Práctica: Capa de Transporte, Control de Congestión en TCP

Matías Robles [*] Andres Barbieri [**]

20 de abril de 2009

1. Configurar con el simulador [NS-2] una topología como la mostrada en la figura 1. Asegúrese que el enlace intermedio tenga menor BW y mayor DELAY que los enlaces de los extremos. Configurar un nodo de recepción y otro de transmisión de tráfico UDP a un data rate constante, uno en cada extremo. Configurar otros dos nodos en primera instancia para que utilice el algoritmo TCP Básico: Tahoe. Luego cambiar a Reno o New-Reno. Comparar como se comportan mirando el throughput alcanzado, y los parámetros de control de congestión: $SSTH$ y $CWND$.

   a) Indicar en que lugar están en la etapa de SS (Slow Start) y cuando en CA (Congestion Avoidance).

   b) Comparar cual obtiene mejor rendimiento.

   c) ¿Qué sucede con la Ventana de Congestión, $CWND$, y con el SS Threshold, $SSTH$ en cada algoritmo?

   d) ¿Qué sucede si se arranca con un $SSTH$ bajo (e.g. 10), analizar con el NS-2?

   e) Comparar el rendimiento de Tahoe, Reno y New-Reno suprimiendo los datos UDP.

   f) Comparar el rendimiento de Tahoe, Reno y New-Reno aumentando el data rate de los datos UDP (e.g 0.2Mbps).

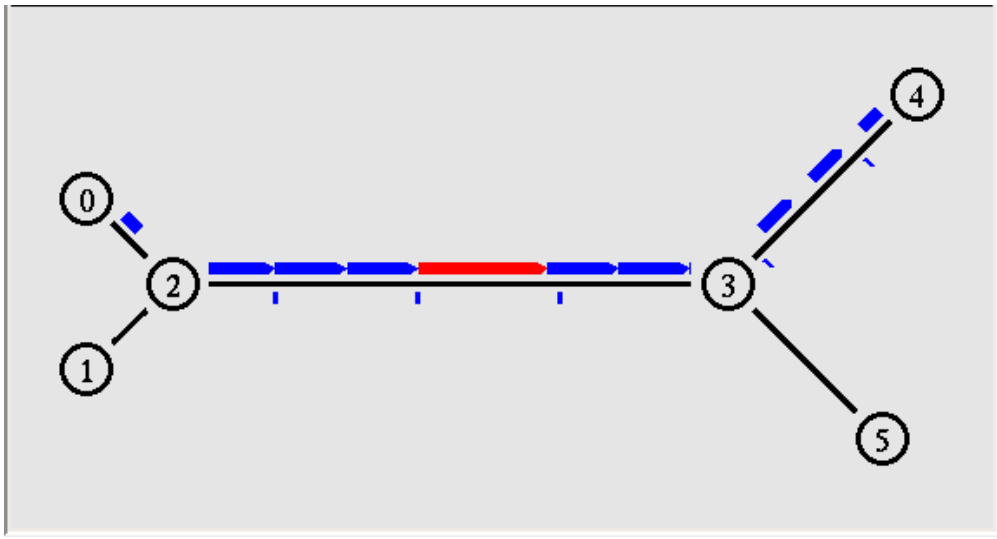

Figura 1: Esquema de la toplogía 1.

A continuación se muestra un ejemplo del script a ejecutar con NS-2.

[*] mrobles@info.unlp.edu.ar
[**] barbieri@cespi.unlp.edu.ar

```
? cat tcp_tahoe.tcl
#!/opt/ns-allinone-2.31/bin/ns

## Configura el prefijo de los nombres de los archivos de salida
set str_outfile "out-tcp_tahoe"

## Crea un objeto Simulador
set ns [new Simulator]

## Define diferentes colores para ser procesados por el NAM
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
$ns color 4 Yellow
$ns color 5 Black
$ns color 6 Chocolate

############## Open Files ##################
## Abre el archivo de salida a ser procesado por NAM
set file_nam [open $str_outfile.nam w]
$ns namtrace-all $file_nam
## Abre el archivo de traza
set file_tr [open $str_outfile.tr w]
$ns trace-all $file_tr
## Abre el archivo donde se almacenará la evolución de la CWND
set file_win [open $str_outfile.win w]
## Abre el archivo donde se almacenará la evolución de la SSTH
set file_ssth [open $str_outfile.ssth w]
#############################################

#############################################
## Define un procedimiento 'finish'
proc finish {} {
        global file_nam file_tr ns str_outfile
        $ns flush-trace
        #Close the trace file
        close $file_nam
        close $file_tr
        #close $file_win
        #close $file_ssth
        set str_nam  "$str_outfile.nam"
        set str_win  "$str_outfile.win"
        set str_ssth "$str_outfile.ssth"
        set str_tr   "$str_outfile.tr"
        exec ./winssth2png.sh $str_win $str_ssth
        exec ./thr2png.sh $str_tr
        # Execute nam on the trace file
        # exec nam $str_nam &
        exit 0
}
#############################################

## Crea 6 nodos
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
```

```
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

## Crea enlaces entre nodos
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns simplex-link $n2 $n3 0.3Mb 100ms DropTail
$ns simplex-link $n3 $n2 0.3Mb 100ms DropTail
$ns duplex-link $n3 $n4 0.5Mb 40ms DropTail
$ns duplex-link $n3 $n5 0.5Mb 30ms DropTail

## Indica las posiciones para ser usadas por NAM
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns simplex-link-op $n2 $n3 orient right
$ns simplex-link-op $n3 $n2 orient left
$ns duplex-link-op $n3 $n4 orient right-up
$ns duplex-link-op $n3 $n5 orient right-down

## Configura el tamaño de cola en el enlace (n2-n3)
$ns queue-limit $n2 $n3 20

## Configura la conexión TCP

## Crea un emisor y un recpetor para
## transferencias BULK.
# Para correr Reno, New-Reno o Vegas cambiar por:
# set tcp [new Agent/TCP/Reno]
# set tcp [new Agent/TCP/Newreno]
# set tcp [new Agent/TCP/Vegas]

# TCP-Tahoe
set tcp [new Agent/TCP]

$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink/DelAck]
$ns attach-agent $n4 $sink
$ns connect $tcp $sink
$tcp set fid_ 1
## Configura SSTH inicial
$tcp set window_ 8000
$tcp set packetSize_ 552

# Configura FTP sobre TCP
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

#Crea una conexión UDP
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n5 $null
$ns connect $udp $null
$udp set fid_ 2
```

```
## Configura CBR (Constant Bit Rate) sobre UDP
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 0.01mb
$cbr set random_ false

## Indica cuando comienzan y cuando terminan
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 124.0 "$ftp stop"
$ns at 124.5 "$cbr stop"

## Define e inicia invocación de procedimientos de
## generación de evolución de CWND y SSTH

proc plotWindow {tcpSource file} {
global ns
set time 0.1
set now [$ns now]
set cwnd [$tcpSource set cwnd_]
set wnd [$tcpSource set window_]
puts $file "$now $cwnd"
$ns at [expr $now+$time] "plotWindow $tcpSource $file"
}
$ns at 0.1 "plotWindow $tcp $file_win"

proc plotSSthresh {tcpSource file} {
global ns
set time 0.1
set now [$ns now]
set ssthresh [$tcpSource set ssthresh_]
puts $file "$now $ssthresh"
$ns at [expr $now+$time] "plotSSthresh $tcpSource $file"
}
$ns at 0.1 "plotSSthresh $tcp $file_ssth"

### MAIN()
$ns at 160.0 "finish"
$ns run

### EOF ###
```

Se deben agregar los siguientes scripts en el mismo directorio del anterior para realizar el procesamiento de las salidas. Para esto se requiere `awk(P)`, `perl(1)`, `bash(1)`, y `gnuplot(1)` [Siever]:

```
? cat throughput.pl
#!/usr/bin/perl

##
## Copied from: NS Simulator for Beginers - Sophia Antipolis, France-2004
##
```

```
# type: perl throughput.pl <trace file> <required node> <granlarity> \
# >  output file

$infile=$ARGV[0];
$tonode=$ARGV[1];
$granularity=$ARGV[2];

#we compute how many bytes were transmitted during time interval specified
#by granularity parameter in seconds
$sum=0;
$clock=0;

      open (DATA,"<$infile")
        || die "Can't open $infile $!";

    while (<DATA>) {
             @x = split(' ');

#column 1 is time
if ($x[1]-$clock <= $granularity)
{
#checking if the event corresponds to a reception
if ($x[0] eq 'r')
{
#checking if the destination corresponds to arg 1
if ($x[3] eq $tonode)
{
#checking if the packet type is TCP
if ($x[4] eq 'tcp')
{
    $sum=$sum+$x[5];
}
}
}
}
else
{   $throughput=$sum/$granularity;
    print STDOUT "$x[1] $throughput\n";
    $clock=$clock+$granularity;
    $sum=0;
}
}
   $throughput=$sum/$granularity;
    print STDOUT "$x[1] $throughput\n";
    $clock=$clock+$granularity;
    $sum=0;

    close DATA;
exit(0);
```

```
? cat winssth2png.sh
#!/bin/bash

if [ $# -lt 2 ]
then
        echo "usage $0 file1.win file2.ssth"
```

```
        exit 1
fi

echo -e " set style data linespoints \n set xlabel \"time (seconds)\" \n \
       set ylabel \"Segments (cwnd)\" \n set terminal png \n set output  \
       \"$1.png\" \n plot \"$1\" using 1:2 title \"snd_cwnd\", \"$2\"    \
       using 1:(\$2>=8000 ? 0 : \$2) title \"snd_ssth\" " | \
       gnuplot -persist
exit 0

### EOF ###
```

```
? cat thr2png.sh
#!/bin/bash

if [ $# -lt 1 ]
then
        echo "usage $0 file1.tr"
        exit 1
fi

perl ./throughput.pl $1 4 1 > $1.thr
echo -e " set style data linespoints \n set xlabel \"time (seconds)\" \n  \
       set ylabel \"Throughput (Mbps)\" \n set terminal png \n set output \
       \"$1.png\" \n plot \"$1.thr\" using 1:(\$2/(1000*1000))            \
       title \"throughput\" " | \
       gnuplot -persist
exit 0
```

Luego ejecutar con los siguientes comandos:

```
? export PATH=$PATH:/opt/ns-allinone-2.31/bin
? ns tcp_tahoe.tcl
```

Este comando genera los gráficos requeridos en formato PNG. Si se desea analizar la evolución se puede ejecutar NAM de la siguiente forma.

```
? export PATH=$PATH:/opt/ns-allinone-2.31/bin
? nam out-tcp_tahoe.nam
```

Para obtener la cantidad de MB (Mega bytes) transferidos ejecutar, por ejemplo:

```
? awk 'BEGIN{COUNT=0}{COUNT=COUNT+$2}END{print COUNT/(1000*1000)}' \
  out-tcp_tahoe.tr.thr
3.88178
```

2. Configurar en paralelo dos sesiones Reno-Reno y en otro ejemplo Reno-Tahoe en lugar de utilizar UDP.

a) Comparar contra cual Reno obtiene mejor rendimiento. ¿Por qué le parece que sucede esto?

Para la sesiones TCP-TCP cambiar en el código anterior, la sesión UDP por otra TCP, por ejemplo:

```
...
set tcp2 [new Agent/TCP/Reno]

## Crea otro emisor y otro recpetor para
## transferencias BULK.
$ns attach-agent $n1 $tcp2
set sink2 [new Agent/TCPSink/DelAck]
$ns attach-agent $n5 $sink2
$ns connect $tcp2 $sink2
$tcp2 set fid_ 2
$tcp2 set window_ 8000
$tcp2 set packetSize_ 552

set ftp2 [new Application/FTP]
$ftp2 attach-agent $tcp2
$ftp2 set type_ FTP

$ns at 0.1 "$ftp2 start"
$ns at 1.0 "$ftp start"
$ns at 124.0 "$ftp stop"
$ns at 124.5 "$ftp2 stop"
...
```

3. Volver a configurar UDP y en paralelo una sesión utilizando TCP Vegas. Volver UDP al valor inicial (e.g. 0.01Mbps).

   a) Comparar como se comportan contra las pruebas anteriormente realizadas de Tahoe y Reno.

   b) ¿Cuál obtiene un mejor rendimiento de los tres algoritmos?

4. Configurar en paralelo dos sesiones Vegas-Vegas y Vegas-Reno en lugar de utilizar UDP.

   a) Comparar como se comporta Vegas contra Reno y como Vegas contra Vegas.

   b) ¿Qué conclusiones se pueden obtener a partir de estas pruebas?

5. Cambiar los parámetros de los enlaces de forma de triplicar o cuadriplicar el delay y duplicar el ancho de banda. Asegurarse que los parámetros de los enlaces locales sean mejores que el troncal. Dejar el tráfico UDP.

   a) Probar Reno y ver como se comporta con estas nuevas condiciones.

   b) Probar Vegas y ver como se comporta con estas nuevas condiciones. Subir el tiempo de corrida.

   c) ¿Cuál funciona mejor, Qué conclusiones se pueden obtener?

   Ejemplo del cambio en el enlace troncal:

```
...
## Crea enlaces entre nodos
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns simplex-link $n2 $n3 0.6Mb 300ms DropTail
```

```
$ns simplex-link $n3 $n2 0.6Mb 300ms DropTail
$ns duplex-link $n3 $n4 2Mb 40ms DropTail
$ns duplex-link $n3 $n5 2Mb 30ms DropTail
...
```

6. Configurar con el simulador NS-2 una nueva topología como la mostrada en la figura 2. En la misma se adicionan dos nodos entre los cuales el enlace se configura con pérdida y un delay variable.

   a) Probar sobre la topología anterior los algoritmos de Reno, New-Reno y Vegas (Creo que Vegas con LOSS no muestra los resultados como deberían).

   b) Cambiar el receptor para que no utilice Delayed Ack y ver si mejora o no, con respecto a Reno o New-Reno.

   c) A la configuración de receptor de Delayed Ack y sin Delayed Ack, adicionar en ambos extremos SACK (Selective Acknowledge) y ver si se obtiene una mejora.

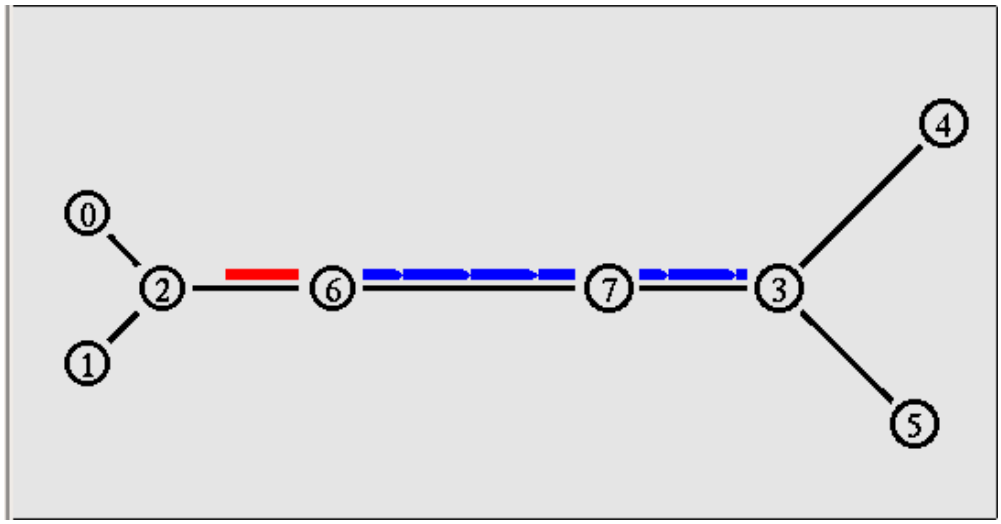

Figura 2: Esquema de la toplogía 2.

A continuación se muestra parte de la configuración de un script para la nueva topología.

```
...
## Crea 6 nodos
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
set ndelay0 [$ns DelayBox]
set ndelay1 [$ns DelayBox]

## Crea enlaces entre nodos
$ns duplex-link  $n0 $n2 2Mb  10ms DropTail
$ns duplex-link  $n1 $n2 2Mb  10ms DropTail

$ns duplex-link  $n2 $ndelay0 0.3Mb 25ms DropTail
```

```
$ns simplex-link $ndelay0  $ndelay1 0.3Mb 50ms DropTail
$ns simplex-link $ndelay1  $ndelay0 0.3Mb 50ms DropTail

$ns duplex-link  $n3 $ndelay1 0.3Mb 25ms DropTail

$ns duplex-link  $n3 $n4 0.5Mb 40ms DropTail
$ns duplex-link  $n3 $n5 0.5Mb 30ms DropTail

## Indica las posiciones para ser usadas por NAM
$ns duplex-link-op  $n0      $n2      orient right-down
$ns duplex-link-op  $n1      $n2      orient right-up

$ns duplex-link-op  $n2      $ndelay0 orient right
$ns simplex-link-op $ndelay0 $ndelay1 orient right
$ns simplex-link-op $ndelay1 $ndelay0 orient left
$ns duplex-link-op  $n3      $ndelay1 orient left

$ns duplex-link-op  $n3 $n4 orient right-up
$ns duplex-link-op  $n3 $n5 orient right-down

## Configura el tamaño de cola en el enlace (n2-n3)
$ns queue-limit $n2 $ndelay0 20

## Configura parámetros del link intermedio
#BW
set sender_bw [new RandomVariable/Constant]
$sender_bw set val_ 0Mb; #0.3Mb
set recvr_bw [new RandomVariable/Constant]
$recvr_bw set val_  0Mb; #0.3Mb

#Delay
#set sender_delay [new RandomVariable/Constant]
#$sender_delay set val_ 0ms; # 100ms
set sender_delay [new RandomVariable/Uniform]
$sender_delay set min_  10
$sender_delay set max_  50
#set recvr_delay [new RandomVariable/Constant]
#$recvr_delay set val_ 0ms;  # 100ms
set recvr_delay [new RandomVariable/Uniform]
$recvr_delay set min_  10
$recvr_delay set max_  50

#Loss Rate
#set loss_rate [new RandomVariable/Constant]
#$loss_rate set val_ 0
set loss_rate [new RandomVariable/Uniform]
$loss_rate set min_ 0
$loss_rate set max_ 0.01

#Configura reglas para DelayBoxes
$ndelay0 add-rule [$n0 id] [$n4 id] $sender_delay $loss_rate $recvr_bw
$ndelay1 add-rule [$n4 id] [$n0 id] $recvr_delay  $loss_rate $sender_bw
...
```

Aquí se muestran los cambios para Deshabilitar DelACK y luego para habilitar SACK en ambos

lados.

```
...
## set sink [new Agent/TCPSink/DelAck]
set sink [new Agent/TCPSink]
...
```

```
...
set tcp [new Agent/TCP/Sack1]
...
set sink [new Agent/TCPSink/Sack1]
...
```

7. Configurar con el simulador NS-2 con el parche para soportar las implementaciones de TCP de GNU/Linux. Usando la primera topología, como la mostrada en la figura 1, configurar para que utilice CUBIC, Hybla y Scalable TCP.

   a) ¿Cómo es el rendimiento obtenido con respecto a las pruebas anteriores, a qué se puede deber esto?. Comparar con New-Reno.

   b) Agregar SACK, Timestamps y soporte de RFC-1323 y volver a realizar las pruebas. Comparar contra Vegas.

   c) Configurar ahora la topología de la figura 2 y realizar las pruebas. ¿Cual obtiene mejor rendimiento? (Las mediciones de Vegas e Hybla con pérdida parecen no ser correctas debido a la simulación).

   A continuación se muestra parte del script para utilizar las implementaciones provistas por GNU/Linux.

```
...
set tcp [new Agent/TCP/Linux]
$ns at 0 "$tcp select_ca $tcp_ca" }

set tcp [new Agent/TCP/Sack1]
...
set sink [new Agent/TCPSink/Sack1]
#set sink [new Agent/TCPSink/DelAck]
...

$sink set ts_echo_rfc1323_ true
$sink set timestamps_ true
```

1. Gráficos

Se debe tener en cuenta que en los gráficos el throughput esta indicado en Mbps (Megabits por segundo), pero en realidad esta medido en MBps (Mega Bytes por segundo).

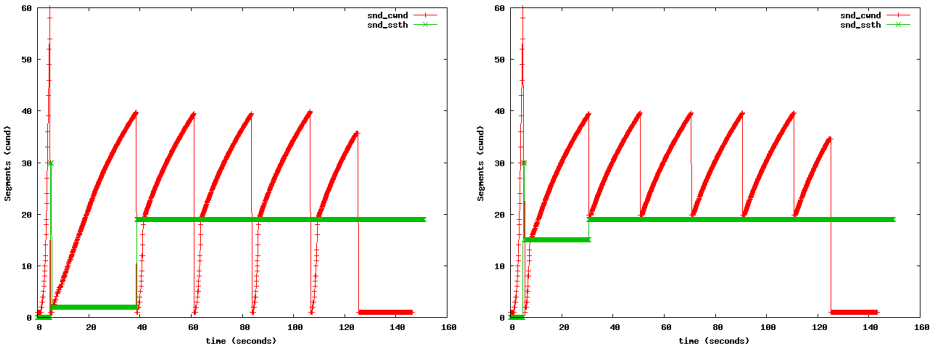

Figura 3: Evolución de CWND y SSTH para Tahoe y Reno compitiendo con tráfico UDP (New-Reno se comporta igual que Reno).

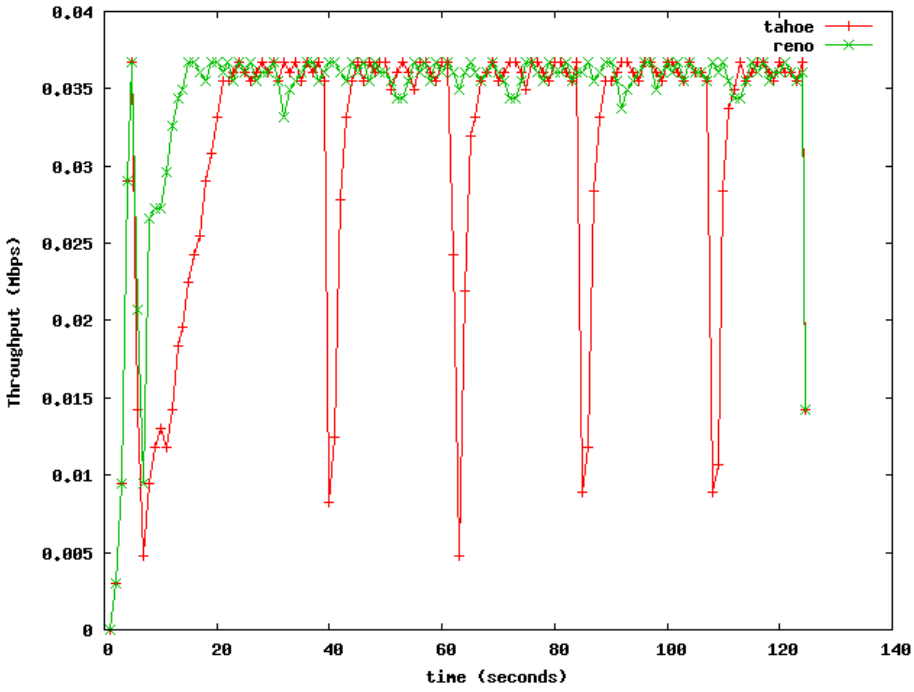

Figura 4: Evolución del Throughput para Tahoe y Reno compitiendo con tráfico UDP (New-Reno se comporta igual que Reno).

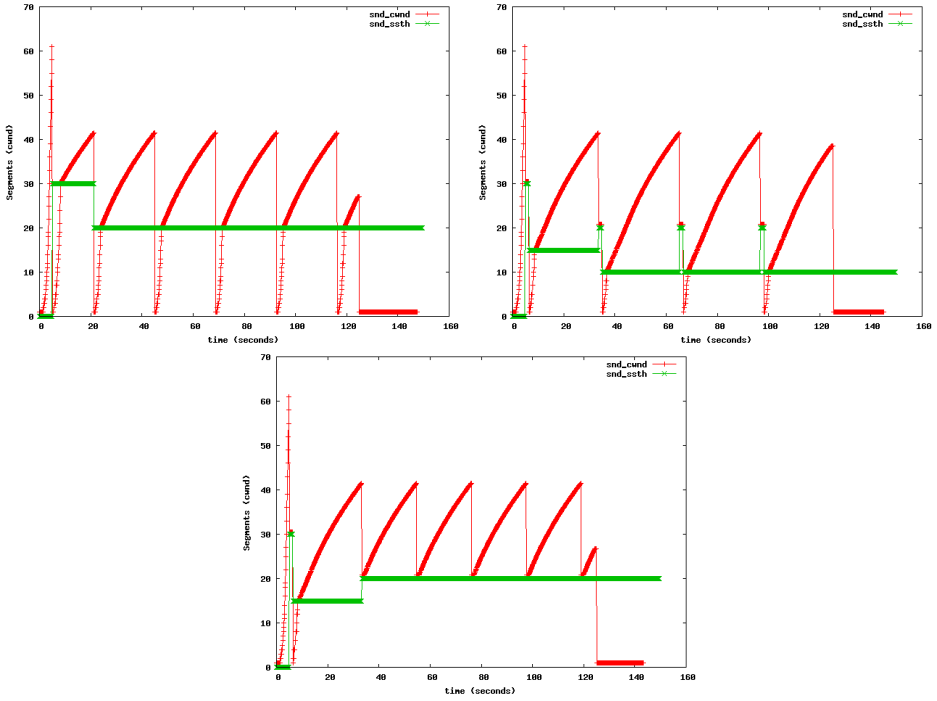

Figura 5: Evolución de CWND y SSTH para Tahoe, Reno y New Reno sin competencia de tráfico (Reno parece tener un rendimiento peor).

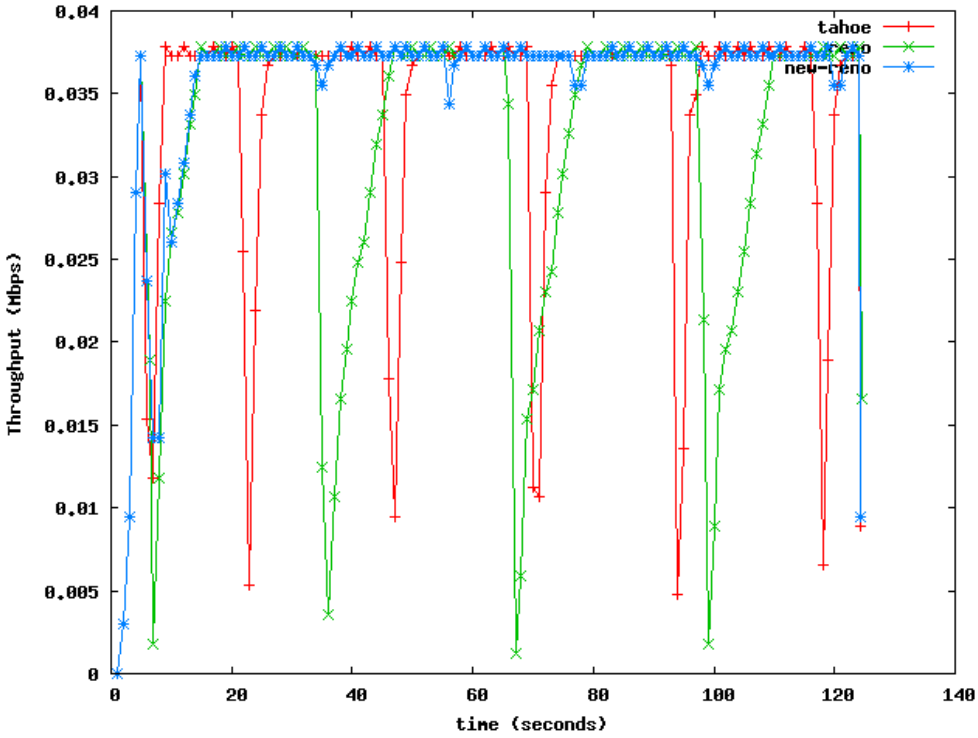

Figura 6: Evolución del Throughput para Tahoe, Reno y New-Reno sin competencia de tráfico (Reno parece tener un rendimiento peor).

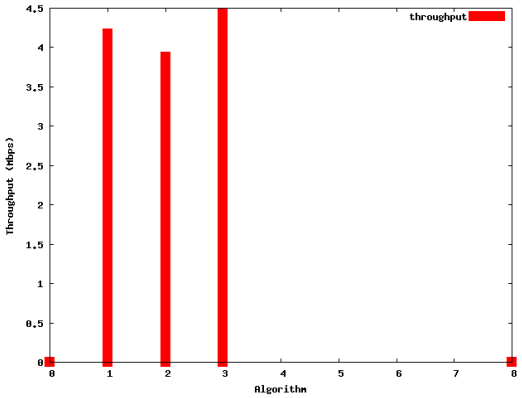

Figura 7: Comparativa de evolución del Throughput sin tráfico UDP, para las tres implementaciones. Ver tabla 1.

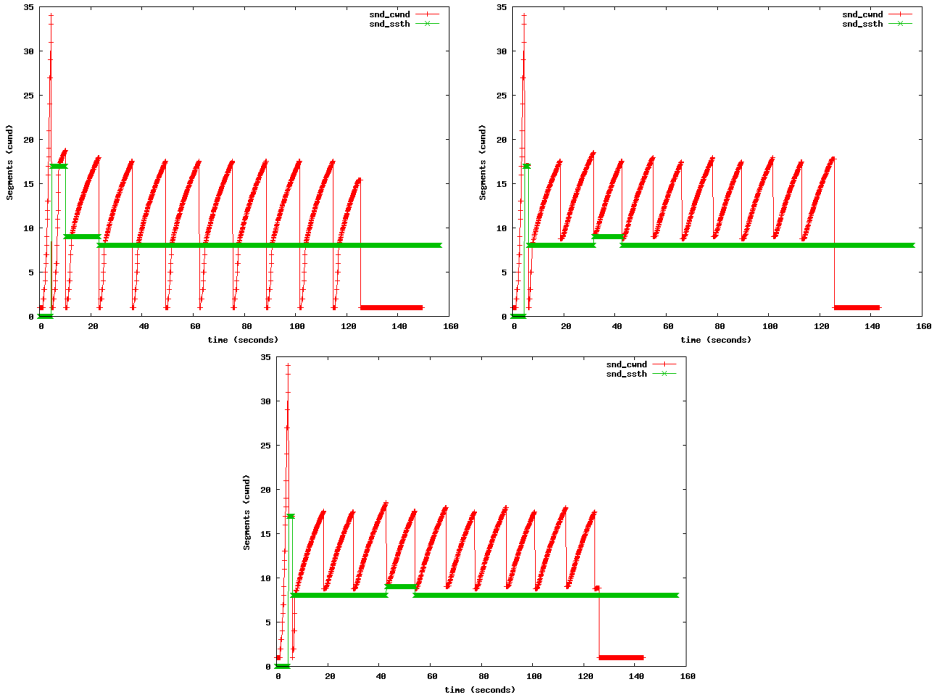

Figura 8: Evolución de CWND y SSTH para Tahoe, Reno y New Reno con competencia de mayor tráfico UDP. (New-Reno se comporta igual que Reno).

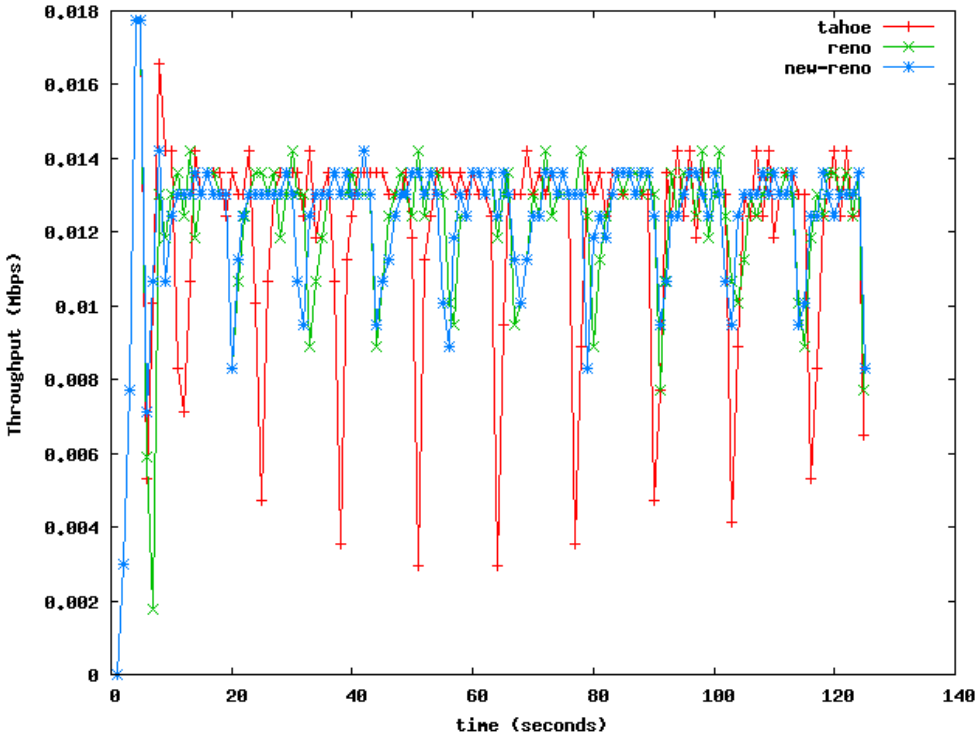

Figura 9: Evolución del Throughput para Tahoe, Reno y New-Reno con competencia de mayor tráfico UDP.

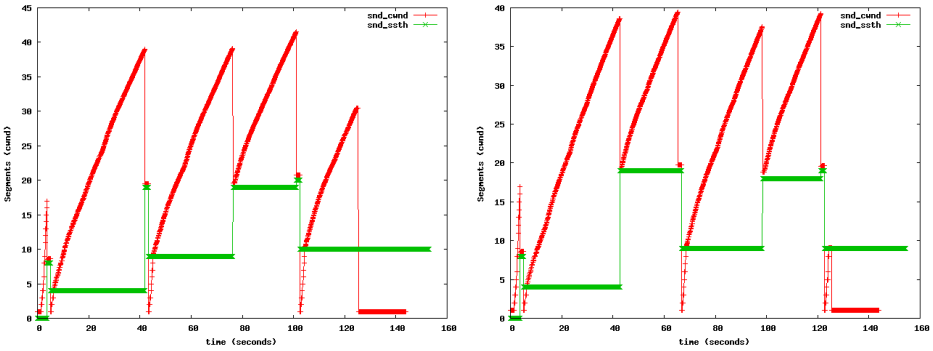

Figura 10: Evolución de CWND y SSTH para Reno vs. Reno y Reno vs. Tahoe (con competencia de dos sesiones TCP, midiendo la primera).

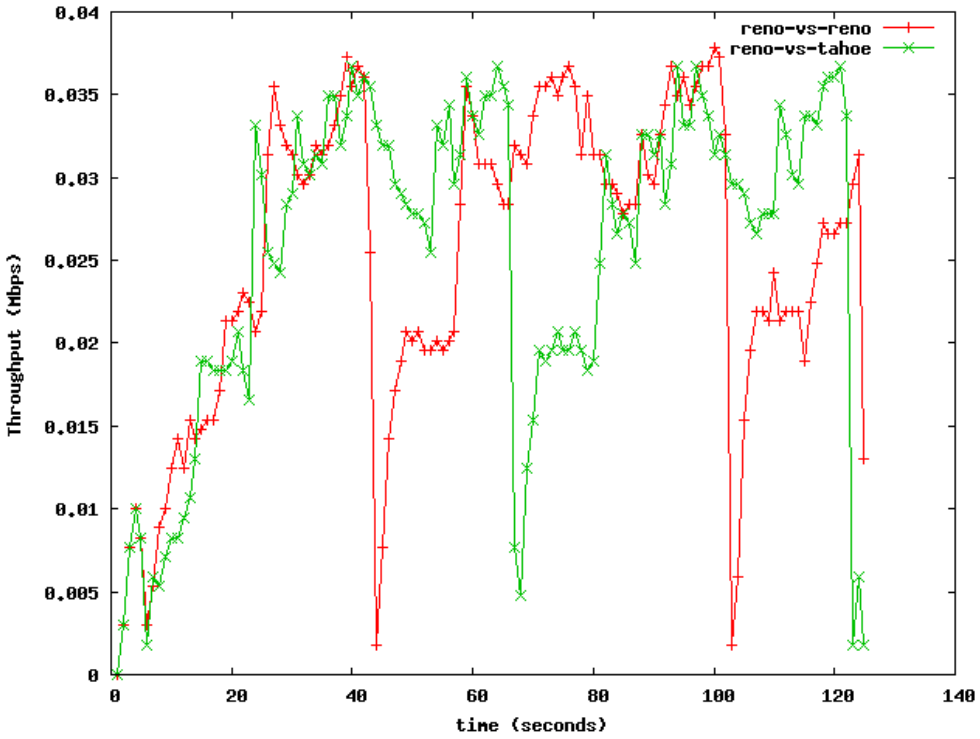

Figura 11: Evolución del Throughput para Reno vs. Reno y Reno vs. Tahoe (con competencia de dos sesiones TCP, midiendo la primera. Reno vs. Reno se equilibran, sobre Tahoe, Reno obtiene ventaja).

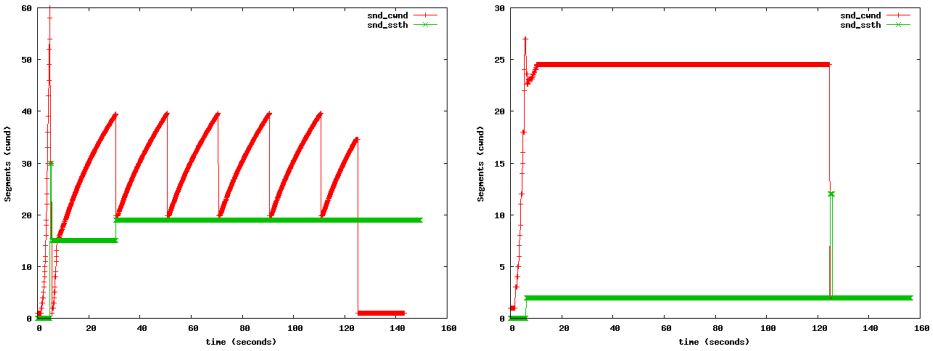

Figura 12: Evolución de CWND y SSTH para Reno y Vegas con competencia de algo de tráfico UDP (Vegas muestra ser muy superior).

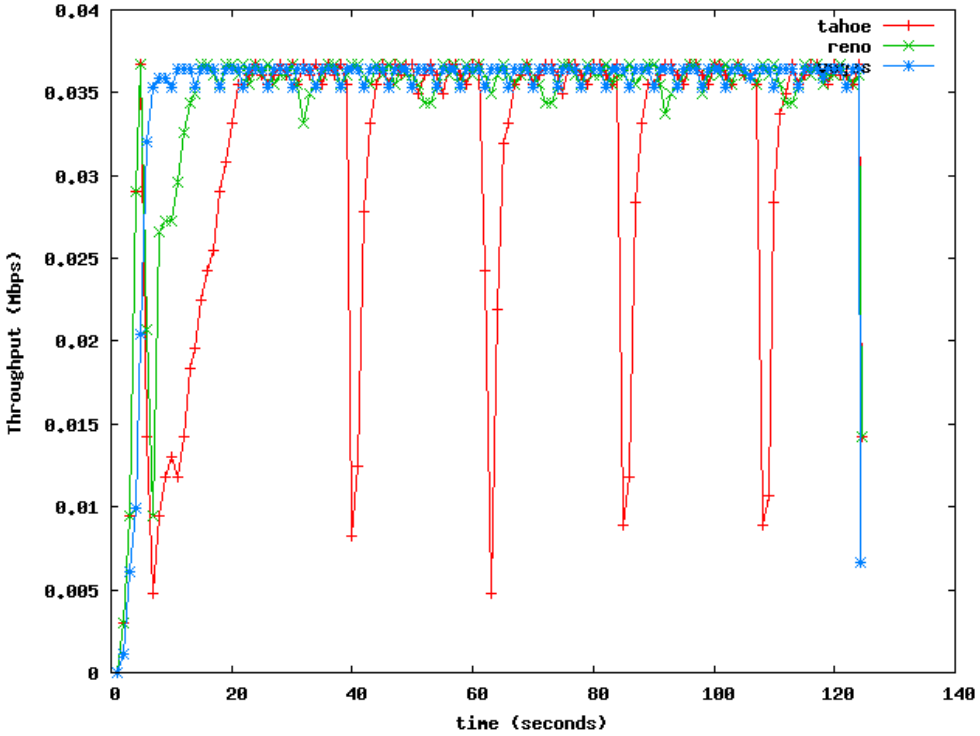

Figura 13: Comparativa de evolución del Throughput para Tahoe, Reno y Vegas con competencia de algo de tráfico UDP (Vegas muestra ser superior).

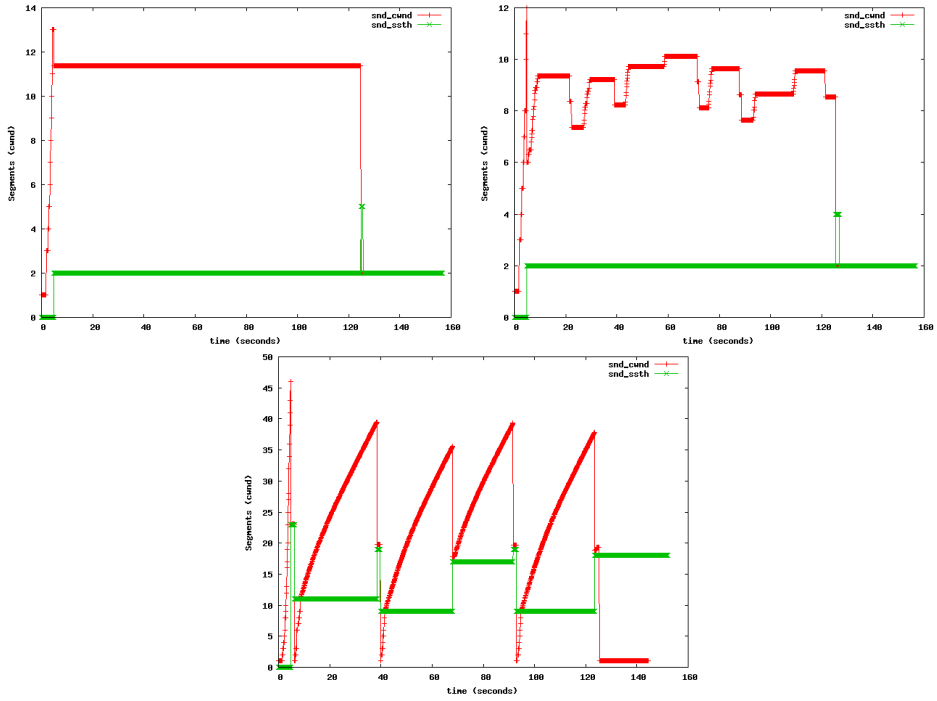

Figura 14: Evolución de CWND y SSTH para Vegas vs. Vegas, Vegas vs. Reno y Reno vs. Vegas, con competencia de dos sesiones TCP, midiendo la primera (Vegas "pierde terreno" sobre Reno (algoritmo poco friendly). Vegas es considerado sobre otras sesiones).

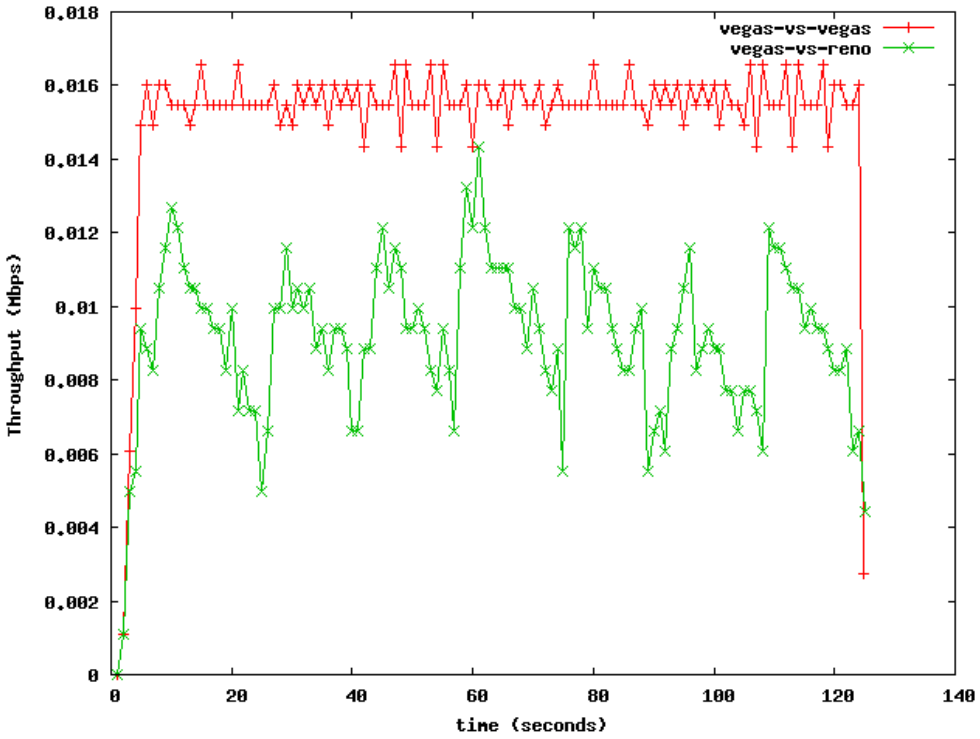

Figura 15: Comparativa de evolución del Throughput para Vegas. vs. Vegas y Vegas vs. Reno (con competencia de dos sesiones TCP, midiendo la primera).

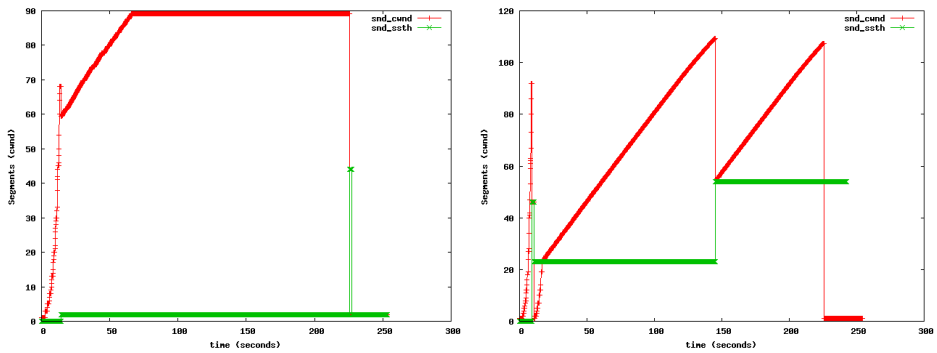

Figura 16: Evolución de CWND y SSTH para Vegas y Reno con un producto DELAY*BW más grande.

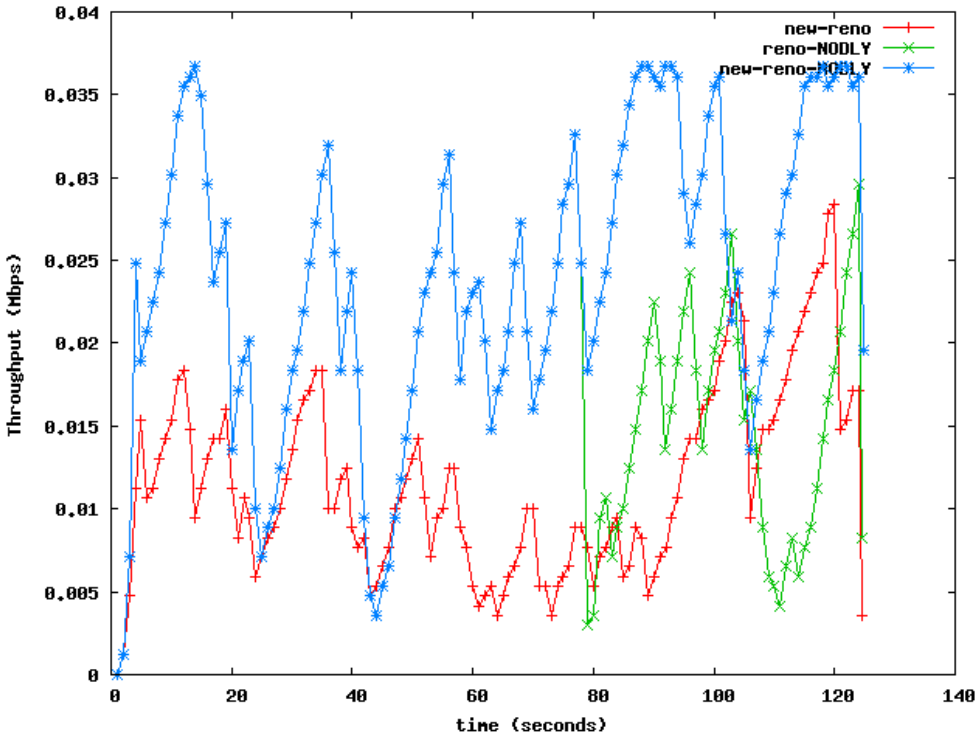

Figura 17: Comparativa de evolución del Throughput con enlace con pérdida y delay variable, para New-reno, Reno sin DELACK y New-Reno sin DELACK.

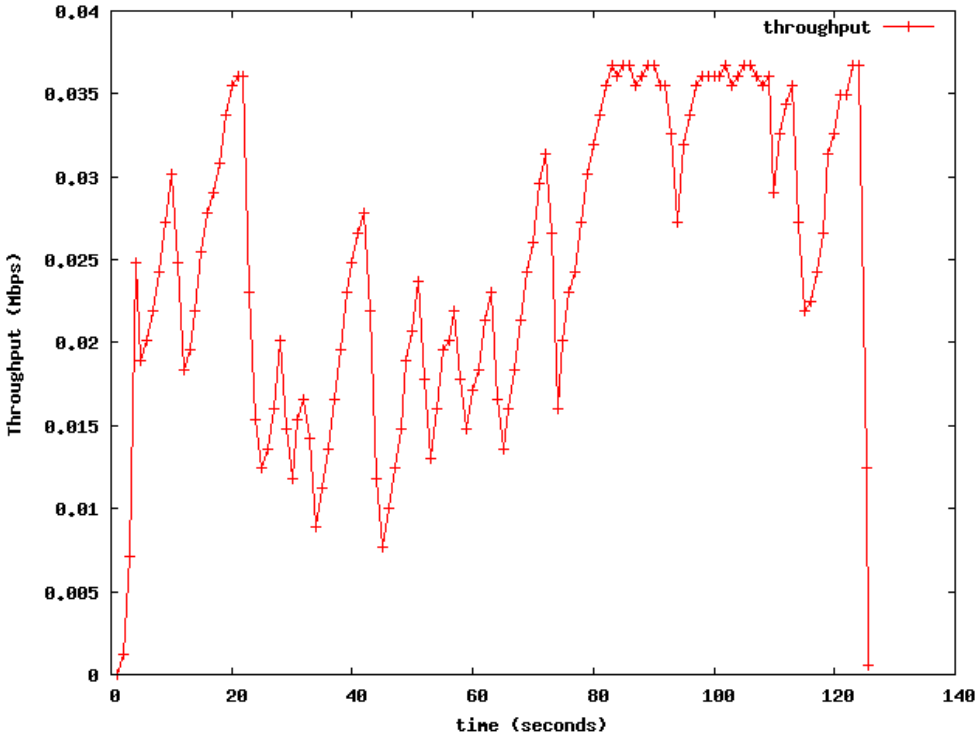

Figura 18: Evolución del Throughput con enlace con pérdida y delay variable, para SACK sin DELACK.

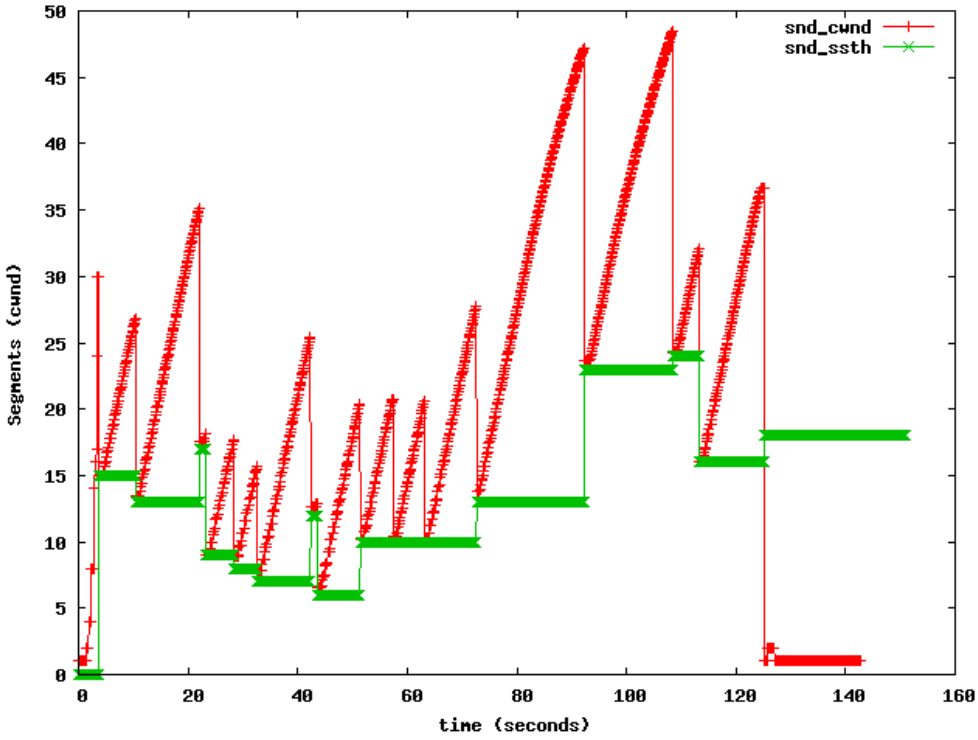

Figura 19: Evolución de CWND y SSTH para SACK sin DELACK.

Código de Algoritmo	Algoritmo y DelACK
1	reno DELAYED
2	tahoe DELAYED
3	newreno DELAYED
4	tahoe NODELAY
5	reno NODELAY
6	newreno NODELAY
7	sack NODELAY

Cuadro 1: Tabla de Algoritmos para la figura 21 y 7.

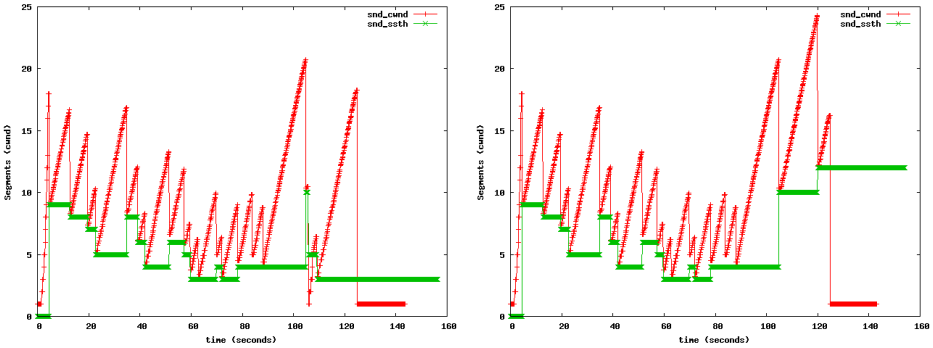

Figura 20: Evolución de CWND y SSTH para Reno y New-Reno sin SACK y con DELACK con un enlace con pérdida (Obtiene mucho pero rendimiento que la opción NODELAY y NODELAY+SACK).

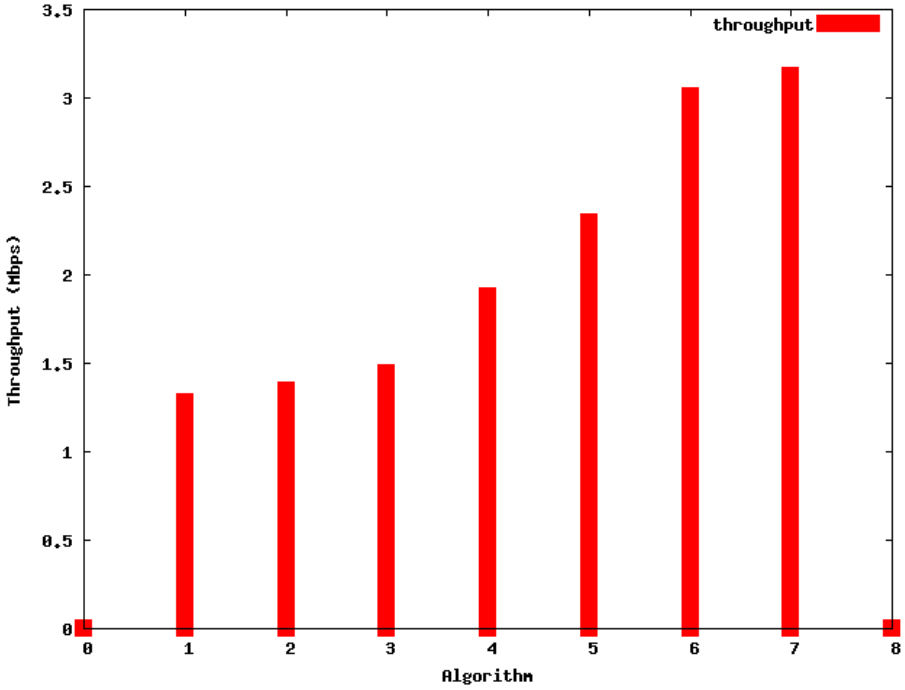

Figura 21: Comparativa de evolución del Throughput con enlace con pérdida y delay variable, para varias implementaciones. Ver tabla. 1.

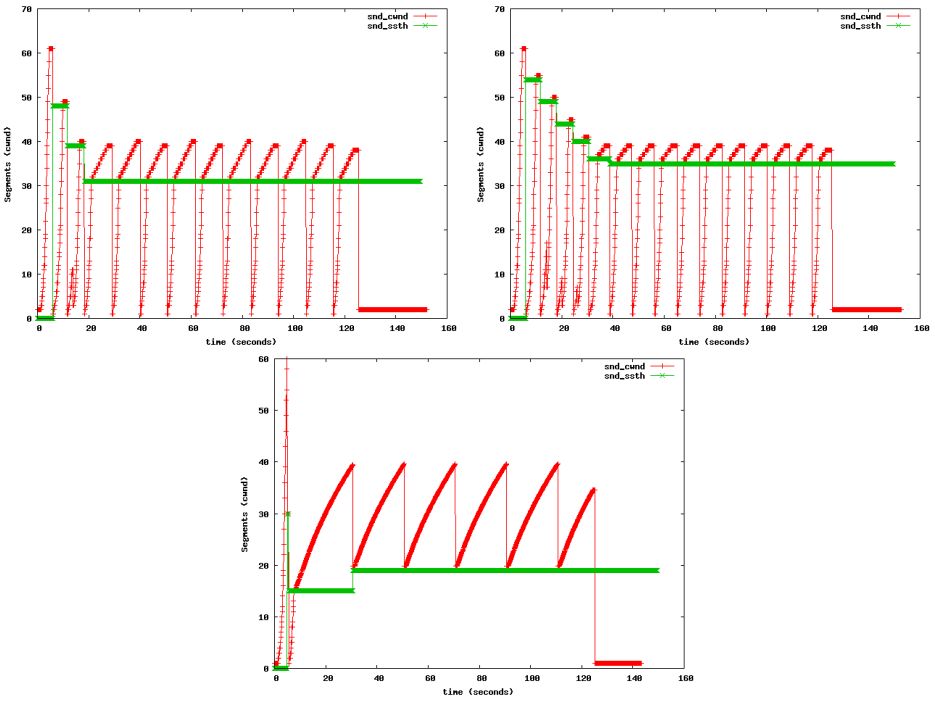

Figura 22: Evolución de CWND y SSTH para Cubic, Scalable y New Reno sin soporte de SACK (Cubic, Scalable e Hybla obtienen peor performance, Hybla la peor).

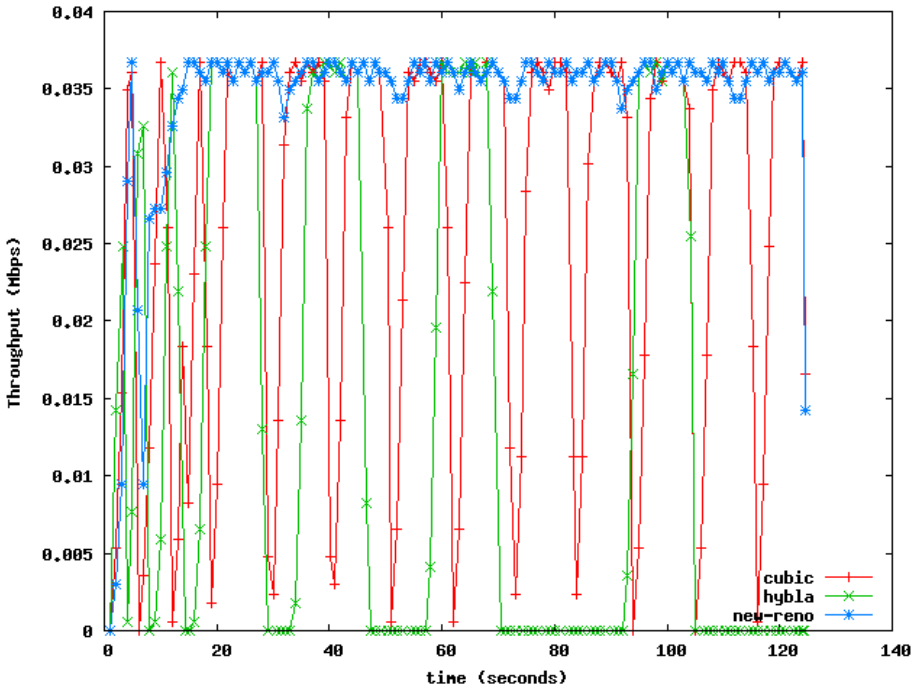

Figura 23: Comparativa de evolución del Throughput entre Cubic, Hybla y New-Reno sin soporte de SACK con algo de tráfico UDP.

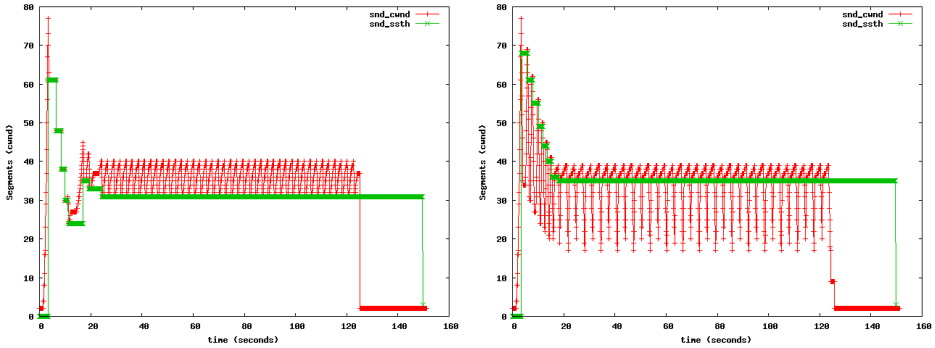

Figura 24: Evolución de CWND y SSTH para Cubic, Scalable e Hybla con soporte de SACK (Cubic, Scalable e Hybla obtienen mejor performance.

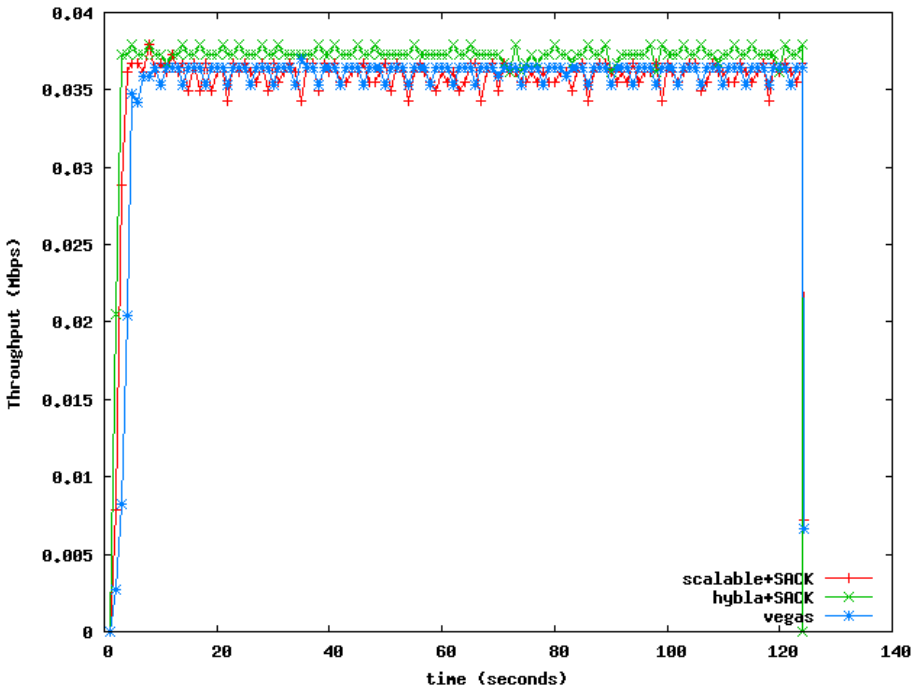

Figura 25: Comparativa de evolución del Throughput entre Scalable, Hybla y Vegas con soporte de SACK, con algo de tráfico UDP

2. Implementaciones TCP

TCP: Diseñado por Vinton Cerf y Robert Kahn. Las primeras implementaciones comenzaban enviando múltiples segmentos a la red hasta llenar la ventana de recepción publicada (window size advertised by the receiver): $RWND$. Los tamaños de segmentos típicos son de 512 o 536 [RFC-879], aunque hoy en día es más común ver tamaños de 1460, $1500 - (20 + 20)$ (MTU basado en Ethernet). Este comportamiento puede ser considerado correcto entre dos hosts en al misma LAN, pero cuando hay routers intermedios esto genera problemas, ya que puede ocurrir congestión en la red, bajando notablemente el rendimiento. Los cálculos mostrados a continuación son realizados en SMSS (Sender Max Segment Size) y no en bytes, salvo que se especifique lo contrario.

TCP-TAHOE: Algoritmo de CC (Congestion Control) diseñado por Van Jacobson e implementado por BSD Unix. Utiliza los ACK para ir "disparando" los nuevos segmentos de datos. Mantiene las variables $CWND$ y $SSTH$ para controlar la congestión. Comienza en etapa de **Slow Start (SS)** en la cual el crecimiento es exponencial y continúa en esta etapa hasta alcanzar $CWND == SSTH$ o detecta congestión por LOSS. Comienza con $CWND = 1$ (1 o 2 segmentos). En [RFC-2581] se indica que la ventana inicial, IW (Initial Window) puede tener valores mayores, de hasta 3 o 4 segmentos, pero sin exceder 4380 bytes. El valor inicial de $SSTH = 65535$ (un valor alto, representando infinito) [RFC-2001]. Si llega a $CWND == SSTH$ entra en etapa de **CA** en la cual el crecimiento es lineal. Es importante destacar que puede detener el envío antes ya que la capacidad a transmitir también esta regida por la ventana de recepción, $RWND$, siendo al ventana de transmisión: $Win = MIN(RWND, CWND)$. La detección de congestión la hace con segmentos perdidos (LOSS), ACK TMOUT. Al detectar congestión configura $SSTH = CWND/2$ y $CWND = 1$. Vuelve a empezar desde el inicio. Presenta el problema de que detecta la congestión de forma tardía y toma un RTT o más (de acuerdo a la "granularidad"). Cada vez que se detecta un TMOUT de T_0, el pipe se vacia y se vuelve a comenzar desde el principio Tiene bajo rendimiento en enlaces con un $DELAY * BW$ alto (Fat pipes). La implementación 4.3BSD Tahoe utilizaba **SS** solo en caso de que el otro extremo este en otra red [RFC-2001].

- **Slow Start:** se comienza enviando 1 segmento y esperando por ACK ($CWND = 1$), al recibir el ACK se incrementa al doble $CWND = 2$, cuando los ACK para los 2 nuevos segmentos llegan se lleva $CWND = 4$, proveyendo un crecimiento exponencial. De caulquier forma el emisor puede usar Delayed ACK (enviando un ACK, por cada 2 segmentos recibidos) por lo que el crecimiento no es tán exponencial.

- **Congestion Avoidance:** incrementa $CWND++$ por cada RTT, crecimiento lineal.

TCP-RENO: Algoritmo de CC (Congestion Control) que mantiene los mismos principios de **TCP-TAHOE**, pero agrega la detección de congestión en una etapa más temprana. Incluye **Fast-Retransmit (FRT)** y **Fast Recovery (FR)** diseñados por Van Jacobson. Detecta congestión de dos forma, una como TCP-TAHOE con TMOUT y otra con ACK duplicados (en realidad triplicados). Agrega 2 nuevas etapas etapas. Si detecta congestión con TMOUT vuelve a comenzar con SS y se configura $CWND = 1$. **Fast ReTransmit (FRT):** si se reciben 3 ACK duplicados (4 ACK para el mismo segmento de datos enviado) se toma como síntoma de segmento perdido y se re-transmite sin necesidad de esperar por TMOUT. El receptor no debe retardar los ACK, y en particular para los segmentos fuera de orden. Luego se entra en etapa de **Fast Recovery (FR):** $SSTH = MIN(CWND/2, 2)$ y $CWND = SSTH + 3$ (el 3 es por los 3 segmentos que han llegado fuera de orden y han dejado lugar en la red). Cada ACK duplicado se incrementa la $CWND++$, pues supone que un segmento fuera de orden ha llegado al extremo y hay un nuevo lugar en el pipe. Si $CWND > DataPipe$ envía un nuevo segmento en cada ACK. Si para el segmento retransmitido por el 3DUP ACK llega su ACK (debería llegar en RTT), al recibirlo vuelve $CWND = SSTH = MIN(CWND/2, 2)$ (valor al inicio de **FR**, se achica) y se termina

con esta etapa y comienza con **CA**. No vacia el pipe ante la detección de congestión. No se comporta bien con pérdidas de más de 1 segmento por ventana, múltiples pérdidas en un *single flight*. Podría llegar a reducirse la ventana dos o más veces en. La implementación 4.3BSD Reno cambia de Tahoe, que utilizaba **SS** solo en caso de que el otro extremo este en otra red, a utilizarlo siempre [StevI]. El algoritmo **FRT** aparecio por primera vez en 4.3BSD Tahoe, pero incorrectamente luego de salir de esta etapa comenzaba con **SS**. El **FR** aparecio por primera vez en 4.3BSD Reno [StevI]. En [RFC-2581] se indica que en lugar de configurar $SSTH = MIN(CWND/2, 2)$ se debe hacer $SSTH = MIN(FlightSize/2, 2)$, siendo $FlightSize$ la cantidad de datos (segmentos) en la red que aún no han sido confirmados.

TCP-NEW-RENO: Es la implementación de **TCP-RENO** actual. Diseñado por S.Floyd, y T.Henderson. Similar a **TCP-RENO** pero permite detectar múltiples pérdidas sin reducir la ventana dos veces o más. Las fases de **SS**, **CA** y **FRT** son iguales. Cambia en la fase **FR**, solo sale de esta etapa cuando todos los segmentos pendientes de ACK de la ventana al momento de detectarse la congestión han sido confirmados. Con ACK nuevos pero habiendo aún segmentos sin confirmar supone que el siguiente fue perdido, lo retransmite y coloca el contador de $DUPACKS = 0$. Recién al estar todo confirmados pone $CWND = SSTH$ y continua con **CA** como **TCP-TAHOE**. Tiene el problema que para detectar pérdidas requiere un RTT.

TCP-SACK: Es la implementación de TCP que agrega a **TCP-RENO** o **TCP-NEW-RENO** la capacidad de manejar de mejor manera múltiples pérdias en un mismo RTT. Requiere que los ACK no sean acumulativos como go-back-N, sino que sean selectivos en bloques. Cuando se entra en **FR** se estima cuantos datos han sido enviados y no se han confirmado $FlightSize$ o $Pipe$ y se configura $CWND = CWND/2$. Cada ACK se lleva $FlightSize - -$ y cada retransmit $FlightSize + +$. Cuando $FlightSize < CWND$ envia todos los segmentos que no tiene confirmados, si no existen envia nuevos entrando en CA. Definido en [RFC-2018]. Se utiliza como una extensión a **TCP-NEWRENO**.

TCP-VEGAS: Desarrollado en la Universidad de Arizona, por Lawrence Brakmo y Larry L. Peterson. Cambia la idea de **TCP-RENO**, en lugar de trabajar de forma re-activa, ante la congestión, intenta hacerlo de forma pro-activa evitándola. Detecta congestión midiendo el DELAY en cada RTT en lugar del LOSS. En la etapa de **SS** solo incrementa de forma exponencial si entre cada RTT el througput alcanzado esta por encima de un umbral (compara el real con el estimado), si esto no sucede, pasa a **CA**. En etapa de **CA** va incrementando y decrementando la $CWND$ de manera de mantener el rendimiento esperado. Al recibir un ACK duplicado chequea que $(CurrentTime - SegmentTxTime) > RTT$ (RTT estimado) reenvía sin espera 3DUP ACK.

Algunas Otras Implementaciones de TCP: Existen varias optimizaciones realizadas a TCP, la mayoría pone énfasis en sacar provecho de LFP (Long Fat Pipes) o LFN (Long Fat Networks): redes con un gran producto BW*DELAY. En general solo se modifica el lado del emisor.

TCP-Hybla: es una implementación que tiene como objetivo mejorar la performance en redes con un RTT muy grande (e.g. enlaces satelitales). Modifica el algoritmo de control de congestión teniedo en cuenta los RTT muy grandes. Para que funcione de forma correcta requiere obligatoriamente la utilización de SACK y hace uso de TIMESTAMPS. Si bien es una modificación del emisor solo, si el receptor no soporta SACK su comportamiento no es bueno. Fue desarrollado en el DEIS, Universidad de Bologna, Italia. Es esponsoreado por las redes satelitales europeas.

HighSpeed TCP (HSTCP): implementación estandarizada en [RFC-3649]. Esta pensada para optimizar el rendimiento de TCP en enlaces con producto BW*DELAY grande, LFP (Long Fat Pipes). Si la ventana de congestión es chica se comporta igual que New-Reno, si la CWND

esta por encima de un umbral los parámetros para calcular como decrece o crece la CWND se calculan en función del tamaño de la ventana, dejando que crezca más rápido en CA.

TCP-BIC: Binary Increase Congestion control, implementacion para redes con producto BW*DELAY grandes. Persigue el mismo objetivo que los anteriores La CWND se calcula de forma cuadrática con respecto al tiempo desde que se detecto congestión. En varias versiones del Kernel GNU/Linux 2.6 era el default.

TCP-Cubic: Una versión de BIC moderado. Simplifica el cálculo de CWND con respecto a BIC y es más amigable con respecto a otras sesiones TCP. Tiene también un buen despempeño en enlaces con pérdidas. En las versiones del Kernel GNU/Linux 2.6 actuales es la default.

Scalable TCP: Implementación de TCP pensada para trabajar sobre LFP y con tiempos de recuperación a pérdidas mucho mejores que las implementaciones tradicionales.

TCP-Westwood y Westwood+ (TCPW,TCPW+): modificación de TCP, del lado del emisor que permite trabajar con LFP y que incluso tienen pérdidas, LEAKY PIPES. Una de sus utilizaciones son los enlaces Wireless. Calcula el rate que debe enviar a partir de los ACK que va recibiendo. Es un protocolo amigable para las demas sesiones TCP. TCP-Westwood+ (plus) soluciona un problema de TCP-Westwood en la estimación al tener tráfico de regreso: "ACK-Comprssed" (Varios ACK que salen con una una separación desde el origen, perod debido a encolado llegan luego todos "juntos" al destino.

TCP-Compound: Implementación de Microsoft que mantiene dos CWND.

TCP-LP (Low Priority): implementación de TCP que intenta utilizar solo el excedente de la capacidad del enlace, la capacidad no utilizada.

TCP-Veno: Implementación que combina características de Reno y Vegas.

TCP-Illinois: variante de TCP desarrollada en la Universidad de Illinois, Urbana-Champaign. Pensado para mejorar rendimiento sobre LFP. A medida que la CWND va cambiando los parámetros para hacer crecer este valor.

H-TCP (Hamilton TCP): Desarrollado en el Hamilton Institute, en Irlanda. Como la gran mayoría de las nuevas implementaciones intenta mejorar el rendimiento sobre LFP, trabaja de forma más moderada que BIC o HSTCP.

Referencias

[StevI] TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, 1994. W. Richard Stevens.

[NS-2] The NS Manual (formerly ns Notes and Documentation). The VINT Project, A collaboratoin between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC. Kevin Fall, Editor, Kannan Varadhan, Editor. http://www.isi.edu/nsnam/ns/doc/index.html.

[NS-2-TCP] A mini-tutorial for TCP-Linux in NS-2. David X. Wei. NetLab Caltech. http://netlab.caltech.edu/projects/ns2tcplinux/ns2linux/ tutorial/index.html#parameters.

[RFC-793] TRANSMISSION CONTROL PROTOCOL. http://tools.ietf.org/html/rfc793. 1981.

[RFC-879] The TCP Maximum Segment Size and Related Topics. J. Postel (ISI) 1983. http://tools.ietf.org/html/rfc879.

[RFC-2001] TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. W. Stevens, 1997. http://tools.ietf.org/html/rfc2001.

[RFC-2018] TCP Selective Acknowledgment Options. M. Mathis, J. Mahdavi S. Floyd, A. Romanow. 1996. http://tools.ietf.org/html/rfc2018.

[RFC-2581] TCP Congestion Control. M. Allman, V. Paxson, W. Stevens. 1999. http://tools.ietf.org/html/rfc2581.

[RFC-2582] The NewReno Modification to TCP's Fast Recovery Algorithm. S. Floyd (ACIRI), T. Henderson (U.C. Berkeley). 1999. http://www.ietf.org/rfc/rfc2582.txt.

[RFC-3390] Increasing TCP's Initial Window. M. Allman, S. Floyd. C. Partridge. 2002. http://tools.ietf.org/html/rfc3390.

[RFC-3649] HighSpeed TCP for Large Congestion Windows. S. Floyd (ICSI). 2003. http://www.ietf.org/rfc/rfc3649.txt.

[ACOMP] A Comparative Analysis of TCP Tahoe, Reno, New-Reno, SACK and Vegas. http://inst.eecs.berkeley.edu/~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf. Contribución de estudiantes de U.C. Berkeley.

[Brak94] TCP Vegas: New techniques for congestion detection and avoidance. L. Brakmo, S. O'Malley, and L. Peterson. n Proceedings of the SIGCOMM '94 Symposium (Aug. 1994) pages 24-35.

[SCA] Scalable TCP: Improving Performance in Highspeed Wide Area Networks. Tom Kelly CERN - IT Division. 1211 Geneva 23. Switzerland. http://www.deneholme.net/tom/papers/scalable_improve_hswan.pdf. On Engineering a Stable and Scalable TCP Variant. Technical Report CUED/F-INFENG/TR.435, Laboratory for Communication Engineering, Cambridge University, June 2002.

[HYBLA] A New Transport Protocol Proposal for Internet via Satellite: the TCP Hybla. C. Caini and R. Firrincieli, in Proc. ESA ASMS 2003, Frascati, Italy, Jul. 2003, .vol. SP-54.

[CUBIC] CUBIC: A New TCP-Friendly High-Speed TCP Variant. Injong Rhee, and Lisong Xu. http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/cubic-paper.pdf. In Proc. Workshop on Protocols for Fast Long Distance Networks, 2005.

[Siever] Linux in a Nutshell, Fourth Edition June, 2003. O'Reilly. Ellen Siever, Stephen Figgins, Aaron Weber.

Índice

Índice de figuras