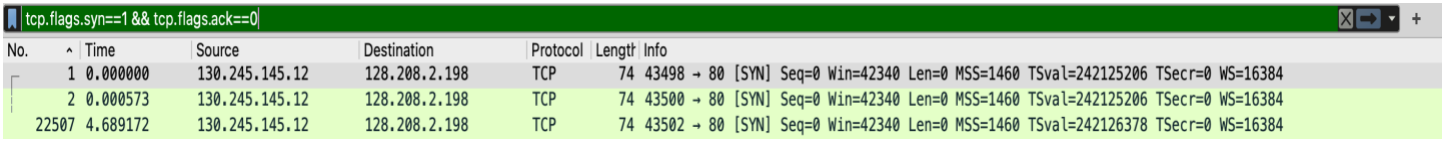
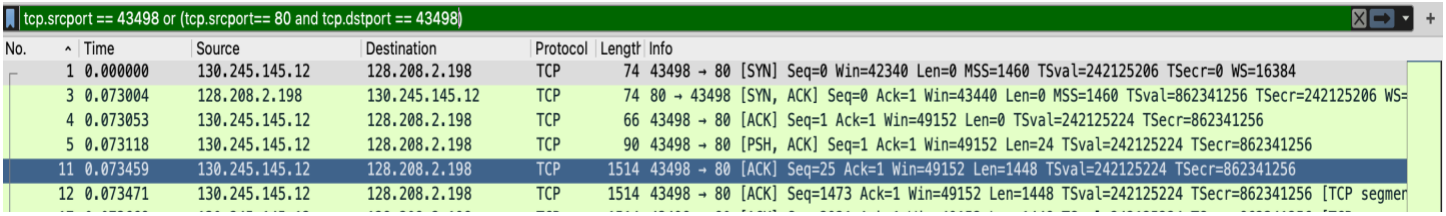
CSE534 HOMEWORK 2:

PART A

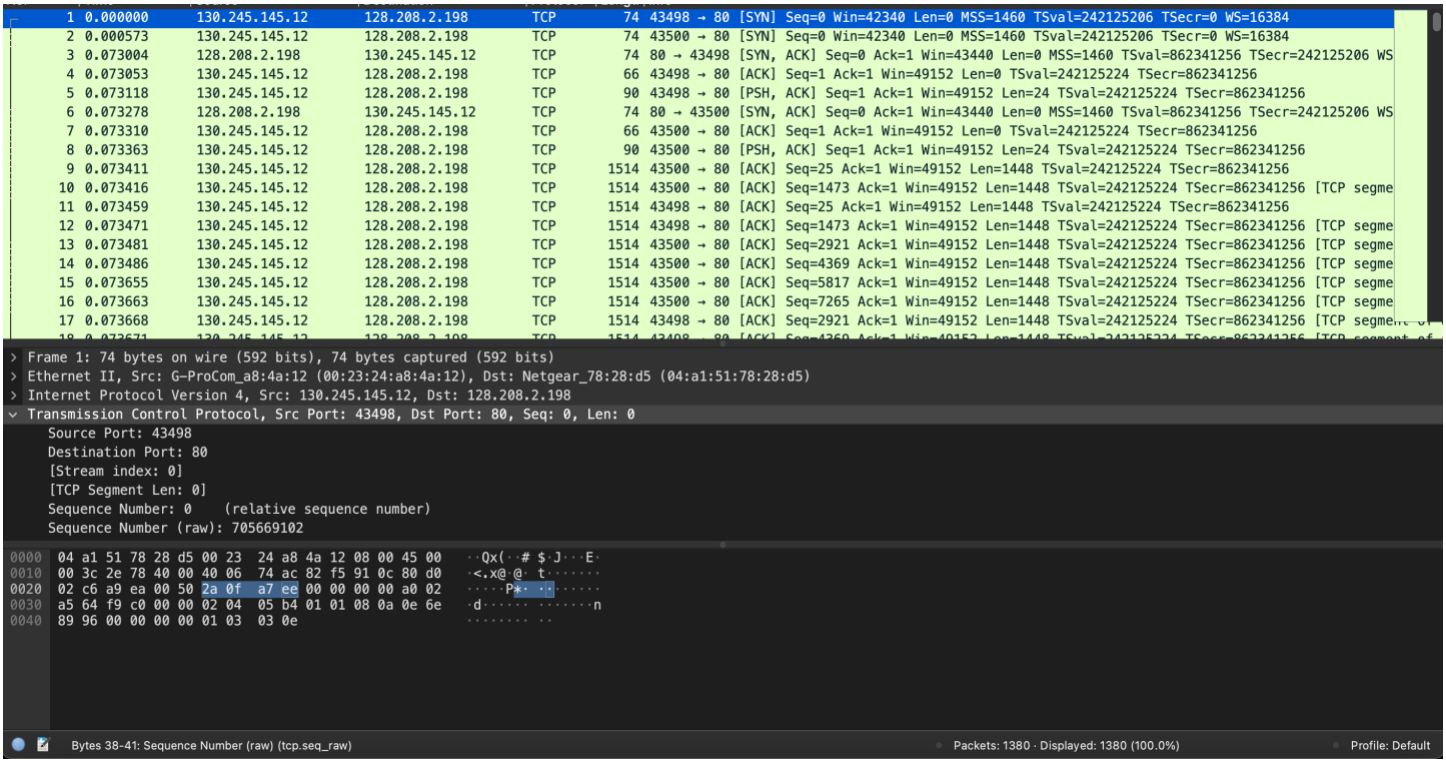
This assignment uses the Python dpkt library to read the packet byte stream from the assignment2.pcap file. To parse each packet and extract all the important TCP header fields, I am making use of the **struct** package in python (https://docs.python.org/3/library/struct.html). To better understand the header formats and the specific byte ranges where pertinent header fields can be found, we can make use of the WireShark tool, which as we can see in the below images, helps us to view and understand packet structures better by providing search and filtering capabilities. Our code will now parse only the necessary fields at the byte ranges we determine from WireShark, which is also very useful when verifying the results we obtain.



1. Filter to understand number of connections/flows established during the 3-Way Handshake phase



2. Filter TCP Flow specific to port 43498 from the source host



3. We can parse the Sequence Number at bytes 38-41 (as seen in bottom of screen)

1. Count the number of TCP flows initiated from the sender
   - **Output:**
     Number of TCP Connections initiated from 130.245.145.12 = 3, on ports 43498,43500,43502
   - **Explanation**:
     Number of TCP flows by is the same as the number of SYN-ACK packets sent from the receiver. An additional check is made to ensure that each of them have a corresponding FIN packet at the end of the flow. A successful handshake, denoted through the SYN-ACK flag, hence also implies a new connection.

2. For each TCP flow
   a. For the first 2 transactions after the TCP connection is set up (from sender to receiver), get the values of the Sequence number, Ack number, and Receive Window size.
   - **Output:**
     ::::::::::::::::::::Flow 1::::::::::::::::::::
     Transaction 1:
     Sender 130.245.145.12: Sequence Number = 705669103, Ack Number = 1921750144, Calculated Receive Window Size = 49152
     Receiver 128.208.2.198: Sequence Number = 1921750144, Ack Number = 705669127, Calculated Receive Window Size = 49152

     Transaction 2:
     Sender 130.245.145.12: Sequence Number = 705669127, Ack Number = 1921750144, Calculated Receive Window Size = 49152
     Receiver 128.208.2.198: Sequence Number = 1921750144, Ack Number = 705670575, Calculated Receive Window Size = 49152

     ::::::::::::::::::::Flow 2::::::::::::::::::::
     Transaction 1:
     Sender 130.245.145.12: Sequence Number = 3636173852, Ack Number = 2335809728, Calculated Receive Window Size = 49152
     Receiver 128.208.2.198: Sequence Number = 2335809728, Ack Number = 3636173876, Calculated Receive Window Size = 49152

     Transaction 2:
     Sender 130.245.145.12: Sequence Number = 3636173876, Ack Number = 2335809728, Calculated Receive Window Size = 49152
     Receiver 128.208.2.198: Sequence Number = 2335809728, Ack Number = 3636175324, Calculated Receive Window Size = 49152

     ::::::::::::::::::::Flow 3::::::::::::::::::::
     Transaction 1:
     Sender 130.245.145.12: Sequence Number = 2558634630, Ack Number = 3429921723, Calculated Receive Window Size = 49152
     Receiver 128.208.2.198: Sequence Number = 3429921723, Ack Number = 2558634654, Calculated Receive Window Size = 49152

     Transaction 2:
     Sender 130.245.145.12: Sequence Number = 2558634654, Ack Number = 3429921723, Calculated Receive Window Size = 49152
     Receiver 128.208.2.198: Sequence Number = 3429921723, Ack Number = 2558636102, Calculated Receive Window Size = 49152

- **Explanation:**
  - As expected from the question, we are considering only the first 2 transactions after the TCP handshake. We will first be considering the sender packet with the PSH flag set, and the corresponding receiver packet whose sequence number is same as the PSH packet's acknowledgement number. These 2 packets comprise the 1st transaction. The 2nd Transaction too is derived similarly.
  - Receiver window size is calculated as the product of the window size (3) and the scaling factor (2^14 = 16384)

b. Compute the throughput at the receiver

- **Output:**
  Flow 1 Throughput = 5.251 MBps
  Flow 2 Throughput = 1.285 MBps
  Flow 3 Throughput = 1.482 MBps
- **Explanation:**
  Throughput, in megabytes per second, is calculated as the number of megabytes transmitted per second. The entire packet size (header + payload length) is considered for calculating the throughput. Since it is asked to calculate throughput at sender, we are considering only sender packets for byte calculation.

c. Compute the loss rate for each flow

- **Output:**
  Flow 1: Number of lost sender packets = 3, Loss Rate = 0.00043
  Flow 2: Number of lost sender packets = 94, Loss Rate = 0.0132994
  Flow 3: Number of lost sender packets = 0, Loss Rate = 0.0
- **Explanation:**
  Loss rate is the number of packets not received at the receiver, over the number of packets sent from the sender. Thus, even here we consider only sender packets. An alternate way to define loss rate would be the number of times packets with duplicate sequence numbers are sent, which happens when the packet corresponding to that sequence number was not received at the receiver. This is the implementation I have gone with.

d. Estimate the average RTT. Now compare your empirical throughput from (b) and the theoretical throughput (estimated using the formula derived in class)

- **Output:**
  Flow 1: Avg RTT = 0.07314, Theoretical Throughput = 1.179 MBps
  Flow 2: Avg RTT = 0.07284, Theoretical Throughput = 0.2129 MBps
  Flow 3: Avg RTT = 0.07222, Theoretical Throughput = inf MBps
- **Explanation:**
  - Average RTT (round-trip time) is calculated as the sum of the individual round-trip times of every transaction, divided by the number of such transactions (or ACKs from the receiver).
  - For calculating avg RTT, we can exclude retransmitted packets from the sample (using Karn's algorithm)
  - Theoretical Throughput = (sqrt(3/2) * MSS) / (RTT * sqrt(loss rate))
  - In the first 2 flows, we see that the actual throughput from b) is greater than the calculated theoretical throughput. This may be due to some of the assumptions we make when calculating the average RTT and the loss rate. Since the loss rate is zero for the third flow, the theoretical throughput becomes infinity, which is not realistic since there is always some packet loss practically.

PART B

1. Print the first ten congestion window sizes. Mention the size of the initial congestion window. You need to estimate the congestion window size empirically since the information is not available in the packet. Comment on how the congestion window size grows.
    - **Output:**

      Flow 1:
      First 10 congestion windows (after handshake): [22800, 36336, 62074, 74186, 112036, 157456, 234670, 281604, 436032, 635880]
      Growth Rate of Congestion Window: [1.594, 1.708, 1.195, 1.51, 1.405, 1.49, 1.2, 1.548, 1.458]

      Flow 2:
      First 10 congestion windows (after handshake): [13716, 30280, 52990, 71158, 95382, 157456, 205904, 331566, 434518, 608628]
      Growth Rate of Congestion Window: [2.208, 1.75, 1.343, 1.34, 1.651, 1.308, 1.61, 1.311, 1.401]

      Flow 3:
      First 10 congestion windows (after handshake): [13716, 30280, 49962, 68130, 99924, 134746, 204390, 274034, 221334]
      Growth Rate of Congestion Window: [2.208, 1.65, 1.364, 1.467, 1.348, 1.517, 1.341, 0.808]

    - **Explanation:**
        - To compute the empirical values of the first 10 congestion window sizes, we make use of the average RTT we calculated in 2c) of section PART A. We exclude the handshake packets from the sample set. Since congestion window determines the rate at which sender packets are sent, we consider only the sender packets.
        - I am estimating the congestion window size to be the total number of bytes sent between 2 RTTs (hence the use of avg RTT, which is average time between 2 successive ACKs). Thus, the total number of bytes sent by the sender in the time-window between 2 RTTs is taken as the congestion window size.
        - Initial Congestion Window size is usually same as the packet MSS, but in our results, we see 22800, 13716 and 13716 in flows 1-3 respectively.
        - We observe that the congestion window is multiplicatively increasing (but not doubling) majority of the times with every ACK received, across all flows (only one time we observe a growth rate of < 1, in flow 3). This implies that congestion has not been encountered yet during the first 10 ACKs.

2. Compute the number of times a retransmission occurred due to triple duplicate ack and the number of times a retransmission occurred due to timeout
    - **Output:**

      Flow 1:
      Number of Triple Duplicate ACKs = 2
      Number of Timeouts = 1

      Flow 2:
      Number of Triple Duplicate ACKs = 36
      Number of Timeouts = 58

Flow 3:
Number of Triple Duplicate ACKs = 0
Number of Timeouts = 0

- **Explanation:**
  - The number of packets lost calculated in 2c) of section Part A is the total number of sender packets lost, that is, the number of sequence numbers for which retransmissions have been requested from the sender.
  - Triple Duplicate Acknowledgements (TDA) is calculated if there are at least three packets from the sender with the acknowledgement number equal to the sequence number of a retransmitted packet.
  - Therefore, packet loss due to timeouts is calculated as (number of packets lost – number of TDA received)

PART C

In this section, we use the Linux **tcpdump** utility to capture, filter, and analyze network traffic originating from the host machine. Per the question, we need to listen to connections made from my local to remote server http://www.sbunetsyslabs.com at port 1080, and to https://www.sbunetsyslabs.com at ports 1081 and 1082 to generate packet capture files containing TCP/HTTP data which we will then analyze. The exact tcpdump command used is as follows from my M1 MacOS machine:

- tcpdump -i en0 -n port 1080 -w http_1080.pcap
- tcpdump -i en0 -n port 1081 -w http_1081.pcap
- tcpdump -i en0 -n port 1082 -w http_1082.pcap

The above commands mean that tcpdump will listen for outgoing requests originating from the en0 interface of my machine, which is the Ethernet interface (i.e Wi-Fi), to ports 1080-82 of remote servers, and write the packets captured during the exchange into the pcap files specified after the -w option. As an example, to monitor the traffic to http://www.sbunetsyslabs.com:1080, we first run the first command, which starts the packet capture process in the foreground, and then hit the URL on a browser with cache disabled. Once the page loads and all requests have been served, we terminate the capture process with a ^C which then creates the output pcap file which we will use for our analysis. The below image explains this process.

```
(base) asomayaj@ASOMAYAJ-MBP-295 ~ % tcpdump -i en0 -n port 1080 -w http_1080.pcap
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C1977 packets captured
5408 packets received by filter
0 packets dropped by kernel
(base) asomayaj@ASOMAYAJ-MBP-295 ~ % tcpdump -i en0 -n port 1081 -w http_1081.pcap
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C22 packets captured
1553 packets received by filter
0 packets dropped by kernel
(base) asomayaj@ASOMAYAJ-MBP-295 ~ % tcpdump -i en0 -n port 1081 -w http_1081.pcap
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C2638 packets captured
2818 packets received by filter
0 packets dropped by kernel
(base) asomayaj@ASOMAYAJ-MBP-295 ~ % tcpdump -i en0 -n port 1082 -w http_1082.pcap
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C2160 packets captured
2359 packets received by filter
0 packets dropped by kernel
```

1. Reassemble each unique HTTP Request/Response for http_1080.pcap. The output of this part should be the Packet type (request or response) and the unique <source, dest, seq, ack> TCP tuple for all the TCP segments that contain data for that request.
   - **Output:**
     Refer to part_c_1.txt in the **PART_C** folder of the project directory.
   - **Explanation:**
     - The HTTP request can be found in the payload that starts with '**GET'.** Similarly, the HTTP response can be found in the payload that starts with '**HTTP'.**
     - We then print all the TCP segments which are part of the flow by matching the acknowledgement numbers of the requests to the sequence numbers of the responses.

2. Identify which HTTP protocol is being used for each PCAP file. Note that two of the sites are encrypted so you must use your knowledge of HTTP and TCP to programmatically solve this question.
   - **Output:**
     Number of Flows/Connections established to the site on port 1080 = 17, which is equal to the number of GET requests made. Hence, it must be using HTTP 1.0, which creates a new connection for every request. Additionally, we can also confirm this with the Response section in part_c_1.txt file, where we can see HTTP/1.0 in the beginning of the response.

     Number of Flows/Connections established to the site on port 1081 = 6, which must imply that it must be using HTTP 1.1 which reduces the number of persistent connections through parallelization of requests (based on Browser settings, which is 6 by default).

     Number of Flows/Connections established to the site on port 1082 = 1, implying that it must be using HTTP 2.0 which is known to make use of a single connection by making use of the pipelining technique to send all objects within one connection itself
   - **Explanation:**
     Explained in Output.

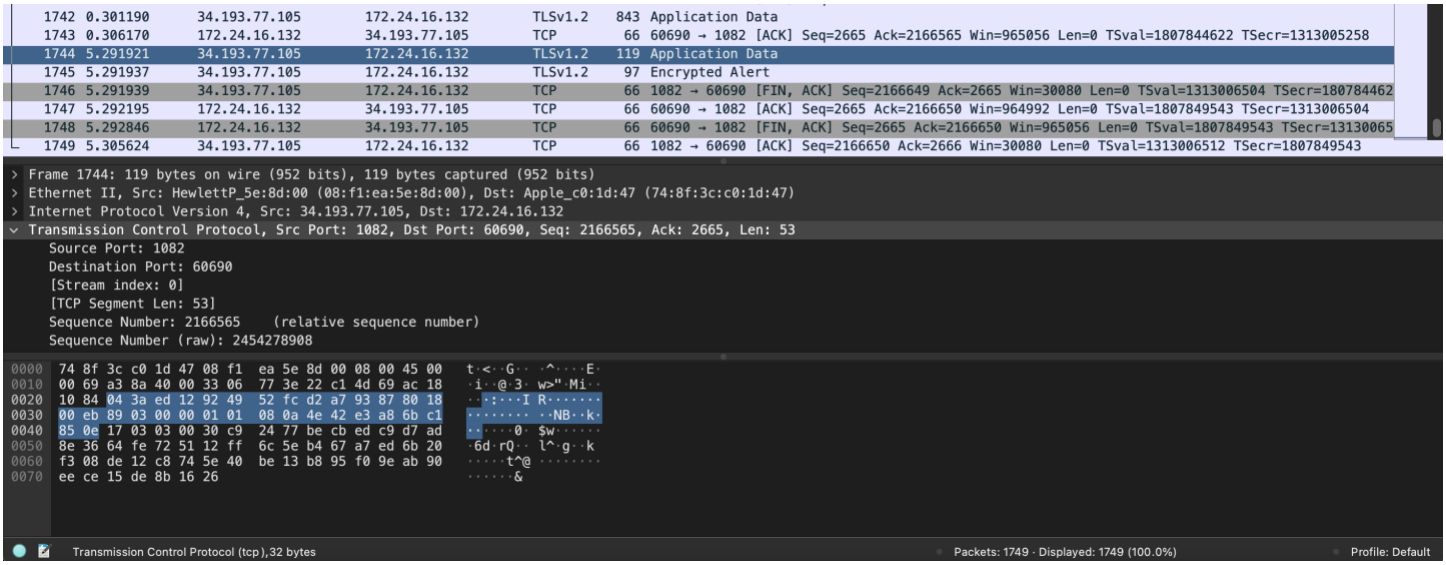
3. Which version of the protocol did the site load the fastest under? The slowest? Which sent the most number of packets and raw bytes? Which protocol sent the least?
   - **Output:**
     HTTP/1.0 Connection Stats:
     Total number of Connections = 17
     Total Packets Transferred = 1966
     Total Transfer Time = 0.3392s
     Total Transfer Bytes = 2276284

     HTTP/1.1 Connection Stats:
     Total number of Connections = 6
     Total Packets Transferred = 1911
     Total Transfer Time = 5.3356s
     Total Transfer Bytes = 2285681

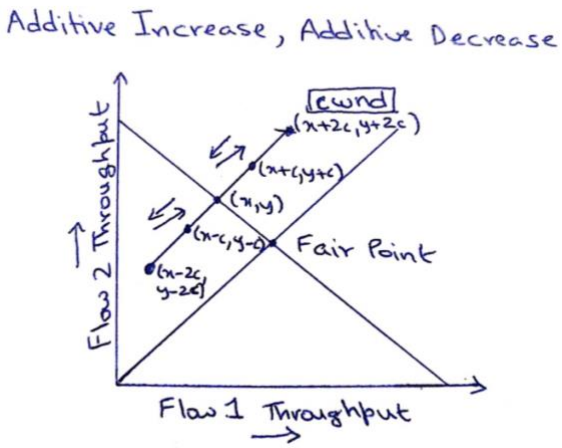
     HTTP/2.0 Connection Stats:
     Total number of Connections = 1
     Total Packets Transferred = 1749
     Total Transfer Time = 5.3056s
     Total Transfer Bytes = 2284766

- **Explanation:**
  - The number of packets transferred seems to be most for the HTTP/1.0 connection, and the least for HTTP/2.0 one.
  - The total transferred bytes seem to be comparable across 1.1 and 2.0, and lesser in 1.0. This is because both 1.1 and 2.0 use compression.
  - A surprising observation we see here is with respect to the total transfer time statistic. Intuitively, both 1.1 and 2.0 should be outperforming 1.0 due to their optimizations with respect to reduced TCP connections. However, the opposite is observed with 1.0 performing best at 0.33s, and the other 2

    exceeding 5.3s. An in-depth inspection of the packet flow on WireShark may throw some light on why this is happening. In row 1744, we can see a sudden jump of the total time from 0.3s to 5.3s. Both 1.1 and 2.0 use TLS, which may be the reason for the spike in total time at row 1745 (Total packets in this flow is 1749).

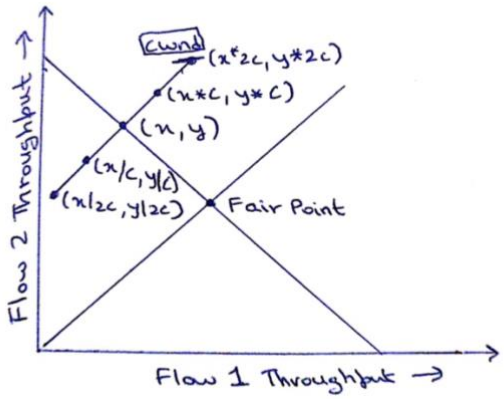


PART D - Fairness



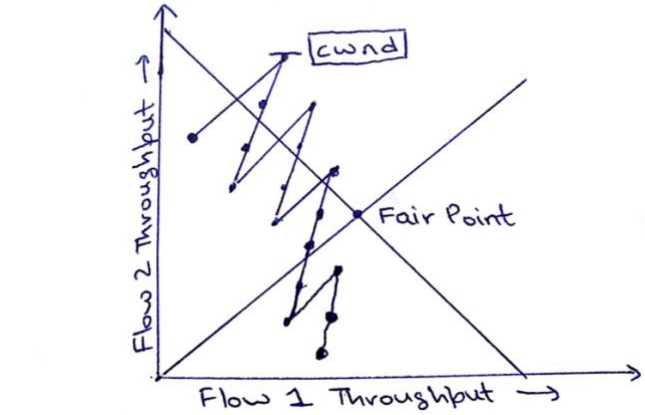
Let us assume that the additive constant is c. Until the cwnd is reached, both the x and y coordinates (which represent throughputs for Flows 1 and 2) iteratively increase by a constant factor c. Once it is reached, it will iteratively decrease by the same constant factor c. Hence as we can see from the above graph, the fair point will never be reached, and hence, fairness cannot be achieved.

Multiplicative Increase, Multiplicative Decrease



Like the AIAD process explained above, let us assume that the multiplicative constant is c. Until the cwnd is reached, both the x and y coordinates (which represent throughputs for Flows 1 and 2) iteratively scale up/multiply by constant factor c. Once it is reached, it will iteratively scale down/divide by the same constant factor c. Hence as we can see from the above graph, the fair point will never be reached, and hence, fairness cannot be achieved.

Multiplicative Increase, Additive Decrease



In this technique, the cwnd is reached very quickly due to multiplicative increase of the throughput, after which the descend back to the ssthreshold happens slowly iteratively (and with cwnd being reduced). Hence, across multiple cycles of ascent and descent, we will eventually descend below the fair point, and it will never be reached.