



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	3
Preparación del Ambiente	4
Código Fuente	5
Etapas de Protocolo TCP	5
Desarrollo	7
Parte 1	7
Parte 2	17
TCP NewReno	17
Parte 3	23
Estándar H323	23
Conclusiones	26
Referencias	27

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.

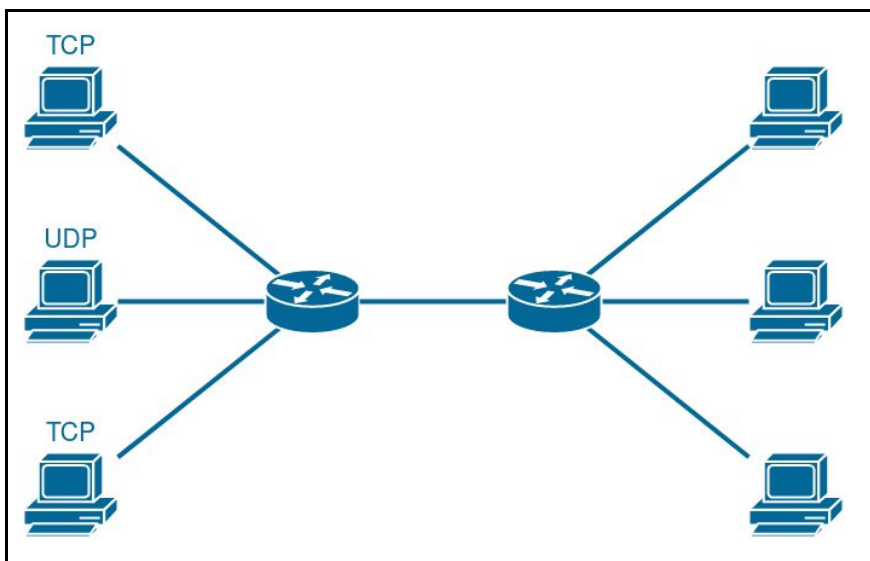


Figura 1. Topología Dumbbell.

En segundo lugar, se explicará el algoritmo *NewReno* y se implementará el algoritmo con otra red Dumbbell 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:

- Ubuntu 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 bzip2 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-dev && 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. Descomprimos 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: TP-SOR2-REDES](https://github.com/TP-SOR2-REDES)

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 provocando una comunicación indirecta hacia el emisor para realizar los ajustes pertinentes al $cwnd$ (ventana de congestión).

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 slow start y 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 entonces el cwnd se “desinfla” sin mantener el valor.

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 de 100Kb/s para todos los enlaces.

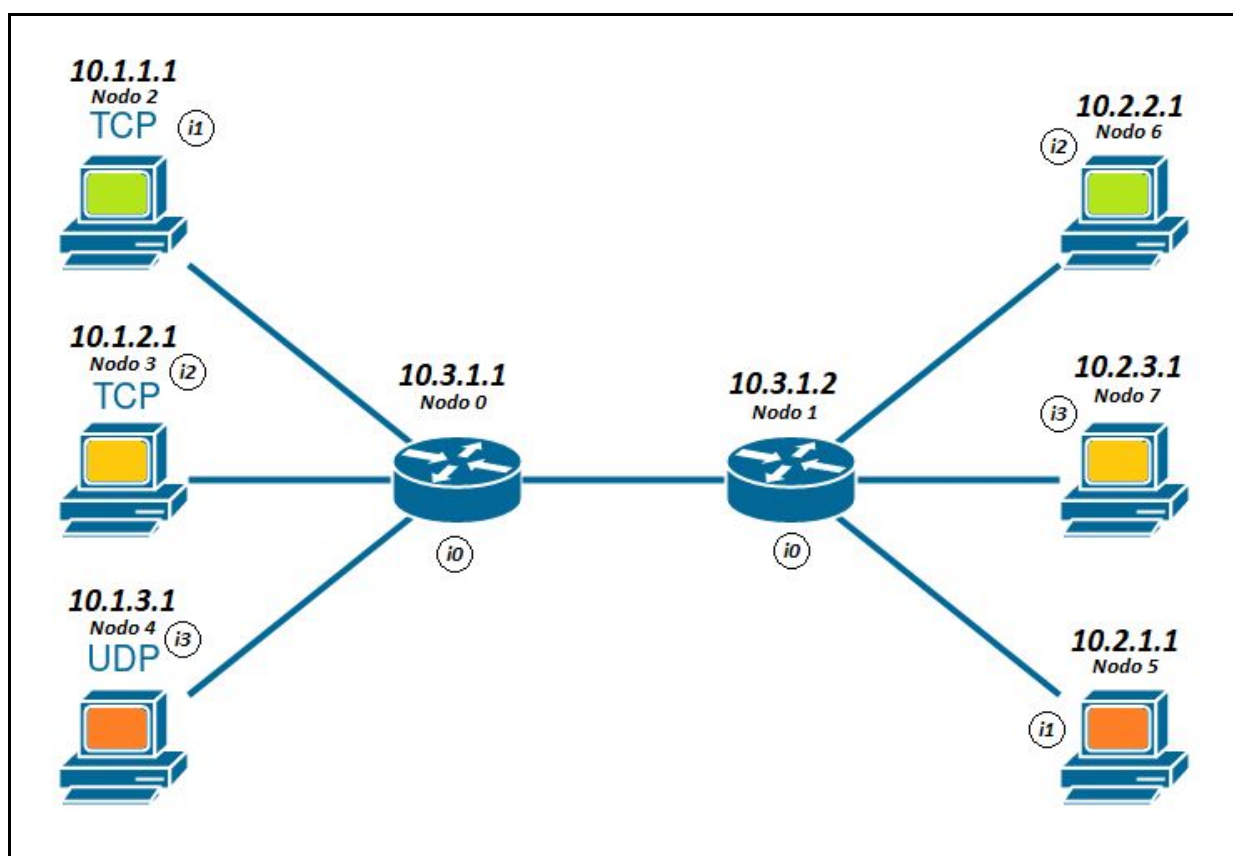


Figura 2. Topología con numeración, IPs, interfaces y tipo de protocolo.

Los archivos **.pcap** son las capturas de los paquetes que viajan a lo largo de las distintas aristas. Por ejemplo, el archivo 'dumbbell-tp2-4-0.pcap', tiene como indicadores el nodo origen (4 en este caso) y seguido la interfaz 0, por lo tanto todos los paquetes que capturará serán los UDP. Notemos que todos los nodos que solo tienen un único vecino, tienen una única interfaz, que es 0. En la Figura 2 se indica sólo las interfaces de los routers (Nodo 0 y 1). Por ejemplo "n0i0" (de ahora en más) correspondería al archivo

.pcap del nodo 0 y a la interfaz 0, es decir, del enlace entre router y router, con lo cual pasarían todos los paquetes de la simulación (TCP y UDP).

Por otro lado, el archivo **.pcap** n0i3 contendrá solo las capturas de los paquetes UDP.

En primer lugar, se realizaron pruebas con los **2 emisores TCP**, es decir, n2i0 y n3i0, para más detalle y conocer acerca de qué nodo e interfaz de la topología ver etiquetas de Figura 2. Dentro de la carpeta **/cases/tcp** del repositorio se pueden obtener los **.pcaps** de la ejecución.

Y en la Figura 3, observamos el gráfico obtenido de la librería *Gnuplot* que nos muestra cómo varía la ventana de congestión y el umbral para los nodos e interfaces mencionadas, es decir, n2i0 y n3i0.

Con lo cual, la ventana de congestión “cwnd-n3i0” y el umbral “ssthresh-n3i0” para n3i0 se visualiza en color violeta y verde respectivamente. Y para el n2i0 se visualiza en color azul “cwnd-n2i0” y en amarillo “ssthresh-n2i0”.

En caso de querer generar el gráfico, estando posicionados por terminal en la carpeta **/cases/tcp/graph/n2i0-n3i0**, ejecutar “**gnuplot config_cwnd_ssthresh.plt**”, esto combinará 4 archivos **.data** con las métricas, nodos e interfaces anteriormente mencionadas. (los colores de las líneas pueden variar en la ejecución, es necesario guiarse por el formato de las etiquetas explicadas en el párrafo anterior). Tener en cuenta, que si se genera una ejecución de la simulación y luego se intenta usar los **.data**, se debe eliminar el primer dato del umbral ya que el primer dato el algoritmo usa un número aleatorio alto y el gráfico se iría de escala.

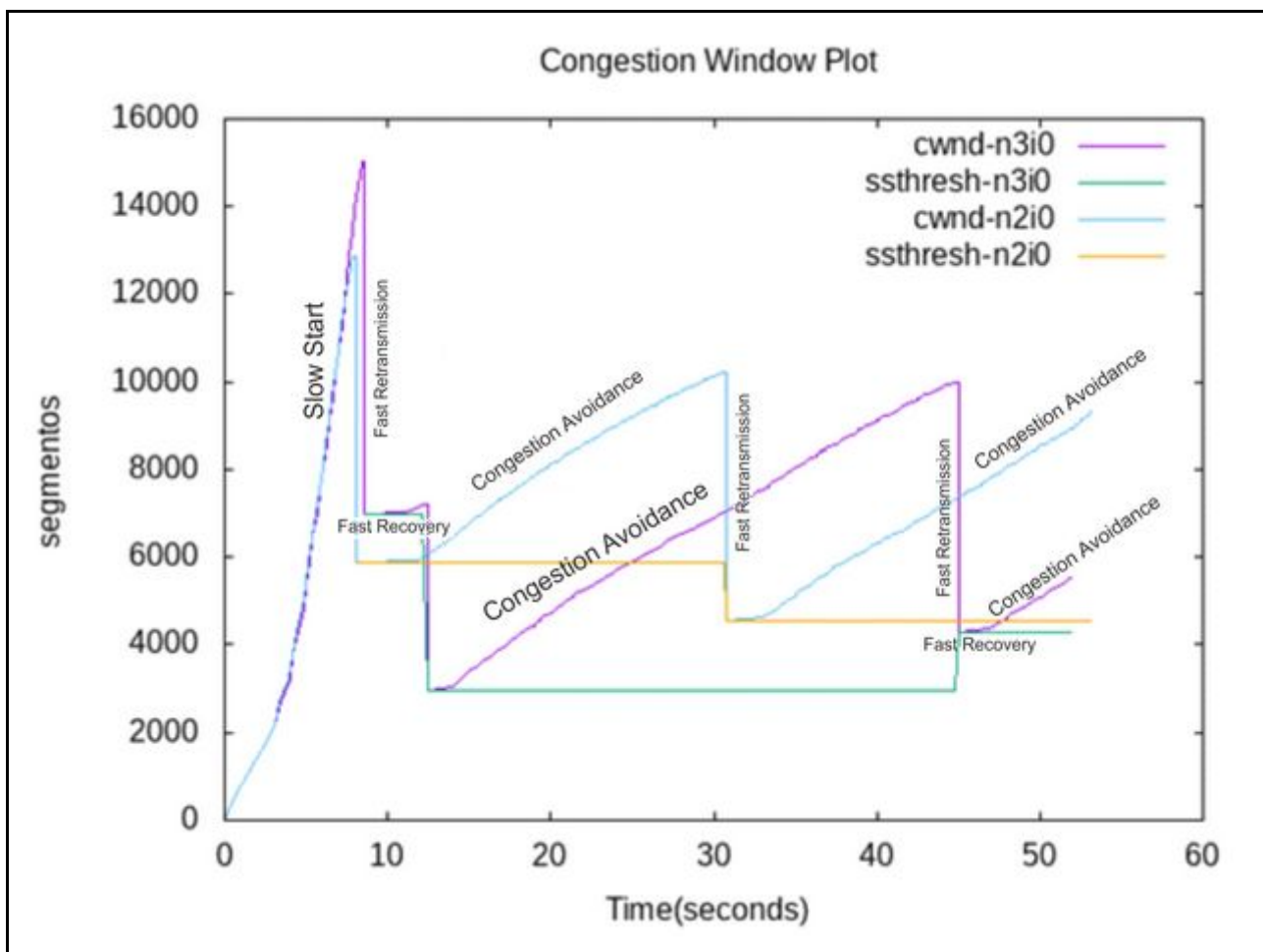


Figura 3. Ventana de congestión de emisores TCP obtenida del código C++ por gnuplot.

Ambos emisores arrancan enviando de manera exponencial (slow start) paquetes a la red, provocando luego en segundo 8, la congestión de los routers. Esto se traduce como pérdidas de paquetes que los emisores reciben como ACK duplicados y comienza la etapa de retransmisión de paquetes y de ahí a Fast Recovery ($ssthresh = cwnd/2$ - $cwnd = ssthresh + 3*MSS$). Luego en el segundo 12 cuando recibe el ACK del paquete perdido se setea $ssthresh = cwnd/2$ - $cwnd = ssthresh + 3*MSS$ y se retoma Congestion avoidance (crecimiento lineal), hasta que nuevamente hay pérdidas de paquetes y entra a Fast Retransmit.

Ahora, si analizamos los bits de transferencia en ese momento, se puede notar una reducción. En las Figuras 4 y 5, es decir, vemos los bits de transferencia correspondientes a los emisores TCP a los 8 segundos de la simulación. Para poder replicar los gráficos de las Figuras 4 y 5 tomar los .pcaps (dumbbell-tp2-3-0 y dumbbell-tp2-2-0) de la ruta /re-ejecuciones/tcp - 100kb-s. Asegurarse de configurar la escala a 12000 Bytes/seg.

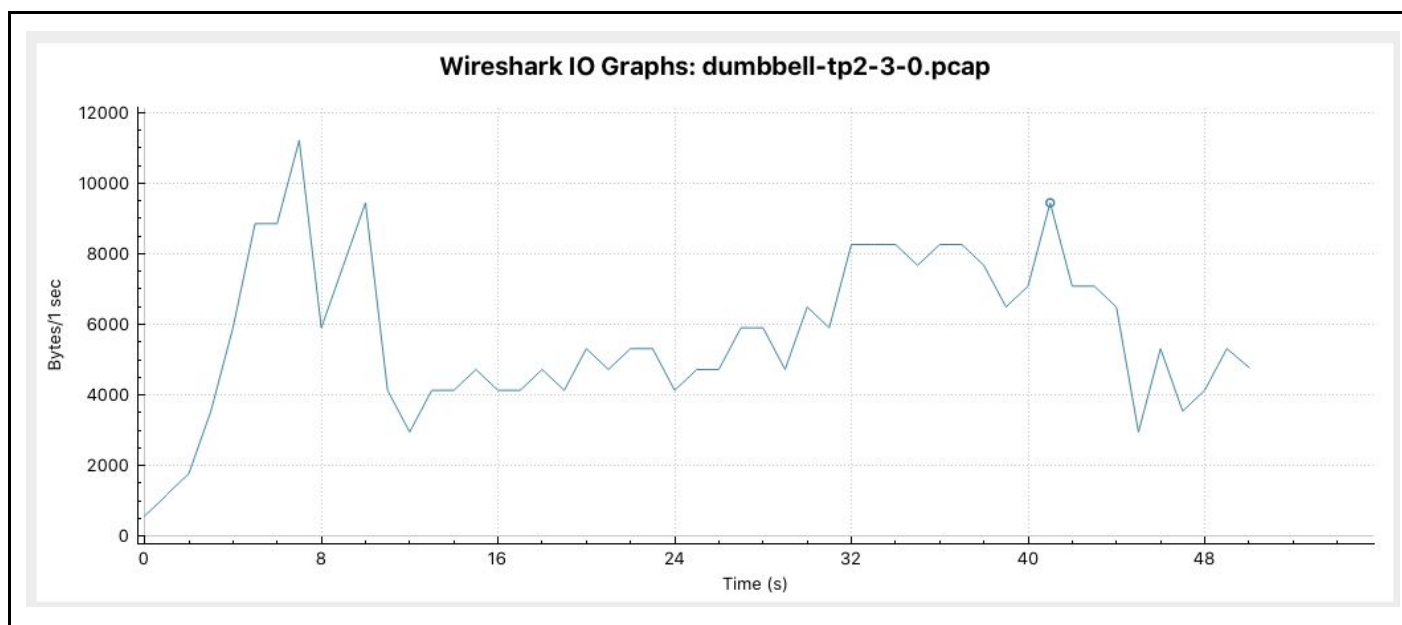


Figura 4. Bits de transferencia del nodo 3 con protocolo TCP.(/re-ejecuciones/tcp - 100kb-s)

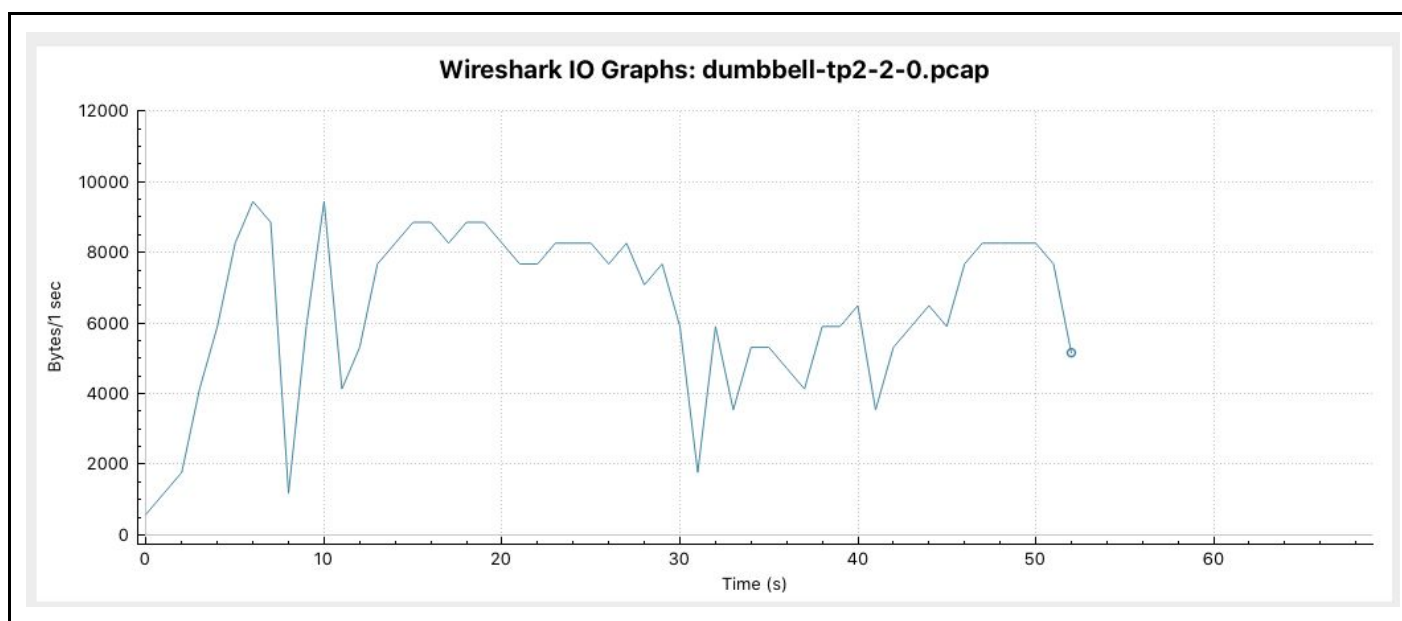


Figura 5. Bits de transferencia del nodo 2 con protocolo TCP. (/re-ejecuciones/tcp - 100kb-s)

Ahora vamos a calcular el ancho de banda. Observamos los gráficos de la opción “Statistics” del menú, más precisamente utilizamos el tipo de gráfica **Flow Graph**. Luego, para medir la velocidad de transferencia tomamos 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 7), que es de 536 bytes más el header nos da un total de 590 bytes.

Si la cantidad de paquetes es igual a 3, luego, obtenemos los bytes transferidos en ese tiempo, que es 590 bytes * 3 = 1770 bytes. La división del total de bytes por el tiempo calculado nos proporciona una aproximación.

Luego, la velocidad de transferencia en el momento analizado es de 72,50 kbit/s.

$$\text{Velocidad} = 1770 \text{ Bytes} / (7,851839 \text{ s} - 7.661119 \text{ s}) = 9280,62 \text{ Bytes/S} = 74245 \text{ bit} / \text{s} = 72,50 \text{ kbit/s}$$



Figura 6. Flow graph del momento en el cual se congestiona la red, visto desde el enlace n2i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp - 100kb-s. Y se visualizan los ACK previos para el cálculo de la velocidad.

No.	Time	Source	Destination	Protocol	Length	Info
84	7.472319	10.1.1.1	10.2.2.1	TCP	590	49153 → 300 [AC
85	7.519519	10.1.1.1	10.2.2.1	TCP	590	49153 → 300 [AC
86	7.661119	10.2.2.1	10.1.1.1	TCP	54	300 → 49153 [AC
87	7.661119	10.1.1.1	10.2.2.1	TCP	590	49153 → 300 [AC
88	7.708319	10.1.1.1	10.2.2.1	TCP	590	49153 → 300 [AC
89	7.755519	10.1.1.1	10.2.2.1	TCP	590	49153 → 300 [AC
90	7.851839	10.2.2.1	10.1.1.1	TCP	62	300 → 49153 [AC
91	7.851839	10.1.1.1	10.2.2.1	TCP	590	49153 → 300 [AC
92	7.899039	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 90

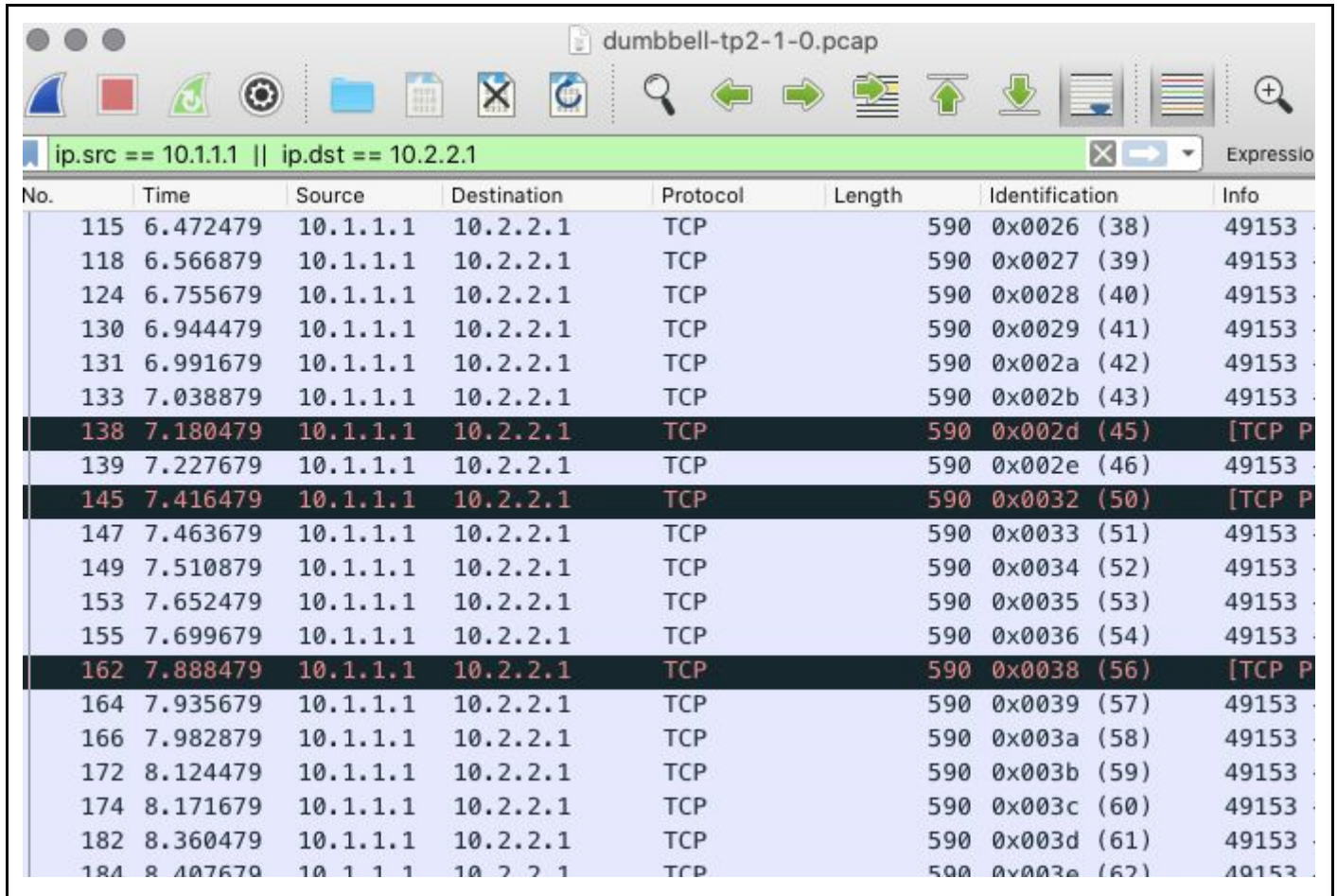
▶ Frame 87: 590 bytes on wire (4720 bits), 590 bytes captured (4720 bits)
 ▶ Point-to-Point Protocol
 ▶ Internet Protocol Version 4, Src: 10.1.1.1, Dst: 10.2.2.1
 ▶ Transmission Control Protocol, Src Port: 49153, Dst Port: 300, Seq: 32697, Ack: 1, Len: 536
 ▶ Data (536 bytes)

Figura 7. Cantidad de bytes de datos de cada paquete, visto desde el enlace n2i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp - 100kb-s. 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, /re-ejecuciones/tcp - 100kb-s/dumbbell-tp2-1-0.pcap):

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

Con lo cual podemos ver los paquetes relacionados (ver Figura 8).



No.	Time	Source	Destination	Protocol	Length	Identification	Info
115	6.472479	10.1.1.1	10.2.2.1	TCP	590	0x0026 (38)	49153
118	6.566879	10.1.1.1	10.2.2.1	TCP	590	0x0027 (39)	49153
124	6.755679	10.1.1.1	10.2.2.1	TCP	590	0x0028 (40)	49153
130	6.944479	10.1.1.1	10.2.2.1	TCP	590	0x0029 (41)	49153
131	6.991679	10.1.1.1	10.2.2.1	TCP	590	0x002a (42)	49153
133	7.038879	10.1.1.1	10.2.2.1	TCP	590	0x002b (43)	49153
138	7.180479	10.1.1.1	10.2.2.1	TCP	590	0x002d (45)	[TCP P
139	7.227679	10.1.1.1	10.2.2.1	TCP	590	0x002e (46)	49153
145	7.416479	10.1.1.1	10.2.2.1	TCP	590	0x0032 (50)	[TCP P
147	7.463679	10.1.1.1	10.2.2.1	TCP	590	0x0033 (51)	49153
149	7.510879	10.1.1.1	10.2.2.1	TCP	590	0x0034 (52)	49153
153	7.652479	10.1.1.1	10.2.2.1	TCP	590	0x0035 (53)	49153
155	7.699679	10.1.1.1	10.2.2.1	TCP	590	0x0036 (54)	49153
162	7.888479	10.1.1.1	10.2.2.1	TCP	590	0x0038 (56)	[TCP P
164	7.935679	10.1.1.1	10.2.2.1	TCP	590	0x0039 (57)	49153
166	7.982879	10.1.1.1	10.2.2.1	TCP	590	0x003a (58)	49153
172	8.124479	10.1.1.1	10.2.2.1	TCP	590	0x003b (59)	49153
174	8.171679	10.1.1.1	10.2.2.1	TCP	590	0x003c (60)	49153
182	8.360479	10.1.1.1	10.2.2.1	TCP	590	0x003d (61)	49153
184	8.407679	10.1.1.1	10.2.2.1	TCP	590	0x003e (62)	49153

Figura 8. Momento en que se detectan la pérdida de los primeros paquetes. Visto desde enlace n1i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp - 100kb-s.

Detección de pérdida de paquetes, son los paquetes que provienen del enlace **n2i0** de tipo emisor al enlace **n6i0** de tipo receptor. Se observa que no le llega al **n6i0** los paquetes nro 44, 47, 48, 49, 55, 64 y 67 . Estos números son identificados en el campo 'identification' de la Figura 8, y se puede observar su pérdida debido a la ausencia del mismo. Esto provocará los mensajes de ACK duplicados hacia el emisor **n2i0**, lo que significa que cuando llegue el tercer ACK duplicado provoca que el nodo 2 entre en Fast Retransmit. El campo "identification" proviene del header de IPV4, es un campo que se utiliza principalmente para identificar de forma unívoca al grupo de fragmentos de un datagrama de IP única.

Lo interesante de este caso es que no sucedieron 3 ACK duplicado tal como lo indica la teoría, sino que con tan sólo 2 ACK duplicados se observó una retransmisión. El caso lo pueden ver en **/re-ejecuciones/tcp - 100kb-s**. Aún no logramos entender cuál fue el motivo pero pudimos notar que todos los enlaces estaban configurados a 100kb/s. El archivo con el código es **dumbbell-tp2-tcp.cc** y el commit es: **75c07110ab81681a4a430964fe56de81d1f8536a**.

A continuación, ya que no pudimos encontrar el caso en el escenario anterior (ver código commit), vemos con el fin de ejemplificar la teoría estudiada la ejecución en donde efectivamente se observan 3 ACK duplicados con la ejecución de un nuevo caso muy similar de congestión ubicado en **/re-ejecuciones/tcp**. En este caso, la idea era dejar los enlaces a 100kb/s y forzar un cuello de botella en el enlace entre los routers a 50kb/s.



Figura 9. 3 ACK y Retransmisión de paquetes. Visto desde el enlace n2i0 del .pcap correspondiente dentro de **/re-ejecuciones/tcp**.

Podemos observar en la figura 9 que surgen ACKs duplicados y a los 9,44 segundos se pasa al mecanismo **“Fast Retransmit”**.

79	8.916159	10.1.1.1	10.2.2.1	TCP	590	0x0039 (57)	49153 → 300 [ACK] Seq=29481 Ack=1 Win=131072 Len=536 TSval=8868 TSecr
80	8.963359	10.1.1.1	10.2.2.1	TCP	590	0x003a (58)	49153 → 300 [ACK] Seq=30017 Ack=1 Win=131072 Len=536 TSval=8868 TSecr
81	9.249119	10.2.2.1	10.1.1.1	TCP	62	0x0015 (21)	[TCP Dup ACK 77#1] 300 → 49153 [ACK] Seq=1 Ack=19297 Win=131072 Len=6
82	9.249119	10.1.1.1	10.2.2.1	TCP	590	0x003b (59)	49153 → 300 [ACK] Seq=30553 Ack=1 Win=131072 Len=536 TSval=9249 TSecr
83	9.343519	10.2.2.1	10.1.1.1	TCP	62	0x0016 (22)	[TCP Dup ACK 77#2] 300 → 49153 [ACK] Seq=1 Ack=19297 Win=131072 Len=6
84	9.343519	10.1.1.1	10.2.2.1	TCP	590	0x003c (60)	49153 → 300 [ACK] Seq=31089 Ack=1 Win=131072 Len=536 TSval=9343 TSecr
85	9.440479	10.2.2.1	10.1.1.1	TCP	70	0x0017 (23)	[TCP Dup ACK 77#3] 300 → 49153 [ACK] Seq=1 Ack=19297 Win=131072 Len=6
86	9.440479	10.1.1.1	10.2.2.1	TCP	590	0x003d (61)	[TCP Fast Retransmission] 49153 → 300 [ACK] Seq=19297 Ack=1 Win=13107
87	9.723679	10.2.2.1	10.1.1.1	TCP	70	0x0018 (24)	[TCP Dup ACK 77#4] 300 → 49153 [ACK] Seq=1 Ack=19297 Win=131072 Len=6
88	9.818079	10.2.2.1	10.1.1.1	TCP	70	0x0019 (25)	[TCP Dup ACK 77#5] 300 → 49153 [ACK] Seq=1 Ack=19297 Win=131072 Len=6

Figura 10. Ventanas de recepción en el momento de congestión de la red (ver variable “Win”). Visto desde el enlace n2i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp.

En la figura 10 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.

Ahora, volvemos al primer caso de la congestión para seguir analizándolo (100 kb/s).

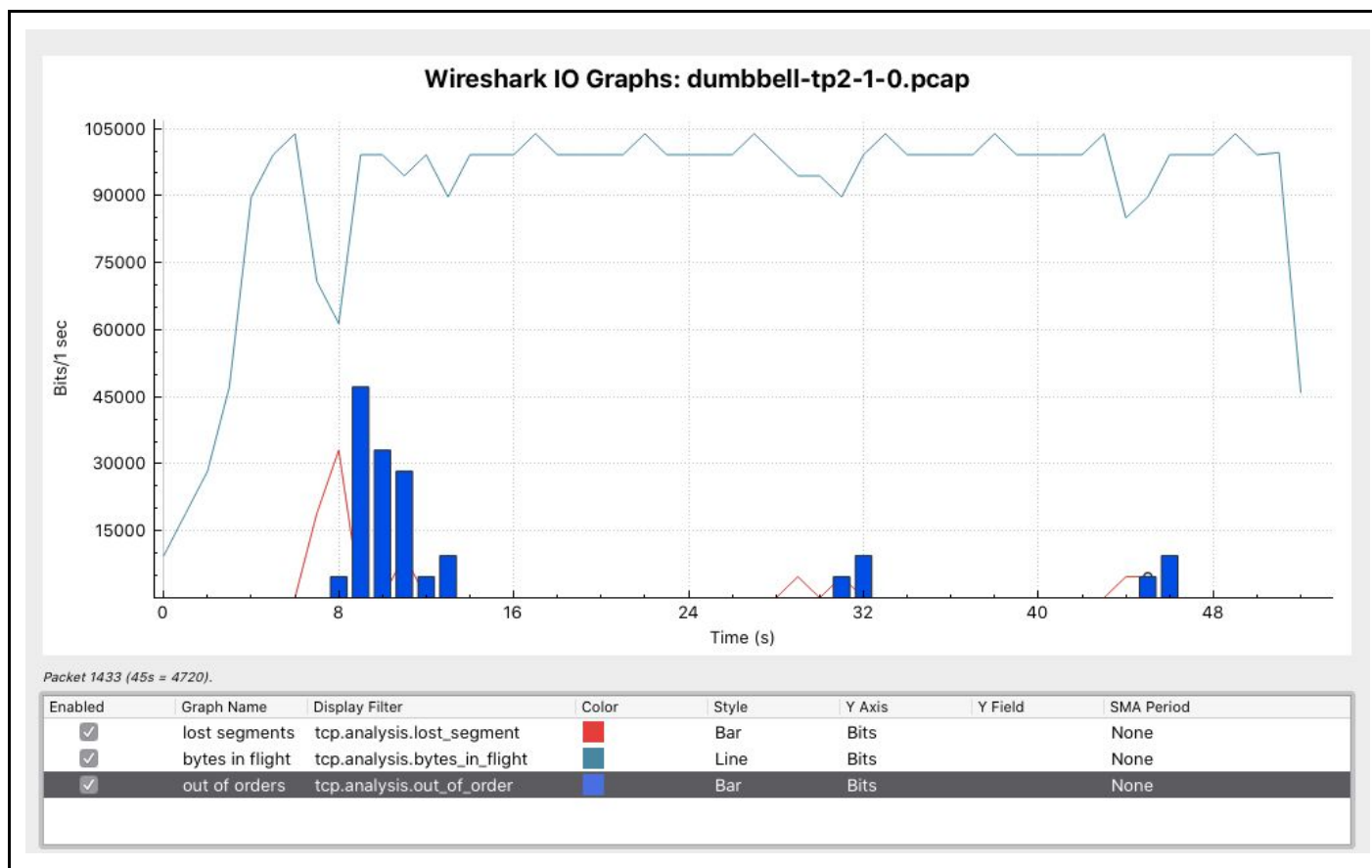


Figura 11. Archivo .pcap correspondiente al enlace n1i0 (nodo 1 e interfaz 0).Visto desde el .pcap correspondiente dentro de /re-ejecuciones/tcp - 100kb-s.

En la Figura 11 podemos ver el mayor pico de pérdida de paquetes en el momento de la congestión seguido de la detección de “paquetes fuera de orden”.

Ahora la topología cuenta con otro protocolo para la transferencia de paquetes, UDP. Cabe aclarar que este protocolo tiene características opuestas a TCP.

UDP no lleva ningún control y en cambio TCP cada vez que pierde un paquete debería bajar su ventana.

El motivo por el cual UDP baja es debido a que la ventana de TCP va creciendo (la cwnd) y se detiene cuando le llega el 3 ACK duplicado, ya que el algoritmo no ve nada más hasta actualizar el tamaño de ventana. Luego cuando la ventana de tcp baja (cwnd), UDP aprovecha para ocupar más ancho de banda. Esto implica que UDP monopolice el ancho de banda.

Además, UDP 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.

En la Figura 12, podemos ver que no ocurre debido a que se configuraron las colas de los routers con un tamaño máximo de sólo 10 paquetes.

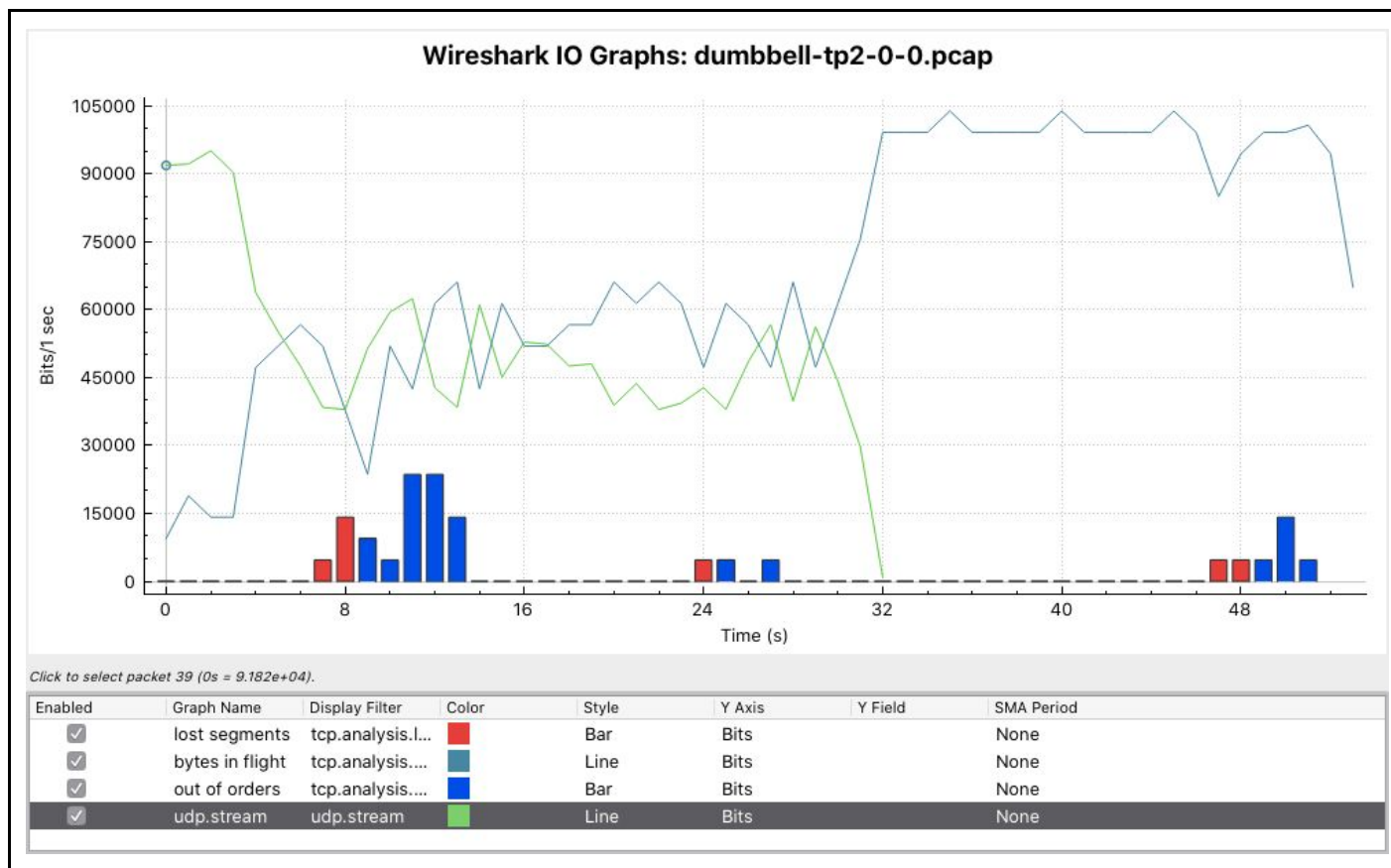


Figura 12. Comparación entre el flujo de datos UDP y TCP. Visto desde el enlace n0i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp-udp.

Luego, se producen pérdidas de paquetes al ser tan pequeño el tamaño de la cola y por eso se observa que UDP comienza en 90000 y termina en 45000. Si la cola del router fuese configurado con un valor mayor, se vería gráficamente de que UDP no bajaría tan rápidamente su ancho de banda.

Los paquetes perdidos de UDP no se recuperan por característica del protocolo, pero en el caso de TCP sí. Por ejemplo, TCP puede demorar más en transmitir toda la información porque puede que tenga que re-transmitir, en cambio UDP no lleva un control de paquetes recibidos.

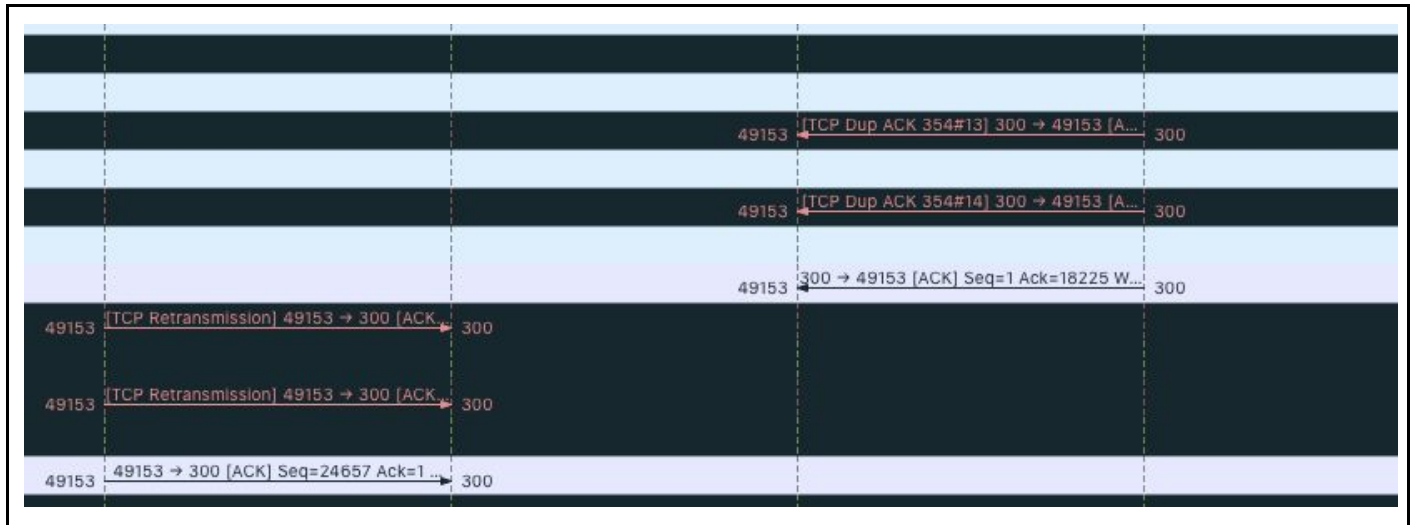


Figura 13. ACK duplicados y retransmisión en simulación con nodos TCP y UDP. Visto desde el enlace n0i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp-udp. Gráfico Flow Graph, a partir de 10,14 seg.

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 SSTHRESH.

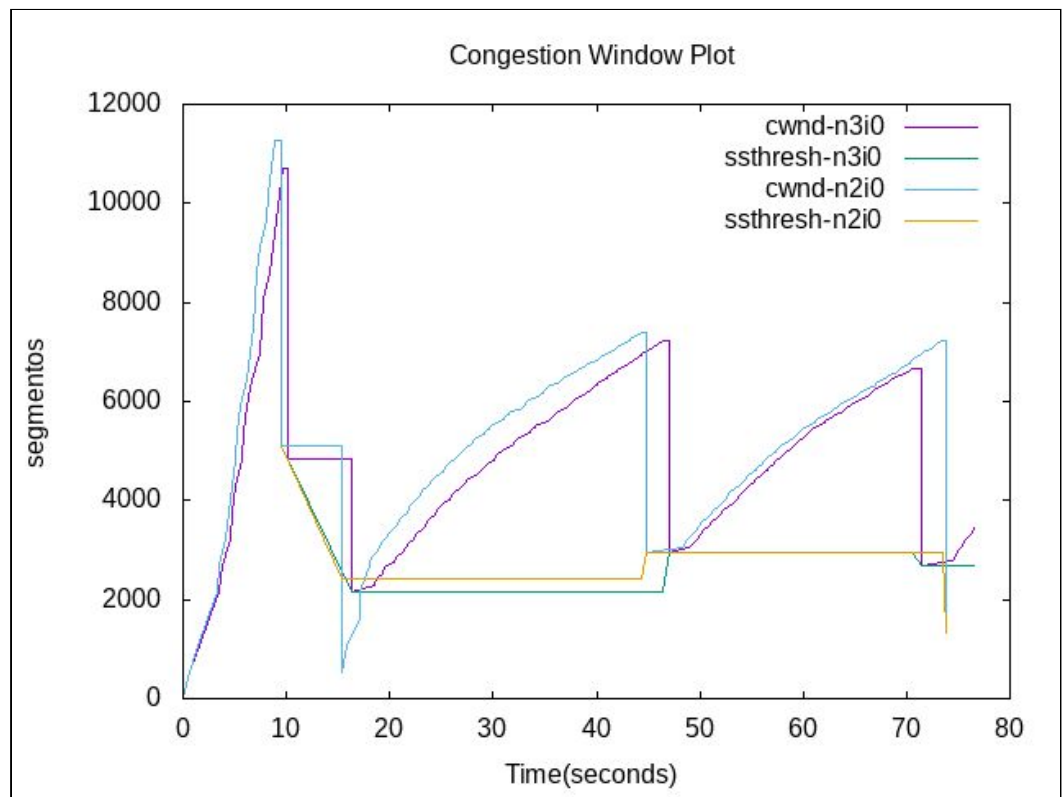


Figura 14. Ventana de congestión para los enlaces emisores/router.

Ambos emisores producen el ajuste de su cwnd al mismo instante en tiempos 11, 45 o 70. Esto es debido a que en el tiempo 0, ambos comienzan con el envío de sus paquetes. Inician con el algoritmo Slow start provocando la congestión rápida sobre el router, y posteriormente cada emisor hará su ajuste

correspondiente según el algoritmo TCP (newReno/default). Como en la simulación ambos implementan el mismo protocolo (TCP), los ajustes son los mismo, provocando que en esos instantes mencionados se apliquen los mecanismos necesarios por la pérdida de paquetes durante la saturación de los routers.

Parte 2

TCP NewReno

TCP Reno proponía como máximo una retransmisión por cada RTT, sin embargo, no tiene en cuenta casos en los que se producen múltiples pérdidas de paquetes en una sola ventana, conocido como, 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 15** 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 (2^n) 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.

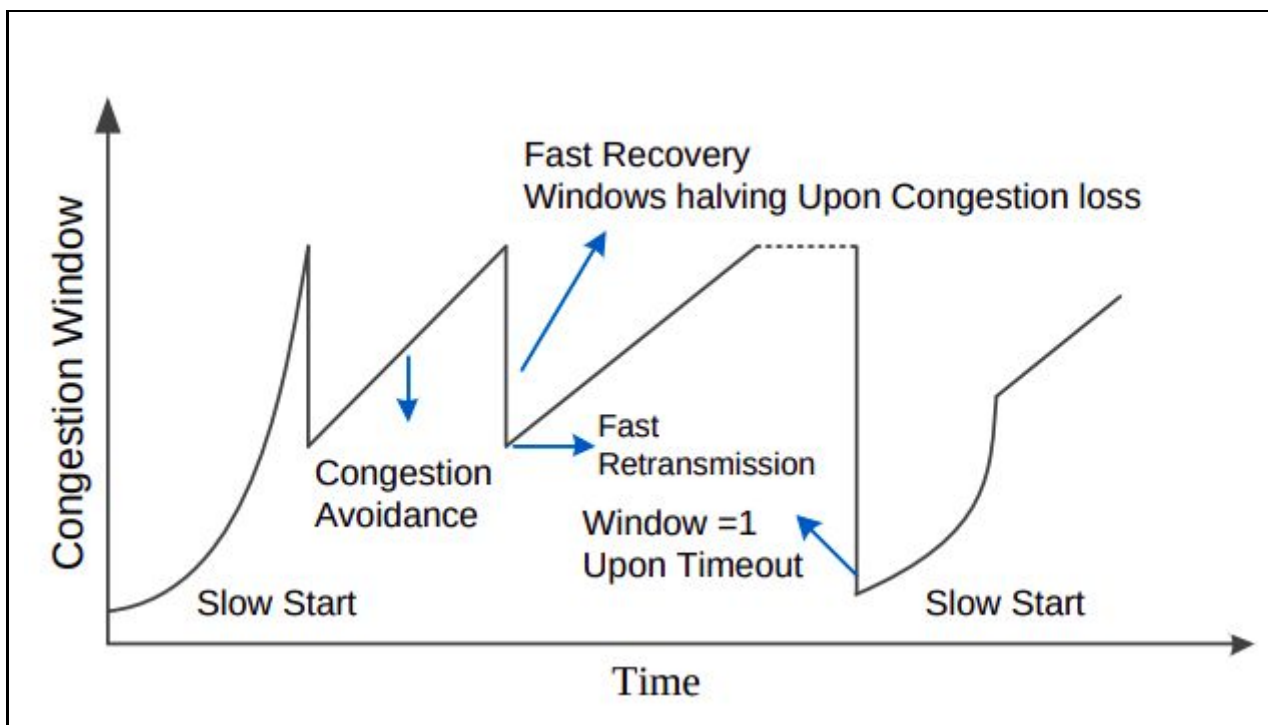


Figura 15. Fases del algoritmo NewReno.

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%.

Una vulnerabilidad de Reno se manifiesta cuando ocurre una **pérdida de múltiples segmentos** en una misma ventana. Dada una pérdida de segmento se entrará en el modo de recuperación rápida reduciendo la ventana de congestión a la mitad, pero como fueron múltiples pérdidas de segmentos se recibirá un reconocimiento parcial de los segmentos. Por lo tanto, se sale de la recuperación rápida y se vuelve a la retransmisión rápida de otro segmento perdido reduciendo otra vez la ventana a la mitad al entrar a la recuperación rápida y así sucesivamente con todos los segmentos perdidos. La ejecución de Reno se alterna entre las fases de evitación de la congestión y la recuperación rápida. Por lo tanto, se provocan múltiples reducciones de la ventana de congestión y del umbral hasta no permitir ACKs duplicados, siendo la expiración del tiempo de espera el único mecanismo posible. Esto provoca una importante disminución en el desempeño (throughput) de TCP.

NewReno a diferencia de Reno, cuando se reciben tres ACKs duplicados no sale de la fase de recuperación rápida hasta recibir los ACKs de todos los segmentos pendientes de la misma ventana hasta el momento de entrar a dicha fase. Para esto, NewReno utiliza una variable adicional donde guarda el número de secuencia del último segmento mencionado.

De esta forma NewReno resuelve el problema de múltiples pérdidas en la ventana de congestión. Por otro lado, también genera un tiempo excesivo en el tiempo de recuperación, ya que demora un RTT en detectar que se perdió.

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 16. 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 17. 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 18. Momento en que se detectan la pérdida de los primeros paquetes del nodo 2 al nodo 6. Visto desde enlace n1i0 del .pcap correspondiente dentro de /re-ejecuciones/tcp - 100kb-s.

En la Figura 19 se pueden ver las diferentes etapas del algoritmo NewReno correspondientes a n2i0 y n3i0. Se puede apreciar la similitud con la Figura 3 debido a que actualmente se utiliza el algoritmo por default en ns3.

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.

Para replicar el gráfico, ir a **cases/tcp-newreno/graph** y combinar mediante un archivo .plt los archivos .data correspondientes a los nodos e interfaces mencionadas ubicados en las carpetas n2i0 y n3i0.

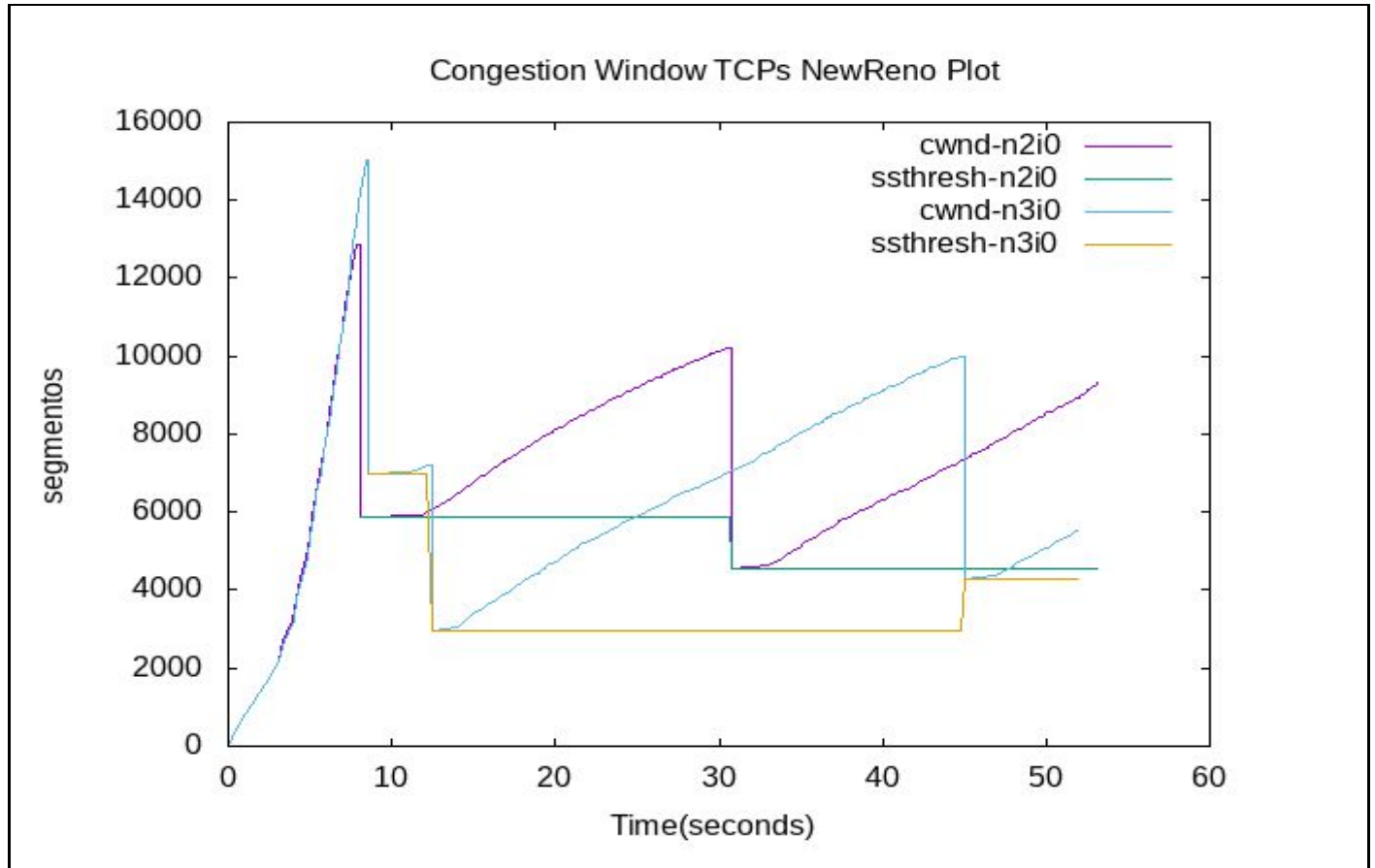


Figura 19. Ventana de congestión de los enlaces entre emisores y routers.

Luego, mirando el pcap n2i0, se puede observar la característica que diferencia NewReno de Reno. En el segundo 8.089 observamos un “**Fast retransmission**”, podemos ver varias retransmisiones (ver Figura 20).

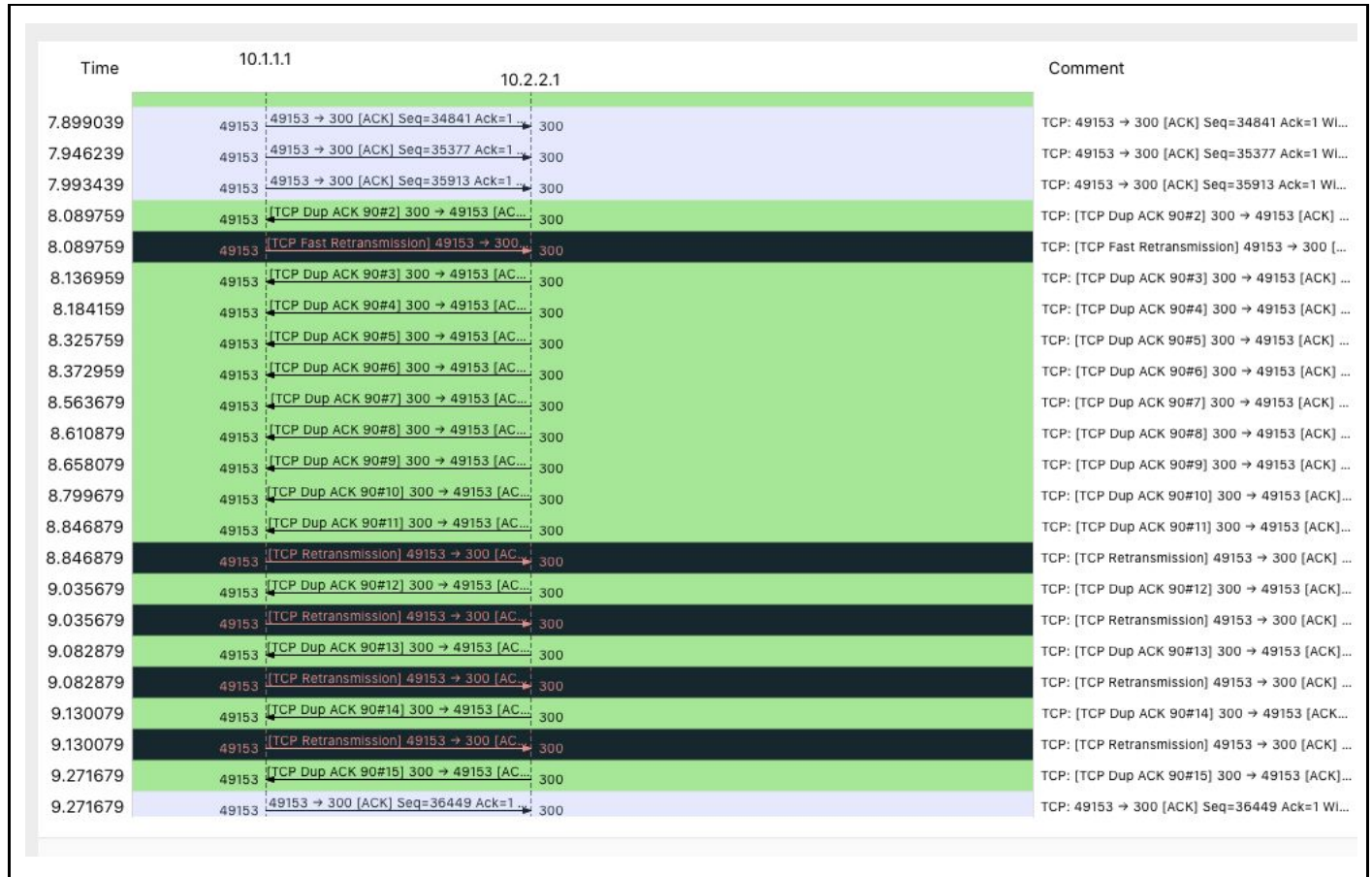


Figura 20. TCP NewReno: Fast Retransmission y varias retransmisiones en un RTT.

En la Figura 21, se puede apreciar los números de secuencias de los distintos paquetes. Las retransmisiones que produce NewReno las podemos notar en cada bajada del número de secuencia. Por ejemplo, en el segundo 8,08 se puede ver el inicio de *Fast Retransmission* del paquete con número de secuencia 22513 (ver Figura 22).

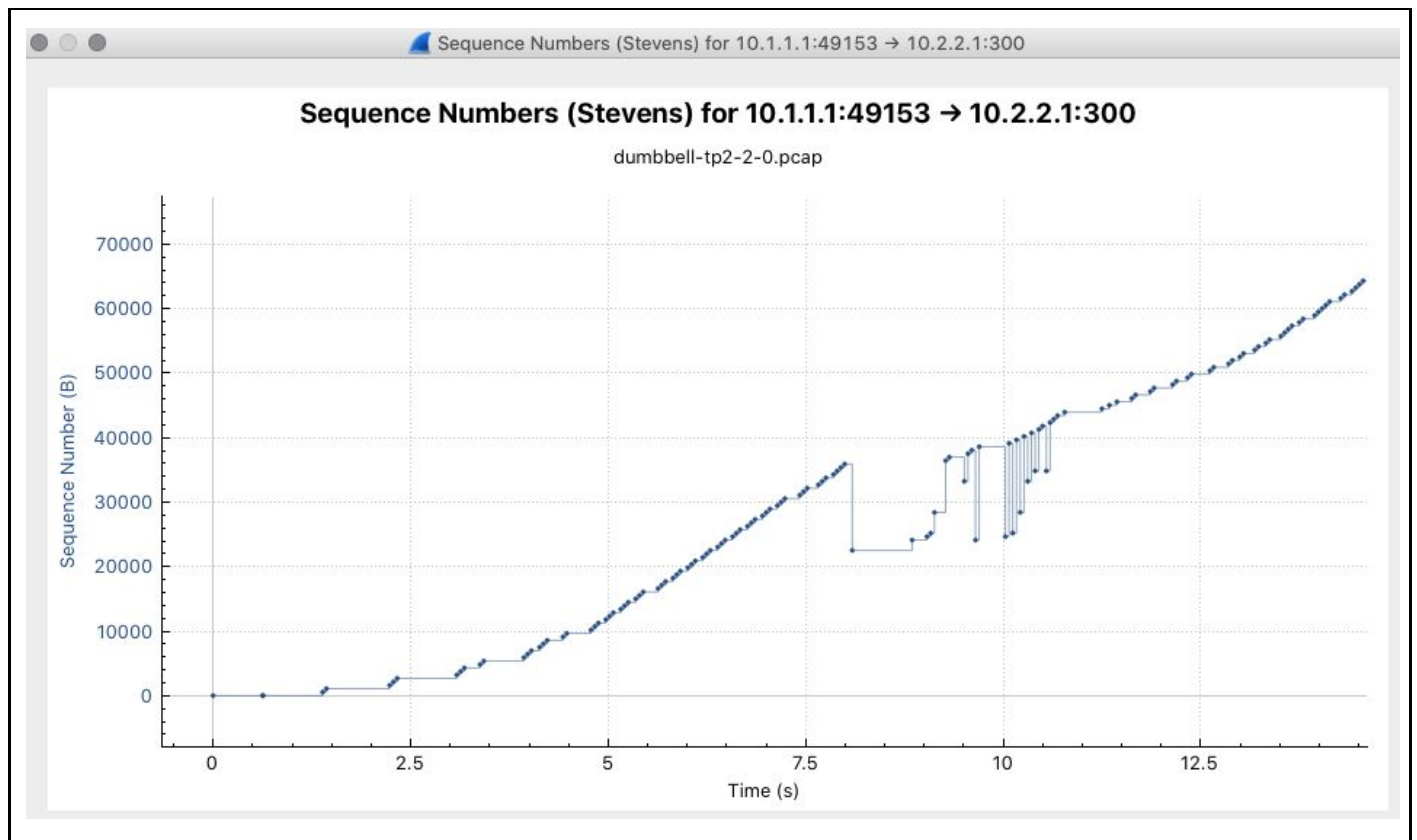


Figura 21. Traza de flujo de datos TCP NewReno durante la simulación.

No.	Time	Source	Destination	Protocol	Length	Identification	Info
84	7.472319	10.1.1.1	10.2.2.1	TCP	590	0x003d (61)	49153 → 300 [ACK] Seq=31625 Ack=1 Win=131072 Len=536
85	7.519519	10.1.1.1	10.2.2.1	TCP	590	0x003e (62)	49153 → 300 [ACK] Seq=32161 Ack=1 Win=131072 Len=536
87	7.661119	10.1.1.1	10.2.2.1	TCP	590	0x003f (63)	49153 → 300 [ACK] Seq=32697 Ack=1 Win=131072 Len=536
88	7.708319	10.1.1.1	10.2.2.1	TCP	590	0x0040 (64)	49153 → 300 [ACK] Seq=33233 Ack=1 Win=131072 Len=536
89	7.755519	10.1.1.1	10.2.2.1	TCP	590	0x0041 (65)	49153 → 300 [ACK] Seq=33769 Ack=1 Win=131072 Len=536
91	7.851839	10.1.1.1	10.2.2.1	TCP	590	0x0042 (66)	49153 → 300 [ACK] Seq=34305 Ack=1 Win=131072 Len=536
93	7.899039	10.1.1.1	10.2.2.1	TCP	590	0x0043 (67)	49153 → 300 [ACK] Seq=34841 Ack=1 Win=131072 Len=536
94	7.946239	10.1.1.1	10.2.2.1	TCP	590	0x0044 (68)	49153 → 300 [ACK] Seq=35377 Ack=1 Win=131072 Len=536
95	7.993439	10.1.1.1	10.2.2.1	TCP	590	0x0045 (69)	49153 → 300 [ACK] Seq=35913 Ack=1 Win=131072 Len=536
97	8.089759	10.1.1.1	10.2.2.1	TCP	590	0x0046 (70)	[TCP Fast Retransmission] 49153 → 300 [ACK] Seq=22513
107	8.846879	10.1.1.1	10.2.2.1	TCP	590	0x0047 (71)	[TCP Retransmission] 49153 → 300 [ACK] Seq=24121 Ack=
109	9.035679	10.1.1.1	10.2.2.1	TCP	590	0x0048 (72)	[TCP Retransmission] 49153 → 300 [ACK] Seq=24657 Ack=
111	9.082879	10.1.1.1	10.2.2.1	TCP	590	0x0049 (73)	[TCP Retransmission] 49153 → 300 [ACK] Seq=25193 Ack=
113	9.130079	10.1.1.1	10.2.2.1	TCP	590	0x004a (74)	[TCP Retransmission] 49153 → 300 [ACK] Seq=28409 Ack=
115	9.271679	10.1.1.1	10.2.2.1	TCP	590	0x004b (75)	49153 → 300 [ACK] Seq=36449 Ack=1 Win=131072 Len=536
117	9.318879	10.1.1.1	10.2.2.1	TCP	590	0x004c (76)	49153 → 300 [ACK] Seq=36985 Ack=1 Win=131072 Len=536
119	9.507679	10.1.1.1	10.2.2.1	TCP	590	0x004d (77)	[TCP Retransmission] 49153 → 300 [ACK] Seq=33233 Ack=
121	9.554879	10.1.1.1	10.2.2.1	TCP	590	0x004e (78)	49153 → 300 [ACK] Seq=37521 Ack=1 Win=131072 Len=536
123	9.602079	10.1.1.1	10.2.2.1	TCP	590	0x004f (79)	49153 → 300 [ACK] Seq=38057 Ack=1 Win=131072 Len=536
124	9.649279	10.1.1.1	10.2.2.1	TCP	590	0x0050 (80)	[TCP Retransmission] 49153 → 300 [ACK] Seq=24121 Ack=
125	9.696479	10.1.1.1	10.2.2.1	TCP	590	0x0051 (81)	49153 → 300 [ACK] Seq=38593 Ack=1 Win=131072 Len=536

Source: 10.1.1.1
Destination: 10.2.2.1

▼ Transmission Control Protocol, Src Port: 49153, Dst Port: 300, Seq: 22513, Ack: 1, Len: 536

Source Port: 49153
Destination Port: 300
[Stream index: 0]
[TCP Segment Len: 536]
Sequence number: 22513 (relative sequence number)

Figura 22. Paquete retransmitido con número de secuencia 22513.

Parte 3

Estándar H323

H.323 (Arquitectura 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-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
- _ 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.

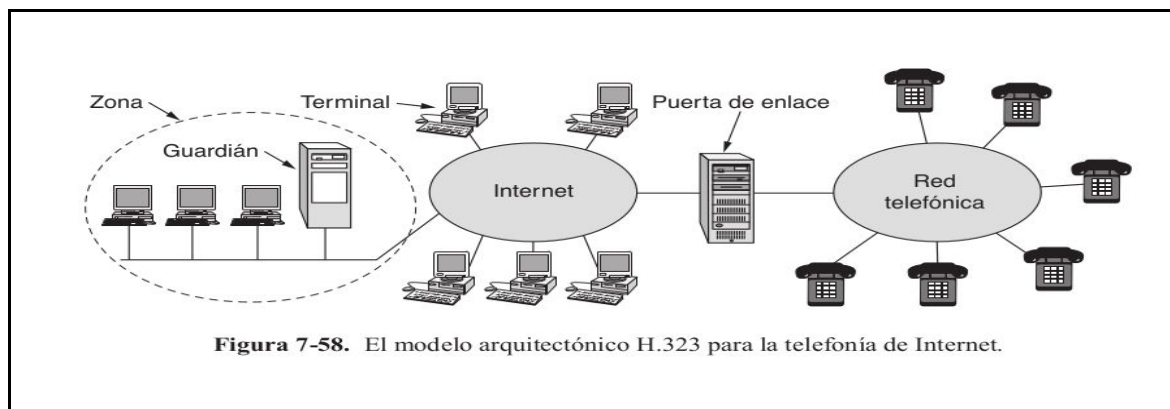


Figura 23. Modelo Arquitectónico H323.

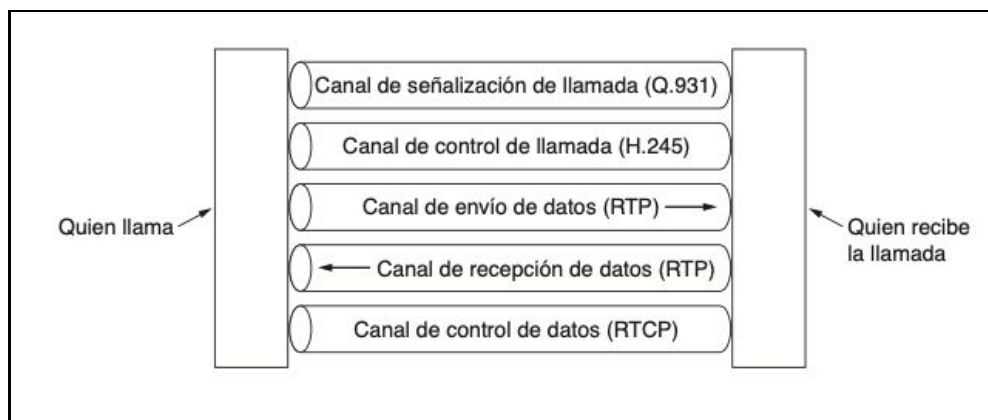


Figura 24. Canales Lógicos entre el que hace la llamada y el que la recibe durante la llamada.

Audio	Video	Control			
G.7xx	H.26x	RTCP	H.225 (RAS)	Q.931 (Señalización)	H.245 (Control de llamadas)
RTP					
UDP				TCP	
IP					
Protocolo de capa de enlace					
Protocolo de capa física					

Figura 25. 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 bits 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 26. 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.

$67,2 \text{ kbps} * \text{cantMaxLLamadasSimultaneas} \leq 200 \text{ kbps}$
 $\text{cantMaxLLamadasSimultaneas} \leq (200 \text{ kbps}) / (67,2 \text{ kbps}) = 2,976$
 $\text{cantMaxLLamadasSimultaneas} = 2$

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**

$11,2 \text{ kbps} * \text{cantMaxLLamadasSimultaneas} \leq 200 \text{ kbps}$
 $\text{cantMaxLLamadasSimultaneas} \leq (200 \text{ kbps}) / (11,2 \text{ kbps}) = 17,857$
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.

Esto es debido a que G.711 tiene un tamaño del paquete más grande (esto debido a que la carga util de G.711 es de 160 bytes mientras que G.729 es de 20 bytes) y por ende ocupa mayor ancho de banda por llamada con respecto a G.729, y por lo tanto ante igual condiciones (mismo ancho de banda del canal) G.729 permite más llamadas en simultáneo.

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](#)