

Universidad Nacional de Río Cuarto

Facultad de Ingeniería



Trabajo final de tráfico:

Análisis de algoritmos de control de congestión TCP: BBRv1, BBRv2, Reno y Cubic en Mininet

Tráfico (Cod. 015)

Integrantes:

- Bernardi Martín: 39024893
- Remedi Augusto: 40202426
- Rittano Ignacio: 39967807

Introducción	3
Desarrollo	4
Configuración de herramientas	4
Máquina virtual con BBRv2	4
Mininet	6
Iperf3	10
Captcp y tcpdump	11
qlen_plot.py y software de pruebas	15
FRRouting y Miniedit	16
Adaptación de Miniedit	16
Escenario de ejemplo	19
Algoritmos de control de congestión	22
TCP Reno	22
TCP CUBIC	23
TCP BBRv1	24
TCP BBRv2	27
Escenario	30
Resultados	31
Escenario 1	31
Escenario 2	35
Escenario 3	37
Escenario 4	39
Escenario 5	41
Conclusión	44
Referencias	45

Introducción

En el siguiente trabajo se procederá a realizar diversas pruebas de algoritmos de control de congestión TCP, centrándonos en el más reciente hecho por Google y publicado como código libre: BBRv2. El objetivo es analizar y comparar su comportamiento frente a su versión anterior y ante alternativas como Reno y Cubic, explicando el funcionamiento de cada uno y corroborando mediante mediciones de throughput, ventana de congestión y latencia, entre otros.

Ya hay disponibles trabajos comparando los algoritmos analizados, pero lo que los diferencia es el ambiente de trabajo. La mayoría utiliza el servicio Google Compute Engine (GCE) o servidores reales que se poseen en una empresa. En nuestro caso buscamos brindar información sobre el uso y configuración de un ambiente de trabajo basado en Mininet para realizar pruebas de todo tipo mediante la virtualización.

Por otro lado, se explicará el uso y configuración de Mininet y Miniedit para la simulación de Routers implementados con FRRouting. Esto es de gran utilidad ya que Mininet no contempla la posibilidad de ejecutar protocolos de ruteo como OSPF y BGP, pero realizando configuraciones menores es posible utilizar esta herramienta para simular escenarios.

Todo este entorno de trabajo lleva un trabajo de configuración considerable, por lo tanto se proveerá el código fuente desarrollado (disponible en <https://github.com/martinber/bbr2-mininet>) y una máquina virtual lista para ser utilizada.

Desarrollo

Configuración de herramientas

A continuación se explica el procedimiento de instalación y configuración de herramientas relacionadas a la simulación y análisis de tráfico en Mininet

Máquina virtual con BBRv2

La máquina virtual (VM) está basada en una distribución Lubuntu, debido a sus bajos requerimientos de hardware. Para instalarla solo se debe abrir el archivo **.ova** y seguir los pasos de instalación. Es posible crear interfaces, aumentar la cantidad de RAM, número de procesadores, etc. Es recomendado usar los valores por defecto que hemos utilizado.

Es necesario utilizar VirtualBox 6.1 o superior ya que brinda soporte a la versión de kernel 5.4 con BBRv2 que utilizamos. También se recomienda asignar más de un núcleo, de lo contrario observamos disminuciones importantes en el throughput.

De forma alternativa a la utilización de la máquina virtual OVA que brindamos, es posible realizar la instalación manualmente.

Instalación

Se puede partir de una instalación estándar de Lubuntu o de una VM proveniente de sitios como www.osboxes.org. Además de necesitar VirtualBox 6.1 o mayor, es necesario instalar dentro de la VM una versión de DKMS compatible.

Por otro lado, se encontró que la VM tenía muy poco espacio en la partición **/boot** para almacenar el Kernel a compilar. En nuestro caso con gparted se agrandó la partición **/boot** a expensas de achicar el swap que estaba adyacente.

Se recomienda instalar el Kernel 5.4.0 personalizado por Google con soporte para BBRv2 proveniente de <https://github.com/google/bbr>. Los pasos de compilación se corresponden a los de cualquier Kernel Linux:

1. Instalar dependencias:

```
sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils  
libssl-dev bc flex libelf-dev bison
```

2. Clonar desde repositorio github:

```
git clone -o google-bbr -b v2alpha --depth 1 https://github.com/google/bbr.git
```

3. Copiar **/boot/config-5.0.0-23-generic** o similar a **bbr/.config**
4. Ejecutar el comando **make**
5. Responder a las preguntas, las cuales se pueden pasar dejando el valor por defecto, a excepción de cuando pregunta si hay desea activar BBRv2, a lo que se debe poner que sí, escribiendo **Y**:

```
BBR2 TCP (TCP_CONG_BBR2) [N/m/y/?] (NEW) y
```

6. Terminar de instalar con

```
sudo make modules_install  
sudo make install
```

Luego de terminar la instalación y compilación del nuevo Kernel, se debe reiniciar la VM. En este momento encontramos que la cantidad de memoria RAM que previamente asignamos a la máquina era insuficiente provocando al inicio el error:

```
Kernel panic - not syncing: System is deadlocked on memory
```

La solución fue aumentar la cantidad de memoria disponible para la VM.

Una vez que ya se ha encendido la máquina con el nuevo Kernel, va a aparecer BBR2 dentro de los algoritmos disponibles, que puede ser comprobado a partir de la herramienta sysctl de siguiente forma:

```
sysctl net.ipv4.tcp_available_congestion_control
```

Es posible observar el algoritmo actualmente activo con:

```
sysctl net.ipv4.tcp_congestion_control
```

Para cambiar el algoritmo se utiliza el comando:

```
sysctl -w net.ipv4.tcp_congestion_control=bbr2
```

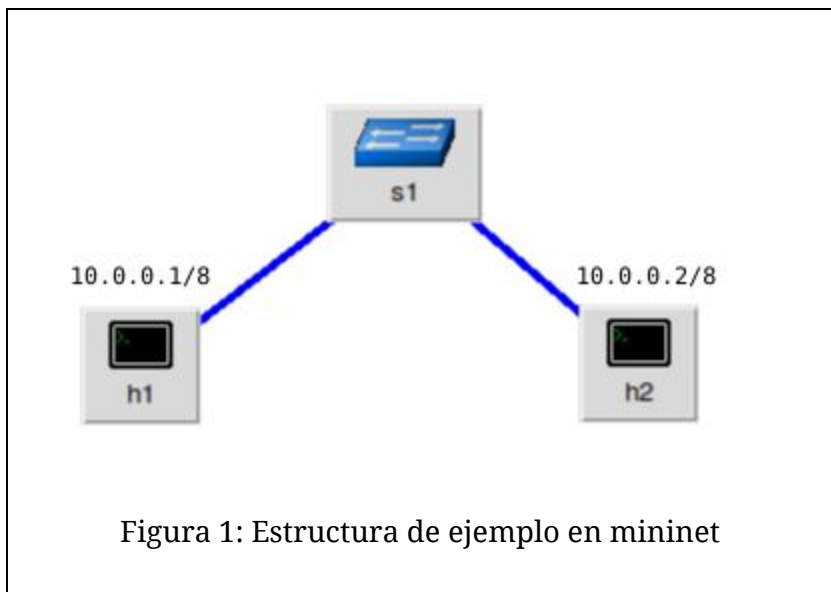
Mininet

Es una herramienta que permite generar una red virtual dentro de una computadora con muy pocos requerimientos. Mininet nos da la capacidad de agregar hosts, switches y routers que comparten el kernel de la máquina que estamos utilizando. Se programa la red utilizando Python 2 y nos brinda la posibilidad de acceder a una CLI (Command Line Interface) que nos permite correr comandos en los distintos componentes que conforman la red.

Los pasos para instalar este programa son:

```
git clone git://github.com/mininet/mininet
./mininet/util/install.sh
```

Por defecto, al ejecutar mininet se crea una topología como la que se muestra en la Figura 1:



Pero este esquema de red no es visible gráficamente, ya que Mininet trabaja solamente mediante una consola de comandos que se abre al ejecutar:

```
sudo mn
```

Es posible que ejecuciones anteriores de Mininet que no se cerraron correctamente produzcan el error de la Figura 2:

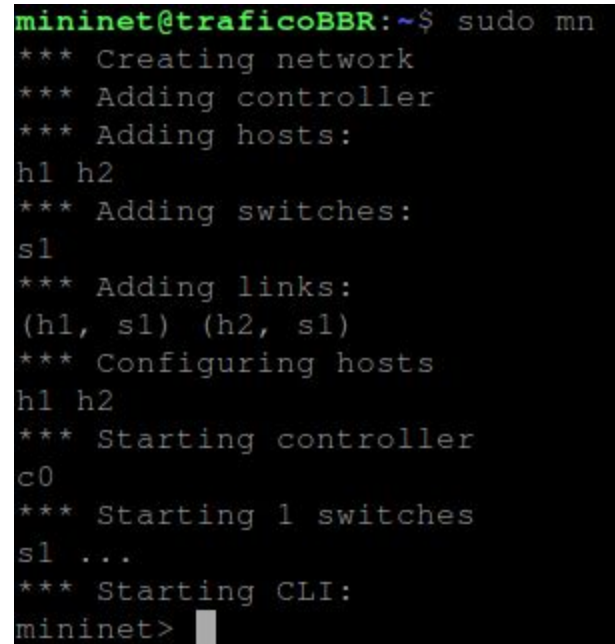
```
mininet@traficoBBR:~$ sudo mn
*** Creating network
*** Adding controller
-----
Caught exception. Cleaning up...

Exception: Please shut down the controller which is running on port 6653:
Active Internet connections (servers and established)
tcp        0      0 0.0.0.0:6653          0.0.0.0:*             LISTEN
758/ovs-testcontrol
tcp        0      0 127.0.0.1:41510       127.0.0.1:6653        TIME_WAIT
-
tcp        0      0 127.0.0.1:41508       127.0.0.1:6653        TIME_WAIT
-
-----
```

Figura 2: Ejemplo de error provocado por un incorrecto cierre de Mininet

Para solucionar esto, se debe correr el siguiente comando:

```
sudo kill -9 `sudo lsof -t -i:6653`
```



```
mininet@traficoBBR:~$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figura 3: Ejecución correcta de Mininet

Luego, al intentar ejecutar de nuevo mininet tenemos acceso a la CLI (ver Figura 3). Para correr comandos como se haría en cualquier linux, se debe especificar el host y el comando. Por ejemplo, si queremos enviar un ping del host 1 hacia el host 2, se debe ejecutar:

```
h1 ping 10.0.0.2
```

De esta forma se puede trabajar con los dos hosts y realizar cualquier tipo de pruebas. Los hosts de Mininet sólo virtualizan las interfaces de red, es decir, que todos los procesos ejecutados corren en la máquina anfitrión (la máquina real). Por ende, se puede ejecutar cualquier programa o comando que se encuentre instalado por fuera de Mininet.

Para este trabajo, en lugar de usar la consola de Mininet, se programaron los escenarios en Python para automatizar las pruebas en gran cantidad de escenarios y algoritmos de control de congestión.

Siguiendo el caso anterior de mininet, para generar la misma topología en Python 2 se deben seguir los siguientes pasos. Primero comenzamos importando todas las librerías necesarias.

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
```

Luego, definimos nuestro código, de la forma que se muestra a continuación:

```
def myNetwork():

    net = Mininet( topo=None,
                  build=False,
                  ipBase='10.0.0.0/8')

    info( '*** Iniciando\n')
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch, failMode='standalone')

    h1 = net.addHost('h1', cls=MiHost, ip='10.0.0.1', defaultRoute=None)
    h2 = net.addHost('h2', cls=MiHost, ip='10.0.0.2', defaultRoute=None)

    net.addLink(h1, s1)
    net.addLink(h2, s1)

    net.build()

    for controller in net.controllers:
        controller.start()

    net.get('s1').start([])

    info( '*** Realizar pruebas\n')

    h1.cmd('ping 10.0.0.2')

if name == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

Al ejecutar este script, se creará la topología y el host 1 realizará un **ping** al host 2.

Iperf3

Iperf3 es una herramienta que permite la transmisión de datos entre un servidor y un cliente. Se puede instalar simplemente utilizando **apt**:

```
sudo apt install iperf3
```

Al usar iperf3, primero se define un servidor, ejecutando el comando:

```
iperf3 -s
```

Este comando abre el servidor en el puerto 5201 por defecto, pero puede ser modificado con la opción **-p port**. Por otro lado en el cliente, se debe ejecutar:

```
iperf3 -c "ip" -p "port"
```

Para el cliente hay otras opciones útiles como **-t "tiempo"** donde con un entero se le indica la cantidad de segundos que dura la transmisión, o **-n "bytes"** para indicar una velocidad de transmisión (por ejemplo 100M) .

Gracias a esta herramienta, en conjunto con **captcp** y **tcpdump**, es posible graficar parámetros de mediciones como son el throughput, retransmisiones, ventana de congestión, rtt, entre otros.

Captcp y tcpdump

Captcp es un software código libre cuya finalidad es el análisis de tráfico TCP de archivos **.pcap** que pueden ser obtenidos mediante el uso de **tcpdump**. **tcpdump** ya viene instalado bajo la distribución de Ubuntu. Para instalar **captcp** comenzamos por obtener las dependencias:

```
sudo apt install aptitude
aptitude install python-dpkt python-numpy make gnuplot texlive-latex-extra \
    texlive-font-utils mupdf
```

Si se desea instalar la versión estándar de **captcp**, se la debe obtener del repositorio oficial:

```
curl -LOk https://github.com/hgn/captcp/archive/master.zip
unzip master.zip
cd captcp-master
su -c 'make install'
```

En nuestro caso realizamos dos modificaciones al programa:

- En la versión original se imprimen demasiados mensajes que impiden observar la información de nuestro interés.
- En la versión original hay situaciones en donde no es posible ejecutar el proceso de **captcp socketstatistic** en el background y luego cerrarlo correctamente.

La segunda modificación es especialmente importante, se modificó la función **process_final()** de **captcp.py** de la siguiente manera:

```

def process_final(self):
    # Se agregaron estas dos lineas
    import signal
    signal.signal(signal.SIGINT, signal.default_int_handler)
    # Fin de modificaciones

    self.logger.warning("Sampling rate: %f Hz" % (self.opts.sampling_rate))
    self.logger.warning("Start capturing socket data, interrupt process with CTRL-C")

    try:
        while True:
            data = self.execute_ss()
            self.process_data(data)
            time.sleep(self.sleep_time)

    except KeyboardInterrupt:
        self.logger.warning("\r# SIGINT received, please wait to generate data (this
may take a while)")

        self.write_db()

        self.logger.warning("Data generated in %s/" % (self.opts.outputdir))

```

Para instalar nuestra versión de captcp, se pueden seguir estos comandos:

```

curl -LOk https://github.com/martinber/bbr2-mininet/archive/master.zip
unzip master.zip
cd bbr2-mininet/captcp-mininet
su -c 'make install'

```

A continuación se mostrará un ejemplo de uso de captcp utilizando una topología básica en mininet de dos hosts y un switch, como se muestra en la figura XXXX. En el host 2 abrimos un servidor iperf3, y en el host 1 ponemos a tcpdump a sniffear. Tener en cuenta escuchar en la interfaz correspondiente y usar el argumento **-w** para que escriba la salida en un archivo como se muestra en la Figura 4.

```

2: h1-eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP group default qlen 1000
    link/ether f2:76:68:bd:10:4f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.1/8 brd 10.255.255.255 scope global h1-eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::f076:68ff:febd:104f/64 scope link
        valid_lft forever preferred_lft forever
root@traficoBBR:~# tcpdump -i h1-eth0 -w trace.pcap
tcpdump: listening on h1-eth0, link-type EN10MB (Ethernet), capture size 262144
bytes
^C50575 packets captured
823274 packets received by filter
772699 packets dropped by kernel

```

Figura 4: Ejecución de tcpdump

Luego hacemos una prueba con iperf3 (ver Figura 5) corriendo en el host 1 el comando ya visto anteriormente: `iperf3 -c "ip" -p "port"`. En este caso el servidor se abrió en el puerto por defecto 5201.

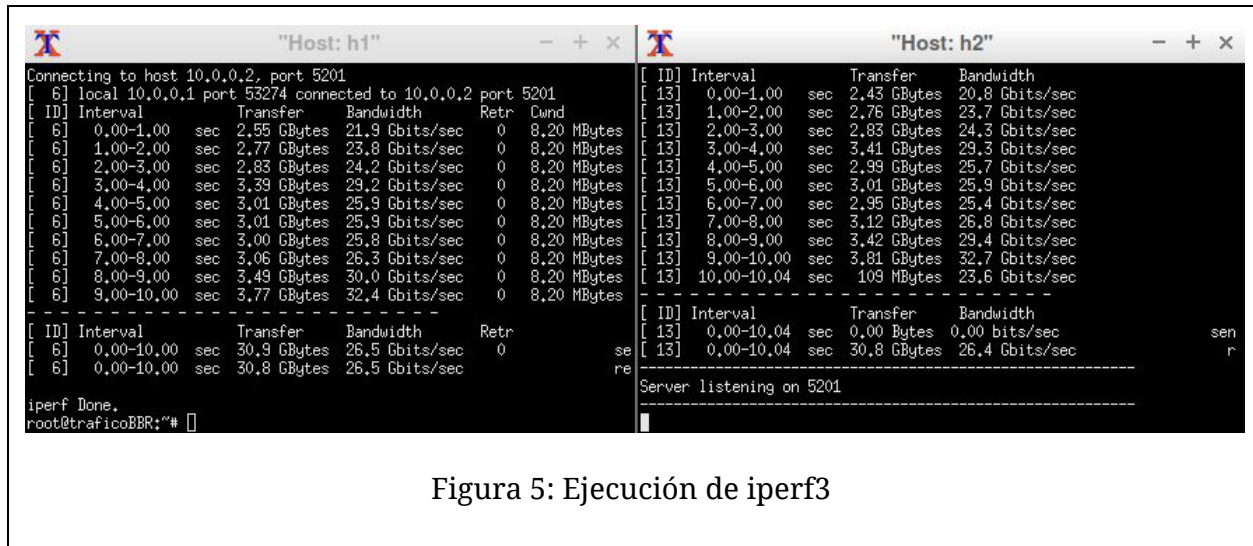


Figura 5: Ejecución de iperf3

Corremos el comando de captcp para analizar el archivo `.pcap` e identificar el flujo de tráfico correcto (ver Figura 6). Recordar que en iperf3 por defecto el host1 se conecta al servidor iperf en el host2 y transfiere tráfico.

```
captcp statistic filename.pcap
```

```

root@traficoBBR:~# captcp statistic trace.pcap
2 10.0.0.1:53274<->10.0.0.2:5201

Packets processed: 50544 (99.9%)
Duration: 10.04 seconds

Flow 2,1 10.0.0.1:53274 -> 10.0.0.2:5201
Flow 2,2 10.0.0.2:5201 -> 10.0.0.1:53274
Packets: 29272 packets
Duration: 10.04 seconds
Data link layer: 1395734869 bytes
Link layer throughput: 1112572731.47 bit/s
Data application layer: 1393802909 bytes
Application layer throughput: 1111032721.21 bit/s
Retransmissions: 28487 packets
Retransmissions per packet: 97.32 percent
ACK flag set but no payload: 1 packets

```

Figura 6: Ejemplo de salida de captcp statistic

Luego generamos la gráfica de throughput con el siguiente comando:

```

captcp throughput -s 0.1 -i -f flowID -o out filename.pcap

```

Donde **flowID** es en este caso **2.1**. Al ejecutar este comando se crea la carpeta **out**, en la cual hay que ejecutar **make** para generar el gráfico, tal como se muestra en la Figura 7

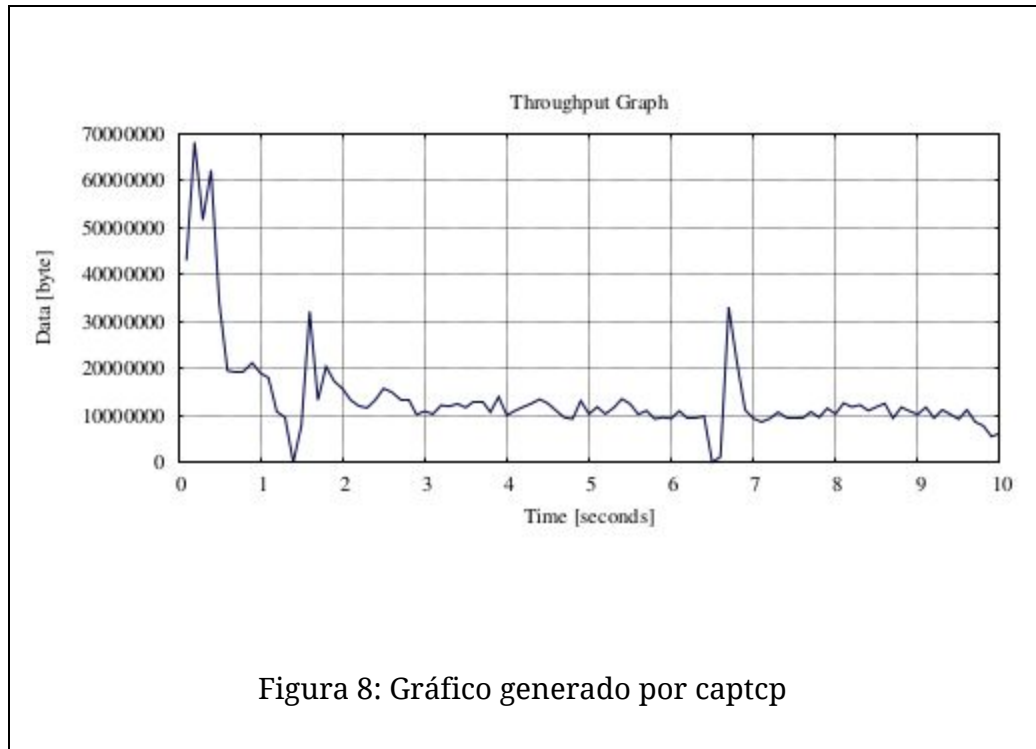
```

root@traficoBBR:~# mkdir out
root@traficoBBR:~# captcp throughput -s 0.1 -i -f 2,1 -o out trace.pcap
# captcp 2010-2013 Hagen Paul Pfeifer and others (c)
# http://research.protocollabs.com/captcp/
# WARNING: graph is scaled to byte per 0.1 seconds
# Use --per-second (-p) option if you want per-second average
# total data (goodput): 1393802909 byte (11.15 Gbit)
# throughput (goodput): 138879090.15 byte/s (1.11 Gbit/s)
# now execute (cd out; make preview)
root@traficoBBR:~# cd out
root@traficoBBR:~/out# make png

```

Figura 7: Generación de gráficos de captcp

Se obtiene de correr el comando **make** un archivo llamado **throughput.pdf** que contiene la gráfica del flujo. En la Figura 8 hay una gráfica de ejemplo.



Además de throughput, captcp contiene varios módulos que permiten observar distinta información.

- throughput: Genera una gráfica del throughput en función del tiempo a partir de una captura **.pcap**.
- statistic: Muestra estadísticas generales sobre las diferentes conexiones TCP capturadas en un archivo **.pcap**.
- inflight: También toma una captura, gráfica entre otras cosas la evolución de los números de secuencia en el tiempo y los momentos en donde ocurren pérdidas de paquetes.
- socketstatistic: A diferencia de los módulos anteriores no utiliza capturas **.pcap** sino que muestrea en tiempo real los tamaños de las ventanas de congestión y los tiempos de ida y vuelta.

qlen_plot.py y software de pruebas

Para realizar las pruebas creamos dos programas adicionales. Uno es **qlen_plot.py**, este programa está basado en el código fuente de Mininet y permite capturar el tamaño de cola presente en el buffer de tipo Token Bucket Filter. Observando el tamaño de cola del host corriendo netem, podemos observar el grado de saturación de la red.

Para hacer las pruebas, programamos en Python un script que permite realizar múltiples pruebas de forma automática, y además, graficar todas las variables de interés mediante matplotlib.

Es necesario instalar la librería matplotlib, y además instalar **qlen_plot.py** que se encuentra disponible en el repositorio:

```
sudo apt install python-matplotlib  
sudo install -m 755 ~/Desktop/bbr2-mininet/qlen_plot/qlen_plot.py /usr/local/bin/
```

Para ejecutar el programa se debe indicar el número de escenario a simular. Por ejemplo, para simular el segundo escenario:

```
sudo python ~/Desktop/bbr2-mininet/tcp-mininet/tcp_mininet.py 2
```

Los resultados se almacenan en la carpeta **/var/tmp/mininet**. Allí dentro, hay una carpeta **res/** con el resumen de los resultados, de lo contrario se pueden explorar los archivos generados por cada host en cada prueba.

FRRouting y Miniedit

Más allá de las pruebas de rendimiento comparativas entre los distintos algoritmos de control de congestión, en esta sección se explica el uso de Mininet y Miniedit para la simulación de Routers que ejecutan FRRouting.

Miniedit es un programa que permite crear topologías listas para ser ejecutadas con Mininet, pero estos programas originalmente sólo permiten crear Hosts y Switches. En este trabajo adaptamos el código fuente de Miniedit para su uso con FRRouting.

FRRouting es un conjunto de programas que permiten efectivamente transformar una computadora Linux en un Router, con soporte para BGP, OSPF, ISIS, MPLS, etc.

Encontramos que para utilizar FRRouting en Miniedit es necesario utilizar las librerías adicionales del proyecto FRRouting (con ciertas modificaciones) y realizar cambios menores a Miniedit. El resultado se puede observar y descargar desde [esta dirección](#). A continuación se explica de manera general en qué consiste el trabajo realizado.

Adaptación de Miniedit

Obtener Miniedit

El primer paso consiste en obtener el código fuente de Miniedit que consiste en un sólo script de Python ubicado en [/mininet/examples/miniedit.py](#)

Obtener librerías de FRRouting para Mininet

El proyecto FRRouting utiliza Mininet para realizar pruebas sobre el software, para ello existen librerías desarrolladas con el fin de integrar ambos programas. Recomendamos descargar todas las librerías presentes en [/frr/tests/topotests/lib/](#) y almacenar esa carpeta junto con el script de Miniedit

Adaptar las librerías de FRRouting

Ya que las librerías están desarrolladas para integrarse a su suite de tests y no para ser usadas con Mininedit, es necesario realizar unos cambios:

En **lib/topotest.py** se debe modificar la línea 775 para definir un nombre de prueba en lugar de obtenerlo desde el entorno. Se puede utilizar cualquier string

```
cur_test = os.environ["PYTEST_CURRENT_TEST"]  
cur_test = "miniedit_topology"
```

En el mismo archivo, alrededor de la línea 790, observar que existen gran cantidad de servicios que componen FRRouting. Activar los que sean necesarios reemplazando los ceros por unos. Por ejemplo:

```
self.daemons = {  
    "zebra": 1,  
    "ripd": 0,  
    "ripngd": 0,  
    "ospfd": 0,  
    "ospf6d": 0,  
    "isisd": 0,  
    "bgpd": 1,  
    "pimd": 0,  
    "ldpd": 0,  
    "eigrpd": 0,  
    "nhrpd": 0,  
    "staticd": 1,  
    "bfdd": 0,  
    "sharpd": 0,  
}
```

En la línea siguiente, agregar al diccionario todos los servicios activados siguiendo el patrón:

```
self.daemons_options = {"zebra": ""}  
self.daemons_options = {"zebra": "", "bgpd": "", "staticd": ""}
```

Ahora en el script de Miniedit, agregar el siguiente import al comienzo del archivo (junto con el resto ya presente:

```
from lib.topotest import Router as LegacyRouter
```

Siguiendo en el mismo archivo, eliminar la clase **LegacyRouter**. Estamos reemplazando el router incluido en Miniedit (básico, sin protocolos de ruteo) por un router proveniente de la librería de FRRouting.

Ahora es necesario editar el uso del router por parte de Mininet para poder ejecutar varios routers a la vez. Si no se realiza el siguiente cambio, la segunda ejecución de FRRouting detectará a la primera y correrán con el mismo PID, compartiendo interfaces.

Para impedir este conflicto se debe asignar como privados los directorios propios de FRRouting, de esta forma ambas instancias de FRRouting no se verán entre sí. Se debe realizar la siguiente modificación en la línea 2772:

```
elif 'LegacyRouter' in tags:
    newSwitch = net.addHost( name , cls=LegacyRouter)
    newSwitch = net.addHost( name , cls=LegacyRouter,
                             privateDirs=["/etc/frr", "/var/run/frr", "/var/log"])
```

Finalmente, se debe indicar a Miniedit para que inicie correctamente los routers. En la línea 3033 agregar a LegacyRouter junto con los demás:

```
if 'LegacySwitch' in tags:
    self.net.get(name).start( [] )
    info( name + ' ' )
if 'LegacyRouter' in tags:
    self.net.get(name).startRouter( [] )
    info( name + ' ' )
```

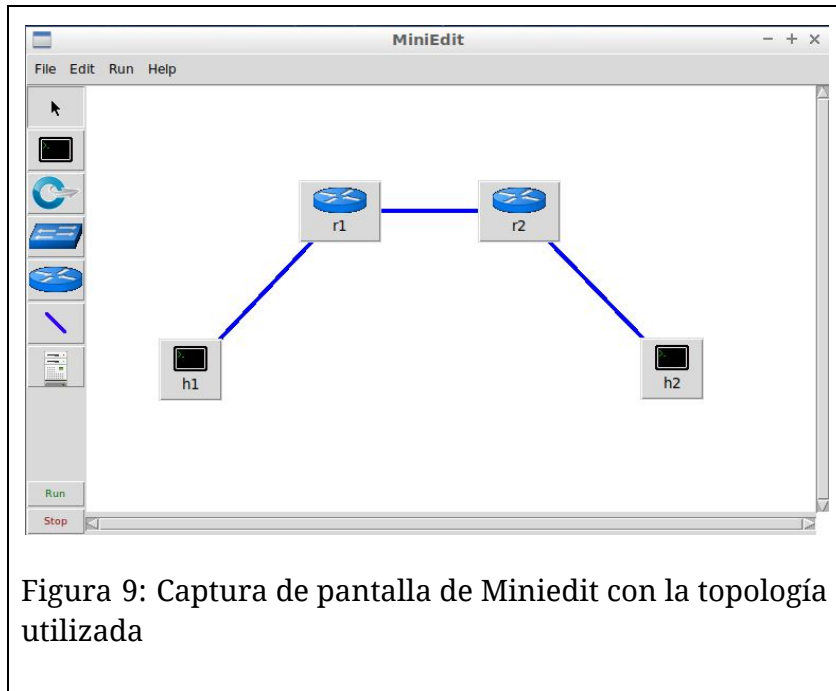
Es importante destacar que el resultado debe mantener la licencia de Mininet para el archivo **miniedit.py** y la licencia GPLv2 para las librerías provenientes de FRRouting.

Para ejecutar el programa, usar:

```
sudo ./miniedit.py
```

Escenario de ejemplo

Se explica a continuación el uso de esta herramienta para realizar pruebas simples del protocolo BGP. En primer lugar se deben crear los elementos de red necesarios como se muestra en la Figura 9.



Luego de presionar *Run*, es posible hacer click derecho a cada elemento y abrir una terminal. Miniedit suele agregar IPs automáticamente a cada interfaz, por lo que se recomienda eliminar las IPs de las interfaces de cada dispositivo (hosts y routers) utilizando un comando similar al siguiente:

```
ip address flush dev h1-eth0
```

Se pueden usar los siguientes comandos para configurar las IPs del host 1 y host 2 respectivamente:

```
ip address add 10.0.0.10/16 dev h1-eth0
Ip route add default via 10.0.0.1

ip address add 10.1.0.10/16 dev h2-eth0
Ip route add default via 10.1.0.1
```

En cada router, se debe activar el forwarding IP y abrir la consola de FRRouting con:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
vtysh
```

La consola de los routers funciona de manera muy similar a Cisco, los comandos ejecutados se muestran a continuación. La lista de comandos para el Router 1 es:

```
configure terminal
interface r1-eth1
ip address 10.0.0.1/16

[Ctrl-Z]
ping 10.0.0.10
[Se observa que se llega al host h1]

configure terminal
interface r1-eth0
ip address 10.9.0.1/16
exit

router bgp 100
network 10.0.0.0/16
neighbor 10.9.0.2 remote-as 200

[Ctrl-Z]
show ip route
```

Y para el Router 2:

```
configure terminal
interface r2-eth0
ip address 10.1.0.1/16

[Ctrl-Z]
ping 10.1.0.10
[Se observa que se llega al host h2]

configure terminal
interface r2-eth1
ip address 10.9.0.2/16

[Ctrl-Z]
ping 10.9.0.1
[Se observa que se llega al router r1]

configure terminal
router bgp 200
network 10.1.0.0/16
neighbor 10.9.0.1 remote-as 100

[Ctrl-Z]
show ip route
```

En la Figura 10 se muestra el resultado de esta configuración

```
"Host: r1"
root@traficoBBR:/Desktop/bbr2-mininet# vtysh
Hello, this is FRRouting (version 7.3).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

traficoBBR# show interface brief
Interface      Status VRF      Addresses
-----
lo             up     default  10.0.0.1/16
r1-eth0       up     default  10.9.0.1/16
r1-eth1       up     default  10.0.0.1/16

traficoBBR# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, D - SHARP,
       F - PBR, f - OpenFabric,
       > - selected route, * - FIB route, q - queued route, r - rejected route

C>* 10.0.0.0/16 is directly connected, r1-eth1, 00:19:26
B>* 10.1.0.0/16 [20/0] via 10.9.0.2, r1-eth0, 00:12:41
C>* 10.9.0.0/16 is directly connected, r1-eth0, 00:19:57
traficoBBR#

"Host: r2"
root@traficoBBR:/Desktop/bbr2-mininet# vtysh
Hello, this is FRRouting (version 7.3).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

traficoBBR# show interface brief
Interface      Status VRF      Addresses
-----
lo             up     default  10.0.0.1/16
r2-eth0       up     default  10.1.0.1/16
r2-eth1       up     default  10.9.0.2/16

traficoBBR# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, D - SHARP,
       F - PBR, f - OpenFabric,
       > - selected route, * - FIB route, q - queued route, r - rejected route

B>* 10.0.0.0/16 [20/0] via 10.9.0.1, r2-eth1, 00:13:06
C>* 10.1.0.0/16 is directly connected, r2-eth0, 00:19:43
C>* 10.9.0.0/16 is directly connected, r2-eth1, 00:18:53
traficoBBR#

"Host: h1"
root@traficoBBR:/Desktop/bbr2-mininet# ip -c a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: h1-eth0@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    group default qlen 1000
    link/ether 06:c1:c0:a0:f4:c5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.10/16 scope global h1-eth0
        valid_lft forever preferred_lft forever
root@traficoBBR:/Desktop/bbr2-mininet# ping 10.1.0.10 -c 3
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=62 time=0.071 ms
64 bytes from 10.1.0.10: icmp_seq=2 ttl=62 time=0.074 ms
64 bytes from 10.1.0.10: icmp_seq=3 ttl=62 time=0.509 ms

--- 10.1.0.10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2065ms
rtt min/avg/max/mdev = 0.071/0.218/0.509/0.205 ms
root@traficoBBR:/Desktop/bbr2-mininet#

"Host: h2"
root@traficoBBR:/Desktop/bbr2-mininet# ip -c a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: h2-eth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    group default qlen 1000
    link/ether 56:c5:1e:d3:e3:97 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.1.0.10/16 scope global h2-eth0
        valid_lft forever preferred_lft forever
root@traficoBBR:/Desktop/bbr2-mininet# ping 10.0.0.1 -c 3
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=63 time=0.047 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=63 time=0.125 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=63 time=0.070 ms

--- 10.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2102ms
rtt min/avg/max/mdev = 0.047/0.080/0.125/0.034 ms
root@traficoBBR:/Desktop/bbr2-mininet#
```

Figura 10: Resultado de la configuración de BGP en Miniedit, se logró la conectividad entre ambos hosts

Algoritmos de control de congestión

TCP Reno

El algoritmo TCP Reno, se basa en pensar que la congestión de la red se produce a causa de las pérdidas de paquetes.

Para determinar la velocidad de transferencia de los datos, Reno utiliza el algoritmo *Slow Start* para calcular el tamaño de la ventana de congestión. Inicialmente, la ventana es de longitud 1 MSS (Maximum Segment Size), y se incrementa de manera exponencial hasta que se produce una retransmisión de 3 ACKs duplicados, lo que produce que el algoritmo coloque el nuevo umbral de tamaño de la ventana a la mitad de la máxima alcanzada hasta que ocurrió este evento.

Luego de esto, la ventana aumenta de manera lineal gracias al algoritmo llamado *Congestion Avoidance*. A partir de esto pueden surgir dos casos:

- Que se repita la recepción de 3 ACKs duplicados, lo cual establece el nuevo tamaño de la ventana a la mitad del nuevo máximo alcanzado,
- Que ocurra un timeout y no se recibe un ACK, lo cual reinicia la ventana al tamaño de 1 MSS, y repite el proceso inicial utilizando slow start, como se puede ver en la Figura 11

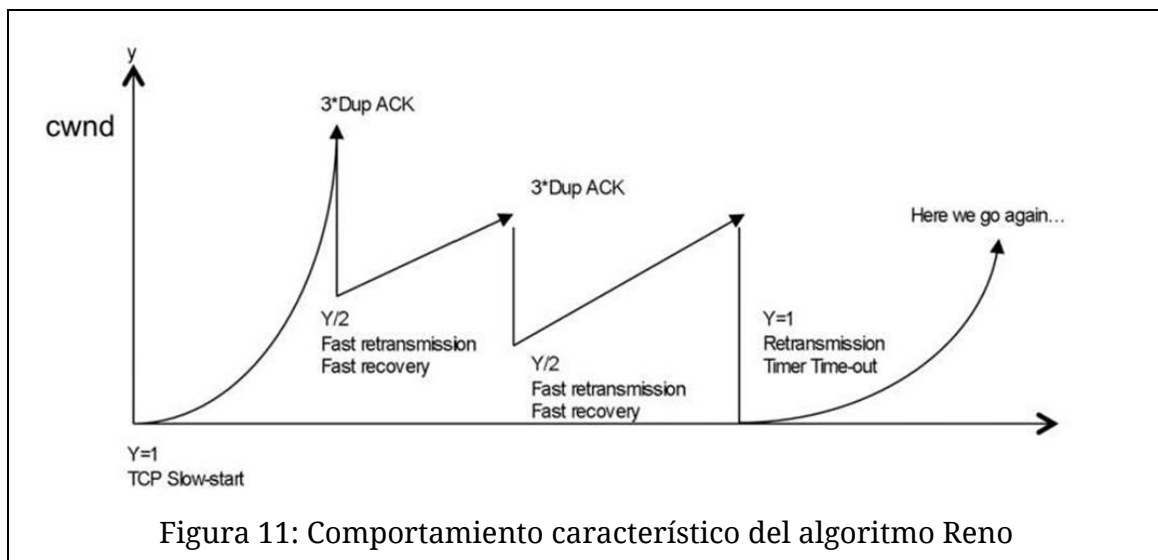


Figura 11: Comportamiento característico del algoritmo Reno

TCP CUBIC

El algoritmo CUBIC surge por el gran crecimiento en ancho de banda de las redes, en las cuales un algoritmo con respuesta lenta, puede desperdiciar la capacidad de los enlaces.

Su funcionamiento se basa en aumentar la tasa de transmisión hasta que ocurra un evento de congestión. Cuando ocurre este evento se establece un umbral (W_{max}) de la ventana y se reinicia la transmisión con una ventana de congestión menor. De no haber congestión, el tamaño va incrementándose rápidamente siguiendo la parte cóncava de una cúbica, como se ve en la Figura 12, y cuando se va acercando al umbral su aumento es menor.

Si al llegar al umbral, no hubo congestión, empieza a aumentar lentamente siguiendo la parte convexa de la cúbica. Cabe aclarar que mientras que el evento de congestión se produzca a un valor de ventana de congestión mayor que el anterior, la curva va a tener como referencia ese valor. Por el contrario si el evento se produce a un valor menor que el anterior, la ventana se reduce a la mitad entre el W_{max} anterior y el tamaño de inicio.

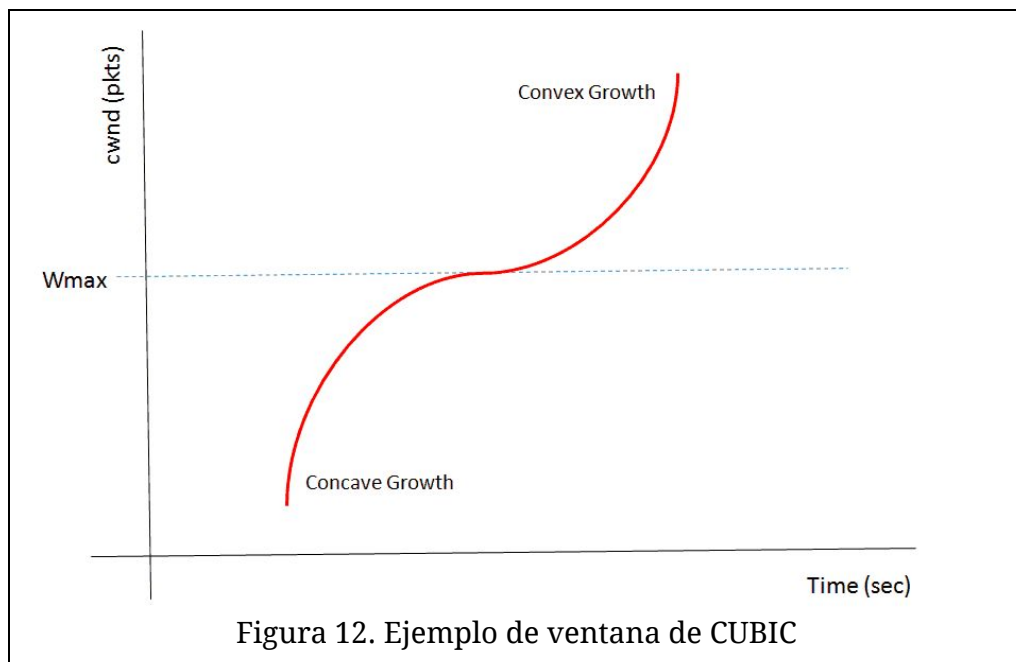


Figura 12. Ejemplo de ventana de CUBIC

La ecuación que rige el tamaño de la ventana de congestión es la siguiente:

$$W_{cubic}(t) = C * (t - K)^3 + W_{max}$$

Donde W_{max} es el último tamaño de ventana antes que ocurriera el evento de congestión, C es una constante que define la agresividad del aumento de la ventana, t es la variable de tiempo y K es una constante determinada por la ecuación:

$$K = \sqrt[3]{(W_{max} * (1 - \beta) / C)}$$

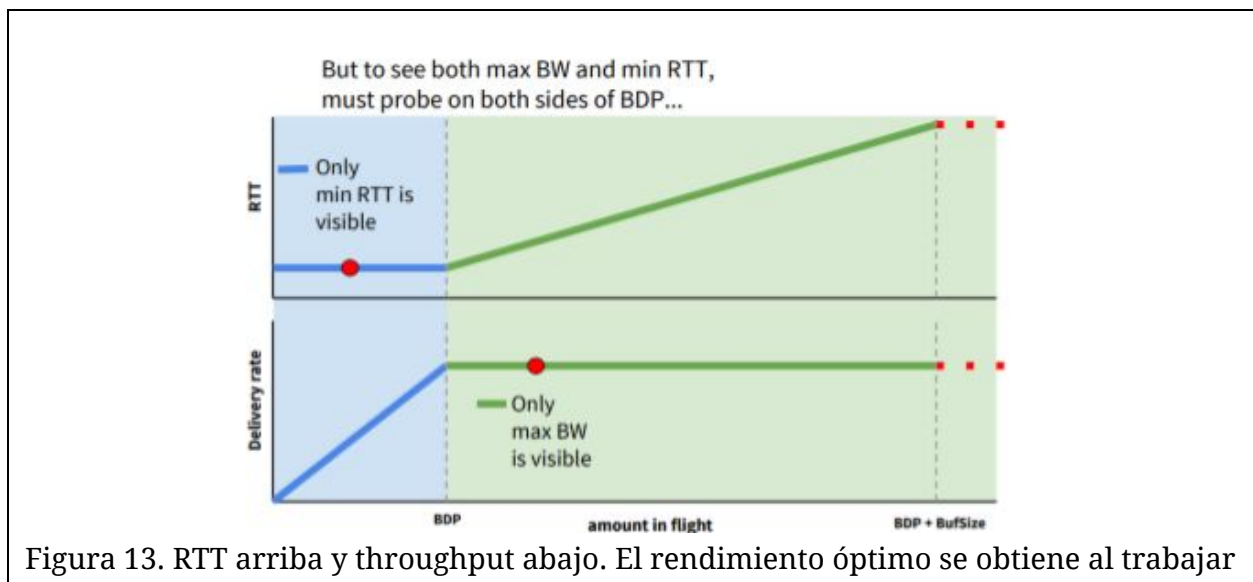
Donde β es el factor de reducción de la ventana, que para el caso $W_{cubic}(0) = W_{max} * \beta$. Este valor se suele establecer en 0.7, ya que un valor menor como 0.5 haría muy lento el aumento y uno mayor estaría muy cerca del umbral lo que no modificaría prácticamente nada el tamaño de la ventana.

TCP BBRv1

Otro algoritmo que hemos utilizado en esta experiencia es BBRv1, el cual utiliza otro concepto de congestión no basado en paquetes perdidos, sino en la latencia y el ancho de banda del cuello de botella de la red.

El funcionamiento de este se basa en ir aumentando la tasa de transmisión hasta encontrarse con un buffer, lo cual indica que este es el máximo ancho de banda disponible en la red. Si la tasa de transmisión se incrementa por encima del cuello de botella, empiezan a encolarse paquetes en el buffer, la latencia comienza a incrementar y el algoritmo debe bajar su velocidad para no llenar el buffer (ver Figura 13 y 14).

El algoritmo busca trabajar en el punto óptimo, donde las latencias son mínimas y el ancho de banda (throughput) utilizado es máximo, contrario a los algoritmos basados en pérdidas, los cuales dejan la latencia incrementar y llenan el buffer hasta que se pierdan paquetes.



en el punto de quiebre

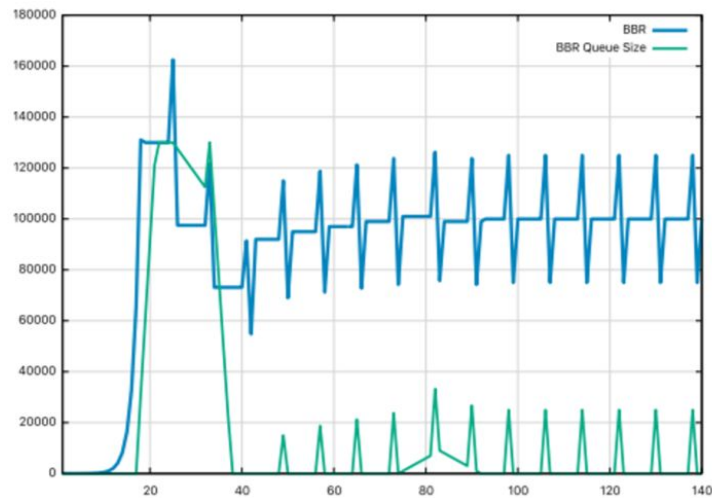


Figura 14. Ejemplo de funcionamiento de BBRv1. Throughput en azul y ocupación de los buffer en verde

El BDP (Bandwidth Delay Product) es la multiplicación de la latencia y el throughput. Representa la cantidad de paquetes circulando en la red en un momento dado y se usa como factor de escalamiento que afecta a la ventana de congestión, lo cual termina afectando al throughput. El algoritmo trabaja en cuatro estados como se muestra en la figura 15:

- **STARTUP:** Al iniciarse la conexión, se incrementa el tamaño de la ventana en un valor de 2.89 (llamado *cwnd_gain*) cada RTT, es decir, se aumenta la tasa de transmisión, de manera exponencial, similar a *slow start* en TCP Reno.
- **DRAIN:** Al estimar que el ancho de banda utilizable no aumenta más de un 25% cuando se reciben 3 ACKs en el proceso STARTUP, se cambia el funcionamiento a modo DRAIN, su objetivo es drenar los buffers que se llenaron por el proceso anterior. Durante esta etapa, se le da una ganancia a la ventana de congestión de $1/cwnd_gain$.
- **PROBE_BW:** Una vez que el algoritmo de BBR termina sus fases iniciales, entra en su estado de PROBE_BW, donde se busca adaptarse al ancho de banda disponible del canal aumentando la tasa de transmisión de datos buscando no aumentar la latencia. Si se detecta un aumento de la latencia, se disminuye la tasa de

transmisión en un factor multiplicativo fijo para vaciar los buffers. BBR se encuentra en este estado la mayoría del tiempo.

- PROBE_RTT: Es un estado que ocurre periódicamente cada 10 segundos y se utiliza para encontrar los valores de RTT mínimos. En este estado se baja la ventana de congestión a un valor mínimo ($4 \times \text{MSS}$) durante 200ms. Acabado este procedimiento, se pasa al estado de PROBE_BW o STARTUP. Este procedimiento se ve reflejado en las zonas vistas en la Figura 15.

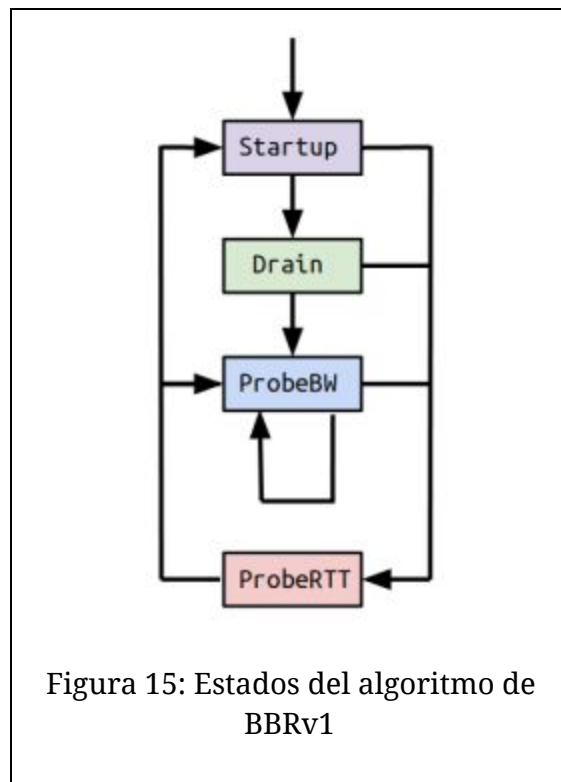


Figura 15: Estados del algoritmo de BBRv1

TCP BBRv2

Está basado en TCP BBR, y está diseñado para solucionar ciertos problemas que se encontraron en la versión anterior:

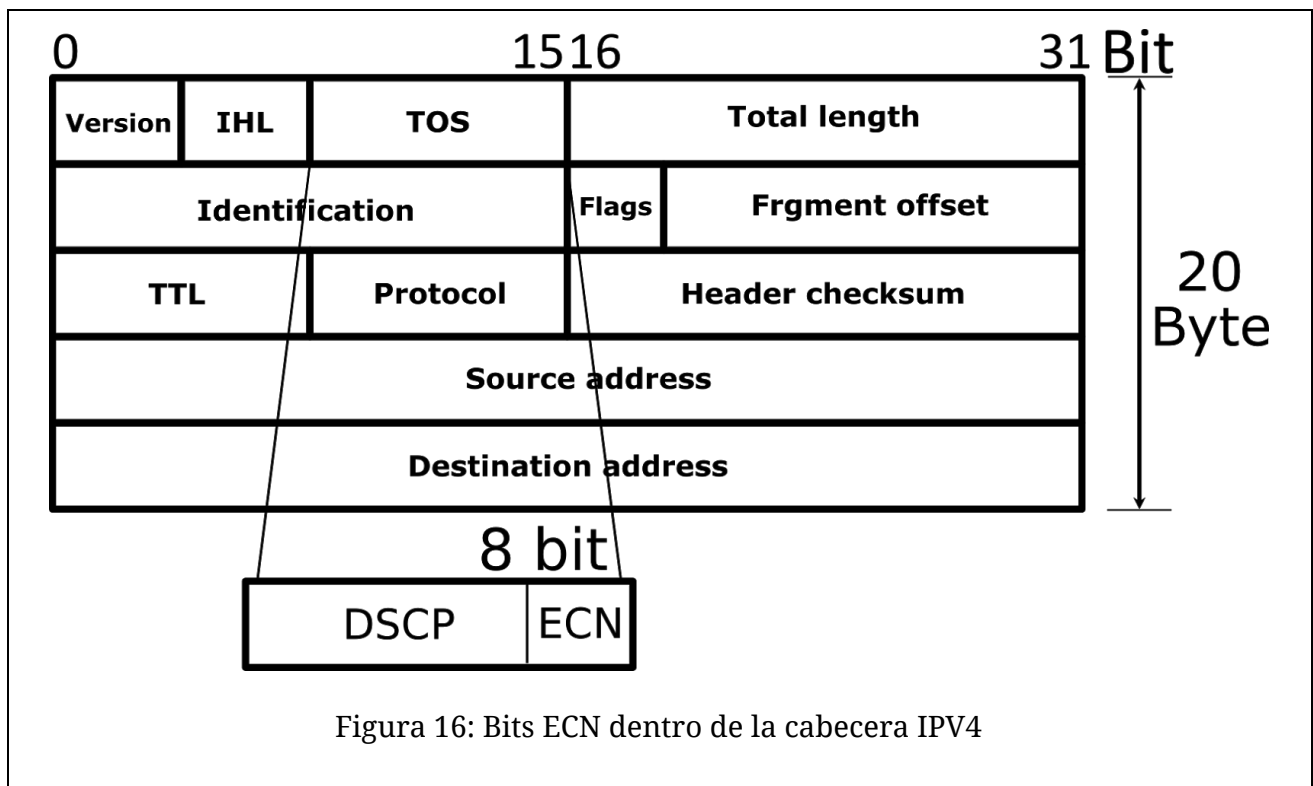
- Los PROBE_RTT ocurren cada intervalos muy grandes de tiempo lo que provoca que BBRv1 sea lento para converger
- El algoritmo toma demasiado ancho de banda en comparación con Reno y Cubic al compartir un cuello de botella

- No toma en cuenta las pérdidas, y puede saturarse mucho si el tamaño de cola es menor a $1.5 \cdot \text{BDP}$.
- En ambientes con alta cantidad de Aggregation, como por ejemplo wi-fi, se experimentaban throughputs muy bajos
- El PROBE_BW es muy drástico y puede provocar una gran latencia por momentos.
- En vez de reducir drásticamente el tamaño de la ventana ($4 \cdot \text{MSS}$) cada vez que hace PROBE_RTT, se reduce ahora a $0.5 \cdot \text{BDP}$ para mantener un alto nivel de throughput.

ECN

BBRv2 se diferencia con BBRv1 en el sentido que se tiene en cuenta el efecto de las pérdidas en el flujo de datos, utilizando del campo TOS de la cabecera IPv4, los últimos dos bits para ECN (Explicit Congestion Notification) como se muestra en la Figura 16. Hay que tener en cuenta que es efectivo el uso de estos bits solamente cuando todos los hosts involucrados de la red soportan el procesamiento del mismo. Hoy en día la mayoría de los sistemas operativos y equipos en data centers incluyen implementaciones de ECN para TCP.

El uso de estos bits para el control de congestión fue popularizado por un algoritmo de control de congestión TCP: DCTP (Data-Center TCP). Google ha decidido implementar de forma similar la utilización de este campo con BBRv2, ya que en BBRv1 para colas pequeñas en los cuellos de botella ($< 1.5 \cdot \text{BDP}$), hay pérdidas de paquetes producto de que no llega a responder el algoritmo. Como resultado, la máquina de estado del algoritmo BBRv2 no solo tiene en cuenta el RTT y el throughput, si no también tiene en cuenta las pérdidas y el ECN.



Aggregation

Es una técnica que se comenzó a implementar en el protocolo 802.11n (WiFi) que permite mejorar la eficiencia de la transmisión cuando se necesita enviar gran cantidad de datos pequeños. Si se envía una trama cada vez que la aplicación necesita enviar datos, es posible que se terminen enviando muchos paquetes pequeños, desperdiciando el ancho de banda.

Por lo tanto, Aggregation consiste en utilizar un buffer, esperando a que la cantidad de información a enviar sea lo suficiente grande antes de enviar un paquete. Para esto se utiliza el algoritmo de Nagle:

```
if there is new data to send then
  if the window size  $\geq$  MSS and available data is  $\geq$  MSS then
    send complete MSS segment now
  else
    if there is unconfirmed data still in the pipe then
      enqueue data in the buffer until an acknowledge is received
    else
      send data immediately
    end if
  end if
end if
```

Esto ocurre en la capa de enlace gracias al protocolo 802.11. Entonces el aggregation también ocurre cuando un host debe enviar ACKs: si por ejemplo se deben enviar 10 ACKs, la placa de red inalámbrica en vez de enviarlos en el momento correcto los enviará a todos juntos.

Esto afecta en gran medida el rendimiento de BBRv1 en redes inalámbricas, ya que éste espera a recibir ACKs para actualizar los tamaños de ventanas de congestión. Como solución a este problema, BBRv2 al recibir gran cantidad de ACKs agrupados, realiza una estimación de su distribución original para actualizar el algoritmo de una manera más suave.

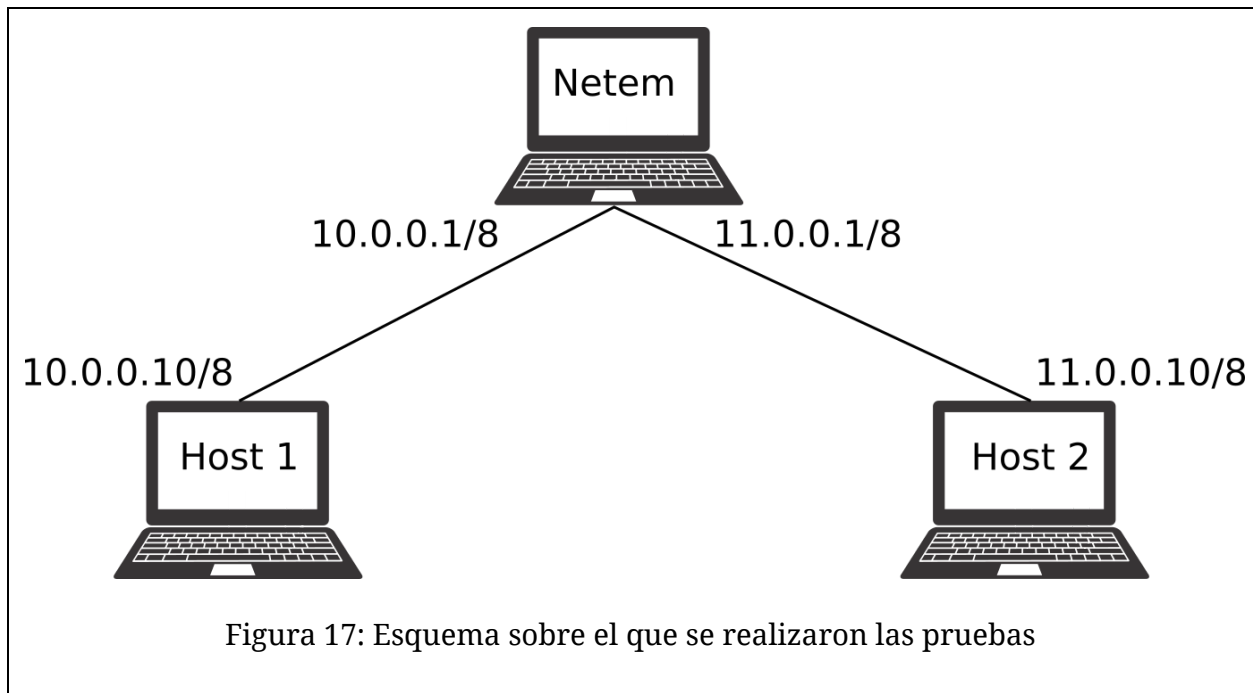
TCO

Es importante no confundir Aggregation con TCO (TCP Offload Engine). TCO es el proceso por el cual el CPU transfiere gran cantidad de datos a la NIC, en donde se segmentarán los paquetes y se colocarán headers Ethernet, TCP e IP a la NIC.

Esto sirve para liberar trabajo al procesador y es necesario si se buscan alcanzar velocidades de varios Gbps. Al sniffear tráfico en una interfaz con TCO se pueden observar paquetes que superan el MTU, pero en realidad estos paquetes son segmentados por la NIC antes de ser enviados.

Escenario

Se creó un escenario simple con dos host conectados a un mismo switch mediante un cable como se ve en la Figura 17. En un principio se utilizaron las funciones que modifican los parámetros del cable respecto al ancho de banda, el retardo, las pérdidas, entre otros, pero los resultados obtenidos no fueron los esperados. Debido a esto se decidió retirar el switch y poner un servidor al cual se le configuró un netem, herramienta que permite reemplazar estas funcionalidades de los cables.



Las pruebas se realizaron utilizando la herramienta iperf3, ya que permite la transmisión de datos a altas velocidades. Se estableció en el host 1 un servidor, mientras que el host 2 actuó como cliente. Se transmitieron datos desde el cliente al servidor y con tcpdump se capturó el tráfico generado, para luego con captcp realizar los gráficos de throughput, RTT, ventana de congestión e inflight. También se fue variando el algoritmo utilizado con el objetivo de poder comparar los rendimientos de cada uno para los distintos casos analizados.

Para automatizar las pruebas se utilizó un script hecho en Python y Mininet. Como dependencia adicional a las indicadas anteriormente en este trabajo se debe instalar:

```
sudo apt install colored-logs
```


Resultados

Escenario 1

El objetivo de este escenario es ver de manera general el funcionamiento de todos los algoritmos de control de congestión analizados.

Al observar las siguientes figuras, especialmente las referidas a los tamaños de ventana de congestión, se pueden notar que todos los algoritmos trabajan de forma distinta para encontrar el cuello de botella de la red, dibujando las curvas características de cada uno descriptas anteriormente.

Se puede ver también que los gráficos de tamaño de ventana de congestión, RTT y longitud de cola son en cierta forma equivalentes. Una mayor ventana de congestión provoca que se envíen más paquetes al mismo tiempo, a partir de cierto punto esto produce saturación en el cuello de botella de la red que se observa como aumento del tamaño de la cola. Por último, al aumentar el tamaño de cola los paquetes demoran más tiempo en transitar por la red (mayor RTT), ya que se encuentran con una cola en donde pasan cierto tiempo esperando el turno para proseguir.

- Reno: El algoritmo va aumentando el tamaño de la ventana de manera lineal hasta que se producen tres ACKs duplicados o se pierdan paquetes por saturación en el buffer y baja el tamaño de la ventana a la mitad. No se observaron eventos donde se produzca un timeout, porque la ventana no bajó nunca a cero.
- Cubic: Se observa que al obtener una pérdida inicia su curva cúbica característica que busca aplanarse al acercarse al valor del umbral. En el caso que el evento de congestión tiene un valor de ventana menor que el evento anterior, el umbral se establece en un valor menor, lo que hace que el crecimiento de la curva no sea tan agresivo sino que más plano.
- BBRv1: se puede ver que el throughput oscila bastante en el límite del ancho de banda. Para el caso de los buffers se ve que va acumulando paquetes, lo cual aumenta el RTT, y lleva a que baje el throughput para vaciarlos.
- BBRv2: se ve un comportamiento similar al de BBRv1 que al empezar a detectar un aumento en el RTT producto de paquetes en el buffer, baja su throughput y los vacía. Al igual que BBRv1, es capaz de mantener un alto throughput y un bajo RTT al mismo tiempo.

Parámetros del escenario:

- Ancho de banda: 10 Mbps
- Delay: 100 ms
- Pérdidas: 0%
- Latencia de buffer: 50 ms

- Duración de prueba: 20s

TCP Reno

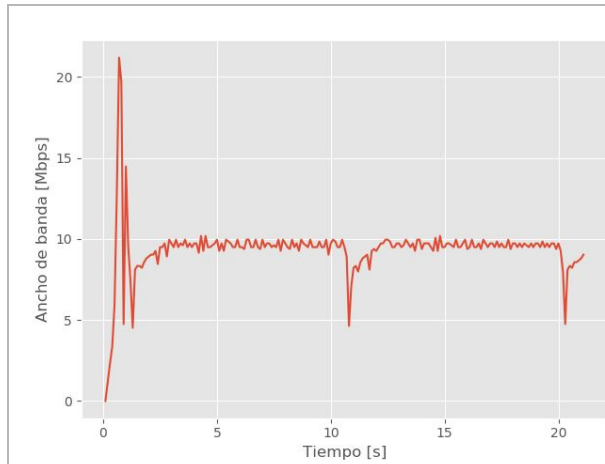


Figura 18.1: Throughput

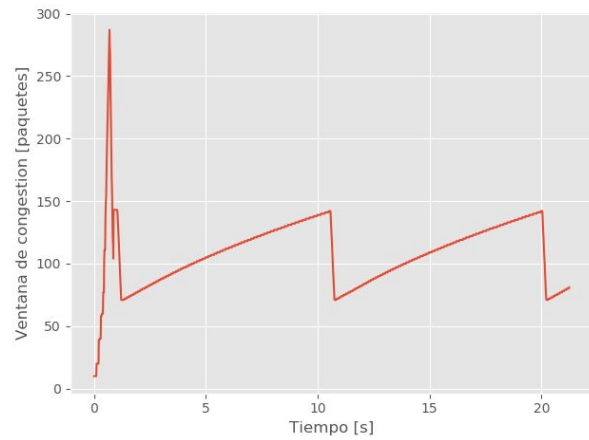


Figura 18.2: Tamaño de ventana de congestión

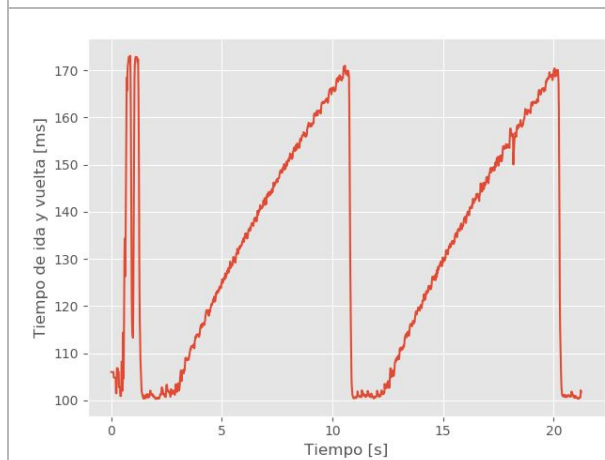


Figura 18.3: Round Trip Time

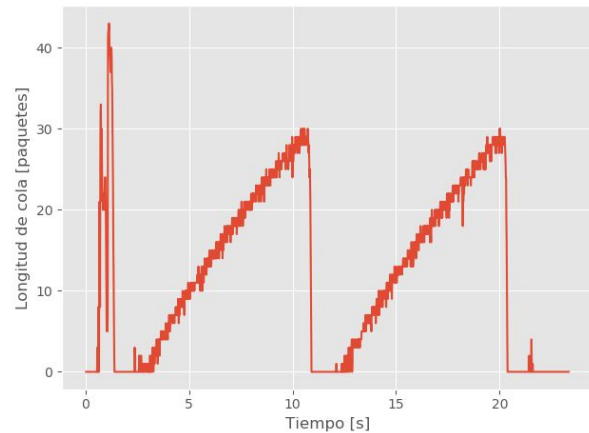


Figura 18.4: Longitud de cola

TCP Cubic

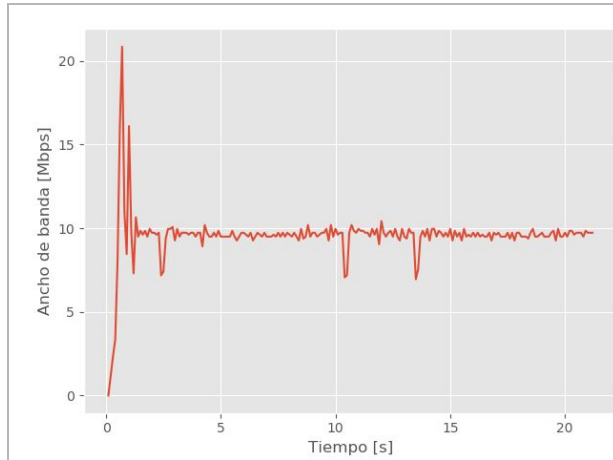


Figura 19.1: Throughput

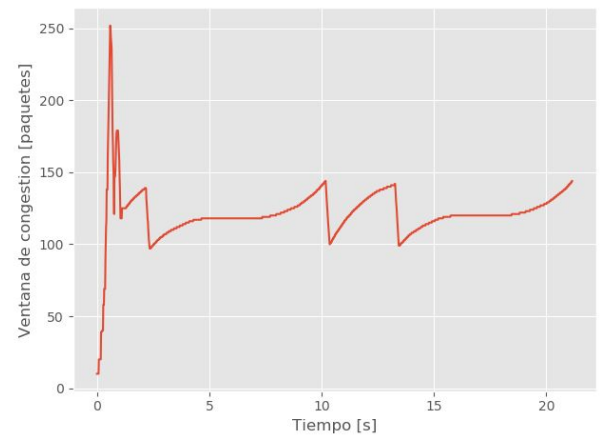


Figura 19.2: Tamaño de ventana de congestión

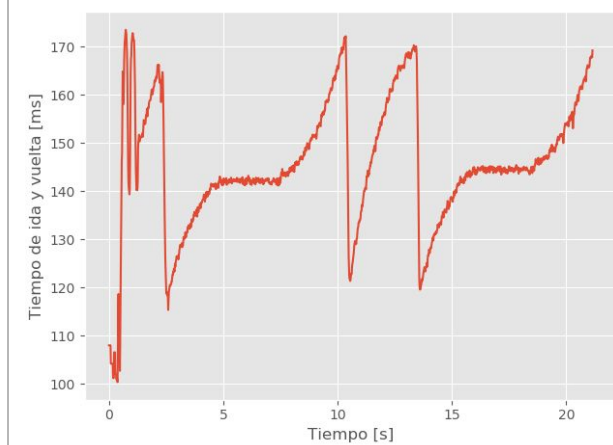


Figura 19.3: Round Trip Time

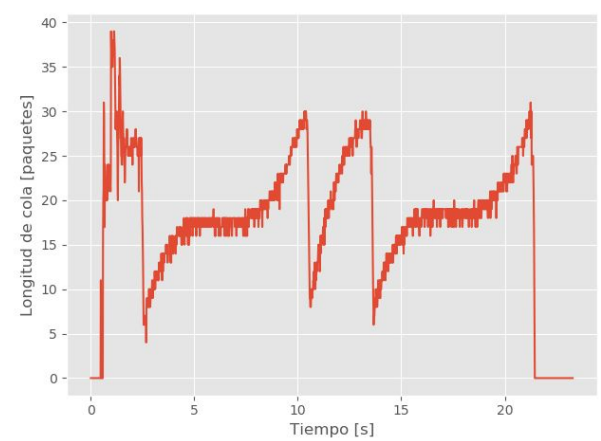


Figura 19.4: Longitud de cola

TCP BBRv1

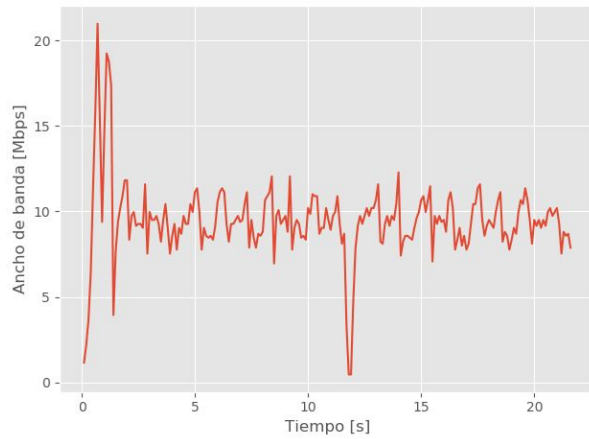


Figura 20.1: Throughput

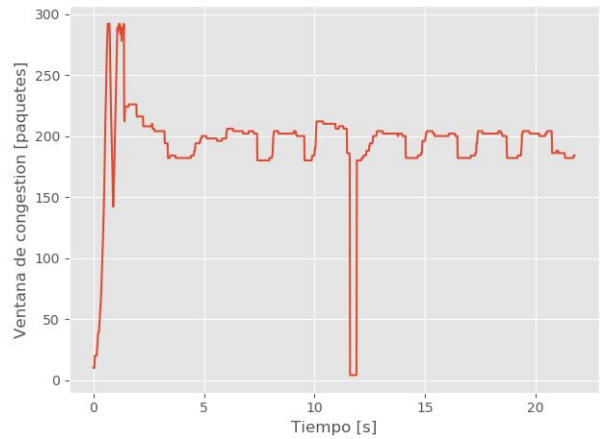


Figura 20.2: Tamaño de ventana de congestión

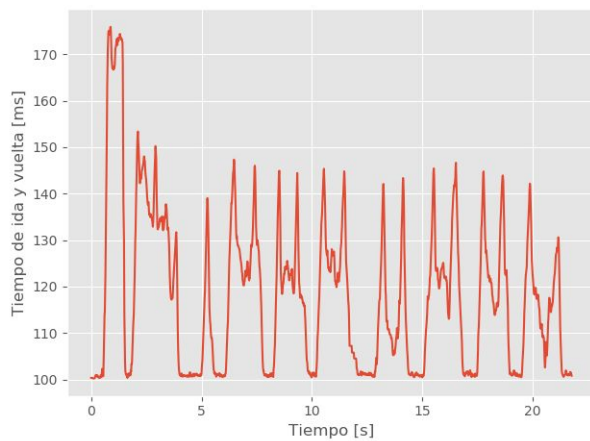


Figura 20.3: Round Trip Time

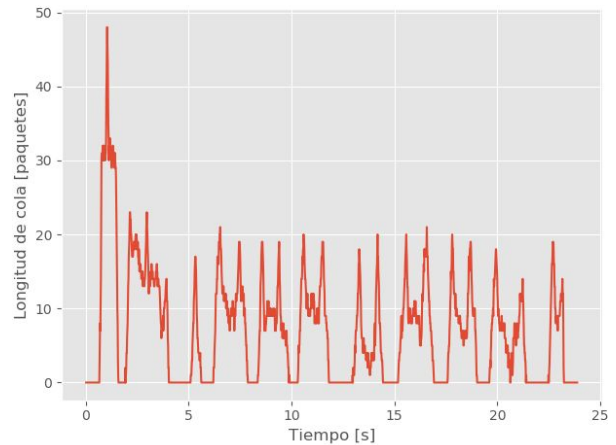
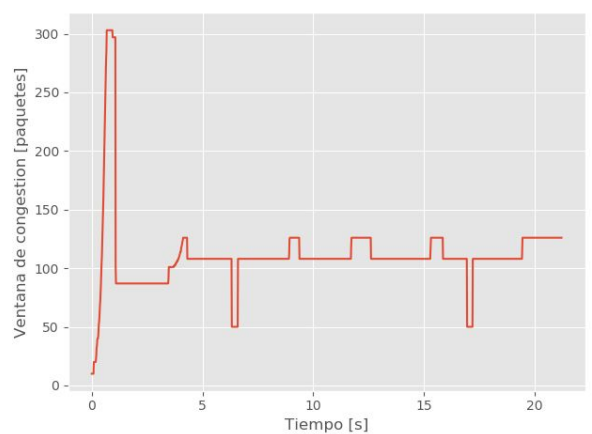
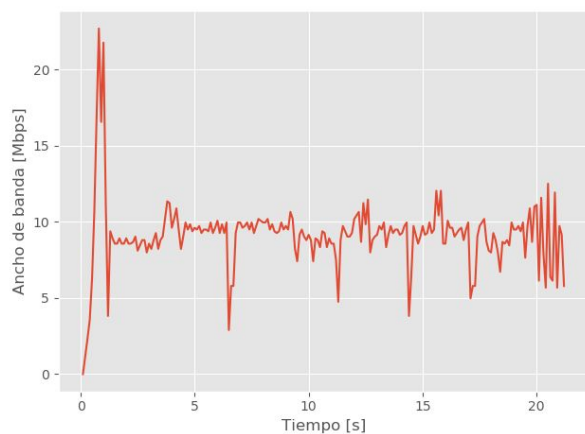
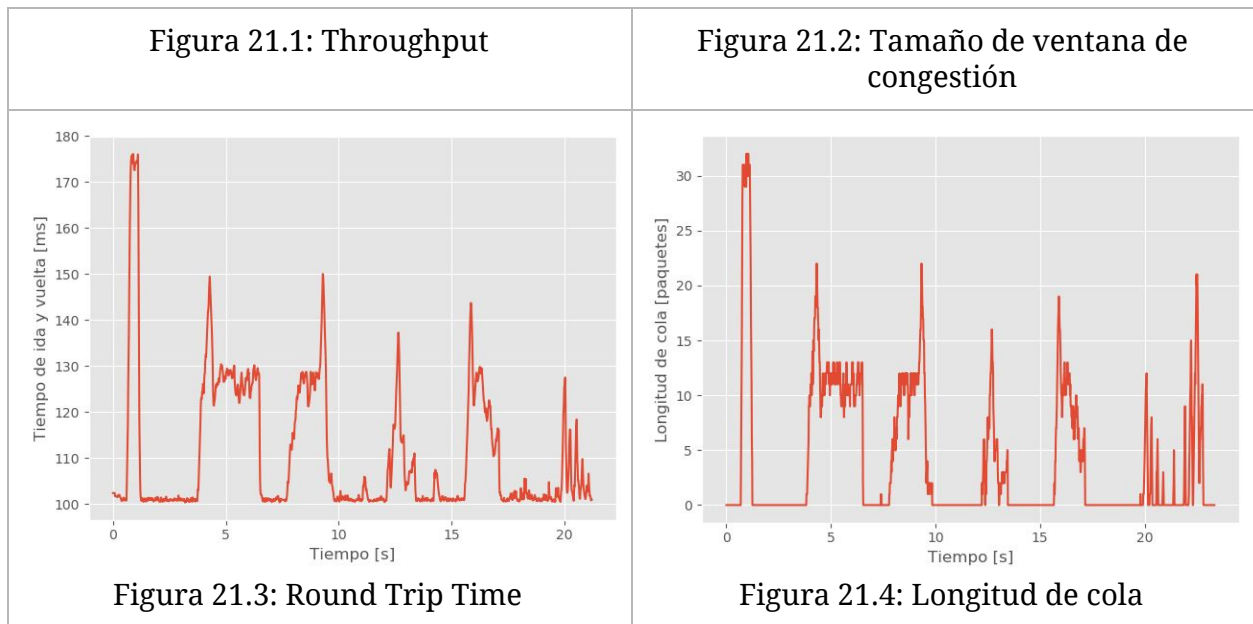


Figura 20.4: Longitud de cola

TCP BBRv2





Escenario 2

Para analizar más en detalle las diferencias entre BBRv1 y BBRv2, creamos tráfico durante un minuto. Se puede observar que ambos algoritmos pudieron detectar correctamente la presencia del cuello de botella sin necesidad de llenar el buffer.

Por otro lado, es posible contemplar cómo en BBRv2 los PROBE_RTT son más frecuentes y no tan profundos. Esta es una de las principales diferencias entre los dos algoritmos.

BBRv1 mantuvo el throughput más cerca del límite de los 100 Mbps a diferencia de BBRv2 que oscila entre los 80 Mbps. En contraparte, en BBRv1 se observa una mayor ocupación de los buffers, lo que genera un aumento en la latencia.

Parámetros del escenario:

- Ancho de banda: 100 Mbps
- Delay: 30 ms
- Pérdidas: 0%
- Latencia de buffer: 30 ms
- Duración de prueba: 60s

TCP BBRv1

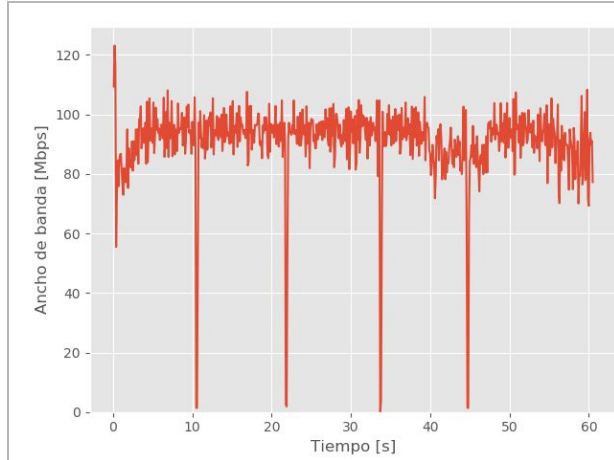


Figura 22.1: Throughput

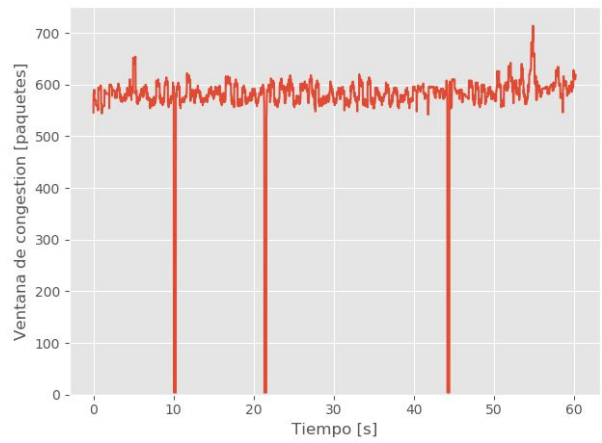


Figura 22.2: Tamaño de ventana de congestión

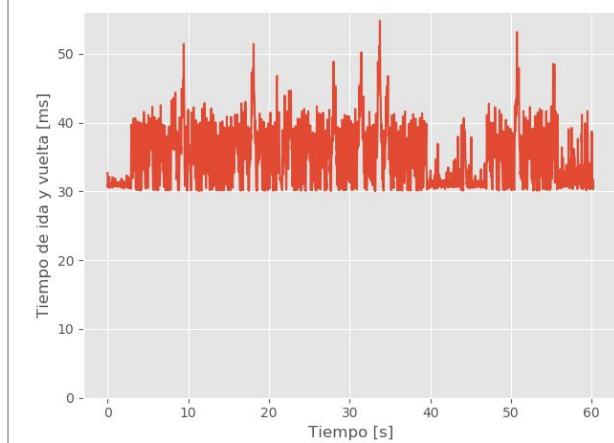


Figura 22.3: Round Trip Time

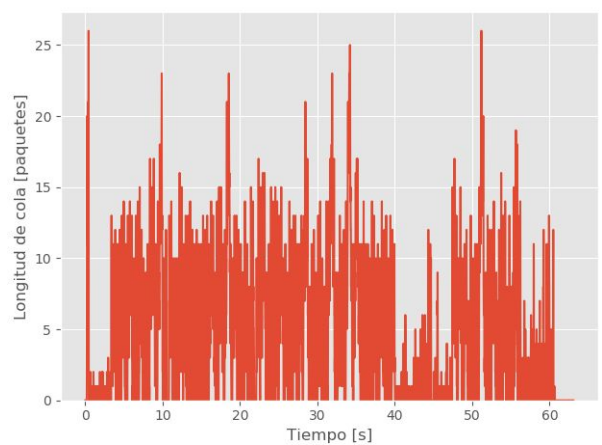
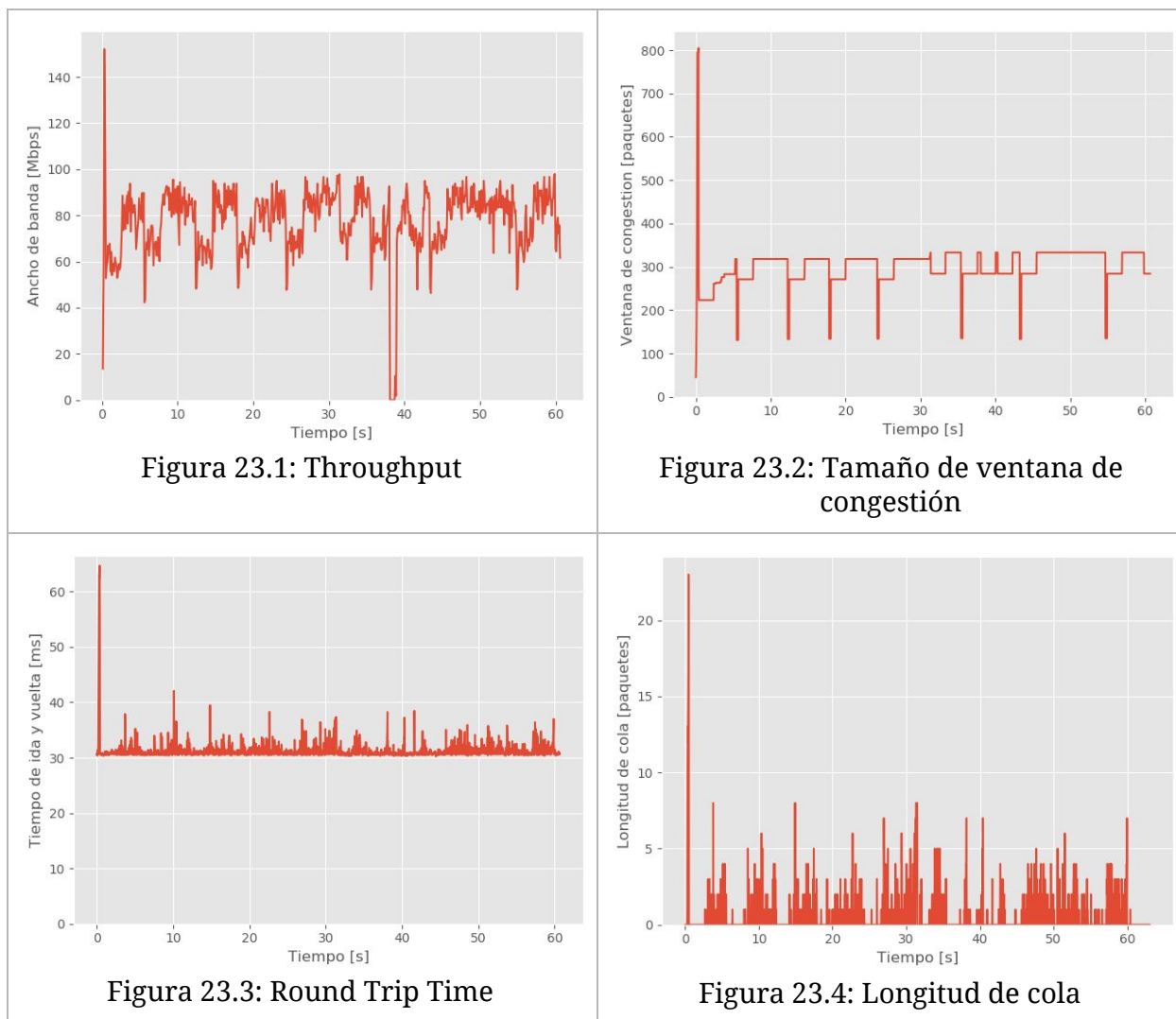


Figura 22.4: Longitud de cola

TCP BBRv2



Escenario 3

Para mostrar el comportamiento diferente de los algoritmos ante las pérdidas, generamos pérdidas aleatorias con netem.

Se puede observar que las pérdidas en BBRv1 no afectan considerablemente el tamaño de ventana de congestión, y por lo tanto su throughput. Debido a que en BBRv2 se corrigió este problema, se puede comprobar que las pérdidas afectan pero en menor medida en comparación con Reno y Cubic.

Respecto a Reno y Cubic, como sí dependen de las pérdidas, su velocidad de transferencia no llega a alcanzar los 5 Mbps.

Parámetros del escenario:

- Ancho de banda: 100 Mbps
- Delay: 100 ms
- Pérdidas: 0.5%, correlación 25%
- Latencia de buffer: 50 ms
- Duración de prueba: 10s

TCP Reno

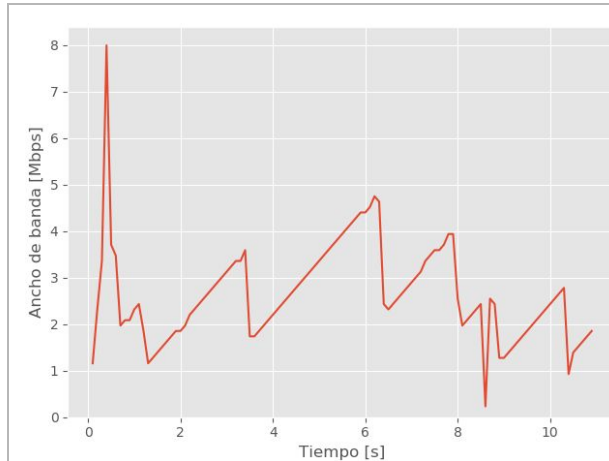


Figura 24.1: Throughput

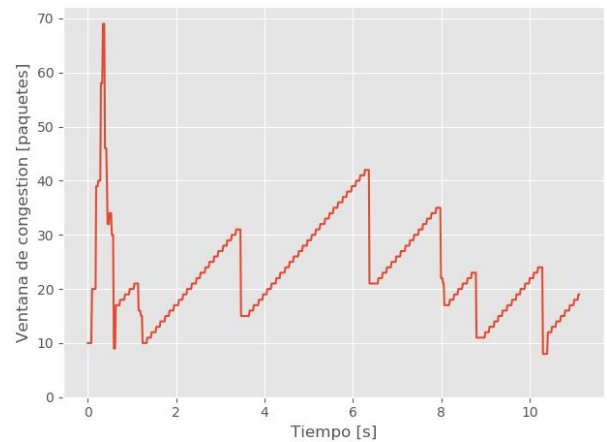


Figura 24.2: Tamaño de ventana de congestión

TCP Cubic

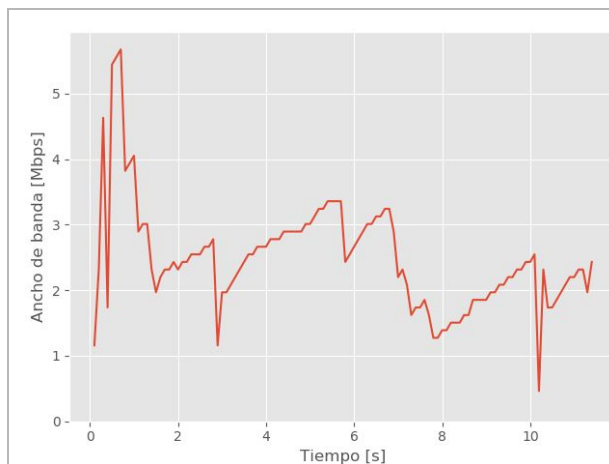


Figura 25.1: Throughput

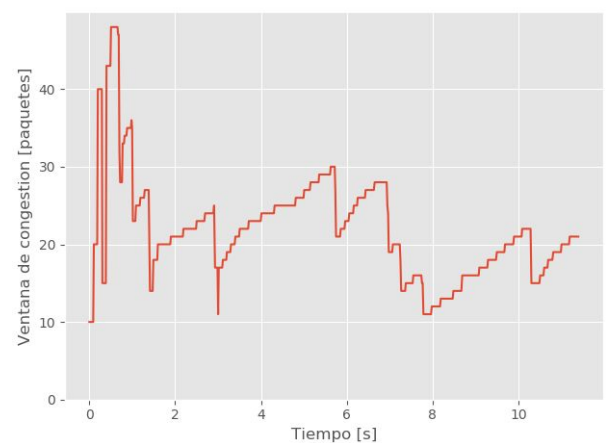


Figura 25.2: Tamaño de ventana de congestión

TCP BBRv1

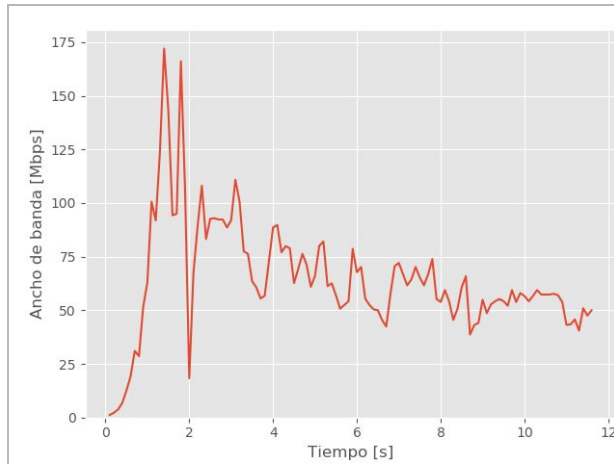


Figura 26.1: Throughput

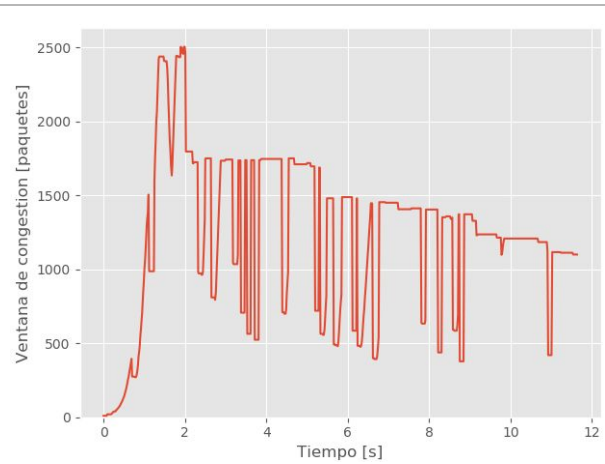


Figura 26.2: Tamaño de ventana de congestión

TCP BBRv2

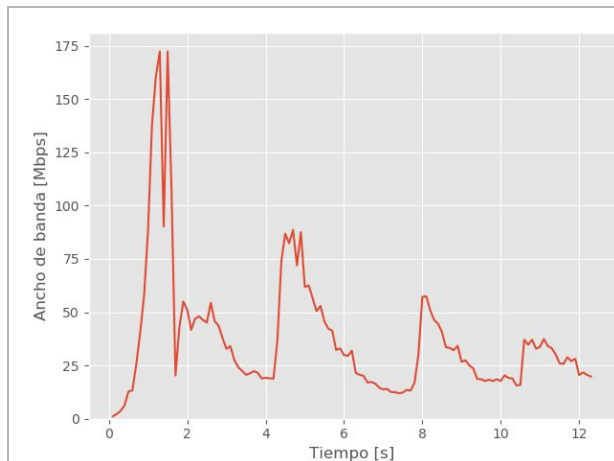


Figura 27.1: Throughput

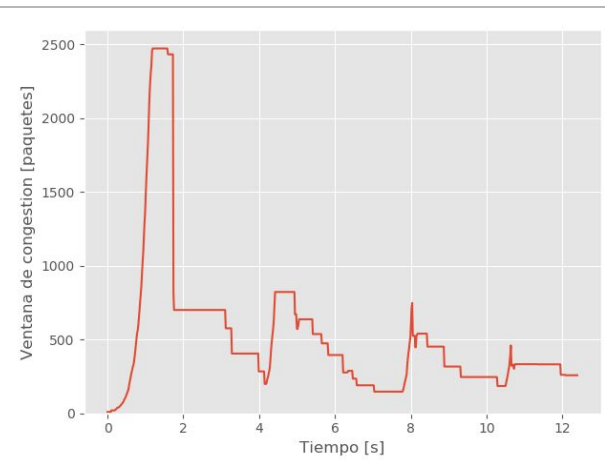


Figura 27.2: Tamaño de ventana de congestión

Escenario 4

En un momento encontramos una situación en la cual el rendimiento de BBRv2 es mucho mayor al de BBRv1. En este escenario mostramos que BBRv1 tiene grandes problemas cuando se le asigna sólo un núcleo a la VM, indicando que probablemente el algoritmo de BBRv1 es más exigente en el uso del procesador para actualizar el tamaño de la ventana de congestión.

Es importante destacar que en este escenario existe un sólo núcleo que tiene que realizar gran cantidad de trabajo:

- Correr el algoritmo de control de congestión en ambos hosts
- Simular un cuello de botella con netem
- Realizar la transferencia de datos con iperf3
- Capturar el tráfico generado con tcpdump
- Interrogar 100 veces por segundo el tamaño de cola presente en el cuello de botella
- Interrogar 100 veces por segundo el RTT, tamaño de ventana de congestión, y más estadísticas que no se utilizan en este trabajo
- Y correr el resto de procesos comúnmente necesarios del sistema operativo

Parámetros del escenario:

- Ancho de banda: 100 Mbps
- Delay: 50 ms
- Pérdidas: 0%
- Latencia de buffer: 50 ms
- Duración de prueba: 60s

TCP Reno

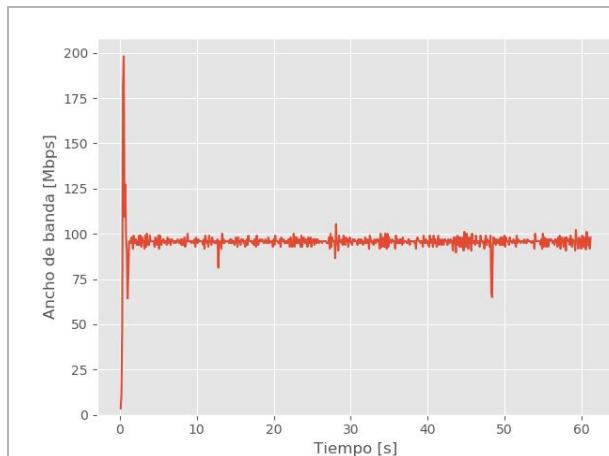


Figura 28.1: Throughput

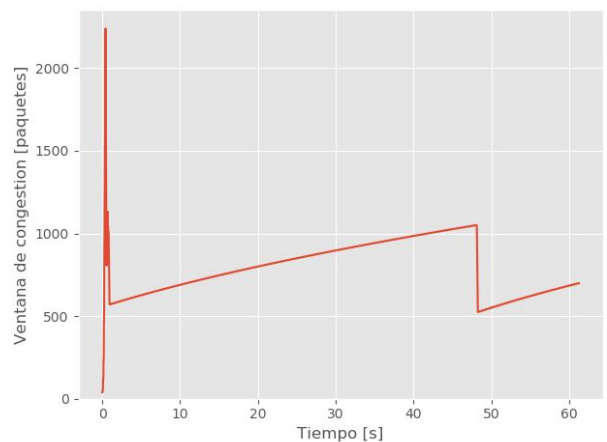


Figura 28.2: Tamaño de ventana de congestión

TCP BBRv1

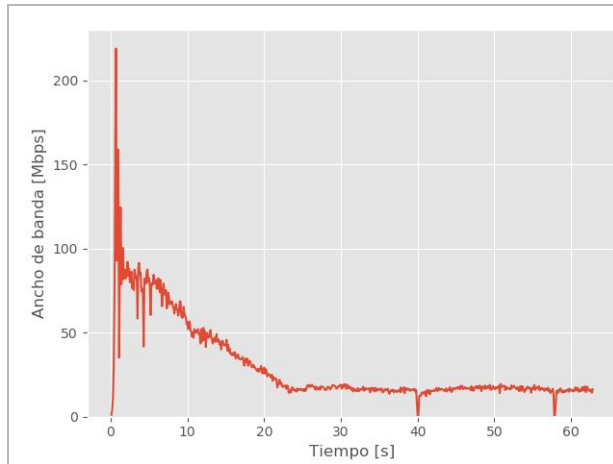


Figura 29.1: Throughput

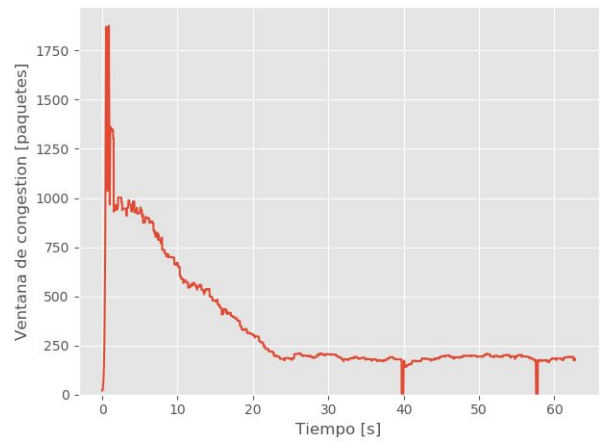


Figura 29.2: Tamaño de ventana de congestión

TCP BBRv2

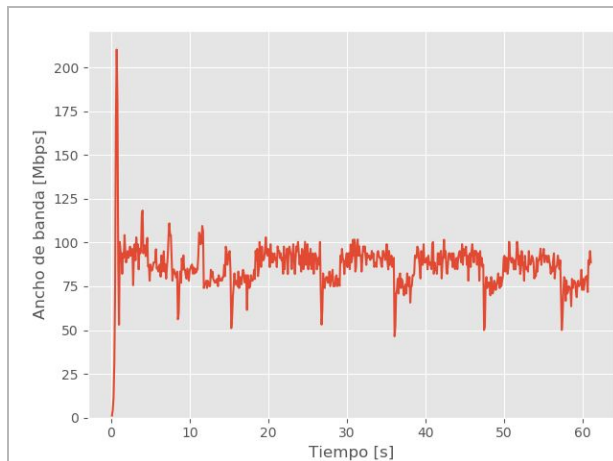


Figura 30.1: Throughput

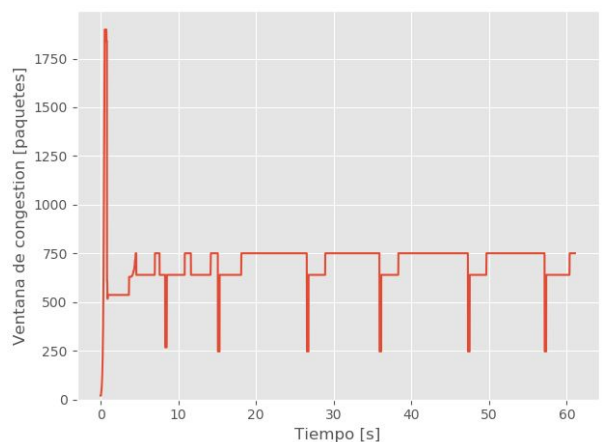


Figura 30.2: Tamaño de ventana de congestión

Escenario 5

Una de las ventajas de BBRv2 sobre BBRv1 es el rendimiento ante redes con buffers relativamente pequeños. Específicamente, BBRv1 puede tener problemas cuando los buffers son menores a 1.5 veces el BDP y existen múltiples flujos compartiendo el cuello de botella.

En nuestra simulación utilizamos un único flujo con un tamaño de buffer muy pequeño (1% del BDP). Observar cómo BBRv1 llenó la cola reiteradas veces (que ronda los 15 paquetes), produciendo una gran cantidad de pérdidas y una posterior disminución en el

throughput, mientras que BBRv2 se mantiene transmitiendo cerca del límite de ancho de banda de 100Mbps.

BBRv1 intenta mantenerse en el punto óptimo de máximo throughput y mínima latencia, pero periódicamente aumenta la tasa de transmisión para observar si los buffers de la red se comienzan a llenar. Este aumento de la tasa de transmisión sigue unos parámetros fijos, que pueden ser muy agresivos y saturar los buffers si éstos son muy pequeños. Esta saturación provoca pérdida de paquetes, en principio BBRv1 ignora las pérdidas pero a partir de cierto punto la gran cantidad de paquetes perdidos afecta al throughput.

BBR2 en cambio realiza este procedimiento adaptativamente, observando las pérdidas e intentando no saturar los buffers. En la prueba se logra apreciar que la nueva versión del algoritmo soluciona este problema.

Finalmente, se puede observar también un problema con Reno, que al disminuir el tamaño de ventana de congestión a la mitad, obtiene caídas considerables de throughput. Esto no se observa de una forma tan acentuada en pruebas anteriores ya que en esos casos el buffer es mayor, creando un margen y suavizando el throughput.

Parámetros del escenario:

- Ancho de banda: 100 Mbps
- Delay: 200 ms
- Pérdidas: 0%
- Tamaño de buffer: 200 Kbytes
- Duración de prueba: 60s

TCP Reno

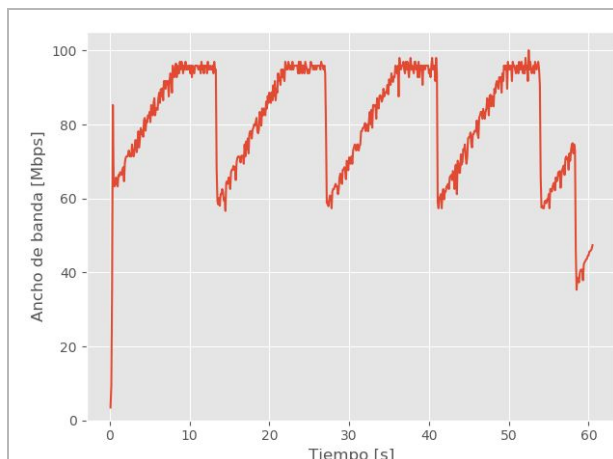


Figura 31.1: Throughput

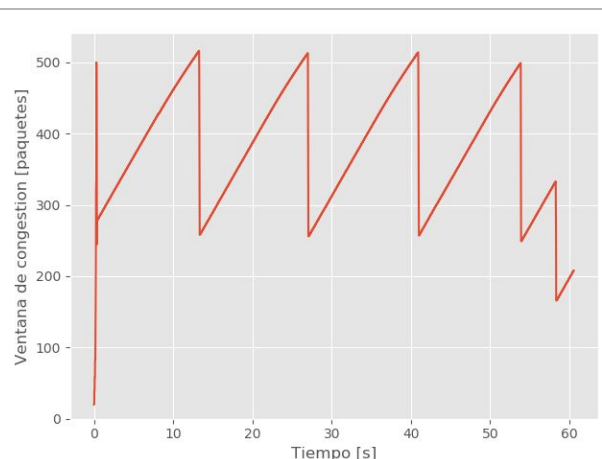
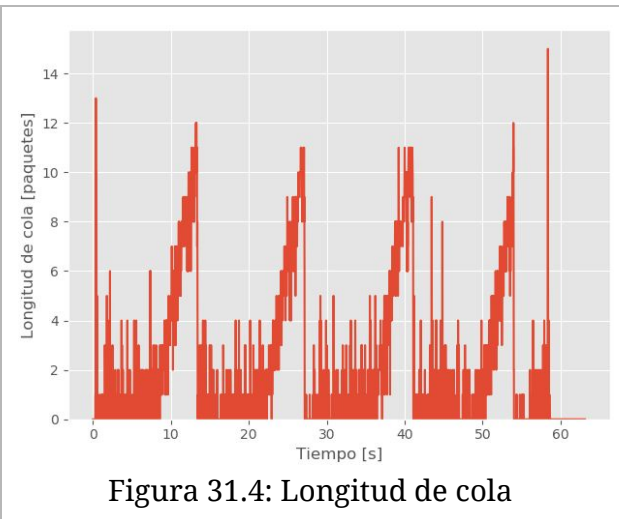
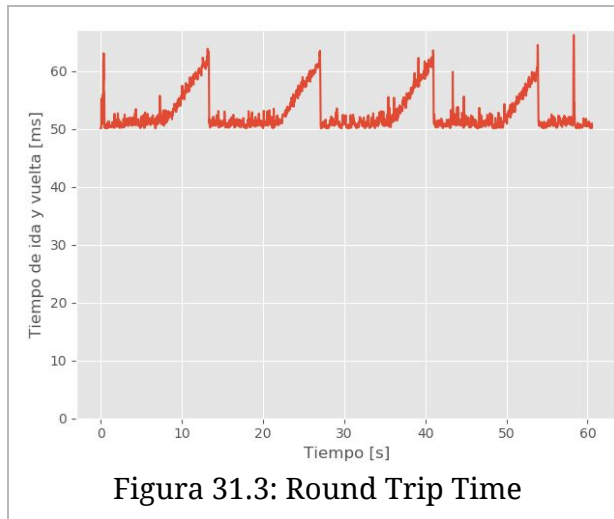
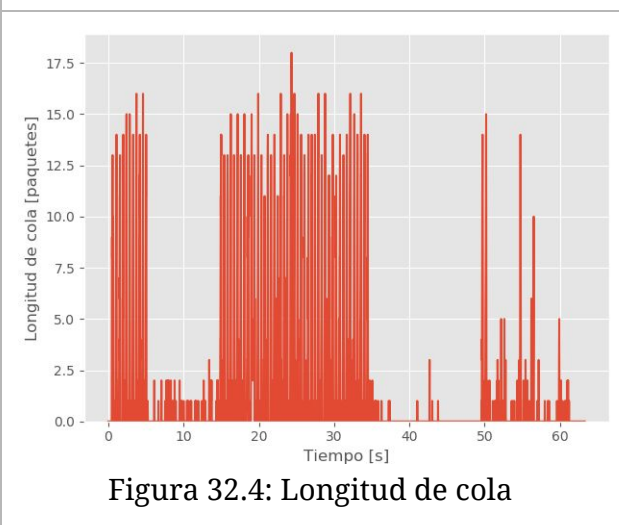
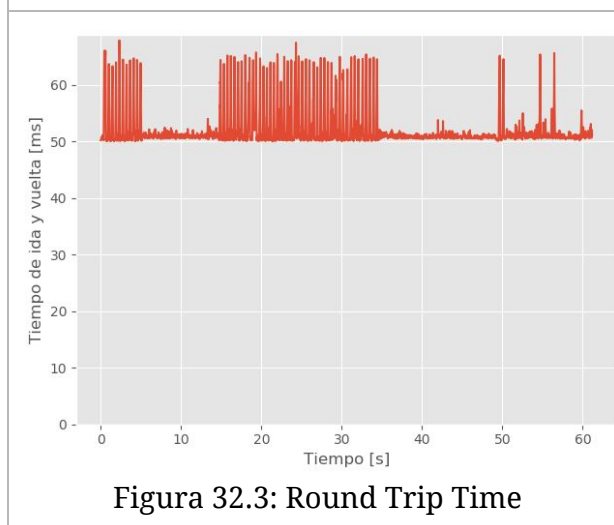
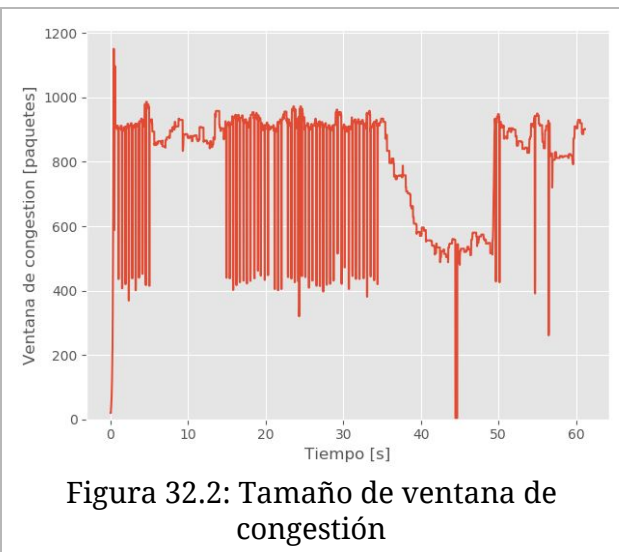
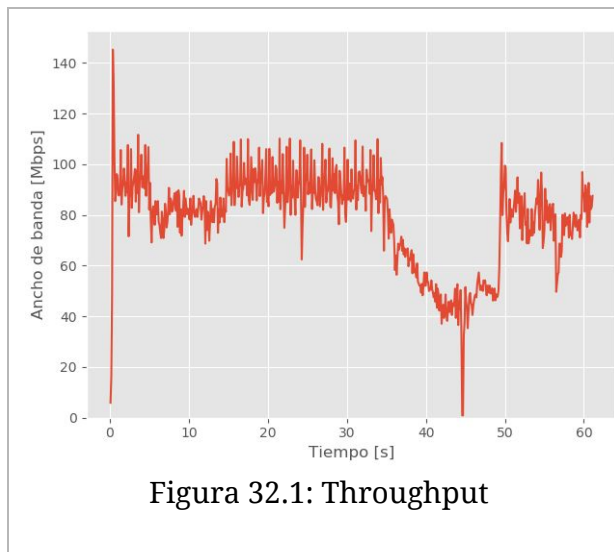


Figura 31.2: Tamaño de ventana de congestión



TCP BBRv1



TCP BBRv2

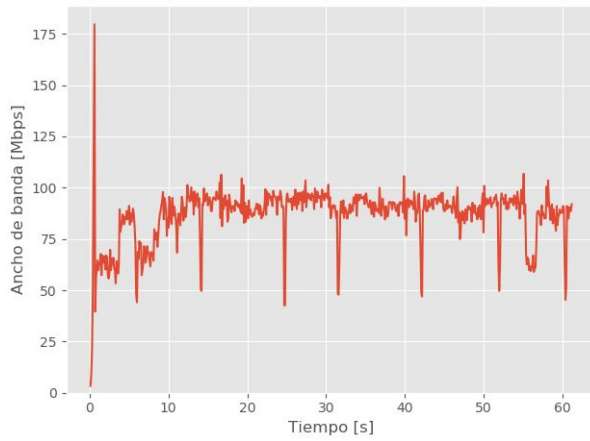


Figura 33.1: Throughput

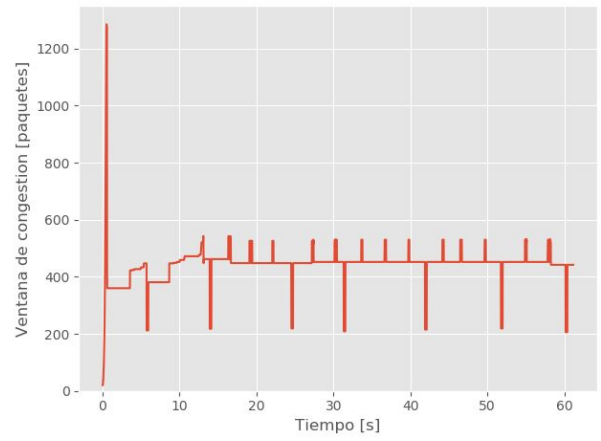


Figura 33.2: Tamaño de ventana de congestión

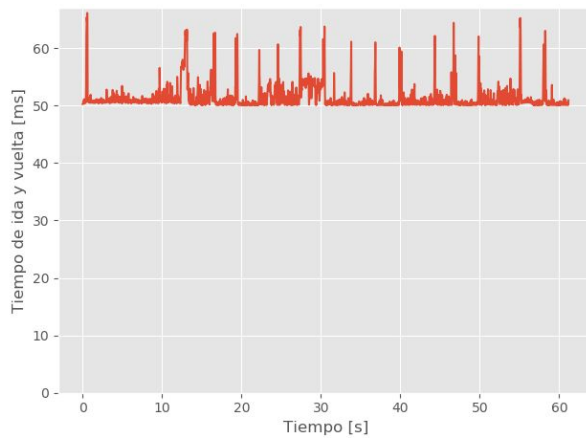


Figura 33.3: Round Trip Time

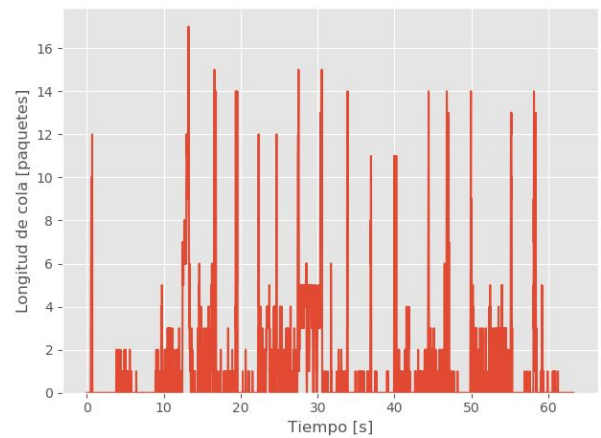


Figura 33.4: Longitud de cola

Conclusión

Uno de los objetivos del trabajo fue lograr la configuración de un entorno de trabajo con Mininet que permita realizar pruebas fácilmente. La configuración inicial toma mucho trabajo, pero esperamos que el código desarrollado y la máquina virtual configurada sirva para trabajos posteriores.

A partir de los escenarios realizados, se puede decir que los resultados fueron los esperados. En escenarios donde las pérdidas son producto de llenado de buffers, Reno aumenta su velocidad de transmisión hasta que se producen pérdidas, para luego bajar el tamaño de la ventana a la mitad. Cubic sigue la forma de una forma de una cúbica y se puede observar que al producirse una pérdida establece el umbral en W_{max} , pero si la siguiente pérdida se produce antes de alcanzar ese valor, se disminuye el umbral a un valor menor.

BBRv1 y BBRv2 logran en general, estimar el throughput máximo del canal sin necesidad de llevar el enlace a la saturación y perder paquetes. Ambos logran aprovechar el canal sin necesidad de aumentar la latencia, lo que es muy importante si en el canal coexisten flujos de voz o aplicaciones en tiempo real.

BBRv2 contiene diversas mejoras con respecto a BBRv1, en este trabajo se logró observar el manejo diferente de las ventanas de congestión para encontrar el límite, las mejoras en redes con buffers pequeños y la respuesta ante pérdidas. Como trabajo futuro se puede activar el uso de ECN y comprobar que el marcado de paquetes es realizado por el buffer en el cuello de botella. Alternativamente, se puede observar la distribución de ancho de banda entre múltiples flujos TCP simultáneos.

Referencias

RFC 5681 - TCP Congestion Control: <https://tools.ietf.org/html/rfc5681>

RFC 8312 - CUBIC for Fast Long-Distance Networks: <https://tools.ietf.org/html/rfc8312>

Why does CUBIC take us back to TCP congestion control?:

<https://pandorafms.com/blog/tcp-congestion-control/>

BBR Development Forum: <https://groups.google.com/forum/#!forum/bbr-dev>

IETF 100 Update: BBR in shallow buffers:

<https://datatracker.ietf.org/meeting/100/materials/slides-100-iccr-a-quick-bbr-update-bbr-in-shallow-buffers>

BBRv2 - A Model-based Congestion Control:

<https://datatracker.ietf.org/meeting/104/materials/slides-104-iccr-a-update-on-bbr-00>

BBRv2: A Model-based Congestion Control - Performance Optimizations:

<https://datatracker.ietf.org/meeting/106/materials/slides-106-iccr-a-update-on-bbrv2>

RFC 3168 - The Addition of Explicit Congestion Notification (ECN) to IP:

<https://tools.ietf.org/html/rfc3168>

tcp_bbr: adapt cwnd based on ack aggregation estimation:

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/net/ipv4/tcp_bbr.c?id=78dc70ebaa38aa303274e333be6c98eef87619e2

TCP small queues and WiFi aggregation: <https://lwn.net/Articles/757643/>