

# Trabajo Práctico 3

## Análisis de Redes

### Sistemas Operativos y Redes II

#### Docentes:

Chuquimango Chilon Luis Benjamin

Echabarri Alan Pablo Daniel

#### Alumnos:

Lopez, Diego (42823629/2019) <dl3464452@gmail.com>

Ruiz, Kevin (41672607/2017) <kevinruiza77@gmail.com>

Sanchez Rodriguez, Camila (41024628/2018) <csanchez\_98@outlook.com>

Santillán, Javier (40226822/2019) <javierignaciosantillan.u@gmail.com>

Soria, Lucía (43911246/2020) <luciacsoria5@gmail.com>

#### Fecha de entrega:

2 de junio de 2024

## Índice

Índice.....	2
Introducción.....	3
Repositorio del proyecto.....	3
Ambiente de desarrollo.....	3
Protocolo UDP.....	4
Protocolo TCP.....	4
Three Way Handshake.....	4
Etapas del protocolo TCP.....	5
Slow Start (Inicio Lento).....	5
Congestion Avoidance (Evitación de congestión).....	6
Fast Retransmit (Retransmisión Rápida).....	7
Fast Recovery (Recuperación Rápida).....	7
Fin de la transmisión - Four Way Handshake.....	8
New Reno.....	9
Desarrollo.....	9
1. Diseño de la topología Dumbbell.....	9
2. Primera parte: Pruebas con dos emisores TCP.....	10
Inicio de la conexión.....	10
Saturación del canal.....	12
Velocidad de transferencia.....	13
Etapas del protocolo TCP.....	16
Ancho de banda del canal.....	19
Anomalías.....	19
Fin de la conexión.....	20
Explicación código.....	21
Preparación del Entorno.....	21
Ejecución de los Archivos.....	22
Descripción del Código.....	22
Funciones.....	22
Flujo de Ejecución.....	24
3. Segunda Parte: Pruebas con dos emisores TCP y uno UDP.....	31
Análisis de la simulación con UDP y TCP.....	31
Conclusión.....	33
Bibliografía.....	33

## Introducción

Dentro del campo de la comunicación, la investigación utilizando la simulación de redes implica un papel fundamental para la innovación y mejora de los sistemas. Dentro de estos se encuentra el simulador NS-3 , *Network Simulator version 3*, el cual es un simulador de código abierto que ofrece la capacidad de diseñar, ejecutar y examinar distintos escenarios en redes de comunicación, resaltando su flexibilidad y fiabilidad.

Esta herramienta es ampliamente utilizada en la investigación y desarrollo de protocolos, así como en la medición del desempeño de los mismos ya que admite una variedad de tecnologías de red, desde redes alámbricas hasta inalámbricas, permitiendo abarcar una gran gama de protocolos de enrutamiento, transporte y aplicaciones.

En este informe, se describe la implementación de una red con una topología **Dumbbell** usando ns-3, donde se llevarán a cabo diversas pruebas, incluyendo la saturación del canal, el análisis de las fases del protocolo TCP, la medición del ancho de banda y la detección de posibles anomalías. Se analizarán los nodos que estén operando bajo el protocolo TCP como aquellos bajo el UDP.

## Repositorio del proyecto

<https://github.com/jisantillan/tp-redes-sor2>

## Ambiente de desarrollo

Para la realización de este trabajo utilizamos la siguiente distribución de Linux: Ubuntu 20.04.2 LTS. Además, se utilizaron las siguientes herramientas: NS-3, NetAnim, Gnuplot y Wireshark.

En primer lugar, Network Simulator 3 (NS-3) es un simulador de red de eventos discretos para sistemas de Internet, desarrollado en C++. Proporciona un ambiente virtual donde estudiantes e investigadores pueden modelar, analizar, y evaluar el rendimiento de redes de comunicación, lo que les permite desarrollar un entendimiento más profundo sobre protocolos y algoritmos. Destacamos que es un software gratuito de código abierto, con licencia GNU GPLv2 y mantenido por una gran comunidad. Existen múltiples versiones de este simulador, y nosotros decidimos utilizar la 3.31.

Para facilitarnos la visualización de la información, también empleamos NetAnim. Esta herramienta permite animar una simulación previamente ejecutada utilizando un archivo de seguimiento XML generado durante la simulación.

Para la creación de gráficos, empleamos Gnuplot, una herramienta gráfica que permite generar visualizaciones en 2D y 3D a partir de conjuntos de datos. Esta utilidad es especialmente importante en contextos donde la representación visual de los datos es fundamental para la toma de decisiones, como en la ciencia e ingeniería. En nuestro caso, resultó muy útil para producir gráficos que representen los distintos estados de las variables analizadas.

Por último, Wireshark es un analizador de paquetes de red que captura y examina el tráfico que pasa a través de una red. Al igual que NS-3, es gratis y de código abierto. Es útil para diagnosticar problemas de red, efectuar auditorías de seguridad y también para aprender más sobre redes informáticas, como es nuestro caso.

## Protocolo UDP

El Protocolo de Datagramas de Usuario, mejor conocido como UDP, es un protocolo de comunicación en redes que proporciona la transmisión de datos sin conexión.

UDP envía datagramas (paquetes de datos) de forma independiente, y sin establecer una conexión antes de la transmisión. Esto hace que UDP sea considerado un protocolo no confiable, ya que no asegura que los paquetes lleguen a destino, y tampoco que lleguen en orden. Sin embargo, la simpleza y velocidad que proporciona este protocolo lo convierten en una gran elección para aplicaciones donde la velocidad y eficiencia son prioritarias. Por ejemplo, aplicaciones que requieren transmisiones en tiempo real y juegos en línea.

## Protocolo TCP

El Protocolo de Control de Transmisión, conocido como TCP, es un estándar de comunicación en redes que proporciona la entrega segura y ordenada de datos entre dispositivos conectados. Su función principal es segmentar los mensajes de la capa de aplicación en unidades más pequeñas (segmentos o datagramas) para su transporte por la capa de red, y luego reensamblarlos en el destino para entregarlos en la capa de aplicación correspondiente. Además, este protocolo maneja la creación de conexiones, confirma la recepción de los paquetes y, en caso de no recibirse, los retransmite.

### Three Way Handshake

Las conexiones TCP se componen de tres etapas:

- Establecimiento de la conexión (3-way handshake)
- Transferencia de datos
- Fin de conexión

Para iniciar una conexión, se emplea un método conocido como *“negociación en tres pasos”* o 3-way handshake. Al establecer la conexión, se ajustan varios parámetros, incluyendo el número de secuencia, lo cual garantiza que los datos se entreguen de manera ordenada y que la comunicación sea estable y confiable.

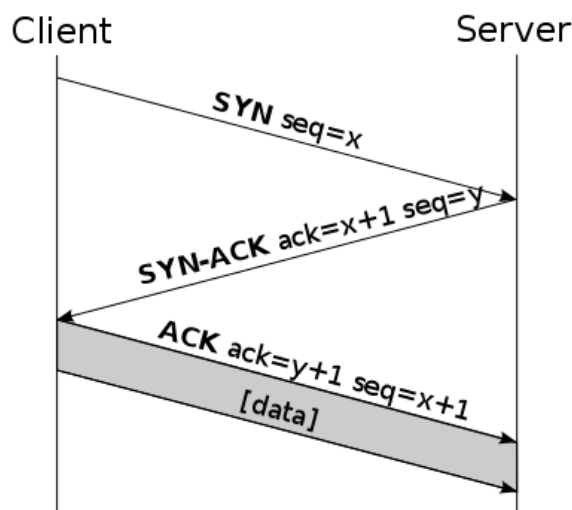


Imagen 1: Three way handshake

El proceso de *handshaking* de tres vías inicia cuando la primera entidad (A) envía un paquete de sincronización, o SYN, a la segunda entidad (B).

Una vez que B recibe el paquete SYN, procede a responder con un paquete SYN-ACK, que sirve tanto para reconocer el paquete SYN recibido como para enviar su propio paquete de sincronización.

Posteriormente, la entidad A recibe el paquete SYN-ACK y responde enviando un paquete de reconocimiento, o ACK, completando así el proceso y estableciendo la conexión entre ambas entidades.

## Etapas del protocolo TCP

En esta sección se explicarán las diferentes etapas del protocolo TCP para la prevención de la congestión de redes. Para lograr esto, TCP emplea un mecanismo de control que involucra al emisor como al receptor.

La gestión de congestión en las redes TCP involucra una serie de estrategias y mecanismos dinámicos diseñados para prevenir la saturación de la red, optimizar el rendimiento y garantizar una transmisión de datos eficiente y ordenada. Este control es crucial, ya que la congestión ocurre cuando los emisores envían más paquetes de los que un router puede procesar en un determinado momento.

Para regular la frecuencia de envío y adaptarse a las variaciones de la red, el protocolo TCP utiliza variables como la “ventana de congestión” (*Congestion Windows* o *CWND*) y el “umbral de congestión” (*Congestion Threshold* o *SSTH*). La ventana de congestión, o *CWND*, limita la cantidad de datos que el emisor puede tener en tránsito, ajustándose según la percepción de la congestión. Por ejemplo, si se detecta poca congestión, el emisor puede incrementar su tasa de emisión, mientras que en situaciones de mayor congestión, la tasa se reducirá para evitar la pérdida de paquetes y otros problemas asociados.

El tamaño de la ventana de congestión se determina también por el valor de la ventana de recepción (*RWND*) y por el valor, en sí, de la ventana de congestión (*cwnd*), que se ajusta en respuesta a eventos de congestión.

Por otro lado, el umbral de congestión, o *SSTH*, actúa como un indicador para cambiar entre diferentes tácticas de control de congestión: por encima de este umbral, el TCP utiliza mecanismos como la evitación de la congestión, mientras que por debajo, puede optar por técnicas como arranque lento para incrementar gradualmente la ventana de congestión hasta encontrar un equilibrio entre eficiencia y estabilidad de la red. La configuración inicial de *SSTH* se establece en un valor alto y se ajusta hacia abajo en respuesta a la detección de la congestión, para prevenir futuras saturaciones.

Este conjunto de herramientas y parámetros permite a TCP adaptarse a las variaciones en la carga de la red, manteniendo una transmisión eficiente y minimizando la pérdida de paquetes.

## Slow Start (Inicio Lento)

El mecanismo de Slow Start en TCP tiene como objetivo incrementar la ventana de congestión (*CWND*) de manera exponencial hasta alcanzar el umbral de congestión (*SSTH*). Inicialmente, al comenzar la transmisión, el valor de *CWND* se establece en un segmento de tamaño máximo (*MSS*), y se incrementa en un segmento por cada acuse de recibo (*ACK*) recibido.

$$CWND_{k+1} = CWND_k + 1x MSS$$

Esto resulta en que, por cada intervalo de tiempo de ida y vuelta (*RTT*), el tamaño de la ventana de congestión se duplica, generando un aumento rápido de la tasa de transmisión.

Durante la fase de *Slow Start*, pueden ocurrir dos situaciones de detección de congestión:

1. Si se detectan tres ACKs duplicados durante la transmisión, esto indica la pérdida de paquetes, finalizando la etapa de *Slow Start*. En este caso, se procede con la ejecución de *Fast Retransmit*.
2. Si se produce una pérdida de paquetes, evidenciada por el vencimiento del tiempo de transmisión (*RTO*), el umbral de congestión se ajusta a la mitad del valor actual de *CWND* ( $SSTH = CWND/2$ ), y *CWND* se reduce a  $1 \times MSS$  reanudando el proceso de *Slow Start*.
3. Por otro lado, si *CWND* alcanza el umbral *SSTHRESH*, se activa la ejecución de *Congestion Avoidance*.

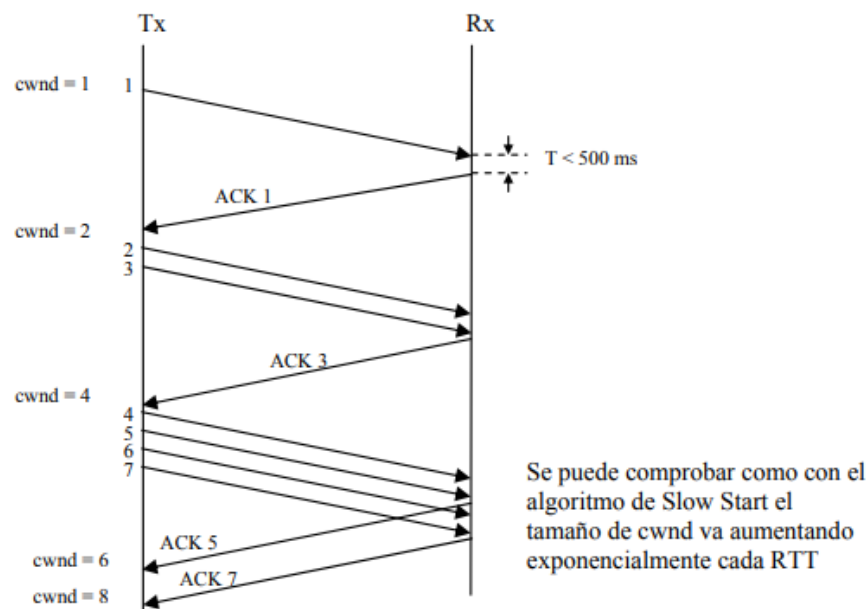


Imagen 2: Ejemplificación de un *Slow Start*

### Congestion Avoidance (Evitación de congestión)

Como se mencionó anteriormente, la ejecución de *Congestion Avoidance* se activa cuando la *CWND* alcanza el umbral *SSTHRESH*. La misma ocurre mientras  $cwnd > ssthresh$ . En esta etapa, la *cwnd* se incrementa un *MSS* por cada *RTT*. Es decir,

$$CWND_{k+1} = CWND_k + MSS, \text{ por cada } RTT$$

De esta forma, la *cwnd* crece de forma lineal. Eso sucede hasta que ocurra alguno de los siguientes sucesos:

1. Ocurre el vencimiento del tiempo de transmisión (*RTO*), es decir, se detecta una pérdida de paquetes. En este caso, se setea el umbral de congestión  $ssthresh = cwnd/2$ , y también se setea  $cwnd = 1 \times MSS$ . Luego, se vuelve a activar la ejecución del *Slow Start*.

2. Se detectan tres ACKs duplicados. En este caso, se realiza *Fast Retransmit*, y se activa la ejecución de *Fast Recovery*.
3. Ocurre que  $cwnd = rwnd$ . En este caso, el  $cwnd$  se mantiene constante.

### Fast Retransmit (Retransmisión Rápida)

Ayuda a aligerar los casos en los que basarse en el *RTO* para la retransmisión de segmentos resulta lento. Tras recibir tres *ACKs* duplicados, *Fast Retransmit* retransmite el segmento sin esperar el *RTO*, ya que se considera que, si bien se ha perdido un segmento, la congestión de la red no es tan grave como para tomar medidas muy drásticas.

Luego de su ejecución, se activa la ejecución de *Fast Recovery*.

### Fast Recovery (Recuperación Rápida)

En este caso, se setea el umbral de congestión *ssthresh* en  $cwnd/2$ , y también la ventana de congestión  $cwnd$  en  $ssthresh + 3 * MSS$ . Luego, cada vez que se reciba un *ACK* duplicado, se actualizará nuevamente el valor de  $cwnd$ , de forma tal que  $cwnd = cwnd + 1 * MSS$ . Esto es para reflejar los segmentos que están en el buffer del receptor, es decir, fuera de la red. Hay dos escenarios posibles para salir de *Fast Recovery*:

1. Si se recibe un *ACK* de un paquete perdido, la ventana de congestión vuelve a tomar el valor seteado al comienzo de la ejecución, es decir,  $cwnd = ssthresh + 3 * MSS$ . Después, se regresa a Congestion Avoidance.
2. Si antes de recibir el *ACK* del paquete perdido ocurre un vencimiento del tiempo de transmisión (*RTO*), también se setea  $cwnd = ssthresh + 3 * MSS$ , pero, en este caso, se regresa a *Slow Start*.

*Fast Recovery* permite la recuperación rápida tras la pérdida de paquetes, ya que implica la retransmisión selectiva de los paquetes que se han recibido correctamente. Ayuda a mantener un flujo de datos constante y eficiente, reduciendo así el impacto de la pérdida de paquetes.

En el siguiente diagrama se pueden apreciar los cambios de estado relativos al control de congestión:

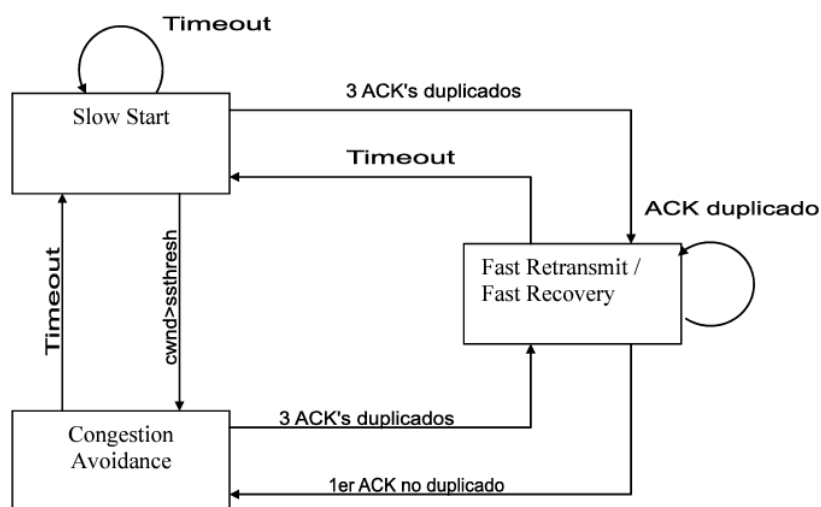


Imagen 3: Etapas del protocolo TCP

Por último, en el siguiente gráfico se puede observar cómo varía el tamaño de la ventana de congestión (*cwnd*), al igual que las distintas etapas o componentes del control de congestión, a lo largo de distintos intervalos de tiempo de ida y vuelta (*RTT*):

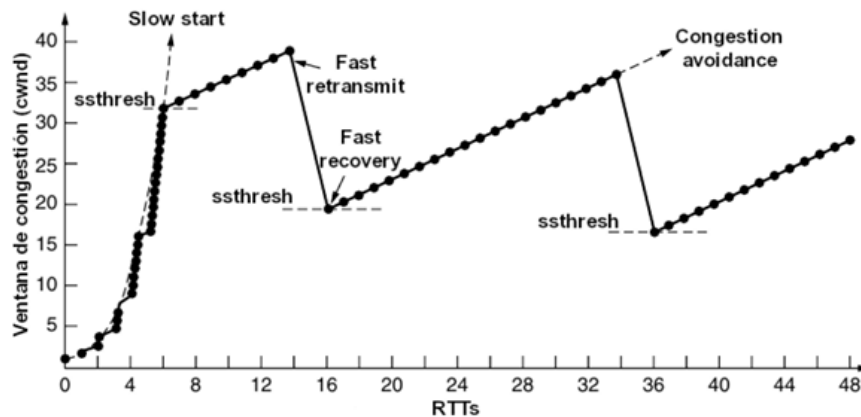


Imagen 4: Gráfica que muestra las etapas del protocolo TCP

## Fin de la transmisión - Four Way Handshake

Para finalizar una transmisión, TCP usa una negociación de cuatro pasos, donde cada lado de la conexión termina la misma de forma independiente. Cuando uno de los lados desea finalizar la conexión, transmite un segmento con el flag FIN encendido. El receptor del mensaje confirma la recepción del segmento enviando un ACK, y envía también un segmento con el flag FIN en 1. Finalmente, quien comenzó esta interacción confirma la llegada de este último segmento con un ACK.

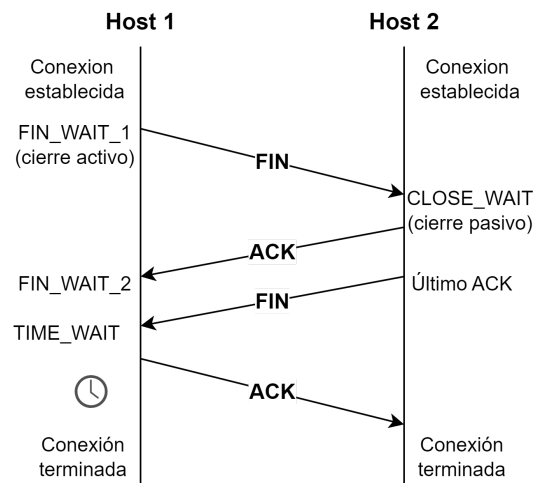


Imagen 5: Comunicación entre dos máquinas, se observa el final de esta

También podría suceder que quien recibe primero el cierre de conexión envíe en un mismo segmento el ACK y el flag FIN en encendido. En ese caso, la negociación es sólo de tres pasos (Three-way handshake).



## New Reno

En el siguiente trabajo, utilizaremos TCP New Reno. Esta es una variante del algoritmo de control de congestión de TCP, diseñada para mejorar el rendimiento en condiciones de pérdida de paquetes. TCP New Reno se introdujo como una mejora sobre TCP Reno, específicamente para manejar situaciones en las que se pierden múltiples paquetes dentro de una misma ventana de transmisión.

A diferencia de TCP Reno, que solo puede manejar una pérdida a la vez, TCP New Reno utiliza un mecanismo de reconocimiento parcial (*Partial Acknowledgment*) para continuar recuperándose rápidamente, incluso si se pierden varios paquetes. Esto permite retransmitir los paquetes perdidos de manera más eficiente sin salir del estado de recuperación rápida (*Fast Recovery*).

Esta variante proporciona, entonces, una recuperación más eficiente de múltiples pérdidas de paquetes en una sola ventana de transmisión, lo cual resulta especialmente útil en redes con alta latencia o alta tasa de pérdidas, como redes de larga distancia o redes inalámbricas.

## Desarrollo

### 1. Diseño de la topología Dumbbell

En primer lugar, se desarrolló el diseño de la Dumbbell Topology solicitada en la consigna del trabajo práctico. Esta topología cuenta con:

- Tres nodos emisores on/off. De estos nodos, dos utilizan TCP y uno UDP.
- Tres nodos receptores. Al igual que los emisores, dos utilizan TCP y uno UDP.
- Dos nodos (routers) intermedios.

Todas las conexiones son cableadas. Definimos que los enlaces tengan un data-rate de **100Kb/s**, a excepción del bottleneck cuyo data-rate es de **50Kb/s**. Se busca generar un cuello de botella (*bottleneck*) en la red y, con esto, provocar una congestión en la misma.

Para la visualización de la topología, se ha utilizado NetAnim para comprender la distribución de los nodos y sus conexiones luego del desarrollo del código. A continuación, se puede observar un gráfico realizado con Drawio.io para una mejor representación.

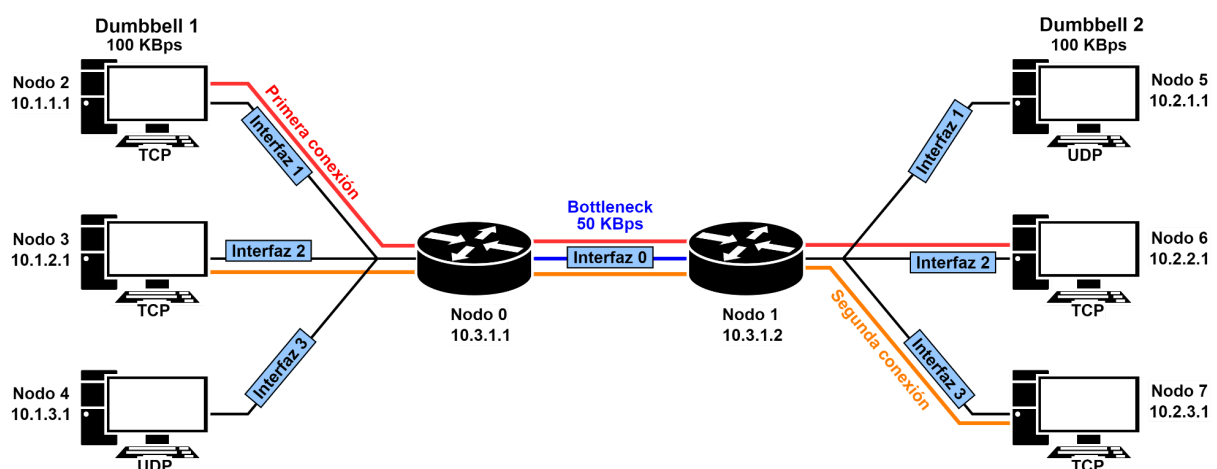


Imagen 6: Topología construida para la simulación

En el repositorio, se encontrará con una serie de archivos .pcap. Dichos archivos contienen las capturas de los paquetes que transitan por las distintas conexiones entre los nodos. Por ejemplo, el archivo “*dumbbell-tp2-4-0.pcap*” indica que el nodo origen es el 4 y la interfaz es la 0, por lo tanto, todos los paquetes capturados serán UDP. Además, debemos aclarar que todos los nodos con un solo vecino tienen una única interfaz, que es la 0.

Por otro lado, se definió la interfaz de los routers, Nodo 0 y Nodo 1, como la interfaz 0, que hará referencia al archivo .pcap del nodo 0 e interfaz 0, es decir, “*dumbbell-tp2-0-0.pcap*”. Esto implica que todos los paquetes de la simulación pasarán por el enlace entre los dos routers. En cambio, el archivo .pcap 0-3 (Nodo 0 y Nodo 4) únicamente contendrá las capturas de los paquetes UDP.

Inicialmente, las pruebas con los emisores TCP se realizaron utilizando el Nodo 2 e interfaz 0 y Nodo 3 Interfaz 0.

## 2. Primera parte: Pruebas con dos emisores TCP

Los archivos .pcap correspondientes a la ejecución del script dumbbell-topology-tcp-wireshark.cc se encontrarán dentro de la carpeta punto2 del repositorio.

### Inicio de la conexión

Utilizando Wireshark, podemos observar el inicio de la conexión mediante el método *Three-way Handshake*, el cuál se ha explicado anteriormente.

Time	Source	Destination	Protocol	Length	Info
0.000000	10.1.1.1	10.2.2.1	TCP	58	49153 → 3000 [SYN] Seq=0 Win=65535 Len=0 TSval=0 TSecr=0 WS=4 SACK_PERM=1
0.004640	10.1.2.1	10.2.3.1	TCP	58	49153 → 3000 [SYN] Seq=0 Win=65535 Len=0 TSval=0 TSecr=0 WS=4 SACK_PERM=1
0.418560	10.2.2.1	10.1.1.1	TCP	58	3000 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSval=313 TSecr=0 WS=4 SACK_PERM=1
0.423200	10.2.3.1	10.1.2.1	TCP	58	3000 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSval=318 TSecr=0 WS=4 SACK_PERM=1
0.627520	10.1.1.1	10.2.2.1	TCP	54	49153 → 3000 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=627 TSecr=313
0.632160	10.1.2.1	10.2.3.1	TCP	54	49153 → 3000 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=632 TSecr=318
0.674719	10.1.1.1	10.2.2.1	TCP	590	49153 → 3000 [ACK] Seq=1 Ack=1 Win=131072 Len=536 TSval=627 TSecr=313
0.721919	10.1.2.1	10.2.3.1	TCP	590	49153 → 3000 [ACK] Seq=1 Ack=1 Win=131072 Len=536 TSval=632 TSecr=318

Imagen 7: Inicio de transmisión de los paquetes visto en Wireshark

Podemos observar, en los recuadros de color rojo, cómo los nodos 2 y 3 inician la sincronización: el nodo 2 (*ip 10.1.1.1*) envía un paquete de sincronización (SYN) al nodo 6 (*ip 10.2.2.1*). De la misma forma, el nodo 3 (*ip 10.1.2.1*) envía un paquete de sincronización (SYN) al nodo 7 (*ip 10.2.3.1*).

Inmediatamente después, los nodos 6 y 7 responden a la sincronización enviando a los nodos 2 y 3 (respectivamente) el correspondiente acuso de confirmación (ACK). Esto podemos verlo en los recuadros de color amarillo.

En Wireshark, si accedemos a la información de cualquiera de estos cuatro paquetes, podemos observar que todos tienen el flag SYN encendido, como se ve en la imagen 8.

```

1001 .... = Header Length: 36 bytes (9)
> Flags: 0x002 (SYN)
  000. .... = Reserved: Not set
  ...0 .... = Nonce: Not set
  .... 0... = Congestion Window Reduced (CWR): Not set
  .... .0.. = ECN-Echo: Not set
  .... ..0. = Urgent: Not set
  .... ...0 = Acknowledgment: Not set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
  > .... .... ..1. = Syn: Set
  .... .... ...0 = Fin: Not set
[TCP Flags: .....S.]

```

Imagen 8: Flags de los paquetes del inicio de transmisión.

Finalmente, podemos ver en la imagen 7, encuadrado en verde, que los nodos 2 y 3 dan por finalizado el three-way handshake enviando otro acuse de confirmación. Luego de esto, la conexión está establecida, por lo que ambos nodos comienzan a transmitir la información. Podemos saber que los paquetes contienen información por el atributo *Len=536*. Si *Len=0*, entonces que el paquete no cuenta con información (como es el caso de todos los paquetes enviados previamente).

Por otro lado, dentro de esta etapa se define la ventana de recepción, es decir, el buffer del lado del emisor y receptor que retiene temporalmente los datos que llegan. Como se muestra en la siguiente imagen, el Three-Way Handshake establece su tamaño de ventana en 65535 bytes. Sin embargo, hace uso de la “Window size”, sección del header de TCP. Este campo ocupa 16 bits en el segmento, por lo que el mayor valor posible es de 65536. A este campo, se lo debe multiplicar por el “Window size scaling factor” y, se obtendría, el verdadero valor del tamaño de la ventana de recepción.

```

> Flags: 0x002 (SYN)
Window size value: 65535
[Calculated window size: 65535]
Checksum: 0x0000 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (16 bytes), Timestamps, Window scale, SACK permitted, End of Option List (EOL)
  > TCP Option - Timestamps: TSval 0, TSecr 0
  > TCP Option - Window scale: 2 (multiply by 4)
  > TCP Option - SACK permitted
  > TCP Option - End of Option List (EOL)
[Expert Info (Note/Protocol): The SYN packet does not contain a MSS option]

```

Imagen 9: Ventana de recepción del nodo 2.

```

> Flags: 0x010 (ACK)
Window size value: 32768
[Calculated window size: 131072]
[Window size scaling factor: 4]
Checksum: 0x0000 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0

```

Imagen 10: Ventana de recepción del nodo 2 en la emisión del primer paquete.

## Saturación del canal

Con el objetivo de realizar un análisis de la red, se fueron recopilando datos de esta para luego generar gráficos que nos ayuden a entender que es lo que estuvo ocurriendo en nuestra simulación.

En la imagen 11, por ejemplo, podemos ver la ventana de congestión del nodo 2, y observamos una congestión de la red alrededor del segundo 5,7.

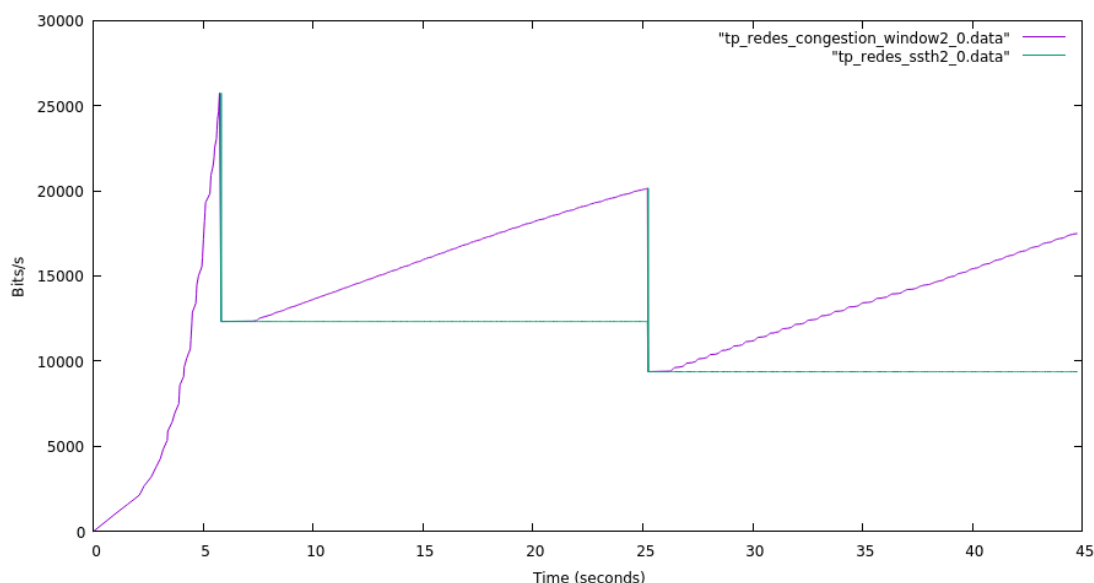


Imagen 11: Gráfica de la ventana de congestión del nodo 2

Este mismo suceso lo podemos ver en los datos de Wireshark para los .pcap del nodo 2 con la interfaz 0 (archivo *tp\_redes\_-2-0.pcap*). En la imagen 12 se visualiza la pérdida de paquetes produciendo los ACK duplicados por parte del nodo 6, que conlleva luego de los 3 ACK duplicados a un Fast Retransmission, que se puede observar en la imagen 11 como la caída de la gráfica.

189	5.760859	10.1.1.1	10.2.2.1	TCP	590	49153 → 3000 [ACK] Seq=73969 Ack=1 Win=131072 Len=536 TSval=5749 TSecr=5490
190	5.761179	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 186#1] 3000 → 49153 [ACK] Seq=1 Ack=48777 Win=131072 Len=0 TS
191	5.766759	10.1.1.1	10.2.2.1	TCP	590	49153 → 3000 [ACK] Seq=74505 Ack=1 Win=131072 Len=536 TSval=5761 TSecr=5500
192	5.772979	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 186#2] 3000 → 49153 [ACK] Seq=1 Ack=48777 Win=131072 Len=0 TS
193	5.772979	10.1.1.1	10.2.2.1	TCP	590	49153 → 3000 [ACK] Seq=75041 Ack=1 Win=131072 Len=536 TSval=5772 TSecr=5520
194	5.820179	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 186#3] 3000 → 49153 [ACK] Seq=1 Ack=48777 Win=131072 Len=0 TS
195	5.820179	10.1.1.1	10.2.2.1	TCP	596	[TCP Fast Retransmission] 49153 → 3000 [ACK] Seq=48777 Ack=1 Win=131072 Len=0 TS
196	5.831979	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 186#4] 3000 → 49153 [ACK] Seq=1 Ack=48777 Win=131072 Len=0 TS

Imagen 12: Pérdida de paquetes en la comunicación entre el nodo 2 y 6

Análogamente, con el nodo 3 tenemos una situación similar. En las imágenes 12 y 13 podemos observar como en el segundo 5,7 aproximadamente ocurre la congestión de la red, se produce la pérdida de paquetes y se acciona el *Fast Retransmission*.

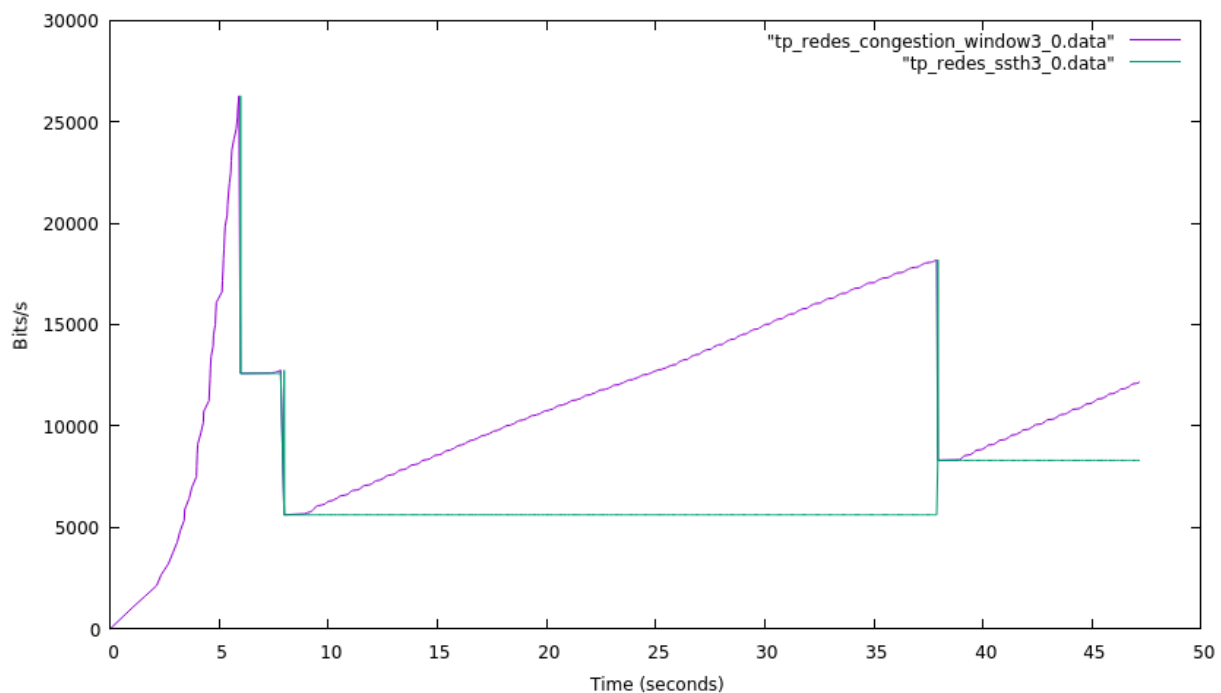


Imagen 13: Gráfica de la ventana de congestión del nodo 3

193	5.961779	10.2.3.1	10.1.2.1	TCP	62	[TCP Dup ACK 189#1]	3000 → 49153	[ACK]	Seq=1 Ack=49313 Win=131072
194	5.961779	10.1.2.1	10.2.3.1	TCP	590	49153 → 3000	[ACK]	Seq=75577 Ack=1 Win=131072 Len=536 TSval=5961 T	
195	5.973579	10.2.3.1	10.1.2.1	TCP	62	[TCP Dup ACK 189#2]	3000 → 49153	[ACK]	Seq=1 Ack=49313 Win=131072
196	5.973579	10.1.2.1	10.2.3.1	TCP	590	49153 → 3000	[ACK]	Seq=76113 Ack=1 Win=131072 Len=536 TSval=5973 T	
197	5.985379	10.2.3.1	10.1.2.1	TCP	62	[TCP Dup ACK 189#3]	3000 → 49153	[ACK]	Seq=1 Ack=49313 Win=131072
198	5.985379	10.1.2.1	10.2.3.1	TCP	590	[TCP Fast Retransmission]	49153 → 3000	[ACK]	Seq=49313 Ack=1 Win=1
199	6.020779	10.2.3.1	10.1.2.1	TCP	62	[TCP Dup ACK 189#4]	3000 → 49153	[ACK]	Seq=1 Ack=49313 Win=131072

Imagen 14: Pérdida de paquetes en la comunicación entre el nodo 3 y 7

## Velocidad de transferencia

Podemos ver como la velocidad de transferencia de datos cambia a lo largo de la simulación, tanto para el nodo 2 como para el nodo 3.

Antes de calcular la velocidad de transferencia, se definirá a la misma como la **velocidad real a la que los datos se transmiten desde un punto a otro a través de una conexión de red**.

Por lo tanto, este cálculo se debe efectuar en el momento donde se da el pico máximo de transferencia, esto se da milisegundos antes de la congestión de la red.

Para hacerlo se utilizará los datos brindados por uno de los gráficos de Wireshark, en los cuales se aplicaran filtros mediante las direcciones IP de los nodos, para visualizar la cantidad de bits por segundo e identificar el tiempo antes de la congestión.



Imagen 15: Gráfico de la velocidad de ambas transferencias en bits por segundo

A partir de esta imagen, se puede observar, que a partir de dicho momento todo el gráfico tiene un comportamiento simétricamente opuesto. Es decir, cuando una de las gráficas disminuye en la cantidad de bits por segundo, la otra aumenta. Esto se debe al cuello de botella establecido para la simulación y a las etapas de control de congestión de TCP.

Además, se puede observar que al momento en que ambas gráficas no tienen el comportamiento descrito es al inicio, cuando ambas están en la etapa de "Slow Start". Posterior a esto, comienza a aplicarse el control de congestión y varían sus comportamientos.

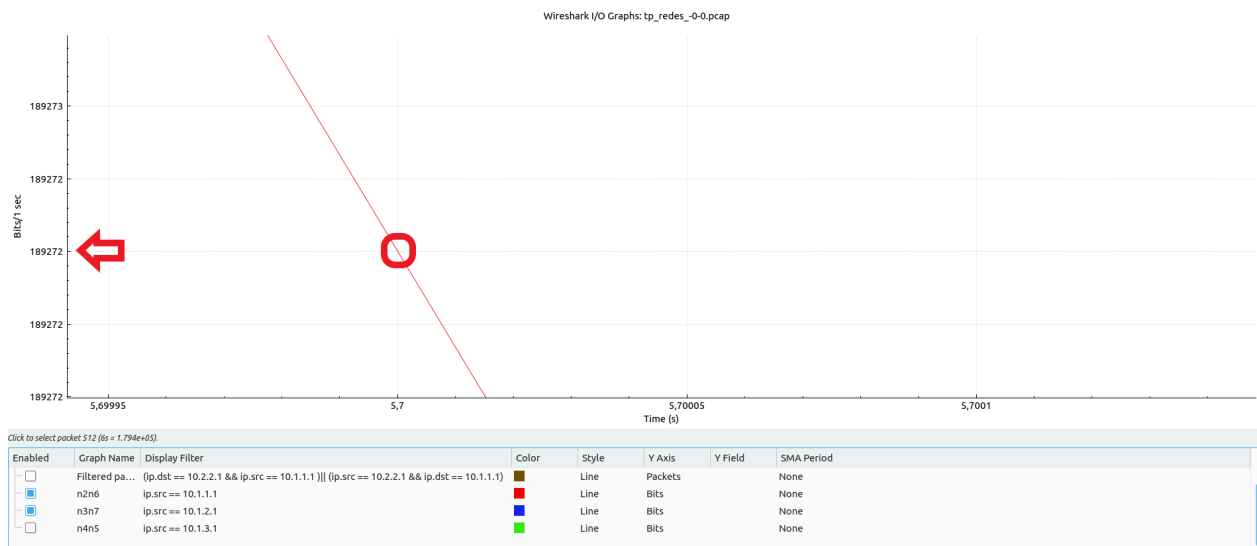


Imagen 16: Velocidad de transferencia del nodo 2 a 6 antes de la congestión.



Imagen 17: Velocidad de transferencia del nodo 3 al 7 antes de la congestión.

Como se puede observar, en el segundo **5,7** la transferencia entre el nodo 2 y el nodo 6 (línea roja) se ve afectada por una pérdida de paquetes, por lo tanto, se analizará en ese instante de tiempo. Esto se puede observar a partir de la imagen 13.

La transferencia del nodo 2 al 6 tiene una tasa de **182272 bits** por segundo y la transferencia del nodo 3 al 7 tiene una tasa de **165672 bits** por segundo. Luego, sumando ambas velocidades se obtiene una velocidad total de **354944 bits** por segundo, que es igual a **43,328125 KBps**.

Al alcanzar el punto de congestión establecido, las velocidades de transferencia se ven afectadas. En este caso, la transferencia del nodo 3 a 7 disminuye su velocidad mientras que la correspondiente del nodo 2 al 6, aumenta de manera simétrica, manteniendo un límite de **50 KBps** establecido. Además, se puede decir que, como se mantiene el comportamiento simétrico a lo largo de la duración restante de la simulación, el límite de velocidad nunca es superado.



Imagen 18: Gráfico completo de la velocidad de ambas transferencias en bits por segundo

## Etapas del protocolo TCP

Si analizamos las variaciones en el tamaño de la ventana de congestión de los nodos emisores, es decir, los nodos 2 y 3, podemos identificar las distintas etapas del protocolo TCP.

Analizaremos primero la ventana de congestión del nodo 2. Podemos observar en la imagen 19 que el nodo envía paquetes a la red de forma crecientemente exponencial (**Slow Start**). Esto genera, como se ha explicado anteriormente, la congestión de la red (segundo 5,7 aproximadamente). La pérdida de paquetes que esto produce se evidencia en que el emisor recibe por parte del receptor varios *ACKs* duplicados. Por lo tanto, se ejecuta la retransmisión rápida de paquetes (**Fast Retransmission**), y, a continuación, la recuperación rápida (**Fast recovery**,  $ssthresh = cwnd/2$  y  $cwnd = ssthresh + 3*MSS$ ). Luego de esto, alrededor del segundo 7, la cantidad de paquetes que se envían a la red crece de forma lineal (se entra en **Congestión avoidance**), hasta que se vuelve a detectar una pérdida de paquetes. Cuando esto sucede, se repite la ejecución de Fast Retransmission y Fast Recovery, y nuevamente se entra en Congestion Avoidance (segundo 26 aproximadamente).

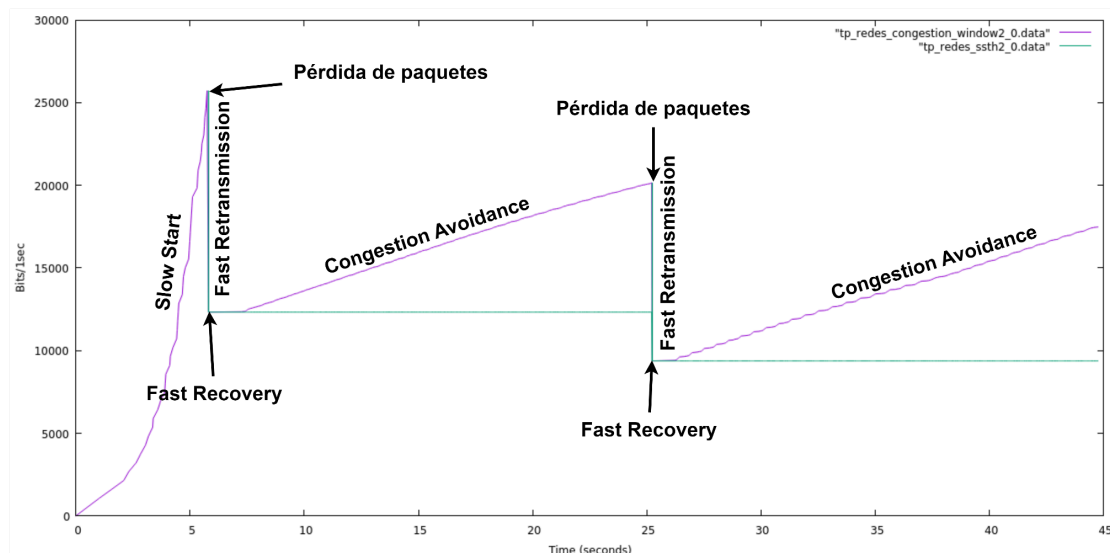


Imagen 19: Gráfico de la variación de la *cwnd* del nodo 2.

En segundo lugar, identificamos las distintas etapas del protocolo a través de los cambios en la ventana de congestión del nodo 3. Al igual que el nodo 2, podemos observar en la imagen 20 que el nodo 3 envía paquetes a la red de forma crecientemente exponencial (**Slow Start**). Esto genera la congestión de la red (segundo 5,7 aproximadamente). Como ya se ha explicado, la pérdida de paquetes que esto produce se evidencia en que el emisor recibe por parte del receptor varios *ACKs* duplicados. Por lo tanto, se ejecuta la retransmisión rápida de paquetes (**Fast Retransmission**), y, a continuación, la recuperación rápida (**Fast recovery**,  $ssthresh = cwnd/2$  y  $cwnd = ssthresh + 3*MSS$ ). Luego, como se ve en la imagen 20, marcado con un círculo rojo, se entra en **Congestión avoidance** por un breve instante, pero en menos de un segundo se vuelve a identificar una pérdida de paquetes (imagen 21). Se repite, entonces, Fast Retransmission y Fast Recovery, y, aproximadamente en el segundo 8, se entra en Congestion Avoidance. Al volver a detectarse una pérdida de paquetes (segundo 37 aproximadamente), se repite la ejecución de Fast Retransmission, Fast Recovery, y de Congestion Avoidance (alrededor del segundo 39).



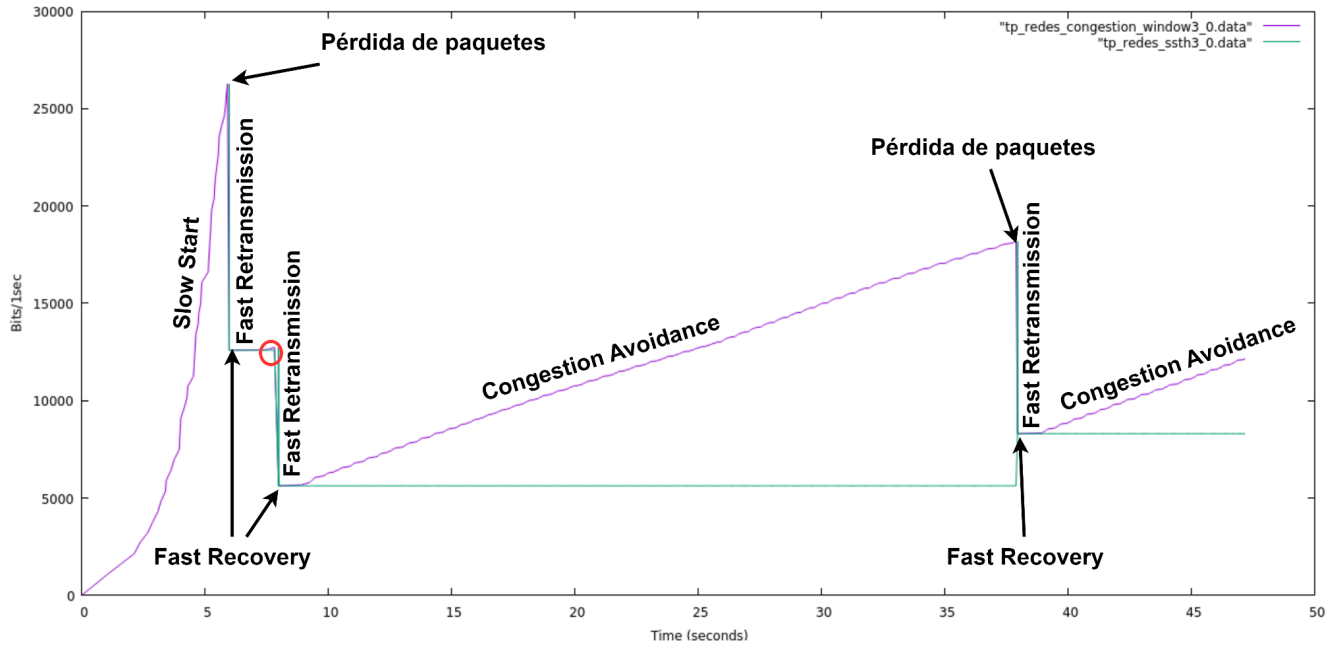


Imagen 20: Gráfico de la variación de la *cwnd* del nodo 3.

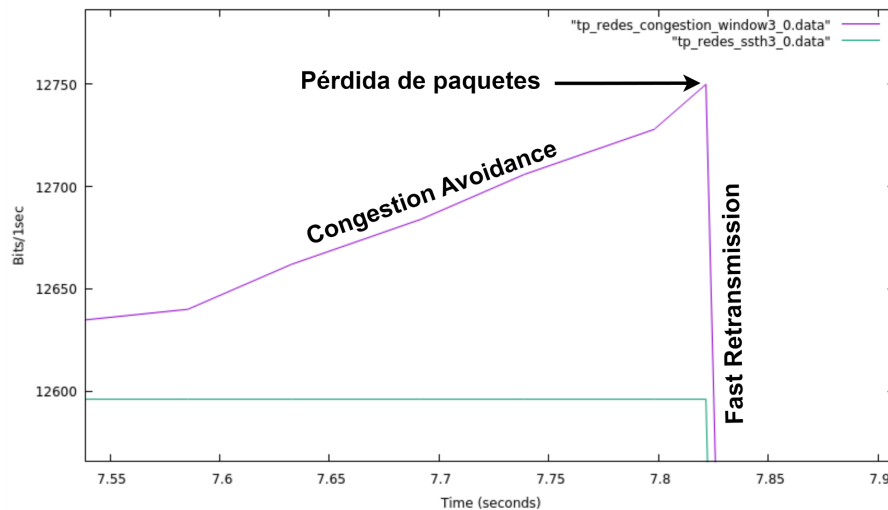


Imagen 21: Gráfico de la variación de la *cwnd* del nodo 3.

También podemos observar las etapas del protocolo TCP analizando los gráficos que nos muestran la cantidad de bytes enviados por los nodos emisores, pero que aún no fueron recibidos por los nodos receptores en un periodo de tiempo.

En primer lugar, podemos ver en el siguiente gráfico la cantidad de bytes que el nodo 2 envía al nodo 6. Se ve que al principio la cantidad de bytes aumenta de forma exponencial, pero que baja de forma abrupta en el segundo 6 aproximadamente, que es cuando finaliza *Slow Start* y se ejecutan *Fast Retransmission* y *Fast Recovery*. Luego, la cantidad de bytes enviados permanece más o menos estable (fase de *Congestion Avoidance*), hasta que en el segundo 25 aproximadamente vuelve a producirse una pérdida de paquetes y, en consecuencia, a ejecutarse *Fast Retransmission* y *Fast Recovery*.

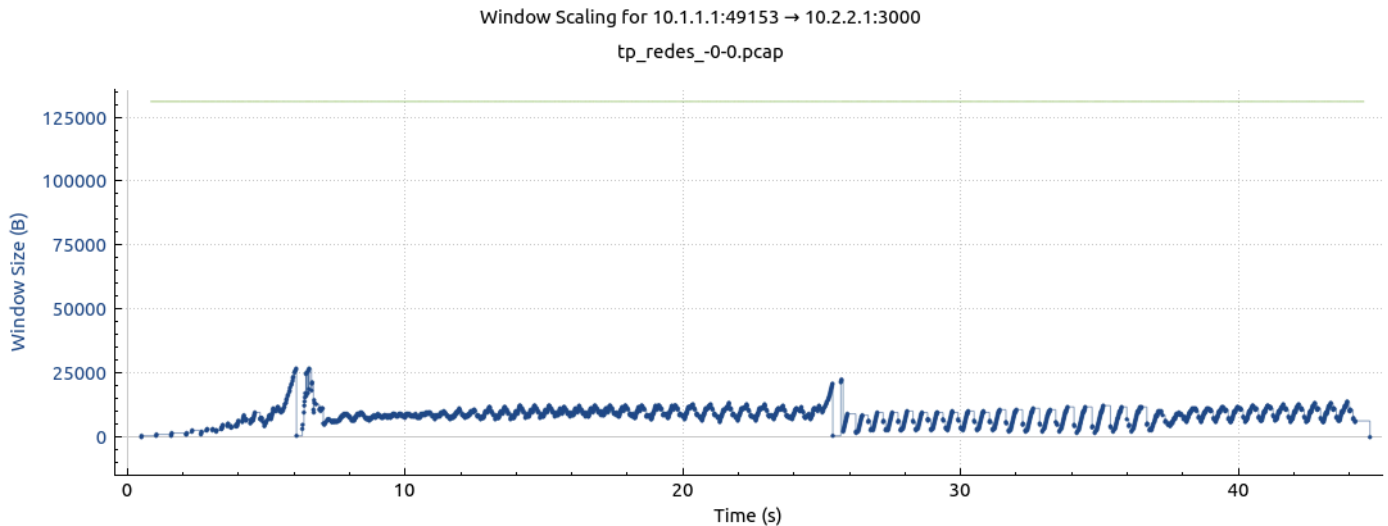


Imagen 22: Gráfico de la variación de *bytes in flight* entre el nodo 2 al nodo 6.

En segundo lugar, también podemos ver el siguiente gráfico que refleja la cantidad de bytes que el nodo 3 envía al nodo 7. Al igual que en el caso anterior, se ve que al principio la cantidad de bytes aumenta de forma exponencial (*Slow Start*), pero que baja de forma abrupta en el segundo 6 aproximadamente (se ejecutan *Fast Retransmission* y *Fast Recovery*). Inmediatamente después, se vuelve a producir un gran aumento en la transmisión de paquetes, pero vuelve a bajar abruptamente. Luego, de eso la cantidad de bytes enviados permanece estable (fase de *Congestion Avoidance*), hasta que en el segundo 37 aproximadamente vuelve a producirse una pérdida de paquetes y, en consecuencia, a ejecutarse *Fast Retransmission* y *Fast Recovery*.

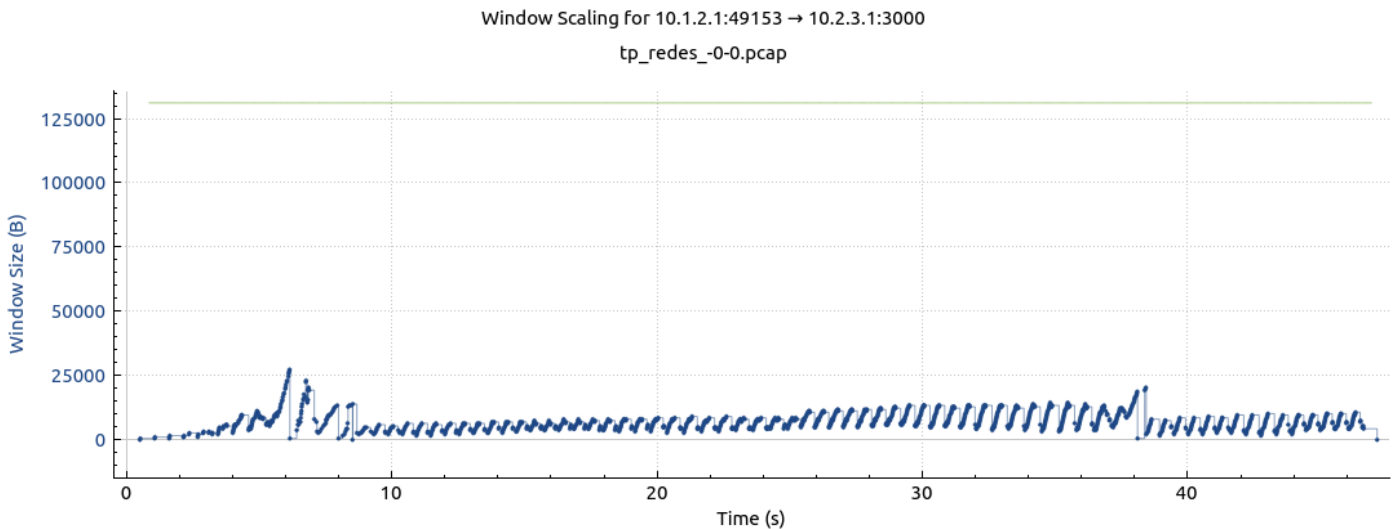


Imagen 23: Gráfico de la variación de *bytes in flight* entre el nodo 3 al nodo 7.

En la parte superior de los dos gráficos anteriores, podemos ver una línea de color verde. La misma representa el tamaño de la ventana de recepción de los nodos 6 y 7 (respectivamente). Como se ve en los gráficos, la misma no sufrió alteraciones durante la transmisión.

## Ancho de banda del canal

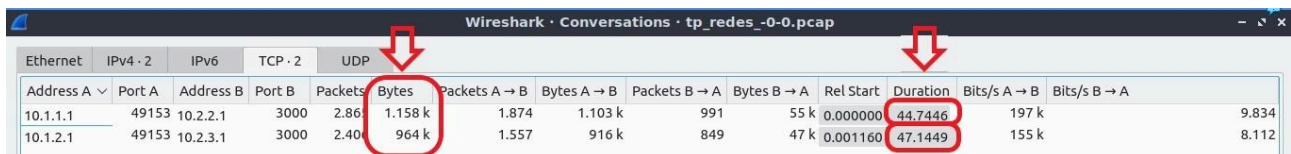
En la siguiente sección se analizará el ancho de banda del canal durante la captura de transferencias de datos que se ha realizado. Para esto, se tomará como ejemplo una captura de las conexiones entre dos nodos TCP y los otros nodos TCP. Pero primero se definirá a qué nos referimos con ancho de banda.

Ancho de banda se refiere a la **capacidad máxima de transferencia de datos de un canal de comunicación**. Esta es una medida teórica que generalmente se expresa en bits por segundo (*bps*) y sus múltiplos como kilobits por segundo (*Kbps*), megabits por segundo (*Mbps*) o gigabits por segundo (*Gbps*).

Para calcular el ancho de banda promedio se puede utilizar la siguiente fórmula:

$$\text{Ancho de banda} = \text{Cantidad total de datos transmitidos} / \text{Tiempo total de transferencia}$$

Para obtener estos valores, se ha utilizado la información proporcionada por la herramienta Wireshark en el menú “Statistics”.



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.1.1.1	49153	10.2.2.1	3000	2.86	1.158 k	1.874	1.103 k	991	55 k	0.000000	44,7446	197 k	9.834
10.1.2.1	49153	10.2.3.1	3000	2.40	964 k	1.557	916 k	849	47 k	0.001160	47,1449	155 k	8.112

Imagen 24: Cantidad de bytes transmitidos y duración tiempo total

Podemos determinar que la cantidad de bytes que se han transmitido es **1158 KB + 964 KB** con una duración total de tiempo de transmisión de **44,7446 y 47,1449 segundos**. Por lo tanto, obtenemos que:

$$\begin{aligned}
 \text{Ancho de banda} &= (1158 \text{ KB} + 964 \text{ KB} / 47,1449 \text{ segundos}) \\
 &= 1158 \text{ KB} / 44,7446 \text{ segundos} + 964 \text{ KB} / 47,1449 \text{ segundos} \\
 &= 25,88 \text{ KB/segundos} + 20,44 \text{ KB/segundos} \\
 &= 46,32 \text{ KB/segundos}
 \end{aligned}$$

En conclusión, podemos determinar que el ancho de banda es aproximadamente **46,32 Kb por segundo**. Podemos ver que hay una coincidencia en promedio con el Data Rate determinado en el Bottleneck de nuestra simulación.

## Anomalías

Analizando los paquetes en Wireshark, además, encontramos una anomalía que nos pareció interesante. Como se ve en la siguiente imagen, el nodo 3 (*ip 10.1.2.1*) esperó a recibir siete ACKs duplicados por parte del nodo 7 (*ip 10.2.3.1*) antes de ejecutar *Fast Retransmission*. Esta anomalía se encuentra en el enlace entre el nodo 3 con el nodo 0 (nodo 3 - interfaz 0), en el *.pcap* correspondiente.

326	7.856979	10.2.3.1	10.1.2.1	TCP	54 [TCP Dup ACK 324#1] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7856979 TSecr=7856979
327	7.869099	10.2.3.1	10.1.2.1	TCP	62 [TCP Dup ACK 324#2] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7869099 TSecr=7869099
328	7.869099	10.1.2.1	10.2.3.1	TCP	590 49153 → 3000 [ACK] Seq=94873 Ack=1 Win=131072 Len=536 TSval=7869099 TSecr=7869099
329	7.916299	10.2.3.1	10.1.2.1	TCP	62 [TCP Dup ACK 324#3] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7916299 TSecr=7916299
330	7.928099	10.2.3.1	10.1.2.1	TCP	62 [TCP Dup ACK 324#4] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7928099 TSecr=7928099
331	7.940219	10.2.3.1	10.1.2.1	TCP	70 [TCP Dup ACK 324#5] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7940219 TSecr=7940219
332	7.940219	10.1.2.1	10.2.3.1	TCP	590 49153 → 3000 [ACK] Seq=95409 Ack=1 Win=131072 Len=536 TSval=7940219 TSecr=7940219
333	7.975619	10.2.3.1	10.1.2.1	TCP	70 [TCP Dup ACK 324#6] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7975619 TSecr=7975619
334	7.987739	10.2.3.1	10.1.2.1	TCP	78 [TCP Dup ACK 324#7] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=7987739 TSecr=7987739
335	7.987739	10.1.2.1	10.2.3.1	TCP	590 [TCP Fast Retransmission] 49153 → 3000 [ACK] Seq=82545 Ack=1 Win=131072 Len=0 TSval=7987739 TSecr=7987739
336	8.023139	10.2.3.1	10.1.2.1	TCP	78 [TCP Dup ACK 324#8] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=8023139 TSecr=8023139
337	8.034939	10.2.3.1	10.1.2.1	TCP	78 [TCP Dup ACK 324#9] 3000 → 49153 [ACK] Seq=1 Ack=82545 Win=131072 Len=0 TSval=8034939 TSecr=8034939

Imagen 25: Fast Retransmit ejecutándose luego de la séptima ACK duplicada. Pcap 3-0.

Una anomalía similar podemos observar en la .pcap del enlace entre el nodo 2 con el nodo 0 (nodo 2 - interfaz 0). En este caso, podemos observar cómo el nodo 2 (ip 10.1.1.1) ejecutó *Fast Retransmission* luego del segundo ACK duplicado por parte del nodo 6 (ip 10.2.2.1), en lugar de esperar a recibir el tercer ACK.

1681	25.225658	10.2.2.1	10.1.1.1	TCP	62 [TCP Dup ACK 1678#1] 3000 → 49153 [ACK] Seq=1 Ack=562265 Win=131072 Len=0 TSval=25225658 TSecr=25225658
1682	25.225658	10.1.1.1	10.2.2.1	TCP	590 49153 → 3000 [ACK] Seq=582633 Ack=1 Win=131072 Len=536 TSval=25225658 TSecr=25225658
1683	25.237458	10.2.2.1	10.1.1.1	TCP	62 [TCP Dup ACK 1678#2] 3000 → 49153 [ACK] Seq=1 Ack=562265 Win=131072 Len=0 TSval=25237458 TSecr=25237458
1684	25.237458	10.1.1.1	10.2.2.1	TCP	590 [TCP Fast Retransmission] 49153 → 3000 [ACK] Seq=562265 Ack=1 Win=131072 Len=0 TSval=25237458 TSecr=25237458

Imágen 26: Fast Retransmit ejecutándose luego de la segunda ACK duplicada. Pcap 2-0.

## Fin de la conexión

Para finalizar, podemos observar mediante Wireshark el fin de la conexión entre los dos pares de nodos. Como se ha explicado anteriormente, el cierre de la conexión se realiza mediante el método *Four-way Handshake*.

Comenzaremos observando el fin de la conexión entre los nodos 2 y 6. Con el fin de que la vista sea más prolija, aplicamos el siguiente filtro en Wireshark: *ip.src==10.1.1.1 or ip.src==10.2.2.1*. El resultado es el siguiente:

5117	44.207057	10.1.1.1	10.2.2.1	TCP	590 49153 → 3000 [ACK] Seq=995353 Ack=1 Win=131072 Len=536 TSval=44125 TSecr=43873
5118	44.218857	10.1.1.1	10.2.2.1	TCP	518 49153 → 3000 [FIN, ACK] Seq=995889 Ack=1 Win=131072 Len=464 TSval=44125 TSecr=43873
5119	44.219577	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [ACK] Seq=1 Ack=991065 Win=131072 Len=0 TSval=44168 TSecr=43818
5120	44.243177	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [ACK] Seq=1 Ack=992137 Win=131072 Len=0 TSval=44192 TSecr=43842
5124	44.337577	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [ACK] Seq=1 Ack=993209 Win=131072 Len=0 TSval=44286 TSecr=43936
5125	44.361177	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [ACK] Seq=1 Ack=994281 Win=131072 Len=0 TSval=44310 TSecr=43960
5133	44.467377	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [ACK] Seq=1 Ack=995353 Win=131072 Len=0 TSval=44416 TSecr=44054
5141	44.536017	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [ACK] Seq=1 Ack=996354 Win=131072 Len=0 TSval=44484 TSecr=44125
5142	44.537097	10.2.2.1	10.1.1.1	TCP	54 3000 → 49153 [FIN, ACK] Seq=1 Ack=996354 Win=131072 Len=0 TSval=44484 TSecr=44125
5156	44.744617	10.1.1.1	10.2.2.1	TCP	54 49153 → 3000 [ACK] Seq=996354 Ack=2 Win=131072 Len=0 TSval=44738 TSecr=44484

Imagen 27: Fin de conexión entre emisor nodo 2 y receptor nodo 6.

Podemos observar, en el recuadro de color rojo, cómo el nodo 2 (ip 10.1.1.1) inicia el cierre de la conexión: envía al nodo 6 (ip 10.2.2.1) un segmento con el flag *FIN* encendido.

Al principio, el nodo 6 continúa enviando los *ACKs* correspondientes a segmentos que el nodo 2 le había enviado anteriormente, y que recién está recibiendo. Sin embargo, y como podemos ver en el recuadro verde, cuando recibe el segmento que indica el cierre de la conexión, responde enviando un segmento con el *ACK* correspondiente junto con flag *FIN* en 1.

Finalmente, el nodo 2 da por finalizado el método enviando otro acuse de confirmación (recuadro amarillo). Luego de esto, la conexión finaliza, por lo que ambos nodos dejan de enviar y recibir información.

Proseguiremos con el fin de la conexión entre los nodos 3 y 7. En este caso, no fue necesario aplicar un filtro en Wireshark, ya que, cuando estos nodos comenzaron el cierre de conexión, los nodos 2 y 6 ya habían dejado de transmitir información. De haber sido necesario un

filtro, el mismo habría sido de la forma: *ip.src==10.1.2.1 or ip.src==10.2.3.1*. Podemos, entonces, ver el cierre de conexión claramente:

5263	46.629417	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[ACK] Seq=1 Ack=818473 Win=131072 Len=0 TSval=46578 TSecr=46281
5264	46.635257	10.1.2.1	10.2.3.1	TCP	78 49153 → 3000	[FIN, ACK] Seq=822761 Ack=1 Win=131072 Len=24 TSval=46606 TSecr=46354
5265	46.653017	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[ACK] Seq=1 Ack=819545 Win=131072 Len=0 TSval=46601 TSecr=46304
5266	46.836577	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[ACK] Seq=1 Ack=820617 Win=131072 Len=0 TSval=46785 TSecr=46500
5267	46.919177	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[ACK] Seq=1 Ack=821689 Win=131072 Len=0 TSval=46868 TSecr=46582
5268	46.942777	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[ACK] Seq=1 Ack=822761 Win=131072 Len=0 TSval=46891 TSecr=46606
5269	46.943857	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[ACK] Seq=1 Ack=822786 Win=131072 Len=0 TSval=46892 TSecr=46606
5270	46.944937	10.2.3.1	10.1.2.1	TCP	54 3000 → 49153	[FIN, ACK] Seq=1 Ack=822786 Win=131072 Len=0 TSval=46892 TSecr=46606
5271	47.146017	10.1.2.1	10.2.3.1	TCP	54 49153 → 3000	[ACK] Seq=822786 Ack=2 Win=131072 Len=0 TSval=47146 TSecr=46892

Imagen 28: Fin de conexión entre emisor nodo 3 y receptor nodo 7. Pcap 0-0.

Podemos observar, en el recuadro de color rojo, cómo el nodo 3 (*ip 10.1.2.1*) inicia el cierre de la conexión enviando al nodo 7 (*ip 10.2.3.1*) el segmento con el flag *FIN* encendido.

Al principio, el nodo 7 continúa enviando los acuses de confirmación correspondientes a segmentos que el nodo 3 había enviado anteriormente. Sin embargo, y como podemos ver en el recuadro verde, cuando recibe el segmento que indica el cierre de la conexión, responde enviando un segmento con el *ACK* correspondiente junto con flag *FIN* en 1.

Finalmente, el nodo 3 da por finalizada la conexión enviando otro acuse de confirmación (recuadro amarillo). Luego de esto, la conexión entre ambos nodos finaliza, por lo que la simulación llega a su fin.

En Wireshark, podemos comprobar que los flags *FIN* están encendidos en los paquetes que aparecen en los recuadros rojo y verde, tal y como se observa en la imagen 29.

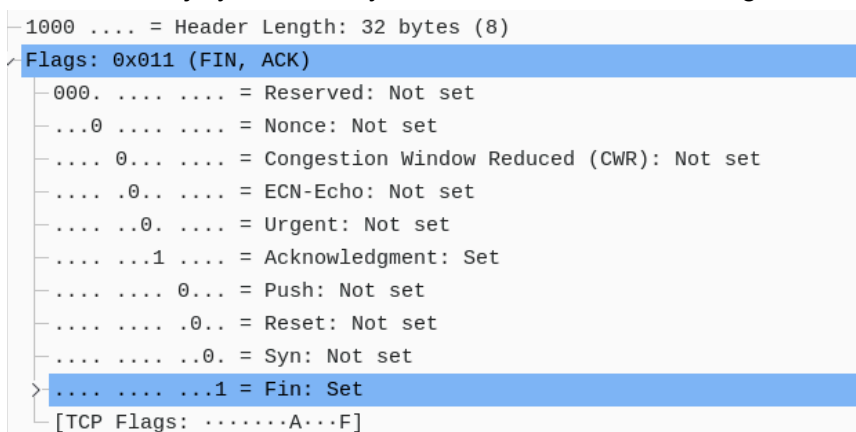


Imagen 29: flags de los paquetes del fin de transmisión.

Algo importante a remarcar en los dos cierres de conexión, es que ambos nodos receptores (es decir, los nodos 6 y 7) enviaron el acuse de confirmación y el flag *FIN* encendido en un mismo segmento. Por lo tanto, el fin de la conexión se dio en sólo tres pasos, en lugar de cuatro. Por lo tanto, no se realizó un método *Four-way handshake*, sino un *Three-way handshake*.

## Explicación código

Una vez desarrollado el punto uno, definiendo la topología sobre la cual se trabajará, se procederá a explicar el código utilizado en NS-3. En esta sección, se detalla el proceso para ejecutar los archivos de código C++ proporcionados, utilizando este entorno de simulación.

## Preparación del Entorno

Los archivos con extensión .cc deben copiarse en la siguiente ruta dentro del directorio de trabajo de NS-3:

```
/USER/ns3/ns3-allinone-3.31/ns3-3.31/scratch
```

## Ejecución de los Archivos

1. Abrir una terminal y navegar a la carpeta que contiene el directorio scratch, específicamente en:  
`/USER/ns3/ns3-allinone-3.31/ns3-3.31`

2. Ejecutar el siguiente comando para compilar y correr los archivos C++:  
`./waf --run scratch/<nombre_archivo>`

Donde <nombre\_archivo> debe ser reemplazado por el nombre del archivo que se desea ejecutar, por ejemplo:

```
./waf --run scratch/dumbbell-topology-tcp-wireshark
```

Por cada ejecución de este comando estamos compilando y corriendo un ejecutable de nuestro archivo C++. Los ejecutables se encuentran en la ruta del directorio `/ns3/ns3-allinone-3.31/ns3-3.31/build/scratch`

3. Si la ejecución del comando fue exitosa, debemos ver en nuestra terminal el mensaje "build finished successfully".

## Descripción del Código

En esta sección, se proporciona una descripción detallada de las funciones implementadas y del funcionamiento general del código. Estas funciones son comunes para los escenarios:

- a) 4 nodos TCP, los dos emisores y los dos receptores.
- b) 4 nodos TCP, los dos emisores y los dos receptores.
- 2 nodos UDP, 1 emisor y 1 receptor

## Funciones

En este apartado, se detallan las funciones principales del código, describiendo su propósito, parámetros de entrada, y salida.

### CreatePointToPointHelper

```
PointToPointHelper CreatePointToPointHelper (  
    const std::string& queue, const std::string& dataRate, const  
    std::string& delay, const std::string& maxSize  
)
```

Descripción: Crea y configura un objeto *PointToPointHelper*.

Parámetros:

*queue*: Tipo de cola a utilizar. Debe ser un string que especifique el tipo de cola.

*dataRate*: La tasa de datos del enlace punto a punto. Debe ser un string que especifique la tasa de datos, por ejemplo, "50KBps".

*delay*: El retraso en el enlace punto a punto. Debe ser un string que especifique el retraso, por ejemplo, "2ms".

*maxSize*: Tamaño máximo de la cola. Debe ser un string que especifique el tamaño máximo, por ejemplo, "100p" para 100 paquetes.

Returns:

Un objeto *PointToPointHelper* configurado con los atributos especificados.

**SetupDumbbellTopology**

```
PointToPointDumbbellHelper SetupDumbbellTopology (  
    uint32_t CLIENT_NODES_COUNT,  
    uint32_t SERVER_NODES_COUNT,  
    uint32_t port,  
    PointToPointHelper &dumbbellHelper, PointToPointHelper  
    &bottleneckHelper, InternetStackHelper &stack  
)
```

Descripción: Crea y configura un objeto *PointToPointDumbbellHelper*.

Parámetros:

*CLIENT\_NODES\_COUNT*: Número de nodos cliente en el lado izquierdo de la topología Dumbbell.

*SERVER\_NODES\_COUNT*: Número de nodos servidor en el lado derecho de la topología Dumbbell.

*port*: Puerto utilizado para las comunicaciones.

*dumbbellHelper*: Objeto *PointToPointHelper* para los enlaces entre los nodos y los routers.

*bottleneckHelper*: Objeto *PointToPointHelper* para representar el bottleneck entre los routers.

*stack*: Objeto *InternetStackHelper* util para representar el stack

Returns:

Un objeto *PointToPointDumbbellHelper* configurado con los atributos especificados.

**ConfigureApplicationDuration**

```
void ConfigureApplicationDuration(  
    ApplicationContainer &apps,  
    Time startTime,  
    Time stopTime  
)
```

Descripción: Configura el tiempo de inicio y final de un *ApplicationContainer*.

Parámetros:

*apps*: Contenedor de aplicaciones *ApplicationContainer* al que se definirán sus tiempos de inicio y detenimiento.

*startTime*: Tiempo de inicio para todas las aplicaciones en el contenedor.

*stopTime*: Tiempo de stop para todas las aplicaciones en el contenedor.

### **InstallApplications**

```
void InstallApplications(  
    uint32_t sourceNode, uint32_t destinationNode, uint32_t  
    port, ApplicationContainer &clientApps,  
    ApplicationContainer &serverApps,  
    PointToPointDumbbellHelper &dumbbell,  
    StringValue socket)
```

**Descripción:** Instala aplicaciones cliente y servidor en los nodos especificados de una topología Dumbbell.

#### **Parámetros:**

*sourceNode*: Índice del nodo cliente en el lado izquierdo de la topología Dumbbell.

*destinationNode*: Índice del nodo servidor en el lado derecho de la topología Dumbbell.

*port*: Puerto utilizado para las comunicaciones entre el cliente y el servidor.

*clientApps*: Contenedor ApplicationContainer donde se agregarán las aplicaciones cliente.

*serverApps*: Contenedor ApplicationContainer donde se agregarán las aplicaciones servidor.

*dumbbell*: Objeto PointToPointDumbbellHelper que representa la topología Dumbbell.

*socket*: Tipo de socket utilizado por las aplicaciones, por ejemplo TcpSocket.

### **Flujo de Ejecución**

En esta sección se explica cómo interactúan las diferentes funciones y se describe el flujo general de ejecución del programa.

#### **- Inicialización de variables**

```
int main (int argc, char *argv[]){  
  
    ApplicationContainer clientApps;  
    ApplicationContainer serverApps;  
  
    StringValue TCP_SOCKET ("ns3::TcpSocketFactory"); // socket tcp  
    StringValue UDP_SOCKET ("ns3::UDPSocketFactory"); // socket udp
```



```
uint32_t NODE_0 = 0;
uint32_t NODE_1 = 1;
uint32_t NODE_2 = 2;

InternetStackHelper stack;
```

Definimos las variables *clientApps* y *serverApps*, que son contenedores que almacenarán las aplicaciones cliente y servidor, respectivamente. Asimismo, declaramos las variables *NODE\_0*, *NODE\_1* y *NODE\_2* para identificar los nodos en la topología.

*InternetStackHelper* stack: Es un helper utilizado para instalar la pila de protocolos de Internet en cada nodo. Este helper se utiliza como parámetro al configurar la topología dumbbell.

#### - Configuración del protocolo TCP

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

A su vez, configuramos el tipo de protocolo TCP a usar en la simulación. En este caso, se ha configurado para usar *TcpNewReno*.

#### - Creación de Helpers Punto a Punto

```
PointToPointHelper dumbbellHelper = CreatePointToPointHelper(
    "ns3::DropTailQueue",
    "100KBps",
    "100ms",
    "10p");

PointToPointHelper bottleneckHelper = CreatePointToPointHelper(
    "ns3::DropTailQueue",
    "50KBps",
    "50ms",
    "10p");
```

Definimos dos *PointToPointHelpers* a través de la función *CreatePointToPointHelper()*.

- **dumbbellHelper**  
Representa el enlace desde los nodos emisores y receptores hasta los nodos intermedios. Está configurado con una cola de tipo *ns3::DropTailQueue*, que representa una cola FIFO (First In, First Out). Tiene un tamaño máximo de 10 paquetes (10p), una tasa de transferencia de datos de 100KBps (100 Kilobytes por segundo) y un delay de 100ms.
- **bottleneckHelper**  
Representa el canal intermedio, donde se produce la saturación. La diferencia con el helper anterior es que este está configurado con una tasa de transferencia de datos de 50KBps (50 Kilobytes por segundo) y un delay de 50ms.

#### - Creación de Helper de Topología Dumbbell

```
PointToPointDumbbellHelper dumbbell = SetupDumbbellTopology(
    CLIENT_NODES_COUNT,
    SERVER_NODES_COUNT,
```

```
port,
dumbbell Helper,
bottleneckHelper, stack);
```

Creamos un *PointToPointDumbbellHelper* a través de la función *SetupDumbbellTopology()*, utilizando los siguientes métodos de ns-3:

```
PointToPointDumbbellHelper dumbbell(
    CLIENT_NODES_COUNT, dumbbellHelper,
    SERVER_NODES_COUNT, dumbbellHelper,
    bottleneckHelper);
dumbbell.InstallStack(stack);
Ipv4AddressHelper leftIP("10.1.1.0", "255.255.255.0");
Ipv4AddressHelper rightIP("10.2.1.0", "255.255.255.0");
Ipv4AddressHelper routersIP("10.3.1.0", "255.255.255.0");

dumbbell.AssignIpv4Addresses(leftIP, rightIP, routersIP);
Ipv4GlobalRoutingHelper::PopulateRoutingTables();
```

Utilizamos el helper *PointToPointDumbbellHelper* para facilitar la creación de la topología dumbbell. Para ello, pasamos como parámetros la cantidad de nodos a la izquierda (CLIENT\_NODES\_COUNT), la cantidad de nodos a la derecha (SERVER\_NODES\_COUNT), junto a sus respectivos enlaces punto a punto (utilizando el mismo helper para ambos lados), y finalmente añadimos el enlace intermedio. De esta forma, definimos nuestro objeto que representa la topología requerida.

Llamando al método *dumbbell.InstallStack(stack)*, instalamos la pila de protocolos de Internet en todos los nodos (clientes, servidores y routers) utilizando el helper *InternetStackHelper*.

Generamos direcciones IPv4 a través del *Ipv4AddressHelper* para el lado izquierdo, derecho y los nodos centrales del dumbbell, comenzando con los valores 10.1.1.0, 10.2.1.0 y 10.3.1.0 respectivamente. Posteriormente, asignamos estas direcciones y poblamos las tablas de ruteo correspondientes.

### - Configuración de Aplicaciones

```
//ONOFFAPPLICATION nodo 0 --> nodo 1
InstallApplications(NODE_0, NODE_1, port, clientApps, serverApps, dumbbell, TCP_SOCKET);
//ONOFFAPPLICATION 2 nodo 1--> nodo 2
InstallApplications(NODE_1, NODE_2, port, clientApps, serverApps, dumbbell, TCP_SOCKET);

ConfigureApplicationDuration(clientApps, Seconds(0.0), Seconds(40.0));
ConfigureApplicationDuration(serverApps, Seconds(0.0), Seconds(100.0));
```

En esta sección, se genera la configuración adicional de los nodos indicando el protocolo a utilizar, así como el nodo de origen y de destino. Esta configuración particular es para el primer escenario con 4 nodos TCP: dos en el lado izquierdo y dos en el lado derecho. Para el segundo escenario, solo sería necesario agregar un nodo UDP en el lado izquierdo que apunte a su par en el lado derecho, de la siguiente forma:

```
InstallApplications(NODE_2, NODE_0, port, clientApps, serverApps, dumbbell, UDP_SOCKET);
```

Instalamos las aplicaciones en los *ApplicationContainer* definidos anteriormente invocando la función *SetupDumbbellTopology()*, la cual hace uso de las siguientes funciones de ns-3:

```
OnOffHelper clientHelper(socket.Get(), Address());
clientHelper.SetAttribute("OnTime",StringValue("ns3::ConstantRandomVariable[Constant=1000.0]"));
clientHelper.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0]"));

AddressValue remoteAddress(InetSocketAddress(dumbbell.GetRightIpv4Address(destinationNode),port));
clientHelper.SetAttribute("Remote", remoteAddress);
clientApps.Add(clientHelper.Install(dumbbell.GetLeft(sourceNode)));
PacketSinkHelper server(
    socket.Get(),
    InetSocketAddress(dumbbell.GetRightIpv4Address(destinationNode),
        port));

serverApps.Add(server.Install(dumbbell.GetRight(destinationNode)));
```

### Configuración Cliente

*OnOffHelper clientHelper*: Se crea un helper *OnOff* para facilitar la creación de una aplicación *OnOff*, con un tiempo de encendido constante de 1000 segundos y un tiempo de apagado constante de 0 segundos.

*AddressValue remoteAddress*: Se configura la dirección IPv4 y el puerto del nodo de destino, y se asigna esta dirección remota configurada a la aplicación *OnOff* cliente a través del atributo "Remote".

*clientApps.Add*: Se añade la aplicación cliente configurada al contenedor *clientApps* y se instala en el nodo de origen especificado por *sourceNode*.

### Configuración Servidor

*PacketSinkHelper* funciona como receptor de los paquetes enviados por la aplicación cliente.

*serverApps.Add*: Se añade la aplicación servidor configurada al contenedor *serverApps* y se instala en el nodo de destino especificado por *destinationNode*.

Finalmente, invocamos la función *ConfigureApplicationDuration(app, startTime, stopTime)* para definir el tiempo de inicio y detención de las aplicaciones. Esta función utiliza los métodos *Start()* y *Stop()* propios de ns-3.

### Tracers

Para la recopilación de información (los archivos *.data*) de la simulación hicimos uso del sistema de Tracers de ns3, y terminamos creando dos tipos de funciones para manejar el flujo de estos datos: *Tracer* y *Trace*.

Las funciones *Trace* son las encargadas de crear el flujo por donde se enviarán los datos para posteriormente ser almacenados en archivos *.data*, además de configurar el tipo de *Tracer* y de cuál nodo/interfaz obtener la información.

En la simulación los tipos de datos que podemos extraer, como la ventana de congestión, los rto, rtt, etc. (que estuvimos usando para el análisis) los podemos encontrar como atributos de

ciertas clases. En nuestro caso particular, los datos que fuimos extrayendo y que acabamos de mencionar son atributos de la clase de ns3 llamada `TcpSocketBase`. Internamente ns3 tiene el sistema de nodos e interfaces que vimos anteriormente cuando explicamos los archivos `.pcap`. Con esa misma idea de indexación, podemos extraer entonces el valor que tiene un atributo en un momento dado de la simulación, y los Tracers son los encargados de obtener esta información.

Los Tracers entonces son una especie de función *callback* que son llamadas cuando un atributo cambia de valor. Cuando esto ocurre, el Tracer recibe el valor anterior al cambio y el nuevo valor del atributo para realizar acciones. En nuestro caso, la acción que realizamos (y para lo cual estamos usando los Tracers) es la de ir guardando estos valores en una tabla tiempo/valor, que indica que valor tenía el atributo que estamos *traceando* en cierto momento de la simulación. De esta forma lo que obtenemos es la variación de ese atributo a lo largo de toda la simulación.

Teniendo todo esto en cuenta, lo que necesitamos entonces para poder *tracear* cierto atributo es: saber qué atributo queremos tracear y saber de qué nodo/interfaz lo vamos a obtener.

De forma general lo que necesitaremos a nivel código es lo siguiente:

**static Ptr<OutputStreamWrapper> atributoStreamNodo:** Este objeto es el encargado de escribir y generar el archivo de datos con la información que vayamos recopilando. **atributo** sería cambiado por el nombre real del atributo y **Nodo** sería el número de nodo del cual estamos hablando (en nuestro caso varía entre 1 y 7).

**static void AtributoTracerNodo\_Interfaz(oldval, newval):** Este método sería nuestro Tracer. **Atributo** sería reemplazado por el atributo a tracear, **Nodo** e **Interfaz** por el número de nodo e interfaz respectivamente. Luego `oldval` y `newval` son el valor viejo y nuevo que tomó el atributo. En nuestro caso estos valores se los pasamos como una entrada al Stream para que los vaya escribiendo en el archivo de datos.

**static void TraceAtributoNodo\_Interfaz(nombre\_archivo):** Este método sería nuestro Trace, que se encarga de instanciar a nuestro `atributoStreamNodo`, define el tipo de Tracer a utilizar y de qué nodo/interfaz/atributo sacar los datos.

A modo de ejemplo, veamos cómo es esta estructura para tracear la ventana de congestión entre el nodo 2 y la interfaz 0. En una primera instancia tendríamos nuestro Stream:

```
static Ptr<OutputStreamWrapper> congestionWindowStream2;
```

El Tracer:

```
//Tracers para recopilar informacion de los cambios en los atributos de la conexión TCP de
static void
CongestionWindowTracer2_0 (uint32_t oldval, uint32_t newval)
{
    //Como empezamos luego de la simulacion (0.1 segundos luego) primero registramos en el s
    if (fCongestionWindow2)
    {
        saveTCPData(congestionWindowStream2, 0.0, oldval);
        fCongestionWindow2 = false;
    }
    else
    {
        saveTCPData(congestionWindowStream2, Simulator::Now ().GetSeconds (), newval);
    }
    congestionWindowValue2 = newval;
    //SSThresh is set according to the CongestionWindow. When the CongestionWindow changes,
    if (!fSSThresh2)
    {
        saveTCPData(ssThreshStream2, Simulator::Now ().GetSeconds (), ssThreshValue2);
    }
}
```

Aquí nos detendremos y veremos un poco qué está haciendo nuestro Tracer. En una primera instancia tenemos un flag para ver si es la primera “iteración” (o mejor dicho si es la primera vez que llamamos a la función) de la simulación. Esto debido a que, como veremos más adelante, el mecanismo de *tracing* comienza en el momento 0.01 de la simulación y como la diferencia es insignificante, “empezamos” en 0 para así también estandarizar el tiempo de comienzo de cada Tracer de atributos que tengamos en nuestra simulación.

Luego vemos que se llama al método `saveTCPData`. Este método toma nuestro Stream, un momento dado y el valor que queramos guardar para la recopilación de información, internamente podemos ver que lo único que hace es meter los datos en el Stream de la forma “tiempo -espacio- valor”. Este formato es necesario para que luego gnuplot pueda leer correctamente los datos y generar los gráficos.

```
static void
saveTCPData(Ptr<OutputStreamWrapper> stream, float time, uint32_t value)
{
    *stream->GetStream () << time << " " << value << std::endl;
}
```

Esta estructura es muy similar o incluso idéntica en todos los Tracers de nuestra simulación.

Por último tenemos al Trace:

```
static void
TraceCongestionWindow2_0 (std::string file_name)
{
    AsciiTraceHelper ascii;
    congestionWindowStream2 = ascii.CreateFileStream (file_name.c_str ());
    Config::ConnectWithoutContext ("/NodeList/2/$ns3::TcpL4Protocol/SocketList/0/CongestionWindow", MakeCallback (&CongestionWindowTracer2_0));
}
```

Aquí podemos ver cómo se inicializa al Stream con el nombre del archivo que le hayamos pasado, pero lo más interesante es la última línea del método. En resumidas cuentas, lo que quiere decir la última línea es lo siguiente: cada vez que el atributo `CongestionWindow` para el nodo 2 e interfaz 0 cambie, realizará una llamada al método `CongestionWindowTracer2_0` con el valor viejo y nuevo del atributo. En este paso hay que tener en cuenta que no podemos pasar cualquier cosa como argumento de `MakeCallback`, justamente `CongestionWindowTracer2_0` esta siguiendo una especie de interfaz que está definida en el archivo cabecera de `TcpSocketBase`. Como habíamos dicho en un principio, este atributo le pertenece a este último objeto, y si vamos a la clase de `ns3`, podemos ver lo siguiente:

```

C tcp-socket-base.h  tcp-socket-base.cc  dumbbell-topology-tcp-wireshark.cc
src > internet > model > tcp-socket-base.cc > {} ns3 > GetTypeId(void)
66 namespace ns3 {
73 TcpSocketBase::GetTypeId (void)
178     "ns3::TcpSocketState::EcnStatesTracedValueCallback")
179     .AddTraceSource ("AdvWND",
180                     "Advertised Window Size",
181                     MakeTraceSourceAccessor (&TcpSocketBase::m_advWnd),
182                     "ns3::TracedValueCallback::UInt32")
183     .AddTraceSource ("RWND",
184                     "Remote side's flow control window",
185                     MakeTraceSourceAccessor (&TcpSocketBase::m_rWnd),
186                     "ns3::TracedValueCallback::UInt32")
187     .AddTraceSource ("BytesInFlight",
188                     "Socket estimation of bytes in flight",
189                     MakeTraceSourceAccessor (&TcpSocketBase::m_bytesInFlightTrace),
190                     "ns3::TracedValueCallback::UInt32")
191     .AddTraceSource ("HighestRxSequence",
192                     "Highest sequence number received from peer",
193                     MakeTraceSourceAccessor (&TcpSocketBase::m_highRxMark),
194                     "ns3::TracedValueCallback::SequenceNumber32")
195     .AddTraceSource ("HighestRxAck",
196                     "Highest ack received from peer",
197                     MakeTraceSourceAccessor (&TcpSocketBase::m_highRxAckMark),
198                     "ns3::TracedValueCallback::SequenceNumber32")
199     .AddTraceSource ("CongestionWindow",
200                     "The TCP connection's congestion window",
201                     MakeTraceSourceAccessor (&TcpSocketBase::m_cWndTrace),
202                     "ns3::TracedValueCallback::UInt32")
203     .AddTraceSource ("CongestionWindowInflated",

```

Aquí podemos ver el atributo y con qué TracedCallback está asociado. En este caso, vemos que está asociado al `m_cWndTrace`, luego si nos dirigimos a la definición vemos lo siguiente:

```

C tcp-socket-base.h .../ns3  tcp-socket-base.cc  tcp-socket-base.h .../model  dumbbell-topology-tcp-wireshark.cc
src > internet > model > tcp-socket-base.h > {} ns3 > TcpSocketBase > m_cWndTrace
36 namespace ns3 {
217 class TcpSocketBase : public TcpSocket
312 void SetRetxThresh (uint32_t retxThresh);
313
314 /**
315  * \brief Get the retransmission threshold (dup ack threshold for a fast retransmit)
316  * \return the threshold
317  */
318 uint32_t GetRetxThresh (void) const { return m_retxThresh; }
319
320 /**
321  * \brief Callback pointer for cWnd trace chaining
322  */
323 TracedCallback<uint32_t, uint32_t> m_cWndTrace;
324
325 /**
326  * \brief Callback pointer for cWndInfl trace chaining
327  */
328 TracedCallback<uint32_t, uint32_t> m_cWndInflTrace;
329

```

Como se puede apreciar, `m_cWndTrace` es un `TracedCallback<uint32_t, uint32_t>`, que es justo los dos parámetros con los que definimos a `CongestionWindowTracer2_0`. Esto es muy importante a tener en cuenta a la hora de crear nuestra estructura para los Tracers, ya que tenemos que seguir esta interfaz para que funcione correctamente.

Una vez que tenemos todo lo anterior implementado, comenzar la recopilación de información es sencillo. Para esto lo que hacemos es llamar a `Simulator::Schedule` al cual le vamos a indicar cuándo queremos que comience a funcionar luego de iniciada la simulación y a qué método (con sus respectivos parámetros) llamar. En nuestro caso, eso se da de esta forma:

```

Simulator::Schedule (Seconds (0.001), &TraceCongestionWindow2_0, prefix_file_name + "congestion_window2_0.data");

```

### 3. Segunda Parte: Pruebas con dos emisores TCP y uno UDP

Los archivos .pcap correspondientes a la ejecución del script dumbbell-topology-tcp-udp-wireshark.cc se encontrarán dentro de la carpeta punto3 del repositorio.

#### Análisis de la simulación con UDP y TCP

Inicialmente, se puede observar como UDP aprovecha el ancho de banda en comparación con TCP. En la imagen 30, la gráfica de UDP (en verde) consistentemente, durante toda la simulación, utiliza el ancho de banda que no está siendo utilizado por TCP. Esto es evidente al inicio de la comunicación, donde UDP alcanza valores muy altos de bits/s mientras TCP realiza el three-way handshake. Estos valores disminuyen gradualmente a medida que TCP empieza a utilizar más ancho de banda de la red. Además, se observan en los segundos 18, 23 y 37 aproximadamente, donde se nota que tan pronto como la gráfica de TCP desciende, UDP aumenta en respuesta.

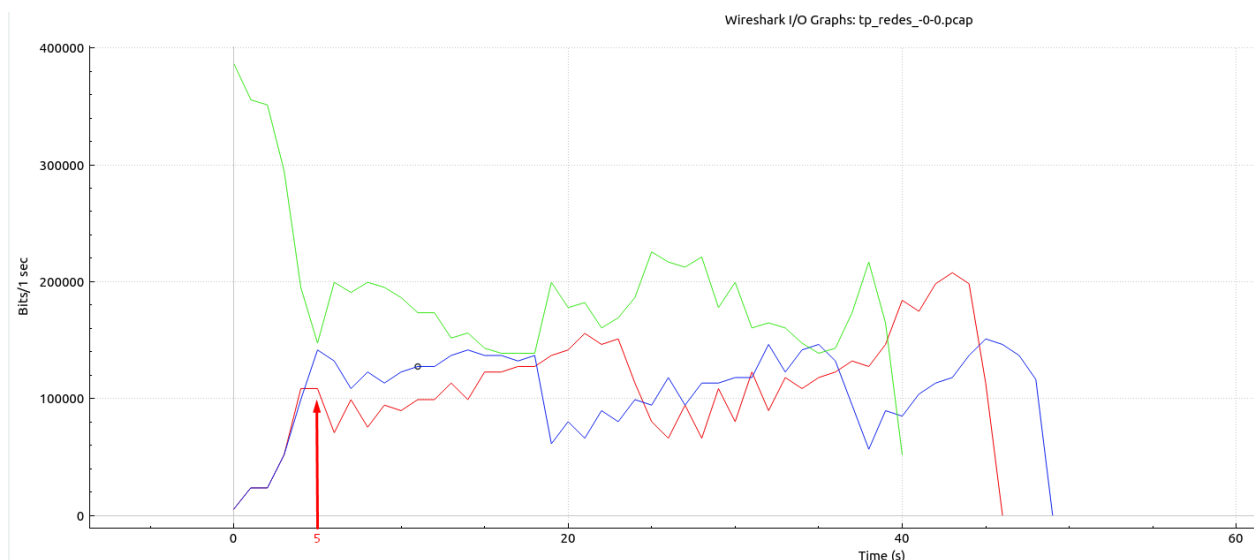


Imagen 30: Gráfica de la simulación UDP y TCP. Se observa marcado en el gráfico el momento aproximado en el que ocurre la congestión de la red

Otro dato a destacar es el momento en donde terminan las gráficas. En nuestra simulación, se ha configurado que el envío de información por parte de los emisores termine en el segundo 40. En la imagen 30, se puede observar que UDP finaliza su transmisión aproximadamente en el segundo 40, mientras que para TCP esto ocurre alrededor de los segundos 47 y 49. Esto sucede a pesar de que ambos dejan de transmitir en el segundo 40. La diferencia radica en el funcionamiento de ambos protocolos: en UDP, el emisor envía información continuamente, sin considerar si esta llega a destino o no, debido a que no está orientado a conexión. Por su lado, TCP utiliza mecanismos con acuses de recibo, en donde espera por las respuestas para seguir enviando información.

Ethernet	IPv4 · 3	IPv6	TCP · 2	UDP · 1							
Address A ▾	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.1.1.1	10.2.2.1	1.679	677 k	1.097	645 k	582	32 k	0.000000	46.0254	112 k	
10.1.2.1	10.2.3.1	1.740	697 k	1.127	663 k	613	34 k	0.001160	49.1491	107 k	
10.1.3.1	10.2.1.1	1.791	970 k	1.791	970 k	0	0	0.013031	40.2328	193 k	

Imagen 31: Conversación total de la red visto en Wireshark. Se puede observar los tiempos de finalización para cada nodo en la columna *Duration*

Continuando con el análisis, en las imágenes 32 y 33 se muestra la ventana de congestión para el nodo 2 y 3 respectivamente, en la interfaz 0. Al centrarnos en la imagen 32, se puede observar las etapas del protocolo TCP, sección explicada anteriormente. En la imagen, podemos visualizar la etapa de **Slow Start** al principio de la simulación, que culmina en la congestión de la red alrededor del segundo 5.5 aproximadamente, dando paso a la etapa de **Fast Retransmission**. Posteriormente, desemboca en un **Fast Recovery** seguido de la etapa de **Congestion Avoidance** hasta el segundo 25 aproximadamente, momento en el cual se produce otro **Fast Retransmission** debido a la pérdida de paquetes. Esto condice con lo discutido anteriormente sobre UDP y el ancho de banda. Si retomamos a la imagen 30, notaremos que en ese momento, como ya se mencionó, disminuye la cantidad de bits transmitidos por el nodo 2 (nodo configurado con TCP), y en su lugar aumenta para el nodo 4 (nodo configurado con UDP).

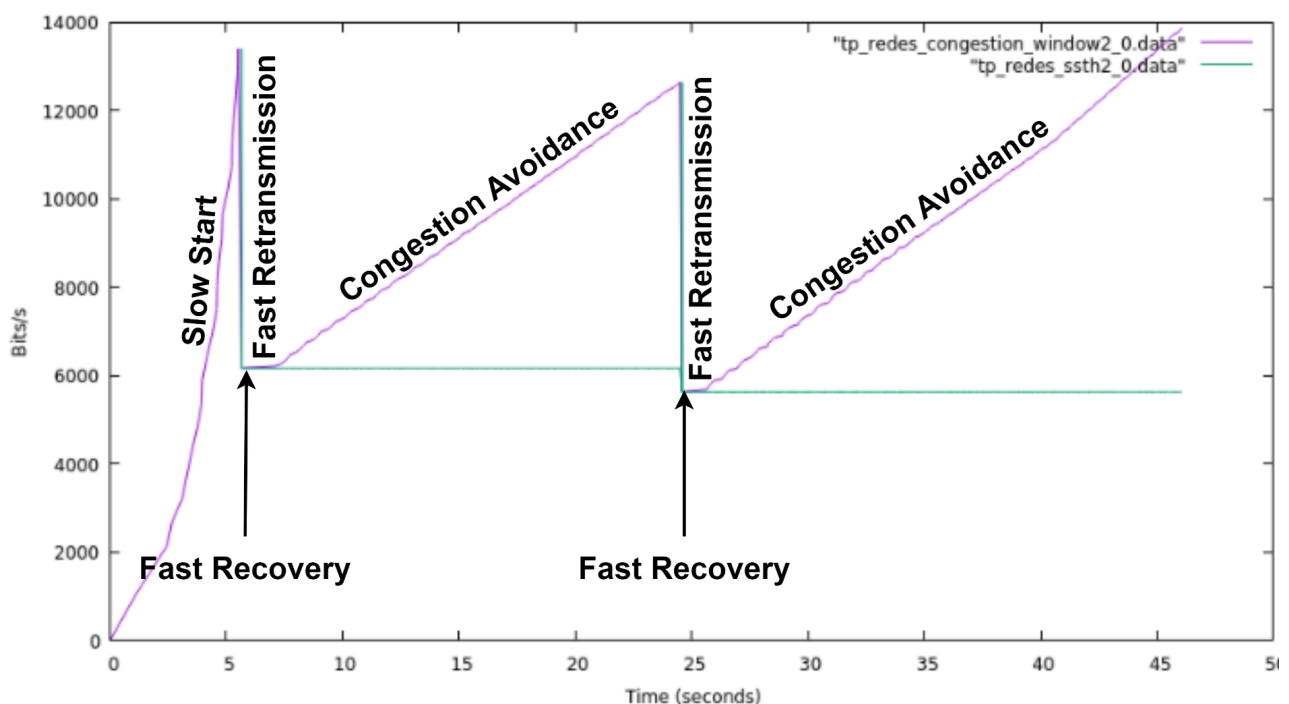


Imagen 32: Gráfica de la ventana de congestión del nodo 2 para la simulación con UDP



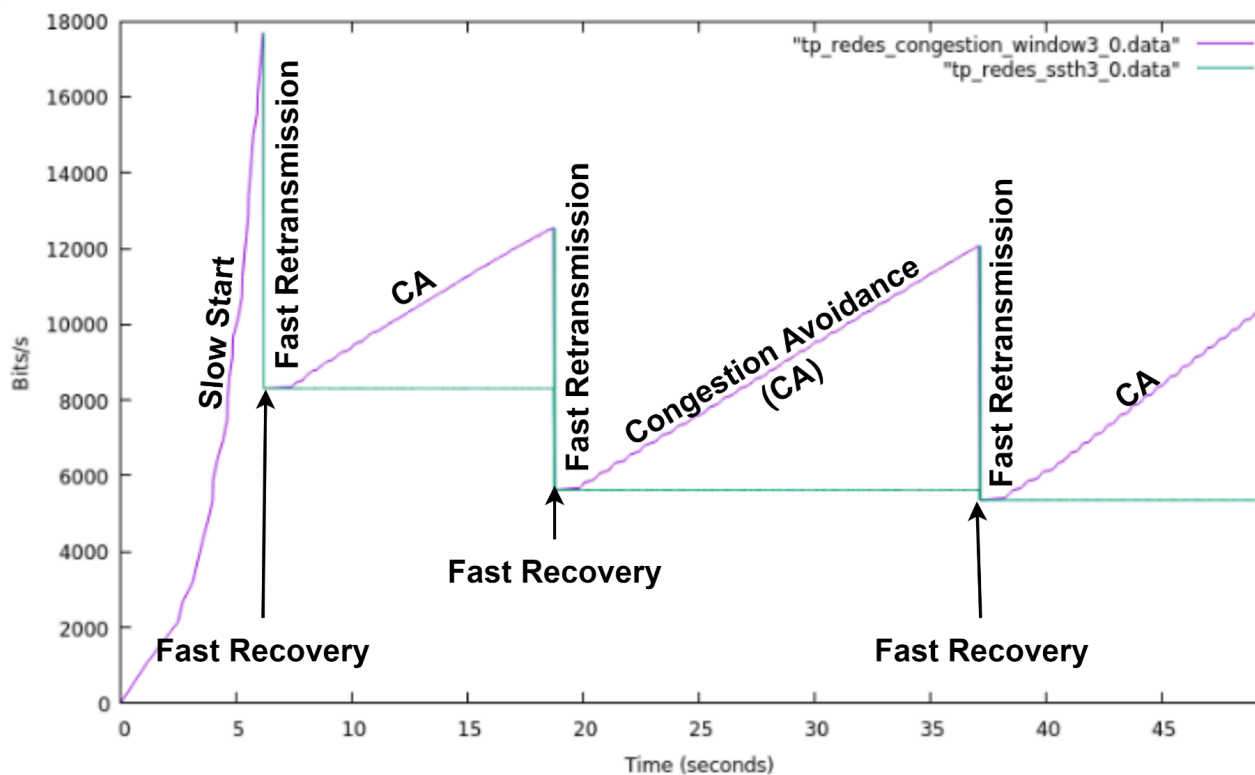


Imagen 33: Gráfica de la ventana de congestión del nodo 3 para la simulación con UDP

## Conclusión

Para concluir, luego de haber realizado el análisis sobre la simulación de una red en la cual se utilizan protocolos TCP y UDP, es evidente que ambos se comportan de forma muy distinta: en términos de rendimiento y latencia, UDP tiende a ser más rápido que TCP debido a su enfoque en la velocidad de transmisión y la falta de handshakes que pueden resultar complejos. Sin embargo, esta ventaja de velocidad puede llevar a la pérdida de datos, los cuales no se pueden recuperar. Por otro lado, TCP se enfoca en la confiabilidad y garantiza la entrega de datos, lo que puede resultar en una latencia ligeramente mayor pero una transferencia de datos más segura.

Coincidimos en que, aunque por momentos nos resultó difícil, indagar e investigar los protocolos resultó muy enriquecedor. Nos pareció especialmente interesante observar la convivencia de diferentes protocolos en transmisión simultánea y también cómo, en algunos casos, la práctica no refleja al 100% lo planteado en la teoría.

Finalmente, consideramos que la experiencia de investigar y simular la topología Dumbbell, junto con todos los escenarios realizados durante el desarrollo de este trabajo, fue positiva. Esto porque pudimos poner en práctica y entender mejor lo que habíamos visto en la parte teórica de la materia.

## Bibliografía

(2. Animation — Model Library, n.d.)  
[NetAnim - Animation - Model Library](#)

(23.6. Queues — Model Library, n.d.)  
[23.6. Queues — Model Library. \(n.d.\). ns-3.](#)  
(Alsuheim, 2020)  
[Alsuheim, A. \(2020, January 1\). ns3 Network Simulator - Using TraceSources. YouTube.](#)  
(Alsuheim, 2023)  
[NS3 Network Simulator, Youtube.](#)  
(Cantera Álvarez, 2023)  
[Estudio de técnicas de control de congestión con diferentes tipos de tráfico de datos en NS3. Wikipedia.](#)  
(Congestión Y Rendimiento De TCP, n.d.)  
[Congestión y rendimiento de TCP](#)  
(CSE 222A – Lecture 14: Congestion Control, n.d.)  
[CSE 222A – Lecture 14: Congestion Control. \(n.d.\). UCSD CSE.](#)  
(Dogan, 2024)  
[Dogan, N. \(2024, January 2\). Measure bandwidth using Wireshark \[Practical Examples\]. GoLinuxCloud.](#)  
(Donnelly, 2022)  
[Donnelly, J. \(2022, February 18\). ¿Qué es el TCP? MASV.](#)  
(Escalante, 2023)  
[Handshaking de 3 vías \(3 way handshaking\) - abcXperts](#)  
(Gonz, 2014)  
[TCP \(Transmission Control Protocol\)](#)  
(Jain, 2021)  
[TCP 3-Way Handshake Process - GeeksforGeeks](#)  
(Jain, 2022)  
[Slow Start Restart Algorithm For Congestion Control - GeeksforGeeks](#)  
(Merritt, 2014)  
[Gnuplot Documentation](#)  
(NS3 Tutorials for Beginners, 2014)  
[NS3 Tutorials for Beginners. Youtube.](#)  
(Ns-3 Documentation, n.d.)  
[ns-3 Documentation. \(n.d.\). ns-3.](#)  
(Ns-3 Tutorial: 3.4 Running a Script, n.d.)  
[ns-3 tutorial: 3.4 Running a Script. \(n.d.\). ns-3.](#)  
(Sellaheewa, n.d.)  
[Sellaheewa, R. \(n.d.\). Creating Graphs in Gnuplot. SEPnet.](#)  
(Plotting Data With Gnuplot, n.d.)  
[Plotting Data with gnuplot. \(n.d.\). Plotting Data with gnuplot.](#)  
(Protocolo De Control De Transmisión, n.d.)  
[Protocolo de control de transmisión - Wikipedia, la enciclopedia libre](#)  
(Trace Sources, n.d.)  
[Trace Sources. \(n.d.\). Wikipedia.](#)  
(TCP Header - Definition, Diagram and Its Format, 2024)  
[TCP Header - Definition, Diagram and its Format. \(2024, January 6\). PyNet Labs.](#)  
(Universidad Politécnica De Cartagena, n.d.)  
[Simulación de los mecanismos de control de congestión en TCP/IP](#)  
(Wireshark · Documentation, n.d.)  
[Wireshark Documentation](#)