Apurva Gandhi, Margaret Haley, Michael Dahlquist
CSCI 356: Computer Networking
December 7th, 2020
Project 3

Zoom Project: Written Report

Preliminary Data Analysis

Running the code as it was given, before making any changes. This is the Stop-and-Wait protocol.

Test 1:

- Running server on 35.221.36.111 (us-east4-c) and client on 34.77.18.50 (europe-west1-b)
- Sent 68 packets, 68 unique, 0 duplicate, 0 misordered, 0 missing
- Elapsed time 5.663 s, total received 95.62 KB, throughput 16.89 KBps
- Estimated time to send 180,000 packets:
    - 5.663 seconds / 68 pkts = 0.0833 seconds/pkt
    - .0833 seconds/pkt * 180,000 pkts = 14,990.294 seconds
- Throughput: 16.89 KBps
- RTT (running ping from client to server with 1440 bytes)
    - 83.315 ms = 0.0833 seconds

Test 2:

- Running server on 34.125.47.95(us-west4-a) and client on 35.236.78.138 (us-west2-a) (hosts are physically close together)
- 22 packets, 22 unique, 0 duplicate, 0 misordered, 0 missing
- Elapsed time 1.533 s, total received 30.94 KB, throughput 20.18 KBps
- Time to sent 180,000 packets
    - 1.533 seconds/22 packets = 0.06968 seconds/packet
    - 0.06968 seconds/packet * 180,000 packets = 12,542.4 seconds
- Throughput: 20.18 KBps
- RTT: 63.502 ms = 0.063502 seconds

Test 3:

- Running both server and client on 34.84.237.14 (asia-northeast1-b) (same host)
    - First time running 25 packet was received
    - Lost packets effects
- 25 packets, 25 unique, 0 duplicate, 0 misordered, 0 missing
- Elapsed time 0.317 s, total received 35.16 KB, throughput 111.05 KBps
- Time to sent 180,000 packets

    - 0.317 seconds / 25 packets = 0.01268 seconds/packet
    - 0.01268 seconda/packet * 180,000 packets = 2,282.4 seconds
- Throughput - 111.05 KBps
- RTT (running ping from client to server with 1440 bytes)
    - 0.473 ms

Analysis:

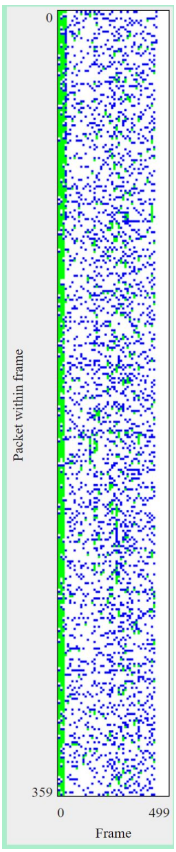
In our tests, there were no duplicate packets and no misordered packets, which makes sense because we are running a Stop-and-Wait protocol. The only way we would get duplicate packets is if some are duplicated within the network, but we did not see this in our tests. The program always halts after some number of packets and does not successfully send all 180,000, due to the fact that some packets just get randomly lost in the network. When a packet is lost, the program halts and never continues because no re-transmission has been implemented yet. Throughput was about 17 KBps when run on two random hosts, increased to 20 when run on closer hosts, and increased all the way to 111 when run on the same host; these results make sense because the data did not have to travel as far. We measured RTT by running ping from client to server with a packet size of 1440 bytes, which was reasonably consistent with our observed protocol behavior for each test. Lost packets severely impede the efficacy of the protocol because packets always get randomly lost in the network, and this protocol does not have any measures to handle that.

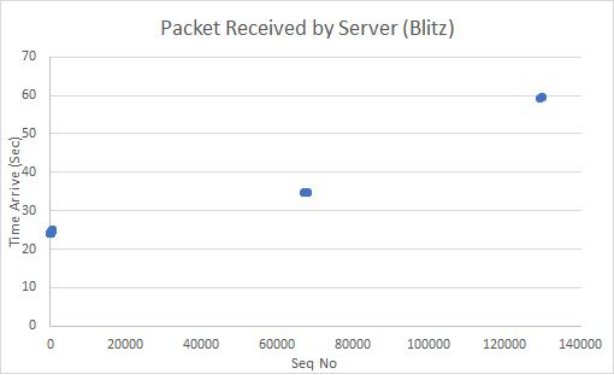
Conclusion: very fair but slow, halts frequently so ineffective

Blitz Protocol

Test:

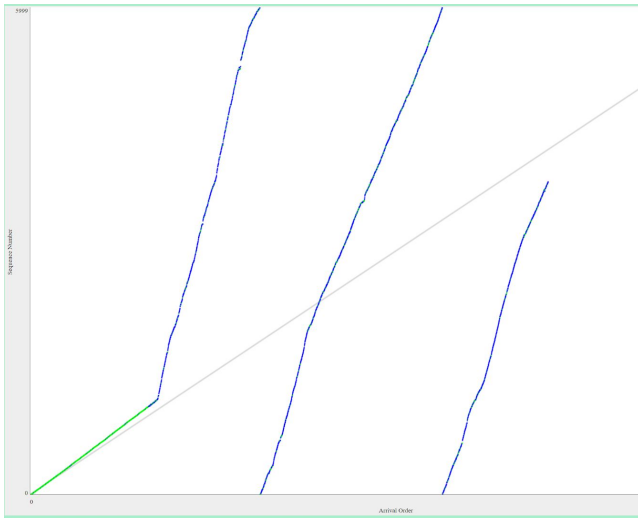
- Running client on 34.125.47.95(us-west4-a) and server on 35.221.36.111 (us-east4-c)
- Sent 8792 packets, 8792 unique, 0 duplicate, 6881 misordered, 68314 missing
    - Client sent 8792 + 68314 packets = 77106
- Elapsed time 41.525 s, total received 12.07 MB, throughput 297.74 KBps
- Time to send 180,000 packets
    - 41.525 seconds / 77106 packets = 0.000539 seconds/packet
    - 0.00053 seconds/packet * 180,000 packets = 96.94 seconds





Analysis:

The estimated time to send 180,000 packets was significantly faster with the blitz protocol than the original stop-and-wait. While removing the lines of code that waited for ACKs caused us to send much quicker, it also caused some other issues. We had packets that came in out of order, as well as many missing packets. In the graph on the previous page, the blue packets indicate packets that came in out of order. The graph above this paragraph, titled "Packets Received by Server," highlights the inconsistency in sequence numbers delivered, as there should ideally be no spaces in between the blue dots in this graph; the spaces indicate those packets that have not arrived yet and may never arrive. In the graph to the right, blue packets harshly deviating from the diagonal grey line indicates packets arriving out of order. In the beginning, you can see that many packets arrived properly (green), but that becomes increasingly rare as more and more packets arrive out of order and packets go missing. It's clear from these three graphs that while this protocol initially delivers packets quickly and (mostly) in order, the speed of sending and lack of re-ordering and retransmission quickly cause problems. The receiver gets very few packets in order after the first few frames, making the protocol very unreliable.



Conclusion: fast but extremely out of order, unreliable, and not very fair because it sends packets as fast as it possibly can

Our Protocol

Our protocol attempts to cope with packet loss, duplication, and misordering, as well as implement a sliding window and avoid excessive retransmission. In order to cope with packet loss and avoid excessive retransmission, we implemented a timeout. If the timeout expires before we have received an ACK for the oldest outstanding packet, we retransmit that packet. We performed a calculation to determine an appropriate timeout that we believe will work for a variety of different servers at varying geographic locations. Initially, we set the timeout to be 0.5 seconds, long enough for a round trip even if the servers were far apart. For every 300 + N (with N being our window size) packets sent by the client, we treated one packet as a probe. We chose the number 300 because we wanted to respond to changing network conditions without overdoing it by constantly sending probe packets. We added N to that 300 in order to ensure we never have more than one probe packet outstanding at a time. By recording the time it took for the probe to get sent and the ACK for it to return, we got an estimate for the RTT. Then, we used Karn's algorithm to do a weighted average of the RTT. Using this implementation, we avoided excessive retransmission by giving the packets enough time to make a round trip, at the same time allowing for the timeout to slowly decrease if the packets do not need that much time. We believe our timeout will work in different scenarios by slowly adjusting to the current conditions (geographic distances, network conditions) based on the data from the probe packets.

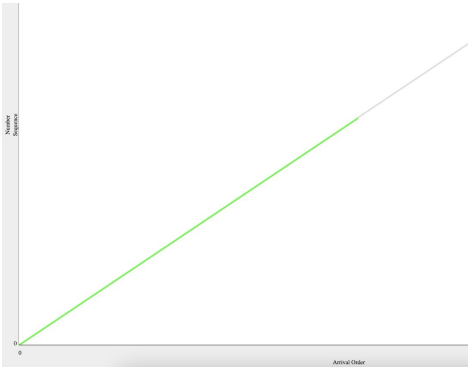
For duplication and misordering, we made use of sequence numbers on the server side. We kept track of the sequence number that we are currently expecting and must pass along next. If the packet that we were waiting for arrives, we call a data sink. If the packet is associated with a sequence number other than what we were waiting for, then we place it in a dictionary. This dictionary contains a key for the sequence number of the packet, and a value for the packet's payload. Once we receive the packet we were waiting for, we check the dictionary to see if the next sequence number to be passed along was previously received (aka the packets came out of order). We do that repeatedly until the sequence number we must pass along next is not in our dictionary. In this case, we are still waiting to receive that packet from the client.

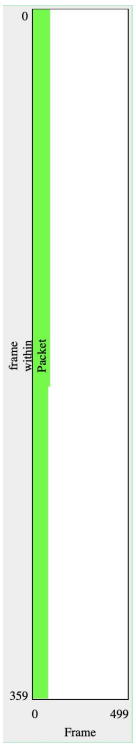
Additionally, we implemented a sliding window to allow our client to have multiple packets outstanding at one time, increasing the speed of the protocol. We have a flexible sliding window of size N, and our client initially sends N packets to the server. Next, we receive an ACK and mark in an array that the ACK number was received. Then we have a loop that checks if the oldest outstanding ACK we were waiting for has been received, according to the array. If it has, we slide the window up so that the bottom of the window equals the new oldest outstanding packet and the top of the window increases by one. We repeat this process until the sequence number associated with the bottom of the window has not yet received an ACK.

Testing Our Protocol

Test 1:

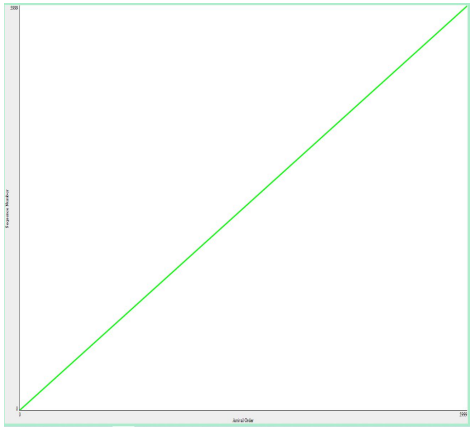
- Testing on server 35.228.5.242 (europe-north1-a) and client 35.244.108.42 (australia-southeast1-b)
- Elapsed time 234.217 s, total received 4.23 MB, throughput 18.47 KBps
- 3077 packets, 3077 unique, 0 duplicate, 0 misordered, 0 missing
- Time to send 180,000 packets
    - 234.217 seconds / 3077 packets = 0.07612 seconds/packet
    - 0.07612 seconds/packet * 180,000 packets = 13,701.6 seconds
- RTT (running ping from client to server with 1440 bytes): 314.855 ms

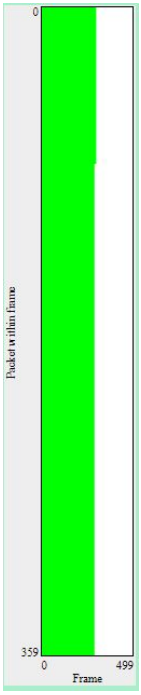




Test 2:

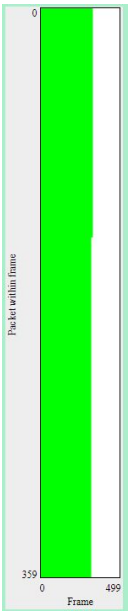
- Client and server on same host: 104.155.201.100 (asia-east1-c)
    - Elapsed time 26.561 s, total received 20.54 MB, throughput 791.72 KBps
    - 14954 packets, 14954 unique, 0 duplicate, 0 misordered, 0 missing
    - Time to send 180,000 packets
        - 26.561 seconds / 14954 packets = 0.00178 seconds/packet
        - 0.00178 seconds/packet * 180,000 packets = 319.71 seconds
- RTT (running ping from client to server with 1440 bytes): 0.489 ms

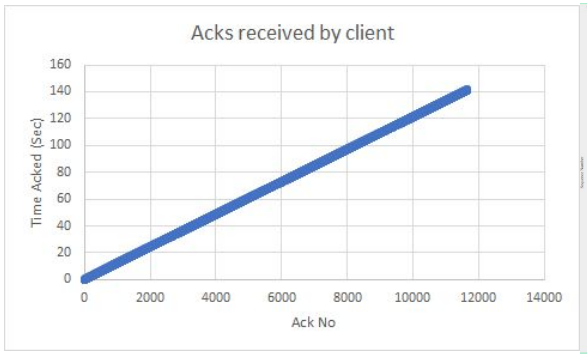


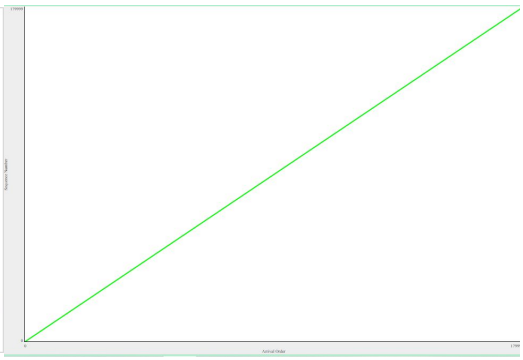


Test 3

- Testing on server 34.87.29.60 (asia-southeast1-b) and client 34.93.46.53 (asia-south1-b)
- Elapsed time 141.366 s, total received 16.02 MB, throughput 116.04 KBps
- 11665 packets, 11665 unique, 0 duplicate, 0 misordered, 0 missing
- Time to send 180,000 packets
    - 141.366 seconds / 11665 packets = 0.0121 seconds/packet
    - 0.0121 seconds/packet * 180,000 packets = 2141.39 seconds
- RTT (running ping from client to server with 1440 bytes): 60.920 ms





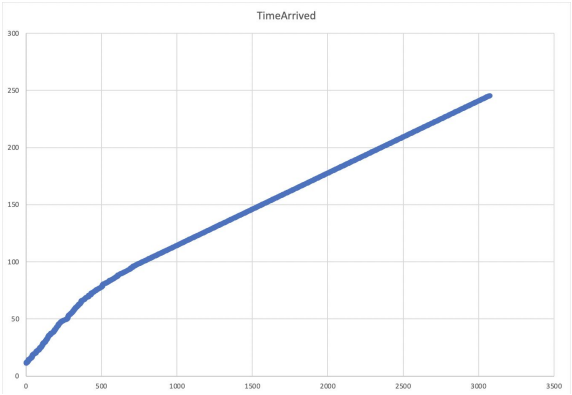


The tests above were performed in order to measure the speed, efficiency, and reliability of our protocol. We tried running the client and server on two different hosts, on hosts close together, and on the same host. From the results above, we determined that our protocol is significantly better than the previous two we had analyzed for a multitude of reasons. Firstly, the graphs taken from the website that plot arrival order on the x-axis and sequence number on the y-axis demonstrate that the packets are being received perfectly in order. This outcome represents the ideal situation, matching the proper ordering from the original stop-and-wait protocol. This graph is significantly better than the blitz protocol graph, which clearly demonstrates deviation from the gray diagonal line and misordering of packets. The graphs that plot packets received clearly shows all green packets, with no duplicate red packets or misordered blue ones. These plots demonstrate improvement from the blitz protocol, which demonstrated many misordered packets. From the graphs alone, the advantage of our protocol over the original stop-and-wait may not be obvious. However, the key difference is in the efficiency and completeness of the sending. Our protocol is significantly faster than the projected time of the original stop-and-wait protocol for all 180,000 packets. Furthermore, our protocol is actually capable of sending all those packets, while the original stop-and-wait would always halt due to random, unavoidable packet loss. Test 2, for example, is estimated to take 2141.39 seconds to send all the packets, which is faster than the original protocol estimations. Test 1 is estimated to take longer, but we

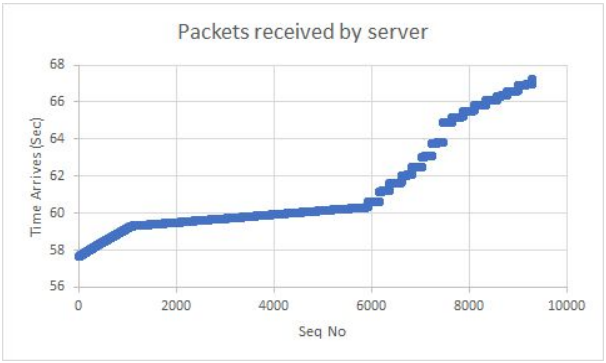
determined this is not due to our protocol by noticing the elevated RTT and running tests on different servers. Although our protocol is slower than blitz, the blitz protocol failed to actually deliver all the packets in order. Our protocol accomplishes this.

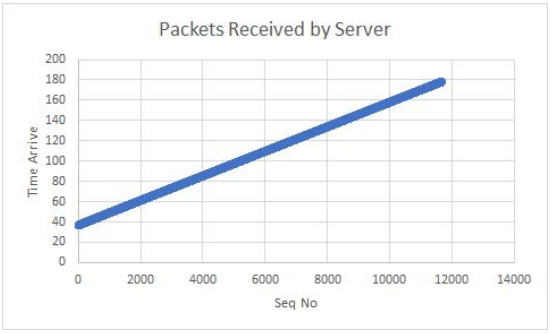
We believe our protocol responds well to packet loss, reordering, and delays. We know this because there are no gaps in the sequence number graphs. Therefore, all the packets were received. We also see that they were received in order, from the solid green plots of packets received shown in the tests above. One interesting thing to note is that the graphs of sequence numbers received by the server (which we included below) are all different shapes. We believe this may have something to do with the network congestion at the time of sending, or the filtration/misordering elements added to the different servers that we tested on. We were somewhat surprised that test 3 was linear rather than test 2, as test 2 was performed with the server and client running on the same host. We originally anticipated that this graph would be the most linear because the packets do not have to travel a far distance, so the packets would not encounter queuing delays and network congestion as much. We wondered if perhaps this server happened to have more of a packet filter, to cause a less-constant stream of packets received.

Test 1:



Test 2:



Test 3: 

Next, we ran tests to explore how our protocol would behave under different conditions. Specifically, we varied different parameters to see how that would affect the efficacy of the protocol. When testing different parameters, we first tried different values of the parameters using two servers that were fairly close together. Comparing those results, we determined the best of the parameter values. Next, we compared that "best" parameter value on different servers to determine if they depend on how far away those servers are from each other.

Varying the size of the sliding window

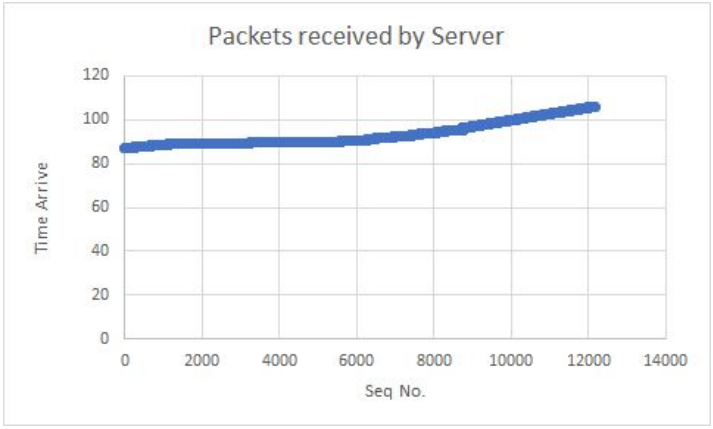
Test 1: changing window size, keeping hosts constant

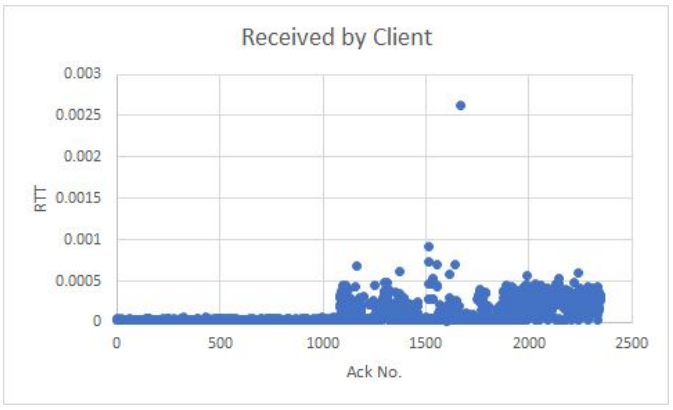
- Testing on server 34.87.29.60 (asia-southeast1-b) and client 34.93.46.53 (asia-south1-b)
    - Window size: 5
        - Elapsed time 141.366 s, total received 16.02 MB, throughput 116.04 KBps
        - 11665 packets, 11665 unique, 0 duplicate, 0 misordered, 0 missing
        - Time to send 180,000 packets
            - 141.366 seconds / 11665 packets = 0.0121 seconds/packet
            - 0.0121 seconds/packet * 180,000 packets = 2141.39 seconds
    - Window size: 50
        - Elapsed time 34.069 s, total received 14.42 MB, throughput 433.27 KBps
        - 10497 packets, 10497 unique, 0 duplicate, 0 misordered, 0 missing
        - Time to send 180,000 packets
            - 34.069 seconds / 10497 packets = 0.003246 seconds/packet
            - 0.003246 seconds/packet * 180,000 packets = 584.20691 seconds
    - Window size: 500
        - Elapsed time 93.856 s, total received 1.29 MB, throughput 14.07 KBps
        - 939 packets, 939 unique, 0 duplicate, 0 misordered, 0 missing
        - Time to send 180,000 packets
            - 93.856 seconds / 939 packets = 0.00999 seconds/packet

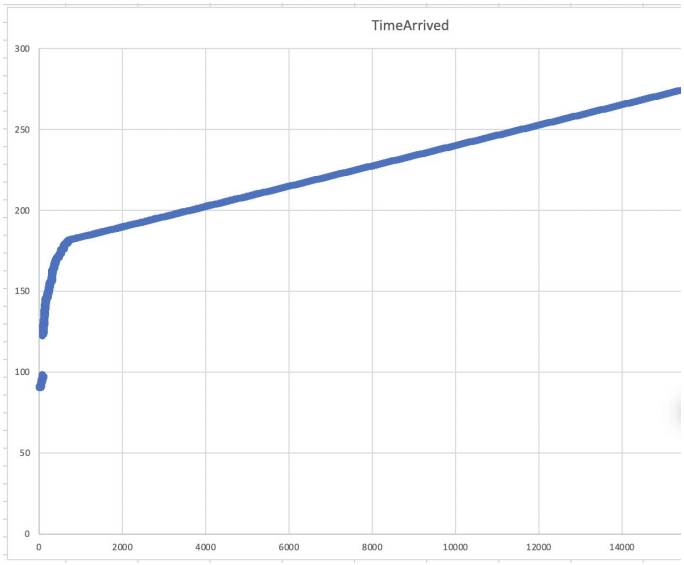
- 0.00999 seconds/packet * 180,000 packets = 17991.565495 seconds

Test 2: changing hosts, keeping window size constant at 50

- Client and server on same host: 104.155.201.100 (asia-east1-c)
  - Elapsed time 35.223 s, total received 23.25 MB, throughput 676.04 KBps
  - 16933 packets, 16933 unique, 0 duplicate, 0 misordered, 0 missing
  - Time to send 180,000 packets
    - 35.223 seconds / 16933 packets * 180,000packets = 374.53 seconds
  - RTT (running ping from client to server with 1440 bytes): 0.489 ms





- Testing on server 35.236.78.138 (europe-north1-a) and client 34.93.46.53 (australia-southeast1-b)
    - Elapsed time 188.720 s, total received 21.77 MB, throughput 118.11 KBps
    - 15850 packets, 15850 unique, 0 duplicate, 0 misordered, 0 missing
    - Time to send 180,000 packets
        - 188.720 seconds / 15850 packets * 180,000 packets = 2143.19 seconds
    - RTT (running ping from client to server with 1440 bytes): 314.855 ms
    - 

Seqno vs time

After testing sizes 5, 50, and 500 for our sliding window in test 1, we determined that the window size of 50 was the best of the three. For all three, the packets still arrived in order due to the design of our protocol, but window size 50 was the fastest according to our time estimate for sending 180,000 packets. In test 1 with a window size of 500, we noticed something really interesting with the sequence numbers, and we included a graph above to illustrate this. We concluded that this window size is too big here because although our protocol still corrected misordering of packets, vastly different sequence numbers come in right next to each other. We believe this is a result of too large a window, and we also noted that it significantly slows down the speed of our sending. From these tests, we conclude that having a window size too small and too large are both inefficient.

In test 2 with a window size of 50 held constant, packets delivered fairly consistently throughout. With client and server on the same host, the graph of sequence number vs time arrived shows that the rate of packet arrival increases slightly at the end of the test. We believe this is due to our timeout estimate getting more accurate over time as more probes come back with quick RTTs. Within that same test, the graph of RTT vs Ackno demonstrated the RTT for some packets increases after 1200 packets, while 0-1200 has an extremely constant RTT. We hypothesize that this might have been due to packet filters beginning to slow some packets down or perhaps there was just some congestion within the server. When

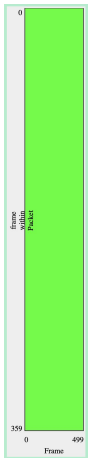
we tested two servers that are far apart, the RTT was significantly larger but because of the sliding window, the packets are delivered relatively fast. The graph shows logarithmic growth until around 1000 packets and switches to linear growth. Perhaps the logarithmic growth was due to the timeout initially changing drastically as RTT probes adjusted it, then approached a more constant rate as the timeout approached an accurate RTT. There is also an interesting burst of packets at the beginning, then a gap. We predict that the small burst in the very beginning was the initial N packets that get sent before our code begins to send packets in a loop by moving the sliding window. Ultimately, with window size held constant, our code sends the fastest when client and server are running on the same host. This outcome makes a lot of sense to us and was not surprising.

Varying the length of the timeout

Here we test two different timeout times, 0 and 1 seconds, on different servers that are geographically close.

Client: us-west2-a: 35.236.78.138, Server: us-west4-a: 34.125.47.95

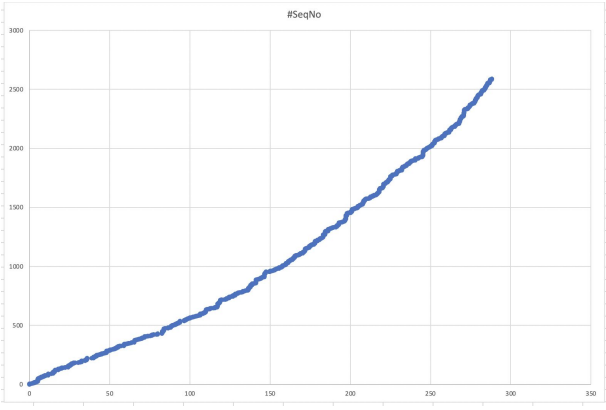
Test 1: Window Size = 5, **Timeout = 0 seconds**



- Elapsed time 112.247 s, total received 27.66 MB, throughput 252.31 KBps
- 20139 packets, 20139 unique, 0 duplicate, 0 misordered, 0 missing
- Time to send 180,000 packets:
    - 112.247 seconds / 20139 packets = 0.005575 seconds/packet
    - 0.005575 seconds/packet* 180,000 packets = 1003.5 seconds
- RTT (running ping from client to server with 1440 bytes): 8.448 ms

Test 2: Window Size = 5, **Timeout = 1 second**

- Elapsed time 288.159 s, total received 3.56 MB, throughput 12.64 KBps
- 2590 packets, 2590 unique, 0 duplicate, 0 misordered, 0 missing
- Time to send 180,000 packets:
    - 288.159 seconds / 2590 packets = 0.1113 seconds/packet
    - 0.1113 seconds/packet* 180,000 packets = 20,034 seconds
- RTT (running ping from client to server with 1440 bytes): 8.448 ms



When the timeout is initialized to 1, the packets initially send at a much lower rate. However, we can see that as time progresses, that rate appears to be growing exponentially. This is because the timeout time will weigh with the actual RTT, which is less than 0, lowering the timeout over time. This will not increase exponentially forever as it will approach the average RTT. Ultimately, the timeout of 1 is still way slower than 0, because the timeout is so large initially that it simply could not catch up with such a significantly smaller timeout (and where timeouts are concerned, a difference of 1 second is actually huge).

After testing timeout time of 0 and timeout of 1 second, we determined that a timeout somewhere in between is probably best. The timeout of 1 is much slower than the RTT time on these servers, causing any lost packet or lost ACK to delay the client way longer than it should. On the other hand, when the initial timeout is 0, we actually retransmit packets more than necessary. While this makes the program run much quicker, it's also sending many packets when the ACK we're waiting for has not arrived yet. The protocol handles duplicate packets, so this excessive retransmission may not seem like an issue. However, excessive retransmission may not be considered fair, because we are taking up much more of the network in order to send more quickly overall. In order to achieve a balance that sends efficiently without bombarding the network with duplicate packets, we need to use a timeout in between 0 and 1.

All of our previous tests were run with a timeout of 0.5. We are including them here again to illustrate the varying results we get when running the protocol on various-distanced data centers when timeout is 0.5.

Test 1:

- Testing on server 35.228.5.242 (europe-north1-a) and client 35.244.108.42 (australia-southeast1-b)
- Elapsed time 234.217 s, total received 4.23 MB, throughput 18.47 KBps
- 3077 packets, 3077 unique, 0 duplicate, 0 misordered, 0 missing
- Time to send 180,000 packets
    - 234.217 seconds / 3077 packets = 0.07612 seconds/packet
    - 0.07612 seconds/packet * 180,000 packets = 13,701.6 seconds

Test 2:

- Client and server on same host: 104.155.201.100 (asia-east1-c)
    - Elapsed time 26.561 s, total received 20.54 MB, throughput 791.72 KBps
    - 14954 packets, 14954 unique, 0 duplicate, 0 misordered, 0 missing
    - Time to send 180,000 packets
        - 26.561 seconds / 14954 packets = 0.00178 seconds/packet
        - 0.00178 seconds/packet * 180,000 packets = 319.71 seconds

Test 3

- Testing on server 34.87.29.60 (asia-southeast1-b) and client 34.93.46.53 (asia-south1-b)
- Elapsed time 141.366 s, total received 16.02 MB, throughput 116.04 KBps
- 11665 packets, 11665 unique, 0 duplicate, 0 misordered, 0 missing
- Time to send 180,000 packets
    - 141.366 seconds / 11665 packets = 0.0121 seconds/packet
    - 0.0121 seconds/packet * 180,000 packets = 2141.39 seconds

We can see from these results that although sending data with a timeout of 0.5 takes longer than a timeout of 0, it is still not as time consuming as a timeout of 1. Unlike the timeout of 0, however, this timeout does not excessively retransmit packets. Therefore, we believe this timeout is better in practice than either 0 or 1.

The features of the protocol that make the most difference in practice are the sliding window and the reordering of packets. Without the sliding window, waiting for each individual ACK slows down the protocol significantly. It would not be feasible in practice to use a Stop-and-Wait protocol because its inefficiency outweighs its fairness and reliability. Blitz protocol, on the other hand, is the opposite: its lack of fairness, reliability, and accurate packet delivery outweighs the benefit of its speed. Our protocol has N packets in flight at all times, and accounts for reordering and packet loss that unavoidably occurs in the network. We believe these are the most important features because they allow the protocol to work effectively in practice. One thing that did not make as much of a critical difference in our protocol is the timeout. We still believe timeout was very important, but slightly less critical than the others because even with poorly-chosen timeouts, our protocol still achieved the basic goals of sending all the packets and getting them in order. If you chose too low of a timeout and packets were re-sent too frequently, all the necessary packets still got sent correctly in the end. Too high of a timeout definitely slowed things down, but again, everything still got sent.

Finally, enjoy this image of us when we finished our final networks project!! We may have beat the record for the longest zoom ever (top right hand corner of the picture, 12.33 hours).