**CSE434 Lab 2 Report**

**Group 82**

**Group Members:**

**Jeet Thakkar**

**Wei Hng Yeo**

# 

[**Part 1: Tests with different TCP Protocols**](#_o1mjauvuea0p) **3**

[Exercise 1.1: Tests with TCP Reno](#_y94m2n1oiylk) 3

[Experiment Screenshot: TCP Reno](#_yky33l7ro0i) 3

[1.1.1: Plot: Annotated](#_74fd5zevo42q) 3

[1.1.2: Explanation:](#_pljkjcojm1jz) 4

[Exercise 1.2: Test with TCP Cubic](#_k9lymdpkah3b) 5

[Experiment Screenshot: TCP Cubic](#_dpr49s2e99u) 5

[1.2.1: Plot: Annotated](#_yni7is9q1c88) 6

[1.2.2 Explanation:](#_d9108f21jear) 7

[1.2.3: TCP cubic reaches faster available bandwidth faster than TCP Reno](#_ddcu1xyvigbm) 7

[Exercise 1.3: Tests with TCP Vegas](#_6mi4u3ynx6v9) 8

[Experiment Screenshot:](#_a5pgpt21n0t4) 8

[Plot of Vegas w/o Reno Running on the same machine (for comparison)](#_jnhn9kj5peyn) 8

[Plot of Reno vs Vegas](#_11uco29zk17i) 9

[1.3.1: iperf3 values for Reno vs Vegas [From the Screenshot]](#_vujc1qze2a38) 10

[1.3.2: Explanation](#_y8q0tuerep3j) 11

[Exercise 1.4: Tests with TCP Reno w/ECN Enabled.](#_wiksskt19u8b) 12

[Experiment Screenshots:](#_ntk1mqctra4s) 12

[Setup (Router Setup)](#_2pruh73saf9h) 12

[Setup (End-Host Setup)](#_3np5chomzvbs) 13

[TCP-ECN Run Time](#_gbctfqxuuil2) 13

[1.4.1: Delay Performance Comparisons](#_p9ugzctybqjg) 14

[Delay Performance of TCP-Reno w/o ECN:](#_ivxlx52qzpjz) 14

[Delay Performance of TCP-Reno w/ECN:](#_mtteqswowg4u) 14

[1.4.2: Examining the ECN pcap](#_5t5vfitmhf0r) 14

[1.4.2.a: Finding the TCP ECN, SYN and ECN-SYN-ACK response](#_wi8snfyw5fkx) 14

[1.4.2.b: Data packet with ECN bits set to 01 or 10](#_n9rimqqeon65) 16

[1.4.2.c: ECN notification of congestion and corresponding reduction of CWND](#_z776s5s9u55h) 16

[**Part 2: Basic Home Gateway Services: DHCP, DNS, NAT**](#_w1idgamdqdm9) **20**

[Exercise 2.1 Observe a DHCP Request and Response](#_ndf081fjeg97) 21

[Exercise 2.2 Observe a DNS Query and Response](#_eezj06q0lyoy) 23

[Exercise 2.3 Use NAT](#_6mnlvvzg7cdl) 25

# 

# Part 1: Tests with different TCP Protocols

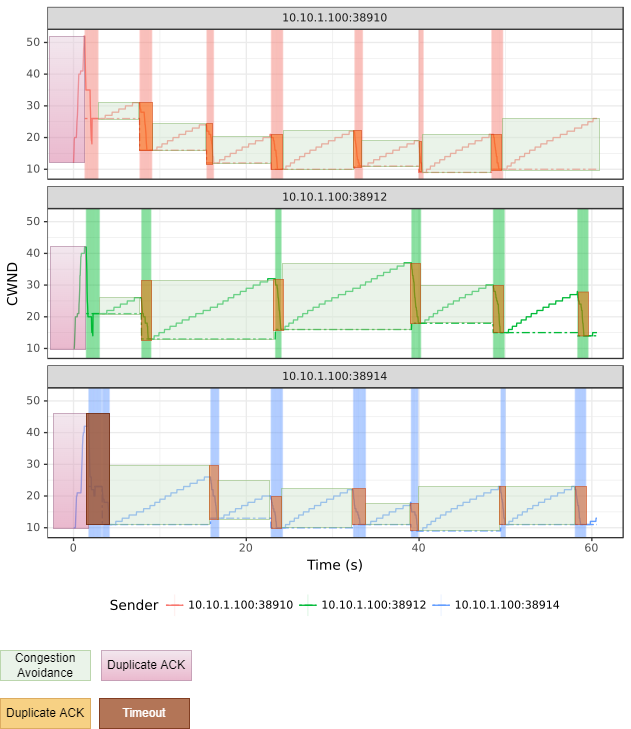
For each experiment we have attached the results, outputs and the files generated in the subfolder of the “Lab2/Lab2\_1/[experiment name like reno, cubic, reno-vs-vegas]” of the folders attached to the report.

## Exercise 1.1: Tests with TCP Reno

### Experiment Screenshot: TCP Reno



### 1.1.1: Plot: Annotated



### 1.1.2: Explanation:

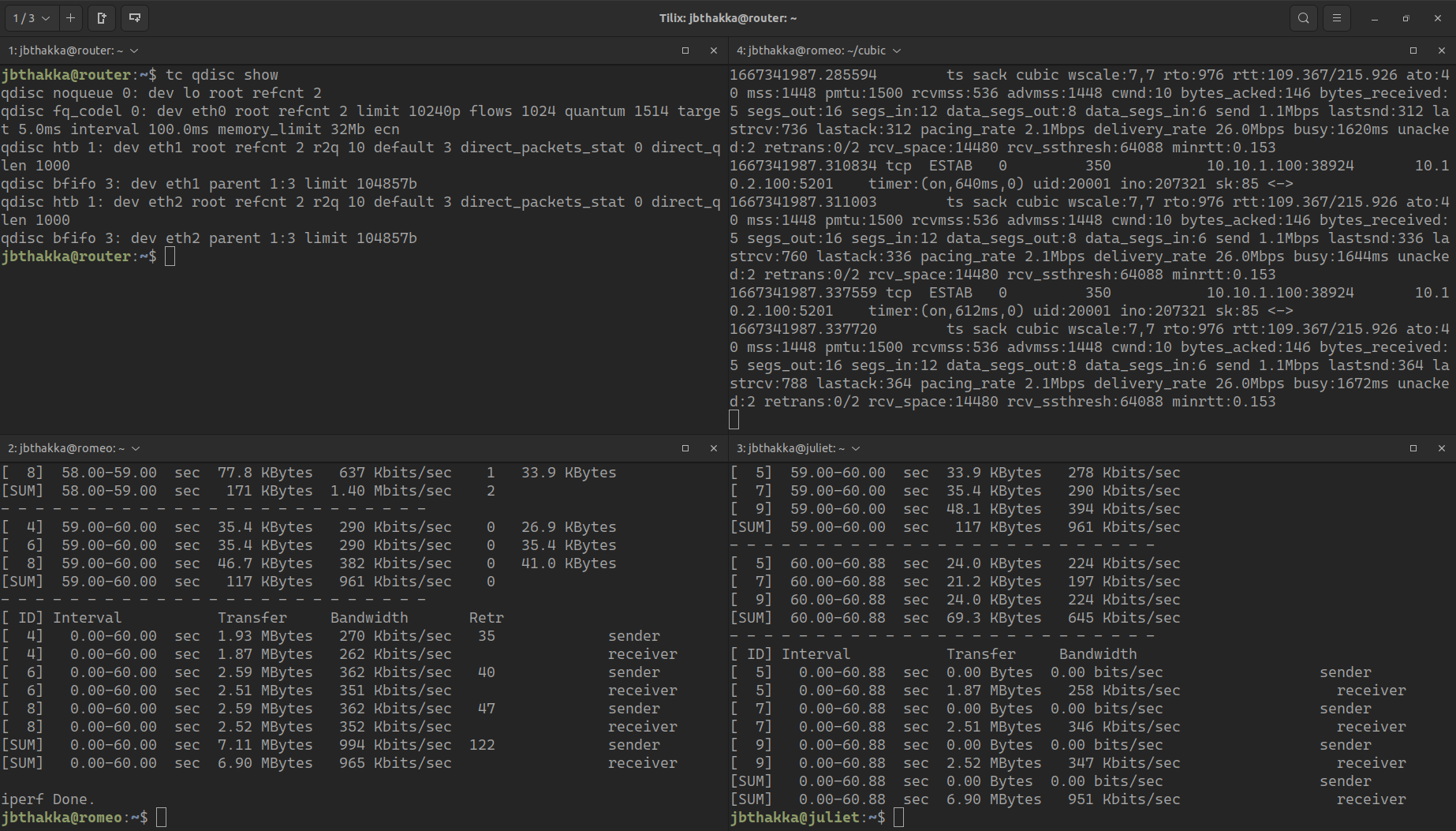
In TCP Reno, the Slow Start Phase consists of setting the CWND to MSS i.e. 10, and then doubling the CWND until a timeout or until a Duplicate ACK is received. In the plot no. 1 and plot no. 2 we see a duplicate ACK, so the new CWND is set to (CWND at duplicate ACK/2), but in plot 3 we see a timeout is received so the CWND value is set back to 10.

After that point congestion control (AIMD) takes over and there is linear increase in the CWND until a Duplicate ACK is received - at which point the CWND is set back to CWND/2 - In any instance of a timeout, CWND is set back to 10.

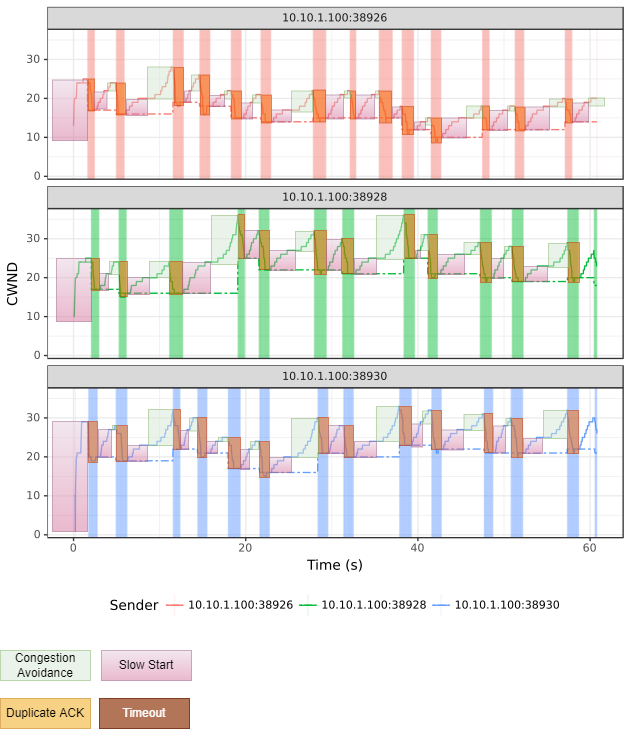
In our plots, we see a timeout occur in only one place which is in Plot 3 for 10.10.1.100:38914 in the beginning - since we do not see any place where CWND comes back to 10 after reaching a high, of greater than 25, we can infer that there is no timeouts.

## Exercise 1.2: Test with TCP Cubic

### Experiment Screenshot: TCP Cubic

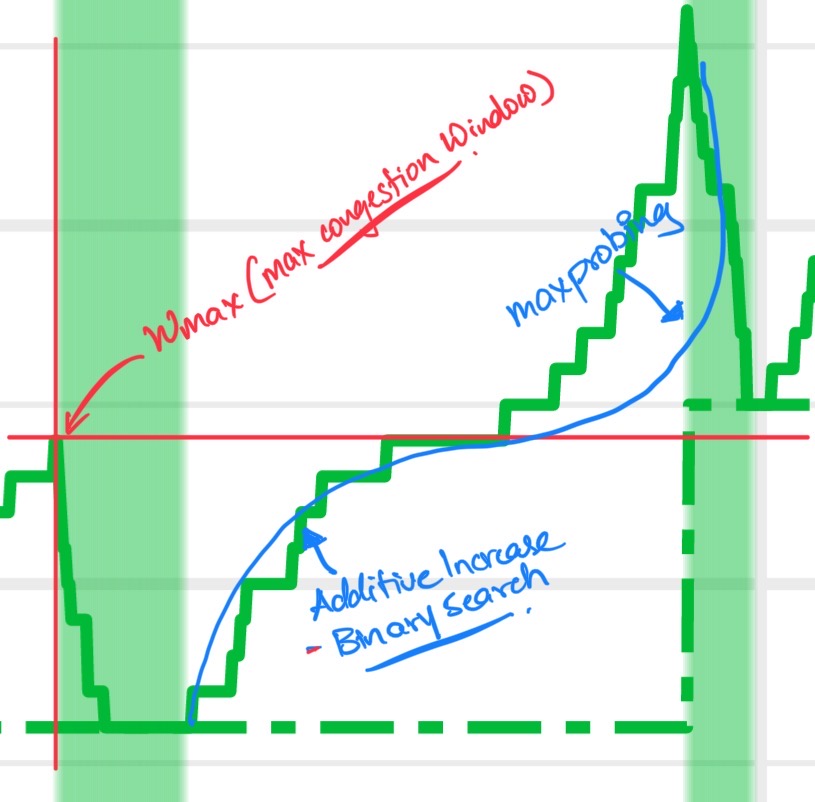


### 1.2.1: Plot: Annotated



### 1.2.2 Explanation:

A maximum window size Wmax is set (usually to the last value of the CWND) before the decrease. The CWND is increased rapidly until it approaches the Wmax value, and then it tapers off. After crossing the threshold Wmax, there are small increases to CWND - followed by greater and greater increases until the next duplicate ACK is received, where the the CWND is set to the next Wmax threshold and the CWND is then decreased by the multiplicative decrease factor specified.



### 1.2.3: TCP cubic reaches faster available bandwidth faster than TCP Reno

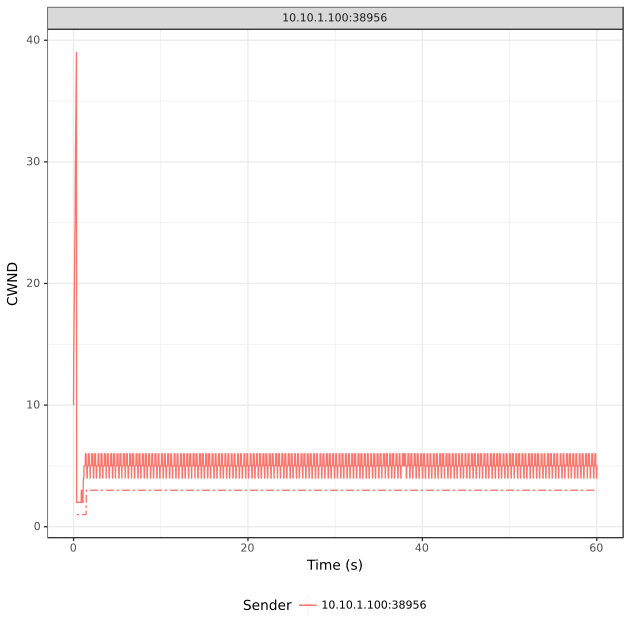
TCP Cubic reaches available bandwidth faster due to the exponential increase &then conservative increase until the Wmax value is reached, and then the max-probing cubic incrementing of the CWND, until it encounters the next duplicate ACK or timeout. Also, unlike Reno, Cubic does not penalize for the receipt of a duplicate ACK by cutting the CWND to half - there is a much smaller factor of decrease in the CWND in Cubic than in Reno.

## Exercise 1.3: Tests with TCP Vegas

### Experiment Screenshot:

### 

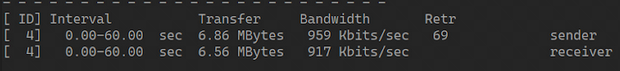
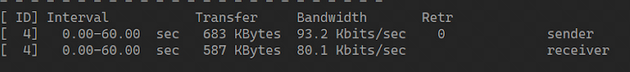
### Plot of Vegas w/o Reno Running on the same machine (for comparison)



Here we see the congestion window oscillate between a=4 and b=6 - The TCP Vegas algorithm is oscillating between these values based on the difference in the expectedOutput and actualOutput.

### Plot of Reno vs Vegas

### 1.3.1: iperf3 values for Reno vs Vegas [From the Screenshot]

* Reno :  
  
  + Bandwidth at sender: 959 Kbits/s
  + Bandwidth at sender: 917 Kbits/s
* Vegas :  
  
  + Bandwidth at sender: 93.2 Kbits/s
  + Bandwidth at sender: 80.1 Kbits/s

### 1.3.2: Explanation

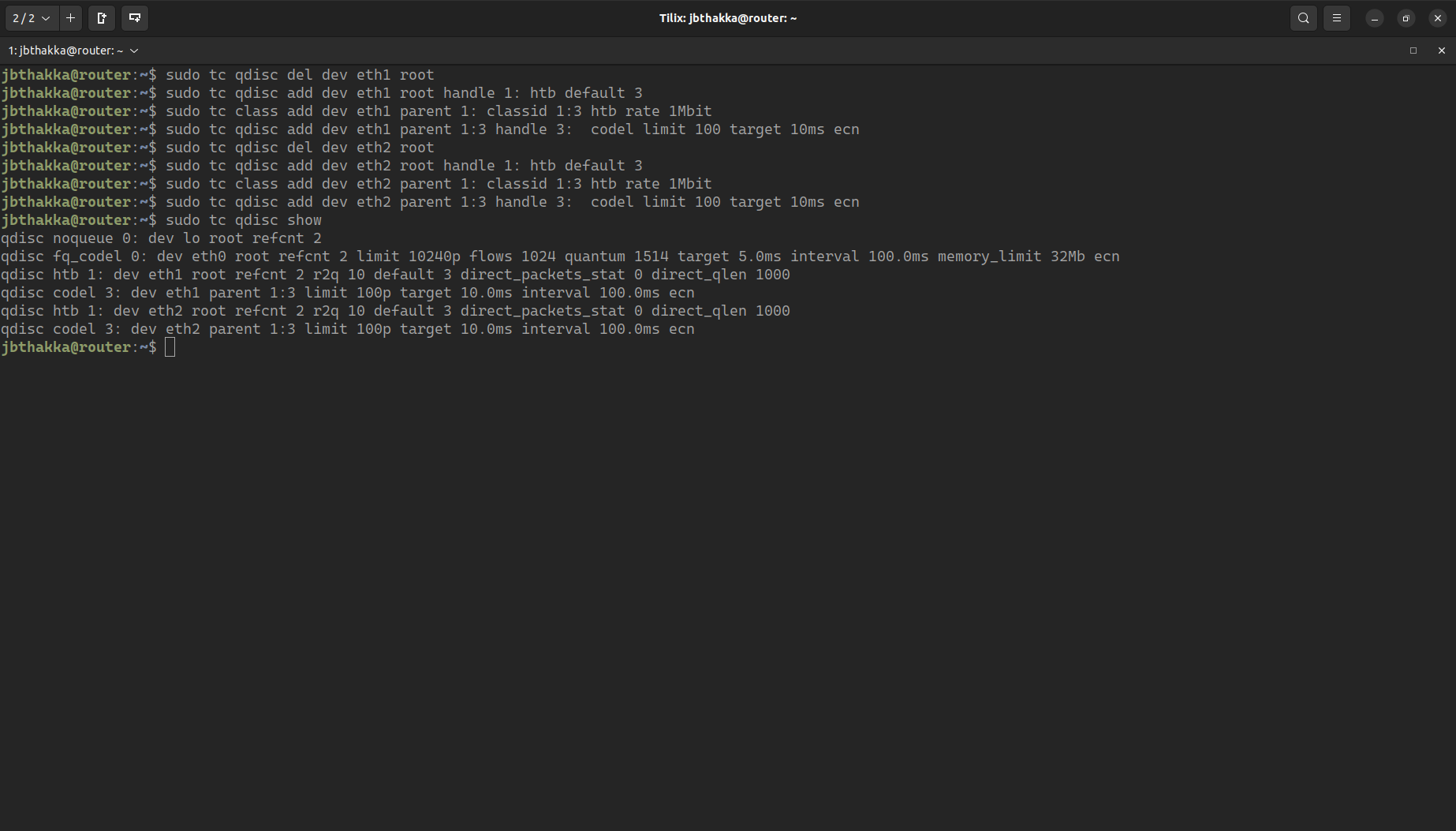
TCP Reno performs better than TCP Vegas, when sharing the same bottleneck link, because TCP Reno infers congestion based on Packet Loss/Duplicate ACKs - so it does not regulate the sender unless such an event occurs. However, TCP Vegas actually infers “congestion” based on packet delay - so it penalizes delay events, long before actual congestion occurs; In this case, the sender regulates its CWND, before it detects actual packet loss event; This strategy works well when you have other senders running the same protocol - a system of “honor” is maintained among the senders, by not grabbing all the available bandwidth.

However, TCP Reno will keep grabbing the bandwidth unless a packet loss event occurs, which causes TCP Vegas to regulate itself, since TCP Reno keeps ramping up the sender, there are delays inferred on TCP Vegas’s Sender side, causing a vicious cycle of self-reduction of CWND until TCP Vegas is reduced to its minimum value.

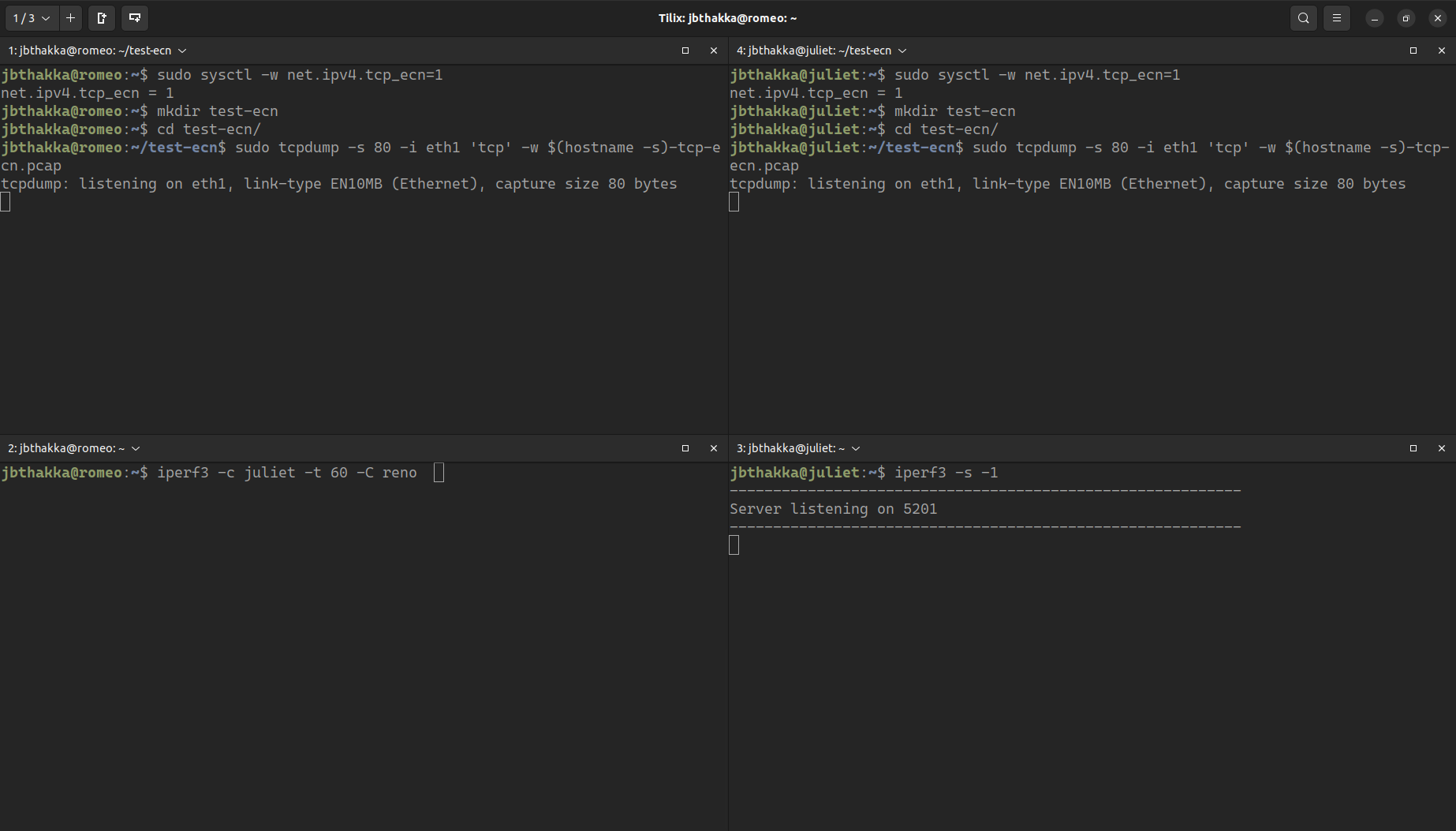
## Exercise 1.4: Tests with TCP Reno w/ECN Enabled.

### Experiment Screenshots:

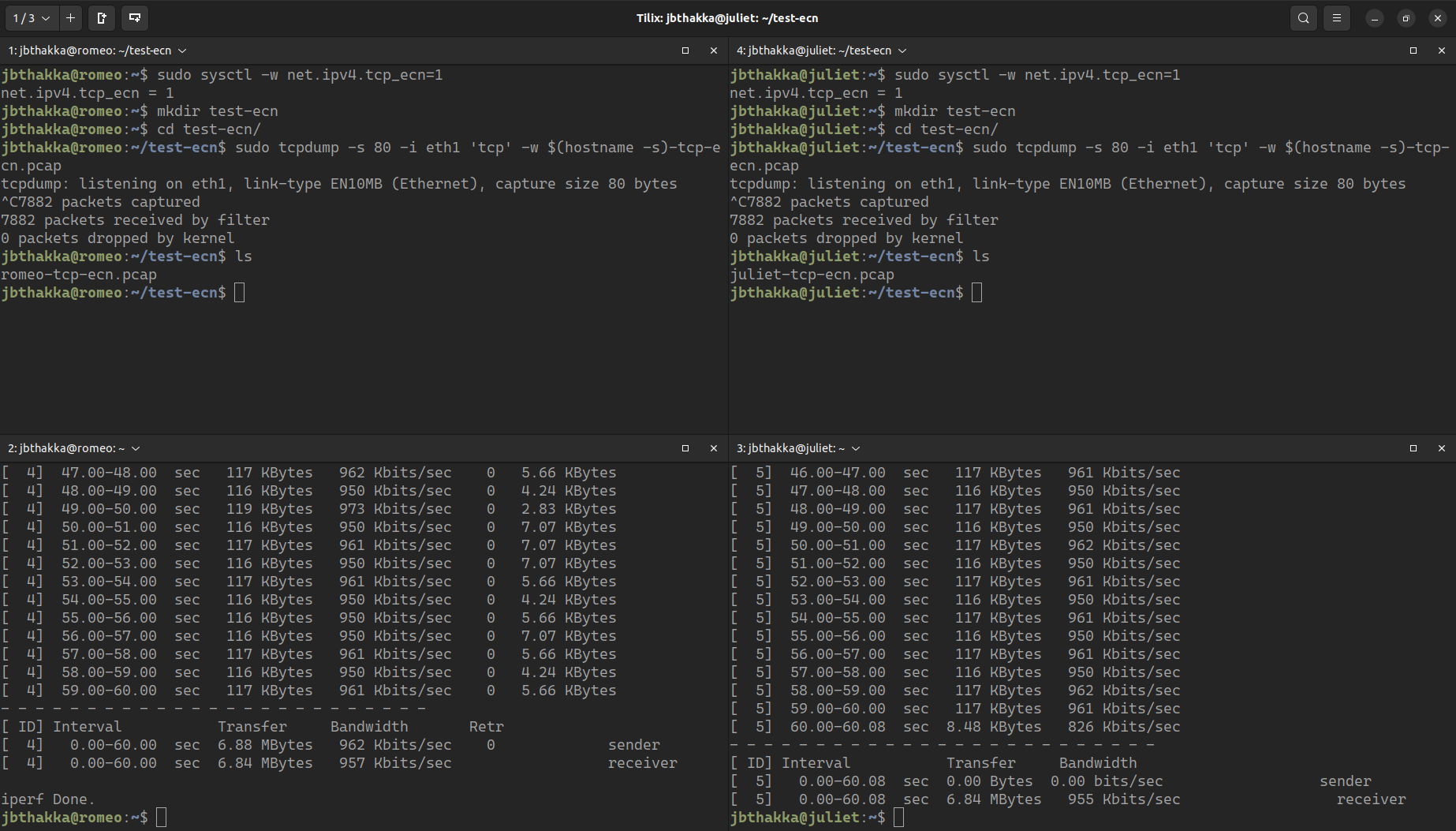
#### Setup (Router Setup)



#### Setup (End-Host Setup)

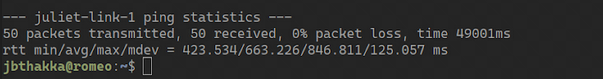


#### TCP-ECN Run Time

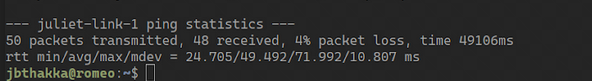


### 1.4.1: Delay Performance Comparisons

#### Delay Performance of TCP-Reno w/o ECN:



#### Delay Performance of TCP-Reno w/ECN:

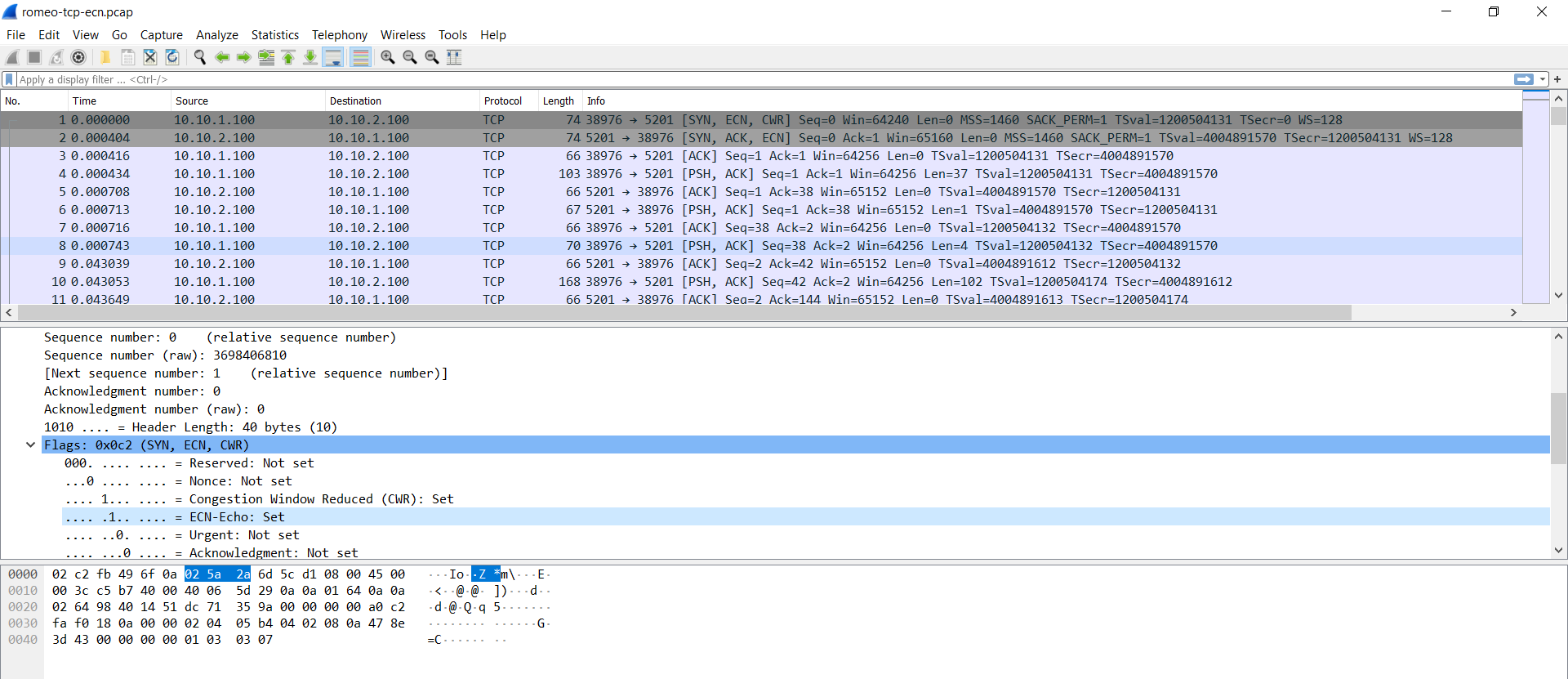


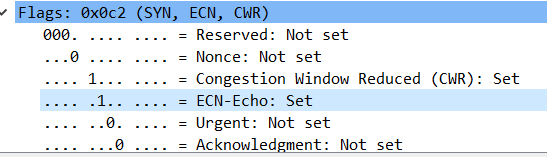
We see the delay in TCP Reno with ECN enabled is significantly lesser than w/o ECN, due to the notifications sent by the router that participates in the connection.

### 1.4.2: Examining the ECN pcap

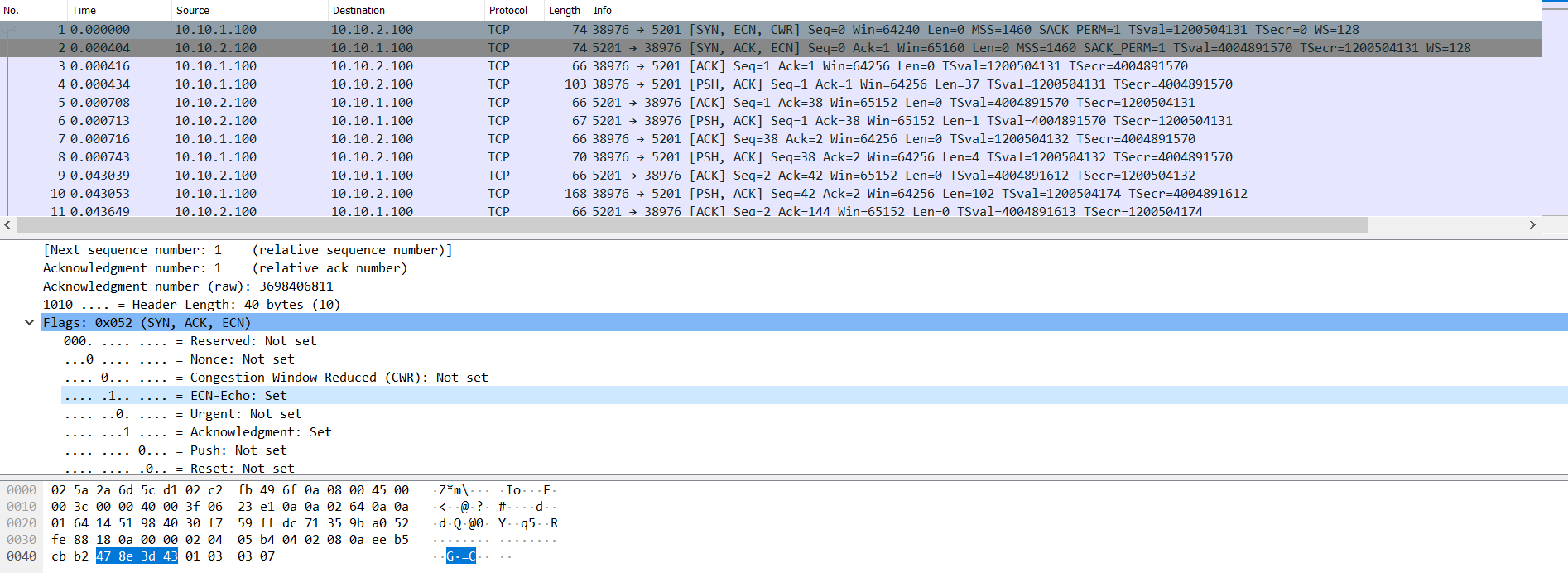
#### 1.4.2.a: Finding the TCP ECN, SYN and ECN-SYN-ACK response

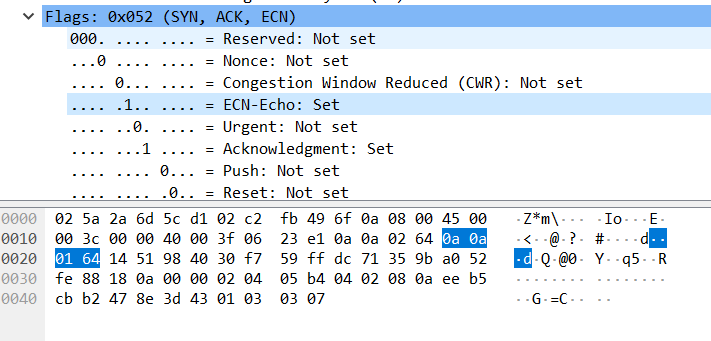
TCP SYN Packet from Romeo to Juliet



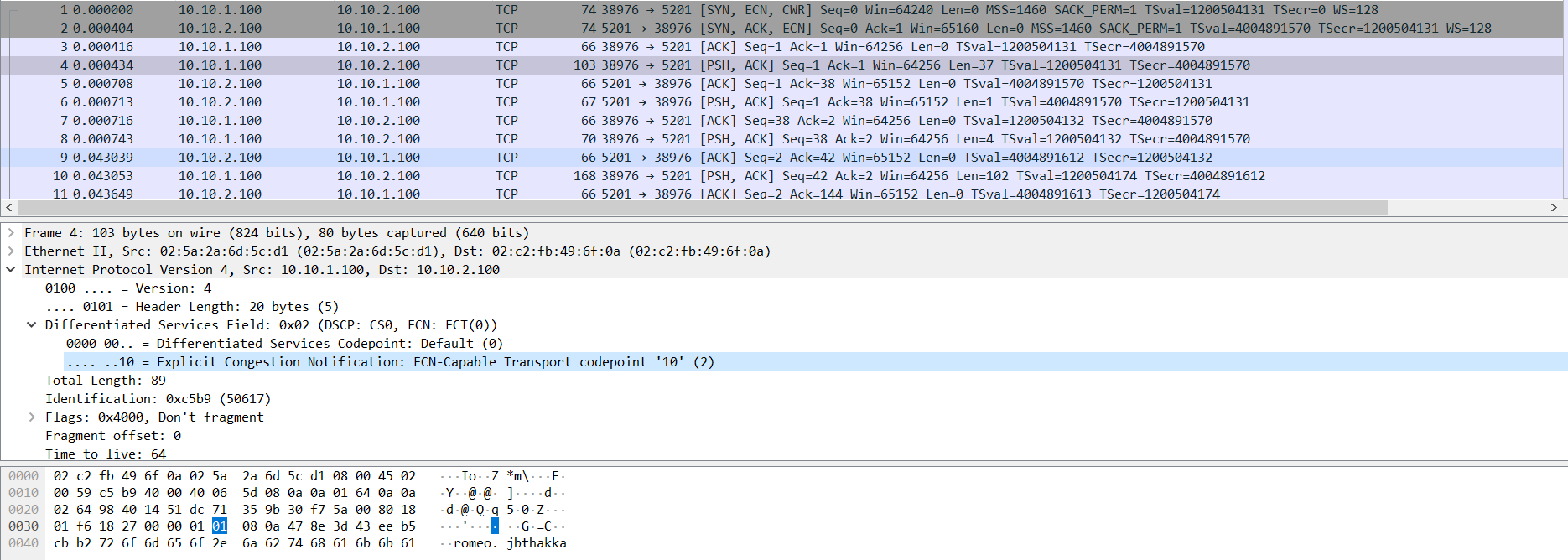


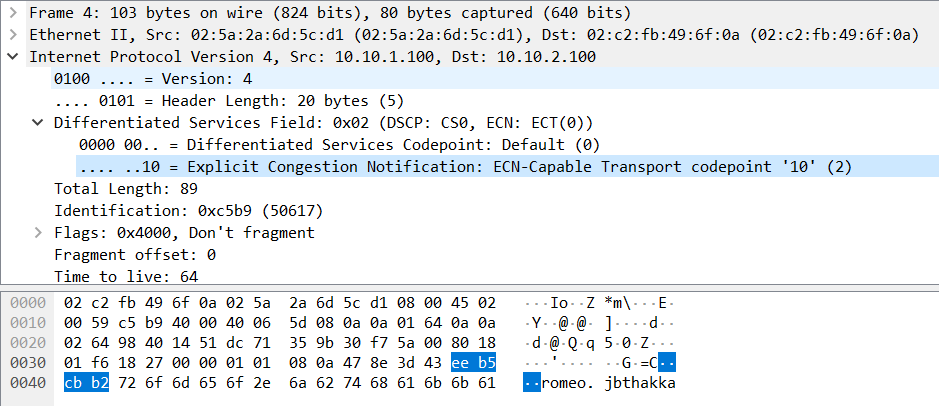
The SYN-ACK Packet from Juliet to Romeo



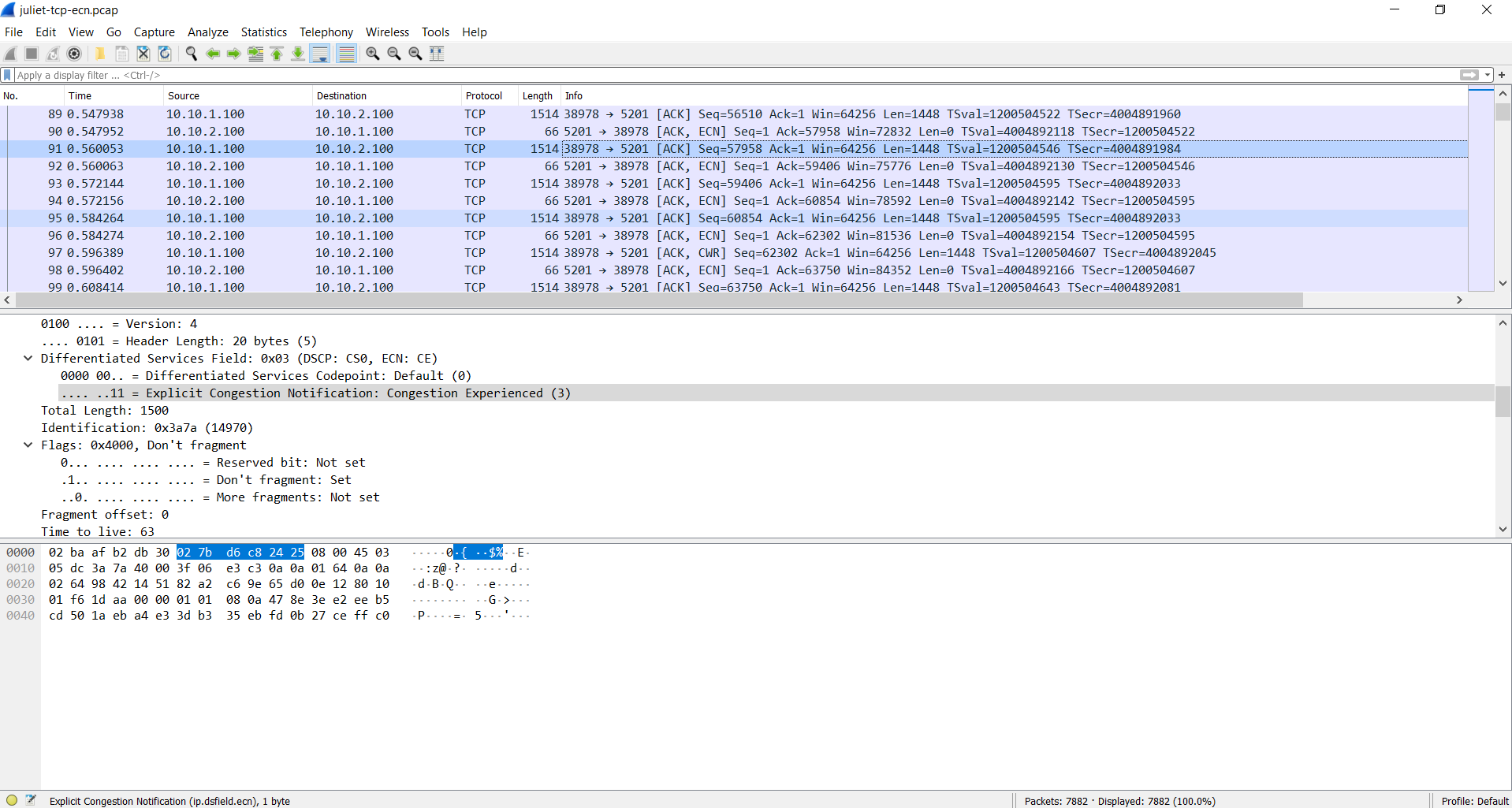


#### 1.4.2.b: Data packet with ECN bits set to 01 or 10

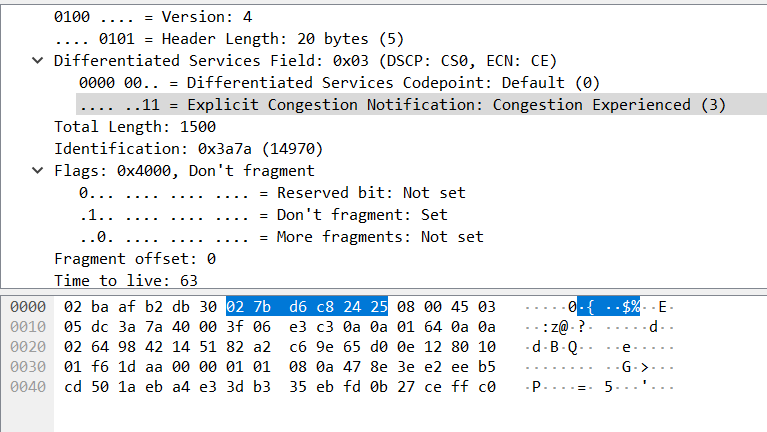




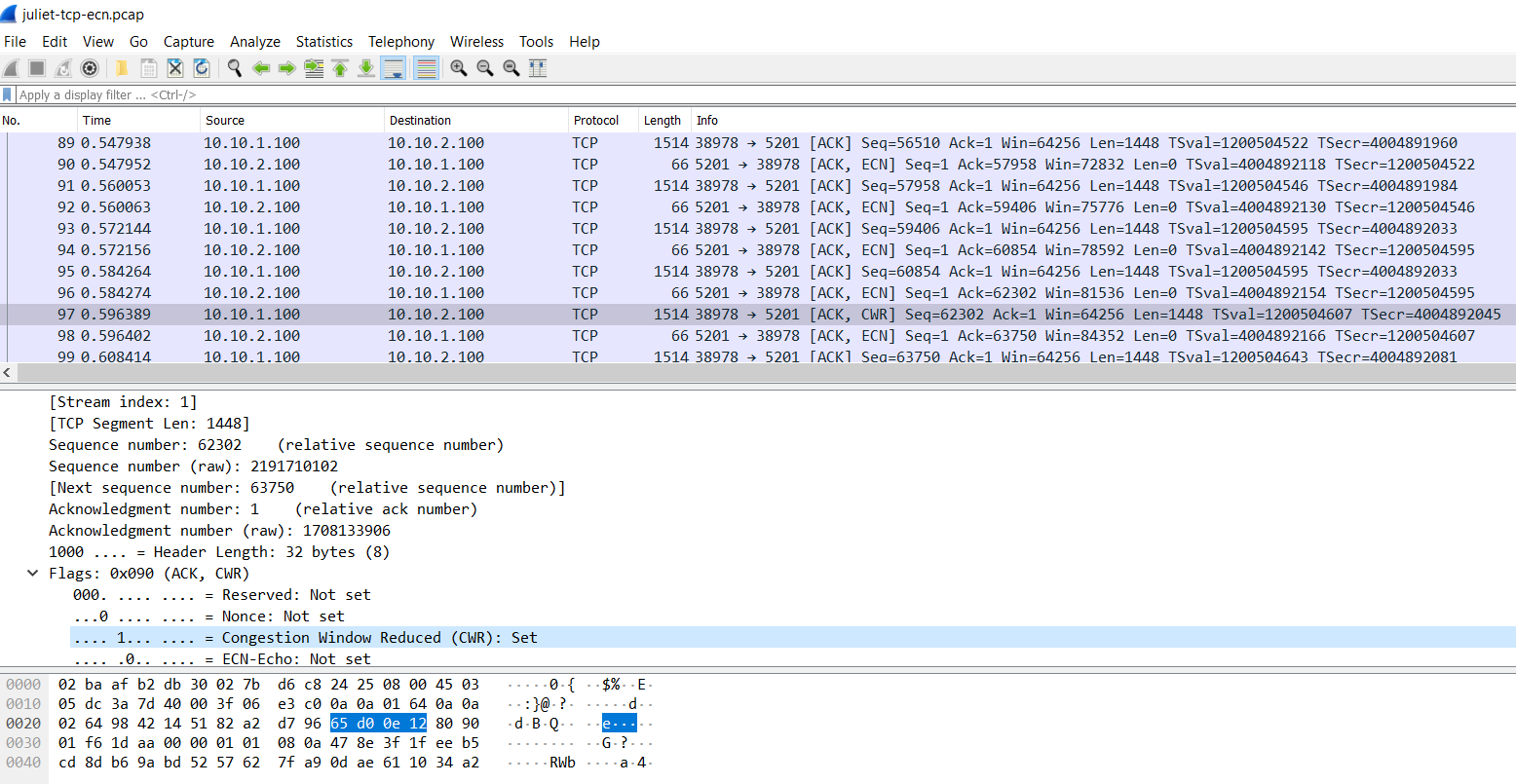
#### 1.4.2.c: ECN notification of congestion and corresponding reduction of CWND

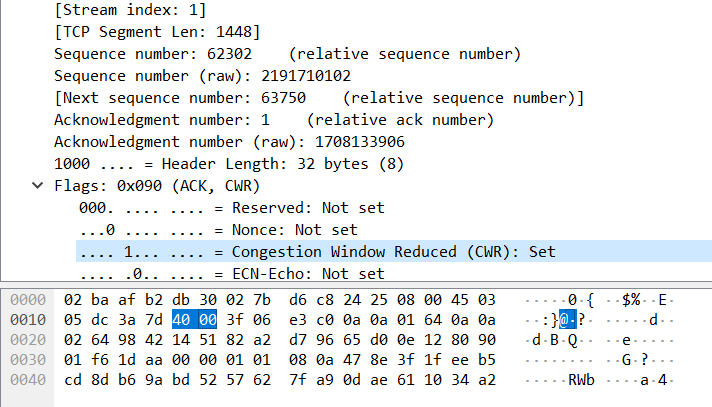
In order to find this we go into Juliet’s pcap and filter using the wireshark parameter ip.dsfield.ecn == 3 because the router will explicitly send the congestion notification to the receiver and the receiver will then signal the sender via the response - we have to look for the corresponding ACK which will contain the notification, the reduced CWND on Romeo’s side.  


In the screenshot we see this on packet with seq number 57958

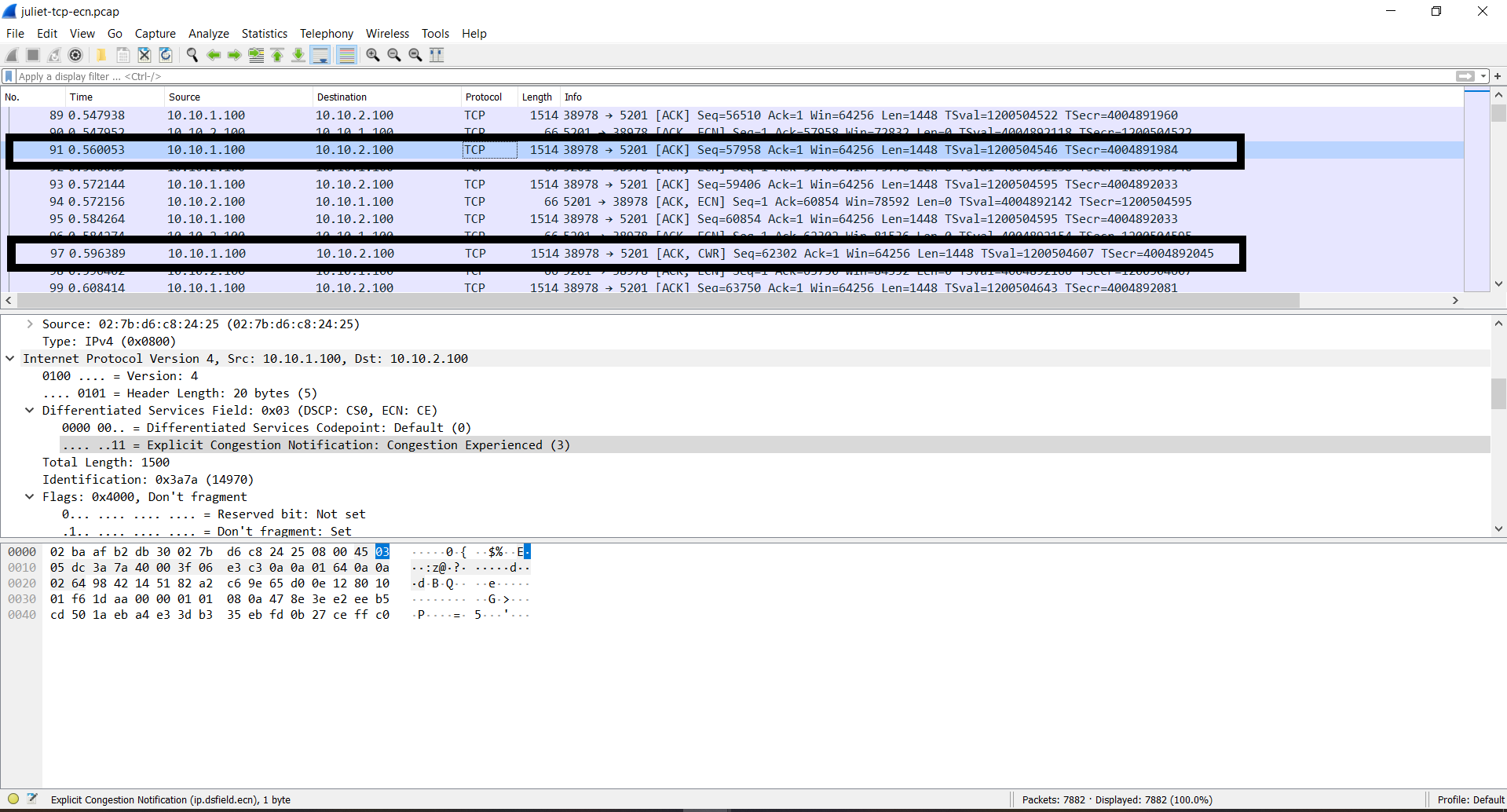


Then the subsequent packet with seq number 62302 comes with the CWR flag set to true.

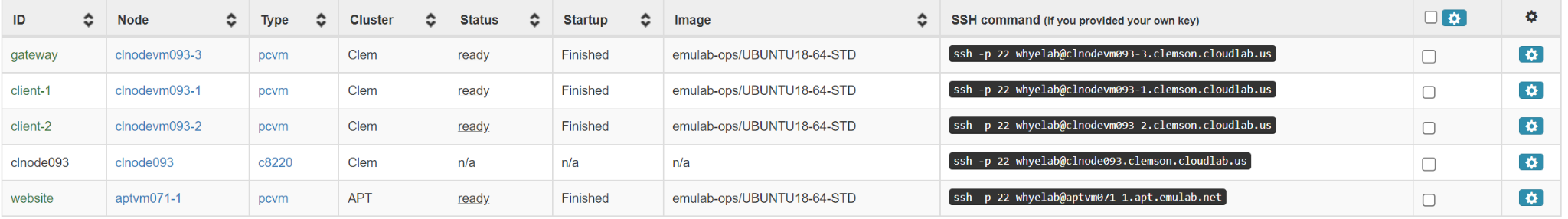




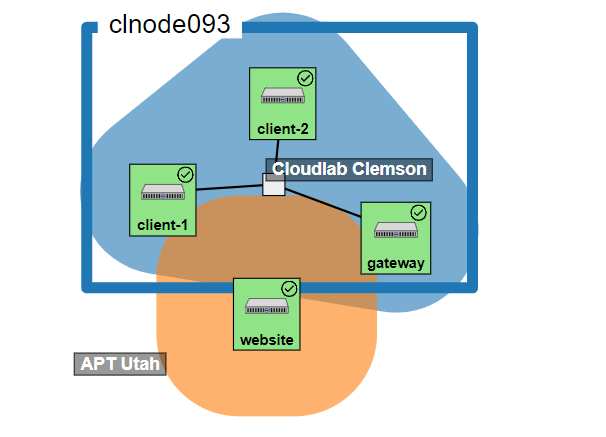
Both the packet details are shown below (from the Lab2/Lab2\_Part1/tcp-ecn/juliet-tcp-ecn.pcap):



# Part 2: Basic Home Gateway Services: DHCP, DNS, NAT



List View



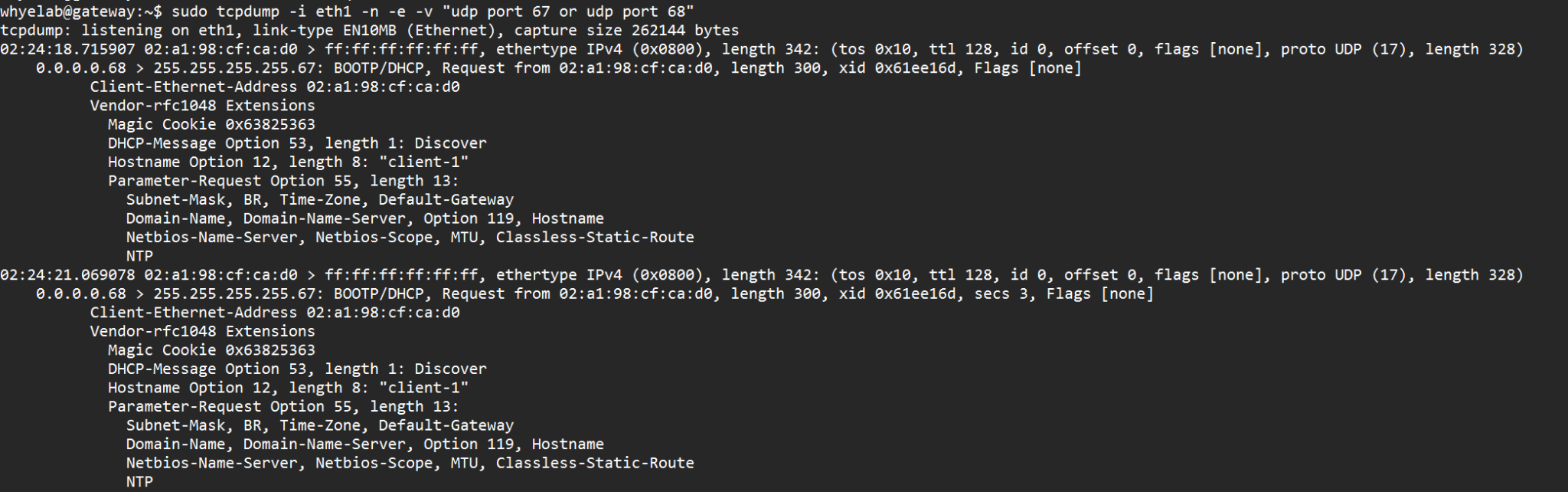
Topology View

## **Exercise 2.1 Observe a DHCP Request and Response**

1. Source IP: 0.0.0.0

Destination IP: 255.255.255.255

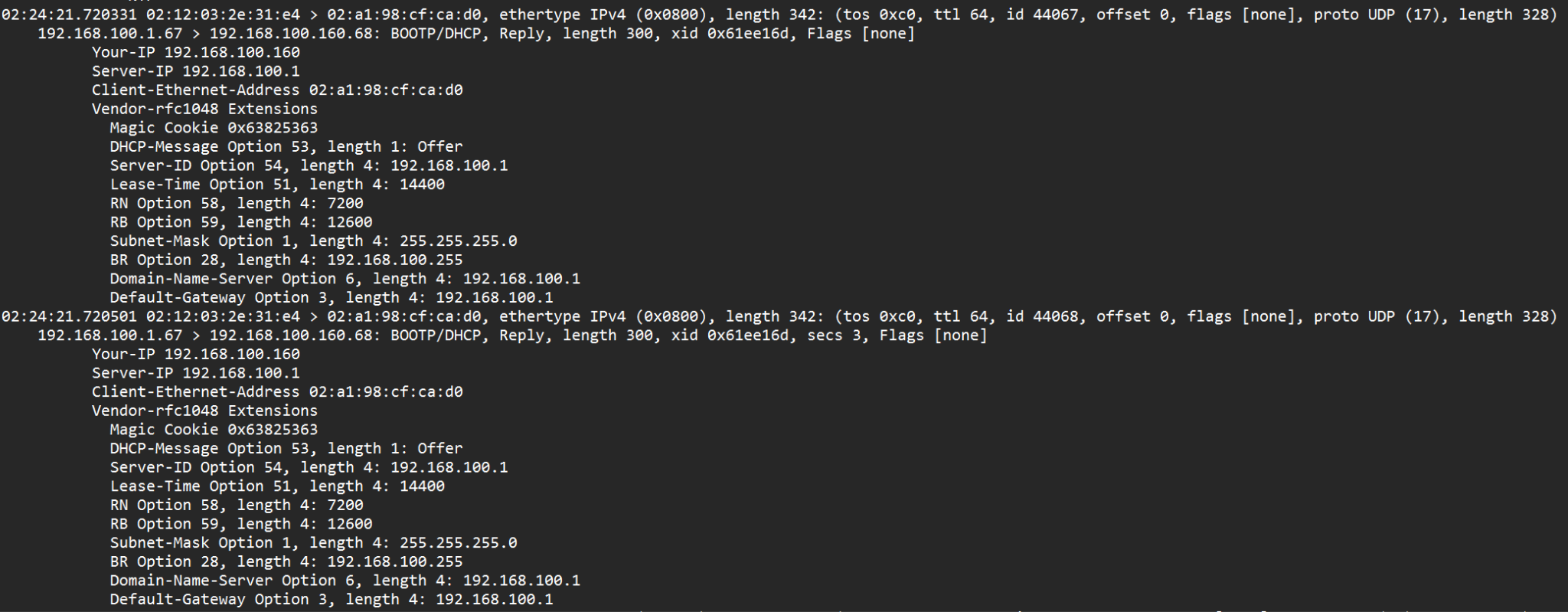
Since the client does not have an IP address yet, the client will send the discover message with the source IP address of “0.0.0.0" (an invalid address). The client also does not know the address of the DHCP server, so it uses the destination as the broadcast IP address "255.255.255.255".

****

Discover by Client

1. The server offered the client with IP address: 192.168.100.160.

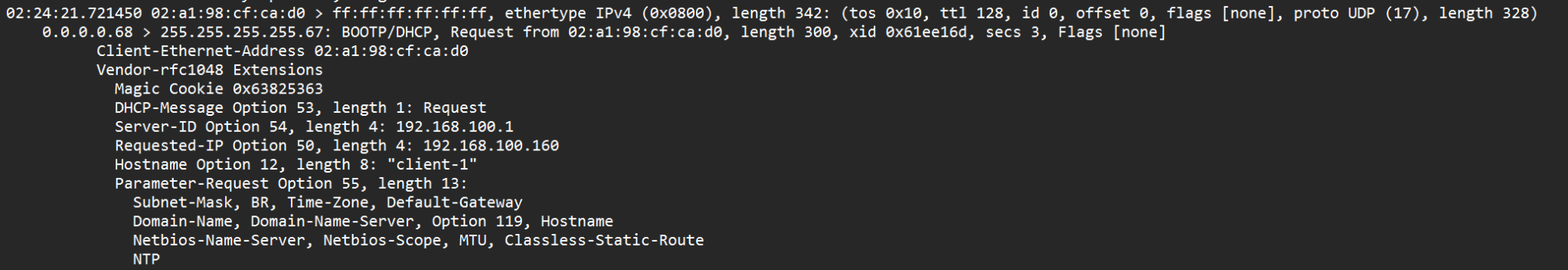
Based on the dnsmasq configuration file, the range of IP provided by the server is 192.168.100.100 - 192.168.100.199.



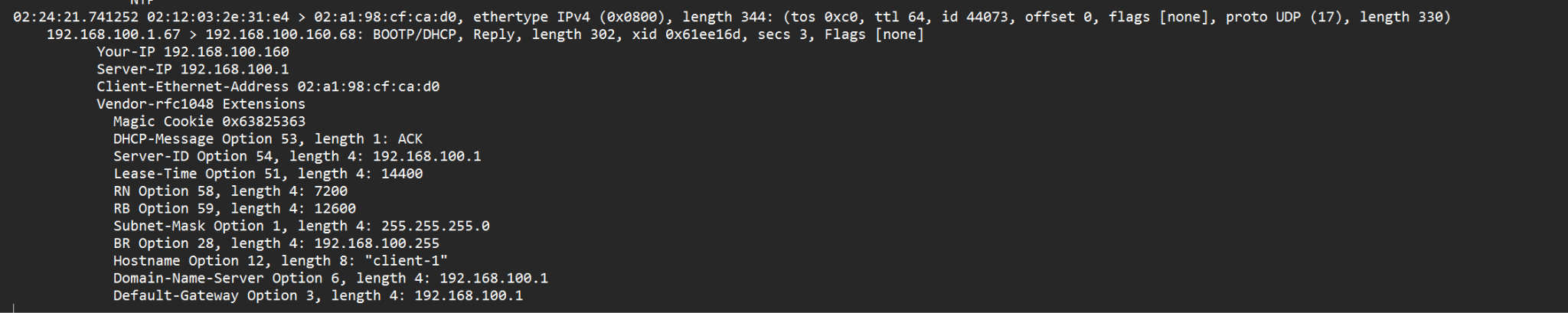
Offer by Server

1. Destination IP of the request: 192.168.100.1

The destination IP is the gateway’s address which the gateway has offered. In this case the client has chosen to take 192.168.100.160 as its IP address.



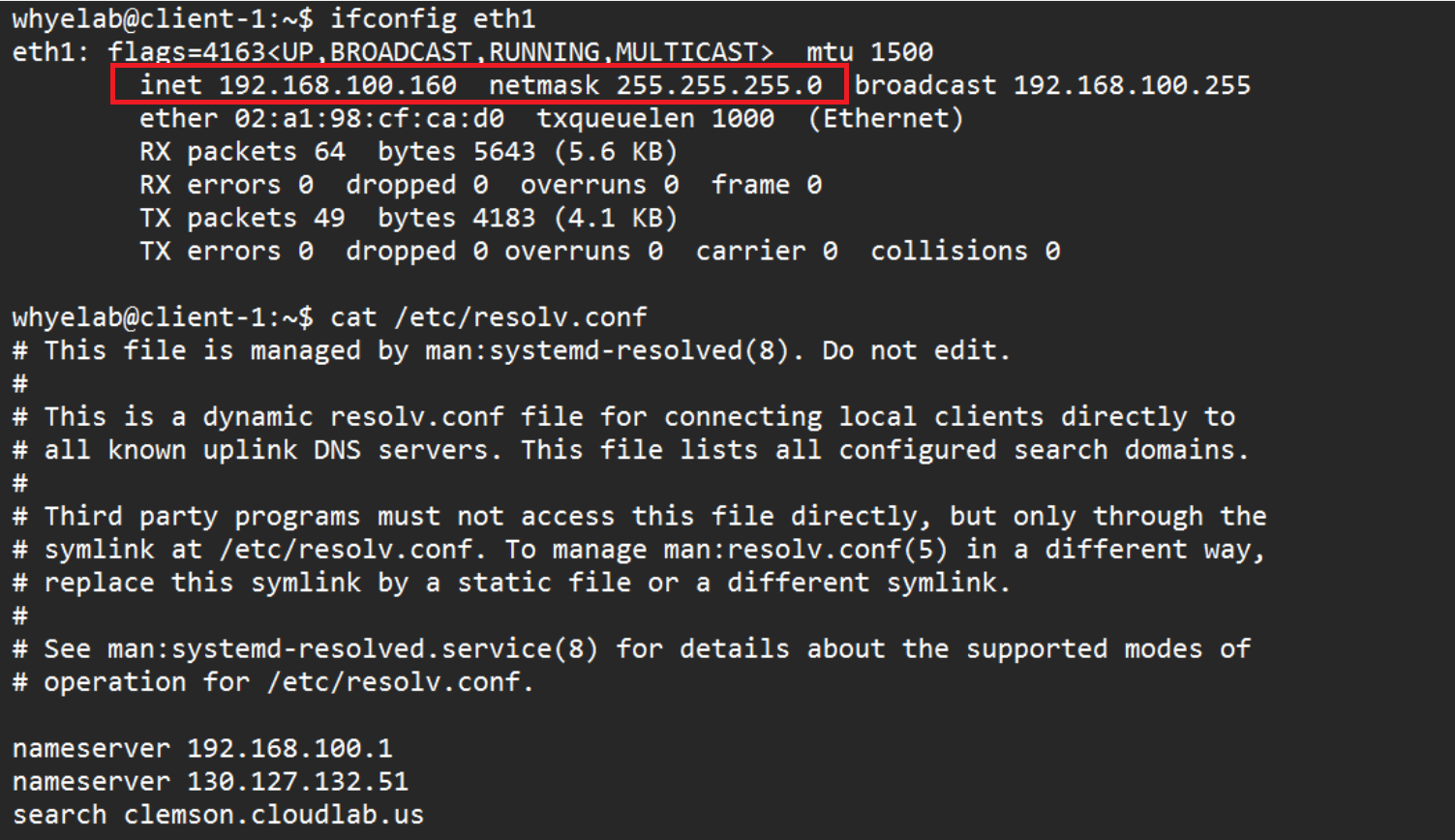
Request by Client



Acknowledge by Server

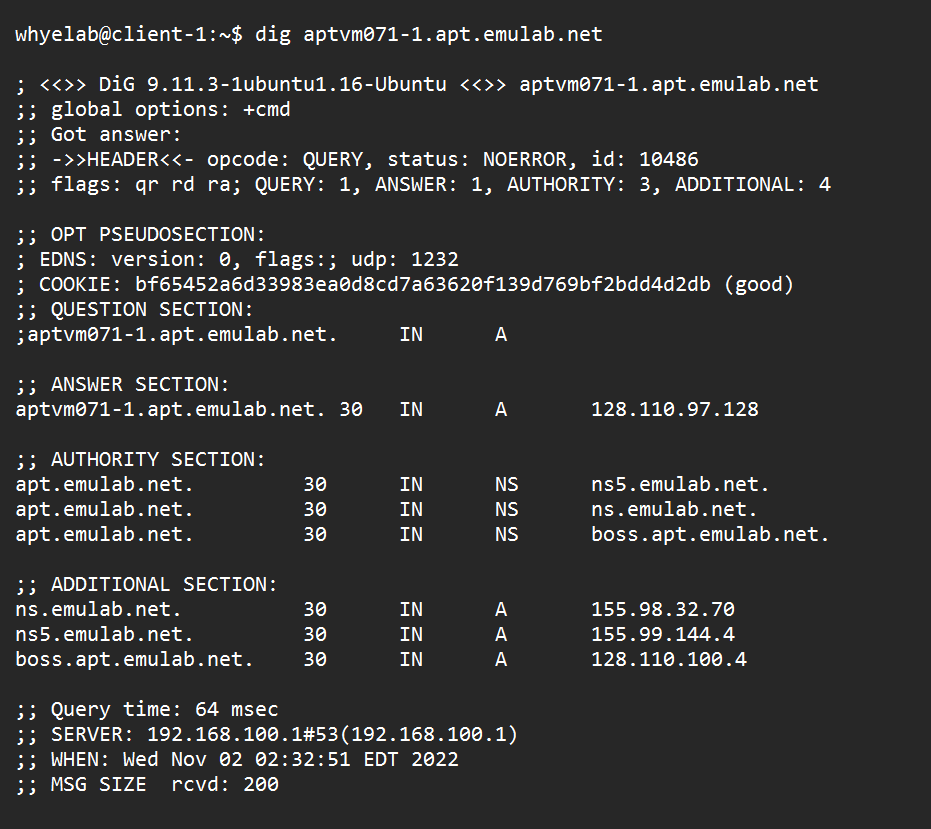
1. The IP address indeed matches the offer that the client has accepted (192.168.100.160).

The server may also include other options, such as the address of a gateway or name server that the client should use.

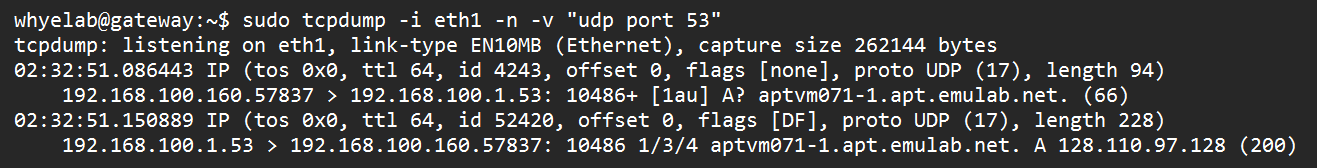


eth1 configuration

## **Exercise 2.2 Observe a DNS Query and Response**



Dig command ran on client



TCP Dump on gateway

(a) The host name I tried to resolve is aptvm071-1.apt.emulab.net



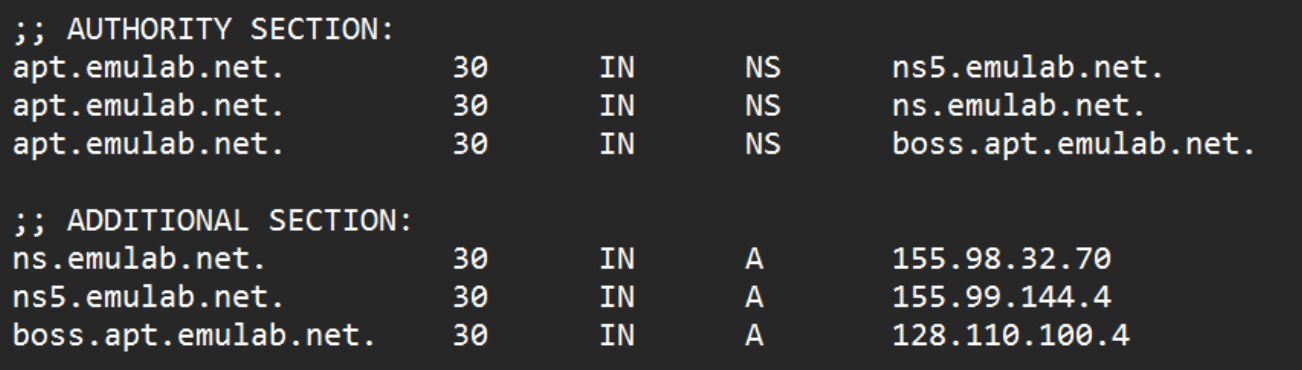
Website Node List View

(b) "A" record type.

(c) No name address is provided, only an IP address of 128.110.97.128 is given.

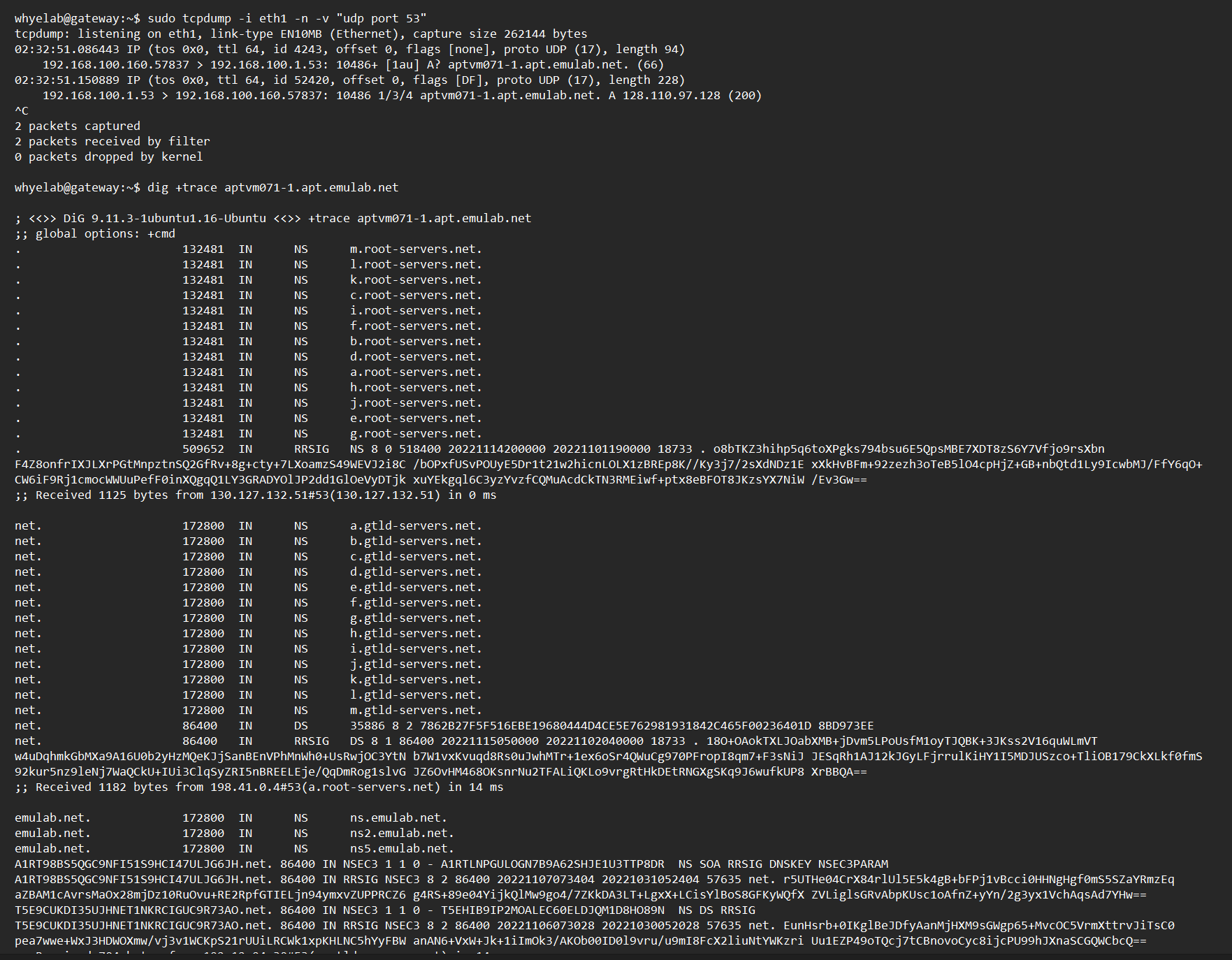
  
Answer Section from “dig” command of client

(d) The first authority server is “apt.emulab.net.” with an IP address of “155.98.32.70”.

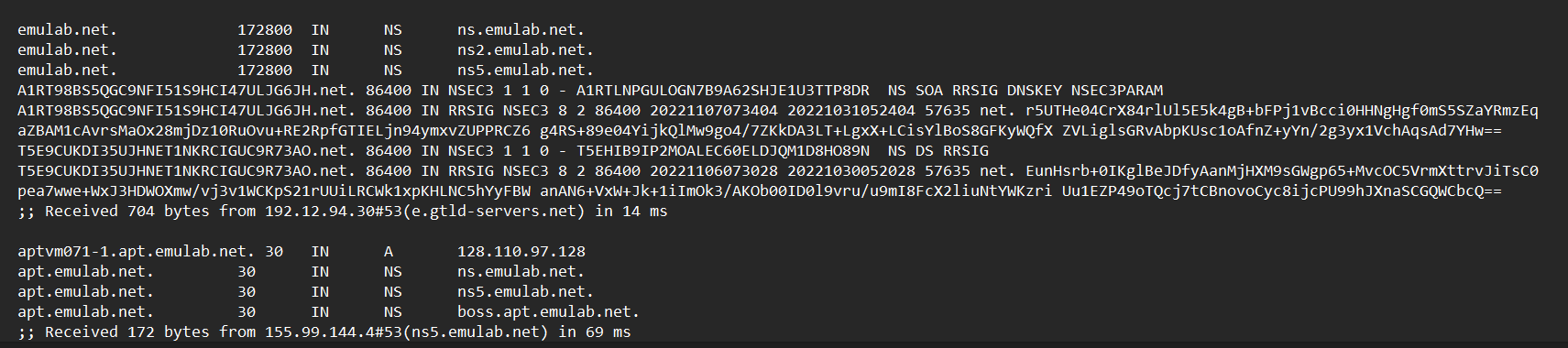


Authority Section and Additional Section

(e) The IP address of the server where the DNS response message is obtained is 128.110.97.128.



DNS resolution with +trace Part 1



DNS resolution with +trace Part 2

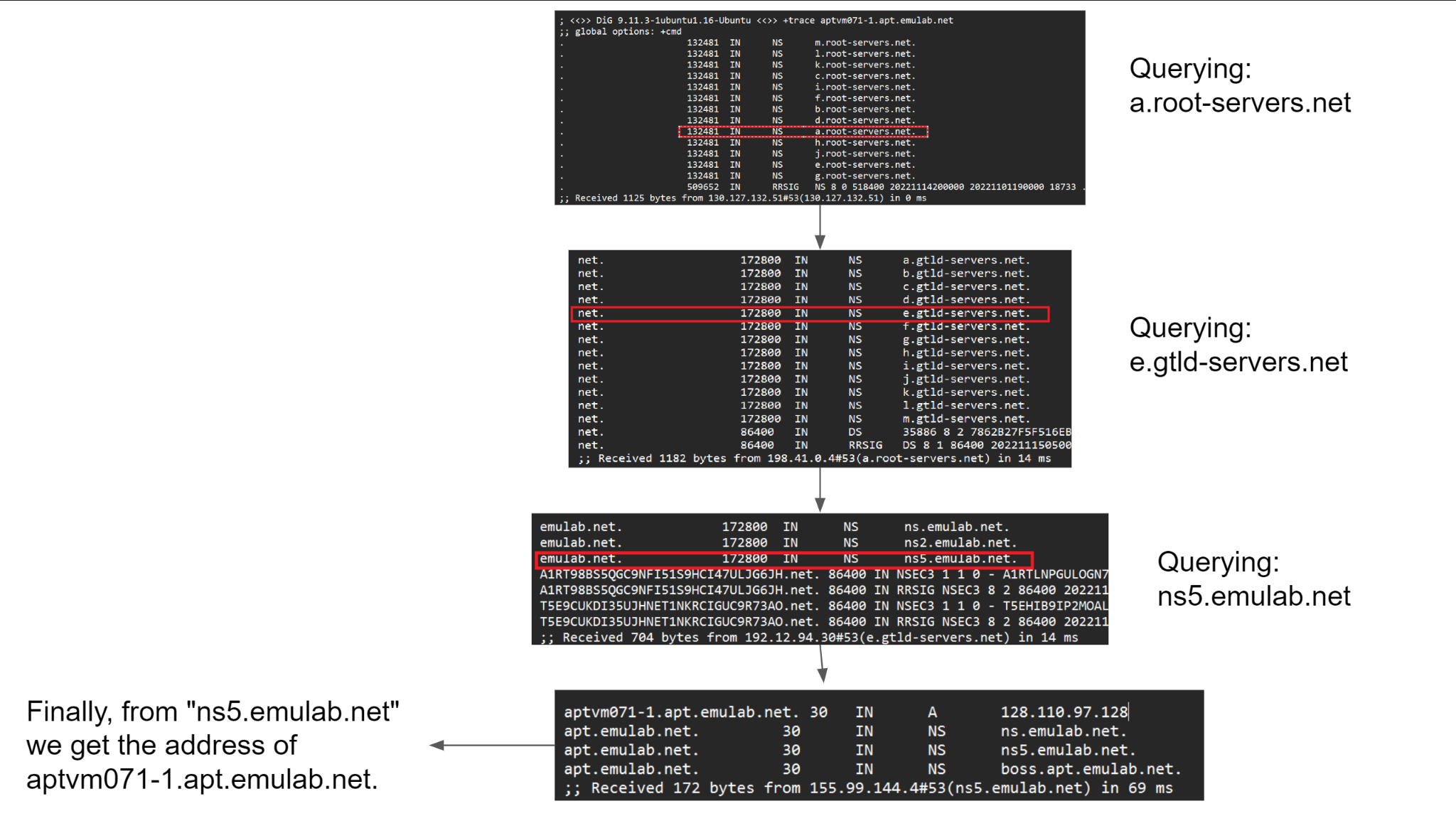
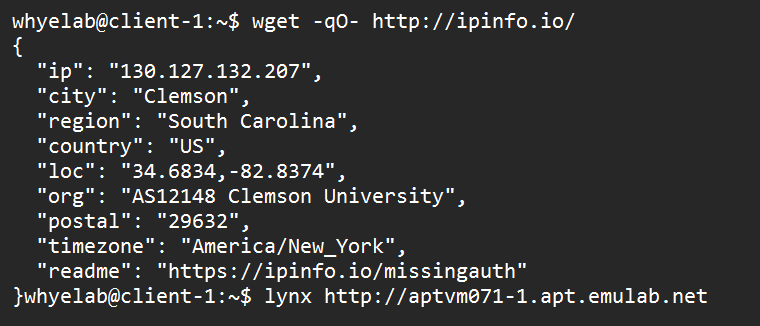


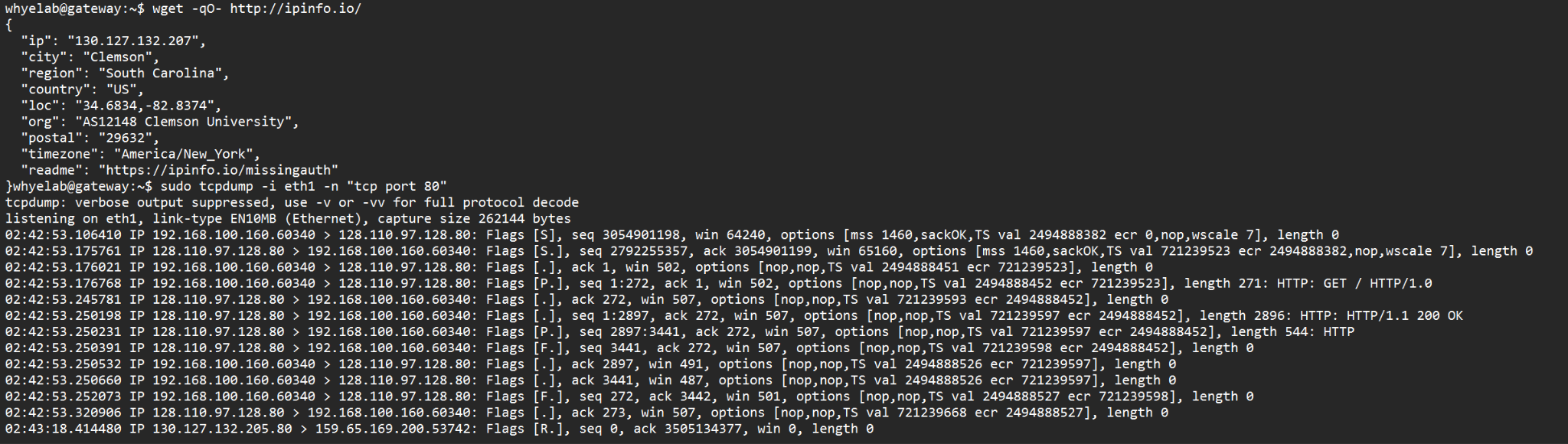
Diagram for showing how hostname is resolved recursively

## **Exercise 2.3 Use NAT**

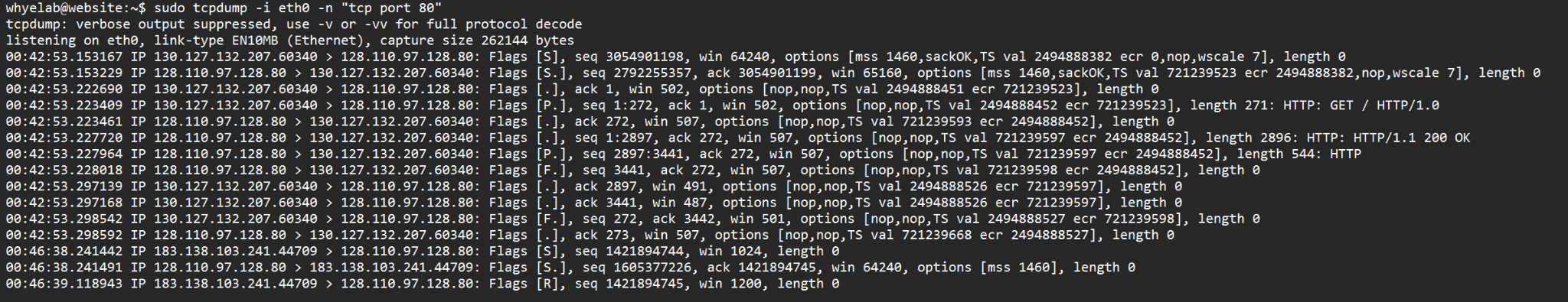
1. The TCP 3 way handshake is as shown below.



Client opening <http://aptvm071-1.apt.emulab.net> from terminal browser



TCP Dump at Gateway



TCP Dump at Website

