

Sistemas Operativos y Redes II

Trabajo Práctico 2. Análisis de Redes.

**Alumnos: Alan Echabarri - alanechabarri@gmail.com Juan M. Corbera - jmcorbera@gmail.com**

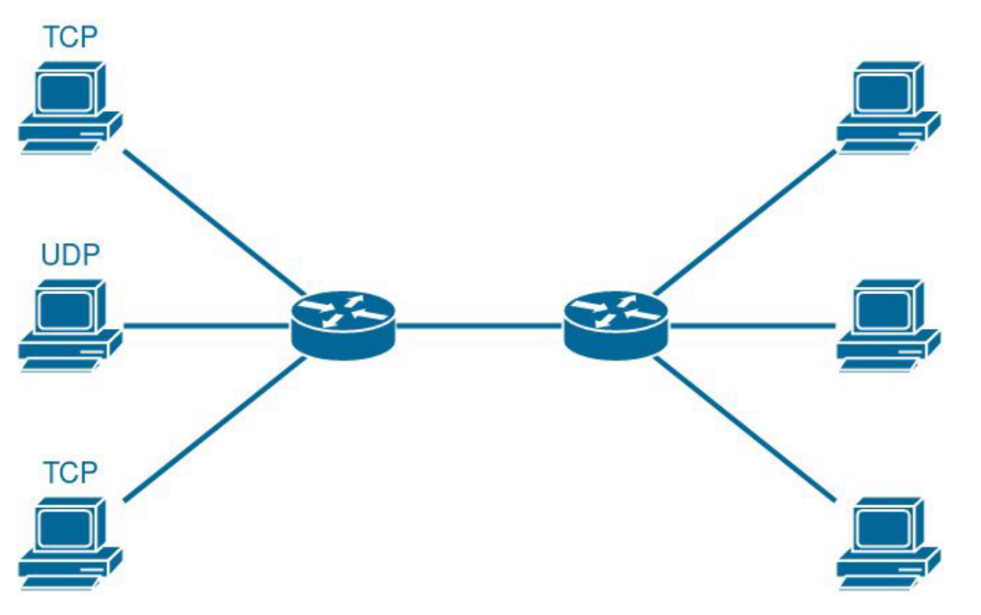
**Profesor: Alexis Tcach**

**Comisión: 1**

Primer Semestre 2022

# Introducción

En el siguiente informe se va a detallar el análisis de redes poniendo el foco en las consignas pedidas en el TP2 de Sistemas Operativos y Redes II.

En primer lugar, implementaremos el escenario sobre el cual realizaremos las diferentes prácticas. Se diseña para esto un escenario con 3 emisores on/off application, 3 receptores y dos nodos intermedios, utilizaremos lo que normalmente es conocido como Dumbbell topology. La red contará con 2 transmisores TCP y un transmisor UDP. Las conexiones serán cableadas.

Para las primeras prácticas, solo utilizaremos 2 transmisores TCP y se buscará saturar el canal de transmisión, medir la velocidad de transferencia. Se mostrará mediante gráficos tamaño de colas de recepción, ventana de TCP y se explicaran las distintas etapas del protocolo TCP. También se calculará el ancho de banda del canal.

Luego con la misma configuración se agregará un nodo UDP y se mostrará los cambios que suceden.

En una segunda etapa se agregará un algoritmo llamado Westwood, se explicará su funcionamiento. El mismo será implementado en el primer escenario probado en el punto anterior, utilizando 2 transmisores TCP y presentaran los datos obtenidos de las muestras realizadas.

# Configuración

Las distribuciones de Linux utilizadas para el desarrollo y las distintas pruebas fueron:

● Ubuntu 18.04 LTS

Para las diferentes prácticas se utilizó un software de simulación de redes con eventos discretos, desarrollado en C++, conocido como Network Simulator 3 (NS3). El mismo es un software libre de licencia GNU GPL v2 (General Public License GNU Versión 2).

NS3 cuenta con una documentación muy extensa de donde se puede obtener los prerrequisitos de instalación, como así también las diferentes configuraciones para poder utilizar la aplicación.

* <https://www.nsnam.org/docs/tutorial/html/index.html>

Para esta serie de prácticas se utilizó la release ns-3.30.1. Descargada desde el siguiente link:

* <https://www.nsnam.org/releases/ns-3-30/>

Instalación Simple:

1. Descomprimimos la carpeta del NS3, nos ubicamos dentro de la misma y ejecutamos:

* build.py: ./build.py –enable-examples –enable-tests
* bake: ./bake.py build

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

Compilar waf

* ./waf clean
* ./waf -d optimized –enable-examples –enable-tests configure
* ./waf clean
* ./waf -d debug –enable-examples –enable-tests configure
* ./waf

1. Para verificar que todo haya quedado bien configurado, colocar algunos de los archivos.cc (ejemplo: ../ns-3.30.1/examples/tutorial/hello-simulator.cc) dentro de la carpeta scratch y ejecutar/.

* ./waf --run nombre\_archivo

Para graficar las pruebas se utilizó **Gnuplot**, que es una herramienta visual para graficar las trazas de las diferentes simulaciones realizadas. Este programa nos permite generar gráficas 2D y 3D. Puede producir resultados directamente en pantalla y en varios formatos de imagen, como PNG, JPEG, etc.

# Desarrollo

Se diseñó un escenario con 3 emisores on/off application, 3 receptores y dos nodos intermedios. Se conectaron los 3 emisores a un nodo, luego éste a otro y finalmente éste a los 3 destinos finales. Uno de los emisores se definió como UDP y los otros 2 TCP. Con conexiones cableadas de 150Kb/s para nodos-router y un enlace Cuello de botella de 50 kb/s entre routers.

Las diferentes lógicas de pruebas que se realizaron, fueron guardadas en sus correspondientes archivos .cc y compiladas con NS3. En los capítulos siguientes desarrollaremos los resultados de las mismas y mostraremos gráficamente las salidas capturadas mediante archivos .pcap y .data.

Los archivos .pcap son las capturas de los paquetes que viajan a lo largo de las distintas interfaces. Dichas capturas son guardadas con el formato “Name-NodeId-interfazId.pcap”

Por ejemplo, el archivo ‘Dumbbell-TCP-2-0.pcap’, tendrá la captura de los paquetes que fueron transmitidos por el nodo 2 usando la interfaz 0 (interface de red que une a los nodos transmisores con el router).

Los archivos .data guardan información de algún evento en específico. Los mismos nos brindarán datos, por ejemplo, de los valores CWND, SSTHRESH, RTT, RTO etc., que luego podrán ser graficados y guardados en formato png a través de la aplicación **Gnuplot**.

Para la compilación de los diferentes archivos, no ubicaremos dentro de la carpeta ns-3.30.1 y ejecutaremos el compilador ./waf del archivo en cuestión. Recordemos que el archivo a compilar debe estar colocado en la carpeta /scratch.

La forma de ejecutar el compilador será la siguiente:

* ./waf --run scratch/nombre\_de\_archivo.cc

# Introducción protocolo TCP

Explicaremos como el protocolo TCP utiliza distintas técnicas para evitar la congestión de la red, para esto el protocolo implementa un mecanismo de control entre el emisor y el receptor. La congestión ocurre cuando los emisores envían un volumen de paquetes mayor al que un receptor puede procesar. Si el emisor percibe que hay poca congestión entre él y el receptor va a incrementar su tasa de emisión. Por lo contrario, si percibe mucha congestión reducirá la misma. Esto lo logrará mediante la variación del tamaño de la ventana SND.WND. Este tamaño se determinará por el valor de la ventana de recepción y por el valor de la ventana de congestión (CWND) que dependerá del algoritmo de control de congestión. Este algoritmo percibe la congestión de la red a través de los eventos de pérdidas de segmentos que pueden ser: la recepción de 3 ACK´s duplicados o la expiración del TIMEOUT (tiempo de espera).

Se han implementado varias técnicas de control de congestión en TCP para limitar la velocidad de envío de datos que ingresan a Internet mediante la regulación del tamaño de la CWND. Las técnicas que utiliza, son los algoritmos **Slow Start**, **Congestion Avoidance**, **Fast Retransmit** y **Fast Recovery**. Las primeras dos fases del mecanismo de control de lacongestión, *Slow Start* y *Congestion Avoidance*, regulan la cantidad de paquetes inyectadosen la conexión y son responsables de la detección de la congestión, controlan las transmisiones. Mientras que los otros dos algoritmos, *Fast Retransmit* y *Fast**Recovery*, reaccionan para superar la congestión y proporcionar un mecanismo que acelera larecuperación de la conexión.

Además del valor CWND, el algoritmo suma otra variable llamada SSTHRESH (umbral *entre slow start* y *congestion avoidance*). Esta variable va a determinar si se va a utilizar el mecanismo de arranque lento o el de evitación de la congestión. Cuando CWND se encuentre por debajo del umbral se utilizará *Slow Start* y cuando se encuentre por encima se utilizará *Congestion Avoidance*. El valor de SSTHRESH se calculará cuando ocurra un evento de pérdida de segmentos. Se tomará el valor de CWND (con al menos dos segmentos) y se lo dividirá por 2, por lo tanto, SSTHRESH = CWND / 2.

**- Slow Start (Arranque lento)**

El algoritmo Slow Start sugiere que se incremente el tamaño de la CWND en el mismo número de segmentos con acuse de recibo, por cada ACK recibido con éxito. Al recibir un ACK, se puede enviar el doble de la cantidad de datos que fueron reconocidos por ese ACK (política de aumento multiplicativo). TCP empieza la comunicación probando la red, enviando un solo segmento, y fija el tamaño de la CWND a un segmento. Es decir, en el primer RTT la ventana aumenta el valor de CWND a dos, con lo que se envían dos segmentos, luego aumenta a cuatro, se envían cuatro segmentos y así sucesivamente utilizando múltiplos de dos. Por lo tanto, esto provoca que se duplique la tasa de envío por cada RTT y que aumente exponencialmente la CWND hasta que perdemos un paquete que es un signo de congestión.

A partir de esto, disminuimos nuestra tasa de envío y reducimos la CWND a uno para garantizar la liberación de recursos de red.

**- Congestion Avoidance (Evitación de la congestión)**

Cuando la CWND alcanza el valor de la variable umbral denominado SSTHRESH (umbral de Slow Start), TCP abandona la fase de Slow Start y entra en una fase Congestion Avoidance para encontrar la capacidad disponible de la red y no saturarla.

Por lo tanto, el valor de la ventana de transmisión en cada momento da una indicación de la tasa de transmisión de TCP, ya que, en ausencia de errores se va a transmitir durante un RTT como máximo tantos segmentos como indique la ventana de transmisión. Se abandona el estado de Congestion Avoidance cuando el emisor TCP detecta un evento de congestión por expiración de timeout o por la recepción consecutiva de tres ACK duplicados.

En el primer caso, se reduce el valor de CWND a 1 y se define el valor de SSTHRESH a la mitad del valor alcanzado por CWND o al menos dos segmentos. En caso contrario, se reconfigura la variable SSTHRESH al valor CWND/2, divide a la mitad el valor de CWND y le suma 3 de los ACK duplicados recibidos luego, establece el valor de SSTHRESH a la mitad del valor de CWND y se invoca a la estrategia Fast Retransmit

**- Fast Retransmit (Retransmisión rápida)**

La fase de Fast Retransmit permite que un emisor conozca que se ha perdido un segmento incluso antes de que venza el temporizador de retransmisión. Cuando se recibe un segmento fuera de orden, el receptor genera lo que se denomina un ACK duplicado, es decir, vuelve a asentir los mismos datos que ya asintió anteriormente. Lo que hace es retransmite el segmento, donde no se requiere tiempo de espera para que expire el temporizador de retransmisión, solo asume en que en cuanto llegan al emisor tres ACKs duplicados (cuatro ACKs idénticos) son un buen indicador de un segmento perdido. Entonces se entra en Fast Recovery

**- Fast Recovery (Recuperación rápida)**

Los segmentos recibidos luego del segmento perdido generaron los ACKs duplicados, esto quiere decir que todavía hay datos entre los dos extremos. TCP no quiere reducir el flujo bruscamente, por lo tanto, evita el ingreso a *Slow Start* y de *Congestion Avoidance*. Luego, el emisor establece SSTHRESH en la mitad de la CWND actual y ajusta el valor de la CWND igual a uno, esto lo hace por cada ACK duplicado que recibió de los tres segmentos que ya habían sido transmitidos luego del segmento perdido CWND = SSTHRESH + 3, reiniciando la fase Slow Start.

## Parte 1

### Prueba con 2 emisores TCP

En primer lugar, se realizaron pruebas con 2 emisores TCP. En donde se simuló una saturación de canal.

Con los archivos .pcap y .data generados de la compilación del archivo **“dumbbell-2TCP”**, analizaremos cómo se satura el canal y la velocidad de transferencia.

Con la ejecución del comando “gnuplot cfg\_cwnd\_ssthresh.plt”, se obtuvo la gráfica que contiene las variaciones en el tiempo de la ventana de congestión y SSTHRESH de los nodos 2 y 3 (TCP’s), vía interfaz 0. Fig. 1

El umbral SSTHRESH comienza con un valor arbitrariamente alto, para este trabajo, ese valor inicial lo omitimos.

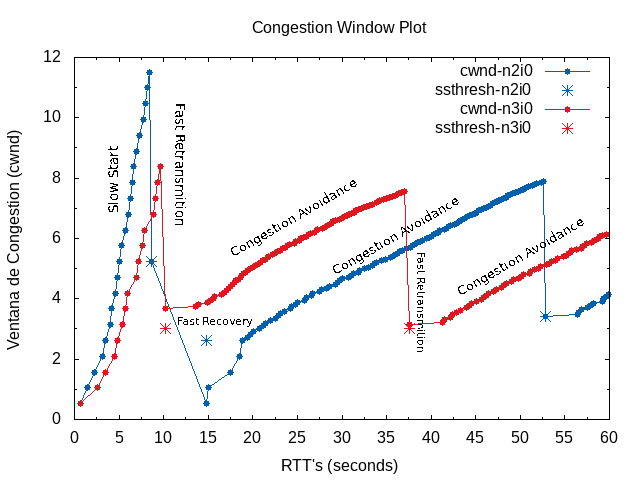
****

Fig. 1

Lo que se observa en la Fig. 1 es como varia con el tiempo la ventana de congestión del emisor, esto es la cantidad de paquetes estimada que se pueden enviar por tiempo sin saturar el canal. Mostrando como varia la CWND podemos indirectamente ver la cantidad de paquetes que llegan a destino por tiempo. Debido a que la CWND solo crece mientras se obtengan acuses de recibos.

**SLOW START**

Como observamos en la Fig.1 que las ventanas de congestión de ambos nodos emisores comienzan en cero, y van incrementando exponencialmente los paquetes que envían por la red. Iniciando la etapa de Slow Start.

**FAST RECOVERY**

Abrimos con Wireshark el archivo generado “SOR2-TP2-Dumbbell-2TCP-2-0” (Nodo 2), se observa que en el segundo 8,64 tras recibir 3 ACKs duplicados, realiza un Fast retransmit y entra en fast recovery. Ver Fig. 2.

En la Fig. 3, vemos que el último ACK fue el 19833, por lo que el CWND de ese momento 10720 Bytes, es el que se va utilizar como referencia.

Como se puede observar en el gráfico con zoom se setea el umbral de congestión a la mitad del CWND 🡪 SSTHRESH=CWND/2 (10720/2=5360).

También se setea CWND = CWND /2 debido al Fast Retransmition (no se espera a un RTO).

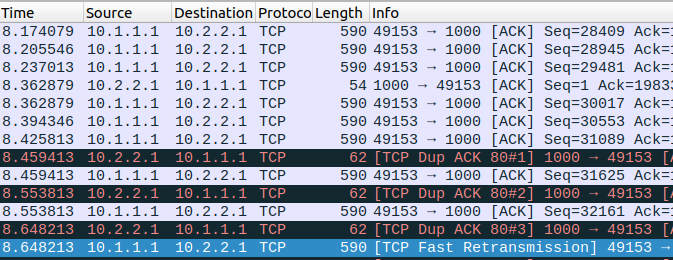


Fig. 2

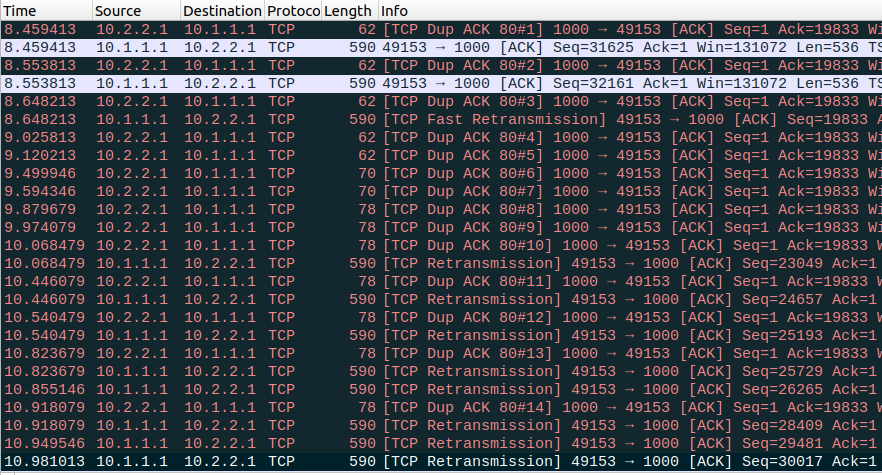
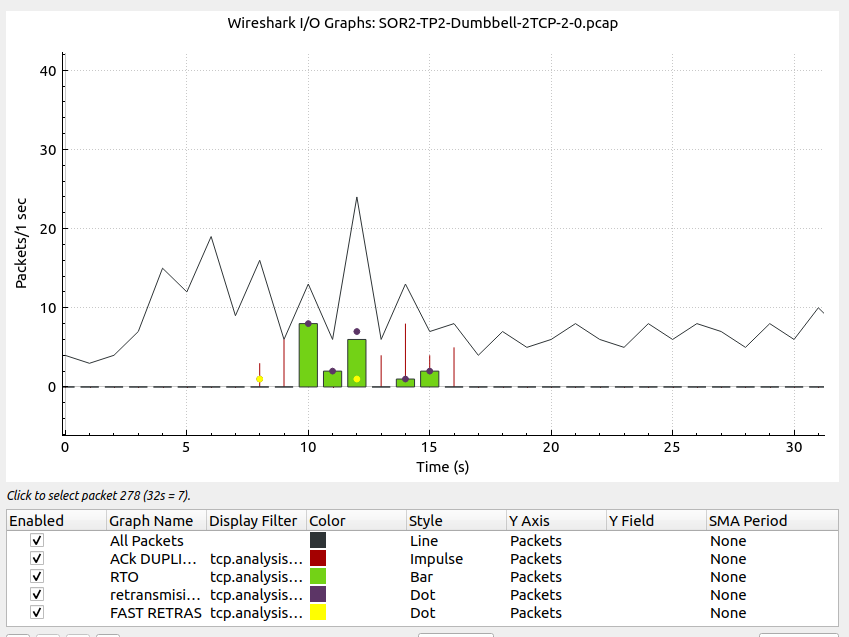


Fig. 3

Del segundo 8.6 al 14 el RTT del nodo 2 es demasiado elevado. Por lo tanto, durante este tiempo, en el grafico no se aprecia un incremento de ventana CWND = SSTHRESH + 3\*MSS ni tampoco CWND = CWND +1MSS \* RTT ya que no se están muestreando (el CWND cambia por RTT no por segundos).

Con un CWND seteado en 5360 bytes, desde el seg 8.64 al 10.068 tcp estuvo esperando a recibir suficientes ACKs duplicados como tamaño de la ventana (5360=10 ACKs). En ese momento es cuando comienza a retransmitir 1 paquete por ACK duplicado, pero se ve interrumpido por un RTO, por lo que debe volver a Slow Start.

Desde el segundo 10.98 al 14.79 ocurre slow start hasta que se vence otro RTO y ocurre slow start otra vez durante ese tiempo no se pudieron recabar datos de como varia la CWND.Sin embargo desde 14.79 CWND baja nuevamente a 1\* MSS = 536 bytes, Y SSTHRESH baja a 2680 bytes, Fig X.



Esto se puede entender mejor abriendo el archivo n2i0cwnd.data

Por otra parte, cabe aclarar que el SSTHRESH solo actualiza su valor 3 veces.

*Anomalía*

En esta ocasión no entendemos exactamente porque entra en fast recovery con solo 2 ACKs duplicados, pero de todas formas notamos que efectivamente se entra en fastRecovery, debido a que CWND se setea = CWND /2 , y como CWND en ese momento es igual a SSTHRESH inmediatamente comienza congestion avoidance, esto lo sabemos no solo por la curvatura del grafico(casi horizontal) sino por la forma en que se  va incrementando congestion windows, esta no se incrementa de a 536 bytes, sino de a fracciones de 536.

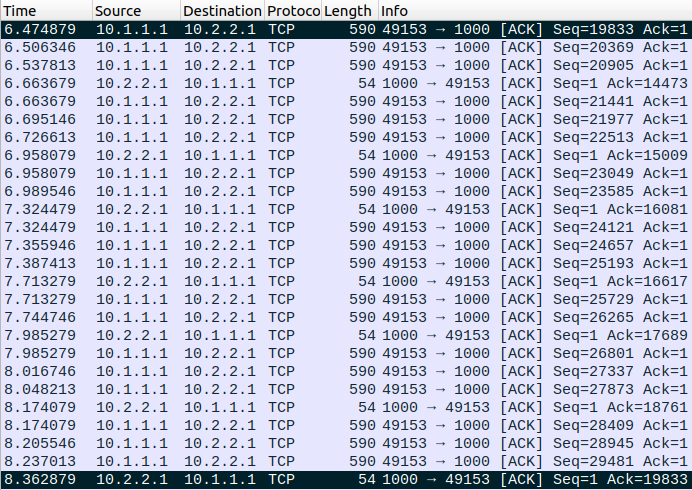
También descartamos la posibilidad de que el fast Retrasmit se inicie debido al flag Explicit Congestion Notification, debido a que revisamos las ventanas de los paquetes desde wireshark, en la sección de protocolo ipv4, y este no aparecía marcado.

**Cola de congestión**

Con respecto a la cola de congestion, la ventana del receptor permanece estática. Esto se puede apreciar en la variante windows, de la información Tcp en wireShark. En la anterior imagen, por ejemplo, se observa como la ventana permanece fija en el máximo valor. Esto se debe a que el buffer del receptor nunca se llega a llenar, ya sea porque el nodo router dropea paquetes o porque la cantidad de bytes por segundo no supera la velocidad de procesamiento.

**Velocidad de transferencia**

Abrimos con Wireshark el archivo generado “SOR2-TP2-Dumbbell-2TCP-2-0”. Ahora vamos a medir la velocidad de transferencia.



**BW =**

**RTT** (tiempo de ida y vuelta): tiempo que pasa entre enviar datos y recibir el ACK correspondiente al nro. de secuencia.

**RTT** = 8.362879 - 6.474879 🡪 1.88

**MSS** = 590

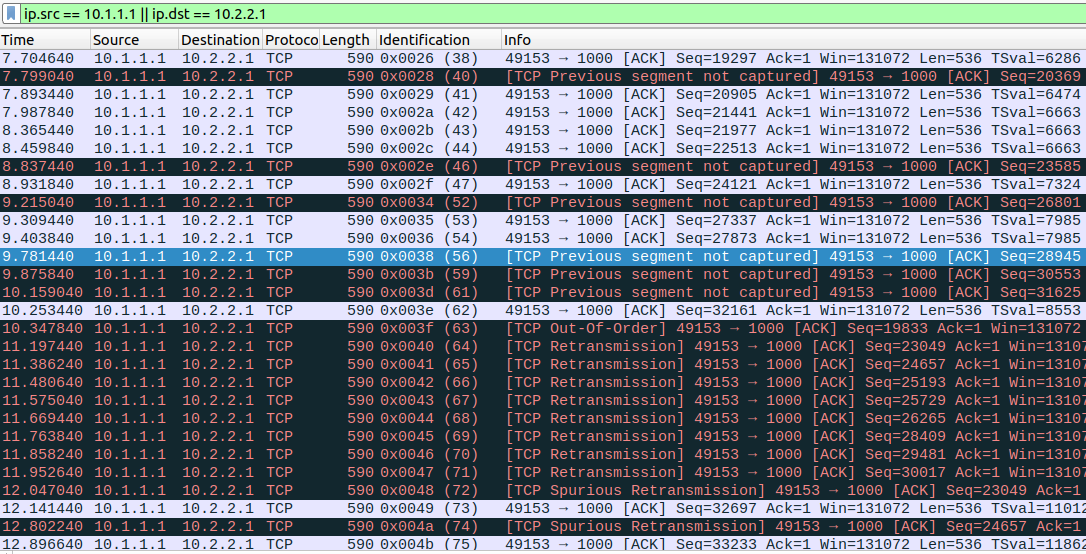
**CantPaq** = 19

=1

**BW =** 🡪 **BW =** 11,92kb / 3.88 = 3.07kb/s

Aproximadamente 3 kb/s cuando la red está configurada a 30kb como mínimo. Esto demuestra que la red no es estable.

Para ver la cantidad de paquetes que llegan a destino, abrimos el archivo n1i0.pcap con wireshark y aplicamos el filtro “ip.src == 10.1.1.1 || ip.dst == 10.2.2.1” en la pestaña debajo expandimos la solapa de protocolo IP , damos click derecho  identification y “add column”

****

En la imagen anterior se puede ver que los paquetes del emisor n2i0 al receptor n6i0, se observan múltiples paquetes que no llegan.Si prestamos atencion a la columna identificacion, se ve que los paquetes 39,45,48,49,50,51,55 y 60 entre otros, brillan por su ausencia. Esta es la causa de los mensajes de ack duplicados que vimos anteriormente desde el punto de vista del nodo 2 interfaz 0. Por lo cual, al llegarle el tercer ack duplicado el nodo 2 entra en fast retransmit y fast recovery.

**Ancho de Banda del Canal**

**Calcular el ancho de banda del canal**

**Explicar que sucede (Anomalías? explicarlas)**

### Prueba con 2 emisores TCP y uno UDP

**Ancho de Banda de los diferentes Emisores**

**TCP distintas acciones del protocolo**

## Parte 2

Una implementación de TCP Westwood.

Westwood emplea el paradigma de control de congestión AIAD (Aumento Aditivo/Disminución Adaptativa). Cuando ocurre un episodio de congestión, en lugar de reducir a la mitad el cwnd, este protocolo intenta estimar el ancho de banda de la red y usa el valor estimado para ajustar el cwnd. Westwood realiza el muestreo del ancho de banda en cada recepción de ACK.

Los dos métodos principales en la implementación son CountAck (const TCPHeader&) y EstimateBW (int, const, Time). El método CountAck calcula el número de segmentos reconocidos al recibir un ACK. El valor de BudgetBW estima el ancho de banda en función del valor devuelto por CountAck y el intervalo de muestreo (tiempo entre llegadas del último ACK).

Un pseudocodigo sencillo para comprender como se calcula el Bandwidth es:

*if (ACK is received)*

*sample\_BWE[k] = (acked\*pkt\_size\*8)/* *(now - lastacktime);*

*BWE[k] = (19/21)\*BWE[k-1] + (1/21)\* (sample\_BWE[k]+ sample\_BWE[k-1]);*

*endif*

donde:

acked=numero de segmentos reconocidos por el ultimo ack.

*pkt\_size*=tamaño del segmento en bytes.

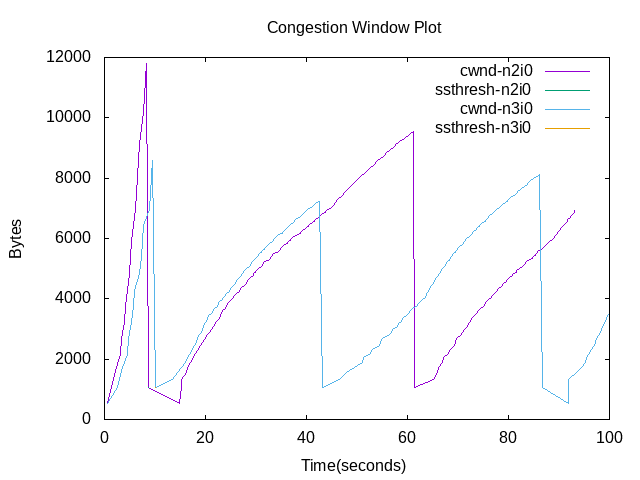
*now*= hora actual

*lastacktime*= hora en que se recibio el ack anterior.

k y k-1 =indican el valor actual y anterior de la variable

*BWE*= medida filtrada de paso bajo del ancho de banda disponible.

Ahora vamos a mostrar el funcionamiento, en el siguiente gráfico se muestra la congestion windows del n2i0 y n3i0(insertar plot)



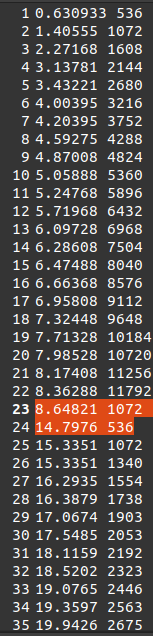
**Slow start**

Como se observa en el gráfico, la congestion windows va incrementando exponeencialmente como en los otros flavors. Sin embargo, algo cambia cuando se producen 3 acks duplicados.

en ese momento ssthresh = (BWE\*RTTmin)/seg\_size; es decir el umbral de congestión se setea en una proporcion del ancho de banda estimado, y luego si el cwnd es mayor a este valor, se entra en Congestion avoidance, cwnd se coloca en **cwnd = ssthresh.**

RTTmin = el valor rtt mas pequeño observado durante la conexion.

Esto se puede observar en el archivo, en el segundo 8.64 ocurren los 3 ack duplicados, ssthresh se setea en 1072, y como cwnd por ese entonces era mayor a ssthresh, tambien se setea en 1072.

Si ocurre un RTO, pasa lo siguiente

if (coarse timeout expires)

ssthresh = (BWE\*RTTmin)/seg\_size;

 if (ssthresh < 2)

ssthresh = 2;

endif;

cwin = 1;

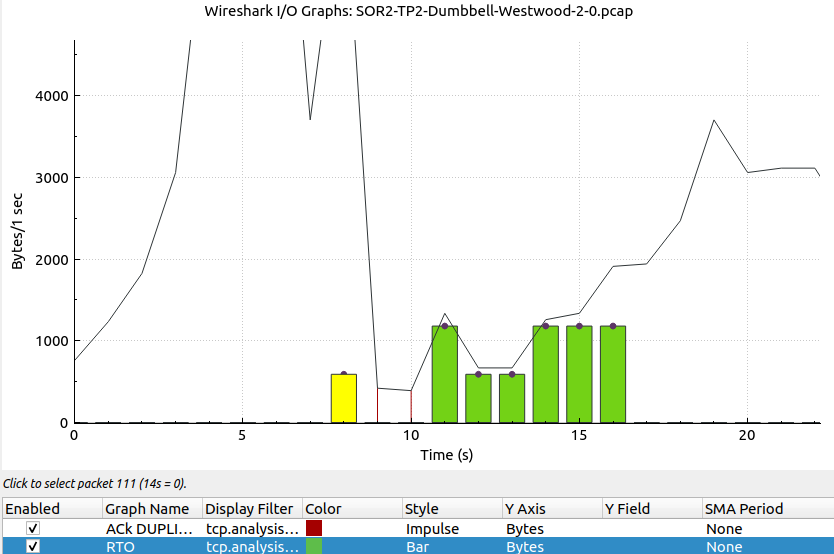
Lo que entendemos es que cwnd se setea en 1cwnd y ssthresh vuelve a setearse como el algoritmo anterior, sin embargo esta vez, si ssthresh es menor a 2cwnd, ssthsh se setea en 2cwnd. Lo que podemos observar en nuestro trabajo en el segundo 14.79 corresponde a un RTO vencido, por lo que la cwnd se estea en 1cwnd=536, desde ahi hace un mini slow start, hasta que supera al ssthresh y comienza congestion avoidance.

Desde aqui no hay mucho que decir, se utiliza el algoritmo de evitacion de congestion anteriormente mostrado, hasta que ocurran 3 acks duplicados, o haya otro rto.

cwnd+mss\*(mss/cwnd).

cwnd=1072+536\*536/1072

cwnd=1340

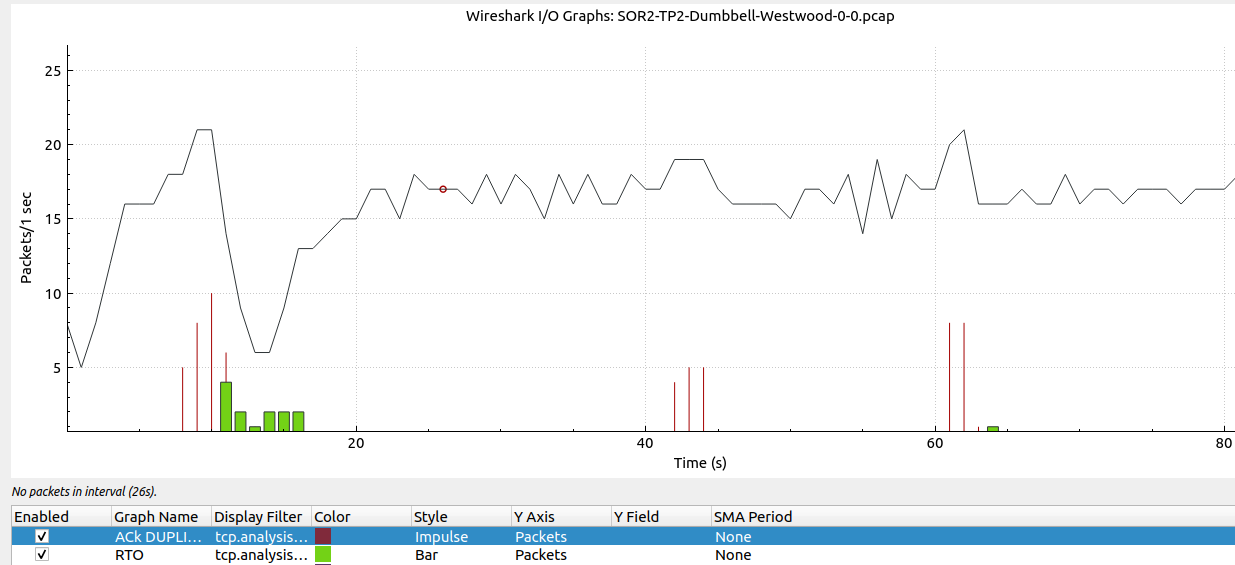


**Comparaciones con Tcp/udp**

Si observamos el grafico a continuación, notamos un incremento de paquetes por segundo al inicio de la conexión, pero un gran pico de caída en cuanto se producen los RTO.Esto se debe a que se vuelve a congestion avoidance de manera mas rápida, y se tarda mas tiempo en alcanzar el umbral de congestión.

 Por otra parte comparando los picos de bajada en la cwnd, podríamos inferir en que la red de westwood entra con menos frecuencia en fastrecovery, y se mantiene mayor tiempo en congestion avoidance.

En consecuencia podríamos decir,basándonos en la red saturada que creamos, que el algoritmo westwood es mas amigable con el resto de los nodos, pero esto tiene el costo de tardar demasiado en retomar al maximo rendimiento.



# Referencias

<https://www.nsnam.org/docs/release/3.30/tutorial/html/index.html/html/index.html>

https://www.nsnam.org/doxygen/dumbbell-animation\_8cc\_source.html

<https://www.ibm.com/docs/ko/psfa/1.6?topic=throughput-tcp-window-size-latency>

http://intronetworks.cs.luc.edu/current2/mobile/ns3.html