Universidad Nacional de General Sarmiento

Sistemas Operativos y Redes II

TPN°2 : Análisis de Redes

***Profesores**:*

- Gutierrez, Pedro.
- Tcach, Alexis.

***Integrantes**:*

- De Benedetti, Lautaro. - lautarodebenedetti@gmail.com
- Galvan, Javier. - fjgalvanx87@gmail.com
- Ramos, Javier. - aranibar.javier@gmail.com
- Saczkowski, Sabrina. - sabrina.sacz@gmail.com

2020

Introducción

En este informe vamos a explicar y detallar el análisis de redes realizado en tres partes.

En primer lugar, implementaremos una red de topología Dumbbell con un simulador de redes y haremos varias pruebas sobre la misma: saturación del canal, detección de las etapas del protocolo TCP, calcular el ancho de banda y verificar si existen anomalías. Se analizará tanto para nodos TCP como UDP.

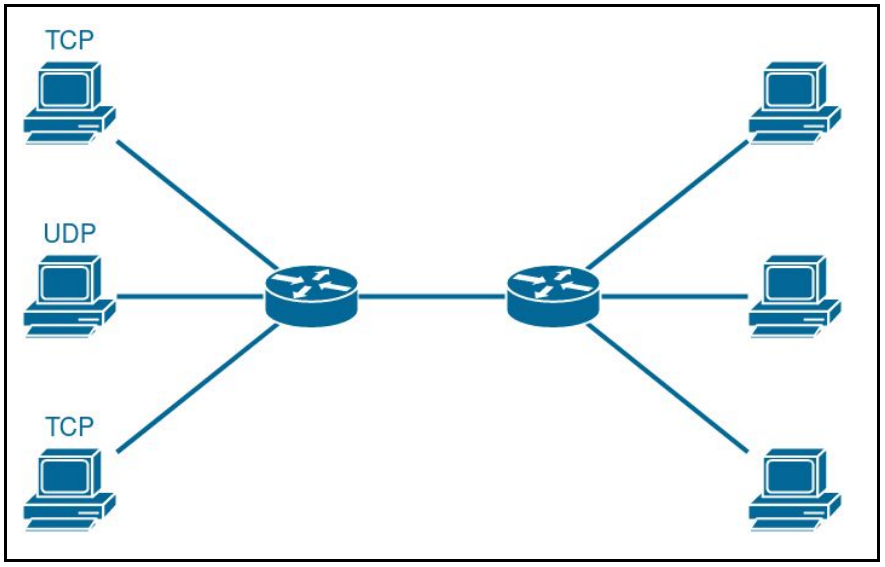

Figura 1. Topología Dumbell.

En segundo lugar, se explicará el algoritmo *NewReno y* se implementará el algoritmo con otra red Dumbell con todos sus nodos TCP.

Luego, se explicará el estándar H323, se analizará un escenario y se realizarán los cálculos necesarios para entender su funcionamiento.

Y finalmente, se detallarán las conclusiones obtenidas.

Preparación del Ambiente

Para el desarrollo y las pruebas del trabajo utilizamos las siguientes distribuciones de Linux:

- Lubuntu 16.04 64Bit.
- Ubuntu 18.04 LTS

A su vez, utilizamos un software de simulación de redes con eventos discretos, desarrollado en C++ y enfocado a sistemas de internet, conocido como *Network Simulator 3* (NS3). Cabe indicar que se trata de un software libre y que cuenta con una licencia GNU GPL v2 (General Public License GNU Versión 2), además se encuentra disponible para cualquier tipo de público, lo cual permite que tenga fines educativos y de investigación.

A continuación se detallarán los pasos para su configuración:

1. Pre-requisitos, instalamos las siguientes **librerías**:

   sudo apt-get install gcc g++ python && sudo apt-get install gcc g++ python && sudo apt-get install mercurial python-setuptools git && sudo apt-get install qt5-default && sudo apt-get install python-pygraphviz python-kiwi python-pygoocanvas libgoocanvas-dev ipython && sudo apt-get install openmpi-bin openmpi-common openmpi-doc libopenmpi-dev && sudo apt-get install autoconf cvs bzr unrar && sudo apt-get install gdb valgrind && sudo apt-get install uncrustify && sudo apt-get install doxygen graphviz imagemagick && sudo apt-get install texlive texlive-extra-utils texlive-latex-extra texlive-font-utils texlive-lang-portuguese dvipng && sudo apt-get install python-sphinx dia && sudo apt-get install gsl-bin libgsl2 libgsl-dev && sudo apt-get install flex bison libfl-dev && sudo apt-get install tcpdump && sudo apt-get install sqlite sqlite3 libsqlite3-dev && sudo apt-get install libxml2 libxml2-dev && sudo apt-get install cmake libc6-dev libc6-dev-i386 libclang-dev && sudo pip install cxxfilt && sudo apt-get install libgtk2.0-0 libgtk2.0-dev && sudo apt-get install vtun lxc && sudo apt-get install libboost-signals-dev libboost-filesystem-dev

2. Descargamos la versión más reciente de NS3. En nuestro caso utilizamos la **release ns-3.30.1.**
   Link: https://www.nsnam.org/releases/ns-3-30/

3. Descomprimimos la carpeta del NS3,nos ubicamos dentro y compilamos:

   ./build.py –enable-examples –enable-tests

4. Nos ubicamos dentro de la carpeta **ns-3.30.1** y ejecutamos:

   ./waf configure

5. Verificamos que tengamos el módulo visual activo, "*PyVisualizer*" en caso de no ser así, instalar las librerías sugeridas entre paréntesis. En nuestro caso se utilizó el gestor de paquetes Synaptic y se tuvieron que instalar las siguientes:

python-dev python3-dev python-gconf python-gi python-gi-cairo python-gobject-2 python-gtk2 python-pyorbit python-redis python-talloc

6. Luego, ubicando los archivos .cc dentro de la carpeta **scratch** se pueden ejecutar los archivos mediante:

   ./waf --run nombre_archivo --vis

Además, se utilizó *Gnuplot,* una herramienta visual para graficar las trazas de las diferentes simulaciones realizadas. Es un programa que permite generar gráficas 2D y 3D. Puede producir resultados directamente en pantalla y en varios formatos de imagen, como PNG, SVG, JPEG, etc. Primero se realizó el seteo de las propiedades de los gráficos, como el nombre de los ejes, el título de los gráficos, los tipos de líneas, y el tipo de salida. Luego, con el comando plot , indicando los archivos de datos a trazar, se realizó la impresión de los datos en gráficos.

Código Fuente

Github: https://github.com/ssaczkowski/TP-SOR2-REDES.git

Etapas de Protocolo TCP

En este apartado vamos a explicar las distintas **etapas del protocolo TCP** para evitar la congestión de la red, para esto el protocolo implementa un mecanismo de control entre el emisor y el receptor.

La congestión ocurre cuando los emisores envían un volumen de paquetes mayor al que un enrutador puede procesar. Si el emisor percibe que hay poca congestión entre él y el receptor va a incrementar su tasa de emisión. Por otro lado, si percibe lo contrario reducirá la misma. Esto lo logrará mediante la variación del tamaño de la ventana *SND.WND*. El tamaño de la ventana *SND.WND* se determinará por el valor de la ventana de recepción y por el valor de la ventana de congestión (cwnd) que dependerá del algoritmo de control de congestión. Este algoritmo percibe la congestión de la red a través de los eventos de pérdidas de segmentos que pueden ser: la recepción de un triple ACK duplicado o la expiración del tiempo de espera (timeout).

El algoritmo de control de congestión se compone de cuatro mecanismos esenciales: arranque lento (***slow start***), evitación de la congestión (***congestion avoidance***), retransmisión rápida (***fast retransmit***) y la recuperación rápida (***fast recovery***). TCP Reno utiliza estos mecanismos y es considerado el algoritmo estándar de control de congestión.

Además del valor cwnd, el algoritmo suma otra variable llamada umbral (***ssthresh***). Esta variable va a determinar si se va a utilizar el mecanismo de arranque lento o el de evitación de la congestión. Cuando la ventana de congestión se encuentre por debajo del umbral se utilizará arranque lento y cuando se encuentre por encima se utilizará evitación de la congestión. El valor de ssthresh se calculará cuando

ocurra un evento de pérdida de segmentos. Se tomará el valor de cwnd (con al menos dos segmentos) y se lo dividirá por 2, por lo tanto, ssthresh = cwnd / 2.

Arranque lento (Slow Start)

Como se explicó antes, este algoritmo se utiliza cuando se está por debajo del umbral, por lo tanto, siempre lo utilizaremos al inicio de la transmisión. Inicialmente el valor de la ventana de congestión se establece en 1 MSS (también puede tener un valor distinto) y se incrementará en un segmento por cada ACK recibido. Por lo tanto, como por cada ACK, cwnd = cwnd + 1, durante esta etapa la ventana de congestión crecerá exponencialmente por cada RTT. No es exactamente exponencial ya que el receptor puede retrasar sus ACK, generalmente enviando un ACK por cada dos segmentos que recibe.

El emisor puede saturar al enrutador intermedio y este comenzará a descartar segmentos para comunicarle al emisor que la ventana de congestión se volvió demasiado grande.

Evitación de la congestión (Congestion Avoidance)

Este algoritmo entra en funcionamiento al momento que el valor de la ventana de congestión supera el valor del umbral. A diferencia del arranque lento, en esta etapa, por cada ACK que se recibe, cwnd = cwnd + 1/cwnd. Por lo tanto, la ventana de congestión se incrementará en 1 MSS por cada RTT, o sea, crecerá de forma lineal.

Retransmisión rápida (Fast Retransmit)

El emisor, controlando los ACKs duplicados, puede detectar una pérdida de segmento antes que expire el tiempo de espera (timeout). El ACK duplicado tendrá el número que correspondía a la secuencia que se esperaba recibir.

Si se reciben tres ACKs duplicados el emisor considera a un segmento como perdido. Aquí es donde el mecanismo de retransmisión rápida se ejecuta y realiza la retransmisión del segmento perdido antes de que el tiempo de espera termine.

El emisor asume que si existió un re-orden en la entrega de segmentos pueden existir uno o dos ACKs duplicados. Por lo tanto, al desconocer si los primeros dos ACKs duplicados son por un reordenamiento o por una pérdida, se debe esperar a recibir tres ACKs duplicados para dar por perdido un segmento.

Recuperación rápida (Fast Recovery)

Este algoritmo se ejecuta cuando el emisor recibe tres ACKs duplicados en la etapa de prevención de la congestión, aquí se asume que el segmento se perdió y lo retransmite inmediatamente. Los segmentos recibidos luego del segmento perdido generaron los ACKs duplicados, esto quiere decir que todavía hay datos entre los dos extremos. TCP no quiere reducir el flujo bruscamente por lo tanto, evita el ingreso a la etapa de arranque lento y se mantiene en la de evitación de la congestión. Luego establece el valor del umbral a la mitad del valor de la ventana de congestión (ssthresh = cwnd / 2) y el valor de la ventana lo establece en el valor del umbral + 3 (cwnd = ssthresh + 3), esto lo hace por cada ACK duplicado que recibió de los tres segmentos que ya habían sido transmitidos luego del segmento perdido. También, si llega otro ACK duplicado se aumenta el cwnd por el segmento adicional recibido, si es posible, se transmitirá un nuevo segmento.

Esta etapa finalizará una vez que se recibió el ACK del segmento retransmitido, luego vuelve a la fase de evitación de la congestión. Si el evento de pérdida fue un timeout se volverá a la fase de arranque lento.

Esta mejora y produce un alto rendimiento cuando se tiene una congestión moderada y especialmente para ventanas grandes.

Desarrollo

Parte 1

Se diseñó un escenario con 3 emisores on/off application, 3 receptores y dos nodos intermedios. Se conectaron los 3 emisores a un nodo, luego éste a otro y finalmente éste a los 3 destinos finales. Uno de los emisores se definió como UDP y los otros 2 TCP. Con conexiones cableadas.

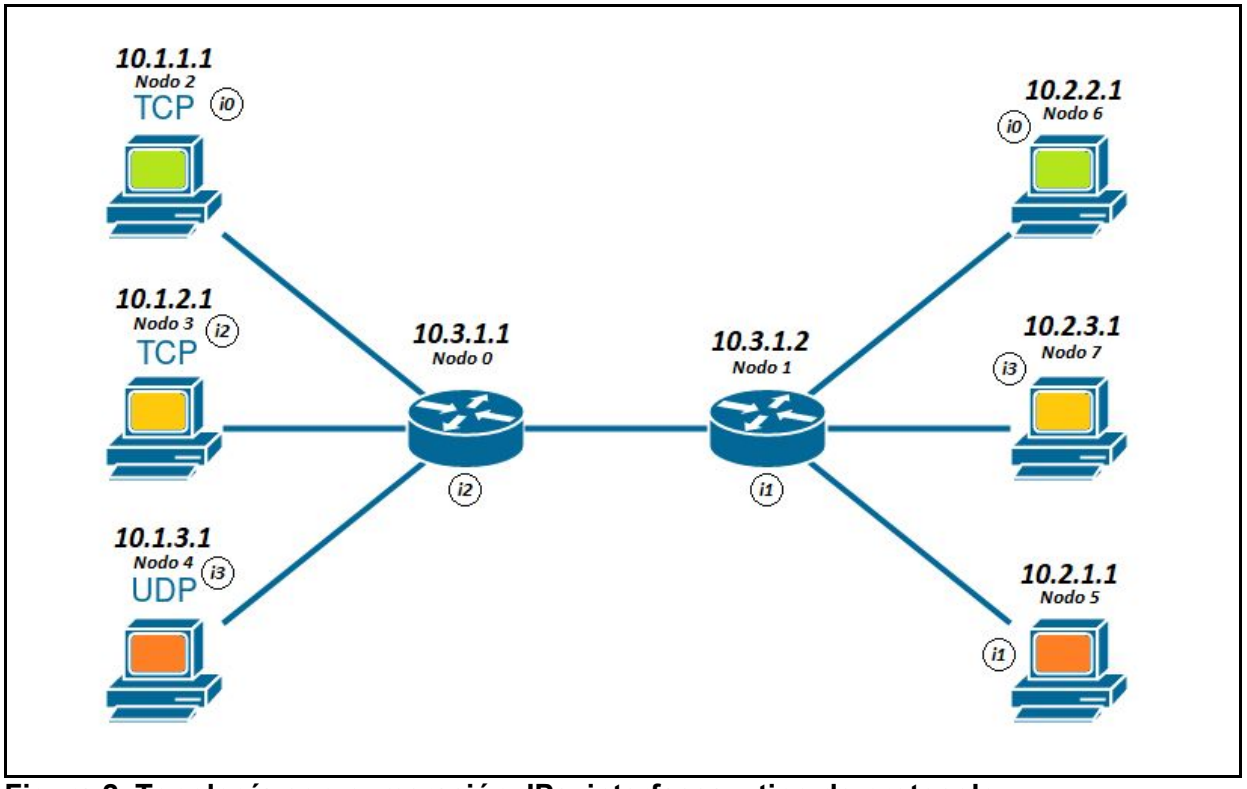

Figura 2. Topología con numeración, IPs, interfaces y tipo de protocolo.

En primer lugar, se realizaron pruebas con los **2 emisores TCP**. El caso se puede obtener dentro de la carpeta **/cases/tcp** del repositorio. Si abrimos los archivos **.pcap** mediante Wireshark, se observa una congestión a los **8 segundos** por ambos emisores TCP (ver Figura 3).

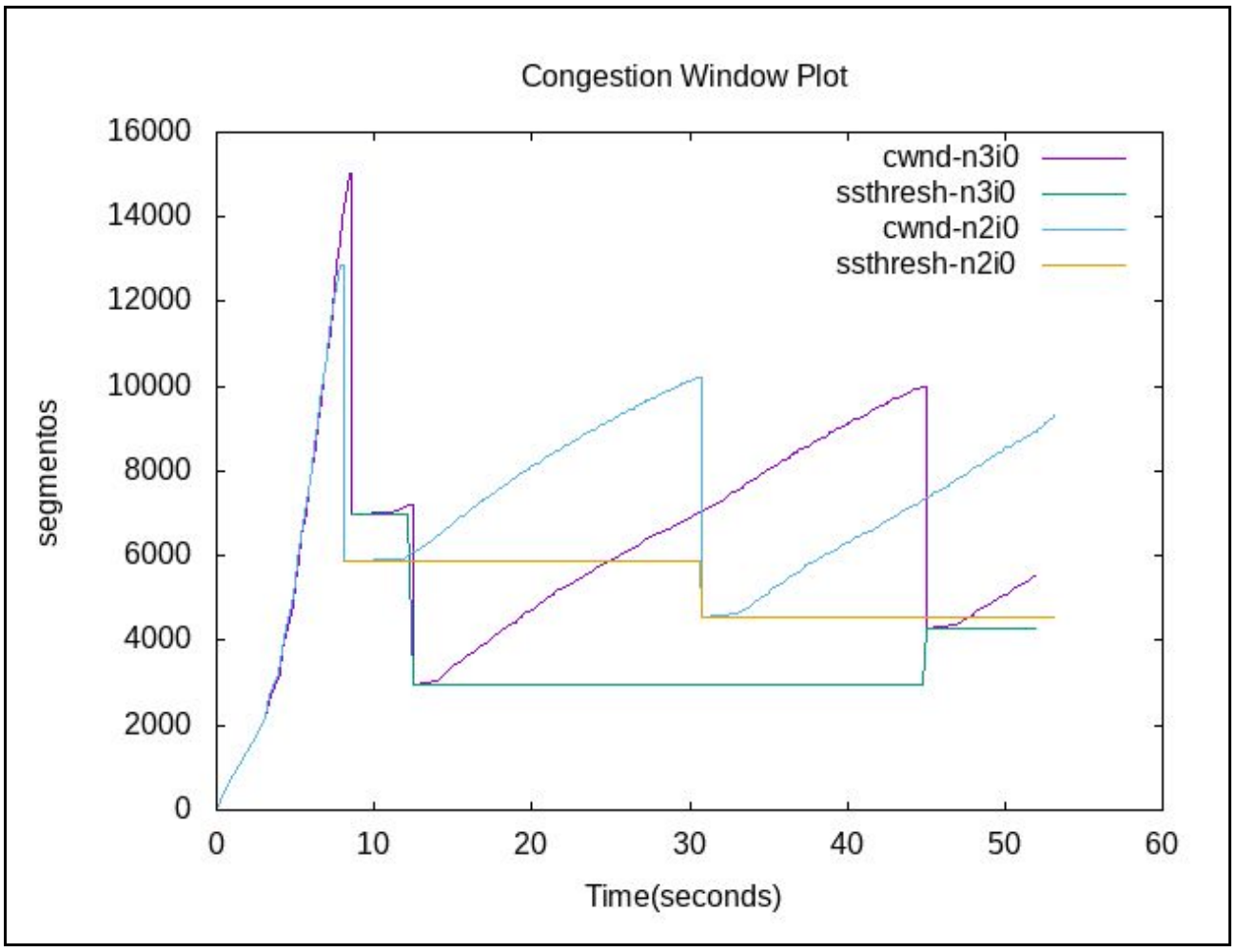

Figura 3. Ventana de congestión de emisores TCP obtenida del código C++ por gnuplot.

Además, si analizamos los bits de transferencia en ese momento, se puede notar una reducción. En la *Figura 4 y 5*, es decir, vemos los bits de transferencia correspondientes a los emisores TCP a los 8 segundos de la simulación.

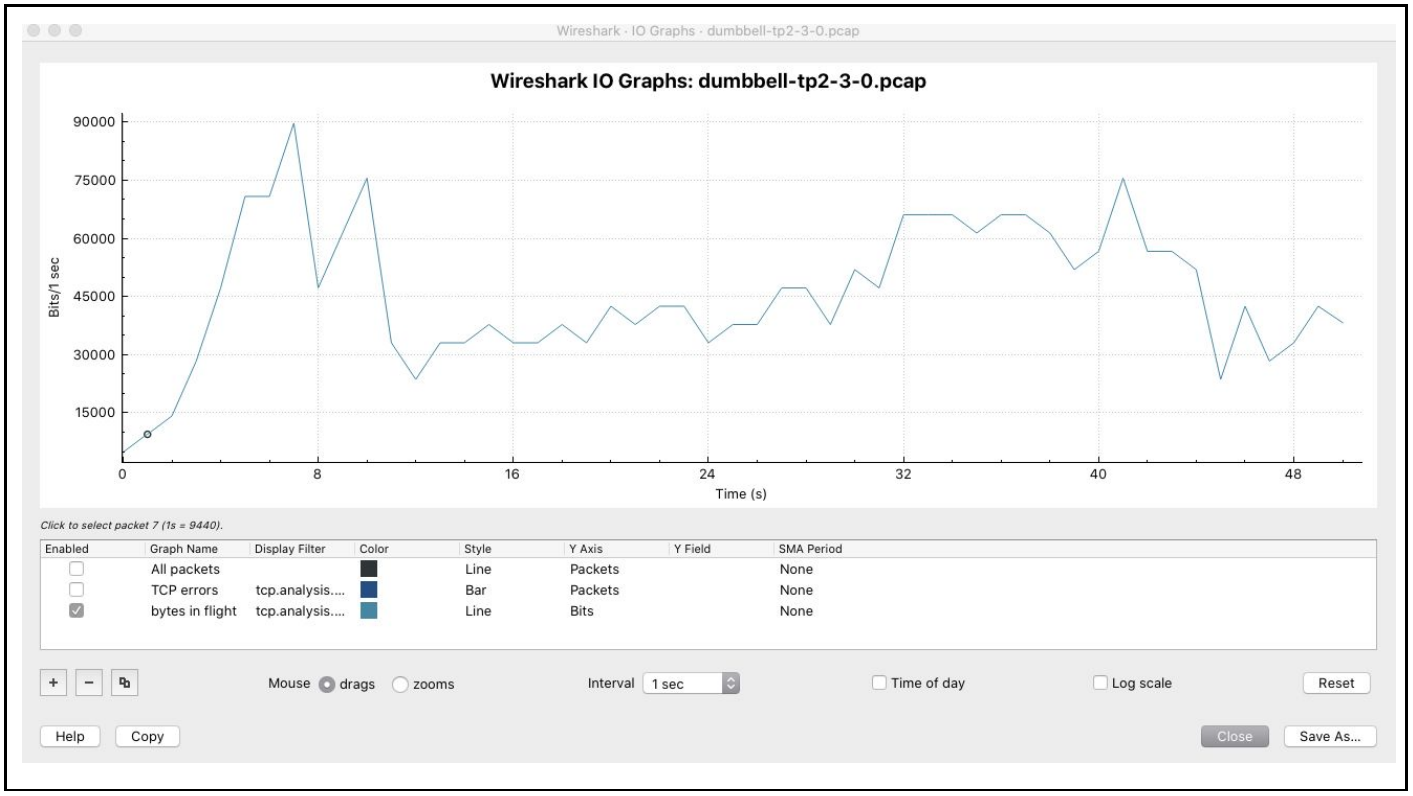

Figura 4. Bits de transferencia del nodo 3 con protocolo TCP.

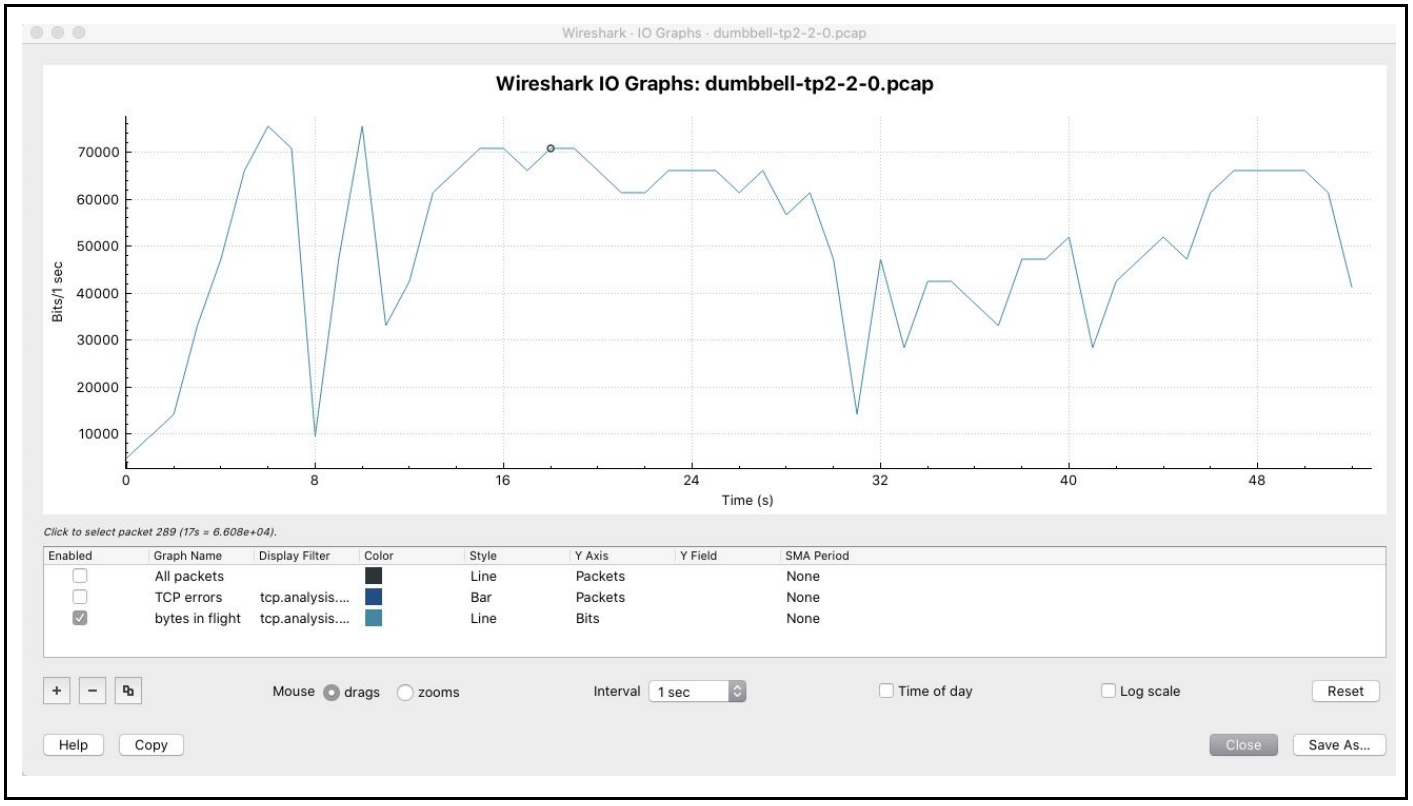

Figura 5. Bits de transferencia del nodo 2 con protocolo TCP.

Ahora vamos a analizar la velocidad de transferencia. Por un lado, observamos los gráficos de la opción “Statistics” del menú, más precisamente utilizamos el tipo de gráfica ***Flow Graph***. Y por el otro, vamos a medir la velocidad de transferencia en el momento anterior a la congestión, es decir, a partir del último ACK de confirmación.

En la *Figura 6* se puede ver el comienzo de la transmisión que vamos a analizar. Notamos los tres paquetes enviados al receptor antes del ACK de confirmación al emisor, esto ocurre a los 7.661119 hasta los 7,85183 segundos.

La resta de estos dos tiempos, es el tiempo que tardó en transferirse estos tres paquetes. Con lo cual, obtenemos el tiempo de 1 RTT más 2 deltas.

Situándonos en estos paquetes en la pantalla principal, podemos obtener la cantidad de datos en bytes de cada paquete (ver Figura), que es de 536 bytes.

Si la cantidad de paquetes es igual a 3, luego, obtenemos los bytes transferidos en ese tiempo, que es 536 bytes * 3 = 1608 bytes. La división del total de bytes por el tiempo calculado nos proporciona una aproximación de la velocidad de transferencia o ancho de banda del enlace.

Luego, la velocidad de transferencia en el momento analizado es de 67,449 kbit/s.

Velocidad = 1608 Bytes / (7,851839 s -7.661119 s) = 8431,21 Bytes/S = 67449,68 bit / s = **67,449 kbit/s**

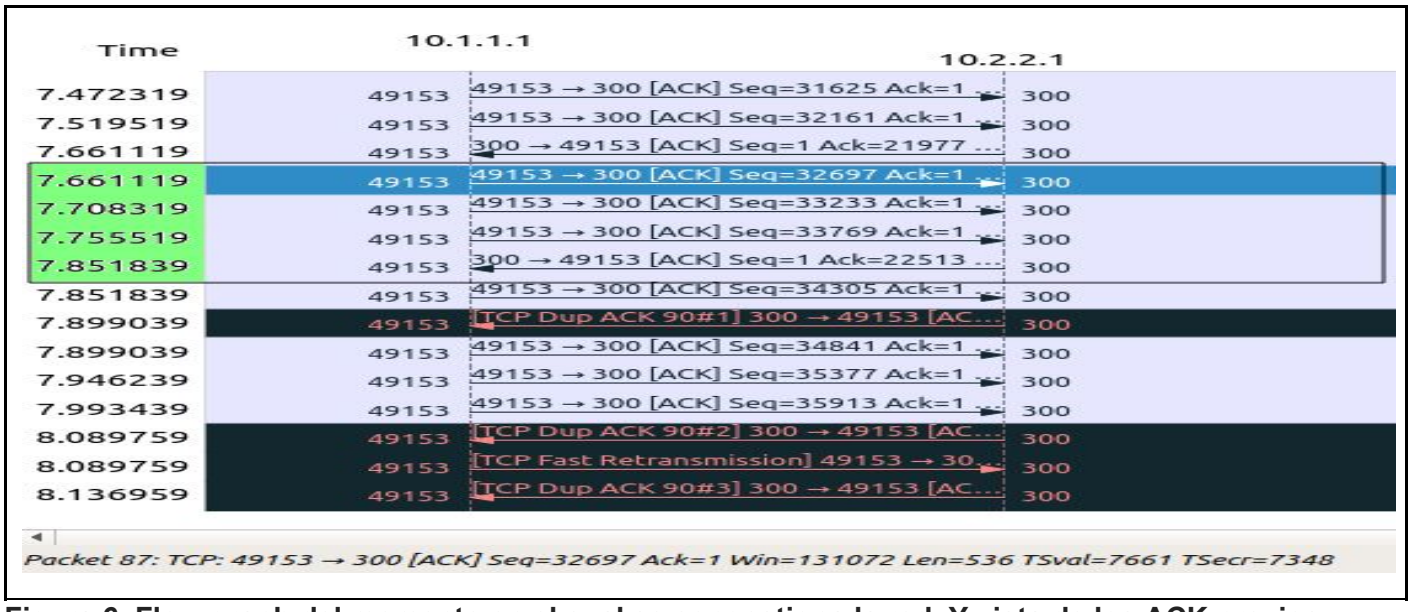

Figura 6. Flow graph del momento en el cual se congestiona la red. Y vista de los ACK previos.

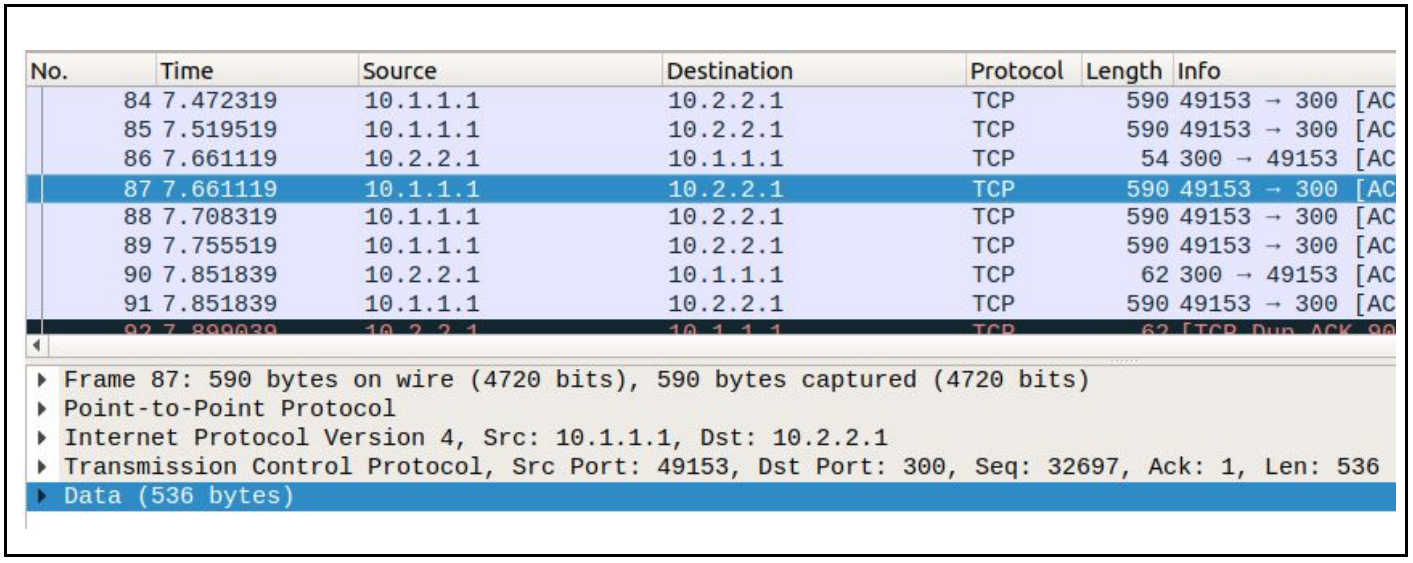

Figura 7. Cantidad de bytes de datos de cada paquete. En este caso observamos el No. 87 de 536 bytes.

Para conocer la cantidad de paquetes que llegaron a destino vamos a aplicar un filtro dentro de *Wireshark* con el .pcap (ubicado en, *cases/tcp/dumbbell-tp2-1-0.pcap*) del enlace del nodo 2 al router del nodo 0:

ip.src == 10.1.1.1 || ip.dst == 10.2.2.1

Con lo cual podemos ver los paquetes relacionados con el nodo 2 y el nodo 6 (ver Figura 8).

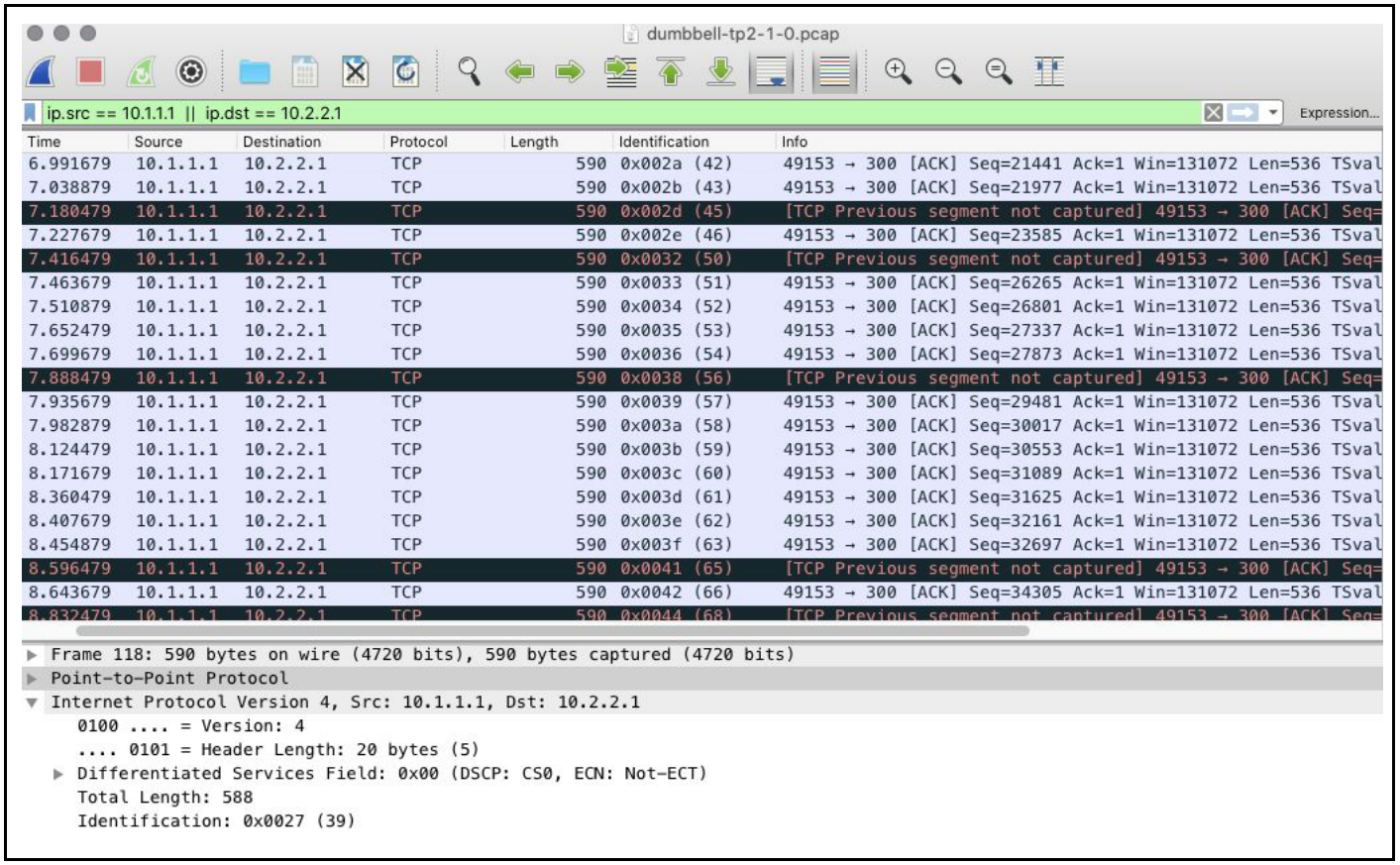

Figura 8. Momento en que se detectan la pérdida de los primeros paquetes del nodo 2 al nodo 6.

Detección de pérdida de paquetes, son los paquetes que provienen del nodo 2 tcp 10.1.1.1 de tipo emisor al nodo 6 tcp 10.2.2.1 de tipo receptor. Se observa que no le llega al nodo 6 receptor tcp el paquete nro 44, 47-48 y 55. Esto provocará el mensaje de ACK duplicados hacia el emisor nodo 2 tcp 10.1.1.1, lo que significa que cuando llegue el tercer ACK duplicado de que se perdió paquetes. Lo que provoca que el nodo 2 entre en en segundo 8,089 que entre en Fast Retransmit debido a que al receptor nodo 6 no le llegó paquetes (ver **Figura 8).**

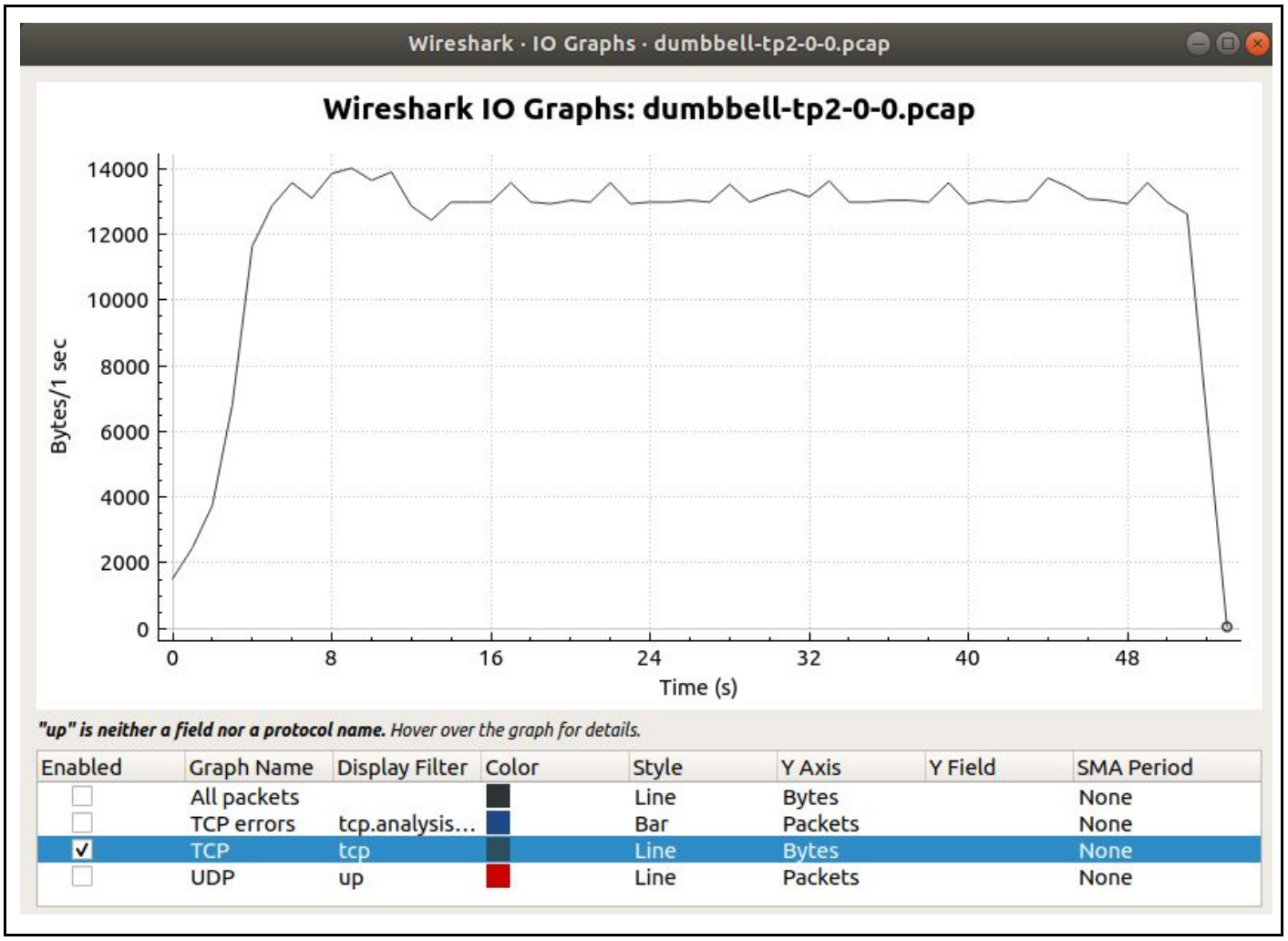

Figura 9. Rendimiento del ancho de banda efectivo del canal, entre routers, solo con emisores y receptores TCP.

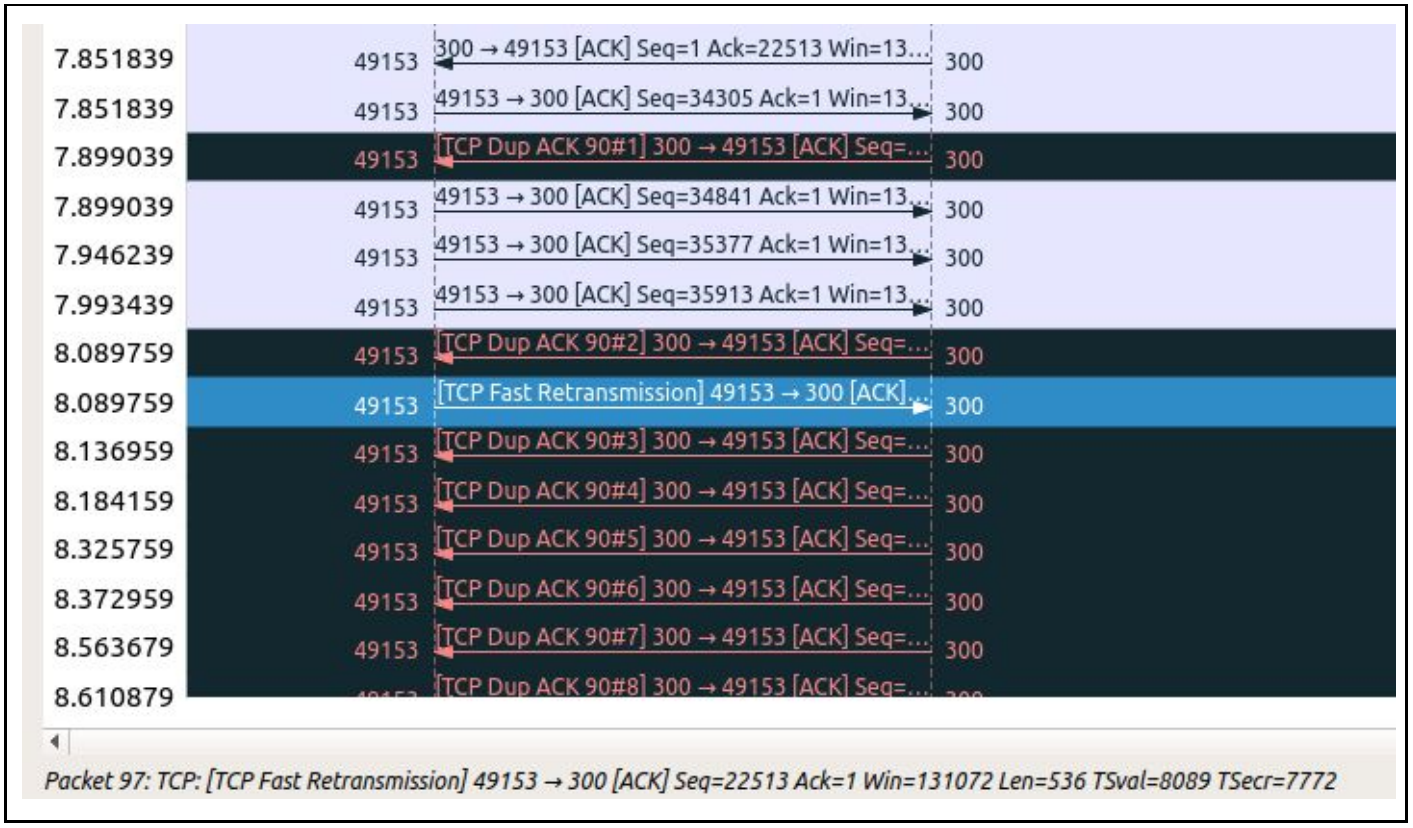

Figura 10. 3 ACK y Retransmisión de paquetes a los 8 segundos.

Podemos observar en la figura 8 que surgen ACKs duplicados y a los 8 segundos se pasa al mecanismo ***"Fast Retransmit".***

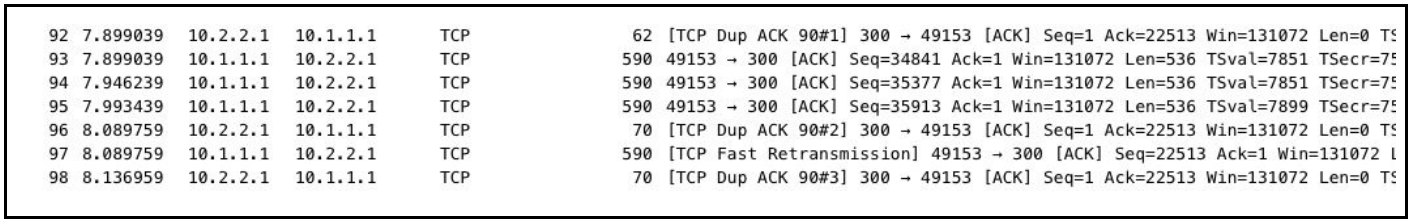

Figura 11. Ventanas de recepción en el momento de congestión de la red (ver variable *"Win").*

En la figura 9 vemos como las colas de recepción se mantienen en el mismo tamaño *"Win=131072".* Esto se debe a que la congestión en el router (nodo 0) no permite que avancen los paquetes. Por lo tanto, el receptor procesa con rapidez los paquetes que le llega (al ser pocos) de tal forma que no actualiza su ventana de recepción.

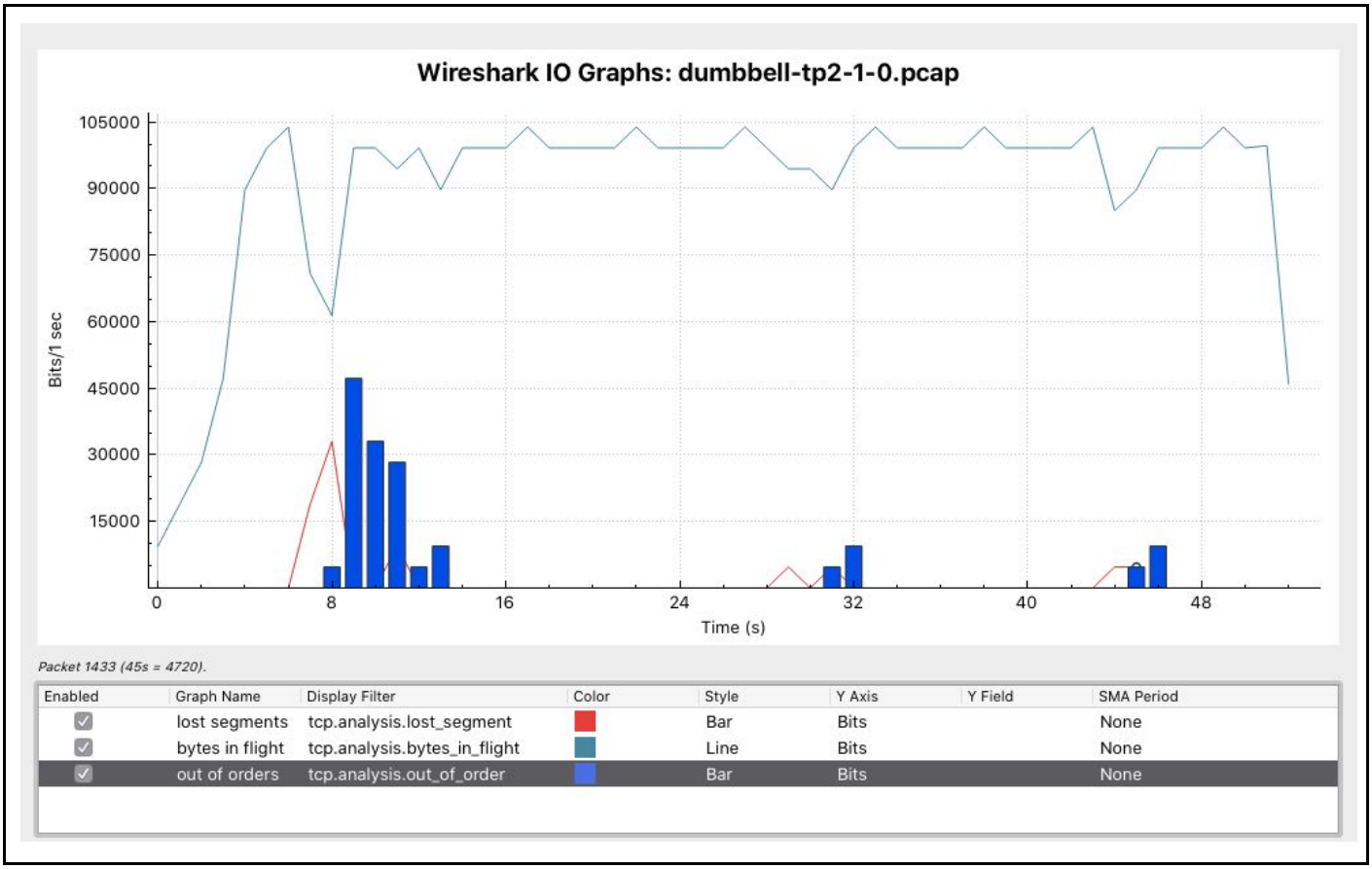

Figura 12. Archivo .pcap correspondiente al nodo 0 (router) y al nodo 6 (receptor).

En la figura 10 podemos ver el mayor pico de pérdida de paquetes en el momento de la congestión seguido de la detección de “paquetes fuera de orden”.

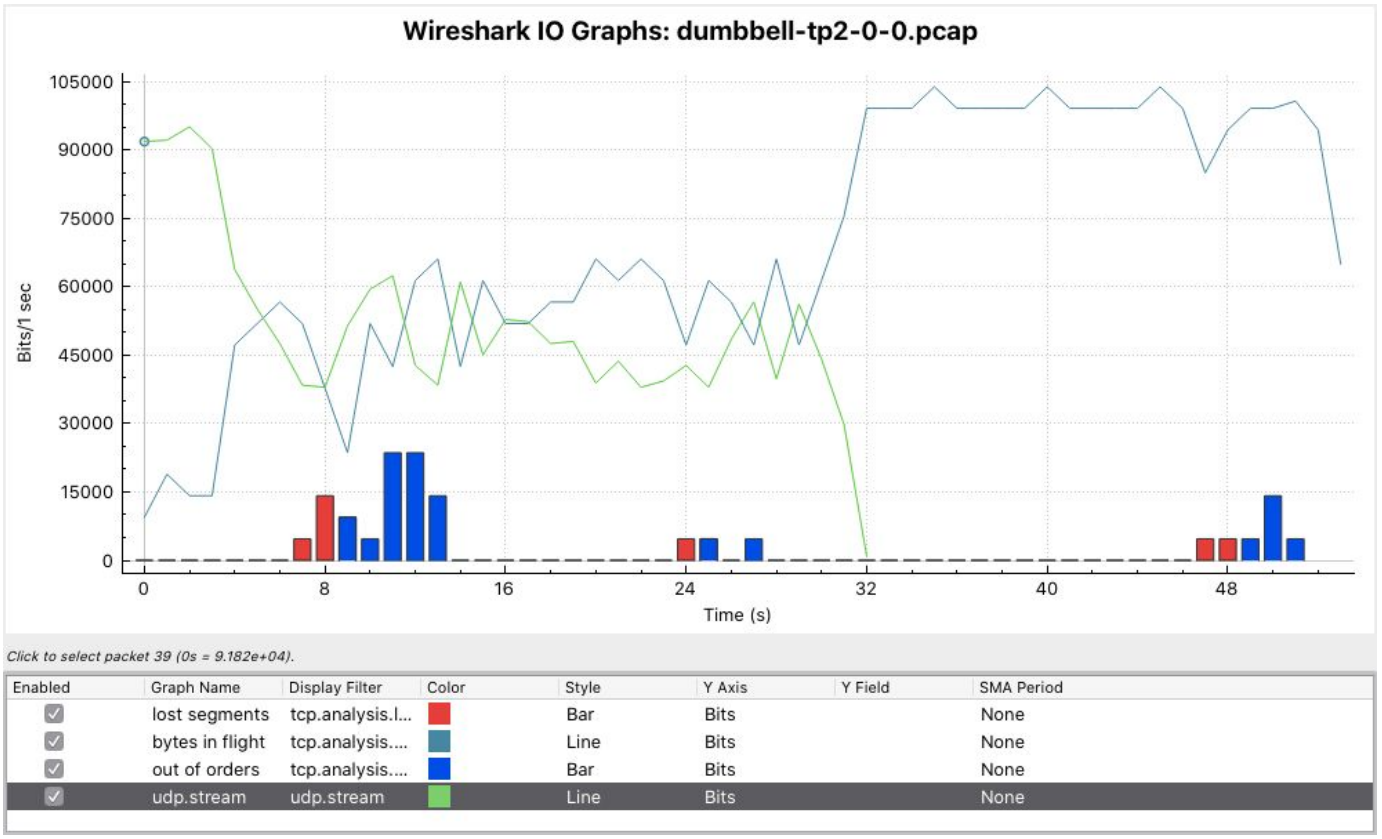

Figura 13. Comparación entre el flujo de datos UDP y TCP.

Ahora la topología cuenta con otro protocolo para la transferencia de paquetes, UDP. Cabe aclarar que este protocolo tiene características opuestas a TCP, como por ejemplo está orientado a la no conexión, lo que significa que no pierde el tiempo en establecer una comunicación sino que envía paquetes ni bien inicia. Por otro lado, sabemos que TCP tiene un set de algoritmos para incrementar en envío de paquetes a la red, por lo cual este envío de paquetes a la red va variando en el tiempo, en cambio UDP no aplica ningún algoritmo de ajuste de envio de paquetes, por lo tanto envía la misma cantidad de paquetes a lo largo de la comunicación (ver Figura 9).

Otra observación que se da en los gráficos, es que UDP finaliza primero en enviar sus paquetes que TCP. Esto es debido a las características ya mencionadas de los protocolos. Por ejemplo, TCP puede demorar más en transmitir toda la información porque puede que se pierdan paquetes y los tenga que re-transmitir, en cambio UDP no lleva un control de paquetes recibidos. Podemos decir entonces que bajo la misma cantidad de paquetes que debemos enviar, el gráfico bps (velocidad de transferencia) de UDP se reduce a 0 mucho antes que TCP por las características de los protocolos.

Podemos ver las diferentes etapas del protocolo en la figura 11.

Figura 14. ACK duplicados y retransmisión en simulación con nodos TCP y UDP.

Luego, se redujo el ancho de banda entre los routers para forzar una congestión más fuerte. Y se obtuvieron los siguientes gráficos que demuestran que fue exitoso ya que se presentan pérdidas de segmentos, ACKs duplicados y *timeouts* que provocan el cambio a los distintos algoritmos y la actualización correspondientes de los parámetros CWND y SSTRESH.

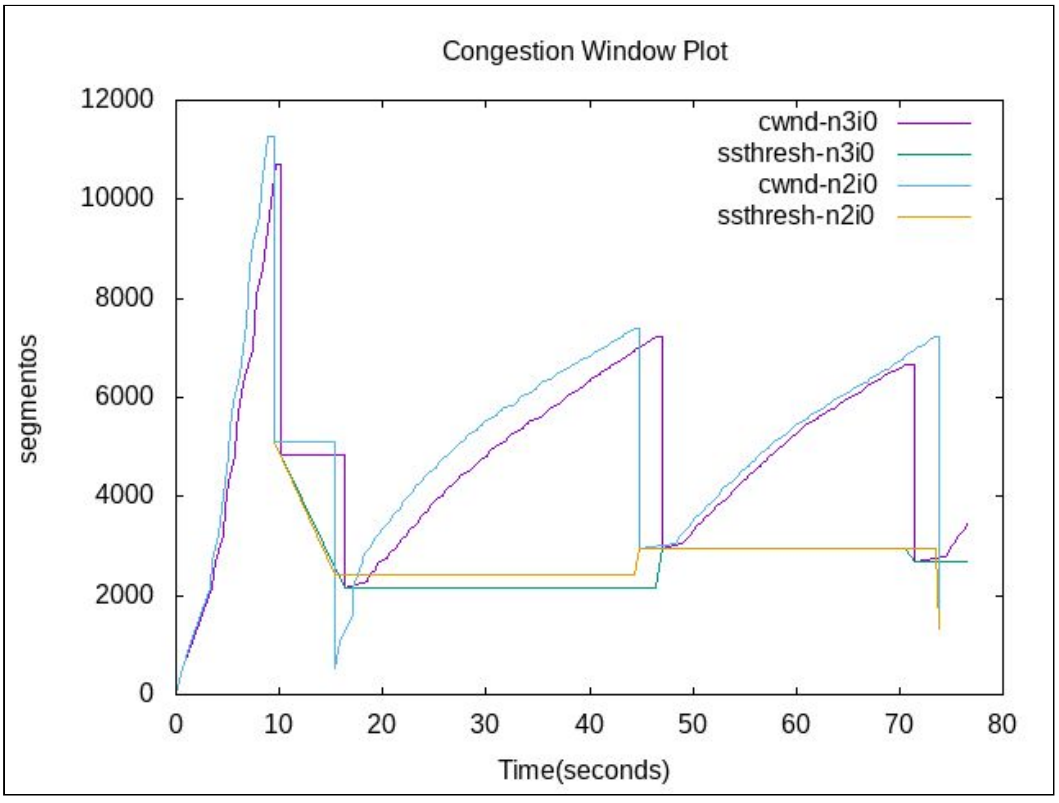

Figura 15. Ventana de congestión para los enlaces emisores/router.

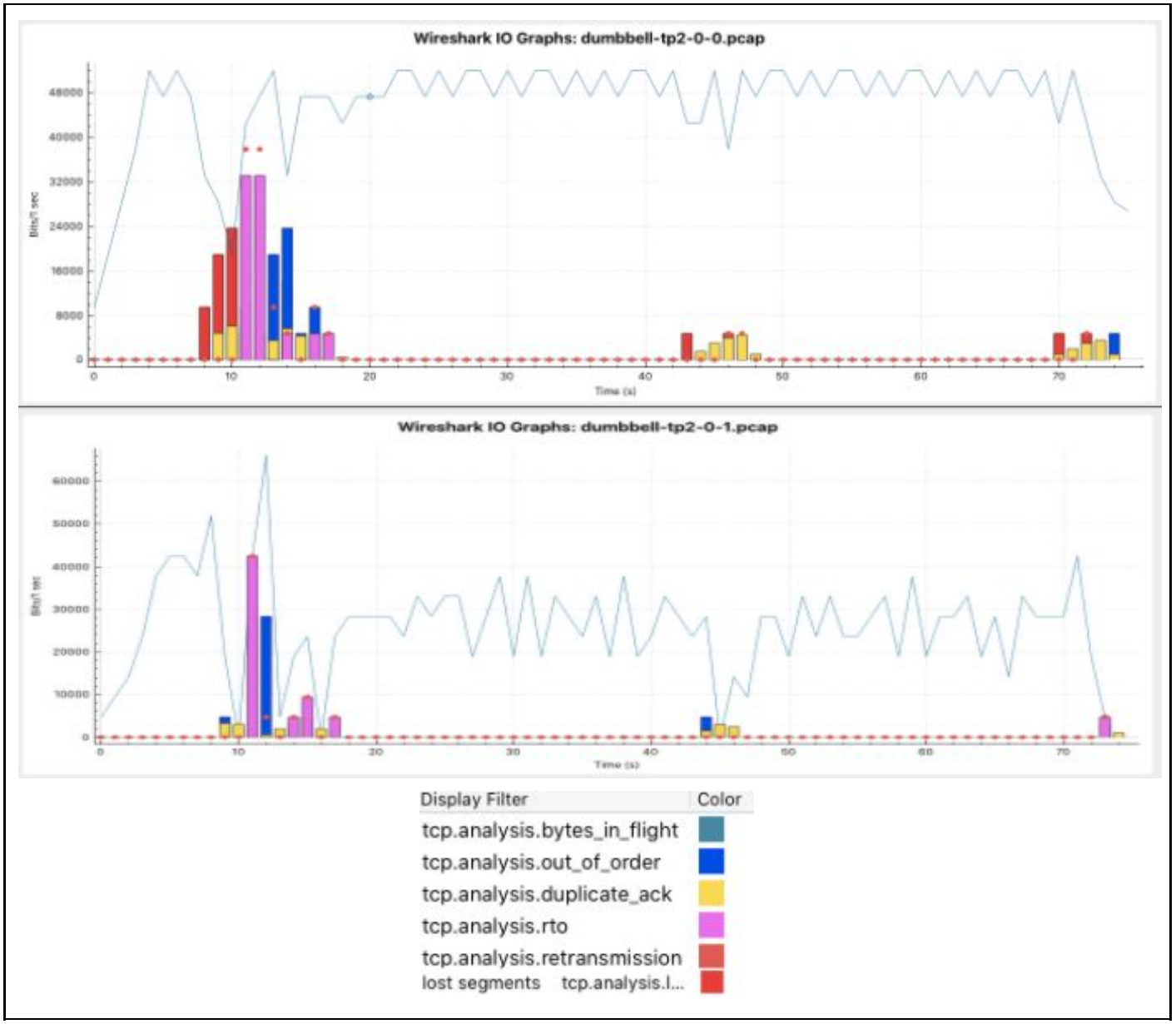

Figura 16. Estadísticas de Wireshark correspondiente al enlace entre los routers.

Parte 2

TCP NewReno

TCP Reno proponía como máximo una retransmisión por cada RTT, sin embargo, no tiene en cuenta casos con múltiples pérdidas consecutivas. Por consiguiente, en 1999 se desarrolló el algoritmo de New Reno. Este NewReno conserva el principio básico de TCP, como los arranques lentos y el temporizador de retransmisión de grano grueso.

El mecanismo de control de congestión existente de NewReno como se ilustra en la **figura X** de la ventana de congestión de NewReno, también conocido como AIMD (Aumento Aditivo, Disminución Multiplicativa), comprende cuatro fases, fase de inicio lento, control de congestión, retransmisión rápida y fase de recuperación rápida.

En la fase de inicio lento, la ventana de congestión (CWND) aumentará en 1 por cada paquete recibido, por lo que el tamaño de cwnd crecerá 2 veces por RTT. Significa que cwnd aumenta exponencialmente (2n) por RTT, mientras que en la fase de prevención de congestión, CWND aumentará 1 / CWND por paquete. Por lo tanto, el tamaño de la CWND aumentará en 1 por RTT. Este mecanismo es muy útil en redes con un ancho de banda amplio, porque el logro del ancho de banda máximo se puede hacer rápidamente.

La ocurrencia frecuente de congestión aumentará la necesidad de recursos. Por lo tanto, es necesario realizar una mejora que pueda minimizar la congestión, pero, al mismo tiempo, la utilización del ancho de banda aún se puede maximizar.

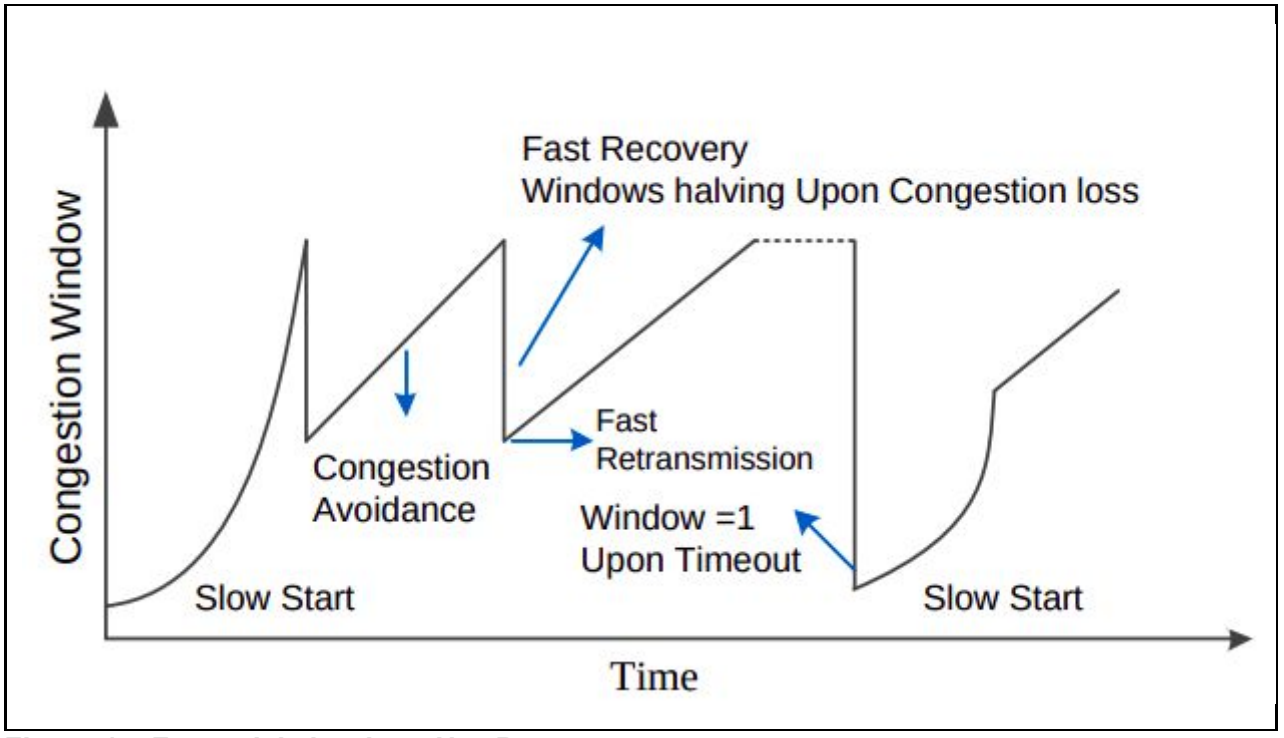

Figura 17. Fases del algoritmo NewReno.

El control de congestión TCP supone que las pérdidas de paquetes ocurren como resultado de la congestión en la red, y responde con una disminución multiplicativa disminuyendo el tamaño del cwnd a la mitad (cwnd = cwnd / 2) o 50%.

NewReno introdujo un pequeño refinamiento en la etapa de recuperación rápida para solventar el problema de múltiples pérdidas consecutivas. A diferencia de Reno, cuando recibe tres ACKs duplicados no sale de la fase de recuperación rápida hasta recibir los ACKs de todos los segmentos pendientes. NewReno utiliza una variable adicional donde guarda el número de secuencia del último segmento. Con lo cual NewReno resuelve el problema de múltiples pérdidas en la ventana de congestión pero genera un exceso en el tiempo de recuperación, ya que demora un RTT en recuperar cada segmento perdido.

Para establecer el tipo de socket predeterminado antes de que se creen objetos relacionados con la pila de Internet, se puede colocar la siguiente declaración en la parte superior del programa de simulación:

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

Figura 18. Implementación algoritmo Newreno.

A continuación se muestra la simulación con 2 emisores TCP con el algoritmo de NewReno. Dejando deshabilitado el del emisor y receptor UDP.

Parámetros	**Valores**
Ancho de banda en nodos intermedios, emisores y receptores.	100 kb/s
Delay en nodos intermedios, emisores y receptores.	100 ms
Capacidad máxima de cola de paquetes.	10 p.
Nodos emisores OnOffHelper. Modo OnTime	1000.0

Figura 19. Tabla de parámetros utilizados durante la simulación con NS-3.

Dentro de Wireshark con el pcap 1-0, aplicando el filtro de **“ip.src == 10.1.1.1 || ip.dst == 10.2.2.1”** para solo ver los paquetes relacionados con nodo 2 emisor y nodo 6 como receptor ambos con tcp newreno.

Se puede observar que al nodo 6 no le llegan los paquetes nros. 44, 47, 48, 49 y 55. Esto provoca la devolución de ACKs duplicados, lo que generará una retransmisión por parte del nodo 2.

ip.src == 10.1.1.1 || ip.dst == 10.2.2.1

No.	Time	Source	Destination	Protocol	Length	Identification	Info
118	6.566879	10.1.1.1	10.2.2.1	TCP	590	0x0027 (39)	49153 → 300 [ACK] Seq=19833 Ack=1 \
124	6.755679	10.1.1.1	10.2.2.1	TCP	590	0x0028 (40)	49153 → 300 [ACK] Seq=20369 Ack=1 \
130	6.944479	10.1.1.1	10.2.2.1	TCP	590	0x0029 (41)	49153 → 300 [ACK] Seq=20905 Ack=1 \
131	6.991679	10.1.1.1	10.2.2.1	TCP	590	0x002a (42)	49153 → 300 [ACK] Seq=21441 Ack=1 \
133	7.038879	10.1.1.1	10.2.2.1	TCP	590	0x002b (43)	49153 → 300 [ACK] Seq=21977 Ack=1 \
138	7.180479	10.1.1.1	10.2.2.1	TCP	590	0x002d (45)	[TCP Previous segment not captured
139	7.227679	10.1.1.1	10.2.2.1	TCP	590	0x002e (46)	49153 → 300 [ACK] Seq=23585 Ack=1 \
145	7.416479	10.1.1.1	10.2.2.1	TCP	590	0x0032 (50)	[TCP Previous segment not captured
147	7.463679	10.1.1.1	10.2.2.1	TCP	590	0x0033 (51)	49153 → 300 [ACK] Seq=26265 Ack=1 \
149	7.510879	10.1.1.1	10.2.2.1	TCP	590	0x0034 (52)	49153 → 300 [ACK] Seq=26801 Ack=1 \
153	7.652479	10.1.1.1	10.2.2.1	TCP	590	0x0035 (53)	49153 → 300 [ACK] Seq=27337 Ack=1 \
155	7.699679	10.1.1.1	10.2.2.1	TCP	590	0x0036 (54)	49153 → 300 [ACK] Seq=27873 Ack=1 \
162	7.888479	10.1.1.1	10.2.2.1	TCP	590	0x0038 (56)	[TCP Previous segment not captured
164	7.935679	10.1.1.1	10.2.2.1	TCP	590	0x0039 (57)	49153 → 300 [ACK] Seq=29481 Ack=1 \
166	7.982879	10.1.1.1	10.2.2.1	TCP	590	0x003a (58)	49153 → 300 [ACK] Seq=30017 Ack=1 \
172	8.124479	10.1.1.1	10.2.2.1	TCP	590	0x003b (59)	49153 → 300 [ACK] Seq=30553 Ack=1 \
174	8.171679	10.1.1.1	10.2.2.1	TCP	590	0x003c (60)	49153 → 300 [ACK] Seq=31089 Ack=1 \

Figura 20. Momento en que se detectan la pérdida de los primeros paquetes del nodo 2 al nodo 6.

En la siguiente imagen se puede mostrar las diferentes etapas de los nodos tcp newreno 2 y 3, al dirigirse a los nodos receptores tcp 6 y 7. Particularmente se puede observar lo que ocurre en el segundo 8, provocado por la pérdida de paquetes mostradas en la imagen anterior. En la siguiente imagen se pueden apreciar las etapas de slow start, fast recovery fast retransmit, congestion avoidance y el ssthresh de los nodos emisores tcp newreno n2 y n3.

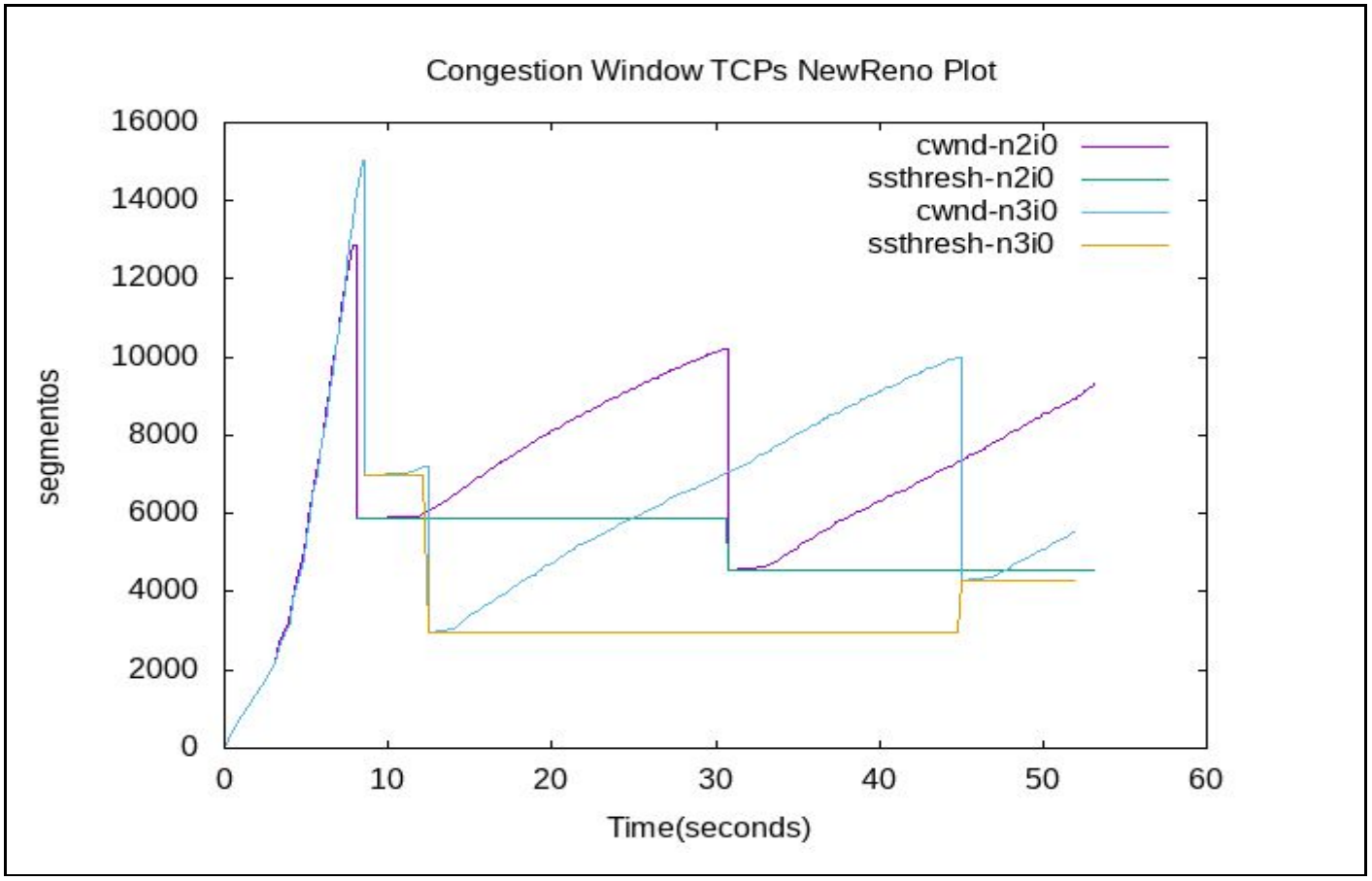

Figura 21. Ventana de congestión de los enlaces entre emisores y routers.

Con wireshark mirando el pcap 2-0, se puede observar el momento en que entra el emisor nodo 2 tcp en que cambia de Slow Start a Fast Retransmit en el segundo 8.089.

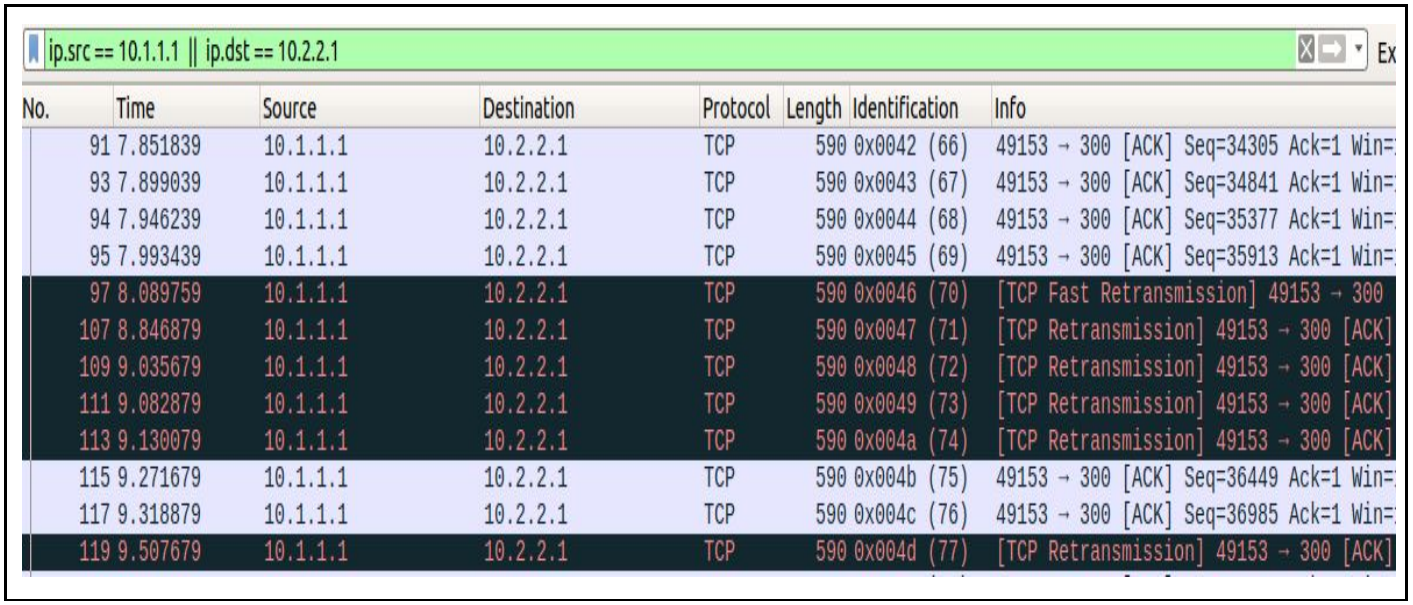

Figura 22. Etapa de Fast Retransmission del TCP NewReno.

Dentro de Wireshark con el pcap 1-0, aplicando el filtro de “ip.src == 10.1.1.1 || ip.dst == 10.2.2.1” para solo ver los paquetes relacionados con nodo 2 emisor y nodo 6 como receptor ambos con tcp newreno.

Se puede observar que al nodo 6 no le llegan los paquetes nros. 44, 47-57, 55. Esto provoca la devolución de ACKs duplicados, lo que generará una retransmisión por parte del nodo 2.

En la siguiente imagen se puede apreciar el ancho de banda del canal de los dos emisores TCP NewReno hacia los dos receptores.

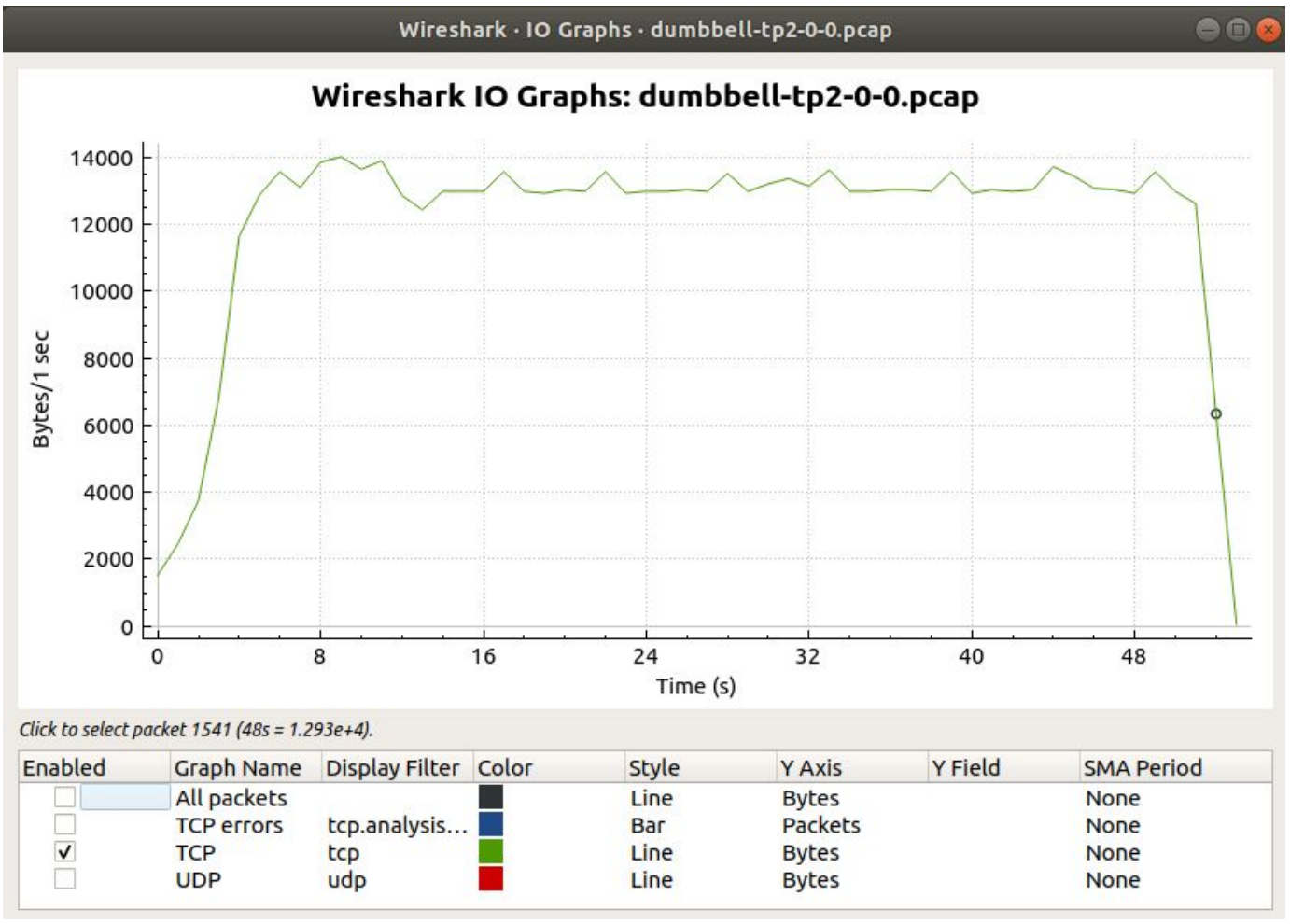

Figura 23. Flujo de datos TCP NewReno por el canal.

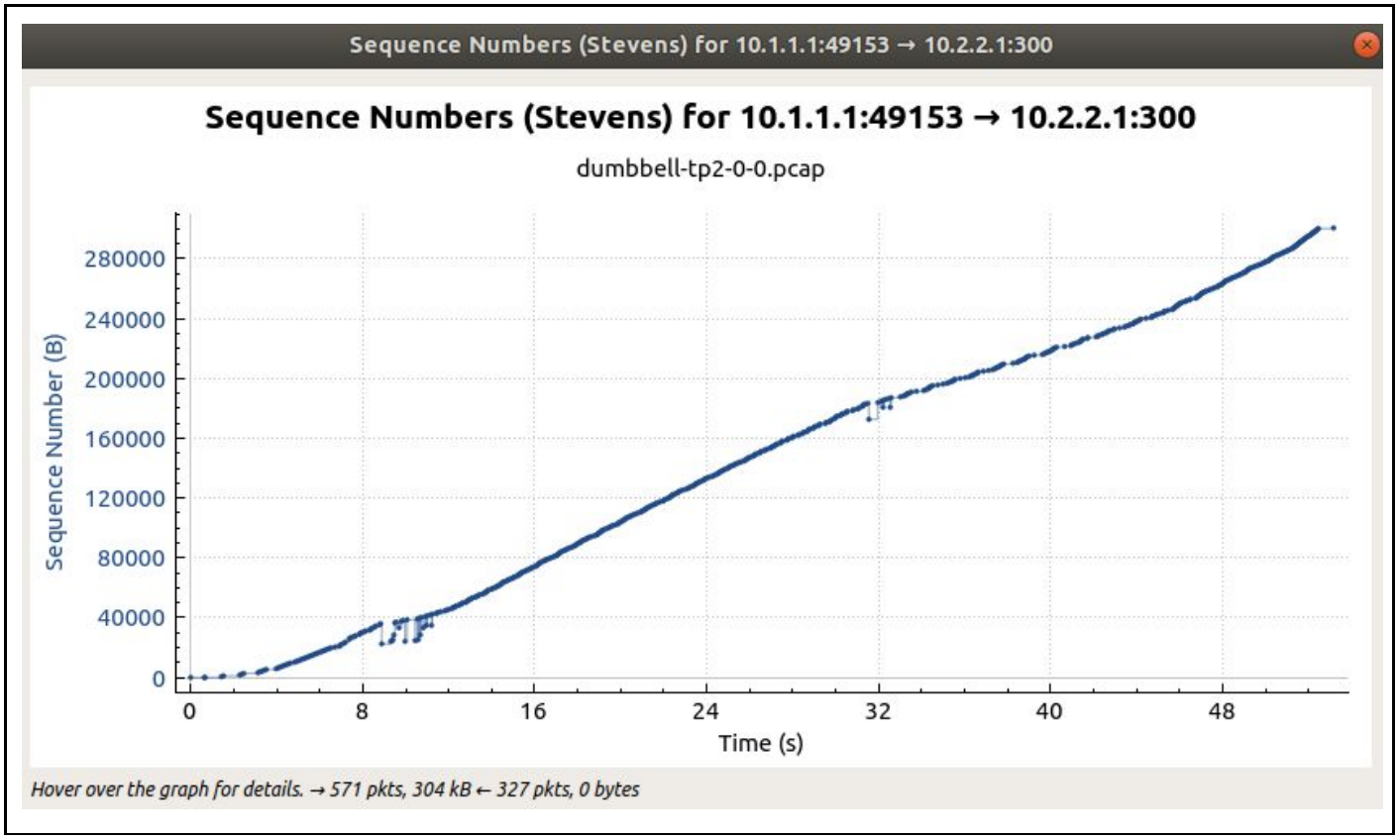

Figura 24. Traza de flujo de datos TCP NewReno durante la simulación.

Parte 3

Estándar H323

H.323 (Arquitecura de telefonía de Internet)

Resuelve el problema de establecer y terminar las llamadas por internet. H.323 "Sistemas de comunicaciones multimedia basados en paquetes" en la revisión de 1998. La recomendación H.323 fue la base para los primeros sistemas de conferencias por Internet que se hicieron muy populares. Sigue siendo la solución más implementada en la actualidad, en su séptima versión a partir de 2009.

Los pasos para establecer y finalizar las llamadas por internet son las siguientes:
_ Terminal difunde paquete UDP.
_ Guardián responde y se obtiene su IP.
_ Terminal envía un RAS solicitando ancho de banda.
_ Guardián acepta, se usa TCP con la Terminal.
_ Terminal envía mensaje SETUP con TCP y Q.931
_ El SETUP especifica n° teléfono, IP y puerto a llamar.
_ Guardian envía CALL PROCEEDING de Q.931.
_ Guardian reenvía el SETUP al IP Puerta De Enlace.
_ Puerta, realiza llamada,envía ALERT Q.931 -Terminal
_ Se envía CONNECT de Q.931 a Terminal.
_ Guardián ya no está dentro del ciclo. Sí la Puerta.
_ Con H.245 se negocia parámetros de la llamada.
_ Se establece el codec.

_ Se crean 2 canales de datos unidireccionales con el codec junto con otros parámetros a cada canal.
_ Es posible que se usen distinto codec las terminales
_ Luego de finalizar las negociaciones, comienza el flujo de datos mediante RTP y para el control de congestión es RTCP el cual sincroniza audio y video.
_ Fin de llamada, cuando alguien cuelga se usa el canal de señalización de Q.931. Se liberan recursos.
_ La terminal que inició la llamada se comunica con Guardián con mensaje RAS para liberar el ancho De Banda o realizar otra llamada.
_ QoS (calidad de servicio) queda fuera del alcance del H.323. Sin embargo, en ninguna parte de la llamada en el lado del teléfono habrá variación en el retardo, debido a que es la forma en que se diseñó la red telefónica.

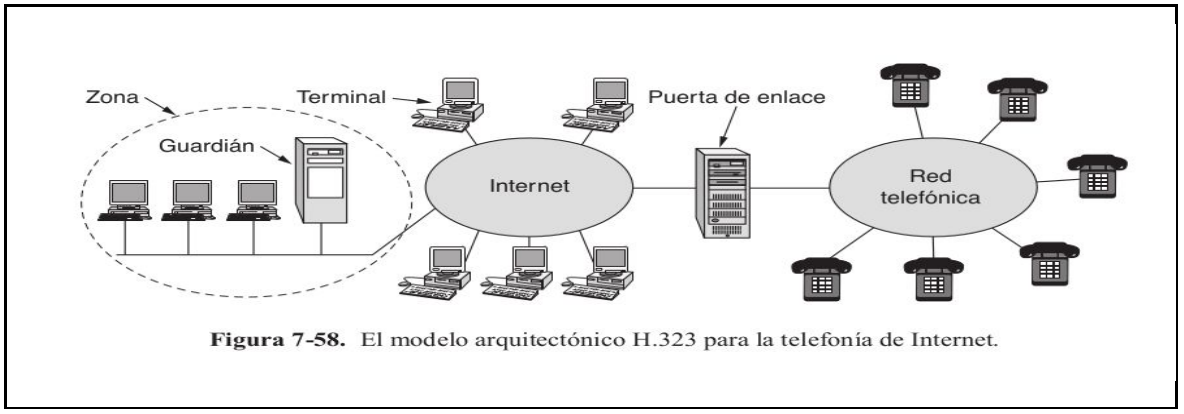

Figura 7-58. El modelo arquitectónico H.323 para la telefonía de Internet.

Figura 25. Modelo Arquitectónico H323.

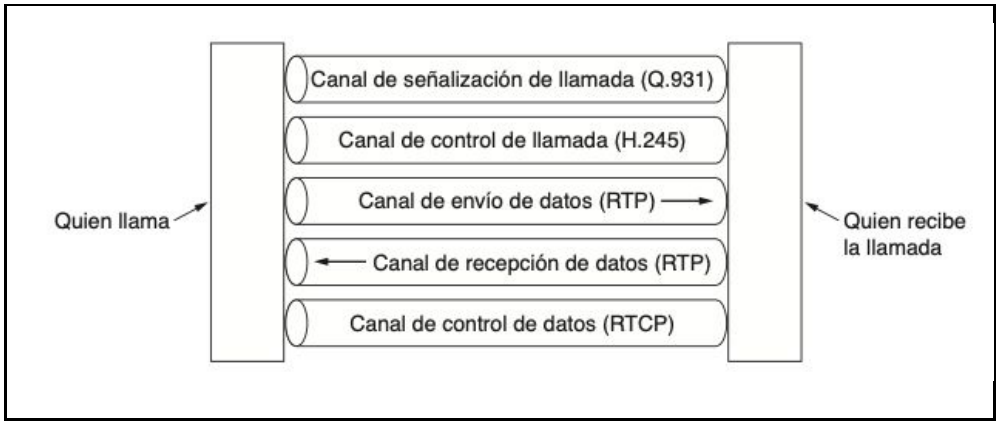

Figura 26. Canales Lógicos entre el que hace la llamada y el que la recibe durante la llamada.

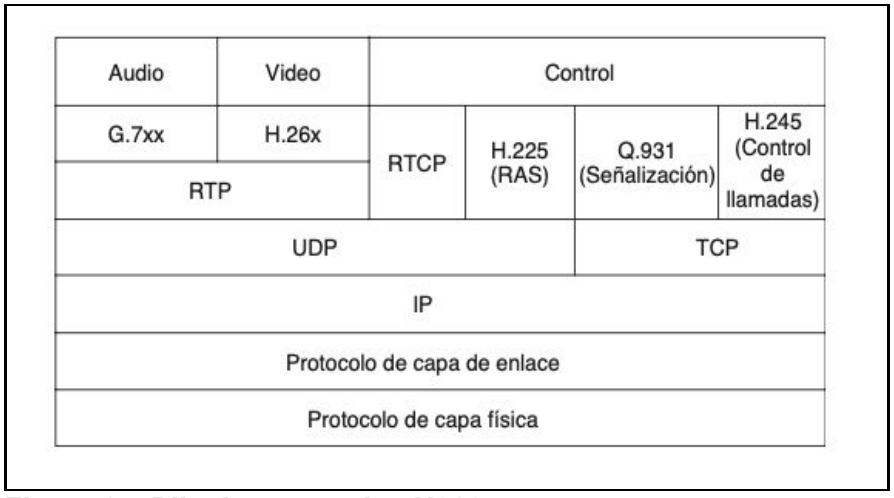

Figura 27. Pila de protocolos H323.

El ancho de banda es lo que se transfiere y en cuanto tiempo se lo transfiere. Su unidad está expresada en kbps= kb/s. O sea cantidad de Bytes por segundo.

Datos:
_ cRTP = 2 bytes
_ Header de capa 2 = 6 bytes
_ Tamaño de carga útil de voz para G.711 = 160 bytes
_ tabla:

Información de códec				Cálculos de ancho de banda					
Velocidad de bits y códec (kbps)	Ejemplo de tamaño del códec (bytes)	Ejemplo de intervalo del códec (ms)	Mean Opinion Score (MOS)	Tamaño de la carga útil de voz (bytes)	Tamaño de la carga útil de voz (ms)	Paquetes por segundo (PPS)	Ancho de banda MP o FRF.12 (Kbps)	Ancho de banda c/cRTP MP o FRF.12 (kbps)	Ancho de banda Ethernet (Kbps)
G.711 (64 Kbps)	80 bytes	10 ms	4.1	160 bytes	20 ms	50	82.8 Kbps	67.6 Kbps	87.2 Kbps
G.729 (8 Kbps)	10 bytes	10 ms	3.92	20 bytes	20 ms	50	26.8 Kbps	11.6 Kbps	31.2 Kbps

Figura 28. Parámetros de G711 y G729.

Las fórmulas de cálculo de ancho de banda por llamada son las definidas a continuación.

- **Tamaño total del paquete = (encabezado L2: MP o FRF.12 o Ethernet) + (encabezado IP/UDP/RTP) + (tamaño de carga útil de voz)**
- **PPS = (velocidad de bits en codec) / (tamaño de la carga útil de voz)**
- **Ancho de banda = tamaño de paquete total * PPS**

Aplicamos las fórmulas a los datos obtenidos:

_ Tamaño del paquete total= 6 bytes + 2 bytes + 160 bytes = 168 bytes

_ Tamaño total del paquete= (168 bytes) * (8 bits/byte) = 1344 bits

OBS: (160 bytes) * (8 bit/byte) = 1280 bits

_ PPS= (64 kbps) / (1280 bits) = 50 pps

_ **AnchoDeBandaPorLLamada**= (1344 bits) * (50 pps) = **67,2 kbps**

Llamadas simultáneas: indica la máxima cantidad de llamadas que el Cliente puede recibir o realizar en un mismo momento, se puede interpretar como máximo número de canales lógicos asignados.

Sabiendo que se usa una red con un ancho de banda máximo de 200 kbps.

67,2 kbps * cantMaxLLamadasSimultaneas <= 200 kbps
cantMaxLLamadasSimultaneas <= (200 kbps) / (67,2 kbps) = 2,976
cantMaxLLamadasSimultaneas = 2

Como calculamos anteriormente, la cantidad máxima de llamadas simultáneas para G.711 es de 2. A continuación se calculará primero el ancho de banda por llamada para G.729, luego la máxima cantidad de llamadas en simultáneo que soporta en una red de 200 kbps. Por último se comparará cuál codec permite mayor cantidad de llamadas.

Datos:
cRTP = 2 bytes
Header de capa 2 = 6 bytes
Tamaño de carga útil de voz para G.729 = 20 bytes

Aplicamos las fórmulas del 3.1 a los datos obtenidos para G.729:

_ Tamaño del paquete total= 6 bytes + 2 bytes + 20 bytes = 28 bytes

_ Tamaño total del paquete= (28 bytes) * (8 bits/byte) = 224 bits

OBS: (20 bytes) * (8 bit/byte) = 160 bits

_ PPS= (8 kbps) / (160 bits) = 50 pps

_ **AnchoDeBandaPorLLamada**= (224 bits) * (50 pps) = **11,2 kbps**

11,2 kbps * cantMaxLLamadasSimultaneas <= 200 kbps
cantMaxLLamadasSimultaneas <= (200 kbps) / (11,2 kbps) = 17,857
cantMaxLLamadasSimultaneas = 17

Podemos decir que siendo 17 las llamadas simultaneas para G.729 y solo 2 para G.711, con G.729 se realiza un mayor cantidad de llamadas en simultáneo respetando todos los datos dados.

Conclusiones

Para concluir podemos decir, que los dos protocolos simulados mediante la herramienta wireshark tiene comportamiento distintos y esperados según las definiciones de las RFC. Esta diferenciación de protocolos es debido a que tienen funcionalidades y aplicaciones distintas. En TCP es importante llevar un control de paquetes y de la red por donde se lo envía. Por el contrario, UDP no lleva ningún control.

Esto hace que sean mejor para una cosa que otra. Por ejemplo, las videoconferencias en UDP y las descargas de archivos para TCP. Se puede ver claramente la importancia de uno y del otro.

Por otro lado TCP New Reno es una variante más de la familia de TCPs. Su comportamiento no se ve reflejado en los gráficos o análisis de los pcap's (con o sin la implementación del protocolo), debido que NS3 implementa por default *TCP:newreno*. Por lo tanto, el comportamiento de las ventanas (cwnd) es la misma.

Referencias

Documentation
Conceptual Overview — ns-3 project ns-3-dev documentation
RFC 6582 - The NewReno Modification to TCP's Fast Recovery Algorithm
TCP LR-Newreno Congestion Control for IEEE 802.15.4-based Network
TCP models in ns-3 — Model Library
H.323 : Packet-based multimedia communications systems
Redes de Computadoras, 5ta Edición
Voz sobre IP – Consumo de Ancho de Banda por Llamada