MPLS EN EL KERNEL DE LINUX

IMPLEMENTACION DE ETHERNET SOBRE MPLS

Versión 1.5.0

Diego Gonzalez – Gustavo Compagnone – Diego Lorenzo
Facultad de Ingeniería - Universidad de la República
Montevideo
2006

Tabla de Contenidos

1. Introducción

Este proyecto tiene como objetivo el agregado de una funcionalidad al proyecto dirigido por James Leu y Ramón Casellas que implementa el sistema MPLS para el kernel de Linux. El agregado que se hizo es la implementación y posteriores pruebas de la funcionalidad de encapsular ethernet sobre MPLS. Esto último es la parte fundamental de las funcionalidades necesarias para implementar pseudos cables.

Para tal fin, es necesario realizar un estudio profundo del sistema MPLS, y (como se verá en el desarrollo del presente documento) para el estudio del citado proyecto también fue necesario un estudio pormenorizado del stack IP del kernel de Linux, dada la gran interrelación entre ambos.

El trabajo se dividió en tres etapas que se describen brevemente a continuación y que serán desarrolladas más adelante.

1) Adquisición de los conocimientos involucrados para poder desarrollar el objetivo final.

Este punto se dividió básicamente en el estudio de tres temas; Lenguaje C, implementación del stack IP en el kernel de linux y fundamentos de MPLS. El estudio del Stack IP del kernel de linux se encuentra en los capítulos 2 y 3, y en el capítulo 4 se presentan los fundamentos de MPLS.

2) Definición del alcance del proyecto.

Para esto se estudió la implementación existente del sistema MPLS en linux desde el punto de vista del usuario (capítulo 5) y del desarrollador (capítulo 6) para luego poder definir las modificaciones a realizar.
Al finalizar la segunda etapa y luego de un análisis de las distintas opciones (hechas en conjunto con los tutores) se decidió agregar soporte para encapsular ethernet sobre MPLS (ver figura 1.1).

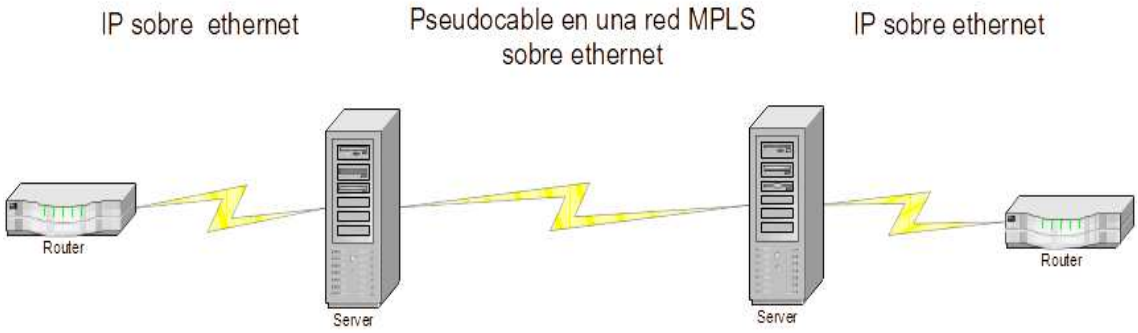

Figura 1.1

La aplicación de esta implementación es dar a Linux la capacidad de tomar todos los frames recibidos por una interfase ethernet y reenviarlos encapsulados sobre MPLS a través de otra interfase ethernet. En el paquete transmitido hacia la red MPLS hay dos encabezados ethernet, el encabezado original que se encuentra encapsulado dentro del paquete MPLS y un segundo encabezado que es agregado por la interfase que transmite el paquete. De la misma forma en el otro extremo de la comunicación se recibe un frame ethernet encapsulado en MPLS, por lo tanto es necesario desencapsularlo y retransmitirlo como un frame ethernet convencional (ver figura 1.2).

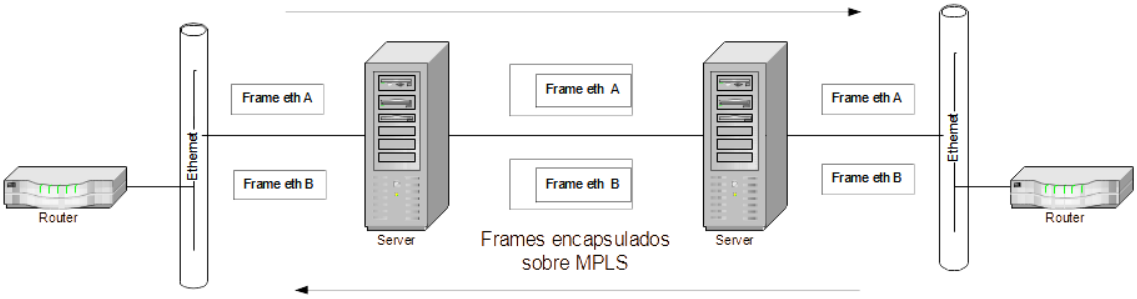

Figura 1.2

Al final de este capítulo describimos las otras opciones que se consideraron y no fueron tomadas en cuenta para continuar el proyecto.

> 3) Finalmente, una vez definido el alcance, la última etapa del proyecto es la modificación del código del sistema MPLS para implementar la funcionalidad requerida.
> A continuación presentaremos (y también de modo de describir más en detalle la nueva funcionalidad) las motivaciones que definieron nuestro objetivo.

Motivación de los Pseudo cables

Para un proveedor que comercializa múltiples servicios, la tendencia actual es usar redes convergentes en lugar de tener infraestructuras paralelas (Frame Relay, ATM, etc.) para dar cada servicio. Esta tendencia está sustentada por la simplificación operativa (mantenimiento, configuración, stock, etc.) y minimización de costos (de capital y operacionales).
En virtud de lo anterior, una red convergente puede ser dimensionada de manera eficiente para dar todos los servicios en conjunto.

Dado que el tráfico de paquetes ocupa cada vez un porcentaje mayor del ancho de banda (recursos) de las redes de operadores, cada vez es más conveniente optimizar las redes para este tipo de tráfico. Por otro lado, si bien el tráfico de Internet (IP) es el que está creciendo más aceleradamente, éste no es el que genera mayor rentabilidad por bit, siendo el tráfico ATM, TDM o Frame Relay el principal generador de rentabilidad por bit.

Por lo expuesto anteriormente y dada la cantidad de equipamiento existente que opera con tecnologías tradicionales, los operadores ven cada vez como una necesidad mayor el interconectar estos equipos a través de las redes de paquetes (IP) optimizadas, manteniendo así los servicios TDM, ATM o Frame Relay tan lucrativos históricamente.

La figura siguiente representa la idea anterior, en donde se utiliza una red de paquetes IP, a la que convergen las tecnologías existentes.

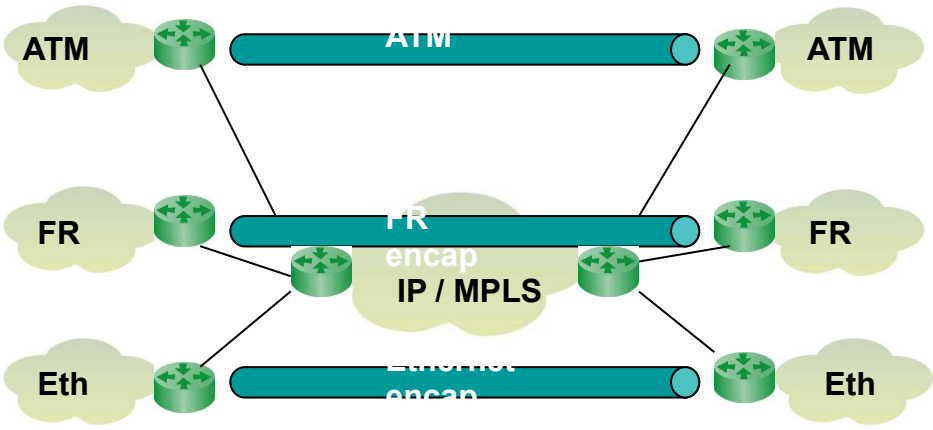

Figura 1.3 – Backbone de paquetes para la convergencia de tecnologías de banda ancha.

Es así, como nace la necesidad de emular servicios utilizando “Pseudo cables” que puedan ser trasportados mediante el backbone IP de forma transparente para el usuario. MPLS en conjunto con las disciplinas de ingeniería de tráfico proporciona una excelente infraestructura para este propósito.

Es importante mencionar que los pseudo cables son ideales para dar servicios de VPN en capa dos sobre un backbone de paquetes, emulando un servicio dedicado para cada cliente en un medio compartido y por lo tanto altamente eficiente. Los servicios de VPN en capa dos sobre una red de paquetes, en contraposición con las VPN en capa tres tienen las siguientes ventajas:

- Independencia del cliente para controlar su red en capa tres. Esto le permite crear sus propias reglas de seguridad y enrutamiento, así como preservar la privacidad respecto del proveedor que únicamente debe proporcionar el medio de transporte.
- Simplifica el mantenimiento del servicio para el proveedor, por no tener que controlar la red del cliente en capa tres, por lo tanto ya no es necesario almacenar tablas de enrutamiento por cada punto del cliente.
- Acota las responsabilidades del proveedor, ya que permite establecer claramente las fronteras del servicio sin necesidad de analizar protocolos de capas superiores.
- No existe la limitante de las VPNs en capa tres, que sólo soportan tráfico IP.

- Es posible dar servicios de altas capacidades en forma eficiente sobre una red convergente, lo cual no es posible con las tecnologías de capa dos convencionales como Frame Relay o ATM.

1.1. Descripción de los Pseudo Cables

El modelo de referencia de los pseudo cables se muestra en la Figura 1.4.

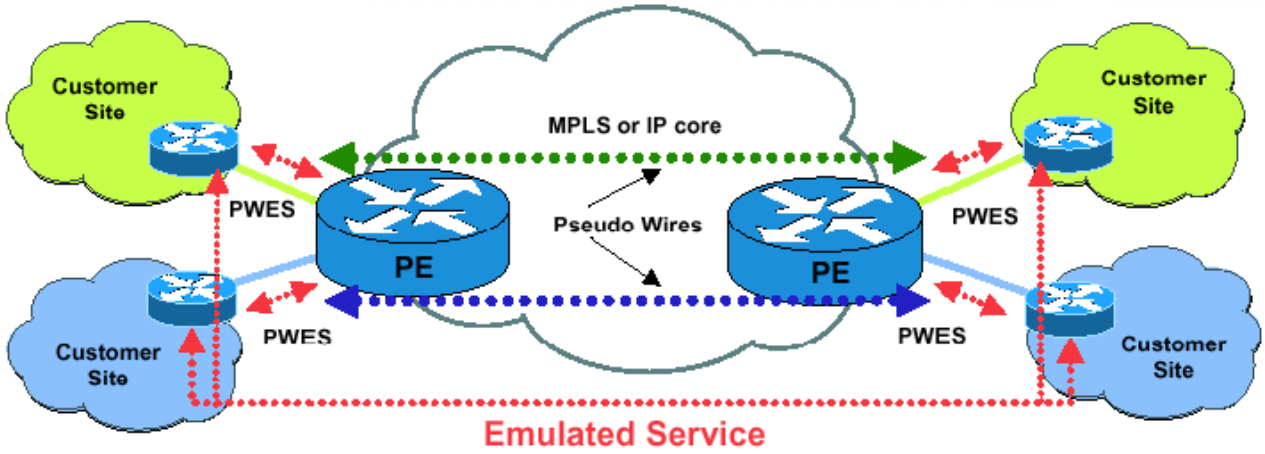

Figura 1.4 [http://www.apricot.net/apricot2005/slides/T5-3_3.pdf**]**

Un dispositivo PE (provider edge) administrado por el proveedor de servicios implementa la emulación del pseudo cable para un dispositivo CE (customer edge) administrado por el cliente.

El vínculo entre el PE y el CE (PWES pseudo wire end services) puede ser a nivel de Ethernet, PPP, Frame relay, ATM, etc.
El pseudo cable es un circuito virtual entre los PE's, los frames recibidos desde los CE son encapsulados dentro del pseudo cable y enviados al PE del otro extremo, una vez arribados allí son desencapsulados y enviados al CE destino.

Para establecer un pseudo cable entre dos PE se pueden utilizar tres métodos.

Configuración manual.
Protocolos de señalización.
Mecanismo de auto-descubrimiento.

A nivel de estandarización, la mayor actividad se da en el grupo de trabajo del IETF, Pseudowire Emulation Edge-to-Edge (PWE3). Este grupo se dedica al desarrollo de recomendaciones para la encapsulación y la emulación de servicios de los pseudo cables. Existen dos líneas en este sentido, la propuesta por el "draft-martini y la propuesta por el "draft-kompella".

El draft-martini se caracteriza por su simplicidad y describe cómo establecer un pseudo cable entre dos dispositivos PE, los métodos de encapsulación y define una extensión para LDP a fin de poder soportar el intercambio de información para establecer el pseudo cable.

El draft-kompella es más complejo y no lo describimos aquí ya que lo consideramos fuera del alcance del proyecto.

La RFC 3916 (Requirements for Pseudo-Wire Emulation Edge-to-Edge) describe los requerimientos básicos para implementar los pseudo cables.
Los requerimientos de la emulación incluyen encapsular PDUs especificas de la tecnología para la cual se está usando el pseudo cable en el punto de ingreso de éste, transportar esta PDU encapsulada hasta el otro extremo y finalmente desencapsularla en el punto de egreso. Todas las operaciones anteriores deben ser realizadas considerando las restricciones de temporización (delay, jitter, etc.), secuencia y cualquier otro atributo impuesto por la tecnología que se está emulando.

1.2. Ethernet sobre MPLS

En el caso de ethernet sobre MPLS, en el PE de ingreso se recibe un frame ethernet, al cual se le agregan las etiquetas de MPLS (se encapsula) y luego se transmite éste por la red MPLS. Hay dos modos de operación definidos [ref. 20] para un pseudo cable que emula ethernet:

Modo Raw (Transparente): En este modo, se pueden recibir en el PE de ingreso, frames con o sin VLAN TAG, pero el pseudo cable es transparente respecto del TAG, es decir que el frame es transmitido al otro extremo sin alterar el TAG y sin que éste se use como información para procesar dicho frame.

Modo Tagged: En este modo, cada frame que se recibe debe tener un TAG de VLAN (802.1Q) y el valor de este TAG se usa como información para procesar el frame en los extremos del pseudo cable.

El procedimiento a seguir ante el arribo de un frame al PE de ingreso es el siguiente:

1. Se elimina el preámbulo y FCS.
2. Se agrega al frame la palabra de control si es necesario (ésta es opcional y su uso se describe más adelante) y la etiqueta correspondiente al pseudo cable (etiqueta MPLS), ésta suele denominarse VC Label (etiqueta de circuito virtual).
3. Se debe encapsular el paquete resultante en un paquete MPLS, el PDU del pseudo cable es el frame ethernet recibido.

Cuando un paquete es recibido por el PE de egreso del pseudo cable, se hace lo siguiente:

1. El encapsulado del túnel y la etiqueta del pseudo cable se eliminan.
2. Se regenera el FCS.

3. Se transmite el frame en su formato original.

Ahora analicemos qué sucede cuando se recibe un frame que contiene un VLAN TAG en el PE de ingreso. Se distinguen los siguientes casos:

El TAG se utiliza como "*delimitador de servicio*". Esto significa que el proveedor del servicio emulado está usando el TAG de VLAN como información para procesar los paquetes en el pseudo cable.
El TAG no se utiliza como "delimitador de servicio". Esto significa que es un TAG que está siendo usando por el cliente del servicio emulado (no por el proveedor), por lo tanto éste no tiene ningún sentido ni aporta información alguna para procesar los paquetes en el pseudo cable.

Si se está operando en modo Tagged, cada frame transmitido en el pseudo cable debe tener un VLAN TAG del tipo "delimitador de servicio", y éste TAG se usa como información para procesar el paquete. En caso que el PE de ingreso reciba un frame sin TAG, éste debe agregar una "dummy" TAG (TAG de relleno).

Si se está operando tanto en modo Tagged como RAW los VLAN TAG que no son delimitadores de servicio deben ser trasmitidos en forma transparente por el pseudo cable.

El TAG que es delimitador de servicio tiene significado local para cada par PE-CE que interviene en el pseudo cable.

1.3. Palabra de Control en los pseudo cables

Si es necesario preservar el orden en el servicio que se está emulando, se hace mediante un mecanismo de números de secuencia implementados en la palabra de control.

La palabra de control es opcional, consta de 4 octetos que se describen en la siguiente figura.

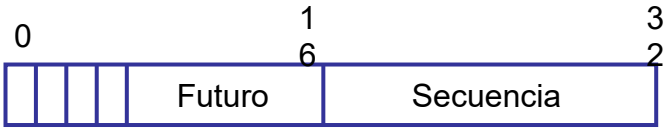

Figura 1.5 – Palabra de control

Los primeros 4 bits de las palabras de control se setean en cero para indicar que se está transmitiendo un paquete de datos.
El resto de los primeros 16 bits están reservados para uso futuro.
Los últimos 16 bits de la palabra de control se usan como número de secuencia. El número de secuencia se inicializa en valor uno y es circular, es decir que cuando se llega al valor máximo comienza en 1 nuevamente, el cero indica que no se está usando secuencias. En caso de estar usando secuencia el transmisor incrementa en uno el número de secuencia por cada paquete transmitido.

La siguiente figura da una idea general del formato del paquete dentro y fuera del pseudo cable.

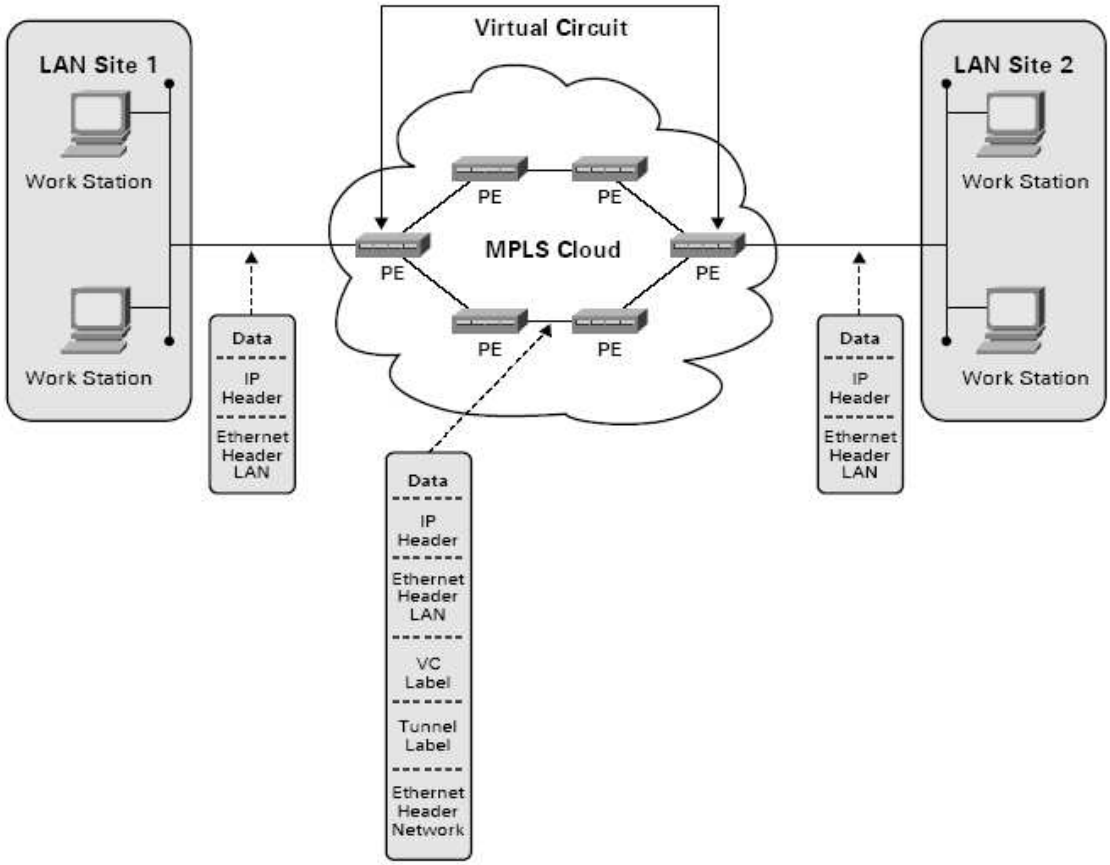

Figura 1.6 - Ethernet sobre MPLS [www.ipinfusion.com]

1.4. Opciones descartadas durante el proces de análisis.

Antes de definir agregar soporte para encapsular ethernet sobre MPLS se manejaron otras opciones. En particular trabajar en algunos de los "ToDo" propuestos por Leu y Casellas.

Las dos opciones consideradas fueron:
a) Implementar en el kernel de linux, MPLS sobre una capa dos Frame Relay (FR).
b) Interconectar dos redes Frame relay a través de una red MPLS mediante la creación de un pseudo cable.

Para poder trabajar con cualquiera de las dos opciones se accedió al siguiente hardware.

Dos PC con Linux, cada uno con una tarjeta de ethernet y una tarjeta Eicon con interfaz X.21 que permiten al PC trabajar con Frame relay.
Dos routers cisco 2500.

En la figuras 1.7 y 1.8 se muestran las maquetas que se utilizarían en cada caso (MPLS sobre Frame relay y frame relay a través de MPLS).

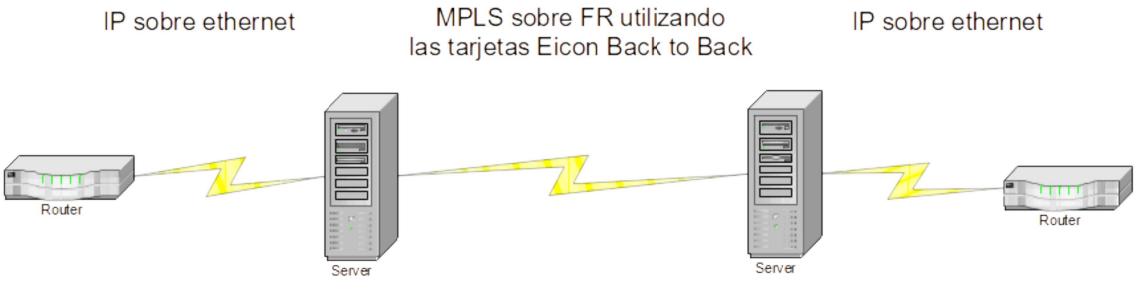

Figura 1.7

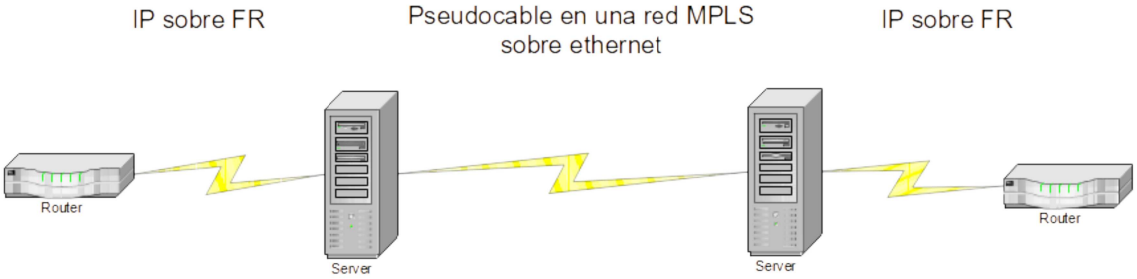

Figura 1.8

En la figura 1.7 se arma la maqueta con las tarjetas Eicon “back to back” a modo de formar la red MPLS sobre capa dos FR, en los extremos se encuentran los routers conectados a las tarjetas LAN de los PC trabajando sobre IP. En este caso el objetivo sería modificar el sistema MPLS para poder trabajar con MPLS sobre capa 2 Frame relay.

En la figura 1.8 las interfases seriales de los routers se conectan a las interfaces X.21 de las tarjetas Eicon de los PC para levantar Frame Relay y las interfases LAN de los PC back to back formando la red MPLS. En este caso se busca modificar el código para poder levantar un pseudo cable que una las redes Frame relay.

El primer paso fue hacer funcionar el hardware sin considerar MPLS, para esto se armó una maqueta como se muestra en la figura 1.9 donde los routers se conectaban contra el PC con la tarjeta Eicon y se levantaba Frame relay.

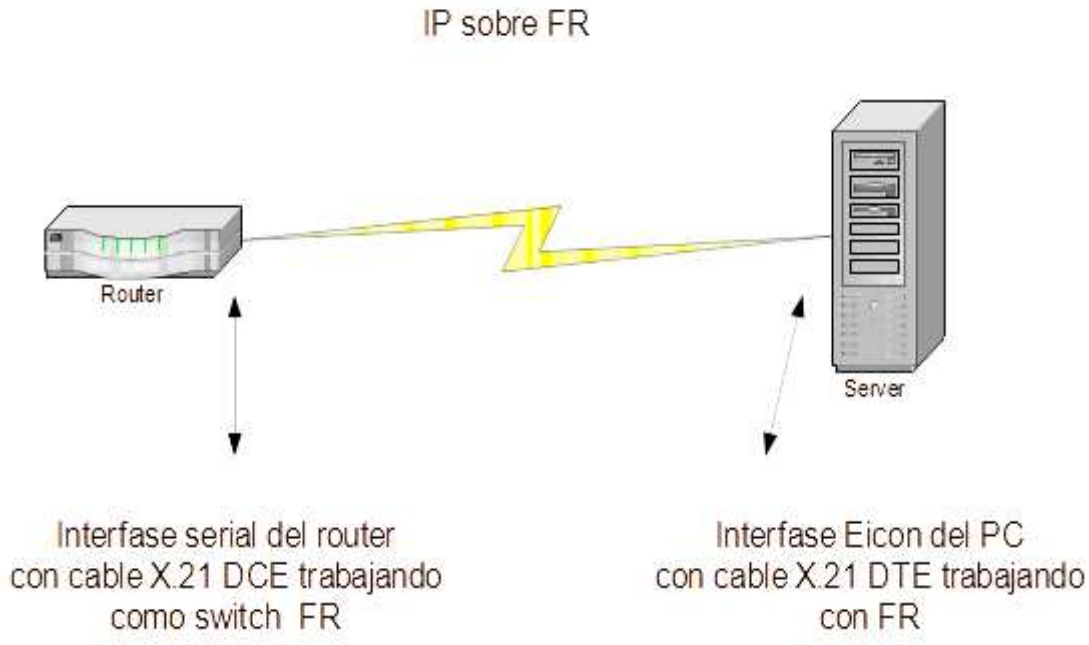

Figura 1.9

Una vez probado el hardware se comenzó a estudiar los pasos a seguir para implementar las nuevas funcionalidades. En este punto se vieron algunas incompatibilidades que nos hicieron desistir de estas opciones.
En particular las tarjetas Eicon estaban diseñadas para trabajar con el kernel 2.4 y el software que manejaban las mismas no era abierto, por otro lado la última versión del sistema MPLS estaba implementada sobre 2.6.

Las posibles opciones eran adaptar la tarjeta al kernel 2.6 ó trabajar con la versión MPLS para el kernel 2.4.

La primera fue descartada por entender que desviaría el curso del proyecto hacia otras áreas diferentes de las acordadas originalmente.
La segunda opción también fue descartada debido a que entendimos que las diferencias entre las versiones para (2.4) y la actual (2.6) eran muy importantes. Por otro lado la interacción con los directores del proyecto original de MPLS sobre Linux sería más fluida al trabajar en la misma versión en la que ellos se encontraban desarrollando.

2. Implementación del stack de TCP/IP en el Kernel de Linux.

Antes de ver los detalles de implementación comenzamos con una breve descripción general de cómo está organizado el kernel y algunos de los componentes principales. Para lograr esta descripción nos basamos en la documentación existente. Dado que la mayoría de ella se refiere al kernel 2.4 tuvimos que hacer un debug básico del código utilizando la función printk() para poder trazar el recorrido de los paquetes por el kernel 2.6.

2.1. Descripción general

Desde un punto de vista muy general podemos identificar tres grandes componentes del sistema operativo con el que estamos trabajando (fig. 2.1). En las siguientes secciones de este capítulo iremos describiendo cada uno de éstos, concentrándonos principalmente en el Kernel y en la implementación del stack de protocolos en linux.

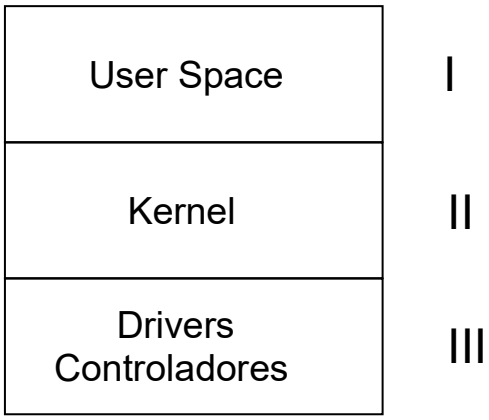

Figura 2.1

2.2. Componentes principales

I) User Space: Las aplicaciones de usuarios corren en este nivel, ej: Mail, browser, etc.

II) Dentro del Kernel de Linux los principales componentes que implementan las funcionalidades de networking son:
a. BSD Sockets (genéricos).
b. Inet Sockets (comunicaciones que utilizan TCP/IP), Unix Sockets (comunicaciones internas de la máquina), etc.
c. Código y/ó funciones que implementan las capas de red, ej. TCP/UDP, IP, ARP.

III) Los Drivers de los dispositivos de red tienen un rol fundamental en el Kernel. Los mismos son como "cajas negras" asociadas a cada dispositivo de hardware, en éstos se implementan una serie de funciones que son los puntos de acceso a las capas superiores, en particular permite conectarse con el stack TCP/IP. Los drivers de dispositivos ocultan completamente los detalles de cómo cada uno de los dispositivos trabaja y dejan disponibles los servicios que éstos brindan.

2.3. Diagrama de interrelación de componentes

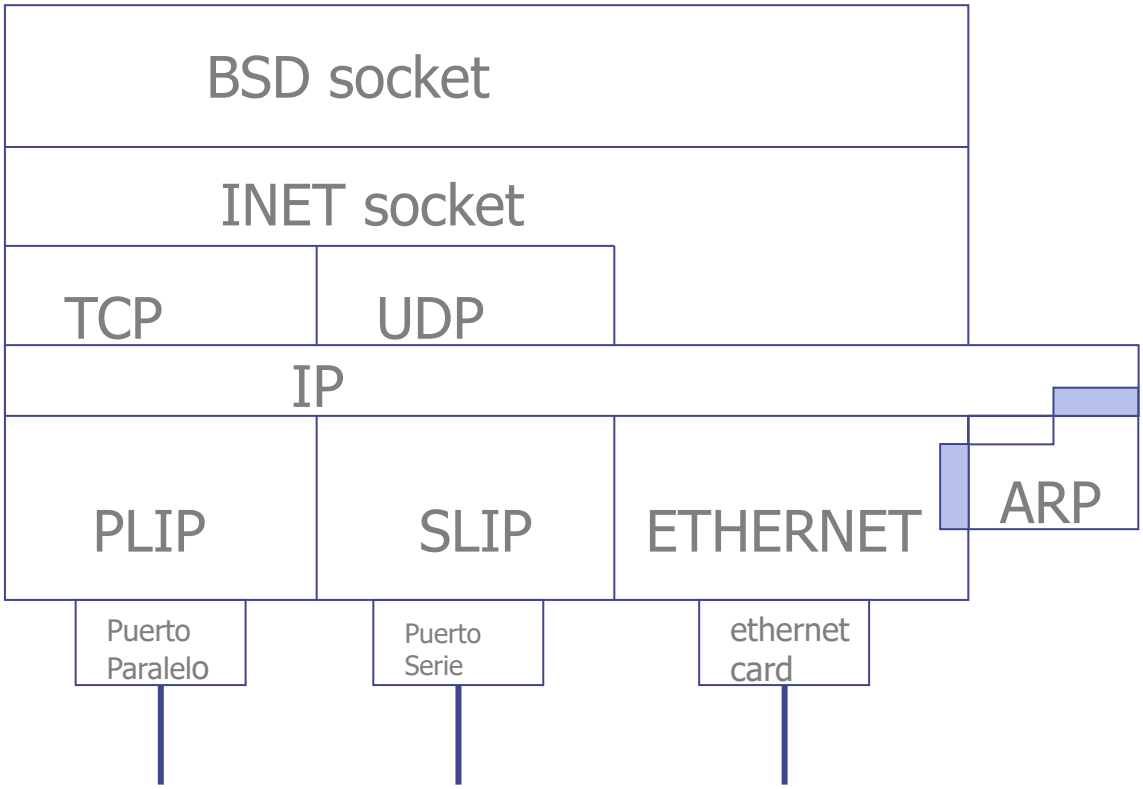

Figura 2.2

2.4. Sockets

El propósito de los BSD Sockets es presentar a las aplicaciones una interfase de comunicación genérica. Las aplicaciones envían y reciben datos escribiendo y leyendo Sockets respectivamente, sin preocuparse de cómo se implementa internamente esta comunicación.

A través de los sockets las aplicaciones pueden comunicarse con otras aplicaciones dentro de la misma máquina o fuera de ella. Dentro de los BSD socket (de propósito general) se incluyen entonces socket particulares. En el caso de las comunicaciones

entre procesos en distintas máquinas a través de TCP/IP, se utilizan los Inet Sockets, para procesos dentro de una misma máquina se utilizan los Unix Sockets.
Existen otros tipos de sockets, los mismos se pueden ver en el archivo socket.h en donde están definidos los “Protocol families”.

Los socket se clasifican también por el tipo de comunicación, los dos más importantes son:

Stream --> Orientados a conexión
Datagram --> No orientados a conexión

2.5. TCP/IP

La capa IP en el kernel de linux está implementada por el código y funciones contenidas en los siguientes archivos:

Ip_input.c -> Contiene todas las funciones necesarias para procesar un paquete a nivel de capa de red. Atiende los paquetes de entrada, es decir que llegan desde la tarjeta de red, luego de ser procesados por las funciones de capa 2.
Ip_output.c -> Ídem anterior, pero para los paquetes salientes.
Ip_forward.c -> Procesa los paquetes que deben ser enrutados por el propio host.

La función *netif_receive_skb()* (incluida en dev.c) es la que pasa el control del paquete a la capa de red invocando a la función *ip_recv()* (incluida en ip_input.c) para el caso de los paquetes IP. Luego de esto se hacen algunas verificaciones para ver si el paquete tiene como destino el host local ó debe ser reenviado. La función *ip_local_delivery()* es quién se encarga de pasar el control del paquete a las capas de transporte (tcp, udp, etc.) en caso de ser necesario (paquete para el propio host).

En caso que el destino del paquete sea otro host, significa que el PC está actuando como un router y por lo tanto debe reenviarlo. Este proceso comienza con una llamada a la función *ip_route_input()* que verifica las tablas de enrutamiento para obtener la ruta y reenviar el paquete. Se hace una descripción un poco más detallada de este procedimiento en el apéndice que describe estas tablas (FIB).

Cuando un paquete llega desde las capas superiores (transporte) para ser manejado por la capa de red, la primera función que maneja el paquete a nivel IP es *ip_queue_xmit()*, luego se lo pasa a *ip_output()*. Estas funciones hacen los últimos cambios y verificaciones del paquete a nivel de capa de red y lo entregan a la función *dev_queue_xmit()* quién encola el paquete para ser transmitido.

En la implementación de la capa de transporte se definen los protocolos TCP y UDP. TCP está implementado por el código de las funciones contenidas en los siguientes archivos:

tcp_input.c –> Esta parte del código trabaja con los paquetes entrantes desde la capa de red.
tcp_output.c –> Funciones que trabajan con los paquetes salientes hacia la capa de red.

tcp.c –> Código general de TCP, verificaciones y opciones.
tcp_ipv4.c –> Código específico de TCP para IP versión 4.
tcp_timer.c –> Manejo de los timers de TCP.
tcp.h –> Aquí se encuentran las definiciones de estructuras y variables de la implementación de TCP.

Dado que UDP es mucho más simple que TCP, todo el código y funciones de su implementación están contenidos en udp.c.

2.5.1. Drivers de red

Los drivers de los dispositivos de red además de controlar al dispositivo brindan una interfase con el stack TCP/IP permitiendo el intercambio de paquetes.

La estructura principal que utilizan los drivers es ***net_device structure*** (que se describe en el punto 2.6.8) y sirve para representar a un dispositivo.
Por cada dispositivo, el driver correspondiente inserta un link a la estructura que lo representa en una lista global de dispositivos de red.
El driver además define la función *Interrupt_Handler()* que será invocada cuando se reciba una interrupción de este dispositivo.

API vs. NAPI :

***A nivel de drivers*:**

Los dispositivos API generan una interrupción por cada paquete recibido, esto hace que en condiciones de alto tráfico el procesador se encuentre permanentemente ejecutando la función *Interrupt_Handler(),* lo que llevaría a un “Interrupt livelock” (los paquetes no pueden ser desencolados debido a que se destinan todos los recursos para encolarlos).

Los drivers que utilizan NAPI interrumpen al procesador cuando reciben el primer paquete, de esta forma el kernel se entera de que debe atender al dispositivo. De aquí en adelante el dispositivo procesa los nuevos paquetes sin interrumpir al procesador.
Luego que el procesador atiende al dispositivo y procesa todos los paquetes pendientes el driver volverá a habilitar las interrupciones.
En caso de llenarse el buffer de recepción de la tarjeta de red, los paquetes se descartan “silenciosamente” o sea sin generar ninguna interrupción.

***A nivel del kernel*:**

En el caso API, la función invocada por el Interrupt Handler es *netif_rx(),* esta función encola el paquete recibido en la backlog_queue de la CPU interrumpida (en realidad coloca un puntero al paquete en la cola) y se agenda un softirq que es responsable del procesamiento posterior del paquete.
Cuando se atiende la softirq, se ejecuta la función *net_rx_action()* que “interroga” todos los paquetes en la backlog_queue. La función que “interroga” la backlog_queue es

process_backlog(). Para cada paquete se llama a la función correspondiente que lo procesará (por Ej. ip_rcv()).

En el caso NAPI, la función invocada por el Interrupt Handler es *netif_rx_schedule(),* esta función en lugar de encolar el paquete en la backlog_queue, coloca una referencia al dispositivo en una cola de la CPU interrumpida (poll-list). Como en el caso anterior se agenda una softirq, pero en este caso las interrupciones debidas a la recepción de paquetes en este dispositivo están deshabilitadas.
Cuando se atiende la softirq, se ejecuta la función net_rx_action() que "interroga" todos los dispositivos presentes en la poll_list, obteniendo los paquetes directamente de sus rx_ring_buffer. La función que "interroga" al dispositivo es implementada por el driver de la tarjeta. Para cada paquete se llama a la función correspondiente que lo procesará (por ej: ip_rcv()).

Para mantener la compatibilidad del nuevo kernel (NAPI) y los drivers se seguirá implementando la backlog_queue y se considerará como un dispositivo más. La función netif_rx() se reescribió para poder encolar a la backlog_queue en la poll_list de la CPU.

2.6. Estructuras de datos de red en el Kernel

A continuación se hace una breve descripción de las estructuras con las que trabajamos al estudiar la implementación del stack TCP/IP en el Kernel de Linux.

2.6.1. Sockets

Está definida en net.h.

Los BSD Sockets se representan con la estructura socket.

Al llamar a la función socket() se crea e inicializa la estructura de datos socket. Dentro de la estructura tenemos entre otros: type, ops y sk.

type --> Indica el tipo de comunicación de socket (ej SOCK_STREAM)
ops --> Apunta a una estructura de datos proto_ops que contiene todas las operaciones que puede efectuar la familia de protocolos a la cual pertenece el socket según el valor del protocol family.
sk --> Apunta a una estructura de tipo sock que se trata en la próxima sección del capítulo.

Nota: El valor del campo ops en esta estructura se obtiene de un vector llamado pops que contiene los punteros a las instancias de tipo proto_ops correspondientes a cada familia de protocolos (una instancia por familia de protocolos) que tiene definida el kernel. En estas instancias de tipo proto_ops se encuentra información o funciones específicas para cada familia de protocolos. Para encontrar el puntero correspondiente a una familia particular se consulta la posición del vector pops igual al valor de la familia

(Address family o AF). Para el caso de PF_INET, AF_INET vale 2 (por definición) y el puntero se encuentra en la posición 2 del vector pops.

```
struct socket {
      socket_state            state;
      unsigned long           flags;
      struct proto_ops  *ops;
      struct fasync_struct    *fasync_list;
      struct file       *file;
      struct sock       *sk;
      wait_queue_head_t wait;
      short             type;
      unsigned char           passcred;
};
```

2.6.2. Sock

La información específica del protocolo del BSD Socket se representa con la estructura sock.
Ej: para un INET socket esta estructura almacenara toda la información específica de los protocolos TCP/IP y UDP/IP

La estructura completa se encuentra en el Apéndice A.

2.6.3. Sk_Buff

Definida en skbuff.h

Cada paquete de datos se representa con la estructura sk_buff.

De forma simplificada se puede decir que la parte de red del kernel usa principalmente dos estructuras de datos: una para mantener el estado de la conexión (sock) y otra para mantener el estado de los paquetes entrantes y salientes (sk_buff).

El sk_buff es una estructura que representa a cada paquete y es común a todas las capas. La estructura contiene punteros a toda la información referente a un paquete (socket, dispositivo, ruta, ubicación de datos, etc.). Para paquetes salientes los protocolos de transporte son los responsables de crear la estructura, para paquetes entrantes son los drivers. Una vez creada la estructura cada capa llena la información que le corresponde a medida que procesa el paquete, esto implica que el paquete es copiado solo una vez desde la tarjeta de red a la memoria del Kernel.

La mayoría (y los más relevantes) de los elementos tienen nombres que describen por sí solos su significado, pero es importante aclarar que los dos primeros miembros ***next, *prev** son iguales que en la estructura ***sk_buff_head.***

La estructura completa se encuentra en el apéndice A.

2.6.4. Sk_Buff_Head

Definida en skbuff.h

Esta estructura se usa para definir los extremos en una lista circular, definen un orden en la lista y se usan entre otras cosas por razones de mantenimiento.

La estructura completa se encuentra en el apéndice A.

2.6.5. List_Head

Esta estructura se utiliza como un "almacén" de listas de tipo ***list_head***, implementada como una "doubled linked list" en que cada miembro de la lista tiene un puntero hacia el siguiente miembro y otro al anterior. De modo que lo que se almacena aquí son miembros de tipo ***list_head*** los cuales pertenecen a otras estructuras que quedan así referenciadas.
A pesar de lo que indica el nombre, la lista no tiene ni cabecera ni final.
Las características Principales de este tipo de listas son;
1.- Agregar ó quitar un miembro requiere el mismo tiempo de CPU independientemente de la cantidad de miembros de la lista.
2.- El espacio en memoria no precisa ser contiguo, por lo que no requiere de reserva previa.
3.- Las estructuras que se agregan como miembros de ***list_head*** no deben ser necesariamente del mismo tipo.
Se dispone además de un "set" de funciones que permiten operar sobre las listas, entre las más importantes tenemos funciones para agregar ó quitar miembros de la lista, recorrerla transversalmente, etc.

La estructura completa se encuentra en el apéndice A

2.6.6. Packet_Type

Definida en netdevice.h

```
struct packet_type {
        unsigned short          type;   /* This is really htons(ether_type).    */
        struct net_device       *dev;   /* NULL is wildcarded here              */
        int                     (*func) (struct sk_buff *, struct net_device *,
                                         struct packet_type *);
        void                    *af_packet_priv;
        struct list_head        list;
};
```

Esta estructura se utiliza para "registrar" los distintos tipos de paquetes o protocolos que serán válidos y la forma en que serán procesados, de forma que para cada tipo de protocolo soportado por el kernel debe crearse una de estas estructuras y registrarla.

Para registrar los distintos ***packet_type*** el kernel inicializa dos tipos de "linked list"; una ***struct list_head ptype_all*** para procesar los paquetes independientemente del protocolo y un array de 16 "linked list" ***struct list_head ptype_base[16]*** para protocolos específicos.

> Nota: Para el caso del protocolo MPLS, es necesario registrar un packet type, se verá con más detalle en el capítulo 6.

Cada ***packet_type*** se registra ó almacena en una de estas "*linked list*"**,** invocando para ello a la función ***dev_add_pack(struct packet_type *pt),*** que en función del valor type y mediante un algoritmo dado, inserta nuestro ***packet_type*** en una de las 16 ***ptype_base*** ó en la ***ptype_all***

Resumiendo, para inicializar una de estas estructuras tenemos que llenar los siguientes campos;

1°.- Una variable "***type***" , el cual será el "protocol ID" con el que identifico este protocolo (por ej: ***type= htons(ETH_P_IP, ETH_P_ARP, etc),*** estos valores están declarados en **packet.h**, que es donde se declaran los distintos tipos de paquetes)**.**
2°.- Un puntero al dispositivo por el que van a salir y entrar estos paquetes. Si asigno ***dev=NULL*** no se especifica ninguno, por lo que cualquier dispositivo será válido.
3°.- Un puntero a la función encargada de recibir este tipo de paquetes, por ej: func=ip_rcv(), mpls_rcv (), etc.
4°.- Una variable "***list***" de tipo ***list_head*** sin inicializar. Al registrarse el ***packet_type*** quedará apuntando a la lista donde se almacena nuestro ***packet_type***.

La estructura completa se encuentra en el apéndice A.

2.6.7. Sofnet_Data

Definida en netdevice.h

Esta estructura almacena punteros a colas de transmisión y recepción de los dispositivos e interfaces (devices ó backlogs) que deben ser atendidas por la CPU más adelante. Para eso tenemos los siguientes campos:

- ***input_pkt_queue:*** una cola de paquetes. Aquí se almacena los paquetes (sk_buff) arribados desde los dispositivos en forma secuencial y son los que están prontos para ser pasados a las capas superiores para su procesamiento.
- **output_queue:** esta cola contiene una lista de dispositivos que tienen paquetes prontos para ser transmitidos.
- **Poll_list:** Una cola conteniendo los dispositivos pendientes para ser atendidos por el scheduler.

Adicionalmente tenemos:
1.-Variable ***throttle*** se usa como "flag" para indicar congestión y por consiguiente descartar todos los paquetes que arriben hasta que se cambie la condición, modificando para ello el valor de este flag.

2.- Una estructura ***backlog_dev*** que indica el dispositivo que se está procesando.
3.- Variable ***cng_level***, es el valor que indica el nivel de congestión actual (no congestión, low, moderate)

La estructura completa se encuentra en el apéndice A.

2.6.8. Net_Device

Definida en netdevice.h

Esta estructura se usa para representar dispositivos (ó interfaces en forma genérica) en Linux, aquí se guarda información y punteros a las distintas funciones que operan sobre los mismos. Por cada dispositivo en el sistema existe una instancia de este tipo. Algunos campos importantes son:

- Name: el nombre del dispositivo, por ejemplo eth0.
- IRQ: La interrupción asociada al dispositivo.
- Init: Puntero a la función de inicialización del dispositivo.
- Poll_list: un campo de tipo list_head que se usa cuando el dispositivo se quiere incluir en la poll_list de la CPU.
- Funciones estándar: son las funciones que el dispositivo tiene disponible para la interacción con el kernel. Generalmente definidas en el driver. Por ejemplo la función "open" inicializa las estructuras de datos internas del dispositivo antes de aceptar paquetes. A continuación se muestan algunos campos importantes de esta estructura con su descripcion:

```
struct  net_device
{
        /* Nombre del dispositivo */
        char                    name[IFNAMESIZ];
        unsigned int            irq;
        /* Función de inicialización del dispositivo*/
        int                     (*init)(struct net_device *dev);
        /*Se utiliza para agregar el dispositivo a la poll list    */
        struct list_head        poll_list ;
        int                     quota;
        void                    *priv;
        /* La estructura declara las funciones que pueden actuar sobre los
        dispositivos*/
        int                     (*open) (struct net_device *dev)
        int                     (*poll) (struct net_device *dev, int quota)
}
```

La estructura completa se encuentra en el apéndice A

3. Recorrido de un paquete IP en el Kernel

3.1. Introducción

Nos concentraremos fundamentalmente en cómo Linux implementa la capa física, capa 2 y algo de capa 3 debido a que esto es lo que entendemos que concierne al proyecto.

Comenzaremos por una revisión general a los efectos de explicar los principios fundamentales con los que funciona el Kernel de Linux y luego analizaremos en detalle el código de algunas funciones (las más importantes) que implementan el stack de protocolos en el Kernel de Linux y por lo tanto procesan cada paquete entrante o saliente. En particular elegimos la recepción de paquetes para estudiar el proceso en detalle.

Durante la transmisión las interfases reciben paquetes de la capa de red, y luego los envían al medio físico, asimismo en la recepción se reciben paquetes de la NIC (Network Interface Card) que se pasan a las capas superiores entregándolos a las funciones correspondientes a los distintos protocolos.

3.2. Recepción

Una descripción de alto nivel es que la recepción de un paquete hace que el dispositivo de red por el que llegan los paquetes genere una interrupción de hardware. En respuesta a esta interrupción el manejador de interrupciones copia el paquete del dispositivo a la memoria y luego lo procesa. El procesamiento del paquete no se debe hacer en el contexto de la interrupción de hardware para evitar tener el procesador mucho tiempo ocupado atendiendo a una interrupción, dado que esto puede hacer que se pierdan otras interrupciones. La implementación de Linux es procesar el paquete en una interrupción de software “softirq” mientras las interrupciones de hardware están habilitadas.

Viendo el proceso más en detalle, cuando un paquete es recibido por el dispositivo éste se almacena en la memoria mediante el manejador de DMA en una cola circular. El manejador usa una lista de descriptores de paquetes que contienen apuntadores a las direcciones de memoria donde los paquetes se pueden almacenar. Cada área de memoria debe ser suficientemente grande para almacenar un paquete con el máximo tamaño de paquete (MTU) que la interfase puede recibir. La interrupción de hardware se genera recién cuando el paquete está almacenado en memoria y listo para ser procesado.

Al generarse la interrupción en la CPU, ésta salta al código ISR (Interrupt Service Routine). El proceso siguiente depende de si la sub-capa de red trabaja con el modelo antiguo API (Application Programing Interfase) ó con el modelo actual NAPI (Network Application Programing Interfase).

3.2.1. Modelo API

En este modelo, a la llegada de una interrupción el manejador de interrupciones llama a la función del Kernel netif_rx(), la cual almacena un puntero al descriptor del paquete recibido en la cola Backlog de la CPU y solicita una softirq (se prende un flag) la cual es responsable del manejo subsiguiente del paquete. La CPU puede recibir una interrupción por paquete o por grupo de paquetes dependiendo de la configuración del dispositivo. Si el Backlog se llena, se entra en un estado de "throttle" en el cual se espera a vaciar la cola antes de volver al estado normal y aceptar paquetes. La siguiente figura representa la topología del modelo API.

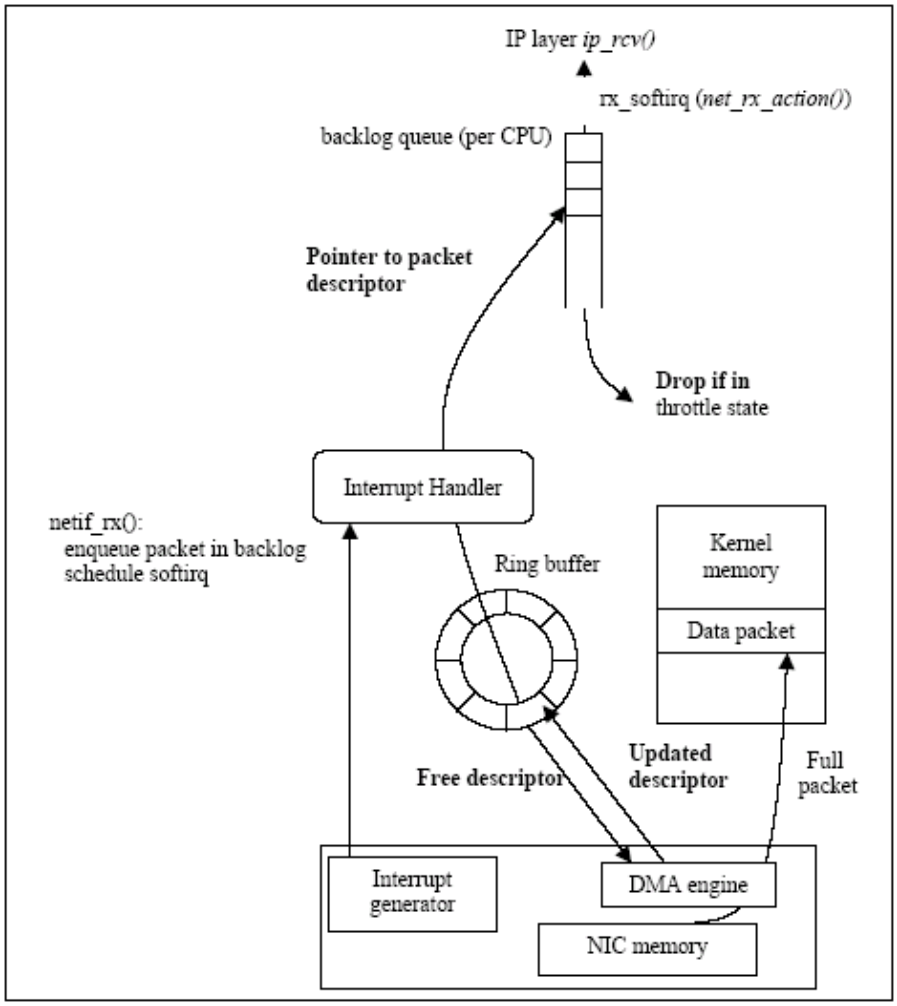

Figura 3.10 – Modelo API [ref 4]

3.2.2. Modelo NAPI

En este caso cuando la CPU recibe una interrupción del NIC debido a la recepción de un paquete, el manejador de interrupciones llama a la función netif_rx_scheduler() y ésta en lugar de utilizar una cola Backlog utiliza una cola llamada poll_list que almacena una referencia al dispositivo. En este momento es solicitada una softirq como en el caso anterior, a partir de este punto no se vuelven a habilitar las interrupciones de hardware de este dispositivo hasta vaciar la cola de paquetes, pero si están habilitadas las interrupciones de harware para otros dispositivos.
Nota: Si durante la sofirq se alcanza el numero máximo de paquetes que pueden ser atendidos se vuelve a poner el dispositivo en la poll_list y se agenda una nueva softirq de baja prioridad llamada ksoftirq, las interrupciones de hardware de este dispositivo permanecen deshabilitadas en este proceso.
En realidad lo que se guarda en el poll_list es un apuntador al dispositivo que generó la interrupción y se genera una softirq, de esta manera queda claro que el dispositivo tiene paquetes pendientes para ser procesados.
Cuando se ejecuta la softirq todos los dispositivos que están registrados en la poll_list son interrogados y se procesan los paquetes pendientes de cada dispositivo.

Para asegurar compatibilidad hacia atrás con los dispositivos que no soportan el modelo NAPI el Backlog aún esta implementado pero la función netif_rx se reescribió de manera que el backlog representa un dispositivo (backlog_dev) que puede ser agregado en la poll_list de la CPU. En la siguiente figura se muestra un esquema lógico del modelo NAPI.

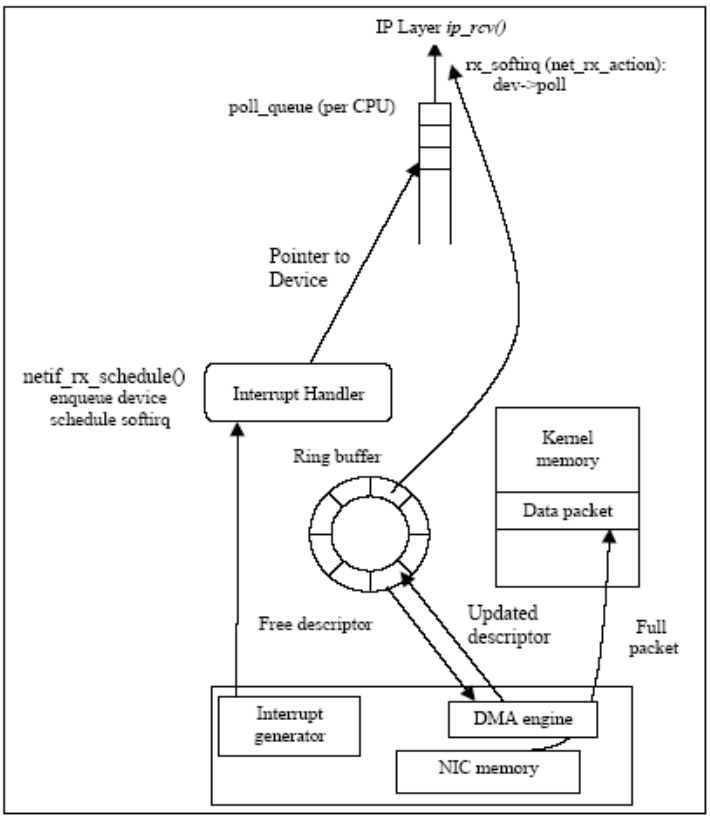

Figura 3.2 – Modelo NAPI [ref 4]

A continuación se muestra un esquema de las funciones que intervienen en la recepción de un paquete en cada uno de los modelos (el camino en rojo representa el modelo NAPI):

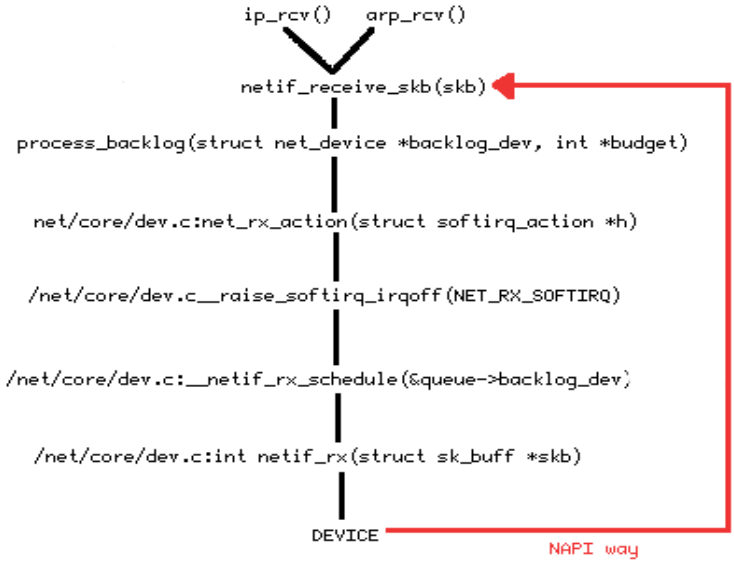

Figura 3.3 – Principales funciones del stack de protocolos en la recepción [

3.2.3. Detalle de las principales funciones

netif_rx()

Esta función es llamada solamente por los drivers de dispositivos API (los NAPI llaman directamente a netif_receive_skb()).

El procesamiento del paquete depende del estado de la cola y del throttle flag (éste indica si estamos en estado de congestión).

1. Si la cola está llena se descarta el paquete y se setea el estado de congestión (en caso de no estar en este estado).
2. Si la cola no está llena pero hay paquetes, se chequea si hay congestión, de ser así se descarta el paquete. Si no existe congestión el paquete se agrega a la input packet queue.
3. Si la cola está vacía, se revisa si hay congestión para pasar el throttle a cero debido a que se vació la cola. Luego se llama a la función netif_rx_schedule() para ingresar el backlog del dispositivo a la poll_list y finalmente se agrega el paquete a la input queue (backlog).

Resumen del código:

```
if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {   (1) Chequea si la cola no
                                                                está llena
     if (queue->input_pkt_queue.qlen) {  (2) Chequea si no está vacía la cola
          if (queue->throttle)                    (3) Chequea si está en throttle
               goto drop;                         (3) Si está en throttle dropea
                  encola;
                  return;
     if (queue->throttle)                  (2) Si la cola esta vacía, revisa el thorttle,
        queue->throttle = 0;                   si esta en throttle sale de ese estado
     netif_rx_schedule(&queue->backlog_dev); Pone la bocklog del dispositivo en la
                                              poll list. Este es el caso de un kernel
                                              con NAPI y un driver que no lo
                                              soporta.
      encola;
      return;}
if (!queue->throttle)  (1) Si no está en throttle pasa a ese estado, a este punto solo
                           llegamos  si la condición en el primer IF es falsa.
   queue->throttle = 1;
dropea;
return;}
```

netif_rx_schedule()

Esta función agrega una interfase (o dispositivo) al final de la poll_list de la CPU. En nuestro caso la interfase es una backlog_queue. Luego de esto genera una softirq. Estrictamente hablando la poll_list (de la CPU) está representada por una estructura list_head y "agregar" un dispositivo a la poll_list (de la cpu) es agregar la lista poll_list (del dispositivo) que es del tipo list_head en la poll_list de la CPU.

Para hacerlo se llama a la función __netif_rx_schedule() y ésta a la función list_add_tail que inserta una nueva entrada al final de la lista (o sea antes de la cabecera de la lista)

list_add_tail(struct list_head *new, struct list_head *head)
Inserta la lista new al final de la lista que es encabezada por la lista head

En nuestro caso la función es invocada de la siguiente forma:
list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);

Como dev es el dispositivo con el que fue invocada la función (netif_rx_schedule(&queue->backlog_dev)), entonces &dev->poll_list es la poll_list del dispositivo backlog_dev. O sea que "agrega" al final de la poll_list de la CPU el dispositivo backlog_dev.

net_rx_action()

Introducción:

Cuando un Kernel NAPI trabaja con una tarjeta API, el backlog asociado a la CPU debe representarse como un dispositivo de red para poder agregarse a la poll_list.
Para esto cada CPU tendrá asociada un estructura softnet_data y el campo reservado para el dispositivo (del tipo net_device) que vamos a utilizar es el de la variable backlog_dev.
También se deberá inicializar "el dispositivo", en particular el método de "interrogación"(polling) que será process_backlog (queue->backlog_dev.poll = process_backlog). Esta inicialización se hace en la función net_dev_init(void) contenida en dev.c. La función se ejecuta durante el "booteo" del sistema.

Descripción de la función:

Esta función contiene el código que atiende a la softirq generada (NET_RX_SOFTIRQ). Cuando es llamada, recorre todos los dispositivos que están en la poll_list (entre ellos el backlog si existe) y llama a la función que se encarga de "interrogar" cada uno de los dispositivos (dev->poll).

Por cada invocación a la funcion net_rx_action() a lo sumo se procesan budget=netdev_max_backlog paquetes (300 por defecto), esto implica que en cada atención a una softirq se atienden a lo sumo 300 paquetes.
Quota es un campo dentro de net_device que indica el número máximo de paquetes que puede pasar el driver al kernel en cada llamada.
En definitiva en cada llamada a process_backlog() se procesara como máximo, el mínimo valor entre quota (del dispositivo) y budget que a su vez se va modificando cada vez que se atiende a un dispositivo.
Para recorrer toda la poll_list se utiliza la función list_empty en la condición de un while. A medida que se recorre la lista se extrae el dispositivo correspondiente a cada entrada, para esto utiliza la función list_entry():

list_entry (ptr, type, member)

Esta función devuelve la estructura del tipo "type" que contiene una variable "member" (del tipo list_head) igual a ptr.

En nuestro caso la función es invocada de la siguiente forma:
dev = list_entry (queue->poll_list.next, struc net_device, poll_list)
Por lo tanto la función devuelve el dispositivo (de tipo net_device) que posee una variable poll_list igual a queue->poll_list.next

Luego de obtener el dispositivo invoca a la función correspondiente que lo "interroga". Esta invocación se hace dentro de un if que controla la cantidad máxima de paquetes que puede procesar, la "interrogación" del dispositivo se hace dentro de la condición del if y sólo se entra al if en el caso que la cola quede vacía o se hayan procesado "budget" paquetes.

La invocación es dev->poll(dev, &budget).
Obs: cuando dev es backlog_dev dev->poll es process_backlog().
La función process_backlog() devuelve 0 si se pudieron procesar todos los paquetes en el backlog y además modifica "budget=budget-paquetes procesados" y resta en "dev->quota" los paquetes procesados. En otros casos devuelve -1.

En caso de procesarse el máximo de paquetes y no haber vaciado la cola del dispositivo éste se quita de la cola y se vuelve a encolar (al final).

netif_receive_skb()

Esta función es llamada (en nuestro caso) desde process_backlog y es la encargada de llamar a la función correspondiente según el tipo de paquete (o protocolo) recibido.

El tipo de paquetes (o protocolo) "type" y la función que se invoca para recibirlo "(*func)" se definen mediante una estructura packet_type que posee una variable dev que puede utilizarse para seleccionar el dispositivo por el que se reciben los paquetes del protocolo, tal como lo vimos en 2.6.6. En caso de ser dev=NULL se reciben paquetes de este protocolo por cualquier interfase.

El kernel posee dos listas posibles para agregar las listas de cada tipo de paquetes.
La estructura packet_type posee una variable list de tipo list_head para poder agregarse a cualquiera de las siguientes listas:

static struct list_head ptype_all;
static struct list_head ptype_base[16];

En la primera lista (ptype_all) se encuentran los protocolos definidos que reciben todos los paquetes (sin importar el protocolo que trae el paquete). Estos protocolos tienen almacenado en type el valor ETH_P_ALL.
En la segunda lista se almacenan los protocolos que tratan específicamente los paquetes con determinado protocolo. Los posibles tipos están definidos en el archivo if_ether.h por ejemplo ETH_P_IP.

Resumen del código:

Básicamente el código recorre las dos listas, primero la que procesa los paquetes independientemente del protocolo y luego la lista que procesa los paquetes con protocolos específicos. Para cada protocolo encontrado en las listas se llama a la función asociada al mismo.

Para recorrer las listas (ptype_all y ptype_base[]) se utiliza la macro *list_for_each_entry_rcu*.

Ejemplo:

list_for_each_entry_rcu(ptype, &ptype_all, list)
En este caso recorre toda la lista ptype_all y para cada lista (en la lista) encuentra la estructura packet_type que posee esta lista en la variable list. Esta estructura packet_type la guarda en ptype (dado que estamos recorriendo ptype_all esta estructura corresponde a protocolos que procesan todos los paquetes).

Recorrida de la lista ptype_all:

```
list_for_each_entry_rcu(ptype, &ptype_all, list) {

        /* Aqui comienza el codigo "sentencia" del for */

        if (!ptype->dev || ptype->dev == skb->dev) {   /* Si acepta paquetes de
cualquier dispositivo o si el dispositivo por el que acepta paquetes es por el que entro  el
paquete  */
          if (pt_prev)  /* esta condición se cumple siempre salvo la 1era vez
                ret = deliver_skb(skb, pt_prev);      /* llama a la función
que procesa al pachetto   */
            pt_prev = ptype;
        }
}
```

Recorrida de la lista ptype_base[]

La lógica es similar salvo que ptype_base posee 16 listas y se debe encontrar la lista adecuada para buscar luego el protocolo.

Las listas se colocan en una hash table con 16 hash list. Los 4 bits menos significativos del protocolo se utilizan como hash key.
Así la llamada a list_for_each_entry_rcu en este caso será así:

```
list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list)
```

donde type = skb->protocol

3.3. Transmisión

Para cada paquete que debe ser transmitido desde la capa IP se llama al procedimiento dev_queue_xmit(), el cual encola el paquete en la cola qdisc asociada al dispositivo de salida asignado por los algoritmos de enrutamiento. Luego si el dispositivo no está parado debido a que el tx_ring está lleno o hay una falla, todos los paquetes son procesados por la función qdisc_restart(), la cual llama al método hard_start_xmit() implementado en el driver del dispositivo. Éste último pone el descriptor del paquete (el cual contiene la dirección de memoria del kernel donde está el paquete de datos) en el tx_ring y le avisa al dispositivo que hay un paquete listo para ser transmitido.

Una vez que el paquete fue transmitido completamente por el hardware, éste le comunica a la CPU que el paquete se transmitió generando una interrupción, la CPU usa esta información para poner el paquete en la cola completation_queue y solicita una softirq la cual se encargará de liberar recursos utilizados para la estructura skbuff y el paquete de datos si estamos seguros que ya no es necesario.

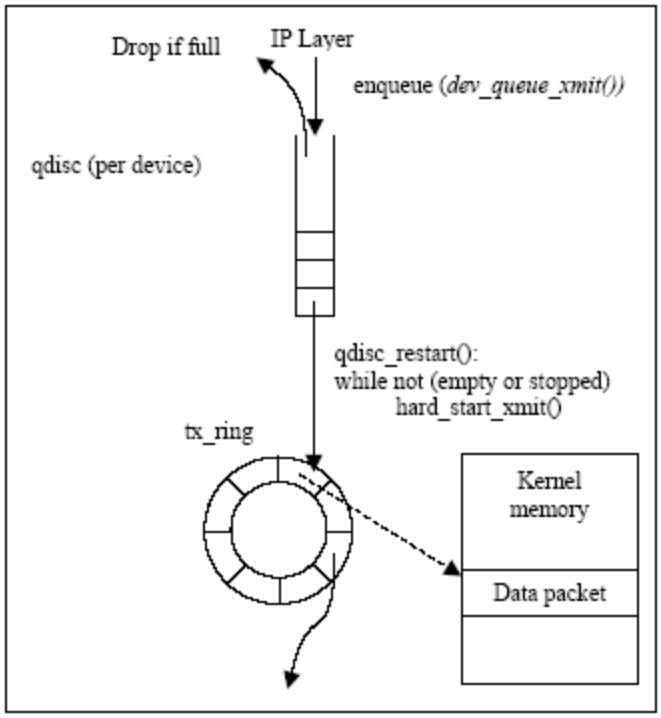

Figura 3.4 – Transmisión [ref 4]

3.4. ARP y tablas de enrutamiento en Linux

Hasta ahora analizamos qué es lo que sucede en el proceso de transmisión y recepción de un paquete desde el nivel físico hasta la capa de red. Hemos procurado intencionalmente no introducir detalles sobre enrutamiento y tareas que se realizan en la capa de red por considerar que este tema escapa a los conocimientos necesarios para elaborar el proyecto, no obstante en el apéndice D incluimos una descripción del protocolo ARP y del enrutamiento en Linux.

4. Fundamentos de MPLS

A continuación se hará una introducción a los principios básicos de funcionamiento de MPLS y a la estructura de las etiquetas MPLS. Los conceptos aquí incluidos son parte de la etapa de adquisición de conocimientos previos del proyecto, esenciales para la comprensión del sistema MPLS implementado en Linux, así como las modificaciones hechas sobre éste. Si el lector está familiarizado con MPLS puede omitir la lectura de este capítulo.

4.1. Estándares

El funcionamiento de MPLS está estandarizado por el IETF en sus recomendaciones RFCs. Las RFCs que describen la operación básica son las siguientes:

- Los principios básicos de operación y arquitectura de MPLS se definen en la RFC 3031.
- Los principios de codificación de etiquetas MPLS en links PPP o LAN se definen en la RFC3032 así como las reglas y procedimientos para procesar los distintos campos de las etiquetas MPLS.
- La RFC 3036 define un protocolo por el cual los dispositivos MPLS pueden informar a otros dispositivos MPLS las etiquetas que se están usando y el significado de éstas.

Adicionalmente hay otras RFCs y draft's que tratan temas avanzados como ingeniería de tráfico (RFC 2702, RFC3209), establecimiento de caminos (LSP) usando el protocolo LDP (RFC 3212), etc.

4.2. Introducción

Comenzaremos intentando contestar la pregunta "¿Qué es MPLS? ".

MPLS es un método de alta performance de "forwardear" paquetes en una red. MPLS permite que un router en el ingreso de la "nube" agregue una etiqueta al paquete y en los hops sucesivos el forwarding del paquete se hace basado en esta simple etiqueta sin la necesidad de analizar el encabezado de la capa de red del paquete.

MPLS integra las ventajas de performance de manejar el tráfico en capa 2 con las ventajas de escalabilidad y flexibilidad del enrutamiento en capa 3.

El concepto básico y fundamental de MPLS es incluir una etiqueta en cada paquete. Esta etiqueta es corta y de tamaño fijo. Luego, entidades/dispositivos que manejan paquetes MPLS hace lookups basados en estas etiquetas simples para determinar a donde debe ser forwardeado el paquete. Esta etiqueta sencilla se define basada en múltiples criterios y contiene resumida entre otras cosas la siguiente información asociada a un paquete convencional:

a) Destino.
b) Precedencia – CoS.

c) Miembro de una VPN.
d) Información de QoS de RSVP.
e) La ruta del paquete de acuerdo a la ingeniería de tráfico aplicada.

De esta forma el análisis completo de toda esta información se realiza solo una vez a la entrada del paquete en la red MPLS, luego a lo largo de toda la red MPLS se hace forwarding del paquete basándose en la etiqueta. Esto es un resultado clave ya que permite tomar decisiones de forwarding para un paquete basándose en múltiples criterios sin afectar la performance de todos los equipos de la red, la decisión se toma solamente en el nodo de ingreso al seleccionar la etiqueta.

4.3. Definiciones

La tarea de realizar el reenvío de un paquete puede ser pensada o dividida en dos funciones:

1) La primera clasifica el conjunto total de paquetes posibles en un conjunto de FECs (Forwarding equivalence Classes)
2) La segunda mapea cada FEC en un “next hop” o próximo salto. Desde el punto de vista de las reglas de reenvío, distintos paquetes que son mapeados en la misma FEC son indistinguibles.

En las redes MPLS (a diferencia de las redes ruteadas convencionalmente con IP) a un paquete particular se le asigna una FEC (un camino a seguir en la red) solo una vez, cuando el paquete entra en la red. Esta FEC es codificada en un campo de largo fijo en el encabezado del paquete conocido como etiqueta. En los saltos ó nodos subsecuentes no se analiza nuevamente el encabezado del paquete, sino que se reenvía de acuerdo a la etiqueta del paquete la cual indica el índice del próximo salto y la nueva etiqueta. Por lo tanto la etiqueta se modifica al nuevo valor y el paquete se reenvía por el índice especificado.

El hecho de no ser necesario analizar el encabezado de un paquete mas que en el ¨nodo de ingreso” de la red MPLS y, en los routers/switches subsecuentes hacer el reenvío basado en la etiqueta tiene varias ventajas:

- El reenvío MPLS puede ser realizado por switches que tienen la habilidad de buscar una etiqueta en una tabla y reemplazarla sin la necesidad de tener la habilidad de analizar el encabezado de capa de red del paquete.
- Dado que el paquete sólo se analiza en el “nodo de ingreso” a los efectos de ser asignado a una determinada FEC, en éste análisis es posible tener en cuenta toda la información contenida en el paquete y no solo la disponible en el encabezado de capa de red, sin que esto signifique una gran sobrecarga de procesamiento en el enrutamiento del paquete en toda la red. En este caso el router de ingreso hace el trabajo pesado de procesamiento.
- Paquetes idénticos que entran a la red MPLS mediante distintos “nodos de ingreso” pueden ser etiquetados en forma independiente. Por lo tanto decisiones de enrutamiento basadas en el nodo de ingreso son fácilmente implementables. Esto es un caso de lo que se conoce en la jerga como Policy Routing.

- Las consideraciones para asignar un paquete a una determinada FEC pueden ser más y más complicadas sin ningún impacto en los requerimientos de performance y procesamiento de los routers ó nodos que solo deben forwardear paquetes.
- Si se quiere implementar “Policy Routing”, es decir forzar el camino a seguir por un paquete aún antes de que éste llegue y de manera que no sea establecido por los algoritmos dinámicos de enrutamiento, esto se puede implementar fácilmente asignando a la ruta específica una etiqueta y sin la necesidad de que el paquete sea codificado con un identificador de la ruta específica (“como en el caso de enrutamiento por origen”) más que la etiqueta MPLS.

Algunos routers analizan el encabezado de capa de red de un paquete para determinar su “precedencia” ó “clase de servicio” y en base a esto aplicar distintos umbrales para descartar tráfico ó distintas disciplinas de encolamiento y transmisión. MPLS permite (pero no requiere) que la información de “precedencia” ó “clase de servicio” sea inferida total ó parcialmente de la etiqueta MPLS. Se puede decir que la etiqueta representa la combinación de FEC y “precedencia” ó “clase de servicio”.

La técnica de MPLS es aplicable a “TODOS” los protocolos de capa de red.

4.3.1. Etiqueta

La etiqueta que se pone en un paquete específico representa la FEC a la cual éste es asignado. Si Ru (Router uplink) y Rd (Router downlink) son LSR (label switch routers), éstos se ponen de acuerdo en que cuando Ru transmite un paquete a Rd, Ru va a etiquetar el paquete con una etiqueta L sí y sólo si el paquete es miembro de la FEC F, por lo tanto se establece una asociación entre la FEC F y la etiqueta L para los paquetes transmitidos desde Ru a Rd.

Se debe notar que L no representa la FEC F necesariamente para otros paquetes que no sean aquellos que están siendo enviados de Ru a Rd. Además Rd no es necesariamente el destino final del paquete.

Es responsabilidad de Rd seleccionar la etiqueta L que identifica los paquetes de una única FEC F y por lo tanto puede interpretar unívocamente la etiqueta L de los paquetes entrantes.

4.3.2. Routers de Upstream y downstream

Decir que un router es el upstream y otro es el downstream respecto de una relación etiqueta L y FEC F significa qué, la etiqueta particular representa a la FEC respectiva para los paquetes viajando desde Ru a Rd, pero no implica que todos los paquetes de la FEC F son enrutados desde Ru a Rd.

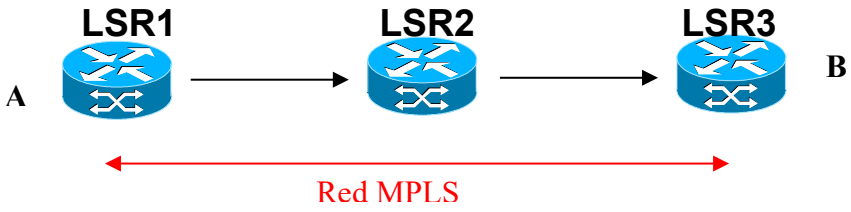

Figura 11

Para un enlace de A -> B:
El LSR1 es el router de ingreso.
El LSR3 es el router de egreso.
El LSR1 es upstream de LSR2 y LSR2 es upstream de LSR3.
LSR3 es downstream de LSR2 y LSR2 es downstream de LSR1.

4.3.3. Paquete etiquetado

Un paquete etiquetado es aquel en el que se ha codificado una etiqueta. En algunos casos la etiqueta reside en un encabezado que existe específicamente para este propósito, mientras que en otros casos la etiqueta puede residir en un campo existente del encabezado de capa 2 ó capa 3. La técnica de codificación de etiqueta debe ser acordada por las dos entidades involucradas, la que codifica y la que decodifica la etiqueta. Se verán más detalles de esto más adelante en este capítulo cuando se aborden las técnicas de codificación de etiquetas.

4.3.4. Asignación y distribución de etiquetas

La decisión de establecer una relación entre la etiqueta L con una FEC F es realizada por el LSR que es downstream respecto de la relación. El LSR downstream informa al LSR upstream de la relación establecida. Las etiquetas son asignadas en el downstream y las asignaciones son transmitidas en la dirección downstream – upstream.

4.3.5. Protocolo de distribución de etiquetas

Un protocolo de distribución de etiquetas es un conjunto de procedimientos por los cuales un LSR informa a otro de las relaciones etiqueta/FEC definidas. Dos LSR que usan un protocolo de distribución de etiquetas para intercambiar información de

relaciones etiquetas/FEC se denominan “Par de distribución de etiqueta”. Si dos LSR forman un “Par de distribución de etiquetas” diremos que hay una “adyacencia de distribución de etiquetas” entre ellos.

4.3.6. Downstream no solicitado vs. Downstream sobre demanda

La arquitectura MPLS permite que un LSR explícitamente solicite a su “next hop” para una FEC en particular la etiqueta correspondiente. Esto se conoce como distribución de etiqueta por “Downstream sobre demanda”.
La arquitectura MPLS también permite que un LSR distribuya relaciones etiqueta/FEC a LSRs que no las han solicitado explícitamente. Esto se conoce como distribución de etiquetas por “downstream no solicitado”. En un par de LSR que forman una adyacencia de distribución de etiquetas, se debe acordar cuál de las dos técnicas se usarán en dicha adyacencia. No obstante una misma plataforma (LSR) puede usar distintas técnicas en distintas adyacencias de distribución.

4.3.7. Stack de etiquetas

En el modelo general de la arquitectura de MPLS, es posible que un paquete tenga más de una etiqueta, esto se maneja en una estructura LIFO, esta estructura se denomina “stack de etiquetas”.
Independientemente de que MPLS soporta niveles de jerarquía, el procesamiento de un paquete etiquetado es independiente del nivel de jerarquía que se esté manejando y siempre se considera la última etiqueta insertada para forwardear o asignar el paquete a una determinada FEC.

4.3.8. NHLFE (next hop forwarding entry)

El NHLFE se usa para reenviar paquetes etiquetados y es una tabla que contiene la siguiente información:

1) El next hop para el paquete.
2) La(s) operación(es) a realizar en el stack de etiquetas del paquete, que puede ser:
    a. Reemplazar la última etiqueta del paquete con una nueva etiqueta.
    b. Hacer un pop del stack de etiquetas.
    c. Reemplazar la última etiqueta del stack y luego hacer uno ó más push en el stack especificando nuevas etiquetas (se incrementa el nivel de jerarquía).

Esta tabla puede contener información adicional como:

3) Encapsulación que se debe usar en el link por el que se va a enviar el paquete.
4) La forma en la que se debe codificar la etiqueta al transmitir el paquete.
5) Cualquier información adicional necesaria para procesar el paquete.

4.3.9. ILM (Incoming Label Map)

El ILM se usa para mapear cada etiqueta entrante en un conjunto de NHLFEs. Esto es útil en aplicaciones donde se desea implementar balanceo de carga.

4.3.10. FTN (FEC to NHLFE mapeo)

Se usa para etiquetar paquetes que llegan sin etiquetas, es decir se asigna el paquete a una determinada FEC.

4.3.11. Swapping de etiquetas

Para forwardear un paquete etiquetado, un LSR examina la etiqueta en la parte de arriba del stack. Luego se usa la información en la ILM para mapear esta etiqueta con un NHLFE. Usando la información en el NHLFE, se determina adónde se debe forwardear el paquete y qué operaciones realizar sobre el stack del paquete antes de forwardearlo. *Es importante destacar que cuando se está haciendo swapping de etiquetas, la información de cuál es el next hop se determina a partir del NHLFE y éste puede ser distinto del next hop si se estuvieran usando los algoritmos tradicionales de enrutamiento.*

4.3.12. Alcance de etiquetas (Label Space)

El alcance de las etiquetas define el espacio donde tienen validez las etiquetas. Éste puede ser definido:

a. Por interfaces, si el LSR es capaz de interpretar paquetes provenientes de distintas interfaces con la misma etiqueta y asignarlos a distintas FEC en base a la interfase de origen.
b. Por plataforma, en este caso la etiqueta es única en la plataforma (LSR) y Rd debe distribuir etiquetas distintas para FECs distintas.

4.3.13. LSP (Label Switching Path), nodo de ingreso y nodo de egreso

El LSP de nivel “m” para un paquete dado es una secuencia de routers , <R1, …., Rn> con las siguientes propiedades:

a. R1, el “ingreso del LSP” es un LSR que hace un push de una etiqueta en el stack de etiquetas del paquete P, lo cual resulta en un stack de etiquetas de profundidad “m”.
b. Para todo i, 1 < i < n, P tiene un stack de etiquetas de profundidad “m” cuando es recibido por Ri.
c. En ningún momento durante la transmisión de P desde R1 hasta R[n-1] el stack de etiquetas tiene profundidad menor a m.

d. Para todo i, 1 < i < n-1, Ri trasmite P a R[i+1] usando la etiqueta de nivel m (la de más arriba del stack) como índice en la ILM.

El LSP termina en el "router de egreso" donde el forwarding del paquete se realiza en un nivel m-k, con k>0, ó por los algoritmos de enrutamiento convencional, es decir no MPLS.

Por lo tanto llamaremos "LSP para una FEC F" particular, a la secuencia de LSR de nivel m para un paquete particular P, donde el nivel "m" en el stack MPLS del paquete P es la correspondencia de etiqueta MPLS a FEC.

La figura siguiente muestra un esquema de las entidades definidas hasta ahora.

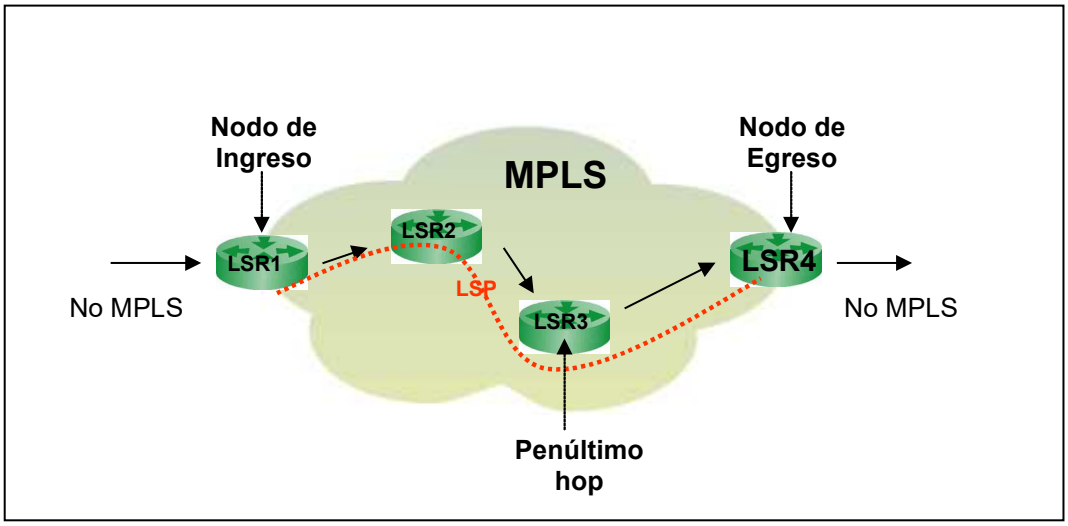

Figura 4.212

4.4. Conceptos básicos de operación en MPLS

4.4.1. Pop en el penúltimo hop

Es posible hacer un pop del stack de nivel "m" de etiquetas en el LSR [n-1], dado que el penúltimo hop es el único que debe tomar una decisión de forwarding basado en la etiqueta de nivel "m" del stack. Desde el punto de vista de la arquitectura esto es perfectamente válido ya que el propósito de la etiqueta de nivel "m" es transportar el paquete P hasta el LSR "n" y una vez que el LSR[n-1] tomó la decisión de enviar P a LSR[n] la etiqueta de nivel "m" ya no cumple ninguna función, por lo tanto no es necesario transmitirla.

La ventaja práctica de hacer pop en el penúltimo hop, es que se ahorra un ciclo innecesario de lookup en el nodo de egreso sin agregar un ciclo en el penúltimo hop, adicionalmente cuando se implementa esto, el último hop ni siquiera necesita ser un LSR. Algunos switches son capaces de "swapear" etiquetas pero nó de hacer un pop,

por lo tanto esto no puede ser universalizado y en el caso que el penúltimo hop sea un switch que no puede hacer pop, esta técnica no puede usarse y el pop debe ser implementado en el nodo de egreso. El pop en el penúltimo hop se implementa solamente en el caso que el LSR de egreso lo solicita específicamente.

4.4.2. Control del LSP: Ordenado vs Independiente

En el modo de establecimiento de LSP independiente, cada LSR hace una decisión independiente de asociar una etiqueta a una FEC particular y distribuye esta asociación a sus pares de distribución.
En el modo Ordenado, un LSR asocia una etiqueta a una FEC si este es el nodo de egreso para esa FEC, o si ya ha recibido una asociación para la FEC en particular desde el próximo hop para esta FEC.

4.4.3. Agregación

Una forma de particionar el tráfico en FECs es crear una FEC para cada prefijo de dirección de capa de red existente en la tabla de enrutamiento. Aunque en algún dominio MPLS esto puede significar que se crea un conjunto de FECs donde todo el tráfico perteneciente a éstas sigue el mismo camino y recibe el mismo tratamiento. En este caso la unión de todas estas FECs es también una FEC. El proceso de crear una sola etiqueta o un subconjunto de etiquetas para identificar la unión de todas las FECs se llama “agregación”.
El proceso de Agregación reduce el número de etiquetas necesarias para manejar un conjunto de paquetes y puede eventualmente reducir el tráfico del protocolo de distribución de etiquetas.

5. Guía de usuario del sistema MPLS

5.1. Introducción

En este capítulo se hará una breve descripción del funcionamiento y aplicaciones de la implementación actual del sistema MPLS para el kernel de Linux. A estos efectos comenzaremos mostrando los comandos de configuración, para luego ver algunos ejemplos y capturas con un analizador de protocolos mostrando el efecto de ls comandos aplicados.

5.2. Funcionalidades y comandos

Las opciones del sistema MPLS en un kernel compilado con soporte MPLS se configuran con uno de los siguientes comandos:

mpls ilm CMD label LABEL labelspace NUMBER [proto PROTO | instructions INSTR]

Este es el formato de la línea de configuración correspondiente a la entrada al sistema de un paquete MPLS. Los valores que pueden tomar cada uno de los modificadores como CMD, LABEL, NUMBER, PROTO y INSTR se aclaran a continuación.

CMD establece si estoy agregando, eliminando o modificando una regla para una etiqueta de entrada al sistema MPLS.
LABEL hace referencia al valor de la etiqueta a la que corresponde la regla.
NUMBER es un número asignado como identificador del Labelspace.
PROTO indica si esta regla es para trafico IP versión 4 o versión 6.
INSTR indica la ó las instrucciones asociadas a la etiqueta entrante, aquí es donde en realidad se configura cómo procesar el paquete que llega con la etiqueta especificada en LABEL.

mpls nhlfe CMD key KEY [mtu MTU propagate_ttl | instructions INSTR]

La idea de esta línea de comando es la misma que para la etiqueta entrante, salvo que esto corresponde a una regla que se aplica a la salida de un paquete del sistema MPLS. El campo KEY se usa debido a la implementación en el kernel, ya que cada regla esta ordenada en una hashtable y esta “key” se usa para buscar en la tabla.

mpls labelspace CMD dev NAME labelspace NUMBER

Esta línea permite definir labelspace y agregar ó eliminar dispositivos físicos (tarjetas de red) en los espacios definidos. Para que una interfase física pueda manejar tráfico MPLS debe estar asociada a un labelspace definido en el sistema.

mpls tunnel CMD dev NAME key KEY

Esta línea permite definir túneles, estos son interfaces virtuales que luego pueden ser usadas para definir rutas y propagarlas mediante los algoritmos de enrutamiento.

Con los siguientes comandos puedo ver estadísticas y configuración del sistema MPLS:

mpls ilm show [label LABEL labelspace NUMBER]
Muestra configuración y estadísticas para las etiquetas de entrada:

[root@ORENSE root]# mpls ilm show
ILM entry label gen 100 labelspace 0 proto ipv4
pop peek (3480 bytes, 9 pkts, 0 dropped)

mpls nhlfe show [key KEY]
Muestra configuración y estadísticas para las etiquetas de salida:

[root@PONTEVEDRA root]# mpls nhlfe show
NHLFE entry key 0x00000002 mtu 1496 propagate_ttl
push gen 100 set eth0 ipv4 192.168.10.10 (420 bytes, 5 pkts, 0 dropped)

mpls labelspace show [dev NAME]
Muestra labelspace definidos y estadísticas.

[root@ORENSE root]# mpls labelspace show
LABELSPACE entry dev eth1 labelspace -1
LABELSPACE entry dev eth0 labelspace 0
LABELSPACE entry dev lo labelspace -1

Opciones de los modificadores:

CMD   := add | del | change
NUMBER := 0 .. 255
TYPE   := gen | atm | fr
VALUE := 16 .. 1048575 | <VPI>/<VCI> | 16 .. 1023
LABEL := TYPE VALUE
KEY   := 0 for add | previously returned key
NAME   := network device name (i.e. eth0)
PROTO := ipv4 | ipv6
ADDR   := ipv6 or ipv4 address
NH   := nexthop NAME [none | PROTO ADDR]
FWD   := forward KEY
PUSH   := push LABEL
INSTR := NH | PUSH | pop | deliver | peek | FWD
set-dscp <DSCP> | set-exp <EXP> |
set-tcindex <TCINDEX> | set-rx-if <NAME>
forward <KEY> | expfwd <EXP> <KEY> ...
exp2tc <EXP> <TCINDEX> ... | exp2ds <EXP> <DSCP> ...
nffwd <MASK> [<NFMARK> <KEY> ...]
nf2exp <MASK> [<NFMARK> <EXP> ...]
tc2exp <MASK> [<TCINDEX> <EXP> ...]

ds2exp <MASK> [<DSCP> <EXP> ...]
dsfwd <MASK> [<DSCP> <KEY> ...]

Nota: Si bien estos son los valores válidos de los modificadores, no todas las funcionalidades están aún implementadas en el sistema MPLS. Por ejemplo aún no está implementado el soporte para Frame Relay o ATM.

5.3. Ejemplos de Configuración

5.3.1. LSP, una sola etiqueta

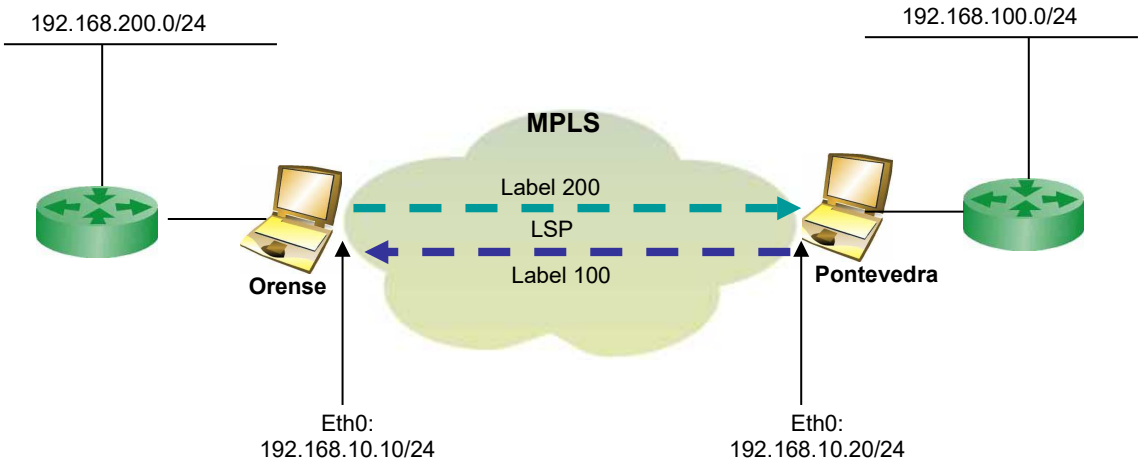

Figura 5.1

A continuación se indica la configuración para levantar los LSP y "mapear" el tráfico a través de ellos.

Configuración de Pontevedra:

```
mpls nhlfe add key 0 instructions push gen 100 nexthop eth0 ipv4
192.168.10.10

ip route add 192.168.200.0/24 via 192.168.10.10 dev eth0 spec_nh
0x8847 0x2
mpls labelspace add dev eth0 labelspace 0
mpls ilm add label gen 200 labelspace 0 instructions pop deliver
```

El primer comando crea una nueva regla para el forwarding. El parámetro "nhlfe" indica que se configura esa tabla y "add key 0" significa una nueva entrada, una aclaración importante es que después de ejecutar el comando, el sistema MPLS devuelve la key de la nueva regla creada (en el ejemplo la key devuelta es 0x2). En caso de querer agregar instrucciones a una regla existente, se debe poner la key asociada.
A continuación van las instrucciones, en este caso "push gen 100", agrega una etiqueta genérica de valor 100 al paquete. Después "nexthop eth0 ipv4 192.168.10.10" indica que se envíe el paquete al siguiente nodo vía la interfase eth0.

El segundo comando mapea tráfico IP hacia un LSP, para eso se debe usar una de las reglas nhlfe creadas (y la key asociada).

El tercer comando asigna el valor de labelspace a una interfase determinada, en este caso eth0.

El cuarto comando crea una nueva regla de mapeo de etiquetas entrantes. El parámetro "ilm" indica que se configura esa tabla. Para eso hay que indicar valor de etiqueta de entrada "label gen 200" y valor de labelspace "labelspace 0". Con estos valores queda identificado cuales son los paquetes que serán procesados por esta nueva regla, falta agregar las instrucciones a ejecutar en la misma, "pop" extrae la primera etiqueta del stack y "deliver" entrega el paquete a las capas superiores.

Configuración de Orense:

```
mpls nhlfe add key 0 instructions push gen 200 nexthop eth0 ipv4
192.168.10.20
ip route add 192.168.100.0/24 via 192.168.10.20 dev eth0 spec_nh
0x8847 0x2
mpls labelspace add dev eth0 labelspace 0
mpls ilm add label gen 100 labelspace 0 instructions pop deliver
```

A continuación se muestra el resultado de una captura usando Ehtereal, luego de hacer la configuracion mencionada en la maqueta.

Ping generado desde un PC con la IP 192.168.100.23 conectado a Pontevedra.

1	0.000000	192.168.100.23	192.168.200.10	ICMP	Echo (ping) request
2	0.000837	192.168.200.10	192.168.100.23	ICMP	Echo (ping) reply
3	1.000867	192.168.100.23	192.168.200.10	ICMP	Echo (ping) request
4	1.001645	192.168.200.10	192.168.100.23	ICMP	Echo (ping) reply
5	2.000852	192.168.100.23	192.168.200.10	ICMP	Echo (ping) request
6	2.001687	192.168.200.10	192.168.100.23	ICMP	Echo (ping) reply
7	3.001784	192.168.100.23	192.168.200.10	ICMP	Echo (ping) request
8	3.002567	192.168.200.10	192.168.100.23	ICMP	Echo (ping) reply

Figura 5.2 – Ping Request y Reply

```
▸ Frame 1 (78 bytes on wire, 78 bytes captured)
▾ Ethernet II, Src: 00:e0:7d:94:ec:40, Dst: 00:e0:4c:9c:84:5b
    Destination: 00:e0:4c:9c:84:5b (Realteks_9c:84:5b)
    Source: 00:e0:7d:94:ec:40 (Netronix_94:ec:40)
    Type: MPLS label switched packet (0x8847)
▾ MultiProtocol Label Switching Header
    MPLS Label: Unknown (100)
    MPLS Experimental Bits: 0
    MPLS Bottom Of Label Stack: 1
    MPLS TTL: 127
▸ Internet Protocol, Src Addr: 192.168.100.23 (192.168.100.23), Dst Addr: 192.168.200.10 (192.168.200.10)
▸ Internet Control Message Protocol
```

Figura 5.3 – Decodificación del Ping Request

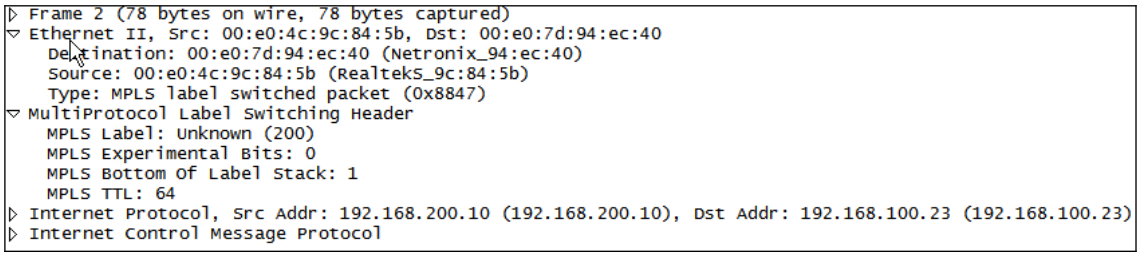

Figura 5.4 – Decodificación del Pin Reply

5.3.2. Ejemplo con dos etiquetas:

```
Configuración Pontevedra
mpls nhlfe add key 0 instructions push gen 300 nexthop eth0 ipv4
192.168.10.10
mpls nhlfe add key 0 instructions push gen 100 forward 0x2
ip route add 192.168.200.0/24 via 192.168.10.10 dev eth0 spec_nh
0x8847 0x3
mpls labelspace add dev eth0 labelspace 0
mpls ilm add label gen 300 labelspace 0 instructions pop peek
mpls ilm add label gen 200 labelspace 0 instructions pop deliver
```

En este caso, cuando llega un paquete a Pontevedra desde eth1 y con IP de destino 192.168.200.x este se manda al LSP que tiene label 100 haciendo en este momento un push con la etiqueta 100 y este a su vez forwardea este paquete al LSP que tiene etiqueta 300, aquí se hace el segundo push con la etiqueta 300. Por lo tanto el paquete sale por eth0 con dos etiquetas (100 y 300) donde la 100 es el BOS (bottom of the stack) por lo tanto la primera operación en el próximo host MPLS será en base a la etiqueta 300.

Configuración Orense

```
mpls nhlfe add key 0 instructions push gen 300 nexthop eth0 ipv4
192.168.200.11
mpls nhlfe add key 0 instructions push gen 100 forward 0x2
ip route add 192.168.200.0/24 via 192.168.10.10 dev eth0 spec_nh
0x8847 0x3
mpls labelspace add dev eth0 labelspace 0
mpls labelspace add dev eth1 labelspace 0
mpls ilm add label gen 300 labelspace 0 instructions pop peek
mpls ilm add label gen 100 labelspace 0 instructions pop deliver
```

```
A continuación se muestra el resultado de una captura usando Ehtereal,
luego de hacer la configuracion mencionada con dos etiquetas MPLS.

Ping generado desde un PC con la IP 192.168.100.23 conectado a
Pontevedra.
```

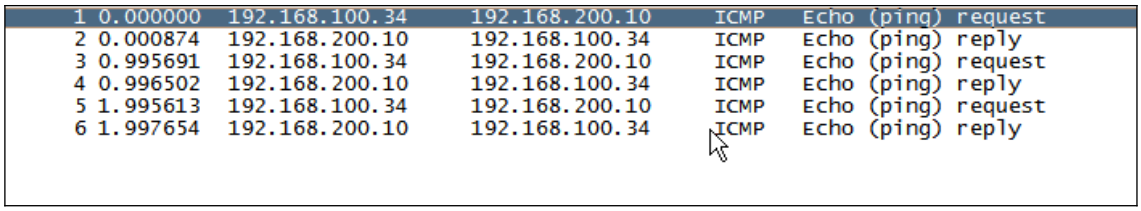

Figura 5.5 – Ping Request y Reply

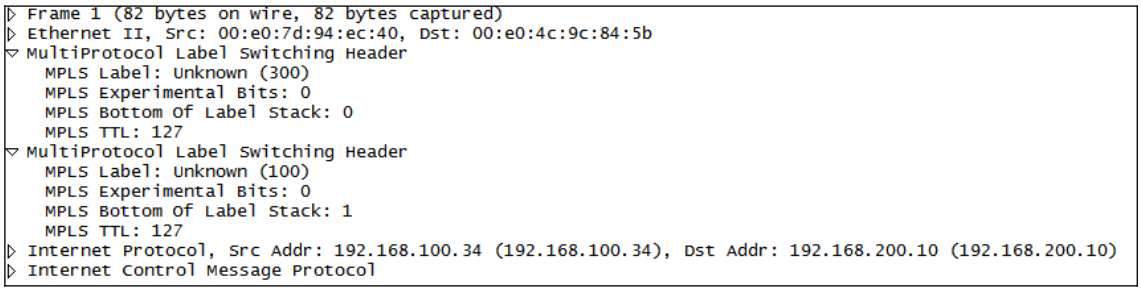

Figura 5.6 – Decodificación del Ping Request

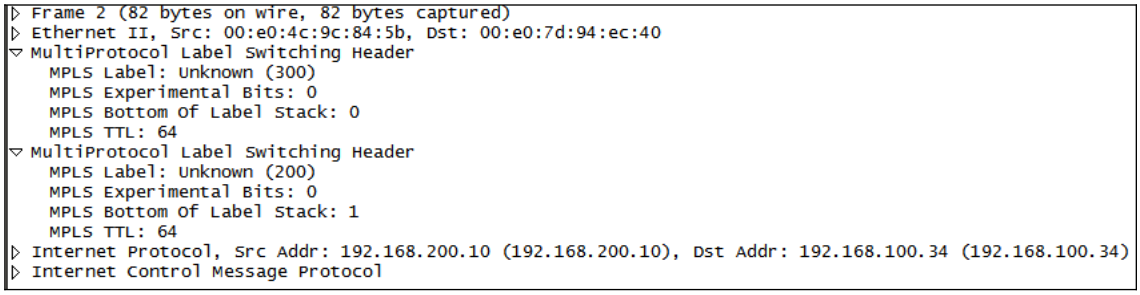

Figura 5.7 – Decodificación del Ping Reply

Implementación actual de MPLS en el Kernel

5.4. Introducción

El objetivo de este capítulo es explicar la implementación y funcionamiento del sistema MPLS.

El sistema MPLS se puede separar en 3 partes:

1.- Una serie de estructuras que sirven para almacenar los datos permanentes, transitorios y/ó de la configuración hecha al sistema MPLS.

2.- Una serie de algoritmos que realizan toda la tarea de enrutamiento y procesamiento de los paquetes que atraviesan el sistema MPLS, además de administrar las estructuras de datos antes mencionadas.

Estas dos partes del código se compilan dentro del kernel, formando entonces parte de kernel mismo. Además el sistema MPLS debe interactuar con el kernel de Linux. Para la implementación del sistema MPLS, se deben alterar estructuras existentes ó modificar funciones del kernel, un ejemplo simple y evidente es la definición de la estructura net_device, la cual debe ser modificada para agregarle un vínculo al sistema MPLS.

3.- Por último, una tercera parte, que es la implementación de la interfase entre el sistema (implementado en el kernel) y el usuario, a fin de que éste lo configure. Esta parte del sistema a diferencia de las otras dos, se compila en el espacio de usuario y corre como una aplicación en el mismo.

Como último comentario vale la pena aclarar que la anterior separación, se hace a los efectos de hacer más fácil la comprensión del código en su conjunto. Dicho esto, comenzaremos viendo entonces algunas de las estructuras de datos del sistema (las más importantes) y también cómo interactúan entre ellas y con el resto del kernel.

5.5. Estructuras de datos

La implementación de MPLS en el kernel de Linux contiene (como todo sistema) una serie de estructuras que se utilizan para almacenar los datos transitorios, permanentes y de configuración del forwarding MPLS, en este punto se hará una descripción de las más importantes.

5.5.1. NETDEVICES, MPLS_INTERFACES Y LABELSPACE.

Recordando que un valor de etiqueta (label) representa una "FEC" (Forwarding Equivalence Class) localmente entre dos LSR (Label Switching Router) y que las operaciones requeridas en la recepción de un paquete dependen del valor de la etiqueta recibido y del LSR que envía el paquete, es que se hace necesario para forwardear correctamente los paquetes (en virtud que diferentes LSR nos pueden enviar paquetes con el mismo valor de etiqueta) conocer el valor de la etiqueta y el labelspace de la interfase por la que entra el paquete.

De este modo se define y asigna un "labelspace" a cada una de las interfaces de red que serán utilizadas para recibir paquetes MPLS, como un atributo de ellas mismas, que será el mismo para todas las interfaces en caso de usar alcance por plataforma ó distinto si se quiere tener un alcance por interfase.

Para identificar las interfaces que están habilitadas para cursar tráfico MPLS, se amplia la estructura net_device con un puntero a una estructura de tipo mpls_interface, la cual contiene básicamente el labelspace asociado a la interfase. Esta extensión se hace agregando (en el archivo /include/linux/netdevice.h) a la definición de la estructura net_device la siguiente línea:

```
void  *mpls_ptr;  /* MPLS specific data */
```

Y la definición de mpls_interface es;

```
struct mpls_interface {

   struct list_head list_out;

   struct list_head list_in;

   /*
    * Label Space for this interface
    */
   int  labelspace;
};
```

De modo que, dada una interfase de red, la existencia ó no del puntero al objeto "mpls_interface" indica si el dispositivo está habilitado a cursar tráfico mpls. Una aclaración importante es que el valor –1 indica que el dispositivo está inhabilitado para cursar tráfico MPLS.

Una última observación es que en la definición de net_device el puntero mpls_ptr no tiene tipo asociado (void) por lo que hay que hacer el "cast" adecuado para referenciarlo.

5.5.2. MPLS INPUT/OUTPUT INFORMATION y MPLS KEY

De manera de obtener en forma sencilla el set de operaciones a realizar a cada paquete que entra ó sale, el sistema MPLS administra un tipo de estructura para almacenar la información de entrada (ILM Incoming Label Map) y otro tipo para la salida (NHLFE Next Hop Label Forwarding Entres), en ellos se guardan todos los datos necesarios para el correcto procesamiento y forwarding de los paquetes. A su vez cada regla que el usuario programa para las etiquetas de entrada ó de salida, genera una instancia de estas estructuras, la cual hay que almacenar para su posterior consulta. Se necesita en consecuencia una herramienta para estas operaciones.

Para almacenar estas instancias se utiliza un "radix tree", en pocas palabras diremos que es una estructura en forma de árbol, este "radix tree" almacena cada instancia con una referencia ó "key", la cual sirve para acceder a cada instancia dentro del árbol con la "key" asociada. El sistema MPLS usa dos radix-tree, uno para la entrada y otro para la salida.

Para clarificar diremos que al crear una regla para los paquetes que arriben con etiqueta 34 y otra con etiqueta 25, se crean dos instancias de tipo mpls_ilm, aunque configuremos el sistema para que actúe en forma idéntica para los dos casos, entonces al llegar un paquete con una de las dos etiquetas, el sistema MPLS buscará dentro del radix-tree la instancia asociada a ese valor de etiqueta.

En el caso entrante y para simplificar el algoritmo, la "key" se obtiene combinando el labelspace y el valor de la etiqueta. Más adelante, a continuación de la descripción de las estructuras ILM y NHLFE se detallará la implementación de las "key".

ILM Object

Como ya se dijo, toda la información relativa a la recepción se almacena en una estructura de tipo ***mpls_ilm*** (ver apéndice B), la cual es posible obtener del radix-tree de entrada, a partir del valor de etiqueta y la interfase de entrada. Los campos más importantes que contiene esta estructura son:

1) Las operaciones MPLS a ejecutar para esta regla de entrada. Para esto se tiene un puntero que referencia a la primera instrucción MPLS que se debe ejecutar (una explicación más detallada de cómo trabajan las instrucciones la haremos más adelante)
2) La clave ó "key" a ser utilizada para buscar esta estructura dentro del radix-tree.
3) El "protocol driver" para los paquetes que usen esta ILM. En la implementación actual del sistema MPLS se pueden procesar dos tipos de paquetes: Ethernet Ipv4 y Ethernet Ipv6, entonces cada tipo de paquete debe ser procesado por un

"protocol driver" distinto y cada regla debe tener un puntero al "protocol driver" adecuado.
4) Otros datos propios de la presente regla. LabelSpace, etc.

NHLFE Object

De manera análoga a la recepción todo lo concerniente a la transmisión se almacena en una estructura llamada *mpls_nhlfe* (ver apéndice B). Los campos más importantes aquí son;

1) El destino del paquete. Para almacenarlo se usa una unión, ya que el destino puede ser otro host, para el cual se usa una estructura *dst_entry* ó también el destino puede ser un Forwarding y en ese caso lo que hay que almacenar es otra estructura *mpls_nhlfe*.
2) Las operaciones MPLS a ejecutar por esta regla de salida. Para esto se tiene un puntero que referencia a la primera instrucción MPLS que se debe ejecutar.
3) La clave usada para buscar el objeto NHLFE en el radix-tree.
4) Otros datos propios de la regla. Device de salida, etc.

5.5.3. MPLS KEY

La implementación se hace con una estructura llamada "mpls_key" que multiplexa (utilizando una unión en "C") entre 3 tipos distintos de etiquetas; ATM , Frame Relay ó Generic Key. Las cuales son siempre equivalentes a un entero de 32 bits formado por diferentes sub-campos. En este punto se aclara que actualmente no está implementado el soporte para estos dos protocolos, por lo que solo detallaremos las "generic keys".

```
struct mpls_gen_key {
        unsigned int type:2;
        unsigned int index:10;
        unsigned int gen:20;
};

struct mpls_key {
        union {
                struct mpls_atm_key atm;
                struct mpls_gen_key gen;
                struct mpls_fr_key  fr;
                unsigned int mark:32;
        } u;
```

```
};
```

5.5.4. MPLS LABEL.

Básicamente la estructura se compone de un índice y un entero de 32 bits que puede ser una “key” ó una etiqueta dependiendo del alcance. En otras palabras, si consideramos el índice como el labelspace, entonces esta estructura contiene la etiqueta, el tipo de etiqueta y el labelspace.

```
struct mpls_label {
      enum mpls_label_type_enum ml_type;
      union {
            u_int32_t ml_key;
            u_int32_t ml_gen;
            u_int32_t ml_fr;
            struct mpls_label_atm ml_atm;
      } u;
      int ml_index;
};

enum mpls_label_type_enum {
      MPLS_LABEL_GEN = 1,
      MPLS_LABEL_ATM,
      MPLS_LABEL_FR,
      MPLS_LABEL_KEY
};
```

5.5.5. Parámetros MPLS

Todos los parámetros relativos al “header” MPLS se almacenan en la siguiente estructura:

```
   struct mpls_skb_parm {
      struct mpls_prot_driver *prot;
      unsigned int  gap;
      unsigned int  label:20;
      unsigned int  ttl:8;
      unsigned int  exp:3;
      unsigned int  bos:1;
      unsigned char flag;
      unsigned char popped_bos;
      unsigned char *top_of_stack;
   };
```

Para almacenar el header MPLS en el socket buffer dentro del kernel se utiliza un puntero de propósito general (cb), recordamos que la definición de socket buffer se puede ver en el apéndice A

Por esta razón y para acceder fácilmente a la estructura mpls_skb_parm desde cualquier punto del código MPLS, hay definida una macro que nos permite acceder al lugar adecuado dentro del skb.

```
#define MPLSCB(skb) ((struct mpls_skb_parm*)((skb)->cb))
```

5.5.6. Operaciones e Instrucciones MPLS

¿Cómo procesar correctamente un paquete entrante ó saliente? Hasta aquí vimos que al configurar el sistema MPLS se crean estructuras ILM ó NHLFE las cuales contienen todos los datos necesarios para el forwarding. Ahora la pregunta es cómo se implementa esto, para eso vamos a ver el caso entrante, cuando se asigna una etiqueta de entrada se genera una estructura ILM de tipo struct mpls_ilm, que contiene entre otros campos un puntero a una estructura de tipo mpls_instr

```
struct mpls_instr {
      struct mpls_instr  *mi_next;
      unsigned short      mi_opcode;
      enum mpls_dir       mi_dir;
      void               *mi_data;
      void *              mi_parent;
};
```

El campo fundamental de esta estructura es ***mi_opcode***, ya que es la referencia a una de las operaciones posibles (MPLS_OP_DLV, MPLS_OP_POP, etc), los otros campos son;

1) *mi_next, un puntero a la siguiente operación (si es necesario), esto permite concatenar diferentes operaciones en serie, aspecto muy importante al irse creando las reglas ILM y NHLFE.
2) mi_dir, indica la dirección (entrada ó salida)
3) *mi_data, puntero a los datos asociados a la instrucción.

Ahora, ¿Cómo se utiliza esta estructura para realizar todas las operaciones requeridas para procesar nuestros paquetes que llegan con determinado valor de etiqueta?

En cada instancia ***mpls_instr*** que se crea, guardamos el código de la operación que se quiere ejecutar, los datos asociados (si es que existen), etc. El código (opcode) es el que nos permite acceder (actuando como clave) a todas las funciones para ejecutar esa operación.

5.5.7. Implementación de las Operaciones

Cada operación tiene asociados varios punteros a funciones que son los que definen cómo se crea, limpia, imprime y/ó ejecuta la correspondiente operación, y se almacenan en una estructura de tipo struct mpls_ops, además se crea un array de la siguiente forma (definido en el archivo mpls_opcode.c);

```
struct mpls_ops mpls_ops[MPLS_OP_MAX] = {
      [MPLS_OP_NOP] = {
            .in      = mpls_op_nop,
```

```
        .out     = mpls_op_nop,
        .build   = NULL,
        .unbuild = NULL,
        .cleanup = NULL,
        .extra   = 0,
        .msg     = "NOP",
    },
    [MPLS_OP_POP] = {
        .in      = mpls_in_op_pop,
        .out     = NULL,
        .build   = mpls_build_opcode_pop,
        .unbuild = NULL,
        .cleanup = NULL,
        .extra   = 0,
        .msg     = "POP",
    }.....
```

Para clarificar lo que se mencionó en el párrafo anterior, el campo .in es el que se accede en el procesamiento entrante, por lo tanto aquí debe estar la función a ejecutar en esta operación para el caso entrante, de igual forma el campo .out es el relativo al procesamiento de salida, etc. Para el caso que no tenga utilidad alguno de los campos, éstos se inicializan en NULL.

Así definido, el array contiene todas las operaciones posibles y accedemos a cada una por su correspondiente op_code (MPLS_OP_POP, etc). De esta manera con el op_code contenido en mpls_instr se puede acceder a los punteros asociados a cada una de las operaciones y/ó los datos asociados.

Veamos algunas de las operaciones;

5.5.8. POP

La operación POP se utiliza para extraer una etiqueta de la pila (no para analizar ni guardar, solo la "saca" del skb), el puntero *.in* apunta a la función *mpls_in_op_pop()* y es la que se ejecuta cada vez que queremos extraer una etiqueta de un paquete entrante. Observar además la asignación *.out=NULL*, esto es coherente, ya que no tiene sentido esta operación para un paquete en sentido saliente.

Lo primero que hace la función es chequear que no se haya extraído la última etiqueta y que sea posible extraerla

```
    if (MPLSCB(*skb)->popped_bos || (((*skb)->data + sizeof(u32)) >=
(*skb)->tail)) {
        return MPLS_RESULT_DROP;
    }
```

Pasada la comprobación anterior hay que ver si la etiqueta a ser extraída es la última (el bit BOS es el que indica esta situación), en ese caso hay que "levantar" el "flag" popped_bos.

```
        if (MPLSCB(*skb)->bos) {
              MPLSCB(*skb)->popped_bos = 1;
        }
```

Ahora si, ya se puede extraer la etiqueta del encabezado usando la función skb_pull(), y se corrigen los punteros de red (nh) y de transporte (h), esto último es necesario porque la función skb_pull() los "corre" al sacar la etiqueta.

```
        skb_pull(*skb, sizeof(u32));
        (*skb)->h.raw     += sizeof(u32);
        (*skb)->nh.raw    += sizeof(u32);
        MPLSCB(*skb)->gap += sizeof(u32);
        return MPLS_RESULT_SUCCESS;
}
```

5.5.9. PEEK

Vimos el caso anterior que extrae la etiqueta, esta operación por el contrario lo que hace es "leer" la que está primero en el stack, NO la extrae. De nuevo esta es una función que se usa para la recepción y el puntero **.in** apunta a *mpls_in_op_peek()*

El funcionamiento es simple, primero se fija si es la última etiqueta del stack y en caso afirmativo devuelve "Delivery" como resultado, esto es así porque al ser la última etiqueta no queda otra opción que entregar el paquete al propio host para que lo analize la capa de red.

```
        if (MPLSCB(*skb)->bos) {
              return MPLS_RESULT_DLV;
```

Si no es la última etiqueta del stack, se obtiene el shim desde el encabezado de red,

```
        shim = ntohl(*((u32*)&skb->nh.raw));
```

y ya con el shim se pueden obtener los distintos campos que componen el encabezado MPLS, incuído claro está la propia etiqueta.

```
        if (!(MPLSCB(skb)->flag)) {
              MPLSCB(skb)->ttl  = shim & 0xFF;
              MPLSCB(skb)->flag = 1;
        }
        MPLSCB(skb)->bos   = (shim >> 8 ) & 0x1;
        MPLSCB(skb)->exp   = (shim >> 9 ) & 0x7;
        MPLSCB(skb)->label = (shim >> 12) & 0xFFFFF;

        return MPLS_RESULT_RECURSE;
```

Observar que el resultado que devuelve esta función es "RECURSE", esto le indica al algoritmo de recepción que debe analizar el valor de la etiqueta para volver a procesar el paquete. Aquí se ve porqué trabajan en conjunto las operaciones POP-PEEK, la primera operación extrae la etiqueta que está encima en el stack y PEEK "lee" el valor de la que viene debajo para que se pueda tomar la decisión de cómo continuar el procesamiento.

5.5.10. DLV

En este caso lo único que hace la función es devolver el valor DLV, para indicar al algoritmo que haga el "delivery" a las capas superiores.

```
return MPLS_RESULT_DLV;
```

5.5.11. PUSH

Esta es una de las operaciones más importantes, ya que es la que agrega el shim-header a los paquetes MPLS y si bien los punteros .in y .out apuntan a *mpls_op_push()*, la función que contiene el código que se ejecuta es *mpls_push()*.

Lo primero que hace esta función es hacer espacio para el shim usando para eso la función *skb_push()*, y desplazando también los puntero *h.raw* y *nh.raw*, aquí obviamos la parte de código que resuelve el problema de falta de espacio para el shim, ya que no aporta a lo sustancial de esta función.

```
skb_push(o,sizeof(u32));
       o->h.raw -= sizeof(u32);
       o->nh.raw -= sizeof(u32);
```

Teniendo ya el espacio suficiente para colocar el shim, lo que resta es agregarlo en la parte de encabezado de red, usando para eso los 4 campos involucrados.

```
shim = htonl(((label & 0xFFFFF) << 12) |
            ((MPLSCB(o)->exp & 0x7) << 9) |
            ((MPLSCB(o)->bos & 0x1) << 8) |
             (MPLSCB(o)->ttl & 0xFF));
memmove(o->data,&shim,sizeof(u32));
```

Finalmente se actualizan los campos de la estructura *mpls_skb_parm*, observar que se "setea" el bit BOS en 0, por lo que si se hace otro PUSH, no quedará marcado ese bit.

```
MPLSCB(o)->label = label;
MPLSCB(o)->bos = 0;
MPLSCB(o)->popped_bos = 0;
```

Cuando se vea la recepción y transmisión se verá más en profundidad como usa el sistema MPLS estas operaciones.

5.6. Algoritmos Principales Del Sistema MPLS

5.6.1. Inicialización del sistema MPLS

Las principales tareas que se realizan cuando se inicializa el sistema MPLS son:

- Registro del ”packet handler” para tráfico MPLS
- Registro del netdevice event notifier

5.6.2. Registro del packet handler

Una de las primeras operaciones que se debe realizar al arrancar el sistema MPLS es registrar un “packet handler” para el protocolo MPLS. El packet handler es el encargado de recibir y procesar correctamente los paquetes MPLS, así como otros packets handler reciben y manejan los paquetes de otros tipos (IP, ARP, etc).

El propio kernel proporciona una función apropiada para este registro,

```
Void dev_add_pack (struct packet_type *pt);
```

Por lo tanto lo único que debe hacer el sistema MPLS es completar correctamente una estructura de tipo packet_type con el “protocol ID” de MPLS (el cual está normalizado tanto para tráfico unicast como para tráfico multicast) y un puntero a una función, que es la que será llamada por el kernel de linux cada vez que éste reciba un paquete MPLS.

```
static struct packet_type mpls_uc_packet_type = {
      .type = __constant_htons(ETH_P_MPLS_UC), /* MPLS Unicast PID */
      .dev  = NULL,                            /* All devices */
      .func = mpls_skb_recv,
}
```

5.6.3. Registro del NetDevice Event Notifier

Un NetDevice Event Notifier es un puntero a una función que es llamada por el kernel de linux cada vez que ocurre un evento en un determinado dispositivo de red, por ejemplo cada vez que el dispositivo se “cae” ó se “levanta”.

```
      register_netdevice_notifier(&mpls_netdev_notifier);
```

5.6.4. Recepción de paquetes MPLS

Cada paquete que arriba por una interfase de entrada genera una serie de eventos y acciones en el stack IP del kernel de linux, una de ellas es decidir, en función del protocol ID ó packet type, cómo procesarlo. Por eso una de las modificaciones que debe realizar el sistema MPLS es agregar una función adecuada para el procesamiento de los paquetes MPLS entrantes (en este caso la función mpls_skb_recv()).

Función mpls_skb_recv:

Vamos a ver más en detalle cómo trabaja esta función.

```
int mpls_skb_recv (struct sk_buff     *skb, struct net_device  *dev,
                   struct packet_type *pt) {
      int labelspace;
      int result = NET_RX_DROP;
      struct mpls_label label;
      struct mpls_interface *mip = dev->mpls_ptr;
```

Lo primero es obtener a partir del dispositivo de entrada el puntero a mpls_interface. A continuación se realizan chequeos para comprobar que se pueda continuar con el procesamiento. El siguiente diagrama muestra en forma esquemática el proceso en la recepción de un paquete MPLS.

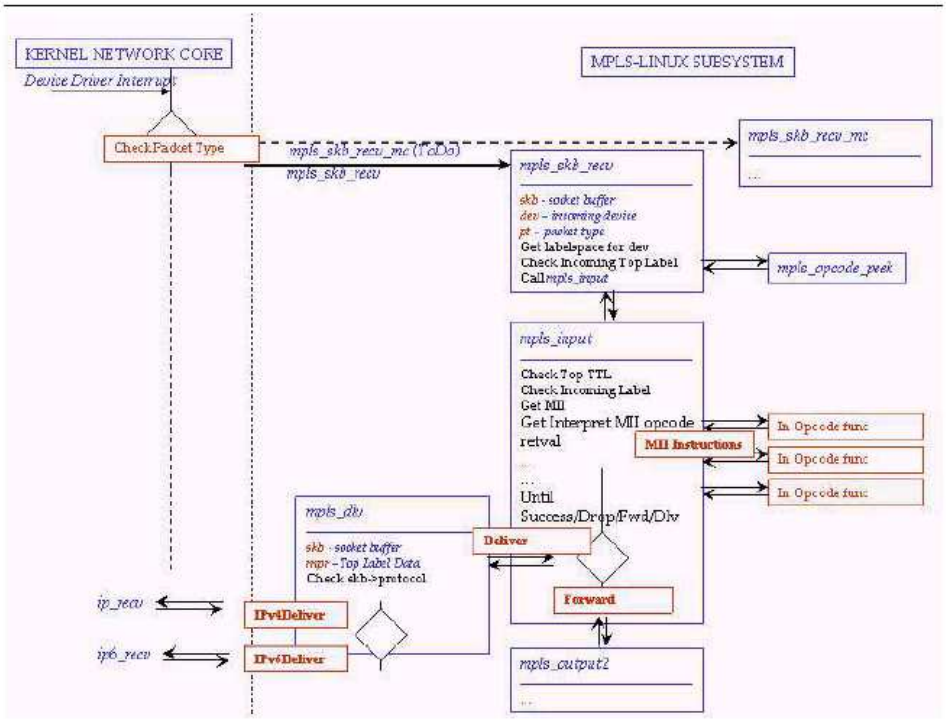

Figure 5.1: MPLS Input Overview

Figura 6.1[http://perso.enst.fr/~casellas/mpls-linux/index.html]

```
memset(MPLSCB(skb), 0, sizeof(*MPLSCB(skb)));
memset(&label, 0, sizeof(label));
```

En estas líneas se inicializan dos estructuras en 0; mpls_skb_parm (recordar que son los datos MPLS almacenados dentro del skb) y mpls_label. A continuación se comienzan a rellenar todos los campos;

```
MPLSCB(skb)->top_of_stack = skb->data;

mpls_opcode_peek (skb);
```

Esta función obtiene la etiqueta que esta en la parte superior del stack y completa el resto de los campos a partir de la etiqueta misma recibida (TTL, BOS Bottom of Stack Bit, EXP bits para E-LSP).

Luego a partir del valor obtenido de etiqueta se llenan los campos de mpls_label.

```
label.ml_type  = MPLS_LABEL_GEN;
label.u.ml_gen = MPLSCB(skb)->label;


if (mpls_input (skb,dev,pt,&label,labelspace))
            goto mpls_rcv_drop;
```

En esta función (mpls_input) se comienza el procesamiento en sí del paquete, para eso se obtiene -a partir de los valores de la etiqueta y del labelspace-, la estructura ILM asociada (la regla para procesar la recepción de los paquetes, como ya se explicó) y con ella se comienzan a aplicar todas las operaciones configuradas para esta ILM.

```
/* Iterate all the opcodes for this ILM */
      for (mi = ilm->ilm_instr; mi; mi = mi->mi_next) {
            data   = mi->mi_data;
            opcode = mi->mi_opcode;
            msg    = mpls_ops[opcode].msg;
            func   = mpls_ops[opcode].in;

            MPLS_DEBUG("opcode %s\n",msg);
            if (!func) {
                  MPLS_DEBUG("invalid opcode for input: %s\n",msg);
                  goto mpls_input_drop;
            }


            switch (func(&skb,ilm,&nhlfe,data)) {
                  case MPLS_RESULT_RECURSE:
                        label->ml_type = MPLS_LABEL_GEN;
                        label->u.ml_gen = MPLSCB(skb)->label;
                        goto mpls_input_start;
                  case MPLS_RESULT_DLV:
                        goto mpls_input_dlv;
                  case MPLS_RESULT_FWD:
                        goto mpls_input_fwd;
                  case MPLS_RESULT_DROP:
                        goto mpls_input_drop;
                  case MPLS_RESULT_SUCCESS:
                        break;
```

```
}
```

En este algoritmo se van ejecutando en forma secuencial todas las operaciones programadas en la ILM, para esto obtenemos de cada operación el opcode y los datos asociados. Estos valores son los fundamentales, ya que a partir del opcode se puede acceder a la función que ejecuta la operación en cuestión.

Ahora, veamos más en detalle el tema de cómo se asigna esa función, por ejemplo si la operación que se está procesando es POP, entonces la variable auxiliar *opcode* es igual a MPLS_OP_POP y la línea

```
func = mpls_ops[opcode].in;
```

Es equivalente a

func = mpls_op_in_pop

Asignada correctamente la función que procesará la operación, se ejecuta dentro del switch de forma de utilizar el resultado de la ejecución para decidir cómo continúa el procesamiento del paquete, que puede ser entregar el paquete a las capas superiores (deliver), forward a través de la red MPLS, etc.

Como forma de reforzar los conceptos y a modo de resumen, en el algoritmo de la función *mpls_input()* se hace la iteración, de ésta surge principalmente el valor del *opcode*, y con este valor se accede a las funciones (definidas en otro archivo) que ejecutan la operación. Esto permite tener un algoritmo central simple, mientras el código de las funciones lo tenemos en otro archivo independiente, permitiendo una fácil implementación de las operaciones.

5.6.5. Envío de paquetes MPLS

En la parte anterior vimos cómo el sistema MPLS junto con el Kernel de Linux realizan la recepción de los paquetes MPLS. En este punto veremos cómo se realizan las funciones de envío y forwarding de las tramas MPLS

Un paquete que necesita ser reenviado (forwarding) puede venir desde las capas superiores (IPV4) ó que como resultado del procesamiento en las funciones de entrada, sea necesario el reenvío (forward) del paquete. El punto de entrada para el envío de paquetes MPLS es la función mpls_output ().

La función mpls_output() recibe como argumento un puntero al skb que debe ser transmitido, por lo tanto el socket debe estar pronto para ser enviado, skb->dst debe ser válido y es parte de una estructura NHLFE válida donde se encuentra toda la información de salida que se utilizará para el envío.

Lo primero que realiza esta función es chequear estas condiciones; campo skb->dst válido y perteneciente a un NHLFE. También aquí se comprueba si el skb está compartido (shared), este chequeo se realiza aquí ya que solo hay que preocuparse de esto para los paquetes que llegan desde un protocolo de capa 3 (IPv4), y justamente esta

es la función de entrada para esos paquetes. En caso de estar compartido, se clona el skb y se descarta el original, todo esto lo hace la función skb_share_check. Finalmente, luego de los chequeos se llama a la función mpls_output_shim ().

```
int mpls_output (struct sk_buff **pskb)
{
      struct sk_buff *skb = *pskb;
      struct mpls_nhlfe* nhlfe = NULL;

      if (unlikely(!skb->dst)) {
            printk("MPLS: No dst in skb\n");
            goto mpls_output_drop;
      }
      if (unlikely(skb->dst->ops->protocol != htons(ETH_P_MPLS_UC))) {
            printk("MPLS: Not a MPLS dst in skb\n");
            goto mpls_output_drop;
      }
      nhlfe = container_of(skb->dst, struct mpls_nhlfe, u.dst);
      if (unlikely(!nhlfe)) {
            printk("MPLS: unable to find NHLFE from dst\n");
            goto mpls_output_drop;
      }

      skb = skb_share_check(skb, GFP_ATOMIC);
      if (unlikely(!skb)) {
            printk("MPLS: unable to share skb\n");
            goto mpls_output_drop;
      }

      return mpls_output_shim(skb,nhlfe);

mpls_output_drop:
      kfree_skb(skb);
      return NET_XMIT_DROP;
}
```

La tarea principal de *mpls_output_shim()* es comenzar a inicializar el encabezado MPLS. Para eso recibe de *mpls_output()* el socket que se pretende enviar y una estructura *mpls_nhlfe* conteniendo toda la información relativa al envío.

Para realizar todo lo anterior se van llenando los campos de la estructura *mpls_skb_parm* con los valores por defecto, salvo TTL y BOS (aquí se hace la propagación del TTL). También se obtiene aquí el protocolo adecuado para el tipo de paquete (IPv4 ó IPv6).

En este punto se está por insertar la primera etiqueta al header, por lo tanto el bit BOS que indica la última etiqueta del stack debe estar “encendido”. Al final se llama a la función *mpls_output2 ()* que es la que hace el “trabajo grueso” en la preparación del envío.

```
int mpls_output_shim (struct sk_buff *skb, struct mpls_nhlfe *nhlfe)
{
      struct mpls_prot_driver *prot;
      int retval = 0;
      int ttl;
```

```
        prot = mpls_get_prot2(skb->protocol);
        if (unlikely(!prot)) {
                printk("MPLS: unable to find a protocol driver(%d)\n",
                        htons(skb->protocol));
                goto mpls_output_error;
        }

        ttl = 255;
        if (nhlfe->nhlfe_propagate_ttl) {
                ttl = prot->get_ttl(skb);
        }

        MPLSCB(skb)->prot = mpls_hold_prot(prot);
        MPLSCB(skb)->label = 0;
        MPLSCB(skb)->ttl = ttl;
        MPLSCB(skb)->exp = 0;
        MPLSCB(skb)->bos = 1;
        MPLSCB(skb)->flag = 0;
        MPLSCB(skb)->popped_bos = 0;
        MPLSCB(skb)->gap = 0;

        retval = mpls_output2(skb,nhlfe);
        mpls_put_prot(prot);
        return retval;

mpls_output_error:
        kfree_skb(skb);
        return NET_XMIT_DROP;
}
```

En la función *mpls_output2 ()* es donde se realiza la iteración a través de todas las operaciones configuradas para el NHLFE dado y finalmente una vez que se terminan de ejecutar todas las operaciones se llama a *mpls_send()*, los argumentos que recibe *mpls_output2()* son el socket_buffer a enviar y el objeto conteniendo toda la información relativa a la regla de envío pertinente.

En la explicación de la función anterior se menciona que se comienza a preparar el encabezado MPLS, en esta función se completa dicha tarea, como ejemplo se va a detallar el caso de agregar el valor de la etiqueta en el encabezado, que es el caso cuando la iteración da por resultado la operación PUSH. En ese caso las asignaciones principales son;

```
        int opcode= mi->mi_opcode (MPLS_OP_PUSH)
        void* data = mi->mi_data
```

Con estos dos datos ya se puede completar la operación, ahora la variable opcode contiene el valor MPLS_OP_PUSH y en data está el valor de la etiqueta que hay que insertar en el encabezado. Entonces ahora ***"mpls_ops[opcode].out"***, apunta a la función apropiada (***"mpls_op_push()"***), que es la que hace todo el trabajo de insertar la etiqueta y lo único que hay que proporcionarle a esta función es el valor de la etiqueta (data) y la estructura *nhlfe*.

```
        func=mpls_ops[opcode].out
              switch ( func (&skb,NULL,&nhlfe,data)) {
```

Con el valor que devuelve la función se toma la decisión de cómo continua el procesamiento, el cual puede ser volver a iterar, entregar a capa superior, Forwarding, desechar paquete ó fin del procesamiento.

Si bien la explicación hecha para PUSH es particular, aplicando un razonamiento similar se puede explicar el funcionamiento para las otras operaciones posibles.

```
int mpls_output2 (struct sk_buff *skb,struct mpls_nhlfe *nhlfe)
{



            if ((func = mpls_ops[opcode].out)) {
                  switch ( func (&skb,NULL,&nhlfe,data)) {
                        case MPLS_RESULT_RECURSE:
                        case MPLS_RESULT_DLV:
                        case MPLS_RESULT_DROP:
                              goto mpls_output2_drop;
                        case MPLS_RESULT_FWD:
                              goto mpls_output2_start;
                        case MPLS_RESULT_SUCCESS:
                              break;
```

Finalmente, la función mpls_send() se encarga de mandar el skb y asume que todo está correctamente asignado para realizar el envío, otra cosa importante es que no se libera aquí el skb (incluso si hay un error de transmisión) por lo que esa tarea la tiene que completar quien llama a esta función. Los valores de retorno son:

MPLS_RESULT_SUCCESS ó MPLS_RESULT_DROP

5.7. Interfase De Usuario

Para que el usuario pueda configurar el sistema MPLS se precisa un intérprete de comandos, aquí se presenta un detalle del código de dicho intérprete. Al contrario que el código del sistema MPLS que se encuentra dentro del Kernel de Linux, este código se compila y ejecuta en el "userspace" y están todas las funciones dentro de un mismo archivo: mpls.c (en el apéndice C)

Para explicar cómo trabaja este código vamos a ver cómo interpreta paso a paso una determinada línea de comando y va completando los datos que serán utilizados posteriormente por el algoritmo de forwarding. Ya que el procedimiento es similar para las distintas alternativas, con un ejemplo se puede ver mejor cómo funciona el intérprete, lo haremos entonces con la creación de una regla para la recepción:

Ej: mpls ilm add label gen 200 labelspace 0 instructions pop peek

Como ya se ha visto los parámetros necesarios para configurar una regla de procesamiento para los paquetes de entrada ó salida son: labelspace, valor de etiqueta y una serie de instrucciones asociadas a la regla. En el caso particular de asignar el labelspace a un dispositivo, éste último hay que indicarlo expresamente. Entonces, en el caso de entrada que vamos a ver, para almacenar estos valores se utilizan 2 estructuras: una para almacenar el labelspace y el tipo y valor de etiqueta (mpls_in_label_req) y otra para almacenar las instrucciones (mpls_instr_req) a ejecutarse.

Vamos a ver cómo se van almacenando estas estructuras con los valores dados por el usario.

5.7.1. Main()

Como toda aplicación que corre en el espacio del usuario, el código comienza a ejecutarse en la función main() dentro del archivo mpls.c, y básicamente hace ciertos chequeos de sintáxis, vuelca en pantalla información de ayuda si la entrada no es correcta, etc. A continuación de todo eso discrimina según sea el primer comando que se haya introducido (labelspace, ilm, nhlfe, etc), para seguir un procesamiento distinto de la línea introducida según lo que queramos configurar. En el caso de nuestro ejemplo es "ilm", entonces desde main() se llama a la función do_ilm () pasándole como argumentos el resto de la línea introducida. Para los otros casos se llama a funciones similares; do_ilm(), do_nhlfe(), etc.

5.7.2. Do_ilm()

En esta función lo que se hace es llamar a la función mpls_ilm_modify() con distintos argumentos según el comando sea de agregar, eliminar ó modificar (add, del, change)

5.7.3. Mpls_ilm_modify()

Aquí se hace una iteración sobre todos los parámetros restantes introducidos, en el caso de nuestro ejemplo, lo primero que encuentra es “label”, seguido de “gen 200”, cuando el parámetro es “label” lo primero que se chequea es el resto de la sintaxis, pues a continuación de “label” deben estar el tipo y el valor de etiqueta, hecha la comprobación, lo siguiente es, usando la función mpls_parse_label(), almacenar el tipo y valor de etiqueta en *mpls_in_label_req*. El parámetro siguiente es “labelspace 0” que en forma similar a lo anterior almacena el valor del labelspace asociado para esta regla también en *mpls_in_label_req*. Después de esto viene el análisis para almacenar las distintas instrucciones que se van a aplicar a esta regla de entrada, en nuestro caso el resto de la línea es “instructions pop peek”, estos valores se pasan a la función mpls_parse_instr(), ésta es la función que describiremos con más detalle por ser la más importante como se verá a continuación.

5.7.4. Mpls_parse_instr()

En esta función también se hace una iteración (con un while) pero para discernir las instrucciones que se van a aplicar en la presente regla, veamos el caso de “pop” (para cada una de las instrucciones el tratamiento es muy similar).

```
.
.
.
} else if (strcmp(*argv, "pop") == 0) {
                  if (direction == MPLS_OUT)
                        invarg(*argv, "invalid NHLFE instruction");
                  instr->mir_instr[c].mir_opcode = MPLS_OP_POP;
            } else if (strcmp(*argv, "peek") == 0) {
.
.
.
```

Aquí se ve que lo primero es verificar la consistencia de esta instrucción, en este caso no tiene sentido para una regla saliente la instrucción POP, por lo que si fuera ese el caso se rechaza la configuración. De no haber problemas, guarda el opcode asociado a POP en *mpls_instr_req* y vuelve a iterar si es que queda otro parámetro por procesar.

6. L2CC (Layer 2 Cross Connect)

6.1. Introducción

En este capítulo se describe conceptualmente la cross-conexión de capa 2 en el contexto de nuestro proyecto. El objetivo de una cross-conexión (L2CC) es realizar una interconexión fija y permanente a nivel de capa 2, esta cross-conexión se puede definir de dos formas.
La primera es entre dos tarjetas ethernet dentro de una misma máquina (ver figura 7.1).

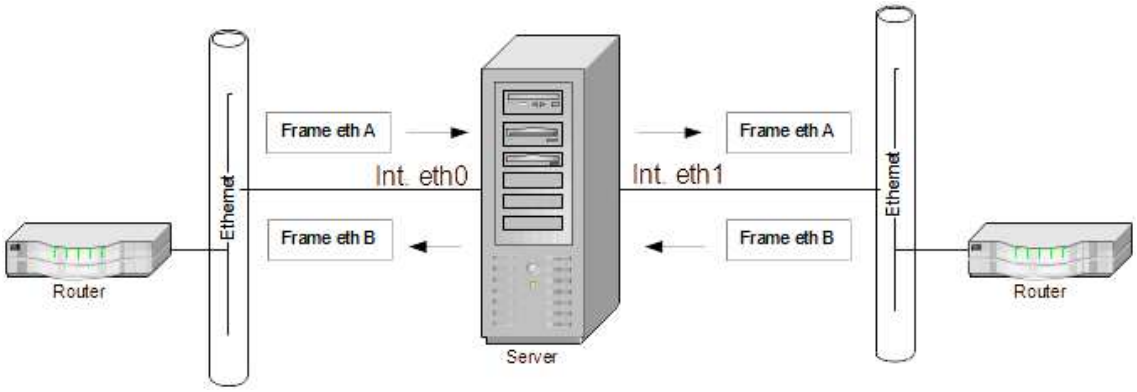

Figura 7.1

En la figura se muestra un diagrama de una cross-conexión entre la interfase eth0 y eth1. Así configurada la cross-conexión, todos los frames que entran por el dispositivo eth0, salen tal cual llegan por el dispositivo eth1 sin ser modificado y sin que intervengan ninguna de las capas superiores (en otras palabras, se saltea todo el procesamiento usual del kernel). Del mismo modo también se configura para una cross-conexión entre la interfase eth1 y eth 0 para que los frames que entran por la interfase eth1 salgan por la interfase eth0.

La segunda forma de definir una cross-conexión es entre una interfase ethernet y un LSP, esto lo utilizamos posteriormente como idea base para implementar ethernet sobre MPLS.
En la figura 7.2 se ve el diagrama donde se levantan dos LSP que permiten encapsular los frames ethernet sobre la red MPLS, uno en cada sentido.

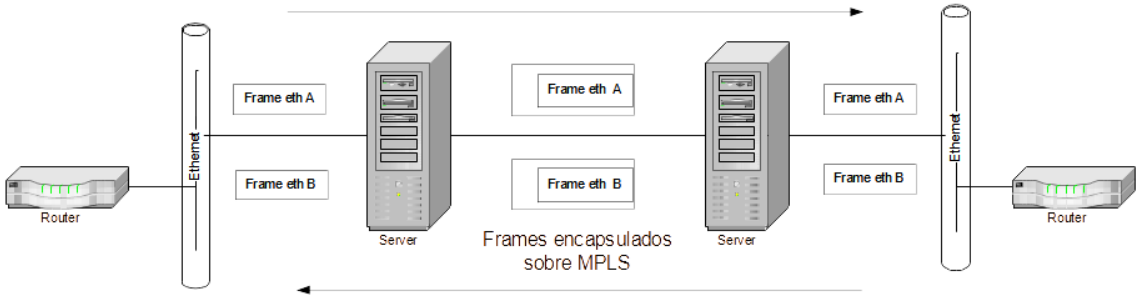

Figura 7.2

En la figura siguiente se muestra en forma esquemática los campos que contiene un frame al ser transmitido por la cross-conexión, luego de ser encapsulado.

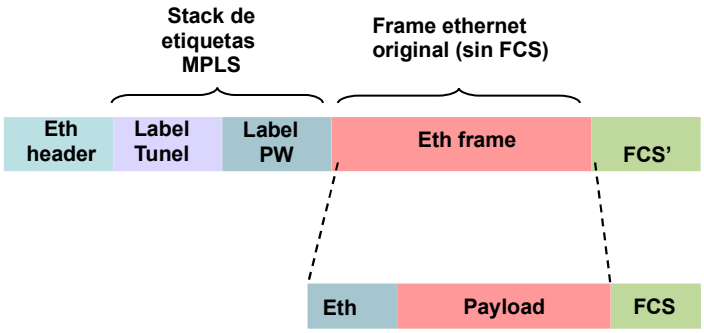

Figura 7.313 – Ehternet frame encapsulado

Se debe notar que el campo FCS en el paquete encapsulado no es el FCS original, ya que éste se elimina y se recalcula antes de enviar el paquete luego que éste fue encapsulado. El receptor debe regenerar el FCS original, luego que el frame fue desencapsulado.

6.2. Implementación del Patch L2CC

La implementación original (desarrollada por James Leu) del código estaba realizada para el kernel 2.4 y la cross-conexión a través de una red MPLS no se encontraba funcionando. A continuación describimos la estructura básica de lo que ya estaba disponible para luego en el capítulo 9 indicar las modificaciones realizadas para adaptarlo al kernel 2.6 y hacer funcionar la cross-conexión a través de una red MPLS.

Como se dijo en la introducción, la cross-conexión es a nivel de capa 2 y, por lo tanto, hay que implementar una manera de "saltear" el camino normal del socket buffer a través del Kernel de Linux, de forma de crear la conexión lógica. Además el punto y la forma en que se debe intervenir la recepción del socket_buffer van a depender de cómo el usuario haya configurado el sistema.

Para configurar una cross-conexionón entre dos interfases de un PC se usa el siguiente comando:

l2ccadm -a -i eth1 -o eth2

Los modificadores tienen el siguiente significado:

-a – Agrega una cross-conexión
-i eth1 – Define la interfase de entrada de la cross-conexión como la eth1
-o eth2 – Define la interfase de salida de la cross-conexión como la eth2.

Para que la cross-conexión quede funcionando debe agregarse en cada uno de los sentidos, es decir de eth0 a eth1 y viceversa.

6.3. L2CC - Estructuras de Datos

La estructura principal y donde se guarda la configuración relativa a la cross-conexión es una estructura del tipo ***l2cc.***

```
struct l2cc {
      int active;
      int num_instr;
      unsigned long packets;
      unsigned long bytes;
      unsigned long drop;
      unsigned long policed;
      unsigned long cn;
      struct l2cc_instr instr[8];
};
```

Como se ve, los campos más relevantes son el array (instr[]) donde se almacena el set de instrucciones a ejecutar para cada socket_buffer que interconectamos y un "flag" que indica si la interconexión está activa.

Al configurar el L2CC se crea un objeto de este tipo y se lo referencia con un puntero dentro de la estructura ***net_device*** correspondiente al dispositivo que estamos configurando para crear la interconexión, por ello se amplía la definición de esta estructura dentro del archivo netdevice.h

```
#ifdef CONFIG_L2CC
        rwlock_t        l2cc_lock;
        struct l2cc           *l2cc;
#endif
```

6.4. Instrucciones L2CC

Para almacenar las instrucciones L2CC se utiliza la siguiente estructura, la cual contiene el *opcode* asociado y una unión en que se guarda la información necesaria, por ejemplo; para el caso de una cross-conexión hacia otro dispositivo, almacenamos un puntero al dispositivo de salida (net_device), en el caso que sea un tunel MPLS, se almacena un puntero a *mpls_nhlfe* qué, como se vió en el cap. 6 contiene toda la información de salida;

```
struct l2cc_instr {
      enum l2cc_opcode opcode;
      union {
            unsigned char dstmac[6];
            struct net_device *dev_tx;
#ifdef CONFIG_MPLS
            struct mpls_nhlfe *mpls_tx;
#endif
      } u;
};
```

Los *opcode* posibles son;

```
enum l2cc_opcode {
        L2CC_OPCODE_NOOP,
        L2CC_OPCODE_SET_DSTMAC,
        L2CC_OPCODE_DEV_TX,
        L2CC_OPCODE_MPLS_TX,
};
```

Recordamos que en el procesamiento de los paquetes entrantes la función netif_rx() es la que recibe los paquetes desde los dispositivos de red y los coloca en cola de espera en el buffer de entrada, quedando de esta manera a la espera que el kernel los procese. Y la función process_backlog() es la encargada de procesar estos paquetes a medida que el CPU asigna recursos y para eso lo que hace básicamente es tomar el primer paquete de la cola y llamar a la función **netif_receive_skb()** (archivo *./net/core/dev.c)* que es quién entrega a las funciones de capa 3 (red) los paquetes para continuar con su procesamiento. Por lo tanto, es en esta función donde se debe interrumpir el flujo normal del procesamiento para implementar la cross-conexión.

```
      int netif_receive_skb(struct sk_buff *skb)
      {
            struct packet_type *ptype, *pt_prev;
            int ret = NET_RX_DROP;
            unsigned short type;

      #ifdef CONFIG_NETPOLL
            if (skb->dev->netpoll_rx && skb->dev->poll &&
netpoll_rx(skb)) {
                  kfree_skb(skb);
                  return NET_RX_DROP;
            }
      #endif

            if (!skb->stamp.tv_sec)
                  net_timestamp(&skb->stamp);

            skb_bond(skb);

            __get_cpu_var(netdev_rx_stat).total++;

      #ifdef CONFIG_L2CC
            if (skb->dev->l2cc) {
                  if (skb->dev->l2cc->active) {
                        l2cc_process(skb);
                  } else {
                        kfree_skb(skb);
                  }
                  return NET_RX_DROP;
            }
      #endif

            skb->h.raw = skb->nh.raw = skb->data;
            skb->mac_len = skb->nh.raw - skb->mac.raw;
```

La forma en que se interviene el flujo normal, es agregando un desvío condicional en función de si el dispositivo contiene un puntero a una estructura del tipo ***l2cc,*** la presencia del

puntero anterior indica si el dispositivo tiene asignada o no una cross-conexión L2CC. En caso afirmativo se invoca la función **l2cc_process()** dentro de *l2cc.c;*

```
void l2cc_process(struct sk_buff *skb){

      struct l2cc *l2cc = skb->dev->l2cc;
      int i;

      l2cc->packets++;
      l2cc->bytes += skb->len;

      for (i = 0;i < l2cc->num_instr;i++) {
            switch(l2cc->instr[i].opcode) {
                  case L2CC_OPCODE_NOOP:
                        break;
                  case L2CC_OPCODE_SET_DSTMAC:
                        memcpy(skb->mac.ethernet->h_dest,
                              l2cc->instr[i].u.dstmac, ETH_ALEN);
                        break;
                  case L2CC_OPCODE_DEV_TX:
                        skb->dev = l2cc->instr[i].u.dev_tx;
                        skb_push(skb, skb->dev->hard_header_len);
                        switch (dev_queue_xmit(skb)) {
                              case NET_XMIT_DROP:
                                    l2cc->drop++;
                                    break;
                              case NET_XMIT_CN:
                                    l2cc->cn++;
                                    break;
                              case NET_XMIT_POLICED:
                                    l2cc->policed++;
                                    break;
                              case NET_XMIT_SUCCESS:
                              case NET_XMIT_BYPASS:
                                    break;
                        }
                        break;
#ifdef CONFIG_MPLS
                  case L2CC_OPCODE_MPLS_TX:
                        skb_push(skb,skb->dev->hard_header_len);
                        skb->nh.raw -= skb->dev->hard_header_len;
                        skb->h.raw = NULL;
                        skb->mac.raw = NULL;
                        if (mpls_output_shim(skb,
                              l2cc->instr[i].u.mpls_tx) !=
                              NET_XMIT_SUCCESS)
                              l2cc->drops++;
                        break;
#endif
                  default:
                        break;
            }
      }
}
```

Esta función es la que hace todo el trabajo de cross-conexión en sí, para eso lo que hace es fijarse en las instrucciones definidas en la estructura del tipo ***l2cc***. Las opciones entonces son;

A. NOOP -- Ninguna operación (no hace nada).

B. SET_DSTMAC -- Cambia la dirección mac_ethernet de destino del socket_buffer recibido, no haremos una descripción aquí de esta alternativa.

C. DEV_TX – Este es el opcode para cross-conectar una interfase de entrada con otra de salida, en este caso se realizan las siguientes operaciones:

1.- Primero se cambia el dispositivo asignado al skb entrante, por el que está almacenado en la estructura l2cc, de esta manera las funciones de salida utilizaran este dispositivo para enviar el skb y queda así hecha la cross-conexión física.

2.- Se agranda el área de datos del skb moviendo el puntero *skb->data* en una cantidad igual a *dev->hard_header_length* (función skb_push()), de este modo todo el frame ethernet queda encapsulado y saldrá por el nuevo dispositivo tal cual ingreso (incluyendo el mac header original), en la figura 7.4 se destaca la línea punteada que marca el puntero después de ser movido.

3.- Por último se llama a la función dev_queue_xmit(skb) que es quién envía el socket a través del dispositivo programado.

Observación: En este punto los punteros h, nh y mac son nulos porque el kernel todavía no los inicializó (dado que el punto en que interrumpimos la recepción del socket_buffer es antes de que el kernel lo haga). De todos modos el bloque de memoria donde reside el skb contiene todos estos datos, para accederlos hay que inicializar los punteros adecuadamente, pero no es necesario para el envío del socket.

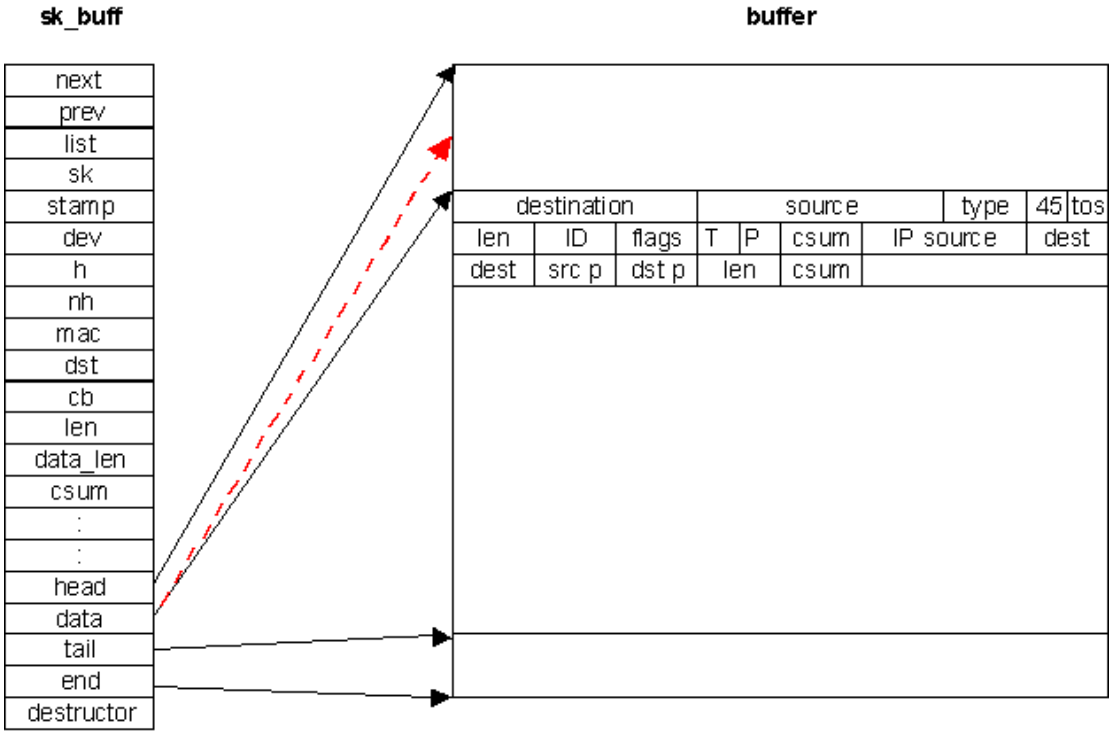

Figura 7.4

D. MPLS_TX – Este opcode es el correspondiente a la cross-conexión con un túnel MPLS y las operaciones que se hacen son;

1. Al igual que en el caso anterior, se agranda el área de datos de modo que todo el frame ethernet quede encapsulado.

2. Se mueve el puntero nh.raw (que apunta al header del protocolo de red) hacia atrás, en el mismo valor que el data pointer, ya que donde apunta data es que se coloca el encabezado MPLS.

3. Se inicializan en NULL los punteros de transporte y de capa 2 (h.raw y mac.raw)

4. Por último se envía el paquete a través de la red MPLS invocando a la función mpls_output_shim ().

7. Adaptaciones Al Sistema MPLS Para Implementar Ethernet Sobre MPLS

7.1. Introducción

Al momento de definir que nuestro proyecto apuntaba a implementar en el kernel de Linux esta funcionalidad, y tomando en cuenta los siguientes aspectos básicos:

1. El estudio que se realizó sobre el sistema MPLS.
2. En la implementación original del l2cc estaba prevista una instrucción que permitía configurar un sentido de la cross-conexión, de manera de enviar todos los frames que llegan por un dispositivo físico a un dispositivo virtual (LSP MPLS). No obstante en el sistema MPLS no estaba previsto el sentido contrario de la cross-conexión de modo de poder enviar los paquetes MPLS que llegan por el túnel al dispositivo físico correcto.

Surgieron en claro dos aspectos:

3. El ingreso de paquetes, si bien estaba previsto, no funciona.
4. No estaba previsto el egreso (no tiene previsto procesar paquetes que llegan desde la red MPLS).

> Nota: Por recepción (ingreso) entendemos los paquetes que arriban por una interfase ethernet y que saldrán encapsuladas hacia la red MPLS, y por transmisión (egreso) los paquetes que llegan desde la red MPLS y luego de ser desencapsulados se envían directamente a una interfase ethernet.

En virtud de lo anterior debíamos corregir los errores y complementar el código para la correcta recepción de las tramas e implementar en su totalidad la transmisión.

Como primer punto se debía definir sobre cuál de los códigos se iba a implementar la cross-conexión en sentido saliente.

Como mencionamos anteriormente el parche L2CC tal cual está implementado no tiene previsto el egreso, por lo que diseñar e implementar el sentido saliente sobre este código aparecía como algo más complejo que hacerlo sobre el sistema MPLS. Por otra parte y reforzando lo anterior, el sistema MPLS es quién hace el forwarding en función del valor de etiqueta del paquete entrante, por lo tanto al implementarlo en el sistema MPLS nuestro trabajo consistía en agregar una instrucción que implementara la cross-conexión de los paquetes con un cierto valor de etiqueta hacia un dispositivo dado. En definitiva nuestra meta es modificar el sistema MPLS para que;

1. Acepte un comando de la forma

mpls ilm add label gen 200 labelspace 0 instructions pop set-rx-if eth1 psw

2. Lo procese en forma adecuada logrando así la funcionalidad de encapsular ethernet sobre MPLS.

La idea del comando anterior es la siguiente:

Este comando deberá aplicar las instrucciones POP, set-rx-if y psw a cada paquete que ingrese con la etiqueta de MPLS 200 desde el labelspace 0.
El Pop elimina la etiqueta del stack (en este punto, esta debería ser la última etiqueta por lo tanto lo que obtenemos es un frame ethernet sin MPLS).
El set-rx-if modifica el dispositivo asociado al socket_buffer, esto es necesario debido a que el skb tiene asociado el dispositivo por el que ingresa y para enviarlo por otro se hace necesario modificar esa asociación. Por último psw deberá enviar el skb por el dispositivo de salida indicado en la instrucción set-rx-if.

7.1.1. Modificaciones en la interfaz de comandos (mpls.c)

Para poder ingresar el comando anterior se debe modificar la interfase de comandos para que admita un nuevo parámetro, indicando que lo que se quiere es cross-conectar una etiqueta de entrada con una interfase de salida. Nuestro nuevo parámetro será la clave “psw”, el cual debe tener un opcode asociado (MPLS_OP_PSW), la implementación del mismo la veremos en el punto posterior a este.

Como vimos en el capítulo anterior, quién interpreta los parámetros de las distintas instrucciones es la función mpls_parse_instr(), por lo tanto en esa función debemos agregar la posibilidad de admitir el nuevo nemotécnico, y se hace de manera similar a los otros:

```
} else if (strcmp(*argv, "psw") == 0) {
                  if (direction == MPLS_OUT)
                        invarg(*argv, "invalid NHLFE instruction");
                  instr->mir_instr[c].mir_opcode = MPLS_OP_PSW;
```

Con este agregado en el código de la función mpls_parse_instr(), es posible ahora configurar el sistema de manera que ejecute la nueva operación que cross-conecta paquetes MPLS etiquetados con cierto valor (y por determinada interfase de entrada) hacia una determinada interfase de salida.

7.1.2. Modificaciones hechas en el código principal para implementar Ethernet sobre MPLS.

Las modificaciones se pueden dividir en dos partes

a) Recepción
b) Transmisión

Por recepción entendemos las tramas que arriban por una interfaz ethernet y que saldrán encapsuladas hacia la red MPLS, y por transmisión los paquetes que llegan desde la red MPLS y luego de ser desencapsulados se envían directamente a una interfase ethernet.

Comenzaremos explicando las modificaciones realizadas para que funcionara la recepción. Previo a las explicaciones se presenta un diagrama de la maqueta de pruebas y su configuración.

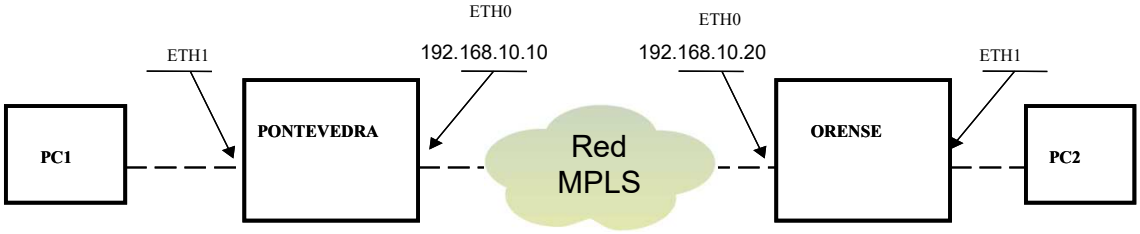

Figura 8-1

7.2. Recepción

En las primeras pruebas que se hicieron en la maqueta, el kernel se colgaba apenas llegaba el primer frame. Luego de cierto período de análisis, se llegó a la conclusión que el problema se encontraba en el código L2CC, más precisamente en las asignaciones que se hacían a los punteros del socket_buffer en cuestión, previo a pasarlo al sistema MPLS.

El problema que provocaba este error estaba en las asignaciones hechas al puntero de red. Aquí se ve el código original;

```
case L2CC_OPCODE_MPLS_TX:
                        skb_push(skb,skb->dev->hard_header_len);
                        skb->nh.raw -= skb->dev->hard_header_len;
```

```
skb->h.raw = NULL;
skb->mac.raw = NULL;
if (mpls_output_shim(skb,
        l2cc->instr[i].u.mpls_tx) !=
        NET_XMIT_SUCCESS)
        l2cc->drops++;
break;
```

Como ya se describió en el punto 7.4, los punteros a los encabezados de capa de red y de transporte no están inicializados en el lugar en el cual se interrumpe la recepción de los paquetes, y lo que la línea marcada pretende hacer es restarle una cierta cantidad a un puntero no inicializado (*nh.raw)*. Para resolver lo anterior reemplazamos la línea marcada en negrita por esta otra:

```
skb->nh.raw = skb->data;
```

Con esta asignación se resuelve entonces el problema y se deja el puntero de capa de red apuntando al lugar que se pretendía. ¿Qué se pretende con esta operación? Se agranda el área de datos (función skb_push()) del skb, moviendo el puntero skb->data en una cantidad igual a dev->hard_header_length, de este modo todo el frame ethernet queda encapsulado como payload del paquete MPLS (incluyendo el mac header), además nh.raw apunta a la misma dirección de memoria que data, que es donde la función mpls_push() va a colocar el encabezado MPLS.

Llegado a este punto encontramos nuevos problemas, cuando generábamos tráfico en la eth0 debía salir por la eth1 con la etiqueta MPLS asignada, pero esto no sucedía y los paquetes generados ni siquiera alcanzaban las funciones de envío del sistema MPLS. Durante la investigación de este problema encontramos que lo que sucedía era que lo que estábamos generando como tráfico en la eth0 era broadcast ARP y la implementación actual del sistema MPLS solamente contempla el forwarding de unicast de IPv4 e IPv6. Una vez comprendida esta limitante ingresamos las MAC de destino manualmente en la tabla ARP del PC que estábamos usando como generador de tráfico (para evitar los broadcast ARP y generar unicast) y en este momento apareció el primer paquete en la salida de la interfase eth1 ¡Eureka! En este punto el paquete no estaba encapsulado correctamente pero al menos era enviado por la interfase eth1 con la etiqueta MPLS correcta. Esto nos dejó una nueva limitante a resolver: ¿Cómo manejar los paquetes broadcast, ya que la implementación del sistema MPLS no lo contempla?

Entonces, repasando ambos códigos tenemos que el parche L2CC entrega los paquetes directamente a la función mpls_output_shim(), para lo cual debe pasarle los punteros del socket a ser enviado y la estructura ***mpls_nhlfe*** conteniendo toda la información necesaria para procesar fructíferamente el envío (el control de la consistencia de estos datos, se hace en este caso dentro del sistema L2CC).

Surge aquí una primera diferencia con el procedimiento habitual dentro del procesamiento MPLS, lo que recibe mpls_output_shim() son paquetes que llegan desde capa 3 (desde IP ó desde el propio MPLS si se trata de un Forwarding). Cuando le hacemos llegar un socket desde L2CC el mismo lo hace desde capa 2, entonces esta es la principal diferencia, tanto conceptual como práctica.

La función mpls_oputput_shim(), lo primero que hace es decidir qué “protocol driver” usar para enviar el paquete (función mpls_get_prot2()), y como se vió se

contemplan únicamente dos posibilidades (IPv4, IPv6), esto genera un problema, ya que ahora existe la posibilidad que desde L2CC le llegan paquetes ARP o eventualmente cualquier tipo de broadcast y unicast ethernet, por lo tanto estos son descartados ya que la función auxiliar no va a encontrar el "protocol driver" adecuado para los paquetes que no sean IPv4 o IPv6.

```
prot = mpls_get_prot2(skb->protocol);
       if (unlikely(!prot)) {
              printk("MPLS: unable to find a protocol driver(%d)\n",
                     htons(skb->protocol));
              goto mpls_output_error;
```

Como todos los frame ethernet deben pasar, entonces la solución que se implementó fue eludir este chequeo para el caso en que el skb proviniera desde el patch L2CC.

```
if (!(skb->dev->l2cc)) {

       prot = mpls_get_prot2(skb->protocol);
       if (unlikely(!prot)) {
              printk("MPLS: unable to find a protocol driver(%d)\n",
              htons(skb->protocol));
       goto mpls_output_error;
       }
}
```

De esta manera los skb que no llegan desde un dispositivo con el sistema L2CC activado, mantienen el procesamiento habitual y para el caso que sea desde L2CC no se realiza el chequeo y todas las tramas pueden pasar.

Este cambio solucionaba en la teoría el problema de que los broadcast no pasaban, pero ahora el kernel se volvía a "colgar" ante la aparición de un broadcast.

Volviendo a investigar el código se ve que lo que sigue a los chequeos mencionados anteriormente, es la parte en que se realiza todo lo concerniente a la propagación del TTL de los paquetes

```
ttl = 255;
       if (nhlfe->nhlfe_propagate_ttl) {
              ttl = prot->get_ttl(skb);
       }
MPLSCB(skb)->ttl = ttl;
```

El problema que ocasiona esto es que para el caso de broadcast no hay encabezado IP y por lo tanto no tienen TTL, entonces la función prot->get_ttl(skb) es en ese caso inconsistente y provoca el "cuelgue" del kernel. La solución entonces es dejar este tramo de código dentro del if que se usó para la parte anterior, de modo que esto no se ejecute para los paquetes que llegan desde el patch L2CC. El código resultante se puede ver al final de este punto.

Ahora bien, si no ejecutamos esta parte del código para los paquetes que llegan desde L2CC, no hay asignación de TTL para los paquetes MPLS generados en este caso, por lo tanto el TTL en la etiqueta MPLS quedaba con valor 0, lo cual seguramente provocaría que cualquier LSR descartara estos paquetes. ¿Qué solución aportamos? Tomando en cuenta que el valor TTL de los paquetes que ingresan al pseudo-cable, queda encapsulado al igual que toda la trama ethernet completa, y ese mismo valor será el que mantenga al egresar del pseudo cable por el otro extremo a los efectos de mantener el concepto de transparencia del pseudo-cable, la decisión fue que los

paquetes MPLS que se generan para el pseudo-cable tengan un valor TTL igual a 255. Para cumplir este objetivo la línea de asignación de la variable auxiliar ttl =255, debe estar al principio del código,

```
int mpls_output_shim (struct sk_buff *skb, struct mpls_nhlfe *nhlfe)
{
        struct mpls_prot_driver *prot;
        int retval = 0;
        int ttl=255;
        prot = mpls_get_prot2(skb->protocol);

        if (!skb->dev->l2cc) {
                if (unlikely(!prot)) {
                        printk("MPLS: unable to find a protocol driver(%d)\n",
                                htons(skb->protocol));
                        goto mpls_output_error;
                }

                if (nhlfe->nhlfe_propagate_ttl) {
                        ttl = prot->get_ttl(skb);
                        }
        }
```

de modo que esté correctamente inicializada para ambos casos. El código final resultante se puede ver en el apéndice C.

7.3. Transmisión

Una vez que se completó la recepción, restaba implementar el sentido inverso a fin de que quedara armado el "pseudo cable", como se mencionó anteriormente la idea para resolver esto es agregar una operación al procesamiento del sistema MPLS, de modo de permitir la nueva configuración.

Como vimos en la explicación de la recepción de paquetes MPLS, la función que recibe los paquetes es *mpls_skb_recv()* que, luego de ciertos chequeos obtiene del paquete entrante los valores de la etiqueta recibida y el labelspace asociado. A continuación se llama a la función *mpls_input()* que es donde se comienza el procesamiento en sí del paquete (la cual ya fue explicada en la sección **6.3.4)**. La idea central es que, una vez obtenida la regla para procesar el paquete (a partir del labelspace y del label), ejecutar en forma secuencial todas las operaciones programadas para la regla.

Entonces es aquí donde agregaremos la nueva operación, la cual optamos por llamar PSW. La explicación de cómo trabaja la función *mpls_skb_recv()* fue hecha en la sección 6.3.4, por lo que aquí presentaremos solo las partes que se agregaron y/ó modificaron, que son:

1) Un opcode para la nueva operación
2) Las funciones asociadas a este opcode
3) Un nuevo resultado posible de la ejecución de las funciones, que implique la funcionalidad del pseudo-cable.

Dentro del archivo mpls.h, que es donde está la parte más significativa de las definiciones, está la enumeración de todas las operaciones posibles.

```
enum mpls_opcode_enum {
        MPLS_OP_NOP = 0x00,
        MPLS_OP_POP,
        MPLS_OP_PEEK,
        MPLS_OP_PUSH,
        MPLS_OP_DLV,
        MPLS_OP_PSW,
        MPLS_OP_FWD,
        MPLS_OP_NF_FWD,
```

Aquí entonces agregamos el nuevo opcode: MPLS_OP_PSW. Ahora hay que agregar las funciones para procesar esta nueva operación. Lo primero es ampliar el array en que están definidos los punteros de las operaciones;

```
        [MPLS_OP_PSW] = {
               .in      = mpls_in_op_psw,
               .out     = NULL,
               .build   = mpls_build_opcode_psw,
               .unbuild = NULL,
               .cleanup = NULL,
               .extra   = 0,
               .msg     = "PSW",
```

No definimos puntero en sentido saliente, ya que sólo va a trabajar en sentido entrante, hechas estas definiciones hay que implementar las operaciones asociadas;

La creación solo debe verificar que se esté creando desde una regla de entrada la operación e iniciar la parte de datos en NULL.

```
MPLS_BUILD_OPCODE_PROTOTYPE(mpls_build_opcode_psw)
{
       *data = NULL;
       if (unlikely(direction != MPLS_IN)) {
              MPLS_DEBUG("PSW only valid for incoming labels\n");
              return -EINVAL;
       }
       *last_able = 1;
       return 0;
}
```

La función mpls_in_op_psw es simple,

```
MPLS_IN_OPCODE_PROTOTYPE(mpls_in_op_psw)
{
       return MPLS_RESULT_PSW;
}
```

En nuestro caso lo que queremos que haga la función es devolver un resultado que le indique a la función mpls_input() que el paquete es parte de un pseudo cable. Entonces el mismo debe ser desencapsulado y luego de esto transmitido directamente por la interfase de salida del pseudo cable, esto último se implementa en mpls_input(), para eso agregamos el siguiente código.

```
switch (func(&skb,ilm,&nhlfe,data)) {
                case MPLS_RESULT_RECURSE:
                        label->ml_type = MPLS_LABEL_GEN;
                        label->u.ml_gen = MPLSCB(skb)->label;
                        goto mpls_input_start;
                case MPLS_RESULT_PSW:
                        goto mpls_input_psw;
                case MPLS_RESULT_DLV:
                        goto mpls_input_dlv;
                case MPLS_RESULT_FWD:
                        goto mpls_input_fwd;
                case MPLS_RESULT_DROP:
                        goto mpls_input_drop;
                case MPLS_RESULT_SUCCESS:
                        break;

mpls_input_psw:

/************************************/
        mpls_ilm_release(ilm);
        skb->mac.raw = skb->data;
        dev_queue_xmit(skb);
        return NET_XMIT_PSW;
/************************************/
```

Primero hay que liberar la estructura *ilm* (aquí termina el procesamiento para este paquete y ya no se va a utilizar), luego se apunta el puntero de la capa 2 al comienzo del paquete (así se desencapsula el paquete original) y finalmente se invoca la función *dev_queue_xmit()* para enviar el paquete por la interfase de red.

Vamos a ver ahora el valor de retorno NET_RX_DROP.

Este fue el primer valor que utilizamos para retornar a mpls_skb_recv(), ¿Por qué este valor? Al llamar a la función *dev_queue_xmit()*, el socket buffer comienza a ser manipulado por la parte de transmisión del kernel y por tanto el procesamiento que hace MPLS sobre el mismo termina aquí y por eso el valor de retorno elegido: dar cuenta al sistema MPLS que el procesamiento del socket_buffer terminó.

Al hacer las pruebas con el código así implementado, se comprobó que el pseudo-cable funcionaba bien, pero después de pasar varios paquetes a través de él, el kernel se "colgaba". ¿Cuál era el problema? El valor de retorno. Cuando estamos devolviendo NET_RX_DROP a *mpls_skb_recv()*, le estamos indicando que hubo un error en la recepción de paquete, por lo tanto esta función "libera" memoria eliminando el socket buffer. Esto induce un error ya que al pasarle el socket a las funciones de transmisión, estas son las que deben liberarlo, una vez completado el procesamiento. Si mpls_skb_recv() libera los socket, se puede dar el caso que lo haga antes de que el kernel lo haya transmitido, debido al funcionamiento de la cola de transmisión y es aquí que se produce el error.

Repasando el código de mpls_skb_recv(), vemos que cuando invoca a mpls_input() el resultado es continuar el procesamiento de recepción ó si se encontró un error, descartar el paquete. Para nuestro propósito tenemos que agregar una tercera

opción que permita interrumpir el procesamiento de recepción sin eliminar el socket_buffer. Para eso lo primero es devolver un valor distinto a DROP ó SUCCESS, para que *mpls_skb_recv()* discrimine el caso del pseudo-cable y no libere el socket_buffer. El código resultante entonces es:

```
mpls_input_psw:

       mpls_ilm_release(ilm);
       skb->mac.raw = skb->data;
       dev_queue_xmit(skb);
       return NET_XMIT_PSW;
```

Y en la función *mpls_skb_recv()*;

```
aux = mpls_input (skb,dev,pt,&label,labelspace);
if (aux )
       goto mpls_rcv_drop;

result = dst_input(skb);

       MPLS_DEBUG("exit(%d)\n",result);
       return result;

mpls_rcv_err:
       /* increment some err counter */
mpls_rcv_drop:
       if (aux ¡= NET_XMIT_PSW )
              kfree_skb (skb);
mpls_rcv_out:
       MPLS_DEBUG("exit(DROP)\n");
       return NET_RX_DROP;
```

8. Pruebas de las funcionalidades que se agregaron

A los efectos de probar las funcionalidades de encapsulado ethernet sobre MPLS que se incorporaron a la implementación del sistema MPLS existente, se realizaron las siguientes pruebas que se describen a continuación.

En primer lugar probamos la encapsulación de un paquete icmp ping entre un router y un PC que se encontraban en los extremos del pseudo cable.

En segundo lugar probamos acceder a internet utilizando el pseudo cable, en este caso el PC utiliza el acceso a internet que dispone el router al otro lado del pseudo cable. En esta prueba veremos la encapsulación de los siguientes protocolos; DHPC, DNS, HTTP y SNMP.

Durante estas pruebas se utilizó el programa ethereal (www.ethereal.com) para poder capturar y analizar el tráfico.

8.1. Maqueta de pruebas

En la figura 9.1 se muestra la maqueta armada para las purebas a través de un pseudo cable y la Tabla 9.1 muestra los datos de cada host. En la práctica esta configuración debería ser equivalente a usar un cable cruzado entre el PC y el router en lugar de interconectarlos utilizan un pseudo cable.

Para poder ver los paquetes encapsulados intercalamos un HUB entre Pontevedra y Orense de forma tal de poder conectar un PC corriendo un SNIFFER.

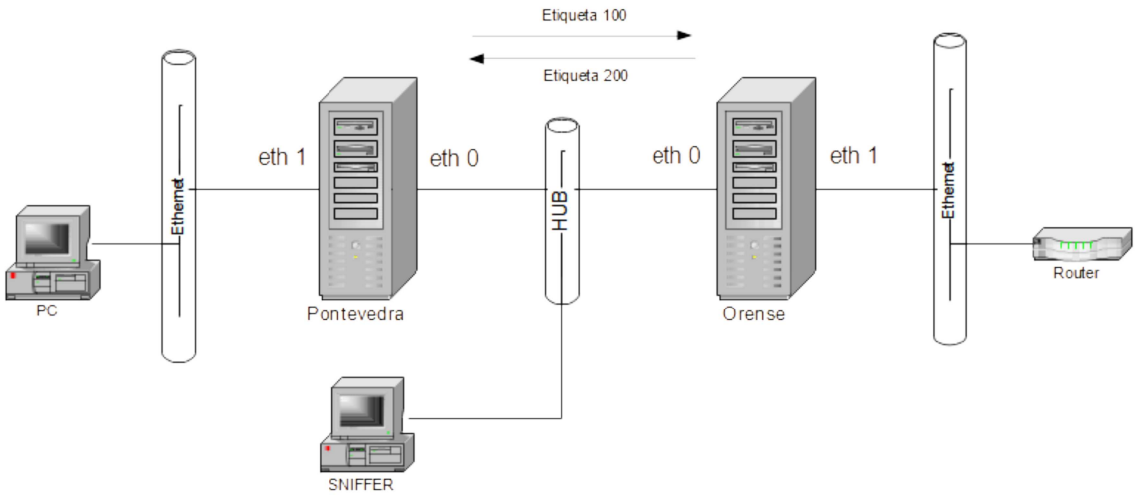

Figura 9.1

	PC	Pontevedra eht0	Orense eth0	Router
MA	00:02:3F:7B:7D:E3	00:E0:7D:94:EC:40	00:E0:4C:9C:84:5B	00:01:36:06:1F:BC

C				
IP	192.168.1.2	192.168.10.20	192.168.10.10	192.168.1.1

Tabla 9.1

Tanto el PC como el router se encuentran en la misma red IP y para ellos el pseudo cable es transparente.

La configuración del pseudo cable de Pontevedara es la siguiente:

```
mpls nhlfe add key 0 instructions push gen 100 nexthop eth0 ipv4
192.168.10.10
mpls labelspace add dev eth0 labelspace 0
mpls labelspace add dev eth1 labelspace 0
mpls ilm add label gen 200 labelspace 0 instructions pop set-rx-if
eth1 psw
l2ccadm3 -a -i eth1 -l 0x2
```

La configuración del Pseudos Cable de Orense es la siguiente:

```
mpls nhlfe add key 0 instructions push gen 200 nexthop eth0 ipv4
192.168.10.20
mpls labelspace add dev eth0 labelspace 0
mpls labelspace add dev eth1 labelspace 0
mpls ilm add label gen 100 labelspace 0 instructions pop set-rx-if
eth1 psw
l2ccadm3 -a -i eth1 -l 0x2
```

8.2. Ping

La prueba consistió en realizar un ping desde el PC hacia el router y capturar los paquetes en el PC que corre el Sniffer.

La captura realizada de los ping´s se muestra en la figura 9.2 y en las siguientes figuras se irán viendo los paquetes más significativos y su decodificación.

```
1 0.000000   192.168.10.20    Broadcast        ARP    Who has 192.168.10.10?  Tell 192.168.10.20
2 0.000080   192.168.10.10    192.168.10.20    ARP    192.168.10.10 is at 00:e0:4c:9c:84:5b
3 0.000202   192.168.1.2      Broadcast        ARP    Who has 192.168.1.1?  Tell 192.168.1.2
4 0.001459   192.168.1.1      192.168.1.2      ARP    192.168.1.1 is at 00:01:36:06:1f:bc
5 0.004058   192.168.1.2      192.168.1.1      ICMP   Echo (ping) request
6 0.005585   192.168.1.1      192.168.1.2      ICMP   Echo (ping) reply
7 1.002759   192.168.1.2      192.168.1.1      ICMP   Echo (ping) request
8 1.004283   192.168.1.1      192.168.1.2      ICMP   Echo (ping) reply
```

Figura 9.2

El paquete número uno y su decodificación se muestra en la figura 9.3.

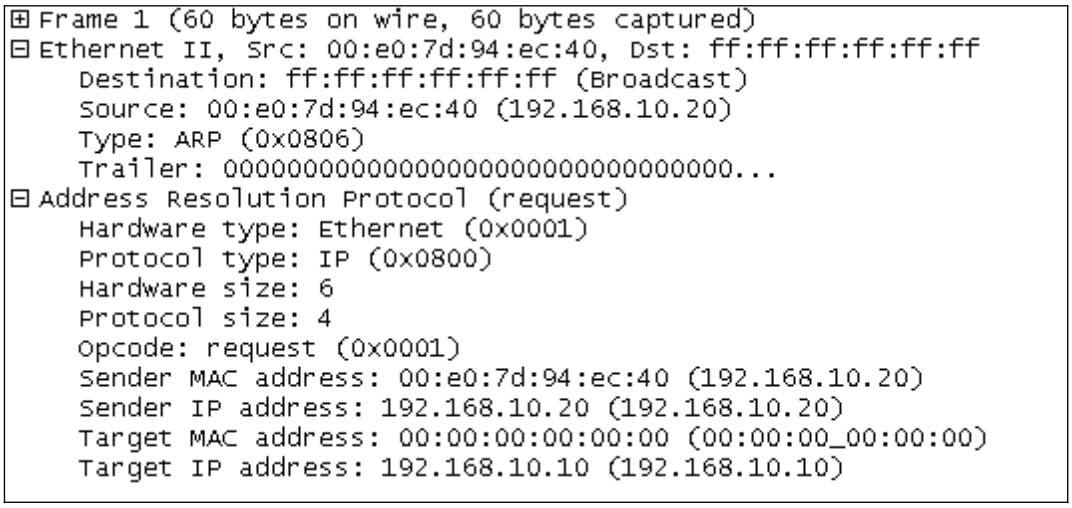

Figura 9.3

Inicialmente las tablas de ARP en todos los equipos están vacías, por lo tanto para que Pontevedra pueda enviar frames hacia Orense es necesario que resuelva su dirección MAC, entonces el primer paquete de la captura es el broadcast ARP que vemos en la figura 9.3 y el segundo será la respuesta al mismo.
Una vez obtenida esta dirección Pontevedra podrá enviar los frames encapsulados a Orense (host con IP 192.168.10.10) a través del pseudo cable.

El primer paquete generado por el PC y enviado a través del pseudo cable será el número tres de la captura (ver figura 9.2), y corresponde al mensaje ARP que se genera al ejecutar el ping en el PC hacia el host 192.168.1.1. La decodificación del mismo se ve en la figura 9.4.

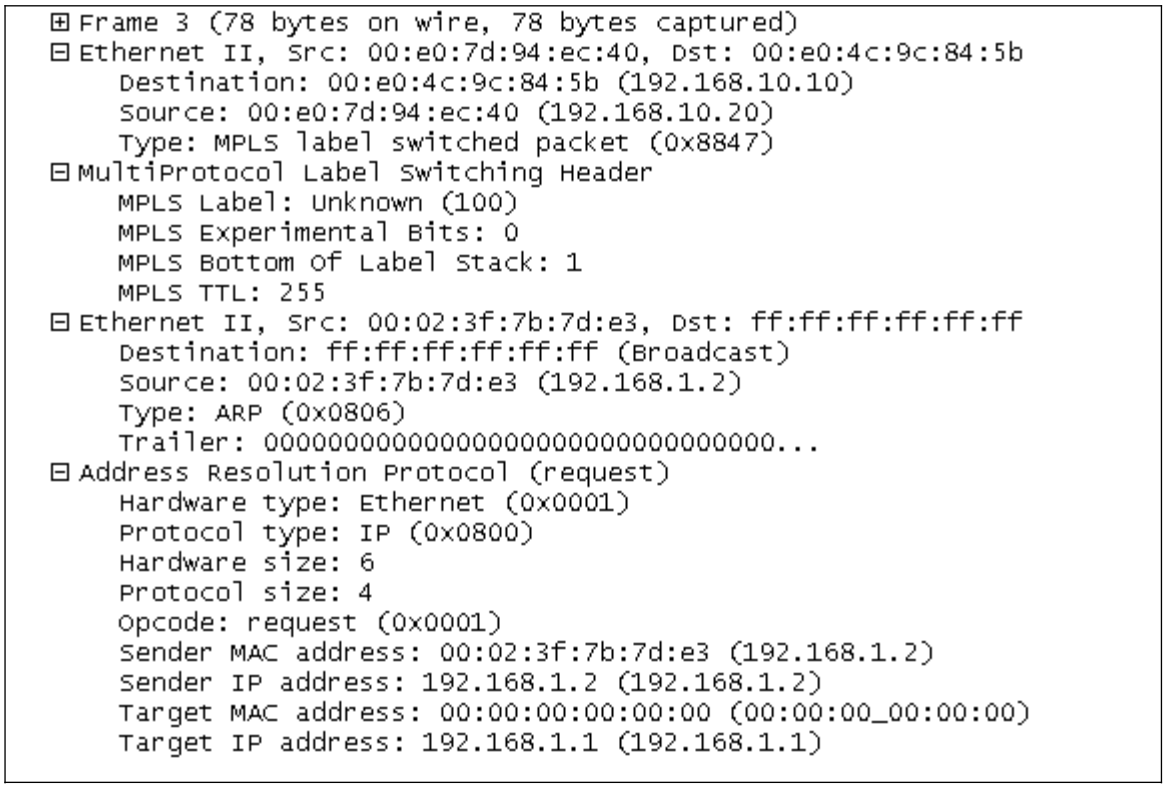

Figura 9.4

Podemos ver que el frame a nivel de capa dos es enviado por Pontevedra hacia Orense (en la tabla 9.1 se pueden ver los datos de Pontevedra y Orense a nivel de IP y capa dos), además se puede ver que el frame lleva un paquete MPLS observando el campo Type que en este caso es “MPLS label switched packet (0x8847)”.
Luego podemos ver en los campos del encabezado MPLS que el valor de la etiqueta es 100. Este valor corresponde al LSP que encapsula los frames en la dirección de Pontevedra hacia Orense.
A continuación del encabezado MPLS podemos ver el mensaje ARP original del PC totalmente encapsulado. Aquí podemos notar que el origen del mismo es la MAC address del PC (00:02:3F:7B:7D:E3) y que se busca resolver la MAC del host de destino del ping (192.168.1.1).

La decodificación del paquete número 4 se muestra en la figura 9.5.

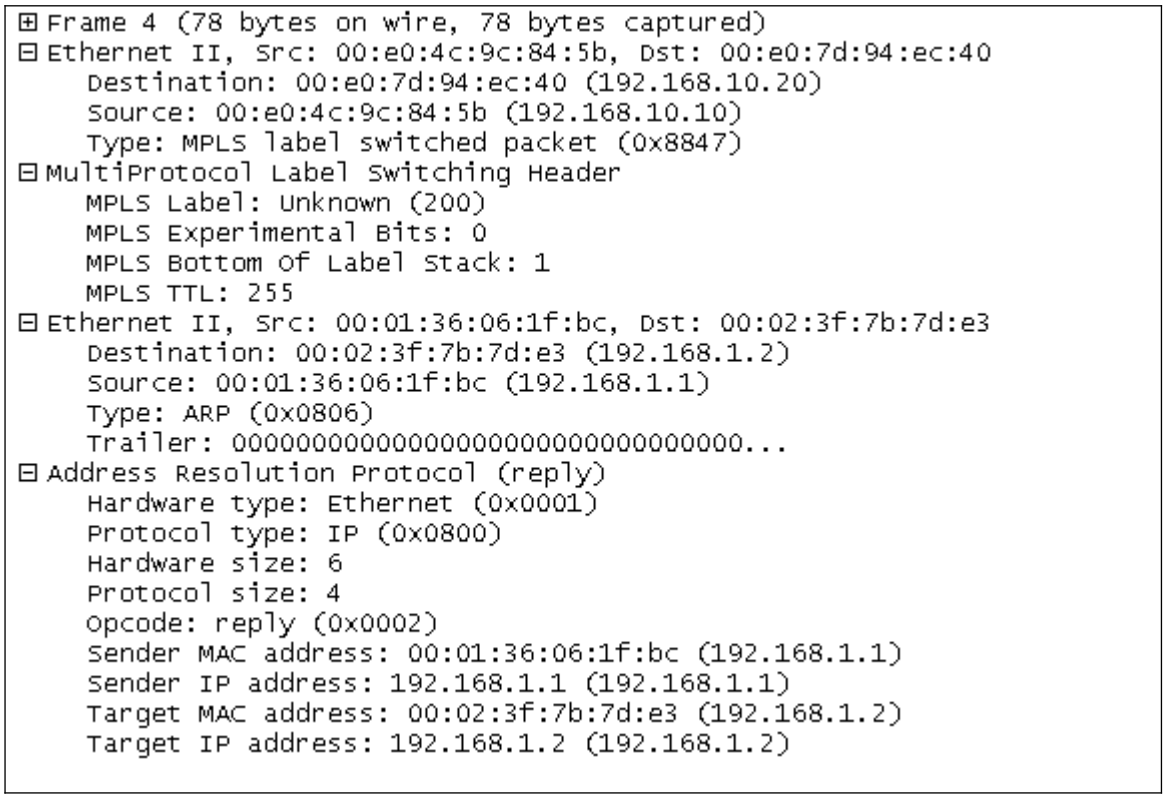

Figura 9.5

El paquete encapsulado es simplemente la respuesta del router al mensaje de ARP enviado por el PC.
A nivel de capa dos podemos observar que el origen es Orense y el destino Pontevedra y a nivel de MPLS se utiliza la etiqueta 200 correspondiente al LSP en ese sentido.

El quinto paquete se ve en la figura 9.6 y es el ping generado por el PC hacia el router encapsulado por el LSP 100. En el frame encapsulado se puede ver que el origen y el destino a nivel de capa dos y capa tres corresponden al PC y al router respectivamente.

```
⊞ Frame 5 (92 bytes on wire, 92 bytes captured)
⊟ Ethernet II, Src: 00:e0:7d:94:ec:40, Dst: 00:e0:4c:9c:84:5b
    Destination: 00:e0:4c:9c:84:5b (192.168.10.10)
    Source: 00:e0:7d:94:ec:40 (192.168.10.20)
    Type: MPLS label switched packet (0x8847)
⊟ MultiProtocol Label Switching Header
    MPLS Label: Unknown (100)
    MPLS Experimental Bits: 0
    MPLS Bottom Of Label Stack: 1
    MPLS TTL: 255
⊟ Ethernet II, Src: 00:02:3f:7b:7d:e3, Dst: 00:01:36:06:1f:bc
    Destination: 00:01:36:06:1f:bc (192.168.1.1)
    Source: 00:02:3f:7b:7d:e3 (192.168.1.2)
    Type: IP (0x0800)
⊞ Internet Protocol, Src Addr: 192.168.1.2 (192.168.1.2), Dst Addr: 192.168.1.1 (192.168.1.1)
⊟ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x305c (correct)
    Identifier: 0x0200
    Sequence number: 0x1b00
    Data (32 bytes)
```

Figura 9.6

Finalmente el último paquete decodificado se muestra en la figura 9.7 y es la respuesta del router al ping generado por el PC.

```
⊞ Frame 6 (92 bytes on wire, 92 bytes captured)
⊟ Ethernet II, Src: 00:e0:4c:9c:84:5b, Dst: 00:e0:7d:94:ec:40
    Destination: 00:e0:7d:94:ec:40 (192.168.10.20)
    Source: 00:e0:4c:9c:84:5b (192.168.10.10)
    Type: MPLS label switched packet (0x8847)
⊟ MultiProtocol Label Switching Header
    MPLS Label: Unknown (200)
    MPLS Experimental Bits: 0
    MPLS Bottom Of Label Stack: 1
    MPLS TTL: 255
⊟ Ethernet II, Src: 00:01:36:06:1f:bc, Dst: 00:02:3f:7b:7d:e3
    Destination: 00:02:3f:7b:7d:e3 (192.168.1.2)
    Source: 00:01:36:06:1f:bc (192.168.1.1)
    Type: IP (0x0800)
⊞ Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 192.168.1.2 (192.168.1.2)
⊟ Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0x385c (correct)
    Identifier: 0x0200
    Sequence number: 0x1b00
    Data (32 bytes)
```

Figura 9.7

Nuevamente observamos origen y destino a nivel de capa dos (Orense y Pontevedra respectivamente), LSP 200 y origen y destino del frame encapsulado a nivel de capa dos y tres (router y PC respectivamente).

8.3. Acceso a Internet

En esta prueba accederemos a internet desde el PC a través de la conexión que posee el router (ver figura 9.8).
El objetivo es mostrar los distintos protocolos encapsulados y su correcto funcionamiento a través del pseudo cable.
Los detalles de la encapsulación se omiten por haber sido descriptos en detalle en el ejemplo del ping.

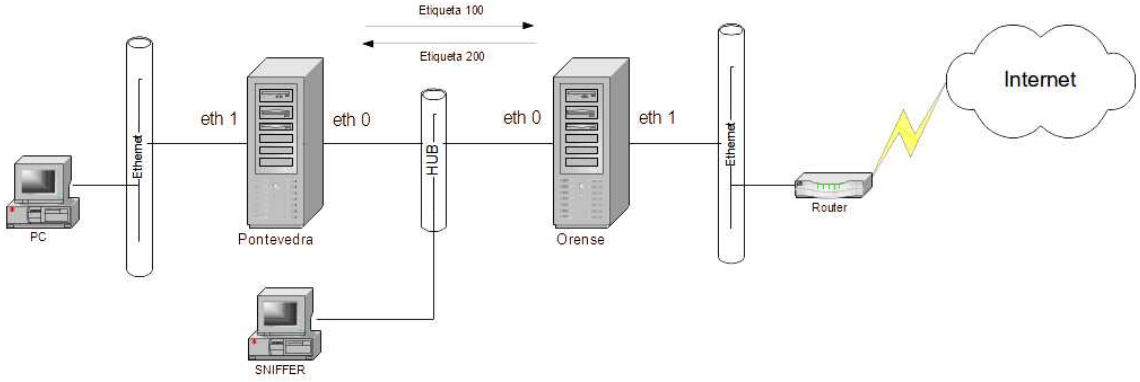

Figura 9.8

Previo a esto veremos cómo el router asigna al PC por DHCP la IP, ruta por defecto y la dirección IP del servidor DNS.

8.3.1. DHCP

En este caso configuramos el PC para que tome sus datos por DHCP (en el caso anterior estos datos se configuraron estáticamente).
En la figura 9.9 se ven los paquetes capturados correspondientes al intercambio de mensajes DHCP.

1	0.000000	0.0.0.0	255.255.255.255	DHCP	DHCP Discover - Transaction ID 0x907c6b40
2	0.004345	192.168.1.1	Broadcast	ARP	Who has 192.168.1.101? Tell 192.168.1.1
3	1.005690	192.168.1.1	255.255.255.255	DHCP	DHCP Offer - Transaction ID 0x907c6b40
4	4.014966	0.0.0.0	255.255.255.255	DHCP	DHCP Discover - Transaction ID 0x907c6b40
5	4.018178	192.168.1.1	Broadcast	ARP	Who has 192.168.1.101? Tell 192.168.1.1
6	4.997986	192.168.10.20	192.168.10.10	ARP	Who has 192.168.10.10? Tell 192.168.10.20
7	4.998061	192.168.10.10	192.168.10.20	ARP	192.168.10.10 is at 00:e0:4c:9c:84:5b
8	5.015544	192.168.1.1	255.255.255.255	DHCP	DHCP Offer - Transaction ID 0x907c6b40
9	11.024924	0.0.0.0	255.255.255.255	DHCP	DHCP Discover - Transaction ID 0x907c6b40
10	11.028111	192.168.1.1	Broadcast	ARP	Who has 192.168.1.101? Tell 192.168.1.1
11	12.025239	192.168.1.1	255.255.255.255	DHCP	DHCP Offer - Transaction ID 0x907c6b40
12	12.032166	0.0.0.0	255.255.255.255	DHCP	DHCP Request - Transaction ID 0x907c6b40
13	12.038294	192.168.1.1	255.255.255.255	DHCP	DHCP ACK - Transaction ID 0x907c6b40

Figura 9.9

En la figura 9.10 se muestra decodificación de uno de los frames DHCP encapsulado.

```
⊞ Ethernet II, Src: 00:e0:4c:9c:84:5b, Dst: 00:e0:7d:94:ec:40
⊞ MultiProtocol Label Switching Header
⊞ Ethernet II, Src: 00:01:36:06:1f:bc, Dst: ff:ff:ff:ff:ff:ff
⊞ Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 255.255.255.255 (255.255.255.255)
⊞ User Datagram Protocol, Src Port: bootps (67), Dst Port: bootpc (68)
⊟ Bootstrap Protocol
    Message type: Boot Reply (2)
    Hardware type: Ethernet
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0x907c6b40
    Seconds elapsed: 0
  ⊞ Bootp flags: 0x0000 (Unicast)
    Client IP address: 0.0.0.0 (0.0.0.0)
    Your (client) IP address: 192.168.1.101 (192.168.1.101)
    Next server IP address: 0.0.0.0 (0.0.0.0)
    Relay agent IP address: 0.0.0.0 (0.0.0.0)
    Client hardware address: 00:02:3f:7b:7d:e3
    Server host name not given
    Boot file name not given
    Magic cookie: (OK)
    Option 53: DHCP Message Type = DHCP Offer
    Option 1: Subnet Mask = 255.255.255.0
    Option 3: Router = 192.168.1.1
  ⊟ Option 6: Domain Name Server
      IP Address: 200.108.192.4
      IP Address: 200.108.192.5
    Option 15: Domain Name = "dedicado.com.uy"
    Option 51: IP Address Lease Time = 1 day
    Option 54: Server Identifier = 192.168.1.1
    End Option
    Padding
```

Figura 9.10

8.3.2. Acceso a Internet

Veremos dos casos, el primero es el acceso a un servidor web y el segundo es el acceso a un equipo con gestión snmp.

En la figura 9.11 se muestran los paquetes que intercambia el PC al acceder a través de internet a un servidor web.

```
 1 0.000000 192.168.1.101  200.108.192.4  DNS    Standard query A www.google.com
 2 0.052795 200.108.192.4  192.168.1.101  DNS    Standard query response CNAME www.l.google.com A 216.239.37.99
 3 0.056373 192.168.1.101  216.239.37.99  TCP    4355 > http [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1260
 4 0.240667 216.239.37.99  192.168.1.101  TCP    http > 4355 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 MSS=1260
 5 0.242125 192.168.1.101  216.239.37.99  TCP    4355 > http [ACK] Seq=1 Ack=1 Win=65535 Len=0
 6 0.246478 192.168.1.101  216.239.37.99  HTTP   GET / HTTP/1.1
 7 0.477741 216.239.37.99  192.168.1.101  TCP    http > 4355 [ACK] Seq=1 Ack=504 Win=7687 Len=0
 8 0.485022 216.239.37.99  192.168.1.101  TCP    [TCP Dup ACK 7#1] http > 4355 [ACK] Seq=1 Ack=504 Win=6432 Len=
 9 0.538350 216.239.37.99  192.168.1.101  HTTP   HTTP/1.1 200 OK
10 0.540099 216.239.37.99  192.168.1.101  HTTP   Continuation
11 0.547208 192.168.1.101  216.239.37.99  TCP    4355 > http [ACK] Seq=504 Ack=1511 Win=65535 Len=0
12 0.574816 192.168.1.101  216.239.37.99  HTTP   GET /intl/en/images/logo.gif HTTP/1.1
```

Figura 9.11

La figura 9.12 muestra el primer paquete decodificado donde se ve la consulta al DNS para resolver www.google.com

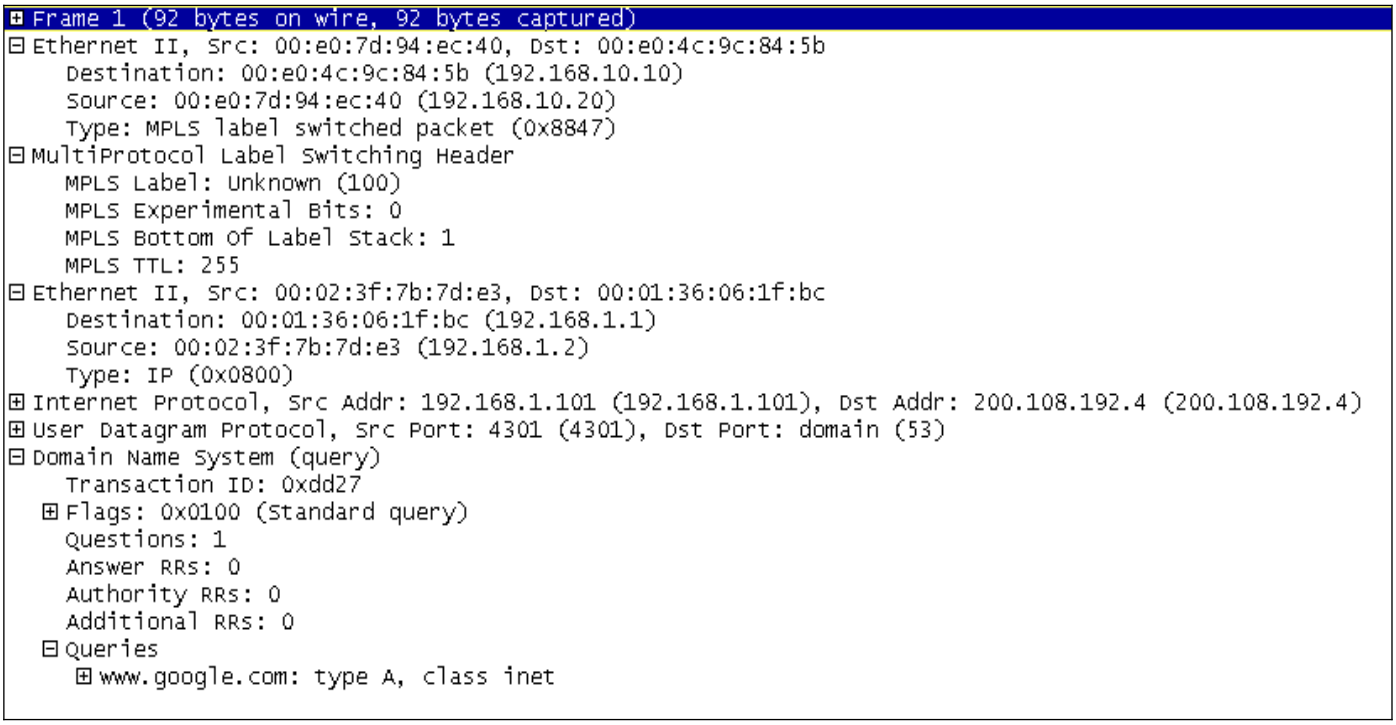

Figura 9.12

La figura 9.13 muestra la respuesta a la consulta DNS vista anteriormente.

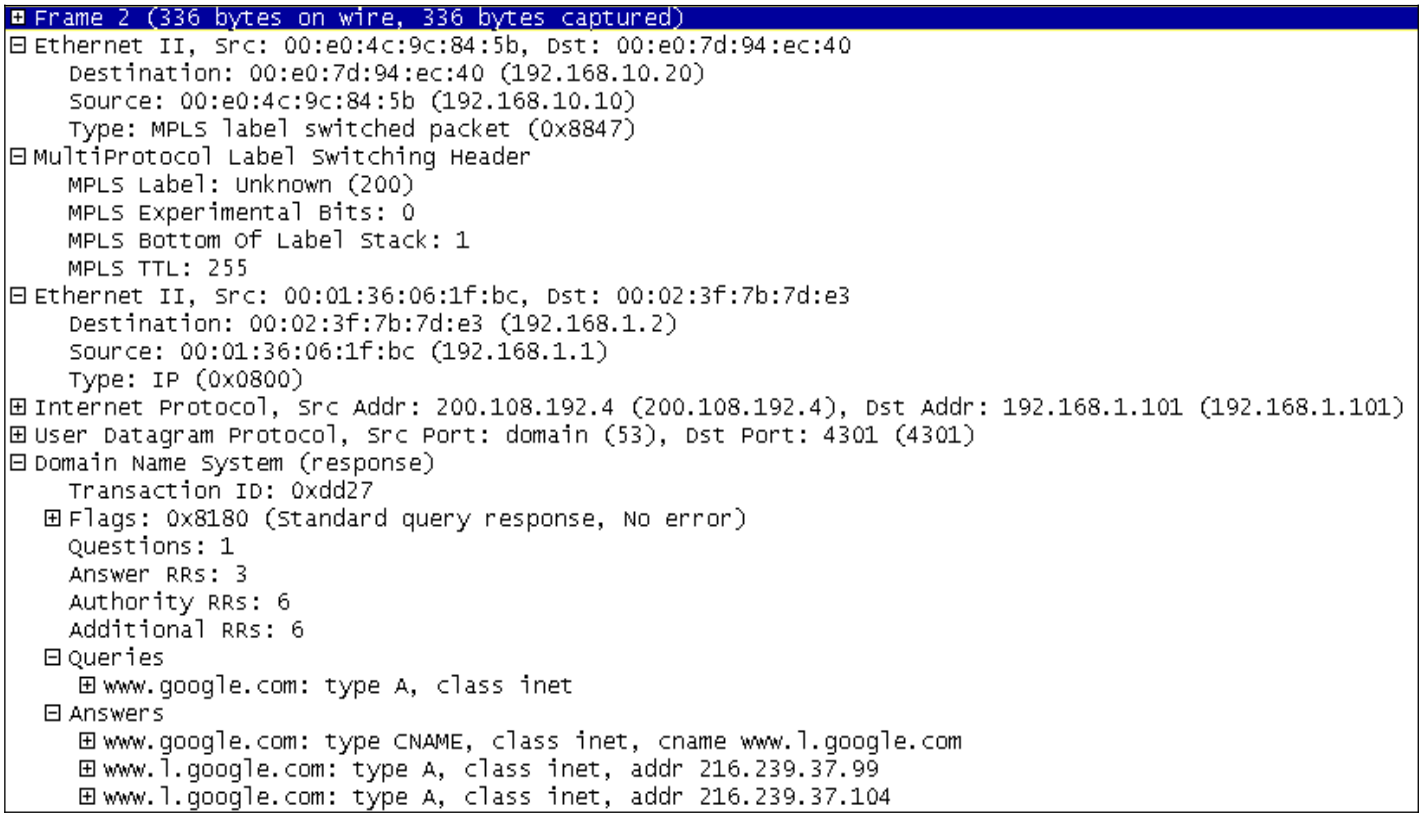

Figura 9.13

Finalmente en la figura 9.14 vemos el get a nivel de puerto 80.

```
⊞ Frame 6 (575 bytes on wire, 575 bytes captured)
⊞ Ethernet II, Src: 00:e0:7d:94:ec:40, Dst: 00:e0:4c:9c:84:5b
⊞ MultiProtocol Label Switching Header
⊞ Ethernet II, Src: 00:02:3f:7b:7d:e3, Dst: 00:01:36:06:1f:bc
⊞ Internet Protocol, Src Addr: 192.168.1.101 (192.168.1.101), Dst Addr: 216.239.37.99 (216.239.37.99)
⊞ Transmission Control Protocol, Src Port: 4355 (4355), Dst Port: http (80), Seq: 1, Ack: 1, Len: 503
⊟ Hypertext Transfer Protocol
  ⊟ GET / HTTP/1.1\r\n
      Request Method: GET
    Host: www.google.com\r\n
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.1) Gecko/20060111 Firefox/1.5.0.1\r\n
    Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5\r\n
    Accept-Language: en-us,en;q=0.5\r\n
    Accept-Encoding: gzip,deflate\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Keep-Alive: 300\r\n
    Connection: keep-alive\r\n
    Cookie: PREF=ID=d1c4ce8570209a82:LD=en:CR=2:TM=1134663312:LM=1140361167:GM=1:S=tSjAQ7Zlhjob3Q67\r\n
    \r\n
```

Figura 9.14

En la figura 9.15 vemos el acceso un equipo con gestión snmp.

```
 1 ( 192.168.1.101  200.58.142.149  SNMP   GET iso.3.6.1.2.1.1.4.0 iso.3.6.1.2.1.1.5.0 iso.3.6.1.2.1.1.6.0 iso.3.6.1.4.1.1
 2 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.2.1.1.4.0 iso.3.6.1.2.1.1.5.0 iso.3.6.1.2.1.1.6.0 iso.3.6.1.
 3 ( 192.168.1.101  200.58.142.149  SNMP   GET iso.3.6.1.4.1.12394.1.2.2.2.1.1.6.5 iso.3.6.1.4.1.12394.1.2.2.2.1.1.8.5 iso
 4 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.2.2.1.1.6.5 iso.3.6.1.4.1.12394.1.2.2.2.1.1.8.
 5 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1 iso.3.6.1.4.1.12394.1.2.6.1.1.2 iso.3.
 6 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.10.10.10.100 iso.3.6.1.4.1.12394.1.2.6
 7 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.10.10.10.100 iso.3.6.1.4.1.12394.1.2.6
 8 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.72 iso.3.6.1.4.1.12394.1.2.6
 9 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.72 iso.3.6.1.4.1.12394.1.2.6
10 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.80 iso.3.6.1.4.1.12394.1.2.6
11 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.80 iso.3.6.1.4.1.12394.1.2.6
12 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.110 iso.3.6.1.4.1.12394.1.2.
13 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.110 iso.3.6.1.4.1.12394.1.2.
14 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.114 iso.3.6.1.4.1.12394.1.2.
15 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.114 iso.3.6.1.4.1.12394.1.2.
16 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.116 iso.3.6.1.4.1.12394.1.2.
17 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.116 iso.3.6.1.4.1.12394.1.2.
18 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.118 iso.3.6.1.4.1.12394.1.2.
19 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.192.168.4.118 iso.3.6.1.4.1.12394.1.2.
20 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.1.200.108.247.88 iso.3.6.1.4.1.12394.1.2
21 ( 192.168.1.101  200.58.142.149  SNMP   GET-NEXT iso.3.6.1.4.1.12394.1.2.6.1.1.1.200.108.247.88 iso.3.6.1.4.1.12394.1.2
23 ( 200.58.142.149 192.168.1.101   SNMP   RESPONSE iso.3.6.1.4.1.12394.1.2.6.1.1.2.10.10.10.100 iso.3.6.1.4.1.12394.1.2.6
```

Figura 9.15

En la figura 9.16 y figura 9.17 vemos la parte principal de la decodificación de los primeros dos paquetes.

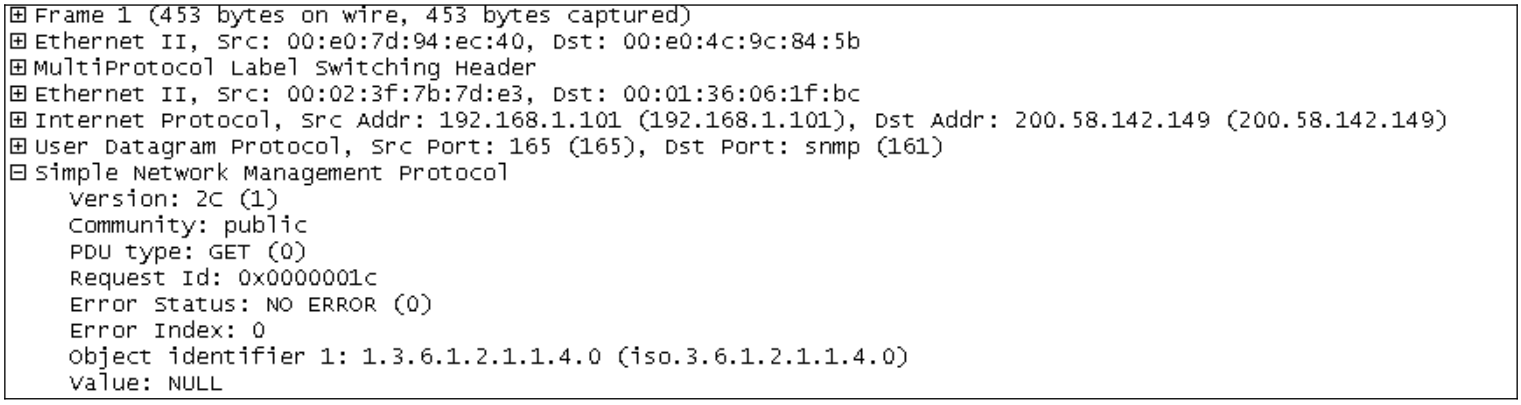

Figura 9.16

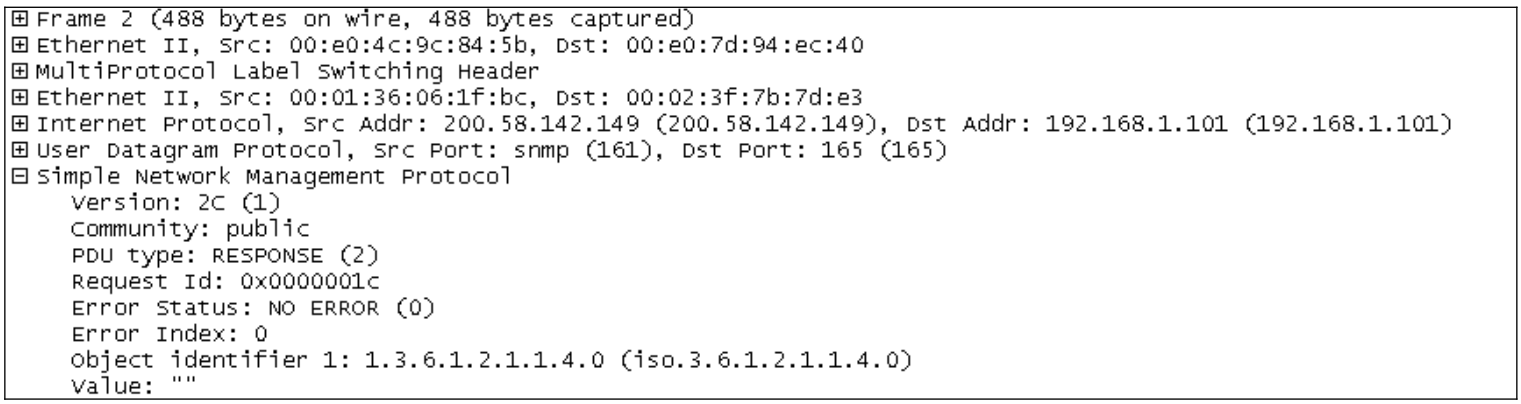

Figura 9.17

9. ¿Cómo seguir desarrollando este proyecto?

9.1. Introducción

El propósito de este capítulo es enumerar los temas que creemos que se podrían seguir desarrollando a partir de este proyecto, mencionando cuáles son las ventajas que se lograrían a partir de estos.

9.2. Funcionalidades de Pseudos cables a ser implementadas

De las funcionalidades que se definen en los pseudos cables para brindar servicios emulados de ethernet, lo que se implementó durante este proyecto es el encapsulado del frame ethernet en un paquete MPLS en el PE de entrada y la funcionalidad de desencapsular el frame ethernet en el PE de salida. Por lo tanto quedan pendientes las siguientes funcionalidades:

- Implementar la palabra de control de acuerdo a los requerimientos de la RFC4385 (Control Word for Use over an MPLS PSN): Esto permite usar el servicio simulado en casos en que se requiere una entrega ordenada de los paquetes en el receptor, esta funcionalidad se logra mediante el número de secuencia implementado en la palabra de control.

- Implementar el modo de operación “tagged”: Esto permite clasificar el tráfico en el pseudo cable de acuerdo a la VLAN. Este modo tiene una importancia general a los efectos de interoperabilidad, debido a que este es el único modo requerido en el draft “Encapsulation Methods for Transport of Ethernet Over MPLS Networks” y es el modo de operación por defecto.

- Se debe implementar la verificación de MTU en los dos extremos PE-CE antes de activar el pseudo cable. Esto es importante porque está definido en el draft “Encapsulation Methods for Transport of Ethernet Over MPLS Networks”, además es bastante común encontrar problemas de incompatibilidad de MTU en redes que implementan VLANs y MPLS debido a que, tanto el TAG de VLAN como la etiqueta MPLS incrementan el tamaño del paquete a transmitir.

- Adicionalmente a la verificación de MTU, sería muy interesante, debido a la frecuencia con que aparecen incompatibilidades de MTU, implementar fragmentación y re-ensamblado de acuerdo al draft “PWE3 Fragmentation and Reassembly”. Esto permite que, en caso que un frame exceda el MTU de la PSN debido a las etiquetas adicionales asociadas al pseudo cable, pueda ser fragmentado y transmitido en partes para luego ser re-ensamblado en el extremo opuesto.

- Implementar QoS: Se debe considerar mapear la marca de QoS en los bits de prioridad del TAG de VLAN (802.1p) en el campo de prioridad del protocolo con el que se encapsula, por ejemplo en el campo EXP de MPLS. Para hacer esto, primero se debe implementar el modo de operación “tagged”.

- Implementar LDP extendido de manera de poder establecer y señalizar pseudos cables automáticamente. Este es un arduo trabajo ya que la extensión LDP para MPLS aún no esta completamente implementado en Linux, por lo tanto antes de comenzar a implementar las extensiones de LDP para pseudos cables tal como está definido en el draft “Pseudowire Setup and Maintenance using the Label Distribution Protocol” habría que terminar la implementación de LDP.

- Hacer más amigable las interfaces de configuración del sistema MPLS y L2CC, por ejemplo hacer una interfaz de configuración grafica.

- Mejorar los mecanismos de debugging.

9.3. Problemas conocidos en la implementación actual

Lo que presentamos aquí son los problemas detectados y no corregidos a la fecha de finalización de este proyecto.

- Al desconectar la interfase que está del lado de la red MPLS, cuando el pseudo cable está activo, éste deja de funcionar aunque las interfases se vuelvan a conectar. La causa de este problema es desconocida.

- No se actualizan los contadores del sistema operativo en caso de descartar paquetes desde los pseudos cables.

- El TTL en la etiqueta MPLS se fija en 255 en el PE de entrada sin importar el valor de TTL del paquete IP. Si bien el sentido común nos hace pensar que al transportar un paquete ethernet y teniendo en cuenta que no hay TTL a nivel de ethernet, entonces esta es una implementación correcta. De todas formas, se recomienda analizar si esta implementación introduce limitaciones en alguna aplicación en particular. No se encuentran definiciones en las RFC o DRAFT asociados a los pseudos cables sobre este tema.

Glosario

API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
CE	Customer Edge
CPU	Central Processing Unit
DCE	Data Circuit Equipment
DLCI	Data Link Circuit Identifier
DMA	Direct Memory Access
DTE	Data Terminal Equipment
FEC	Forwarding Equivalence Classes
FIB	Forward Information Base
FIFO	First In First Out
FR	Frame Relay
FTN	FEC To NHLFE map
IETF	Internet Engineering Task Force
ILM	Incoming Label Map
IP	Internet Protocol
IPv4	IP versión 4
IPv6	IP versión 6
IRQ	Interrupt Request
ISR	Interrupt Service Routine
L2CC	Layer 2 Cross Connect
LAN	Local Area Network
LDP	Label Distribution Protocol
LIFO	Last In First Out
LSP	Label Switched Path
LSR	Label Switching Router
MAC	Media Access Control
MPLS	Multiprotocol Label Switchig Architecture
MTU	Maximunm Transfer Unit
NAPI	New Application Programming Interface
NAT	Network Address Translation
NHLFE	Next Hop Label Forwarding Entry
NIC	Network Interface Card
PC	Personal Computer
PE	Provider Edge
PPP	Point to Point Protocol
PSN	Packet Switched Network
PWE3	Pseudowire Emulation Edge-to-Edge workig group
PWES	Pseudo Wire End Services
QOS	Quality of Service
Rd	Router downlink
RFC	Request For Comment
RSVP	Resource-Reservation Protocol

Ru	Router uplink
SLIP	Serial Line IP
TCP	Transmission Control Protocol
TDM	Time Division Multiplexing
TTL	Time To Live
UDP	User Datagram Protocol
VPN	Virtual Private Network

Referencias

1. El lenguaje de programación C – Brian W. Kernighan
2. RFC 3031 – MPLS
3. Linux IP Networking - May 2000 - http://www.cs.unh.edu/cnrg/gherrin/linux-net.html
4. A Map of the Networking code in Linux Kernel 2.4.20 - http://www.datag.org/
5. Linux Networking Kernel – Feb 2003 - http://www.ecsl.cs.sunysb.edu/elibrary/linux/network/LinuxKernel.pdf
6. Network Protocol Stack and TCP hacking – feb 2004 - http://www.linuxgazette.com/node/8781
7. Short description abaut the Linux networt buffers (skb) – Oct 2000 - http://gnumonks.org/ftp/pub/doc/skb-doc.html
8. Informal Kernel hacking guide http://kernelnewbies.org/documents/kdoc/kernel-hacking.pdf
9. FIB Structures and lookup code - http://www.cl.cam.ac.uk/Teaching/2002/DigiComm2/sheet6.pdf
10. Linux Network documentation - http://www.ecsl.cs.sunysb.edu/elibrary/linux/network/
11. ARP Cache poisoning detection and prevention – Dec 2003 - http://www.cs.sjsu.edu/faculty/stamp/students/Silky_report.pdf
12. Linux Networking, muy buena referencia sobre NAPI – Sep 2004 - http://limnos.csrd.uiuc.edu/notes/linux-networking/core.html
13. Linux 2.6 Network Stack - http://www.spine-group.org/papers/stack/stack2.6.html
14. Linux Kernel Architecture - http://www.stillhq.com/extracted/phd/references/welte-kernel_smp.pdf
15. Link con detalles y explicación de algunas de las funciones del Kernel - http://www.cs.clemson.edu/~westall/881/notes/
16. Linux Network Stack Walkthrough (2.4.20) - http://edge.mcs.drexel.edu/GICL/people/sevy/network/Linux_network_stack_walkthrough.html
17. Linux Kernel 2.4 – Internals - http://www.tldp.org/LDP/lki/lki.pdf
18. RFC 3916 -Requirements for Pseudo-Wire Emulation Edge-to-Edge (PWE3) (http://www.ietf.org/rfc/rfc3916.txt)
19. RFC 3985 - Pseudo Wire Emulation Edge-to-Edge (PWE3) Architecture (http://www.ietf.org/rfc/rfc3985.txt)

20. http://tools.ietf.org/wg/pwe3/draft-ietf-pwe3-ethernet-encap/draft-ietf-pwe3-ethernet-encap-11.txt
21. Layer 2 VPN Architectures – Ciscopress.com (ISBN 1-58705-168-0)