SIMULACIÓN EN NS3 DEL PROBLEMA DENOMINADO CUELLO DE BOTELLA COMPARTIDO QUE SE PRESENTA EN EL PROTOCOLO MP-TCP

SIMULATION IN NS3 OF THE PROBLEM SO CALLED SHARED NECK OF BOTTLE THAT APPEARS IN THE PROTOCOL MP-TCP

Christian Alexander Valdivieso Pinzón, Carlos David Cajas Guijarro, Raúl David Mejía Navarrete, Iván Marcelo Bernal Carrillo
Departamento de Electrónica, Telecomunicaciones y Redes de Información, Escuela Politécnica Nacional, Ecuador
E-mail: [christian.valdivieso, carlos.cajas, david.mejia, ivan.bernal]@epn.edu.ec

(Enviado Mayo 14, 2017; Aceptado Mayo 29, 2017)

Resumen
MP-TCP (*MultiPath-TCP*) es un protocolo que permite el envío de datos por múltiples caminos en dispositivos que poseen varias interfaces de red. Una conexión MP-TCP se divide en varias conexiones TCP denominadas subflujos, de esta manera se puede incrementarla tasa efectiva de la conexión y la resistencia a fallos. Sin embargo, en un escenario en el que se emplee MP-TCP con un cliente y un servidor, cada uno con una sola interfaz de red, se puede tener que los subflujos creados sigan el mismo camino a pesar que existan varios disponibles; esto genera el problema denominado cuello de botella compartido (*shared bottleneck)*. El artículo describe el desarrollo del código en C++ para realizar la simulación del problema del *shared bottleneck* en el simulador de redes NS3. La simulación está desarrollada con la infraestructura DCE (*Direct Code Execution*) para NS3 de tal manera que se puede utilizar una implementación existente de MP-TCP instalada en un *kernel* de Linux. Con la implementación desarrollada se realizan pruebas y se presentan y discuten los resultados obtenidos, lo que permite complementar el estudio y análisis de este problema del que adolece MP-TCP. Entre los resultados se resalta la estimación de la tasa efectiva de transferencia que se obtiene variando el número de subflujos y caminos en escenarios en los que se presenta el problema del *shared bottleneck*.

Palabras clave: *MP-TCP, Shared Bottleneck, NS3-DCE, Ndiffports, Throughput.*

Abstract
MP-TCP (MultiPath-TCP) is a protocol that allows the sending of data through multiple paths in devices that have several network interfaces. An MP-TCP connection is divided into several TCP connections called subflows, in this way the effective connection rate and the fault resistance can be increased. However, in a scenario where MP-TCP is used with a client and a server, each with a single network interface, you can have the created subflows follow the same path even if several are available; this generates the problem called shared bottleneck. The article describes the development of the C ++ code to simulate the shared bottleneck problem in the NS3 network simulator. The simulation is developed with the DCE (Direct Code Execution) infrastructure for NS3 in such a way that an existing implementation of MP-TCP installed in a Linux kernel can be used. With the developed implementation, tests are performed and the results obtained are presented and discussed, which makes it possible to complement the study and analysis of this problem that MP-TCP suffers from. Among the results, the estimation of the effective transfer rate obtained by varying the number of subflows and roads in scenarios in which the shared bottleneck problem occurs is highlighted.

Keywords: *MP-TCP, Shared Bottleneck, NS3-DCE, Ndiffports, Throughput.*

1 INTRODUCCIÓN

Hoy en día, dispositivos tales como computadoras portátiles, teléfonos inteligentes y servidores, poseen varias interfaces de red (*multihomed*). Estos dispositivos, por lo general, han venido utilizando en estas últimas décadas la arquitectura tradicional de TCP/IP (*Transmission Control Protocol/Internet Protocol*). A pesar que esta arquitectura es funcional y aprovecha la diversidad de caminos o rutas en una red a nivel de la capa Internet, esta presenta ciertas limitaciones a nivel de capa transporte como [1]:

- Establecer un solo camino por conexión TCP, limitando así el *throughput* de dicha conexión, debido a que no se aprovecha el conjunto de interfaces disponibles en algunos hosts, ya que las conexiones TCP solo se establecen con un par de puertos origen-destino asociados a un par de direcciones IP.

- En la arquitectura TCP/IP, para diferenciar las distintas conexiones TCP, el receptor demultiplexa los paquetes de acuerdo a la denominada tupla de 4 elementos, los cuales son: direcciones IP origen y destino, y número de puertos origen y destino; esto hace que las conexiones estén relacionadas con los elementos de la tupla, por tanto, las conexiones TCP no pueden moverse de una red a otra sin que se reinicie la conexión TCP y se cambie la configuración de las direcciones IP y los puertos TCP.

De este modo, si se desea mejorar la resistencia a fallas y aumentar la tasa efectiva de transferencia (*throughput*) o recepción de datos, según corresponda, es necesario buscar nuevas formas de usar múltiples caminos en la red para una misma conexión TCP empleando una o varias de las interfaces que estén disponibles en un host [2].

El protocolo MP-TCP es una alternativa que permite usar varios caminos en la red. MP-TCP hace uso de varias conexiones TCP, denominadas subflujos que en conjunto forman una conexión MP-TCP [3]. El protocolo MP-TCP permite un aumento de *throughput* y una resistencia a fallas en dispositivos *multihomed*, en condiciones en las que cada subflujo pueda asociarse a una interfaz de red diferente.

Sin embargo, hay ocasiones cuando estos subflujos tienden a seguir un único camino (varios subflujos asociados a una misma interfaz). A este problema se lo denomina *shared bottleneck* o cuello de botella compartido [3]. Por tanto, para realizar un adecuado análisis de este problema que se presenta en MP-TCP, en este trabajo se plantea realizar la simulación que permita establecer escenarios en los que se reproduzca el problema del *shared bottleneck*, desarrollando el código en C++para el simulador de redes NS3[4] y empleando la implementación de MP-TCP que se dispone a nivel del *kernel* de Linux. Además, se realizan las simulaciones empleando varias topologías de red y se observa el efecto que produce en el performance de la red el aumentar el número de subflujos y que estos subflujos sigan una misma trayectoria.

Las demás secciones de este artículo están organizadas de la siguiente manera: la Sección 2 presenta aspectos del protocolo MP-TCP considerados necesarios para el desarrollo del trabajo. Se discute el administrador de caminos y el problema de cuello de botella compartido. La Sección 3 presenta aspectos básicos del simulador NS3,así como las abstracciones empleadas en NS3 y las bibliotecas que se usarán para la simulación del problema a resolver. La Sección 4 presenta las ideas principales que se usaron para el desarrollo del código C++ para la simulación en NS3 del problema de cuello de botella compartido. La Sección 5 describe los resultados obtenidos de la simulación, empleando el código escrito, en particular las medidas de *throughput* que se obtienen en una conexión MP-TCP. El artículo concluye con la Sección 6 en la que se presentan las conclusiones desprendidas del trabajo realizado.

2 MP-TCP

Con MP-TCP se pueden aprovechar los casos en los que un host tenga a disposición varias interfaces para conectarse a una red (interfaz *WiFi*, interfaces LAN, interfaz para acceso a la red móvil, etc.). Al crear una conexión MP-TCP con estas interfaces, considerando que ambos extremos (cliente y servidor) soportan el protocolo MP-TCP, se crean varios subflujos, los cuales pueden usar caminos disjuntos.

Cuando se hace mención al término "camino" en este artículo, se hace referencia al camino que los paquetes de una conexión tienden a seguir; esta aclaración es debido a que cada paquete podría seguir un camino diferente. Los caminos disjuntos se definen como las trayectorias que siguen los paquetes de datos, que no comparten ningún nodo en común, siendo un nodo un *router* o un dispositivo de red en general. Para evitar ambigüedades en este artículo, a partir de este punto se usará el término "camino" para definir una trayectoria por la cual circulan paquetes de datos de una conexión o subflujo que tiene al menos un nodo en común, por tanto, diferenciando este término del de "caminos disjuntos". La Fig. 1 presenta un conjunto de caminos que no son disjuntos porque, por ejemplo, el nodo al cual está conectado el host cliente es común a las tres trayectorias que se visualizan claramente en la figura.

Cada subflujo, con una diferente tupla de elementos, crea una conexión TCP, y, en conjunto, estos subflujos forman una conexión MP-TCP. MP-TCP se encargará de enviar los datos a través de los subflujos asociados a los caminos sean o no disjuntos menos congestionados [5]; de esta manera se maximiza el uso de los recursos que dispone un host, y, en ciertos escenarios, se obtiene una mejor tasa efectiva y se incrementa la resistencia a fallas de la conexión establecida.

2.1 Administrador de Caminos

La implementación más conocida, activa y estable de MP-TCP ha sido desarrollada por el grupo de trabajo de MP-TCP de la Universidad Católica de Lovaina [6], sobre el sistema operativo Linux. Esta es la versión que se ajusta de mejor manera al RFC-6824 y es la recomendable para realizar ensayos y análisis del protocolo en un ambiente real [6]. Esta implementación permite la configuración del administrador de caminos, lo cual permite configurar cómo se van a crear los subflujos en función de las interfaces de red disponibles.

Las opciones que se pueden configurar en el administrador de caminos son:

- *Fullmesh:* Se crean subflujos considerando todas las posibles combinaciones de direcciones IP asociadas a dos *hosts*.

- *Ndiffports*: Se crean subflujos por cada par de direcciones IP origen y destino, modificando el puerto origen. Esta configuración será de utilidad para la simulación del *shared bottleneck* que se discute en este trabajo.

2.2 Problema de Cuello de Botella Compartido

Cabe mencionar que para establecer varios subflujos en una conexión MP-TCP, no es necesario el uso de varias interfaces de red [3], se lo puede hacer con una sola en cada host que participa en la conexión; para esto se debe modificar la configuración de MP-TCP; en concreto, cambiando la configuración del administrador de caminos de *fullmesh* (por defecto) a *ndiffports* y el puerto de origen, con esto se pueden establecer varios subflujos con el mismo par de direcciones IP [4]. Este escenario implica que no existen caminos disjuntos en la conexión MP-TCP y los subflujos creados poseen el mismo par de direcciones IP origen y destino, van a compartir los nodos y enlaces que conforman el único camino existente, los datos se distribuyen entre los subflujos, pero no se incrementa el *throughput*. A este problema se le conoce como el cuello de botella compartido (*shared bottleneck*). Este problema causa una sobre utilización del ancho de banda en los enlaces donde circulan todos los subflujos de la misma conexión MP-TCP [3].

Un escenario en el cual se produce el problema del *shared bottleneck* se observa en la Fig. 1, en la cual se visualiza que, aunque existen tres trayectorias posibles del cliente hacia el servidor, todos los subflujos asociados a la conexión en MP-TCP tienden a seguir el mismo camino.

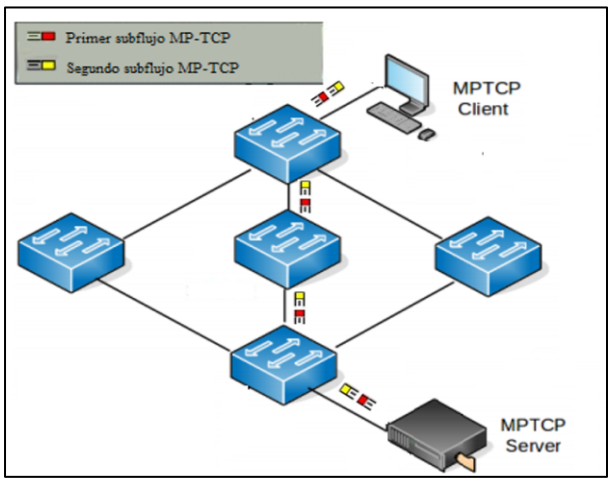


Figura 1 Problema del cuello de botella compartido.

3 NS-3

Direct Code Execution (DCE) es una infraestructura para NS3 que provee facilidades para ejecutar, dentro de NS3, implementaciones existentes de protocolos de red sea en el espacio de usuario o del *kernel* del sistema operativo, incluso puede ejecutarse aplicaciones; en ambos casos sin la necesidad de realizar cambios en código fuente. Por ejemplo, en lugar de usar una implementación de una aplicación similar a ping, se puede usar la aplicación real de ping. Además, se puede utilizar el *stack* de protocolos de Linux en las simulaciones.

Debido a que no existe un módulo oficial de MP-TCP desarrollado sobre NS3, se usará el módulo DCE sobre la versión 3.14 de NS3 [7]. Al cargar un sistema operativo Linux con la versión MP-TCP de la Universidad Católica de Lovaina dentro del módulo DCE, se podrá simular con NS3 un entorno cliente-servidor con MP-TCP habilitado en lugar de TCP.

3.1 Abstracciones empleadas en NS3

Para trabajar en NS3 se consideran algunas abstracciones [4] que usa el simulador como:

1) Nodos: Un nodo en NS3 es una computadora o *host* a la cual se le pueden añadir ciertas funcionalidades como el *stack* de protocolos, aplicaciones o interfaces de red. Esta abstracción se emplea mediante la clase `Node`. `Node` provee funciones para el manejo de los *hosts* en la simulación. Otra clase relacionada es `NodeContainer`, la cual provee la abstracción de un arreglo de nodos.

2) Aplicación: En NS3, una aplicación es un programa de software que corre sobre los nodos para realizar ciertas tareas a nivel de usuario. Esta abstracción es implementada por la clase `Application`, la cual provee funciones para el manejo de aplicaciones a nivel de usuario.

3) Canal: Un canal en NS3 permite la conexión entre nodos. Es una representación del medio físico por el cual fluyen los datos entre dos nodos. La clase que realiza esta abstracción es `Channel`, la cual provee funciones para el manejo de la conexión entre los nodos. Una clase específica derivada de`Channel` es `PointToPointChannel`.

4) Dispositivos de Red: Para NS3, un dispositivo de red cubre el software y el hardware de una tarjeta de red. Cuando un dispositivo de red se asocia a un nodo, el nodo está en la capacidad de comunicarse con otros mediante un canal. A la clase que maneja esta abstracción se la denomina `NetDevice`, la cual maneja las funciones para comunicar los nodos a través de un canal. En NS3 se debe tener una asociación correcta entre un tipo de canal y un tipo de dispositivo de red, pues una clase específica como `PointToPointNetDevice` solo podrá asociarse a un canal de la clase `PointToPointChannel`. La clase `NetDeviceContainer` permite crear un arreglo de dispositivos de red.

5) Topology Helper: Esta abstracción en NS3 facilita la asociación entre los distintos nodos, dispositivos de red y canales. Clases específicas son, por ejemplo: `DceManagerHelper`, `InternetStackHelper`.

3.2 Bibliotecas utilizadas

1) Core-module: [8] Biblioteca núcleo de NS3que permite el uso de clases para la ejecución de las simulaciones. Las clases y funciones del *core* son comunes para todos los protocolos y nodos de red usados en los modelos. Las clases del módulo *core* permiten: manejo de variables aleatorias como número de secuencia inicial, manejo de los eventos en el tiempo de una simulación, uso de punteros, entre otros. Las clases y funciones empleadas de esta biblioteca son:

- `CommandLine`: Clase que permite modificar valores por defecto mediante la línea de comandos, durante el inicio de la ejecución de la simulación.
- `GlobalValue::Bind("variable_a_cambiar", "estado_lógico_de_la_variable")`: Permite modificar el valor por defecto de una variable global, en este caso se usará para calcular los *checksums* de los protocolos TCP e IP.
- `Simulator::Run()`: Función global que ejecuta la simulación en NS3.
- `Simulator::Stop("tiempo_en_segundos")`: Función global que detiene la simulación luego que han transcurrido el número de segundos que se indica en el argumento de la función.
- `Simulator::Destroy()`: Función global que permite liberar los recursos comprometidos con la simulación como memoria, objetos creados, entre otros.

2) Internet-module [9]*:* Biblioteca que provee clases para que los nodos posean el stackTCP/IPv4. Las clases usadas son:

- `InternetStackHelper`: Agrega el stackTCP/IP sobre los nodos ya existentes.
- `Ipv4AddressHelper`: Clase que permite establecer el direccionamiento IPv4 sobre un enlace entre dos nodos.
- `Ipv4InterfaceContainer`: Clase que permite tener un contenedor entre las interfaces de los nodos conectados y sus respectivas direcciones IPv4.
- `Ipv4GlobalRoutingHelper::PopulateRoutingTables`: Esta función construye una base de datos de enrutamiento e inicializa las tablas de enrutamiento estática de todos los nodos en la simulación; esto se realiza cuando se invoca la función, una vez que se construye la topología y se realiza la asignación de direcciones IP.

3) Dce-module [10]**:** Biblioteca que posee clases de NS3 que permiten utilizar los protocolos de red disponibles en el sistema operativo subyacente, sea en el espacio de *kernel* o en el espacio de usuario, como, por ejemplo, el *stack* TCP/IP de una versión de Linux. Entre las clases que se usarán de esta biblioteca están:

- `DceManagerHelper`: Clase para instalar sobre los nodos, las funcionalidades de DCE. Mediante la función`SetNetworkStack` se puede seleccionar el sistema operativo a instalar en los nodos. En este caso permitirá instalar en los nodos cliente y servidor un sistema operativo Linux con MP-TCP.
- `DceApplicationHelper`: Esta clase define qué aplicación DCE se ejecutará dentro de los nodos, especificando la aplicación a ejecutar y los parámetros de esta. Para este trabajo, con esta clase se ejecuta la herramienta *iperf* en los nodos cliente y servidor.
- `LinuxStackHelper`: Clase que permite configurar algunos parámetros del *kernel* de Linux. Cabe mencionar que en las simulaciones con NS3 será necesario configurar el administrador de caminos de MP-TCP para administrar la creación de los subflujos.

4) Point-to-point module*:* Biblioteca que implementa un enlace entre dos nodos con el protocolo Point-to-Point (PPP). Esta biblioteca dispone de clases que pueden definir características del enlace como ancho de banda, latencia, entre otros. La clase a usarse es `PointToPointHelper`, la cual ayuda a crear un enlace PPP entre dos nodos.

5) Netanim-module: Biblioteca que permite llamar a la función `AnimationInterfaceanim("nombre.xml")`, para la creación de un archivo de formatoXML*,* el cual, usando NetAnim, permite visualizar los nodos en la red y el intercambio de flujos de paquetes entre ellos.

6) Constant-position-mobility-model: Biblioteca que dispone de clases para modelos de movilidad entre nodos. La clase a usar es `ConstantPositionMobilityModel`, la cual define un modelo de movilidad que hace que la posición actual de un nodo no cambie, hasta que de forma explícita sea cambiada. Esto es para uso con NetAnim.

4 DESARROLLO DEL CÓDIGO PARA LA SIMULACIÓN

Como punto de partida para desarrollar el código para la simulación del cuello de botella compartido en NS3, se empleó [7] que aborda aspectos sobre DCE. Se inició el desarrollo creando el archivo *sharedbottleneck.cc*.

Al inicio del archivo sharedbottleneck.cc se declaran las bibliotecas utilizadas; en este caso: core-module, internet-module, dce-module, point-to-point-module, netanim-module y constant-position-mobility-model.

Luego, se procede a la declaración de las variables que se van a utilizar, son ejemplos: el número de *routers* que conectan a los hosts cliente y servidor, duración de la ejecución de la aplicación *iperf* (16 segundos), tamaño de la ventana de TCP (256 KBytes). El valor de `numero_routers` serán los *routers* que crearán diferentes caminos y este valor podrá ser modificado al

momento de ejecutar la simulación adicionando `-numero=x`, donde x indica el número de routers.

Con ayuda de la clase `CommandLine` se pueden obtener del usuario valores mediante la línea de comandos al ejecutar la simulación. En nuestro caso se podrá configurar el número de routers al añadir `-numero=x`. A continuación, el código procede a realizar un chequeo, si `numero_routers`es cero o negativo, la simulación no se iniciará y en el terminal donde se haya ejecutado la simulación despliega un mensaje de error.

Se debe activar obligatoriamente el cálculo de*checksums* de los protocolos IP o TCP para las simulaciones a realizarse. Este cálculo está deshabilitado por defecto y se lo pueda cambiar con la función:

```
GlobalValue::Bind(
   "ChecksumEnabled",
   BooleanValue(true))
```

El siguiente paso es la creación de los nodos cliente, servidor y los *routers*, empleando la función CreateObject<`Node`>()y `NodeContainer.Create()`. Para todos los routers, es necesario la instalación del *stack* TCP/IP, lo que se realiza con la ayuda de la clase `InternetStackHelper`.

Ahora se realiza la instalación del *stack* `dceManager` y `linuxstack`, tanto en el cliente, así como en el servidor. Con `dceManager` se instala un *kernel* de Linux y con `linuxstack` se puede configurar parámetros del *kernel* como los del administrador de caminos de MP-TCP.

Para compilar la biblioteca: liblinux.so, la cual permitirá manejar el *stack* de protocolos y el *kernel* de Linux con MP-TCP se emplea la siguiente función:

```
dceManager.SetNetworkStack(
   "ns3::LinuxSocketFdFactory",
   "Library”,
   StringValue("liblinux.so"))
```

Posteriormente, se procede a la creación de los enlaces con ayuda de la clase `PointToPointHelper`. En cada enlace se establece un ancho de banda (1Gbps ó 10Mbps) y un retardo de 1µs. Los enlaces de 10Mbps serán para las conexiones entre los *routers*, mientras que los de 1Gbps son para los enlaces entre los hosts cliente y servidor con sus respectivos *routers*.

En este punto del código corresponde realizar la asignación de direcciones IP. Entre el cliente y el *router*, la dirección de red que usan estos dos nodos es la 10.0.0.0/24. Esto se consigue con las funciones `setBase()` y `setAssign()`de la clase `Ipv4AddressHelper`.De igual manera se realiza este procedimiento de asignación entre el servidor y su *router*, para estos dos nodos se usó la dirección de red 10.1.0.0/24.

Se realiza un barrido de todos los *routers*, de acuerdo al número que haya ingresado el usuario, para poder realizar la asignación de direcciones IP y la instalación de los enlaces entre los *routers*, de acuerdo a la topología que se visualiza en la Fig. 2. Para llenar las tablas de enrutamiento, se emplea la siguiente línea de código:

```
Ipv4GlobalRoutingHelper::PopulateRoutin
gTables().
```

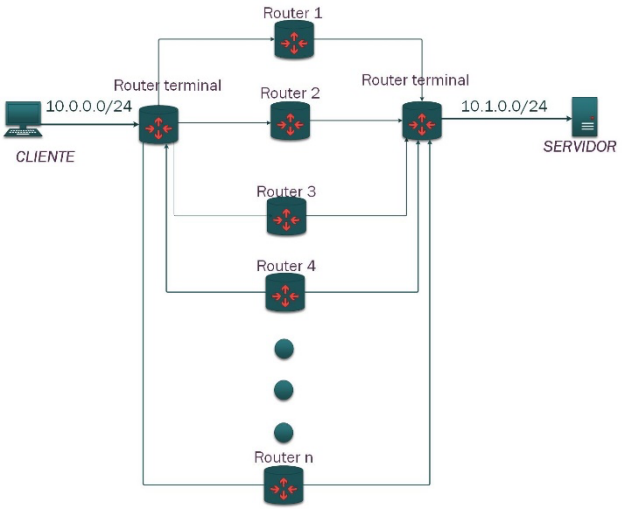


Figura 2 Topologías a implementarse.

Luego se procede activar al protocolo MP-TCP. Esto requiere la siguiente línea de código:

```
linuxStack.SysctlSet(
   client,
   ".net.mptcp.mptcp_enabled",
   "1")
```

La función SysctlSet permite el uso del comando *sysctl* de Linux con el cual se puede configurar parámetros del *kernel*, en este caso la activación o la desactivación de MP-TCP (1 activación y 0 para desactivar).

Para configurar el administrador de caminos de MP-TCP con la opción *ndiffports* se utiliza:

```
linuxStack.SysctlSet(
   client,
   ".net.mptcp.mptcp_path_manager",
   "ndiffports")
```

Para establecer el número de subflujos que se van a crear con *ndiffports*, es necesario modificar el valor del parámetro `num_subflows_read_mostl`, ubicado en el archivo `mptcp_ndiffports.c`. Es pertinente mencionar que la implementación de Linux para NS3 empleada soporta solo hasta 30 subflujos.

Luego es necesaria la ejecución de la aplicación *iperf*, lo que se consigue con ayuda de un objeto `dce` de la clase `DceApplicationHelper`. Con *iperf*, el cliente se conecta al servidor en el puerto 5001, por defecto. El comando a adicionarse en el objeto `dce` del host cliente es:

```
iperf -c 10.1.0.1 -i 2 --time 16 -w 256k
```

La opción -c indica, el modo de operación cliente de *iperf*, 10.0.0.1es la dirección IP que corresponde a la dirección con la cual se va a conectar, en este caso la dirección IP del servidor; -i indica el intervalo de generación de los reportes (cada dos segundos); --time 16 indica que el tiempo de ejecución de la aplicación es de 16 segundos; -w 256k indica que el tamaño de la ventana TCPes de 256 KBytes.

Para instalar el objeto `dce` en el nodo cliente, se debe crear un objeto de la clase `ApplicationContainer` y emplear la función `Install()`. Con las funciones `Start()` y `Stop()` del objeto aplicación se establece el intervalo de tiempo en el que se ejecutará dicho objeto `dce`.

Luego, de igual manera se realiza la ejecución de *iperf* en el servidor. El comando añadido en el objeto `dce_servidor` es: `iperf -s -P 1 -w 256k`, donde: -s indica el modo de operación servidor; -P 1 indica que solo se hará una prueba (-P 2 permite ejecutar dos pruebas en paralelo); y, -w 256k indica el tamaño de ventana en el servidor. De igual manera que en el caso del cliente, se requiere de un objeto `ApplicationContainer`, para la instalación del objeto `dce_servidor` en el nodo servidor.

A continuación, se escribe el código para generar un archivo con extensión .cap, que captura todos los paquetes en el lado del cliente; esto permite saber si el protocolo MP-TCP se inicializó y si se crearon los subflujos que se configuraron. Además, la simulación crea un archivo de formato .xml, que permite visualizar la topología creada y el flujo de datos en NetAnim.

Para el posicionamiento de los nodos para la animación NetAnim, se escribió la función denominada *configurar_posicion(),*la cual adicionará las coordenadas de la posición y agregará el modelo `ConstantPositionMobilityModel` a todos los nodos que usen esta función.

Con `Simulator::Run()`se ejecuta la simulación. Con `Simulator::Stop(Seconds())` se procede a detener la simulación luego del tiempo indicado en segundos. Se escogió 18 segundos para que la aplicación *iperf* corra sin ser interrumpida.

Finalmente, con `Simulator::Destroy()`se procede a la destrucción de todos los objetos creados en la simulación luego que todas las tareas programadas hayan finalizado. Los procedimientos descritos se pueden resumir en el diagrama de flujo de la Fig. 3.

5 SIMULACIÓN Y RESULTADOS

En esta sección se presentan los resultados de las simulaciones obtenidas al ejecutar las pruebas y al realizar la medición del *throughput*. Se pretende que evidencie la disminución del *throughput* que ocasiona el problema del cuello de botella compartido, para ello, se implementan las topologías de red indicadas en la Fig. 2. Se simula el número de subflujos acorde al número de *routers*, por ejemplo, si se configura *ndiffports* con 4 subflujos, entonces se crean 4 *routers*. En caso de que el número de subflujos sea mayor al número de *routers*, estos igual seguirían el mismo camino.

Los enlaces de las redes 10.0.0.0/24 y 10.1.0.0/24 tendrán una velocidad de 1 Gbps, mientras que los demás enlaces una velocidad de 10 Mbps.

El hacer que el número de subflujos creados varíen con el número de *routers* creados, permite observar que, aunque se tenga la misma cantidad de posibles caminos que subflujos, estos seguirán uno solo de ellos. Aunque exista una gran cantidad de subflujos y *routers*, el *throughput* entre cliente y servidor no sobrepasará los 10 Mbps, que es el ancho de banda de los enlaces entre los *routers*, aunque en el enlace del cliente o el servidor con sus respectivos *routers* se dispone de 1 Gbps. Esto se debe a que todos los subflujos siguen el mismo camino, el cual estará limitado a 10 Mbps, y no se aprovechan los demás caminos.

Para la medición de *throughput* se realizaron pruebas con un número de subflujos en el rango de 1 a 30, utilizando una máquina física con 8GB de memoria RAM, con Linux, NS3, DCE y MP-TCP.

5.1 Resultados

Se presentan únicamente los resultados cuando se usaron 16 subflujos, ya que los procedimientos para las demás pruebas son similares. Sin embargo, se presenta una gráfica de la variación del *throughput* en función del número de subflujos, con todas las pruebas que se hicieron.

5.2 Ejecución de la simulación

El comando que se muestra continuación será necesario para ejecutar la simulación del*shared bottleneck*:

```
$./ waf –run “sharedbottleneck –numero=16”
```

5.3 Resultados con *iperf*

En el archivo sharedbottleneck.cc se incluye código que comanda la ejecución de la aplicación *iperf*, la cual permite obtener la tasa efectiva entre dos *hosts*. Luego de compilado el archivo sharedbottleneck.cc, se crea una carpeta denominada files-0, la cual almacena los resultados de la ejecución de *iperf*. Para poder observar los resultados se ejecuta el comando:

```
# cat files-0/var/log/*/stdout
```

Los resultados de la ejecución del comando se muestran en la Fig. 4. Como se puede observar, la tasa efectiva promedio entre el cliente y el servidor es de 8.20 Mbps en un intervalo de entre 0 y 16.1 segundos. Debido a que la topología tenía 16 *routers* y por tanto 16 posibles caminos se hubiera esperado que el *throughput* fuera de 160 Mbps ya que la interfaz que conecta los enlaces que conectan al cliente y servidor con sus respectivos *routers* iniciales poseen velocidades de un 1Gbps.

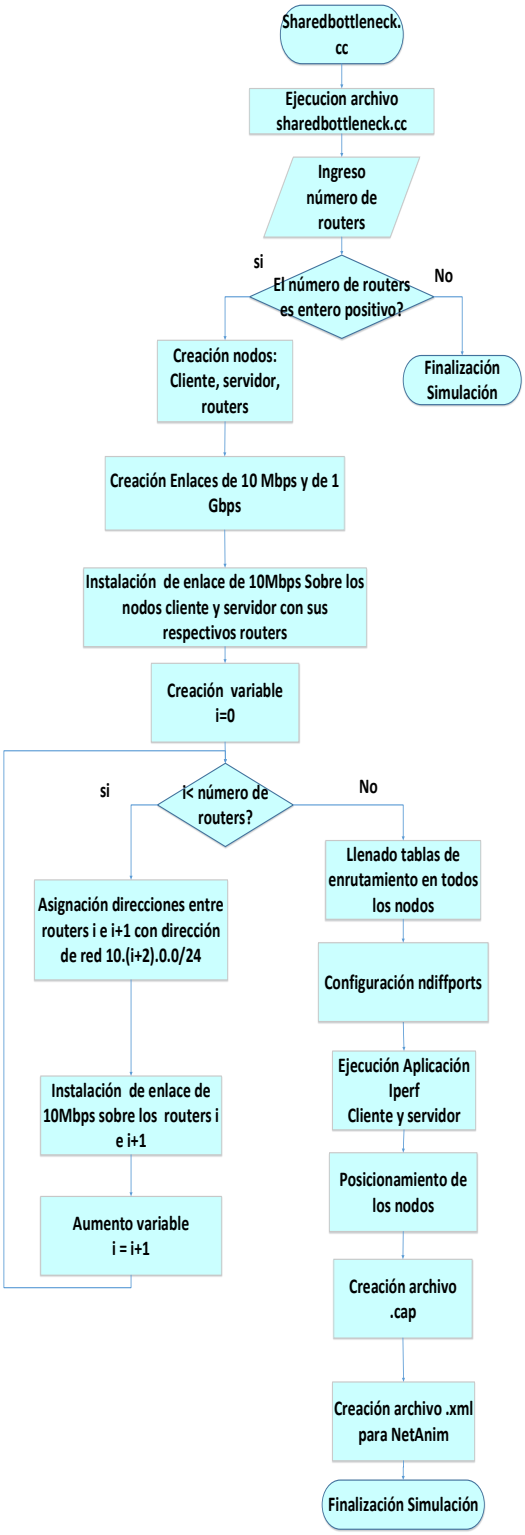


Figura 3 Tareas realizadas en sharedbottleneck.cc.

```
[  3] local 10.0.0.1 port 40289 connected with 10.1.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec  1.62 MBytes  6.82 Mbits/sec
[  3]  2.0- 4.0 sec  2.12 MBytes  8.91 Mbits/sec
[  3]  4.0- 6.0 sec  2.00 MBytes  8.39 Mbits/sec
[  3]  6.0- 8.0 sec  2.25 MBytes  9.44 Mbits/sec
[  3]  8.0-10.0 sec  2.12 MBytes  8.91 Mbits/sec
[  3] 10.0-12.0 sec  1.75 MBytes  7.34 Mbits/sec
[  3] 12.0-14.0 sec  2.00 MBytes  8.39 Mbits/sec
[  3] 14.0-16.0 sec  1.75 MBytes  7.34 Mbits/sec
[  3]  0.0-16.1 sec  15.8 MBytes  8.20 Mbits/sec
```

Figura 4 Captura del tráfico obtenido con *iperf*.

5.4 Resultados con NetAnim

Con NetAnim se puede observar la topología usando el archivo sharedbottlneck.xml como se observa en la Fig. 5. Existen dieciséis caminos para llegar desde el nodo 0 (cliente) al nodo 6 (servidor). De esos dieciséis posibles caminos, el flujo de datos (flechas azules) sigue un solo camino, aunque la conexión MP-TCP posea 16 subflujos.

La Fig. 6 muestra la captura de tráfico sobre la subred 10.0.0.0/24 (nodo cliente y el nodo 4), donde se puede visualizar que se trabaja con MP-TCP, en la columna de protocolo. Además, en la pestaña estadísticas en la opción "conversaciones", que se muestra en la Fig. 7, se observa el número de conexiones TCP establecidas. Se puede observar que son 16 conexiones TCP, lo cual era de esperar, ya que se configuró la opción *ndiffports* para 16 subflujos con un mismo par de direcciones IP.

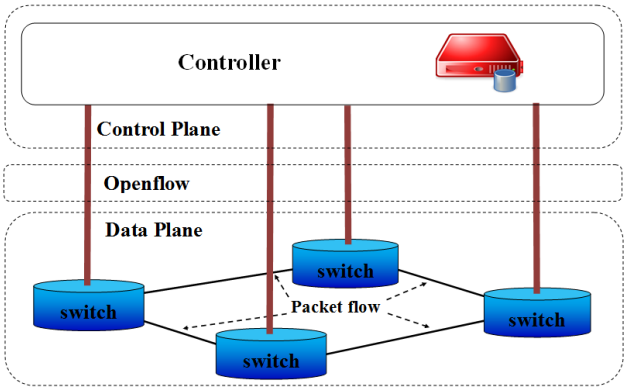


Figura 9 Arquitectura de las SDN.

Finalmente, en la Fig. 8, se presentan los resultados obtenidos al simular mediante la creación de distinto número de subflujos. Cabe mencionar que a medida que el número de subflujos aumenta, y considerando el tiempo de simulación, el *throughput* tiende a disminuir; esto se debe a que la conexión MP-TCP tiene varias conexiones de subflujos abiertas y estas toman un tiempo hasta que se establezcan. Como se observa en la Fig. 8, la disminución del *throughput*es proporcional al aumento del número de subflujos dado que en al tener 30 subflujos, el *throughput* es de 6.11 Mbps.

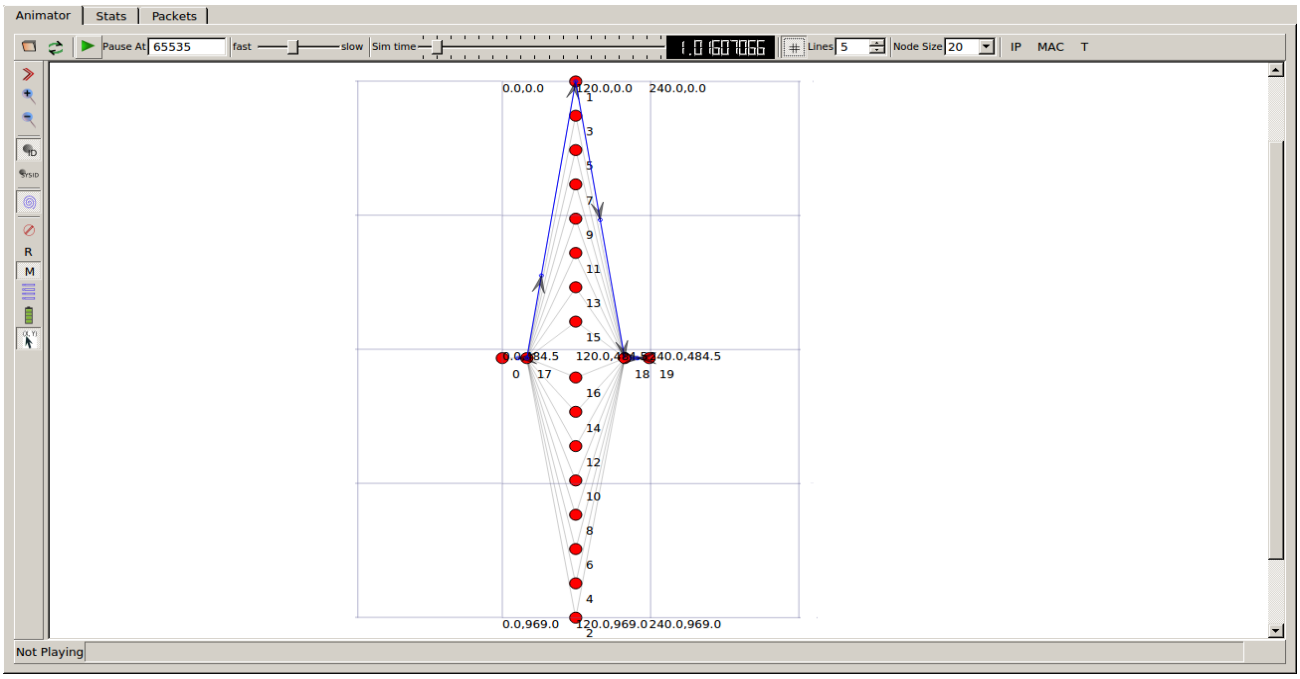


Figura5 Visualización de la topología de *shared bottleneck* en NetAnim.

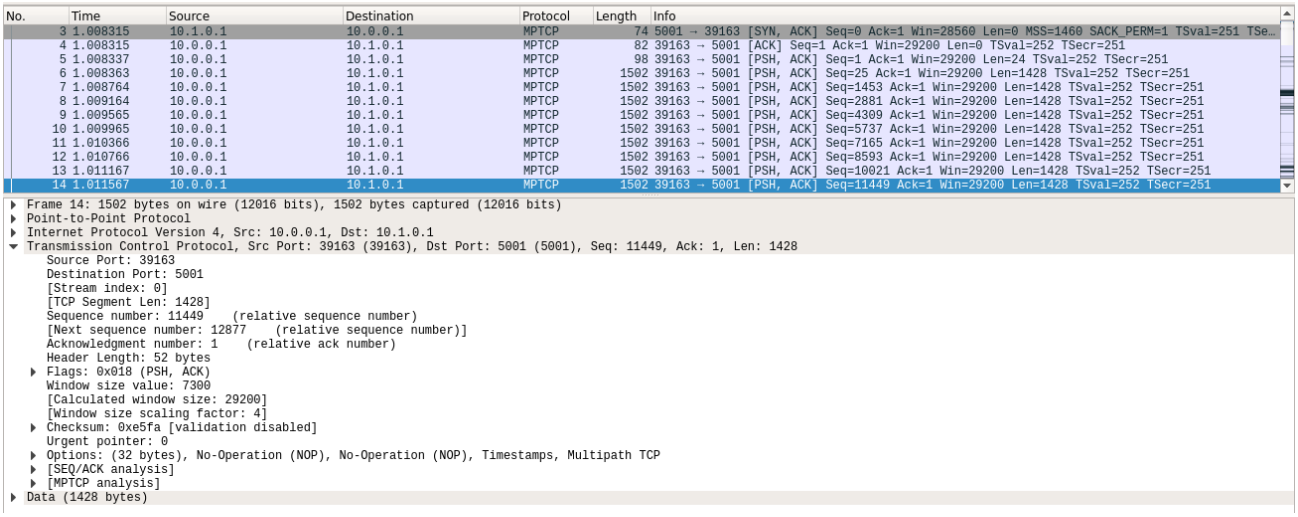
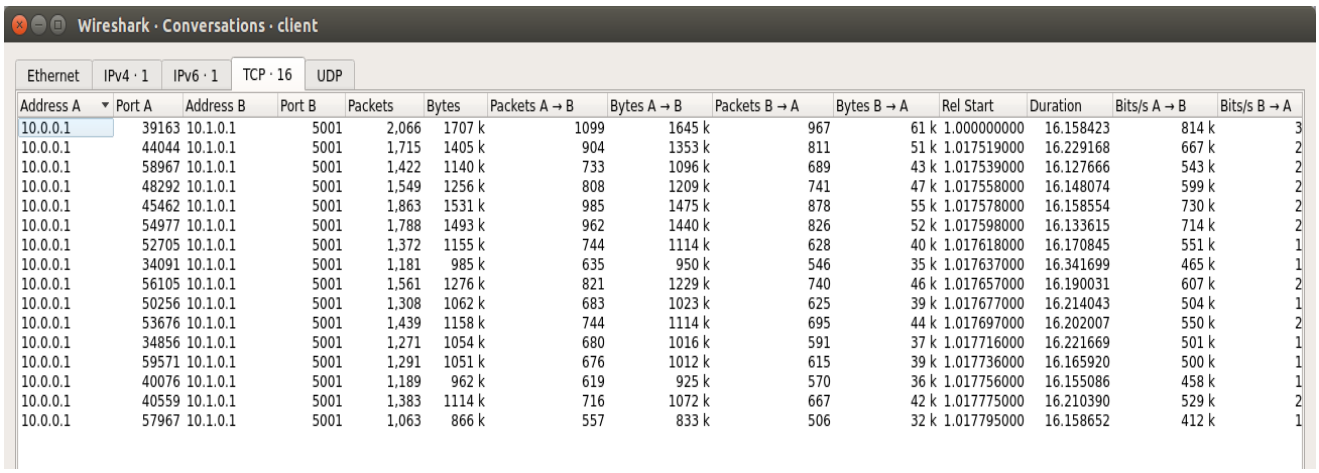


Figura 6 Visualización en *Wireshark* de los paquetes capturados en el lado del cliente.



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.0.0.1	39163	10.1.0.1	5001	2,066	1707 k	1099	1645 k	967	61 k	1.000000000	16.158423	814 k	3
10.0.0.1	44044	10.1.0.1	5001	1,715	1405 k	904	1353 k	811	51 k	1.017519000	16.229168	667 k	2
10.0.0.1	58967	10.1.0.1	5001	1,422	1140 k	733	1096 k	689	43 k	1.017539000	16.127666	543 k	2
10.0.0.1	48292	10.1.0.1	5001	1,549	1256 k	808	1209 k	741	47 k	1.017558000	16.148074	599 k	2
10.0.0.1	45462	10.1.0.1	5001	1,863	1531 k	985	1475 k	878	55 k	1.017578000	16.158554	730 k	2
10.0.0.1	54977	10.1.0.1	5001	1,788	1493 k	962	1440 k	826	52 k	1.017598000	16.133615	714 k	2
10.0.0.1	52705	10.1.0.1	5001	1,372	1155 k	744	1114 k	628	40 k	1.017618000	16.170845	551 k	1
10.0.0.1	34091	10.1.0.1	5001	1,181	985 k	635	950 k	546	35 k	1.017637000	16.341699	465 k	1
10.0.0.1	56105	10.1.0.1	5001	1,561	1276 k	821	1229 k	740	46 k	1.017657000	16.190031	607 k	2
10.0.0.1	50256	10.1.0.1	5001	1,308	1062 k	683	1023 k	625	39 k	1.017677000	16.214043	504 k	1
10.0.0.1	53676	10.1.0.1	5001	1,439	1158 k	744	1114 k	695	44 k	1.017697000	16.202007	550 k	2
10.0.0.1	34856	10.1.0.1	5001	1,271	1054 k	680	1016 k	591	37 k	1.017716000	16.221669	501 k	1
10.0.0.1	59571	10.1.0.1	5001	1,291	1051 k	676	1012 k	615	39 k	1.017736000	16.165920	500 k	1
10.0.0.1	40076	10.1.0.1	5001	1,189	962 k	619	925 k	570	36 k	1.017756000	16.155086	458 k	1
10.0.0.1	40559	10.1.0.1	5001	1,383	1114 k	716	1072 k	667	42 k	1.017775000	16.210390	529 k	2
10.0.0.1	57967	10.1.0.1	5001	1,063	866 k	557	833 k	506	32 k	1.017795000	16.158652	412 k	1

Figura 7 Visualización en *Wireshark* de los subflujos creados en la conexión MP-TCP del cliente.

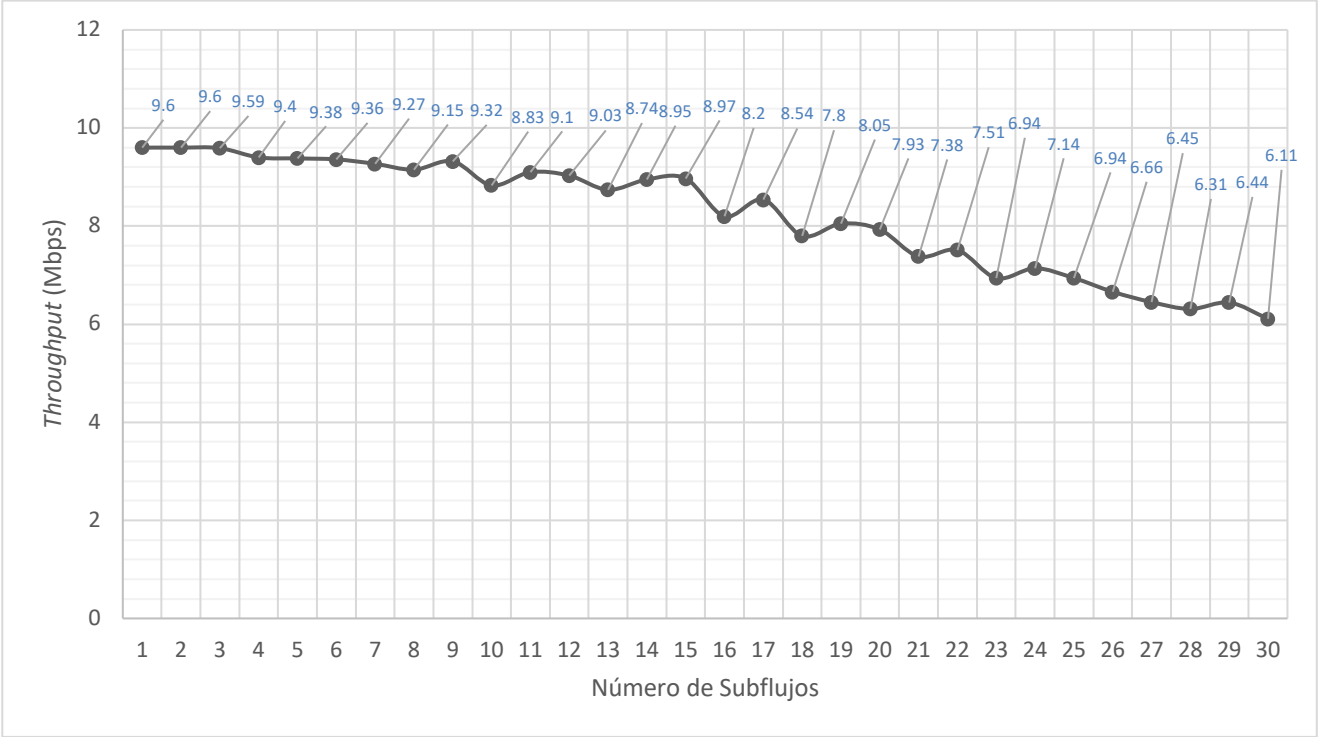


Figura 8 *Throughput* obtenido en todas las pruebas en función del número de subflujos.

6 CONCLUSIONES

En este documento, se presentan los resultados de la simulación del problema del *shared bottleneck* del que adolece MP-TCP, variando el número de subflujos y el número de *routers* de la topología.

Con la simulación del problema del *shared bottleneck* usando NS3 se crearon varios subflujos por una sola interfaz en el cliente y en el servidor; de esto se puede concluir que la creación de subflujos es independiente del número de interfaces de red, aunque se esperaría tener asociado un subflujo a una interfaz de red para tener un aumento de *throughput*.

Además, se pudo observar que al momento de aumentar el número de subflujos, el *throughput* de la conexión MP-TCP tiende a disminuir, esto se produce debido a que esta conexión maneja varias conexiones TCP abiertas; las creaciones de estas conexiones hacen que se genere un cuello de botella y los datos tengan que esperar la finalización del establecimiento de la conexión de los subflujos, para poder enviar datos.

Aplicando el criterio de redes tradicionales, al momento no existe una manera simple de resolver este problema del que adolece MP-TCP, pueden plantearse cambios en el *kernel* de MP-TCP, así en los algoritmos de control de congestión.

Una solución a este problema es factible y de manera relativamente simple usando las Redes Definidas por Software (*Software Defined Networks*, SDN) [3]. Las SDN constituyen una arquitectura de red cuya característica fundamental es desacoplar físicamente el plano de control (inteligencia) del plano de datos, derivando el control a una computadora (controlador) y esperando contar con dispositivos muy rápidos en las tareas de conmutación, aunque con limitada inteligencia (Fig. 9). El control de la red se realiza con la instalación, en los dispositivos de red, de reglas que definen patrones de comportamiento del tráfico. Dichas reglas se programan en el controlador empleando lenguajes de alto nivel para procesar el tráfico, definir las reglas y enviarlas a los dispositivos de red para que se apliquen sobre el tráfico que circula por ellos.

Para la topología planteada, en los dispositivos de red a los que se conectan los hosts cliente y servidor se deberían instalar reglas que permitan evitar el problema del cuello de botella compartido. Esto se encuentra en desarrollo utilizando el controlador *OpenDaylight*. Laque se genere un cuello de botella y los datos tengan que esperar simulación de los dispositivos de red y hosts se realiza con NS3, tomando como base lo desarrollado y presentado en este artículo.

7 AGRADECIMIENTOS

Los autores desean expresar su reconocimiento y agradecimientos a la Escuela Politécnica Nacional por el soporte financiero y logístico, en particular durante el desarrollo de las actividades del Proyecto de Investigación Interno PII-15-14.

8 REFERENCIAS

[1] Raiciu,C.; Paasch, C.; Barre, S. Ford, A. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. Presented as part of the 9th USENIX Symposium on Networked, 2012, pp 13-14.

[2] Raiciu, C.; Barre, S.; Pluntke, C.; Greenhalgh, A.;Wischik, D.; Handley, M. Improving Datacenter Performance and Robustness with Multipath TCP, In Proceedings of the ACM SIGCOMM conference (SIGCOMM '11), 2011.

[3] Sandri, M.; Silva, A.; Rocha, L.; Verdi, F. On the Benefits of Using Multipath TCP and Openflow in Shared Bottlenecks, IEEE, 2015.

[4] nsman, ns-3 Tutorial 3.14 version. URL: https://www.nsnam.org/docs/release/3.14/tutorial/ns-3 tutorial.pdf. (16.12.2016).

[5] Ford, A.; Raiciu, C.; Handley, M.; Bonaventure, O. TCP Extensions for Multipath Operation with Multiple Addresses, IETF RFC-6824, 2013.

[6] Méndez, S. Á. Análisis de protocolo MPTCP en plataformas Linux, Madrid, 2015. Universidad Carlos III de Madrid. Departamento de. Ingeniería Telemática, 2015. URL: http://e-archivo.uc3m.es/handle/10016/23646.

[7] thehajime, github, thehajime, URL: https://github.com/direct-code-execution/ns-3-dce/blob/master/test/dce-mptcp-test.cc. (27.12.2016).

[8] nsnam, ns-3 Manual, 22 6 2012. URL: https://www.nsnam.org/docs/release/3.14/manual/ns-3-manual.pdf. (11.10.2016).

[9] nsnam, ns3 Model Library Release 3.14. URL: https://www.nsnam.org/docs/release/3.14/models/ns-3-model-library.pdf. (12.10.2016).

[10] Lacage, M. ns-3 Direct Code Execution (DCE), URL: https://www.nsnam.org/docs/dce/release/1.9/manual/ns-3-dce-manual.pdf. (16.10.2016).

