Nombre: Kristian Mendoza

Fecha: 16/05/2023

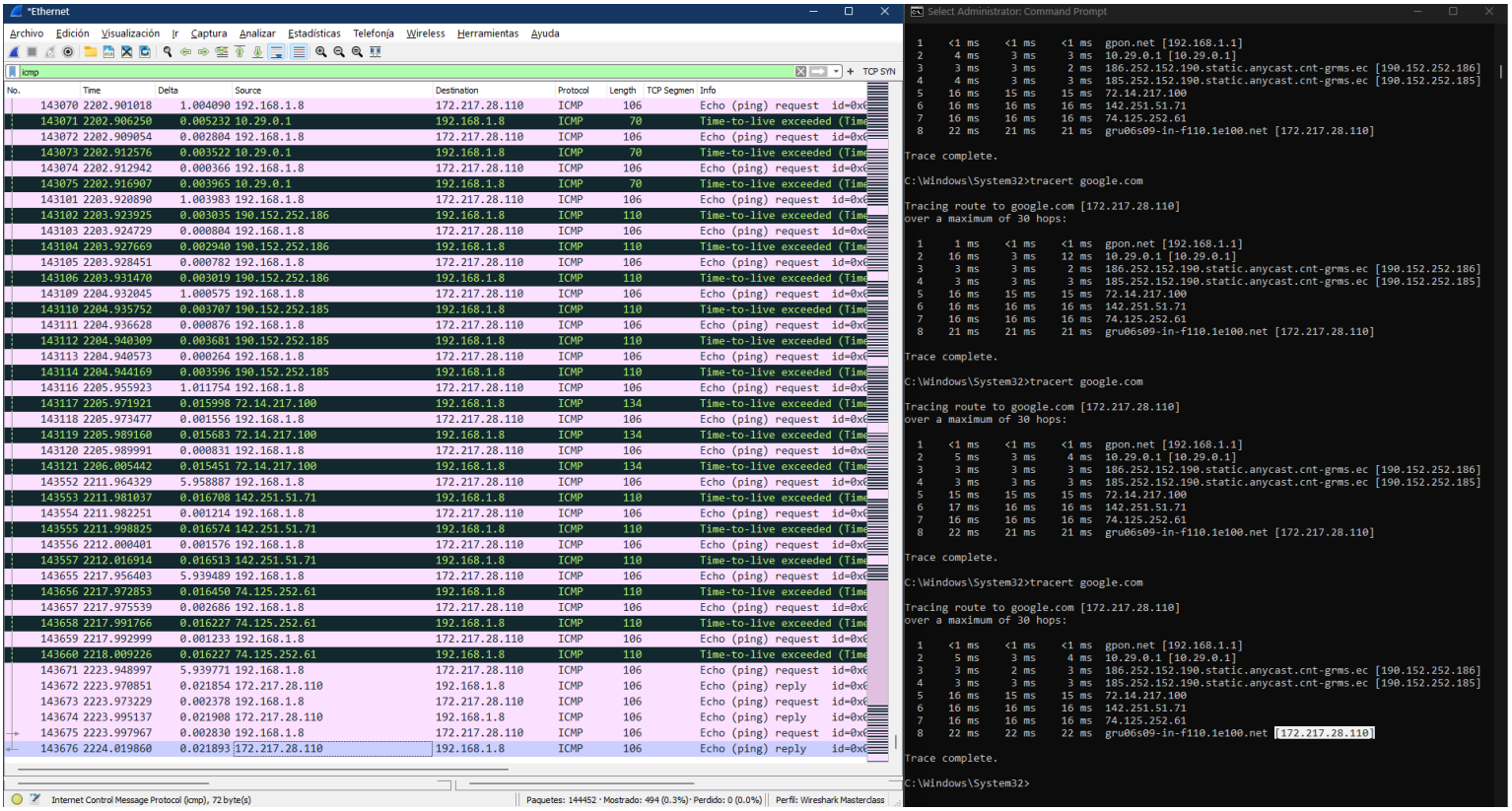
Curso: Redes

NRC: 4005

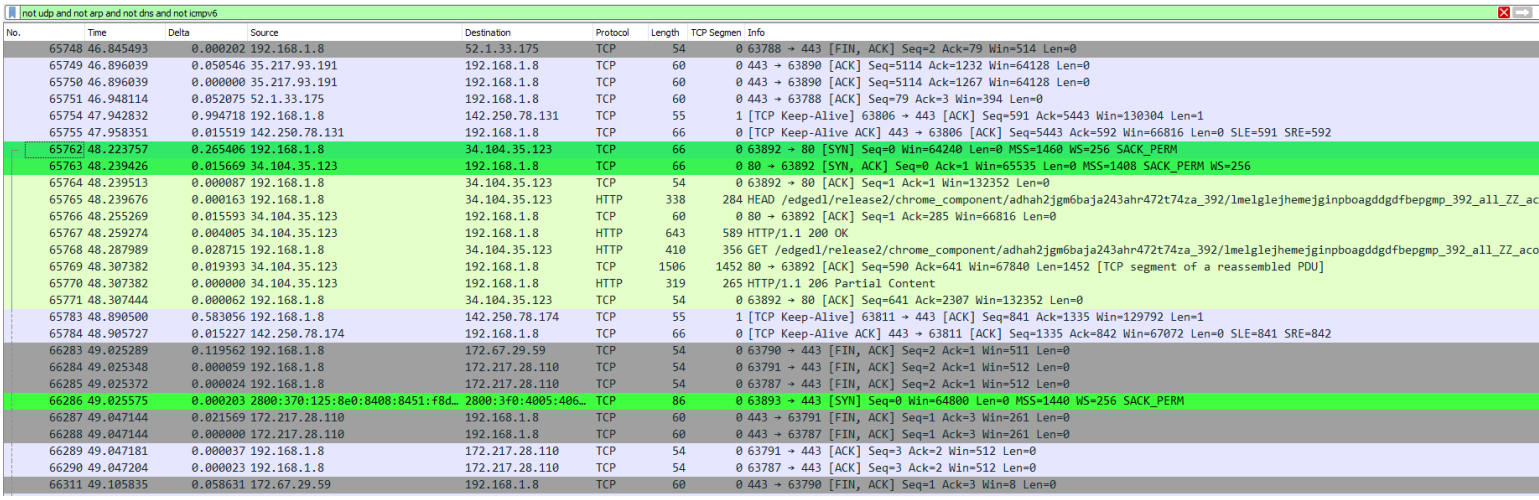
Homework 003

Answer the following questions.
1) Read the following Wireshark tutorial and use it to capture traffic from the following scenarios. Use screenshots to show your results.

a) Run 10 traceroute commands against google.com



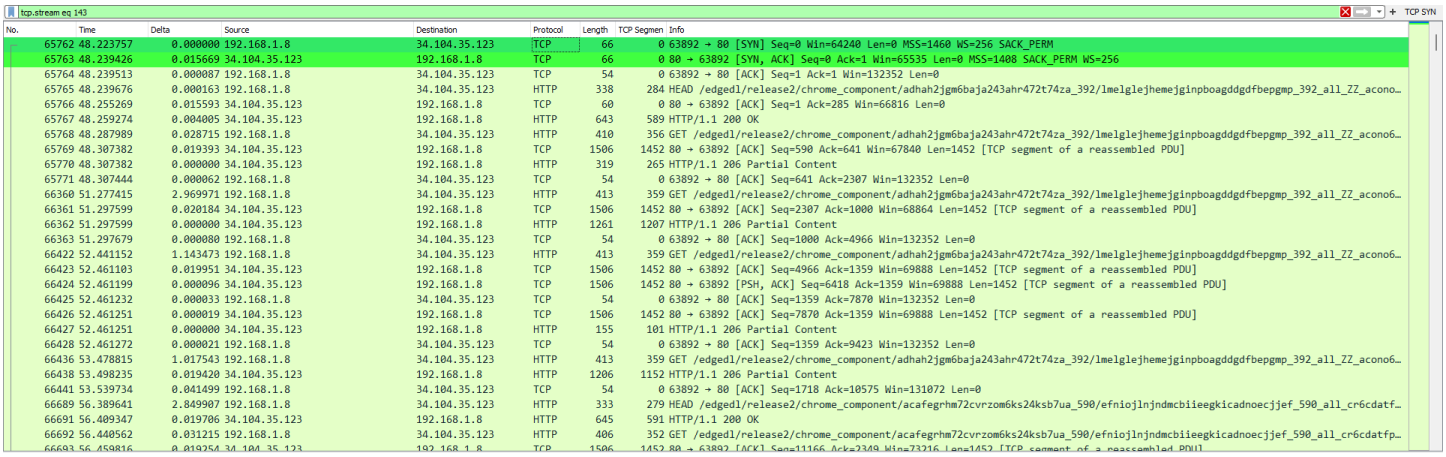
b) Watch a video from youtube.com. Capture the TCP handshake, and the congestion window.



No.	Time	Delta	Source	Destination	Protocol	Length	TCP Segmen	Info
65748	46.845493	0.000202	192.168.1.8	52.1.33.175	TCP	54	0	63788 → 443 [FIN, ACK] Seq=2 Ack=79 Win=514 Len=0
65749	46.896039	0.050546	35.217.93.191	192.168.1.8	TCP	60	0	443 → 63890 [ACK] Seq=5114 Ack=1232 Win=64128 Len=0
65750	46.896039	0.000000	35.217.93.191	192.168.1.8	TCP	60	0	443 → 63890 [ACK] Seq=5114 Ack=1267 Win=64128 Len=0
65751	46.948114	0.052075	52.1.33.175	192.168.1.8	TCP	60	0	443 → 63788 [ACK] Seq=79 Ack=3 Win=394 Len=0
65754	47.942832	0.994718	192.168.1.8	142.250.78.131	TCP	55	1	[TCP Keep-Alive] 63806 → 443 [ACK] Seq=591 Ack=5443 Win=130304 Len=1
65755	47.958351	0.015519	142.250.78.131	192.168.1.8	TCP	66	0	[TCP Keep-Alive ACK] 443 → 63806 [ACK] Seq=5443 Ack=592 Win=66816 Len=0 SLE=591 SRE=592
65762	48.223757	0.265406	192.168.1.8	34.104.35.123	TCP	66	0	63892 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
65763	48.239426	0.015669	34.104.35.123	192.168.1.8	TCP	66	0	80 → 63892 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1408 SACK_PERM WS=256
65764	48.239513	0.000087	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=1 Ack=1 Win=132352 Len=0
65765	48.239676	0.000163	192.168.1.8	34.104.35.123	HTTP	338	284	HEAD /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_ac
65766	48.255269	0.015593	34.104.35.123	192.168.1.8	TCP	60	0	80 → 63892 [ACK] Seq=1 Ack=285 Win=66816 Len=0
65767	48.259274	0.004005	34.104.35.123	192.168.1.8	HTTP	643	589	HTTP/1.1 200 OK
65768	48.287989	0.028715	192.168.1.8	34.104.35.123	HTTP	410	356	GET /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_aco
65769	48.307382	0.019393	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [ACK] Seq=590 Ack=641 Win=67840 Len=1452 [TCP segment of a reassembled PDU]
65770	48.307382	0.000000	34.104.35.123	192.168.1.8	HTTP	319	265	HTTP/1.1 206 Partial Content
65771	48.307444	0.000062	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=641 Ack=2307 Win=132352 Len=0
65783	48.890500	0.583056	192.168.1.8	142.250.78.174	TCP	55	1	[TCP Keep-Alive] 63811 → 443 [ACK] Seq=841 Ack=1335 Win=129792 Len=1
65784	48.905727	0.015227	142.250.78.174	192.168.1.8	TCP	66	0	[TCP Keep-Alive ACK] 443 → 63811 [ACK] Seq=1335 Ack=842 Win=67072 Len=0 SLE=841 SRE=842
66283	49.025289	0.119562	192.168.1.8	172.67.29.59	TCP	54	0	63790 → 443 [FIN, ACK] Seq=2 Ack=1 Win=511 Len=0
66284	49.025348	0.000059	192.168.1.8	172.217.28.110	TCP	54	0	63791 → 443 [FIN, ACK] Seq=2 Ack=1 Win=512 Len=0
66285	49.025372	0.000024	192.168.1.8	172.217.28.110	TCP	54	0	63787 → 443 [FIN, ACK] Seq=2 Ack=1 Win=512 Len=0
66286	49.025575	0.000203	2800:370:125:8e0:8408:8451:f8d…	2800:3f0:4005:406…	TCP	86	0	63893 → 443 [SYN] Seq=0 Win=64800 Len=0 MSS=1440 WS=256 SACK_PERM
66287	49.047144	0.021569	172.217.28.110	192.168.1.8	TCP	60	0	443 → 63791 [FIN, ACK] Seq=1 Ack=3 Win=261 Len=0
66288	49.047144	0.000000	172.217.28.110	192.168.1.8	TCP	60	0	443 → 63787 [FIN, ACK] Seq=1 Ack=3 Win=261 Len=0
66289	49.047181	0.000037	192.168.1.8	172.217.28.110	TCP	54	0	63791 → 443 [ACK] Seq=3 Ack=2 Win=512 Len=0
66290	49.047204	0.000023	192.168.1.8	172.217.28.110	TCP	54	0	63787 → 443 [ACK] Seq=3 Ack=2 Win=512 Len=0
66311	49.105835	0.058631	172.67.29.59	192.168.1.8	TCP	60	0	443 → 63790 [FIN, ACK] Seq=1 Ack=3 Win=8 Len=0

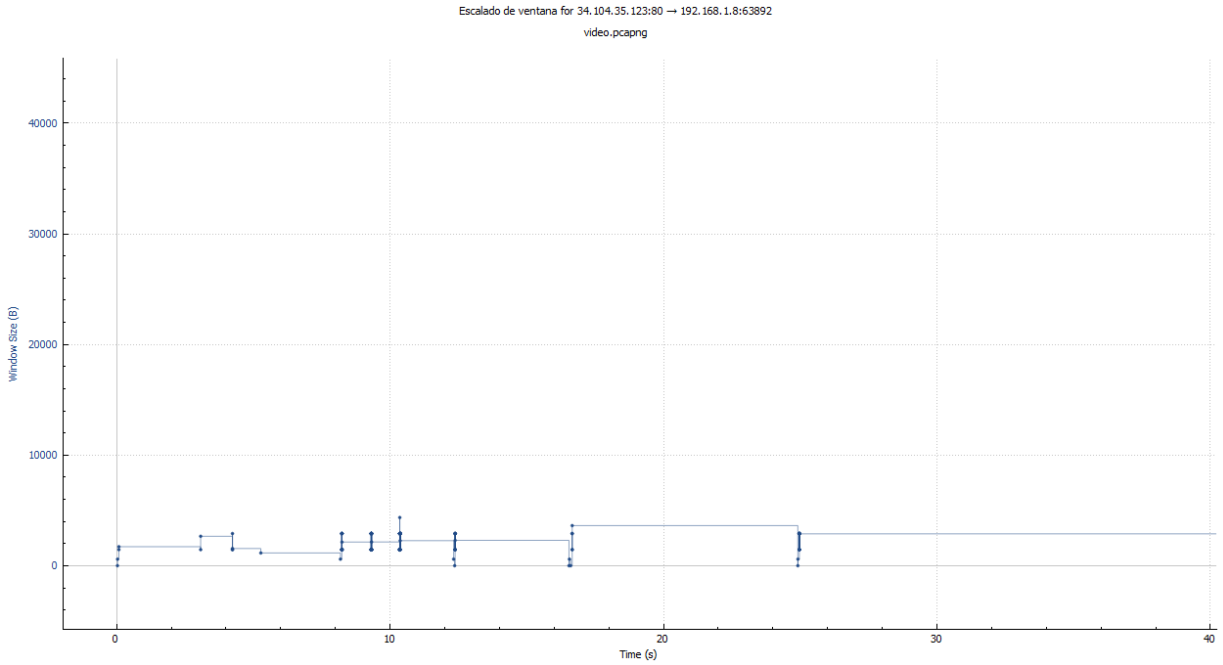
Se puede ver que el paquete 65762 es el inicio del handshake con el host que tiene la ip 34.104.35.123, la cual es perteneciente a Google.

Y luego al hacer un "follow TCP stream" se obtiene esta versión filtrada de la conexión:

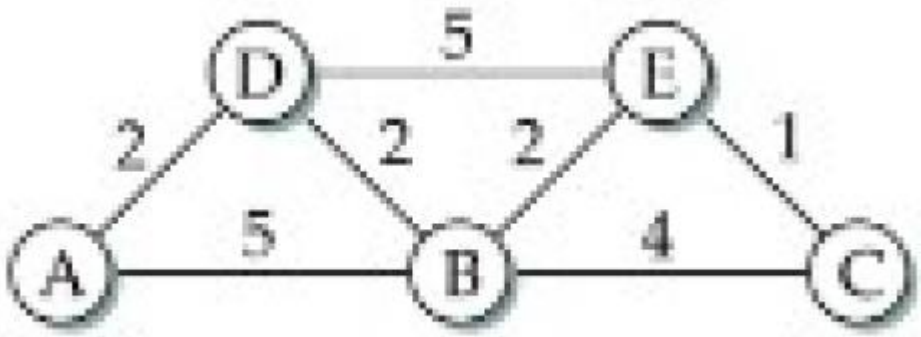


No.	Time	Delta	Source	Destination	Protocol	Length	TCP Segmen	Info
65762	48.223757	0.000000	192.168.1.8	34.104.35.123	TCP	66	0	63892 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
65763	48.239426	0.015669	34.104.35.123	192.168.1.8	TCP	66	0	80 → 63892 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1408 SACK_PERM WS=256
65764	48.239513	0.000087	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=1 Ack=1 Win=132352 Len=0
65765	48.239676	0.000163	192.168.1.8	34.104.35.123	HTTP	338	284	HEAD /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_acono…
65766	48.255269	0.015593	34.104.35.123	192.168.1.8	TCP	60	0	80 → 63892 [ACK] Seq=1 Ack=285 Win=66816 Len=0
65767	48.259274	0.004005	34.104.35.123	192.168.1.8	HTTP	643	589	HTTP/1.1 200 OK
65768	48.287989	0.028715	192.168.1.8	34.104.35.123	HTTP	410	356	GET /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_acono6…
65769	48.307382	0.019393	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [ACK] Seq=590 Ack=641 Win=67840 Len=1452 [TCP segment of a reassembled PDU]
65770	48.307382	0.000000	34.104.35.123	192.168.1.8	HTTP	319	265	HTTP/1.1 206 Partial Content
65771	48.307444	0.000062	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=641 Ack=2307 Win=132352 Len=0
66360	51.277415	2.969971	192.168.1.8	34.104.35.123	HTTP	413	359	GET /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_acono6…
66361	51.297599	0.020184	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [ACK] Seq=2307 Ack=1000 Win=68864 Len=1452 [TCP segment of a reassembled PDU]
66362	51.297599	0.000000	34.104.35.123	192.168.1.8	HTTP	1261	1207	HTTP/1.1 206 Partial Content
66363	51.297679	0.000080	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=1000 Ack=4966 Win=132352 Len=0
66422	52.441152	1.143473	192.168.1.8	34.104.35.123	HTTP	413	359	GET /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_acono6…
66423	52.461103	0.019951	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [ACK] Seq=4966 Ack=1359 Win=69888 Len=1452 [TCP segment of a reassembled PDU]
66424	52.461199	0.000096	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [PSH, ACK] Seq=6418 Ack=1359 Win=69888 Len=1452 [TCP segment of a reassembled PDU]
66425	52.461232	0.000033	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=1359 Ack=7870 Win=132352 Len=0
66426	52.461251	0.000019	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [ACK] Seq=7870 Ack=1359 Win=69888 Len=1452 [TCP segment of a reassembled PDU]
66427	52.461251	0.000000	34.104.35.123	192.168.1.8	HTTP	155	101	HTTP/1.1 206 Partial Content
66428	52.461272	0.000021	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=1359 Ack=9423 Win=132352 Len=0
66436	53.478815	1.017543	192.168.1.8	34.104.35.123	HTTP	413	359	GET /edgedl/release2/chrome_component/adhah2jgm6baja243ahr472t74za_392/lmelglejhemejginpboagddgdfbepgmp_392_all_ZZ_acono6…
66438	53.498235	0.019420	34.104.35.123	192.168.1.8	HTTP	1206	1152	HTTP/1.1 206 Partial Content
66441	53.539734	0.041499	192.168.1.8	34.104.35.123	TCP	54	0	63892 → 80 [ACK] Seq=1718 Ack=10575 Win=131072 Len=0
66689	56.389641	2.849907	192.168.1.8	34.104.35.123	HTTP	333	279	HEAD /edgedl/release2/chrome_component/acafegrhm72cvrzom6ks24ksb7ua_590/efniojlnjndmcbiieegkicadnoecjjef_590_all_cr6cdatf…
66691	56.409347	0.019706	34.104.35.123	192.168.1.8	HTTP	645	591	HTTP/1.1 200 OK
66692	56.440562	0.031215	192.168.1.8	34.104.35.123	HTTP	406	352	GET /edgedl/release2/chrome_component/acafegrhm72cvrzom6ks24ksb7ua_590/efniojlnjndmcbiieegkicadnoecjjef_590_all_cr6cdatfp…
66693	56.459816	0.019254	34.104.35.123	192.168.1.8	TCP	1506	1452	80 → 63892 [ACK] Seq=11166 Ack=2349 Win=73216 Len=1452 [TCP segment of a reassembled PDU]

Y la congestion Window se ve algo así:

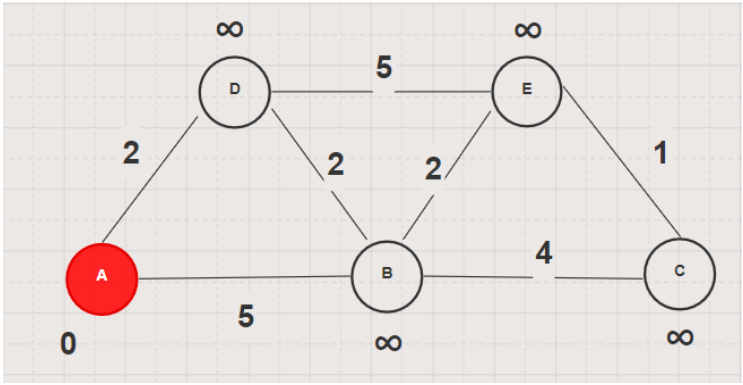


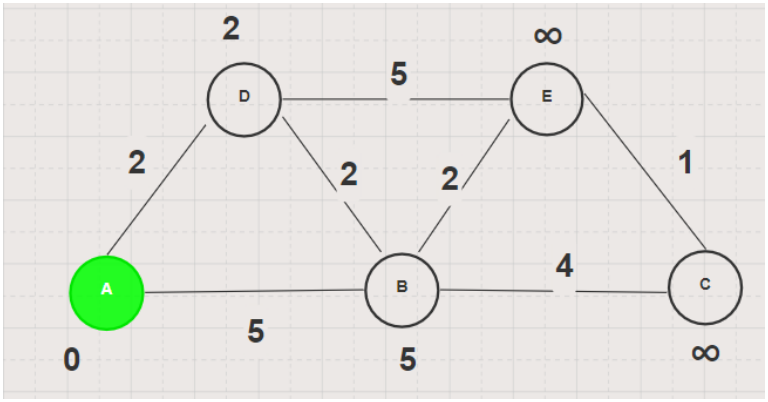
2) Use Dijkstra's to get the routing tables for nodes A, B and E.

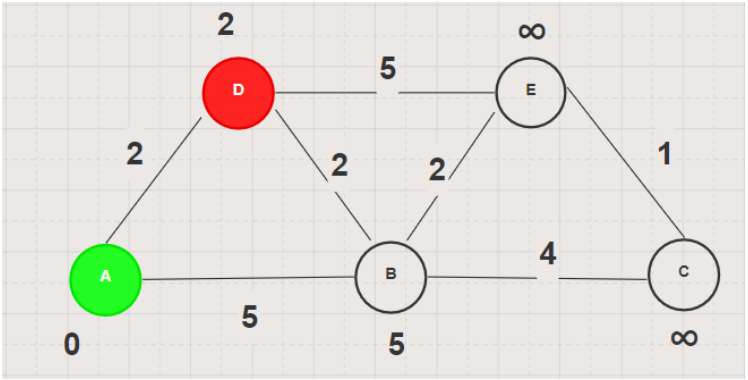


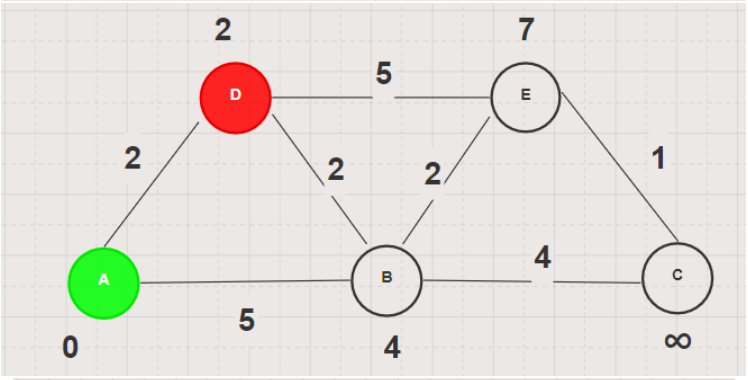
Nodo A

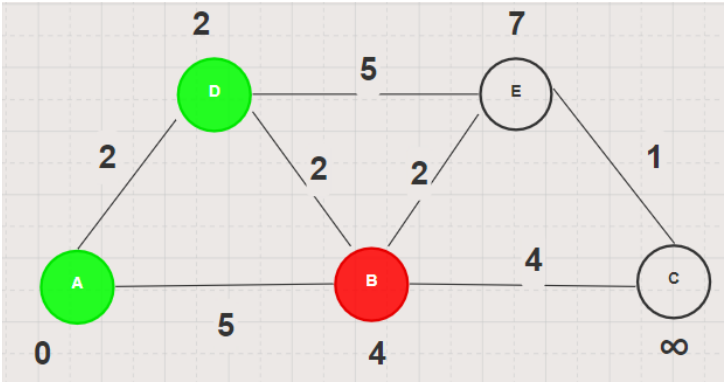
Se van a exponer los pasos del Algoritmo de Dijkstra. Los nodos visitados serán pintados con verde y los tentativos son los rojos

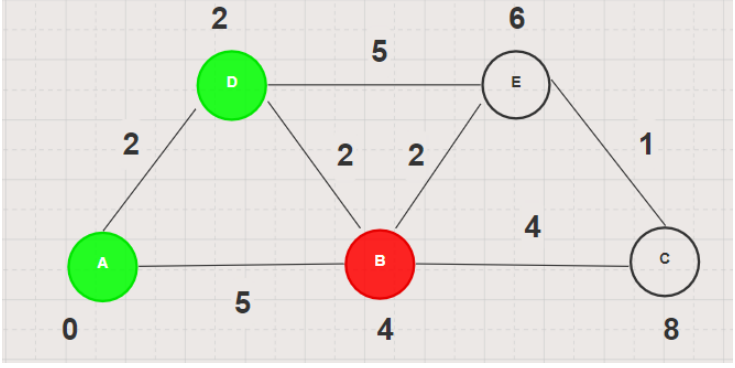


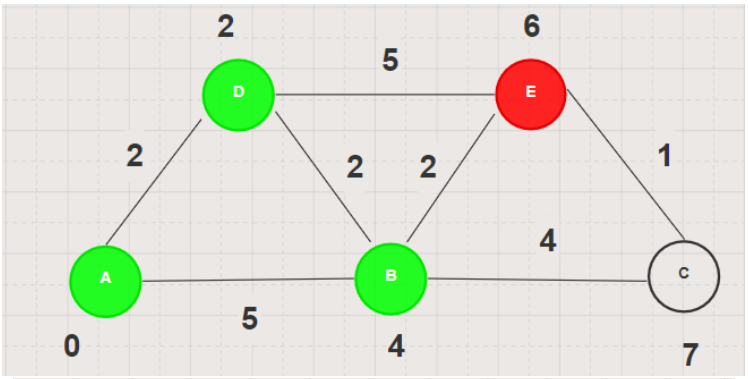

2
∞
D
5
E
2
2
2
1
A
B
4
C
5
0
5
∞

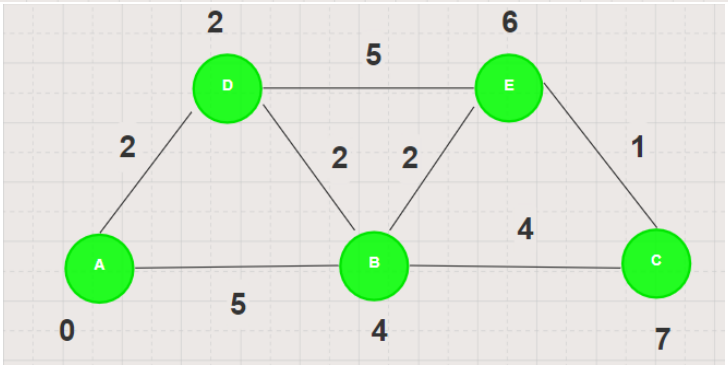

2
∞
D
5
E
2
2
2
1
A
B
4
C
5
0
5
∞


2
7
D
5
E
2
2
2
1
A
B
4
C
5
0
4
∞


2
7
D
5
E
2
2
2
1
A
B
4
C
5
0
4
∞


2
6
D
5
E
2
2
2
1
A
B
4
C
5
0
4
8

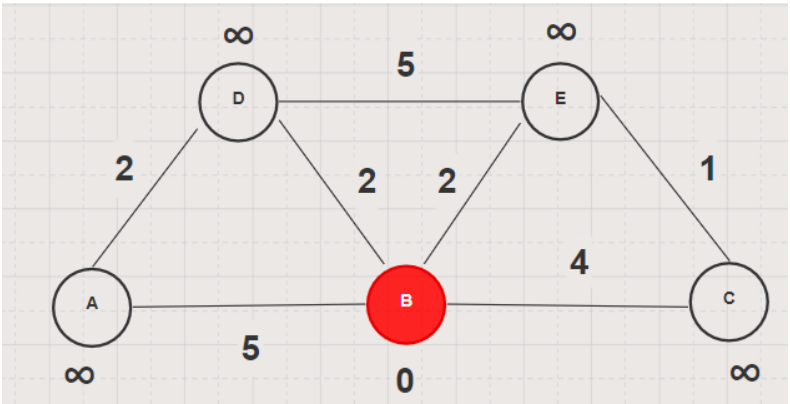


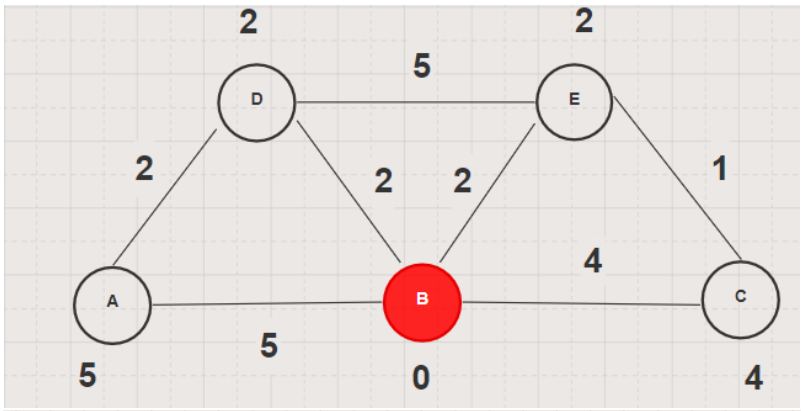


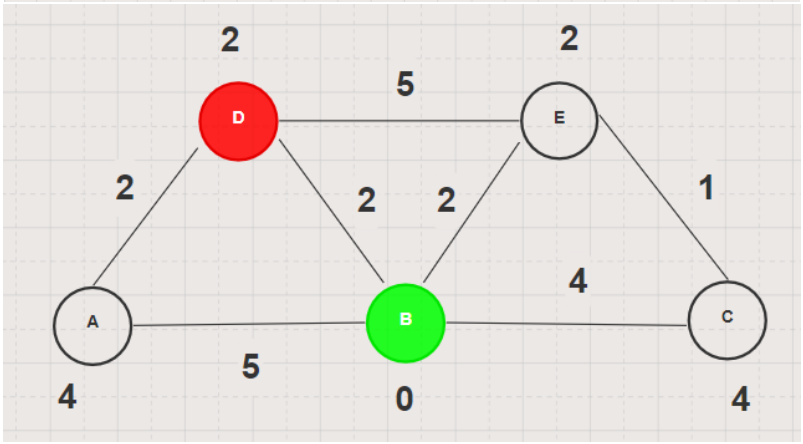
Entonces la forwarding table es la siguiente:

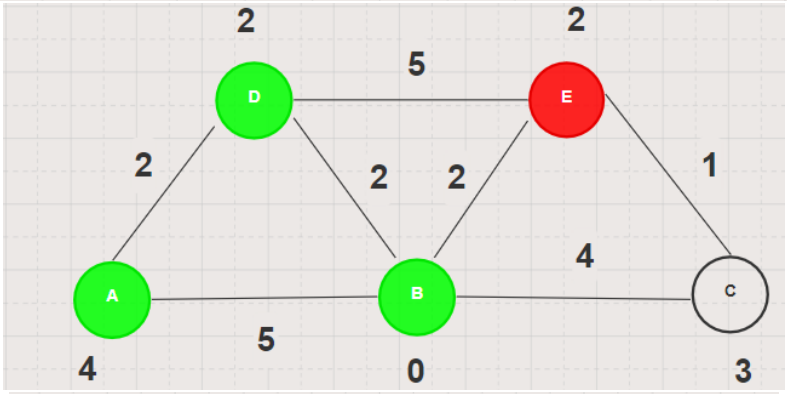
Destination	Next Hop	Cost
B	D	4
D	D	2
E	D	6
C	D	7

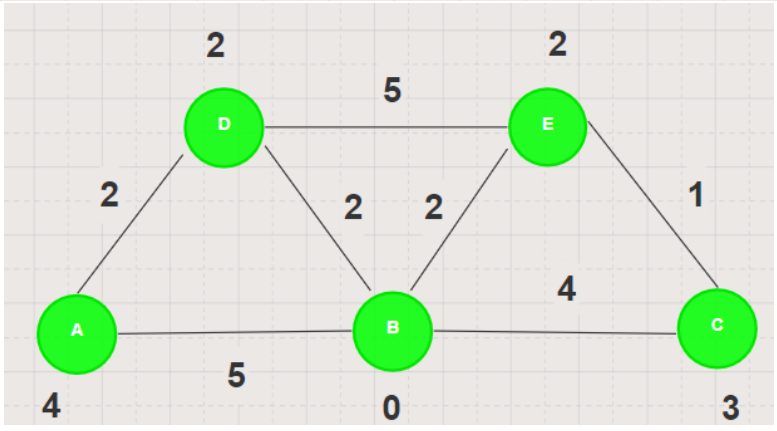
Nodo B







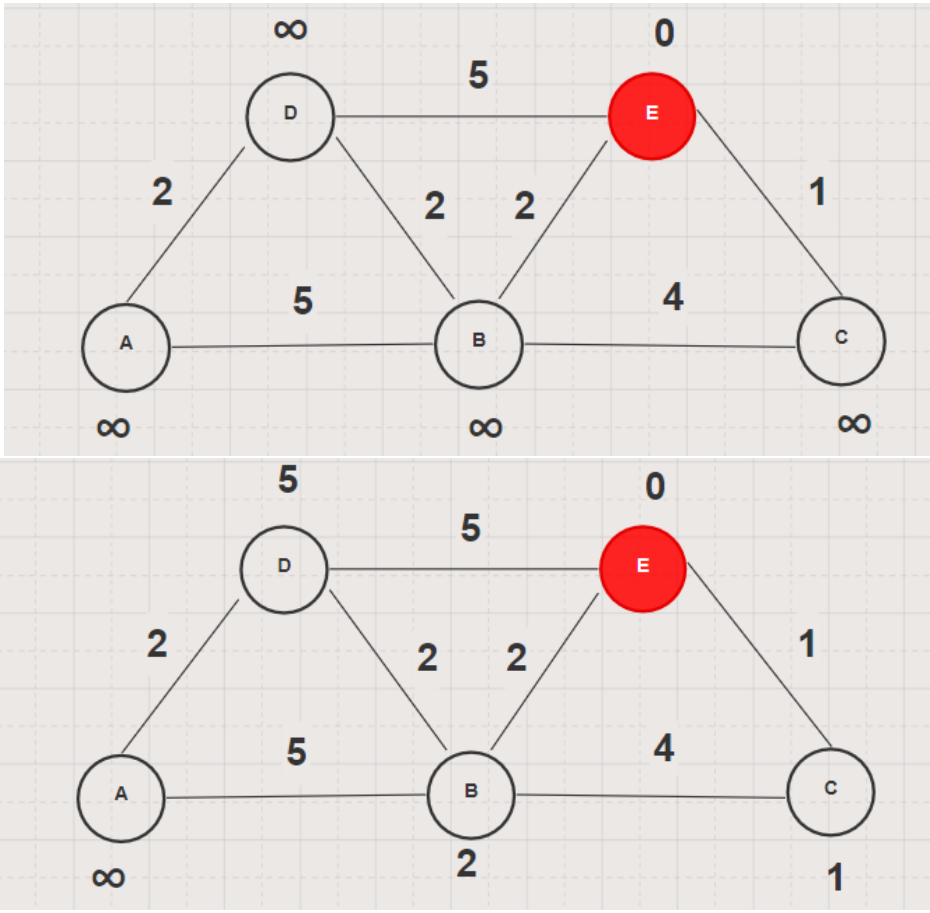


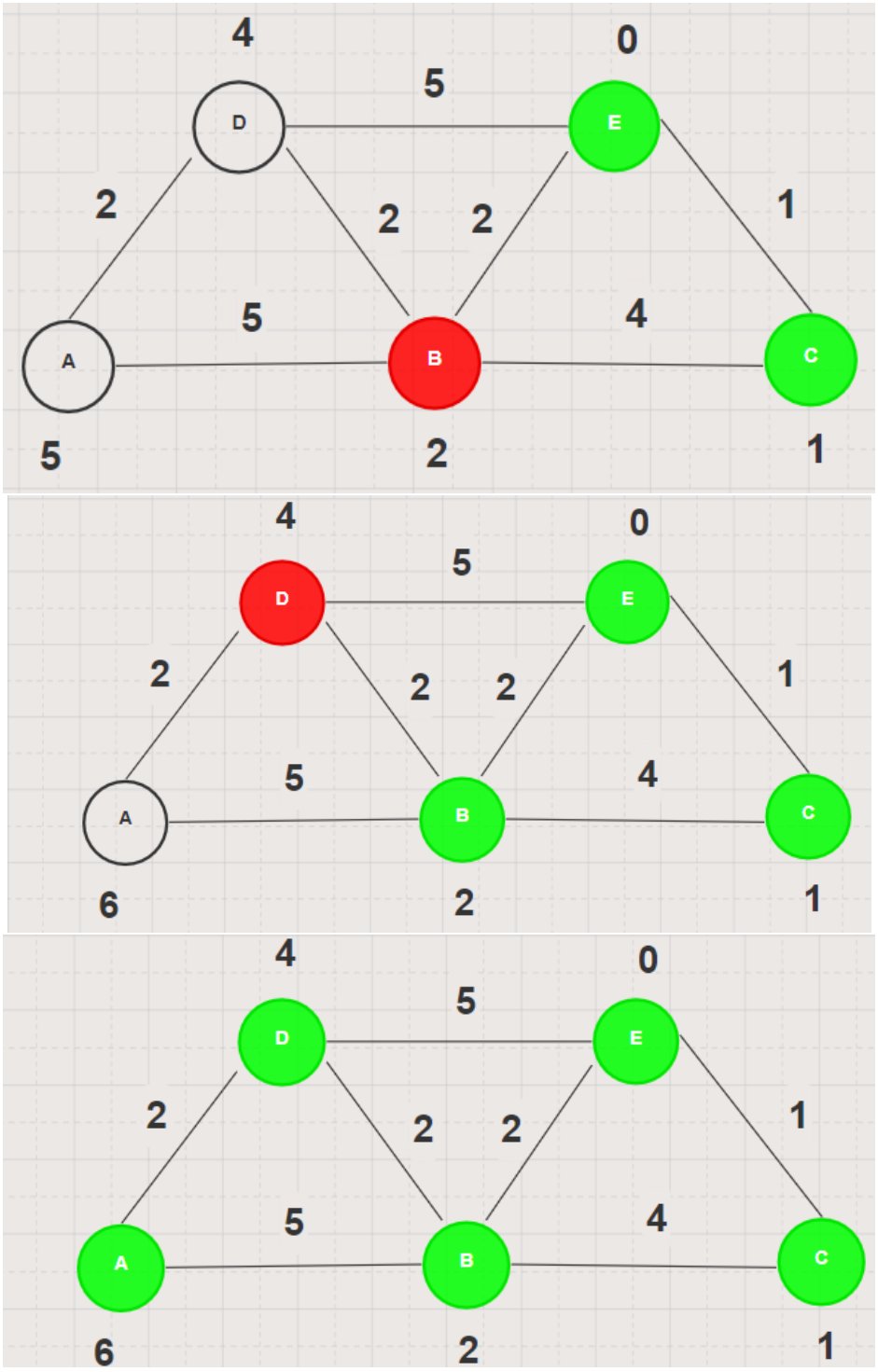


Y la forwarding table sería:

Destination	Next Hop	Cost
A	D	4
D	D	2
E	E	2
C	E	3

Nodo E





Y el forwarding table sería:

Destination	Next Hop	Cost
A	B	6
D	B	4
C	C	1

B	B	2

3) Suppose a host wants to establish the reliability of a link by sending packets and measuring the percentage that are received; routers, for example, do this. Explain the difficulty of doing this over a TCP connection.

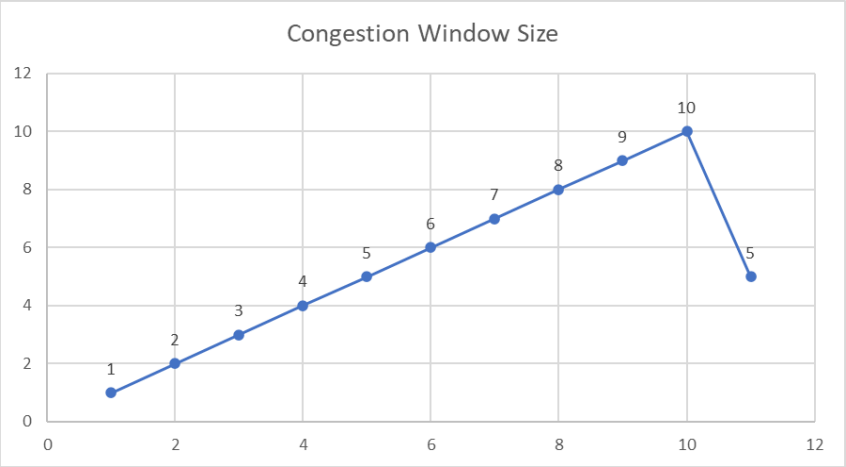
Los problemas de intentar establecer la fiabilidad de la red con una conexión de TCP es que al tener medios para reenviar paquetes por si no se reciben aknowledgments tales como los time outs o los métodos existentes en el slidiwing window protocol, se puede de alguna manera "cubrir" que está habiendo una perdida de paquetes en la red. Además de que con el control de congestión de TCP pueden haber confusiones entre perder paquetes por deficiencias de una conexión, o perderlos por que hay demasiados paquetes y no entran en el congestion window.

4) Consider a simple congestion control algorithm that uses linear increase and multiplicative decrease (no slow start). Assume the congestion window size is in units of packets rather than bytes, and it is one packet initially.

a) Give a detailed sketch of this algorithm.

Ya que se considera un algoritmo simple, se van a especificar los valor de los factores de adición y multiplicación como: 1 para el linear increase y 2 para el multiplicative decrease. Con esto se puede describir el funcionamiento del algoritmo cómo:

1. Se empieza la Congestion Window con 1 paquete
2. Si se recibe el ACK del paquete se aumenta el tamaño de la Congestion Window en 1, ahora siendo de 2 paquetes.
3. Se envian dos paquetes y si se reciben los dos ACKS, se aumenta el tamaño de la Congestion Window en 1, ahora siendo de 3.
4. El algoritmo continua añadiendo 1 paquete al Congestion Window por cada vez que se reciben todos los ACKs de los paquetes que se enviaron.
5. Si en algún momento, no se recibe un paquete ya sea porque pasó el tiempo de timeout o se perdió el ACK, se divide el número de paquetes del Congestion Window para 2. Por ejemplo, si la ventana era de tamaño 10, ahora pasa a ser de tamaño 5.



b) Assume the delay is latency only, and that when a group of packets is sent, only a single ACK is returned.

Si solo se recibe un ACK por grupo de paquetes enviados, no hay efectos negativos en el incremento lineal del algoritmo, ya que realmente no importa la cantidad de ACKs que llegan, sino, si llega el ack que significa que hubo éxito en enviar los paquetes. El problema surge cuando se piede un paquete. Ya que no se tiene ACKs individuales, si no llega un ACK, se va a suponer que todo el grupo de paquetes debe ser reenviado, y se disminuye el tamaño del Congestion Window a la mitad. Por lo que esto lo vuelve menos eficiente.

c) Plot the congestion window as a function of RTT for the situation in which the following packets are lost: 9, 25, 30, 38 and 50. For simplicity, assume a perfect timeout mechanism that detects a lost packet exactly 1 RTT after it is transmitted.

Se va a realizar una tabla para mostrar cómo cambia el tamaño de la congestion window en función del RTT y los paquetes que se pierden

RTT	Paquetes enviados	Congestion window Size
1	1	1
2	2,3	2
3	4,5,6	3
4	7,8,9,10	4
5	9,10	2
6	11,12,13	3
7	14,15,16,17	4
8	18,19,20,21,22	5
9	23,24,25,26,27,28	6
10	25,26,27	3
11	28,29,30,31	4
12	30,31	2
13	32,33,34	3
14	35,36,37,38	4
15	38,39	2
16	40,41,42	3

17	43,44,45,46	4
18	47,48,49,50,51	5
19	50,51	2