|  |  |  |
| --- | --- | --- |
| **[CS301]-Computer Networks** | Homework: 02 | Fall 2022-23 |
| Name: **Sudhir Sharma**  [All THE PCAP FILES](https://drive.google.com/drive/folders/1213JD_mI_gzS-bJ4QTL_oWMCBaxImuxv?usp=sharing) | RollNo: **12041500** | email: sudhirsharma@iitbhilai.ac.in |

**Solution of** PART 1:

**Wireshark** is used to capture packets when the ubuntu image is downloaded. The packet capture is carried out for about 75 seconds.

**a. INTERNET**

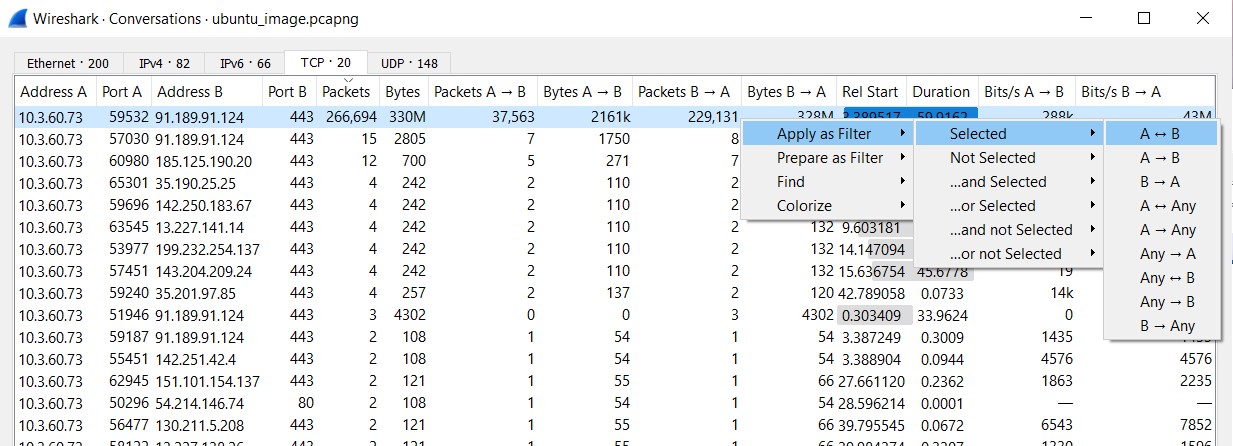
­­­

Figure 1.1: Applying filter

To find the IP used for downloading we go in **statistics→ conversations sort the packets in descending order**. Then we apply the filter for bidirectional. **The IP used is 91.189.91.124.**

We also apply the filter **ip.addr == 91.189.91.124 .**to filter out the packets involved in the download and proceed to plot the graph.

While downloading ubuntu iso image from internet we have run the wireshark for 75 sec and approx**. 400MB** out **of 3.6 GB** has been downloaded at the 75 sec.

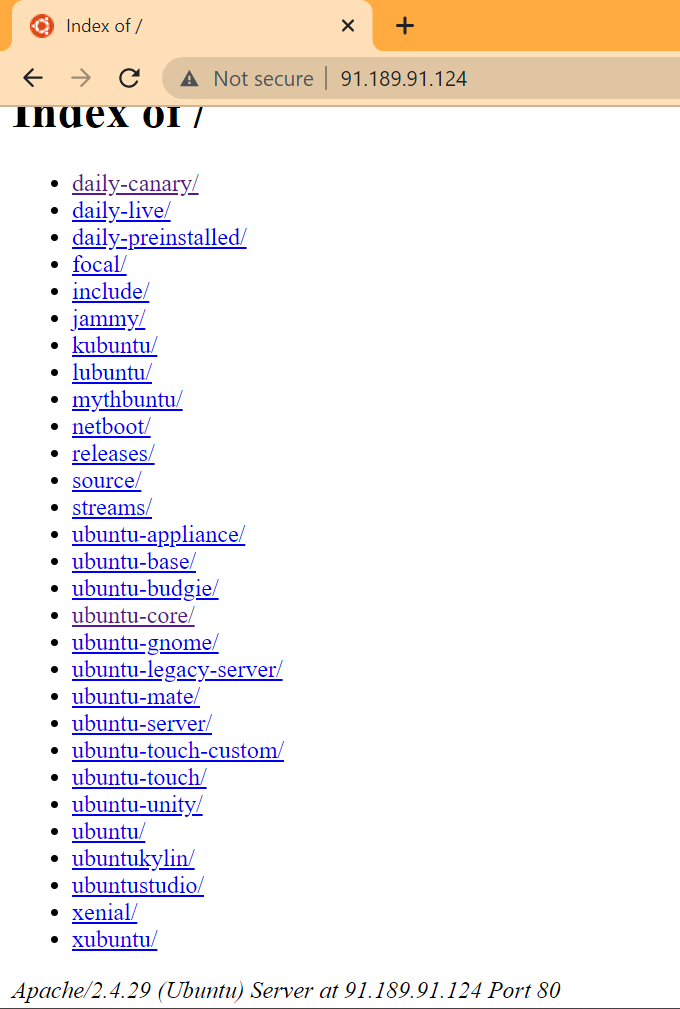


Fig shown the index of ubuntu server at 91.189.91.124

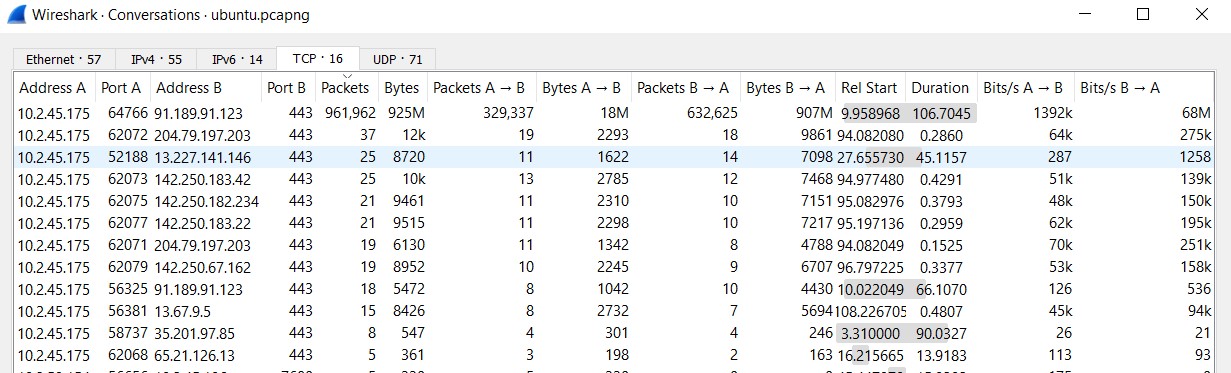
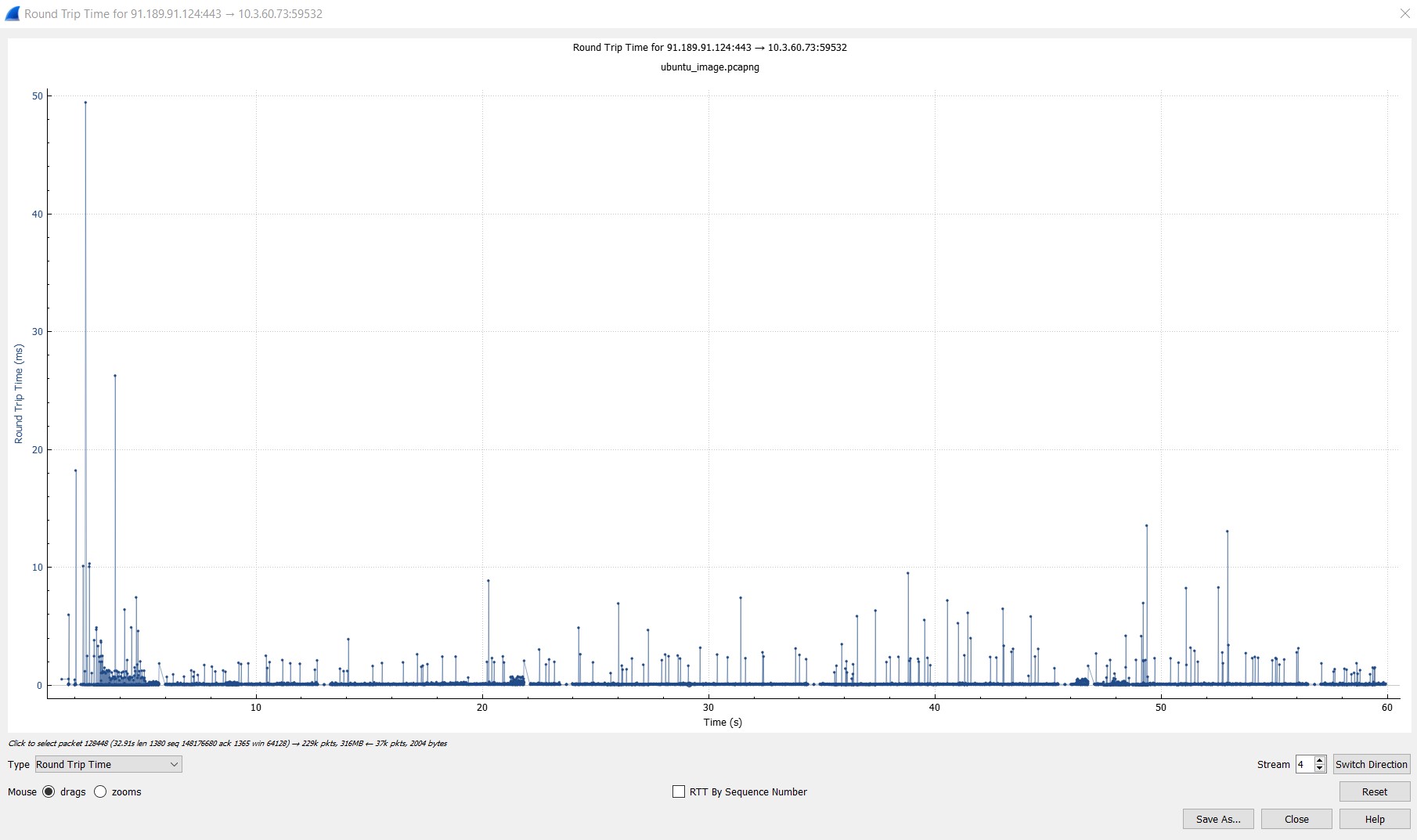


Figure 1.2: IP of download

For finding the round-trip time go to **statistics→TCP** stream graphs→ round trip time

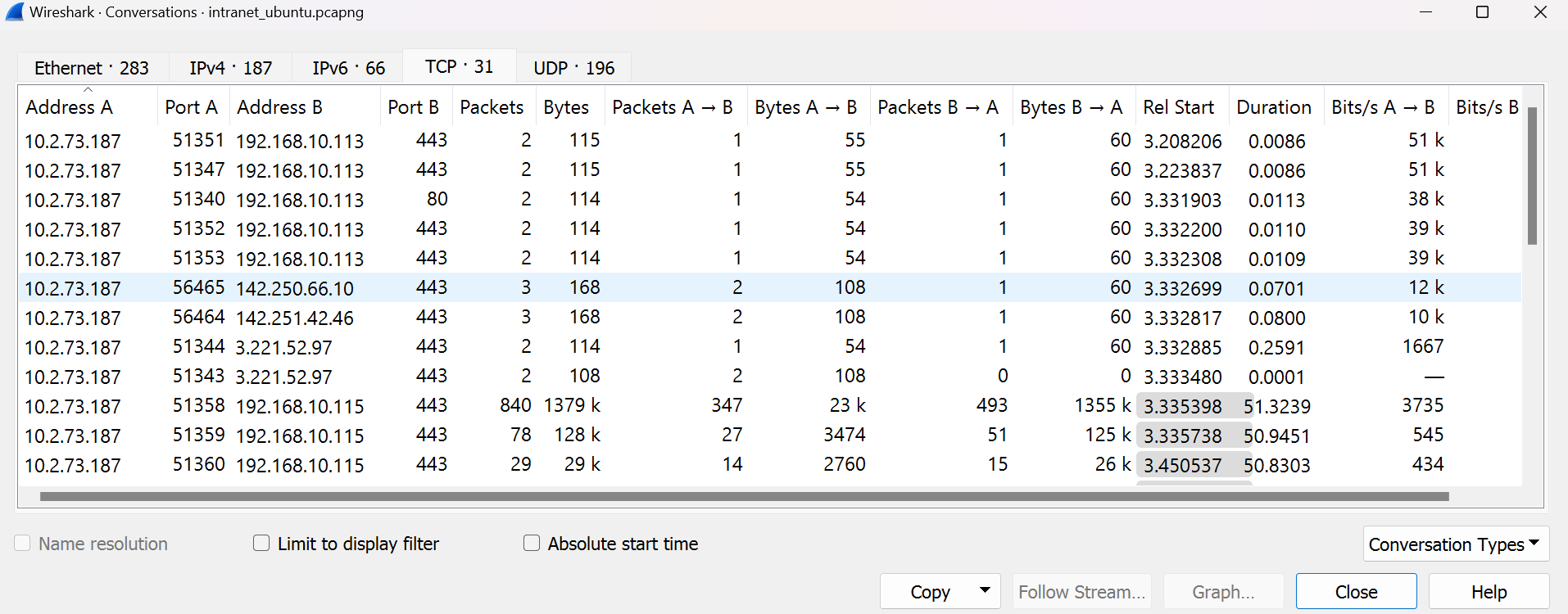
­­

**Figure 1.3: Round Trip Time For INTERNET**

**INTRANET**

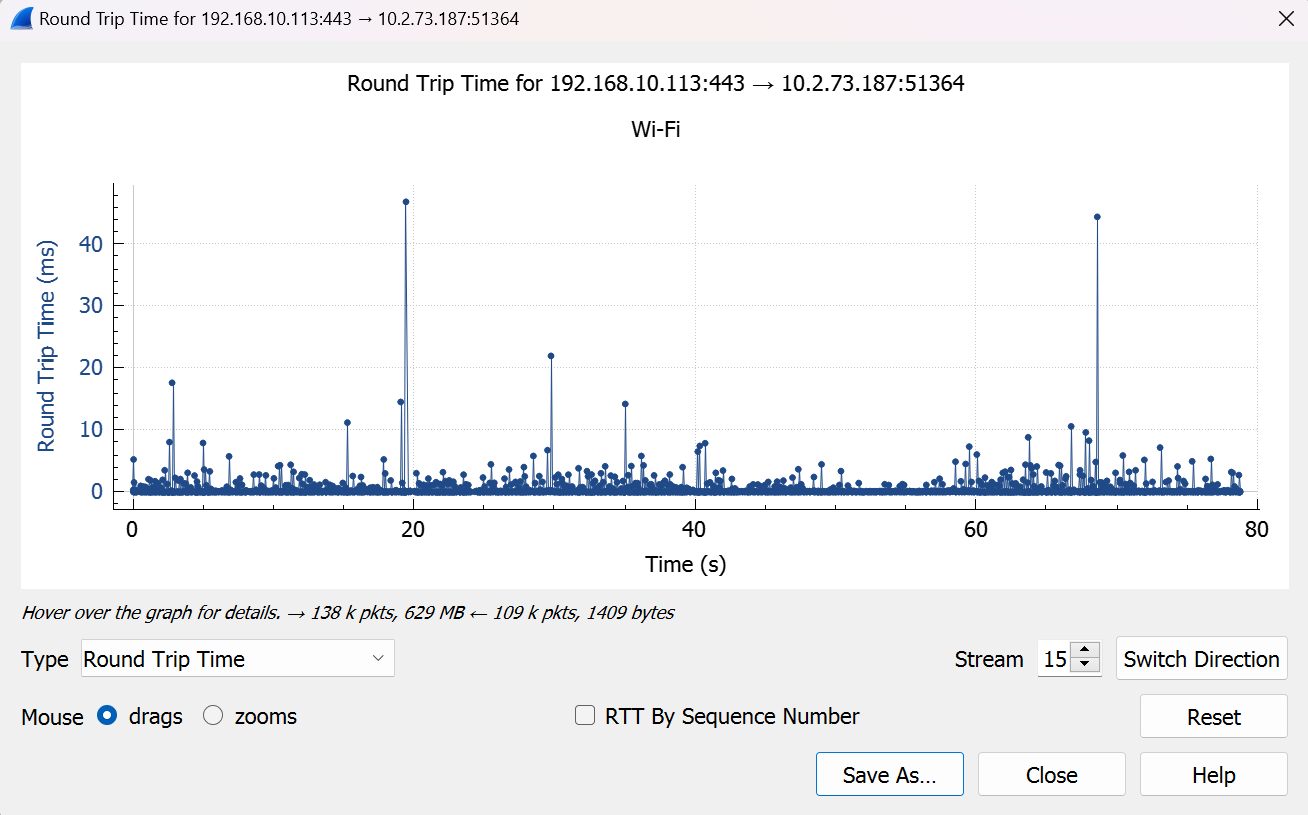
The pcap file given as intranet\_ubuntu.pcapng

To find the IP used for downloading we go in **statistics→ conversations sort the packets in descending order**. Then we apply the filter for bidirectional. **The IP used 192.168.10.113**

****

For finding the round-trip time go to **statistics→TCP** stream graphs→ round trip time.

While downloading ubuntu iso image from internet we have run the wireshark for 68 sec and approx**. 470MB** out **of 3.6 GB** has been downloaded at the 68 sec.

****

RTT for INTRANET

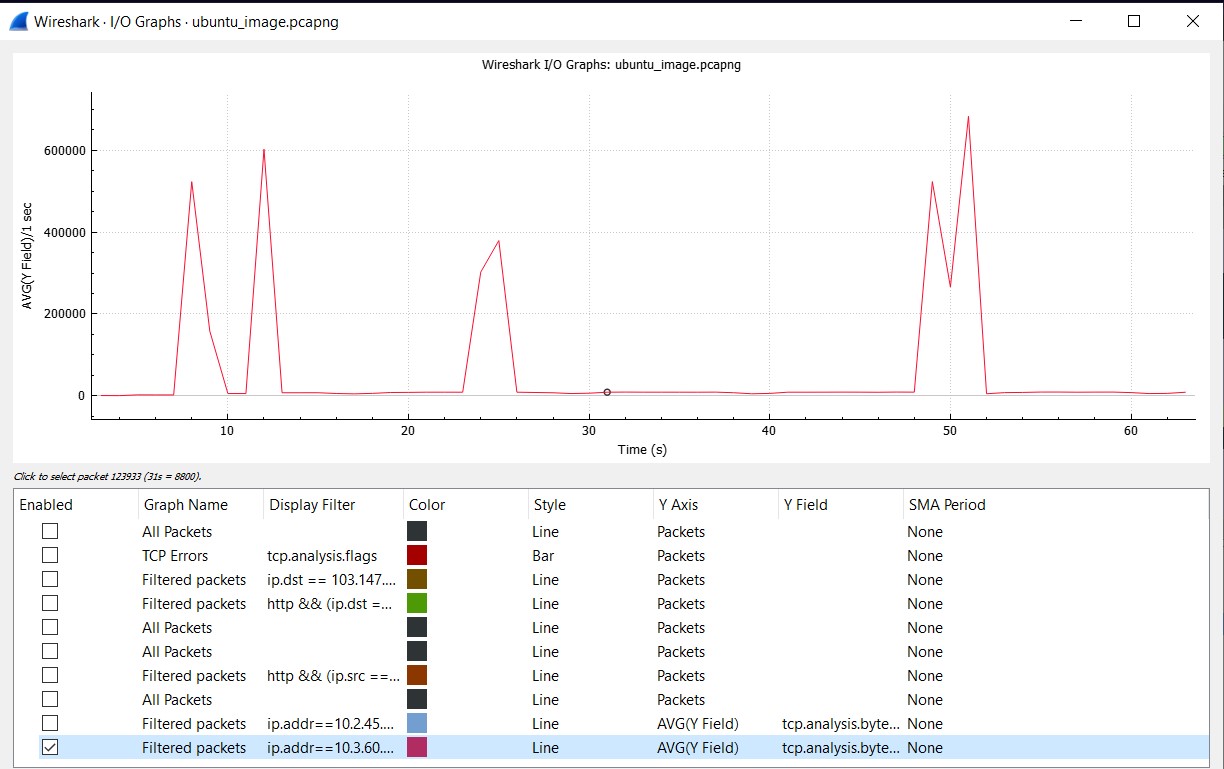
# Inference

**As speed of the internet increases RTT decreases**. Initially the RTT was very high but decreases gradually. Also most of the packets were from single IP 91.189.91.124.

**b.**

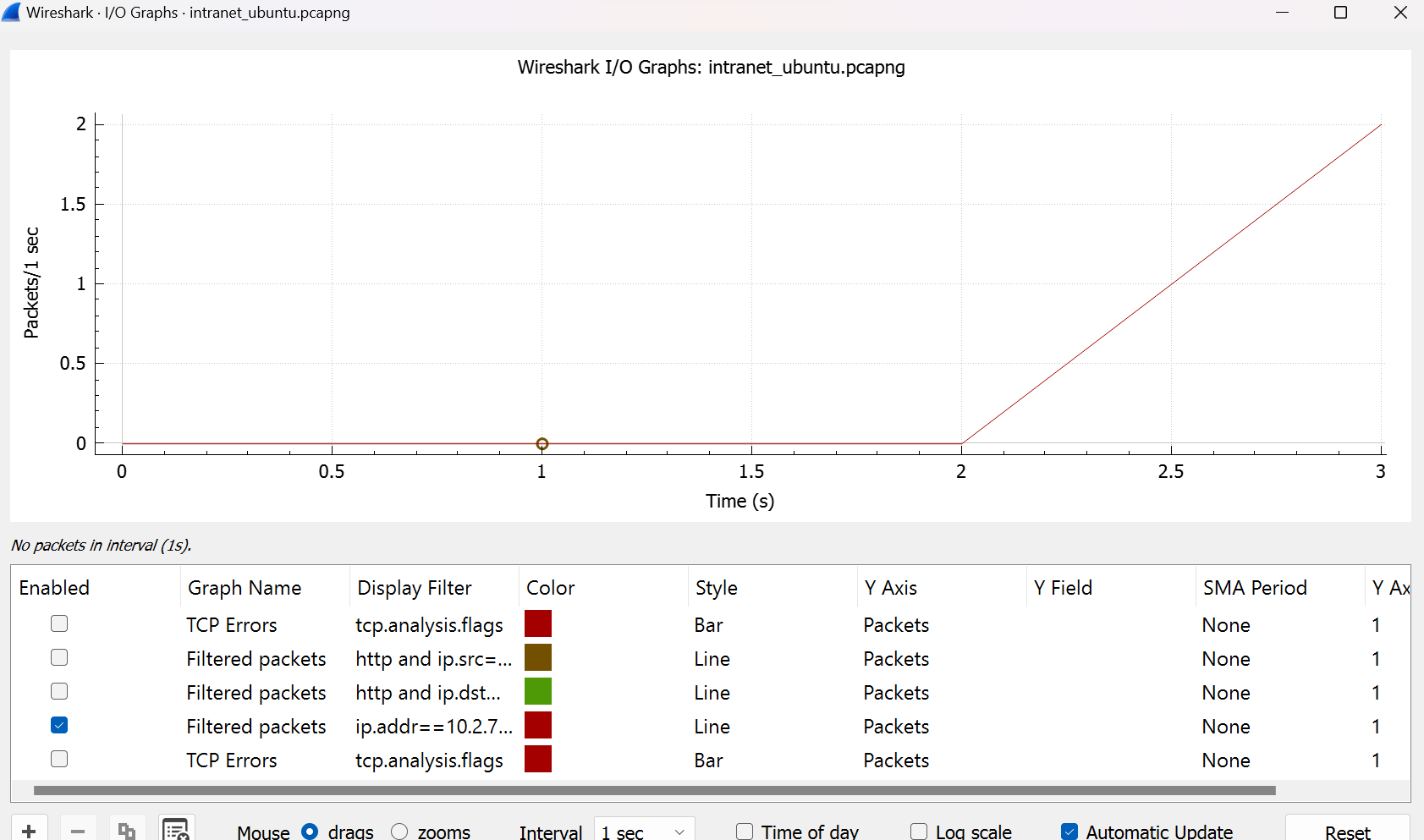
**For INTERNET**

We use Y-axis as avg of Y-field and set Y-field to tcp.analysis.bytes in flight. The graph is at **statistics→I/O** graphs.



**Figure 1.4: TCP congestion window**

**For INTRANET**

****

**Figure 1.5: TCP congestion window**

# **Inference**

At some points avg field is zero. This implies that all the bytes have been acknowledged till that point. The value of the congestion increases as congestion window decreases. Also the sender reduces the speed of sending whenever the bytes in flight increases to certain extent. Sender speed is set to zero if some packets are lost.

**c. statistics→flow graph for internet**

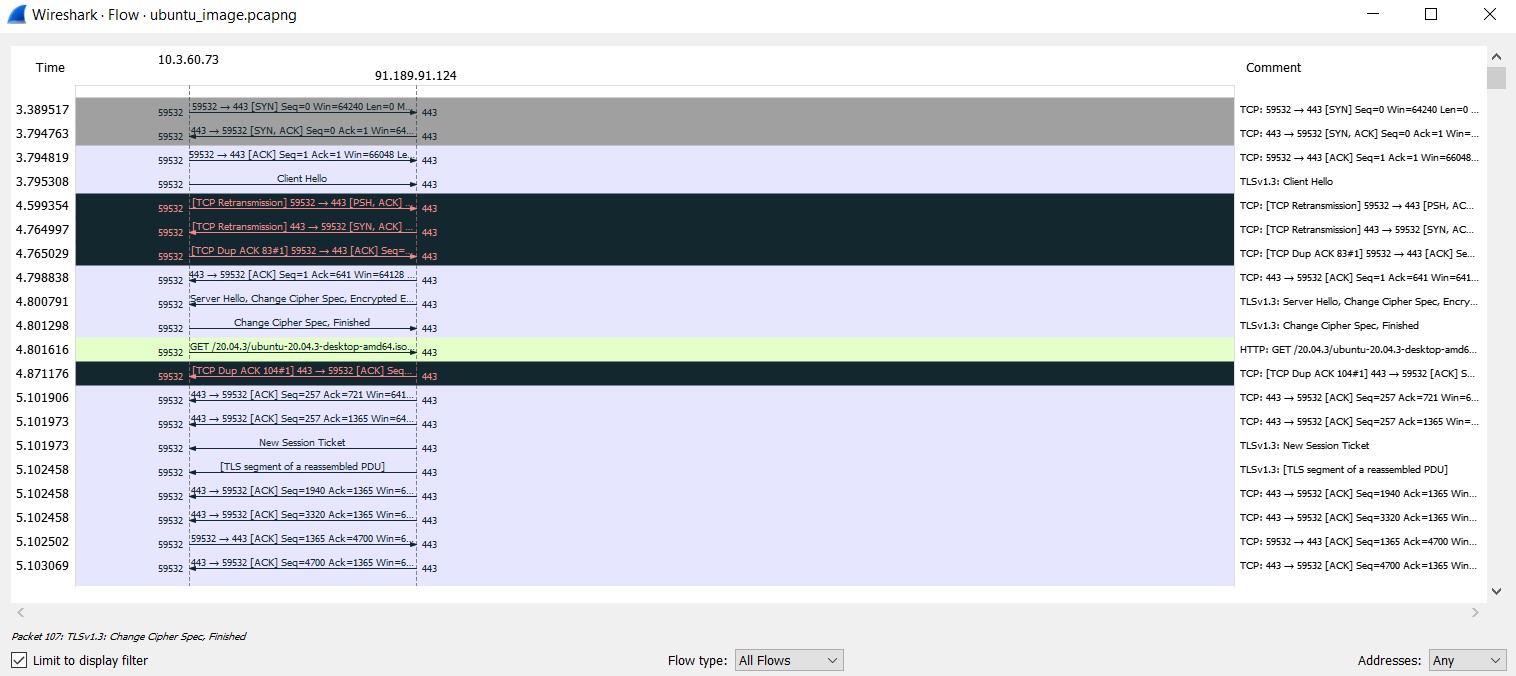
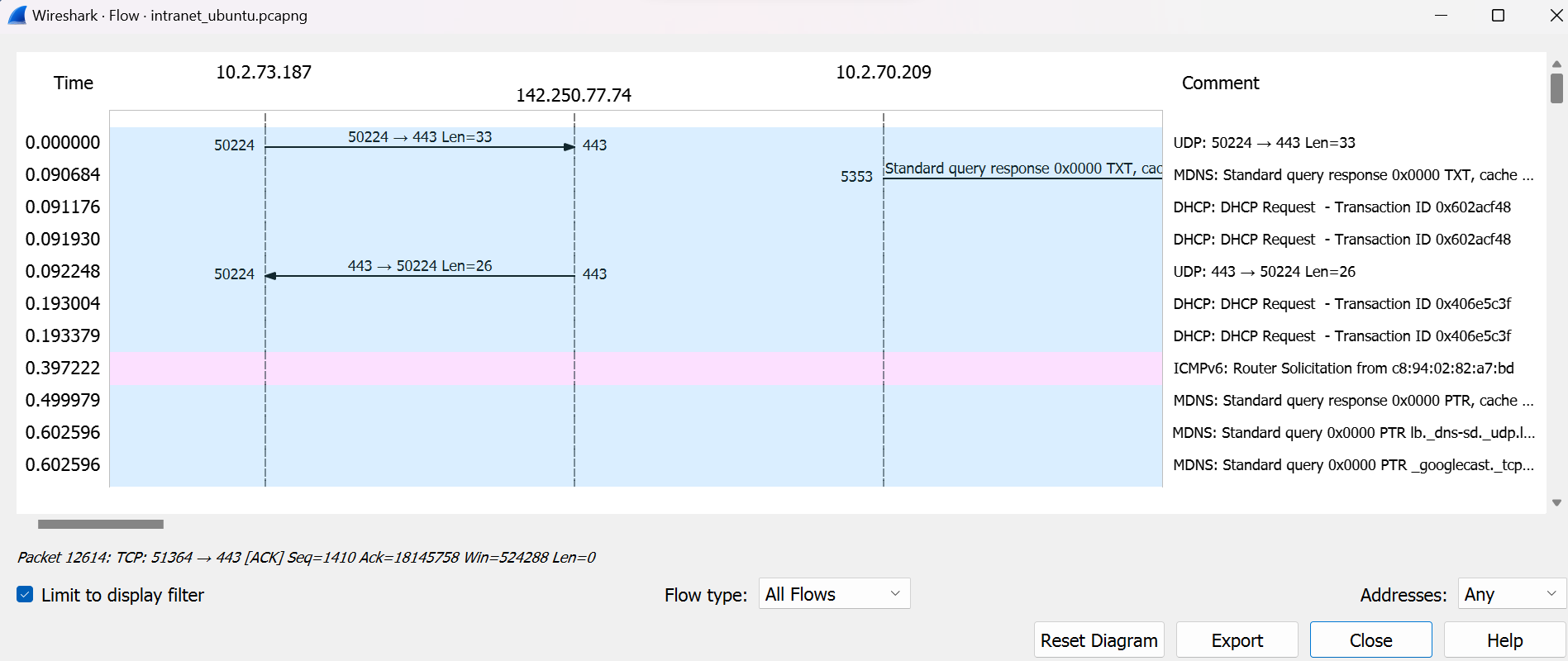
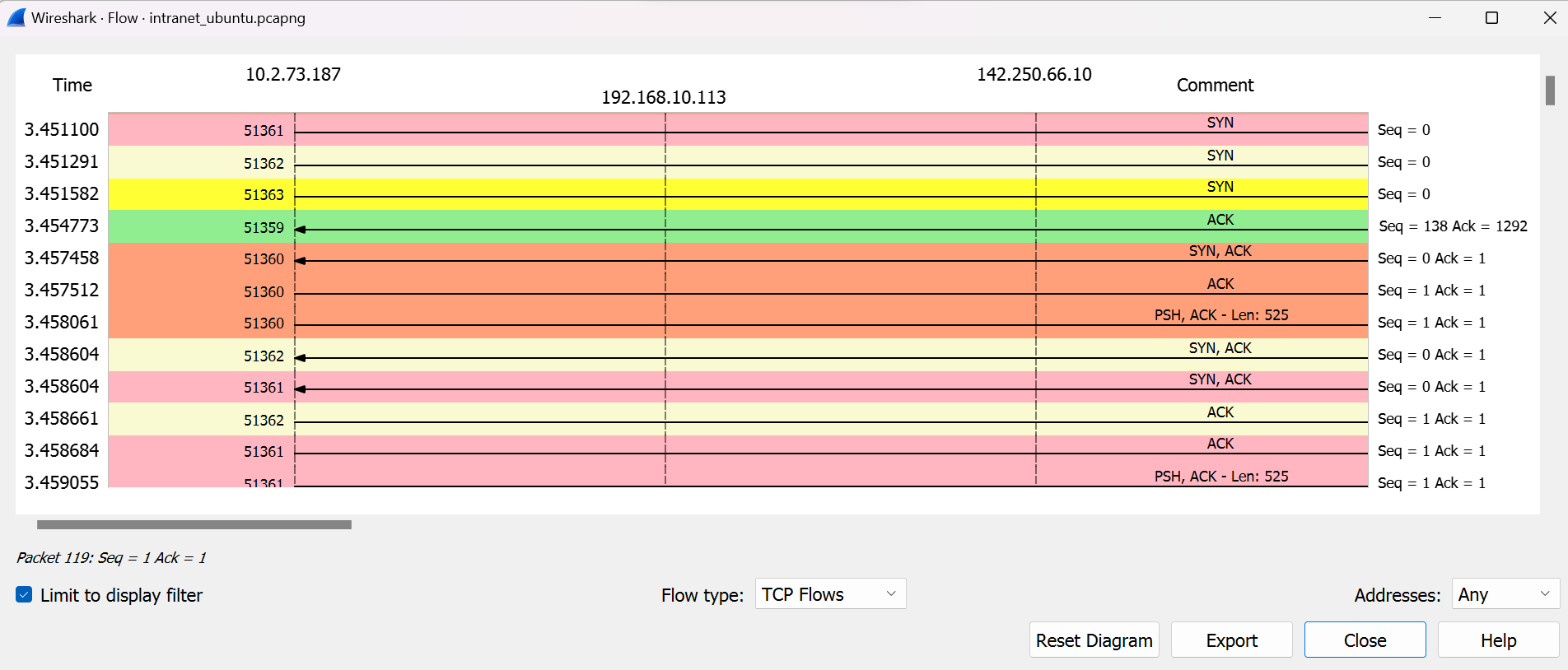


Figure 1.5: Flow Graph for internet

**For INTRANET**

****

**TCP Flow For INTRANET**

****

**Inference**

The Flow Graph shows connections between hosts. It highlights the packet time, direction, ports and comments for each captured connection. Each vertical line represents the specific host. The numbers in each row at the very left of the window represent the time packet. The numbers at the both ends of each arrow between hosts represent the port numbers.

1. We use the filter *ip.addr == 91.189.91.124* and then go to statistics→ capture file properties. The average throughput in bits/s is 41M.

**For INTERNET** ( you can also go to tcp stream-> thoughput)

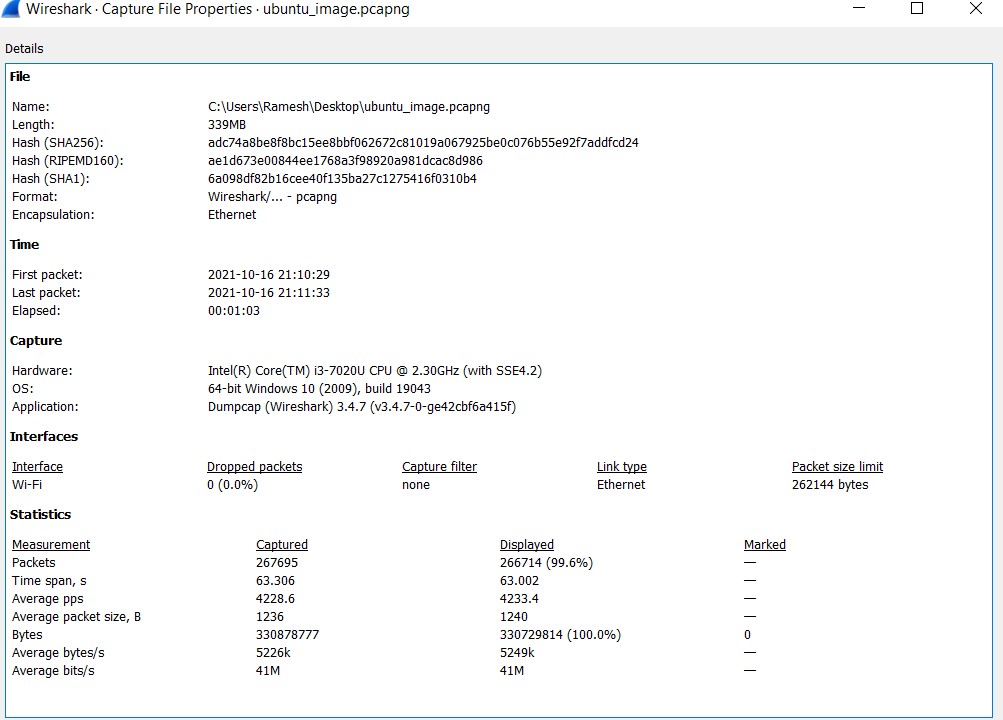
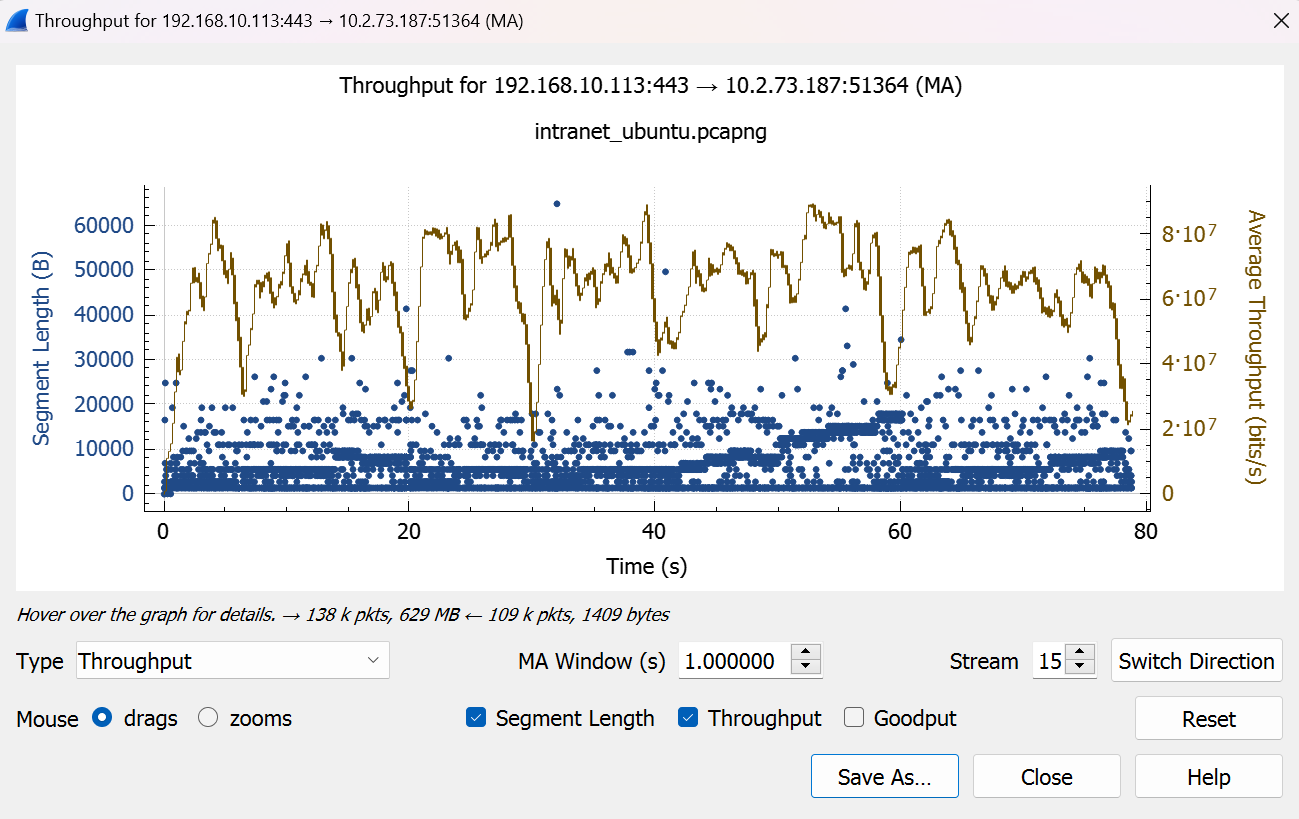
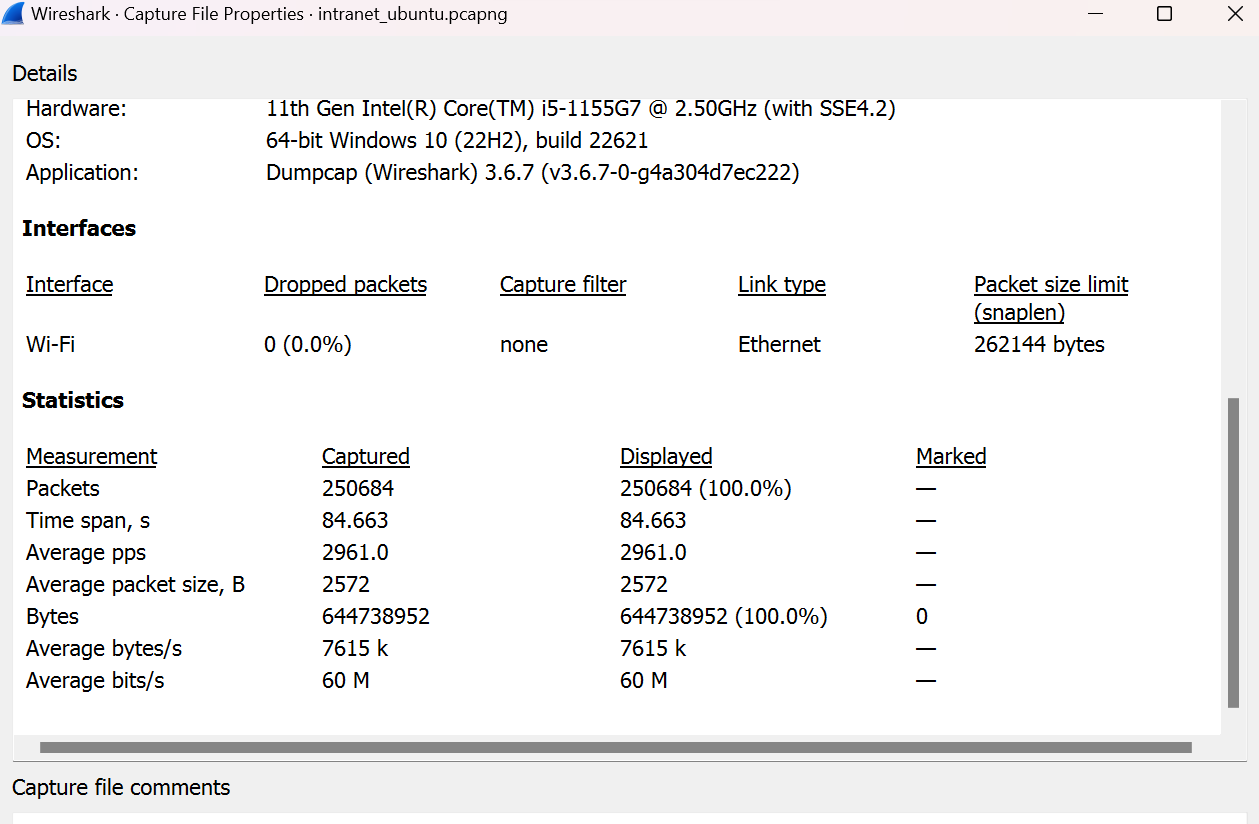


Figure 1.6: Average throughput for INTERNET

**For INTRANET**

****

****

The average throughput in bits/s is 60M.

1. For Receiver Window: **Statistics− *>*TCP Stream Graphs− *>*Window Scaling In this only check the rwnd option.**

**for internet**

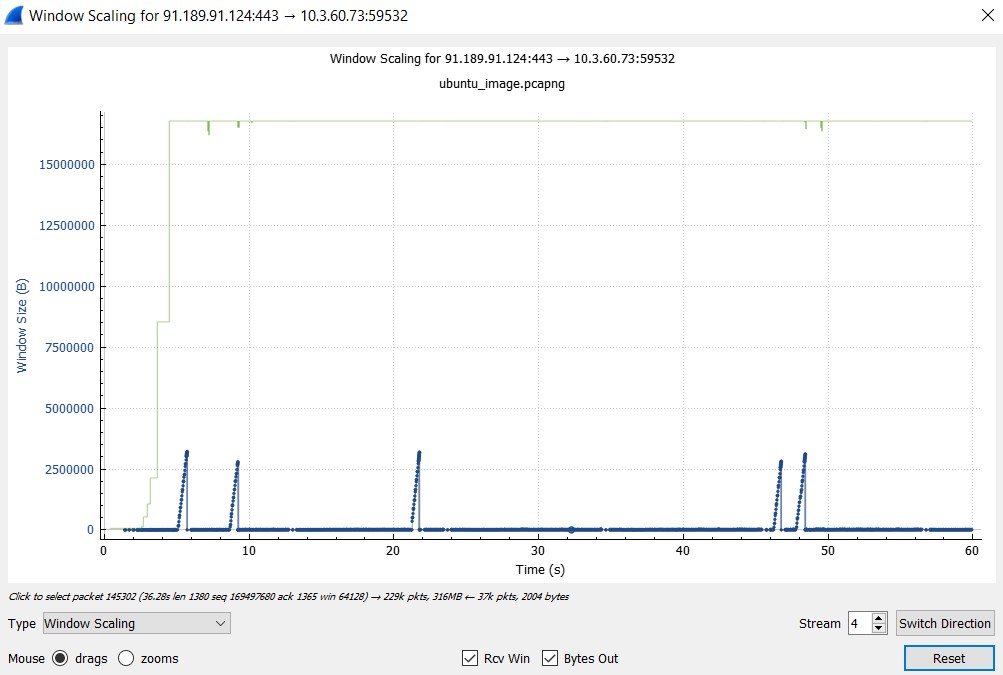


Figure 1.7: Receiver congestion window advertised over time for internet

**For intranet**

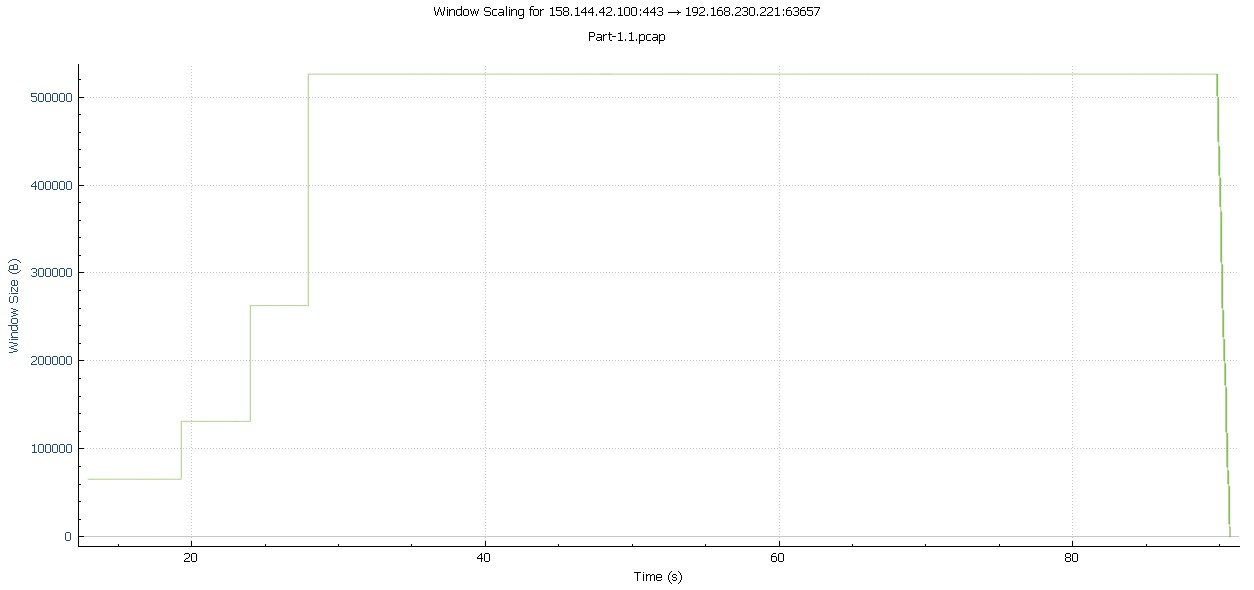


Figure 1.7: Receiver congestion window advertised over time for INTRANET

**Inference** We can see that as the size of congestion window decreases congestion increases. If packet is lost then the size is reduced to zero.

1. **statistics→ IO graphs**

Use display filter: tcp.analysis.duplicate\_ack\_num==1

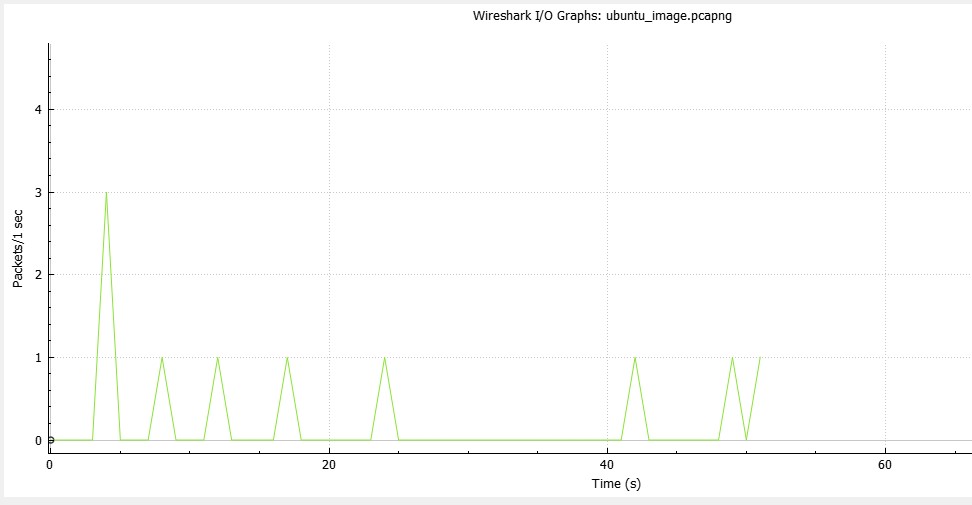


Figure 1.8: 1-duplicate ack

Use display filter: **tcp.analysis.duplicate\_ack\_num==2**

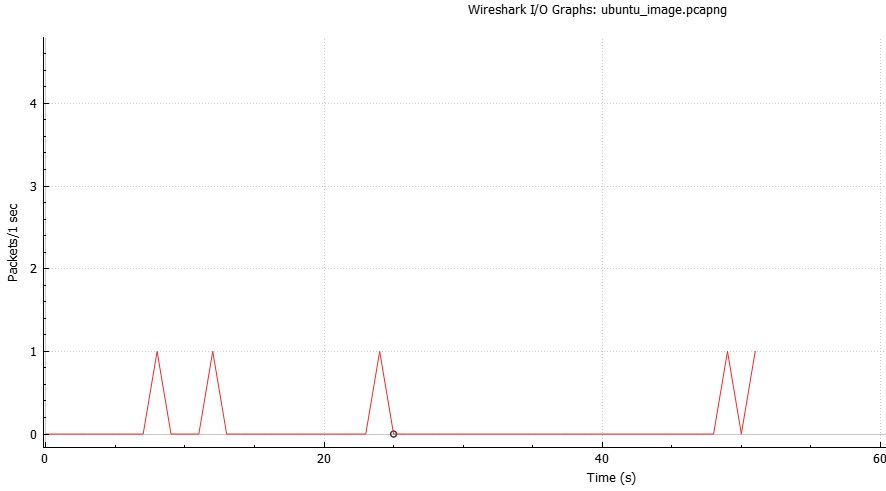


Figure 1.9: 2-duplicate ack

Use display filter: tcp.analysis.duplicate\_ack\_num==3

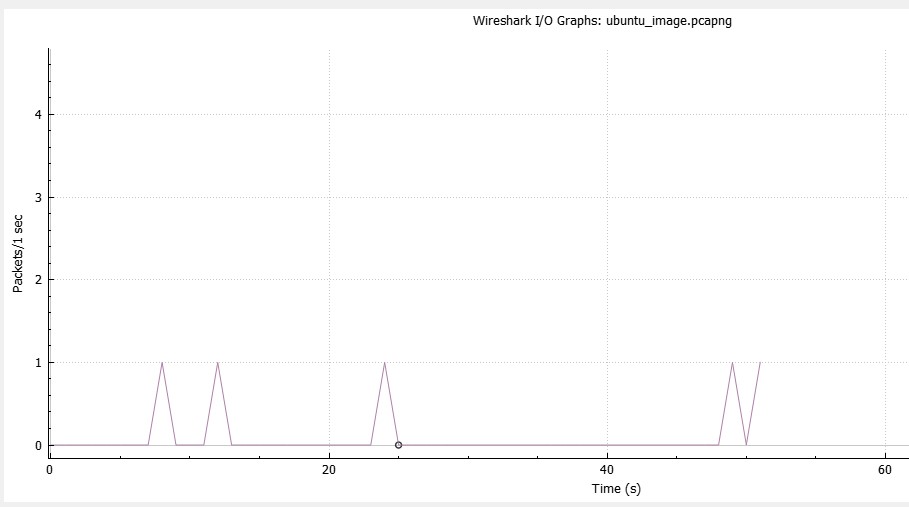


Figure 1.10: 3-duplicate ack

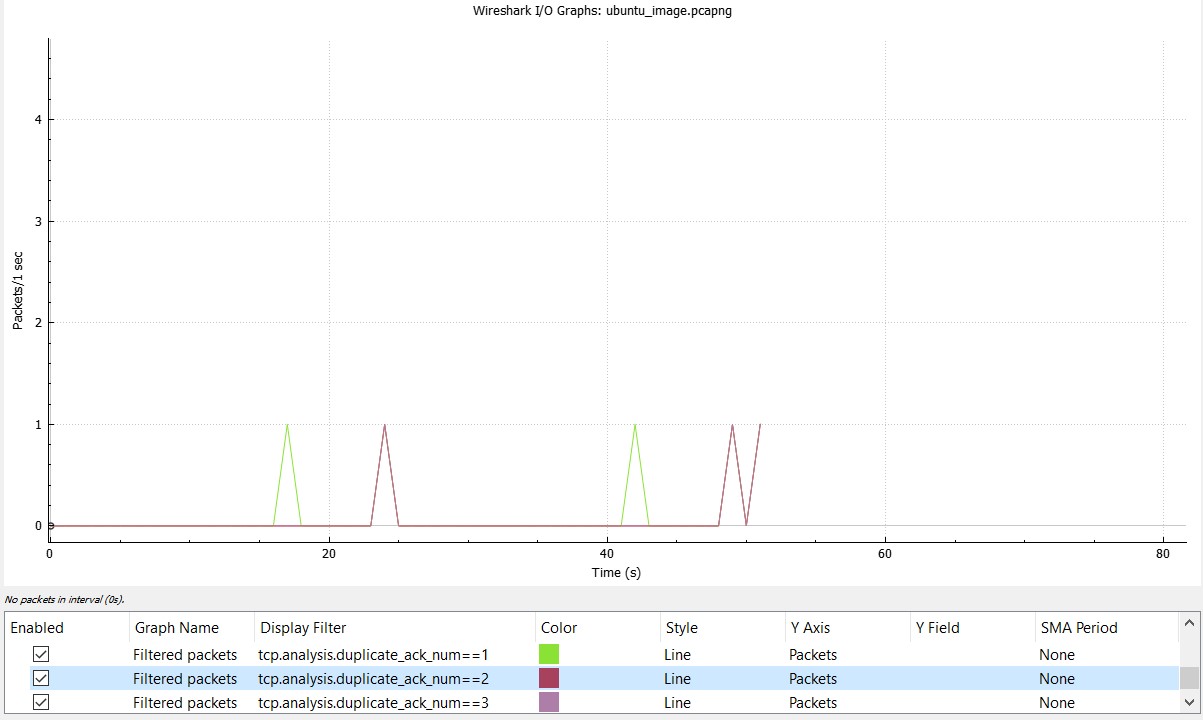


Figure 1.11: 1,2,3-duplicate acks

# Inference

We can infer that the order of probability for duplicate acks is *P*(1) ≥ *P*(2) ≥ *P*(3)

2. A file of size 21 mb was downloaded.

The 3 -way handsake is shown below no. 1163->1172->1173

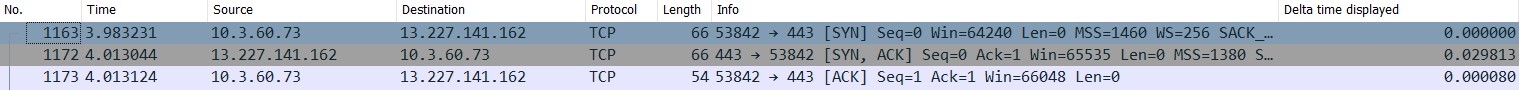


Figure 1.12: TCP 3-way handshake

# Inference

In the first step the sender sends a segment with syn set to true which implies that the sender is active open and wants to establish a connection. The sequence number here chosen is zero.

The reciever send a segment with syn + ack. ack is the acknowledgment which tells the sender that the packet was received.

At last the sender acknowledges the segment and sends a segment with previous ack + 1 (0 + 1 = 1) and the reciever opens the connection.

**3.** We ping to www.hacktoberfest.com The Ip address 103.224.182.242

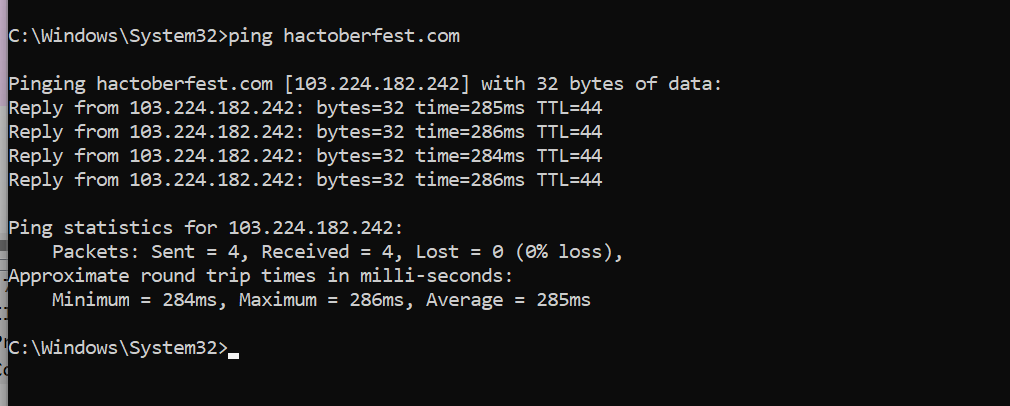


Figure 1.13: Ping to [www.hacktoberfest.com](http://www.hacktoberfest.com)

Using filter ip.addr==103.224.182.242

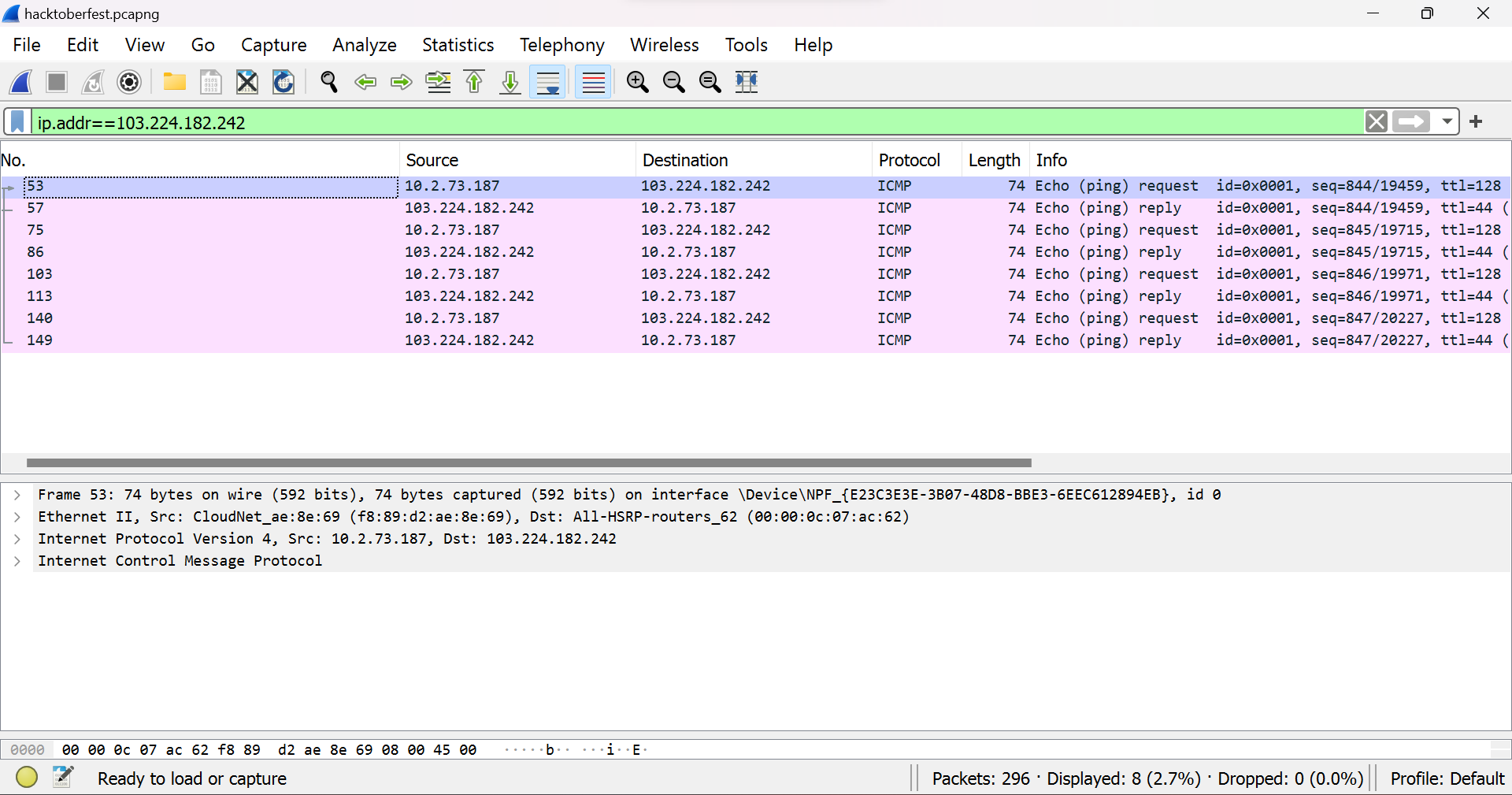
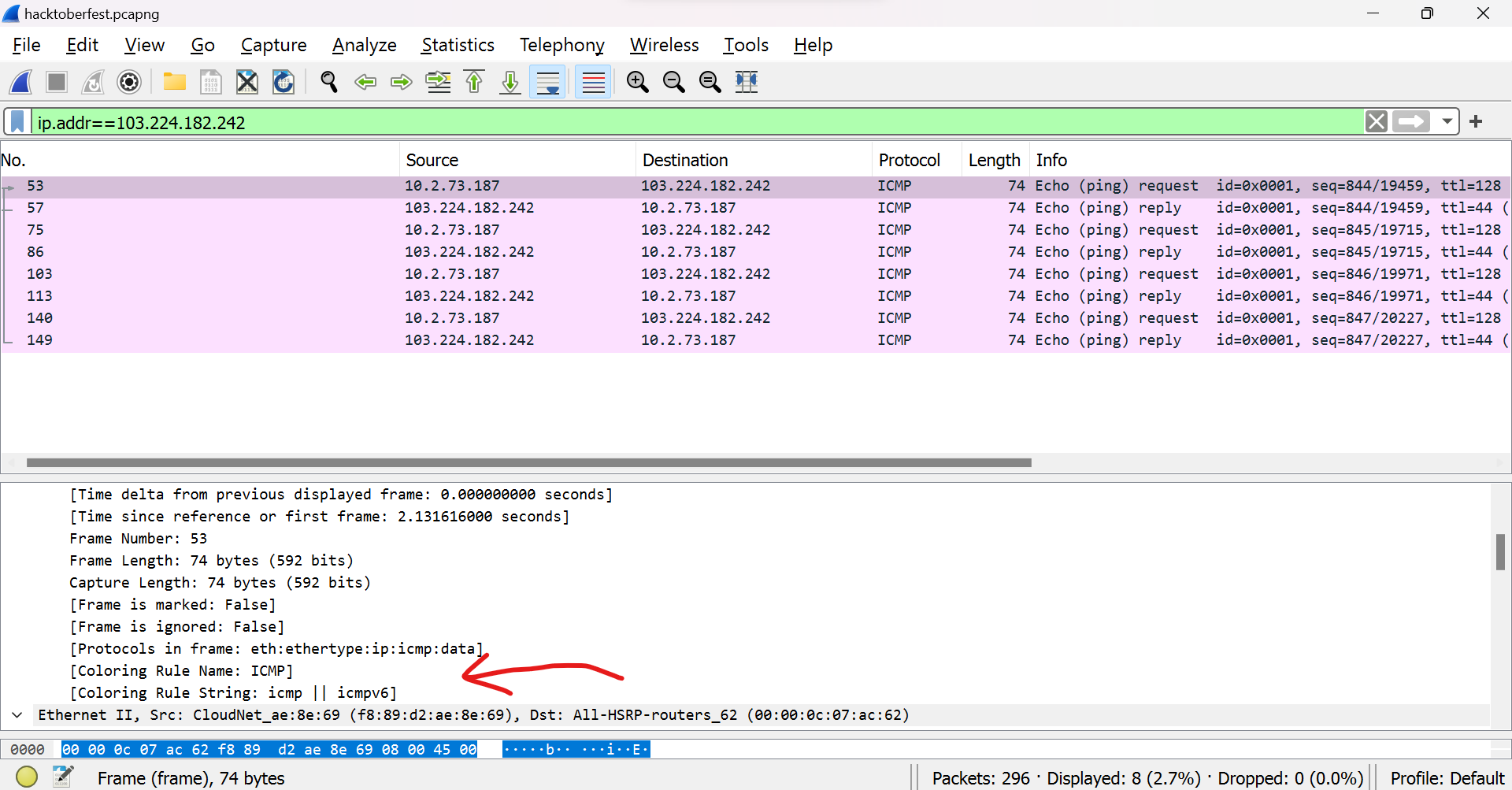


Figure 1.14: Packets captured for ping command

# Ping operates by means of **Internet Control Message Protocol (ICMP)** packets. Pinging involves sending an ICMP echo request to the target host and waiting for an ICMP echo reply.

The packets captured by Wireshark is of ICMP

All details are given in **hacktoberfest packets details.txt**



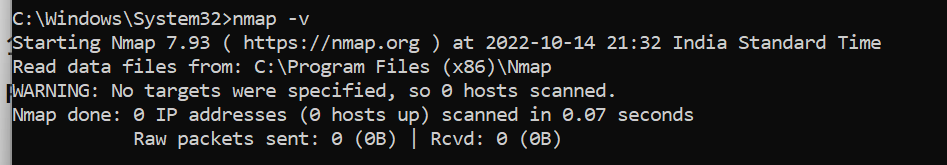
# Inference

ICMP packets are generated.

4.

We have to first install nmap in the operating system . To check weather nmap is installed or not we have to run

”**$nmap -v**”



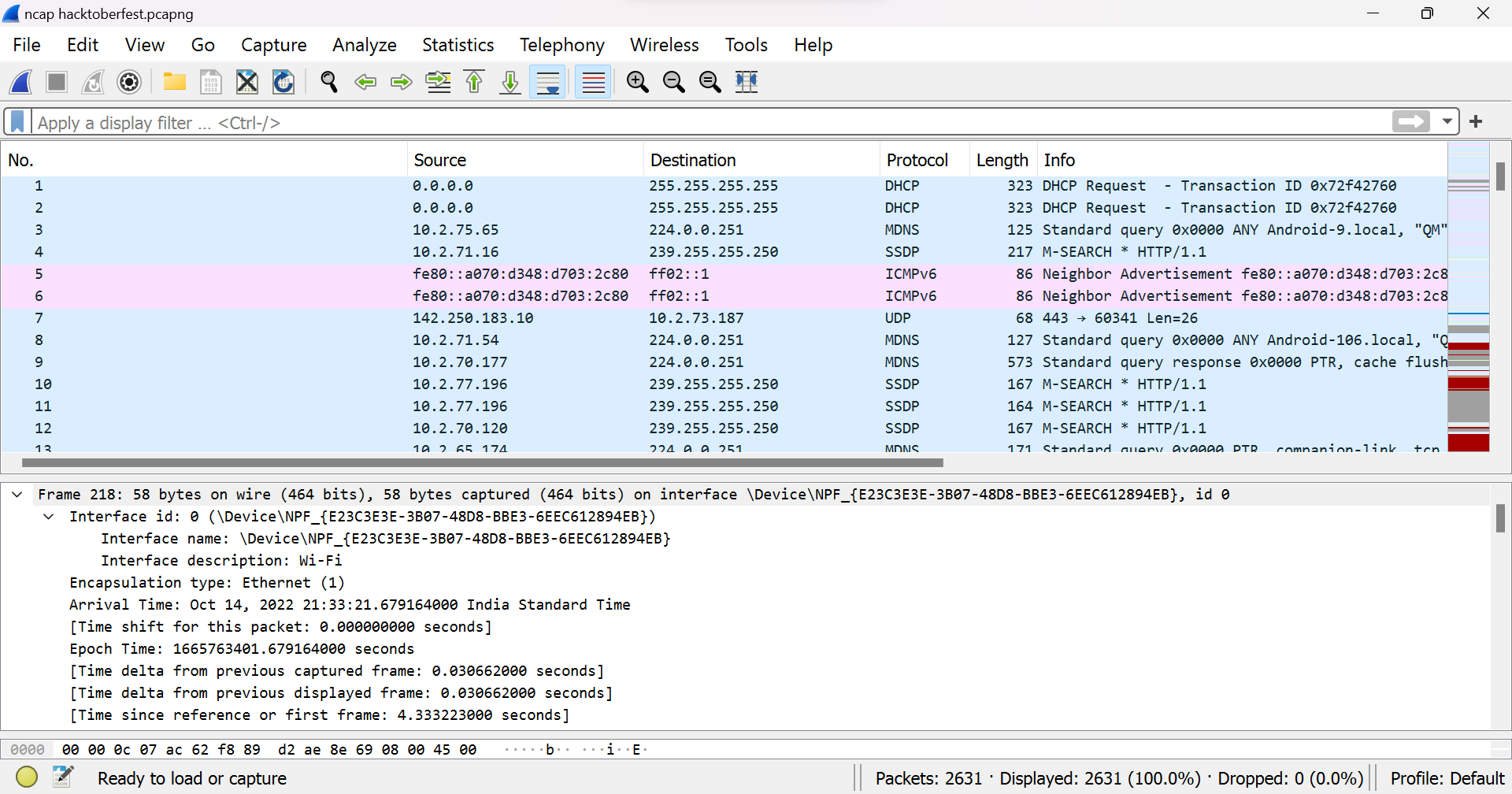
Start Wire shark and run the command ***$ nmap -PS 103.224.182.242***



**Fig showing closed and open tcp ports**

Above fig showing 996 ports closed

* **Open:** Open indicates that a service is listening for connections on this port.
* **Closed:** Closed indicates that the probes were received, but it was concluded that there was no service running on this port.
* **Filtered:** Filtered indicates that there were no signs that the probes were received and the state could not be established. This could indicate that the probes are being dropped by some kind of filtering.
* **Unfiltered:** Unfiltered indicates that the probes were received but a state could not be established.
* **Open/Filtered:** This indicates that the port was filtered or open, but the state could not be established.
* **Closed/Filtered:** This indicates that the port was filtered or closed but the state could not be established.



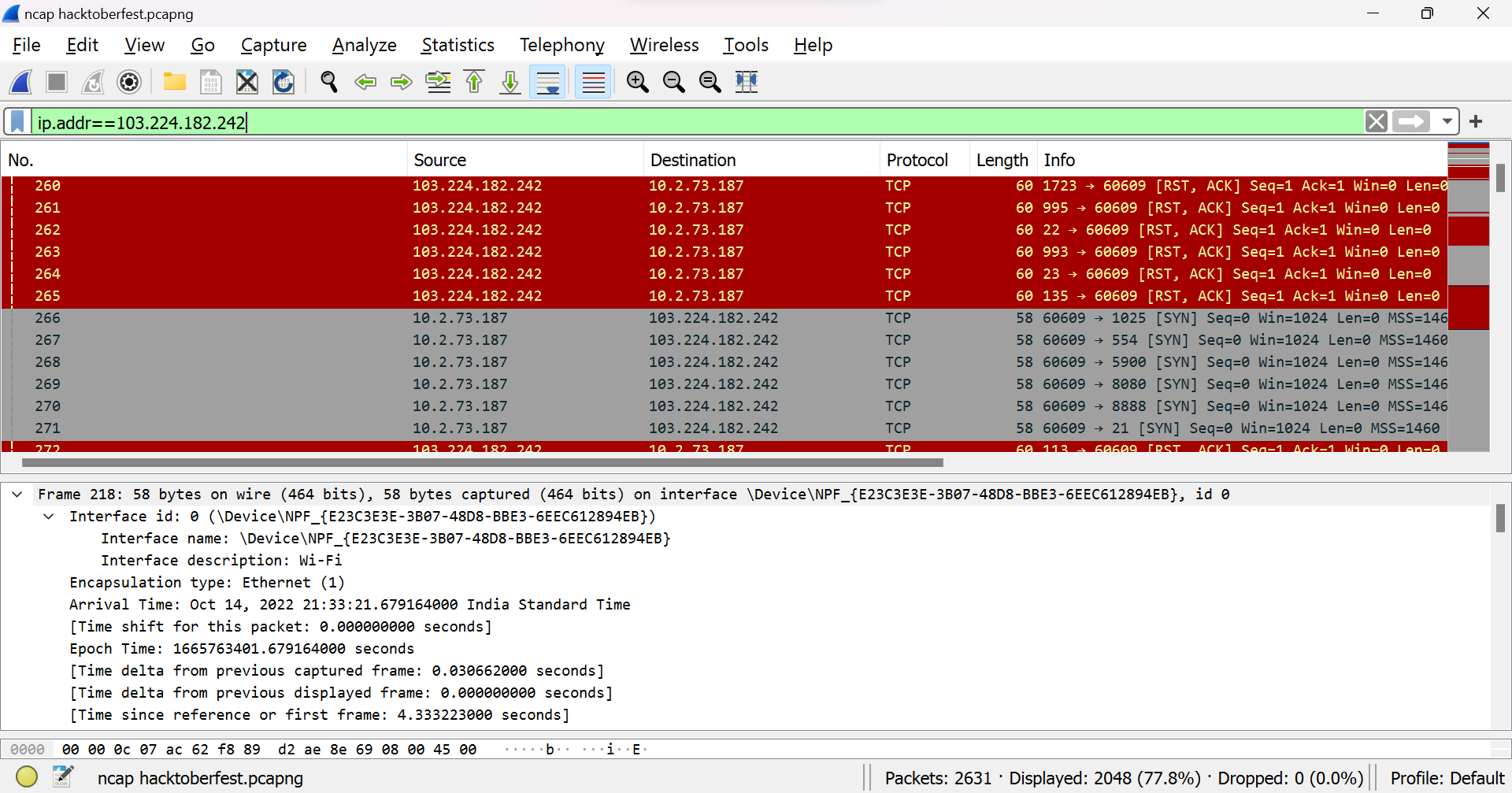
Packets captured by nmap -ps 103.224.182.242

Total 2631 packets have been captured

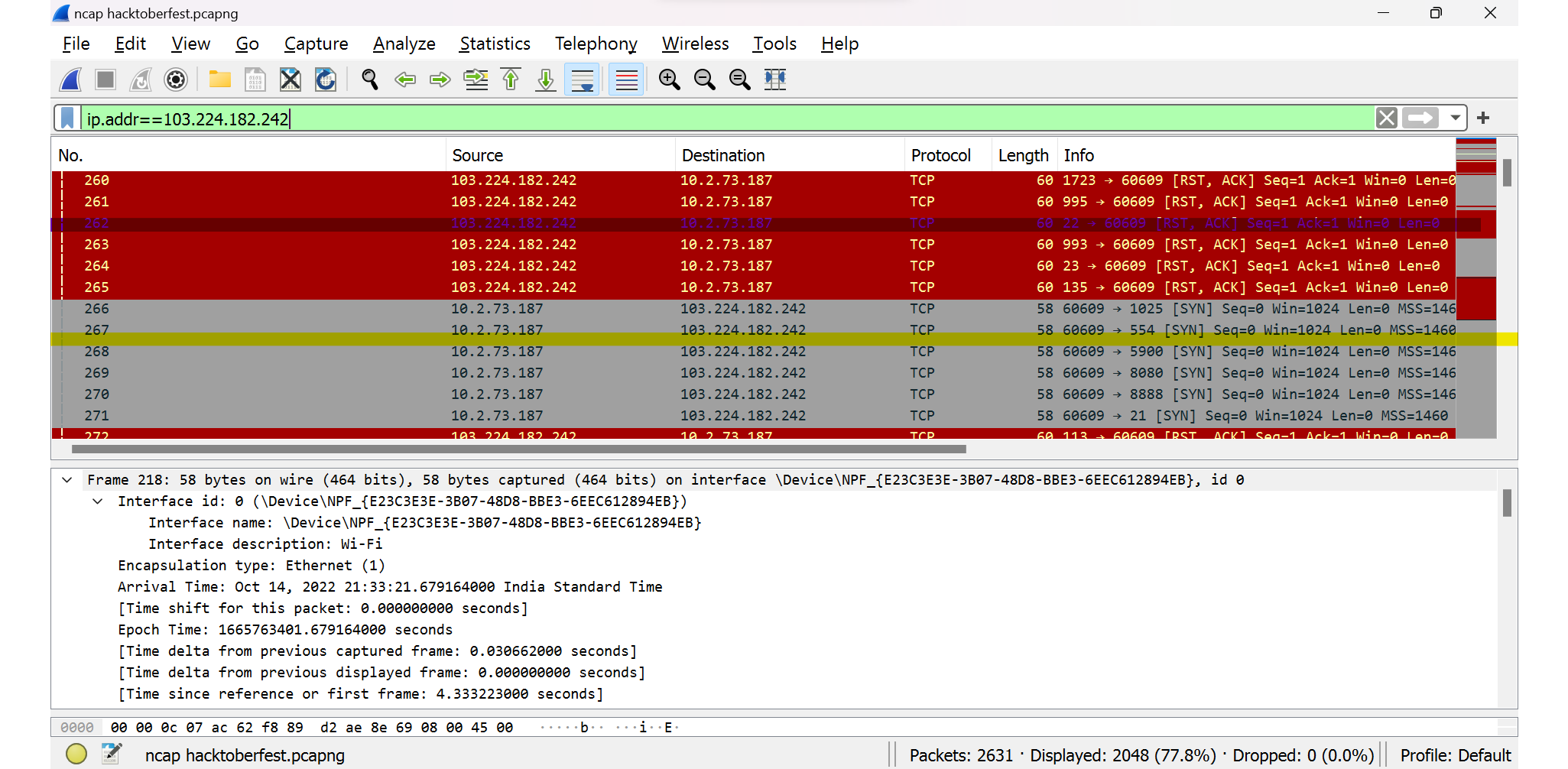
***-P options are used to select different ping methods. When -PS is selected, nmap will check if hosts are online by sending single SYN packet. -Pn will skip this phase and jump right to port scan***

-PS: TCP SYN/ACK, UDP or SCTP discover to given ports

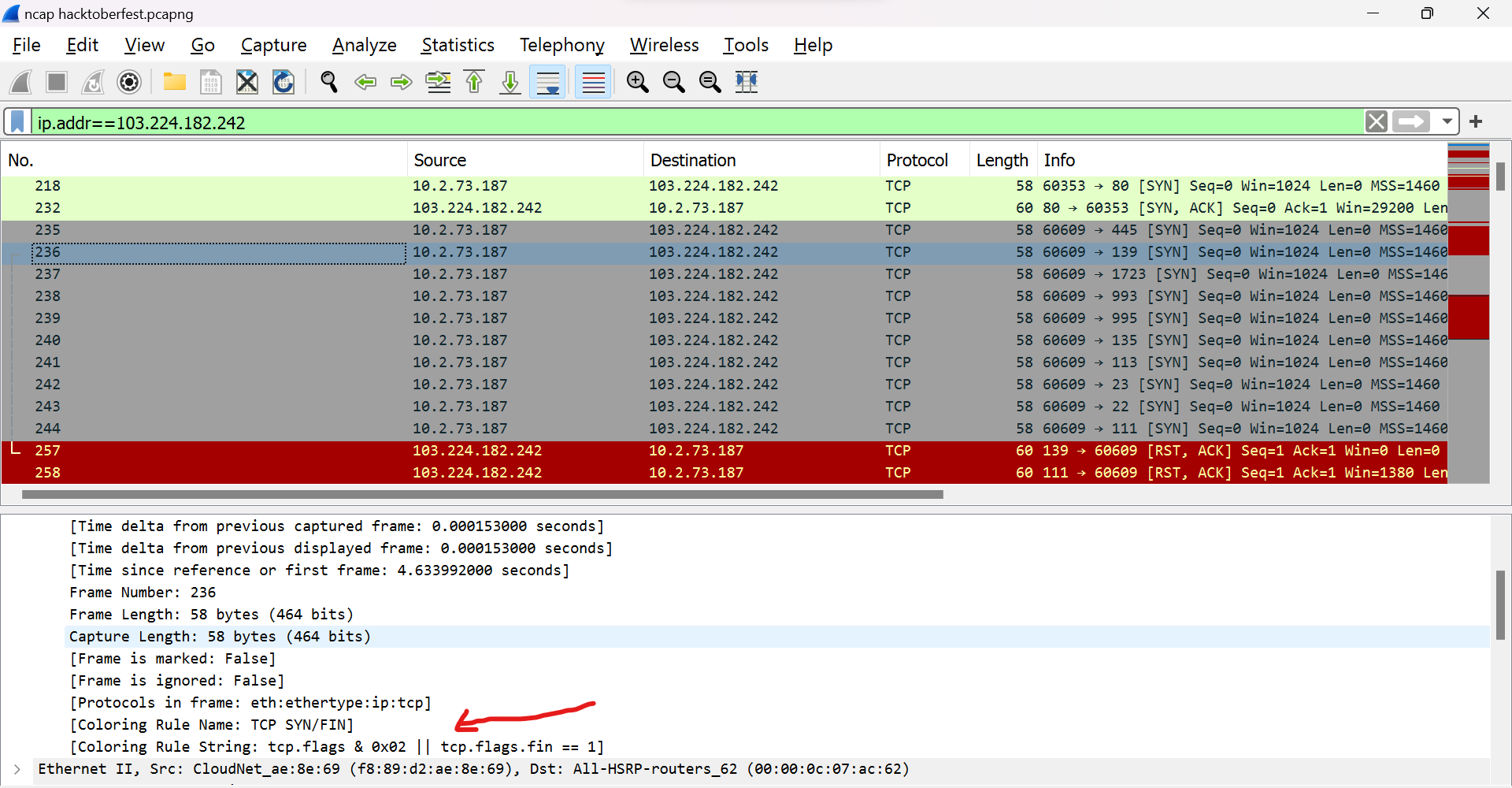
Using filter **ip.addr==103.224.182.242**

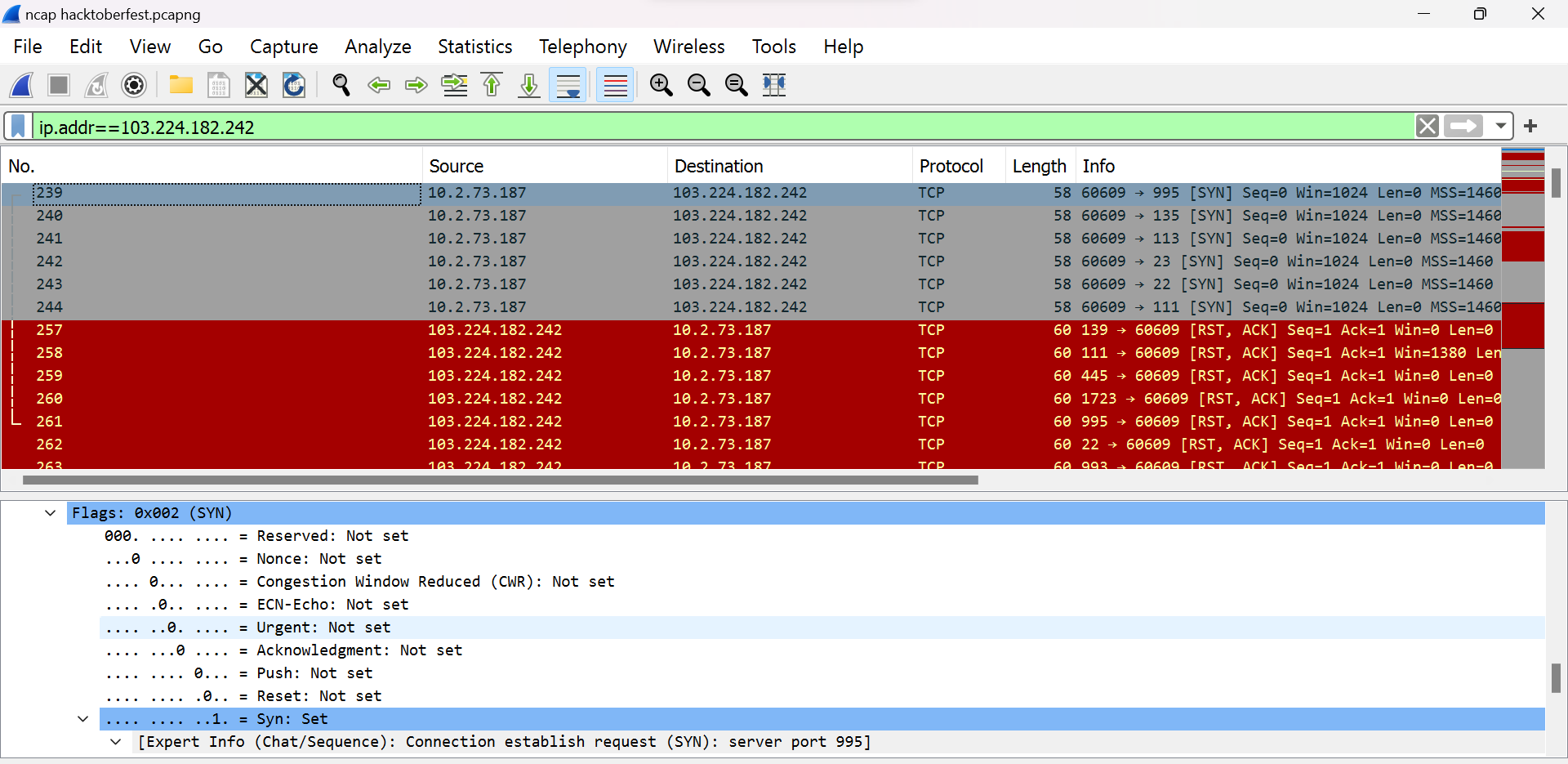


**Nmap receives several SYN/ACK TCP packets** . Many operating systems use a simple counter for this which starts at zero at boot time then increments at a constant rate such as twice per second.



Packtets captured SYN/ACK



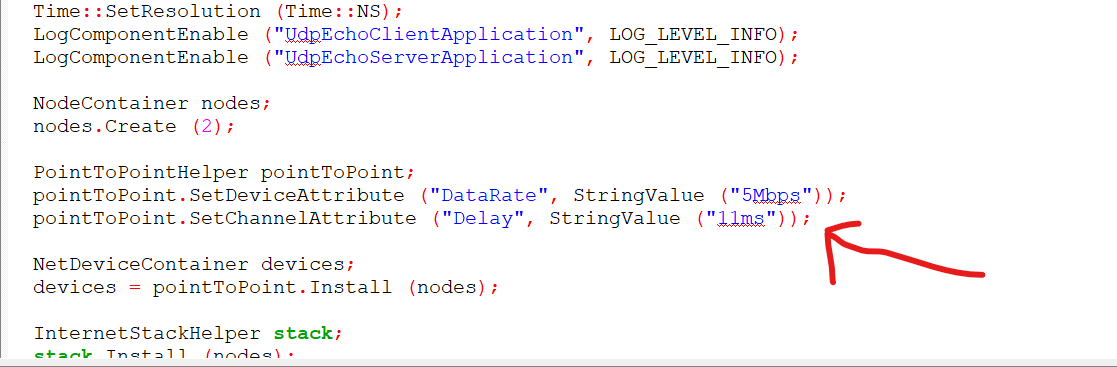


The pcapng file Is given as **ncap hacktoberfest.pcapng**

**Solution of Problem 2**

**A)**

**Open and explore the first\_FM.cc file**  given in lec11 also given in **q2\_1** folder.

****

These line is the code for Delay . Change the value and in each step run ***./ns3 run scratch/first\_FM.cc*** and note the **Total throughput of system.**

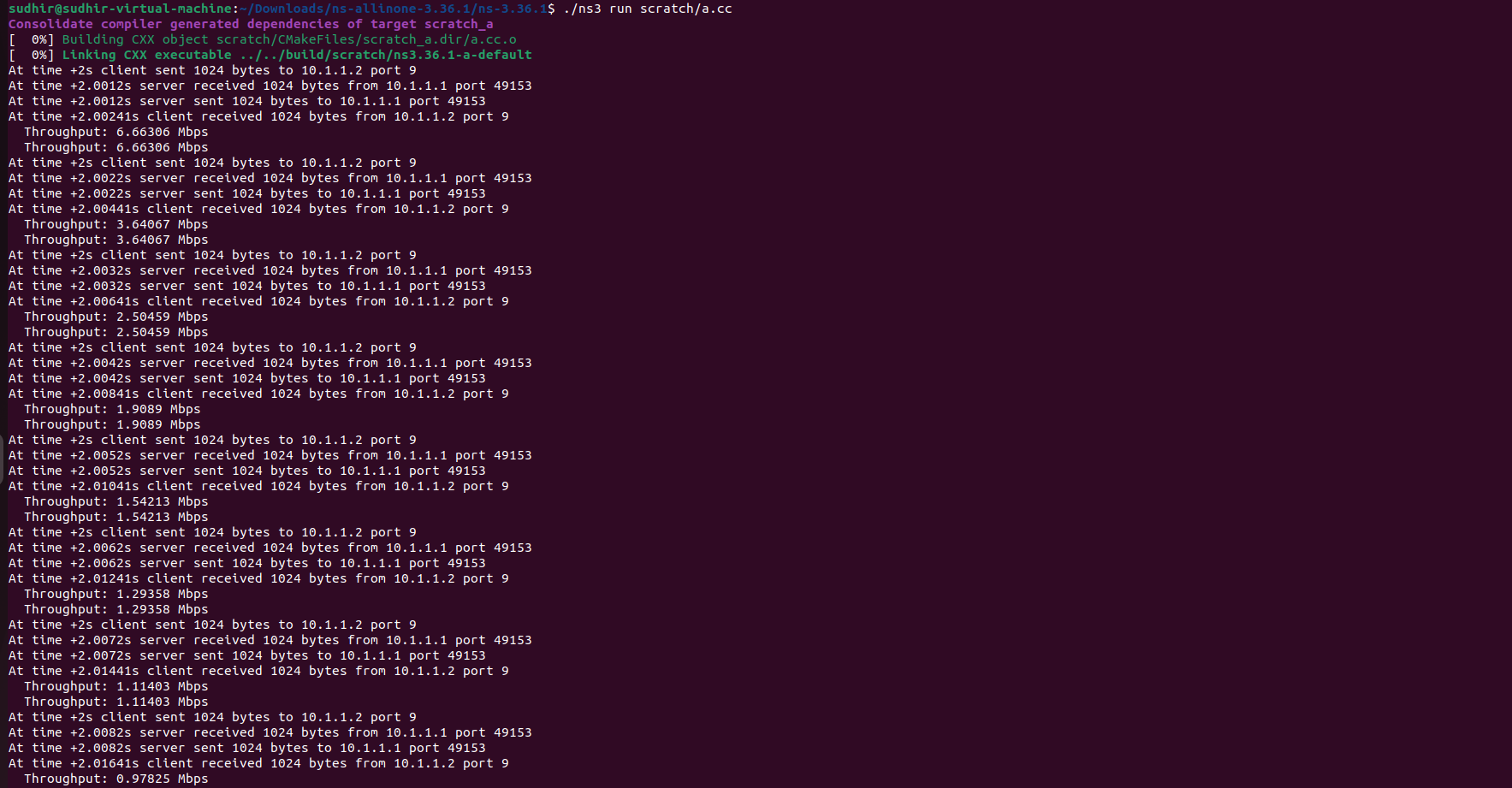
**Throughput determines how much of an object can be delivered over a period of time and delay determines how long it takes to deliver an object**.

**The first\_FM.cc is Renamed as a.cc**  for practice.

I have created a loop inside (delay 1->100) the a.cc for calculating the throughput corresponding to respective delay.

And all the [ delay , throughput ] in the range is stored.

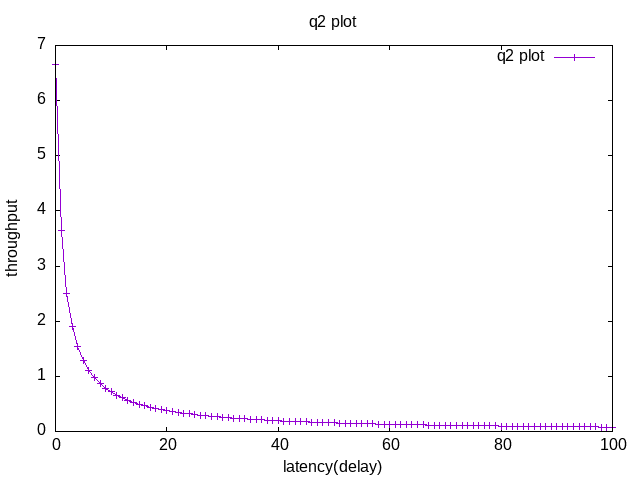
When we run the file ***“ ./ns3 run scratch/a.cc “*** the following out put will be shown.



It shows the throughput with respective delays. At last a **q2.plt**  file will be generated



Plot the **q2.plt** file using **gnuplot q2.plt.** the following graph will be ploted.



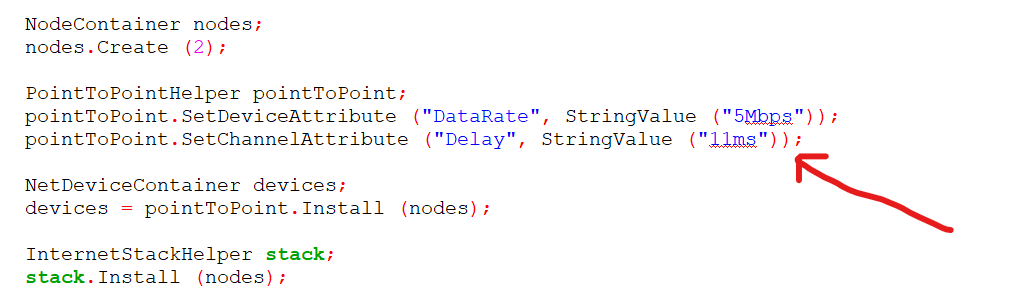
**They are inversely related**

The delay before acknowledgment packets are received (= latency) will have an impact on how fast the TCP congestion window increases (hence the throughput). **When latency is high, it means that the sender spends more time idle (not sending any new packets), which reduces how fast throughput grows**.

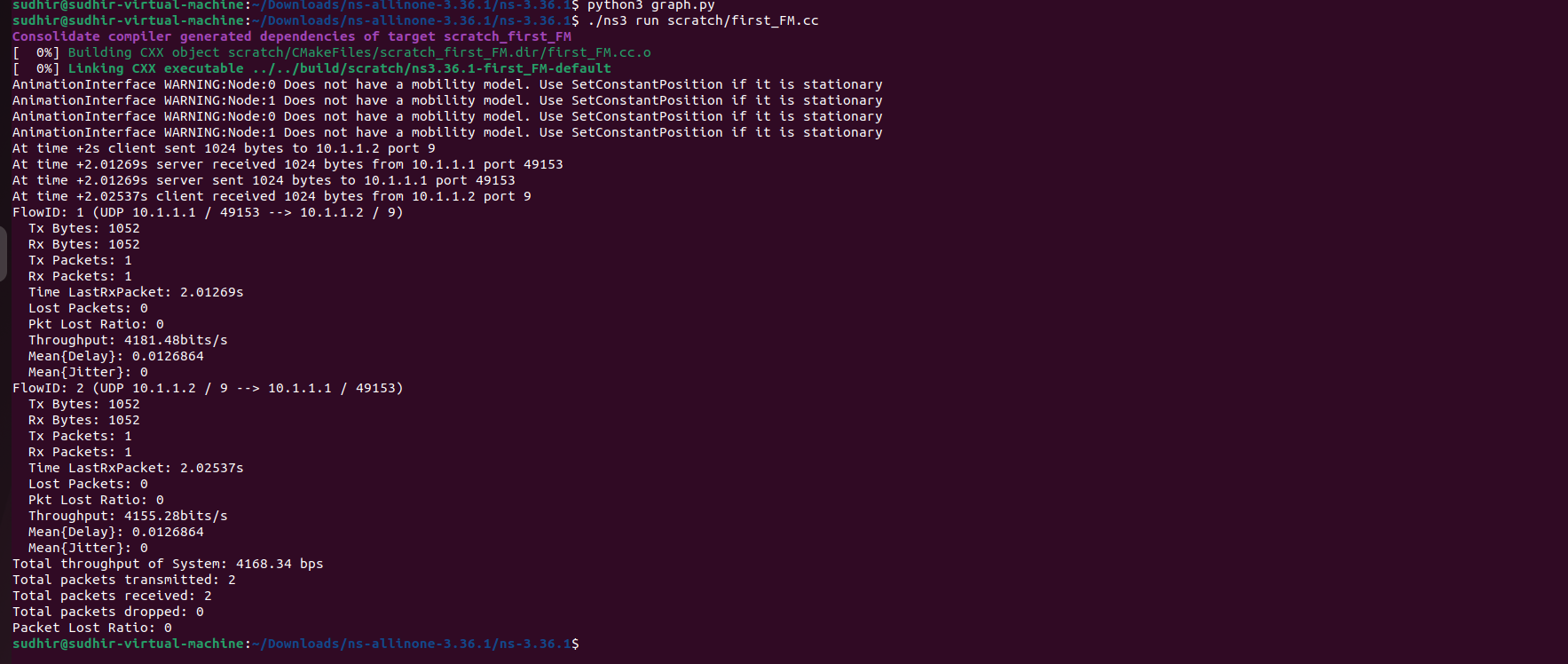
Delay is simply the time taken for a data packet to reach its destination after being sent. We measure network latency as round trips, In other words: throughput is the number of data packets being successfully sent per second, and latency is the actual time those packets are taking to get there. So, the terms are related - they both relate to data transfer and speed. They’re basically two sides of the same coin, but still different metrics. You want to maximize throughput but minimize latency

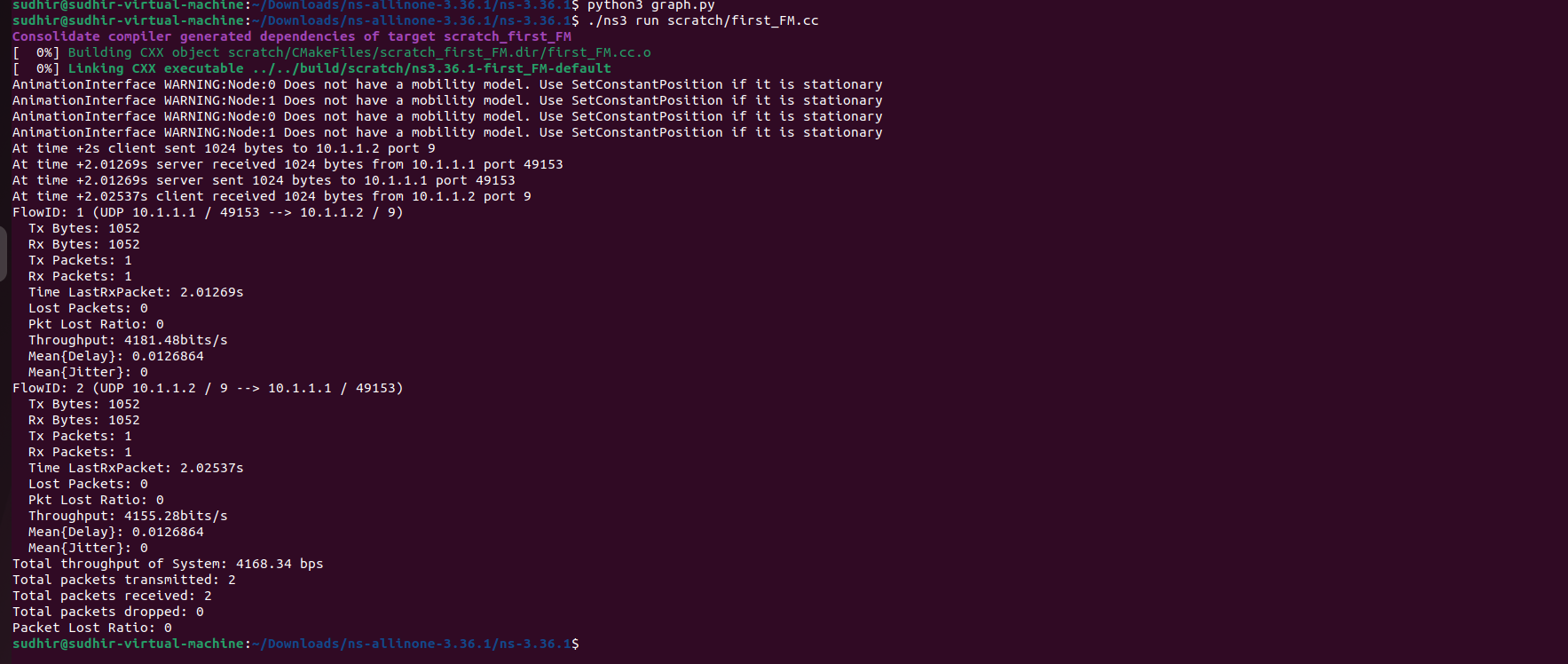
.

Or we can manually just enter in the file and change the



Change manually and note the **delay and througput** and plot the graph .





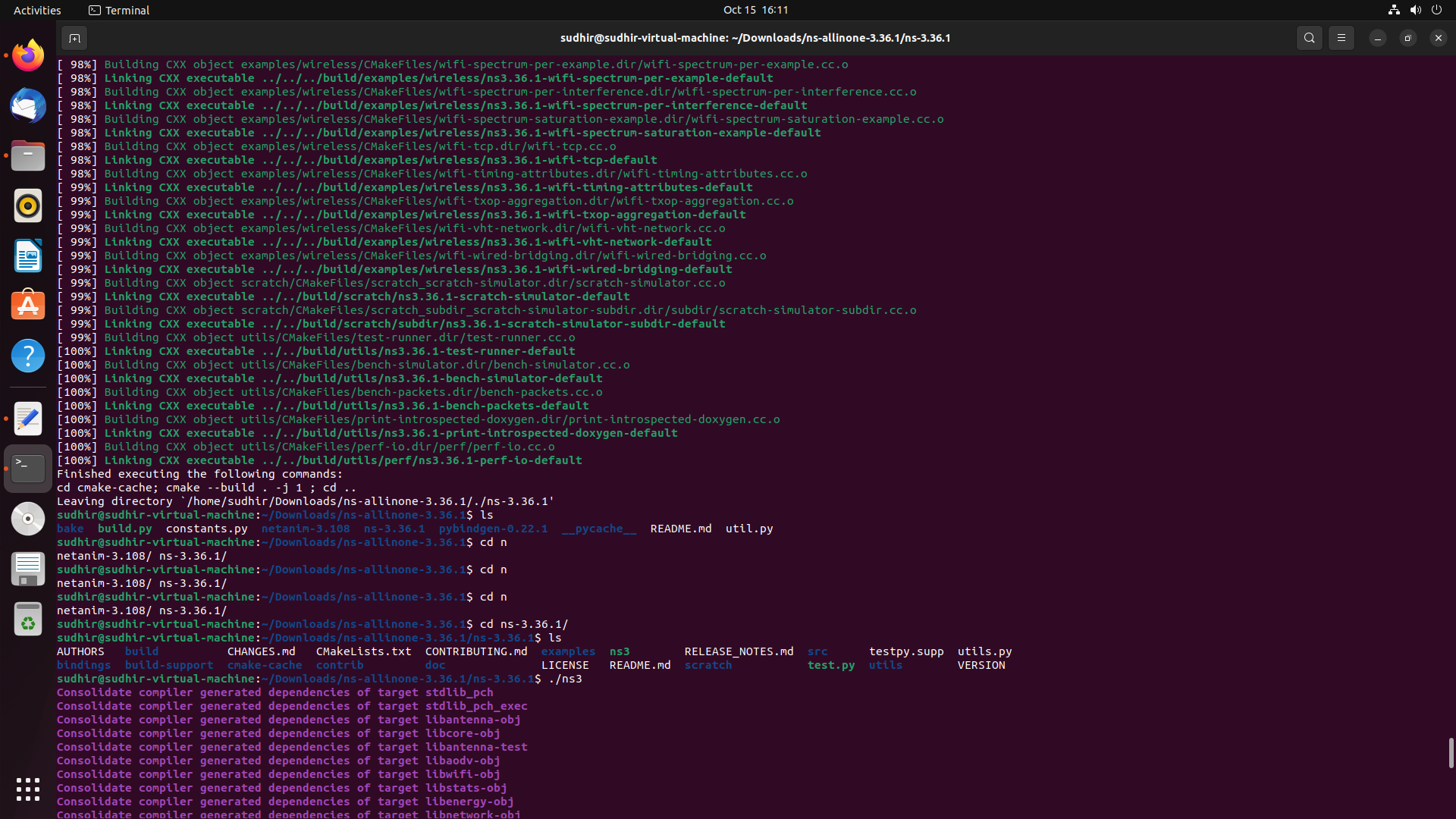
The python file for Graph is given as **graph.py**

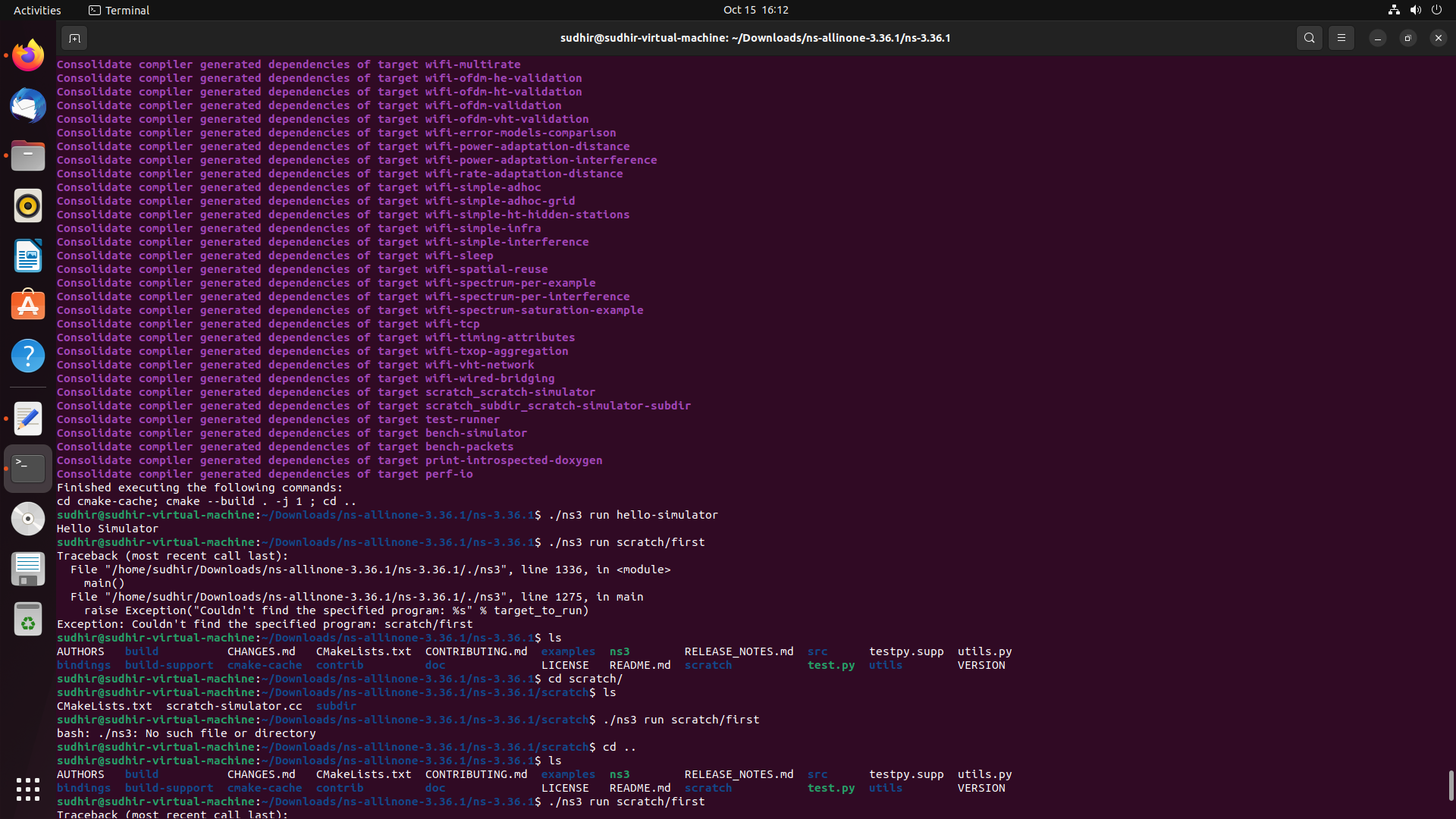
**B)**

For these questions we have to install ns3 [installation](https://karimmd.github.io/post/tutorial/ns3_installation/).

After installation check weather its installed correctly or not.

Explore the **tutorial scratch**  and **examples**  in the folder build after installation of ns3.





You can also run ***./ns3 run hello-simulator*** for conformation

Copy the **first.cc** located in **examples/tutorials** to **scratch** folder.

Run the following command for checking ***“ ./ns3 run scratch/first.cc***

*Building successful indicates that headers were enabled and many more libraries installed.*



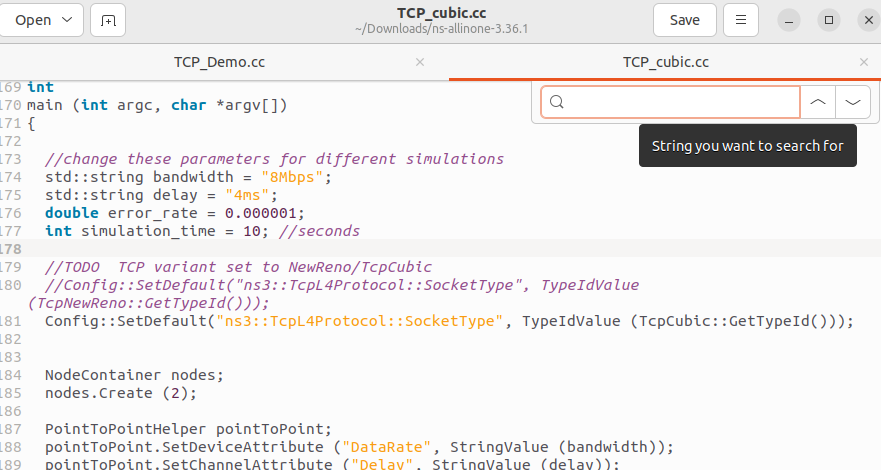
End on confirmation of ns3

**NOW**  two files have been provided to us **TCP\_demo.cc** and **CW.plt**.

Explore the file TCP\_demo.cc

Here we can see the code for TCP variant (NewReno and CUBIC).

These Is currently in TCPCubic (the uncommented line)



**Code for TCP\_cubic**

1 ) Put the file in **scratch** folder and remane as **TCP\_Cubic.cc**

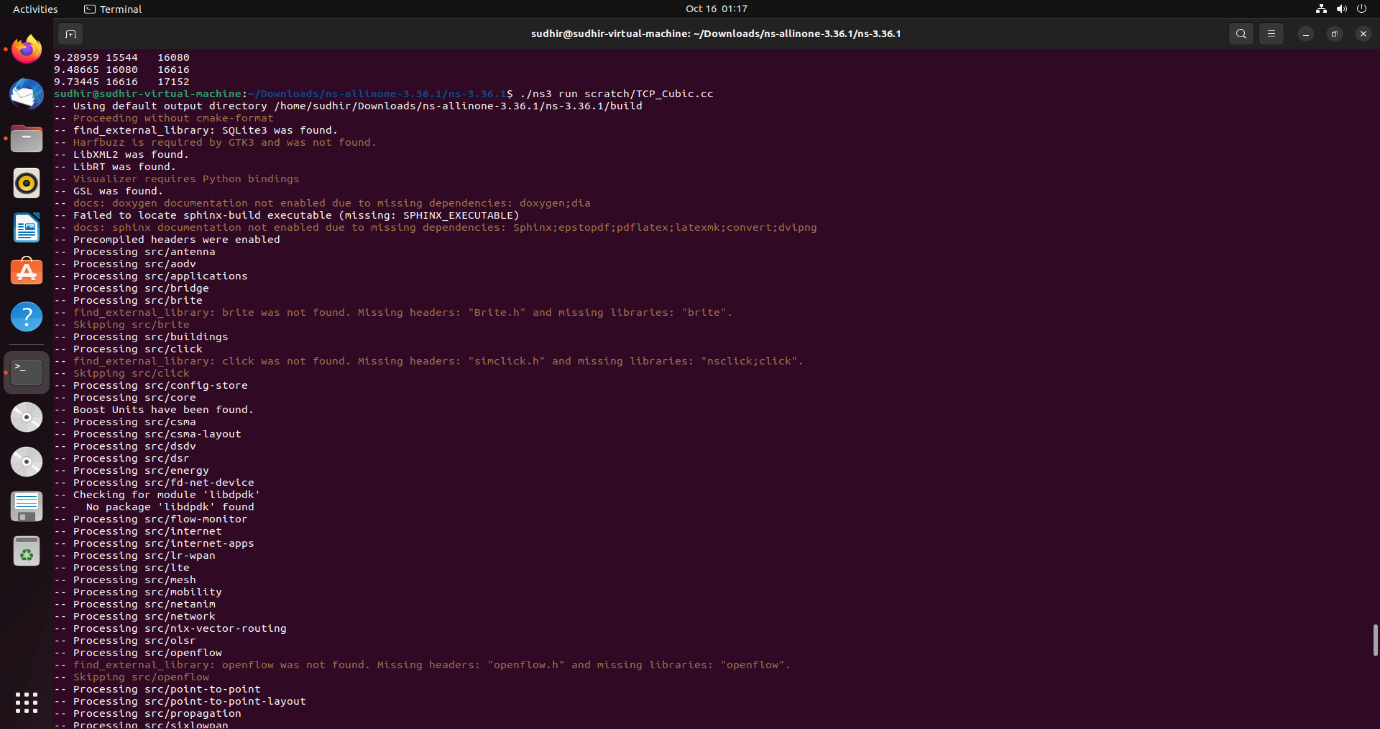
2) run the following command ***”./ns3 run scratch/TCP\_Cubic.cc***

3) we can see that some new file have been created like **TCP\_Cubic.cwnd** file some **pcap** files given in the folder.

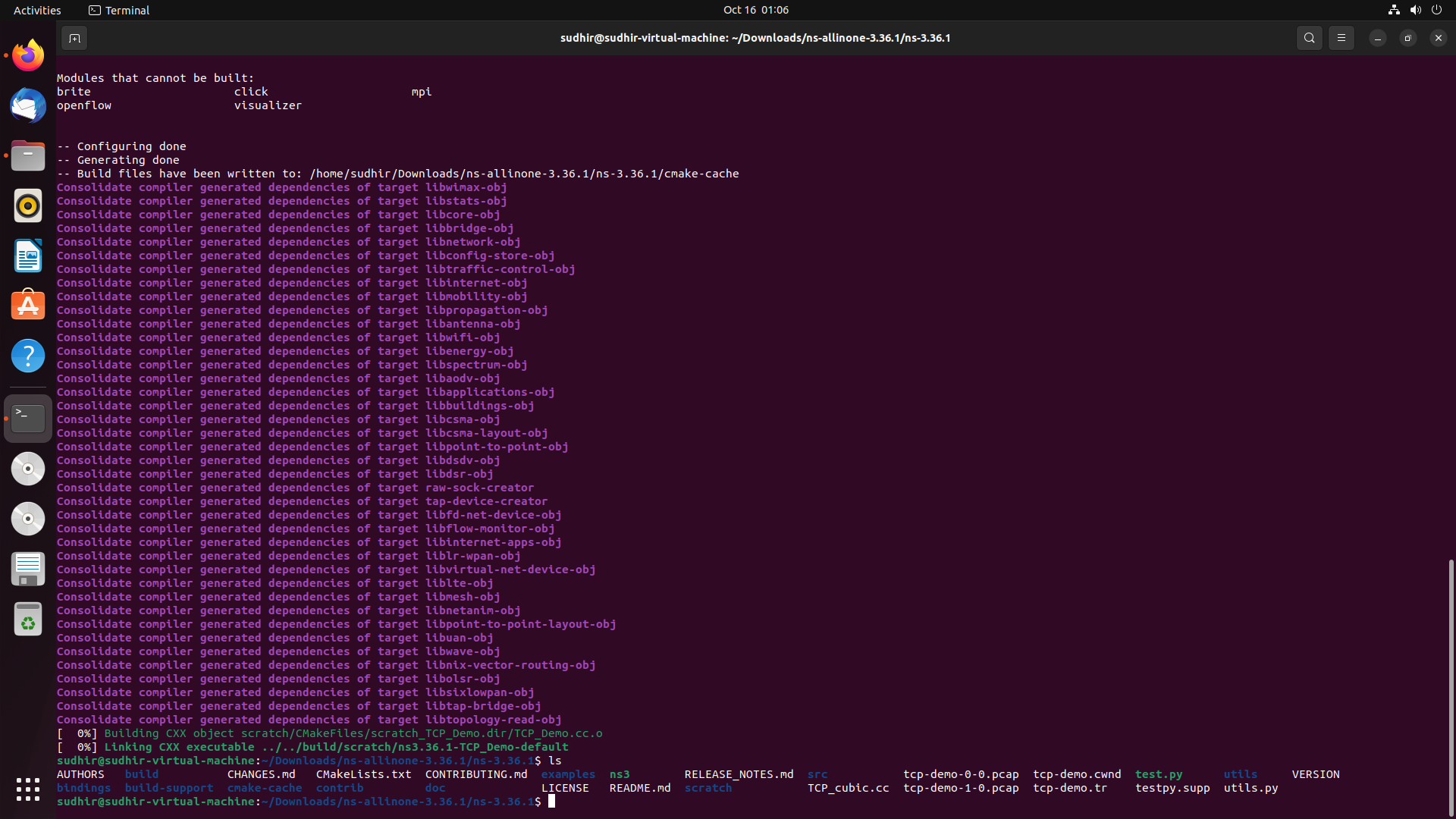
4) now add the **CW.plt**  file in the folder where **TCP­\_Cubic.cwnd**. is generated and then

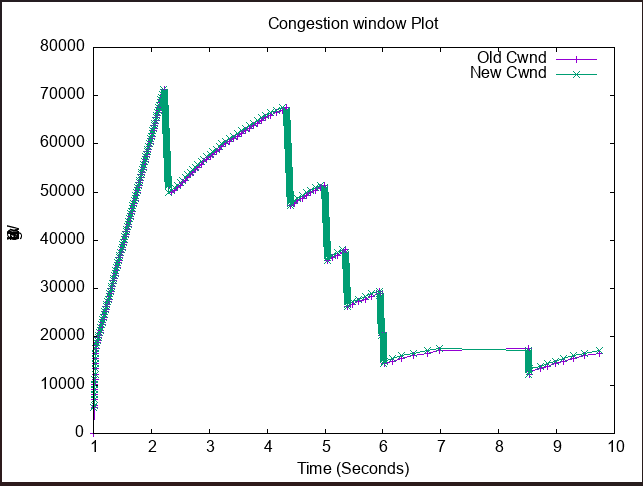
5) run ***“gnuplot CW.plt”*** (to install “ sudo apt install gnuplot “) .

6) run “ls” we will able to see the **CW.png** file .



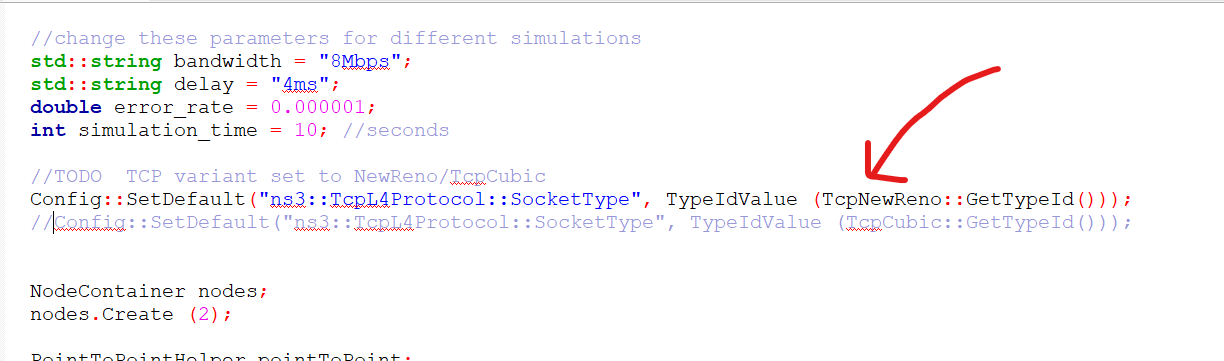
Respective file generated throughout the process of TCP\_cubic id given in **TCP\_Cubic**  folder.





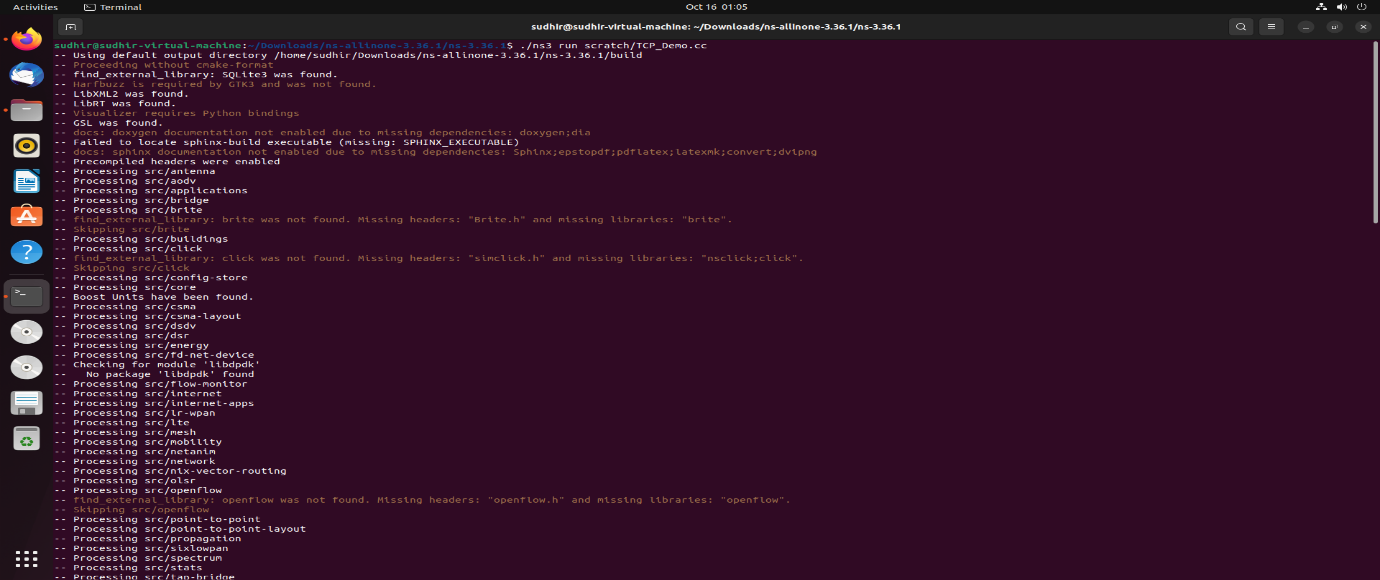
TCP­\_cubic

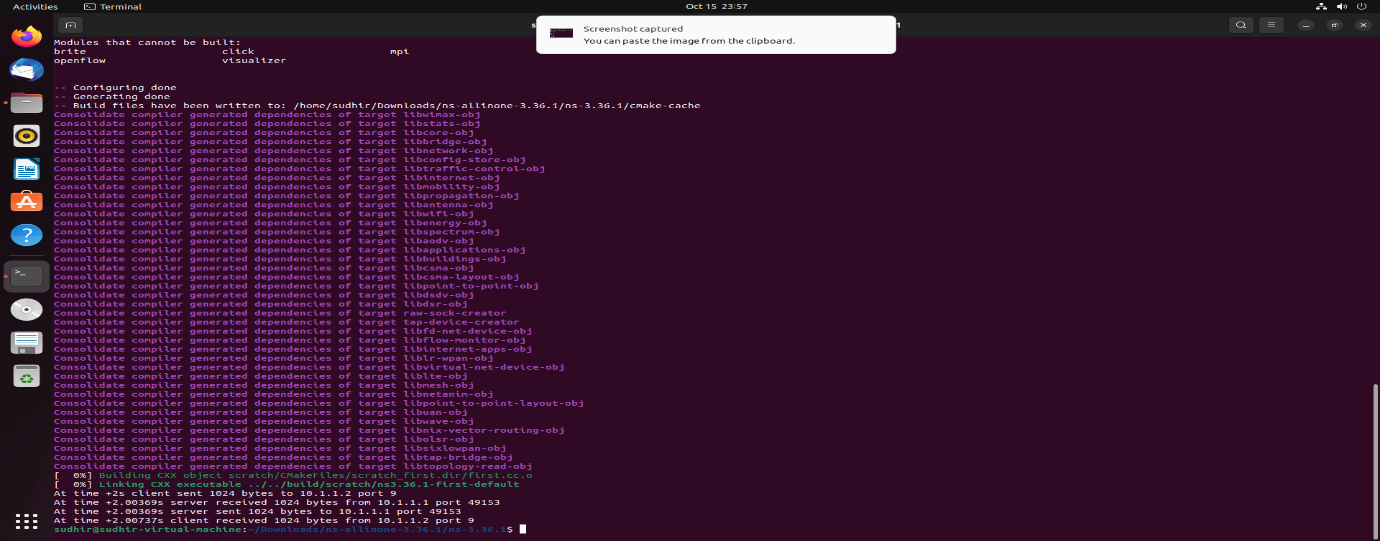
**IN CASE OF TCP NEW Reno we have to uncomment the line of code for tcp reno in TCP\_Demo.cc file.**

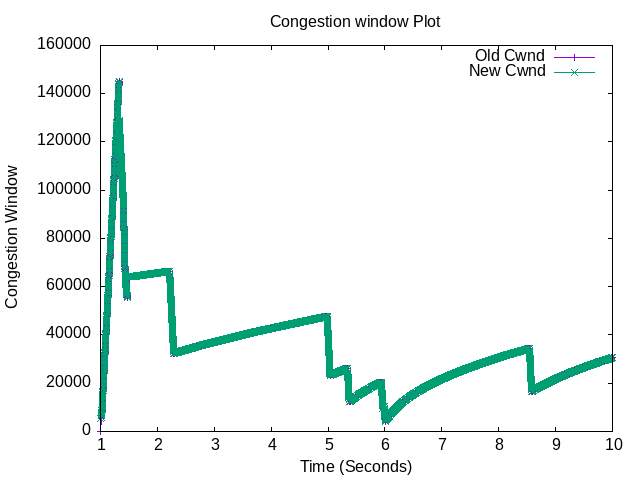
****

**Code for TCP\_NewReno**

Then Do the Same procedure as explained in case of TCP­\_Cubic (Explained above);  **TCP\_demo.cc==TCP\_reno.cc**







Congestion window TCP\_NewReno

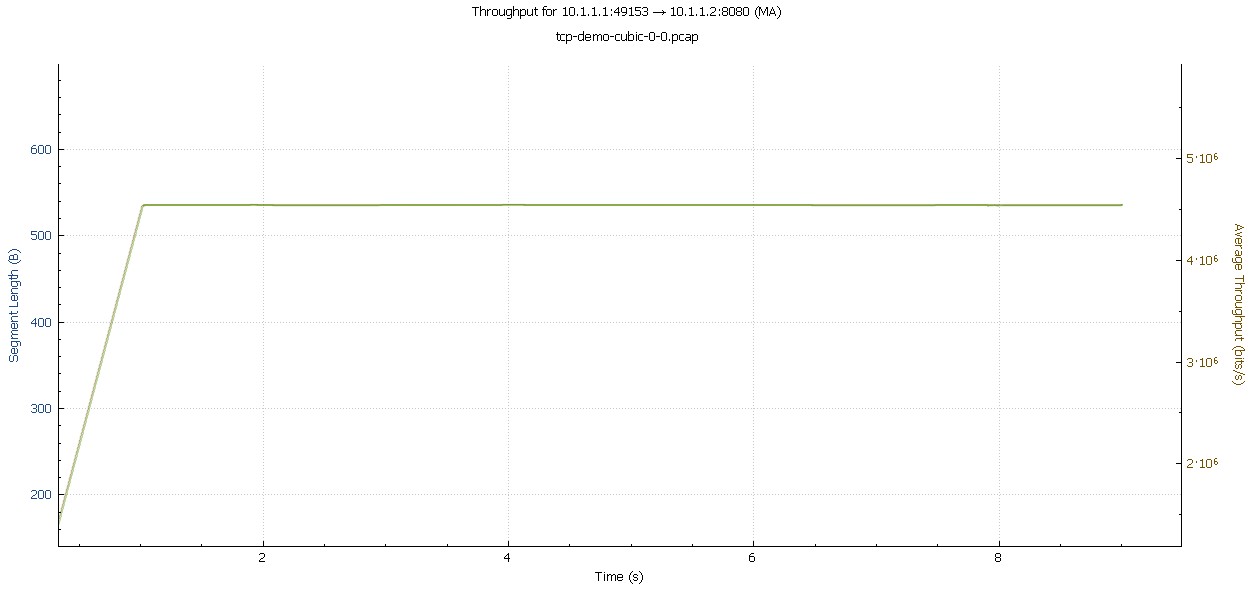
Respective file generated throughout the process of TCP\_NewReno id given in **TCP\_Reno**  folder.

**Now,**  There are some **pcap file**  is, given in respective TCP Variant Folder. Now open the tcp-demo-**cubic/newreno**-0-0.pcap file in wireshark. And find the throughput. Shown below—

**Or we can also increase the values of bandwidth , delay and error rate**

**Answer the following questions for both the TCP variants:**

1. In both cases the window is reduced by 6 times. All of these times it is packet loss and duplicate ACKS. Thiscan be verified from the pcap files generated by the simulation. Go to the time instances where the reduction happens and you will see dup Acks in that region in pcap file. Note that pcap file starts 1s late so a reduction at t=9s in graph corresponds to t=8s in pcap file.
2. Comparing the CWND the New Reno is agressive and gets a very high CWND initially but then drops ofquickly. While Cubic is less aggressive so it reaches less initially but due to the stability it maintains quite a steady CWND. The average CWND is larger in case of Cubic. From the pcap the throughput and goodput graphs can be obtained.



TCP Cubic Throughput and Goodput

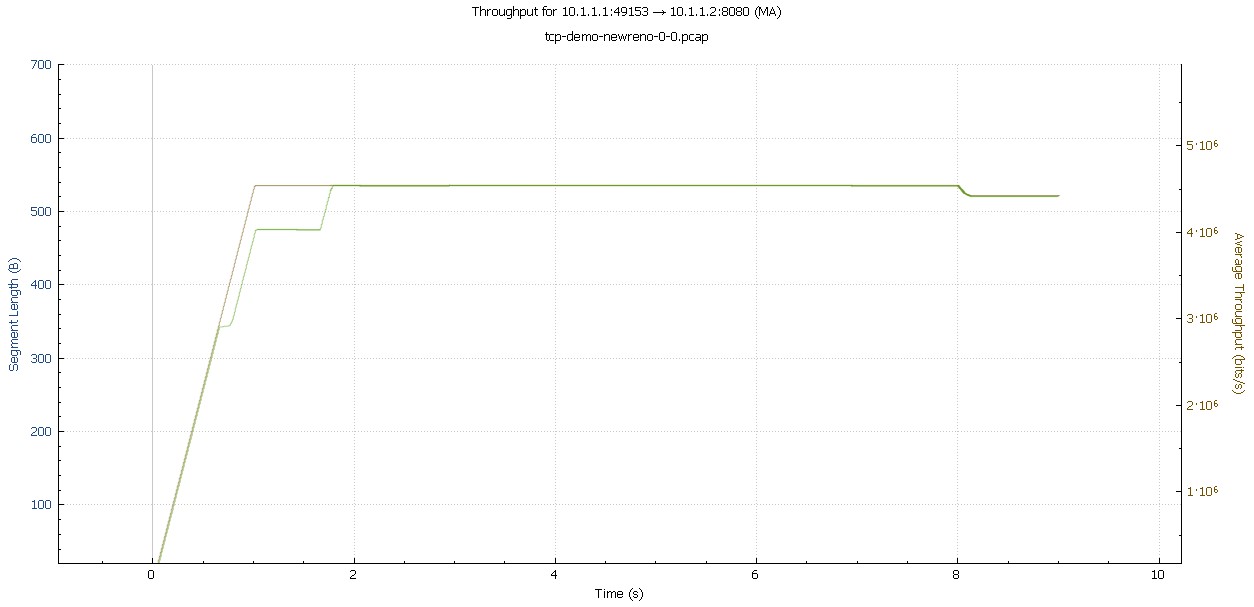
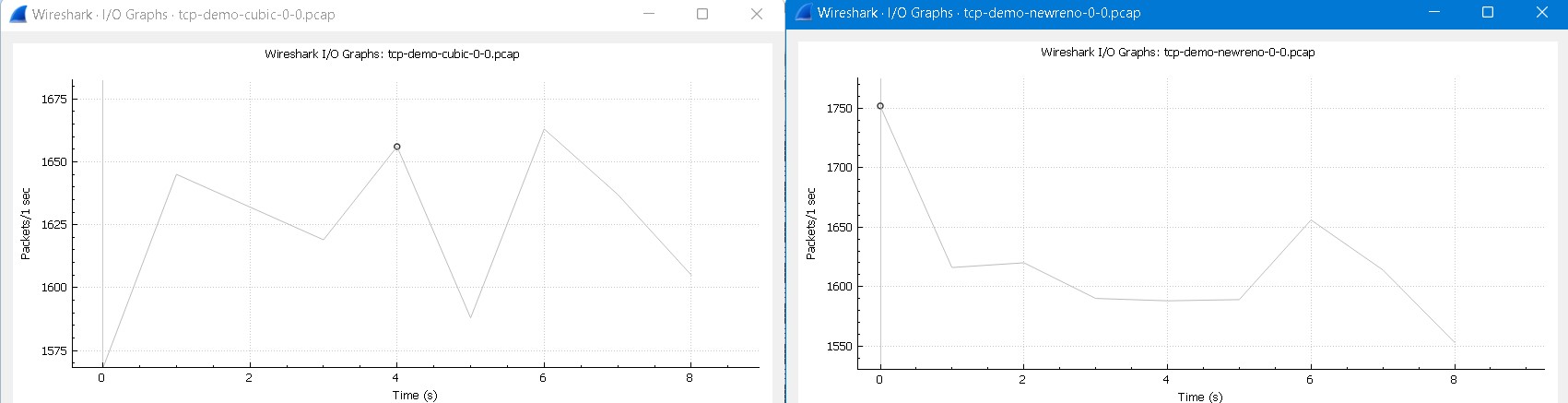


Figure 2.17: TCP New Reno Throughput and Goodput

Though there is not much difference it can be seen that for New Reno there are places where some decrease is observed.



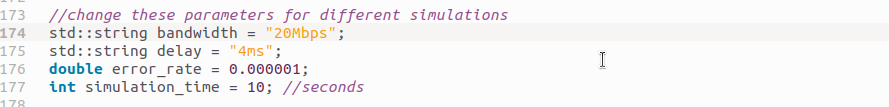
TCP Cubic Throughput and Goodput

The packets/s is also better on average for Cubic than New Reno.

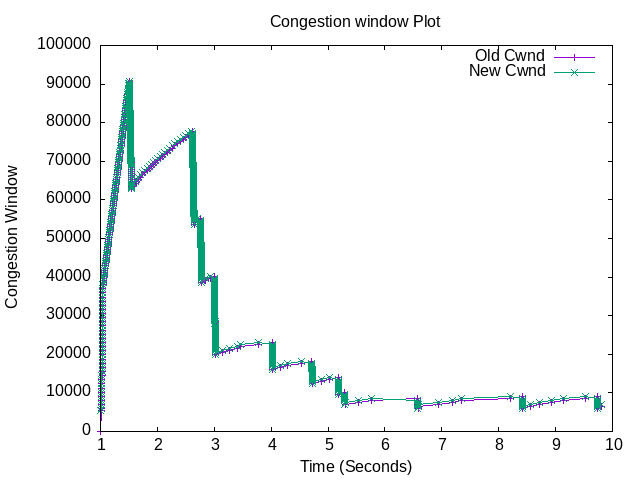
Hence I conclude that Cubic is better than New Reno.

3)

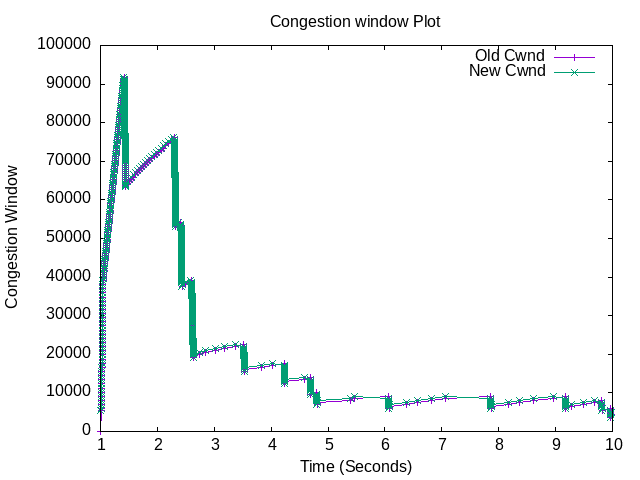
a)



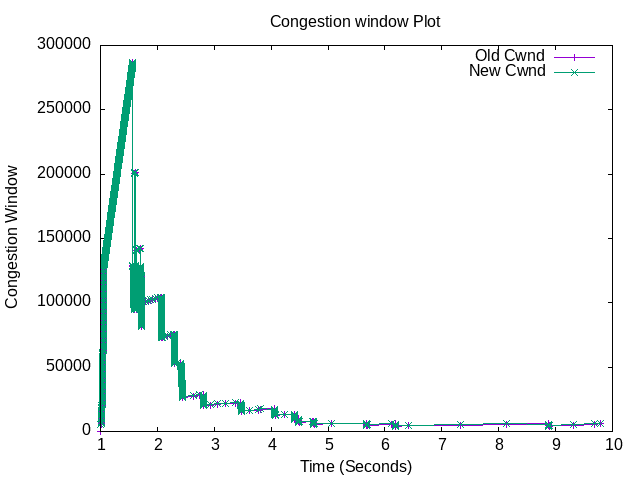
Increasing the bandwidht in TCP\_demo.cc file and ploting respective CW.plt file



**CW\_20.png (20 Mbps)**



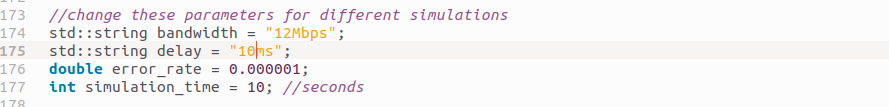
**CW\_25.png (25 Mbps)**



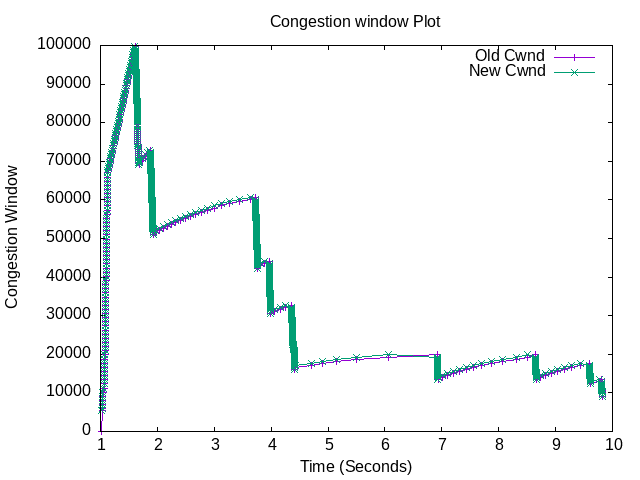
**CW\_55.png (55 Mbps)**

***”* THE ABOVE OBSERVATION SHOWS THAT ON INCREASING BANDWIDTH COONGESTION WINDOW INCREASES “**

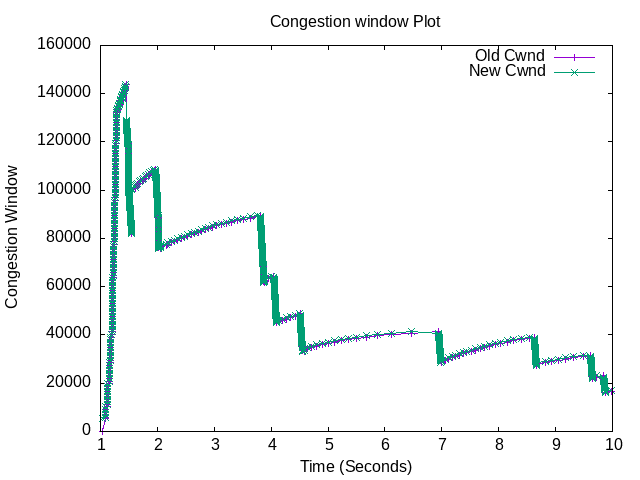
b)



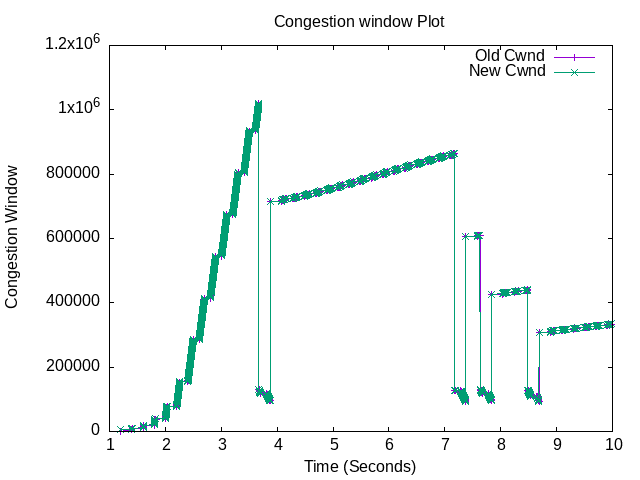
Increasing the delay in TCP\_demo.cc file and ploting respective CW.plt file



**CW\_10.png (10ms delay)**



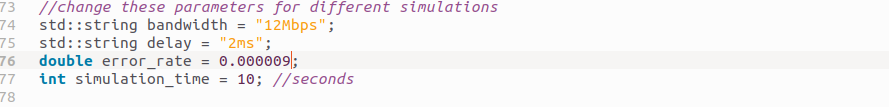
**CW\_20.png (20ms delay)**



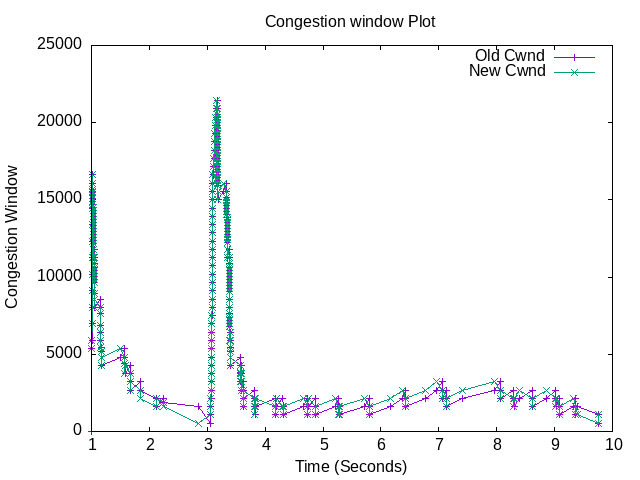
**CW\_100.png (10 0ms delay)**

***”* THE ABOVE OBSERVATION SHOWS THAT ON INCREASING DELAY CONGESTION WINDOW DECREASES “**

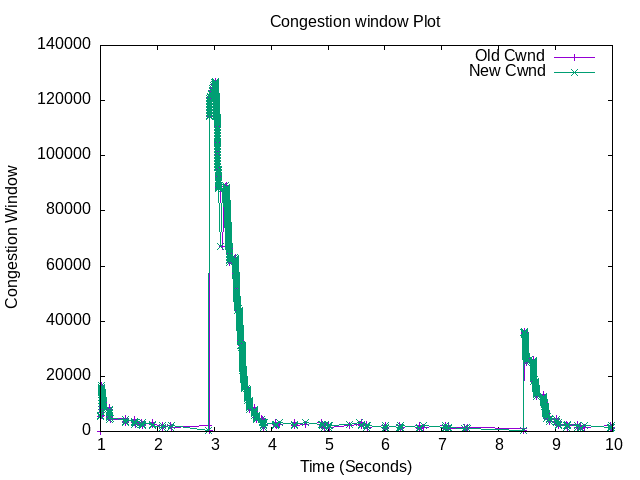
c)



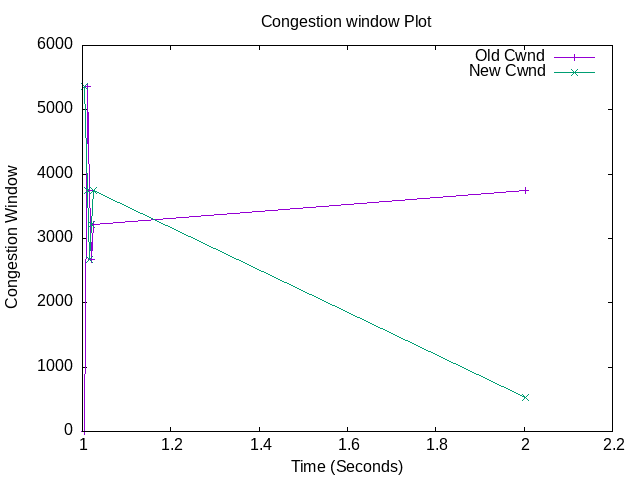
Increasing the error\_delay in TCP\_demo.cc file and ploting respective CW.plt file



**ERR\_9.png (0.000009)**

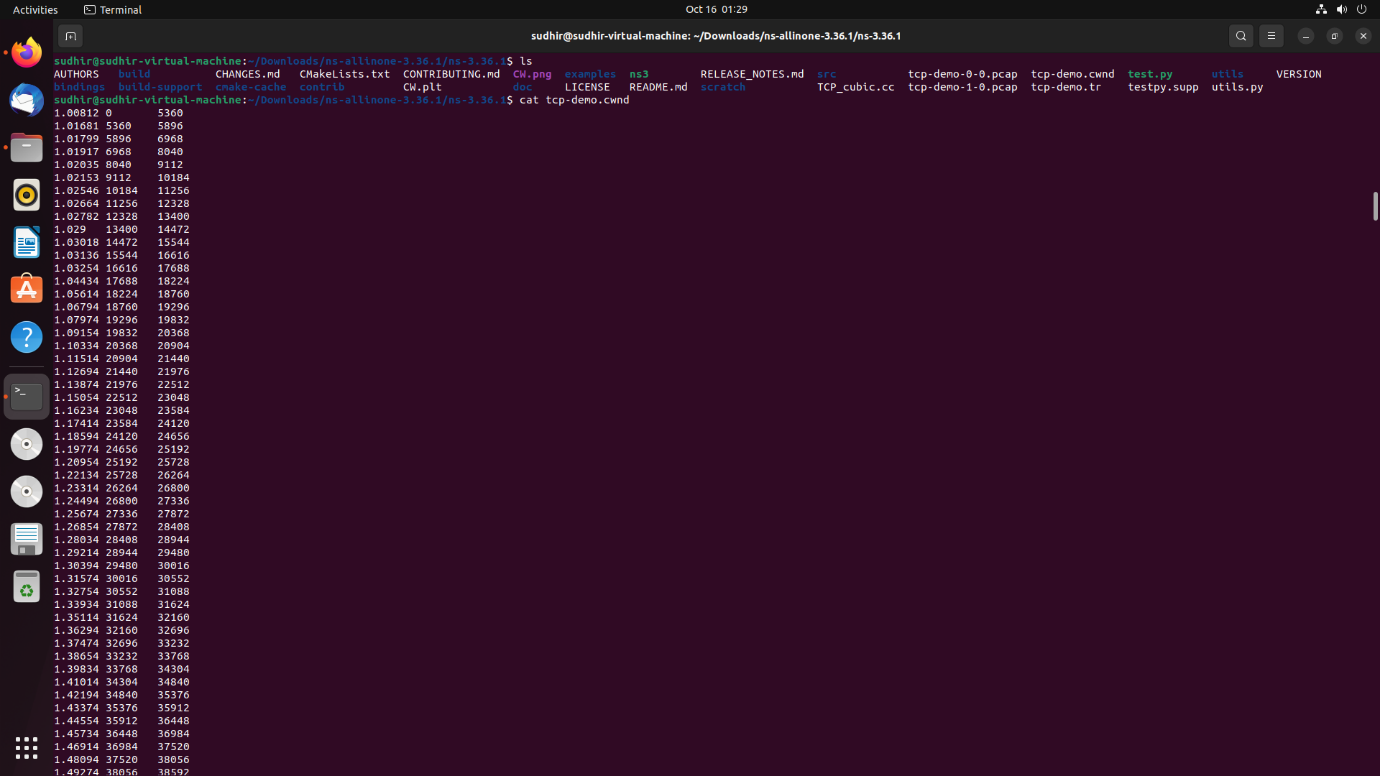


**ERR\_12.png (0.000012)**



**ERR\_2.png (0.0002)**

***”* THE ABOVE OBSERVATION SHOWS THAT ON INCREASING ERROR CONGESTION WINDOW DECREASES “**



**The tcp-demo.cwnd file contains the coordinates for the graph;**

**Solution of problem 3.**

Here are the 4 algorithms I explain. I have not used pseudocode as it can be found in the RFCs. I have explained the algorithms in English with their different stages and using formulas wherever needed.

# TCP Reno (Loss based) Alogrithm

* If three duplicate ACKS are recieved, then assume that the segement is lost and retransmit the segment and enter into fast-recovery mode.
* Half of the current CWND is saved as ssthresh (*ssthresh = CWND/2*) and reduce CWND by half (*CWND= CWND/2*) in case of three duplicate ACKS.
* At the arrival of another duplicate ACK the size of the congestion window increases by one until congestion is detected. *cwnd += 1*
* On ACK acknowledging new data, *CWND = ssthresh*, invoke congestion avoidance (linear increase in CWND)

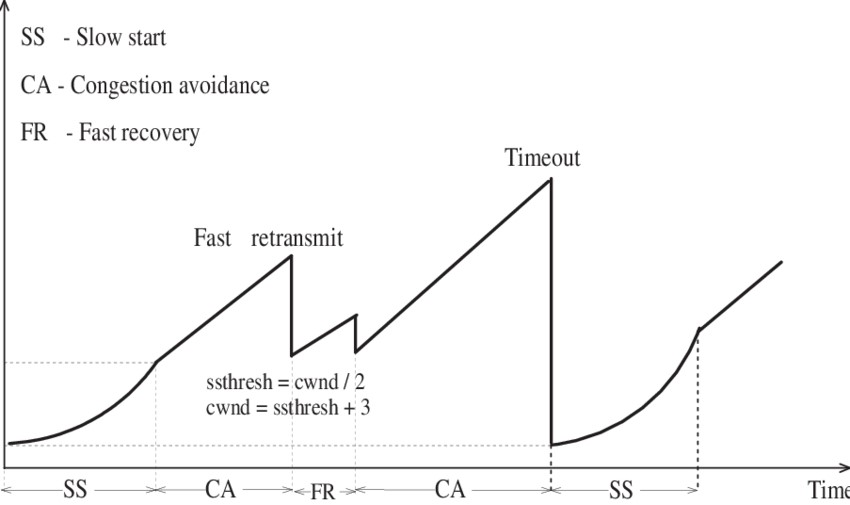


Figure 1.17: TCP Reno [1]

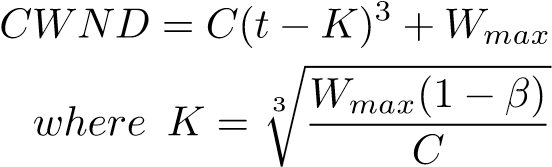
# Suitable for which scenarios and where it fails

* TCP Reno efficiently recover from a single packet loss, but the disadvantage is that it suffers from performance problems when multiple packets are dropped and makes the size of congestion window to 1.
* There exists fluctuations in round trip times, causing delay jitter and inefficient bandwidth utilization, and cause retransmissions of the same packet after a packet loss.
* If the size of the window is kept very small then the sender will not get enough DUPACKS for a fast retransmit and will wait for timeout.

TCP Cubic (Loss based)

# Alogrithm

* CUBIC increases its window to be real-time dependent, not RTT dependent.
* CUBIC has the following terms using in congestion control
  + *Wmax*: Window size just before the last reduction.
  + *β* : Mulitplicative decrease factor.
  + C : Scaling constant.
  + CWND: The congestion window at the current time.
  + t: Time elapsed since the last window reduction.
* After the window reduction followed by a loss event, CUBIC will make *Wmax* the new window size where the loss event occurred then there will be a multiplicative decrease of congestion window by a factor of *β* followed by regular fast recovery and retransmission of TCP.
* Then window size is increased using the concave profile of the cubic function, after it enters into congestion avoidance from fast recovery. The cubic function is set to have its narrow change at *Wmax* so the concave growth continues until the window size becomes *Wmax*.
* Next, the cubic function turns into a convex profile and the convex window growth begins. This style of window adjustment improves protocol and network stability while maintaining high network utilization because the window size remains almost constant, forming a plateau around *Wmax* where network utilization is deemed highest and under steady state, most window size samples of CUBIC are close to *Wmax*.



* After receiving ACK during congestion avoidance, CUBIC computes the window growth rate during the next RTT period using above equations. It sets W(t + RT T) as the target value of congestion window. Let the current window size is CWND. Depending on the value of CWND, CUBIC runs in three different modes.
  + If the CWND is lesser than window size that TCP would reach at time t after the last loss event, then CUBIC is in the TCP mode.
  + If cwnd is lesser than *Wmax*, then CUBIC is in the concave region.
  + If cwnd is larger than *Wmax*, then CUBIC is in the convex region.

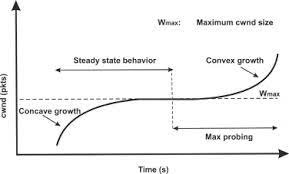
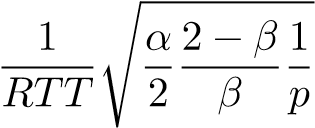
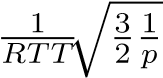


Figure 1.18: TCP Cubic [3]

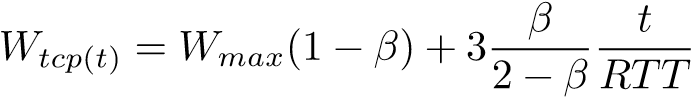
From the above graph we can see the following regions:

* TCP-friendly region: While receiving an ACK in congestion avoidance, first check whether the protocol is in the TCP region or not. We analyze the window size of TCP in terms of the elapsed time t. We then find the average window size of AIMD with an additive factor *α* and a multiplicative factor *β* using



From the above equation we can get the average window size of TCP with *α* = 1 and *β* = 0*.*5 i.e. .

For these equations to be same as that of TCP,. If TCP increases window size by *α* per RTT, then the window size of TCP in terms of t is:



* Concave: If CWND is less than *Wmax*, then the protocol is in the concave region. In this region, CWND is incremented by
* Convex: When the window size of CUBIC is larger than *Wmax*, it passes the plateau of the cubic function after which CUBIC follows the convex profile of the cubic function. If the protocol is the convex region outside the

TCP mode, CWND is incremented by.

* Multiplicative decrease: When a packet loss occurs, CUBIC reduces its window size by a factor of *β*.
* Fast Convergence: When a new flow joins the network, existing flows in the network need to give up their bandwidth shares to allow the new flow some room for growth. To increase this release of bandwidth by existing flows, the following mechanism called fast convergence is used:
  + The protocol remembers the last value of *Wmax* as *Wlastmax* before it updates *Wmax* for the current loss event.
  + At packet loss, if the current value of *Wmax* is less than the last value of *Wlastmax*, this indicates that the saturation point experienced by this flow is getting reduced because of the change in available bandwidth.
  + Then this flow is allowed to release more bandwidth by reducing *Wmax* further.
  + This allows more time for the new flow to catch up its window size.

# Suitable for which scenarios and where it fails

We can see that if C is excellent in high speed wired environment then throughput of CUBIC is not affected in wired network in the presence of reverse traffic. CUBIC perform bad in wireless network. It does not achieve throughput and better level of fairness in wireless network.Therefore we can say that it is poor for wireless or low speed networks. TCP Vegas (Delay based)

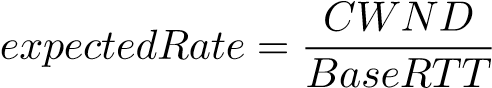
# Alogrithm

* TCP Vegas uses an Additive Increase Additive Decrease(AIAD) approach.
* During congestion avoidance, TCP vegas calculates the difference between the expected and actual sending rate to estimate the data currently queued at the bottleneck.

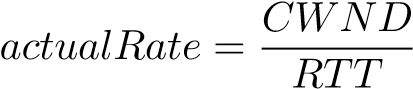
*Diff* = (*Expected* − *Acutal*)∗ *BaseRTT* (1.1)

where BaseRTT is the minimum round trip time.

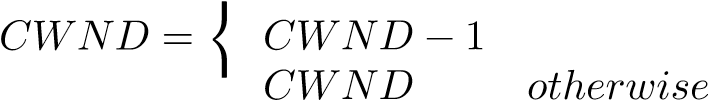
* The expected sending rate is calculated by dividing CWND (current window size) by BaseRTT



* The actual sending rate is calculated using actual BaseRTT



* The bottleneck is estimated using equation 1. Based on Diff the source updates its window size. *α* and *β* are algorithm parameters which control the length and the stability of the bottleneck queue.

 *CWND* +1 if *Diff < α* if *Diff > β*



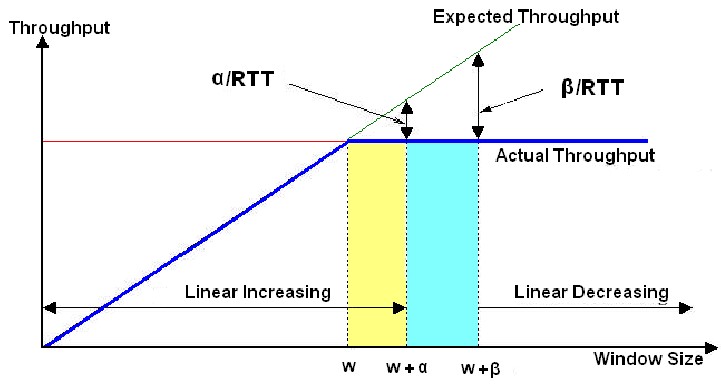


Figure 1.19: TCP Vegas [2]

The idea of this algorithm is that if a sender can send a CWND size of data without having a large increase in RTT , it means that the link is under-utilised and one can increase CWND. Alternatively, if the increase in RTT is such that Diff exceeds *β* threshold, it means the link is overly utilised and we should reduce CWND.

# Speedier Retransmission Mechanism

Since TCP Vegas has a precise timer, the packet loss can be determined with one DUPACK only. When a DUPACK is received, the algorithm checks *Current Time - Packet Transmission Time* ¿ *Timeout* for packet. If the inequality is true TCP Vegas will retransmit the packet immediately (hence it will not wait for 2 more DUPACKs)

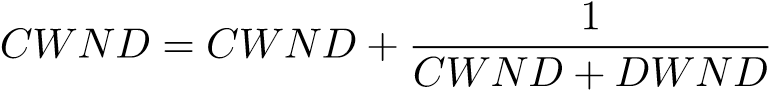
# Suitable for which scenarios and where it fails

The Speedier Retransmission Mechanism of Vegas is efficient in congestion control. However Vegas suffers from loss-based flows. If there is some retransmission within the sampleRTT duration, Vegas will not perform congestion avoidance algorithm step because CWND would have been changed. It’s performance degrades substantially in asymmetric networks (i.e., ADSL (Asymmetric Digital Subscriber Line))

TCP Compound (CTCP) (Hybrid)

# Alogrithm

* A new state variable is introduced in the current TCP Control Block (TCB), called, *dwnd* (Delay Window), which controls the delay-based component in CTCP, *cwnd* remains same. The CTCP sending window now is controlled by both cwnd and dwnd.
* The TCP sending window is *win* = *min*(*CWND* + *DWND,AWND*) where where awnd is the advertised window from the receiver.
* The update of CWND is same as in the TCP in the congestion avoidance phase. The CWND is increased by one every RTT and halved when packet loss occurs. But, here CTCP may send CWND+DWND packets in one RTT. Therefore, the increment of CWND on the arrival of an ACK is modified as



* CTCP has the same slow start mechanism of TCP. Initially DWND is set to 0 if the connection is in a slowstart state, and the delay-based component is effective only when the connection is working at the congestion avoidance phase.
* CTCP uses the delay-based algorithm from TCP Vegas.
* A state variable, baseRTT, is used for estimation of the transmission delay of a packet. When the connection is started, baseRTT is updated by the min RTT that has been observed so far. An exponentially smoothed current RTT, sRTT, is also kept.Then, the number of backlogged packets of the connection can be estimated by:



Expected estimates the throughput we get if we do not overrun the network path and Actual stands for the throughput we really get.

**Comparisons:**

Among New Reno and Cubic, Cubic is less aggressive and more stable while New Reno is more aggressive. Compared to them Vegas less aggressive. But is more adjustable according to congestion. It can detect congestion early compared to loss-based variants. It will give lesser RTT. In straightforward data transfers not involving much loss Vegas can give bad performance compared to the loss-based versions. Compound TCP being a mix of both loss and delay based algorithms can handle these cases in a balanced fashion as it takes aggressiveness from loss based and adjustability from delay based.

# References

1. Zhang, Hui and Liu, Mingjian and Vukadinovic, Vladimir and Trajkovic, Ljiljana. (2006). Modeling TCP/RED: a Dynamical Approach. 10.1007 10973509 11.
2. Abed, Dr.Ghassan and Ismail, Mahamod and Jumari, Kasmiran. (2011). Characterization and observation of (transmission control protocol) TCP-Vegas performance with different parameters over (Long term evolution) LTE networks. Scientific Research and Essays. 6. 2003-2010. 10.5897/SRE11.252.
3. Ahmad, Mudassar and Ngadi, Md and Ahmad, Usman and Amjad, Kashif and Ghafir, Ibrahim and Arioua, Mounir. (2017). A New Linux Based TCP Congestion Control Mechanism for Long Distance High Bandwidth Sustainable Smart Cities. Sustainable Cities and Society. 37. 10.1016/j.scs.2017.11.005.
4. https://www.fastly.com/blog/quic-handshake-tls-compression-certificates-extension-study
5. Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor

Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM ’17). Association for Computing Machinery, New York, NY, USA, 183–196. DOI:https://doi.org/10.1145/3098822.3098842

1. https://www.ionos.com/digitalguide/hosting/technical-matters/quic-the-internet-transport-protocol-based-on-udp/