

INFORME

TP2: ANÁLISIS DE REDES

Sistemas Operativos y Redes 2

Integrantes del grupo:

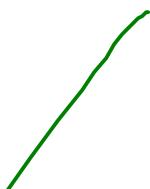
• Luis Curto	luisenriquecurto@gmail.com	31232760
• Sebastian Pintos	spintos100@gmail.com	41688321
• Matias Morales	matias.agustin.morales@gmail.com	42956612
• Vanesa Vera	vanesa.vera1127@gmail.com	43089000

Docentes:

- Alexis Tcach
- Alan Echabbarri

Índice

Introducción.....	2
Herramientas Utilizadas.....	2
NS3.....	2
NetAnim.....	2
WireShark.....	3
TcpTrace.....	3
Gnuplot.....	3
Descripción de la red generada.....	4
Pruebas realizadas.....	8
Primer escenario (TCP).....	9
Velocidad de Transferencia.....	9
Análisis de los gráficos.....	10
Etapas del protocolo TCP.....	15
Segundo escenario (TCP - UDP).....	23
Ancho de Banda.....	25
Análisis de los paquetes incluyendo nodos UDP.....	27
Problemas Encontrados.....	33
Conclusiones.....	33



Introducción

El objetivo principal del trabajo práctico es realizar una configuración de red utilizando la topología *Dumbbell*. Este escenario consta de tres emisores *on/off application*, tres receptores y dos nodos intermedios, donde los emisores se conectarán a través de los nodos para enviar datos a los destinos finales. A través de pruebas y análisis, se busca comprender el comportamiento de los protocolos **UDP** y **TCP**, así como las interacciones entre ellos.

Durante este informe se desarrollará como fue la resolución del trabajo, para ello, en primer lugar mencionaremos y daremos una breve explicación de las herramientas utilizadas. Luego, daremos un descripción de la red generada donde mencionaremos el desarrollo de la misma. Una vez explicada la red haremos una división para desarrollar las dos pruebas de emisores, una solamente con emisores **TCP** y otra con **TCP-UDP**. Cada una de ellas contará con un análisis de su red.

Al finalizar los análisis mencionaremos los problemas con los que nos encontramos durante el desarrollo del trabajo práctico y daremos una conclusión para finalizar con el informe.

Herramientas Utilizadas

Para lograr el objetivo del trabajo práctico utilizamos el sistema operativo **Linux** junto con el simulador de redes **NS3**, **netAnim**, **Wireshark**, **TcpTrace** y **Gnuplot**. A continuación se detallarán más las herramientas utilizadas.

NS3

NS3 es un simulador de redes basado en eventos discretos. Permite simular el comportamiento de los nodos de red y analizar el rendimiento de los protocolos de comunicación, o sea, permite la simulación de redes cableadas e inalámbricas y ofrece herramientas de análisis para medir métricas de rendimiento. En este caso, utilizamos la versión que se encontraba instalada en la máquina virtual ofrecida por la materia.

NetAnim

NetAnim es una herramienta de visualización y animación utilizada en conjunto con el simulador **NS3**. Proporciona una interfaz gráfica para representar visualmente la ejecución de las simulaciones de redes realizadas en **NS3**. Permite visualizar la topología de red creada en **NS3**, mostrando los nodos de red, enlaces y sus propiedades.

WireShark

Wireshark es una herramienta de análisis de redes de código abierto y ampliamente utilizada. Permite capturar y analizar el tráfico de red en tiempo real o a partir de archivos de captura previamente almacenados. Permite examinar y comprender el contenido de los paquetes de datos que se transmiten a través de una red. Además, mediante una interfaz gráfica que muestra los paquetes capturados y permite filtrar entre paquetes.

TcpTrace

Tcptrace es una herramienta open-source para analizar archivos de capturas de redes del tipo **.pcap**. Acepta como entrada archivos producidos por programas de capturas de paquetes.

Gnuplot

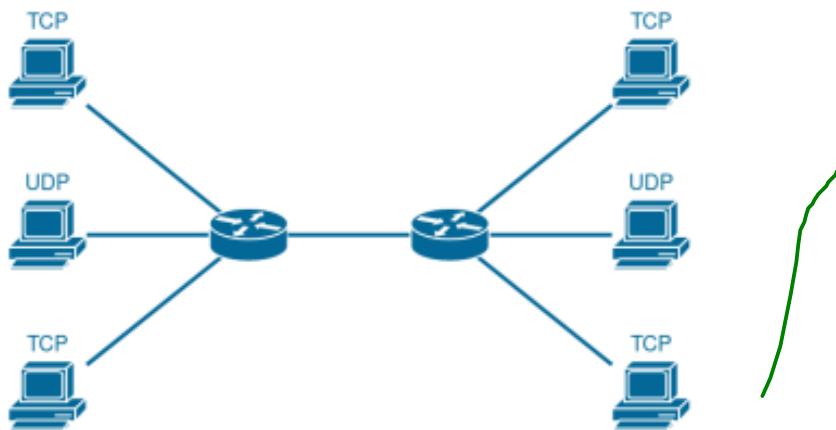
Gnuplot es un programa de trazado de gráficos en dos y tres dimensiones que permite generar gráficas y visualizar datos de manera interactiva en una variedad de formatos.

Bien organizada la introducción

Descripción de la red generada

Para llevar a cabo la simulación, primero se debió realizar la configuración del entorno de ejecución en **NS3**.

Nuestra configuración cuenta con 3 emisores *on/off application*, 3 receptores y dos nodos intermedios, se conectarán los 3 emisores a un nodo, luego este a otro y finalmente éste a los 3 destinos finales. Uno de los emisores es **UDP**. los otros 2 **TCP** y siempre usará conexiones cableadas, todo esto siguiendo la topología *Dumbbell*.



Dumbbell topology

La topología *Dumbbell* (mancuerna) es un diseño comúnmente utilizado en redes de computadoras para representar una conexión entre dos redes distintas a través de un enlace central.

A continuación describiremos la estructura principal del código para lograr el funcionamiento correspondiente al de la topología solicitada.

Como primer paso se crean los contenedores de los nodos para los *gateways*, emisores y receptores en una simulación de red. Los *gateways* son los nodos que actúan como puntos de entrada/salida entre diferentes redes, mientras que los emisores generan flujos de datos y los receptores son los destinos de esos flujos.

```
//Se crean los nodos (emitters = emisor, receivers = receptores)
NS_LOG_UNCOND ("Create nodes");
NodeContainer gateways;
gateways.Create (2);
NodeContainer emitters;
emitters.Create (num_flows);
NodeContainer receivers;
receivers.Create (num_flows);
```

Creación de los nodos

Luego se configuran los enlaces derecho, izquierdo y central de la topología dumbbell utilizando la clase *PointToPointHelper*. Los enlaces se configuran con sus respectivas tasas de transferencia de datos, retardos y modelos de error de recepción. Luego, se instala el enlace central entre los dos *gateways*. Anteriormente definimos las variables con los valores para el ancho de banda y el *delay*.

```
//Creamos los canales (PointToPoint)
NS_LOG_UNCOND ("Create channels");
PointToPointHelper rightLink;
rightLink.SetDeviceAttribute("DataRate", StringValue("50KBps"));
rightLink.SetChannelAttribute("Delay", StringValue("1ms"));

PointToPointHelper leftLink;
leftLink.SetDeviceAttribute ("DataRate", StringValue ("50KBps"));
leftLink.SetChannelAttribute ("Delay", StringValue ("1ms"));

PointToPointHelper centerLink;
centerLink.SetDeviceAttribute ("DataRate", StringValue ("50MBps"));
centerLink.SetChannelAttribute ("Delay", StringValue ("10ms"));
centerLink.Install (gateways.Get (0), gateways.Get (1));
```

Enlaces entre nodos

Una vez que tenemos los enlaces, se instala la pila de protocolos de Internet en todos los nodos de la simulación, lo que es necesario para permitir la comunicación y el enrutamiento de datos en la red simulada.

Ojo Este código no es el que pusieron en el código. Este está mal, el otro bien. OJO

```
//Instalo la pila de protocolos de Internet
NS_LOG_UNCOND ("Install internet protocol stack");
InternetStackHelper stack;
stack.InstallAll ();
```

Instalación de pila de protocolos

Las siguientes líneas de código establecen la dirección IP base y la máscara de subred utilizadas para asignar direcciones IP a los nodos de la simulación. En nuestro caso definimos la dirección base 10.0.0.0 y la máscara 255.255.255.0.

```
//Establecemos la dirección IP base
NS_LOG_UNCOND ("Stable IP address");
Ipv4AddressHelper address;
address.SetBase ("10.0.0.0", "255.255.255.0");
```

Definimos dirección base

Ahora creamos las aplicaciones de envío de datos *on/off* en los nodos emisores y aplicaciones de recepción de datos en los nodos receptores utilizando sockets **UDP** y **TCP**, respectivamente. Las aplicaciones se configuran con direcciones IP y puertos específicos y se establecen los tiempos de inicio y finalización de las aplicaciones.

```
for (uint16_t i = 0; i < emitters.GetN (); i++)
{
    //Se crean las aplicaciones UDP en nodos 1 de los emitters y receivers
    if(i == 1){
        OnOffHelper ftp ("ns3::UdpSocketFactory", InetSocketAddress (sink_interfaces.GetAddress (i, 0), port));
    }
    //Se crean las aplicaciones TCP en nodos 0 y 2 de los emitters y receivers
    else{
        OnOffHelper ftp ("ns3::TcpSocketFactory", InetSocketAddress (sink_interfaces.GetAddress (i, 0), port));

        ApplicationContainer sourceApp = ftp.Install (emitters.Get (i));
        sourceApp.Start (Seconds (start_time * i));
        sourceApp.Stop (Seconds (stop_time - 3));

        ApplicationContainer sinkApp = sinkHelper.Install (receivers.Get (i));
        sinkApp.Start (Seconds (start_time * i));
        sinkApp.Stop (Seconds (stop_time));
    }
}
```

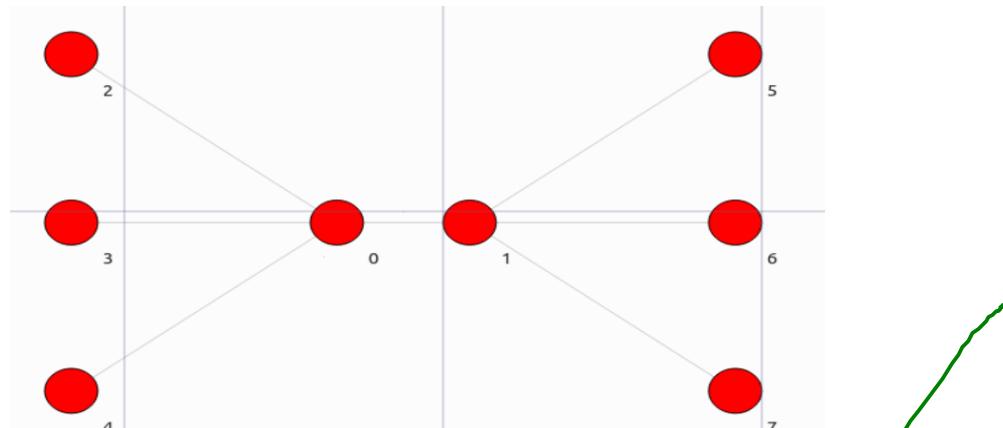
Instalación de las aplicaciones on/off y configuración de las mismas

Después, se utiliza el nombre de archivo "tráfico" para la captura de paquetes, y se establece el segundo argumento como true para habilitar el modo promiscuo. El modo promiscuo permite capturar todos los paquetes que pasan por las interfaces de red, incluso aquellos que no están destinados a la máquina en la que se ejecuta la simulación.

```
//Captura el tráfico en un archivo pcap
NS_LOG_UNCOND ("Capturing traffic");
if (pcap)
{
    rigthLink.EnablePcapAll ("tráfico", true);
}
```

Captura del tráfico en un archivo pcap

Por otro lado, se utiliza la clase *AnimationInterface* para crear una animación de la simulación y establecer las posiciones constantes de los nodos en la misma. La estructura de animación generada se muestra a continuación en NetAnim.



Dumbbell topology en NetAnim

```
//Crea el archivo animacion.xml para vizualizar en NetAnim
NS_LOG_UNCOND ("Create the animacion.xml file for NetAnim");
AnimationInterface anim ("animacion.xml");
anim.SetConstantPosition(emitters.Get(0), 20, 10);
anim.SetConstantPosition(emitters.Get(1), 20, 25);
anim.SetConstantPosition(emitters.Get(2), 20, 40);
anim.SetConstantPosition(gateways.Get(0), 40, 25);
anim.SetConstantPosition(gateways.Get(1), 50, 25);
anim.SetConstantPosition(receivers.Get(0), 70, 10);
anim.SetConstantPosition(receivers.Get(1), 70, 25);
anim.SetConstantPosition(receivers.Get(2), 70, 40);
anim.EnablePacketMetadata(true);
```

Generación y configuración de la animación para NetAnim

Finalmente se configura la simulación para que después de 150 segundos se detenga y luego la ejecuta hasta que se complete o hasta que se alcance el tiempo especificado (150s).

¿Por qué definieron este tiempo?

Muy bien como aplicaron tracers y las estrategias en NS3 !

```

double duration = 150
NS_LOG_UNCOND ("Start simulation");
Simulator::Stop (Seconds(duration));
Simulator::Run ();
NS_LOG_UNCOND ("Stop simulation");

```

Simulación

Pruebas realizadas

En este apartado definiremos y desarrollaremos los dos escenarios en los que se realizarán las pruebas para analizar sus resultados. El primer escenario será solamente con los dos emisores **TCP** transmitiendo y el segundo escenario pondremos a transmitir todos los emisores, es decir los dos **TCP** y el **UDP**. Con esto se busca poder ver las diferencias entre los protocolos y cómo afectan a la red los mismos.

Para entender mejor los resultados obtenidos y el análisis de los gráficos, daremos algunas características principales de cada protocolo para tener en cuenta:

Característica	TCP	UDP
Orientación de conexión	Orientado a la conexión	Sin conexión
Fiabilidad	Entrega confiable de datos	No hay garantía de entrega
Control de congestión	Control de congestión integrado	No hay control de congestión
Sobrecarga y eficiencia	Mayor sobrecarga	Menor sobrecarga

Ahora que ya tenemos en cuenta esto, vamos a destacar los parámetros y valores que se utilizaron en la configuración para ambos escenarios, ya que será de gran influencia para el análisis:

Parámetros	Valor
Ancho de banda en nodos intermedios	50KBps
Delay de los nodos intermedios	10ms
Tiempo de simulación	150s
Ancho de banda de nodos emisores	50MBps
Delay de los nodos emisores	1ms

Si cambian 2 parámetros (BW y delay), no van a saber bien cuál es el motivo...
Eso no se debe hacer nunca

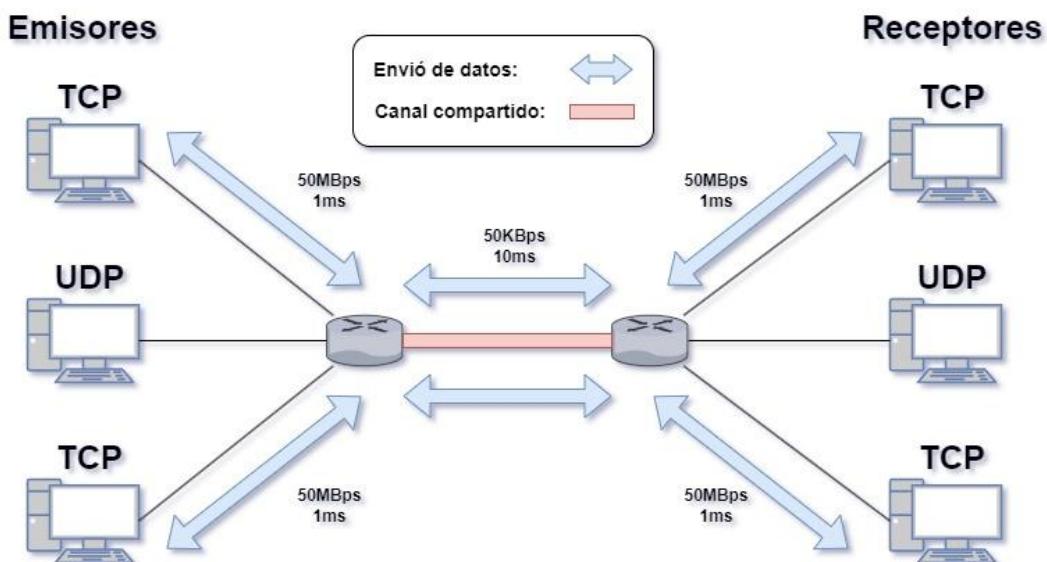
Primer escenario (TCP)

En esta sección solo veremos las pruebas realizadas con los nodos TCP emitiendo.

Velocidad de Transferencia

Para realizar las pruebas con solamente los emisores **TCP** en primer lugar debemos dejar sin transmitir al emisor **UDP**.

Luego, los nodos emisores **TCP** deben generar una gran cantidad de envío de datos para que el canal se sature y provocar la pérdida de paquetes junto con fluctuaciones en el rendimiento del protocolo **TCP**, ocasionando la ejecución del mecanismo de control de congestión. En la siguiente imagen mostramos el flujo de los datos a través del canal compartido, donde solo transmiten los emisores **TCP**.



Representación gráfica del envío de datos por el canal compartido

En la siguiente imagen se pueden observar las conversaciones de los dos emisores **TCP** junto con su información correspondiente a la cantidad de paquetes enviados, su tamaño y la duración de la conversación. Estos datos los obtuvimos gracias a la herramienta *conversations* que nos ofrece **Wireshark**.

Podemos observar como el emisor con la Ip: 10.0.2.1 se comunica con el receptor cuya Ip es igual a 10.0.3.2, también la comunicación entre el emisor con Ip: 10.0.6.1 con el receptor de Ip:10.0.7.2.

Ethernet	IPv4 · 2	IPv6	TCP · 2	UDP							
Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	End	
10.0.2.1	10.0.3.2	7.959	3.230 k	5.232	3.082 k	2.727	148 k	0.000000	149.9875		
10.0.6.1	10.0.7.2	11.289	4.571 k	7.385	4.353 k	3.904	217 k	0.200000	149.7737		

Conversaciones desde Wireshark

Para calcular la velocidad de transferencia tomamos los *bytes* y la duración indicada para cada uno, luego sumamos esa cantidad y obtenemos la velocidad de transferencia en *bytes*.

Cálculos y datos

	Bytes	Duración	Cálculo	Total
Comunicación 1	3,230 k	149.9875 seg	3,230 k / 149.9875 s	21,536.20Bps
Comunicación 2	4,571 k	149.7737 seg	4,571 k / 149.7737 s	30,520.13Bps
Comunicación total	-	-	-	52,056.33Bps

Las tablas también deberían tener la bajada

Finalmente vemos que la velocidad de transferencia del canal cuando se satura es de 52,056.33Bps bytes por segundo, lo cual si pasamos a kilobytes es 50.83KBps. Es mayor al ancho de banda proporcionado (50KB), esto se debe a que lo brindado por wireshark es un promedio del ancho de banda y puede verse afectado por la congestión de la red.

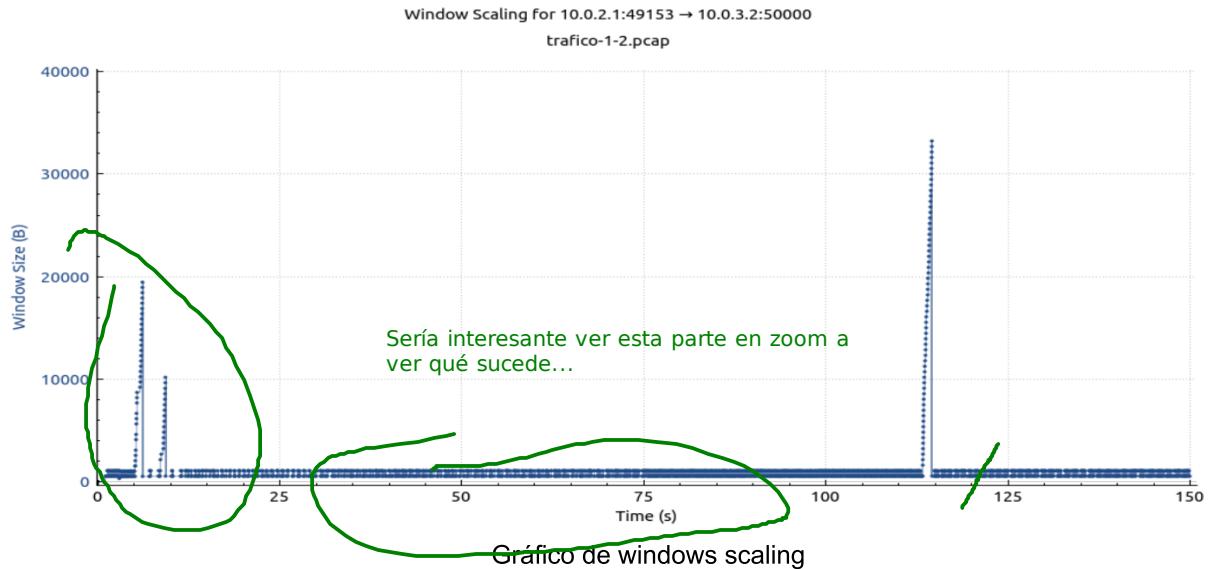
Análisis de los gráficos

A continuación se mostrarán y describirán los gráficos obtenidos para la visualización de datos más relevantes sobre lo que sucede en nuestra red previamente configurada. Cada gráfico contará con una explicación de qué características de la red podemos obtener a través de él.

En primer lugar analizaremos una aproximación del tamaño de la cola de recepción mediante el comportamiento del tamaño de la ventana de recepción. Para entender mejor el análisis, antes de pasar al gráfico definiremos los siguientes concepto:

- Cola de recepción: es una estructura de datos utilizada en los dispositivos de red para almacenar temporalmente los paquetes recibidos antes de su procesamiento.
- Ventana de recepción: es el espacio disponible en el búfer del receptor para recibir datos enviados por el emisor.

Una vez dicho esto, mediante el gráfico **Window Scaling** proporcionado por **wireshark**, explicaremos el comportamiento de la red y cómo va cambiando el tamaño de la cola de recepción durante el mismo.



En el gráfico podemos ver como tamaño de la ventana comenzó a fluctuar de manera regular luego de que inició el envío de datos, esto indica que la cola de recepción experimentó cierta congestión pero pudo manejar la data de manera adecuada. Eso se mantuvo hasta el momento en el que la ventana produjo un pico, o sea que aumentó significativamente su tamaño permitiendo más flujo de datos entre el emisor y el receptor, cuando se produce esto generalmente la cola de recepción también incrementa temporalmente, en ese momento manejó una mayor cantidad de datos. *¿y por qué sucede esto ?*

Luego, el pico sufrió una caída en el tamaño de la ventana, esto significa que la cola de recepción pudo estar saturada y fue descargando paquetes debido a la falta de capacidad.

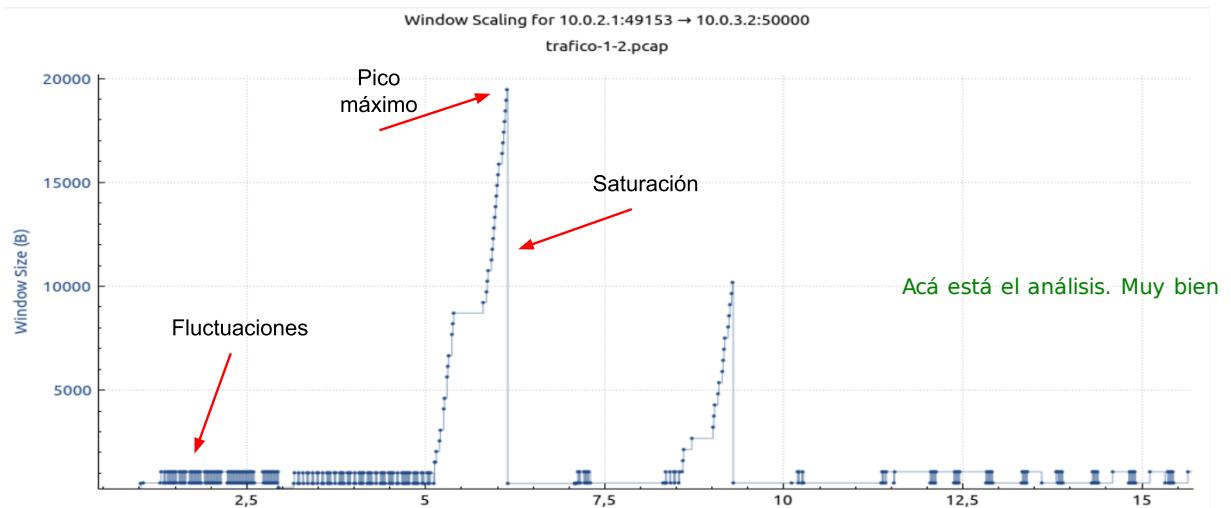


Gráfico de windows scaling con las indicaciones

Ahora vamos a analizar características importantes sobre los paquetes de la red, como el orden, la identificación de la pérdida y la duplicación de ellos, además, mostraremos un análisis sobre la congestión y el flujo de los datos.

Comenzaremos utilizando el gráfico **Sequence Numbers (tcptrace)** de wireshark que se muestra a continuación.

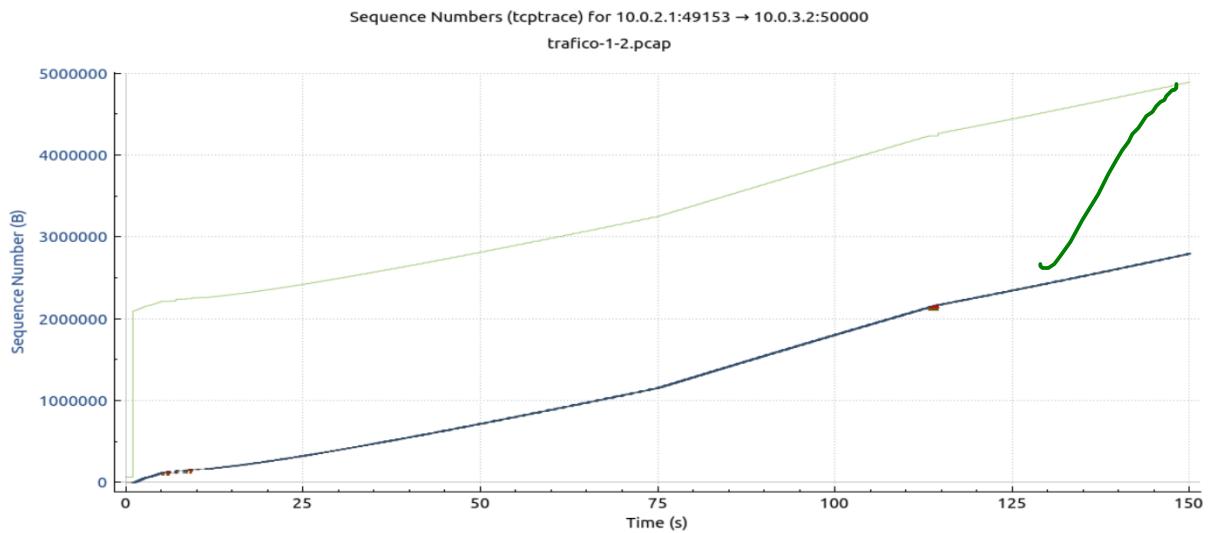
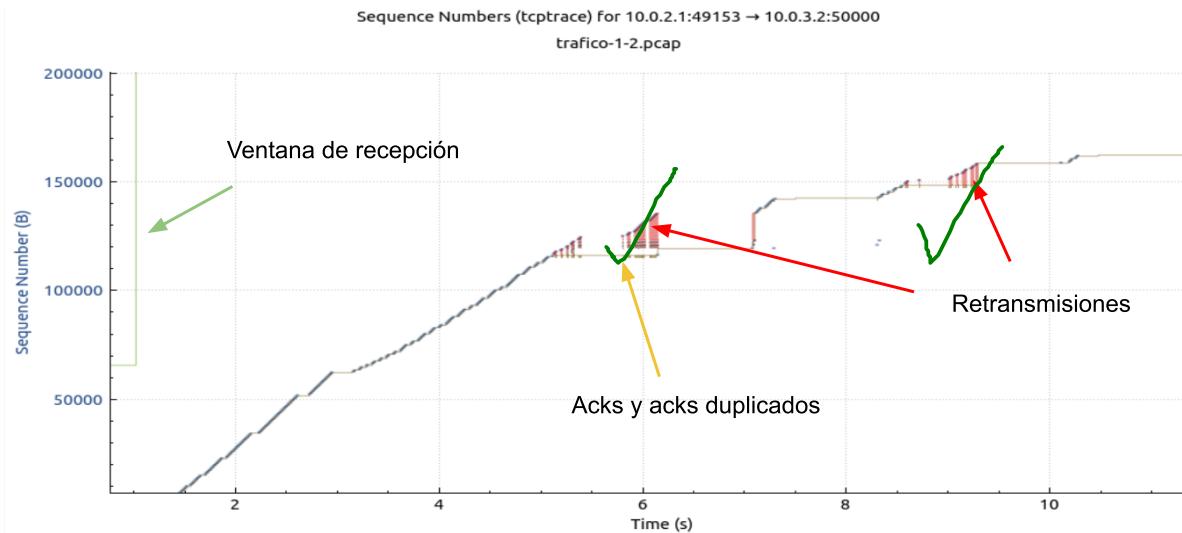


Gráfico Sequence Numbers tcptrace

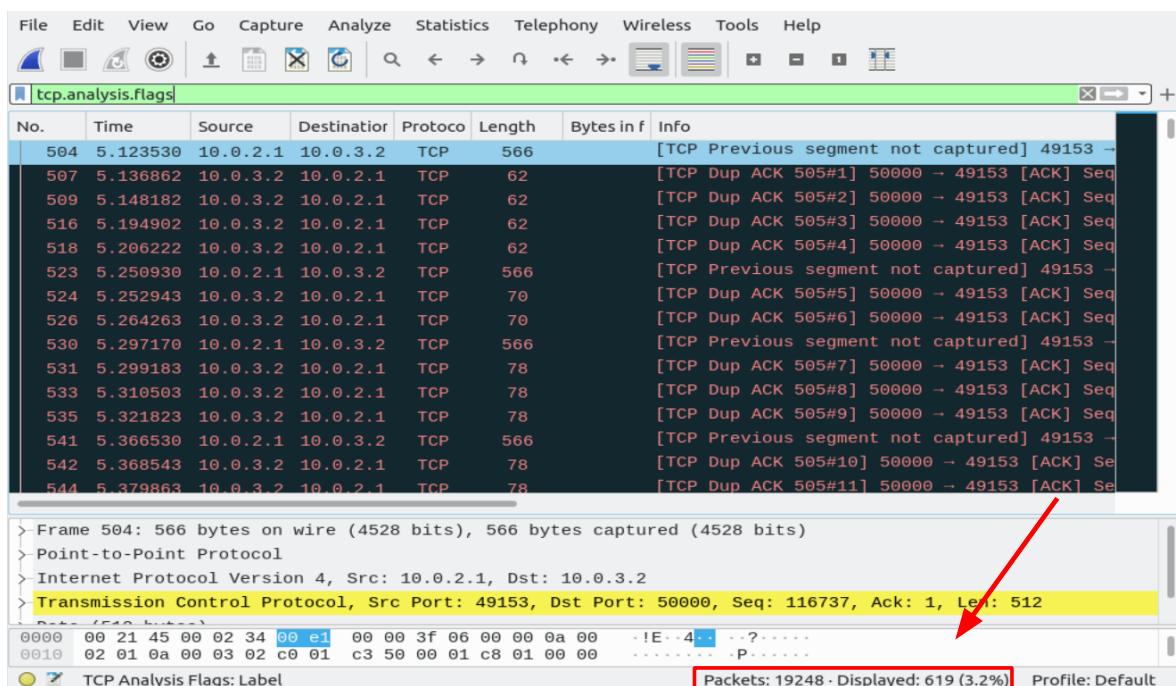
El eje X es el tiempo y el eje Y son los números de secuencia de TCP. Los números de secuencia son representativos de los bytes enviados. Lo ideal sería ver una línea suave hacia arriba y hacia la derecha. Cuanto más inclinada sea la línea, mayor será el rendimiento. Los pequeños rayos negros representan segmentos de datos TCP. La línea gris debajo de esos son los ACK del receptor. La distancia entre los ACK y los datos TCP en un momento dado representa los bytes en vuelo. La línea superior representa la ventana de recepción calculada del cliente.

Como podemos observar, alrededor de los 5 segundos comenzó a presentar anomalías con el envío de paquetes, esto indica que hubo pérdidas o retransmisiones de ellos, que resulta en problemas con la congestión de la red y por ende con el rendimiento de la misma. A continuación se muestra la parte del gráfico en la que se realiza la mayor pérdida de paquetes por la congestión de la red.



Zoom gráfico secuence numbers tcptrace

Por otro lado en la pantalla inicial de wireshark se puede utilizar un filtro para obtener todos los paquetes que tuvieron problemas durante la transmisión. En la siguiente imagen podemos observar la lista con los paquetes afectados en wireshark que equivale a un 3.2% de los paquetes totales durante la simulación de la red.



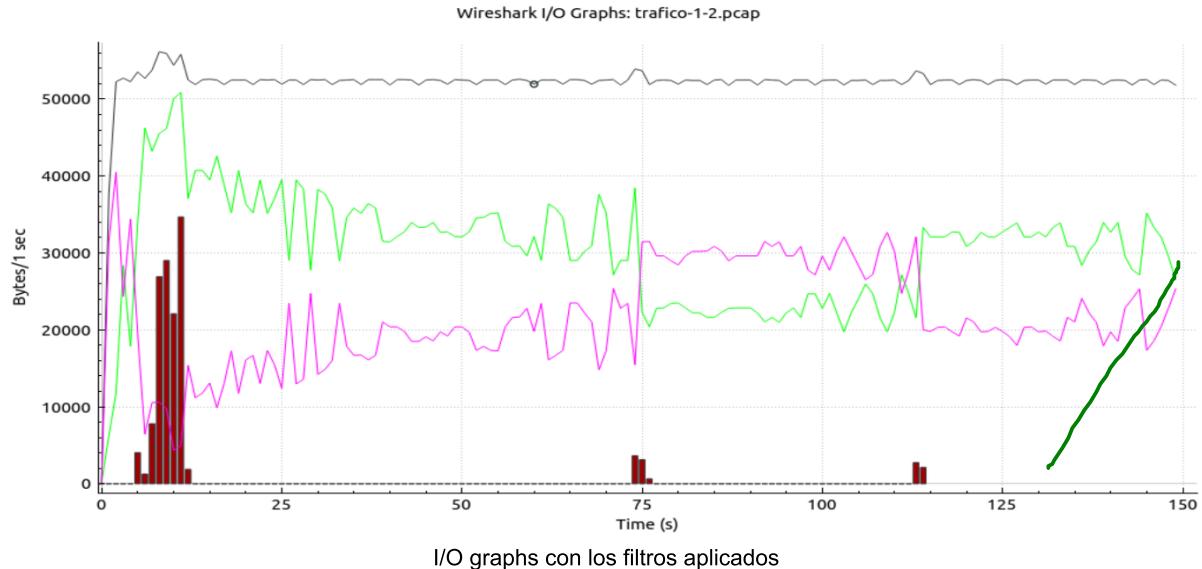
Lista de paquetes afectados

Por último, podemos mostrar la transferencia de los paquetes a lo largo de la simulación con el gráfico I/O Graph que nos proporciona Wireshark. En este gráfico añadimos filtros personalizados para poder observar el flujo de paquetes de cada emisor por separado. Para ello filtramos según la IP 10.0.2.1 que corresponde al primer emisor con una linea **rosa** y la IP 10.0.6.1 que corresponde al segundo emisor con una línea de color **verde**. Tambien colocamos unos filtros para los paquetes perdidos, el cual se puede observar con unas barras de color **rojo**. Luego se puede observar una línea **negra** que indica el ancho de banda a lo largo de la simulación.

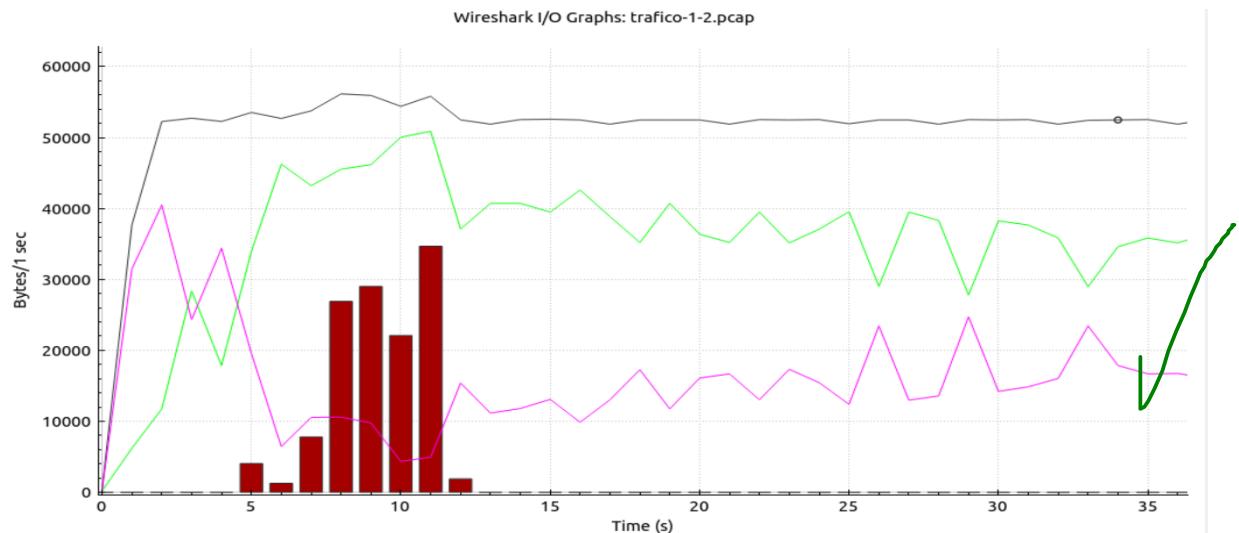
Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period
<input type="checkbox"/>	All Packets		■	Line	Bytes		None
<input type="checkbox"/>	TCP Errors	tcp.analysis.flags	■	Bar	Bytes		None
<input type="checkbox"/>	TCP 10.0.6.1	ip.addr == 10.0.6.1	■	Line	Bytes		None
<input type="checkbox"/>	TCP 10.0.2.1	ip.addr == 10.0.2.1	■	Line	Bytes		None

Filtros del I/O graphs

En el eje vertical observaremos los bytes que son transmitidos por segundo, mientras que el eje horizontal se observa el tiempo transcurrido en segundos. Por lo tanto el gráfico se ve la siguiente manera:



I/O graphs con los filtros aplicados



Se suele poner toda la descripción

Podemos observar como las barras **rojas** crecen alrededor de los 5 segundos, lo que indica el momento en que hubo una gran cantidad de pérdida de paquetes. Además, en ese momento se ve como el segundo emisor (línea **verde**) intenta enviar una gran cantidad de datos, lo que provoca la saturación. Por lo que el ancho de banda sube y tiene una caída debido a que los emisores superaron el ancho de banda disponible. Una vez que pasa la saturación, el canal ya se estabiliza.

Como se observa en la figura anterior, una de las conexiones TCP en primer momento envía más datos que la otra, esto luego se va regulando hasta que las dos se ajustan en cantidad de datos siendo más equitativo el envío.

También vemos como el **ancho de banda** siempre está cerca de los 51200 bytes/s (equivale a los 50KB) lo que previamente se estableció en la configuración de nuestra red.

Mediante estos tres gráficos mostramos algunas propiedades de la red generada como el ancho de banda, el tamaño próximo de la cola de recepción, la ventana TCP, información sobre los paquetes transmitidos y el flujo de los datos. En la próxima sección del informe mencionaremos las etapas del protocolo TCP por las que pasa la misma durante toda su simulación.

Etapas del protocolo TCP

El protocolo TCP (Transmission Control Protocol) consta de varias etapas que se llevan a cabo durante el establecimiento, mantenimiento y finalización de una conexión. Las etapas son las siguientes:

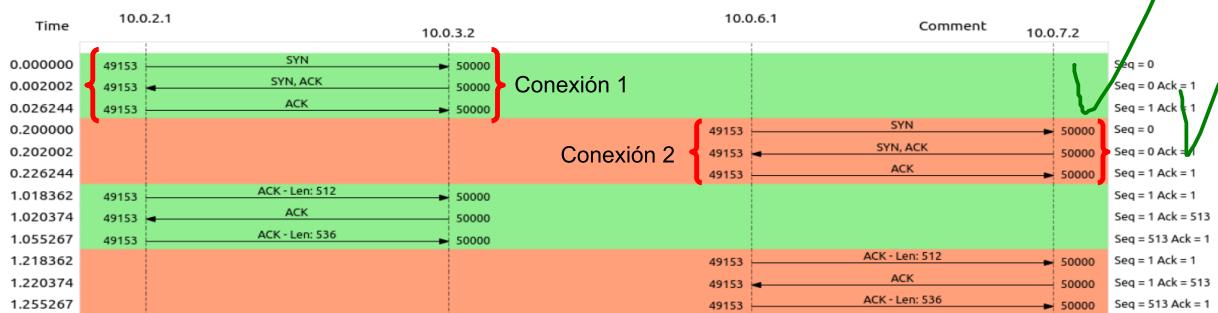
1. Establecimiento de conexión (TCP handshake)

2. Transferencia de datos
3. Control de congestión
4. Finalización de la conexión (TCP termination)

Ahora vamos a explicar cada etapa por separado mostrando cómo se visualizan en nuestra red. En primer lugar, el establecimiento de conexión, para esto se siguen los siguientes pasos:

1. Solicitud de conexión (SYN): El emisor envía un segmento TCP con el flag SYN activado para solicitar una conexión al receptor.
2. Respuesta de conexión (SYN-ACK): El receptor responde con un segmento TCP que tiene los flags SYN y ACK activados para indicar que acepta la solicitud de conexión y está listo para recibir datos.
3. Confirmación de conexión (ACK): El emisor envía un segmento TCP con el flag ACK activado para confirmar la conexión establecida. A partir de este punto, la conexión TCP está establecida y lista para la transferencia de datos.

En la siguiente imagen mostramos el establecimiento de la conexión con **flow graph** a través de **wireshark**. Se puede observar la comunicación entre los emisores y receptores.



Establecimiento de la conexión en wireshark

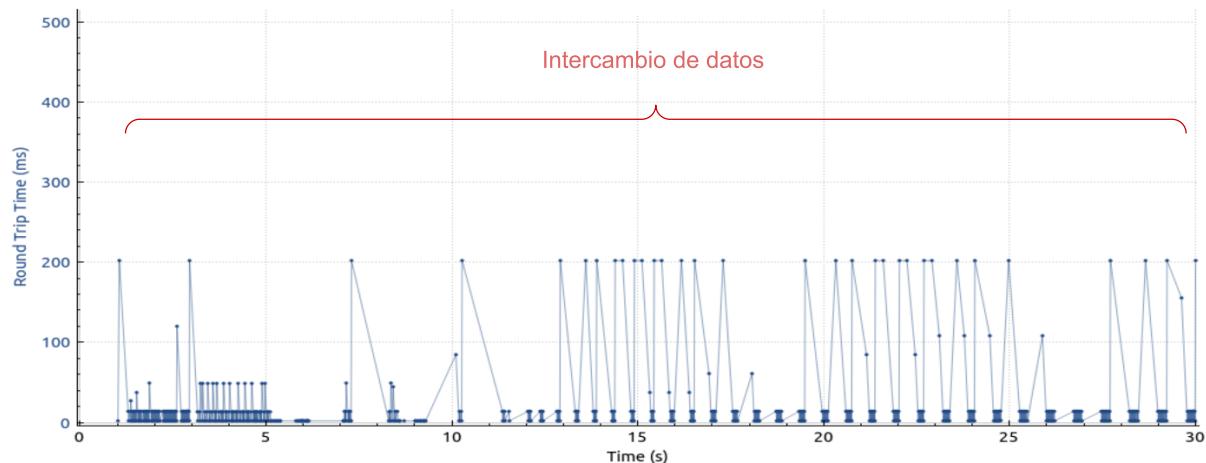
Una vez establecida la conexión, el emisor y el receptor pueden intercambiar datos en segmentos TCP. Los datos se dividen en segmentos y se envían a través de la red. El receptor envía acuses de recibo (ACK) al emisor para confirmar la recepción de los datos. A través de wireshark podemos observar el intercambio de paquetes de la siguiente forma.

Time	10.0.2.1	10.0.3.2	10.0.6.1	Comment	10.0.7.2
4.789970			49153	ACK - Len: 536	50000
4.791983			49153	ACK	50000
4.801770			49153	ACK - Len: 536	50000
4.813570			49153	ACK - Len: 536	50000
4.815583			49153	ACK	50000
4.824890	49153	ACK - Len: 512	50000	Seq = 51433 Ack = 1	
4.836210	49153	ACK - Len: 512	50000	Seq = 1 Ack = 51969	
4.838222	49153	ACK	50000	Seq = 51969 Ack = 1	
4.847530	49153	ACK - Len: 512	50000	Seq = 52505 Ack = 1	
4.858850	49153	ACK - Len: 512	50000	Seq = 1 Ack = 53041	
4.860862	49153	ACK	50000	Seq = 107521 Ack = 1	
4.870170	49153	ACK - Len: 512	50000	Seq = 108033 Ack = 1	
4.881490	49153	ACK - Len: 512	50000	Seq = 108545 Ack = 1	
4.883502	49153	ACK	50000	Seq = 109057 Ack = 1	
4.892810	49153	ACK - Len: 512	50000	Seq = 1 Ack = 109569	
				Seq = 109569 Ack = 1	
				Seq = 110081 Ack = 1	
				Seq = 1 Ack = 110593	
				Seq = 110593 Ack = 1	

Intercambio de datos entre emisores y receptores

Otra forma de verlo es a través del gráfico de RTT en el cual luego de la conexión se muestra el tiempo que tarda un paquete de datos en viajar desde el punto de origen (10.0.2.1) hasta el destino (10.0.3.2) y regresar nuevamente.

Round Trip Time for 10.0.2.1:49153 → 10.0.3.2:50000



Intercambio de datos en gráfica RTT

Ahora haremos especial mención a la etapa de control de congestión que es uno de los aspectos más importantes del protocolo TCP, junto con la posibilidad de recuperar paquetes perdidos. Esto está claro en la RFC 4614. TCP suele tratar la pérdida de paquetes como relativos a una pérdida de paquetes por congestión.

A finales de los años 80, el uso de las redes empezaba a ser cada vez más importante, por lo cual se comenzó a detectar que en el protocolo TCP se producían fenómenos de sub-uso del ancho de banda de las redes o la pérdida masiva de paquetes en algunos nodos. A partir de un trabajo publicado por Van Jacobson (1988) que daba cuenta de estos fenómenos, se incorporaron algoritmos al protocolo TCP que son los listados:

- (i) round-trip-time variance estimation
- (ii) exponential retransmit timer backoff
- (iii) slow-start

- (iv) more aggressive receiver ack policy
- (v) dynamic window sizing on congestion
- (vi) Karn's clamped retransmit backoff
- (vii) fast retransmit

Obsoleted by: [5681](#)
 Updated by: [3390](#)
 Network Working Group
 Requests for Comments: 2581
 Obsoletes: [2001](#)
 Category: Standards Track

PROPOSED STANDARD
Errata Exist
 M. Allman
 NASA Glenn/Sterling Software
 V. Paxson
 ACIRI / ICSI
 W. Stevens
 Consultant
 April 1999

TCP Congestion Control

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

This document defines TCP's four intertwined congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery. In addition, the document specifies how TCP should begin transmission after a relatively long idle period, as well as discussing various acknowledgment generation methods.

1. Introduction

This document specifies four TCP [[Pos81](#)] congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery. These algorithms were devised in [[Jac88](#)] and [[Jac90](#)]. Their use with TCP is standardized in [[Bra89](#)].

This document is an update of [[Ste97](#)]. In addition to specifying the

Congestion Avoidance and Control*

Van Jacobson[†]
 Lawrence Berkeley Laboratory
 Michael J. Karels[‡]
 University of California at Berkeley

November, 1988

Introduction

Computer networks have experienced an explosive growth over the past few years and with that growth have come severe congestion problems. For example, it is now common to see internet gateways drop 10% of the incoming packets because of local buffer overflows. Our investigation of some of these problems has shown that much of the cause lies in transport protocol implementations (*not* in the protocols themselves): The 'obvious' ways to implement a window-based transport protocol can result in exactly the wrong behavior in response to network congestion. We give examples of 'wrong' behavior and describe some simple algorithms that can be used to make right things happen. The algorithms are rooted in the idea of achieving network stability by forcing the transport connection to obey

Paper Congestion Avoidance and Control por Jacobson y Karels

A pesar que el RFC original sobre TCP (RFC 793) no contenía ningún mecanismo de control de congestión, hoy en día hay varios RFC que amplían este original (RFC 2581). Esto es porque luego de observar estas pérdidas de paquetes, Jacobson empezó a proponer la idea de que el problema no venía del protocolo si no de la implementación del mismo. Las maneras más sencillas de implementar un protocolo basado en el tamaño de ventana trajeron los problemas de comportamiento y los problemas de congestión. Los algoritmos propuestos por Jacobson están profundamente relacionados con el concepto de la estabilidad de la red, la forma de lograr esto es a través de la política de conservación de paquetes como principio.

Luego de una introducción histórica con respecto a el control de congestión, hoy en día existen muchos algoritmos para ello, entre los más importantes se encuentran:

- ❖ TCP Tahoe
- ❖ TCP Reno
- ❖ TCP NewReno
- ❖ TCP Vegas
- ❖ TCP CUBIC

En nuestro caso, decidimos utilizar el algoritmo TCP NewReno el cual es una mejora adicional del algoritmo TCP Reno. Aborda una limitación en el manejo de múltiples pérdidas de paquetes consecutivos en su antecesor. Cuando se producen múltiples pérdidas de paquetes en rápida sucesión, puede interpretarlas como una única pérdida y reducir drásticamente la ventana de congestión. TCP NewReno soluciona este problema permitiendo que el emisor retransmite los paquetes perdidos de forma selectiva y continúe aumentando la ventana de congestión de manera más eficiente.

El algoritmo TCP NewReno consta de varios mecanismos clave para controlar la congestión en una red. A continuación, se explica brevemente el funcionamiento de cada uno de estos mecanismos:

- **Slow Start:** En la fase de Slow Start, la ventana de congestión se incrementa exponencialmente en cada ronda de transmisión. Cuando se establece una nueva conexión TCP o después de una pérdida de paquete, la ventana de congestión se inicializa con un valor bajo y se duplica en cada ronda de transmisión exitosa. Este mecanismo permite que TCP NewReno aumente rápidamente la velocidad de transmisión hasta que se alcanza un umbral determinado.
- **Fast Retransmit:** El mecanismo de Fast Retransmit es una mejora que permite una retransmisión rápida de paquetes perdidos. Si el receptor recibe paquetes fuera de orden y detecta una brecha en la secuencia, envía una señal de ACK duplicada al emisor para indicar la pérdida. En lugar de esperar a que se agote el temporizador de retransmisión, el emisor retransmite de inmediato el paquete perdido antes de que se reciba cualquier retransmisión adicional.
- **Fast Recovery:** El mecanismo de Fast Recovery está estrechamente relacionado con el Fast Retransmit. Cuando se detecta una pérdida de paquete utilizando Fast Retransmit, TCP NewReno entra en el modo de Fast Recovery. En lugar de reducir la ventana de congestión a la mitad, como lo hace TCP Tahoe, TCP NewReno reduce la ventana de congestión de manera más suave y continúa transmitiendo paquetes adicionales más allá del paquete perdido. Esto acelera la recuperación y evita una reducción drástica de la velocidad de transmisión.
- **Congestion Avoidance:** Despues de la fase de Slow Start y Fast Recovery, TCP NewReno entra en la fase de Congestion Avoidance. Durante esta fase, la ventana de congestión se incrementa de manera lineal en lugar de exponencial. La ventana de congestión se incrementa lentamente para evitar una congestión excesiva en la red. TCP NewReno utiliza el algoritmo AIMD (Additive Increase, Multiplicative Decrease) para ajustar dinámicamente la ventana de congestión en función de las señales de congestión recibidas.

En conjunto, estos mecanismos permiten a TCP NewReno regular el flujo de datos y controlar la congestión en una red. El algoritmo comienza en Slow Start para aumentar rápidamente la velocidad de transmisión, utiliza Fast Retransmit y Fast Recovery para una rápida recuperación de paquetes perdidos, y luego entra en Congestion Avoidance para mantener un equilibrio entre el rendimiento y la congestión en la red.

Ahora vamos a identificar en un gráfico las etapas del control de congestión. Para obtener el gráfico de la ventana de congestión utilizamos las herramientas **tcpdump** junto a **gnuplot**.

Para poder obtener los archivos .data que utilizaremos luego en **gnuplot**, debemos agregar el siguiente código a nuestra simulación de ns3.

```
if (tracing)
{
    std::ofstream ascii;
    Ptr<OutputStreamWrapper> ascii_wrap;
    ascii.open ((prefix_file_name + "ascii").c_str ());
    ascii_wrap = new OutputStreamWrapper ((prefix_file_name + "ascii").c_str (),
                                         std::ios::out);
    stack.EnableAsciiIpv4All (ascii_wrap);

    Simulator::Schedule (Seconds (0.00001), &TraceCwnd, cwndName + ".data");
    Simulator::Schedule (Seconds (0.00001), &TraceSsThresh, prefix_file_name + "ssth.data");
    Simulator::Schedule (Seconds (0.00001), &TraceRtt, prefix_file_name + "rtt.data");
    Simulator::Schedule (Seconds (0.00001), &TraceRto, prefix_file_name + "rto.data");
    Simulator::Schedule (Seconds (0.00001), &TraceNextTx, prefix_file_name + "next-tx.data");
    Simulator::Schedule (Seconds (0.00001), &TraceInFlight, prefix_file_name + "inflight.data");
    Simulator::Schedule (Seconds (0.1), &TraceNextRx, prefix_file_name + "next-rx.data");
}
```

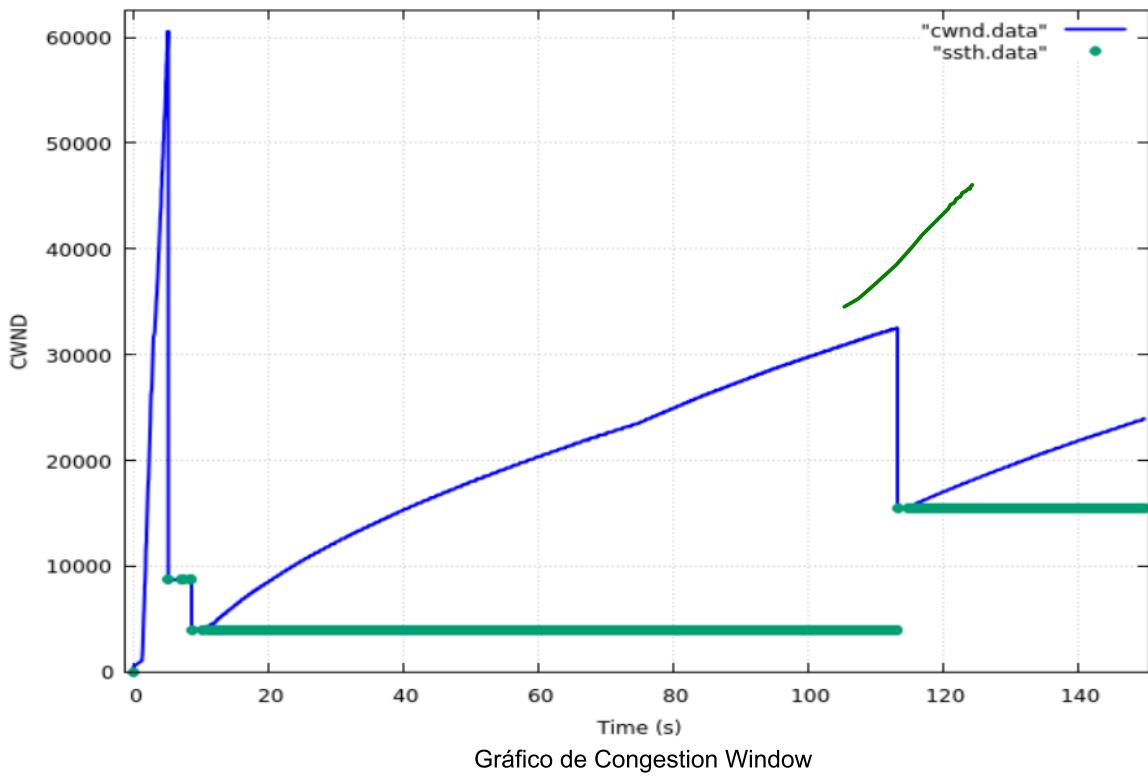
Código para hacer el trace

Ahora para graficar debemos abrir **gnuplot** y escribir el siguiente comando:

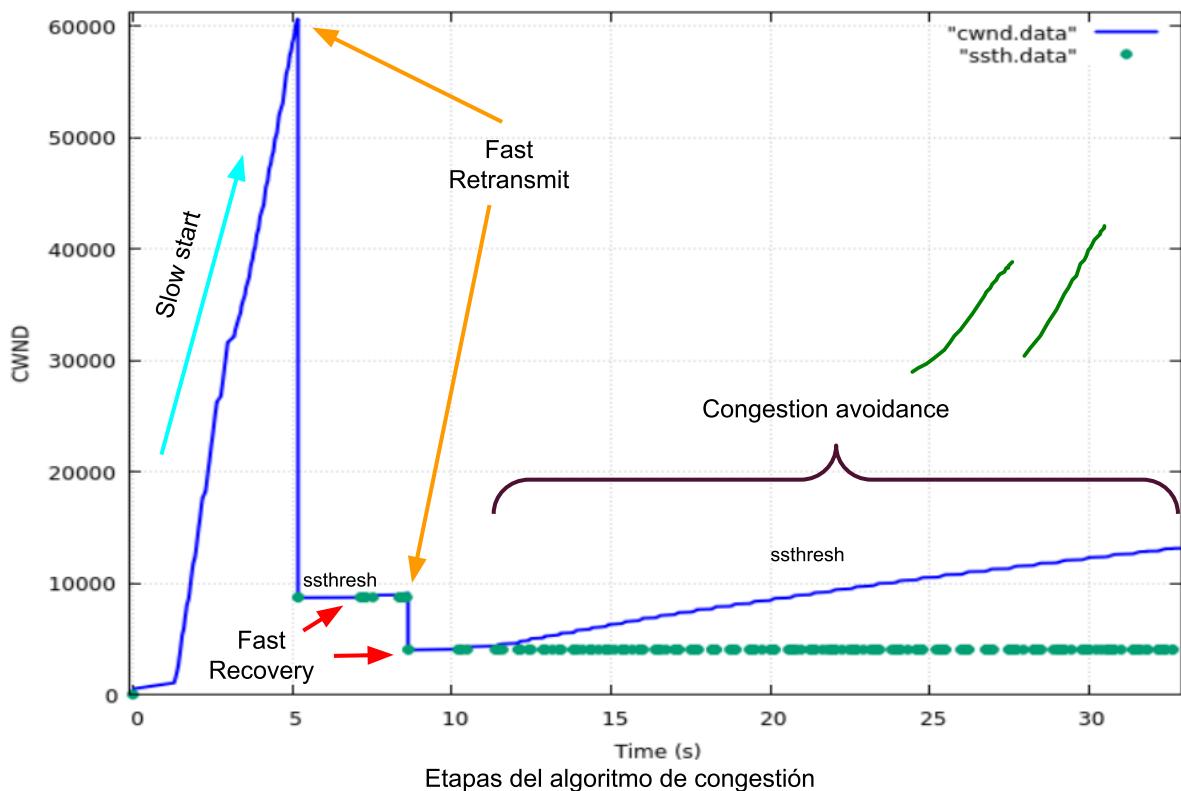
plot "cwnd.data" with lines linewidth 2 linetype 1 linecolor "blue", "ssth.data" with points pointtype 7.

Obtendremos el gráfico que se muestra a continuación, él mismo nos representa en el eje X, el tiempo y el eje Y la congestion window (cwnd). Además la líneas **azules** muestran la cwnd y los puntos **verdes** el ssthresh a lo largo del tiempo. Definiremos estos conceptos.

- Congestion Window (cwnd): La ventana de congestión es un parámetro utilizado por TCP para regular la cantidad de datos que un emisor puede enviar antes de recibir una confirmación (ACK) del receptor.
- Ssthresh (Slow Start Threshold): ssthresh es un umbral utilizado por TCP durante la fase de Slow Start para determinar cuándo cambiar del modo de crecimiento exponencial al modo de crecimiento lineal.



Ya que tenemos el gráfico de congestion window, a continuación podemos observar las distintas etapas del algoritmo que se presentan en el mismo. Entre las etapas están las anteriormente mencionadas, tal y como están explicadas en el algoritmo de control de congestión, que cumplen las características según nuestra red.



Algo a destacar es que en **wireshark** podemos observar como se lanza un fast retransmission cuando se reciben tres ack duplicados, lo cual coincide con la teoría estudiada.

476	8.570139	10.0.2.1	10.0.3.2	TCP	590	9112 49153 → 50000 [ACK] Seq=157121 Ack=1 Win=131072 Len=0
477	8.602703	10.0.3.2	10.0.2.1	TCP	62	[TCP Dup ACK 473#1] 50000 → 49153 [ACK] Seq=1 Ack=1
478	8.604716	10.0.3.2	10.0.2.1	TCP	590	9648 49153 → 50000 [ACK] Seq=157657 Ack=1 Win=131072 Len=0
479	8.614503	10.0.3.2	10.0.2.1	TCP	62	[TCP Dup ACK 473#2] 50000 → 49153 [ACK] Seq=1 Ack=1
480	8.616516	10.0.2.1	10.0.3.2	TCP	590	10184 49153 → 50000 [ACK] Seq=158193 Ack=1 Win=131072 Len=0
481	8.626303	10.0.3.2	10.0.2.1	TCP	62	[TCP Dup ACK 473#3] 50000 → 49153 [ACK] Seq=1 Ack=1
482	8.628316	10.0.3.2	10.0.2.1	TCP	590	10184 [TCP Fast Retransmission] 49153 → 50000 [ACK] Seq=1 Ack=1
483	8.739983	10.0.3.2	10.0.2.1	TCP	62	[TCP Dup ACK 473#4] 50000 → 49153 [ACK] Seq=1 Ack=1
484	9.035263	10.0.3.2	10.0.2.1	TCP	62	[TCP Dup ACK 473#5] 50000 → 49153 [ACK] Seq=1 Ack=1

Captura de 3 ACK Dup antes del Fast Retransmission

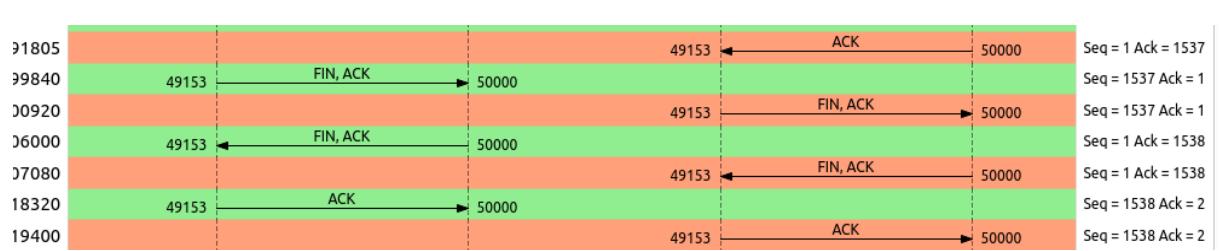
Excelente unión de esto con lo anterior !

Por otro lado, analizamos que en nuestra red, no llegamos a obtener un congestion avoidance en la etapa inicial de slow start, como indica en la teoría. Esto pasa porque el ssthresh se setea en un valor tan alto que nunca llegamos a alcanzarlo.

Por último se encuentra la etapa de finalización de la conexión la cual consta de los siguientes paso para su realización exitosa:

1. Cierre de conexión (FIN): Una de las partes envía un segmento TCP con el flag FIN activado para indicar que ha finalizado la transferencia de datos y desea cerrar la conexión.
2. Confirmación de cierre (ACK): La otra parte responde con un segmento TCP con el flag ACK activado para confirmar el cierre de la conexión.
3. Finalización de la conexión (FIN): La otra parte también envía un segmento TCP con el flag FIN activado para indicar que ha finalizado la transferencia de datos y desea cerrar la conexión.
4. Confirmación de cierre (ACK): La parte original responde con un segmento TCP con el flag ACK activado para confirmar el cierre de la conexión. A partir de este punto, la conexión TCP se cierra.

A continuación se muestra el final de la transmisión desde el flow graph de **wireshark** en el que se puede ver los pasos anteriormente explicados.



Finalización de la conexión en wireshark

Segundo escenario (TCP - UDP)

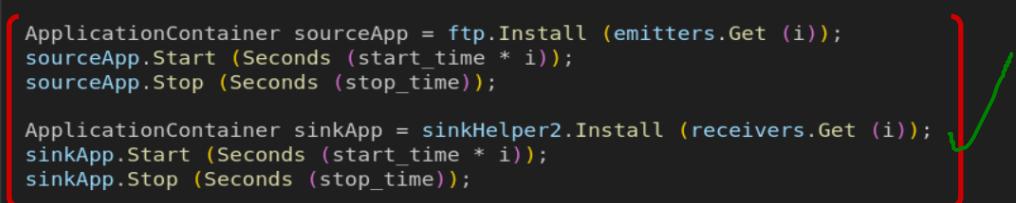
Ahora vamos a realizar las pruebas con la misma configuración que en el escenario anterior pero incluyendo el nodo UDP emitiendo junto a los demás.

Primero debemos poner a emitir el nodo UDP al mismo tiempo que los demás nodos para ver cómo cambia el flujo al tenerlo activo. Para hacer esto debemos agregar en nuestro código de ns3 las siguientes líneas:

```

for (uint16_t i = 0; i < emitters.GetN (); i++)
{
    //Se crean las aplicaciones UDP en nodos 1 de los emitters y receivers
    if(i == 1){
        OnOffHelper ftp ("ns3::UdpSocketFactory", InetSocketAddress (sink_interfaces.GetAddress
        ApplicationContainer sourceApp = ftp.Install (emitters.Get (i));
        sourceApp.Start (Seconds (start_time * i));
        sourceApp.Stop (Seconds (stop_time));
        ApplicationContainer sinkApp = sinkHelper2.Install (receivers.Get (i));
        sinkApp.Start (Seconds (start_time * i));
        sinkApp.Stop (Seconds (stop_time));
    }
    //Se crean las aplicaciones TCP en nodos 0 y 2 de los emitters y receivers
    else{
        OnOffHelper ftp ("ns3::TcpSocketFactory", InetSocketAddress (sink_interfaces.GetAddress
        ApplicationContainer sourceApp = ftp.Install (emitters.Get (i));
        sourceApp.Start (Seconds (start_time * i));
        sourceApp.Stop (Seconds (stop_time));
        ApplicationContainer sinkApp = sinkHelper.Install (receivers.Get (i));
        sinkApp.Start (Seconds (start_time * i));
        sinkApp.Stop (Seconds (stop_time));
    }
}

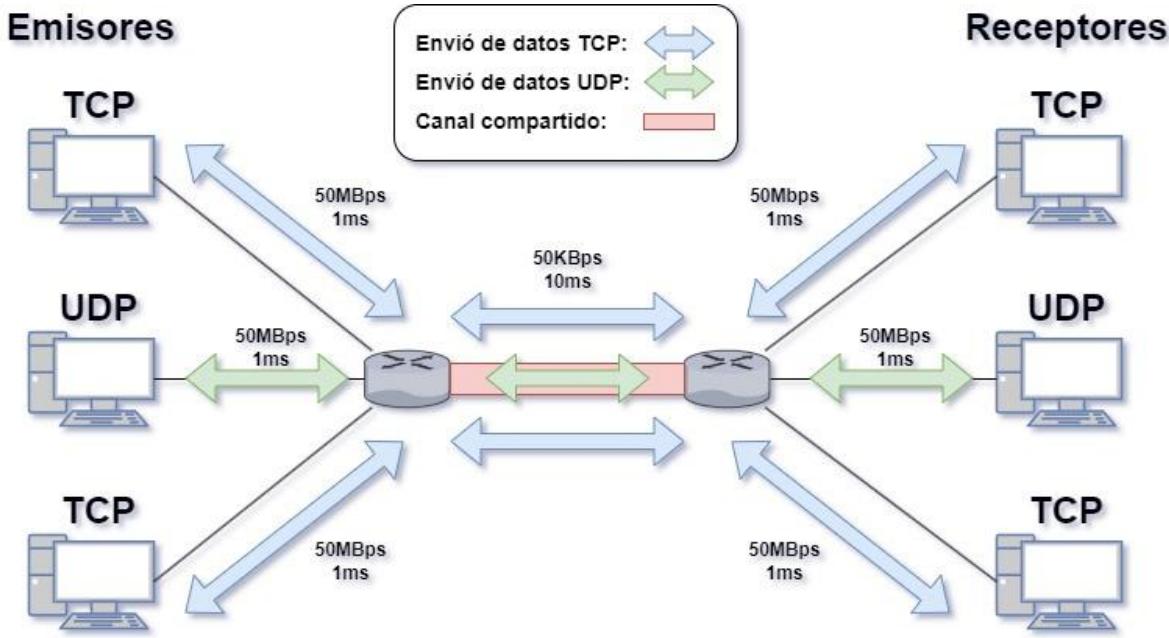
```



Código para poner a emitir al nodo UDP

Una vez que tenemos el código con las nuevas líneas, compilamos nuevamente nuestro archivo de ns3 y podemos ver los cambios en la red.

El objetivo en esta sección es ver cómo al tener a los 3 emisores transmitiendo datos por el mismo canal compartido, esto afecta a nuestra red. A continuación mostramos la estructura con los tres emisores transmitiendo.



Representación gráfica del envío de datos por el canal compartido

Ahora que ya se explicó el nuevo escenario vamos a mostrar qué pasa con el ancho de banda utilizado por cada emisor y mostraremos en los paquetes TCP donde se ven las distintas acciones del protocolo

Ancho de Banda

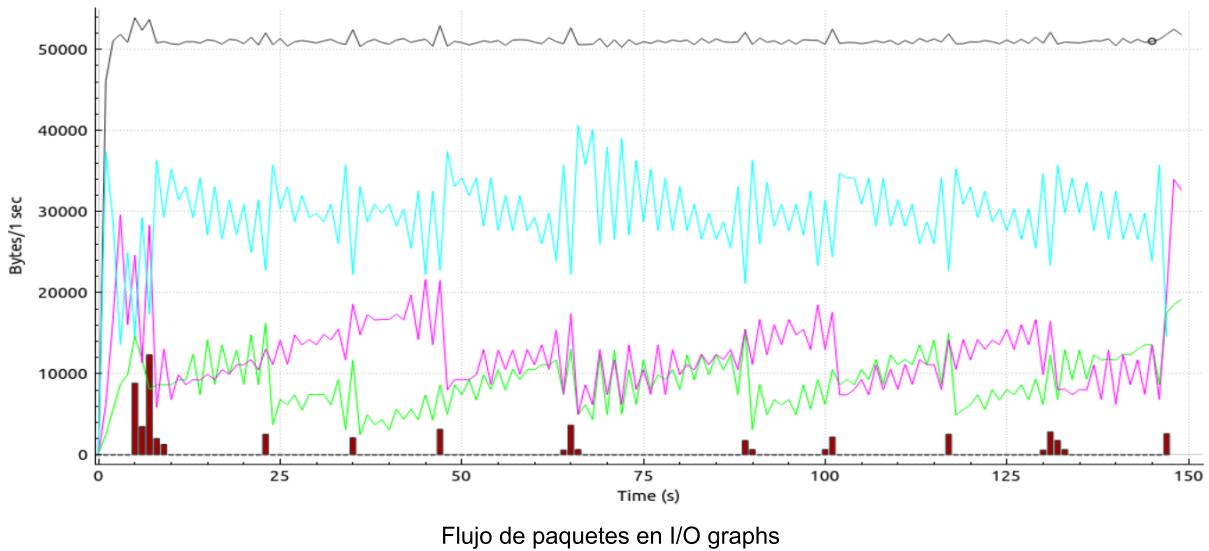
El tráfico UDP no participa en el control de congestión de TCP, lo que significa que los paquetes UDP pueden ocupar parte del ancho de banda disponible sin ajustarse a las señales de congestión detectadas por TCP. Esto puede llevar a situaciones en las que los paquetes UDP ocupen un porcentaje significativo del ancho de banda, lo que afecta negativamente la capacidad de TCP para ajustar su tasa de transmisión de manera óptima.

Por lo tanto vamos a ver el gráfico de **I/O graph** que nos brinda **wireshark** pero esta vez le agregaremos un filtro personalizado más para mostrar el flujo de los paquetes UDP en la simulación, lo representaremos con un color **celeste**.

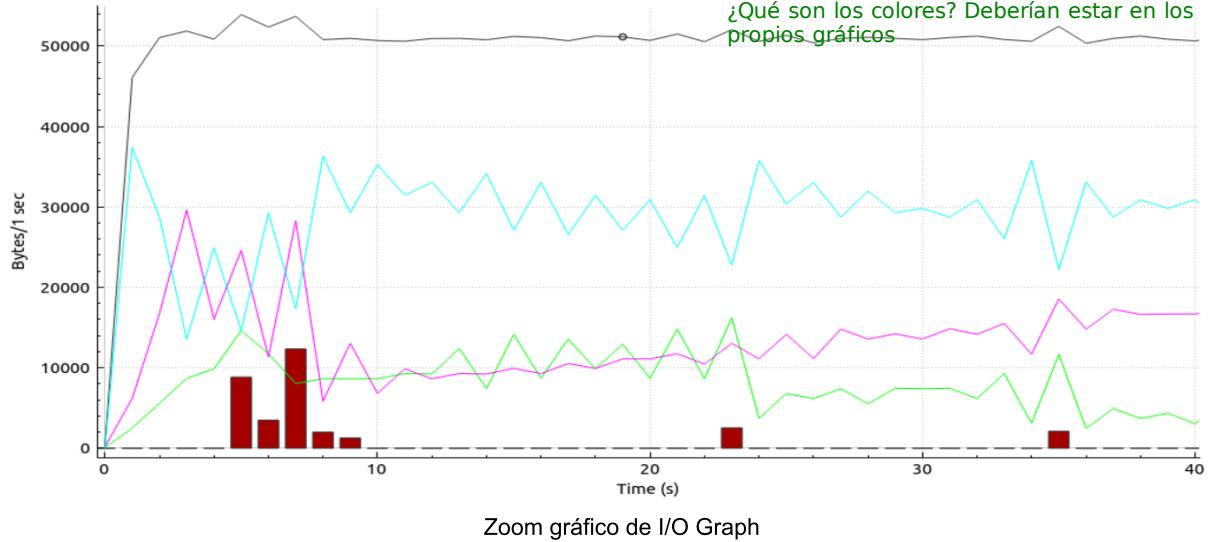
Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period
<input checked="" type="checkbox"/>	TCP 10.0.6.1	ip.addr == 10.0.6.1	Green	Line	Bytes		None
<input checked="" type="checkbox"/>	TCP 10.0.2.1	ip.addr == 10.0.2.1	Magenta	Line	Bytes		None
<input checked="" type="checkbox"/>	UDP 10.0.4.1	ip.addr == 10.0.4.1	Cyan	Line	Bytes		None

Filtro para UDP del I/O graph

En la siguiente imagen se muestra el gráfico resultante de la red con el nuevo filtro aplicado (línea celeste).



Flujo de paquetes en I/O graphs



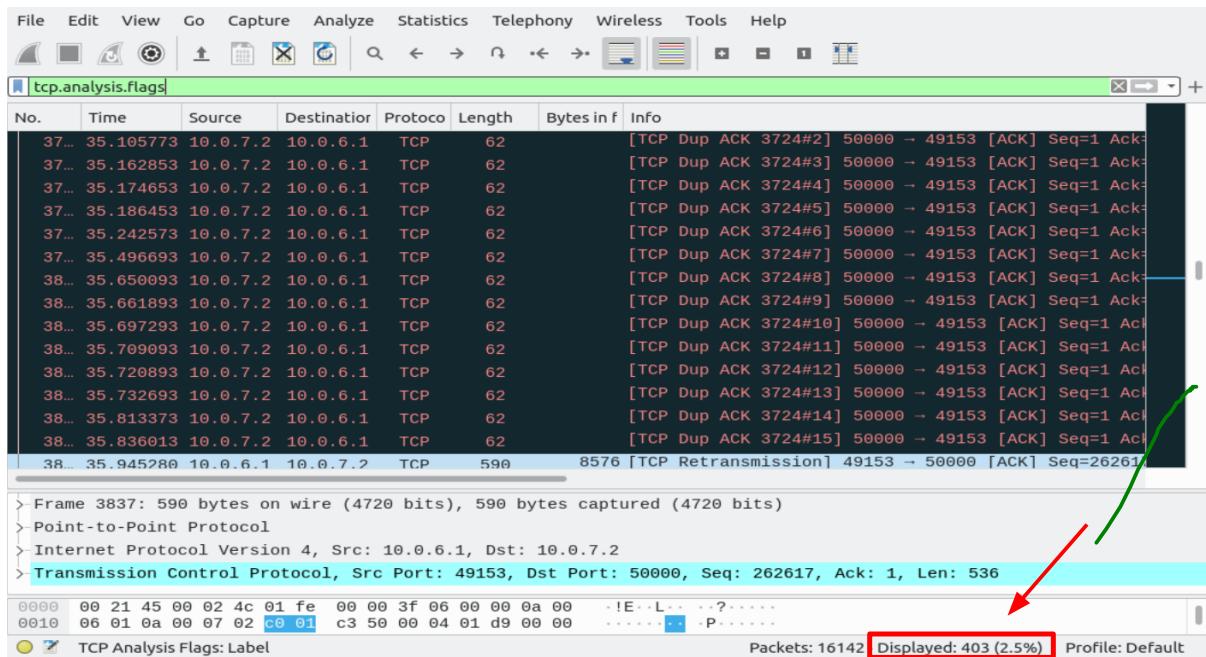
Zoom gráfico de I/O Graph

En el gráfico anterior podemos observar como el flujo de los paquetes UDP toma gran parte del ancho de banda del canal, lo que produce que los emisores TCP tengan que reducir su ancho de banda para no saturarlo. Por lo tanto se muestra como la línea **celeste** tiene flujos aproximados de entre 23000 a 35000 bytes/s mientras que los emisores TCP (líneas **verdes** y **rosa**) bajan sus flujos a un aproximado de 3000 a 22000 bytes/s.

A comparación del gráfico de la sección análisis de gráficos TCP en el primer escenario, se puede observar como la pérdida de paquetes, en este caso, disminuyó debido a que al tener los dos protocolos, se pierden menos cantidad de datos que en TCP, ya que UDP no controla la pérdida de paquetes. Esta pérdida se ve en el tamaño de las barras rojas de ambos gráficos. Si vamos a la página principal de wireshark y aplicamos el filtro

Si udp no controla y tenía 50M, y estaba en 50k el BW de los intermedios, porque acá da 30?

para los paquetes perdidos, podemos ver como el porcentaje de paquetes baja a un 2.5% cuando tenemos TCP y UDP.



Lista de paquetes afectados mostrados en wireshark

También si vamos al apartado conversation de wireshark, podemos observar como cada conversación UDP transmite más paquetes que las dos conversaciones TCP.

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	Bi
10.0.2.1	10.0.3.2		4.659	1.866 k			3.010	1.774 k		1.649		92 k	0.000000	149.9830	
10.0.4.1	10.0.5.2		8.080	4.379 k			8.080	4.379 k		0		0	1.117881	146.3517	
10.0.6.1	10.0.7.2		3.403	1.357 k			2.190	1.291 k		1.213		66 k	0.200000	149.7614	

Conversations TCP-UDP

Análisis de los paquetes incluyendo nodos UDP

Comencemos analizando la simulación que solo contiene a los nodos TCP transmitiendo (primer escenario) para realizar una comparativa.

Para comprender mejor la información recibida en el header del paquete, utilizamos la [RFC 793](#). Donde encontramos una descripción de la ventana y los números de secuencia que analizaremos a continuación.

En este caso podemos ver como el tamaño de la ventana se setea en 32768, aunque estamos utilizando window scaling, por lo que el valor final de la ventana será este número multiplicado por 4. Lo que nos da 131072 bytes. A su vez, podemos ver como los

bytes in flight van aumentando y, el límite para el receptor será el tamaño de su ventana nombrado anteriormente.

```
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 512]
Sequence Number: 1      (relative sequence number)
Sequence Number (raw): 1
[Next Sequence Number: 513    (relative sequence number)]
Acknowledgment Number: 1      (relative ack number)
Acknowledgment number (raw): 1
1000 .... = Header Length: 32 bytes (8)
Flags: 0x010 (ACK)
Window: 32768
[Calculated window size: 131072]
[Window size scaling factor: 4]
Checksum: 0x0000 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
- Options: (12 bytes), Timestamps, End of Option List (EOL), End of Option List (EOL)
  ▶ TCP Option - Timestamps
```

Tamaños de las ventanas en wireshark

No.	Time	Source	Destination	Protocol	Length	Bytes in f	Calculated window	Info
1	0.000000	10.0.2.1	10.0.3.2	TCP	58		65535	49153 → 50000 [SYN] Seq=0 Win=65535 Len=0 TSval=0 TSecr=0 WS=4 SACK_PERM=1
2	0.024322	10.0.3.2	10.0.2.1	TCP	58		65535	50000 → 49153 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 TSval=13 TSecr=0 WS=4 S
3	0.026324	10.0.2.1	10.0.3.2	TCP	54		131072	49153 → 50000 [ACK] Seq=1 Ack=1 Win=131072 Len=0 TSval=26 TSecr=13
4	1.008202	10.0.2.1	10.0.3.2	TCP	566	512	131072	49153 → 50000 [ACK] Seq=1 Ack=1 Win=131072 Len=512 TSval=1008 TSecr=13
5	1.042614	10.0.3.2	10.0.2.1	TCP	54		131072	50000 → 49153 [ACK] Seq=1 Ack=513 Win=131072 Len=0 TSval=1031 TSecr=1008
6	1.044627	10.0.2.1	10.0.3.2	TCP	590	536	131072	49153 → 50000 [ACK] Seq=513 Ack=1 Win=131072 Len=536 TSval=1044 TSecr=1031
7	1.279520	10.0.3.2	10.0.2.1	TCP	54		131072	50000 → 49153 [ACK] Seq=1 Ack=1044 Win=131072 Len=0 TSval=1268 TSecr=1044
8	1.281533	10.0.2.1	10.0.3.2	TCP	590	536	131072	49153 → 50000 [ACK] Seq=1044 Ack=1 Win=131072 Len=536 TSval=1281 TSecr=1268
9	1.281545	10.0.2.1	10.0.3.2	TCP	590	1072	131072	49153 → 50000 [ACK] Seq=1585 Ack=1 Win=131072 Len=536 TSval=1281 TSecr=1268
10	1.328226	10.0.3.2	10.0.2.1	TCP	54		131072	50000 → 49153 [ACK] Seq=1 Ack=2121 Win=131072 Len=0 TSval=1317 TSecr=1281
11	1.330239	10.0.2.1	10.0.3.2	TCP	590	536	131072	49153 → 50000 [ACK] Seq=2121 Ack=1 Win=131072 Len=536 TSval=1330 TSecr=1317
12	1.330250	10.0.2.1	10.0.3.2	TCP	590	1072	131072	49153 → 50000 [ACK] Seq=2657 Ack=1 Win=131072 Len=536 TSval=1330 TSecr=1317
13	1.330262	10.0.2.1	10.0.3.2	TCP	590	1608	131072	49153 → 50000 [ACK] Seq=3193 Ack=1 Win=131072 Len=536 TSval=1330 TSecr=1317
14	1.376931	10.0.3.2	10.0.2.1	TCP	54		131072	50000 → 49153 [ACK] Seq=1 Ack=3193 Win=131072 Len=0 TSval=1365 TSecr=1330
15	1.378944	10.0.2.1	10.0.3.2	TCP	590	1072	131072	49153 → 50000 [ACK] Seq=3729 Ack=1 Win=131072 Len=536 TSval=1378 TSecr=1365
16	1.378956	10.0.2.1	10.0.3.2	TCP	590	1608	131072	49153 → 50000 [ACK] Seq=4265 Ack=1 Win=131072 Len=536 TSval=1378 TSecr=1365
17	1.378968	10.0.2.1	10.0.3.2	TCP	590	2144	131072	49153 → 50000 [ACK] Seq=4801 Ack=1 Win=131072 Len=536 TSval=1378 TSecr=1365
18	1.413837	10.0.3.2	10.0.2.1	TCP	54		131072	50000 → 49153 [ACK] Seq=1 Ack=4265 Win=131072 Len=0 TSval=1402 TSecr=1330
19	1.415850	10.0.2.1	10.0.3.2	TCP	590	1608	131072	49153 → 50000 [ACK] Seq=5337 Ack=1 Win=131072 Len=536 TSval=1415 TSecr=1402
20	1.415862	10.0.2.1	10.0.3.2	TCP	590	2144	131072	49153 → 50000 [ACK] Seq=5873 Ack=1 Win=131072 Len=536 TSval=1415 TSecr=1402
21	1.415874	10.0.2.1	10.0.3.2	TCP	590	2680	131072	49153 → 50000 [ACK] Seq=6409 Ack=1 Win=131072 Len=536 TSval=1415 TSecr=1402
22	1.437437	10.0.3.2	10.0.2.1	TCP	54		131072	50000 → 49153 [ACK] Seq=1 Ack=5337 Win=131072 Len=0 TSval=1426 TSecr=1378
23	1.439450	10.0.2.1	10.0.3.2	TCP	590	2144	131072	49153 → 50000 [ACK] Seq=6945 Ack=1 Win=131072 Len=536 TSval=1439 TSecr=1426
24	1.439462	10.0.2.1	10.0.3.2	TCP	590	2680	131072	49153 → 50000 [ACK] Seq=7481 Ack=1 Win=131072 Len=536 TSval=1439 TSecr=1426
25	1.439474	10.0.2.1	10.0.3.2	TCP	590	3216	131072	49153 → 50000 [ACK] Seq=8017 Ack=1 Win=131072 Len=536 TSval=1439 TSecr=1426

Tabla de wireshark donde se ven los bytes in flight

Sin embargo, antes de llegar al límite de la ventana, podemos ver que comienza la pérdida de paquetes. El nodo receptor avisa que hubo un salto en el número de secuencia, por lo que el emisor activa un *fast retransmission* para reenviar el paquete perdido.

368	5.113546	10.0.2.1	10.0.3.2	TCP	566	16384	131072 49153 → 50000 [ACK] Seq=131585 Ack=1 Win=131072 Len=512 Tval=5113 TSecr=5078
369	5.121738	10.0.2.1	10.0.3.2	TCP	566	16896	131072 49153 → 50000 [ACK] Seq=132097 Ack=1 Win=131072 Len=512 Tval=5121 TSecr=5078
370	5.129930	10.0.2.1	10.0.3.2	TCP	566	17408	131072 49153 → 50000 [ACK] Seq=132609 Ack=1 Win=131072 Len=512 Tval=5129 TSecr=5078
371	5.138122	10.0.2.1	10.0.3.2	TCP	566	17920	131072 49153 → 50000 [ACK] Seq=133121 Ack=1 Win=131072 Len=512 Tval=5138 TSecr=5078
372	5.146314	10.0.2.1	10.0.3.2	TCP	566	18432	131072 49153 → 50000 [ACK] Seq=133633 Ack=1 Win=131072 Len=512 Tval=5146 TSecr=5078
373	5.147942	10.0.3.2	10.0.2.1	TCP	62		131072 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tval=5136 TSecr=3859 S
374	5.154506	10.0.2.1	10.0.3.2	TCP	566	18432	131072 49153 → 50000 [ACK] Seq=134145 Ack=1 Win=131072 Len=512 Tval=5154 TSecr=5136
375	5.159262	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 373#1] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
376	5.162691	10.0.2.1	10.0.3.2	TCP	566	18944	131072 49153 → 50000 [ACK] Seq=134657 Ack=1 Win=131072 Len=512 Tval=5162 TSecr=5148
377	5.179582	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 373#2] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
378	5.170890	10.0.2.1	10.0.3.2	TCP	566	19456	131072 49153 → 50000 [ACK] Seq=135169 Ack=1 Win=131072 Len=512 Tval=5170 TSecr=5148
379	5.172595	10.0.2.1	10.0.3.2	TCP	566	19456	131072 [TCP Fast Retransmission] 49153 → 50000 [ACK] Seq=116225 Ack=1 Win=131072 Len=0 Tsv
380	5.217302	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 373#3] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
381	5.228622	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 373#4] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
382	5.275563	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#5] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
383	5.286823	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#6] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
384	5.321903	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#7] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
385	5.333223	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#8] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
386	5.344543	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#9] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
387	5.391263	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#10] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
388	5.402583	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#11] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv
389	5.413903	10.0.3.2	10.0.2.1	TCP	78		131072 [TCP Dup ACK 373#12] 50000 → 49153 [ACK] Seq=1 Ack=116225 Win=131072 Len=0 Tsv

Tabla de wireshark donde se ve el fast retransmission

Luego de esto, notamos que el número de *bytes in flight* se redujo, y aumentaba más lento que previo a la congestión.

No.	Time	Source	Destination	Protocol	Length	Bytes in flight	Calculated window	Info
433	7.115075	10.0.2.1	10.0.3.2	TCP	566	20384	131072 [TCP Out-Of-Order] 49153 → 50000 [ACK] Seq=122881 Ack=1 Win=131072 Len=51	
434	7.115087	10.0.2.1	10.0.3.2	TCP	590	20840	131072 49153 → 50000 [ACK] Seq=143185 Ack=1 Win=131072 Len=536 Tval=7115 TSecr=	
435	7.124222	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=135681 Win=131072 Len=0 Tsv=7113 TSecr=59	
436	7.126235	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=143721 Ack=1 Win=131072 Len=536 Tval=7126 TSecr=	
437	7.147823	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=136753 Win=131072 Len=0 Tsv=7136 TSecr=60	
438	7.149836	10.0.2.1	10.0.3.2	TCP	590	8040	131072 49153 → 50000 [ACK] Seq=144257 Ack=1 Win=131072 Len=536 Tval=7149 TSecr=	
439	7.149847	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=144793 Ack=1 Win=131072 Len=536 Tval=7149 TSecr=	
440	7.171423	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=137825 Win=131072 Len=0 Tsv=7160 TSecr=60	
441	7.173436	10.0.2.1	10.0.3.2	TCP	590	8040	131072 49153 → 50000 [ACK] Seq=145329 Ack=1 Win=131072 Len=536 Tval=7173 TSecr=	
442	7.173447	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=145865 Ack=1 Win=131072 Len=536 Tval=7173 TSecr=	
443	7.230423	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=138897 Win=131072 Len=0 Tsv=7219 TSecr=60	
444	7.232436	10.0.2.1	10.0.3.2	TCP	590	8040	131072 49153 → 50000 [ACK] Seq=146401 Ack=1 Win=131072 Len=536 Tval=7232 TSecr=	
445	7.232447	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=146937 Ack=1 Win=131072 Len=536 Tval=7232 TSecr=	
446	7.254023	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=139961 Win=131072 Len=0 Tsv=7242 TSecr=61	
447	7.256036	10.0.2.1	10.0.3.2	TCP	590	8040	131072 49153 → 50000 [ACK] Seq=147473 Ack=1 Win=131072 Len=536 Tval=7256 TSecr=	
448	7.256047	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=148009 Ack=1 Win=131072 Len=536 Tval=7256 TSecr=	
449	7.277623	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=141041 Win=131072 Len=0 Tsv=7266 TSecr=61	
450	7.279636	10.0.2.1	10.0.3.2	TCP	590	8040	131072 49153 → 50000 [ACK] Seq=148545 Ack=1 Win=131072 Len=536 Tval=7279 TSecr=	
451	7.279647	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=149081 Ack=1 Win=131072 Len=536 Tval=7279 TSecr=	
452	7.301223	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack=142113 Win=131072 Len=0 Tsv=7290 TSecr=61	
453	7.303236	10.0.2.1	10.0.3.2	TCP	590	8040	131072 49153 → 50000 [ACK] Seq=149617 Ack=1 Win=131072 Len=536 Tval=7303 TSecr=	
454	7.303247	10.0.2.1	10.0.3.2	TCP	590	8576	131072 49153 → 50000 [ACK] Seq=150153 Ack=1 Win=131072 Len=536 Tval=7303 TSecr=	

Aumento de los bytes in flight

Finalmente esto nos lleva al gráfico de la *congestion window(cwnd)* o ventana de congestión. En donde podemos notar que la misma crece hasta el momento de la congestión. Luego a eso, se reduce drásticamente y comienza el lento crecimiento que vimos en los paquetes.

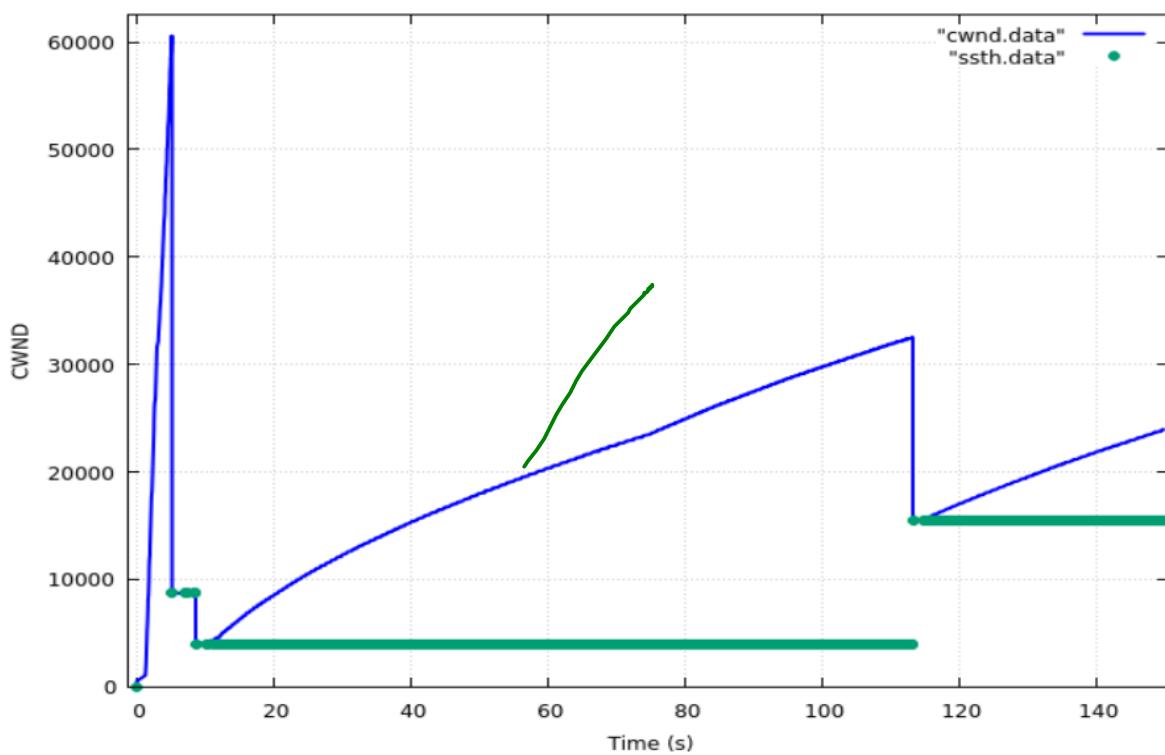


Gráfico de congestión de redes TCP.

Al analizar el segundo escenario (TCP-UDP), notamos algunas similitudes. Principalmente notamos que el seteo inicial de la ventana es el mismo, utilizando el mismo *scaling factor*.

```
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 512]
Sequence Number: 1025      (relative sequence number)
Sequence Number (raw): 1025
[Next Sequence Number: 1537      (relative sequence number)]
Acknowledgment Number: 1      (relative ack number)
Acknowledgment number (raw): 1
1000 .... = Header Length: 32 bytes (8)
Flags: 0x010 (ACK)
Window: 32768
[Calculated window size: 131072]
[Window size scaling factor: 4]
Checksum: 0x0000 [unverified]
[checksum status: unverified]
Urgent Pointer: 0
Options: (12 bytes), Timestamps, End of Option List (EOL), End of Option List (EOL)
  ▶ TCP Option - Timestamps
```

Tamaño de la ventana en wireshark.

Luego, en el análisis de los paquetes, el aumento de la *congestion window* y los *bytes in flight* es similar.

4	1.008202	10.0.2.1	10.0.3.2	TCP	566	512	131072 49153 → 50000 [ACK] Seq=1 Ack
5	1.042614	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack
6	1.044627	10.0.2.1	10.0.3.2	TCP	590	536	131072 49153 → 50000 [ACK] Seq=513 A
7	1.279520	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack
8	1.281533	10.0.2.1	10.0.3.2	TCP	590	536	131072 49153 → 50000 [ACK] Seq=1049 .
9	1.281545	10.0.2.1	10.0.3.2	TCP	590	1072	131072 49153 → 50000 [ACK] Seq=1585 .
10	1.404694	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack
11	1.406707	10.0.2.1	10.0.3.2	TCP	590	536	131072 49153 → 50000 [ACK] Seq=2121 .
12	1.406719	10.0.2.1	10.0.3.2	TCP	590	1072	131072 49153 → 50000 [ACK] Seq=2657 .
13	1.406731	10.0.2.1	10.0.3.2	TCP	590	1608	131072 49153 → 50000 [ACK] Seq=3193 .
14	1.602694	10.0.3.2	10.0.2.1	TCP	54		131072 50000 → 49153 [ACK] Seq=1 Ack
15	1.604707	10.0.2.1	10.0.3.2	TCP	590	1072	131072 49153 → 50000 [ACK] Seq=3729 .

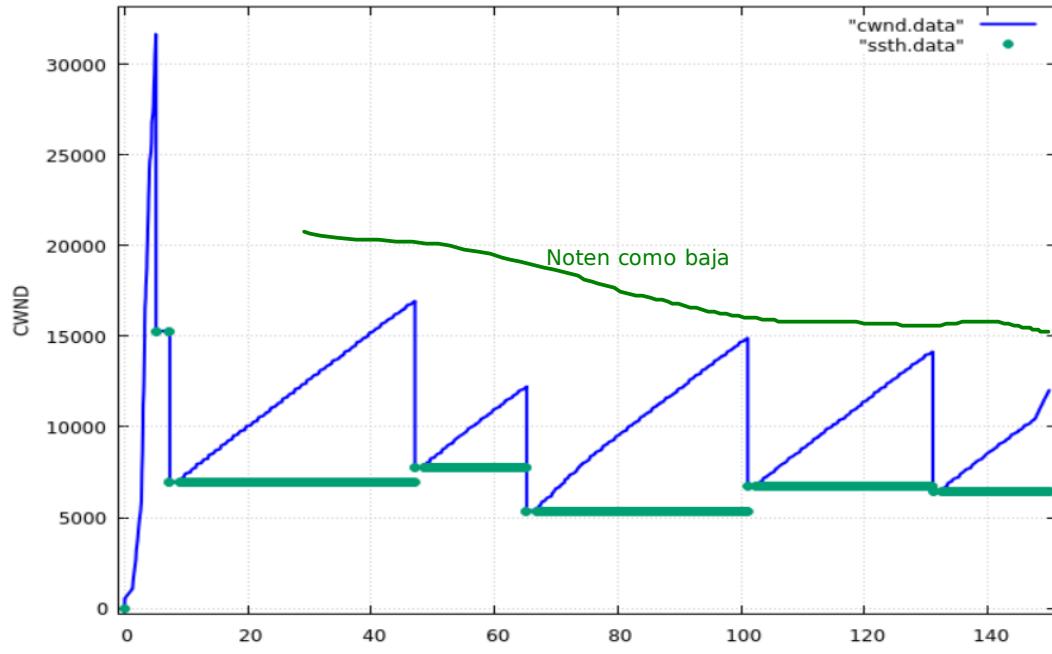
Tabla proporcionada por Wireshark donde se ven los bytes in flight.

Una diferencia que vimos, fue que la saturación del canal surgió 0.10 segundos antes.

435	4.992281	10.0.4.1	10.0.5.2	UDP	542		49153 → 50000 Len=512
436	5.003121	10.0.4.1	10.0.5.2	UDP	542		49153 → 50000 Len=512
437	5.013961	10.0.4.1	10.0.5.2	UDP	542		49153 → 50000 Len=512
438	5.025761	10.0.2.1	10.0.3.2	TCP	590		131072 [TCP Previous segment not captured]
439	5.027774	10.0.3.2	10.0.2.1	TCP	62		131072 50000 → 49153 [ACK] Seq=1 Ack=59473
440	5.037561	10.0.2.1	10.0.3.2	TCP	590	1072	131072 49153 → 50000 [ACK] Seq=60545 Ack=1
441	5.039574	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 439#1] 50000 → 49153
442	5.048401	10.0.4.1	10.0.5.2	UDP	542		49153 → 50000 Len=512
443	5.059241	10.0.4.1	10.0.5.2	UDP	542		49153 → 50000 Len=512
444	5.071041	10.0.2.1	10.0.3.2	TCP	590	1608	131072 49153 → 50000 [ACK] Seq=61081 Ack=1
445	5.073054	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 439#2] 50000 → 49153
446	5.082841	10.0.2.1	10.0.3.2	TCP	590	2144	131072 49153 → 50000 [ACK] Seq=61617 Ack=1
447	5.084854	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 439#3] 50000 → 49153
448	5.094641	10.0.2.1	10.0.3.2	TCP	590	2680	131072 49153 → 50000 [ACK] Seq=62153 Ack=1
449	5.096654	10.0.3.2	10.0.2.1	TCP	62		131072 [TCP Dup ACK 439#4] 50000 → 49153

Tabla de Wireshark donde se ven los paquetes perdidos y duplicados.

Como consecuencia de lo observado anteriormente, la *congestion window* sufre una caída en su tamaño unos segundos antes en cuanto al tiempo. Y por ende su pico quedó bastante lejos de el tamaño de ventana máximo seteado anteriormente



A su vez, otra cosa que podemos analizar en esta comparativa, es la cantidad de pérdida de paquetes y la continuidad en el tiempo de esta pérdida.

En el primer escenario (TCP), una vez saturada la red y luego de que la ventana de congestión se redujo, la red volvió a presentar la pérdida de datos de forma más prolongada por parte del receptor. Incluso luego de aumentar paulatinamente la *congestion window* nuevamente.

En cambio, en el segundo escenario (TCP-UDP), la red volvió a presentar pérdida de paquetes luego de un tiempo pasado la primera congestión. Gracias a esto, el protocolo TCP redujo la ventana de congestión nuevamente, por lo que se redujo su ancho de banda ya que UDP tomó parte de él.

Problemas Encontrados

Durante el desarrollo del trabajo práctico nos encontramos con algunos problemas que nos gustaría detallar.

- ❖ Ejecutar ns3

Nos encontramos con la dificultad de compilar la herramienta cuando quisimos instalar una versión más actual en la máquina virtual, esto pasaba ya que la máquina no tenía suficiente memoria asignada al momento de querer instalar la herramienta, así que compilaba hasta cierto punto y luego se tildaba la máquina virtual o arrojaba error. Eso fue solucionado utilizando la versión instalada en la máquina virtual.

- ❖ Creación de la red

La creación de la red fue un problema ya que en un primer momento no conseguimos que realice lo que precisábamos y no podíamos concretar la correcta conexión, no podíamos hacer que sature el canal, no podíamos obtener los datos de salida para obtener los gráficos. Esto a medida que investigamos un poco más pudimos solucionarlo.

- ❖ Análisis de las redes

Algunos datos nos costó conseguirlos por el hecho de que no entendíamos qué era lo que proporcionaba cada gráfico o como encontrar cierta información necesaria. Como por ejemplo el cálculo de la velocidad de transmisión o las etapas del protocolo TCP.

En general todos los problemas encontrados fueron por desinformación con los temas anteriormente mencionados y todos se solucionaron investigando más a fondo cada uno para llegar a una conclusión acorde a los datos que teníamos.

Conclusiones

Este trabajo se realizó con el objetivo de implementar y simular una red Dumbbell Topology mediante la herramienta ns3 y configurar la red en dos escenarios distintos. El primer escenario, en el que tenemos dos nodos TCP emitiendo y el segundo, donde tenemos dos nodos TCP y uno UDP. Esto con la finalidad de visualizar, mediante gráficos, el flujo de datos y las características de la red.

Para lograr el objetivo se tuvo que investigar sobre la utilización y configuración de ns3, junto con los distintos gráficos de wireshark y otras herramientas para saber que tipo de información nos brindaba.

Finalmente, llegamos a la conclusión de que el objetivo del trabajo se cumplió pudiendo simular correctamente la red y analizar el comportamiento de las pruebas solicitadas con éxito.