

Contents

1	Introducción	2
2	Estado del arte	3
3	Entorno virtual	4
3.1	Requerimientos del entorno virtual	4
3.2	Elección de la herramienta	5
3.3	Diseño e implementación del entorno	6
3.3.1	RAUController	7
3.3.2	RAUSwitch	7
3.3.3	QuaggaRouter	8
3.3.4	RAUHost	9
4	Pruebas de escala	10
4.1	Funcionamiento básico para distintas topologías	10
4.1.1	Descripción del escenario	10
4.1.2	Resultados y observaciones	12
4.2	Escala de servicios y flujos	12
4.2.1	Descripción del escenario	12
4.2.2	Resultados y observaciones	13
5	Conclusiones	16
	Mencionar: Cambiar SNMP por OVS	

Chapter 1

Introducción

Chapter 2

Estado del arte

Chapter 3

Entorno virtual

Uno de los principales objetivos de este trabajo es realizar pruebas funcionales y de escala sobre la arquitectura del prototipo. Es de interés generar distintas realidades, y así detectar puntos de falla o variables clave en la performance de la arquitectura. Para esto se puede utilizar dos parámetros: topología y servicios. Es importante poder aplicar topologías complejas y relativamente grandes a la arquitectura, así como grandes cantidades de servicios, y de esta forma encontrar posibles problemas con la arquitectura, y su respectiva solución. Dado que no es realista hacer este tipo de pruebas con un prototipo físico, por temas económicos y prácticos, se observa la necesidad de un entorno virtual capaz de simular las características del prototipo.

3.1 Requerimientos del entorno virtual

Los requerimientos de este entorno se pueden dividir en dos grupos. En primer lugar, la idea es que el entorno virtual se comporte de una forma lo más fiel posible al prototipo físico. Esto no quiere decir que deba usar las mismas herramientas, pero es deseable que así sea. En segundo lugar, hay que considerar los requerimientos inherentes de un entorno de simulación como el que se pretende. El primer grupo se detalla a continuación.

- Se debe poder simular múltiples RAUSwitch virtuales, y los mismos deben tener las mismas capacidades funcionales que sus pares físicos. A partir de esto, se desprenden los siguientes sub-requerimientos.
 - Deben poder ejecutar el protocolo de enrutamiento OSPF. Es deseable que lo hagan mediante la suite de ruteo Quagga.
 - Deben soportar el protocolo OpenFlow 1.3. Esto se debe a que la aplicación que implementa VPNs depende de que los switches

tengan soporte para MPLS, y OpenFlow ofrece esta funcionalidad a partir de la versión 1.3 (???). Es muy deseable que lo hagan mediante OpenVSwitch, ya que es lo que utilizan los RAUSwitch físicos.

- Se debe poder simular múltiples hosts, ya que son los agentes que se conectan a la red y se envían tráfico entre sí, para corroborar que los flujos de datos son correctos.
- La aplicación RAUFlow debe ejecutarse y comunicarse correctamente con los RAUSwitch. Esto implica que el controlador Ryu debe ser soportado por el entorno.

Cabe remarcar que los módulos SNMP y LSDB Sync quedan por fuera de los requerimientos principales, por ser no esenciales.

El segundo grupo de requerimientos es más genérico, ya que son los que surgen para casi cualquier entorno de simulación de redes.

- Facilidad de configuración. Es importante que el entorno pueda generar distintas topologías y escenarios sin demasiado esfuerzo de configuración.
- Escalabilidad. Dado que uno de los objetivos es realizar pruebas de escala, el entorno debería ofrecer buena escalabilidad. Esto se traduce a que una computadora promedio de uso personal pueda levantar algunas decenas de nodos virtuales como mínimo.

3.2 Elección de la herramienta

Se estudió el estado del arte en lo que respecta a opciones de emulación o simulación para SDN. A continuación se detallan las principales herramientas analizadas al momento de hacer esta investigación.

NS-3

ns-3 fue descartado debido a que no ofrece soporte para Quagga ni OpenFlow 1.3 al momento de realizar esta investigación.

Estinet

Estinet requiere licencias pagas, y se optó por elegir herramientas open source. Debido a la falta de documentación de libre acceso, no se sabe que

tipo de capacidades ofrece.

Mininet

Mininet es un emulador de redes SDN que permite emular hosts, switches, controladores y enlaces. Utiliza virtualización basada en procesos para ejecutar múltiples instancias (hasta 4096) de hosts y switches en un único kernel de sistema operativo. También utiliza una capacidad de Linux denominada *network namespace* que permite crear "interfaces de red virtuales", y de esta manera dotar a los nodos con sus propias interfaces, tablas de ruteo y tablas ARP. Lo que en realidad hace Mininet es utilizar la arquitectura *Linux container*, que tiene la capacidad de proveer virtualización completa, pero de un modo reducido ya que no requiere de todas sus capacidades. Mininet también utiliza *virtual ethernet (veth)* para crear los enlaces virtuales entre los nodos.

LXC

La opción de crear nodos con Linux containers resuelve el problema de Quagga y OpenFlow 1.3, pero llevaría una gran cantidad de trabajo construir distintas topologías (sobre todo si son grandes), ya que casi todo debe ser configurado manualmente por el usuario. Es una opción similar a Mininet, solo que sin gozar de todas las facilidades que ofrece esta última.

Máquinas virtuales

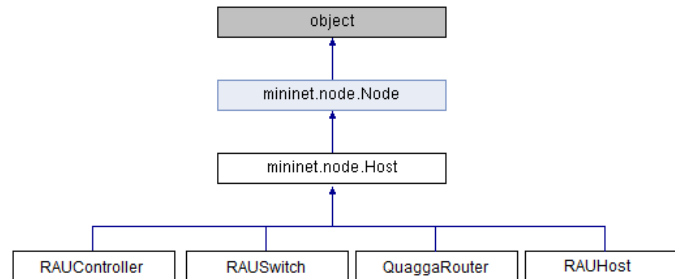
Es una opción similar a LXC (Linux Containers), sólo que menos escalable.

3.3 Diseño e implementación del entorno

El entorno está construido alrededor de Mininet, y se podría pensar como una extensión de la misma. *Out of the box*, Mininet ya cumple tres de los cuatro requerimientos explicados anteriormente. Está diseñada para ser escalable, ya que usa containers reducidos, tiene soporte para OpenFlow 1.3 mediante OpenVSwitch, y es muy fácil de usar. El aspecto en el que falla es en el soporte para Quagga. Dado que Mininet es una herramienta de prototipado para SDN puro, no está pensado para un esquema híbrido como el que se propone. Los switches compatibles con OpenVSwitch que ofrece no pueden tener su propio network namespace, por lo tanto, no pueden tener su propia tabla de ruteo ni interfaces de red aisladas, así que no es posible que utilicen Quagga.

Por otro lado, los hosts de Mininet sí tienen su propio network namespace, y gracias a su capacidad de tener sus propios procesos y directorios, podemos

Figure 3.1: Diagrama de clases del entorno.



ejecutar una instancia de Quagga y OpenVSwitch para cada host. De esta forma es posible crear un router como el requerido por la arquitectura. Esta extensión de las funcionalidades de los hosts es posible ya que Mininet está programado con orientación a objetos y permite al usuario crear subclases propias de las clases que vienen por defecto. En la figura 3.1 se puede ver la estructura de clases del entorno construido. En las siguientes secciones se procederá a estudiar cada una de ellas.

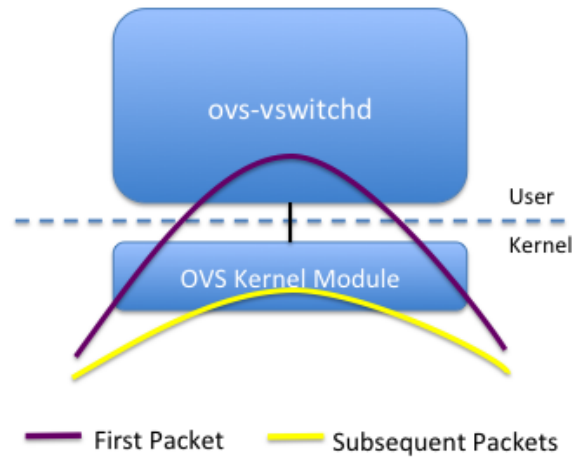
3.3.1 RAUController

En el uso típico de Mininet, la comunicación entre el controlador y el switch se da a través de la interfaz de loopback. Esto es así porque los switches no tienen su propio namespace. Para lograr dicha comunicación, no hace falta un objeto en Mininet que represente el controlador, ya que ejecutar la aplicación en el sistema operativo base ya habilita al switch a comunicarse con ella a través de la interfaz de loopback. Esta situación cambia en este diseño, porque los switches pasan a tener su propio network namespace. Esto lleva a la necesidad de crear un host virtual, que ejecute la aplicación de RAUFlow y se comunique con los switches a través de enlaces virtuales. Para satisfacer esta necesidad se usa la clase RAUController.

3.3.2 RAUSwitch

La clase RAUSwitch es el núcleo del entorno de simulación. Es un Host extendido de tal forma para que, gracias a la funcionalidad de directorios privados, ejecute su propia instancia de Quagga y OpenVSwitch. Cada RAUSwitch tiene los siguientes directorios privados: /var/log/, /var/log/quagga, /var/run, /var/run/quagga, /var/run/openvswitch. Cada RAUSwitch también usa un directorio bajo /tmp, para almacenar sus archivos de configuración.

Figure 3.2: Arquitectura de OpenVSwitch.



OpenVSwitch básicamente consiste de 2 demonios (`ovs-vswitchd` y `ovsdb-server`) que ejecutan en el user-space, y un módulo en el kernel que actúa como cache para los flujos recientes. Utiliza el protocolo 'netlink' para comunicar el user-space con el módulo en el kernel. Poder tomar decisiones sobre los paquetes a nivel del kernel, sin tener que pasar por el user-space, explica en gran medida el buen nivel de performance que ofrece OpenVSwitch. Sin embargo, tener múltiples módulos de kernel ejecutando en el mismo sistema operativo puede crear comportamientos impredecibles e incorrectos, ya que no está previsto para trabajar de esa forma.

Afortunadamente, OpenVSwitch puede ejecutarse completamente en modo user-space, es decir, sin soporte del módulo del kernel. Esto implica que podemos ejecutar tantas instancias de OpenVSwitch como queramos, pero la performance va a ser significativamente peor. Esto no es una desventaja muy seria, ya que el objetivo del entorno no es ser performante al procesar paquetes. Cabe aclarar que en este modo OpenVSwitch continúa haciendo cacheo de flujos, pero ahora lo hace en el user-space.

3.3.3 QuaggaRouter

Es una clase similar al RAUSwitch pero sin OpenVSwitch, es decir, sólo usa Quagga. Apunta a representar el router CE que utilizaría una subred para conectarse a la red. Está conectado a un RAUSwitch de borde.

3.3.4 RAUHost

Es una clase auxiliar, sin ninguna particularidad. Sirve para evitar determinadas configuraciones manuales, como por ejemplo, el *default gateway*.

Chapter 4

Pruebas de escala

En esta sección se listarán las distintas pruebas de escala realizadas. Se explicará el propósito de cada prueba, las características de cada una de ellas (topologías, tipos de tráfico, etc) y por último, los resultados que arrojan.

4.1 Funcionamiento básico para distintas topologías

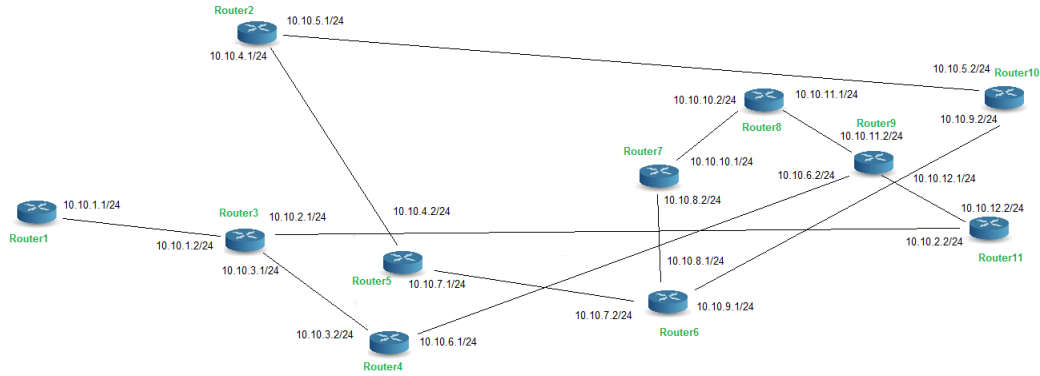
La idea de esta prueba es en cierta manera extender las pruebas que ya fueron realizadas por el proyecto RRAP, y corroborar el funcionamiento básico de la arquitectura y de la aplicación para topologías de diferentes tamaños y características.

4.1.1 Descripción del escenario

El objetivo es crear una VPN de capa 3 entre dos subredes CE, que permita tráfico de ethertype **0x0800**, es decir, tráfico del protocolo IPv4. Luego se generan datos y se envían por la VPN. De esta forma, se prueba que funcionan correctamente las siguientes funcionalidades.

- Algoritmo de ruteo. Se verifican dos aspectos claves: que el camino se corresponde con el camino esperado (calculado previamente de forma manual), y que el camino es correctamente instalado en forma de reglas de reenvío (en base a conmutación de etiquetas MPLS) en las respectivas tablas de flujos OpenFlow de cada nodo del camino. Todo esto se puede comprobar analizando las tablas de flujos de cada nodo, que se pueden ver utilizando el comando **dump-flows** de OpenVSwitch. También se puede utilizar la interfaz gráfica de RAUFlow. Desde las

Figure 4.1: Topología chica



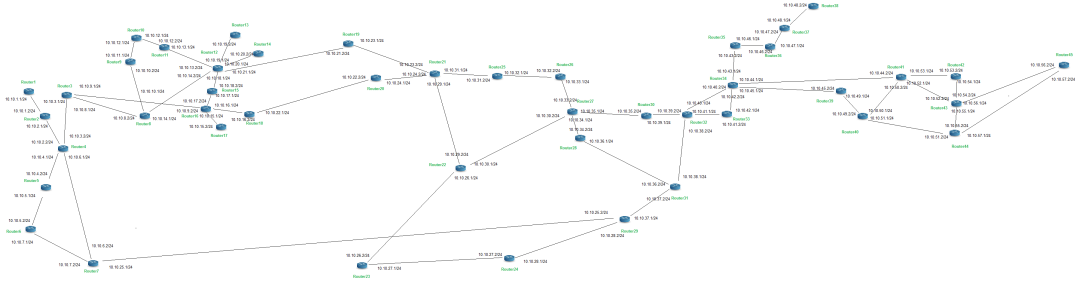
tablas de flujos se puede reconstruir el camino que computó la aplicación, y también comprobar que los flujos configuran correctamente las etiquetas MPLS.

- Clasificación de tráfico. La idea es verificar que realmente se están asignando las etiquetas MPLS al tráfico entrante, así como comprobar que el mismo es reenviado por los nodos correctos. Para ello se genera tráfico utilizando el comando **ping** y la herramienta **iperf**. Con la herramienta **tcpdump**, se verifica el tráfico que pasa por cada nodo del camino.

Para comprobar que el comportamiento es consistente, las pruebas son realizadas con las siguientes topologías:

- Básica: 4 RAUSwitch, en topología de full mesh. Es la utilizada en el prototipo físico.
- Chica: topología arbitraria de 11 nodos (fuente: Topology Zoo). Figura 4.1.
- Mediana: topología arbitraria de 45 nodos (fuente: Topology Zoo). Figura 4.2.
- Grande: topología arborescente de 100 nodos

Figure 4.2: Topología mediana



4.1.2 Resultados y observaciones

Problema del MTU al usar iperf

Hay que reducir 5 o 10 bytes (dependiendo de si el servicio usa 1 o 2 niveles de etiquetas MPLS) al MTU para que el tráfico pase.

Bug en código (ruta)

Error en el código que hacía que se instalaran mal los flujos en los nodos. Tenían incorrectos puertos de entrada y salida.

Bug en código (Dijkstra)

Error en el código del algoritmo de Dijkstra que hacía que se calcularan mal los costos, ya que se sumaba como strings (concatenación) en vez de sumar enteros.

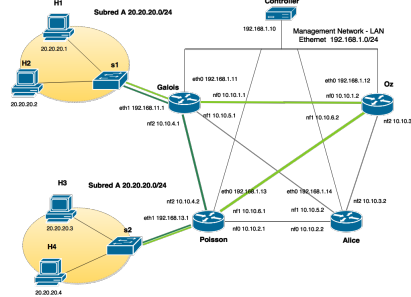
4.2 Escala de servicios y flujos

Es probable que si una arquitectura de este estilo fuera desplegada en la Red Académica Uruguaya, sería sujeta a grandes cantidades de tráfico y, en particular, grandes cantidades de servicios. Por eso es de gran interés realizar pruebas sobre la misma que analicen su comportamiento cuando debe manejar decenas de miles, o incluso millones de flujos distintos. De esta forma se podrían identificar posibles puntos de falla, o umbrales bajo los cuales debe mantenerse la red para funcionar con buen rendimiento.

4.2.1 Descripción del escenario

Para estas pruebas se utiliza una topología básica de 4 RAUSwitch y dos subredes, como indica la figura 4.3. Se crean 15.000 VPNs de capa 2 entre las

Figure 4.3: Topología para prueba de escala de servicios.



subredes, variando los cabezales OpenFlow para que toda VPN sea distinta de las demás. Además, se crea una VPN de capa 3 entre los mismos nodos, para permitir pasar el tráfico IP entre los hosts, que es el que se utiliza en las pruebas.

Interesa realizar estas pruebas estudiando dos aspectos claves:

- Escalabilidad interna del RAUSwitch. Se estudian posibles limitaciones internas que puedan tener los dispositivos, cuando deben manejar grandes cantidades de flujos.
- Escalabilidad en servicios. Se estudian posibles problemas que puedan tener la arquitectura de la red o la aplicación del controlador para manejar grandes cantidades de servicios e información.

4.2.2 Resultados y observaciones

El comportamiento de la arquitectura al manejar esa cantidad de servicios es consistente, por lo tanto, es posible afirmar que la arquitectura de la red no tiene limitaciones con respecto a la cantidad de servicios. Sin ser una limitación, pero sí un factor importante, hay que recordar que los datos que maneja el controlador (entre ellos, los servicios) están en memoria. Por lo tanto se podrá agregar servicios mientras la computadora subyacente tenga suficiente memoria. La creación de 15.000 servicios aumenta el consumo de memoria del controlador en 205 Mb (CONFIRMAR), por lo que un servicio ocupa alrededor de 14 Kb. A modo de ejemplo, si extrapolamos ese número a una computadora que puede dedicar 4 Gb de RAM al controlador, llegamos a que dicho controlador podrá mantener alrededor de 300.000 servicios.


El otro aspecto de interés, la escalabilidad interna del RAUSwitch, arroja resultados similares. En teoría, cuantos más flujos en la tabla, más debería demorar el switch OpenFlow en encontrar el flujo que corresponde con el tráfico que está analizando, y por lo tanto el paquete demora más en ser

forwardado. Esto debería tener un impacto directo en el throughput. Como ya fue explicado, para comprobar esto se crearon 15.000 VPNs de capa 2. Esto implica alrededor de 1.260.000 flujos en ambos nodos, ya que cada VPN consiste de 2 servicios, y cada servicio de capa 2 instala 42 flujos en los nodos involucrados. Con la herramienta 'iperf', se crea tráfico TCP entre hosts de distintas subredes y se mide el throughput promedio sin las VPNs y con ellas. Ambos números resultan iguales, lo cual implica que la velocidad de transferencia no es afectada por más de un millón de flujos.

La explicación de este resultado se encuentra en la especificación de la herramienta OpenvSwitch, que utiliza la arquitectura para implementar OpenFlow. Dicha herramienta realiza cacheo de flujos, es decir, cuando el paquete de un determinado flujo llega por primera vez a un nodo, este paquete es enviado al pipeline de OpenFlow para determinar que acción se debe tomar. Luego de realizada, esta acción es escrita en la caché, y tiene un tiempo de vida de entre 5 y 10 segundos. Si en ese período de tiempo llega otro paquete del mismo flujo, no hay necesidad de enviar el paquete al pipeline, porque ya se sabe cuales son las acciones a tomar para ese paquete. Por lo tanto, si un flujo de datos es constante y rápido, no importa cuántos flujos OpenFlow tenga el nodo, ya que sólo el primer paquete de ese flujo deberá pasar por el pipeline, y por ende, solo él se verá demorado por la existencia de muchos flujos.

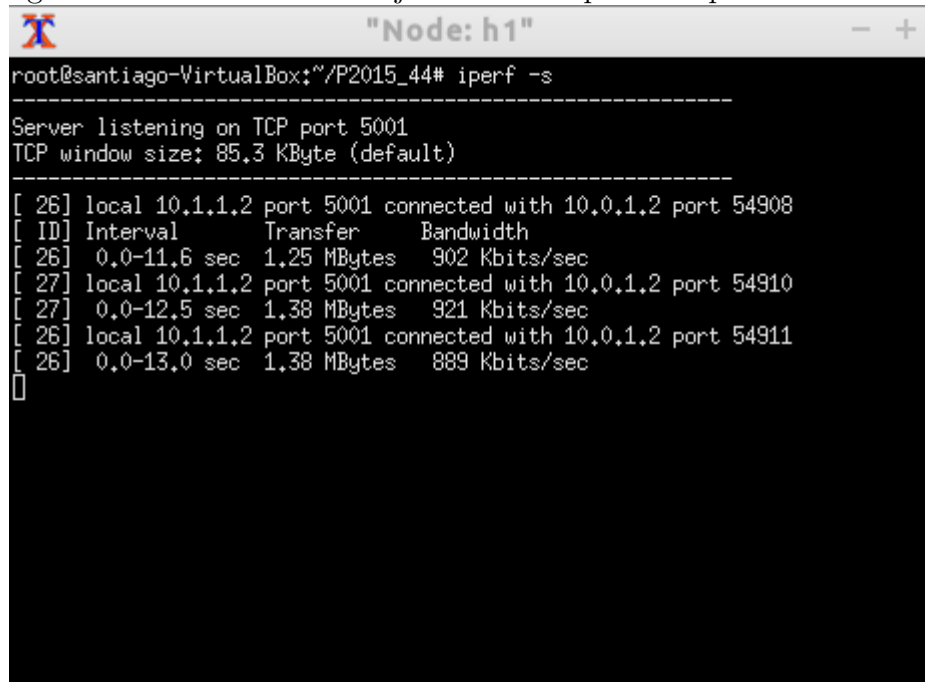
Mediante el comando 'ovs-appctl dpctl/show' de OpenVSwitch, podemos examinar las estadísticas del datapath para cada instancia de OpenVSwitch de nuestro entorno. En las figuras 4.5 y 4.4 se observa, por un lado, la salida de 'iperf' luego de hacer tres ejecuciones, y por otro, las estadísticas del nodo 'alice' luego de dichas ejecuciones. En la sección 'lookups' se detallan cuantos 'hits' y 'miss' de caché han ocurrido hasta el momento, y 'flows' indica cuantos flujos activos hay en el momento en la caché.

Figure 4.4: Estadísticas de cache de flujos del nodo 'alice'.



```
root@santiago-VirtualBox:~/P2015_44# ovs-appctl --target=/tmp/alice/ovs/ovs-vswn
itcd.9999.ctl dpctl/show
netdev@ovs-netdev:
    lookups: hit:6999 missed:95 lost:0
    flows: 10
    port 1: alice (tap)
    port 7: valice-eth2 (tap)
    port 2: alice-eth1
    port 3: alice-eth2
    port 5: alice-eth4
    port 6: valice-eth1 (tap)
    port 0: ovs-netdev (internal)
    port 4: alice-eth3
    port 8: valice-eth3 (tap)
    port 9: valice-eth4 (tap)
system@ovs-system:
    lookups: hit:0 missed:0 lost:0
    flows: 0
    masks: hit:0 total:1 hit/pkt:0.00
    port 0: ovs-system (internal)
root@santiago-VirtualBox:~/P2015_44#
```

Figure 4.5: Resultado de la ejecución de 3 pruebas iperf en el host h1.



```
root@santiago-VirtualBox:~/P2015_44# iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 26] local 10.1.1.2 port 5001 connected with 10.0.1.2 port 54908
[ ID] Interval      Transfer    Bandwidth
[ 26] 0.0-11.6 sec  1.25 MBytes  902 Kbits/sec
[ 27] local 10.1.1.2 port 5001 connected with 10.0.1.2 port 54910
[ 27] 0.0-12.5 sec  1.38 MBytes  921 Kbits/sec
[ 26] local 10.1.1.2 port 5001 connected with 10.0.1.2 port 54911
[ 26] 0.0-13.0 sec  1.38 MBytes  889 Kbits/sec
█
```

Chapter 5

Conclusiones