Estudio de estrategias de implementación del Plano de Control en un Sistema Autónomo de Internet

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

Santiago Colman **Federico Godán**

Tutor: Eduardo Grampín

Facultad de Ingeniería

Universidad de la República

Instituto de Computación Julio 2016

Índice

Lista de Figuras

Lista de Tablas

Nomenclatura

Acrónimos

AS Autonomous System

ASBR Autonomous System Border Routers

ASN Autonomous System Number

BGP Border Gateway Protocol

EAS Experimental Autonomous System

eBGP External Border Gateway Protocol

EGP Exterior Gateway Protocol

IANA Internet Assigned Numbers Authority

iBGP Internal Border Gateway Protocol

IGP Interior Gateway Protocol

ISP Internet Service Provider

LLDP Link Layer Discovery Protocol

OSPF Open Shortest Path First

PGM Pragmatic General Multicast

RC Router Confederation

RIP Routing Information Protocol

RMTP Reliable Multicast Transport Protocol

RR Route Reflector

SDN Software-Defined Networking

SmartPGM Smart Pragmatic General Multicast

TCP Transmission Control Protocol

TRDP TIBCO Reliable Datagram Protocol

UDP User Datagram Protocol

Sección 1

Introducción y objetivos

La Internet se compone de un amplio número de dominios llamados Sistemas Autónomos (AS). Éstos a su vez son conformados por múltiples redes que responden a una misma política de ruteo. Ejemplos pueden ser: redes empresariales, redes universitarias o proveedores de servicios de Internet (ISP). Dicha política es implementada por un protocolo interno (IGP) para el enrutamiento dentro del AS, como puede ser RIP u OSPF, y por un protocolo externo (EGP) para el enrutamiento entre los AS propiamente dichos. Actualmente, el protocolo externo de hecho en la Internet es BGP.

BGP es un protocolo mediante el cual se intercambia información de enrutamiento entre ASs. Podemos distinguir dos formas de realizar este intercambio: external BGP (eBGP) se encarga del intercambio de información entre sistemas autónomos diferentes, mientras que internal BGP (iBGP) se encarga de la distribución de las rutas aprendidas por eBGP dentro de un mismo sistema; para esto mantiene sesiones activas entre todos los routers de la red (full mesh) a fin de comunicar y tener actualizados a los routers internos sobre cómo alcanzar los destinos externos. Esto, si bien funciona correctamente, consume una gran cantidad de recursos ya que cada router debe encargarse de mantener activas las sesiones iBGP, diseminar las rutas aprendidas a los demás y guardar las mismas.

A fin de reducir la cantidad de sesiones iBGP dentro del sistema, a la fecha se han investigado e implementado distintas alternativas; sin embargo éstas han traído consigo nuevos problemas y desafíos a superar. A causa de esto es de interés encontrar otras aproximaciones para la resolución de este problema.

Por otra parte, en los últimos años se ha buscado desacoplar las funcionalidades del Plano de Control de los dispositivos que componen la red; es así como surgió el paradigma de Redes Definidas por Software (SDN), dando lugar a nuevas aproximaciones a problemas existentes en el área.

Motivados por esto, planteamos investigar la factibilidad y posible implementación de una alternativa al uso del full mesh iBGP empleando el paradigma SDN. Se verificará que su comportamiento es equivalente al que se obtiene con el full mesh y además se registrarán métricas comparativas sobre el funcionamiento de ambas opciones.

Sección 2

Estado del arte

2.1 Protocolo BGP

Border Gateway Protocol (BGP) es un protocolo de ruteo intra dominios utilizado para poder dirigir el tráfico IPv4/IPv6 entre distintos ASs. Un AS puede definirse como un conjunto de redes IP bajo una misma administración y con un conjunto de políticas definidas en común. Los ASs poseen globalmente un identificador único conocido como Autonomous System Number (ASN) el cual es designado y regulado por la Internet Assigned Numbers Authority (IANA). La existencia de BGP está motivada en el hecho de que los protocolos utilizados tradicionalmente para ruteo IGP se ven limitados en cuanto a las dimensiones de las redes que pueden manejar, ya que no escalan demasiado bien. Además las configuraciones necesarias comienzan a volverse muy tediosas a medida que la red crece.

BGP es un protocolo de vector de distancia; en ausencia de políticas se comporta considerando como métrica el largo del atributo AS_PATH de cada ruta aprendida. Utiliza conexiones TCP en el puerto 179 para llevar a adelante su función. Maneja cuatro tipos de mensajes [1]:

Open: luego de que un router vecino (o *peer*) es configurado, BGP envía un mensaje de este tipo para tratar de establecer la conexión con el mismo en un proceso que denominaremos *peering*. Este mensaje incluye información como ASN, identificador de router y tiempo de espera (*hold time*).

Update: mensaje utilizado para transmitir información de ruteo entre los routers con los que ya se estableció una conexión. Estos mensajes incluyen información sobre nuevas rutas disponibles, rutas fuera de funcionamiento y otros atributos sobre éstas.

Keepalive: los *peers* BGP que han establecido una conexión intercambian este tipo de mensajes cada cierto período de tiempo, por defecto 60 segundos, para poder mantener la conexión activa.

Notification: cuando ocurre algún tipo de problema que hace que la conexión entre *peers* se pierda, un mensaje de este tipo es enviado al resto de sus *peers* y la conexión se cierra.

Las conexiones BGP establecidas entre dos *peers* son denominadas de distinta forma, dependiendo del AS al que pertenezcan los mismos. Si los routers se encuentran en el mismo AS, entonces la conexión es denominada iBGP. Si se encuentran en ASs diferentes entonces la conexión es llamada eBGP; a aquellos routers que tienen este tipo de conexiones se les conoce como routers de borde (ASBR).

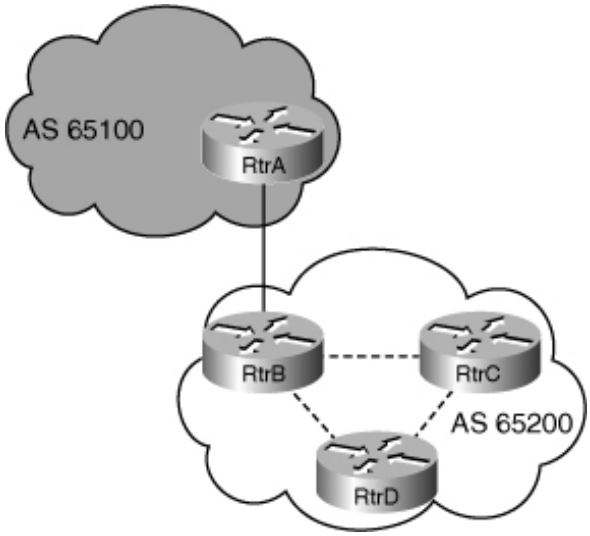

Fig. 2.1 El router RtrA ubicado en AS65100 mantiene una sesión eBGP con RtrB en AS65200, por lo tanto estos routers son los denominados ASBR. Por otro lado RtrB mantiene sesiones iBGP con RtrC y RtrD. Estos últimos también mantienen una sesión iBGP entre ellos. Dado que RtrC y RtrD no mantienen ningún tipo de sesión eBGP los denominaremos routers internos [2]

BGP trata de distinta manera los mensajes UPDATE dependiendo de si su procedencia es una conexión iBGP o eBGP. Si el mensaje UPDATE proviene de una conexión eBGP, entonces el router va a recalcular las mejores rutas y en caso de seleccionar una nueva como la mejor, la comunicará a todos los vecinos que tiene configurados, tanto por iBGP como eBGP; en este último caso, se agrega al AS_PATH del mensaje el ASN del AS al que pertenece. Sin embargo, si el mensaje proviene de una conexión iBGP el router sólo computa la mejor solución y no la retransmite a sus vecinos conectados por iBGP sino que sólo lo hace a aquellos conectados a través de eBGP.

BGP rige su funcionamiento por una máquina de estados, y alterna entre los mismos dependiendo del tipo de mensaje que reciba. La máquina esta compuesta por seis estados:

Idle: no existe conexión con el *peer*. El router se encuentra actualmente tratando de encontrar el *peer* configurado, para ello revisa su tabla de ruteo en busca de maneras de llegar al mismo.

Connect: el *handshake* TCP con el *peer* se ha completado.

OpenSent: se envió un mensaje Open con los parámetros del AS a fin de lograr establecer la conexión.

Active: similar a OpenSent, se diferencian en que para llegar a este estado pueden no haberse aceptado inicialmente los parámetros enviados en el mensaje Open.

OpenConfirm: se ha recibido respuesta positiva al mensaje Open enviado.

Established: es el estado deseable, la conexión ha sido establecida satisfactoriamente y puede comenzar el intercambio de mensajes UPDATE con el *peer*.

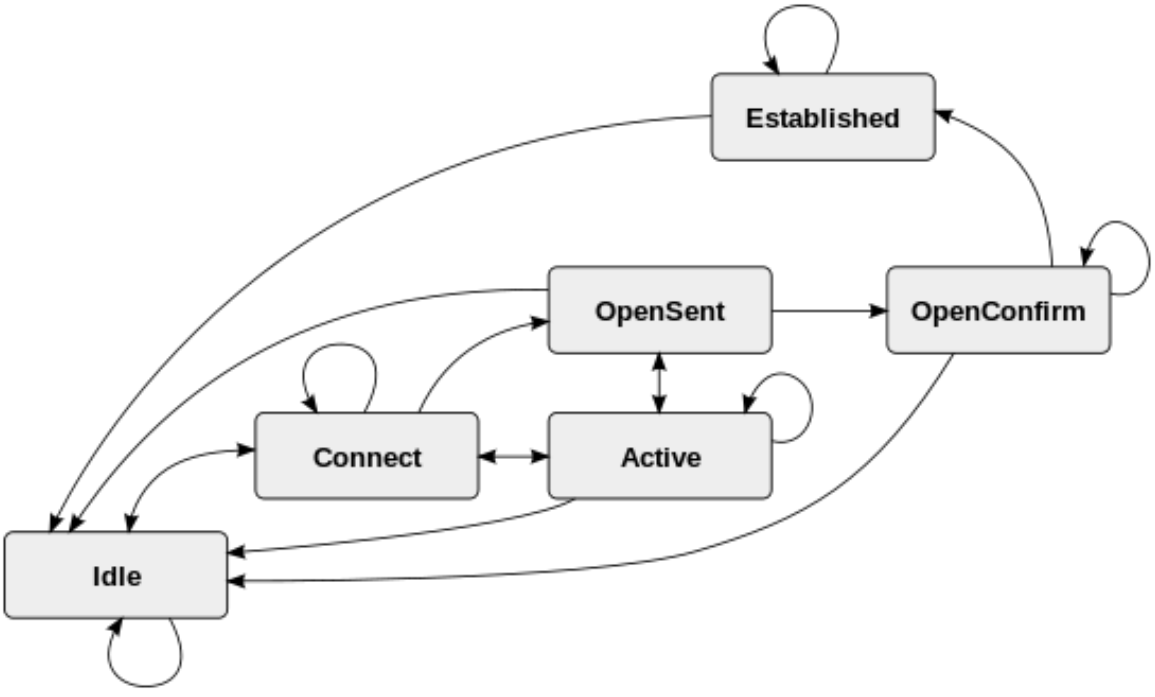

Fig. 2.2 Máquina de estados BGPv4[3]

BGP asume por defecto que dos routers vecinos en AS diferentes se encuentran conectados directamente; para ello establece el parámetro TTL=1 de los mensajes a través de

conexiones eBGP (esto puede modificarse en la mayoría de los motores de ruteo mediante el parámetro *ebgp-multi-hop*). Para los routers conectados por iBGP esta restricción no existe.

Ante la llegada de un mensaje UPDATE, BGP debe decidir las mejores rutas a utilizar teniendo en cuenta la nueva información recibida y lo actualmente presente en su tabla de ruteo. El proceso de decisión es realizado siguiendo secuencialmente el siguiente conjunto de reglas:

1. Ignorar rutas si el NEXT-HOP no es alcanzable.
2. Preferir las rutas con mayor LOCAL_PREF.
3. Preferir las rutas con el AS_PATH más corto.
4. Preferir las rutas que tengan el menor ORIGIN.
5. Para rutas recibidas desde el mismo AS, preferir aquellas con el MED menor.
6. Preferir rutas aprendidas por eBGP sobre las aprendidas por iBGP
7. Preferir rutas con el costo IGP menor.
8. Preferir rutas de *peer* con dirección IP menor.

El resultado de la evaluación de las reglas 1 a 5 es el mismo para todos los routers iBGP pues tienen en consideración atributos globales que no suelen ser modificados por iBGP. Estas variables son:

LOCAL_PREF: es una variable estática definida por el administrador, por defecto vale 100. En base a la configuración es asignada y transmitida por un ASBR a sus pares a través de iBGP.

AS_PATH: es una lista con los ASN de todos los ASs que ha atravesado la información de ruteo.

ORIGIN: dato cargado por el *peer* que genera la información de enrutamiento del mensaje. Se asigna un valor numérico dependiendo de qué tipo es, tomando valores dependiendo del protocolo por el cual es enviado: IGP=0, EGP=1 e INCOMPLETE=2.

MED: utilizado cuando existe más de un enlace a un AS vecino. Sirve para discriminar entre estas opciones.

2.1.1 Dificultades de BGP

BGP fue pensado para brindar confiabilidad, escalabilidad y control, no velocidad[2]. Por lo que su convergencia en general es lenta.

A esto se suma que en redes donde se utiliza iBGP se debe tener una conexión full mesh entre todos los routers de la red a fin de garantizar el correcto funcionamiento del protocolo. Esto resulta en un problema de escalabilidad en redes grandes, pues para una red de N routers es necesario contar con: $\frac{n(n-1)}{2}$ sesiones TCP activas, lo cual evidentemente consume una gran cantidad de memoria y capacidad de procesamiento en los routers[4][5].

Por otro lado, dado que BGP no cuenta con ningún método de descubrimiento, antes de poder hablar con cualquier *peer*, éste debe ser definido de forma estática en el router, lo que se convierte en un trabajo tedioso si se debe configurar todos los routers para estar conectados utilizando full mesh[2].

Los Reflectores de Rutas (RR) y las Confederaciones de Routers (RC) han sido propuestas como una alternativa al full mesh pero han traído consigo nuevas dificultades.

Confederaciones de Routers

El enfoque de las RCs propone formar agrupaciones de routers dentro del AS a modo de "pequeños sistemas autónomos" para luego conectarlos utilizando eBGP entre los mismos. De este modo se reduce la cantidad de sesiones iBGP necesarias dentro del AS compuesto por las RC. Sin embargo, dentro de cada RC se deberá seguir teniendo un full mesh de sesiones iBGP entre todos los routers pertenecientes. Esto por lo tanto sólo permite trasladar el problema original a la interna de cada RC, teniendo como contrapartida una mayor complejidad en la configuración.

Reflectores de Rutas

El uso de RR permite que algunos routers BGP (denominados Reflectores de Rutas) puedan anunciar rutas aprendidas por iBGP a sus clientes dentro del AS en que se encuentra. Esto lleva a reducir la cantidad de sesiones iBGP frente a la configuración full mesh, ya que las sesiones iBGP se establecen únicamente entre los clientes y los RR además de entre RRs. Esta alternativa trae consigo otros problemas si a los RRs no se los localiza y configura correctamente. Estos problemas pueden ir desde la posible divergencia del protocolo, generación de loops al momento de realizar el forwarding de paquetes, reducción de opciones a la hora de seleccionar rutas hasta selección de soluciones no óptimas de enrutamiento.

La selección de los routers que actuarán como RR y sus clientes son un área de investigación al día de hoy.

2.2 Redes Definidas por Software

Se denomina Redes Definidas por Software (SDN) a aquel conjunto de técnicas que permiten la separación del plano de control del plano de datos.

La arquitectura general aceptada de SDN puede verse resumida en tres componentes fundamentales: aplicaciones, controladora y dispositivos SDN[6].

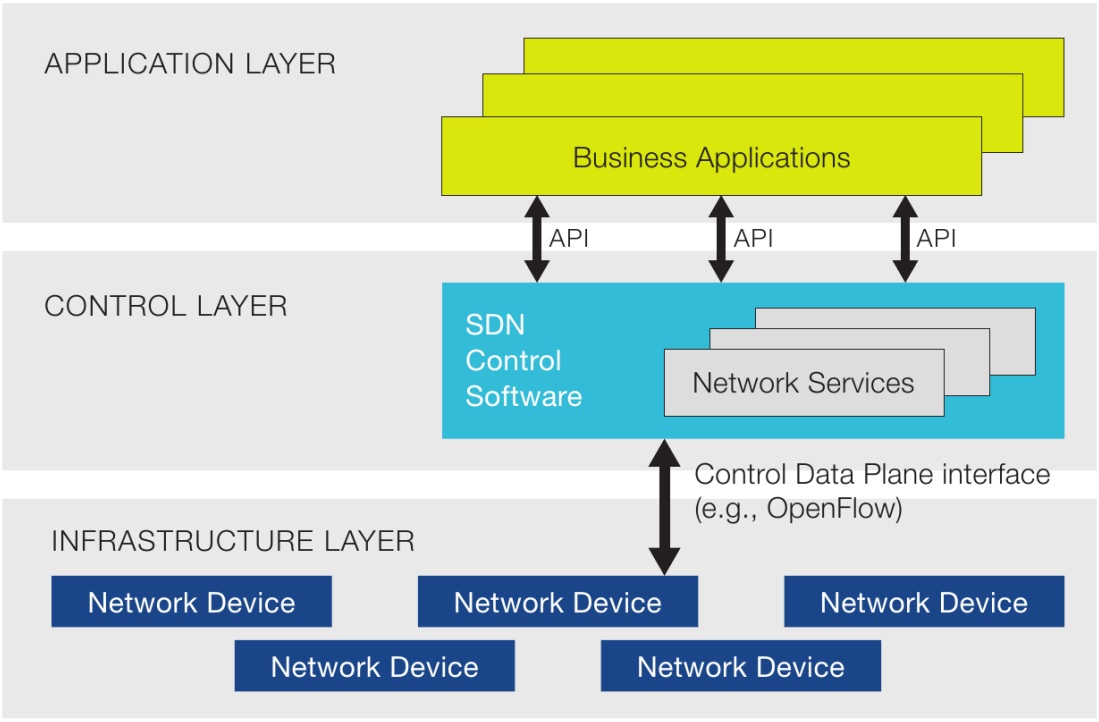

Fig. 2.3 Arquitectura de funcionamiento SDN[7]

2.2.1 Aplicaciones

Son programas que se comunican con la controladora a través de una API bien definida. Estas aplicaciones tienen acceso a todas las funcionalidades disponibles en la controladora, con lo que pueden obtener información sobre la red o indicar la instalación, borrado o modificación de nuevos flujos.

2.2.2 Controladora

La controladora es una componente fundamental de un sistema de redes definidas por software ya que se encarga de ejecutar toda la lógica de las aplicaciones desarrolladas, funcionando de esta forma como el "cerebro de la red".

Es una entidad lógica que recibe peticiones por parte de las aplicaciones, las traduce a reglas que entrega a los dispositivos de la red. Adicionalmente a esto, es responsable de hacer cumplir las políticas de la red y monitorear el estado de ella, obteniendo estadísticas y notificando a las aplicaciones sobre los distintos eventos que acontecen en la misma.

La comunicación desde las aplicaciones hacia ella y desde la misma a sus módulos es realizada a través la llamada NorthBound API. Esta API constituye uno de los componentes críticos de las controladoras y del paradigma SDN en general, ya que las funcionalidades que brinde van a ser lo que permita a los desarrolladores crear aplicaciones para la controladora definiendo de esta forma las capacidades, flexibilidad y posibles restricciones del entorno. Al día de hoy, no hay estándares que definan estas APIs, por lo que, existe una gran variedad funcionando y dependen puramente de la controladora que se vaya a utilizar.

En otro sentido, la comunicación con los dispositivos SDN es llevada adelante a través de la SouthBound API. OpenFlow es el protocolo más conocido que implementa la SouthBound API, sin embargo no es el único; existen otros como Cisco OpFlex[8], NetConf [9] o Lisp. Como tal permite a la controladora administrar la estructura lógica del dispositivo SDN sin entrar en detalle sobre como el mismo implementa el protocolo.

2.2.3 Dispositivos SDN

Dispositivos encargados de realizar el *forwarding* de los distintos mensajes que circulan por la red. Como se dijo anteriormente se comunican con la controladora mediante la SouthBound API de la misma, por lo que para funcionar ejecutan algún software, posiblemente a nivel de firmware, que implementa el manejo de los flujos a través del mismo.

OpenFlow

OpenFlow[10][11] como interfaz de comunicación especifica tres tipos de tablas a mantener en el dispositivo SDN:

Flow Table: es el tipo de tabla más importante. Puede haber una o múltiples; en este último caso, las mismas son numeradas de 0 en adelante y encadenadas en forma de lista simple ordenada. Compara todos los paquetes que llegan al dispositivo y especifica la acción que debe ejecutarse para los mismos.

Un paquete que llegue recorrerá las reglas de comparación secuencialmente en las Flow Tables, tabla a tabla, hasta encontrar una regla que coincida con sus atributos. En ese momento se ejecutará la acción que allí se indique. En caso de que no encuentre ninguna coincidencia, se enviará una copia del mensaje a la controladora notificando de lo ocurrido y preguntando cómo proceder. La controladora notificará a la aplicación que esté pendiente de ese mensaje para que lo procese y defina las acciones a tomar y en base a lo que defina notificará al router.

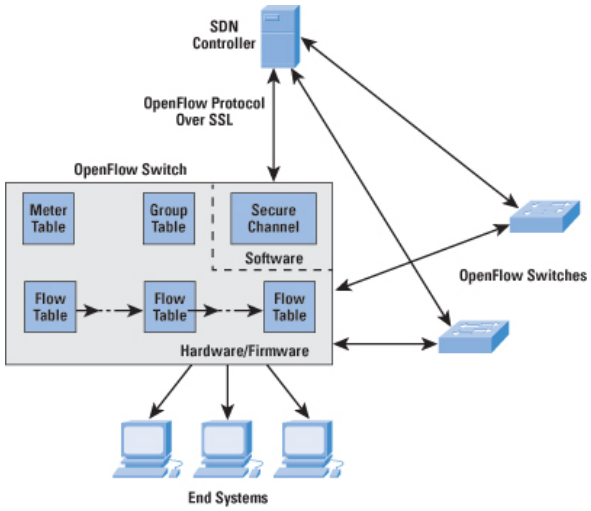

Fig. 2.4 Arquitectura de funcionamiento de dispositivo SDN que implementa OpenFlow [12]

Está compuesta por los campos:

Match Fields: usado para evaluar la correspondencia de la regla con los paquetes que lleguen al dispositivo.

Priority: prioridad relativa de la regla dentro de la tabla actual.

Counters: contadores que registran el flujo de paquetes a través del dispositivo.

Instructions: acciones que deben realizarse si el paquete cumple con las Match Fields. Éstas pueden indicar a donde reenviar el paquete, descartarlo o redirigirlo a otra Flow Table para que continúe su procesamiento.

Timeouts: tiempo máximo que esperará el dispositivo para hacer expirar una regla de flujo.

Cookie: es definido por la controladora. Puede ser utilizado para obtener estadísticas de los flujos, borrado de los mismos u otros.

El campo Match Field a su vez contiene los siguientes parámetros para compararlos con los del paquete recibido:

Ingress Port: número de puerto del dispositivo por el que llegó (físico o virtual).

Ethernet Source or Destination Addresses: MAC de origen del paquete.

IPv4/IPv6 Source Address and Destination Address: IP de origen o destino del paquete.

IPv4/IPv6 Protocol Number: número de protocolo del paquete recibido.

TCP Source/Destination Port: puerto TCP de origen o destino.

UDP Source/Destination Port: puerto UDP de origen o destino

Adicionalmente, la tabla puede contener algunos otros campos opcionales.

Group Table: un flujo puede ser redirigido desde una entrada de la Flow Table a ésta. Se encarga de agrupar conjuntos de acciones a aplicar a los flujos.

Meter Table: se encarga de almacenar y disparar un conjunto de acciones relacionadas a la performance de un determinado flujo.

OpenFlow permite a la controladora agregar, modificar y borrar Flow Tables y reglas definidas en las mismas. Adicionalmente a estas tablas el protocolo define distintos tipos de mensajes que garantizan la comunicación e interacción con la controladora. Los mensajes pueden ser clasificados en tres tipos:

Controladora a dispositivo SDN: son mensajes iniciados por la controladora con destino a un dispositivo SDN de la red, con el fin de administrar las distintas tablas y estados lógicos del mismo. Entre ellos encontramos dos de especial interés:

Packet-Out: mensaje dirigido directo a un puerto de un dispositivo SDN.

Modify-State (FlowMod, GroupMod, PortMod y MeterMod): permite agregar, borrar y modificar entradas de la Flow Table así como configurar algunas opciones del dispositivo SDN.

Asíncronos: son mensajes enviados sin solicitud explícita de la controladora. Los mismos son usados para notificar distintos sucesos a la controladora. Entre estos mensajes cabe destacar el llamado Packet-In, el cual es usado por los dispositivos SDN para preguntar a la controladora acerca de cómo actuar cuando no existe una entrada en la Flow Table que lo contemple.

Simétricos: son mensajes enviados sin que hayan sido solicitados explícitamente por los dispositivos SDN o la controladora. Entre ellos están los mensajes Hello y Echo, utilizados para establecer y verificar la conexión entre los dispositivos SDN y la controladora (así como obtener métricas de esta conexión).

2.2.4 SDN/BGP

En los últimos años, se han explorado e implementado algunas soluciones a los problemas de BGP utilizando el paradigma de SDN[13][14][15]. Si bien en su mayoría estas soluciones son experimentales, han servido de base para abordar los problemas inherentes a BGP sin necesidad de alterar la implementación del protocolo. De esta forma se busca lograr integraciones con las redes actualmente en producción, las que están compuestas mayormente por routers legados que no dan soporte a SDN.

2.2.5 Routing Control Platform

La Routing Control Platform (RCP) es un intento de reducir las sesiones iBGP dentro del sistema. Para ello plantea la creación de una entidad centralizada con la que los routers del sistema mantendrán sesiones iBGP y comunicarán las rutas aprendidas por eBGP. Además de esto, se define para el servidor un módulo de aprendizaje de la topología (e.g, utilizando OSPF)[16][17].

Dado que esta entidad tiene una visión total de todos los mensajes BGP recibidos por la red y conoce la topología de la misma, puede computar el algoritmo de selección de rutas BGP para cada router, como si el mismo estuviera configurado en un esquema full mesh. Las rutas seleccionadas son comunicadas por iBGP al router[18][19], lo que lo diferencia de los RRs, quienes solo comunican a todos sus clientes las mejores rutas en base al conocimiento parcial de éstas, reduciendo su diversidad.

Si bien las conexiones con el servidor son realizadas a través de iBGP, en posteriores estudios se han realizado revisiones de la idea considerando SDN en la implementación[14].

Sección 3

Solución propuesta

3.1 Multicast sobre SDN

Nuestro diseño parte de la observación de que el motivo de la necesidad del full mesh iBGP es garantizar la visibilidad completa de todas las rutas comunicadas al AS. Por lo que se puede pensar el problema enfocándose en la manera cómo estas rutas son distribuidas.

Tomando un AS cualquiera, los puntos de entrada de estas rutas son los ASBRs del mismo y sería deseable que las mismas sean luego distribuidas a los demás routers del AS. En otras palabras, queremos distribuir paquetes desde un origen (uno de los ASBR) a un grupo específico de receptores (el resto de los routers del AS); esto es precisamente lo que hace Multicast [20].

Multicast funciona transmitiendo datagramas IP a un grupo de hosts identificados con una única dirección IP de destino. Los datagramas son entregados a todos los hosts miembros del grupo con el mismo mecanismo de mejor esfuerzo que utiliza un datagrama IP común.

La pertenencia o no de un host al grupo de Multicast es dinámica; el host puede unirse y/o dejar el grupo en cualquier momento, puede pertenecer a más de un grupo simultáneamente y no existen restricciones para el número de hosts miembros.

Los grupos Multicast pueden ser permanentes o temporales. Los primeros tienen asignadas IPs de destino bien definidas; los segundos en cambio, pueden escoger entre las direcciones no asignadas a grupos permanentes y existirá mientras haya miembros en el grupo.

Pensamos entonces que cada ASBR podría establecer un grupo de Multicast en el cual distribuir los mensajes BGP que él reciba, asegurándonos así que todos los routers del AS tengan una visibilidad completa de los mensajes que arriben. Esta idea de hecho fue esbozada en [21] aunque, hasta donde sabemos, no implementada.

Como se menciona en [22], mantener los árboles de distribución Multicast suma complejidad y aumenta el tráfico de la red, por lo que sería deseable desligar a los routers del AS de esta responsabilidad.

Es aquí donde el paradigma SDN resulta ideal para desacoplar estas tareas de control. Una controladora SDN, centralizada y con una visión completa de la topología del AS, podría calcular cada uno de los árboles de distribución y, utilizando el protocolo OpenFlow, instalar reglas de flujo acordes en cada router para retransmitir los paquetes de cada árbol. De esta forma, los routers no tendrán que determinar activamente como distribuir los paquetes de cada grupo. Existen trabajos hechos que exploran la viabilidad de utilizar SDN para administrar Multicast [22] [23] [24].

3.1.1 Desafíos

Detallaremos los desafíos más importantes que encontramos para implementar esta propuesta en un AS. De aquí en adelante, nos referiremos al mismo como AS experimental (EAS).

Eliminación del full mesh

La primer implicancia de esta propuesta es que el EAS eliminará por completo el full mesh de sesiones iBGP, reemplazándolo por un conjunto de árboles Multicast.

Ahora bien, esto también quiere decir que los mensajes BGP van a ser transportados utilizando el protocolo UDP en lugar de TCP. Esto, como es bien conocido, acarrea una serie de problemas, algunos de los cuales son:

- No se garantiza la entrega de los mensajes.
- El orden de entrega de los mensajes puede no corresponderse con el orden en que fueron enviados.
- No intenta recuperarse ante la detección de un error en el mensaje.

De todos estos, probablemente el más crítico sea la falta de garantías al entregar los mensajes, ya que si no podemos contar con ésta, nuestra solución potencialmente podría ser incorrecta. Por lo tanto debemos contar con algún mecanismo para garantizar la entrega de los mensajes BGP a través de todo el árbol.

Otra consecuencia importante de eliminar el full mesh refiere al comportamiento de BGP ante el cierre de estas sesiones. Como está especificado en [25], al cerrarse una sesión BGP, cada uno de los routers eliminará las rutas que aprendió de su *peer*; en consecuencia, debemos asegurarnos de mantener este comportamiento.

De estas consideraciones se desprende que, sin importar el software de ruteo que se utilice, éste deberá ser modificado para poder soportar estos cambios.

Cabe destacar que si bien a la fecha existen protocolos que implementan la distribución confiable de mensajes a múltiples receptores, tales como Pragmatic General Multicast (PGM) [26], SmartPGM, TIBCO Reliable Datagram Protocol (TRDP) y Reliable Multicast Transport Protocol (RMTP) [27], se verá en la siguiente sección que los mismos no cuentan con implementaciones estables integradas a los motores de ruteo open-source.

Árboles Multicast

Si bien más arriba ya mencionamos la idea de tener un árbol por ASBR, sería bueno comentar otras posibilidades que consideramos:

- Tener un único árbol Multicast.
- Tener un árbol Multicast por cada una de las sesiones eBGP que tenga cada ASBR.

La primera de estas opciones resulta claramente poco eficiente, ya que estaríamos sobrecargando el tráfico en los links que formen parte del árbol. Por otra parte, creemos que la segunda de estas opciones agrega complejidad a la solución sin redundar en un beneficio claro, además del hecho de que se necesitarán mas entradas en las tablas OpenFlow para soportar el mayor número de árboles. Por estos motivos consideramos más adecuado que cada ASBR tenga un árbol, sin importar cuántas sesiones eBGP haya establecido.

Habiendo decidido esto, debemos determinar cómo sabrán los routers del EAS cuáles son los grupos de Multicast por los que los mensajes BGP serán distribuidos. Dado que a priori se conoce cuáles de los routers del EAS son de borde, se podría asignar una dirección Multicast a cada uno de ellos y configurar a todos los routers para que se unan a los grupos de cada dirección. Este enfoque nos parece demasiado estático y consideramos que, al estar empleando el paradigma SDN, la tarea de asignar las direcciones Multicast a las que anunciará cada ASBR y notificar éstas al resto del EAS puede ser realizada por la controladora.

Visión de la topología

Debemos implementar también un mecanismo para que la aplicación de la controladora determine activamente la topología del EAS, para así determinar correctamente los árboles de distribución. Dicho mecanismo deberá ser capaz de responder rápidamente a los posibles cambios que haya en la topología, para así evitar que no se distribuyan paquetes por árboles potencialmente inválidos.

Por supuesto que los desafíos detallados previamente no son exhaustivos y existen otros problemas que deberían considerarse, pero hemos destacado los que nos parecen fundamentales para brindar una solución funcionalmente correcta y que cumpla con los objetivos que nos planteamos.

3.1.2 Diseño de alto nivel

Routers híbridos

Dado que pretendemos que la distribución a través de los árboles se haga mediante reglas de flujo OpenFlow, resulta evidente que los routers del EAS deben ejecutar software que implemente las funcionalidades de un switch OpenFlow [11].

Por otra parte, aún necesitamos ejecutar un software de ruteo IP que implemente el protocolo BGP, ya que no pretendemos cambiar la forma como los routers procesan los mensajes BGP, solo su recepción y distribución.

Se concluye entonces que los routers del EAS deben ser "híbridos", una idea que se encuentra prevista en la especificación de los switches OpenFlow [11].

Anuncios de los ASBR

Si bien decidimos que la controladora sea la que anuncie a los routers del EAS cuáles son las direcciones de Multicast a las que deben escuchar, ésta también debe saber cuáles son los ASBR; es por esto que decidimos que cada ASBR envíe un mensaje a la controladora (dentro de un Packet-In) para notificar a ésta de que él es un ASBR.

El contenido de este mensaje únicamente debe hacer saber a la controladora que el router que lo envía es un ASBR y que ha establecido una sesión eBGP; por esto decidimos que el contenido del mensaje fuera lo más simple posible, siendo tan solo el string *EXTERNAL_SESSION*.

La controladora, al recibir este mensaje, asignará una dirección Multicast al ASBR y, mediante un Packet-Out, enviará un mensaje a cada uno de los routers del EAS, conteniendo la dirección del grupo a la que éstos deberán unirse.

Hay que tener en cuenta que es posible que no todos los routers hayan establecido aún la conexión con la controladora, por lo tanto, ésta debe asegurarse de que cada vez que un router establece una conexión con ella, éste es notificado de todos los grupos a los que debe unirse.

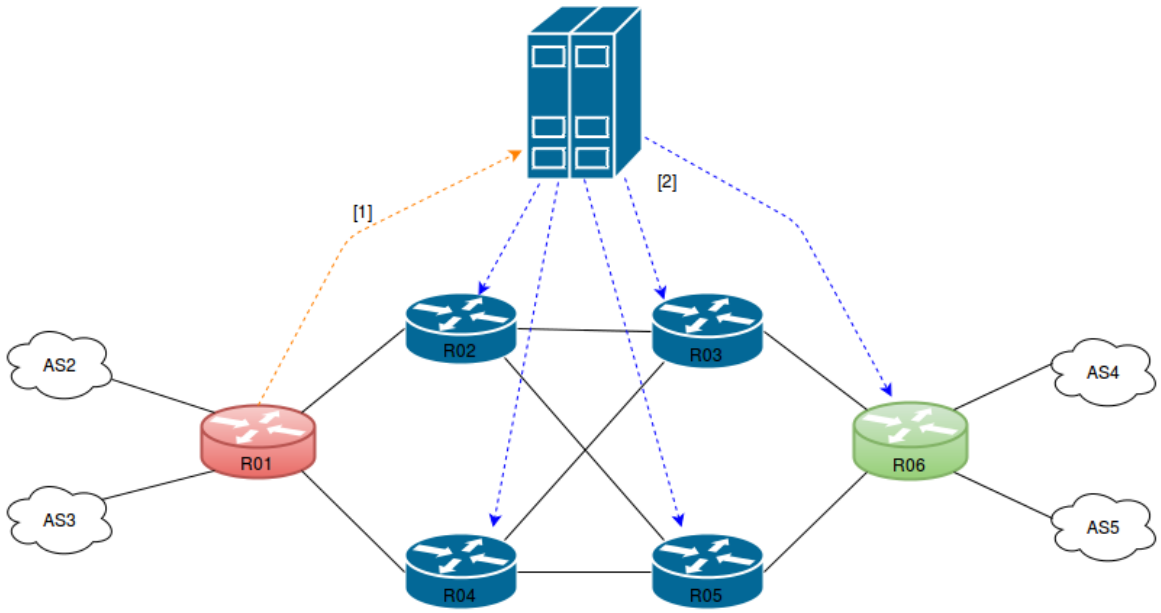

Fig. 3.1 (1) El ASBR envía un Packet-In notificando a la controladora de que es un ASBR. (2) La controladora envía un Packet-Out al resto de los routers con la dirección Multicast asignada al grupo del ASBR.

Descubrimiento de la topología

Dado que el EAS deberá ejecutar algún tipo de protocolo IGP, pensamos en usar OSPF en particular; la ventaja de usar este protocolo es que cada router dentro del EAS tendrá una visión completa de la topología, por lo que, si permitimos que la controladora intercambie mensajes OSPF con el resto de los routers, ésta también tendrá esa visibilidad.

Habiendo dicho esto, en este momento no descartamos que exista alguna controladora SDN que nos provea de alguna funcionalidad para descubrir la topología; pero al menos sabemos que si no es así, el método mencionado antes cumplirá esta función.

Sin importar de qué forma obtengamos la topología del EAS, es claro que hasta que nuestra aplicación no haya determinado que su visión de la misma es estable, i.e no varía en cierto lapso de tiempo, los árboles de distribución no pueden ser determinados y en consecuencia los mensajes BGP que lleguen de AS externos no pueden ser distribuidos.

Surge la pregunta entonces, ¿qué hacer con los mensajes BGP que lleguen a los ASBR mientras la topología no está estable? Inicialmente se podría pensar que cada ASBR almacene estos mensajes para luego, una vez armados los árboles, distribuirlos por el que corresponda. Sin embargo, no nos parece razonable utilizar recursos del ASBR para que actúe como una especie de "buffer" mientras los árboles no han sido generados; consideramos más escalable que estos mensajes sean almacenados en la controladora, que sin duda tendrá más recursos

para soportar el almacenamiento de éstos. Cuando la topología esté lista, la aplicación realizará las siguientes acciones:

1. Determinará el árbol de distribución de cada ASBR.
2. Instalará las reglas acordes en cada router para soportar la distribución a través de los árboles.
3. Enviará cualquier mensaje BGP Update que haya llegado al EAS mientras la topología no se encontraba estable.

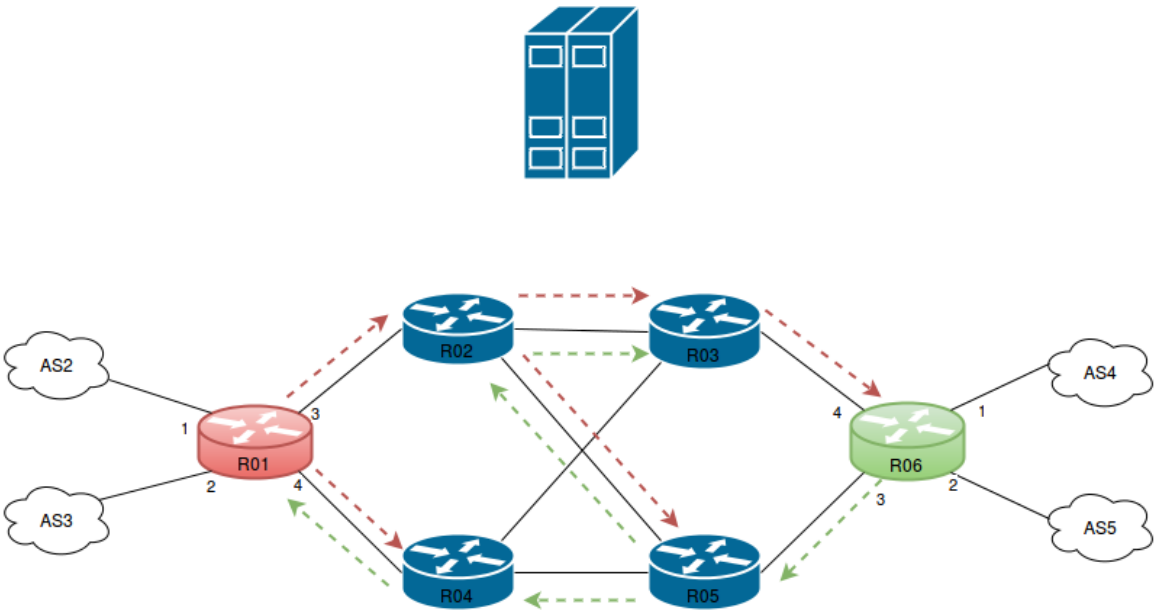

Fig. 3.2 Ejemplo de árboles de distribución para R_{01} y R_{06}

.

Decidimos entonces que los mensajes BGP sean enviados a la controladora mientras la topología no es estable; ahora bien, vale preguntarse si es esta la única situación en la que deban ser enviados a la controladora.

Si pensamos en el caso de un router nuevo que se suma al EAS, en una configuración full mesh este router establecería sesiones iBGP con el resto de los routers y recibiría la información que llegó al AS antes de que él haya entrado en funcionamiento. La única manera que vemos de replicar este comportamiento en nuestra solución, es que la controladora almacene los mensajes BGP que lleguen al EAS y, cuando un nuevo router establezca una conexión con la controladora, ésta envíe los mensajes que fueron previamente distribuidos por los distintos grupos Multicast. De esto se desprende que cada mensaje BGP Update que llegue al EAS será enviado a la controladora.

Vale aclarar también que la aparición de este nuevo router hará que la controladora considere a la topología como inestable y no se distribuirán paquetes hasta que vuelva al estado estable, momento en el cual se volverá a realizar el procedimiento descrito anteriormente.

Distribución de mensajes BGP

Como se mencionó en el punto anterior, todo mensaje BGP que llegue al EAS será enviado a la controladora; ésta, si la topología no se encuentra estable, guardará el paquete para ser distribuido cuando efectivamente lo esté. Debemos definir ahora como se distribuirán los mensajes BGP, ya sea porque llegó en un momento en que la topología se encontraba estable o porque lo hizo un tiempo antes de que se estabilizara.

Cuando un mensaje BGP es recibido por la controladora y la topología se encuentra en un estado estable, la controladora toma el mensaje BGP y encapsula a éste dentro de un segmento UDP, cuya dirección de destino es la dirección Multicast asignada al ASBR que recibió el mensaje originalmente. Este segmento UDP será entonces enviado al ASBR que recibió el mensaje BGP dentro de un Packet-Out, cuya única acción de salida sea el puerto reservado *OFPP_TABLE*, para que sea procesado por las Flow Tables.

Las reglas de flujo instaladas previamente en el ASBR (y en el resto de los routers) por la controladora son relativamente sencillas. El "matcheo" se realiza en base a la IP destino del paquete (la dirección Multicast) y la acción a ejecutar será reenviar este paquete por los puertos que están "linkeados" a los routers que son hijos en el árbol Multicast.

Para dejar esto más claro, si consideramos la figura 3.2, donde R_{01} y R_{06} son ASBR y sus direcciones Multicast asignadas son 239.192.1.1 y 239.192.1.2 respectivamente, las reglas de flujo en cada router serán:

- En R_{01}, si la dirección de destino es 239.192.1.1, el paquete será reenviado por los puertos 3 y 4.
- En R_{06}, si la dirección de destino es 239.192.1.2, el paquete será reenviado únicamente por el puerto 3.

Nótese además que éstas serán las únicas reglas que cada uno de estos routers tendrá, ya que R_{06} es hoja del árbol de R_{01} y viceversa.

Reglas análogas a éstas se encontrarán en los routers internos, variando únicamente en el puerto por el que reenviarán el paquete, garantizando entonces que todos los routers del EAS reciban el mensaje BGP.

Por supuesto que, debido a que los routers recibirán el mensaje BGP dentro de un segmento UDP, tendremos que modificar el software de ruteo para recibir y procesar adecuadamente el mensaje. Se hablará de esto más adelante cuando se detalle la implementación.

Eliminación de rutas por caída de sesión eBGP

Como ya fue mencionado, cada vez que una sesión eBGP se cierra, todas las rutas anunciadas por el *peer* del AS externo deben ser eliminadas. Para realizar esto, decidimos que los ASBR, al momento de determinar que la conexión se ha cerrado, notifiquen a la controladora de ésto enviando un mensaje dentro de un Packet-In. El contenido de este mensaje será un simple string con el siguiente formato:

- DEL#PEER_IP

Donde PEER_IP es la dirección IP del *peer* con el que el ASBR estableció la sesión eBGP.

Luego, de la misma forma que con los mensajes BGP, la controladora encapsulará este mensaje (tal cual fue recibido) dentro de un segmento UDP siendo su destino la dirección Multicast asignada al ASBR. El segmento es enviado al ASBR dentro de un Packet-Out para que sea procesado por las Flow Tables y reenviado a través del árbol.

Se modificará el software de ruteo para que acepte estos mensajes y, utilizando la IP enviada, elimine de sus tablas internas todas las rutas anunciadas por el *peer* externo.

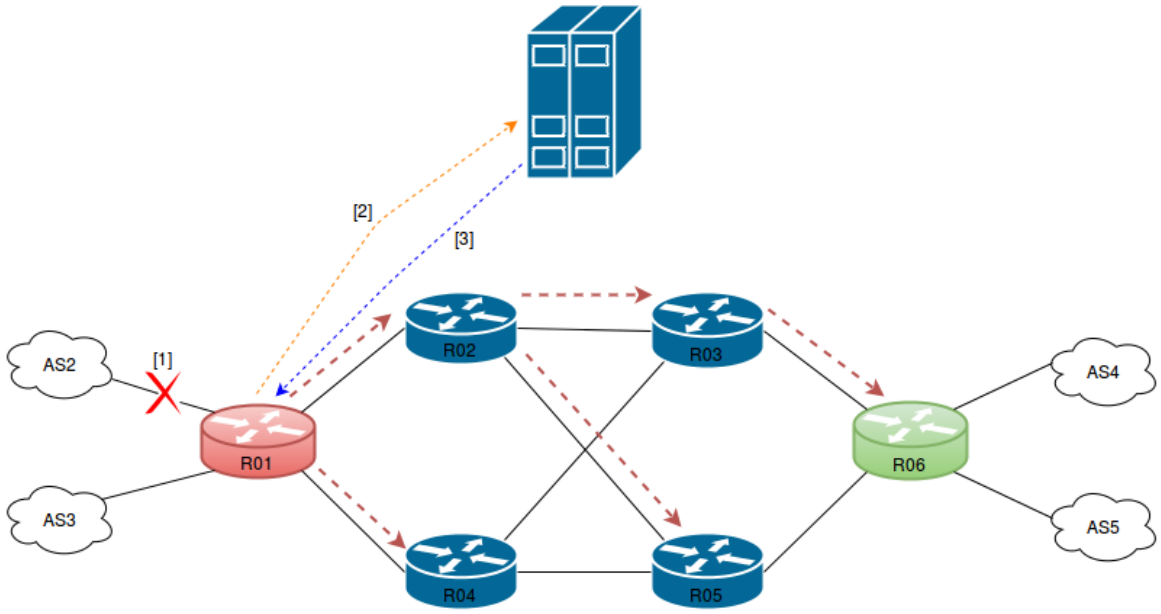

Fig. 3.3 (1) Se cierra la sesión eBGP. (2) El ASBR notifica con un Packet-In a la controladora del cierre. (3) La controladora envía un Packet-Out para distribuir la orden de borrado por el EAS.

Eliminación de rutas por caída de ASBR

Existe también la posibilidad de que por algún motivo un ASBR deje de funcionar o sea dado de baja; en ese caso, todas las sesiones eBGP que estaban activas en ese momento se cerrarán, por lo que también se deben eliminar las rutas que fueron anunciadas por todos sus *peers*.

Sin embargo, dado que el ASBR no está en funcionamiento, éste no puede notificar a la controladora de que las sesiones eBGP se han cerrado. Pero, por otra parte, la controladora debería ser capaz de identificar que este router ha fallado, ya sea porque la conexión con el router se ha cerrado o porque identifique que la topología ha cambiado, por lo que está en condiciones de notificar a los demás routers del cierre de las sesiones.

A diferencia del caso anterior, la cantidad de IPs que debe incluir el mensaje puede ser más de una, por lo que el formato del mensaje será en este caso:

- DEL#PEER_IP[#PEER_IP]*

Nótese que la controladora puede saber cuáles son las IPs de los *peers* del ASBR caído dado que éste ha tenido que enviarle todos los mensajes que ha recibido de sus *peers*; la controladora puede inspeccionar estos mensajes para determinar las distintas IPs de los routers externos.

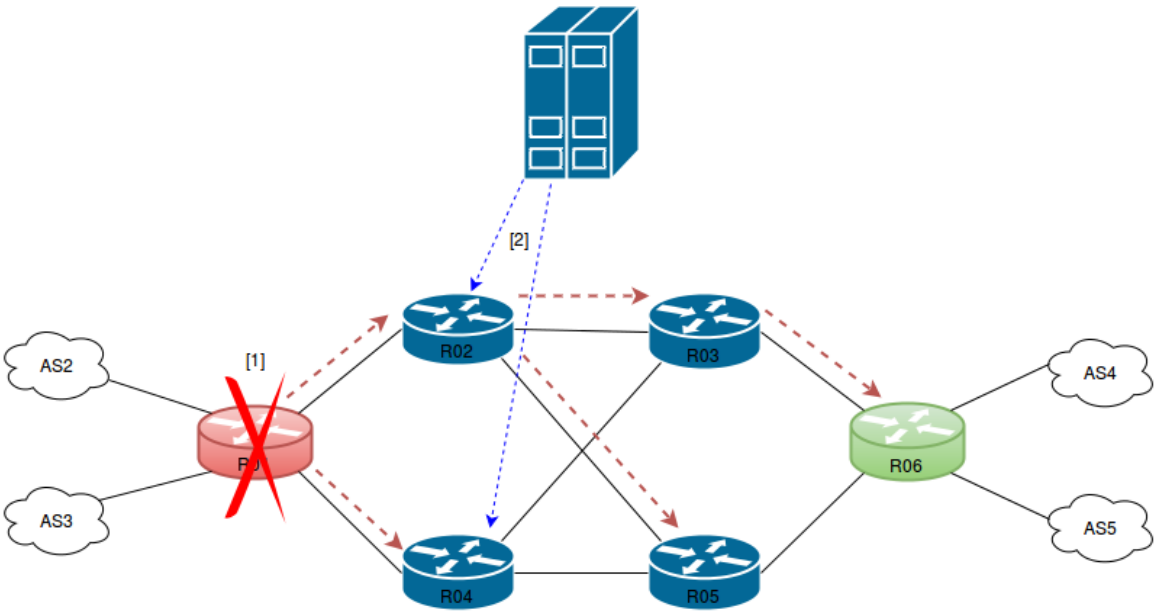

Fig. 3.4 (1) El ASBR deja de funcionar. (2) La controladora envía la notificación de borrado en un Packet-Out a los hijos directos en el árbol del ASBR

Otra diferencia con el caso anterior es que no podemos mandar el mensaje directamente a la raíz del árbol ya que el ASBR no está en funcionamiento; para resolver esto y además

utilizar las reglas ya instaladas en los demás routers del EAS, la controladora enviará un Packet-Out a cada uno de los routers que sean hijos directos en el árbol de distribución del ASBR. De esta forma aseguramos que el mensaje sea distribuido por las distintas ramas del árbol.

Confiabilidad

Para garantizar la entrega tanto de los mensajes BGP como de los mensajes de borrado de rutas, consideramos que la mejor forma de hacerlo es modificar el software de ruteo de los routers para que éstos envíen confirmaciones a la controladora sobre la recepción de los mensajes.

Cuando la controladora envíe un Packet-Out conteniendo un mensaje a ser distribuido por alguno de los árboles, generará un hash del mensaje enviado e inicializará una serie de *timers* para cada uno de los routers del EAS.

Los routers del EAS, luego de recibir y procesar el segmento UDP, enviarán dentro de un Packet-In un mensaje a la controladora con el siguiente formato:

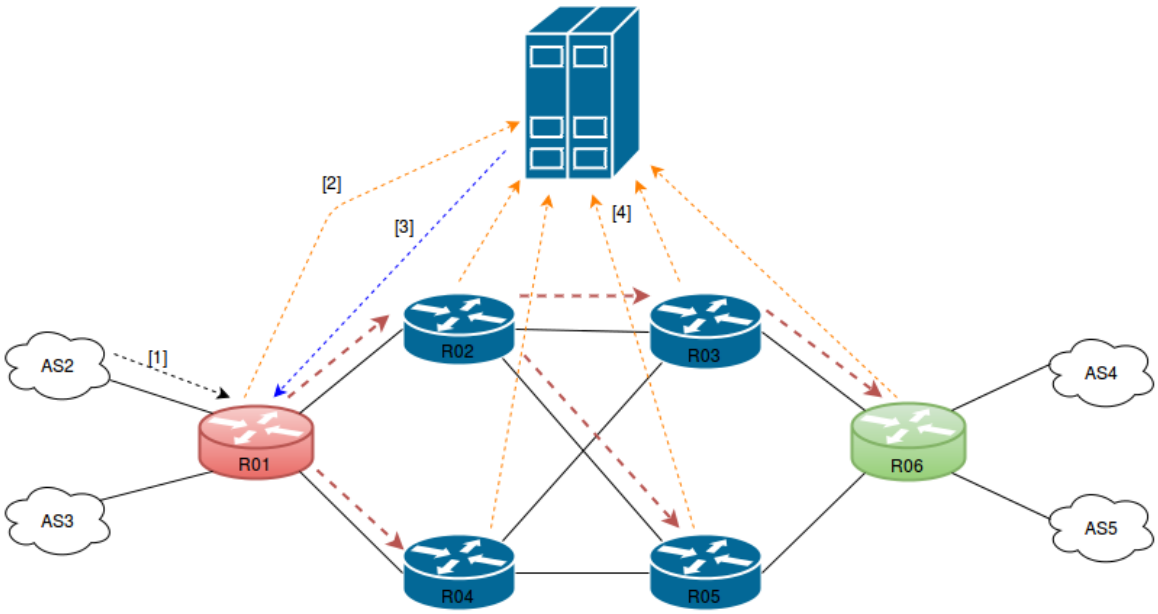

Fig. 3.5 (1) Router externo envía BGP Update. (2) ASBR envía un Packet-In con el mensaje BGP a la controladora. (3) La controladora envía un Packet-Out con el mensaje BGP para que se distribuya por el árbol. (4) Cada router del EAS confirma la recepción enviando un Packet-In.

- ACK#Multicast_IP#HASH

Donde "Multicast_IP" es la dirección IP del grupo Multicast y "HASH" es el hash del paquete que se quiere confirmar.

El hash es claramente necesario para que la controladora pueda saber cuál es el mensaje que un router está confirmando; la dirección IP del grupo en cambio, si bien no es absolutamente requerida para identificar el mensaje, creemos que será de utilidad para que la controladora pueda identificarlo con mayor facilidad.

La controladora, al recibir el mensaje de confirmación, identificará qué mensaje se confirma de acuerdo al hash enviado por el router y, de ser correcto, cancelará el *timer* asignado al router para ese mensaje en particular. Nótese que la controladora sabe qué router envió la confirmación debido a la conexión establecida con él.

Si la controladora no recibe un mensaje de confirmación en el tiempo establecido para el *timer* asignado a un router, asumirá que el paquete no fue recibido por éste y enviará un nuevo Packet-Out específicamente a dicho router.

Vale mencionar que el tiempo de *timeout* que será asignado al *timer* de cada router, variará según el nivel de profundidad en que éste se encuentre en el árbol de distribución. Observando la figura 3.6, si suponemos que el router R_{02} no recibe el mensaje enviado por R_{01}, resulta evidente que tampoco lo recibirán ninguno de sus descendientes, a saber: R_{03}, R_{05} y R_{06}; si todos los tiempos de *timeout* fueran iguales, la controladora enviaría un Packet-Out no solamente a R_2, sino a todos sus descendientes, lo cual sería innecesario ya que, al enviar el Packet-Out a R_2, las reglas de flujo OpenFlow se encargarían de reenviar el paquete a sus descendientes.

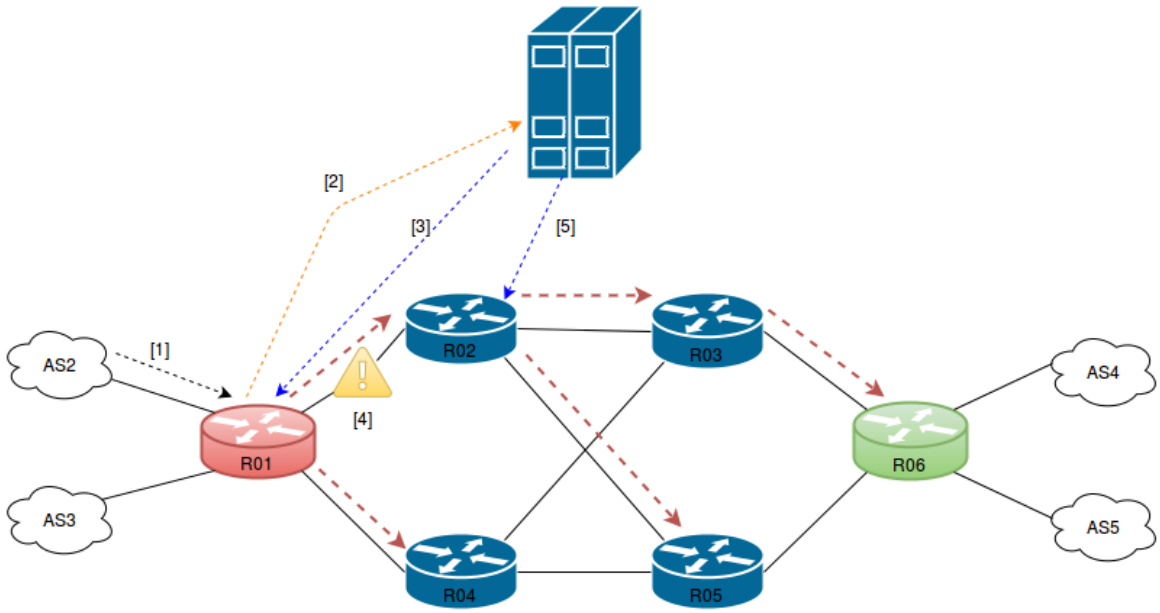

Fig. 3.6 (1) Router externo envía BGP Update. (2) ASBR envía un Packet-In con el mensaje BGP a la controladora. (3) La controladora envía un Packet-Out con el mensaje BGP para que se distribuya por el árbol. (4) El segmento UDP no es entregado. (5) La controladora envía un Packet-Out con el mensaje BGP al router que no confirmó la recepción.

Para evitar este problema y evitar que ocurran *timeouts* innecesarios, definimos el tiempo de *timeout* en un árbol de distribución para cada router R_i de la siguiente forma:

$$T_{R_i} = T_{BASE} + N_{R_i} * T_{DELAY\_DE\_NIVEL}$$

Donde:

- T_{BASE} es el tiempo base de espera para la recepción de las confirmaciones
- $T_{DELAY\_DE\_NIVEL}$ es un tiempo de demora agregado entre cada nivel del árbol
- N_{R_i} es el nivel de profundidad del router R_i dentro del árbol que se esté considerando

De esta forma, consideramos que evitaremos el problema mencionado anteriormente. Ambos tiempos se definirán durante la implementación.

Sección 4

Evaluación de Herramientas

4.1 Controladoras

A la fecha se disponen de distintas opciones a la hora de desarrollar este tipo de sistemas. Entre las opciones propietarias se encuentran: Cisco Application Policy Infrastructure Controller (APIC)[28], HP Virtual Application Networks (VAN)[29], NEC ProgrammableFlow PF6800[30], entre otros. Del lado de las opciones gratuitas y open-source están: NOX/POX, Beacon, RYU, FloodLight y Trema. Dado que analizar y comparar todos los controladores disponibles llevaría demasiado tiempo, centraremos nuestra atención en algunas de las alternativas open-source gratuitas que consideramos de interés para el desarrollo del proyecto. De esta forma tendremos información suficiente para definir cual de las mismas utilizar para la implementación de prototipos.

4.1.1 NOX/POX

Tiene soporte para Openflow v1.0, NOX provee un entorno para la ejecución de aplicaciones SDN programadas en C++. Por su parte POX, que es en cierta forma su sucesor, provee una interfaz para su programación utilizando Python. Las comunidades de esta controladora se encuentran a la fecha poco activas, y los repositorios donde almacenan el código no reciben actualizaciones hace más de tres años. Por otro lado en su sitio web existen muchas secciones no disponibles, lo que hace que gran parte de la documentación no sea accesible. NOX brinda un entorno de programación *multi-thread* mientras que en POX el entorno es *single-thread*. Como características, tiene disponible funcionalidades avanzadas que ayudan al desarrollo de las aplicaciones como ser descubrimiento de topología, entre otras.

4.1.2 Beacon

Da soporte a programación orientada a eventos en un entorno *multi-thread* en Java, trabajando sobre el framework Spring. Este proyecto actualmente no se encuentra muy activo y ha sido sobrepasado por las funcionalidades de otros controladores más modernos. Su desarrollo, al igual que NOX, se encuentra detenido desde hace más de tres años y sus foros no presenten actividad por el mismo período de tiempo.

4.1.3 Trema

Desarrollado teniendo como filosofía facilitar la escritura de código y aumentar la performance, esta controladora combina Ruby (a fin de facilitar la escritura) y C (a fin de lograr mayor performance). Actualmente brinda soporte para OpenFlow v1.3.0 y v1.3.1; no soporta la versión v1.0. Si bien su comunidad se encuentra activa y en su sitio web se dispone de documentación sobre su arquitectura y la API de programación, no destacan demasiados desarrollos realizados con la misma y los ejemplos expuestos en su sitio son muy básicos como para poder tener una idea de su verdadero potencial, sin contar el hecho de que hay parte de la documentación disponible únicamente en idioma japonés.

4.1.4 FloodLight

El desarrollo de este proyecto está contenido dentro de una suite que incluye Floodlight Controller, Indigo, LoxiGen y OFTest. Basado en su versión comercial Big Network Controller (BNC), propone un entorno de desarrollo de módulos y aplicaciones SDN utilizando Java (mediante una Java API) o cualquier otro lenguaje utilizando una REST API que permite interoperar con la controladora. El entorno de ejecución brindado es *muti-thread*, basado en Java y el framework Netty. A la fecha soporta OpenFlow v1.0, v1.3 y v1.4.

Su arquitectura está compuesta por un conjunto de módulos donde cada uno de ellos provee de funcionalidades a otros módulos más complejos y a las aplicaciones desarrolladas. Sus múltiples módulos proveen nativamente gran cantidad de funcionalidades, como ser el descubrimiento de topología.

Su sitio web brinda acceso a toda su documentación y a ejemplos de uso de los distintos módulos, así como detalla algunas de las restricciones conocidas a la fecha.

4.1.5 Ryu

Controladora basada en componentes y programada en Python. Tiene un conjunto predefinido de componentes integrados internamente; éstos pueden ser modificados, extendidos

o combinados a fin de construir distintos tipos de aplicaciones en la controladora. Al igual que FloodLigth, puede utilizarse cualquier lenguaje para programar aplicaciones sobre la misma, utilizando la API provista, pero los componentes internos están desarrollados sobre Python. Si bien el entorno de ejecución es *thread*, su paradigma de programación está basado en programación orientada a eventos, utilizando la biblioteca gevent[31]. Al igual que el resto, cuenta con módulos que le proveen de múltiples funcionalidades, entre ellas el descubrimiento de topología.

A la fecha es una de las pocas controladoras que da soporte a casi toda la gama de versiones de OpenFlow soportando en el momento desde la versión 1.0 a la 1.5.

4.1.6 Comparación

Existen al día de hoy una gran cantidad de estudios comparando las distintas controladoras en base a múltiples parámetros y tomando como variables de referencia distintos elementos de las mismas. El estudio [32] cataloga a Ryu como la mejor opción teniendo en cuenta disponibilidad de interfaz gráfica, soporte a API REST, lenguajes de programación soportados, performance, soporte *virtual switching*, madurez y versiones de OpenFlow soportadas, entre otras. Parte de este estudio identifica las distintas interfaces de comunicación y componentes de las controladoras, como puede observarse en Fig. 4.1.

Por otro lado [33] define un conjunto de índices (performance, escalabilidad, seguridad y confiabilidad) mediante los cuales concluye que la mayoría de las controladoras analizadas (Ryu, Beacon, NOX/POX, FloodLight, MUL y Maestro) son capaces de soportar una carga media de trabajo, pero el que obtuvo mejor resultado fue Beacon. También identifica que, en el momento del estudio, las controladoras tienen fallas de seguridad que pueden ser explotadas por distintos tipos de malware.

Siendo críticos con estos estudios, en los mismos hay algunos detalles que deben ser tenidos en cuenta:

- no quedan del todo justificadas las razones para considerar como positiva o negativa una funcionalidad de la controladora.
- la cantidad de interfaces y switches que puede soportar cada controladora no es estudiada, por lo que parámetros como escalabilidad no son del todo tenidos en cuenta.
- las NorthBound APIs así como los protocolos soportados por la SouthBound API no son analizadas en demasiada profundidad.
- no se realizan comparaciones utilizando los paradigmas de controladora activa/reactiva, lo que podría suponer alguna diferencia en los resultados.

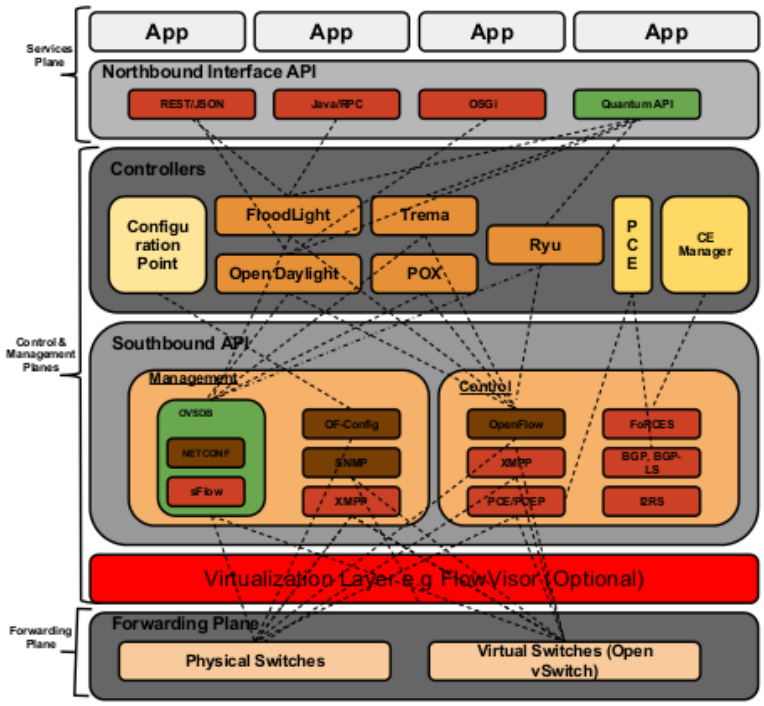

Fig. 4.1 Comparación de controladoras SDN y sus interfaces

- los índices y casos utilizados en los estudios no siempre coinciden, por lo que la comparación entre estudios es muy subjetiva.

De todos modos, es entendible que todos estos estudios enfoquen su atención en las funcionalidades en lugar de otros aspectos, ya que las funcionalidades serán las que en principio determinen los casos de uso donde puedan ser utilizadas. Podemos decir entonces que si bien estos estudios permiten tener un primer punto de partida para quien desee elegir una plataforma de desarrollo, los mismos no tienen en cuenta muchos factores, por los que no se los puede considerar totalmente concluyentes y la elección de cada plataforma queda liberada a las necesidades propias de cada proyecto.

Sin embargo, entre las opciones analizadas, Ryu parece ser la opción que más se adecua a nuestras necesidades ya que, además de contar con una enorme comunidad impulsando su desarrollo, es uno de los entornos mejor documentados teniendo ejemplos y *plugins* que facilitan enormemente el trabajo con el mismo. Adicionalmente, en los anteriormente mencionados estudios comparativos con necesidades similares a las nuestras ha sido considerado una de las mejores opciones.

Como se verá en en las siguientes secciones, Ryu se integra muy fácilmente al entorno de simulación elegido para llevar a cabo la implementación de prototipos, y nos permite

llevar adelante el desarrollo de manera más sencilla, desvinculándonos de preocupaciones adicionales como interoperabilidad entre distintos lenguajes de programación o consumo extras de recursos al necesitar contar con múltiples entornos de ejecución (e.g una máquina virtual Java y entorno de ejecución Python o Ruby).

4.2 Software de ruteo

Existe una gran variedad de alternativas a la hora de querer implementar distintos protocolos de ruteo sobre una red, muchas de ellas propietarias (como CISCO o JUNIPER) así como muchas open-source. Sobre estas últimas se centra la atención, ya que las mismas nos permiten realizar las modificaciones necesarias para realizar el prototipo. Entre las más populares opciones open-source destacan principalmente Quagga Routing Suite y Bird Internet Routing Daemon

También existen otras menos utilizadas o más específicas, tales como OpenBGPD, XORP y FreeRouting, las cuales inicialmente se descartan como posibles bases para el desarrollo del proyecto por contar con implementaciones parciales de los protocolos, su falta de mantenimiento por la comunidad desde hace mucho tiempo, así como carencias a la hora de querer combinar distintos protocolos; por ejemplo, OpenBGPD sólo ejecuta BGP, por lo que para poder utilizar otros protocolos como OSPF sería necesario agregar software extra que lo complemente.

En la siguientes secciones se centra el estudio sobre las opciones consideradas más viables para el desarrollo del proyecto, sus principales características y diferencias.

4.2.1 Quagga Routing Suite

Es el más popular de los software de routing open-source a la fecha, dando soporte a los protocolos RIPv1, RIPv2, RIPng, Babel, OSPFv2, OSPFv3 y BGPv4 [34].

Debido a que lleva muchos años en el mercado, la documentación y ejemplos de uso del mismo es abundante. Esto además se ve complementado por la interfaz y comandos de configuración que utiliza, los cuales son muy similares a los presentes en routers de CISCO, abriendo la puerta de ese modo a una gran cantidad de documentación extra (trasladable casi que directamente). A pesar de llevar varios años en desarrollo y dar soporte a múltiples protocolos, en su lista de futuras funcionalidades a implementar no figuran ninguno de los protocolos de distribución confiable mencionados en la sección anterior.

Implementado en C, su arquitectura fue desarrollada teniendo en mente la utilización de hilos para la gestión de los distintos eventos a dar respuesta (en todos aquellos sistemas

que fuera posible). Sin embargo, debido a que las librerías necesarias para implementar la gestión de los hilos presentan algunos inconvenientes, se utilizan los hilos de forma parcial, complementándose en su lugar con el uso de la llamada al sistema SELECT (en el caso de BGP se crea un thread por cada vecino configurado, a fin de controlar la máquina de estados del protocolo para ese vecino).

La ejecución de Quagga implica inicializar un demonio por cada protocolo y cada uno de ellos inicializará hilos acorde a sus necesidades. El demonio Zebra siempre debe ser inicializado y es el responsable de realizar la comunicación entre las rutas aprendidas por los protocolos y la tabla de ruteo del Kernel.

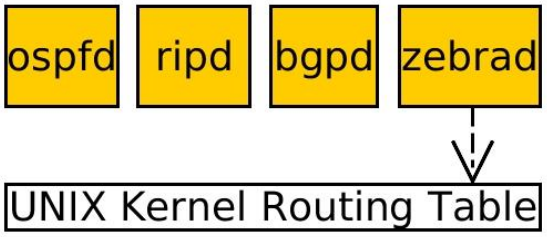

Fig. 4.2 Esquema de funcionamiento de Quagga Routing Suite

Cada demonio es configurado desde un archivo propio (tanto IPv4 como IPv6 son configurados y manejados por el mismo demonio). El acceso a la información de los mismos puede ser realizada mediante telnet una vez inicializados. Dado que esto es bastante engorroso, también se puede acceder e interactuar con los mismos mediante la interfaz vtysh/vty.

En caso de ser necesario realizar algún cambio en los protocolos, se debe editar los archivos y reiniciar el router o demonio que responde al mismo, lo cual puede producir en general efectos no deseados (e.g, períodos de no disponibilidad durante el reinicio).

Su código sigue en general el estándar GNU para desarrollo de software con algunas pequeñas variantes. Si bien esto es bueno, su código es bastante complejo y presenta varias secciones ambiguas y difíciles de interpretar, siendo necesario muchas veces analizar y modificar muchos módulos para lograr extender alguna de las funcionalidades implementadas.

No es raro encontrar zonas con código comentado o no vigente, conteniendo funciones, principalmente experimentales, sin documentar.

4.2.2 BIRD Internet Routing Daemon

Es un motor de routing con menos historia que Quagga, con lo cual muchas funcionalidades avanzadas presente en este último aún no están desarrolladas. Sin embargo, ha ido ganando

popularidad en los últimos años. A la fecha cuenta con implementaciones para los protocolos RIPv2, OSPFv2, OSPFv3 y BGPv4 [35]. Cabe destacar que en su lista de futuras mejoras, a la fecha, aún no aparece la incorporación de los protocolos de distribución confiable mencionados en la sección anterior sino que las mismas se centran mayormente en mejorar el soporte a los protocolos ya implementados.

A pesar de ser más reciente que Quagga, cuenta con una gran cantidad de documentación sobre su uso y su estructura de código en el sitio oficial del proyecto, así como también con varios ejemplos de uso y traducciones de comandos en sintaxis Quagga a Bird. Sin embargo, muchas funcionalidades no están del todo documentadas, y los ejemplos no terminan de ser claros sobre cómo utilizar algunas funciones (por ejemplo el claro manejo de filtros). Esto puede ser un problema en varias ocasiones, ya que la configuración de los distintos protocolos es propia del software de ruteo (utiliza archivos en formato similar a JSON para definir las distintas opciones a tener en cuenta al correr cada protocolo).

Implementado en C, su arquitectura está desarrollada pensando en eventos, descartando el uso de hilos para el manejo de los mismos, ya que el *overhead* generado por su creación y posterior manejo se considera fuertemente negativo para la aplicación. Es por esto que para dar respuesta a los distintos eventos, implementa un planificador propio encargado de la gestión de los mismos. Siguiendo esta línea de desarrollo también se reducen las llamadas al sistema operativo, haciéndolas solo cuando es estrictamente necesario.

El código se encuentra dividido en varios módulos bien diferenciados, cada uno responsable de ejecutar las distintas particularidades de cada protocolo; las acciones y flujo de acciones son manejadas dentro del código por funciones llamadas *hooks*, las cuales pueden alterar o extenderse fácilmente.

El funcionamiento general del motor de ruteo consiste en una tabla primaria donde se almacenan todas las rutas aprendidas a través de los protocolos y, opcionalmente, varias tablas secundarias que pueden definirse para uso en algunos protocolos. Cada protocolo definido cuenta con dos filtros, uno de importación y otro de exportación; esto hace referencia a las rutas aprendidas que deben ser "importadas" a la tabla primaria y las que deben ser exportadas (i.e anunciadas) a los vecinos respectivamente. La configuración de estos filtros es definida en la configuración del protocolo.

La estructura utilizada para la configuración responde a un único archivo de configuración por instancia de Bird y, a diferencia de Quagga, todos los protocolos corren sobre un único proceso (en realidad son dos procesos, uno para IPv4 y otro para IPv6, en caso de darle soporte). El archivo de configuración contiene soporte para un lenguaje básico de *scripting* en el cual se pueden definir reglas semi-dinámicas para los filtros, lo que brinda una mayor flexibilidad a la hora de definir qué rutas anunciar o aceptar de los vecinos.

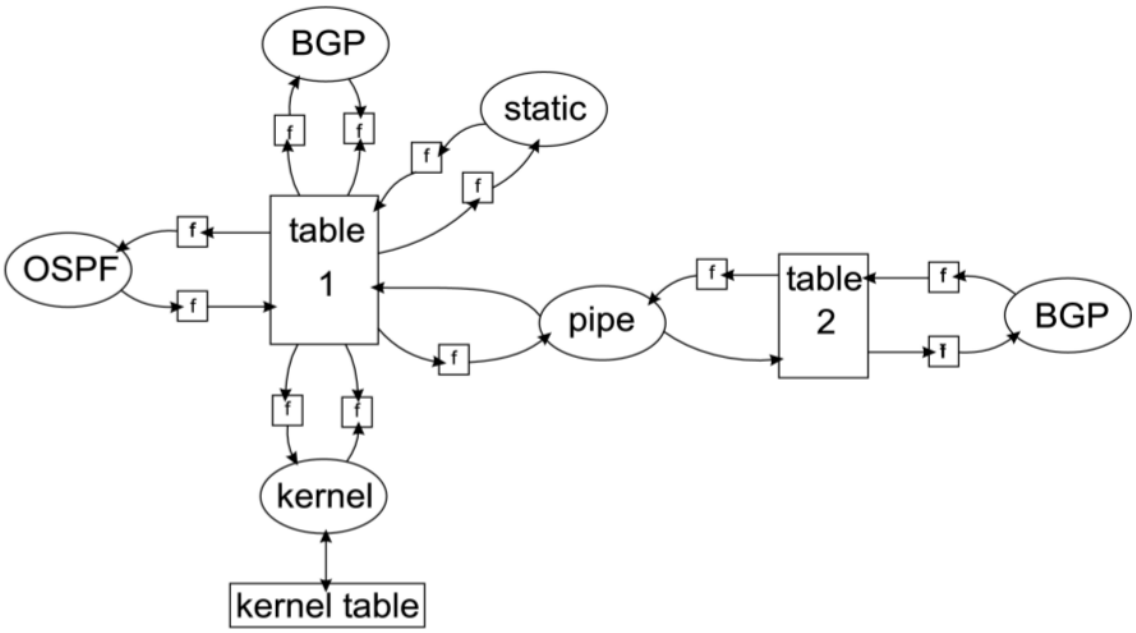

Fig. 4.3 Esquema de funcionamiento de Bird Internet Routing Daemon

La interacción con los distintos protocolos se realiza a través de un cliente que se mantiene como una implementación independiente al motor de ruteo. Este cliente permite acceder a la información de configuración y tablas de ruteo asociadas a cada protocolo en tiempo real, así como ejecutar algunos comandos específicos a éstos (por ejemplo, reinicios). Cabe destacar que algunos protocolos cuentan con la opción de reinicio sin desechar las tablas armadas al momento de cerrar las sesiones activas mientras sea posible (*graceful restart*).

Si bien cumple con la mayoría de las funcionalidades básicas, carece un poco de la flexibilidad presente en vtysh, ya que no permite escribir el archivo de configuración dinámicamente desde el mismo.

4.2.3 Comparación

A la fecha existen varios estudios que demuestran una mejor gestión de memoria y mejores tiempos de respuesta por parte de Bird frente a otros motores de ruteo como Quagga.

Tras realizar pruebas con ambos, optamos por usar como motor de ruteo base a Bird, principalmente debido a su implementación modular y "prolijidad" en el código, así como los distintos estudios sobre su eficiencia.

4.3 Entornos de simulación

En esta primera parte centraremos la atención en discutir las distintas alternativas disponibles para la ejecución y prueba de prototipos, para ello será necesario analizar algunas de las técnicas de virtualización existentes al momento. En las siguientes secciones entraremos en detalles sobre las mismas.

El uso de hardware real para prototipar es inicialmente descartado ya que, además de ser un enfoque caro y del cual no disponemos, resulta engorroso la realización de pruebas sobre el mismo.

Para usar las alternativas de virtualización resulta conveniente distinguir la diferencia entre emuladores y simuladores de red. Por un lado los simuladores intentan brindar un entorno donde lo observado muestre un comportamiento lo más parecido posible en términos de performance a lo que se observaría si se implementara el modelo en hardware. Esto permite fijar, controlar y medir distintas variables dentro del sistema simulado como latencia o pérdida de paquetes.

Su implementación es llevada a cabo en general a nivel de procesos, siendo cada proceso una entidad simulada y el conjunto de los mismos una máquina virtual.

Por otro lado, los emuladores se centran en brindar funcionalidades como protocolos, arquitecturas y configuraciones, dejando de lado la performance de los entornos creados.

La simulación en general tiene como contrapartida su difícil traslado a hardware real o su replicación por parte de otros equipos de investigación que quieran replicar el trabajo realizado. Esto plantea un problema a la hora de querer compartir las tareas realizadas. El enfoque de emulación por el contrario plantea facilidades en ambos aspectos.

La emulación presenta múltiples variantes:

Full Stack Virtualization Los distintos componentes a emular ejecutan su propia versión del sistema operativo y procesos, gestionando sus recursos como si fueran propios; todo esto corriendo sobre el sistema residente que garantizará el acceso al hardware real del equipo. Si bien se logra aislar la ejecución de los elementos, presenta problemas de escalabilidad pues el consumo de recursos de cada máquina virtual es elevado.

Lightweight Virtualization también llamada virtualización a nivel de sistema operativo, se diferencia a la anterior en que los componentes a emular comparten un mismo Kernel sobre el cual, mediante algún mecanismo de aislamiento (e.g. LXC), ejecutan los distintos procesos que sea necesario emular de forma independiente y aislada. Dado que el Kernel compartido entre los componentes es el mismo que ejecuta el host, se pierde la posibilidad de correr múltiples sistemas operativos. Sin embargo presenta algunas ventajas en la cantidad de recursos necesarios para su funcionamiento.

Puesto que nuestro objetivo es poder realizar un prototipo funcional del diseño planteado y estudiar su comportamiento (eventualmente trasladándolo a hardware real) el enfoque de emulación nos resulta más conveniente. En la siguiente sección se analizan brevemente dos de las alternativas open-source más conocidas a la fecha.

4.3.1 Netkit

Es un emulador de redes, cuyo esquema de virtualización se basa en el denominado User Mode Linux (UML). Éste consiste en un Kernel incluido dentro del sistema operativo Linux que puede ser ejecutado como si de un proceso se tratase, dentro de un contenedor Linux (Linux Box). Su configuración y puesta en funcionamiento se realiza mayormente mediante scripts (bash). En este esquema de virtualización, se denomina virtual host a un proceso corriendo en modo UML, mientras que el Linux Box que lo contiene es llamado host.

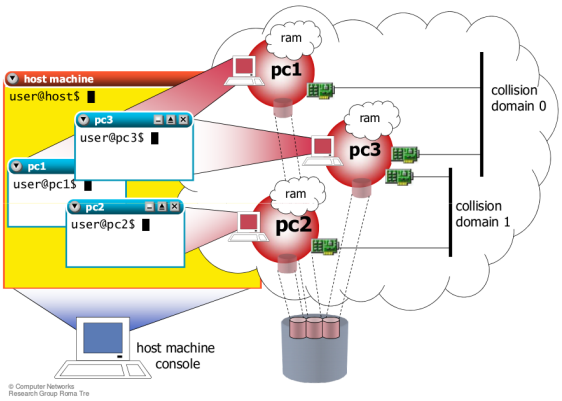

Fig. 4.4 Esquema de funcionamiento de Netkit

El espacio en disco de cada máquina virtual es mapeado a un archivo dentro del sistema operativo residente, mientras que la memoria RAM es reservada directamente por cada una de las mismas, como se ve en Fig. 4.4. Por otro lado las interfaces y sus conexiones son simuladas utilizando dominios de colisión virtuales, los cuales pueden o no estar conectados a las interfaces reales del host.

Según la documentación oficial, son requeridos 10 MB de memoria y 600MB (más un adicional de 1MB a 20MB) de espacio en disco por máquina virtual. Este emulador es mantenido a la fecha por una pequeña comunidad y no recibe actualizaciones desde el año 2011. Actualmente no cuenta con un soporte directo a OpenFlow pero dado su esquema de configuraciones mediante scripts la extensión para poder integrarlo con el mismo no debería resultar compleja.

4.3.2 Mininet

Es un emulador de redes con una API de programación basada en Python. La virtualización en este caso es realizada a nivel de procesos (virtualización ligera a nivel de sistema operativo) y network namespaces.

Su característica principal es la flexibilidad que brinda su API a la hora de generar entornos de pruebas o extender las mismas funcionalidades del emulador, además de la pequeña cantidad de recursos consumidos para su funcionamiento. En la web del proyecto puede encontrarse una gran cantidad de documentación y tutoriales. Otra característica no menor es la facilidad que existe en el traslado de lo emulado a hardware real.

La principal desventaja de este entorno es que hace difícil poder medir la performance a nivel de red. Por otro lado cabe destacar que se cuenta con un solo Kernel disponible para ejecutar, compartido por todos los elementos de la red (frente a Netkit que tiene uno por máquina virtual).

Dada la forma en que se realiza la virtualización en este emulador, el entorno sólo funciona en Linux, lo cual puede resultar un inconveniente si se desea realizar implementaciones sobre otros sistemas operativos.

MiniNext

Es una extensión desarrollada para Mininet que habilita la ejecución de Quagga Routing Suite u otro servicio dentro de los host del entorno simulado. Además de esto, su API habilita la creación de directorios privados para cada host emulado; esto último, si bien en la documentación se especifica que es brindado nativamente por Mininet, en la práctica no es posible realizarlo según explica la documentación, por lo que, la incorporación de esta extensión resulta importante.

Comparación

Ambos emuladores cuentan con los elementos suficientes para llevar a cabo el desarrollo como ya se ha discutido a lo largo de su análisis.

Optamos por utilizar Mininet+MiniNext ya que nos brinda un entorno de emulación programable mas flexible y ágil que Netkit consumiendo una menor cantidad de recursos (lo que permitirá simular redes mas complejas)[36]. Además de esto, Mininet está diseñado pensando en SDN, por lo que contempla la emulación de todos los componentes que conforman el entorno. Netkit por su parte no cuenta con soporte para SDN nativamente, por lo que sería necesario centrarse en la emulación del mismo.

Sección 5

Implementación

Habiendo elegido las herramientas que consideramos más adecuadas para llevar a cabo nuestra propuesta, describiremos ahora los detalles de su implementación, comenzando por la construcción del entrono de simulación para luego centrarnos en los cambios realizados en el software de ruteo y la aplicación SDN desarrollada sobre el controlador RYU.

5.1 Entorno de trabajo

La operabilidad entre RYU y Mininet esta garantizada desde el comienzo ya que además de utilizar las mismas tecnologías, en la documentación de Mininet tanto como en la de RYU se puede encontrar una sección dedicada a configurar esta controladora en el entorno de simulación.

5.1.1 Router Híbrido

La construcción de un router híbrido es realizada utilizando las técnicas del proyecto "OSHI - Open Source Hybrid IP/SDN" [37]. La misma, propone utilizar la funcionalidad de puertos virtuales disponible en OpenvSwitch para crear tantos puertos virtuales como puertos físicos haya disponibles, creando entre ellos una correspondencia uno a uno. Posteriormente se conectan los puertos físicos y virtuales al software del dispositivo SDN y se configura el software de ruteo para que utilice los puertos virtuales a modo de interfaz como si de las físicas se tratara. De esta forma, se proceden a instalar reglas de baja prioridad en la Flow Table 0 del dispositivo SDN, para que todos los flujos IP procedentes de fuera del router sean redirigidos desde el puerto físico al virtual correspondiente entregando de este modo el control sobre el procesamiento del paquete al software de ruteo. Adicionalmente, también se

instala una regla para redirigir el flujo desde el software de ruteo hacia el exterior del router y permitir comunicación con el resto de los dispositivos de la red.

Este enfoque tiene como contrapartida el deterioro de la performance en el procesamiento de paquetes en el dispositivo, ya que se agrega un pequeño *delay* entre que el mensaje es efectivamente recibido y procesado. Además, el mismo *delay* existe a la hora de querer enviar un mensaje desde el software de ruteo hacia afuera del dispositivo. La prioridad de estas reglas es definida como baja para que las posteriores instaladas desde la controladora puedan sobreescribirlas si es necesario.

La arquitectura de un router híbrido queda por lo tanto resumida en la siguiente imagen:

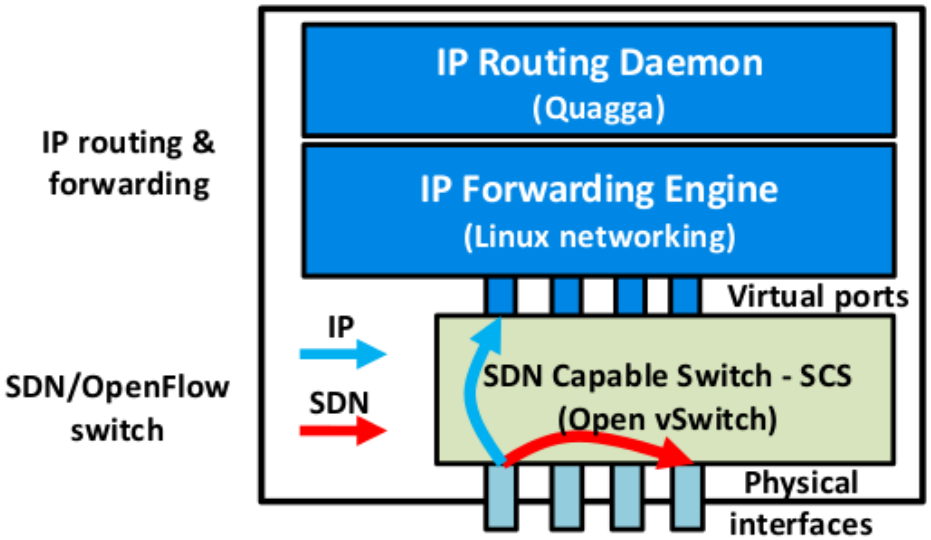

Fig. 5.1 Arquitectura de router híbrido

A continuación se muestran los flujos que son instalados al momento de inicializar un router híbrido dentro de una simulación en Mininet, para realizar la correspondencia entre los puertos físicos y virtuales:

```
#flujo Bird<-puerto virtual<-Switch SDN<puerto físico#
ovs-ofctl -O OpenFlow13 add-flow nombre_switch "table=0,
 ↪  hard_timeout=0, priority=400, in_port=puerto_físico,
 ↪  actions=output:puerto_virtual"

#flujo Bird->puerto virtual->Switch SDN->puerto físico#
ovs-ofctl -O OpenFlow13 add-flow nombre_switch "table=0,
 ↪  hard_timeout=0, priority=400, in_port=puerto_virtual,
 ↪  actions=output:puerto_físico"
```

Siendo:

- nombre_switch el nombre que se le dió al switch definido.
- puerto_físico y puerto_virtual son los números de puertos que se quieren hacer corresponder

Por otra parte, como fue mencionado en la sección correspondiente al diseño, necesitamos que todos los mensajes BGP Update recibidos por los ASBR sean enviados a la controladora dentro de un Packet-In. No solo eso, sino que todos los routers del EAS deben poder enviar dentro de un Packet-In los mensajes de confirmación, además de poder recibir las notificaciones de la controladora para unirse a los grupos de Multicast correspondientes.

Para posibilitar la primera de estas consideraciones, cada ASBR tendrá un flujo adicional por cada puerto físico que esté conectado a un router no perteneciente al EAS. Como queremos que este flujo aplique únicamente a mensajes BGP, el "matcheo" será realizado considerando que el tipo de protocolo sea TCP y el número de puerto origen/destino sea 179. El puerto de salida, además de su correspondiente puerto virtual, será el puerto reservado *CONTROLLER*; esto nos asegurará de que el paquete llegue al software de ruteo y además sea enviado a la controladora dentro de un Packet-In.

```
#flujo Bird<-puerto virtual, controladora<-Switch SDN<-puerto físico#
ovs-ofctl -O OpenFlow13 add-flow nombre_switch "table=0,
 ↪ hard_timeout=0, priority=500, dl_type=0x0800, nw_proto=6,
 ↪ tcp_dst=179, in_port=puerto_fisico,
 ↪ actions=output:puerto_virtual,output:controller"

ovs-ofctl -O OpenFlow13 add-flow nombre_switch "table=0,
 ↪ hard_timeout=0, priority=500, dl_type=0x0800, nw_proto=6,
 ↪ tcp_src=179, in_port=puerto_fisico,
 ↪ actions=output:puerto_virtual,output:controller"
```

Dónde los parámetros que realizan el "matcheo" son:

- dl_type es el tipo de trama ethernet, requerido para que se considere el parámetro nw_proto. El valor 0x0800 corresponde a una trama IP.
- nw_proto es el tipo de protocolo IP/Transporte usado. El valor 6 corresponde a TCP.
- tcp_dst es el número de puerto destino del mensaje.

- tcp_src es el número de puerto origen del mensaje.

En cuanto al segundo punto, surge el problema de cómo especificar el "matcheo" para estos paquetes. Nos pareció razonable que sea por la dirección IP destino del paquete, ya que de todas las formas de identificar un flujo consideramos ésta como una de las más específicas; pero se tiene que ver cuál será esa IP destino.

Optamos por utilizar una dirección particular de Multicast para este propósito, para así evitar cualquier tipo de colisión con otros flujos que potencialmente puedan existir en el router.

Por lo tanto, cada router del EAS tendrá dos flujos adicionales por cada par de puertos físicos/virtuales. Uno de ellos tendrá como puerto de entrada el puerto virtual y como puerto de salida el puerto reservado *CONTROLLER*; para el otro su puerto de entrada será el puerto físico y su puerto de salida será el virtual.

```
#flujo Bird->puerto virtual->Switch SDN->Controladora#
ovs-ofctl -O OpenFlow13 add-flow nombre_switch "table=0,
 ↪ hard_timeout=0, priority=800, dl_type=0x0800,
 ↪ nw_dst=CONTROLLER_MULTICAST_GROUP, in_port=puerto_virtual,
 ↪ actions=output:controller"

#flujo Bird<-puerto virtual<-Switch SDN<-puerto físico<-Controladora#
ovs-ofctl -O OpenFlow13 add-flow nombre_switch "table=0,
 ↪ hard_timeout=0, priority=800, dl_type=0x0800,
 ↪ nw_dst=CONTROLLER_MULTICAST_GROUP, in_port=puerto_físico,
 ↪ actions=output:puerto_virtual"
```

Dónde los parámetros que realizan el "matcheo" son:

- dl_type como ya fue dicho, es el tipo de trama ethernet.
- nw_dst es la dirección IP destino del paquete. CONTROLLER_MULTICAST_GROUP será la IP Multicast asignada para esta comunicación particular entre los routers y la controladora.

5.2 Bird

La configuración de Bird nos permite definir diferentes instancias de cada protocolo; para BGP en particular, cada una de estas instancias se corresponde con uno de los *peers* con

los que se comunicará el router. Dado que no es parte de nuestra solución modificar nada referente a las sesiones eBGP, es razonable mantener una instancia por cada *peer* externo de los ASBR; pero como planeamos eliminar las sesiones iBGP dentro del EAS, es pertinente cuestionarnos cuántas instancias extras tendremos. Las opciones naturales son: tener una sola instancia que recepcione los paquetes distribuidos por cada árbol de difusión o tener una instancia por cada uno de estos árboles. Nos parece más adecuado tener una única instancia, principalmente porque nos permite un manejo más dinámico ante la aparición de nuevos ASBRs (y, por lo tanto, de un nuevo árbol de difusión). De aquí en adelante nos referiremos a ésta como instancia interna.

Surge entonces la cuestión de cómo identificar, en la configuración y dentro del código, esta instancia interna del protocolo. Para no tener que definir una configuración específica y hacer modificaciones al analizador léxico y al *parser*, optamos por definir la instancia como si fuera una sesión iBGP habitual como se muestra a continuación:

```
protocol bgp {
        local as 1;
        neighbor 100.0.1.1 as 1;
        source address 100.0.2.2;
}
```

Lo importante de esta configuración es que el ASN del vecino sea igual al ASN local (el EAS), de esta manera cuando Bird "parsee" el archivo de configuración y, en base a éste, inicialice las estructuras *bgp_config* y *bgp_proto* (habrá una instancia de las dos estructuras por cada *protocol bgp* definido en la configuración), podamos determinar fácilmente si la instancia no corresponde a una sesión eBGP. Para esto basta verificar si el miembro *is_internal* de la estructura *bgp_proto* es falso. Nótese que, si bien la dirección IP del vecino y la local están definidas, esto es solo porque son requeridas por la configuración, sin embargo no son relevantes para la instancia interna ya que el concepto de router vecino no aplica, de la misma forma que tampoco la instancia interna estará identificada con una interfaz (puerto) específica.

Una vez inicializada la instancia interna, debemos determinar las secciones de código que es pertinente modificar. Para esto tuvimos en cuenta dos aspectos fundamentales: primero el análisis y comprensión de cómo Bird implementa la máquina de estados de BGP y luego cuáles de estos estados son relevantes dada la naturaleza de nuestra instancia interna y el tipo de protocolos que usa.

Probablemente esta segunda consideración sea la más importante; como la instancia interna no necesita establecer una sesión TCP con otro router del EAS, muchos de los estados

por los que pasaría una instancia normal del protocolo BGP no van a tener gran significado para ella (la instancia interna). De hecho, estas instancias internas van a ser mayormente entidades pasivas que recepcionarán los paquetes que se distribuyan por el árbol (recuérdese que el reenvío de los paquetes es realizado por los flujos definidos en cada router). En base a esto es que decidimos que estas instancias internas asumirán el estado "Established" desde que inician su ejecución hasta que finalizan.

El hecho de contar con solo un estado nos permite, entre otras cosas, prescindir de la implementación de funciones que manejen los *timeouts* que se producen durante los diferentes estados de BGP.

Otro factor que diferencia la instancia interna de la original es que no necesitamos un socket TCP sino UDP, para así escuchar los distintos grupos de Multicast por los que se distribuirán los mensajes BGP. Aun más, dado que a priori los mensajes pueden llegar por cualquier interfaz necesitamos tener un socket UDP por cada una de ellas, a diferencia de las instancias regulares que tan solo tienen un socket TCP.

Continuando con los sockets, debemos también considerar la comunicación de Bird con la controladora, en particular cuáles de las instancias del protocolo necesitarán comunicarse con ella. En primer lugar, la instancia interna necesitará enviar los ACK correspondientes a los paquetes distribuidos por los árboles Multicast. Por otra parte, las instancias que establecen sesiones eBGP necesitarán notificar a la controladora de la existencia de dicha sesión para que ésta genere un nuevo árbol de Multicast (de no existir ya). De esto se desprende que todas las instancias necesitarán enviar algún tipo de mensaje a la controladora.

Teniendo esto en cuenta consideramos más eficiente utilizar un socket estático global a todas las instancias en ejecución, en lugar de tener uno por cada instancia. Es de notar que Bird utiliza un socket de esta naturaleza para escuchar pasivamente intentos de establecer sesiones BGP.

Al ser este un socket estático, nos aseguramos de inicializarlo una única vez sin importar el número de instancias existentes. Esto se realiza en el siguiente método:

```
static sock *setup_controller_sk(struct bgp_conn *conn, ip_addr addr,
 ↪  unsigned port, u32 flags)
```

Aquí, se creará un socket que será asignado a la variable estática *controller_sk* y que se unirá al grupo Multicast definido específicamente para comunicarse con la controladora. La unión al grupo Multicast se realiza utilizando las siguientes funciones, definidas ya por Bird:

```
int sk_setup_multicast(sock *s)

int sk_join_group(sock *s, ip_addr maddr)
```

La primera de ellas configura el socket *s* para utilizar Multicast y la segunda efectivamente lo une al grupo especificado por el parámetro *maddr*.

La estructura *birdsock*, definida por Bird para representar y extender un socket, permite que se le asignen métodos denominados *hooks*, que no son otra cosa que manejadores de la recepción y del envío de mensajes. A este socket estático se le asignará como manejador de la recepción de mensajes el método:

```
static int rx_controller(sock *sk, int size)
```

Más adelante se detallará las acciones realizadas por éste.

5.2.1 Inicio de Ejecución

Nos dispusimos entonces a encontrar el lugar más apropiado para marcar el estado de la instancia interna como *ESTABLISHED* e inicializar los sockets encargados de escuchar los distintos árboles Multicast. Examinado el código, el siguiente método:

```
static void bgp_connect(struct bgp_proto *p);
```

nos pareció el más adecuado, ya que, entre otras cosas, aquí se inicializa el socket TCP que intentará establecer una sesión BGP con su *peer*.

Entonces, como se dijo anteriormente, si el miembro *is_internal* de la estructura *bgp_proto* es falso, se ejecutará el código normal de Bird; en el caso contrario, se agregó código para iterar sobre la lista de interfaces (inicializada por Bird) y crear un socket UDP Multicast ligado a cada interfaz (puerto físico). El método asignado para manejar los mensajes recibidos por cada socket es el siguiente:

```
int proy_bgp_connected(sock *sk, int size)
```

Este método deberá manejar adecuadamente tanto la recepción de mensajes BGP como las notificaciones de borrado de rutas. El detalle de su funcionamiento será referido más adelante.

Volviendo al método *bgp_connect*, también se creará una estructura de tipo *proy_bgp_conn*, que será asignada al miembro *proy_udp_conn* de la estructura *bgp_proto*. El miembro *proy_udp_conn* fue agregado por nosotros para que cumpla una función equivalente al miembro *conn*, que en la implementación de Bird es la responsable de manejar la sesión; dicho miembro es una estructura de tipo *bgp_conn*, la cual fue tomada como punto de partida para que definamos el tipo *proy_bgp_conn*.

El principal motivo de crear estas nuevos miembros y estructuras, es tratar de interferir lo menos posible en el funcionamiento de Bird y que todo lo que modifiquemos sea lo más independiente posible de lo ya implementado.

Por último, el conjunto de sockets es asignado a un array que se asigna al miembro *sk* de *proy_udp_conn* y también se setea la constante *ESTABLISHED* a su miembro *state*, indicando así que la "conexión" fue establecida.

5.2.2 Notificaciones de ASBR

Si bien anteriormente mencionamos que no vamos a modificar el comportamiento de las sesiones eBGP, necesitamos sí agregar un método para que las instancias con estas sesiones envíen la notificación a la controladora de que han establecido una sesión eBGP.

Para esto se implementó el método:

```
int proy_bgp_send_edge_notification(sock *sk)
```

Que enviará un mensaje con el string *EXTERNAL_SESSION* a la dirección Multicast definida para la comunicación con la controladora. Necesitamos ahora determinar el lugar más apropiado para realizar la invocación.

Examinando el código, encontramos que en el método:

```
void bgp_conn_enter_established_state(struct bgp_conn *conn)
```

Es donde efectivamente la conexión eBGP es establecida y la instancia eBGP cambia de estado; agregamos entonces al final de este método una llamada a *proy_bgp_send_edge_notification* pasando como parámetro el socket estático *controller_sk*.

5.2.3 Recepción de notificación Multicast

Los mensajes que la controladora envía a los routers cada vez que asigna una nueva dirección de Multicast son manejados por el método *rx_controller*. En éste, se toma la dirección IP que viene en el mensaje y se itera sobre los sockets UDP de la estructura *proy_udp_conn* para que cada uno de ellos se una al grupo correspondiente a la dirección recibida.

5.2.4 Recepción de mensajes BGP

Una vez que los sockets UDP se unen a los grupos de Multicast notificados por la controladora, están en condiciones de recibir los mensajes BGP. Ahora, la forma cómo se procesan estos paquetes no debería cambiar demasiado, ya que no es nuestra intención modificar como Bird realiza esto. Observando el código, encontramos que el método:

```
int bgp_rx(sock *sk, int size)
```

Es el *hook* que Bird define para los sockets TCP que establecen la sesión BGP. Éste se encarga de delimitar correctamente los mensajes BGP para pasárselos al método *bgp_rx_packet*, el cual únicamente inspecciona el tipo de mensaje BGP que es y así llamar al método que corresponda. En particular, el método que procesa los mensajes UPDATE es:

```
static void bgp_rx_update(struct bgp_conn *conn, byte *pkt, int len)
```

Éste, luego de hacer otros chequeos particulares a los mensajes UPDATE llama al método:

```
static void bgp_do_rx_update(struct bgp_conn *conn, byte *withdrawn,
 ↪  int withdrawn_len, byte *nlri, int nlri_len, byte *attrs, int
 ↪  attr_len)
```

El cual es el que efectivamente toma la información del mensaje y comienza el proceso de actualizar las tablas de ruteo internas de Bird.

Con el objetivo de aislar nuestro código del ya existente en Bird, se crearon tres métodos alternativos que se corresponden con los tres anteriormente mencionados:

```
int proy_bgp_connected(sock *sk, int size)

static void proy_bgp_rx_packet(struct proy_bgp_conn *conn, byte *pkt,
 ↪  unsigned len, ip_addr sourceAddr)

static void proy_bgp_do_rx_update(struct proy_bgp_conn *conn, byte
 ↪  *withdrawn, int withdrawn_len, byte *nlri, int nlri_len, byte
 ↪  *attrs, int attr_len, ip_addr sourceAddr)
```

Nótese que el primero de estos es el *hook* que asignamos a los sockets UDP.

Inicialmente, tomamos el cuerpo de los tres métodos de Bird y fuímos adaptándolos para que utilicen las estructuras que nosotros definimos (e.g, usar *proy_bgp_conn* en lugar de *bgp_conn*) e incluso remover código que fuera innecesario (e.g, lo referente a la máquina de estados habitual de Bird). El procesamiento en sí del *payload* del mensaje BGP sigue siendo enteramente responsabilidad de Bird.

Una vez que Bird termina de actualizar sus tablas, generamos un hash MD5 del mensaje BGP y utilizamos el socket estático *controller_sk* para mandar el mensaje ACK a la controladora con el formato especificado en la etapa de diseño.

5.2.5 Notificación de cierre de sesión eBGP

En el momento en que una sesión eBGP se cierre, Bird debe enviar un mensaje a la controladora con la dirección IP de su *peer*. Para esto, se implementó el método:

```
void send_bgp_del(struct bgp_conn* conn)
```

El cual tomará la dirección IP del *peer* utilizando el miembro *conn* y se encargará de crear el mensaje definido durante la etapa de diseño (i.e *DEL#PEER_IP*). Este mensaje será enviado por el socket estático *controller_sk*.

Nuevamente, debemos inspeccionar el código de Bird para encontrar el lugar más apropiado para invocar a nuestro método. Hallamos que en el siguiente método:

```
static void bgp_conn_leave_established_state(struct bgp_proto *p)
```

Es dónde inicia el proceso de cierre de sesión y por lo tanto aquí fue agregada esa llamada, pasando como párametro el atributo *conn* de la estructura *bgp_proto*.

5.2.6 Recepción de notificaciones de borrado de rutas

Una vez más inspeccionamos el código de Bird, ahora para determinar el procedimiento que realiza para eliminar las rutas de sus tablas internas cuando se cierra una sesión BGP.

Encontramos que durante el proceso de cierre de sesión se agrega al planificador interno de Bird un evento que invoca al método:

```
int rt_prune_loop(void) // en nest/proto.c
```

El cual itera sobre las tablas internas de Bird y elimina las rutas aprendidas por instancias de protocolos que por algún motivo ya no son válidas; en particular para BGP, esto sucede cuando se cierra una sesión.

Observamos también que, dado que las tablas pueden contener numerosas entradas, Bird realiza un número limitado de iteraciones cada vez que el método es invocado; si llega a ese límite, el método agrega un nuevo evento al planificador para continuar con la iteración, permitiendo de esta forma que el planificador continúe con la ejecución de algún otro protocolo activo.

Teniendo en cuenta estas consideraciones, creamos un método análogo:

```
static void proy_flush_routes(void *vp)
```

Para que realice el borrado de rutas que hayan sido comunicadas por routers específicos; esto es posible dado que Bird almacena en cada entrada de sus tablas de ruteo la dirección IP origen del mensaje que envió la ruta.

En lugar de iterar por todas las tablas internas, nuestro método lo hará solo por la correspondiente a la instancia interna. No obstante el hecho de que recorramos una única tabla, ésta potencialmente podría tener un gran número de entradas; es por esto que también limitaremos el número de iteraciones y de ser necesario se agregará un nuevo evento al planificador para continuar con el recorrido.

Dentro del método *proy_bgp_connected*, si el mensaje recibido es una notificación de borrado, se parseará el mensaje para obtener la lista de IPs enviadas la cual será almacenada en el atributo *proy_ip_list_ready* de la estructura *bgp_proto*. Luego de hecho esto, se agregará un evento al planificador para que se haga la llamada inicial a *proy_flush_routes*.

Dentro del método se tomará la lista de IPs de *proy_ip_list_ready* y se las moverá a *proy_ip_list_execute*, siendo esta lista la que el método utilizará para determinar si debe borrar una ruta o no.

El motivo de estas dos listas se debe a que puede darse la posibilidad de que la instancia tenga una tabla con un gran número de entradas y que, mientras aún no haya terminado de recorrer toda la tabla para realizar el borrado, llegue una nueva notificación indicando que tienen que borrarse rutas de otros orígenes. Si usáramos una única lista y le agregáramos estas nuevas IPs, potencialmente estaríamos dejando entradas al inicio de la tabla que deberíamos haber borrado. Por esto es que mantenemos esta lista *proy_ip_list_ready*; cuando la iteración sobre la tabla finalice, el método consultará si esta lista no es vacía y en caso de ser así, volverá a agregar otro evento para que comience una nueva iteración sobre la tabla, esta vez utilizando la nueva lista de IPs.

5.2.7 Cuantificación de cambios en el código

Con el objeto de dar un panorama cuantitativo de lo que fue necesario modificar en el código de Bird, presentamos en la tabla 5.1 los archivos a los que se le realizaron cambios, la cantidad de líneas de código en el archivo original y la cantidad de líneas una vez modificado. Nótese que estas cantidades refieren estrictamente a líneas de código, es decir, no incluyen líneas en blanco ni comentarios; la obtención de estos valores fue realizada utilizando el aplicativo cloc [38].

Tabla 5.1 Resumen de archivos modificados en Bird

Archivo	Cant. líneas original	Cant. líneas modificado	% de nuevas líneas
nest/iface.c	586	603	2.82
nest/iface.h	115	116	0.86
nest/protocol.h	259	263	1.52
proto/bgp/attrs.c	1352	1499	9.81
proto/bgp/bgp.c	1058	1487	28.85
proto/bgp/bgp.h	290	329	11.85
proto/bgp/packets.c	1310	1495	12.37

5.3 Controladora RYU

5.3.1 Aspectos generales

La implementación del controlador RYU fue realizada dentro de una clase que llamamos *Multicast_BGP*. Como es requerido al utilizar RYU, la misma es sub-clase de *app_manager.RyuApp*.

En su constructor, se inicializan todas las estructuras que se necesitarán para su correcta ejecución. A continuación se describen brevemente las más significativas.

Routers

Diccionario que tiene como clave el Id del datapath con cada router y como valor una estructura que contendrá información sobre los puertos del router así como también una referencia al objeto datapath (creado por la aplicación base de RYU).

Topology

Instancia de la clase *DiGraph* (grafo dirigido) de la librería *NetworkX* [39], representa el estado actual de la topología. La razón de por qué se escogió este tipo de grafo será explicada más adelante cuando se haga referencia a la forma en que se descubre la topología.

External_Sessions

Diccionario que contiene la información correspondiente a cada árbol de Multicast generado por la controladora. Esto incluye: la dirección del grupo Multicast, el árbol propiamente dicho (también una instancia de la clase Di-Graph), las IPs de los *peers* que mantienen sesiones eBGP con el ASBR y otros que se mencionarán más adelante.

Otro aspecto importante al momento de inicializar la controladora es que aquí creamos e iniciamos un objeto *Timer* de la librería *Threading* de Python ([40]). Este objeto permite ejecutar una función luego de que un determinado tiempo haya pasado. Abajo incluímos el código necesario para crear e iniciar un objeto *Timer*:

```
self.STABLE_TOPOLOGY_TIMER =
 ↪  threadLib.Timer(self.WAIT_TIME_FOR_STABLE_TOPOLOGY,
 ↪  self.stableTopology)
self.STABLE_TOPOLOGY_TIMER.start()
```

El primer parámetro que toma el objeto *Timer* al ser creado es el tiempo a esperar para ejecutar la función que es especificada en el segundo parámetro. El método *start* inicia el *Timer*. Sobre la función *stableTopology* se hablará más adelante.

5.3.2 Información de routers

Si bien al inicio, cuando RYU se encarga de establecer el datapath con los routers, la mayoría de los mensajes intercambiados no son de mayor interés para nuestra aplicación, la excepción a esto son las respuestas de los routers a los mensajes *OFPT_MULTIPART_REQUEST* de tipo *OFPMP_PORT_DESCRIPTION* que son realizados internamente por RYU.

Los routers responden a estos *requests* con información de sus puertos, la cual es necesaria para nuestra implementación. Para poder acceder a dicha información, definimos la siguiente función:

```
@set_ev_cls(ofp_event.EventOFPPortDescStatsReply, CONFIG_DISPATCHER)
def port_desc_stats_reply_handler(self, ev):
```

Decorada con el nombre correspondiente al evento que queremos manejar.

Es en esta función que creamos la estructura que usaremos para representar a cada router; se listan a continuación los atributos más significativos:

- id: usado para identificar al router, se le asigna el nombre del puerto reservado *Local*.
- path: referencia al objeto datapath manejado por RYU.
- interfaces: diccionario que contiene estructuras que representan al par puerto físico/virtual con los siguientes atributos:
  - logicalMac: MAC asignada al puerto virtual

- logicalPort: número de puerto virtual
- physicalMac: MAC asignada al puerto físico
- physicalPort: número de puerto físico

Una vez inicializada esta estructura será agregada al diccionario Routers.

Como este evento sucede cuando la conexión con la controladora acaba de establecerse, consideramos adecuado que sea aquí que se notifique al router de cuáles son los grupos Multicast a los que debe unirse. Sin embargo, debido a que durante la simulación la conexión con la controladora es establecida antes de que se inicie Bird, que es quien tiene que recibir estas notificaciones, decidimos retrasar un poco el envío utilizando un objeto *Timer*. La función a ejecutar luego de pasado el tiempo de espera es:

```
def sendPendingNotifications(self, currentDP)
```

Donde *currentDP* es el Id del datapath por el cual debemos mandar las notificaciones.

5.3.3 Descubrimiento de la Topología

Como fue dicho anteriormente, RYU cuenta con un mecanismo para descubrir la topología que forman los dispositivos con los que ha establecido un canal OpenFlow, utilizando Link Layer Discovery Protocol (LLDP). Sin entrar en mayores detalles del funcionamiento de este protocolo, RYU permite que se agreguen funciones para que manejen los distintos eventos; en particular, los eventos que nos interesan son: aparición y pérdida de routers y links. Se crearon los siguientes métodos para manejar cada uno de estos eventos:

```
@set_ev_cls(event.EventSwitchEnter)
def routerDiscovered(self, ev)

@set_ev_cls(event.EventSwitchLeave)
def routerLost(self, ev)

@set_ev_cls(event.EventLinkAdd)
def linkDiscovered(self, ev)

@set_ev_cls(event.EventLinkDelete)
def linkLost(self, ev)
```

En cada una de estas funciones realizamos la operación correspondiente sobre la instancia *Topology*: agregar un nodo, remover un nodo, agregar un link o remover un link. Cualquiera de estos eventos hará que se reinicie el *Timer* que espera que la topología se estabilice.

En el método *routerLost* en particular, si el router con el que ya no hay conexión es un ASBR, se invocará al método *notificarCaidaSesionEBGP*, encargado de enviar las notificaciones de borrado de rutas. Más adelante detallaremos el funcionamiento de éste método.

Un aspecto importante de los eventos que refieren a los links es que se generan un par de eventos por cada link, cada uno de ellos especificando a uno de los routers como el origen, al otro como destino y adicionalmente el número de puerto del router origen que lo vincula al router destino; i.e, si R_1 y R_2 tienen un link entre ellos, R_1 por su puerto número 1 y R_2 por su puerto número 2, se generará un evento con origen R_1, destino R_2, puerto origen número 1 y otro con origen R_2, destino R_1, puerto origen número 2. Es por esto que optamos por utilizar una instancia de la clase *DiGraph*, además de que utilizar un grafo dirigido nos ayudará para el armado de los árboles Multicast y la definición de los flujos OpenFlow.

5.3.4 Manejador de Packet-In

Para manejar los Packet-In que lleguen a la controladora se definió la siguiente función:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev)
```

Aquí hacemos uso de las librerías provistas por RYU para determinar el tipo de paquete que fue enviado dentro del Packet-In y en base a esto determinar como manejarlo. A continuación listaremos los paquetes esperados.

Notificaciones de ASBR

Si recibimos una notificación desde un datapath cuyo Id no es clave del diccionario *External_Sessions*, crearemos una estructura para el mismo, siendo sus atributos más significativos:

- *multicastIP*: dirección Multicast asignada al ASBR.
- *multicastMac*: MAC generada a partir de la dirección Multicast [41].
- *tree*: una vez que la topología se encuentre estable, se le asignará una instancia *DiGraph* que represente el árbol de distribución para este ASBR.
- *peer*s: conjunto que contendrá las direcciones IP de los *peers* del ASBR.

- *packets*: lista que almacenará los mensajes BGP Update que ha recibido el ASBR mientras la topología no se encontraba en un estado estable.
- *packets_in_transit*: diccionario con los mensajes BGP Update que han sido distribuidos por el árbol Multicast, siendo la clave su hash MD5.
- *notification*: mensaje de notificación a ser enviado a los demás routers del EAS para que se unan al grupo Multicast asignado al ASBR.

Mensajes BGP Update

Al recibir un mensaje BGP Update, usando como clave el Id del datapath obtendremos del diccionario *External_Sessions* la dirección Multicast por la que se deberá distribuir el mensaje y encapsularemos a éste dentro de un segmento UDP. Aquí pueden suceder dos cosas

- Si la topología no está estable, el segmendo UDP es guardado en la lista *packets*, para ser distribuido una vez la topología se estabilice.
- Si la topología está estable y los flujos ya han sido instalados, el segmento UDP será a su vez encapsulado dentro de un Packet-Out y enviado por el mismo datapath que recibió el mensaje BGP; su única acción de salida sea reenviarlo al puerto reservado OFPP_TABLE, para que sea procesado por las Flow Tables.

En este último caso debemos además inicializar un objeto *Timer* que será asignado a cada router del EAS. El método encargado de hacer esto es:

```
def inicializarReceptoresACK(self, datapathId, bgpPkt, hashedPkt,
 ↪  tipo_paquete)
```

Siendo sus parámetros:

- *datapathId*: Id del datapath correspondiente al ASBR.
- *bgpPkt*: segmento UDP conteniendo un mensaje (puede ser BGP o notificación de borrado).
- *hashedPkt*: hash MD5 del mensaje (i.e no incluyendo el header UDP).
- *tipo_paquete*: constante que indica si el mensaje es BGP o una notificación de borrado.

El método a ejecutar una vez que expire el *Timer* es:

```
def ackNoRecibido(self, currentDP, rootDP, hashPkt)
```

Tomando como parámetros:

- *currentDP*: Id del datapath por el que se esperaba recibir el ACK.
- *rootDP*: Id del datapath correspondiente al ASBR.
- *hashPkt*: hash MD5 del mensaje para el que se esperaba confirmación.

Este método únicamente obtendrá el mensaje BGP del diccionario *packets_in_transit* y lo enviará por el datapath encapsulándolo dentro de un Packet-Out y siendo su acción de salida reenviarlo al puerto reservado OFPP_TABLE. Se debe tener especial cuidado de especificar el parámetro *in_port* al enviar el Packet-Out; este debe corresponderse con el puerto por el que el paquete debió haber llegado de no haberse perdido.

Notificaciones de cierre de sesión eBGP

Este mensaje es manejado por el método:

```
def notificarCaidaSesionEBGP(self, currentDP, routerCaido, peerIPs)
```

Siendo sus parámetros:

- *currentDP*: Id del datapath correspondiente al ASBR.
- *routerCaido*: valor booleano que es verdadero si la notificación de borrado se debe a que se ha perdido la conexión con el router o falso si solo se ha cerrado una sesión eBGP.
- *peerIPs*: lista con las IPs de los *peers* cuyas rutas anunciadas deben ser borradas.

Generaremos aquí la notificación de borrado de rutas con las IPs incluídas en *peerIPs* que será enviada dentro de un Packet-Out, siendo su acción de salida reenviarlo al puerto reservado OFPP_TABLE. El datapath por el que se enviará este Packet-Out depende de si la conexión con el ASBR se ha perdido o no:

- Si no se perdió, se envía por el datapath del ASBR.
- Si se perdió, se utiliza el árbol de distribución del ASBR para determinar sus hijos directos y el Packet-Out será enviado por los datapaths de éstos.

Mensajes ACK

Cuando se recibe un mensaje ACK, verificamos que tanto la dirección del grupo Multicast como el hash MD5 se correspondan con valores válidos y de ser así, se cancela el *Timer* asignado al router que envió el ACK para este mensaje en particular.

5.3.5 Estabilización de topología

Una vez que no existan nuevos eventos generados por el descubrimiento de la topología, el método *stableTopology* será ejecutado. Lo primero que hará es armar los árboles Multicast, invocando al método:

```
def generateTrees(self)
```

El cual iterará sobre el diccionario *External_Sessions* para generar un árbol por cada ASBR utilizando el algoritmo de Prim; el cual es relativamete sencillo de implementar y además se acopla bien a la forma en como pretendíamos armar el árbol; recuérdese que en Prim se parte de un nodo al azar, que en nuestro caso sería el ASBR. Luego de tener los árboles, se llama al método:

```
def installRules(self)
```

El cual por cada árbol realizará las siguientes acciones:

- Por cada router identificado como origen de una arista (en otras palabras, todo router que no sea hoja del árbol) se tendrá una acción de salida hacia el puerto físico identificado como origen de la arista (garantizando el reenvío al router destino) y otra acción de salida al puerto virtual correspondiente (garantizando que Bird lo reciba).
- Se enviará un mensaje *OFPT_FLOW_MOD* de tipo *ADD*, para agregar un flujo cuyo "matcheo" sea según la dirección IP destino del paquete (la correspondiente al grupo Multicast) y cuyas acciones serán el conjunto determinado en el punto anterior.

Una vez enviados los mensajes para instalar las reglas, se creará un nuevo objeto *Timer* que al expirar invocará al método:

```
def sendPendingPackages(self)
```

El cual se encargará de distribuir aquellos mensajes BGP que llegaron mientras la topología no se encontraba estable. La razón para que este método sea invocado con cierto retraso es para asegurarnos de que las reglas fueron efectivamente instaladas en los routers.

Sección 6

Pruebas

Para verificar que nuestra solución funciona de acuerdo al diseño que nos propusimos y que efectivamente su comportamiento tiene resultados equivalentes a la configuración full mesh, diseñamos un conjunto de pruebas funcionales para simular las distintas dificultades que identificamos durante la fase de diseño.

Además se realizó una serie de pruebas de escalabilidad utilizando los diseños de un par de topologías reales, para identificar posibles problemas que puedan no ser tan claros al utilizar una topología de menor porte y además obtener métricas con el objetivo de hacer un análisis cualitativo de la solución.

El conjunto de pruebas fue ejecutado en una notebook con las siguientes especificaciones:

- Modelo: Dell Inspiron 5720
- CPU: Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
- RAM: 8 GB
- SO: Ubuntu 14.04 Kernel 3.13.0

6.1 Pruebas Funcionales

Para cada una de estas pruebas se escribió un par de scripts de Mininet; uno con los routers configurados para establecer un full mesh dentro del AS, el otro utilizando los routers híbridos tal cual fueron especificados en la sección de implementación.

La idea es ejecutar primero el script que implementa el full mesh para así, una vez que sucedan todos los hechos significativos de la prueba, obtener la tabla de ruteo de todos los routers del AS. Luego se ejecuta el script con los routers híbridos y se compara una a una las

tablas de ruteo de cada router. La prueba es considerada exitosa si todas las tablas tienen las mismas entradas. La topología empleada para todas las pruebas es la siguiente:

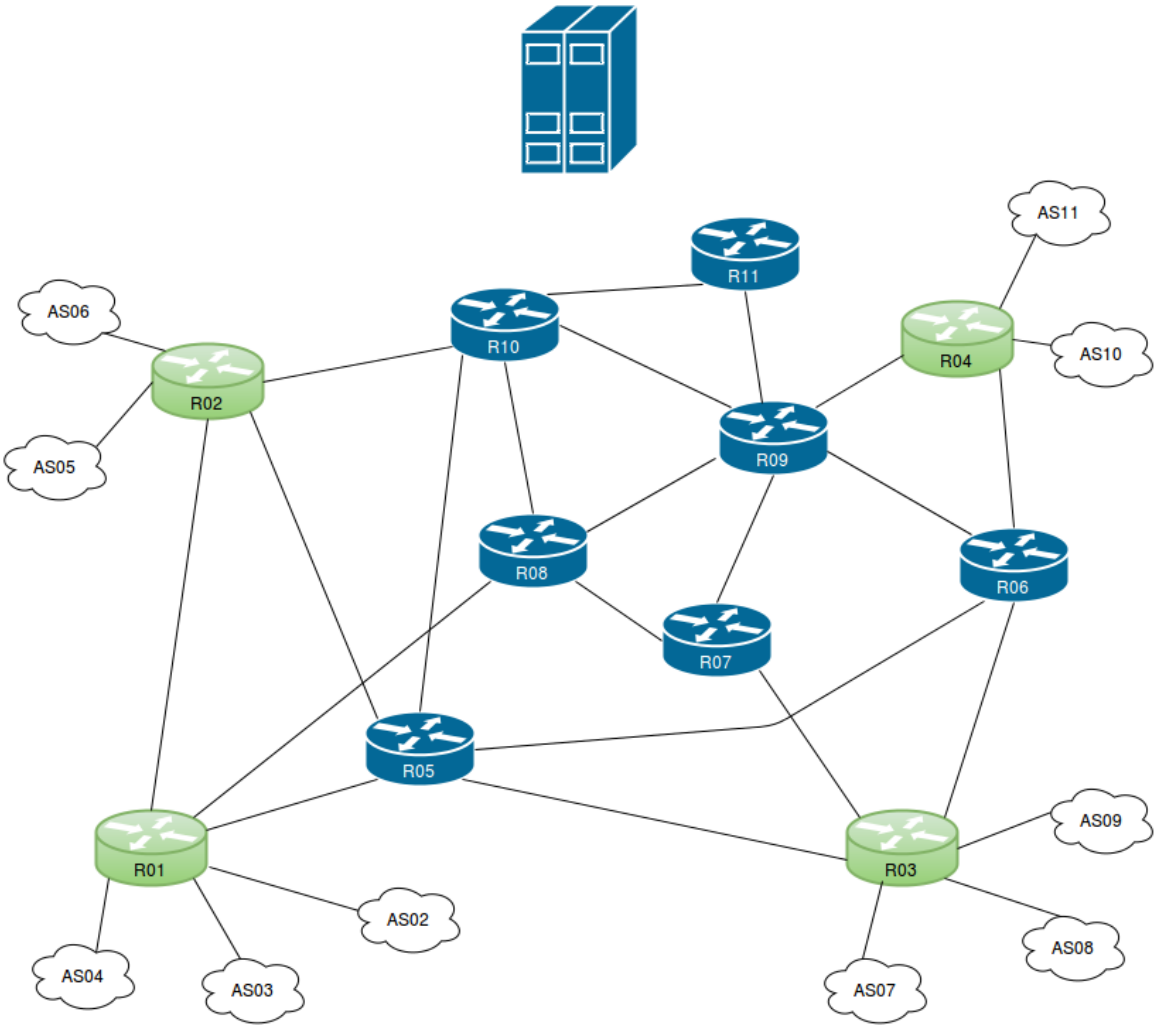

Fig. 6.1 Topología de pruebas funcionales

Con las siguientes consideraciones:

- Los routers de color verde son ASBRs.
- Existe un enlace implícito desde todos los routers del EAS a la controladora.
- Los AS externos constan únicamente de un router.

6.2 Pruebas funcionales

Con este conjunto de pruebas se pretende comprobar el correcto funcionamiento de los distintos eventos a los que debe dar respuesta el EAS. Para ello, se recrearán los mismos y el EAS deberá responder provocando que todos los dispositivos del EAS terminen seleccionando las mismas rutas que seleccionarían si estuvieran conectados en modalidad full mesh.

6.2.1 Prueba 1: correctitud

Esta prueba presenta un escenario básico para verificar que obtenemos un resultado correcto bajo condiciones en que no hay cambios en la topología y que tanto los routers como los links funcionan sin problemas.

6.2.2 Prueba 2: aparición de nuevo router interno

Se busca con esta prueba comprobar el comportamiento del EAS ante la aparición de un nuevo router interno, en particular para verificar que todas las rutas previas que fueron distribuidas por el EAS sean aprendidas por este nuevo router.

6.2.3 Prueba 3: aparición de nuevo ASBR

Se probará en este caso el comportamiento del EAS ante la aparición de un nuevo ASBR. El comportamiento esperado es la generación de nuevos árboles de Multicast por parte de la controladora y la redistribución de las rutas aprendidas por el EAS hacia el nuevo ASBR. También es de esperar que éste anuncie las rutas que aprenda de sus *peers* externos a todo el EAS.

6.2.4 Prueba 4: pérdida de mensajes BGP

El objetivo de esta prueba es simular la pérdida de un mensaje BGP en algún link del árbol, para que ocurra un *timeout* y la controladora tenga que reenviar el mensaje al router que no lo recibió.

La "pérdida" del mensaje es simulada al remover uno de los puertos de salida de un flujo en un ASBR, correspondiéndose el flujo con el árbol del que este ASBR es raíz.

6.2.5 Prueba 5: cierre de sesión eBGP

En este caso, se verificará que ante el cierre de sesión con un *peer* externo, los routers del EAS efectivamente borren las rutas que fueron comunicadas por éste.

El cierre de la sesión es logrado terminando el proceso *bgpd* de uno de los routers externos.

6.2.6 Prueba 6: falla de ASBR

Con esta prueba se busca simular la falla de un ASBR para verificar que la controladora reconoce ésta y actúa acordemente: rearmando los árboles de distribución y enviando las notificaciones de borrado con las direcciones IPs de todos los *peers* del router fuera de funcionamiento.

De forma similar a la prueba anterior, la falla es simulada terminando los procesos *ovs-vswitchd* y *bird* de un ASBR.

6.3 Pruebas de Escalabilidad

En estas pruebas se pretende verificar la escalabilidad del EAS en los siguientes escenarios; por un lado, al inyectar un gran volumen de mensajes BGP en un corto lapso de tiempo, por otro, al utilizar topologías de mediano y gran porte.

Para los últimos dos casos, emularemos algunas topologías reales tomadas del proyecto The Topology Zoo[42]. Se buscó que estas topologías ya incluyeran routers definidos como externos, a fin de no tener que asignar los ASBRs según nuestro criterio y acercarnos más a un diseño topológico real.

En todas estas pruebas los routers de los ASs externos inyectarán rutas utilizando ExaBGP [43], el cual leerá un archivo con trazas tomadas de repositorios públicos de rutas reales. En particular, utilizamos las del proyecto RIPE [44].

6.3.1 Prueba con alto tráfico de mensajes BGP

En este escenario se realizará una carga de un gran volumen de rutas en el EAS. La topología empleada se corresponderá con la utilizada en las pruebas funcionales (Fig. 6.1).

Cada uno de los 10 routers externos enviará 100 mensajes UPDATE con un intervalo de 0,1 segundos entre cada mensaje.

6.3.2 Prueba con topología de mediano porte

Esta prueba emula la red UUNET[42] compuesta por 49 routers (7 de ellos externos) dispuestos como se muestra en Fig 6.2.

En este caso, los 7 routers externos enviarán solo 20 mensajes UPDATE con un intervalo de 7 segundos entre cada mensaje, de forma tal que nos centremos en como influye el tamaño de la topología en lugar del volumen de mensajes.

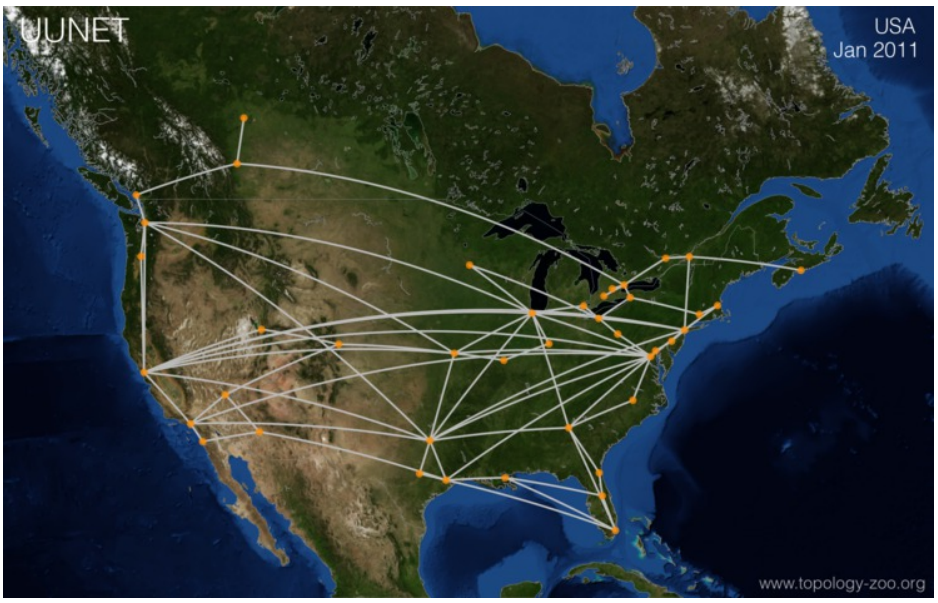

Fig. 6.2 Imagen de la red UUNET

6.3.3 Prueba con topología de gran porte

En esta prueba consideraremos la red Interoute[42] la cual está compuesta por más de 100 routers (5 de ellos externos) interconectados según se muestra en Fig 6.3.

Aquí también, cada uno de los 5 externos enviará 20 mensajes UPDATE con un intervalo de 7 segundos entre cada mensaje.

6.4 Métricas

A fin de poder comparar el funcionamiento del EAS con el de un AS tradicional, se tomaron métricas en las pruebas de escalabilidad definidas. Éstas son:

Fig. 6.3 Imagen de la red Interoute

- Cantidad de memoria utilizada por Bird, categorizándola según en qué está siendo utilizada: Routing Tables, Route attributes, ROA tables, Protocols y Otros. Para esto utilizamos el cliente ligero de Bird (birdcl).

- Cantidad de tráfico enviado por las interfaces de cada router, clasificándolo según su tipo: BGP, OpenFlow, LLDP, Tree Updates (mensajes BGP distribuídos por los árboles) y Notificaciones (mensajes entre los routers y la controladora). Esto lo mediremos utilizando el aplicativo tshark[45] sobre todas las interfaces filtrando por los tipos de mensajes que nos interesa medir.

6.4.1 Resultados de prueba con alto tráfico BGP

En la tabla 6.1 pueden verse los resultados de uso de memoria promedio de los routers de borde e internos en el EAS (EAS ASBR y EAS internos respectivamente) y en configuración full mesh (FM ASBR y FM internos) así como también el promedio general.

La razón de promediar los valores según el tipo de router se debe a que, dentro de cada uno de los tipos, los routers no presentan variaciones demasiado significativas para que valga la pena considerarlos individualmente.

Como puede apreciarse, el uso de memoria es bastante similar en ambos casos. Sin embargo, vemos una reducción en la cantidad de memoria dedicada a los protocolos, lo cuál es esperable ya que el funcionamiento original de Bird requiere que se defina una instancia del protocolo BGP por cada *peer* configurado; mientras que en la variante implementada en el EAS esto se realiza de esa forma sólo para los *peers* externos, ya que para los internos basta definir únicamente una instancia para su funcionamiento.

Para medir el tráfico, dado el reducido tamaño de la topología, optamos por seleccionar un router interno y un ASBR cuyo grado corresponda al grado promedio de cada tipo de router. Tomamos como representativo de los ASBRs a *AS01-R02* (siendo su grado igual a

Tabla 6.1 Memoria prueba con alto tráfico BGP (KB)

Router	Routing tables	Route attributes	ROA tables	Protocols	Otros	Total
EAS ASBR	183.5	140.5	0.19	128.55	37.81	490.55
FM ASBR	166.6	117.4	0.19	219.2	40.51	543.9
EAS Internos	163	115	0.19	115.29	32.67	426.15
FM Internos	165	117.46	0.19	205.97	37.44	526.06
EAS	170.45	124.27	0.19	120.11	34.54	449.56
FM	165.58	117.43	0.19	210.78	38.55	532.53

Tabla 6.2 Tráfico en topología de prueba con alto tráfico BGP (KB)

Router	AS	BGP	LLDP	OpenFlow	Tree Updates	Notificaciones	Total
AS01-R02	EAS	213925	6518	665455	41507	145155	1072560
	FM	329287	0	0	0	0	329287
AS01-R06	EAS	0	6273	416989	159326	173160	755748
	FM	119187	0	0	0	0	119187

5 y el promedio 5.25) y de los routers internos a *AS01-R06* (siendo su grado igual a 4 y el promedio 4.14).

Se observa que el tráfico generado utilizando una configuración full mesh es mucho menor que el tráfico del EAS. Dentro del tráfico del EAS, los mensajes OpenFlow dan cuenta de más del 50% del tráfico en cada router. Observamos también que los mensajes LLDP, necesarios para mantener el conocimiento de la topología, tienen un impacto mínimo en el tráfico total de cada router. Por otra parte, como era esperable al eliminar las sesiones iBGP, el router interno del EAS no genera mensajes BGP.

6.4.2 Resultados de pruebas con topologías de distinto porte

Analizaremos a continuación el comportamiento de la simulación al ir aumentando el tamaño de la topología, manteniendo constante la cantidad de rutas inyectadas al EAS; en cada caso los routers externos enviarán 20 mensajes UPDATE con un intervalo de 7 segundos entre cada mensaje.

Tabla 6.3 Memoria topología porte pequeño (KB)

Router	Routing tables	Route attributes	ROA tables	Protocols	Otros	Total
EAS ASBR	65	41.5	0.19	128.75	38.16	273.6
FM ASBR	66.3	39	0.19	219.65	40.46	365.6
EAS Internos	59	34.94	0.19	115.86	32.67	242.66
FM Internos	65	39	0.19	207.11	36.98	348.28
EAS	61.18	37.33	0.19	120.55	34.57	253.92
FM	65.47	39	0.19	211.67	38.24	354.57

Tabla 6.4 Tráfico en topología de porte pequeño (KB)

Router	AS	BGP	LLDP	OpenFlow	Tree Updates	Notificaciones	Total
AS01-R02	EAS	52300	10649	173038	9015	22040	267042
	FM	220126	0	0	0	0	220126
AS01-R06	EAS	0	8242	97955	20739	22608	149544
	FM	91157	0	0	0	0	91157

Resultados de prueba con topología de porte pequeño

En la tabla 6.3 vemos el promedio de la memoria utilizada por Bird, agrupando los routers de la misma forma que en la prueba anterior.

Como puede verse, el uso de memoria es algo menor en los routers del EAS, siendo la memoria dedicada a los protocolos la que genera esta diferencia. Ésta existe por el motivo explicado anteriormente: los routers del EAS solo utilizan una instancia del protocolo BGP.

Dado que esta prueba utiliza la topología definida por nosotros (Fig. 6.1), utilizamos el mismo criterio que en la prueba de alto tráfico para seleccionar los routers representativos, siendo estos *AS01-R02* y *AS01-R06*.

Se observa en la tabla 6.4 que el tráfico en el EAS es mayor (64% más en el caso de *AS01-R06*). En éstos, la mayor parte del tráfico corresponde a mensajes OpenFlow; esto es aun más notorio en *AS01-R02* dado que, al ser ASBR, debe enviar un mensaje Packet-Out por cada mensaje BGP recibido. El hecho de que *AS01-R06* tenga mayor cantidad de tráfico correspondiente a la categoría Tree Updates, puede deberse a su centralidad en la topología, lo cual hace menos probable que resulte elegido como hoja en alguno de los árboles.

Tabla 6.5 Memoria topología porte mediano (KB)

Router	Routing tables	Route attributes	ROA tables	Protocols	Otros	Total
EAS ASBR	63	32.53	0.19	179.47	38.71	313.9
FM ASBR	81.1	40.07	0.19	591.33	58.64	771.33
EAS Internos	59.11	28.13	0.19	127.57	33.27	248.27
FM Internos	80.78	40.28	0.19	528.44	53.96	703.65
EAS	59.66	28.75	0.19	134.82	34.03	257.45
FM	80.82	40.25	0.19	537.42	54.63	713.31

Resultados de prueba con topología de porte mediano

En la tabla 6.5 vemos el promedio de la memoria utilizada por Bird, agrupando los routers de la misma forma que en la prueba anterior.

Se ve más claramente aquí que los routers en configuración full mesh consumen una mayor cantidad de memoria en comparación a los del EAS. Si bien las tablas de rutas por instancia del protocolo BGP son más pequeñas en la configuración full mesh, la necesidad de éste de contar con múltiples instancias hace que terminen consumiendo más memoria que en la variante EAS.

La selección de los routers en los cuales se midió el tráfico fue hecha nuevamente considerando los dos tipos de routers (ASBR e internos) y el grado promedio de éstos. Dado que se cuenta con un mayor número de routers totales, calculamos también la desviación estándar sobre el grado de los routers, para obtener de esta forma un grupo más representativo.

Considerando los ASBRs, el grado promedio es 8.17 y la desviación estándar es 2.67; es por esto que elegimos a los routers *Los Ángeles* (grado 6), *New York* (grado 8) y *San Francisco* (grado 10). En los internos, el grado promedio es 3.11 y la desviación estándar es 2.18; los elegidos fueron *Portland* (grado 2), *Boston* (grado 3) y *Cleveland* (grado 5).

Podemos ver que el tráfico es bastante diverso. Considerando primero los routers *Portland* y *Cleveland*, vemos que el tráfico del primero es mayor en el EAS que bajo la configuración full mesh, sin embargo *Cleveland* tiene mucho menos tráfico en el EAS que en el full mesh. Esto nos lleva a pensar que el hecho de utilizar los árboles multicast generados por la controladora hace que se distribuya de manera más uniforme el tráfico sobre los routers que al utilizar el enrutamiento IGP dentro del AS.

Nuevamente es notorio que los mensajes OpenFlow dan cuenta de la mayor parte del tráfico en cada router del EAS y el mismo es particularmente alto en los ASBRs.

Tabla 6.6 Tráfico en topología de porte mediano (KB)

Router	AS	BGP	LLDP	OpenFlow	Tree Updates	Notificaciones	Total
Portland	EAS	0	1805	53730	0	17676	73211
	FM	45156	0	0	0	0	45156
Boston	EAS	0	2203	58582	7421	18756	86962
	FM	171424	0	0	0	0	171424
Cleveland	EAS	0	2774	60014	30686	17694	111168
	FM	290416	0	0	0	0	290416
Los Ángeles	EAS	18505	6416	103644	68472	17428	214465
	FM	395239	0	0	0	0	395239
New York	EAS	17985	5161	100031	79129	17068	219374
	FM	950027	0	0	0	0	950027
San Francisco	EAS	13210	2917	69175	1719	14076	101097
	FM	316310	0	0	0	0	316310

Tabla 6.7 Memoria topología gran porte (KB)

Router	Routing tables	Route attributes	ROA tables	Protocols	Otros	Total
EAS ASBR	65.67	31.33	0.19	177.87	36.68	311.74
FM ASBR	127	60	0.19	1202.47	90.07	1479.73
EAS Internos	62.95	25.97	0.19	156.96	32.11	278.18
FM Internos	127	60.05	0.19	1184.25	88.23	1459.72
EAS	63.03	26.12	0.19	157.56	32.24	279.14
FM	127	60.05	0.19	1184.77	88.28	1460.29

Resultados de prueba con topología de gran porte

Finalmente, utlizando una topología de gran porte, puede verse en la tabla 6.7 que el uso de memoria en los routers del EAS es considerablemente menor a la configuración full mesh, confirmando de esta forma las observaciones realizadas en las pruebas anteriores.

Siguiendo el criterio tomado para la topología de mediano porte, calculamos el grado promedio y la desviación estándar para los ASBRs y los routers internos. Por un lado, el grado promedio de los ASBRs es 4.33 y la desviación estándar 0.47, mientras que en los internos, el grado promedio es 2.69 y la desviación estándar 0.98. Dentro de los ASBRs tomamos a *Bari* (grado 4) y a *Marseille* (grado 5), y en los internos a *Oslo* (grado 2), *Munich* (grado 3) y *Rome* (grado 4).

En todos estos el tráfico observado es significativamente menor en el EAS. Podemos constatar nuevamente algunas observaciones ya realizadas como: el tráfico OpenFlow es el

Tabla 6.8 Tráfico en topología de gran porte (KB)

Router	AS	BGP	LLDP	OpenFlow	Tree Updates	Notificaciones	Total
Bari	EAS	12976	3366	74660	18044	12118	121164
	FM	2159495	0	0	0	0	2159495
Marseille	EAS	26208	6732	107883	11851	12248	164922
	FM	2461666	0	0	0	0	2461666
Oslo	EAS	0	3131	54856	14578	12798	85363
	FM	605220	0	0	0	0	605220
Munich	EAS	0	1387	43144	29413	12924	86868
	FM	2243660	0	0	0	0	2243660
Rome	EAS	0	6579	67390	24210	13266	111445
	FM	2175925	0	0	0	0	2175925

de mayor peso en el total (particularmente en los ASBRs), el tráfico LLDP no tiene un gran impacto aun con una topología de mayor tamaño, etc.

Es aquí que vemos el costo real de utilizar una configuración full mesh en una red con un número importante de routers, lo cual permite pensar que el uso de SDN para implementar alternativas al modelo full mesh es una opción viable.

Sección 7

Conclusiones

7.1 Conclusiones

Durante el transcurso del proyecto se buscó una alternativa al problema de escalabilidad de la configuración full mesh en un AS utilizando SDN. Luego de analizar diversas posibilidades se optó por una opción que combina el paradigma SDN con el método de distribución Multicast. Se diseñó una solución enfocada en garantizar la equivalencia funcional con respecto a la configuración full mesh. Se llevó a cabo su implementación y se obtuvieron resultados cuantitativos a partir de simulaciones.

De estos resultados concluímos que en un AS pequeño nuestra solución no aporta mejoras e incluso genera una cantidad de tráfico muy superior a la configuración full mesh. Esto es debido en gran medida a las sesiones OpenFlow requeridas para la comunicación con la controladora SDN.

Por otra parte, a medida que aumenta la cantidad de routers en el AS, comienza a resultar más atractiva la opción SDN, ya que el aumento cuadrático de las sesiones *iBGP* en una configuración full mesh hace que el tráfico aumente proporcionalmente, superando ampliamente al que se genera para mantener las sesiones OpenFlow y para distribuir los mensajes BGP por los árboles Multicast.

Sumado a lo anterior, los routers se ven beneficiados al reducirse la cantidad de memoria necesaria para funcionar correctamente. Vale mencionar que esto es cierto si el router utiliza como software de ruteo a Bird; no podemos asegurar que esto sea así para otros.

7.2 Dificultades

Inicialmente se necesitó de una etapa de asimilación de conocimientos sobre el paradigma SDN y emuladores de red que resultó ser más larga de lo esperado. De igual manera, por tratarse de un proyecto de investigación, se debió leer un gran número de artículos algunos de los cuales resultaron de poca relevancia para nuestro proyecto.

Otra dificultad importante fue la creación del entorno de emulación; en particular, utilizar Mininet para crear routers híbridos.

Por otra parte, las modificaciones realizadas al software de ruteo Bird requirieron el entendimiento de una gran parte de su código, lo cual no se vio ayudado por su documentación.

7.3 Mejoras

Si bien la solución implementada satisface los objetivos establecidos, podemos plantear cuatro mejoras puntuales.

Durante el establecimiento de una sesión BGP, los *peers* determinan la cantidad de bytes (2 o 4) utilizados para representar los ASNs [46]. La solución implementada asume que se utilizan siempre 4 bytes, dado que la instancia interna del protocolo BGP no tiene un *peer* con quién negociar dicho número. Idealmente, cuando el ASBR notifique a la controladora que ha establecido una sesión *eBGP*, deberá especificarle también con cuántos bytes se codificarán los ASNs. La controladora luego comunicaría esto al resto de los routers.

Actualmente, todo mensaje BGP UPDATE que llegue al EAS es enviado a la controladora, la cual mantiene una copia de los mismos. Consideramos que sería mejor que solo se envíen estos mensajes si efectivamente causan una modificación en la tabla de ruteo del ASBR. Para lograr esto, deberíamos permitir que Bird procese el paquete y, en caso de que su proceso de decisión haya seleccionado una nueva ruta, encapsule el mensaje BGP y lo envíe a la controladora (similar a cómo se envían las notificaciones de establecimiento de sesiones externas).

Por otra parte, somos conscientes que en un ambiente real, la acumulación de estos mensajes debe ser administrada de una forma más eficiente; por ejemplo, utilizando una base de datos.

La incorporación de los protocolos de distribución confiable al software de ruteo podría resultar beneficioso para realizar la distribución de mensajes BGP a través de los árboles multicast, ya que liberarían a la controladora de asegurar la entrega de los mensajes y delegarían esta responsabilidad a los routers. Esto, si bien simplificaría la lógica implementada por la controladora, como contrapartida acarrearía la necesidad de mayores recursos en los routers

para llevarlo adelante. En cuanto al tráfico en la red, dependiendo del protocolo utilizado y su funcionamiento, puede reducirse o aumentarse el tráfico general en la red frente a la implementación actual.

Referencias

[1] L. L. Karl Solie, *CCIE Practical Studies*, vol. 2. Cisco Press, 2003.

[2] B. S. Denise Donohue, "Ccnp routing and switching quick reference: Bgp and internet connectivity," *Cisco Press*, 2010.

[3] D. Software.

[4] V. Dario, "A survey of bgp session maintenance issues and solutions," vol. 2, no. 1, 2010.

[5] A. Narayanan, "A survey on BGP issues and solutions," *CoRR*, vol. abs/0907.4815, 2009.

[6] "What's software defined networking (sdn)? definition." http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2010-07-02.

[7] "What is the sdn?." http://fiberonsale.blogspot.com.uy/2015/07/what-is-sdn.html. Accessed: 2010-07-02.

[8] "Opflex: An open source approach." http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731304.pdf. Accessed: 2016-07-31.

[9] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (netconf)," RFC 6241, RFC Editor, June 2011. http://www.rfc-editor.org/rfc/rfc6241.txt.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[11] O. N. Foundation, "Openflow switch specification," tech. rep., Open Networking Foundation, June 2012. http://www.rfc-editor.org/rfc/rfc4271.txt.

[12] W. Stallings, "Software-defined networks and openflow," *The Internet Protocol Journal*, vol. 16, no. 1, 2013.

[13] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–98, Apr. 2014.

[14] W. Duan, L. Xiao, D. Li, Y. Zhou, R. Liu, L. Ruan, Y. Xia, and M. Zhu, "Ofbgp: A scalable, highly available bgp architecture for sdn," in *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems*, pp. 557–562, Oct 2014.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[16] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, (New York, NY, USA), pp. 5–12, ACM, 2004.

[17] C. Pelsser, A. Masuda, and K. Shiomoto, "A novel internal bgp route distribution architecture."

[18] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, (Berkeley, CA, USA), pp. 15–28, USENIX Association, 2005.

[19] R. Govindan, C. Alaettinoglu, K. Varadhan, and D. Estrin, "Route servers for inter-domain routing," in *Computer Networks and ISDN Systems 30*, pp. 1157–1174, 1998.

[20] S. Deering, "Host extensions for ip multicasting," STD 5, RFC Editor, August 1989. http://www.rfc-editor.org/rfc/rfc1112.txt.

[21] A. Greenhalgh, "Tree based ibgp replacement." No publicado.

[22] S. Q. Zhang, Q. Zhang, H. Bannazadeh, and A. Leon-Garcia, "Routing algorithms for network function virtualization enabled multicast topology on sdn," *IEEE Transactions on Network and Service Management*, vol. 12, pp. 580–594, Dec 2015.

[23] A. Menabo, *Evaluating SDN and SDN-based Multicast for Network Intensive Services in UNINETT*. PhD thesis, Norwegian University of Science and Technology, 2015.

[24] A. Iyer, P. Kumar, and V. Mann, "Avalanche: Data center multicast using software defined networking," in *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–8, Jan 2014.

[25] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (bgp-4)," RFC 4271, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4271.txt.

[26] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, and L. Vicisano, "Pgm reliable transport protocol specification," RFC 3208, RFC Editor, December 2001. http://www.rfc-editor.org/rfc/rfc3208.txt.

[27] J. Pradip M., A. Dr. M. S., F. Sakina, M.R.Parate, and J. Prof., "Reliable multicast transport protocol: Rmtp.," *International Journal of Advanced Computer Sciences and Applications*, no. 5, 2010.

[28] "Cisco application policy infrastructure controller." http://www.cisco.com/c/en/us/products/cloud-systems-management/application-policy-infrastructure-controller-apic/index.html. Accessed: 2016-07-31.

[29] "Hp virtual application networks." http://h17007.www1.hp.com/za/en/solutions/technology/van/index.aspx. Accessed: 2016-07-31.

[30] "Nec programmableflow pf6800." https://www.necam.com/sdn/doc.cfm?t=PFlowController. Accessed: 2016-07-31.

[31] "gevent." http://www.gevent.org/. Accessed: 2016-07-31.

[32] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of software defined networking (sdn) controllers," in *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pp. 1–7, Jan 2014.

[33] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of sdn/openflow controllers," in *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, (New York, NY, USA), pp. 1:1–1:6, ACM, 2013.

[34] "Quagga routing suite." http://www.nongnu.org/quagga/. Accessed: 2016-07-31.

[35] "The bird internet routing daemon." http://bird.network.cz/. Accessed: 2016-07-31.

[36] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," 2012.

[37] S. Salsano, P. L. Ventre, L. Prete, G. Siracusano, M. Gerola, and E. Salvadori, "Oshi - open source hybrid ip/sdn networking (and its emulation on mininet and on distributed sdn testbeds)," in *2014 Third European Workshop on Software Defined Networks*, pp. 13–18, Sept 2014.

[38] "Cloc." https://github.com/AlDanial/cloc. Accessed: 2016-08-31.

[39] "Networkx." https://networkx.github.io. Accessed: 2016-07-02.

[40] "Threading library." https://docs.python.org/2/library/threading.html. Accessed: 2016-07-02.

[41] "Mapping ip multicast to mac-layer multicast." https://technet.microsoft.com/en-us/library/cc957928.aspx. Accessed: 2016-07-02.

[42] "The topology zoo." http://www.topology-zoo.org/. Accessed: 2016-07-02.

[43] "Exabgp." https://github.com/Exa-Networks/exabgp/wiki. Accessed: 2016-07-31.

[44] "Ripe network coordination centre." https://www.ripe.net//. Accessed: 2016-07-31.

[45] "Wireshark." https://www.wireshark.org/. Accessed: 2016-07-31.

[46] Q. Vohra and E. Chen, “Bgp support for four-octet as number space,” RFC 4893, RFC Editor, May 2007.

[47] L. Blunk, M. Karir, and C. Labovitz, “Multi-threaded routing toolkit (mrt) routing information export format,” RFC 6396, RFC Editor, October 2011.

Anexo A

Máquina Virtual entregada

Para facilitar la ejecución de la prueba de concepto realizada entregamos una máquina virtual creada en Virtualbox 5.0.26 con las siguientes características:

- CPU: 4 CPUS
- RAM: 6 GB
- DISCO: 16 GB
- SO: XUbuntu 14.04 Kernel: 3.19.0 (a la fecha es requisito de mininet que la versión del kernel no sea superior a la versión 3.x).

La contraseña de usuario y root necesarias para ejecutar las pruebas es "usuario" sin las comillas.

A.1 Software pre-instalado

El software a destacar instalado en la máquina virtual es el siguiente:

- Ryu v3.26
- Mininet v2.1.0
- MiniNext v1.4.0
- tshark v1.10.6
- Wireshark v1.10.6

- Quagga Routing Suite v0.99.22.4
- Bird Internet Routing Daemon v1.5.0
- ExaBGP v3.4.16
- Otros (requeridos por dependencias): flex, bison, libncurses5-dev, libncursesw5-dev, python-webob, python-routes, python-eventlet, libssl-dev, libreadline6, libreadline6-dev, help2man, python-setuptools y python-pip.

A.2 Estructura de directorios

La entrega se encuentra en el directorio Proyecto del home del usuario. El mismo cuenta con la siguiente estructura:

bgpdump: contiene el código de la utilidad para traducir trazas del formato nativo de Ripe a MRT[47], necesario para poder utilizarlo en la inyección de rutas a los AS.

mininext: código fuente de la extensión mininext.

BIRD: agrupa el código fuente de las versiones de Bird utilizadas. Bird_EAS contiene el código utilizado para las simulaciones en el entorno del EAS, mientras que Bird_Original contiene el código de Bird versión 1.5.0 sin modificaciones.

Compilado: contiene los ejecutables de las versiones de Bird Original y Bird EAS contenidas en el directorio BIRD descrito anteriormente.

ryu-3.26: código de la controladora SDN Ryu. En el subdirectorio ryu/app se encuentra la aplicación Multicast_BGP.py desarrollada para la simulación del EAS.

Trazas: conjunto de archivos con trazas en formato MRT para ser utilizadas en la carga de rutas a los AS.

Pruebas: contiene el código de las pruebas desarrolladas así como los archivos de configuración para su ejecución. Dentro de cada uno de los subdirectorios con pruebas puede encontrarse dos directorios: Full Mesh y SDN; los mismos contienen los elementos mínimos necesarios para poder realizar una simulación. Para una prueba "X" Su estructura es definida de la siguiente manera:

- Archivos Quagga Conf: contiene los archivos de configuración para cada uno de los ASBR externos al AS a simular. También contiene la configuración utilizada por ExaBGP para la carga de trazas en el AS (o EAS).

- Archivos Bird Conf: contiene los archivos de configuración de los routers Bird presentes en el AS ó EAS.
- Memoria: en esta carpeta se guardan los resultados de la memoria consumida por cada router del AS simulado.
- Capturas: aquí se guardan las capturas realizadas con tshark en los routers del AS.
- Rutas Full Mesh/Rutas SDN: los archivos en este directorio son dumps de las tablas de rutas de cada uno de los routers del AS.
- Prueba_X.py: es el script que ejecuta la prueba en sí. En base al archivo topology.json construye la topología e incializa los routers según las configuraciones definidas en Archivos Quagga Conf y Archivos Bird Conf.
- Topologia.gml: archivo que describe la topología de la red a simular.
- PathConfig.json: se definen en él las configuraciones necesarias.
- topology.json: archivo generado a partir de Topologia.gml, contiene una definición más detallada de la topología que es utilizada por el script Prueba_X.py.
- Ryu_Log.txt: log de la aplicación Multicast_BGP corriendo en el controlador Ryu.

Adicionalmente a estos directorios, se pueden encontrar un conjunto de scripts Python con utilidades globales a todas las pruebas y que además facilitan la creación de nuevas pruebas:

comparar_rutas.py: se encarga durante la ejecución de las pruebas de proveer la funcionalidad de comparación de las tablas de rutas entre los casos SDN y Full Mesh.

ExaBGPBulkUpdate.py: script a ejecutarse en los routers externos (si la prueba lo requiere). A partir de una archivo de trazas, envía 100 mensajes UPDATE al AS simulado, con un leve intervalo entre cada mensaje.

ExaBGPLoopUpdate.py: similar al script anterior, con la diferencia de que el número de mensajes es solo 20 y el intervalo entre cada uno de éstos es mayor.

Generar_Configuracion.py: genera los archivos de configuración de los routers Quagga, Bird y ExaBGP en base a un archivo GML y un archivo de configuración PathConfig.json. La configuración generada puede ser utilizada para ejecutar una simulación de EAS. Como salida genera dos carpetas en el directorio donde está el archivo GML leído: Archivos Quagga Conf y Archivos Bird Conf.

Generar_Configuracion_Full_Mesh.py: similar al anterior, en este caso la configuración generada puede ser utilizada para ejecutar una simulación en un AS tradicional.

nodos.py: define los distintos tipos de routers disponibles para simular y las interfaces para utilizarlos desde los scripts de prueba.

Anexo B

Instrucciones para la ejecución

En este anexo describiremos el proceso para ejecutar las pruebas desarrolladas en el entorno de la máquina virtual descrita en el Anexo A.

B.1 Ejecución de pruebas

La ejecución de las pruebas puede ser llevada adelante posicionándose en el subdirectorio Prueba X/[Full Mesh | SDN] dentro del directorio Pruebas del proyecto.

Primero debe ejecutarse la versión full mesh para obtener los valores de las tablas de ruteo bajo esta configuración; luego se ejecuta la versión SDN, la cual comparará las tablas de los routers del EAS con las del caso anterior.

Una vez en el directorio corrsepondiente al tipo de prueba que se desee evaluar, el script es ejecutado de la siguiente forma:

```
sudo python Prueba_X.py
```

Durante la simulación se verán múltiples mensajes sobre los eventos que van ocurriendo. Para el caso SDN, al finalizar la simulación se desplegará la lista de resultados de la comparación router a router contra el caso full mesh.

Luego de completada la prueba, se retorna el control del terminal de mininet sobre la red simulada, desde donde es posible acceder a todos los routers de la red mediante el comando:

```
xterm nombre_router
```

y ejecutar los comandos que se desee (e.g. ping).

Para salir de la simulación basta con ejecutar el comando exit. Se recomienda que una vez hecho esto se ejecute el comando de *cleanup* de Mininet:

```
sudo mn -c
```

En las carpetas Memoria queda almacenado el registro de uso de memoria de los routers, mientras que en la carpeta Rutas Full Mesh o Rutas SDN, dependiendo de la simulación, quedan almacenados los *dumps* de las tablas de rutas de cada router.

El éxito de la prueba queda marcado por un mensaje "OK" por router de la topología en la lista de resultados; si alguna de las tablas de routers difiere se desplegará el mensaje "error" junto con la lista de diferencias entre las mismas.

B.2 Implementación de nuevas pruebas

Para la implementación de una nueva prueba se deben generar dos directorios, uno para la versión Full Mesh y otro para la versión SDN. Dentro de los mismos debe crearse un archivo de nombre PathConfig.json con la siguiente estructura:

```
{
        "ryuPath" : "dirección a la raíz del código del controlador
         ↪ Ryu",
        "appPath" : "dirección al código de la aplicación SDN
         ↪ Multicast_BGP.py",
        "baseDir" : "dirección al directorio donde los routers crearán
         ↪ sus directorios privados",
        "birdExecutionPath" : "dirección al directorio raíz donde se
         ↪ compiló Bird",
        "zebraExecutionPath" : "dirección al ejecutable zebrad",
        "bgpExecutionPath" : "dirección al ejecutable bgpd",
        "ospfExecutionPath" : "dirección al ejecutable ospfd",
        "exaBGPScript" : "dirección al script ExaBGPBulkUpdate.py,
         ↪ ExaBGPLoopUpdate.py u otra implementación de inyección de
         ↪ rutas con ExaBGP",
        "traceFile" : "dirección al archivo de trazas a utilizar para
         ↪ la inyección de rutas"
}
```

Las direcciones definidas en él son absolutas en todos los casos.

Adicionalmente a PathConfig.json, se debe agregar un archivo GML con la topología a simular. Este archivo debe cumplir con las siguientes restricciones:

- Los nombres de los routers no pueden exceder los 8 caracteres, no pueden contener espacios y no pueden ser iguales a *None*.
- No puede haber más de un enlace entre un par de routers.
- No puede haber enlaces cuyo origen y destino sea un único router.

Vale mencionar que un router es considerado interno o externo si el atributo *Internal* es igual a 1 o a 0 respectivamente.

Con los dos archivos anteriores, se deben ejecutar los scripts que generan las configuraciones para los routers de la red dependiendo del entorno a simular:

- Generar_Configuracion_Full_Mesh.py: en el caso full mesh.
- Generar_Configuracion.py: para el caso SDN.

La sintaxis de ejecución de los mismos es la siguiente:

```
python Generar_Configuracion_Full_Mesh.py direccion_de_gml
 ↪  direccion_de_PathConfig.json

python Generar_Configuracion.py direccion_de_GML
 ↪  direccion_de_PathConfig.json
```

Su ejecución generará el archivo topology.json el cual, además de tener una descripción de la topología definida en el archivo GML, tendrá un serie de atributos necesarios por cada router para la correcta ejecución de la prueba (IPs asignadas, número de AS, entre otros). También creará las carpetas Archivos Bird Conf y Archivos Quagga Conf dentro del directorio donde se encuentre el archivo GML pasado como parámetro.

En Archivos Quagga Conf además de las configuraciones para los routers Quagga se generan los archivos de configuración necesarios para que ExaBGP pueda realizar la inyección de rutas.

Para completar la prueba se debe programar un script que genere la topología de la red utilizando Mininet. Se puede tomar como base el script Prueba_1.py de la Prueba 1, el cual iniciará todos los routers Bird y Quagga con las configuraciones generadas en el paso anterior. En caso de desear realizar una inyección de rutas de un archivo de trazas, se recomienda tomar como base Prueba_2.py de la Prueba 2 y, si se desea, implementar un script python a ser utilizado por ExaBGP para anunciar rutas.