Facultad de Informática

PROYECTO DE SISTEMAS INFORMATICOS

CURSO 2010/2011

Incorporación de soporte nativo SNMP en el paquete Quagga.

Autores: Rubén Gamarra Rodríguez

Luis Villacastin Candil

Director: Juan Carlos Fabero Jiménez

Índice de contenido

1 AGRADECIMIENTOS

De Rubén: a mis padres y mi hermana por su apoyo y paciencia durante los años de mi periodo universitario, tanto en esta ingeniería superior como de la técnica. También a todas aquellas personas que me han aportado conocimientos que han resultado útiles para el desarrollo de este proyecto.

De Luis: Primero de todo, a los desarrolladores de Net-SNMP, y de Quagga, por sus respuestas en la lista de distribución ante mis dudas iniciales. A todos mis compañeros y amigos por su apoyo durante la carrera. Y a mis padres por permitirme estudiar una carrera.

2 PALABRAS CLAVE

QUAGGA
ZEBRA
SNMP
TRAP
ROUTING

3 RESUMEN / SUMMARY

Este proyecto de Sistemas Informáticos se planteó como una forma de aumentar las capacidades de la *suite* o paquete de aplicaciones de encaminamiento Quagga. Se pretende que el paquete de aplicaciones pueda entenderse de forma eficaz con SNMP (cuyas siglas provienen de *Simple Network Management Protocol* o protocolo simple de administración de redes) sin que sea necesaria la utilización de servidores o dispositivos *proxy* para llevar a cabo este fin.

El proyecto consiste en dotar a Quagga con la capacidad de enviar alertas, a través del protocolo SNMP, cada vez que el demonio principal Zebra añada o elimine una ruta de una red IPv4 al *kernel* o núcleo de una estación agente donde se esté ejecutando. Esas alertas se envían usando una notificación a través de un tipo mensaje especificado en SNMP llamado TRAP. La notificación será enviada automáticamente a una lista de estaciones gestoras cuyas direcciones han sido especificadas en la configuración de la estación agente.

De esta manera un administrador puede consultar y revisar los cambios producidos en las rutas de una estación agente desde una estación gestora sin tener que acceder a la estación agente directamente estableciendo una conexión.

Para el proyecto se ha utilizado la última versión estable de Quagga hasta mayo del año 2011, la versión 0.99.18, aunque los cambios realizados son muy extensibles, y convenientemente documentados, lo que facilita la posibilidad de que en futuras versiones oficiales de Quagga pueda realizarse esta misma mejora con relativa facilidad en caso de que la distribución oficial no lo contemple.

This end-of-studies project was posed as a way of increasing the capabilities of Quagga routing suite. The main goal is to provide the suite the mechanisms to communicate in an efficent way with SNMP (SNMP stands for Simple Network Management Protocol), while avoiding the use of any *proxy* server or *proxy* decive to accomplish this function.

The project consists in giving the Quagga software capabilities to send alerts through SNMP, whenever the main daemon, Zebra, adds or removes an IPv4 route to or from the device kernel of the agent station where it is running. These alerts are sent using a notification implemented as a SNMP message called TRAP. The notification will be send automatically to a set of management stations whose addresses have been entered in the agent station configuration.

This will allow an administrator to check or browse the changes occurred in an agent routing table from the management station without having to log in the agent station for operations, therefore no direct access to the agent station is needed.

During project development the last Quagga stable version (0.99.18 as May 2011) was used, although the changes are very extensibles and duly documented should future official Quagga versions do not include theese changes they can be included with relative ease.

4 INTRODUCCIÓN AL PROYECTO

Este proyecto consiste en dar la capacidad a la *suite* o paquete de aplicaciones de encaminamiento Quagga de enviar alertas, a través del protocolo SNMP, cada vez que el demonio principal Zebra añada o elimine una ruta al núcleo de una estación agente donde se esté ejecutando. Esas alertas se envían usando una notificación a través de un tipo mensaje especificado en SNMP llamado TRAP. La notificación será enviada automáticamente a una lista de estaciones cuyas direcciones han sido especificadas en la configuración de la estación que detecte los cambios.

El proyecto se dividirá en varios apartados, comenzando con la recolección de documentación sobre el software que va a ser modificado, en este caso *Quagga Routing Software Suite,* cuya licencia es GPL. Esto implica también realizar labores de documentación sobre los estándares y protocolos implementados dentro del software citado, así como su configuración y modos de funcionamiento. A continuación se describe la librería que implementa SNMP, *Net-SNMP*, software utilizado para dotar a *Quagga* capacidad para transmitir las notificaciones a través de una red.

Net-SNMP es una *suite* de software o paquete de aplicaciones que implementa los protocolos *SNMP v1*, *SNMP v2c* y *SNMP v3,* anteriormente conocida como UCD-SNMP con licencia BSD. Este paquete incluye, entre otras cosas, lo siguiente, que será explicado más adelante, un agente SNMP capaz de realizar todo tipo de consultas y un demonio SNMP capaz de recibir esas consultas y responderlas.

Aparte del demonio y el agente, también cuenta con varias herramientas de orientadas a SNMP, como un generador de alertas, un *handler* o manejador de esas alertas, un navegador de MIBs (que son las estructuras en las que se puede obtener y almacenar estos datos) aparte de otras aplicaciones relacionadas con la implementación del protocolo.

Una vez que la información y documentación acerca del software citado anteriormente fue recopilada se procedió con las fases de desarrollo, en las que se produce la incorporación de código a software existente a la vez que se realizan pruebas. En el caso de este proyecto se han realizado de forma entrelazada según avanzaba el desarrollo de las funciones o características a implementar.

Para el proyecto se ha utilizado la última versión estable de Quagga hasta mayo del año 2011, la versión 0.99.18, aunque los cambios realizados son muy extensibles, y convenientemente documentados en el código, lo que facilita la posibilidad de que en futuras versiones oficiales de Quagga pueda realizarse esta misma mejora con relativa facilidad en caso de que la distribución oficial no lo contemple.

4.1 Modelo OSI

Antes de iniciar cualquier otra introducción, se procederá a explicar en qué consiste el modelo OSI, para entender el funcionamiento general de cualquier arquitectura de red.

El modelo OSI *Open Systems Interconnection* o interconexión de sistema abiertos se propone con la finalidad de facilitar y maximizar la compatibilidad para intercambiar y transmitir información entre computadores. Al ser un sistema "abierto" para su comunicación con otros, sus especificaciones pueden ser consultadas por cualquier fabricante. Esta estandarización y especificación pretende favorecer la posibilidad de conectarse e intercambiar información entre sistemas de distintos fabricantes en virtud del mutuo uso de los estándares aplicables[1]. También recibe la denominación ISO OSI

El modelo se organiza en 7 capas. La figura 1 muestra de manera simplificada su estructura. Cada una de ellas define funciones específicas dentro de la red. La distribución en capas y el uso de un diseño de estas que sea coherente en conjunto permite que se cambie el contenido de una capa sin afectar a las demás.

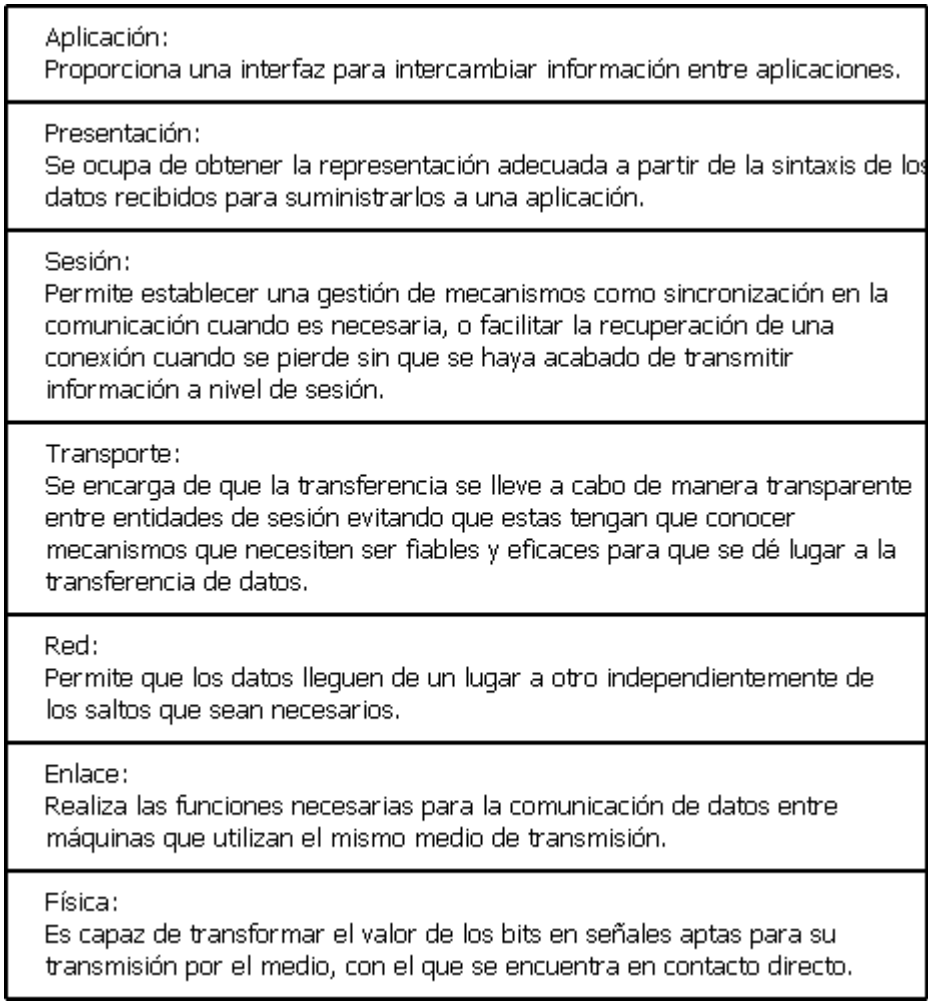

Figura 1: capas del modelo OSI.

Cuando se quiere enviar información los datos "atraviesan" hacia abajo las distintas capas en el extremo emisor de la conexión y recorren la pila de capas hacia arriba en el extremo receptor.

Cada capa ofrece lo que se denominan servicios a la capa superior y utiliza los de la capa inferior.

El modelos OSI no especifica los protocolos que se deben utilizar en cada capa, sino sus funciones. Existen otros modelos como TCP/IP[2].

4.1.1 Capa física

Representa la información en alguna magnitud física adecuada al medio de transmisión, por ejemplo, para el cable puede modula el voltaje o la intensidad de la corriente, para las ondas de radio su amplitud y frecuencia, para los pulsos de luz su modo y frecuencia. Transmite cadenas de bits sin entender su significado.

Debe implementar mecanismos para anunciar y negociar el tipo de flujo de la información que puede ser símplex, dúplex o semidúplex. Define cómo se establece y desmantela la comunicación. Debe especificar la funcionalidad de cada una de las patillas del conector.

4.1.2 Capa de enlace de datos

Los medios físicos no pueden ser considerados ideales ya que en la práctica se ven afectados por ruidos de diversa naturaleza.

La capa de enlace debe implementar mecanismos para intentar solventar o gestionar los siguientes tipos de errores:

- Errores de transmisión del medio físico.
- Agrupación de los datos o bits en tramas (marcos o *frames*).
  Se define como trama un conjunto de datos con estructura determinada.
- Control de flujo: un receptor con capacidades de transmisión lentas en comparación con un emisor debe poder solicitar el cese temporal de transmisión de datos.
- Si la red es un medio de difusión debe implementar mecanismos para detectar o evitar colisiones.
  Se define como colisión el fenómeno que se produce cuando dos o más estaciones están escribiendo información a un medio de difusión de manera que los destinatarios de esa información no serán capaces de obtener los datos de la manera prevista por el emisor.

4.1.3 Capa de red

Realiza funciones como:

- Encaminamiento de mensajes. Toma decisiones acerca de por dónde se enviará un mensaje desde un emisor a un receptor. Decide a qué nodos de una red entregar los datos.
- Interconexión de redes heterogéneas.
  Se define como red heterogénea aquella en la que se utilizan distintos medios de transmisión.

- Facturación: puede establecer métricas que definan la contabilización del tráfico de datos.
- Establecimiento y cierre de conexión entre máquinas.

La capa de red debe proporcionar independencia y abstracción de las capas inferiores.

4.1.4 Capa de transporte

Esta capa permite establecer una comunicación de proceso a proceso con el fin de que se produzca un envío de datos. El modelo OSI sitúa aquí las operaciones de fragmentación y reensamblado, aunque en la práctica, en la capa de red, protocolos como IP implementan soporte para fragmentación y reensamblado. Esta capa también debe ser capaz de gestionar errores. En modo de funcionamiento orientado a conexión debe permitir detectar y recuperar errores que se produzcan entre extremos de la conexión. En modo de funcionamiento no orientado a conexión deber permitir detectar errores que se produzcan entre extremos de la conexión. En ambos modos de funcionamiento debe supervisar la calidad del servicio (QOS *Quality Of Service*).

Se define como calidad de servicio de una conexión de red un conjunto de valores de las magnitudes utilizadas para medir aspectos como: disponibilidad, período de latencia, fluctuaciones en la señal y capacidad[3], de tal manera que el funcionamiento y rendimiento de la red sea el deseado.

Debe desempeñar tareas de control de tráfico mediante control de congestión (solicitar una pausa breve de envío de datos) y control de flujo (solicitar el cese de envío de datos).

Puede ofrecer 4 tipos de servicios:

- No fiable sin conexión. (Solicitudes, preguntas y respuestas, resolución de nombres de dominio, sincronización de relojes entre estaciones).
- No fiable con conexión. (Retransmisión de contenido multimedia en vivo o s*treaming*)
- Fiable sin conexión. (Envío de información con acuse de recibo, códigos intransferibles de un solo uso).
- Fiable con conexión. (Envío de ficheros).

Se define como conexión aquel proceso de comunicación entre dos dispositivos o estaciones en el que previamente se debe establecer la comunicación entre extremos con la finalidad de permitir recibir y discriminar el orden de los datos a transferir.

Se define como fiable la capacidad de la capa de transporte para corregir errores derivados de la pérdida de datos que se producen cuando éstos no son recibidos en el destino.

4.1.5 Capa de sesión

Se encarga de facilitar que los usuarios establezcan conexiones. Una sesión establece el control de una conexión. Establece mecanismos necesarios para desempeñar

funciones como sincronización en la comunicación cuando es necesaria o facilitar la recuperación de una sesión cuando se produce la pérdida de conexión a nivel de sesión de manera no solicitada por ninguno de los extremos y es necesario continuar con la transmisión de información.

4.1.6 Capa de presentación

Esta capa es la encargada de añadir la estructura a los datos en un formato conveniente como pueden ser ASCII, ISO, UTF, EBCDIC, XDR, etc. En el caso de que los datos se pretendan transmitir de manera cifrada especificará el convenio o función que se debe usar.

4.1.7 Capa de aplicación

Proporciona una interfaz para intercambiar información entre aplicaciones o usuarios.

4.2 Introducción al encaminamiento

Las principal función de la capa de red es el encaminamiento. El encaminamiento consiste en la correcta toma de decisiones con el fin de seleccionar un ruta óptima por la que se puede enviar cada datagrama de manera que este llegue a su destino.

4.2.1 Requisitos de un algoritmo de encaminamiento

- Ha de ser correcto. Para toda entrada o situación la salida o resultados han de ser los esperados.
- Sencillo. Debe mantener un rendimiento aceptable frente a una carga elevada de peticiones por unidad de tiempo.
- Robusto. Su funcionamiento debe ser capaz de contemplar situaciones en las que se produzcan fallos en la red.
- Estable. En situaciones en las que las condiciones de la red no cambien, la elección de rutas tampoco deberá hacerlo.
- Equitativo. No debe priorizar o favorecer a determinadas máquinas sobre otras.
- Óptimo. Debe encontrar los mejores valores en según criterios como número de saltos, tiempo, coste económico o fiabilidad.

4.2.2 Clases de algoritmos

Los algoritmos de encaminamiento se pueden clasificar en 3 tipos; estáticos, dinámicos e inundación. Los estáticos y dinámicos están basados en tablas que es la estructura que utilizan para albergar la información de encaminamiento.

4.2.2.1 Estáticos

Son aquellos algoritmos que trabajan con información que no va a cambiar con el paso del tiempo.

4.2.2.2 Dinámicos

Son aquellos algoritmos capaces de construir las tablas a partir de información proveniente de otros encaminadores. A su vez los algoritmos dinámicos se pueden clasificar en dos tipos: **estado del enlace** y **vector distancia** (Bellman-Ford).

Los algoritmos de estado del enlace son útiles con distintas métricas. En los algoritmos de vector distancia se envía una considerable cantidad de información pero sólo a los encaminadores vecinos.

En la sección 5.1.3 se puede encontrar más información sobre implementaciones concretas de algoritmos de encaminamiento dinámicos.

4.2.2.3 De inundación

Son aquellos algoritmos en los que los datagramas o mensajes se envían por varios enlaces a la vez (o incluso todos los que estén disponibles en el encaminador). De esta manera al menos uno de los datagramas utilizará siempre la ruta óptima. Al recorrerse todas las rutas se garantiza que el algoritmo es robusto. Se evita que los datagramas recorran enlaces de manera indefinida gracias a mecanismos como el TTL tiempo de vida (*Time To Live*) o retransmisión selectiva.

Algunas de las posibles aplicaciones de este tipo de algoritmos de encaminamiento pueden ser aplicaciones de ámbito militar.

4.3 Fases del proyecto

El proyecto se dividirá en una serie de fases, cuya descripción puede resumirse en:

- Documentación: Período durante el cual se hace acopio de información relativa al software utilizado, así como de sus principales características, y de aquellas propiedades que queremos agregar al software existente.
- Desarrollo: Se comienza en esta fase con una serie de preparativos relacionados con el desarrollo, tales como el diseño de máquinas virtuales que albergarán los diferentes procesos software, así como la aplicación que usaremos para programar y modificar, en este caso, la *suite* o paquete de aplicaciones Quagga, para que acepte el código nuevo que le será añadido y así ampliar la funcionalidad ya descrita. Al mismo tiempo, se extrae de la la *suite* o paquete de aplicaciones Net-SNMP, la parte del código necesaria, se modifica y adapta. Paralelamente al desarrollo, deben realizarse una serie de pruebas, para comprobar que los cambios que se van realizando funcionan correctamente y no se interponen en el funcionamiento normal del software original.

4.4 Estado del arte

El software Quagga no soporta de manera oficial SNMP. Sin embargo, la comunidad de desarrolladores de SNMP diseñó en 1991 un protocolo nuevo y provisional, que permite dotar

a los dispositivos sin soporte para SNMP de la posibilidad de utilizar sus funcionalidades. Esto se logra mediante SMUX, que se encuentra definido en el RFC 1227 *"SNMP MUX protocol and MIB"*[4], el cual está catalogado como histórico, debido a que casi todos los dispositivos que son puestos a la venta disponen y soportan SNMP, por lo que no es necesario el uso de SMUX.

La ausencia de soporte SNMP de la algunos dispositivos se debe a que el uso de SNMP no estaba extendido o no se había definido aún cuando fueron diseñados. Las estaciones que ejecuten SMUX llevan a cabo funciones de *proxy* o adaptador entre los mensajes SNMP y los dispositivos a administrar. Aún así, las limitaciones son importantes ya que en la práctica sólo se permiten un conjunto reducido de operaciones básicas de lectura.

Aparte de lo expuesto anteriormente, el uso de SMUX no permite una de las funcionalidades más extendidas y demandadas a la hora de administrar dispositivos de red de las que dispone SNMP, el envío de mensajes o alertas de tipo TRAP.

En la documentación oficial de Quagga se explica cómo usar el soporte SMUX, y al mismo tiempo, qué se debe hacer para que el agente SNMP pueda comunicarse con el proceso SMUX de Quagga.

Sobre la implementación del envío de mensajes TRAP de SNMP desde Quagga no se ha encontrado nada relevante para nuestro proyecto, tan solo algunos intentos de desarrolladores de Quagga de añadir el envío de alertas usando el protocolo SMUX, aunque sólo para uno de sus demonios de encaminamiento, en concreto el que implementa OSPF[5]. En este caso, se envían *traps* cuando se realizan algunos cambios en los estados del OSPF, lo cual dista del objetivo de este proyecto, enviar alertas cuando se añadan o eliminen redes al *kernel* o núcleo del sistema operativo mediante el software Quagga.

Entre otros proyectos basados en ampliar Quagga, destacamos el QuaggaMR, también conocido como *Quagga Mobile Routing*. Consta de un Quagga con extensiones para encaminamiento móvil, añadiendo un nuevo demonio que implementa una modificación de OSPFv3 llamada OSPF-MANET, para su uso en redes móviles *ad hoc*[6].

5 DOCUMENTACIÓN

5.1 QUAGGA

Quagga es un paquete de software de encaminamiento formado por diferentes demonios que soportan los protocolos RIPv1, RIPv2, RIPng, OSPFv2, OSPFv3, BGP-4 y BGP-4+. Los protocolos de encaminamiento basados en IPv6 de los citados anteriormente son RIPng y OSPFv3. Quagga también soporta el comportamiento especial de BGP *Route Reflector* y *Route Server.*

Utiliza una arquitectura de software avanzada para proporcionar una gran calidad, con un motor multiservidor de encaminamiento. Tiene un interfaz de usuario interactivo para cada protocolo de encaminamiento. Debido a su diseño, es posible añadir nuevos demonios de protocolos dentro del paquete y comunicarse con el demonio principal Zebra. Zebra se puede utilizar también como librería para un programa cliente de interfaz de usuario.

Quagga es una bifurcación o *fork* de la *suite* o paquete de aplicaciones GNU Zebra, el cual es un software oficial GNU y está distribuido bajo la Licencia General Pública GNU. La ultima versión liberada de GNU Zebra data del año 2005[7].

5.1.1 Arquitectura de Quagga

El software tradicional de encaminamiento está compuesto por un programa o proceso único que proporciona todas las funcionalidades de los protocolos de encaminamiento. Quagga, sin embargo, tiene una visión distinta. Está compuesto por una colección de varios demonios que trabajan juntos para construir una tabla. Hay varios demonios de encaminamiento específicos que se ejecutan junto con zebra, el *kernel* o núcleo gestor del encaminamiento.

El demonio *ripd* maneja el protocolo RIP, mientras que el demonio *ospfd* controla el protocolo OSPFv2. *bgpd* soporta el protocolo BGP-4. Para realizar cambios en la información de la tabla de encaminamiento del núcleo del sistema operativo y la redistribución de rutas entre distintos protocolos de encaminamiento tenemos el demonio *zebra* como se muestra en la figura 2. Es relativamente sencillo añadir demonios que implementen nuevos protocolos de encaminamiento al paquete de encaminamiento sin afectar a otras partes del software.

Para ello hay sólo es necesario ejecutar los demonios asociados a los protocolos de encaminamiento a utilizar. Realizando esta operación, el usuario puede ejecutar un determinado demonio y enviar informes a la consola central del software de encaminamiento. No es necesario tener en ejecución, en una máquina, los demonios de encaminamiento de aquellos protocolos que no se vayan a utilizar. Esta arquitectura pretende aumentar la flexibilidad de configuración del software de encaminamiento.

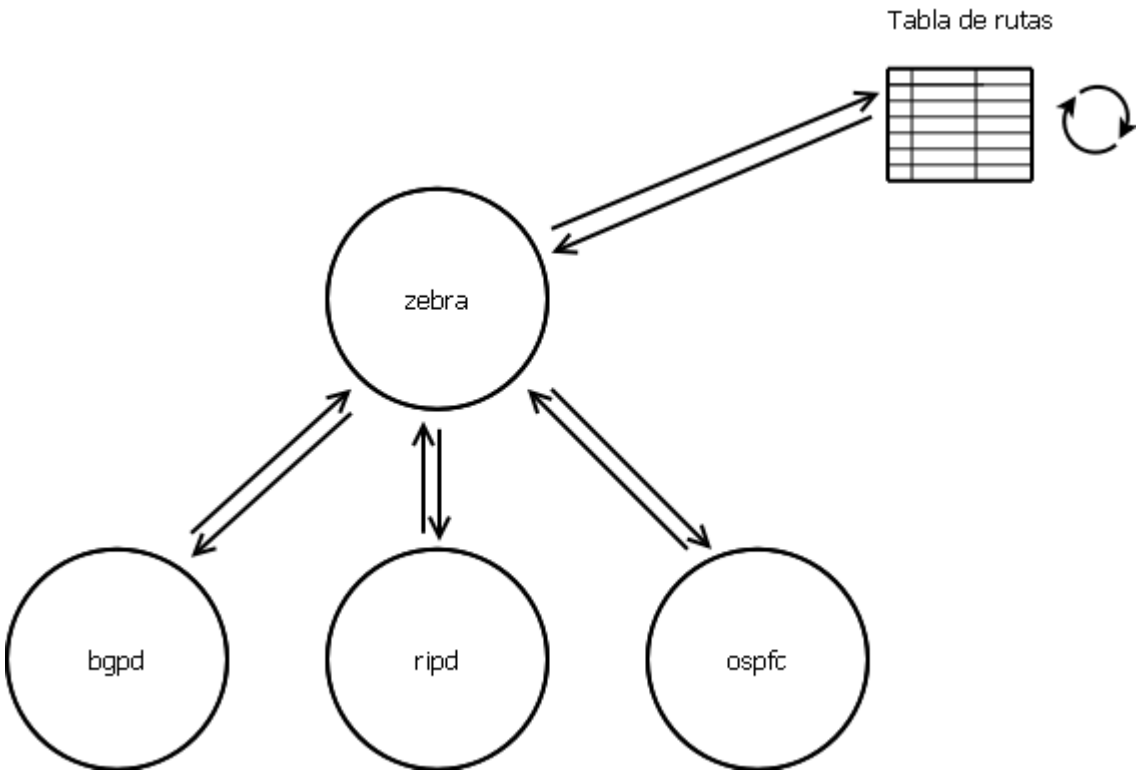

Figura 2: Arquitectura del sistema Quagga.

El demonio principal y gestor Zebra, funciona como un servidor, el cual escucha las peticiones y mensajes de los clientes, que son los demonios que implementan los protocolos de encaminamiento. Estos clientes se comunican con Zebra a través de un protocolo de mensajes propio de Quagga, cuyo formato ya está explicado en la documentación oficial[8].

La arquitectura multiproceso nos permite un sistema más fácilmente extensible y gestionado y por supuesto nos permite un sistema totalmente modular, a la vez que distribuye la configuración en varios ficheros.

Hay dieciséis (enumerados en la tabla 1) tipos de mensajes que utilizan el servidor (demonio Zebra) y los clientes (resto de demonios).

COMANDO/ORDEN	VALOR
ZEBRA_INTERFACE_ADD	1
ZEBRA_INTERFACE_DELETE	2
ZEBRA_INTERFACE_ADDRESS_ADD	3
ZEBRA_INTERFACE_ADDRESS_DELETE	4
ZEBRA_INTERFACE_UP	5
ZEBRA_INTERFACE_DOWN	6
ZEBRA_IPV4_ROUTE_ADD	7
ZEBRA_IPV4_ROUTE_DELETE	8
ZEBRA_IPV6_ROUTE_ADD	9
ZEBRA_IPV6_ROUTE_DELETE	10
ZEBRA_REDISTRIBUTE_ADD	11
ZEBRA_REDISTRIBUTE_DELETE	12
ZEBRA_REDISTRIBUTE_DEFAULT_ADD	13
ZEBRA_REDISTRIBUTE_DEFAULT_DELETE	14
ZEBRA_IPV4_NEXTHOP_LOOKUP	15
ZEBRA_IPV6_NEXTHOP_LOOKUP	16

Tabla 1: Tipos de mensajes utilizados en la comunicación con zebra.

Para nuestro propósito nos interesan aquellos mensajes de tipo:
ZEBRA_IPV4_ROUTE_ADD y ZEBRA_IPV4_ROUTE_DELETE

Como veremos más adelante, en la fase de desarrollo, el objetivo será interceptar esos mensajes, y utilizar su información para enviar la alerta.

Al disponer cada demonio de su propio fichero de configuración e interfaz de terminal, cuando se quiere configurar una ruta estática, ésta se configura en el fichero de configuración *zebra*. Para configurar una red BGP se debe hacer en el fichero de configuración *bgpd*, y para el resto de protocolos se procede de manera análoga, lo cual en algunos casos podría convertirse en una labor tediosa. Para solucionar este problema existe un interfaz *shell* integrada llamado *vtysh*. A través de *vtysh* se puede conectar cada demonio que se encuentre en funcionamiento como un *proxy* o adaptador que recibe la entrada del usuario. Esta interfaz es muy similar a la utilizada en otro conjunto de software de encaminamiento, como por ejemplo CLI de CISCO[9].

5.1.2 Plataformas soportadas

Actualmente, Quagga soporta GNU/Linux, BSD y Solaris. La lista oficial es la siguiente:

- GNU/Linux 2.4.x – en adelante.
- FreeBSD 4.x – en adelante.
- NetBSD 1.6 – en adelante.

- OpenBSD 2.5 – en adelante.
- Solaris 8 – en adelante.

El software Quagga, gracias a su relativa facilidad para ser migrada a otros entornos, puede funcionar en otras plataformas que no aparecen en la lista anterior, aunque su funcionalidad en estos casos puede ser limitada dependiendo de la compatibilidad del sistema con el diseño de Quagga[8].

En el apéndice B se pueden consultar los RFCs actualmente soportados por el paquete Quagga.

5.1.3 Protocolos

Como se ha descrito anteriormente Quagga está formado por varios demonios de encaminamiento más uno, zebra, que es el gestor encargado de procesar y comunicar la información entre la tabla de rutas del *kernel* o núcleo del sistema operativo y estos demonios.

Cada uno de esos demonios de encaminamiento implementa un protocolo. A continuación se describe brevemente el algoritmo de encaminamiento que implementan:

5.1.3.1 RIP:

Este protocolo es implementado por el demonio *ripd* de Quagga. RIP son las siglas en inglés de protocolo de información de encaminamiento (*Routing Information Protocol*). Es un protocolo de puerta de enlace interna o IGP (*Internal Gateway Protocol*).

El origen del RIP fue el protocolo de Xerox, GWINFO. Una versión posterior fue conocida como *routed*, distribuida con BSD Unix, distribución estándar de Berkeley (*Berkeley Standard Distribution*) en 1982[10]. RIP evolucionó como un protocolo de encaminamiento de Internet, y otros protocolos propietarios utilizan versiones modificadas de RIP. El protocolo *AppleTalk Routing Table Maintenance Protocol* (RTMP) y el *Banyan VINES Routing Table Protocol* (RTP), por ejemplo, están los basados en una versión del protocolo de encaminamiento RIP.

La última mejora o actualización hecha a RIP es la especificación RIPv2, que permite incluir más información en los paquetes RIP y provee un mecanismo de autenticación muy simple.

En la actualidad existen tres versiones diferentes de RIP, las cuales son:

RIPv1: No soporta subredes ni direccionamiento CIDR. Tampoco incluye ningún mecanismo de autenticación para los mensajes. Es un protocolo de encaminamiento basado en clases. No goza de un amplio uso en la actualidad.

RIPv2: Soporta subredes, CIDR y VLSM. Soporta distintos tipos de autenticación: no autenticación, autenticación mediante contraseña y autenticación

mediante contraseña cifrada mediante MD5 (desarrollado por Ronald Rivest). Su especificación está recogida en RFC 4822 "*RIPv2 Cryptographic Authentication*"[11].

RIPng: RIP siguiente generación para redes IPv6. Su especificación está recogida en el RFC 2080 "*RIPng for IPv6*"[12].

El protocolo RIP es una implementación directa del encaminamiento por vector-distancia. Utiliza UDP a través del puerto 520. Calcula el camino de acuerdo a una métrica establecida en la configuración del encaminador, que es el número de saltos, minimizando este valor.

MODO DE OPERACIÓN:

Cuando RIP se inicia, envía un datagrama a cada uno de sus vecinos (hacia el puerto bien conocido número 520) pidiendo una copia de la tabla de encaminamiento del vecino. Este mensaje es una solicitud (el campo *command* tiene el valor 1) con *address family* vale 0 y *metric* vale 16. Los encaminadores vecinos envían una respuesta a partir de sus tablas de encaminamiento.

Si está en modo activo envía toda o parte de su tabla de encaminamiento a todos los vecinos por *broadcast* o difusión en RIPv1 o multidifusión en RIPv2 y RIPng y/o mediante el uso de enlaces punto a punto. Esto se hace cada 30 segundos. La tabla de encaminamiento se envía como respuesta (*command* vale 2, aunque no haya habido petición).

Cuando RIP descubre que una métrica ha cambiado, la transmite a los demás encaminadores. En el momento en que se recibe una respuesta el mensaje se valida y la tabla local se actualiza si fuese necesario. Para mejorar el rendimiento y la fiabilidad, RIP especifica que una vez que un encaminador ha aprendido una ruta a través de otro encaminador, debe guardar la misma hasta que conozca una mejor (de coste estrictamente menor); esto evita que entradas de las tablas de rutas cambien o oscilen entre dos o más rutas de igual coste.

Si recibe una petición, distinta de la solicitud de su tabla, se devuelve como respuesta la métrica para cada entrada de dicha petición fijada al valor de la tabla local de encaminamiento. Si no existe ruta en la tabla local, se pone a 16 que es el valor que codifica una ruta como inalcanzable o distancia infinita.

Las rutas que RIP aprende de otros encaminadores caducan o expiran a menos que se vuelvan a recibir dentro de un periodo de 180 segundos (tiempo equivalente a 6 ciclos de difusión). Cuando una ruta expira, su métrica se pone a infinito, y la invalidación de la ruta se envía a los vecinos y 60 segundos más tarde se borra de la tabla.

5.1.3.2 RIPng

Este protocolo es implementado por el demonio *ripngd* de Quagga.

RIPng (siguiente generación de RIP), definido en RFC 2080, es una extensión de RIPv2 para soportar IPv6, la siguiente generación del protocolo de Internet. Las principales diferencias entre RIP y RIPng son:

- Soporta redes IPv6.
- Mientras RIPv2 soporta actualizaciones de autenticación de RIPv1, RIPng no lo hace. Los encaminadores de IPv6, ya usan IPsec, por tanto no hace falta esta capacidad.

5.1.3.3 OSPFv2

Este protocolo es implementado por el demonio *ospfd* de Quagga.

El protocolo está descrito en el RFC 2238[13] *"Definitions of Managed Objects for HPR using SMIv2"*. OSPF, camino abierto más corto primero (*Open Shortest Path First)* es un protocolo de encaminamiento jerárquico de pasarela interior o IGP (*Interior Gateway Protocol*), que usa el algoritmo Dijkstra de estado del enlace (LSA *Link State Algorithm*) para calcular la ruta más corta posible. Utiliza el coste de la ruta como su medida para la métrica. Además, construye una base de datos estado del enlace (*link-state database*, LSDB) con información idéntica en todos los encaminadores del área.

5.1.3.4 OSPFv3

Este protocolo es implementado por el demonio *ospf6d* de Quagga.

Es un protocolo de encaminamiento por estado del enlace el cual fue descrito por primera vez en el RFC 5340 *"OSPF for IPv6"*[14]. El protocolo OSPFv3 trabaja con direcciones IPv6, distribuyendo por la red solamente el prefijo de estas direcciones. No posee soporte para direcciones IPv4, razón por la cual si se desea tener dentro de la misma red direcciones IPv6 y direcciones IPv4 se deben configurar tanto el protocolo OSPFv2 como OSPFv3.

El protocolo OSPFv3 tiene los mismos fundamentos que el protocolo OSPFv2 (Algoritmo SPF, inundaciones, elección del DR, áreas, métricas, temporizadores), pero a pesar de tener ciertas similitudes también poseen diferencias entre las cuales tenemos:

OSPFv3 trabaja sobre un enlace en vez de hacerlo sobre subred.
La topología OSPFv2 no soporta el protocolo IPv6.
OSPFv3 posee un mecanismo de autenticación (RFC 4552 *"Authentication / Confidentiality for OSPFv3"*).
OSPFv3 posee múltiples instancias por enlace.

5.1.3.5 BGP-4

Este protocolo es implementado por el demonio *bgpd* de Quagga.

Existe numerosa documentación sobre este protocolo de la cual facilitamos algunas referencias[15][16], por lo que se muestra a continuación una introducción a BGP.

El protocolo BGP se ha convertido en el principal protocolo de encaminamiento externo utilizado en Internet. Casi todo el tráfico que fluye entre los ISPs es encaminado a través de BGP. La versión actual, BGP-4, se encuentra descrita en el RFC 4271[15].

Con el fin de reducir el tamaño de las tablas de encaminamiento y de facilitar su gestión, Internet se encuentra dividido en sistemas autónomos (AS).

Un sistema autónomo es un conjunto de redes administradas por una misma organización que tiene definida una política de encaminamiento. Esta política de encaminamiento decide las rutas admitidas desde los sistemas autónomos vecinos y las rutas que se envían hacia estos sistemas autónomos. En su interior, el AS utiliza un protocolo interno de encaminamiento como, por ejemplo, OSPF. El protocolo BGP un protocolo de encaminamiento entre sistemas autónomos.

Inicialmente, cada sistema autónomo en Internet tenía un identificador (ASN) formado por 16 bits, lo que permitía hasta 65536 sistemas autónomos teóricos diferentes, si bien el rango de 64512 a 65535 se encontraba reservado para uso privado. Pero a partir del año 2007 se empezó a formar identificadores de 32 bits, ya que con el rango anterior, los ASN libres empezaron a escasear.

Las tablas de encaminamiento de BGP-4 almacenan rutas para alcanzar redes, más concretamente prefijos de cierto número de bits. Las rutas están formadas por una secuencia de números de sistemas autónomos que se deben seguir para alcanzar el prefijo indicado. El último número de AS de la ruta se corresponde con la organización que tiene registrado el prefijo. El principal motivo para almacenar la ruta completa es la detección y eliminación de bucles, es decir, evitar que los paquetes se reenvíen de forma infinita entre unos mismos sistemas autónomos sin alcanzar nunca el destino.

Según el número de conexiones con otros sistemas autónomos y las políticas definidas, un sistema autónomo puede ser de diferentes tipos. El más sencillo (denominado stub AS) tiene una única conexión con otro AS, que será normalmente su ISP o proveedor de servicios de internet. Por este sistema autónomo únicamente circula tráfico local. Si el AS tuviese más de una conexión a otros sistemas, por motivos de redundancia generalmente, se denominaría *multihomed*. El tráfico que circula dentro del AS seguiría siendo local. Por último, un sistema autónomo de tránsito es un sistema con varias conexiones, el cual reenvía tráfico de una conexión a

otra. Por supuesto, los sistemas autónomos pueden decidir los tipos de tráfico que transportan, mediante el establecimiento de políticas.

5.2 Net – SNMP

Net-SNMP, es un conjunto de aplicaciones utilizadas para implementar SNMP v1, SNMP v2c y SNMP v3 usando IPv4 y/o Ipv6. Anteriormente era conocido como UCD – SNMP (*University of California, Davis*) y tiene licencia BSD.

Net-SNMP incluye:

- Aplicaciones de línea de comandos:
  - → Recuperar información de un dispositivo que soporte SNMP, ya sea mediante peticiones individuales (*snmpget*, *snmpgetnext*), o múltiples solicitudes (*snmpwalk*, *snmptable*, *snmpdelta*).
  - → Manipular la información de configuración en un dispositivo con capacidad SNMP (*snmpset*).
  - → Recuperar una colección fija de información de un dispositivo que soporte SNMP (*snmpdf*, *snmpnetstat*, *snmpstatus*).
  - → La conversión entre las formas textuales y numéricas de las tablas MIB dando un OID (*snmptranslate*).

- Un visor gráfico MIB (TKMib), usando Tk / perl.
- Una aplicación de demonio para recibir notificaciones SNMP (snmptrapd).
- Un agente extensible para responder consultas SNMP y gestionar la información (snmpd). Esto incluye soporte integrado para una amplia gama de módulos de información MIB, y se puede ampliar usando los módulos cargados dinámicamente, scripts externos, comandos, y usando la multiplexación SNMP (SMUX) o Agente de extensibilidad (protocolo AgentX).
- Una biblioteca para el desarrollo de nuevas aplicaciones SNMP, tanto en C como en Perl.

5.2.1 SNMP

SNMP son la siglas de *Simple Network Management Protocol* o Protocolo Simple para Gestión de Redes.

Este protocolo se utiliza para gestionar y configurar estaciones que poseen capacidad de conexión a una red, ya sea pública como Internet o privada. Para ello se basa en la existencia de 2 tipos de estaciones o dispositivos: los agentes y los gestores. SNMP se sitúa en la capa de aplicación (capa 7) según el modelo OSI. El método mas común de intercambio de información para SNMP es mediante datagramas UDP.

Dentro de un entorno de red gestionado con SNMP, habrá un conjunto de nodos que se encarguen de la gestión y un conjunto de componentes de la red (*hosts* o estaciones*,* módem*, routers* o encaminadores*, switchs* o conmutadores, etc.), que podrán

ser gestionados por estas estaciones. En la figura 3 se muestra de manera simplificada cómo se comunican las estaciones.

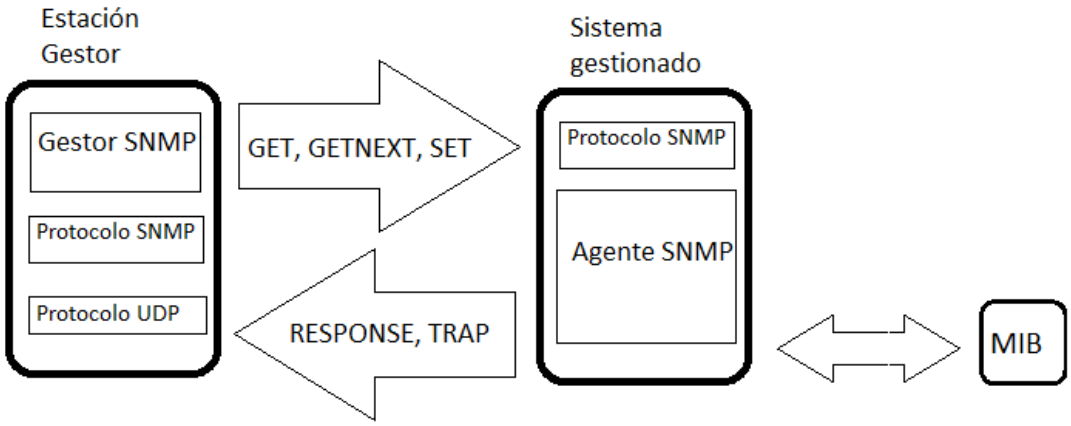

Figura 3: Esquema general del funcionamiento de SNMP.

Para que el software encargado de la gestión de la red en las estaciones de gestión pueda obtener información de los elementos de la red, es necesario que dichos elementos cuenten con un software que permita su comunicación con la estación de gestión. Este software se denomina *agente*.

Por último, hay otra pieza importante en este entorno, y es la base de datos donde se encuentra toda la información que se gestiona. Esta base de datos se denomina MIB (*Management Information Base*).

5.2.2 Tipos de dispositivos

Por lo tanto, tal y como se ha visto, dentro de SNMP, debemos distinguir los dispositivos involucrados en la ejecución del protocolo.

- Agente: Se encarga de supervisar un elemento de la red. Es aquel proceso capaz de informar y atender (por defecto escucha en el puerto 161) peticiones provenientes de un gestor. Para ello puede usar notificaciones de tipo "trap" dirigidas a un gestor o responder a esas peticiones de forma síncrona. Suele residir físicamente en el elemento gestionado.

- Gestor: Es un proceso que se comunica con un agente y que ofrece al usuario una interfaz para obtener información de los recursos gestionados. Además recibirá las notificaciones o alertas enviadas por los agentes. Por defecto escucha en el puerto 162.

5.2.3 MIB

La Base de Información de Gestión (*Management Information Base* o *MIB*) es un tipo de base de datos que contiene información jerárquica, estructurada en forma de árbol, de todos los dispositivos gestionados en una red de comunicaciones. Es parte de la

gestión de red definida en el modelo OSI. Define las variables usadas por el protocolo SNMP para supervisar y controlar los componentes de una red.

Está compuesta por una serie de objetos que representan los dispositivos (como encaminadores y conmutadores) en la red. Cada objeto manejado en un MIB tiene un identificador de objeto único e incluye el tipo de objeto (tal como contador, o secuencia), el nivel de acceso (tal como lectura y escritura), restricciones de tamaño, y la información del rango del objeto.

Un nodo se identifica unívocamente con una secuencia de números enteros que identifican los nodos a través de los cuales hay que pasar para llegar desde la raíz al nodo que nos interese.

La MIB-II es la base de datos común para la gestión de equipos en Internet, la cual se ha actualizado en numerosas ocasiones. Originalmente estaba definida en el RFC 1213. Con la aparición de SNMPv2 y SNMPv3 esta MIB se amplió y se dividió en varios RFCs:

- RFC 4293 *"Management Information Base for the Internet Protocol (IP)"[17]*.
- RFC 4022 *"Management Information Base for the Transmission Control Protocol (TCP)"[18]*.
- RFC 4113 *"Management Information Base for the User Datagram Protocol (UDP)"[19]*.
- RFC 2863 *"The Interfaces Group MIB"[20]*.
- RFC 3418 *"Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)"*[21].

Se apoya en el modelo de información estructurada definido en el RFC 1155 [2], que establece las bases para definir la MIB, indica los tipos de objetos que se pueden usar y define el uso de ASN.1.

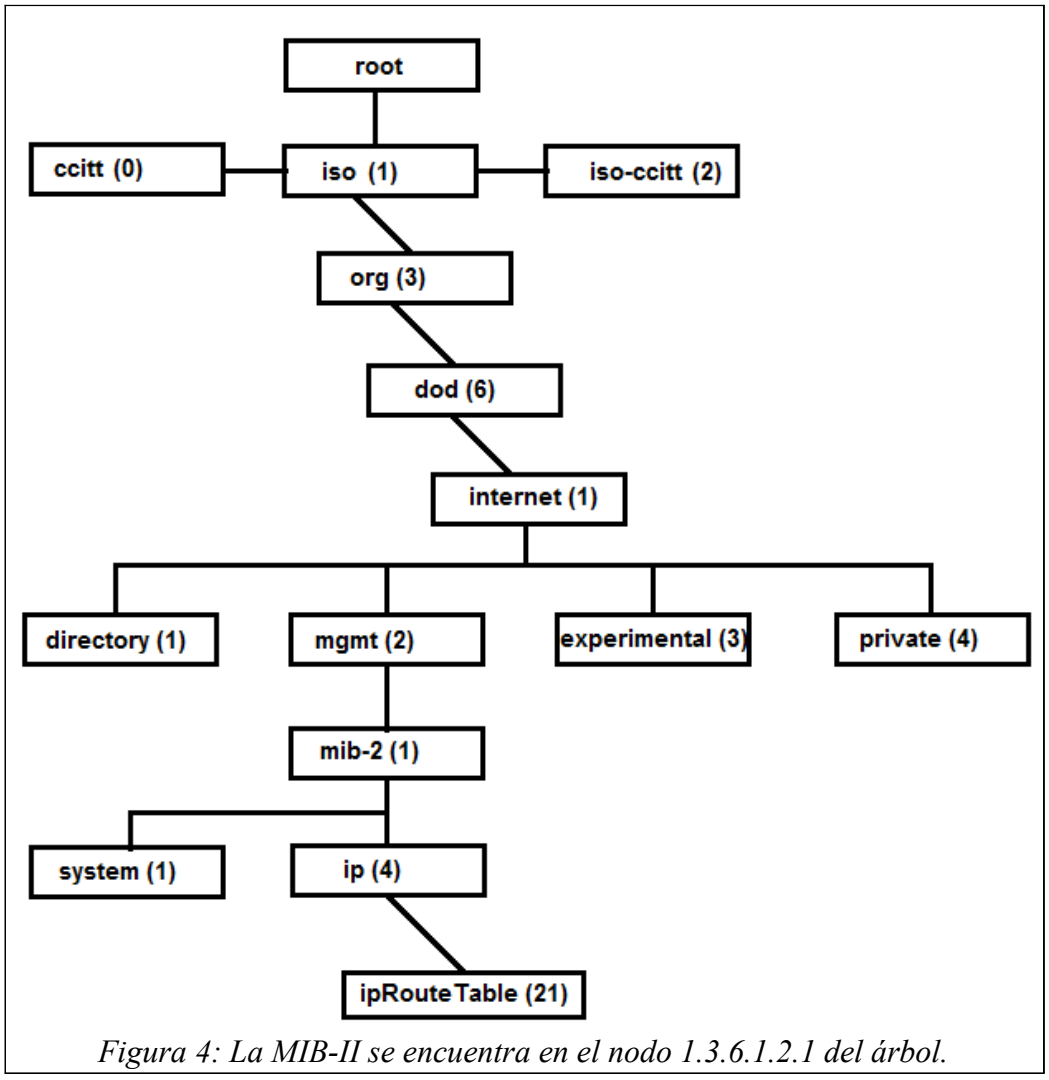

Figura 4: La MIB-II se encuentra en el nodo 1.3.6.1.2.1 del árbol.

En la figura 4 podemos ver como el objeto *ipRouteTable* podría ser identificado por el nombre: *iso.org.dod.internet.mgmt.mib-2.ip.ipRouteTable*, o por el descriptor de objeto equivalente *.1.3.6.1.2.1.4.21.*

Los identificadores de los objetos ubicados en la parte superior del árbol pertenecen a diferentes organizaciones estándares, mientras los identificadores de los objetos ubicados en la parte inferior del árbol son definidos por las organizaciones asociadas.

Los vendedores pueden definir ramas privadas que incluyen los objetos administrados para sus propios productos. Las MIB's que no han sido estandarizadas, normalmente están localizadas en la rama experimental.

5.2.3.1 Estructura

La MIB-II se compone de los siguientes nodos estructurales:

- *System*: de este nodo cuelgan objetos que proporcionan información genérica del sistema gestionado. Por ejemplo, dónde se encuentra el sistema, quién lo administra...
- *Interfaces*: en este grupo está la información de las interfaces de red presentes en el sistema. Incorpora estadísticas de los eventos ocurridos en el mismo.
- *At* (*address translation* o traducción de direcciones): Este nodo es obsoleto, pero se mantiene para preservar la compatibilidad con la MIB-I. En él se almacenan las direcciones de nivel de enlace correspondientes a una dirección IP.
- *Ip*: en este grupo se almacena la información relativa a la capa IP, tanto de configuración como de estadísticas.
- *Icmp*: en este nodo se almacenan contadores de los paquetes ICMP entrantes y salientes.
- *Tcp*: en este grupo está la información relativa a la configuración, estadísticas y estado actual del protocolo TCP.
- *Udp*: en este nodo está la información relativa a la configuración, estadísticas del protocolo UDP.
- *Egp*: aquí está agrupada la información relativa a la configuración y operación del protocolo EGP.
- *Transmission*: de este nodo cuelgan grupos referidos a las distintas tecnologías del nivel de enlace implementadas en las interfaces de red del sistema gestionado.

5.2.3.2 Sintaxis

Los tipos de objetos están definidos en el RFC 1155[2], pero los más importantes son:

- Integer: para objetos que se representen con un número entero.
- Octet String: para texto.
- Null: cuando el objeto carece de valor.
- Object Identifier: para nodos estructurales.
- Sequence y Sequence of: para vectores.
- IpAddress: para direcciones IP
- Counter: número de 32 bits para objetos que representen un contador.
- Gauge: número de 32 bits para objetos medidores.
- Timeticks: para medir tiempos. Cuenta en centésimas de segundos.
- Opaque: para cualquier otra sintaxis ASN.1.

5.2.3.3 Definición de tablas

Las tablas son un tipo estructurado. Se definen usando el tipo *"Sequence of"*. La tabla consiste en un vector ("*sequence of*") de filas, cada una formada por un "*Sequence*" que define la columna.

5.2.4 ASN.1

Abstract Syntax Notation One (notación sintáctica abstracta uno) está diseñado para definir información estructurada de tal forma que sea independiente de la máquina utilizada. Para hacer esto, ASN.1 define tipos de datos básicos, como enteros y strings, y además permite construir nuevos tipos de datos a partir de los ya definidos. Este protocolo fue desarrollado como parte de la capa 6 (presentación) del modelo OSI. El protocolo SNMP usa el ASN.1 para describir la estructura de los objetos.

La notación de Backus-Naur (BNF) es la utilizada por ASN.1 para describir la forma en que se almacena la información.

TIPOS DE DATOS:

En ASN.1 los tipos de datos están clasificados en primitivos, construidos y definidos. Por convención, los tipos comienzan con una letra mayúscula.

5.2.4.1 Tipos primitivos

- INTEGER es un tipo primitivo cuyos valores son números enteros positivos o negativos incluyendo el cero.
- OCTET STRING es un tipo primitivo cuyos valores son una secuencia ordenada de cero, uno, o más bytes. SNMP usa tres casos especiales del tipo OCTET STRING:
  - DisplayString (caracteres ASCII)
  - OctetBitString (cadenas de bits mayores de 32)
  - PhysAddress (direcciones del nivel de enlace).
- OBJECT IDENTIFIER es un tipo cuyos valores representan el identificador de un objeto dentro de la MIB.
- BOOLEAN es un tipo para representar valores que sólo pueden ser verdadero o falso.
- NULL es un tipo carente de valor.

5.2.4.2 Tipos construidos

Los tipos construidos definen tablas y filas (entradas) dentro de dichas tablas. Por convención, los nombres para los objetos tabla terminan con el sufijo *Table*, y los nombres para las filas terminan con el sufijo *Entry*

- SEQUENCE es un tipo estructurado definido mediante una lista de tipos fijos y ordenados. Algunos de los tipos pueden ser opcionales y todos pueden ser

diferentes tipos definidos en ASN.1. El SEQUENCE como un conjunto, define una fila dentro de un tabla.

- SEQUENCE OF es un tipo construido que está definido mediante una lista de tipos definidos en ASN.1. Cada valor es una lista ordenada de cero, uno, o más valores de dicho tipo existente. Es el tipo utilizado para definir las columnas en una tabla, y a diferencia de SEQUENCE, SEQUENCE OF sólo usa elementos del mismo tipo en ASN.1.
- SET y SET OF son tipos equivalentes a SEQUENCE y SEQUENCE OF respectivamente, pero con la única diferencia de que las listas no están ordenadas.
- CHOICE es un tipo construido en el que hay que elegir uno de entre los tipos disponibles en una lista.

5.2.4.3 Tipos definidos

Los tipos definidos son derivados de los tipos anteriores pero son más descriptivos:

- NetworkAddress fue ideado para representar una dirección IP de alguna de las muchas familias de protocolos. Son 4 bytes y se define como OCTET STRING (SIZE (4)).
- Counter es un tipo definido que representa un entero no negativo que se incrementa hasta alcanzar un valor máximo (2^{32}-1), luego se reinicia y vuelve a comenzar desde cero.
- Gauge es un tipo definido que representa un entero no negativo. Este puede incrementarse o decrementarse, pero no sobrepasa el valor máximo (2^{32}-1).
- TmeTicks es un tipo definido que representa un entero no negativo que calcula el tiempo en centésimas de segundos desde algún periodo o instante de tiempo. Es un entero de 32 bits.

5.2.5 Comunidad

Introducimos un concepto muy importante en el protocolo SNMP.

Se denomina comunidad (*community*) a un conjunto de gestores y a los objetos gestionados. A las comunidades se les asignan nombres, de tal forma que este nombre junto con cierta información adicional (se verá más adelante los campos de un paquete SNMP) sirva para validar un mensaje SNMP y al emisor del mismo.

Así, por ejemplo, si se tienen dos identificadores de comunidad: “total” y “parcial”, se podría definir que el gestor que use el identificador “total” tenga acceso de lectura y escritura a todas las variables del MIB, mientras que el gestor con nombre de comunidad “parcial” sólo pueda acceder para lectura a ciertas variables del MIB.

5.2.6 Formato del mensaje SNMP

El formato del mensaje de alto nivel en SNMP está formado por 3 campos:

- *version:* es de tipo entero (*INTEGER*). Para SNMPv1 toma valor 0, para SNMPv2c toma el valor 1.
- *community:* este campo es una cadena binaria con longitud en bits múltiplo de 8, cada carácter se representa con 8 bits (*OCTET STRING*), almacena el nombre de la comunidad SNMP a la que pertenece el paquete.
- *data:* puede ser de cualquier tipo (*ANY)*, se usa para almacenar la PDU (Unidad de Datos de Protocolo, en inglés *Protocol Data Unit*). En SNMPv2c puede ser de tipo PDU o BulkPDU.

Dentro del campo *data* nos encontramos con la PDU, que dependiendo de la versión, puede ser de cinco (SNMP v1) o siete tipos (SNMP v2c).

La figura 5 muestra el formado de algunas PDU utilizadas por SNMP.

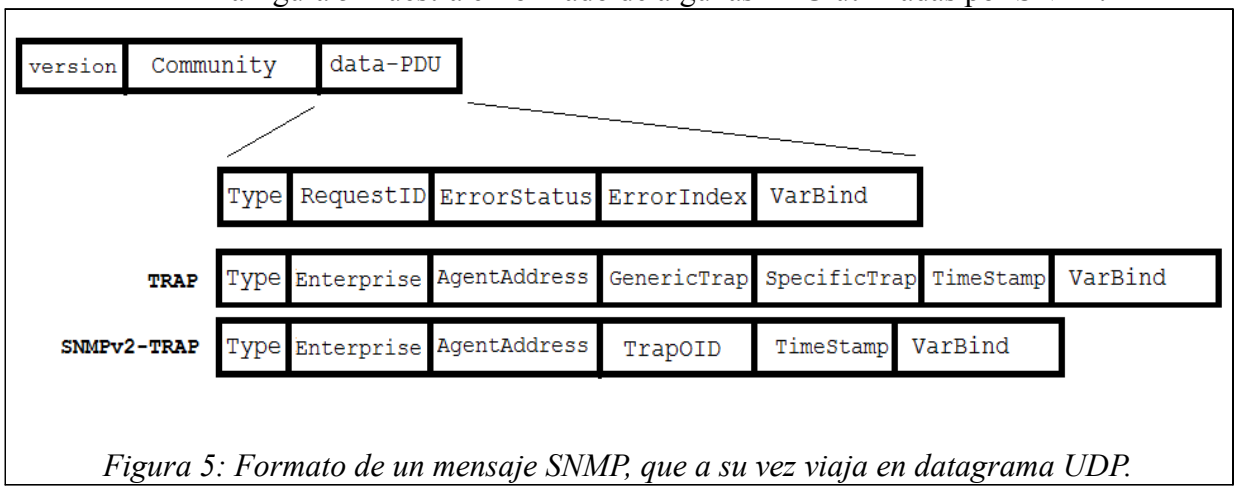

Figura 5: Formato de un mensaje SNMP, que a su vez viaja en datagrama UDP.

Al mismo tiempo, cada PDU está formada por una serie de campos, que se nombran a continuación:

5.2.6.1 Campos de una PDU.

- *Type:* es de tipo entero (*INTEGER*), implícito y se sitúa al comienzo de la PDU para distinguir el tipo de PDU que contiene el mensaje, puede tomar valores de 0 a 4:
  - 0 (GetRequest-PDU): con esta PDU se puede solicitar el valor de una o varias variables. Las variables de las cuales queremos conocer su valor se listan en el campo *VariableBinding* (la estructura de VariableBinding se detalla más abajo).
    Denominaremos *VaribleBinding, VarBind*, *Binding* o enlazado a aquel campo de una PDU que representa un par atributo-valor, pudiendo en algunos casos ser el valor del *binding* irrelevante para la operación actual (puede que el *binding* tenga como finalidad indicar el nombre de la variable que se solicita al receptor de la PDU).
  - 1 (GetNextRequest-PDU): con esta PDU se solicita el valor de una o varias varibles indicadas.

    - 2 (GetResponse-PDU): esta PDU se genera como respuesta a peticiones de tipo *GetRequest*, *GetNextRequest* y *SetRequest.*
    - 3 (SetRequest-PDU) con esta PDU se solicita el establecimiento del valor de la variable o variables que indiquemos en *VariableBinding*.
    - 4 (Trap-Pdu) estas PDU permiten a los agentes comunicar de manera asíncrona a los gestores cualquier evento que haya sucedido al objeto gestionado y en el cual el gestor tiene interés de ser informado .
- *RequestID:* es de tipo entero (*INTEGER*), permite establecer correspondencias entre respuestas y peticiones como se ha indicado más arriba.
- *ErrorStatus:* es de tipo entero (*INTEGER*), puede tomar valores de 0 a 5 siendo:
    - 0 (noError): Sin error.
    - 1 (tooBig): Tamaño demasiado grande.
    - 2 (noSuchName): El nombre no existe.
    - 3 (badValue) Valor incorrecto.
    - 4 (readOnly): Sólo lectura.
    - 5 (genErr): Error genérico.
- *ErrorIndex* : es de tipo entero (*INTEGER*), indica la posición de la variable que ha causado el error, 0 en caso contrario.
- *VariableBinding* o enlazado*:* este campo es una estructura que almacena un nombre identificativo y su valor, el nombre es de tipo *ObjectName* definido en el estándar ASN.1, el valor es de tipo *ObjectSyntax* según ASN.1.En algunos casos el valor del enlazado puede ser irrelevante para la operación actual (puede que el enlazado tenga como finalidad indicar el nombre de la variable que se solicita al receptor de la PDU). Se pueden formar listas de objetos simplemente poniendo unos a continuación de otros.

5.2.6.2 SNMP v1

SNMP versión 1 (SNMPv1) es la implementación inicial del protocolo SNMP, descrita en el RFC 1157 [22]. SNMPv1 opera a través de protocolos como el protocolo de datagramas de usuario (UDP). Aunque UDP es no fiable y no orientado a conexión, SNMP permite al receptor detectar paquetes duplicados mediante el campo *RequestID*. Este mecanismo funciona a nivel de la capa de aplicación. Además el campo *RequestID* sirve para emparejar respuestas con peticiones, en aquellas PDU que necesitan respuesta, es decir, todas menos la Trap-PDU.

La versión 1 ha sido criticada por su falta de seguridad y autenticación de los clientes, la cual se realiza sólo por una "cadena de comunidad", que en realidad es un tipo de contraseña, que se transmite en texto plano. El diseño de los años 80 del SNMP v1 fue realizado y diseñado para las plataformas de computación de la época. Además, SNMP fue aprobado con respecto a la creencia de que se trataba de un protocolo provisional necesario en un despliegue a gran escala de Internet y su comercialización. A esta situación se debe sumar que el tema de la seguridad no era una de las principales áreas de interés en las arquitecturas de red.

Otros RFCs sobre SNMPv1 consultados son los siguientes:

- RFC 1089 *"SNMP over Ethernet"*[23].
- RFC 1418 *"SNMP over OSI"*[24].
- RFC 1187 *"Bulk Table Retrieval with the SNMP"*[25].
- RFC 1270 *"SNMP Communications Services"*[26].
- RFC 1303 *"A Convention for Describing SNMP-based Agents"*[27].
- RFC 2573 *"SNMP Applications"*[28].
- RFC 3413 *"Simple Network Management Protocol (SNMP) Applications"*[29].
- RFC 5592 *"Secure Shell Transport Model for the Simple Network Management Protocol (SNMP)"*[30].
- RFC 5675 *"Mapping Simple Network Management Protocol (SNMP) Notifications to SYSLOG Messages"*[31].
- RFC 5935 *"Expressing SNMP SMI Datatypes in XML Schema Definition Language"*[32].

Los tipos válidos para representar información de los paquetes se toman del estándar ASN.1, además de definir tipos compuestos nuevos a partir de los existentes en ASN.1.

SNMP v1 especifica cinco tipos de paquetes PDU. Todos están compuestos por los campos antes descritos, excepto el último, (Trap- PDU), que contiene los suyos propios:

- GetRequest-PDU: esta PDU se utiliza para solicitar los valores de las variables que estén en el campo *VariableBinding*. En la respuesta, el campo *RequestID* debe tener el mismo valor que el de esta petición. Los campos *ErrorStatus* y *ErrorIndex* siempre toman el valor 0. Como respuesta recibiremos una PDU de tipo GetResponse-PDU, con los valores de las variables solicitadas establecidos en *Variable-Binding.*

- GetNextRequest-PDU: esta PDU se puede utilizar para solicitar el nombre y valor de las variables lexicográficamente siguientes a las indicadas en la lista *Variable-Binding* (permite recorrer tablas). Los campos *ErrorStatus* y *ErrorIndex* siempre toman el valor 0. El contenido de sus campos es análogo al de la PDU GetRequest.

- GetResponse-PDU: esta PDU se genera como respuesta a las PDUs GetRequest, GetNextRequest, SetRequest, y devuelve la información solicitada en los GET o el valor escrito en los SET (a continuación).

- SetRequest-PDU: esta PDU se puede utilizar para establecer los valores de las variables que estén en el campo *VariableBinding*. Los campos *ErrorStatus* y *ErrorIndex* siempre toman el valor 0.

- Trap-PDU: esta PDU se puede utilizar para enviar notificaciones a un gestor SNMP. La PDU Trap está formada por los siguientes campos:
  - *PDU Type* y *Variable Binding*: Estos dos campos son los mismos explicados anteriormente.
  - *Enterprise*: Es de tipo identificador de objeto (*OBJECT IDENTIFIER*), se utiliza para almacenar el tipo de objeto que genera el trap.
  - *Agent Address*: Es de tipo dirección de red (*NetworkAddress*), se utiliza para almacenar la dirección del dispositivo que genera el trap.
  - *Generic Trap Type*: Es de tipo entero (*INTEGER*), indica el tipo del trap y puede tomar valores de 0 a 6 siendo:
    - 0 (*coldStart*): arranque en frío, significa que el agente que envía el trap se está reinicializando y su configuración podría ser alterada.
    - 1 (*warmStart*): arranque en caliente, significa que el agente se está reincializando pero su configuración no va a ser alterada.
    - 2 (*linkDown*): enlace caído, significa que el agente ha detectado que uno de sus enlaces está caído. El primer elemento de la lista del campo *VariableBinding* de este trap contendrá el nombre y valor de la interfaz en la que se ha detectado como caída.
    - 3 (*linkUp*): enlace levantado, significa que el agente ha detectado que uno de sus enlaces se ha levantado. El primer elemento de la lista del campo *VariableBinding* de este trap contendrá el nombre y valor de la interfaz en la que se ha detectado como levantada.
    - 4 (*authenticationFailure*): fallo de autenticación, significa que la entidad receptora de un mensaje no ha podido certificar la autenticación con los datos suministrados.
    - 5 (*egpNeighborLoss*): pérdida de vecino EGP, significa que uno de las estaciones vecinas EGP (Exterior Gateway Protocol o protocolo de puerta de enlace exterior) del agente ha sido anotada como caída y ya no debe ser considerado como vecina. El primer elemento de la lista del campo *VariableBinding* de este trap contendrá el nombre y valor de la estación vecina que se ha detectado como caída.
    - 6 (*enterpriseSpecific*): específico de empresa, este valor del tipo de trap se utiliza para indicar que el tipo de notificación debe ser considerado como definido por los administradores que gestionan la red. El campo *Specific Trap Type* (a continuación) debe contener el código que identifica el tipo de trap particular que ha sucedido.

  - *Specific Trap Type:* Es de tipo entero (*INTEGER*), indica el código especifico del trap ocurrido. Este campo debe estar presente aunque el valor del campo generic-trap no tenga el valor *enterpriseSpecific*. Es decir, si *Generic Trap Type* no tiene el valor 6, entonces este valor valdrá 0.

- *Time-stamp:* cuenta de reloj (TimeTicks), indica el tiempo transcurrido desde que la estación fue encendida o reiniciada, hasta que se generó el trap.

5.2.6.3 SNMP v2c

Antes de la aparición de SNMP v2c, definido en el RFC 1901 "*Introduction to Community-based SNMPv2*"[33], surgió SNMP v2 (sin la 'c'), que es una revisión de la versión 1, e incluye mejoras en las áreas de rendimiento, seguridad, confidencialidad, y gestión. Otros RFCs sobre SNMPv2 consultados son los siguientes:

- RFC 1445 *"Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)*"[34].
- RFC 1446 *"Security Protocols for version 2 of the Simple Network Management Protocol (SNMPv2)*"[35].
- RFC 1447 *"Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2)"[36]*.
- RFC 3416 *"Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)*"[37].
- RFC 5590 *"Transport Subsystem for the Simple Network Management Protocol (SNMP)*"[38].
- RFC 3418 *"Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)*"[21].
- RFC 3584 *"Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework*"[39].
- RFC 1503 *"Algorithms for Automating Administration in SNMPv2 Managers"*[40].

Sin embargo, el nuevo sistema de seguridad implementado en SNMPv2 fue muy criticado por su complejidad, y no fue muy aceptado, por lo que surgió el SNMPv2c, que es una versión cuyos permisos estaban basados en una comunidad, al igual que SNMPv1. SNMPv2c es actualmente el tipo de protocolo SNMP más utilizado y lo podemos encontrar definido en los RFC´s desde el 1901 al 1908.

Poco después aparece SNMPv2u, definido en el RFC 1910 "*User-based Security Model for SNMPv2*"[41], que consta de un sistema de permisos y accesos basados en usuarios, que aportaba bastante seguridad sin la complejidad de SNMPv2.

SNMPv2u ha sido sustituido por SNMPv3 (RFC 3414 "*User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)*" [42]), última versión hasta ahora del protocolo SNMP, que aún hoy en día no ha tenido tanta aceptación como la versión 2c.

De forma oficial, la versión 2 del protocolo SNMP es SNMPv2c, por esa razón lo podemos ver escrito de las dos formas dependiendo de la fuente. Como apunte

adicional, en el RFC 3584 "*Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework*"[39], encontramos la definición de la coexistencia entre las distintas versiones del protocolo.

SNMPv2 introduce algunos cambios con respecto a SNMPv1. Principalmente añade 3 nuevos tipos de PDU:

- *GetBulkRequest:* una alternativa a *GetNextRequest-PDU,* pero no lo sustituye. Sirve para solicitar grandes cantidades de información dejando que el agente fije el tamaño del mensaje.
- *SNMPv2-Trap:* sustituye al Trap-PDU de la versión 1, su función es la misma, pero existe una ligera variación en los campos del PDU:
  - Desaparecen los campos *Generic Trap Type* y *Specific Trap Type.* Y aparece uno nuevo llamado: *TrapOID.* En realidad, este nuevo campo es el equivalente a juntar los dos anteriores.
  - Este tipo de mensaje no tiene ningún tipo de confirmación asociada.
  - Las dos primeras variables de la lista *varibale-binding*, deben ser: *sysUpTime* y *snmpTrapOID.* Se pueden añadir a continuación las *VariableBindings* que se deseen enviar, como se procedía en el Trap-PDU de SNMPv1.
    - *SysUpTime:* es de tipo *timeTicks,* definido en el RFC 3418[21], página 3. Es una medida del tiempo transcurrido (cada unidad equivale a la centésima parte de un segundo) desde que la región de la red, en la que está el proceso que genera valor, fue reinicializada por última vez.
    - *SnmpTrapOID:* es de tipo *OBJECT IDENTIFIER* definido en el RFC 3418[21], página 9. Es una secuencia de enteros que codifica un algoritmo, tipo de atributo o tipo definido por una autoridad de registro (*registration authority*). Se utiliza como identificación de la notificación actual que se envía.
- *InformRequest:* es usado para notificar cambios, es una alerta "trap", pero síncrona, el receptor enviará confirmación de la recepción.

6 Desarollo

6.1 Fase 1: Inicio del proyecto

Debido a que el paquete de Quagga está construido en lenguaje C, y diseñado para sistemas GNU/Linux, el primer preparativo es tener disponible un equipo con GNU/Linux instalado, y a su vez, tener instalado alguna herramienta de programación en C, en este caso se ha utilizado el programa ECLIPSE, con el respectivo *plugin* de C, llamado CDT.

Con ECLIPSE abrimos el código fuente de Quagga y a partir de entonces nos encontramos con uno de los principales problemas que se arrastrará durante todo el proyecto, la falta de documentación sobre el paquete Quagga. En su página oficial, como documentación, encontramos únicamente un manual de uso, con algunas explicaciones de funcionamiento, protocolos, RFCs, formas de usar su consola de configuración, etcétera. Pero nada de

documentos sobre el código fuente, API´s utilizadas, funciones, unidades, forma de organizar el código.

La única documentación sobre el código fuente son los escasos comentarios dentro del mismo código C, y la suposición del comportamiento de las funciones al leer su identificador.

6.2 Fase 2: Implementación

Después de los preparativos, lo siguiente es comenzar el desarrollo. En esta fase, incluiremos el código necesario para añadir esa nueva capacidad al software Quagga.

Se intenta minimizar el cambio dentro del código de Quagga, intentando llevar el mismo orden al nombrar variables, funciones y bloques de código. Todo cambio realizado se ha comentado ampliamente, en inglés por supuesto, para poder seguir de la mejor manera posible el funcionamiento.

Nos centramos pues, en los objetivos de la ampliación. Sabemos que el demonio principal Zebra gestiona la tabla de rutas del *kernel* o núcleo, independientemente del otro demonio o demonios de encaminamiento que tenemos ejecutándose. Por ejemplo, si elegimos usar RIP en nuestra máquina encaminadora, a la vez se estará ejecutando Zebra, que gestionará esas rutas "aprendidas" por RIP y decidirá si agregarla o eliminarla.

Nuestro objetivo es que Quagga, o en este caso el demonio principal Zebra, nos avise o envíe una alerta, cuando añada o elimine esas rutas aprendidas por los diferentes protocolos de encaminamiento. La citada alerta será un Trap-PDU, de esta forma aprovechamos las funcionalidades que nos ofrece SNMP.

La escasa documentación sobre el código, la cual se limita a simples comentarios dentro del código fuente, nos limita enormemente el desarrollo. Sobre todo en invertir más tiempo para averiguar qué realiza cada parte del código, a no ser que se ejecute el software en modo depuración o deduciendo el funcionamiento a través de los nombres de las funciones o de los escasos comentarios.

6.2.1 Introducción y pasos a seguir

Ya que no sabemos nada sobre la estructura del código fuente, ni tampoco sobre las acciones realizadas en cada parte del código, los pasos a seguir serán los siguientes:

- Averiguar la estructura del código fuente de Quagga: su división en directorios y la relación entre ellos.
- Detectar los directorios y archivos donde se encuentra la parte que nos interesa en el proyecto.
- Modificar esa parte de código, y añadir las funciones que extraeremos de Net-SNMP, más algunas propias, para conseguir que Quagga alerte al agregar/eliminar una ruta.
- Realizar las pruebas necesarias, para ello necesitaremos simular varios encaminadores.

6.2.2 Estructura del código fuente

Resumamos la estructura del código fuente de Quagga, el cual se divide en una serie de módulos. Cada módulo está dentro de su correspondiente directorio, que pueden ser, entre otros, un directorio para las cabeceras, otro para las librerías y el resto contiene los demonios de encaminamiento implementados en Quagga, todo ello siguiendo un estricto orden en el nombre de las funciones y archivos. Los módulos más significativos que encontramos son:

Directorio raíz: quagga-0.99.18.tar.gz

- include
- lib
- bgpd
- ospfd
- ospf6d
- ripd
- ripngd
- zebra

A modo de ejemplo, en la carpeta *ripd* encontramos todo el código perteneciente a la implementación del protocolo RIP, dividido en sus correspondientes archivos *.c.

Conviene aclarar el funcionamiento en general de todo el software Quagga, ya explicado en el apartado de documentación. La base es la siguiente: Zebra opera como un servidor que escucha a los clientes, que son los demonios. Los demonios se comunicarán con Zebra, usando un protocolo de mensajes propio de Quagga. Sabiendo que en cada carpeta tenemos los archivos agrupados de cada demonio, una de las primeras investigaciones consistirá en averiguar qué parte del código de cada demonio se encarga de comunicarse con Zebra, ya que es en ese momento cuando empieza la gestión de la información, es decir, de las rutas.

Una de las primeras deducciones al ver el nombre de los archivos de código es que siguen un orden en su nomenclatura, dependiendo de su función. Siempre comienzan con el nombre del demonio implementado, después un guión bajo, y luego su función. Por ejemplo los archivos encargados de hablar con Zebra, serán nombrados así: `protocolo_zebra.c`, es decir, si estamos con OSPF, hablamos del archivo `ospfd/ospf_zebra.c`, y si estamos con BGP, sería el archivo `bgpd/bgp_zebra.c`.

De la misma forma, los archivos que contienen el código principal de cada demonio llevarán su nomenclatura, por ejemplo, si estamos con RIP, tendremos el archivo `ripd/rip_main.c` y si estamos con BGP, tendremos `bgpd/bgp_main.c`. Todo esto es una de las pocas ayudas que nos ofrece este software a la hora de su desarrollo.

6.2.2.1 Archivos y directorios a tener en cuenta.

El directorio más importante es *zebra*, ya que aquí encontramos los archivos del demonio principal, que es el encargado de decidir si agrega o elimina la ruta. Comenzaremos analizando la función que recibe la ruta enviada desde algún demonio encaminador, y seguiremos el camino de funciones hasta que finalmente esa ruta es añadida o eliminada del *kernel* o núcleo del sistema operativo.

6.2.3 Primeros problemas

Con poca documentación y unos comentarios escasos, la investigación sobre la funcionalidad del código se convierte más que nunca en una serie de "prueba y error". Cada vez que encontramos una parte de código o una serie de funciones que creemos funcionan de una forma, tenemos que comprobar que de hecho, ese funcionamiento se cumple, por que nada ni nadie nos está explicando si lo que deducimos es verdad, excepto el código en sí. Todo esto se complica de forma exponencial cuando comenzamos a mirar ese código fuente.

Imaginemos una función, por cuyo nombre se puede deducir que realiza una acción determinada, pero no hay una documentación donde se especifique la finalidad de esa función, por lo que sólo queda seguir el código para averiguarlo. En ese código hay llamadas a más funciones, cada una de la cuales tiene sus instrucciones, y más llamadas a funciones. Por tanto se debe saber qué realizan esas otras funciones para saber lo que realiza la función *padre*. En resumen, el problema aumenta exponencialmente.

6.2.4 Profundizamos en el código

Después de numerosas comprobaciones con el código, y depuraciones, acabamos descubriendo unas funciones comunes que utilizan todos los demonios para comunicarse con el demonio principal Zebra. Cada demonio tiene en su directorio un archivo con esas funciones citadas, ripd, ospfd y bgpd nombran esos archivos: `rip_zebra.c`, `ospf_zebra.c` y `bgp_zebra.c`, respectivamente.

De esas comunicaciones y mensajes entre clientes y servidores, los que nos interesan son aquellos que se refieren a rutas nuevas para añadir al *kernel* o núcleo, o rutas obsoletas para eliminar.

Empecemos por el módulo de RIP. Dentro del archivo `ripd/rip_zebra.c`, tenemos la función `rip_zebra_ipv4_add()`. Con el nombre podemos deducir que se adapta bien a nuestro propósito, así que analizamos su contenido y encontramos una serie de variables sobre la ruta (IP, la máscara de red, el siguiente salto, la métrica) y una llamada a la función: `zapi_ipv4_route()`, enviando esos parámetros.

Nota importante: Los parámetros de entrada de las funciones no se indican aquí, solamente el nombre para simplificar la lectura de esta memoria. Para ello se adjunta un CD con todo el código fuente, y la URL de descarga del software.

Seguimos investigando el curso de esas llamadas, y analizamos el interior de `zapi_ipv4_route()`, donde, gracias a unos pocos comentarios, podemos leer que se trata de una API para comunicarse con Zebra. De entre todas las funciones, instrucciones y parámetros que hay dentro de esta última función, hay una que nos llama la atención: `zclient_send_message()`.

Si recordamos que los demonios en realidad funcionan como clientes de Zebra, ésta función de seguro que nos ayudará en nuestro objetivo. Pero continuamos.

Nos vamos ahora a OSPF, dentro del archivo `ospfd/ospf_zebra.c`, tenemos la función: `ospf_zebra_add()`, y dentro nos encontramos una llamada a la función: `zclient_send_message()`. Hemos llegado de nuevo a esta función. Con lo cual nuestra investigación se encamina algo mejor.

Esta última función, a la cual se llama de forma común por parte de los dos demonios, forma parte de una API que sirve para comunicarse con el demonio gestor Zebra. De forma que cuando un demonio quiere enviar algún mensaje a Zebra perteneciente a una ruta (añadir, borrar, modificar), usa la citada API.

Sólo nos queda BGP, nos vamos a `bgpd/bgp_zebra.c` y buscamos la función encargada de enviar mensaje a Zebra: `bgp_zebra_announce()`, dentro de la cual, se realiza una llamada a la función `zapi_ipv4_route()`, y que como sabemos nos lleva finalmente a `zclient_send_message()`.

Tanto en la función de OSPF y de RIP, podemos ver en el nombre de las funciones la palabra *add*, que instintivamente nos informa de que son funciones dedicadas a enviar el "mensaje" a Zebra para el caso de rutas a añadir. Si resulta que se quiere eliminar esa ruta (porque ya no es alcanzable), existen las mismas funciones pero con la palabra *delete*.

Por ejemplo, para la función `ospf_zebra_add()`, tenemos la función `ospf_zebra_delete()`. El cual llama a `zapi_ipv4_route()`, que ya sabemos que llamará a `zclient_send_message()`.

Para RIP, tenemos `rip_zebra_ipv4_delete()`, y para BGP, la función es `bgp_zebra_withdraw()`, cuyo nombre vemos no es un simple cambio de *add* a *delete,* sino que es un cambio de *announce* a *withdraw*, cuya traducción es *anunciar* y *retirada*, respectivamente. De esa forma se sigue la nomenclatura usada en la definición del protocolo BGP.

Hasta este momento hemos visto únicamente los demonios que implementan RIP, OSPF y BGP, todos en su versión IPv4. No se considerarán en este proyecto sus respectivas versiones IPv6 debido a la incompatibilidad de anotar una dirección IP en formato IPv6 dentro de un Trap-PDU. De la misma forma, para el caso de Zebra no se considerarán aquellas funciones específicas para el tratamiento de mensajes en IPv6. En el apartado *Propuestas de futuras mejoras*, se explica detalladamente esta incompatibilidad.

6.2.5 Comunicación entre demonios y Zebra

Nos centramos ahora en la función `zclient_send_message()`.

Su único parámetro de entrada es un "*struct"* (`struct zclient *zclient`). La citada estructura "zclient" consiste en un registro donde se guarda toda la información del cliente, el socket, el búfer de escritura y lectura, el mensaje a enviar, etc. Por el lado del servidor, veremos más adelante que la función final que analiza el mensaje es: `zread_ipv4_add()`, que se encuentra dentro del archivo `zebra/zserv.c`.

"`zserv.c`" contiene todas las funciones que Zebra utiliza para recibir y tratar los mensajes de los demonios clientes. Según la definición original, se trata de "*zebra daemon server routine*", traducido, *rutina servidora del demonio Zebra*. Dentro de esta última función (`zread_ipv4_add()`), se descomponen los mensajes recibidos en versión IPv4, obteniendo los valores de dirección IP, máscara de red, métrica, siguiente salto, tipo de ruta, etc.

A modo de resumen, hasta ahora, nos encontramos con una serie de llamadas se muestra en la figura 6:

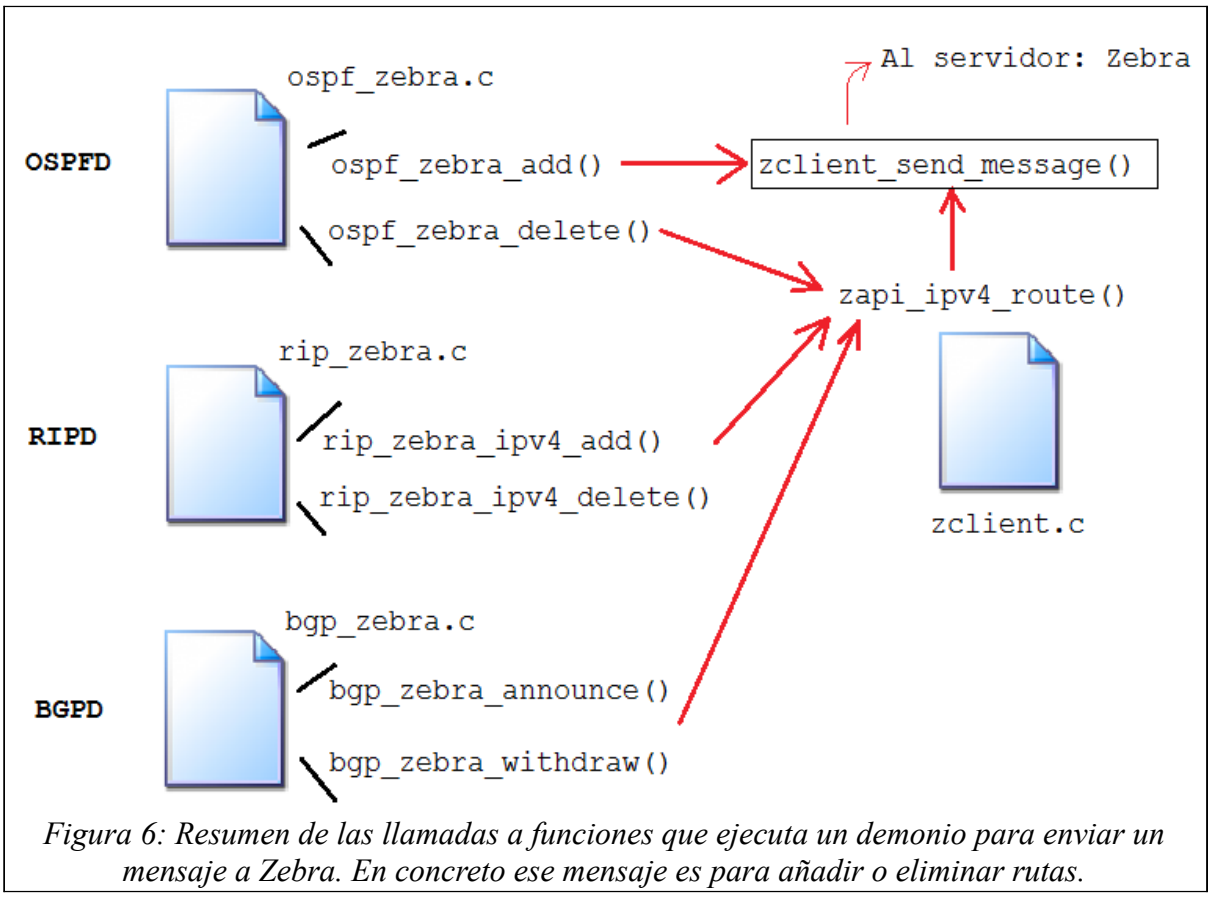

Figura 6: Resumen de las llamadas a funciones que ejecuta un demonio para enviar un mensaje a Zebra. En concreto ese mensaje es para añadir o eliminar rutas.

Ya sabemos el orden de llamadas que realiza el proceso-demonio cliente para enviar a Zebra un mensaje que contiene la información de una ruta, ya sea para añadir o eliminar. Recordemos que entre todos los tipos de mensajes que usan para comunicarse, buscamos aquellos que indiquen la adición o eliminación de rutas en formato IPv4:

ZEBRA_IPV4_ROUTE_ADD y ZEBRA_IPV4_ROUTE_DELETE

Por tanto, llega el turno de ver desde el lado del servidor Zebra los mensajes recibidos y comprobar si es alguno de los ya citados.

Dentro de la unidad "*zserv.c*" se encuentran las funciones encargadas de ejecutar el proceso servidor perteneciente a Zebra. Al mismo tiempo, para poder escuchar a los clientes, iniciará nuevos hilos de ejecución, uno por cada cliente, con su correspondiente *socket* para manejar el flujo de datos. Cuando se recibe algún mensaje del cliente, se genera un evento y se procede a leer el tipo de mensaje, por lo que llegamos a la función: `zread_ipv4_add()`, en caso de que el tipo sea ZEBRA_IPV4_ROUTE_ADD y a la función `zread_ipv4_delete()` en caso de ser ZEBRA_IPV4_ROUTE_DELETE.

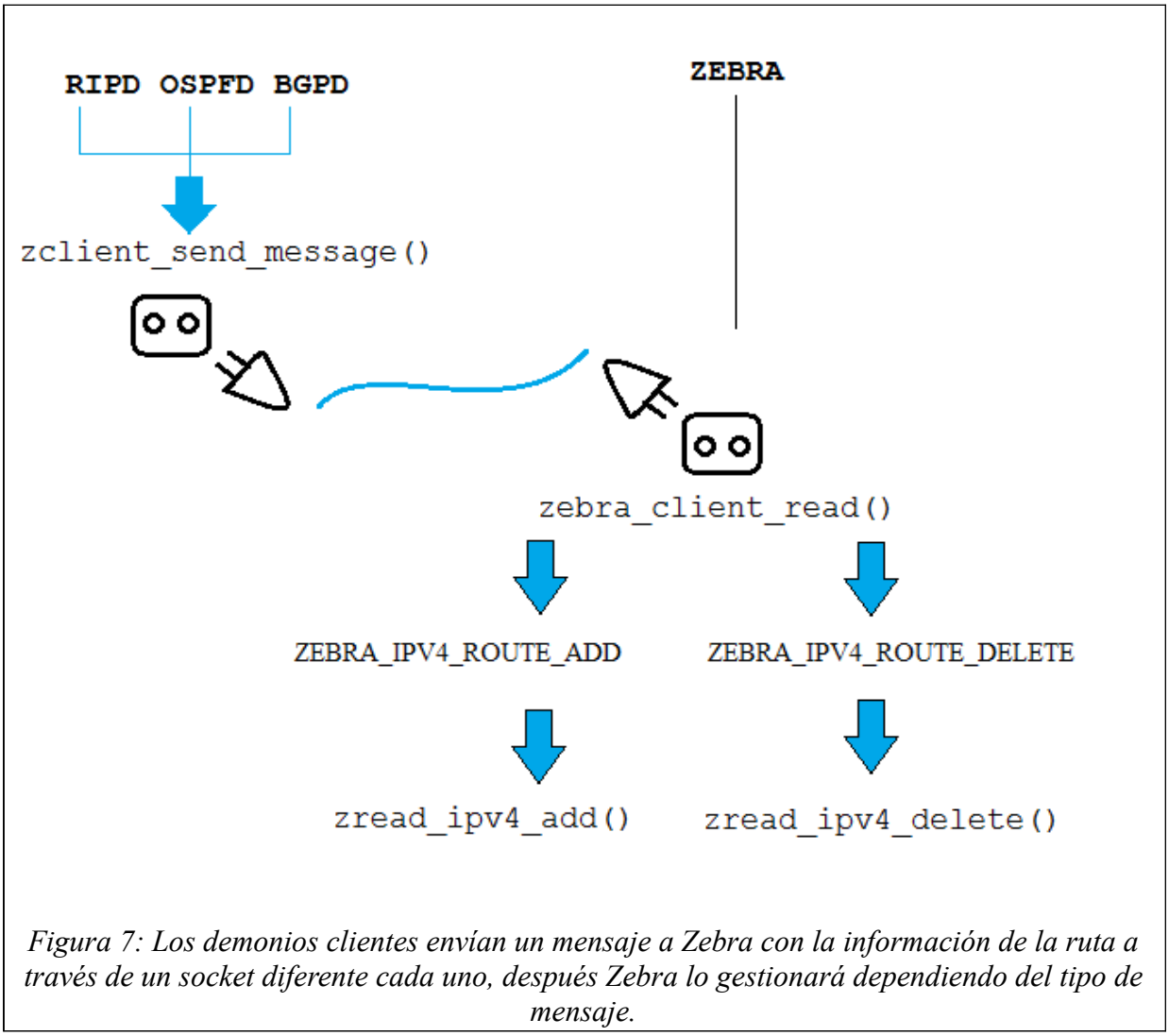

Figura 7: Los demonios clientes envían un mensaje a Zebra con la información de la ruta a través de un socket diferente cada uno, después Zebra lo gestionará dependiendo del tipo de mensaje.

En la figura 7 se detalla el orden de llamadas realizadas, primero cuando un demonio de encaminamiento se comunica a través de su *socket* cliente con Zebra, utilizando para ello la función ya comentada `zclient_send_message()`. Después, a través del *stream buffer*, llega a Zebra, donde se recibe el mensaje y se procede a su lectura y análisis.

Es importante señalar que por cada demonio cliente, existirá un *socket* para cada uno, aunque solo se represente como uno. De los dieciséis tipos de mensajes que existen en este protocolo de Zebra, buscamos aquellos que transportan la información sobre una red que se debe añadir o eliminar. Por tanto, tras la comprobación de tipos, se ejecutará la función correspondiente.

6.2.6 Gestión de Zebra

Una vez que la información de la ruta ha llegado a Zebra, llega el turno de analizar el proceso servidor. Zebra recibirá de forma concurrente toda la información proveniente de los clientes a través de los sockets.

6.2.6.1 Manejando la información

Partimos desde las funciones citadas anteriormente: `zread_ipv4_add()` y `zread_ipv4_delete()`. Ambas se encuentran en el archivo "*zserv.c*", donde, como ya sabemos, están todas las funciones que forman parte del proceso servidor de Zebra.

Aquí se extrae la información de la ruta o red que envía el demonio cliente, la cual consta de una estructura llamada "*rib*". Este *struct* cuyo significado es "*route information base*", (*base de información de la ruta*), guarda todos los parámetros que puede tener una red alcanzable, es decir, dirección IP, Netmask, métrica, número de saltos, protocolo que descubrió la ruta, distancia, etc.

Toda esa información se introduce en un nodo que se añadirá a una cola, llamada "*work queue*" (cola de trabajo). Esa cola está formada por nodos, en cuyo interior se almacena el *struct rib* de cada ruta que haya llegado a Zebra por medio de un mensaje. Esos nodos reciben el nombre de "*route node*" (nodos de ruta). Al mismo tiempo, otro hilo de ejecución de Zebra, se encarga de procesar esa cola, y su funcionamiento se basa en ir cogiendo el primer nodo de esa cola, la cabecera.

La existencia de esa cola de trabajo es esencial, ya que por un lado tendremos uno o más procesos enviando información, y por otro tenemos a Zebra gestionando y manejando esos mensajes. Por tanto, es necesario un búfer para coordinar esa tarea. Además, no solo estarán los clientes enviando rutas para agregar a esa cola, sino que el propio zebra puede agregar nuevas redes en caso de detectar que una interfaz de red se inicie o se caiga. En ese caso, todas las redes conectadas o alcanzables a esas interfaces se agregarán a la cola de trabajo, para ser tratadas.

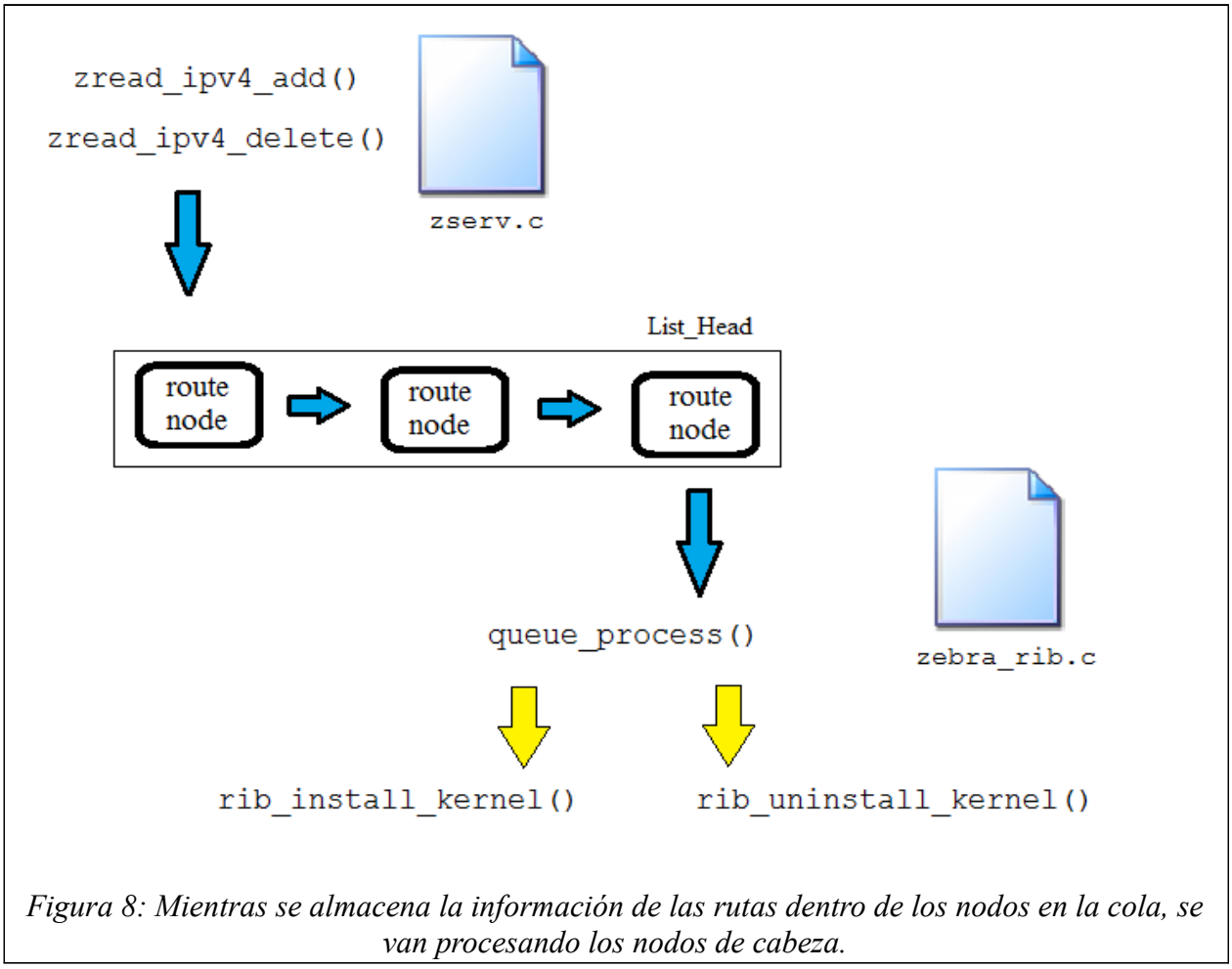

Figura 8: Mientras se almacena la información de las rutas dentro de los nodos en la cola, se van procesando los nodos de cabeza.

Como vemos en la figura 8, la función `queue_process()` va extrayendo los nodos de la cabeza de cola, para tratarlos y gestionarlos. Una vez comprobado cada nodo, si la información que guarda corresponde a una ruta que se debe añadir o eliminar del *kernel* o núcleo, llamará a las funciones encargadas de hacerlo, pasando a tales funciones, el nodo de ruta, donde va almacenada toda la información de la ruta y qué debe hacerse con ella (añadirla, eliminarla o modificarla).

En este momento llegamos a las funciones `rib_install_kernel()` y `rib_uninstall_kernel()`. Estas funciones se encuentran en el archivo "*zebra_rib.c*", que también está en el directorio "zebra", junto a "*zserv.c*". Ambas funciones son las encargadas de agregar o eliminar una ruta o red, en la tabla de rutas del *kernel* o núcleo Linux. Llegamos por fin al momento exacto que buscábamos, en el cual podremos enviar una alerta por SNMP con la información de esa ruta, añadida o eliminada.

Para ello se invocará, dentro de cada una de estas dos funciones anteriores, a una función nueva, llamada `send_trap()`, donde se introducirán los parámetros que maneja la función anterior (`rib_install_kernel` o

`rib_uninstall_kernel`), y que se encargará de construir el Trap-PDU y enviarlo. Esos parámetros se extraen del *struct* de tipo "rib" (los relacionados con la ruta) que ya hemos comentado, y del archivo de configuración "*sendsnmptrap.conf*" (los relacionados con el destino, comunidad, etc.). Este archivo de configuración es necesario añadirlo al software, para gestionar esta nueva funcionalidad que agregamos a Quagga. El diseño de este archivo y de la función `send_trap()`, están explicados en la figura 9.

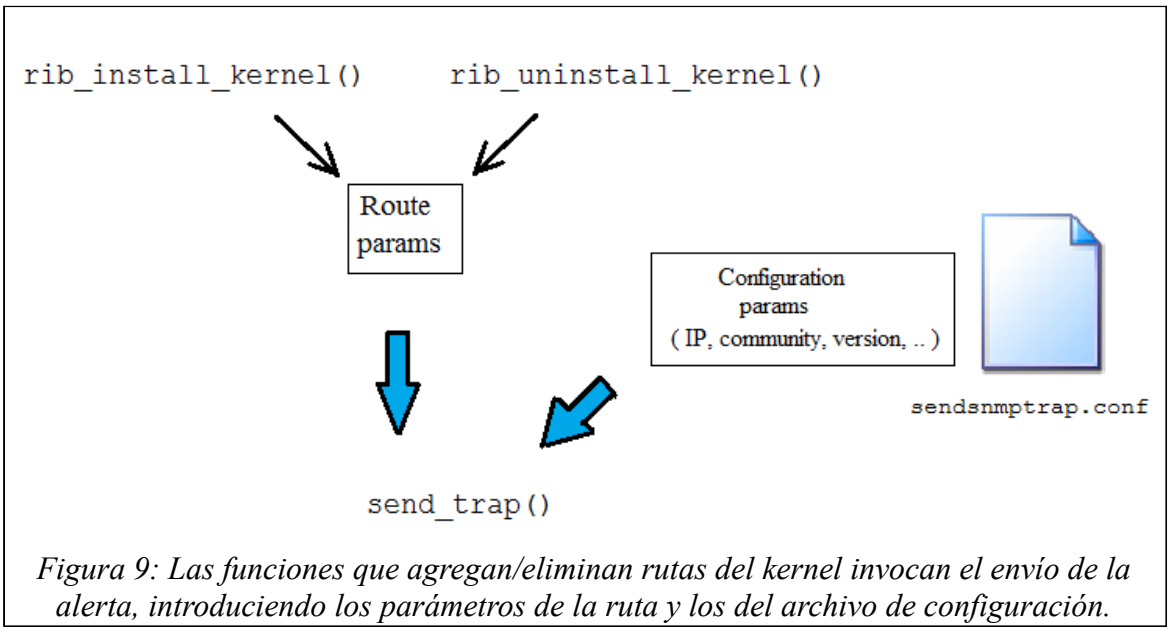

Figura 9: Las funciones que agregan/eliminan rutas del kernel invocan el envío de la alerta, introduciendo los parámetros de la ruta y los del archivo de configuración.

6.2.7 Construyendo el TRAP

Para enviar el Trap de SNMP tenemos que fabricar un datagrama Trap-PDU, con sus correspondientes campos, e introducir dentro la información que queremos enviar, el destino, la comunidad, y todos los que hemos visto que forman parte de un Trap-PDU. La información que corresponde con el destino, la versión a utilizar y la comunidad se recolecta de el archivo de configuración ya citado, muy similar a los ya existentes en Quagga.

6.2.7.1 Nuevas funciones para construir y enviar Trap-PDU

Para la construcción del Trap-PDU, aprovecharemos las funciones implementadas en el software Net-SNMP. Como sabemos, Net-SNMP cubre todas las áreas relacionadas con el protocolo SNMP, es decir, implementa un agente SNMP, un cliente SNMP, las MIB estándares más usadas, aplicaciones de consola para enviar paquetes SNMP a medida y una serie de herramientas para su desarrollo. Aprovechamos entonces la capacidad de este software de enviar paquetes SNMP usando comandos de la consola, extrayendo aquellas funciones que construyen un Trap-PDU usando datos introducidos por consola.

Si nos vamos al código fuente de Net-SNMP, en la carpeta "*apps*", podemos encontrar todas estas aplicaciones de consola por separado, y cada una de ellas corresponde a un archivo en C. Por ejemplo, usando la consola de Linux podemos hacer un "get" en SNMP, con el siguiente comando:

$ snmpget -v 1 -c community destination variable

Donde, "-v" es la versión a utilizar, "-c" es el nombre de comunidad, "destination" es la IP del agente SNMP al que preguntar, y "variable" es la variable que queremos preguntar. Este comando es ejecutado por "*snmpget.c*". Y si queremos enviar un Trap, desde la consola, podemos hacer:

$ snmptrap -v 2c -c community destination uptime trapOid

Donde, "-v", "-c" y "destination" son los mismos campos que en el anterior, "uptime" es un valor que representa el número de segundos que lleva la máquina encendida, si se deja vacío, el agente rellenará automáticamente ese campo con el *uptime* sistema. Y por último "trapOid", que es el OID que genera el Trap. En este caso, como es generado manualmente por línea de comandos, el campo toma el valor que se haya escrito en la consola. Este comando es ejecutado por "*snmptrap.c*"

Esta última aplicación es la que nos interesa, ya que construye un Trap-PDU, con los valores que nosotros damos. Dentro del archivo fuente "*snmptrap.c*" están todas las funciones necesarias para moldear el PDU, rellenarlo con los datos introducidos y enviarlo al destino fijado. Por tanto, basándonos en ese archivo fuente, diseñaremos el nuestro, perfectamente adecuado y modificado para albergar los nuevos parámetros.

La nueva unidad se llama "*sendsnmptrap.c*". Y dentro tenemos la función principal, `send_trap()`, que será invocada cuando se quiera enviar el Trap. Esta función realiza lo siguiente:

- Recibe los parámetros relacionado con la ruta:
  - IP, Netmask, interfaz, métrica, protocolo que generó la información, tipo de ruta (para añadir o eliminar), versión de SNMP, destino y comunidad.
- Fabrica el Trap-PDU, usando las funciones extraídas del software Net-SNMP.
- Introduce dentro del PDU los valores a enviar, es decir, los parámetros de la ruta.
- Establece la sesión SNMP, envía el Trap y cierra la sesión.

6.2.7.2 Envío del Trap-PDU

En este apartado veremos las modificaciones realizadas en las funciones `rib_install_kernel()` y `rib_uninstall_kernel()`. Ya sabemos que dentro de cada una de ellas se invocará al envío del Trap-PDU, llamando a la función `send_trap()`.

Pero antes de invocar a dicha función, se deben realizar una serie comprobaciones, así como obtener los parámetros. Lo primero de todo es comprobar que se ha podido conseguir la información del archivo de configuración. Tal acción se realizó, como veremos en el apartado siguiente, al iniciarse Zebra, y si no se produce ningún error, esos parámetros se almacenan en un vector global. En caso de que no se pudiera obtener información correcta del archivo (mala sintaxis, imposibilidad de leer archivo, etc.), el envío de Traps se desactivará, alertando al usuario, quien tiene la posibilidad de corregirlo.

Lo segundo es consultar la información de ese vector para obtener los datos de envío del Trap. Como veremos en el siguiente apartado, dentro de esta información se encuentran los nombres de aquellos demonios para los cuales el usuario desea recibir las alertas. Por tanto, la alerta se enviará solo si coincide el protocolo generador de la información con el protocolo especificado en el archivo de configuración.

Una vez dentro de las funciones `rib_install_kernel()` y `rib_uninstall_kernel()`, se obtienen los parámetros que vienen almacenados en los nodos de ruta. Dentro de esos nodos de ruta hay muchísima información, la mayoría de esos parámetros son de gestión (valores que usan los demonios para comunicarse entre sí y sincronizarse), y los restantes son los datos de la ruta.

De todos los valores sobre la ruta, extraeremos en concreto ocho, por una sencilla razón. Cuando lanzamos una consulta "GET" SNMP a un dispositivo, y preguntamos sobre su tabla de rutas, sabemos que el agente que recibe esa consulta, cogerá los datos de esa variable (en este caso una tabla) a partir de la MIB correspondiente y nos lo enviaría, en concreto la variable sería:

.1.3.6.1.2.1.4.21 = .iso.org.dod.internet.mgmt.mib-2.ip.ipRouteTable

Al respondernos con la tabla, podremos ver qué valores se almacenan (figura 10) como descriptivos e imprescindibles de una ruta. Esos valores son:

- IP: La dirección IP de la red en cuestión.
- IfIndex: El número de interfaz por donde es alcanzable la IP anterior.
- Metric: La métrica correspondiente a esa red.
- NextHop: El siguiente salto, aquella estación donde debo enviar el paquete destinado a la IP en cuestión.
- Type: El tipo de ruta que es, contiene un *integer*:
  - 2 = invalid; está obsoleta y será eliminada.
  - 3 = direct; es una red directa.
  - 4 = indirect; se accede indirectamente, a través de un nexthop.
- Proto: El protocolo que *aprendió* esa ruta, contiene valores *integer*:
  - RIP = 8
  - OSPF = 13
  - BGP = 14

- Mask: Máscara de red de la IP en cuestión.
- Info: Información de la ruta, Este campo almacena otra variable de tipo OID, que por defecto es {0,0}. Sirve para ampliar y añadir más información a la ruta, pero prácticamente casi nunca se hace, por lo que vale siempre {0,0}.

Por tanto, todos estos valores serán los que extraeremos de los nodos de ruta, y pasaremos a `send_trap()` para que ésta construya y envíe el Trap-PDU.

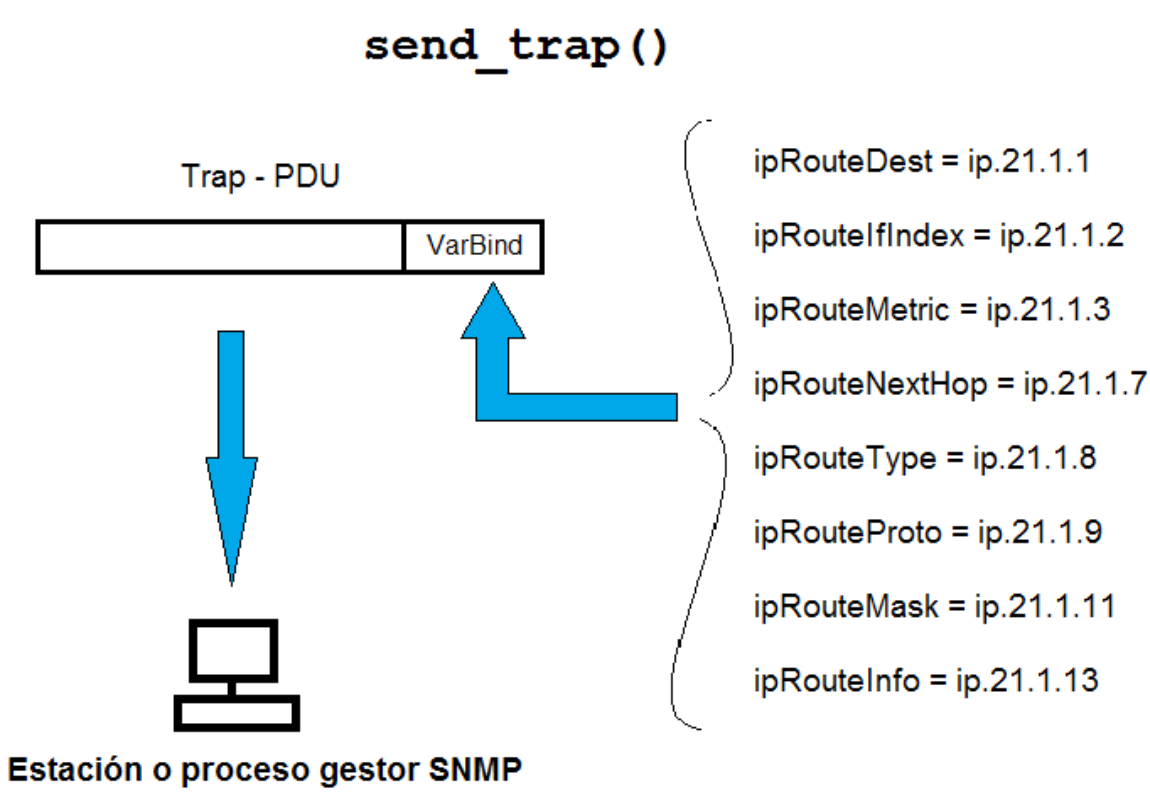

Figura 10: Los valores que llegan a la función se introducen en el mensaje, para ser enviado como Trap al destino prefijado en el archivo de configuración.

Para que todo esto funcione, es necesaria la utilización de funciones de Net-SNMP, por lo que debemos almacenar las librerías que nos brinda el citado software, el cual podemos descargar desde su sitio oficial. Esas librerías se guardan junto a las de Quagga, en el directorio "/lib". Al mismo tiempo, al usar funciones externas, se deben añadir ciertas cabeceras de archivos fuente, para ello se crea el directorio "/include", donde se almacenan todas las cabeceras de Net-SNMP. Por último, se añade una nueva directiva al compilador dentro del archivo *Makefile*, para suministrarle la información sobre dónde encontrar las nuevas librerías y cabeceras:

```
-I$(top_srcdir)/include -L../lib -lsnmp
```

Ya que las modificaciones han sido realizadas todas en el mismo archivo "*zebra_rib.c*", el cual se encuentra en el directorio de "/zebra", el *Makefile* que debe modificarse en concreto sería el que está en el citado directorio.

6.2.7.3 Leyendo del archivo de configuración

Igual que los demonios de encaminamiento tienen sus archivos de configuración para poder explotar al máximos sus posibilidades y adaptarse a las necesidades del usuario, esta nueva funcionalidad necesita de unos ciertos valores para funcionar, que deben ser añadidos por el administrador usuario de este software. Por tanto, en el nuevo archivo de configuración, el administrador debe agregar los parámetros necesarios para el envío del TRAP.

Ejemplo de archivo de configuración "*sendsnmptrap.conf*":

```
!  ** SEND SNMP TRAP WHEN ADD/DELETE ROUTE TO KERNEL **
!
!  ** SYNTAX AND ORDER:
!     daemon snmp_version(v1|v2c) destination(IP) community
!
!  ** SAMPLES: **
!  ripd v1 localhost public
!  ospfd v2c 10.0.0.1 private
!  bgpd v1 192.168.10.12 public
!
!
ripd v2c 192.168.128.1 password
ripd v1 192.168.128.2 password
ospfd v1 192.168.128.1 password
bgpd v2c 192.168.128.1 password

!ripd v1 10.0.0.1 password
!ospfd v2c localhost password
!bgpd v2c 10.0.0.1 password
!
!
```

Dentro del archivo, y siguiendo el orden y la sintaxis indicados, el usuario agrega los parámetros necesarios. Cada línea consta de un TRAP que se enviará cuando el demonio indicado en el primer campo genere la información sobre la red, se enviará el TRAP en la versión indicada en el segundo campo, a la dirección que aparece en el tercer campo, y con la clave de comunidad indicada en el cuarto y último campo.

Al mismo tiempo cabe la posibilidad de enviar varios TRAP a la vez, es decir, si queremos enviar una alerta a dos destinos diferentes cuando *ripd* aprenda una ruta, añadimos una línea nueva, con los campos adecuados. Es una mejora que fue agregada prácticamente al final del desarrollo, y creemos que aumenta de manera significativa la potencia de la nueva funcionalidad agregada a Quagga.

Una vez que sabemos la estructura del archivo nuevo de configuración que hemos diseñado, procedemos a leerlo. Como todas las operaciones de lectura/escritura que se realizan en el software en general, se debe intentar realizar al inicio del programa, una sola vez, y almacenar sus parámetros. Es recomendable, siempre que se pueda, realizar operaciones de L/E el mínimo número de veces, ya que es una acción que requiere más tiempo y recursos que el resto de procesos.

Añadimos por tanto una función nueva en el código, dentro de la unidad "zebra_rib.c". Elegimos esta unidad por una razón, porque aquí se encuentran todas las funciones que ya hemos modificado de Quagga para enviar el Trap-PDU, y de esa forma se pueden realizar todas las modificaciones en el mínimo número de archivos. Al mismo tiempo, esta nueva función sólo será invocada una vez, al inicio del demonio Zebra, para así realizar una sola operación de L/E.

Esta función es:

```
read_snmptrap_config_file(char *commands[]).
```

Cuyo contenido podemos ver dentro del código fuente, tanto en el CD adjunto como en la URL suministrada al final del documento para su descarga. En la línea 922 de la unidad "*zebra_rib.c*" se define la citada función y su funcionamiento es el siguiente:

- Lee línea a línea el archivo de configuración.
- En cada línea, omite los espacios en blanco y analiza cada comando
- Si el primer carácter del primer comando es un "!" entonces es un comentario, y omite toda la línea.
- Comprueba que el primer comando de la línea corresponde con el nombre de alguno de los tres demonios de encaminamiento: *ripd, ospfd, bgpd*.
- Comprueba que el siguiente comando corresponda con alguna de las dos versiones de SNMP que se implementan.
- Comprueba que no falten los comandos de destino y comunidad.
- Si algo falla, se mostrará un error por consola al iniciar Zebra, y el envío de alertas se desactivará.
- El usuario, si quiere utilizar la funcionalidad, debe corregir y reiniciar el demonio.

Si nada falla, todos los comandos se guardan en un vector, para que estén disponibles cuando se realice en envío del Trap-PDU.

6.3 Fase 3: Pruebas y resultados

La fase de pruebas se desarrolló prácticamente al mismo tiempo que las fases anteriores, ya que con cada modificación realizada en el código fuente, no bastaba con compilar y ejecutar el programa, sino que había que probar si el paquete funcionaba correctamente, y realizaba los cambios y acciones que se acababan de hacer.

Debido a que la finalidad del proyecto es dotar a Quagga la capacidad de alertar ante cambios y su tabla de rutas, era necesario realizar unas pruebas que simularan en tiempo real los cambios.

Para ello, la mejor opción consiste en tener una serie de máquinas virtuales conectadas a través de redes virtuales, de tal forma que se diseña una "red de redes", de pequeño tamaño, para que el software de encaminamiento pueda mostrarnos todo su potencial.

6.3.1 Vmware Workstation

Para la realización de las pruebas, y por tanto, la simulación de las máquinas y redes virtuales, se ha utilizado VMware Workstation, que es un sistema de virtualización por software perteneciente a VMware Inc., filial de EMC Corporation.

En un software comercial, aunque existen versiones de prueba, muy extendido. Para la fase de pruebas se usó la versión 7.1.

6.3.2 Máquinas y redes virtuales

En la figura 11 podemos ver de forma muy intuitiva las redes y equipos citados anteriormente. Se crean cinco redes virtuales, la identificación de cada red se escoge de forma arbitraria, a las cuales se conectan varias máquinas de la forma descrita.

Después se crean cuatro máquinas virtuales cuyo S.O. es Ubuntu 10.10 (aunque cualquier distribución que soporte Quagga es válida), donde, en dos de ellas (A y B), se estará ejecutando algún demonio o suite de encaminamiento (por ejemplo Quagga oficial), en la máquina C se ejecutará el Quagga modificado en este proyecto, y en la máquina D se ejecutará el proceso gestor de SNMP que será el encargado de recibir las alertas enviadas por C.

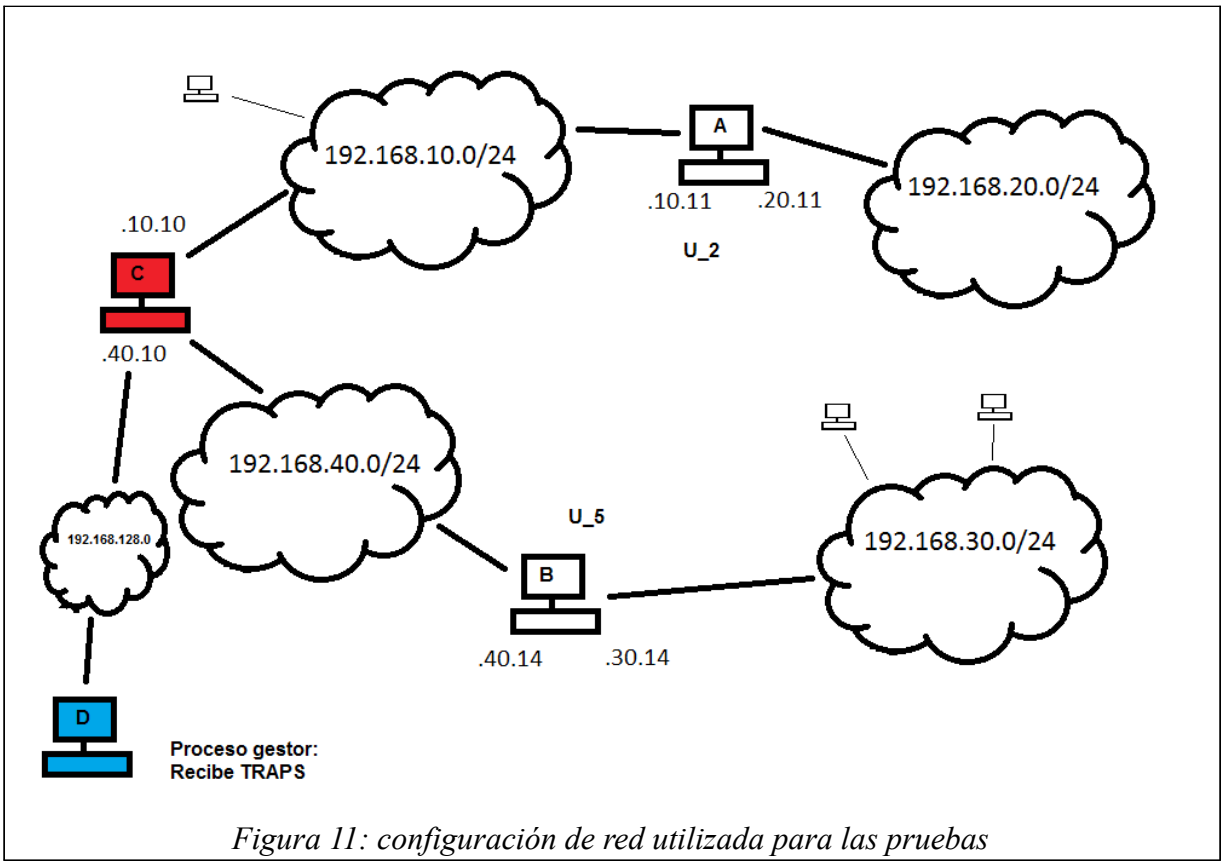

Figura 11: configuración de red utilizada para las pruebas

El objetivo es que la máquina C descubra las redes que no tiene a su alcance, mediante los protocolos de encaminamiento que ya hemos visto anteriormente, y por tanto ejecute las acciones que han sido añadidas, es decir, enviar una alerta usando SNMP, o mejor dicho, un TRAP.

Para ello, en un principio todas las máquinas están apagadas, entonces se encienden en primer lugar las máquinas C y D. Cuando C ha terminado de iniciarse, sólo conoce las tres redes que tiene conectadas directamente (192.168.10.0, 192.168.40.0 y 192.168.128.0). En el caso de la máquina D, comenzará la ejecución del demonio *snmptrapd* perteneciente al software Net-SNMP para poder recibir correctamente los Traps.

Seguidamente, C ejecuta el software Quagga modificado en este proyecto, escuchando por las diferentes interfaces de red y usando los protocolos de encaminamiento que decidamos. En este ejemplo utilizaremos el protocolo RIP, por ser el más sencillo, el cual explicaremos su funcionamiento de forma básica, ya que en el apartado de documentación está convenientemente explicado.

Recordemos que de momento, sólo la máquina C está encendida y ejecutando este software, veamos pues su tabla de rutas actual en la tabla 2.

RED	MÉTRICA	A TRAVÉS DE:
192.168.10.0	1	DIRECTA
192.168.40.0	1	DIRECTA
192.168.128.0	1	DIRECTA

Tabla 2: Tabla de rutas inicial de la estación C.

Suponemos que la máquina C quiere enviar un mensaje/paquete a un equipo que pertenezca la red 192.168.20.0/24, todavía no puede, ya que no aparece en su tabla de rutas.

Acto seguido, encendemos las otras dos máquinas (A y B), que recordemos, ejecutarán el software Quagga original, sin la modificación, por tanto, ahora estas máquinas las podríamos llamar encaminadores. Estos encaminadores, que están utilizando el protocolo RIP para informar de sus redes accesibles y a la vez descubrir las redes que tengan otros, empiezan a comunicarse con nuestro encaminador C.

Supongamos que el encaminador A que está conectado a las redes 192.168.10.0 y 192.168.20.0 se inicia antes que B. Gracias al protocolo RIP informará al encaminador C que está conectado a la red 192.168.20.0.

De esta forma, el encaminador C “aprenderá” que existe una red/ruta nueva, llamada 192.168.20.0/24 a través de 192.168.10.11 (que es la IP del encaminador que informó a C). Por tanto, la tabla de rutas que vimos más arriba, ahora contendrá lo mostrado en la tabla 3.

RED	MÉTRICA	A TRAVÉS DE:
192.168.10.0	1	DIRECTA
192.168.40.0	1	DIRECTA
192.168.128.0	1	DIRECTA
192.168.20.0	2	192.168.10.11

Tabla 3: Tabla de rutas de la estación C con información de la estación A.

A continuación, el encaminador B, el cual está conectado a las redes 192.168.40.0 y 192.168.30.0, termina de iniciarse, y comienza a comunicarse con C. De esta forma, el encaminador C “aprende” que B está conectado a la red 192.168.30.0/24. Así que, C lo anota en su tabla, quedando de la que muestra la tabla 4:

RED	MÉTRICA	A TRAVÉS DE:
192.168.10.0	1	DIRECTA
192.168.40.0	1	DIRECTA
192.168.128.0	1	DIRECTA
192.168.20.0	2	192.168.10.11
192.168.30.0	2	192.168.40.14

Tabla 4: Tabla de rutas de la estación C con información de la estación B.

La finalidad de todo esto radica en que ahora se comprueba si al añadir al *kernel* o núcleo esa ruta nueva, nuestro software que se ejecuta en el encaminador C, ejecutará su mejora, y enviará esa alerta Trap a través de SNMP, al destino que haya sido prefijado en el archivo de configuración, en este caso a D.

La siguiente prueba consiste en comprobar si el encaminador C alerta al gestor D sobre una ruta que se ha convertido en inaccesible. Para conseguir tal fin, simulamos una caída del encaminador A deteniendo los procesos pertenecientes a Quagga. Ahora C deja de mantener contacto con A, y por tanto termina la recepción de información sobre la red 192.168.20.0, por lo que pasado el tiempo establecido en el protocolo RIP para el caso de una red que se ha vuelto inaccesible, el proceso *ripd* que se ejecuta en C marca a esa red como inválida. Acto seguido y gracias a la modificación realizada, la máquina C alerta sobre el cambio que acaba de producirse enviando un Trap al gestor (D).

De esta forma, comprobamos si las modificaciones que realiza este proyecto sobre el software Quagga son correctas, y cumplen las funciones descritas en las fases anteriores.

7 MODO DE USO

El modo de uso de este software es prácticamente el mismo que Quagga, con la excepción de que ahora existe un nuevo archivo de configuración, y un nuevo directorio donde el usuario "quagga" debe tener permisos.

7.1 Instalación

El método de instalación es igual que en Quagga original. Primero se descomprime todo el contenido en una carpeta y luego fijamos el directorio actual de la consola dentro de esta carpeta. Ejecutamos los comandos típicos de instalación de software:

```
$ ./configure
$ make
# make install
```

A la hora de hacer el "./configure", Quagga nos ofrece múltiples opciones, que podemos encontrar explicadas en su documentación oficial. A continuación mostramos las más comunes:

- `--disable-ipv6` , instalar deshabilitando *ipv6*.
- `--disable-ripd` , instalar deshabilitando *ripd* (o el demonio que se ponga).
- `--sysconfdir=dir` , instalar archivos de configuración en *dir*.

Después de la configuración, Quagga nos pide que se agregue un usuario llamado "quagga", que sea propietario de unas carpetas.

```
# adduser quagga
# chown quagga /usr/local/etc
# chown quagga /var/run
# chown quagga /var/lib/snmp
```

El último comando es nuevo y propio de esta versión de Quagga. Es necesario para que pueda comunicarse con el agente SNMP que reside en el mismo dispositivo. Si no se ejecuta, el software funcionará correctamente aunque obtendremos una advertencia cada vez que se envíe un mensaje SNMP.

Se instalan los archivos de configuración (más bien se les cambia el nombre). Por defecto se instalan en: "/usr/local/etc", o a veces se instalan en /etc/quagga, depende de la distribución que usemos:

```
$ cd /usr/local/etc/
# cp zebra.conf.sample zebra.conf
# cp ripd.conf.sample ripd.conf
# cp ospfd.conf.sample ospfd.conf
# cp bgpd.conf.sample bgpd.conf
```

El nuevo archivo de configuración, "sendsnmptrap.conf", lo emplazaremos dentro del directorio que hayamos elegido para albergar al resto de archivos de configuración de Quagga. A continuación modificamos el propietario del fichero y concedemos permisos de lectura.

```
# chown quagga /usr/local/etc/sendsnmptrap.conf
# chmod +r /usr/local/etc/sendsnmptrap.conf
```

Recordamos que debe estar instalado el software agente de Net-SNMP, para utilizar la funcionalidad del envío de Traps de esta versión de Quagga. Si no está instalada, podemos usar el gestor de paquetes *APT* o el que corresponda a la distribución usada, y descargar el paquete *snmpd*, ya que es el software SNMP por defecto en la comunidad GNU/Linux y soportado por casi todas las distribuciones:

```
# apt-get install snmpd
```

7.2 Ejecución

Para ejecutarlo, igual que Quagga original, primero iniciamos Zebra y después el/los demonio/s encaminadores que queramos:

```
$ zebra -d && ripd -d
```

Este comando ejecuta Zebra y ripd con la opción -d (modo demonio).

8 CONCLUSIONES, LOGROS Y PROPUESTAS FUTURAS

8.1 Conclusiones

Con este proyecto se han conseguido ampliaciones importantes en el paquete de encaminamiento Quagga. Hasta ahora sólo era posible instalar Quagga con soporte SMUX, tal y como vimos en el apartado *Estado del arte.*

Zebra utiliza SMUX para establecer la comunicación con el agente SNMP, y su funcionamiento es el siguiente.

- Se instala un agente SNMP en el mismo dispositivo que alberga a Quagga.
- Quagga se compila e instala con la configuración necesaria para ejecutar SMUX.
- Se habilita el agente SNMP para que también use SMUX.
- El agente SNMP sabe que cuando le pregunten sobre variables pertenecientes a Quagga, éste le pasará la consulta al agente *proxy*, que a su vez ejecutará la consulta y devolverá el valor al agente principal.

Cuando se realiza una consulta de tipo “Get-PDU” a este dispositivo preguntando sobre alguna variable de Quagga, el agente SNMP le pasa la consulta al *proxy*, que es el encargado de obtener la información y se la devuelve al agente. Como hemos visto, esta operación es normalmente ejecutada por el agente únicamente, pero en estos casos, en los que el dispositivo no admite SNMP, se necesita el *proxy* para acceder.

Por esta razón, la utilización de SMUX solo nos permite realizar un “GET”, y no permite el envío de Trap-PDU, que es la funcionalidad más importante de cualquier dispositivo SNMP. Al mismo tiempo se complica la instalación y usabilidad del software Quagga.

Con nuestra solución, Quagga no se complica, y mejora notablemente su funcionalidad y potencia de administración remota, ya que el envío de alertas Trap es la parte de SNMP más demandada por los administradores de red.

8.2 Logros

Se ha conseguido lograr el principal objetivo de este proyecto. Quagga puede enviar Traps-PDU a través de SNMP, al destino y comunidad que se elija, en la versión 1 ó 2c.

Al principio del proyecto, una vez encontrados los métodos que usan los demonios para comunicarse con Zebra, se procedió a realizar las modificaciones en cada uno de los demonios. Pero a la hora de depuraciones y extensibilidad, pensamos que la mejor opción era realizar la mejora sólo en el demonio principal. De esa forma, ganamos en simplicidad y a la hora de realizar un archivo de configuración, ya que de esta forma sólo es necesario uno, de la otra forma eran necesarios uno por demonio.

También al comienzo, el envío de alertas Trap se realizaba hacia un único destino, y en una versión y comunidad prefijada dentro del código. Así surgió el archivo de configuración nuevo, y los métodos para extraer información, por lo que se ganó en potencia y adaptabilidad a la hora de elegir los parámetro de envío, tal y como quedó explicado en el apartado 6.2.7.3 del presente documento.

8.3 Propuestas de futuras mejoras

8.3.1 IPv6

Una de las tareas que quedan pendientes, sería adaptar esta funcionalidad para ser utilizada con los demonios encargados de implementar protocolos con IPv6, es decir, ripngd, ospf6d, etc, ya que por ahora sólo funciona con aquellos que trabajan con IPv4.

Esta limitación se debe a la hora de enviar el Trap-PDU, que sólo permite almacenar ciertos tipos de variables, entre los que no se encuentra el tipo de dirección IPv6. En caso de que, en un futuro, se llegara a admitir ese tipo, el cambio a realizar sería muy similar al realizado en este proyecto, pudiendo aprovechar numerosas funciones.

Para empezar, si partimos de `rib_install_kernel(..)`, donde ahora lanzamos la llamada a `send_trap()`, hay dos apartados, el primero para rutas de tipo IPv4, en el cual está realizada la modificación, y el segundo para rutas de tipo IPv6. En este apartado deberíamos llamar a una función similar a `send_trap()`, pero con los parámetros adecuados respecto a IPv6.

8.3.2 Enviar Traps en otros casos

Otra mejora futura, puede consistir en enviar alertas sobre más tipos de eventos, no solo de rutas nuevas o eliminadas. Por ejemplo, cuando se levanten o caigan interfaces de red, o al iniciarse o pararse demonios de encaminamiento. Cualquier evento del cual se necesite estar informado en tiempo real es una buena opción.

8.3.3 Futuras versiones

Todos y cada uno de los cambios realizados están documentados y explicados, tanto en esta memoria, como en el código fuente y demás archivos README. Este proyecto será enviado a la comunidad de desarrolladores de Quagga oficial, para incluir las modificaciones realizadas.

Se habilitará un sitio web, indicado en el apartado de Referencias, para hospedar este proyecto. En caso de salir nuevas versiones de Quagga, en las que todavía no se hayan incluido estas mejoras, se procederá a lanzar este software en la versión actualizada en el sitio web de hospedaje. Ese sitio web también servirá para recibir consultas, dudas, o bugs si los hubiera de este software, y a la vez fomentar el desarrollo del mismo.

9 PROBLEMAS Y SOLUCIONES

DOCUMENTACIÓN.

El primer problema que nos encontramos a la hora de realizar cualquier mejora o modificación a la suite de encaminamiento Quagga es la falta de documentación técnica. Nos referimos en concreto a alguna serie de documentos donde poder encontrar referencias del código fuente, sobre las unidades de las que se compone el software y de las funciones usadas en cada una de las unidades.

De todo lo concerniente a la documentación del propio desarrollo del software y de la gestión del proyecto, pasando por modelaciones (UML), casos de uso, diagramas, pruebas, manuales de usuario, manuales técnicos, etc; todo con el propósito de eventuales correcciones, usabilidad, mantenimiento futuro y ampliaciones al sistema, sólo podemos encontrar de forma oficial un manual de uso del software, con las órdenes de consola para manejar la suite, los comandos CLI, instrucciones de instalación y algún ejemplo de configuración.

Con lo cual, a la hora de seguir el propio código fuente, la tarea se complica enormemente, ya que nos vemos con la necesidad de *deducir* el funcionamiento del software, el orden de las llamadas a funciones, el comportamiento en cada etapa, etc.

A todo lo anterior, hay que sumar la escasez de comentarios dentro del código fuente del paquete Quagga, llegando a casos tan extremos como que podemos encontrar un par de líneas de comentarios para cientos de líneas de código, entremezcladas con funciones, bucles, y demás sintaxis difícil de seguir sin los comentarios adecuados.

EXPERIENCIA EN C.

El siguiente problema encontrado durante la realización del proyecto tiene que ver con nuestra experiencia individual con el lenguaje de programación del cual está construido el software usado. Todos los miembros del equipo teníamos gran experiencia en Java y C++, todo ellos relacionado con programación a objetos y utilización de clases. Por tanto, el cambio de enfoque de la programación con objetos, hacia una programación en C sin objetos, nos condujo a pequeños errores a la hora de programar, que eran fácilmente solucionables pero que la falta de experiencia nos influyó a tener algunas pérdidas de tiempo.

FALTA DE MEDIOS.

Este problema no es grave en sí, porque el desarrollo del proyecto ha podido ser llevado a cabo perfectamente con los medios disponibles, pero debido a los escenarios donde luego podrá ser utilizado este software final modificado, hemos echado en falta poder realizar más pruebas y tests en escenarios reales, con tráfico real. Pero a pesar de todo, consideramos suficientes las pruebas realizadas en las redes simuladas con las ventajas que supone la virtualización.

LICENCIAS.

Debido a la falta de experiencia en el desarrollo de software libre, nos encontramos con algo de desinformación sobre las licencias del software utilizado y sobre el software nuevo que se ha implementado. Aunque es una situación de fácil solución, cabe destacar la

importancia de los distintos tipos de licencias de software, así como saber y conocer la más adecuada para nuestro proyecto.

10 REFERENCIAS BILBIOGRÁFICAS

[1] *"ISO/IEC 7498-1. Information Technology Open Systems Interconnection – Basic Refrence Model: The Basic Model International Standard."*

[2] RFC 1155 *"Structure and Identification of ManagementInformation for TCP/IP-based Internets."* M. Rose, K. McCloghrie. May 1990

[3] *"Using Content Networking to Provide QoS."* http://www.cisco.com/en/US/products/ps6612/products_white_paper09186a00804fce7f.shtml - Cisco Systems Inc.

[4] RFC 1227 *"SNMP MUX protocol and MIB."* M. Rose. May 1991

[5] *"SMUX configuration."* http://www.quagga.net/docs/docs-multi/SMUX-configuration.html - Quagga support

[6] *"Mobile Routing variant of Quagga."* http://downloads.pf.itd.nrl.navy.mil/ospf-manet/quagga-0.99.17mr2.0/RELEASE_NOTES.MobileRouting - Richard Ogier, Phillip Spagnolo

[7] *"GNU Zebra."* http://www.zebra.org/ - GNU Zebra Team

[8] *" Quagga Documentation A routing software package for TCP/IP networks."*

[9] *"Cisco IOS Command Line Interface Tutorial."* http://www.cisco.com/warp/cpropub/45/tutorial.htm - Cisco Systems Inc.

[10] *"What is RIP? Routing Information Protocol."* http://www.pulsewan.com/data101/rip_basics.htm - Pulse Inc.

[11] RFC 4822 *"RIPv2 Cryptographic Authentication."* R. Atkinson, M. Fanto. February 2007

[12] RFC 2080 *"RIPng for IPv6."* G. Malkin, R. Minnear. January 1997

[13] RFC 2238 *"Definitions of Managed Objects for HPR using SMIv2."* B. Clouston, Editor, B. Moore, Editor. November 1997

[14] RFC 5340 *"OSPF for IPv6."* R. Coltun, D. Ferguson, J. Moy, A. Lindem. July 2008

[15] RFC 4271 *"A Border Gateway Protocol 4 (BGP-4)."* Y. Rekhter, Ed; T. Li. January 2006

[16] *"BGP: the Border Gateway Protocol Advanced Internet Routing Resources."* www.bgp4.as - BGP4.AS

[17] RFC 4293 *"Management Information Base for theInternet Protocol (IP)April 2006."* S. Routhier. April 2006

[18] RFC 4022 *"Management Information Base for theTransmission Control Protocol (TCP)."* R. Raghunarayan. March 2005

[19] RFC 4113 *"Management Information Base for theUser Datagram Protocol (UDP)."* B. Fenner, J. Flick. June 2005

[20] RFC 2863 *"The Interfaces Group MIB."* K. McCloghrie, F. Kastenholz. June 2000

[21] RFC 3418 *"Management Information Base (MIB) for theSimple Network Management Protocol (SNMP)."* R. Presuhn, Authors of previous version: J. Case, K. McCloghrie et al. December 2002

[22] RFC 1157 *"A Simple Network Management Protocol (SNMP)."* J. Davin. May 1990

[23] RFC 1089 *"SNMP over Ethernet."* C. Davin, M. Fedor, J. Case. February 1989

[24] RFC 1418 *"SNMP over OSI."* M. Rose. March 1993

[25] RFC 1187 *"Bulk Table Retrieval with the SNMP."* M. Rose, K. McCloghrie, J. Davin. October 1990

[26] RFC 1270 *"SNMP Communications ServicesApplications."* F. Kastenholz. October 1991

[27] RFC 1303 *"A Convention for Describing SNMP-based Agents."* K. McCloghrie, M. Rose. February 1992

[28] RFC 2573 *"SNMP Applications."* D. Levi, P. Meyer, B. Stewart. December 2002

[29] RFC 3413 *"Simple Network Management Protocol (SNMP)Applications."* D. Levi, P. Meyer, B. Stewart. December 2002
[30] RFC 5592 *"Secure Shell Transport Model for the SimpleNetwork Management Protocol (SNMP)."* D. Harrington, J. Salowey, W. Hardaker. June 2009
[31] RFC 5675 *"Mapping Simple Network Management Protocol (SNMP)Notifications to SYSLOG Messages."* V. Marinov, J. Schoenwaelder. October 2009
RFC 5935Expressing SNMP SMI Datatypes in XML Schema
[33] RFC 1901 *"Introduction to Community-based SNMPv2."* J. Case, K. McCloghrie, M. Rose, S. Waldbusser. January 1996
[34] RFC 1445 *"Administrative Model for version 2 of the SimpleNetwork Management Protocol (SNMPv2)."* J. Galvin, K. McCloghrie. April 1993
[35] RFC 1446 *"Security Protocols for version 2 of the SimpleNetwork Management Protocol (SNMPv2)."* J. Galvin, K. McCloghrie. April 1993
[36] RFC 1447 *"Party MIB for version 2 of the Simple NetworkManagement Protocol (SNMPv2)."* K. McCloghrie, J. Galvin. April 1993
[37] RFC 3416 *"Version 2 of the Protocol Operations for theSimple Network Management Protocol (SNMP)."* R. Presuhn, Authors of previous version: J. Case, K. McCloghrie, et al. December 2002
[38] RFC 5590 *"Transport Subsystem for the Simple NetworkManagement Protocol (SNMP)."* R. Presuhn, Authors of previous version: J. Case, K. McCloghrie, et al. June 2009
[39] RFC 3584 *"Coexistence between Version 1, Version 2,and Version 3 of the Internet-standard NetworkManagement Framework."* R. Frye, D. Levi, S. Routhier, B. Wijnen. August 2003
[40] RFC 1503 *"Algorithms for Automating Administration in SNMPv2Managers."* K. McCloghrie, M. Rose. August 1993
[41] RFC 1901 *"Introduction to Community-based SNMPv2."* J. Case, K. McCloghrie, M. Rose, S. Waldbusser. January 1996
[42] RFC 3414 *"User-based Security Model (USM) for version 3of the Simple Network Management Protocol (SNMPv3)."* U. Blumenthal, B. Wijnen. December 2002

11 OTROS DOCUMENTOS CONSULTADOS

A continuación se detalla una lista de otros documentos que se han consultado durante el desarrollo del proyecto, pero que no han sido citados como referencias en el contenido de la memoria.

- Interconectividad de redes con TCP/IP: Volumen II Diseño e Implementación.
  - Douglas E. Comer, David L.Stevens.
  - Prentice Hall, ISBN: 970-26-0000-6
- Protocolos de internet: Diseño e implementación en sistemas Unix
  - Ángel López, Alejandro Novo.
  - RA-MA, ISBN: 9788478973828
- TCP/IP: Arquitectura, protocolos e implementación con IPv6 y seguridad de IP.
  - Dr. Sidnie Feit.
  - McGraw-Hill, ISBN 84-481-1531-7

- Sitio web del software Quagga y su documentación oficial:
  - http://www.quagga.net
  - http://quagga.net/docs.php
- Documentación en español
  - http://www.microalcarria.com/descargas/documentos/Linux/redes/routing/Quagga/Manual_Quagga_castellano/
- Descarga Quagga oficial:
  - http://quagga.net/download.php
- Sitio web del software Net-SNMP y su documentación:
  - http://www.net-snmp.org
  - http://www.net-snmp.org/wiki/index.php/Tutorials
- Funcionamiento del TRAP con Net-SNMP:
  - http://www.net-snmp.org/wiki/index.php/TUT:snmptrap
- Lista de distribución de los desarrolladores de Net-SNMP:
  - http://sourceforge.net/mailarchive/forum.php?forum_name=net-snmp-coders
- Lista de distribución de los desarrolladores de Quagga:
  - http://lists.quagga.net/pipermail/quagga-dev/
- Página de CISCO donde navegar por el árbol jerárquico de los objetos MIB:
  - http://tools.cisco.com/Support/SNMP/public.jsp
- RFCs relacionados con SNMP:
  - http://datatracker.ietf.org/doc/search/?name=snmp&rfcs=on
- Licencia GPL:
  - http://www.gnu.org/licenses/gpl.html
- Otros enlaces de interés:
  - http://www.gossamer-threads.com/lists/quagga/dev/10858
  - http://quagga.net/faq/zebra-hacking-guide.txt

- Sitio web de Google Code donde se hospeda este proyecto:
  - http://code.google.com/p/quagga-trap-version/

RFC
RFCs soportados por Quagga: ver apéndice B

Otros RFCs:

- RFC 5000 *"Internet Official Protocol Standards"* May 2008.
- RFC 4552 *"Authentication/Confidentiality for OSPFv3"* M. Gupta, N. Melam. June 2006.
- RFC 1155 *"Structure and Identification of Management Information for TCP/IP-based Internets"* M. Rose, K. McCloghrie. May 1990.

12 CONTENIDO DEL CD

Este proyecto adjunta un CD con contenido de interés para futuros desarrollos, en concreto, se adjunta todo el código fuente de Quagga snmpTRAP 0.99.18, junto con las instrucciones de funcionamiento e instalación. Además de toda la documentación del código generada en el proyecto.

A continuación se detalla la estructura de carpetas:

- quagga-snmpTRAP-0.99.18 : Todo el código fuente del proyecto.
- Scripts de ejemplo: Este directorio contiene los siguientes archivos:
  - quagga-snmpTRAP.sh : una ayuda para iniciar Quagga, simplemente ejecuta algunos comandos sobre permisos.
  - trap_handler.sh : Este script de ejemplo puede utilizarse en la estación que ejecuta el demonio que recibe los Trap-PDU, es decir, que ejecuta *snmptrapd*. Net-SNMP nos permite llamar a un script propio cuando recibe algún Trap-PDU y "trap_handler.sh" es un buen ejemplo. Simplemente se limita a guardar en un log toda la información ordenada del Trap, y realizar pruebas hasta que tengamos nuestro propio script.
- Archivos de configuración: Aquí encontramos los archivos de configuración comunes en Quagga, pero modificados para poder realizar las pruebas mostradas en esta memoria. Al mismo tiempo agregamos en este directorio un ejemplo del nuevo archivo de configuracón, "sendsnmptrap.conf", y un ejemplo de archivo de configuración del proceso *snmptrapd*: "snmptrapd.conf"

A. CAMBIOS REALIZADOS SOBRE QUAGGA ROUTING SUITE

Como sabemos, este software es una versión de Quagga, basado en la versión oficial, con nuevas funcionalidades. Durante toda la fase de desarrollo, se ha ido comentando una parte del código fuente, y explicando los lugares donde realizar las modificaciones y su motivo. Para futuras consultas, modificaciones o ampliaciones, ponemos en este apartado todos y cada uno de los cambios realizados sobre el código fuente de Quagga-0.99.18.

- Archivo: zebra/zebra_rib.c
  - Línea 44: Se añade: `#include "sendsnmptrap.c"`

- Línea 55-58: declaración variables globales para el archivo de configuración:
  ```
  char *snmptrap_commands[CONFIG_COMMANDS_MAXSIZE];
  int snmptrap_state;
  ```

- Línea 922: nueva función para leer el archivo de configuración.
  ```
  int read_snmptrap_config_file(char *commands[])
  ```

- Línea 982-1025: dentro de la función `rib_install_kernel(..)`, se agrupan los parámetros y se llama a la función `send_trap()`.

- Línea 1058-1097: dentro de la función `rib_uninstall_kernel(..)`, se agrupan los parámetros y se llama a la función `send_trap()`.

- Línea 3061-3063: dentro de `rib_close()`, liberación del vector archivo de configuración.

- Línea 3073-3083: dentro de `rib_init()`, se llama la función de la línea 922, para leer el archivo de configuración.

- Archivo zebra/Makefile.in
  - Se añaden los comandos para enlazar e incluir:
    - Línea 153: `-I$(top_srcdir)/include -L../lib -lsnmp`
    - Línea 310: `../lib/libsnmp.la`
    - Línea 311: `../lib/libsnmp.la`
      Esta última línea corresponde a los comandos para enlazar librerías cuando se ejecutan los test de Zebra, por eso parece repetitiva.

- Archivo README
  - Línea 9: Se avisa de que existe otro README específico de esta versión.

- Archivos incluidos:
  - Librería *libsnmp* dentro de la carpeta /lib de Quagga.
  - Archivo nuevo, sendsnmptrap.c, dentro del directorio /lib de Quagga.
  - Se agrega el directorio “include” al directorio raíz, que contiene todas las cabeceras de Net-SNMP.
  - *Se añade un archivo README llamado: README-Quagga-snmpTrap.txt, en el cual se explican todas las mejoras con respecto al Quagga original, direcciones y sitio web en el que enviar comentarios o posibles bugs.*

B. RFCS SOPORTADOS POR QUAGGA

- RFC 4822
  "RIPv2 Cryptographic Authentication" R. Atkinson, M. Fanto. February 2007.

- RFC 2080
  "RIPng for IPv6" G. Malkin, R. Minnear. January 1997.
- RFC 5709
  "OSPFv2 HMAC-SHA Cryptographic Authentication" M. Bhatia, V. Manral, M. Fanto, R. White, M. Barnes, T. Li, R. Atkinson. October 2009.
- RFC 5250
  "The OSPF Opaque LSA Option" L. Berger, I. Bryskin, A. Zinin, R. Coltun. July 2008.
- RFC 3101
  "The OSPF Not-So-Stubby Area (NSSA) Option" P. Murphy. January 2003.
- RFC 5340
  "OSPF for IPv6" R. Coltun, D. Ferguson, J. Moy, A. Lindem, Ed. July 2008.
- RFC 4271
  "A Border Gateway Protocol 4 (BGP-4)" Y. Rekhter, Ed., T. Li, Ed., S. Hares, Ed. January 2006.
- RFC 5065
  "Autonomous System Confederations for BGP" P. Traina, D. McPherson, J. Scudder. August 2007.
- RFC 1997
  "BGP Communities Attribute" R. Chandra, P. Traina, T. Li. August 1996.
- RFC 2545
  "Use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain Routing" P. Marques, F. Dupont. March 1999.
- RFC 4456
  "BGP Route Reflection:An Alternative to Full Mesh Internal BGP (IBGP)" T. Bates, E. Chen, R. Chandra. April 2006.
- RFC 2858
  "Multiprotocol Extensions for BGP-4" T. Bates, R. Chandra, D. Katz, Y. Rekhter. January 2007.
- RFC 5492
  "Capabilities Advertisement with BGP-4" J. Scudder, R. Chandra. February 2009.
- RFC 3137
  "OSPF Stub Router Advertisement" A. Retana, L. Nguyen, R. White, A. Zinin, D. McPherson. June 2001.

Cuando el soporte para SNMP por proxy está habilitado, también soporta:

- RFC 1227
  "SNMP MUX protocol and MIB" M. Rose. May 1991.
- RFC 4273
  "Definitions of Managed Objects for BGP-4" J. Haas, Ed., S. Hares, Ed. January 2006.
- RFC 1724
  "RIP Version 2 MIB Extension" G. Malkin, F. Baker. November 1994.
- RFC 4750
  "OSPF Version 2 Management Information Base" D. Joyal, Ed., P. Galecki, Ed., S.

Giacalone, Ed. Original Authors: R. Coltun, Touch Acoustra, F. Baker. December 2006.

Los alumnos abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Rubén Gamarra Rodríguez

Luis Villacastin Candil