Universidad Nacional de General Sarmiento

Sistemas Operativos y Redes II

Trabajo Práctico 2. Análisis de Redes.

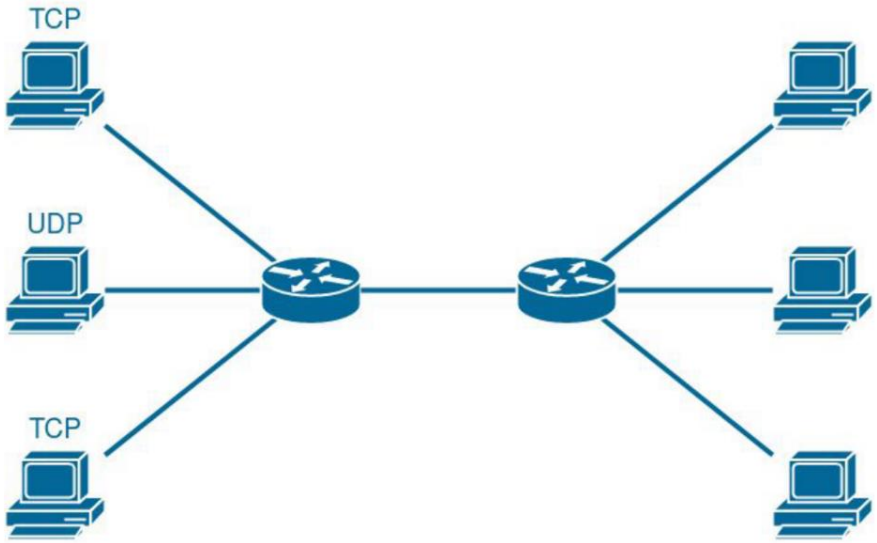
Alumnos: Alan Echabarri - alanechabarri@gmail.com
Juan M. Corbera - jmcorbera@gmail.com
Profesor: Alexis Tcach
Comisión: 1

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
2. 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
3. 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 la congestión, *Slow Start* y *Congestion Avoidance*, regulan la cantidad de paquetes inyectados en 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 la recuperación 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.

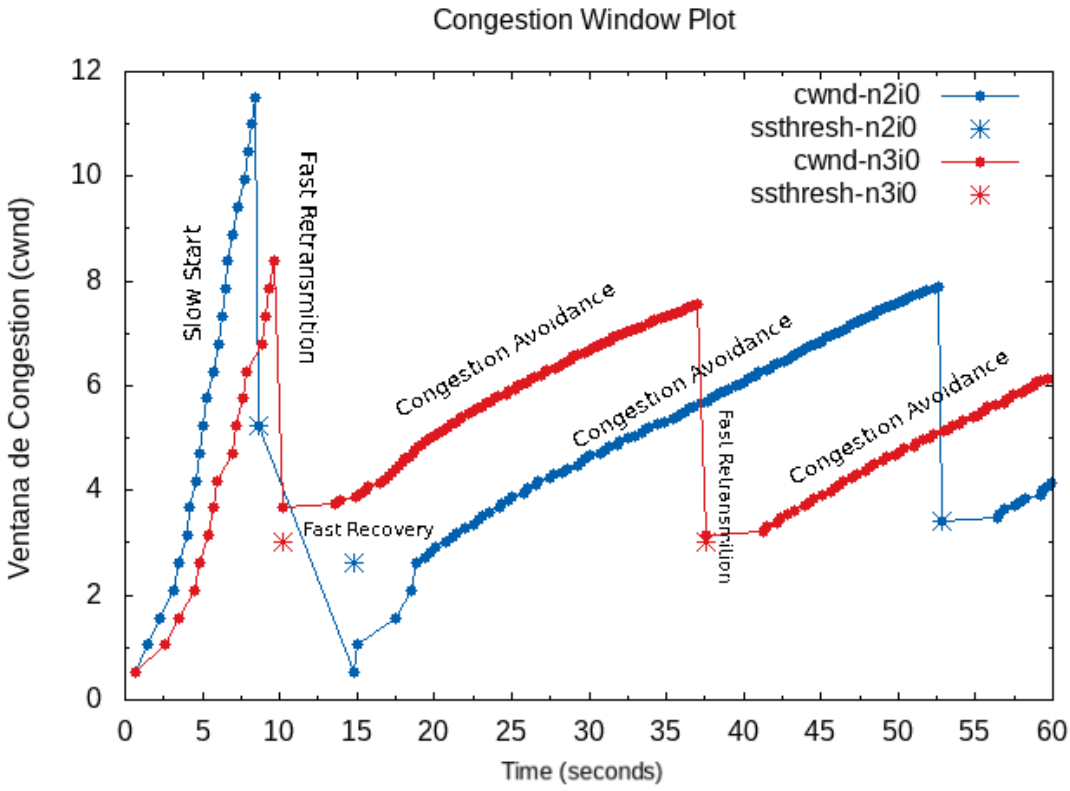


Fig. 1

Velocidad de transferencia

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.

A continuacion describiremos las distintas etapas de TCP desde la perspectiva del nodo 2 interfaz 0.

SLOW START

El algoritmo de Slow Start se utiliza siempre que se esté por debajo del umbral. Inicialmente el valor de la ventana de congestión se establece en 1 y se incrementará en por cada ACK recibido. Como observamos en la Fig., las ventanas de congestión de ambos nodos emisores comienzan en cero, y van incrementando exponencialmente los paquetes que envían por la red. Aproximadamente en el segundo 8,64 el nodo 2, sin que CWND llegara a superar el umbral y tras recibir 3 acks de paquetes duplicados, realiza fast retrasmit y entra inmediatamente en fast recovery.

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).ver fig 4
También se setea CWND = CWND /2 debido al Fast Retransmition (no se espera a un RTO).

Time	Source	Destination	Protoco	Length	Info
8.174079	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=28409 Ack=
8.205546	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=28945 Ack=
8.237013	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=29481 Ack=
8.362879	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=1983
8.362879	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=30017 Ack=
8.394346	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=30553 Ack=
8.425813	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=31089 Ack=
8.459413	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#1] 1000 → 49153 [
8.459413	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=31625 Ack=
8.553813	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#2] 1000 → 49153 [
8.553813	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=32161 Ack=
8.648213	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#3] 1000 → 49153 [
8.648213	10.1.1.1	10.2.2.1	TCP	590	[TCP Fast Retransmission] 49153 →

Fig. 2

Time	Source	Destination	Protoco	Length	Info
8.459413	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#1] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
8.459413	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=31625 Ack=1 Win=131072 Len=536 TS
8.553813	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#2] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
8.553813	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=32161 Ack=1 Win=131072 Len=536 TS
8.648213	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#3] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
8.648213	10.1.1.1	10.2.2.1	TCP	590	[TCP Fast Retransmission] 49153 → 1000 [ACK] Seq=19833 A
9.025813	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#4] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
9.120213	10.2.2.1	10.1.1.1	TCP	62	[TCP Dup ACK 80#5] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
9.499946	10.2.2.1	10.1.1.1	TCP	70	[TCP Dup ACK 80#6] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
9.594346	10.2.2.1	10.1.1.1	TCP	70	[TCP Dup ACK 80#7] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
9.879679	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#8] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
9.974079	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#9] 1000 → 49153 [ACK] Seq=1 Ack=19833 Wi
10.068479	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#10] 1000 → 49153 [ACK] Seq=1 Ack=19833 W
10.068479	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=23049 Ack=1
10.446079	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#11] 1000 → 49153 [ACK] Seq=1 Ack=19833 W
10.446079	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=24657 Ack=1
10.540479	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#12] 1000 → 49153 [ACK] Seq=1 Ack=19833 W
10.540479	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=25193 Ack=1
10.823679	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#13] 1000 → 49153 [ACK] Seq=1 Ack=19833 W
10.823679	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=25729 Ack=1
10.855146	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=26265 Ack=1
10.918079	10.2.2.1	10.1.1.1	TCP	78	[TCP Dup ACK 80#14] 1000 → 49153 [ACK] Seq=1 Ack=19833 W
10.918079	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=28409 Ack=1
10.949546	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=29481 Ack=1
10.981013	10.1.1.1	10.2.2.1	TCP	590	[TCP Retransmission] 49153 → 1000 [ACK] Seq=30017 Ack=1

Fig. 3

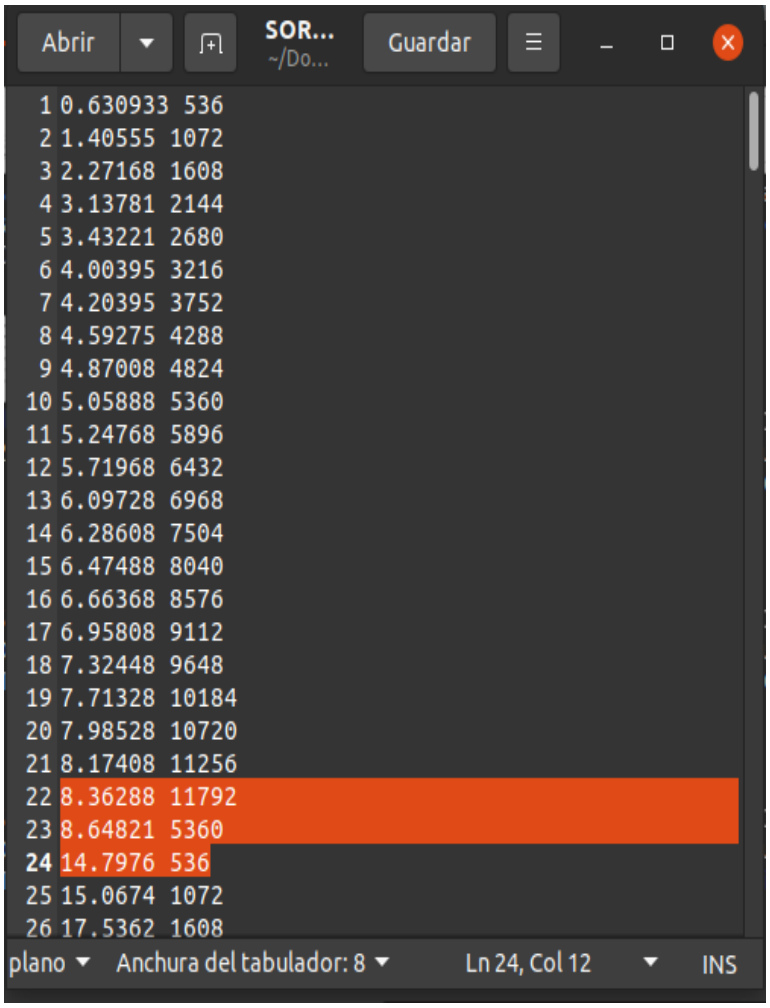


Fig. 4

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). Vea fig 3.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. Vea fig 5.
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, confirmando la teoria.

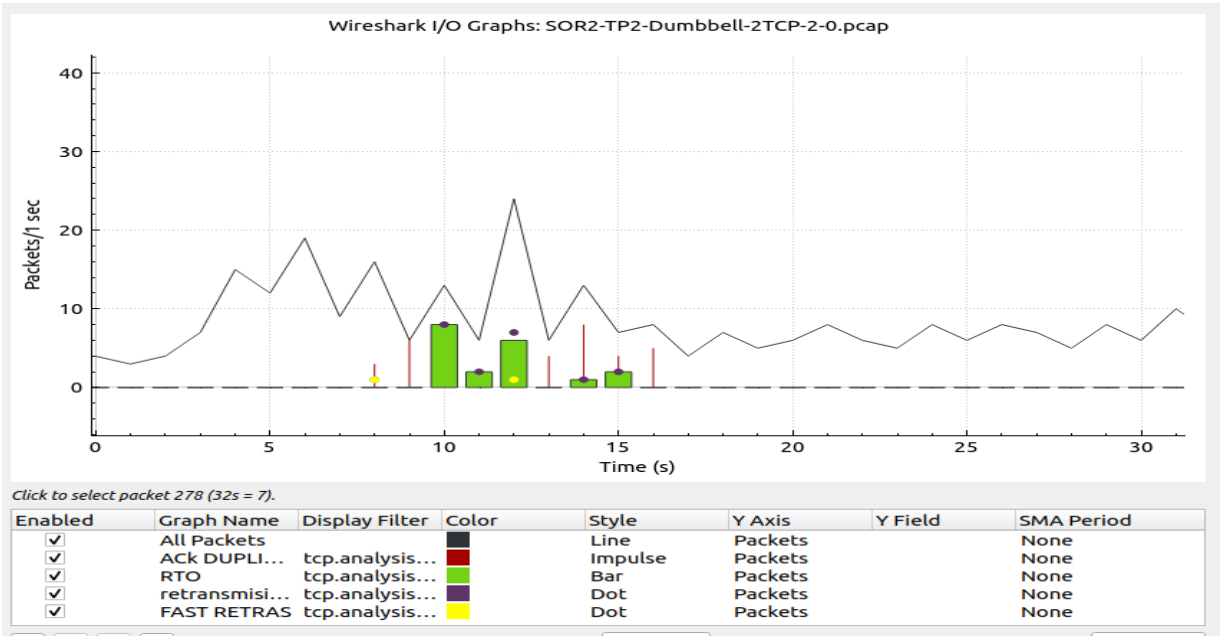


Fig. 5

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

Ssthresh =cwnd/2 = sec :8.64821 bytes:5360

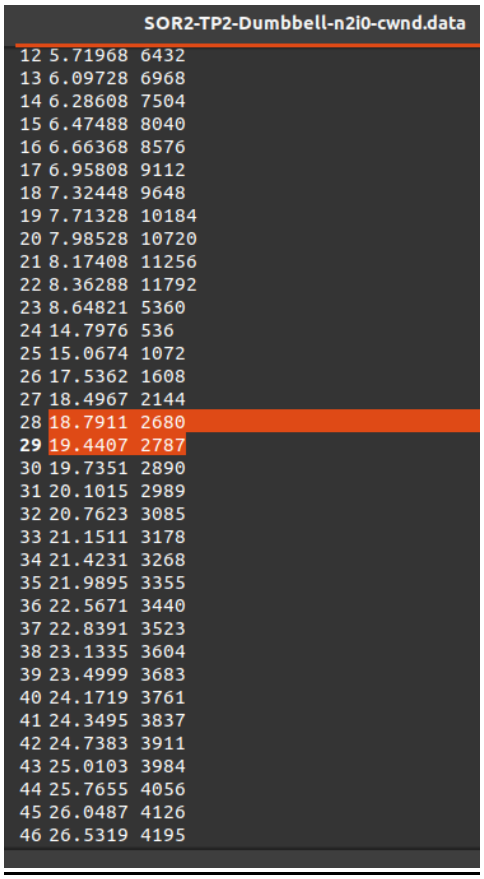
Ssthresh =cwnd/2 = sec :14.7976 bytes: 2680

Ssthresh =cwnd/2 = sec :52.8605 bytes: 3484

SLOW START (otra vez)
Desde el seg 14,79 se ingresa en slow start, debido a un RTO vencido. Y sigue creciendo exponencialmente hasta que supera el ssthresh (2680 bytes) en el segundo 18, desde ese momento comienza Congestion Avoidance.

Congestion Avoidance

Como se puede apreciar en el grafico (fig 1), el incremento del Congesion Windows , es menor que en la anterior etapa. El cwnd va incrementando en cwnd+mss*(mss/cwnd). Esto es en nuestro caso 2680 + 536*(536/2680). Se puede corroborar esta ecuación abriendo el archivo cwnd.data, vea fig 6.



SOR2-TP2-Dumbbell-n2i0-cwnd.data

```
12 5.71968 6432
13 6.09728 6968
14 6.28608 7504
15 6.47488 8040
16 6.66368 8576
17 6.95808 9112
18 7.32448 9648
19 7.71328 10184
20 7.98528 10720
21 8.17408 11256
22 8.36288 11792
23 8.64821 5360
24 14.7976 536
25 15.0674 1072
26 17.5362 1608
27 18.4967 2144
28 18.7911 2680
29 19.4407 2787
30 19.7351 2890
31 20.1015 2989
32 20.7623 3085
33 21.1511 3178
34 21.4231 3268
35 21.9895 3355
36 22.5671 3440
37 22.8391 3523
38 23.1335 3604
39 23.4999 3683
40 24.1719 3761
41 24.3495 3837
42 24.7383 3911
43 25.0103 3984
44 25.7655 4056
45 26.0487 4126
46 26.5319 4195
```

fig 6,congestion avoidance

Finalmente ocurre otro evento de congestion y se vuelve a repetir el ciclo

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 size, 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.

Bandwidth

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

Time	Source	Destination	Protoco	Length	Info
6.474879	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=19833 Ack=1
6.506346	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=20369 Ack=1
6.537813	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=20905 Ack=1
6.663679	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=14473
6.663679	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=21441 Ack=1
6.695146	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=21977 Ack=1
6.726613	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=22513 Ack=1
6.958079	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=15009
6.958079	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=23049 Ack=1
6.989546	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=23585 Ack=1
7.324479	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=16081
7.324479	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=24121 Ack=1
7.355946	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=24657 Ack=1
7.387413	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=25193 Ack=1
7.713279	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=16617
7.713279	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=25729 Ack=1
7.744746	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=26265 Ack=1
7.985279	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=17689
7.985279	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=26801 Ack=1
8.016746	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=27337 Ack=1
8.048213	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=27873 Ack=1
8.174079	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=18761
8.174079	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=28409 Ack=1
8.205546	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=28945 Ack=1
8.237013	10.1.1.1	10.2.2.1	TCP	590	49153 → 1000 [ACK] Seq=29481 Ack=1
8.362879	10.2.2.1	10.1.1.1	TCP	54	1000 → 49153 [ACK] Seq=1 Ack=19833

BW = $\frac{CantPaq*MSS}{RTT+2\cdot\delta}$

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

δ =1

BW = $\frac{19*590}{1.88+2}$ → **BW =** 11,92kb / 3.88 = 3.07kb/s

Aproximadamente 3 kb/s cuando la red está configurada a 50kb 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”

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

Time	Source	Destination	Protoco	Length	Identification	Info
7.704640	10.1.1.1	10.2.2.1	TCP	590	0x0026 (38)	49153 → 1000 [ACK] Seq=19297 Ack=1 Win=131072 Len=536 TSval=6286
7.799040	10.1.1.1	10.2.2.1	TCP	590	0x0028 (40)	[TCP Previous segment not captured] 49153 → 1000 [ACK] Seq=20369
7.893440	10.1.1.1	10.2.2.1	TCP	590	0x0029 (41)	49153 → 1000 [ACK] Seq=20905 Ack=1 Win=131072 Len=536 TSval=6474
7.987840	10.1.1.1	10.2.2.1	TCP	590	0x002a (42)	49153 → 1000 [ACK] Seq=21441 Ack=1 Win=131072 Len=536 TSval=6663
8.365440	10.1.1.1	10.2.2.1	TCP	590	0x002b (43)	49153 → 1000 [ACK] Seq=21977 Ack=1 Win=131072 Len=536 TSval=6663
8.459840	10.1.1.1	10.2.2.1	TCP	590	0x002c (44)	49153 → 1000 [ACK] Seq=22513 Ack=1 Win=131072 Len=536 TSval=6663
8.837440	10.1.1.1	10.2.2.1	TCP	590	0x002e (46)	[TCP Previous segment not captured] 49153 → 1000 [ACK] Seq=23585
8.931840	10.1.1.1	10.2.2.1	TCP	590	0x002f (47)	49153 → 1000 [ACK] Seq=24121 Ack=1 Win=131072 Len=536 TSval=7324
9.215040	10.1.1.1	10.2.2.1	TCP	590	0x0034 (52)	[TCP Previous segment not captured] 49153 → 1000 [ACK] Seq=26801
9.309440	10.1.1.1	10.2.2.1	TCP	590	0x0035 (53)	49153 → 1000 [ACK] Seq=27337 Ack=1 Win=131072 Len=536 TSval=7985
9.403840	10.1.1.1	10.2.2.1	TCP	590	0x0036 (54)	49153 → 1000 [ACK] Seq=27873 Ack=1 Win=131072 Len=536 TSval=7985
9.781440	10.1.1.1	10.2.2.1	TCP	590	0x0038 (56)	[TCP Previous segment not captured] 49153 → 1000 [ACK] Seq=28945
9.875840	10.1.1.1	10.2.2.1	TCP	590	0x003b (59)	[TCP Previous segment not captured] 49153 → 1000 [ACK] Seq=30553
10.159040	10.1.1.1	10.2.2.1	TCP	590	0x003d (61)	[TCP Previous segment not captured] 49153 → 1000 [ACK] Seq=31625
10.253440	10.1.1.1	10.2.2.1	TCP	590	0x003e (62)	49153 → 1000 [ACK] Seq=32161 Ack=1 Win=131072 Len=536 TSval=8553
10.347840	10.1.1.1	10.2.2.1	TCP	590	0x003f (63)	[TCP Out-Of-Order] 49153 → 1000 [ACK] Seq=19833 Ack=1 Win=131072
11.197440	10.1.1.1	10.2.2.1	TCP	590	0x0040 (64)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=23049 Ack=1 Win=1310
11.386240	10.1.1.1	10.2.2.1	TCP	590	0x0041 (65)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=24657 Ack=1 Win=1310
11.480640	10.1.1.1	10.2.2.1	TCP	590	0x0042 (66)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=25193 Ack=1 Win=1310
11.575040	10.1.1.1	10.2.2.1	TCP	590	0x0043 (67)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=25729 Ack=1 Win=1310
11.669440	10.1.1.1	10.2.2.1	TCP	590	0x0044 (68)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=26265 Ack=1 Win=1310
11.763840	10.1.1.1	10.2.2.1	TCP	590	0x0045 (69)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=28409 Ack=1 Win=1310
11.858240	10.1.1.1	10.2.2.1	TCP	590	0x0046 (70)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=29481 Ack=1 Win=1310
11.952640	10.1.1.1	10.2.2.1	TCP	590	0x0047 (71)	[TCP Retransmission] 49153 → 1000 [ACK] Seq=30017 Ack=1 Win=1310
12.047040	10.1.1.1	10.2.2.1	TCP	590	0x0048 (72)	[TCP Spurious Retransmission] 49153 → 1000 [ACK] Seq=23049 Ack=1
12.141440	10.1.1.1	10.2.2.1	TCP	590	0x0049 (73)	49153 → 1000 [ACK] Seq=32697 Ack=1 Win=131072 Len=536 TSval=1101
12.802240	10.1.1.1	10.2.2.1	TCP	590	0x004a (74)	[TCP Spurious Retransmission] 49153 → 1000 [ACK] Seq=24657 Ack=1
12.896640	10.1.1.1	10.2.2.1	TCP	590	0x004b (75)	49153 → 1000 [ACK] Seq=33233 Ack=1 Win=131072 Len=536 TSval=1186

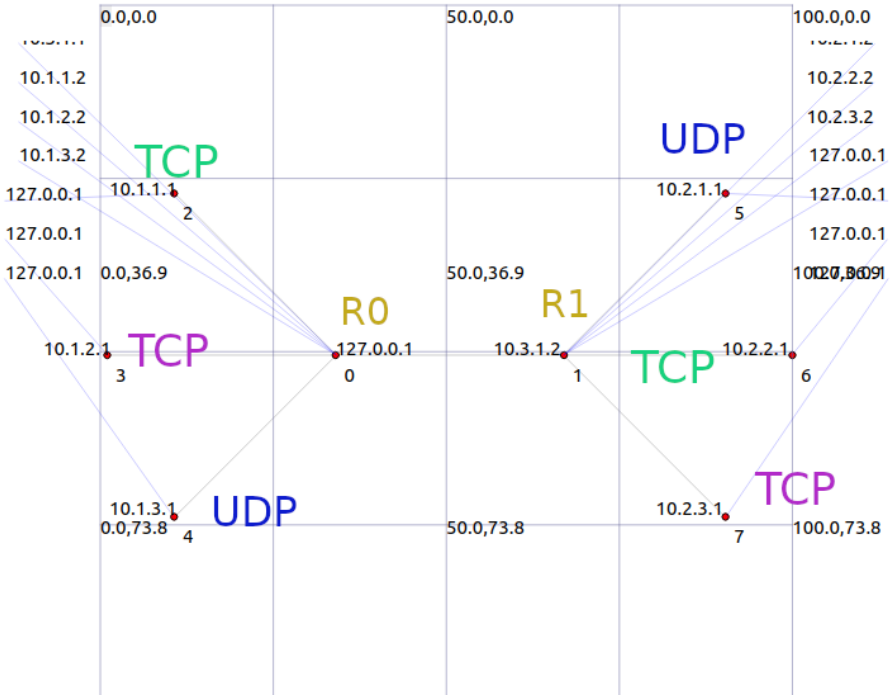
En la imagen anterior se puede ver que los paquetes del emisor n2i0 al receptor n6i0, se observan múltiples paquetes que no llegan.Si prestamos atencion a la columna identificacion, se ve que los paquetes 39,45,48,49,50,51,55 y 60 entre otros, brillan por su ausencia. Esta es la causa de los mensajes 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.

Prueba con 2 emisores TCP y uno UDP

TCP distintas acciones del protocolo

Ahora vamos a analizar el comportamiento de la red incorporando 1 emisor udp y un receptor UDP. UDP permite enviar datos sin establecer conexión previa, no tiene control de flujo ni confirmaciones (acks).

Continuamos con el siguiente esquema de red:



esquema de red

Para empezar, comparamos los paquetes que informa el nodo emisor con los que recibe el nodo receptor. Observamos que hay paquetes que se envían, pero desde el otro punto de no se reciben, pero como se esperaba, UDP no hace nada al respecto, y continúa enviando paquetes. Vea capturas de wireshark a continuación

Time	Source	Destination	Protoco	Length	Info	Identification
0.000000	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0000 (0)
0.086720	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0001 (1)
0.173440	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0002 (2)
0.260160	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0003 (3)
0.346880	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0004 (4)
0.433600	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0005 (5)
0.520320	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0006 (6)
0.607040	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0007 (7)
0.711040	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0008 (8)
0.892160	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0009 (9)
0.978880	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000a (10)
1.065600	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000b (11)
1.152320	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000c (12)
1.329600	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000e (14)
1.416320	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000f (15)
1.503040	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0010 (16)
1.589760	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0012 (18)
1.676480	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0015 (21)
1.763200	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0018 (24)
1.849920	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x001d (29)
1.936640	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x001f (31)
2.023360	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0020 (32)
2.110080	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0021 (33)
2.196800	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0023 (35)
2.283520	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0026 (38)
2.370240	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x002a (42)

Time	Source	Destination	Protoco	Length	Info	Identification
0.000000	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0000 (0)
0.028907	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0001 (1)
0.057814	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0002 (2)
0.086720	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0003 (3)
0.115627	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0004 (4)
0.144534	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0005 (5)
0.173440	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0006 (6)
0.202347	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0007 (7)
0.231254	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0008 (8)
0.260160	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0009 (9)
0.289067	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000a (10)
0.317974	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000b (11)
0.346880	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000c (12)
0.375787	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000d (13)
0.404694	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000e (14)
0.433600	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x000f (15)
0.462507	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0010 (16)
0.491414	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0011 (17)
0.520320	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0012 (18)
0.549227	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0013 (19)
0.578134	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0015 (21)
0.607040	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0016 (22)
0.635947	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0017 (23)
0.664854	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x0018 (24)
0.693760	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x001a (26)
0.722667	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x001b (27)
0.751574	10.1.3.1	10.2.1.1	UDP	542	49153 → 1000 Len=512	0x001d (29)

Ahora visualizaremos desde el nodo 0 interfaz 0, (router a router) la cantidad de paquetes TCP, comparado con la cantidad de UDP. Desde un gráfico que nos ofrece wireshark en static i/o.vea fig 7

Como se puede apreciar, UDP (azul), a pesar de ser solo un nodo emisor, comienza con una gran cantidad de paquetes por segundo, acaparando gran parte de la red sin tener consideración por el resto de nodos.

En cambio, TCP incrementa exponencialmente su tasa de paquetes por segundo, iniciando por 1, hasta el punte en que se encuentra con 3 ACKS duplicado, aproximadamente segundo 10, entonces comienza con su protocolo de Congestion Avoidance. Hasta ese momento el pico de UDP iba cayendo porque TCP se estaba ocupando, con sus 2 nodos una buena porción de red. Estas bajadas no significan que UDP este utilizando alguna clase de algoritmo para enviar menos paquetes, sino que directamente que se envían y no están llegando a destino.

Continuando con el análisis, vemos que cada vez que TCP baja su CWND, UDP concreta más paquetes. Incluso en el segundo 40, aproximadamente UDP llega a superar la frecuencia de transmisión de los 2 nodos TCP. Finalmente observamos que en el segundo 50, (cuando la aplicación se detiene) UDP deja de emitir instantáneamente paquetes, sin importar si llegaron todos a destinos o en orden, mientras que de parte de TCP se siguen emitiendo los paquetes restantes, para obtener la información completa y en orden.

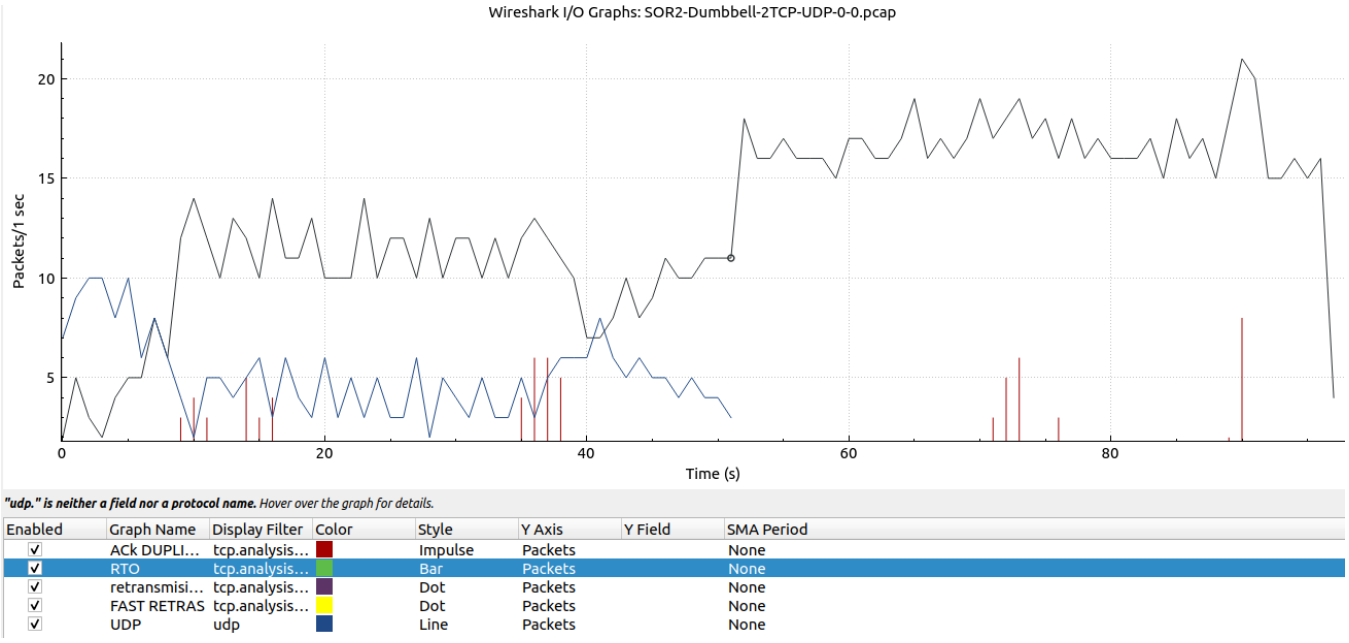


fig 7, paquees/s wireshark

Etapas tcp

Ahora analizaremos las diferentes instancias del protocolo, pero esta vez incorporamos el nodo UDP. Vea fig 8

Como observamos desde la congestion windows, los dos nodos TCP inician slow start, hasta el segundo 10, cuando ocurre un evento de 3 ACKS duplicados.

En ese momento se hace fast retransmit y fast recovery, pero otra vez la CWND se actualiza solo cada RTT, y el RT es demasiado largo.

La diagonal que vemos a continuación es el evento de Congestion Avoidance, que se da cuando el CWND supera el SSTHRESH (asteriscos), entonces la CWND incrementa su valor en fracciones de CWND.

A continuación, repite las etapas.

Podemos destacar en el grafico del CWND, después de que el nodo UDP deja de transmitir, (segundo 60), se libera ancho de banda, por lo cual a partir de ese momento los dos nodos TCP deberían incrementar su tasa de paquetes (en bytes) por segundo, sin embargo, el incremento no es exponencial, ya que en ese momento se encontraban en Congestion Avoidance. Esto se debe a que el SSTHRESH <= CWND.

A continuación ocurre un evento de congestión y se inicia Slow Start otra vez, sí bien ya se liberó ancho de banda del emisor UDP, TCP ya había seteado el SSTHRESH demasiado bajo, por lo que el Slow Start dura demasiado poco y se inicia Congestion Avoidance nuevamente.

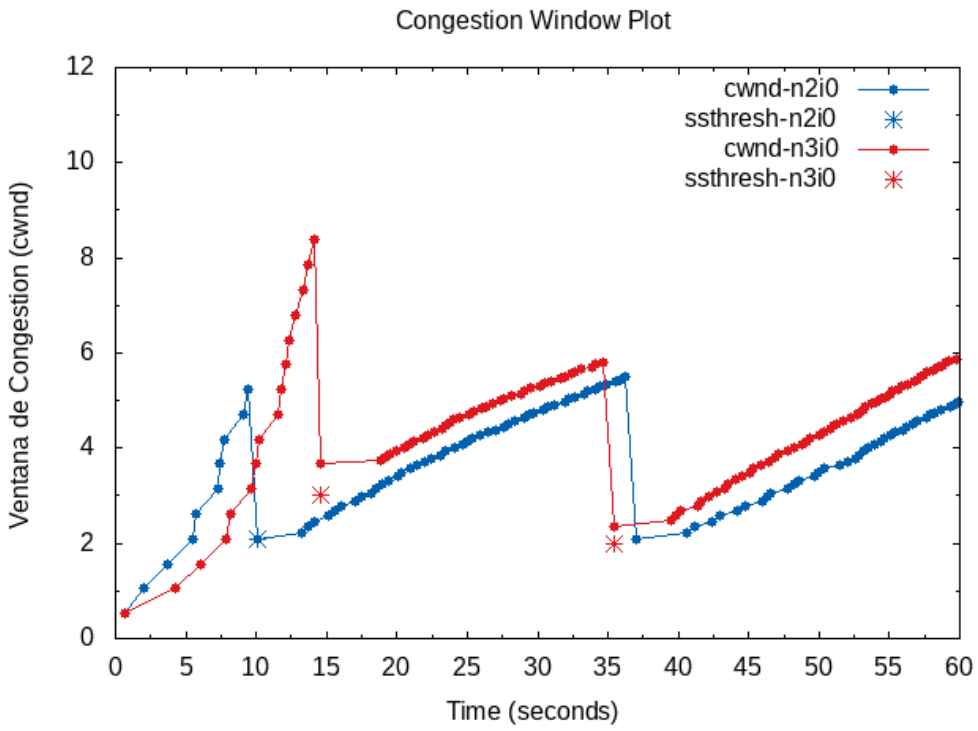


Fig. 8

Uso justo del canal

A continuación, mostraremos las diferencias entre el uso justo del canal por parte de TCP, y el uso desmedido por parte de UDP.

Comparamos desde wireshark, utilizando la herramienta de fusión, combinamos los dos archivos pcap del primer caso (solo dos emisores TCP) Y observamos la red configurada a unos 150 kbs con un cuello de botella de 50 kbs. (vea figura 9 TCP emisores)

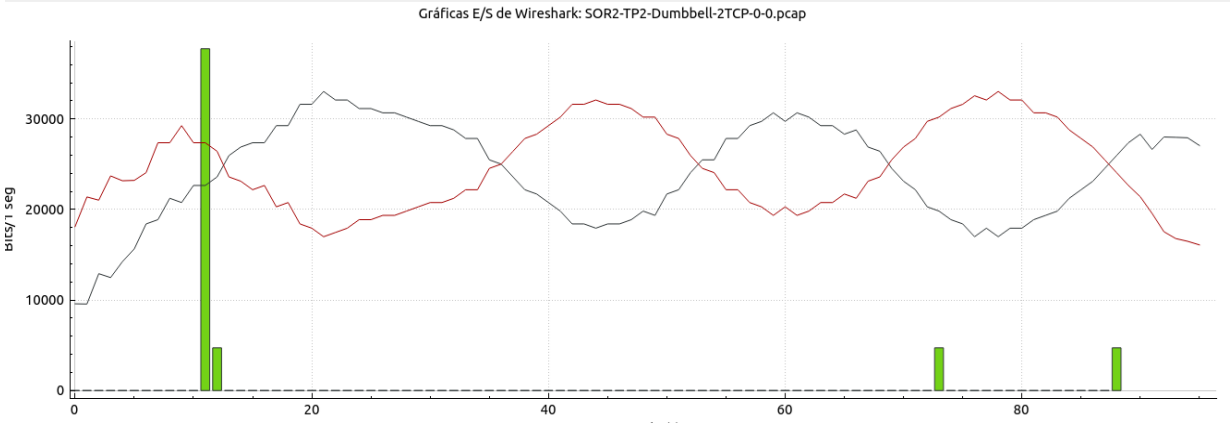
Figure 9: tcp emisores caso 1(solo dos tcp)

Los dos emisores TCP venían incrementando el uso del canal, hasta que ocurre un evento de congestión, debido al cuello de botella. Nótese que a partir de la línea amarilla (FastRetrasmit) Los dos emisores TCP cambian rotundamente su uso del canal, y comienzan su mecanismo contra congestión. A partir de este momento, la suma de los bits/s de los dos emisores TCP no superara los 50kb/s.

Podemos visualizar esto mismo desde la interfaz Router-Router, sin embargo, como en esta misma es donde se da el cuello de botella, no veremos nunca a los nodos TCP superar entre ellos 2 los 50/kbs, (lo que si vimos anteriormente antes del FastRetrasmit). Vea fig 10.

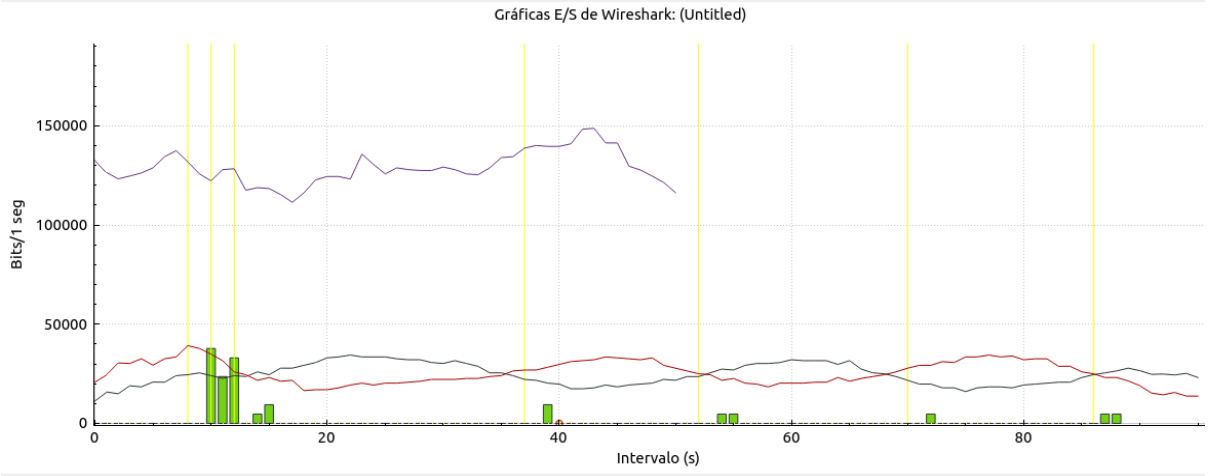
Tomando en cuanta todo lo visto anteriormente, ahora vamos a observar cómo UDP hace uso desmedido del canal, sin considerar los demás nodos. Y como TCP se adapta a la situación de la red.

Figure 10: TCP desde interfaz Router-Router.



Desde la interfaz de los emisores, hemos fusionado los 3 archivos pcap para visualizar cómo se comporta cada emisor con respecto al ancho del canal. Recordemos, que en esta interfaz se tiene unos 150kb/s. Vea fig11.

Figura 11: TCP Y UDP emisores.

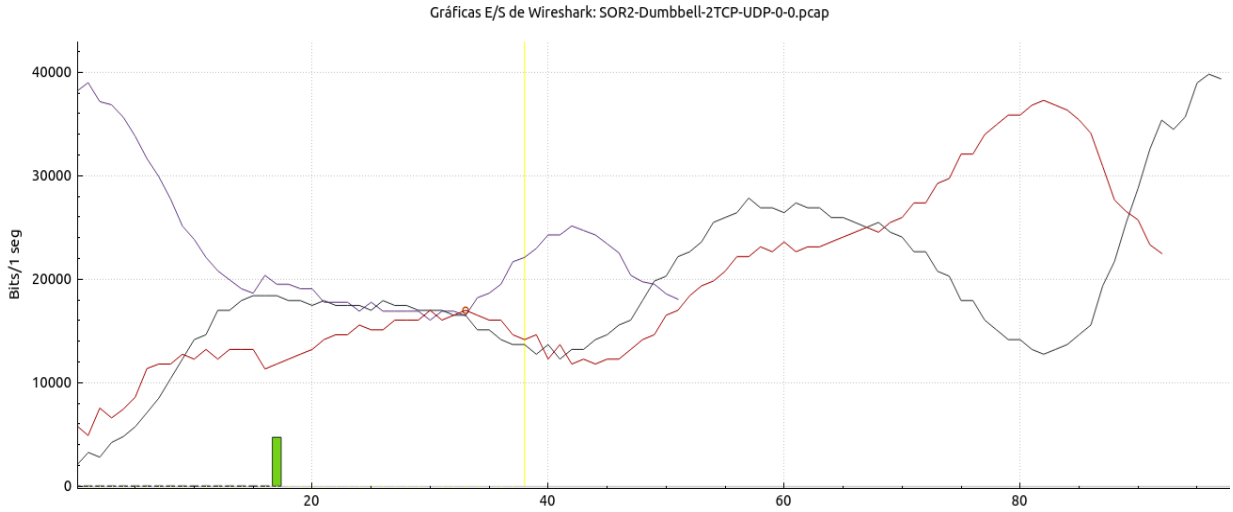


En este gráfico, desde la interfaz de emisores, podemos visualizar el emisor UDP (azul), emite ocupando la mayor parte de la red posible, sin tener en cuenta el cuello de botella que se le avecina. Mientras que los emisores TCP van incrementando su uso del canal, hasta el evento de congestión (Fast retransmit línea amarilla) Desde ese momento la suma de los emisores TCP no superara los 50kb/s. Podríamos decir que TCP “se entera que de que hay un cuello de botella más adelante, y por eso toma medidas, reduciendo su uso del canal”.

Finalmente, el emisor UDP deja de transmitir, entonces ¿si UDP libero el canal, porque los emisores TCP no usan una mayor parte de esos 150kbs? ¿Por qué se queda con una suma menor a 50kbs? Como ya habíamos mencionado, TCP “sabe” que, en alguna otra parte de la red, hay congestión, por lo tanto, hace uso justo del canal, y no congestiona la red.

Ahora analizaremos lo que sucede desde la interfaz R-R.

Figure 12 TCP Y UDP interfaz Router-Router



Lo que observamos es como UDP, (línea azul) comienza con una gran porción del canal, mientras los nodos TCP (negro y rojo) incrementan gradualmente su uso. Aproximadamente en el segundo 15, suceden los eventos de congestión. Por lo cual, los dos nodos TCP comienzan a regular su uso, para ser más justos con el resto de nodos. Durante un tiempo vemos que los 3 nodos comparten aproximadamente la misma porción de la red.16/kbs.

Finalmente, el emisor UDP deja de transmitir, y los nodos TCP comienzan a ocupar una mayor porción de red, siempre teniendo en cuenta de no superar los 50kbs entre ambos nodos.

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.

Westwood independientemente de la red puede mejorar significativamente la eficiencia de la transferencia de datos en redes propensas a errores. Westwood hace una modificación del lado del emisor sobre parte del algoritmo new reno, para que se estime el ancho de banda disponible en la red. La idea es monitorear la tasa de ACKS y utilizarla para calcular la CWND y el umbral de congestión.

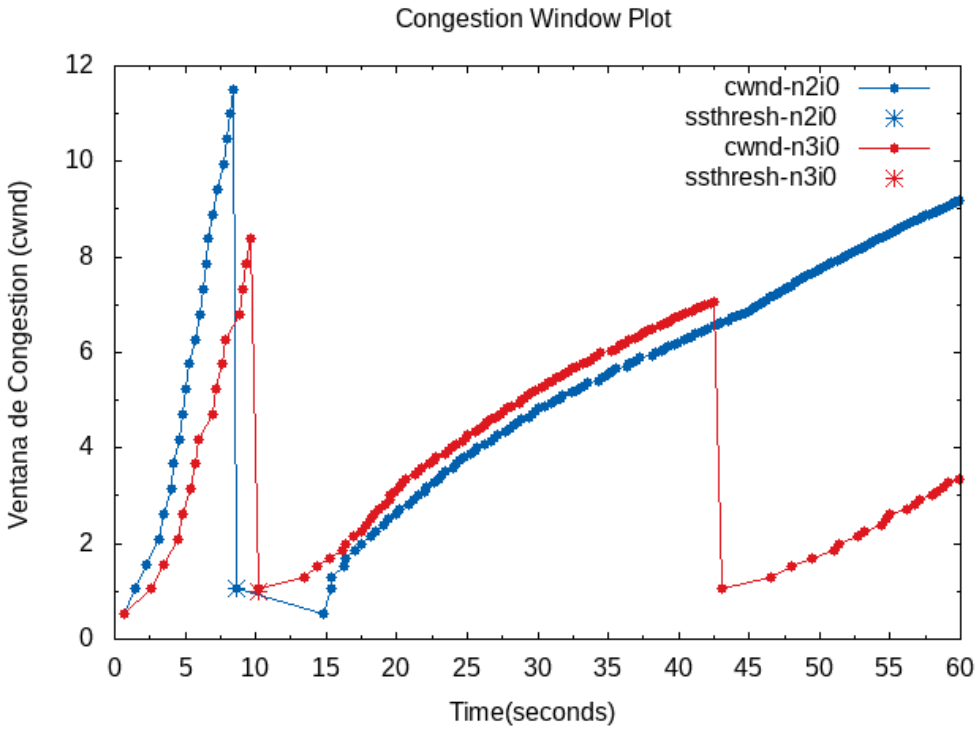
Un seudocódigo sencillo para comprender como se calcula el Bandwidth es:

```
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=número de segmentos reconocidos por el ultimo ACK.
pkt_size=tamaño del segmento en bytes.
now= hora actual
lastacktime= hora en que se recibió 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 congestión Windows del n2i0 y n3i0(insertar plot)



Slow start

Como se observa en el gráfico, la Congestion Windows va incrementando exponencialmente 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 proporción 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 ACKs duplicados, SSTHRESH se setea en 1072, y como CWND por ese entonces era mayor a SSTHRESH, también se setea en 1072.

1	0.630933	536
2	1.40555	1072
3	2.27168	1608
4	3.13781	2144
5	3.43221	2680
6	4.00395	3216
7	4.20395	3752
8	4.59275	4288
9	4.87008	4824
10	5.05888	5360
11	5.24768	5896
12	5.71968	6432
13	6.09728	6968
14	6.28608	7504
15	6.47488	8040
16	6.66368	8576
17	6.95808	9112
18	7.32448	9648
19	7.71328	10184
20	7.98528	10720
21	8.17408	11256
22	8.36288	11792
23	8.64821	1072
24	14.7976	536
25	15.3351	1072
26	15.3351	1340
27	16.2935	1554
28	16.3879	1738
29	17.0674	1903
30	17.5485	2053
31	18.1159	2192
32	18.5202	2323
33	19.0765	2446
34	19.3597	2563
35	19.9426	2675

Si ocurre un RTO, pasa lo siguiente:

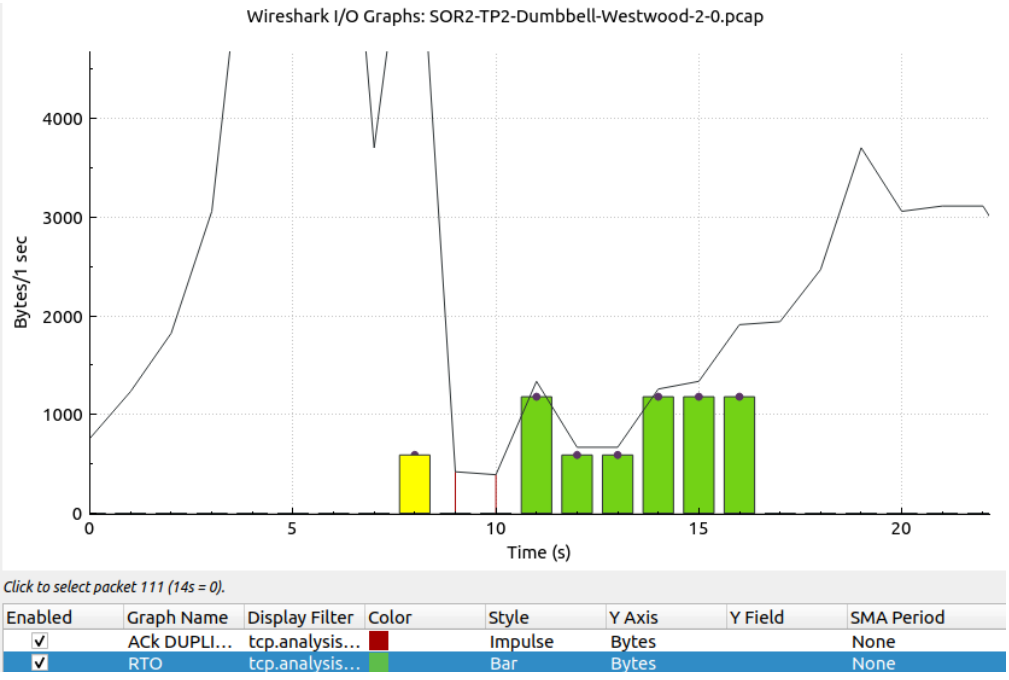
```
if (coarse timeout expires)
ssthresh = (BWE*RTTmin)/seg_size;
 if (ssthresh < 2)
ssthresh = 2;
endif;
cwin = 1;
```

Lo que entendemos es que CWND se setea en 1 y SSTHRESH vuelve a setearse como el algoritmo anterior, sin embargo esta vez, si SSTHRESH es menor a 2 CWND, SSTHRESH se setea en 2 CWND. 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 1 CWND =536, desde ahí hace un mini Slow Start, hasta que supera al SSTHRESH y comienza Congestion Avoidance.

Desde aquí no hay mucho que decir, se utiliza el algoritmo de Congestion Avoidance anteriormente mostrado, hasta que ocurran 3 ACKs duplicados, o haya otro RTO.

cwnd+mss*(mss/cwnd).
cwnd=1072+536*536/1072

cwnd=1340

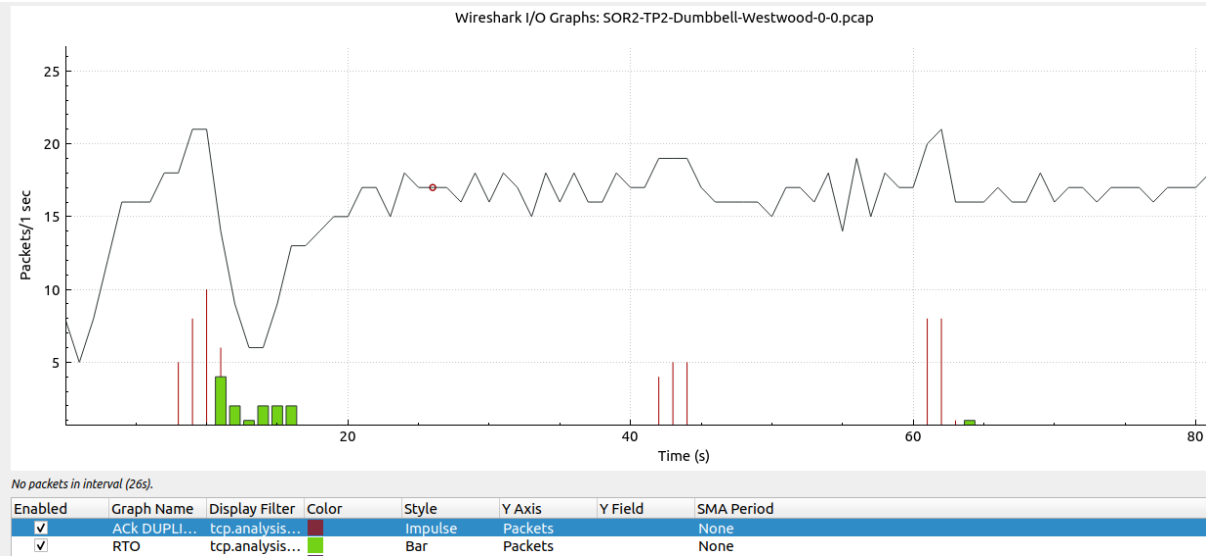


Comparaciones con TCP / UDP

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 produce los RTO. Esto se debe a que se vuelve a Congestion Avoidance de manera más rápida, y se tarda más 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 fast recovery, y se mantiene mayor tiempo en Congestion Avoidance.

En consecuencia, podríamos decir, basándonos en la red saturada que creamos, que el algoritmo Westwood es más amigable con el resto de los nodos, pero esto tiene el costo de tardar demasiado en retomar al máximo rendimiento. Debido a que calcula un ancho de banda bajo, y setea el umbral en relación a esa estimación.



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

https://ocw.unican.es/pluginfile.php/268/course/section/183/tema_04.pdf

http://www.signal.uu.se/Research/PCCWIP/Mobicom01Mascolo.pdf