being set to 2 for LAN.

WAN Tuning

throughput = window_size*packet_size/RTT
(100 Megabits/sec * 1000000 bit/Megabit = window_size * (1400 bytes * 8 bits/byte)/(0.04 s)
window_size = 357 packets

I attempted to run the transfer with the ideal window size of 357 packets. However, the receiver did not operate fast enough to process the incoming packets, so the sender had to resend packets excessively. This led to congestion. For example, in one run, to send a 100 MB file, the sender ended up sending 1330 MB of data! I used a binary search style algorithm, cutting the window in half and retesting, to land on the optimal window size. For each window size, I ran 3 tests and recorded the time to transfer and the total data sent, including retransmissions. After this process, I found the optimal window size was 45 packets. I tested the values 357, 100, 50, 25, 40, and finally 45.

In order to calculate the receiver message timeout, I subtracted the round trip time, 40 ms, from the timeout that was calculated from the LAN tuning, resulting in a message timeout of 85 ms.

WAN-EC Tuning
RTT = 26.3 ms = 0.0263 sec
Throughput = window_size*packet_size/RTT
(100 Megabits/sec * 1000000 bit/Megabit = window_size * (1400 bytes * 8 bits/byte)/(0.0263 s)
window_size = 234.8 packets

With all loss rates, an ideal window size of 245 did not perform well. Data was being sent far too quickly and this led to very high loss; for 0% loss, 10x the original file size ended up being sent. From here, the window size of 45 that performed well for both LAN and artificial WAN was used. It performed well for 0% to 5% loss rate, but the transfer took a little too long for the higher loss rates. Thus, 50 and 55 were experimented with; neither led to a high degree in loss due to sending too fast for the lower loss rates. However, 55 performed better with the high

loss scenarios which was why it was selected as the final window size. 60 and a couple of other higher window sizes were used; the extra data being sent for the 0% loss began to exceed 20% of the original file, and this did not seem acceptable.

The timeouts were tuned by starting with the timeouts that were found to perform well for LAN. Similar to LAN, they were then gradually decreased until no more improvements were seen, and the tuning order was also the same. This ended up being a timeout of 0.05 second for the receiver timeout and 0.5 second for the window timeout.

Finally, the number of packets to resend on timeout did not really affect performance here much either. Thus, this parameter ended up being set to 2 here as well.

Results Discussion

One issue we ran into which prevented us from reaching peak performance was that the server (receiver) was too slow to handle all of the data sent by the client (sender) when window sizes were large. We believe the internal buffer of the socket in the server could not hold all of these packets, leading to packets being dropped, even in the 0% loss case. One solution could be to increase the buffer size, or to increase the speed of the server by optimizing the algorithm or by multithreading. When it came to tuning the window size, we decided to improve transfer time in exchange for increasing the amount of excess data sent that could have congested the network for others. With this all in mind, we saw 1.1-2.3x the file size being sent in LAN, 1-2.1x the file size being sent in emulated WAN, 1.1-2.8x the file size being sent in WAN.

The other issue we saw was that the server was not sending messages to the client often enough, leading to downtime on the client, as it was not able to move its window up until the acknowledgement was received. We attempted to solve this problem by sending ARUs when the receiver received half of the packets in the window without sending a message. But we could improve this by sending cumulative ACKs even more frequently. This does have the drawback of increasing the amount of congestion in the network, so there is a tradeoff.

In all of our data, we saw a trend of transfer time increasing at a square root function of loss rate. This is consistent with the predictions of the Mathis equation, which relates TCP throughput to packet loss via a square root function. We had one outlier in our data. At a 30% loss rate in the LAN environment, we had an unexplained spike in transfer time. However, all other values seem to follow this broad trend, so we are not sure why this was the case in this situation.

We used a 100 Mbps link, so in theory, we should have been able to achieve a 12.5 MB/s transfer rate. However, in the best case of 0% loss, we were only able to achieve an average of 6.95 MB/s in the LAN, 1.45 MB/s in the simulated WAN, and 2.05 in the actual WAN. This corresponds to a link utilization rate of 55.6%, 11.6%, and 16.4% respectively. In contrast, TCP was able to reach transfer rates of 11.45 MB/s, 10.39 MB/s, and 11.82 MB/s.