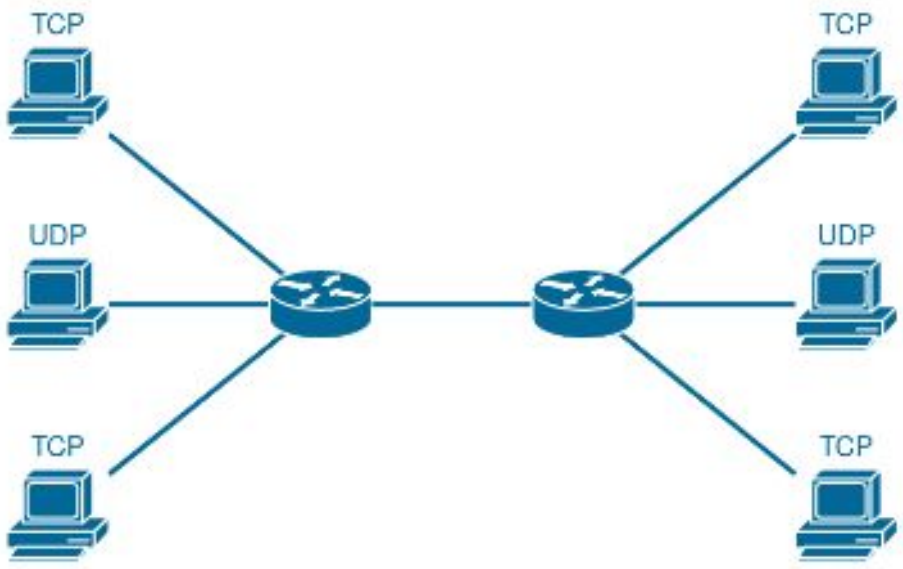
Universidad Nacional de General Sarmiento

Trabajo Práctico N° 2

Sistemas Operativos y Redes II

Análisis de redes

TCP y UDP (opción 1)



Docentes

Alexander, Agustín
Gutierrez, Pedro

Alumnos:

Costilla, Germán Alfredo - costillagermanalfredo@gmail.com
Dougan, Vanesa Solange - vanesa.dougan@hotmail.com
Sosa, Noelia - noesosa88@gmail.com

Segundo cuatrimestre 2020

Parte 1

- Hacer pruebas solo con los 2 emisores TCP.
- a) Hacer que sature el canal. Medir esa velocidad de transferencia (la cantidad de paquetes que llegan a destino).
- b) Mostrar mediante gráficos, tamaño de colas de recepción, ventana de TCP y cualquier mecanismo que muestre lo que sucede.
- c) Explicar en el gráfico las distintas etapas del protocolo TCP.
- d) Calcular el ancho de banda del canal. Explicar qué sucede. ¿Ve alguna anomalía ? Explicarla.
- Realizar pruebas (con la misma configuración), pero ahora con el nodo UDP también emitiendo.
- a) Explicar qué sucede con el ancho de banda utilizado por cada uno.
- b) Mostrar en los paquetes TCP dónde se ven las distintas acciones del protocolo

Parte 2

- Funcionamiento del algoritmo Hybla
- Realizar una configuración de red como la definida en la parte 1 del desarrollo pero sólo con nodos TCP.
- Implementar el algoritmo.
- Presentar datos (tomados por wireshark o ns-3) de las corridas realizadas que muestren que el funcionamiento (según la teoría) es correcto

Parte 1

Hacer pruebas solo con los 2 emisores TCP.

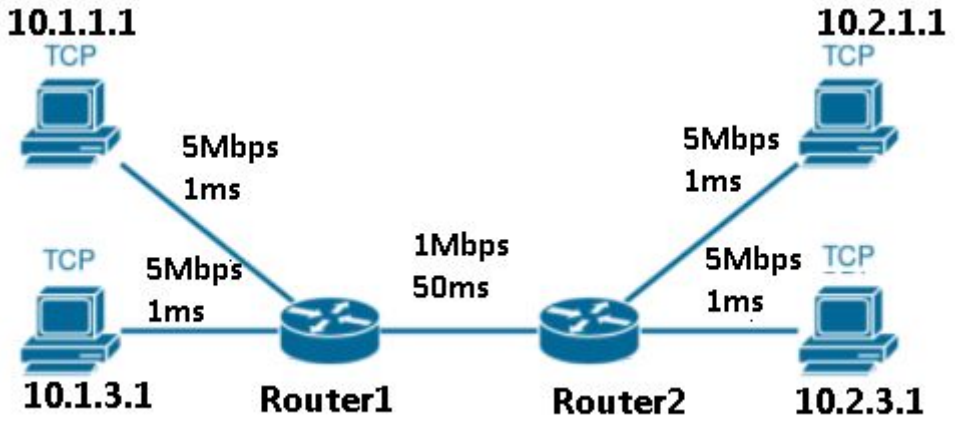
Para realizar pruebas con dos emisores TCP se deben colocar estas variables de la siguiente manera:

```
15   bool habilitarUDP=false;
16   bool todosTCP=false;
```

a) Hacer que sature el canal. Medir esa velocidad de transferencia (la cantidad de paquetes que llegan a destino).

El canal lo saturamos de la siguiente manera:
Por un lado, se configura el DataRate que es la cantidad de bits que se transmiten por unidad de tiempo entre los nodos. Por otro lado, se configura el Delay, que es el tiempo de retardo.

```
NS_LOG_INFO ("Create channels.");
PointToPointHelper p2Central;
p2Central.SetDeviceAttribute ("DataRate", StringValue ("1Mbps"));
p2Central.SetChannelAttribute ("Delay", StringValue ("50ms"));

PointToPointHelper p2Izq;
p2Izq.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
p2Izq.SetChannelAttribute ("Delay", StringValue ("1ms"));

PointToPointHelper p2Der;
p2Der.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
p2Der.SetChannelAttribute ("Delay", StringValue ("1ms"));
```



Como se puede observar en la Figura X, la conexión entre los routers tiene un DateRate de 1 Megabit por segundo y un delay de 50 milisegundos, mientras que las demás conexiones tienen 5Mbps y 1 milisegundo

respectivamente.
Esta configuración, provocará que se genere un cuello de botella entre el Router 1 y el Router 2, ya que procesan mucho más lento que los otros nodos, saturando así el canal.

Por medio de NatAnim, se puede encontrar los datos de cada uno de los flujos.

- Flow ID 1: Emisor TCP 1 (envía a Flow ID 3).
- Flow ID 2: Emisor TCP 2 (envía a Flow ID 4).
- Flow ID 3: Receptor TCP 1.
- Flow ID 4: Receptor TCP 2.

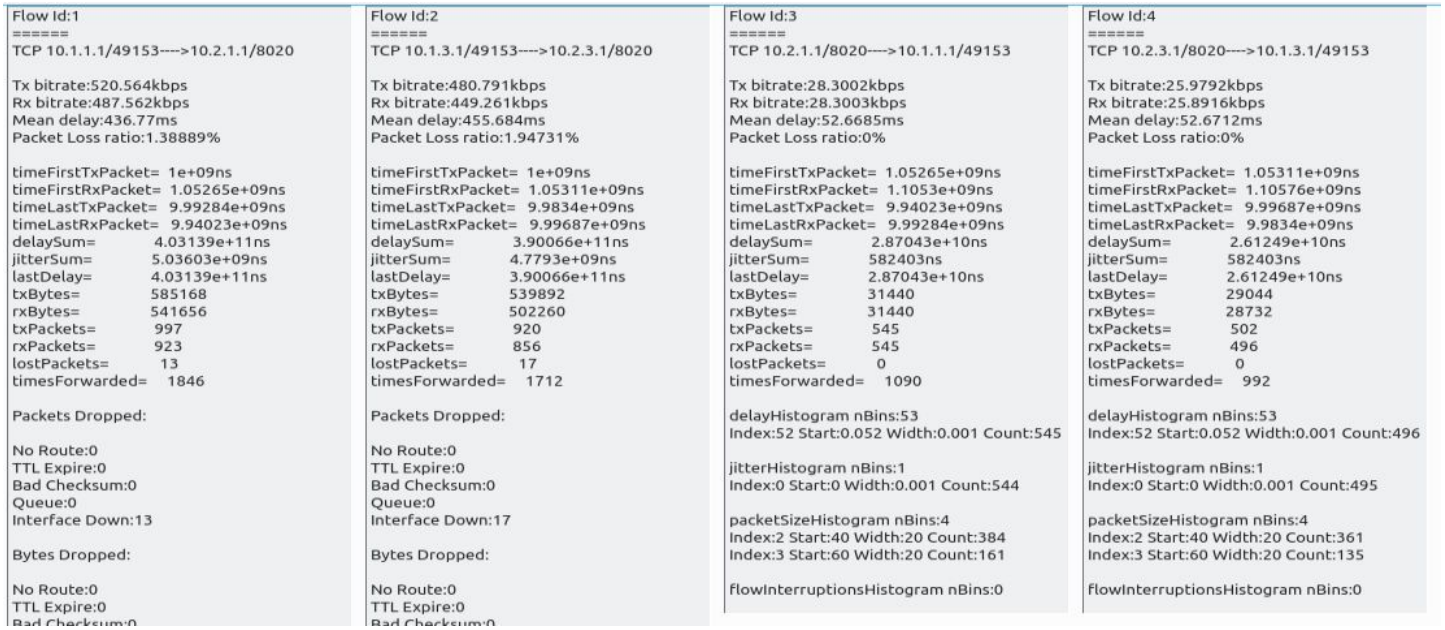


Figura 1. Flow Monitor, tomada de NetAnim

Emisor TCP1:
Cantidad de paquetes enviados: 585168
Cantidad de paquetes perdidos: 13
Porcentaje de paquetes perdidos: 1.38% aprox
Bits transmitidos: 520kbps

Emisor TCP2:
Cantidad de paquetes enviados por el emisor 2: 539892
Cantidad de paquetes perdidos: 17
Porcentaje de paquetes perdidos: 1.94% aprox
Bits transmitidos: 480 kbps

A partir de los datos obtenidos en la figura 1, observamos que el porcentaje de pérdida de paquetes (Packet Loss ratio) es baja para ambos nodos emisores (Flow Id:1, 2) Esto se debe a que TCP detecta la congestión y se ajusta a la capacidad de la red. A su vez, si sumamos Tx Bitrate de Flow 1 + Flow 2, esto suma a 1Mbps, que es el data rate del cuello de botella, lo que significa que el 'fairness' de TCP se ajusta correctamente, repartiendo de manera equitativa el ancho de banda del cuello de botella.

b) Mostrar mediante gráficos, tamaño de colas de recepción, ventana de TCP y cualquier mecanismo que muestre lo que sucede.

Manteniendo la misma configuración del punto a, utilizando los pcaps generados, se obtienen los siguientes gráficos con la herramienta WireShark.

Emisor TCP 1

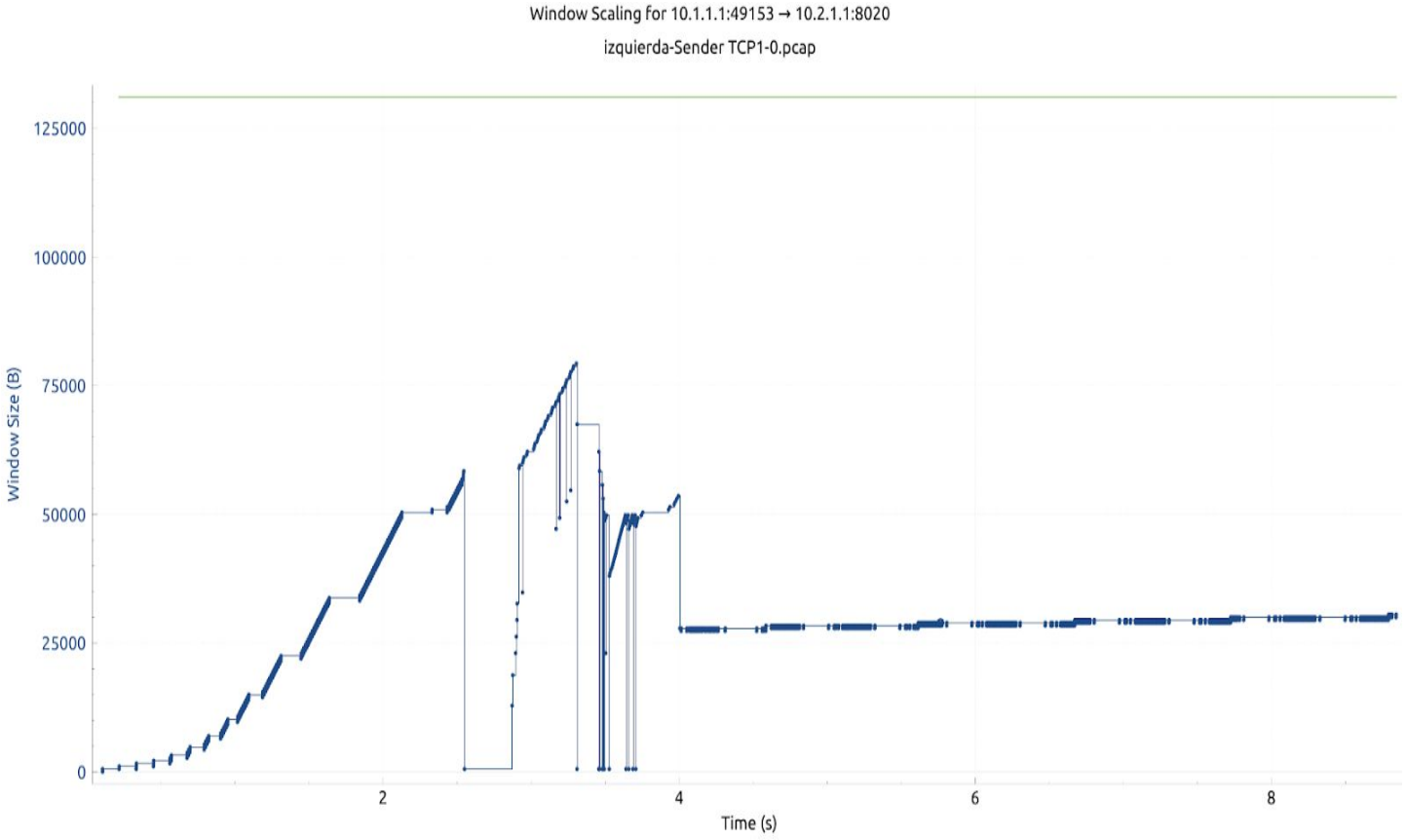


Figura 2: Ventana de congestión TCP1

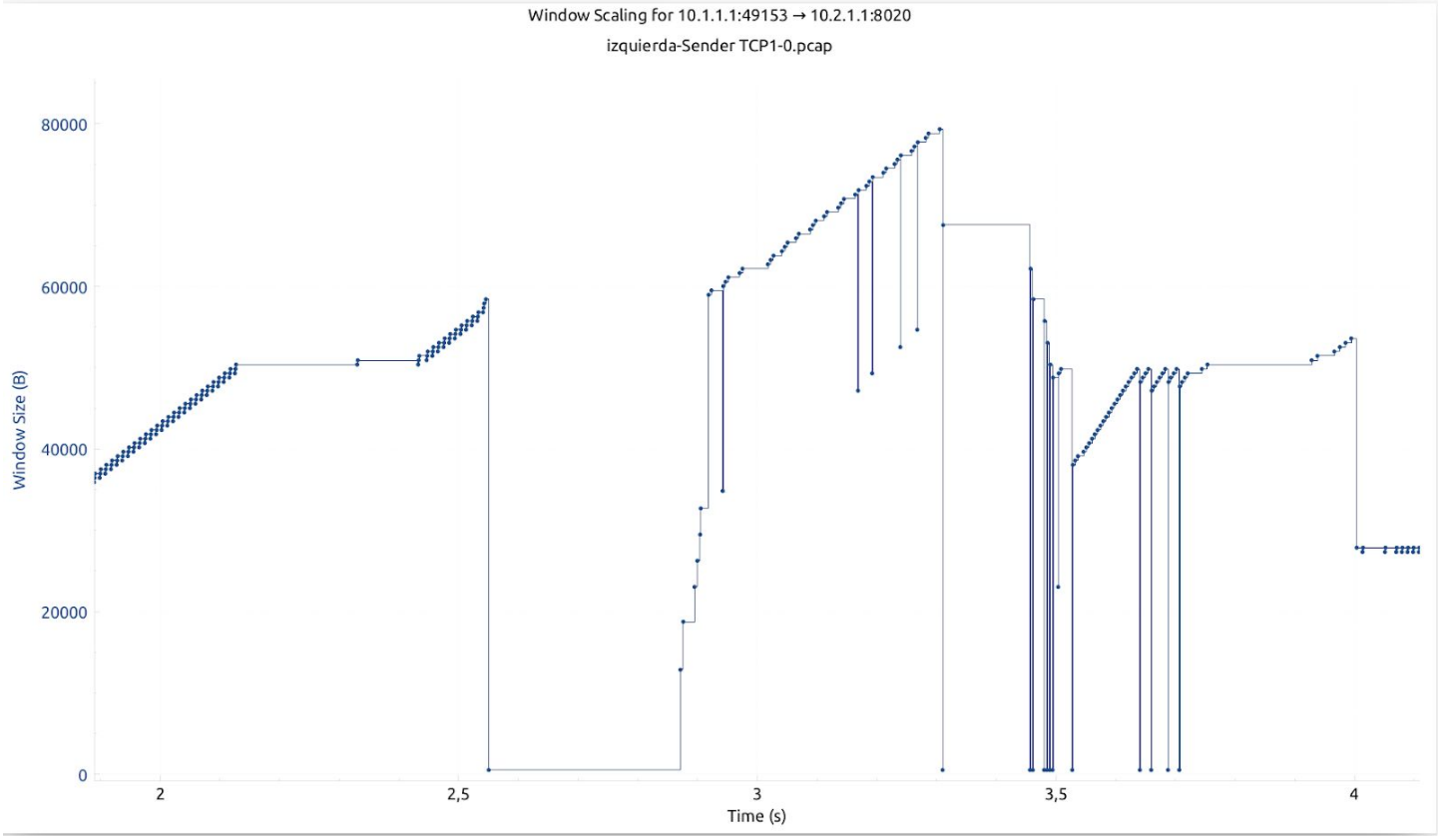


Figura 3: Ventana de congestión TCP1 con zoom

En la figura 2 y 3 se puede observar el comportamiento de la ventana de congestión del emisor TCP1.

Al inicio de la transmisión, comienza la etapa llamada Slow Start en la cual se setea la ventana de congestión (cwnd) en 1*MSS (Tamaño máximo de segmento) y además se configura un umbral de congestión (Este número se puede setear con la capacidad máxima que puede procesar el nodo receptor, es decir, la ventana de control de flujo). Luego, se va a ir aumentado el tamaño de la cwnd por cada segmento que llegue a destino, es decir por cada ACK se va a incrementar un MSS la cwnd, lo cual va a hacer que crezca exponencialmente. El objetivo de esta etapa es testear la capacidad de la red.
Esta etapa puede terminar por tres condiciones:

- Expira un temporizador (No llega el ACK a tiempo de un paquete)
- Se detectan 3 ACK duplicados (Tanenbaum explica que es un criterio elegido por TCP),
- Supera el umbral de slow start

En nuestra simulación, se puede observar que la etapa Slow Start inicia en el segundo 0 hasta llegar al 2.55 aprox ya que en ese momento se detectan 3 ACKs duplicados (Ver la figura 4) lo cual provoca una caída drástica de la ventana, comenzando así la etapa de **Fast Retrasmission.**

Time	Source	Destination	Protocol	Length	Info
2.545404	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#1] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…
2.545404	10.1.1.1	10.2.1.1	TCP	590	49153 → 8020 [ACK] Seq=169913 Ack=1 Win=131072 Len=536 TSval=3545 TSecr=3492
2.550124	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#2] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…
2.550124	10.1.1.1	10.2.1.1	TCP	590	[TCP Fast Retransmission] 49153 → 8020 [ACK] Seq=112025 Ack=1 Win=131072 Len=536…
2.554844	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#3] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…
2.559564	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#4] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…
2.564284	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#5] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…
2.569004	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#6] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…
2.573724	10.2.1.1	10.1.1.1	TCP	62	[TCP Dup ACK 423#7] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=3…

Figura 4: Tomada de WireShark

En la etapa de Fast Retransmission, se reenvían los paquetes sin esperar que el RTO esté vencido y luego se activa la etapa de Fast Recovery que sirve para evitar arrancar nuevamente en Slow Start. Es decir, el incremento en esta etapa no es exponencial como antes, ya que esto va a sobrecargar la red
En Fast Recovery, se realizan las siguientes acciones:

- Se divide el umbral de congestión (ssthresh) a la mitad
- Se configura la ventana de congestión 3 MSS por encima del ssthresh
- Por cada ACK duplicado que sigue recibiendo, se incrementa la cwnd en 1 MSS, inflando la ventana de congestión. Al momento que se recibe el ACK del paquete perdido, se desinfla la cwnd y coloca la ventana=ssthresh +3 MSS y se retoma la etapa de Congestion Avoidance.

Volviendo a la figura 3, se ven estos mecanismos en funcionamiento, a partir del segundo 2,87 aprox. Por otra parte, en la figura 5, se pueden ver las retransmisiones.

Time	Source	Destination	Protocol	Length	Info
2.875983	10.1.1.1	10.2.1.1	TCP	590	[TCP Retransmission] 49153 → 8020 [ACK] Seq=130249 Ack=1 Win=131072 Len=536 TSva…
2.894863	10.2.1.1	10.1.1.1	TCP	78	[TCP Dup ACK 423#54] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=…
2.894863	10.1.1.1	10.2.1.1	TCP	590	[TCP Retransmission] 49153 → 8020 [ACK] Seq=134537 Ack=1 Win=131072 Len=536 TSva…
2.899583	10.2.1.1	10.1.1.1	TCP	78	[TCP Dup ACK 423#55] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=…
2.899583	10.1.1.1	10.2.1.1	TCP	590	[TCP Retransmission] 49153 → 8020 [ACK] Seq=137753 Ack=1 Win=131072 Len=536 TSva…
2.904303	10.2.1.1	10.1.1.1	TCP	78	[TCP Dup ACK 423#56] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=…
2.904303	10.1.1.1	10.2.1.1	TCP	590	[TCP Retransmission] 49153 → 8020 [ACK] Seq=140969 Ack=1 Win=131072 Len=536 TSva…
2.905247	10.1.1.1	10.2.1.1	TCP	590	[TCP Retransmission] 49153 → 8020 [ACK] Seq=144185 Ack=1 Win=131072 Len=536 TSva…
2.918463	10.2.1.1	10.1.1.1	TCP	78	[TCP Dup ACK 423#57] 8020 → 49153 [ACK] Seq=1 Ack=112025 Win=131072 Len=0 TSval=…
2.918463	10.1.1.1	10.2.1.1	TCP	590	49153 → 8020 [ACK] Seq=170449 Ack=1 Win=131072 Len=536 TSval=3918 TSecr=3865

Figura 5: Ventana de congestión TCP1

A partir del segundo 4 se puede ver que la ventana se estabiliza en un rango de valores constantes entre (23-27 mil bytes aprox) como se puede ver en el figura 2 hasta el fin de la simulación. Entrando en la etapa de **Congestion Avoidance.**

La etapa de Congestion Avoidance finaliza cuando ocurren alguno de los siguientes eventos:

- Se vence el RTO: Se configura el umbral de congestión a la mitad de la ventana, se configura la ventana en 1 MSS y vuelve a Slow Start.
- Se detectan 3 ACKs duplicados: Se realiza Fast Retransmit y Fast Recovery.
- Ventana de congestión = Ventana de receptor: Se mantiene constante.

Emisor TCP 2

En cuanto al otro emisor, no se ven diferencias significativas en el gráfico de la ventana de congestión (Ver figura 6).
Lo que se puede destacar, es que hubo más descensos debido a que el emisor TCP 2 perdió más paquetes que el emisor TCP 1.

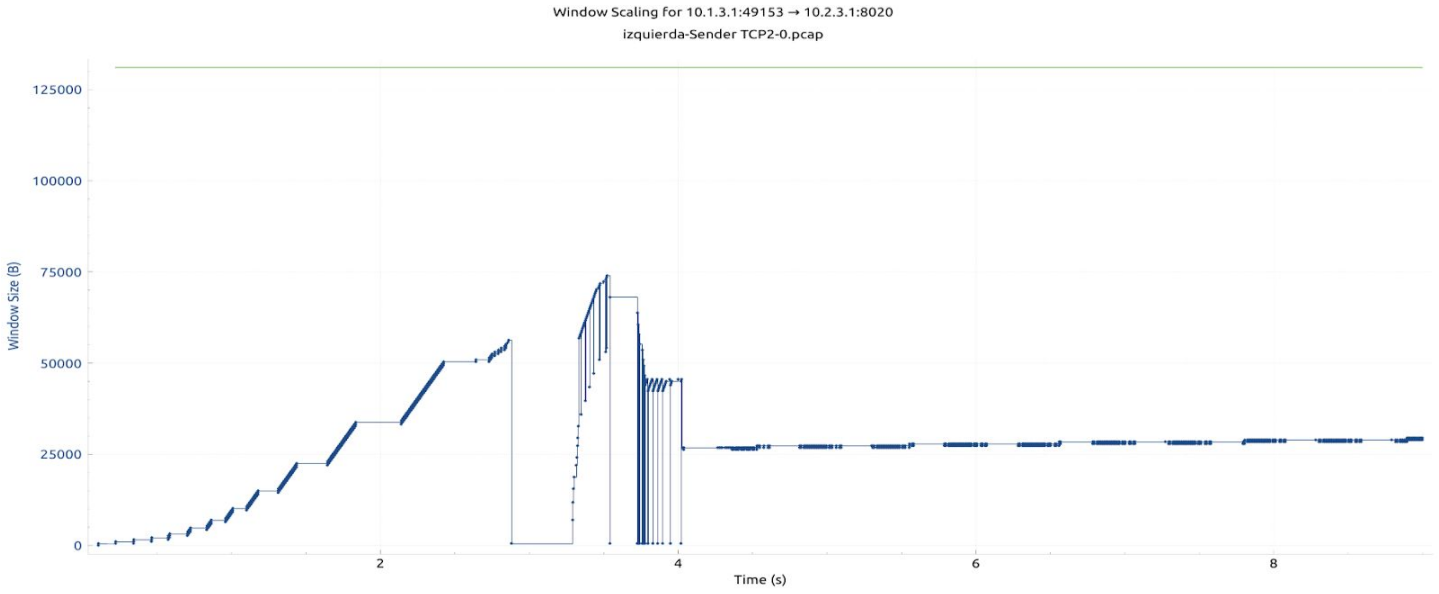


Figura 6: Ventana de congestión TCP2

c) Explicar en el gráfico las distintas etapas del protocolo TCP.

Manteniendo la misma configuración de los puntos anteriores, se explicarán las siguientes etapas del protocolo TCP:

- Three way HandShake (Inicio de conexión).
- Transferencia de datos (Explicado en el punto b)
- Fin de conexión

Three way HandShake:

Para comenzar la conexión, el protocolo TCP utiliza el mecanismo de Three way handshake. Debido a que TCP bidireccional tanto el emisor y receptor deben sincronizarse.
Los pasos son los siguientes:

1. Uno de los lados se encuentra escuchando.
2. El emisor envía una primitiva CONNECT especificando dirección y puerto con el que desea conectarse, el tamaño máximo del segmento dispuesto a aceptar y otros datos opcionales. La primitiva CONNECT envía un segmento TCP con el bit SYN encendido y ACK apagado y espera. (SYN SENT). (Primer paquete TCP)
3. Si algún proceso está escuchando en el puerto, este recibe el segmento TCP y puede aceptar o rechazar la conexión.
4. Si la acepta devuelve el segmento de confirmación de recepción. Enviando un mensaje al emisor con su número de secuencia y el ACK hasta el que recibió. (SYN_RECEIVED) (Segundo paquete TCP)

5. El emisor le envía el tercer mensaje para confirmar que la conexión está establecida. (Tercer paquete.

Por medio de WireShark podemos observar los tres paquetes que son enviados durante el Three Way Handshake.

Time	Source	Destination	Protocol	Length	Info
0.000000	10.1.1.1	10.2.1.1	TCP	58	49153 → 8020 [SYN] Seq=0 Win=65535 Len=0 TSval=1000 TSecr=0 WS=4 SACK_PERM=1
0.105299	10.2.1.1	10.1.1.1	TCP	58	8020 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSval=1052 TSecr=1000 WS=4 SACK_PERM
0.105299	10.1.1.1	10.2.1.1	TCP	54	49153 → 8020 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=1105 TSecr=1052

Figura 6: 3-Way-Handshake tomado de WireShark

En este caso el Emisor TCP1 (10.1.1.1) envía un paquete con el flag SYN prendido y su número de secuencia (seq=0) al receptor TCP1 (10.2.1.1). Luego, el receptor TCP 1 (10.2.1.1) responde con un paquete en el cual le envía el flag SYN prendido, el ACK indicando que recibió hasta el 1 y su número de secuencia (seq=0). Finalmente, el Emisor TCP1 (10.1.1.1), le confirma al receptor TCP1 (10.2.1.1) que la conexión fue establecida enviándole un paquete con ACK=1 y actualizando su número de secuencia (seq=1).

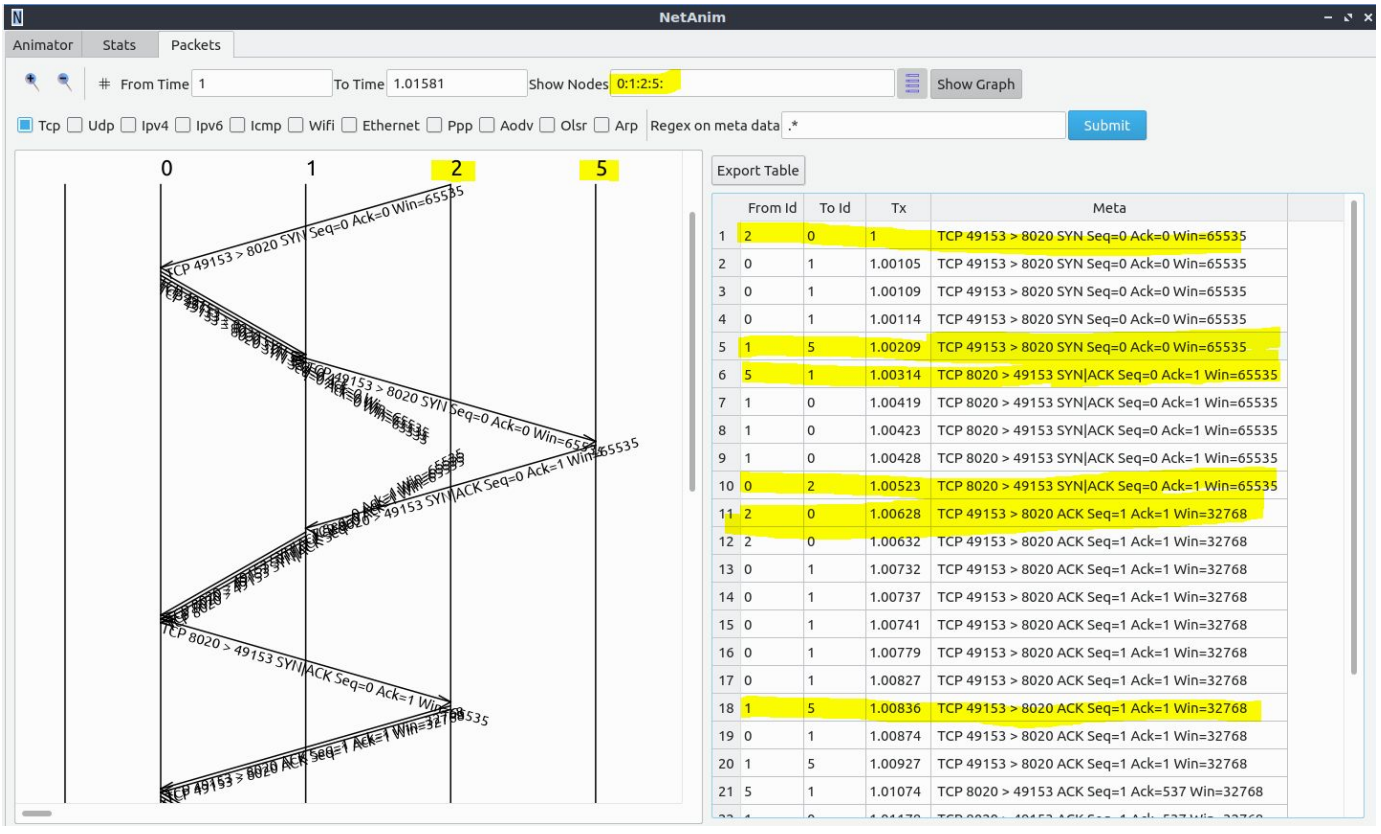


Figura 7: Inicio de conexión pasando por los routers

En la figura 7, se puede ver el recorrido de los paquetes del Three Way Handshake, en donde el nodo 2 es el Emisor, el nodo 5 es el receptor y los nodos 0 y 1 son los routers.

Fin de conexión:

Cuando uno de los nodos ha terminado de transferir, ejecuta una primitiva CLOSE, esto hace que la entidad TCP local envíe un segmento FIN y espere el ACK correspondiente. Al llegar el ACK, se hace una transición

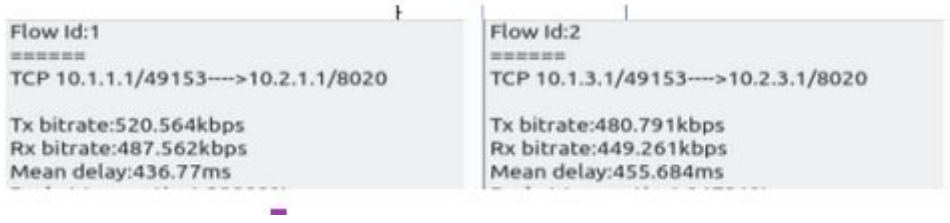
al estado FIN WAIT 2 y se cierra un sentido de la conexión. Cuando cierra también el otro lado llega un FIN, para el cual se envía una confirmación de recepción. Si bien en ese momento ambos están cerrados, TCP espera un tiempo igual al doble del tiempo de vida máximo del paquete para garantizar que todos los paquetes de la conexión hayan expirado, sólo por si acaso se perdió la confirmación de recepción. Al expirar el temporizador, TCP borra el registro de la conexión.

En nuestra simulación, no se puede ver el fin de la conexión debido a que la simulación se realiza por un tiempo determinado, y termina cuando se alcanza ese tiempo sin realizar el fin de la conexión.

d) Calcular el ancho de banda del canal. Explicar qué sucede. ¿Ve alguna anomalía ? Explicarla.

Como se mencionó previamente, los nodos emisores y los receptores, tienen una velocidad de transferencia (Datarate) de 5 Megabits por segundo, mientras que los nodos centrales (Routers) tienen una velocidad de 1 Megabit por segundo. La velocidad de la red es determinada por los nodos de menor velocidad, en este caso la velocidad de los routers. Por lo tanto, el ancho de banda del canal es como máximo de 1 Mbps.

Para comprobar que en la simulación se cumple con lo que afirma el párrafo anterior, se puede realizar una sum del txbiterate (velocidad de transferencia) de los emisores



Nodos emisores:
txbitrate_TCP1_Emisor + txbitrate_TCP2_Emisor =
520, 564 kbps + 480,791 kbps = kbps (aprox 1.01 Mbps)

Puede ser que debido a la simulación, se haya pasado 0.05Mbps del máximo que era 1Mbps.

Realizar pruebas (con la misma configuración), pero ahora con el nodo UDP también emitiendo.

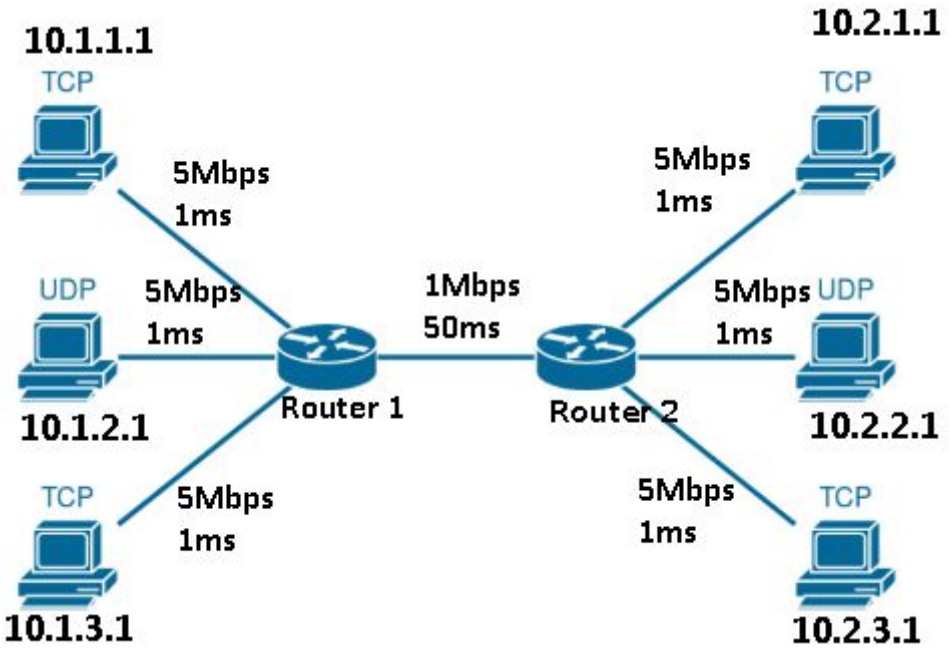
Para realizar las pruebas con dos emisores TCP y un emisor UDP se deben colocar estas variables de la siguiente manera:

```
15   bool habilitarUDP=true;
16   bool todosTCP=false;
```

a) Explicar qué sucede con el ancho de banda utilizado por cada uno.

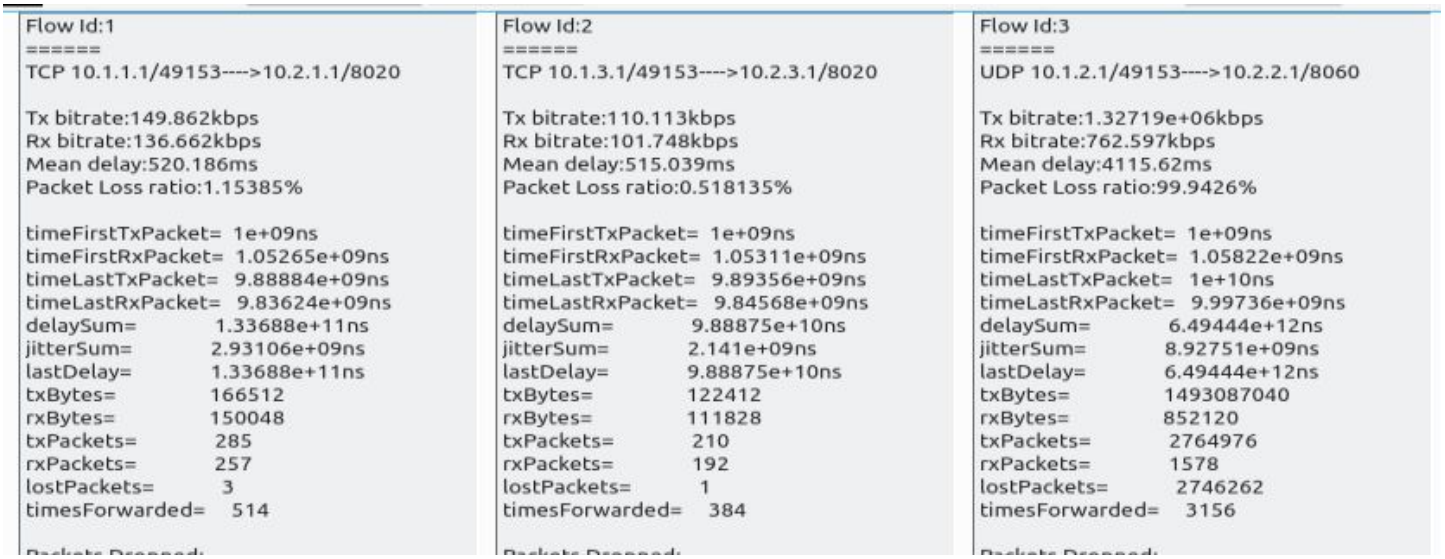
Realizamos las pruebas con la siguiente configuración (Mismo DataRate y Delay que en el punto anterior):

```
NS_LOG_INFO ("Create channels.");
PointToPointHelper p2Central;
p2Central.SetDeviceAttribute ("DataRate", StringValue ("1Mbps"));
p2Central.SetChannelAttribute ("Delay", StringValue ("50ms"));

PointToPointHelper p2Izq;
p2Izq.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
p2Izq.SetChannelAttribute ("Delay", StringValue ("1ms"));

PointToPointHelper p2Der;
p2Der.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
p2Der.SetChannelAttribute ("Delay", StringValue ("1ms"));
```



Por medio de NatAnim, se puede encontrar los datos de cada uno de los flujos.

- Flow ID 1: Emisor TCP 1
- Flow ID 2: Emisor TCP 2
- Flow ID 3: Emisor UDP



```
Flow Id:1
======
TCP 10.1.1.1/49153---->10.2.1.1/8020

Tx bitrate:149.862kbps
Rx bitrate:136.662kbps
Mean delay:520.186ms
Packet Loss ratio:1.15385%

timeFirstTxPacket=  1e+09ns
timeFirstRxPacket=  1.05265e+09ns
timeLastTxPacket=  9.88884e+09ns
timeLastRxPacket=  9.83624e+09ns
delaySum=          1.33688e+11ns
jitterSum=         2.93106e+09ns
lastDelay=         1.33688e+11ns
txBytes=           166512
rxBytes=           150048
txPackets=         285
rxPackets=         257
lostPackets=       3
timesForwarded=    514
```

```
Flow Id:2
======
TCP 10.1.3.1/49153---->10.2.3.1/8020

Tx bitrate:110.113kbps
Rx bitrate:101.748kbps
Mean delay:515.039ms
Packet Loss ratio:0.518135%

timeFirstTxPacket=  1e+09ns
timeFirstRxPacket=  1.05311e+09ns
timeLastTxPacket=  9.89356e+09ns
timeLastRxPacket=  9.84568e+09ns
delaySum=          9.88875e+10ns
jitterSum=         2.141e+09ns
lastDelay=         9.88875e+10ns
txBytes=           122412
rxBytes=           111828
txPackets=         210
rxPackets=         192
lostPackets=       1
timesForwarded=    384
```

```
Flow Id:3
======
UDP 10.1.2.1/49153---->10.2.2.1/8060

Tx bitrate:1.32719e+06kbps
Rx bitrate:762.597kbps
Mean delay:4115.62ms
Packet Loss ratio:99.9426%

timeFirstTxPacket=  1e+09ns
timeFirstRxPacket=  1.05822e+09ns
timeLastTxPacket=  1e+10ns
timeLastRxPacket=  9.99736e+09ns
delaySum=          6.49444e+12ns
jitterSum=         8.92751e+09ns
lastDelay=         6.49444e+12ns
txBytes=           1493087040
rxBytes=           852120
txPackets=         2764976
rxPackets=         1578
lostPackets=       2746262
timesForwarded=    3156
```

Figura 8. Flow Monitor, tomada de NetAnim(2 TCP y 1 UDP)

Emisor TCP1:
Cantidad de paquetes enviados: 285
Cantidad de paquetes perdidos: 3
Porcentaje de paquetes perdidos: 1.15% aprox
Bits transmitidos: 149 kbps

Emisor TCP2:
Cantidad de paquetes enviados: 210
Cantidad de paquetes perdidos: 1
Porcentaje de paquetes perdidos: 0.5% aprox
Bits transmitidos: 110 kbps

Emisor UDP:
Cantidad de paquetes enviados: 1493087040
Cantidad de paquetes perdidos: 2746262
Porcentaje de paquetes perdidos: 99.94% aprox
Bits transmitidos: 1.32719e+06 kbps

Por lo que se puede apreciar en los datos obtenidos, se observa que la tasa de los paquetes perdidos de UDP es altísima llegando al 99.94%. En cuanto a los emisores TCP la tasa de pérdida de paquetes no superan el 1,2%. Esto se debe a que TCP cuenta con el mecanismo de control de congestión, mientras que UDP no tiene ningún mecanismo y por lo tanto envía si importar si los paquetes llegaron a destino o no.
Otra diferencia que se puede observar es que, a diferencia de la prueba anterior en la cual el ancho de banda del cuello de botella (nodo central) se dividía equitativamente entre los emisores , en este caso, el nodo UDP rompe esta relación. Como mencionamos anteriormente, el protocolo UDP solo se dedica a enviar paquetes sin realizar ninguna validación, por lo abarcara la mayor parte de la red y forzará a los otros dos nodos a adaptarse al ancho de banda restante.
En el siguiente gráfico se puede ver que se cumple lo mencionado en el párrafo anterior.

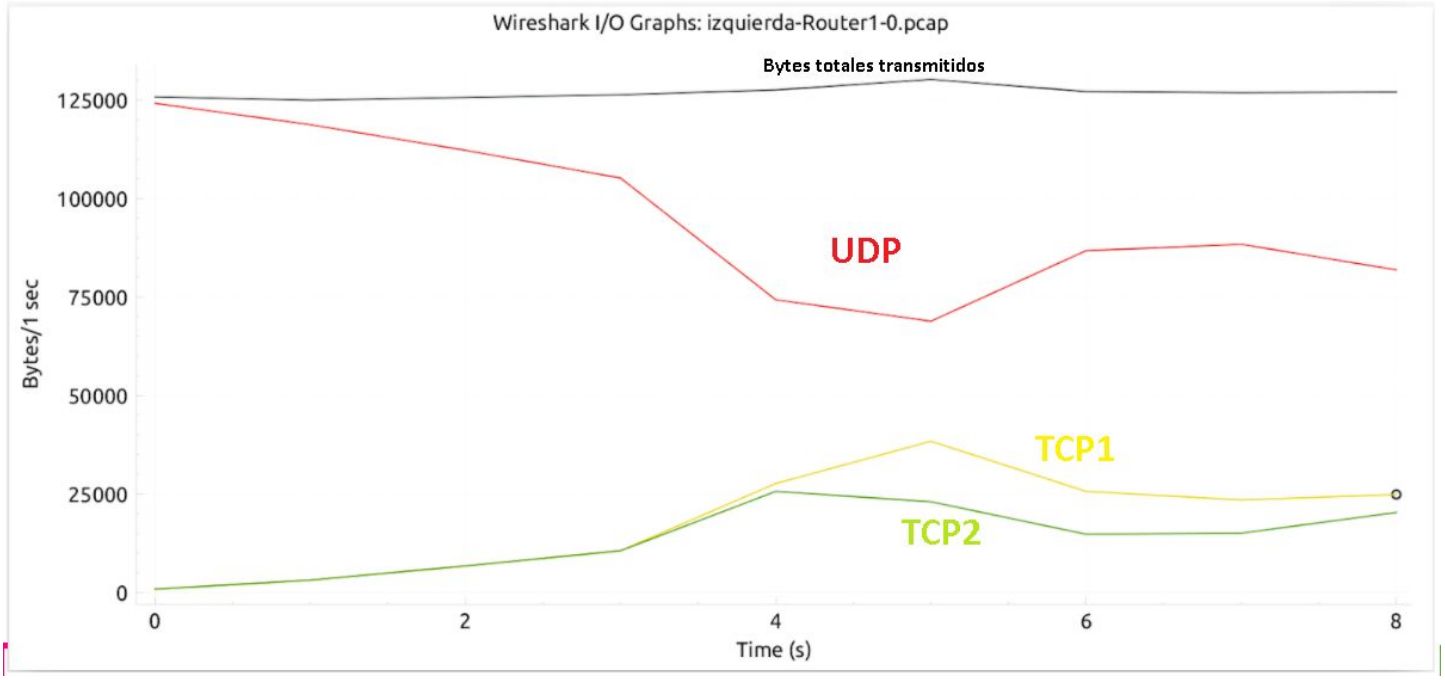


Figura 9. Bytes/Segundos de TCP1, TCP2 y UDP

En el inicio del gráfico 9 (instante 0) se puede ver como UDP consume todo el ancho de banda, mientras que, como vimos en las anteriores secciones, los nodos TCP comienzan con la etapa de Slow Start para ir testeando la capacidad de la red . Esta etapa se mantiene hasta el segundo 5,5 en el emisor 1 y el segundo 2,8 aprox en el emisor 2 (ver figura 10 y 11).
A partir del segundo 6,5 para el emisor TCP 1 y el segundo 4,1 del emisor TCP 2 comienza el Congestion avoidance. En esos instantes se puede observar cómo se estabiliza la ventana de congestión en las figuras 10 y 11. En este punto los dos nodos TCP ya ajustaron su ventana de congestión y el valor de la ventana estará en un intervalo que irá incrementándose por cada ACK recibido y le quitará ancho de banda a UDP.
En el gráfico 9 se puede observar como UDP desde que comenzó fue disminuyendo su velocidad, en cambio ambos nodos TCP fueron incrementando.

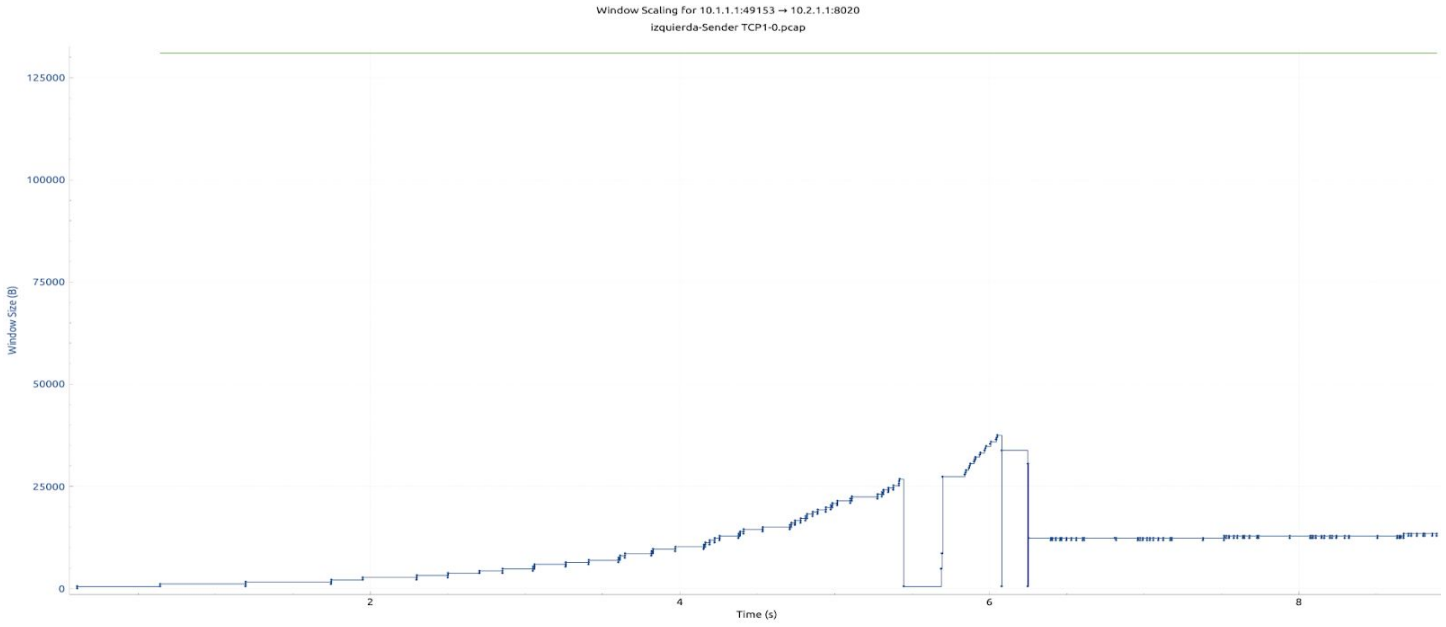


Figura 10. Ventana de congestión de TCP1

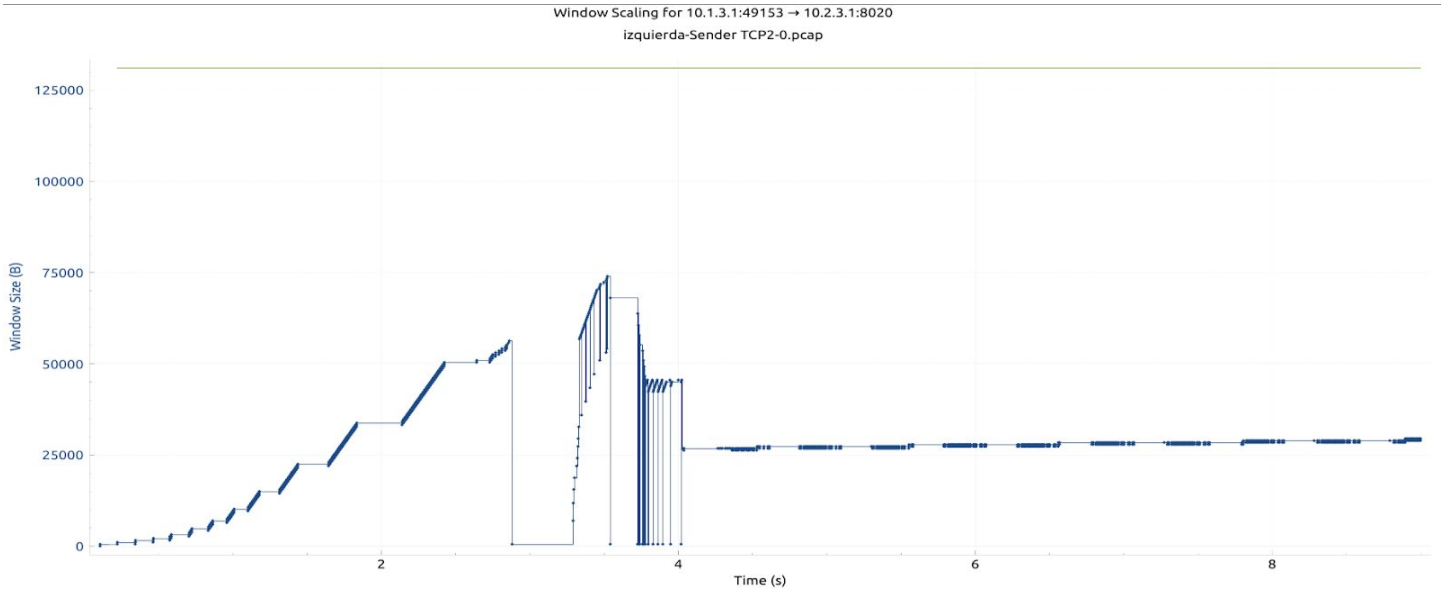


Figura 11. Ventana de congestión de TCP2

b) Mostrar en los paquetes TCP dónde se ven las distintas acciones del protocolo

Analizamos el pcap del nodo 0 (el router del lado izquierdo) generado en la prueba anterior en wireshark, ya que en ese nodo convergen los paquetes enviados por los 3 emisores.

Los nodos emisores TCP comienzan con el mecanismo de Three Way Handshake anteriormente explicado. A su vez, el cuello de botella está saturado al instante con los paquetes UDP, lo que lleva a que la recepción de esos ACKs correspondientes al Three way Handshake se demoren.

3	0.000928	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
4	0.005264	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
5	0.009600	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
6	0.013936	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
7	0.018272	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
8	0.022608	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
9	0.026944	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
10	0.031280	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
11	0.035616	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
12	0.039952	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
13	0.044288	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
14	0.048624	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
15	0.052960	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
16	0.057296	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
17	0.061632	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
18	0.065968	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
19	0.070304	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
20	0.074640	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
21	0.078976	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
22	0.083312	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
23	0.087648	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
24	0.091984	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
25	0.096320	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
26	0.100656	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
27	0.103114	10.2.1.1	10.1.1.1	TCP	58 8020 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSval=105…
28	0.103578	10.2.3.1	10.1.3.1	TCP	58 8020 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSval=105…

Finalmente, en el segundo 0,5255 termina el inicio de conexión

125	0.521248	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
126	0.525584	10.1.1.1	10.2.1.1	TCP	54 49153 → 8020 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=1105 TS…
127	0.526016	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
128	0.530352	10.1.3.1	10.2.3.1	TCP	54 49153 → 8020 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=1105 TS…
129	0.530784	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=1 Ack=1 Win=131072 Len=536 TSval=1105 …
130	0.535504	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
131	0.539840	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
132	0.544176	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
133	0.548512	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=1 Ack=1 Win=131072 Len=536 TSval=1105 …

El Nodo UDP sigue acaparando la red, pero notamos que la cantidad de paquetes enviados por TCP se va duplicando debido al Slow Start. En la siguiente imagen se observa que los emisores mandan 6 paquetes

516	2.178736	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
517	2.183072	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=3217 Ack=1 Win=131072 Len=536 TSval=27…
518	2.187792	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=3753 Ack=1 Win=131072 Len=536 TSval=27…
519	2.192512	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=4289 Ack=1 Win=131072 Len=536 TSval=27…
520	2.197232	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
521	2.201568	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=3217 Ack=1 Win=131072 Len=536 TSval=27…
522	2.206288	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=3753 Ack=1 Win=131072 Len=536 TSval=27…
523	2.211008	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=4289 Ack=1 Win=131072 Len=536 TSval=27…
524	2.215728	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512

Y luego envían 12.

832	3.491824	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=11793 Ack=1 Win=131072 Len=536 TSval=4…
833	3.496544	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=12329 Ack=1 Win=131072 Len=536 TSval=4…
834	3.501264	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=12865 Ack=1 Win=131072 Len=536 TSval=4…
835	3.505984	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
836	3.510320	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=11793 Ack=1 Win=131072 Len=536 TSval=4…
837	3.515040	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=12329 Ack=1 Win=131072 Len=536 TSval=4…
838	3.519760	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=12865 Ack=1 Win=131072 Len=536 TSval=4…
839	3.524480	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
840	3.528816	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=13401 Ack=1 Win=131072 Len=536 TSval=4…
841	3.533536	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=13937 Ack=1 Win=131072 Len=536 TSval=4…
842	3.538256	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=14473 Ack=1 Win=131072 Len=536 TSval=4…
843	3.542976	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=13401 Ack=1 Win=131072 Len=536 TSval=4…
844	3.547696	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=13937 Ack=1 Win=131072 Len=536 TSval=4…
845	3.552416	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=14473 Ack=1 Win=131072 Len=536 TSval=4…
846	3.557136	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512

Mientras ocurre este incremento exponencial, los ACKs que llegan del emisor son cada vez más tardíos (Esto es señal de que la red se está congestionando). Finalmente al detectar ACKs duplicados, los nodos TCP disparan el mecanismo de Fast Recovery y Fast Retransmissions explicados en el punto 2)b)

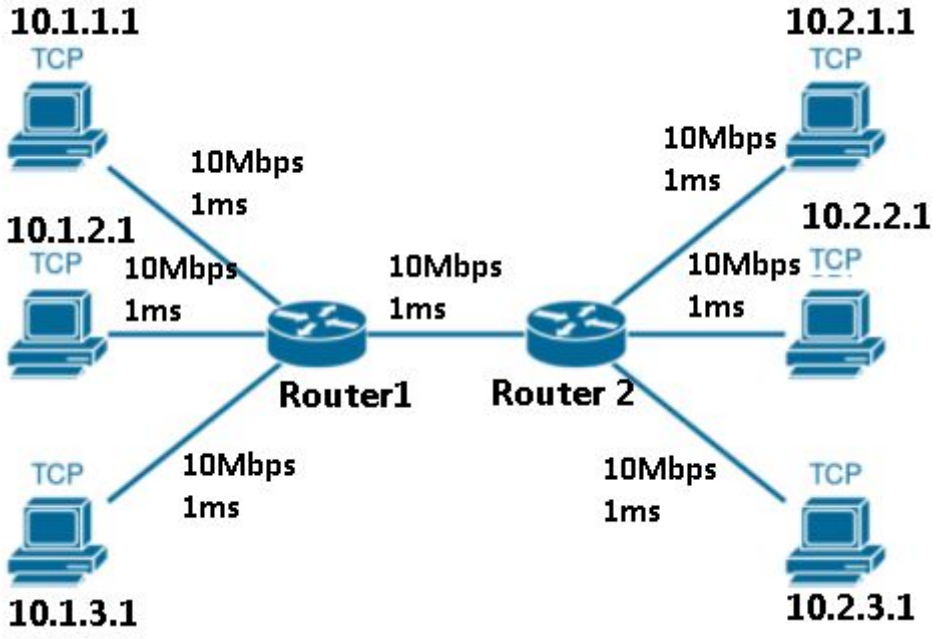
1176	4.840547	10.2.3.1	10.1.3.1	TCP	62 [TCP Dup ACK 1174#1] 8020 → 49153 [ACK] Seq=1 Ack=32697 Win=1…
1177	4.840959	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
1178	4.845267	10.2.3.1	10.1.3.1	TCP	62 [TCP Dup ACK 1174#2] 8020 → 49153 [ACK] Seq=1 Ack=32697 Win=1…
1179	4.845295	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
1180	4.849631	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=35913 Ack=1 Win=131072 Len=536 TSval=5…
1181	4.854351	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=36449 Ack=1 Win=131072 Len=536 TSval=5…
1182	4.854630	10.2.1.1	10.1.1.1	TCP	54 8020 → 49153 [ACK] Seq=1 Ack=34841 Win=131072 Len=0 TSval=580…
1183	4.859071	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=36985 Ack=1 Win=131072 Len=536 TSval=5…
1184	4.863791	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=37521 Ack=1 Win=131072 Len=536 TSval=5…
1185	4.868511	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=38057 Ack=1 Win=131072 Len=536 TSval=5…
1186	4.872435	10.2.3.1	10.1.3.1	TCP	62 [TCP Dup ACK 1174#3] 8020 → 49153 [ACK] Seq=1 Ack=32697 Win=1…
1187	4.873231	10.1.1.1	10.2.1.1	TCP	590 49153 → 8020 [ACK] Seq=38593 Ack=1 Win=131072 Len=536 TSval=5…
1188	4.877155	10.2.3.1	10.1.3.1	TCP	62 [TCP Dup ACK 1174#4] 8020 → 49153 [ACK] Seq=1 Ack=32697 Win=1…
1189	4.877951	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
1190	4.882287	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=37521 Ack=1 Win=131072 Len=536 TSval=5…
1191	4.886518	10.2.1.1	10.1.1.1	TCP	54 8020 → 49153 [ACK] Seq=1 Ack=35913 Win=131072 Len=0 TSval=583…
1192	4.887007	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=38057 Ack=1 Win=131072 Len=536 TSval=5…
1193	4.891727	10.1.3.1	10.2.3.1	TCP	590 49153 → 8020 [ACK] Seq=38593 Ack=1 Win=131072 Len=536 TSval=5…
1194	4.896447	10.1.2.1	10.2.2.1	UDP	542 49153 → 8060 Len=512
1195	4.899987	10.2.3.1	10.1.3.1	TCP	62 [TCP Dup ACK 1174#5] 8020 → 49153 [ACK] Seq=1 Ack=32697 Win=1…

Parte 2

Funcionamiento del algoritmo Hybla

TCP estándar sufre una baja performance en redes con alto delay como puede ser una conexión satelital o Wifi. El alto delay hace que la ventana de congestión no crezca, ya que esta depende de que el ACK llegue en un tiempo razonable para aumentar su tamaño. Para abordar el problema del aumento lento de la ventana de congestión (cwnd), TCP Hybla elimina la dependencia del RTT del algoritmo de congestion avoidance. Esto se hace ajustando el tamaño del cwnd a una relación normalizada de la ventana anterior, lo que da como resultado un cwnd promedio más grande.
Para aliviar la pérdida de paquetes en la ventana, TCP Hybla usa Reconocimiento selectivo (SACK) que permite al remitente saber exactamente qué paquetes se han enviado con éxito y la capacidad de enviar más de un paquete por RTT.
TCP Hybla incorpora la variable ρ=RTT/RTTo, que es un RTT normalizado, donde RTTo es el RTT de referencia al cual se quiere igualar su desempeño. Con el fin de no disminuir la conexión que presente un RTT < RTTo, TCP Hybla fija a ρ en un valor mínimo de 1 y de esta manera se logra aprovechar la estimación del ancho de banda.

Realizar una configuración de red como la definida en la parte 1 del desarrollo pero sólo con nodos TCP.



Para realizar las pruebas con tres emisores TCP se deben colocar estas variables de la siguiente manera:

```
15    bool habilitarUDP=false;
16    bool todosTCP=true;
```

Implementar el algoritmo.

Para realizar la implementación del algoritmo se agregó en el código esta línea:

```
Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue ("ns3::TcpHybla"));
```

Presentar datos (tomados por wireshark o ns-3) de las corridas realizadas que muestren que el funcionamiento (según la teoría) es correcto

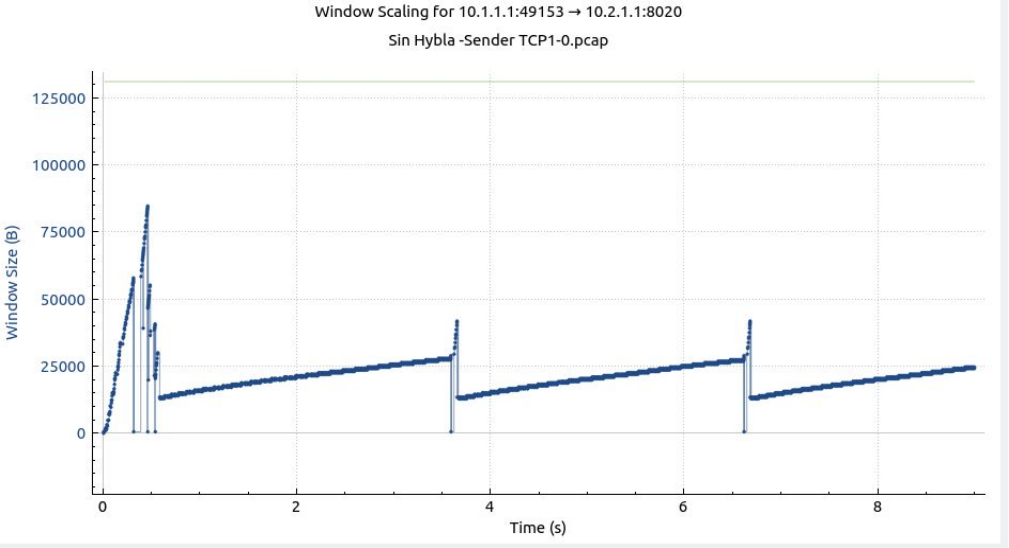


Figura 12: Ventana de congestión sin Hybla

Flow Id:1	Flow Id:2	Flow Id:3
======	======	======
TCP 10.1.1.1/49153---->10.2.1.1/8020	TCP 10.1.2.1/49153---->10.2.2.1/8020	TCP 10.1.3.1/49153---->10.2.3.1/8020
Tx bitrate:3437.42kbps	Tx bitrate:3161.22kbps	Tx bitrate:3405.2kbps
Rx bitrate:3413.13kbps	Rx bitrate:3145.6kbps	Rx bitrate:3380.03kbps
Mean delay:49.9975ms	Mean delay:49.4746ms	Mean delay:49.9816ms
Packet Loss ratio:0.0918274%	Packet Loss ratio:0.116183%	Packet Loss ratio:0.092707%
timeFirstTxPacket= 1e+09ns	timeFirstTxPacket= 1e+09ns	timeFirstTxPacket= 1e+09ns
timeFirstRxPacket= 1.00314e+09ns	timeFirstRxPacket= 1.00319e+09ns	timeFirstRxPacket= 1.00323e+09ns
timeLastTxPacket= 9.99929e+09ns	timeLastTxPacket= 9.99693e+09ns	timeLastTxPacket= 9.99881e+09ns
timeLastRxPacket= 9.99757e+09ns	timeLastRxPacket= 9.99993e+09ns	timeLastRxPacket= 9.99946e+09ns
delaySum= 3.26384e+11ns	delaySum= 2.97738e+11ns	delaySum= 3.23181e+11ns
jitterSum= 6.45092e+09ns	jitterSum= 5.79621e+09ns	jitterSum= 6.47958e+09ns
lastDelay= 3.26384e+11ns	lastDelay= 2.97738e+11ns	lastDelay= 3.23181e+11ns
txBytes= 3866796	txBytes= 3555156	txBytes= 3830340
rxBytes= 3837396	rxBytes= 3537516	rxBytes= 3800940
txPackets= 6578	txPackets= 6048	txPackets= 6516
rxPackets= 6528	rxPackets= 6018	rxPackets= 6466
lostPackets= 6	lostPackets= 7	lostPackets= 6
timesForwarded= 13056	timesForwarded= 12036	timesForwarded= 12932

Figura 13: Datos sin Hybla

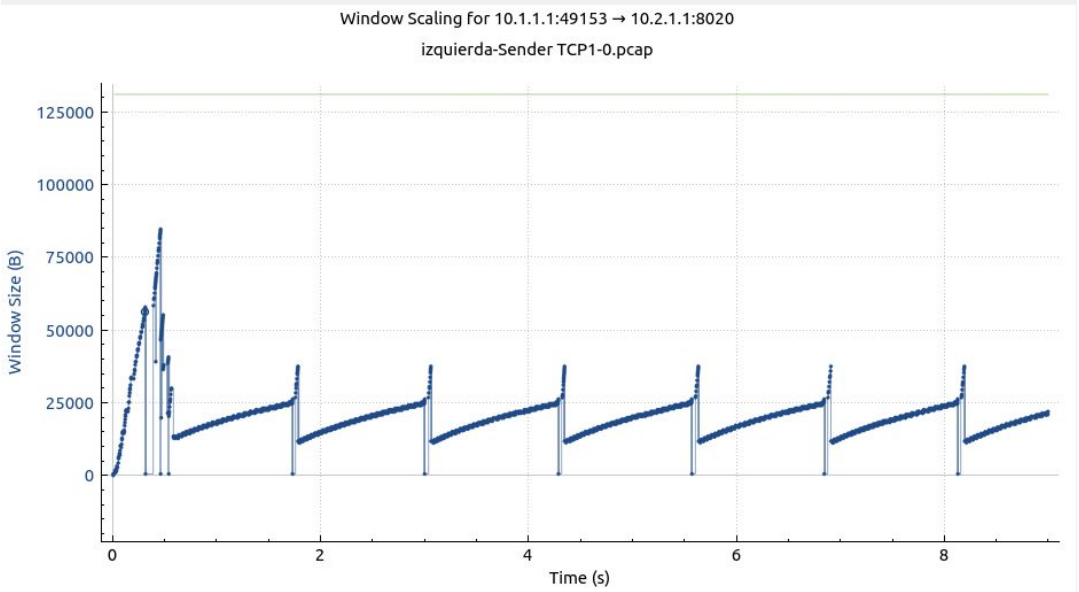


Figura 14: Ventana de congestión con Hybla

Flow Id:1	Flow Id:2	Flow Id:3
======	======	======
TCP 10.1.1.1/49153---->10.2.1.1/8020	TCP 10.1.2.1/49153---->10.2.2.1/8020	TCP 10.1.3.1/49153---->10.2.3.1/8020
Tx bitrate:3351.19kbps	Tx bitrate:3347.18kbps	Tx bitrate:3312.5kbps
Rx bitrate:3326.84kbps	Rx bitrate:3323.02kbps	Rx bitrate:3290.97kbps
Mean delay:47.5083ms	Mean delay:47.8548ms	Mean delay:47.8114ms
Packet Loss ratio:0.156986%	Packet Loss ratio:0.172766%	Packet Loss ratio:0.158579%
timeFirstTxPacket= 1e+09ns	timeFirstTxPacket= 1e+09ns	timeFirstTxPacket= 1e+09ns
timeFirstRxPacket= 1.00314e+09ns	timeFirstRxPacket= 1.00319e+09ns	timeFirstRxPacket= 1.00323e+09ns
timeLastTxPacket= 9.99645e+09ns	timeLastTxPacket= 9.9974e+09ns	timeLastTxPacket= 9.99929e+09ns
timeLastRxPacket= 9.99333e+09ns	timeLastRxPacket= 9.99805e+09ns	timeLastRxPacket= 9.99993e+09ns
delaySum= 3.02153e+11ns	delaySum= 3.04165e+11ns	delaySum= 3.01021e+11ns
jitterSum= 6.14322e+09ns	jitterSum= 6.2887e+09ns	jitterSum= 5.99598e+09ns
lastDelay= 3.02153e+11ns	lastDelay= 3.04165e+11ns	lastDelay= 3.01021e+11ns
txBytes= 3768600	txBytes= 3764484	txBytes= 3726264
rxBytes= 3738612	rxBytes= 3736260	rxBytes= 3700980
txPackets= 6411	txPackets= 6404	txPackets= 6339
rxPackets= 6360	rxPackets= 6356	rxPackets= 6296
lostPackets= 10	lostPackets= 11	lostPackets= 10
timesForwarded= 12720	timesForwarded= 12712	timesForwarded= 12592

Figura 15: Datos con Hybla

Teniendo en cuenta que el algoritmo de Hybla deja de darle importancia al RTT, observamos cómo este hecho se refleja en el manejo del control de congestión, ya que se alcanza con más frecuencia la cota de la ventana debido al aumento de la pérdida de paquetes (Ver figura 12 y 14).

Por otro lado, también se puede evidenciar que al aplicar el algoritmo de Hybla, a pesar de que se pierden más paquetes, la media del ancho de banda no está tan alejada a la actual. (Ver figura 13 y 15)

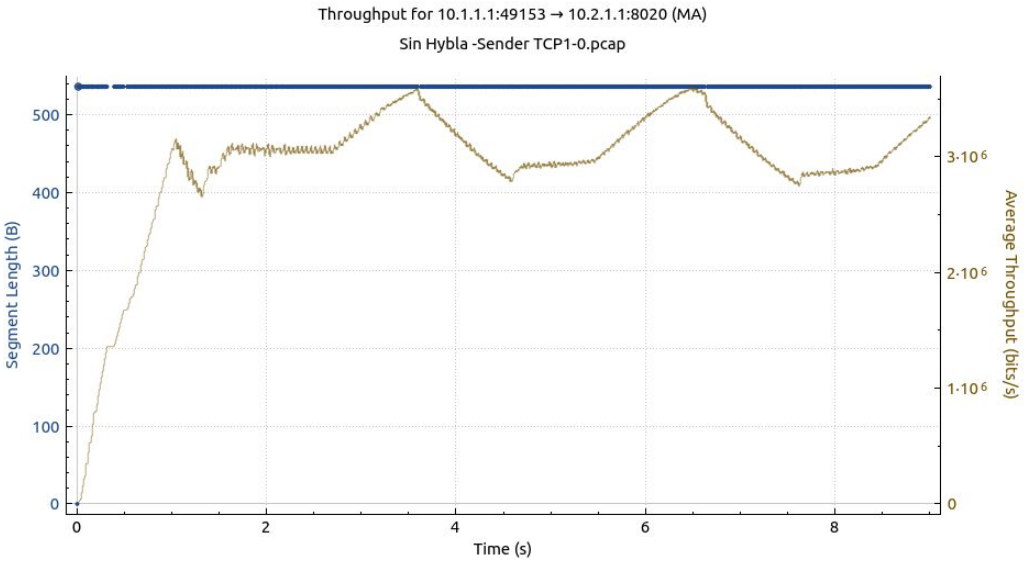


Figura 16: Throughput sin Hybla

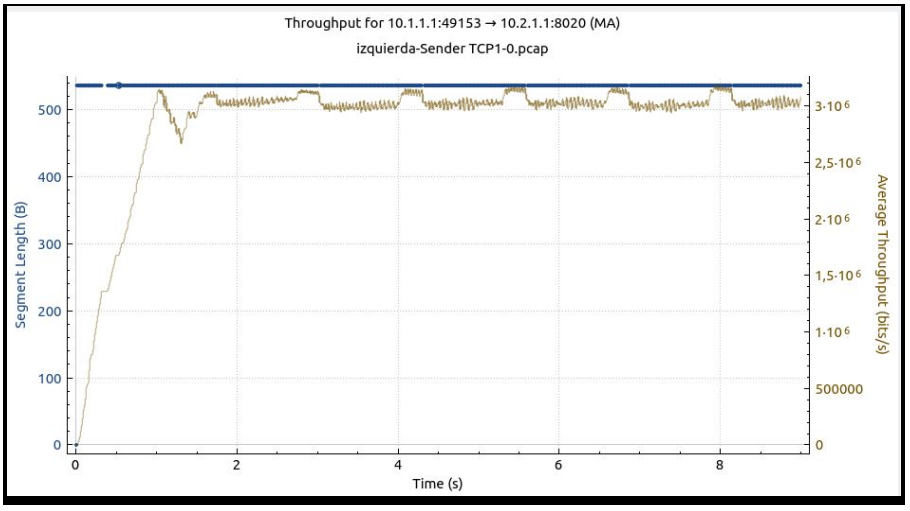


Figura 17: Throughput con Hybla

Podemos observar como la simulación con el algoritmo Hybla mejora el throughput, haciéndolo más estable. Ver imagen 16 y 17.

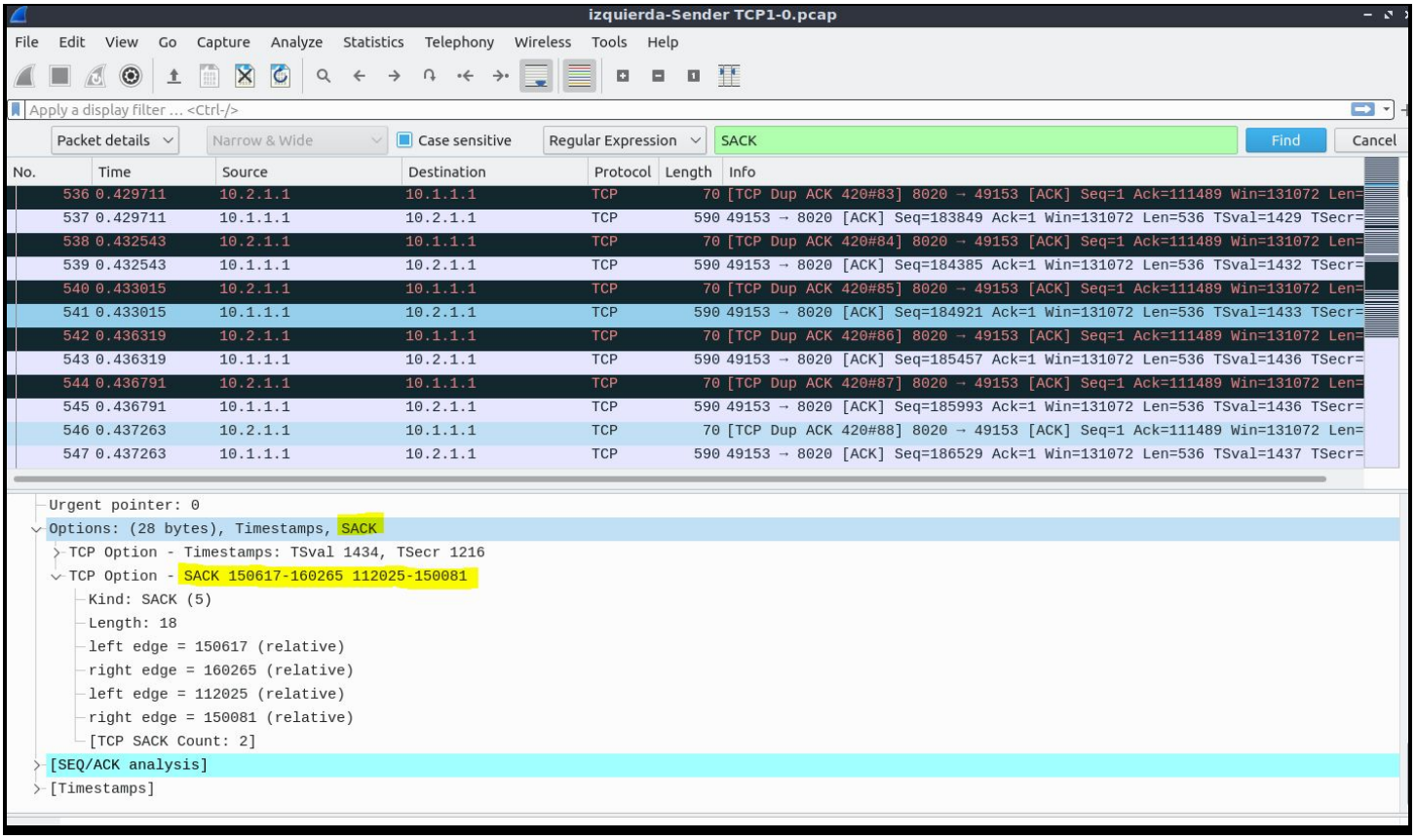


Figura 16: Ejemplo de implementación SACK

En la figura 16, se puede observar que el receptor está intentando enviar un SACK al emisor para avisarle cuales son los paquetes que llegaron correctamente para que retransmita los demás.

Llegamos a la conclusión de que el caso de prueba en el cual utilizamos TCP Hybla es un ambiente favorable para la implementación de este algoritmo. Por más que la red sea cableada y no haya un alto delay (como en Wifi por ejemplo), el ‘delay’ se va a generar en el cuello de botella solo por tener menos capacidad que la suma de los tres emisores TCP. Por lo tanto se activarán los mecanismo del algoritmo y se generará una ventana de congestión al estilo ‘serrucho’ (Figura 14) la cual indica una mayor convergencia.