Project 3 Report

Alexa Furnari and Jacob McDonald

Analysis of Stop and Wait Protocol**:**

1. How fast is it? That is, how long does it take to send all 180,000 packets? If you can't be bothered to wait until the end, calculate about how long it *would* take, if you waited long enough and/or got lucky enough to have all packets arrive in the proper order. What throughput does the protocol achieve? Here and elsewhere, strive to use appropriate units, like Kbps, Mbps, KBps, MBps.

    a. 160.9425716 seconds to send 1000 packets
    b. 28,969.662888 seconds for 180k packets
    c. 482.8277148 minutes for 180k packets
    d. 8.0471285 8 hours for 180k packets

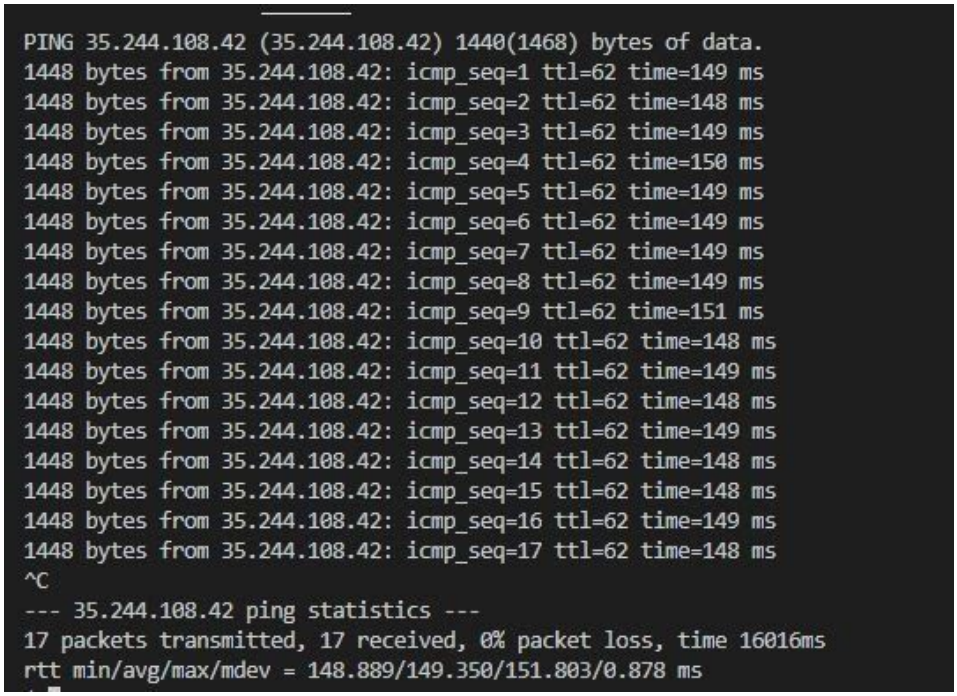
More math

- 1000 packets at 1440 bytes each == 160.942571640014 seconds
- 1,440,000 bytes in 160.94...seconds
- Comes out to about 8,947.29… bytes per second (8.947Kbps)

Packet Number	Time Elapsed since Send of First Packet (seconds)
1000	160.9425716
2000	321.0130517
3000	479.6497021
4000	632.0773678
5000	789.9358652
6000	944.2588165
7000	1096.670143
8000	1249.709512

Chart shows that it takes roughly 160 seconds to send 1000 packets, therefore it will take about 8 hours for 180,000 packets to send.

2. From the client, ping the server to estimate the actual RTT. Is this reasonably consistent with the observed protocol behavior, given the known packet size (1440 bytes payload)?



   On average, the RTT of a 1448 byte packet is around 0.149 seconds. The average time for a packet to get sent and for the sender to receive an ACK is around 0.160 seconds. For example, packet 1000 was sent at time 160.94257164s and the ACK was received at time 161.10176897s (the difference is 0.159198s). For packet 2000 the difference between the send time and the ACK is about 0.158s. The actual RTT from the ping (0.149s) is slightly faster than the RTT we observed (≈ 0.160s), however we don't think the difference between the RTTs is unreasonable.

3. Do you observe any lost or duplicate packets? If so, do they affect the reliability (or efficiency, or speed) of the protocol?

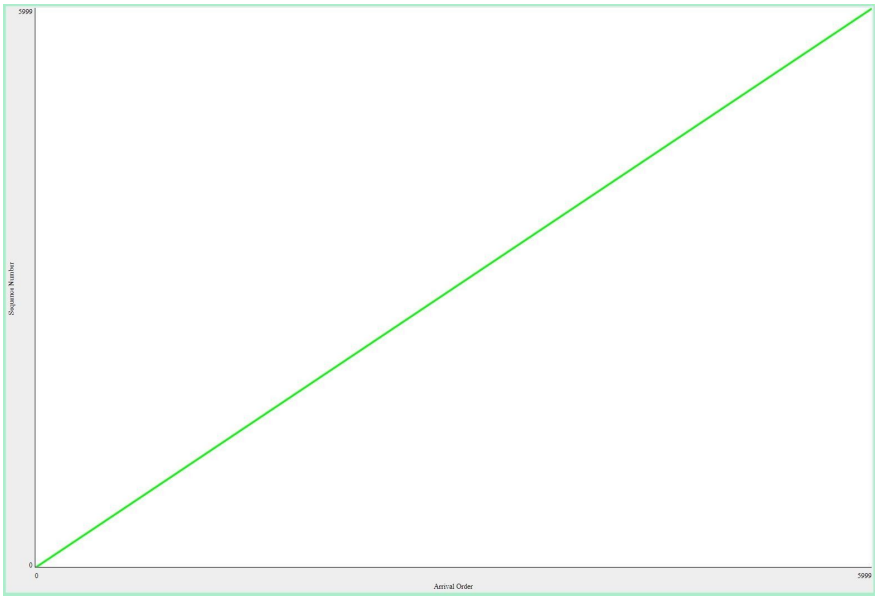
   Since this is a stop-and-wait protocol, if packets are lost, then the program stops entirely; in some of our trials we only managed to get 6 packets out and in another case we only got 16 out before killing the program. Fortunately, with Alexa's machine as the server (atfurn21) and Gianna and Matt's machine as the client, we achieved relatively good results to analyze and got 8260 packets out the door.

4. Try a different pair of data centers and repeat your experiments. Or try two hosts within the same datacenter. Or try running the client and server on the same host. Do your results change substantially? Why or why not? You may need to investigate or speculate as to causes.

Alexa's Machine (australia-southeast1-b) as Server and Gianna and Matt's Machine as Client (asia-south1-b)

8261 pkts, expecting seqno 8261, 11.38 MB in 1353.235 s = 8.61 KBps, approx 0.0 frames per second

```
Connecting to ws://35.244.108.42:8100/ ...
Connected! Waiting for stream to start ...
Stream is starting!
Received packet with sequence number 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49... 99... 199... 299... 399... 499... 599... 699... 799... 899... 999... 1999...
2999... 3999... 4999... 5999... 6999... 7999Lost Connection!
  Elapsed time: 1353.23 s
  Total Packets: 8261
  Unique Packets: 8261
  Missing packets: 0
  Duplicate packets: 0
  Out-of-order packets: 0
  Data: 11.38 MB
  Throughput: 8.61 KBps
  Complete images: 22
  Partial images: 1
```



Alexa’s machine as client (australia-southeast1-b), waffle-town (europe-west1-b) as server

Trial 1

2 pkts, expecting seqno 2, 2.82 KB in 79.543 s = 36.30740605710119 Bps, approx 0.0 frames per second

```
Connecting to ws://34.77.18.50:8100/ ...
Connected! Waiting for stream to start ...
Stream is starting!
Received packet with sequence number 0 1
```

Trial 1 Webpage Picture

Trial 2

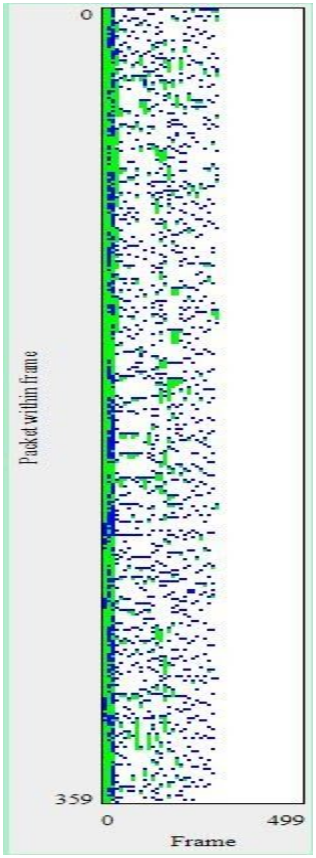
```
Connecting to ws://34.77.18.50:8100/ ...
Connected! Waiting for stream to start ...
Stream is starting!
Received packet with sequence number 0 1 2 3 4 5 6
```

Analysis

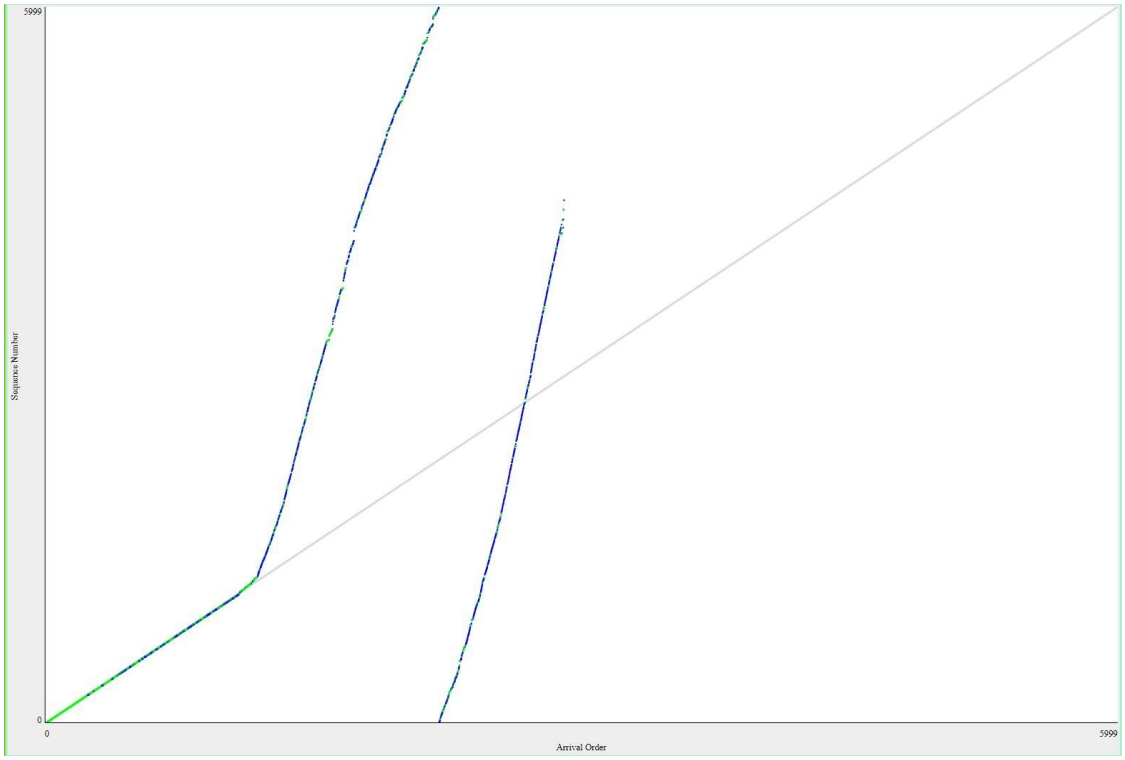
Between the two trials of the Waffletown Server and the Atfurn client, we see that the Waffletown server has been severely damaged by Professor Walsh's filters as in both trials it did not get close to 10 ACKs back to the client (of the 180,000 that it was supposed to end up sending). Client_saw.py utilizes a stop and wait protocol. Meaning packets will never be out of order and the server will never send the next packet without receiving an ACK for the previous packet. Therefore, it gets stuck waiting for the 3rd packet's ACK in trial one and the 7th packet's ACK in trial 2 and never moves on. Comparing these two trials with that of the Atfurn server and GiannaAndMatt client, the latter performed significantly better.

Blitz Protocol:

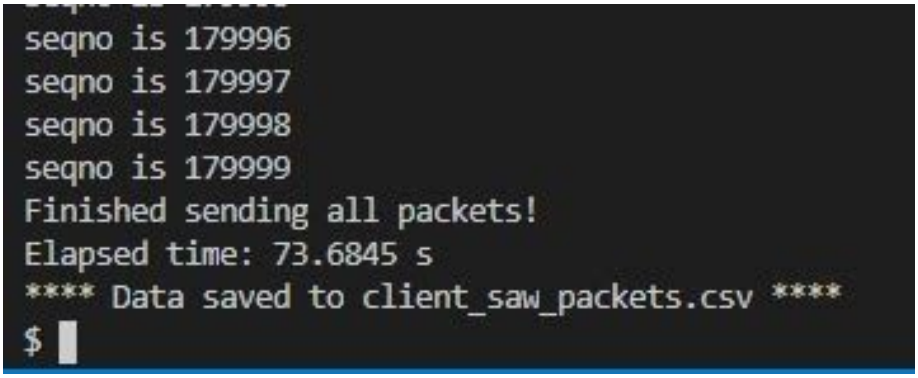
Graph 1



Graph 2



Graph 3



The almost vertical lines in Graph 2 show that the Blitz protocol is quite literally sending as fast as it can (since it is not waiting for ACKs of any kind). The Blitz Protocol showed marvellous results; Graph 3 shows that all 180k packets were sent in less than a minute and a half. At first the packets arrived mostly in order, as seen by the steady green line in Graph 2. However, the scattered blue dots in Graph 1 show the protocol's inability to handle out of order packets.

Documentation of better_client.py protocol with retransmissions, timeouts, and sliding window

Description: Our code initially sends 10 packets without acks and for every ACK that comes back, it sends a new packet. The variable xpctACKnum holds the ACK number we

are expecting to receive based on the packets that were just sent and their corresponding sequence numbers. If the sequence number of the ACK we receive does not match xpctACKnum, our code handles this issue by retransmitting the packet. This causes packets to be received out of order on the client's side, but all 180,000 packets do get sent. To make the picture on the webpage clear without any streaks, we would have to make changes to the server code (which we did not do).

ATFURN as server, Gianna and Matt as the Client

Below is a photo of our client running with ATFURN as the server and Matt and Gianna's machine as the client. Matt and Gianna's machine does not reorder packets, which is why the picture is very clear and the webpage reports no out of order packets. We killed both programs early which is why it does not finish sending all 180,000 packets but if we had let it keep running all 180,000 packets would have been sent.

Speedy Gonzales as Server and ATFURN as the Client

The photo above is with the same protocol, but ATFURN is known for reordering packets which can be seen by the streaky lines in the photo. Again, we killed the program early, but all 180,000 packets would have been sent.

Documentation of better_client.py protocol with retransmissions, timeouts, sliding window and handling of duplications and misordering of packets

Overall, our protocol performs quite well - significantly better than that of client_saw.py. While using a sliding window of size 10, retransmissions, etc. along with a timeout of .5 seconds we see only a handful of glitches here and there (see photo 1). We tried our client with first a timeout of three seconds, then a timeout of one second, and finally ending with half a second; we noticed no change in quality of performance, only that the one with the shorter timeout simply performs faster. Based on the data from figure 3 and some math, we approximate that sending 180,000 packets would take about 54,306 seconds or 15 hours. Interestingly enough this is almost double the amount of time of the

original protocol! We assume this has to do with the significant amount of retransmissions it is doing (almost every time); furthermore, if we wished to experiment with continuously reducing the timeout time we could likely improve performance significantly. The only reason it is retransmitting (for the most part) is due to the misordering of the packets and many duplicate packets were sent (see figure 3); if that were not the case then undoubtedly our protocol would perform better. One solution could be switching the client to another machine instead of ATFURN, like Gianna and Matt's machine. Another would be adding to the protocol to handle duplications better.

Photo 1

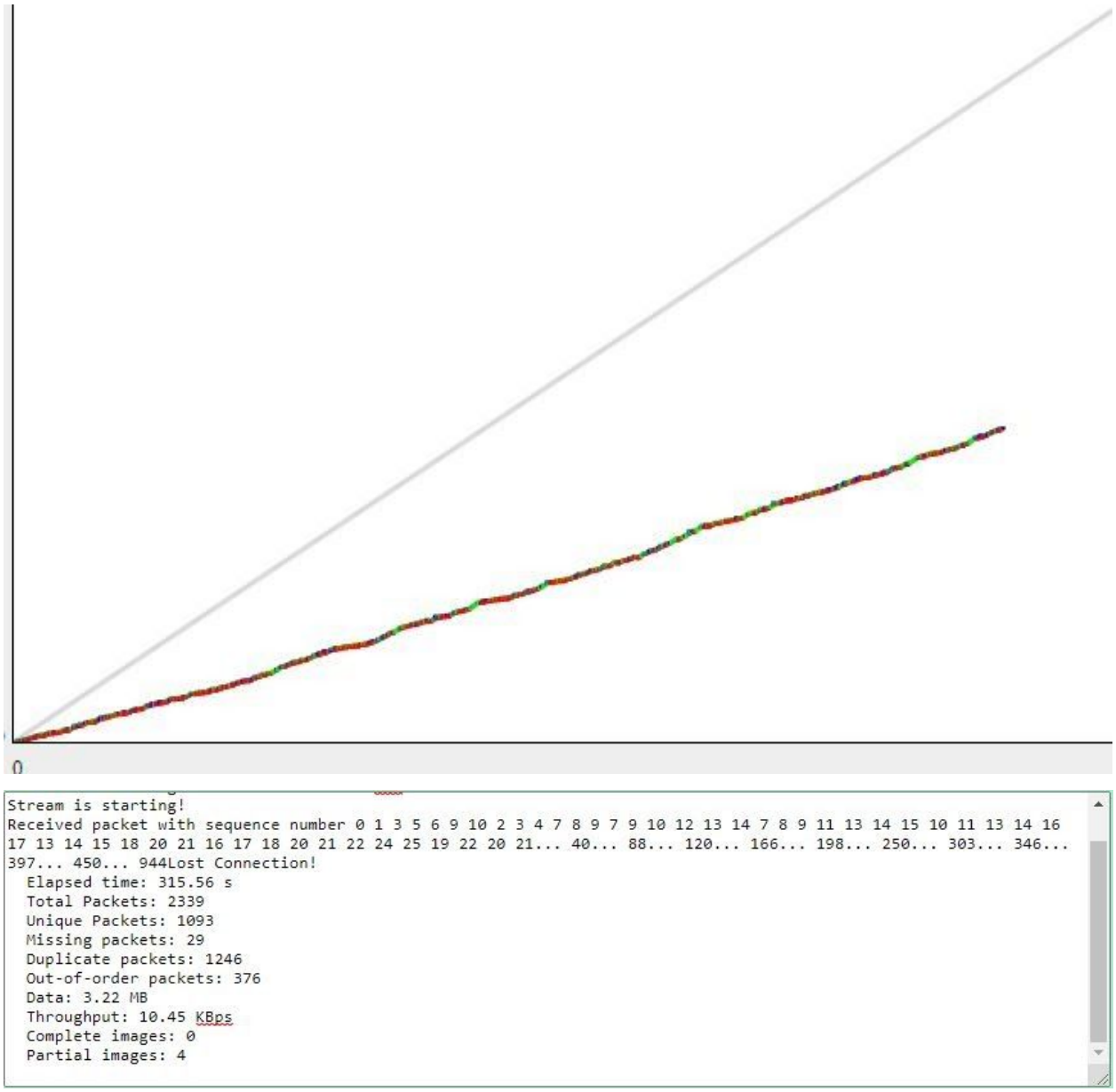


Figure 3

Collaboration Log:

Worked with Alexa Furnari for the duration of the project.

12.06.2020 Messaged Evo regarding misordering and duplicate packets.

Met with Professor Walsh several times over the duration of the project.