Escola Politècnica Superior
de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO DE FINAL DE CARRERA

TITULO: Nuevos protocolos de encaminamiento para redes móviles Ad-Hoc

AUTOR: Javier Martinez Solera

DIRECTOR: David Remondo Bueno

FECHA: 23 de febrero de 2006

Titulo: Nuevos protocolos de encaminamiento para redes móviles Ad-Hoc

Autor: Javier Martinez Solera

Director: David Remondo Bueno

Fecha: 23 de febrero de 2006

Resumen

La comunicación a través de redes heterogéneas constituye un campo de investigación con gran interés. Dentro de éste contexto, la investigación en redes móviles Ad-hoc es una de las áreas con un mayor potencial.

Las redes Ad-hoc están constituidas por terminales inalámbricos que se comunican entre ellos sin necesidad de utilizar una infraestructura pre-existente, como sería una estación base.

El interés de la comunidad científica en éste nuevo tipo de redes, que implican implícitamente la existencia de routers móviles, es considerable y creciente. Se puede esperar que las redes Ad-hoc empiecen a proliferar a medio plazo como complemento a redes celulares (por ejemplo para cubrir zonas de alta densidad de comunicación o para prolongar la duración de las baterías de terminales inalámbricos), o bien para aplicaciones concretas que no pueden depender de infraestructura pre-existente (catástrofes, equipos en ambulancias, etc.).

Recientemente, se ha implementado en la EPSC un banco de pruebas con cinco terminales móviles y un protocolo de encaminamiento dinámico sobre IEEE 802.11.

Como continuación al proyecto anterior, el presente TFC se enfoca a la implementación de un protocolo de encaminamiento orientado al mantenimiento de la Calidad de Servicio (Quality of Service, QoS), mediante el estudio y el análisis del Kernel de Linux en la parte de capa de red, y su modificación para poder tratar la QoS.

En éste proyecto, se realiza el estudio de cómo se añade una ruta en el Kernel de Linux y de cómo trata éste la entrada o salida de un paquete. Además, se da una posible solución para el mantenimiento de QoS.

Palabras Claves: MANET, redes Ad-Hoc, AODV, QoS, Linux, Kernel.

Title: New routing protocols for movable Ad-Hoc networks

Author: Javier Martinez Solera

Director: David Remondo Bueno

Date: February, 23th 2006

Overview

The communication through heterogenous networks constitutes an interesting investigation field. Within this context, the investigation in movable ad-hoc networks is one of the greatest potential areas.

The ad-hoc networks are constituted by wireless terminals that communicate among them with no need to use a preexisting infrastructure, as it would be a station base.

The interest of the scientific community in this new type of networks, that implicitly imply the existence of movable routers, is considerable and increasing. It is possible that ad-hoc networks start to proliferate in the mid term as complement to cellular networks (for example, to cover zones with high density of communication or to prolong the duration of the batteries of wireless terminals), or for concrete applications that can not depend on preexisting infrastructure (catastrophes, equipment in ambulances, etc.).

Recently, a proving stand with five movable terminals and a dynamic routing protocol based on IEEE 802.11 have been implemented in the EPSC.

In continuation to the previous project, the present TFC focuses to the implementation of a routing protocol oriented to maintain the Quality of Service (QoS), through the study and the analysis of the Linux Kernel in the network layer, and its modification to be able to treat the QoS.

In this project, we study how to add a route in the Linux Kernel and how this treats the entrance or exit of a packet. In addition, a possible solution for the maintenance of QoS is given.

Key words: MANET, Ad-Hoc networks, AODV, QoS, Linux, Kernel.

Agradecimientos:

Me gustaría dar las gracias al director de éste proyecto, David Remondo, por su gran ayuda y orientación en todo momento para poder llevar a cabo éste trabajo. También quiero agradecer la ayuda y colaboración Jordi Martínez Martos.

Y como último, pero no menos importante, dar las gracias a mi familia y amigos por haberme apoyado, animado y aguantado en todo momento.

Javier.

INDICE

INTRODUCCIÓN

Las redes móviles Ad-hoc (MANET) son redes constituidas por nodos móviles que se pueden conectar entre ellos de forma arbitraria sin la necesidad de disponer de una infraestructura y de una administración centralizada. De esta forma, son los propios nodos quienes adoptan funciones para el mantenimiento de la red y para su autoconfiguración (por ejemplo en el encaminamiento).

La investigación de las MANET está en pleno auge. Los beneficios de éste tipo de redes, como la no necesidad de centralización, y su utilidad, como la interconexión de nodos aislados, están impulsando el desarrollo de diversas aplicaciones. Las aplicaciones multimedia, debido a sus requerimientos de ancho de banda y sobretodo de retardo, necesitan el diseño de protocolos capaces de asegurar una mínima calidad de servicio (QoS) en sus conexiones.

Éste trabajo de final de carrera es la continuación de un trabajo final de carrera anterior [1], en el cual se realizó una implementación real de una red Ad-hoc con un protocolo de encaminamiento específico, el Ad-hoc On-demand Distance Vector (AODV) [4]. Además, éste trabajo pretendía modificar el AODV para que pudiera ofrecer la posibilidad de enviar la información por multicamino según la QoS del paquete, pero se encontró con la dificultad de que el Kernel de Linux no aceptaba la QoS en sus tablas de encaminamiento, debido a que las tablas de encaminamiento no poseen una casilla para guardar la información y por la cual cosa era imposible la implementación del protocolo con QoS.

El objetivo del presente trabajo es el diseño y la implementación de un protocolo de encaminamiento que asegure QoS, a partir de la modificación de una implementación existente realizada en el proyecto anterior [1] que incorpora mecanismos para asegurar QoS. Como se verá más adelante esta implementación comportará la modificación del Kernel del sistema operativo Linux.

CAPITULO 1. REDES AD-HOC

1.1. Definición

Las redes Ad-hoc, también conocidas como MANET (*Mobile Ad-.Hoc Networks*), son redes formadas por dispositivos capaces de comunicarse entre ellos utilizando el medio radio sin la necesidad de disponer de ningún tipo de infraestructura física ni de ninguna administración centralizada.

Las redes Ad-hoc son redes muy flexibles, donde todos los dispositivos actúan como emisores y receptores y ofrecen los servicios de encaminamiento para permitir la comunicación en la red.

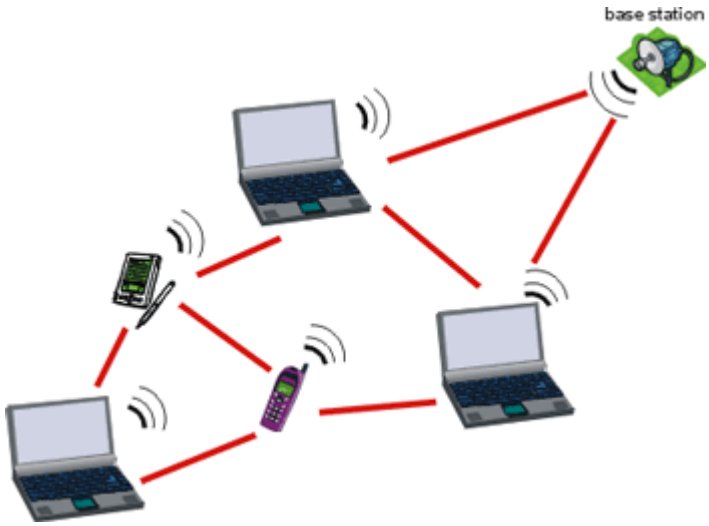


Fig. 1.1 Ejemplo de una red Ad-hoc.

Éstas redes pueden funcionar de una manera autónoma, formando redes aisladas, o algunos de sus dispositivos pueden hacer las funciones de Gateway para la conexión a otras redes tanto celulares como fijas.

1.2. Historia

Las redes móviles Ad-hoc nacen de la necesidad militar de interconectar las diferentes unidades tácticas desplegadas en zona de conflicto sin necesitar la presencia de una red fija.

Es así como los orígenes de la formación de las redes Ad-hoc se remontan a 1968 cuando se iniciaron los trabajos de la red ALOHA, cuyo objetivo era interconectar los centros educativos del archipiélago de Hawai.

La primera red Ad-hoc surgió en los años 70 cuando el Departamento de Defensa (DoD:Department of Defense) de los EEUU patrocinó la investigación de lo que en ese momento se conoció como Red de Radio Paquetes (PRNET: Packet Radio NETwork), la cual usaba una combinación de los protocolos ALOHA y CSMA como técnica de acceso al medio.

El DoD continuó apoyando las investigaciones en éste tipo de redes, hasta lograr la que actualmente es la red Radio Digital de Alcance Próximo (NTDR: Near Term Digital Radio) la cual es la única red Ad-hoc funcional de hoy día en la armada de los Estados Unidos.

Por otra parte, debido al incremento en el uso de dispositivos portátiles y móviles y a la sofisticación de los mismos, la IETF hacia mediados de los años 90 creó el grupo de trabajo MANET, buscando estandarizar los aspectos más importantes de las redes Ad-hoc para ser usados en aplicaciones comerciales.

1.3. Características

Las características más importantes a destacar de las redes Ad-hoc son las siguientes:

- **No utilizan infraestructura:** Las redes Ad-hoc no requieren de la existencia de ningún tipo de infraestructura para poder operar. Una red Ad-hoc puede tener su propio conjunto de protocolos de gestión de red, de encaminamiento, de establecimiento de comunicación e intercambio de información.

- **Topología de red dinámica:** Éste tipo de redes esta formada por nodos móviles con lo que los nodos pueden estar en movimiento, lo cual hace que la topología sea dinámica.

- **Variabilidad en el canal:** Las redes Ad-hoc presentan una gran variabilidad de las condiciones de propagación en el canal radio debido a diferentes aspectos. Algunos de estos aspectos pueden ser a la variabilidad de la energía de transmisión de los nodos, a los efectos de interferencia entre dispositivos y a la topología dinámica que presenta esta red.

- **Ancho de banda limitado:** Las redes Ad-hoc utilizan enlaces radio, los cuales presentan una capacidad más reducida que los enlaces cableados.

- **Comunicación multi-salto (multi-hop):** A diferencia de las redes convencionales, las redes Ad-hoc realizan sus comunicaciones basándose en enlaces con múltiples saltos, cada uno de ellos con diferentes condiciones de propagación radio-eléctrica a través de los diferentes nodos de la red.

- **Utilización de baterías:** Los dispositivos que están pensados para trabajar en redes Ad-hoc (alta movilidad) normalmente basan su fuente de energía en baterías, las cuales tiene una vida útil limitada.

- **Protocolos de comunicación distribuidos**: Los protocolos de una red Ad-hoc son, por definición, distribuidos (es decir, no centralizados). Éstos no dependen de una entidad central que se encargue de la gestión o administración de la red.

1.4. Utilización

En sus inicios, las redes Ad-hoc se consideraban únicamente para aplicaciones militares ya que una arquitectura de red descentralizada es una ventaja y a la vez una necesidad.

Pero poco a poco se ha ido encontrando la ventaja de usar éste tipo de redes en el ámbito comercial. Ejemplos de esto son:

- **Entorno de la ciudad:** La formación de redes Ad-hoc podría también servir como acceso inalámbrico público en zonas urbanas, proporcionando un rápido despliegue y una extensión de la cobertura de redes celulares ó WLANs. Además se pueden tener redes de taxis, comunicación en los estadios deportivos…

- **Nivel local:** A nivel local, las redes Ad-hoc que enlazan ordenadores portátiles o de mano (palmtop) podrían ser usadas para difundir y compartir información entre los participantes de una conferencia.

- **Situaciones extremas y de emergencia:** las operaciones de búsqueda y rescate en zonas remotas, o cuando la cobertura local debe ser desplegada rápidamente en un sitio de construcción.

- **Redes Domésticas:** También podrían ser apropiados para la aplicación en redes domésticas, donde los dispositivos pueden comunicarse directamente para intercambiar información como audio/vídeo, sensores, alarmas, y actualizaciones de configuración.

1.5. Protocolos de encaminamiento

Existe una gran cantidad de protocolos de encaminamiento en redes fijas, pero las técnicas que estos emplean no pueden aplicarse a las redes Ad-hoc. Los algoritmos usados en las redes cableadas presuponen que la topología de la red es poco cambiante, todo lo contrario que en las redes Ad-hoc donde la topología cambia continuamente debido a la movilidad de sus nodos.

Además, el ancho de banda y la batería de los nodos son reducidos y se saturaría muy pronto la red debido al denso tráfico de control desplegado en éste tipo de algoritmos y al rápido crecimiento de las tablas de encaminamiento.

Para intentar solventar estos aspectos, encontramos protocolos de encaminamiento específicos para redes inalámbricas. No existe ninguno que se comporte bien en todos los entornos, sino que se tendrá que escoger en función de nuestras necesidades.

1.5.1. Tipos: Proactivo vs Reactivo

En función del tipo de información que se intercambian los nodos y con la frecuencia con la que lo hacen, los protocolos de encaminamiento en redes Ad-hoc se pueden dividir en:

- **Protocolos proactivos:** En los protocolos de encaminamiento proactivos, también conocidos como con tabla de encaminamiento, todas las rutas a todos los posibles destinos se calculan a priori y, además, éstas se mantienen actualizadas y consistentes en todo momento, utilizando para ello mensajes de actualización periódicos y broadcast.

  Estos protocolos introducen cierto nivel de sobrecarga (overhead) ya que incluso cuando no se transmiten datos, es necesaria la señalización. Sin embargo, presentan la ventaja de tener una latencia pequeña y por lo tanto poder seleccionar rutas válidas de forma prácticamente inmediata.

- **Protocolos reactivos:** Los protocolos de encaminamiento reactivo, también conocido como encaminamiento bajo demanda, se basan en calcular la ruta óptima hacia un determinado destino solamente cuando es necesario.

  Estos protocolos, intentan reducir así la sobrecarga generada por los mensajes de actualización de rutas periódicos de los protocolos proactivos.

  El principal inconveniente de los protocolos reactivos es el retardo inicial que introducen ya que la ruta de X a Y se busca solo cuando X quiere mandar un paquete a Y.

  Ejemplos de protocolo reactivo serían DSR (Dynamic Source Routing) y AODV (Ad-Hoc On-Demand Distance Vector).

- **Protocolo híbrido:** Los protocolos híbridos combinan las características de los reactivos y los proactivos.

CAPITULO 2. Ad-hoc On-demand Distance Vector

2.1. ¿Qué es Ad-hoc On-demand Distance Vector?

Ad-hoc On-demand Distance Vector (AODV) es un protocolo para el encaminamiento de la información en redes móviles Ad-Hoc y es uno de los cuatro protocolos más estudiados y utilizados por la comunidad científica. Éste tipo de protocolo es capaz de encaminar paquetes tanto en modo unicast como multicast.

AODV es un protocolo de encaminamiento reactivo, es decir, solamente se establece una ruta hacia el destino si llega una petición de envío de información, y como su propio nombre indica es un protocolo basado en vector de distancia.

2.2. Funcionamiento

En éste apartado detallaremos un poco el funcionamiento de éste protocolo de encaminamiento debido a que éste proyecto no va destinado al estudio del protocolo. Para obtener más información, se puede consultar el proyecto anterior [1] o el RFC del protocolo [4].

Una de las características que define a AODV es el uso de tablas de encaminamiento en cada nodo para evitar transportar rutas en los paquetes. Cada destino de la tabla de encaminamiento lleva asociado un número de secuencia y un temporizador o *lifetime*. Éste número permite distinguir entre información nueva e información antigua, de tal manera, que se evita la formación de bucles y la transmisión de rutas antiguas o caducadas por la red. La función del temporizador es evitar usar enlaces de los que no se conoce su estado desde hace mucho tiempo.

AODV no mantiene rutas para cada nodo de la red. Estas rutas son descubiertas según se vayan necesitando. AODV es capaz de proveer de transmisión unicast, multicast y broadcast. La transmisión unicast consiste en enviar paquetes de un nodo a otro, la transmisión multicast consiste en enviar paquetes de un nodo a un grupo de nodos y la transmisión broadcast consiste en enviar paquetes de un nodo al resto de nodos de la red. Los descubrimientos de rutas son siempre bajo demanda y siguen una estructura de petición-respuesta de ruta. Las peticiones son enviadas usando un paquete especial denominado Route Request (RREQ). A su vez, las respuestas son enviadas en un paquete denominado Route Reply (RREP).

A continuación se resume la secuencia de pasos para descubrir una ruta:

- Cuando un nodo desea conocer una ruta hacia un nodo destino, envía por *broadcast* un RREQ. Un ejemplo de esto lo podemos ver en la figura 2.1.

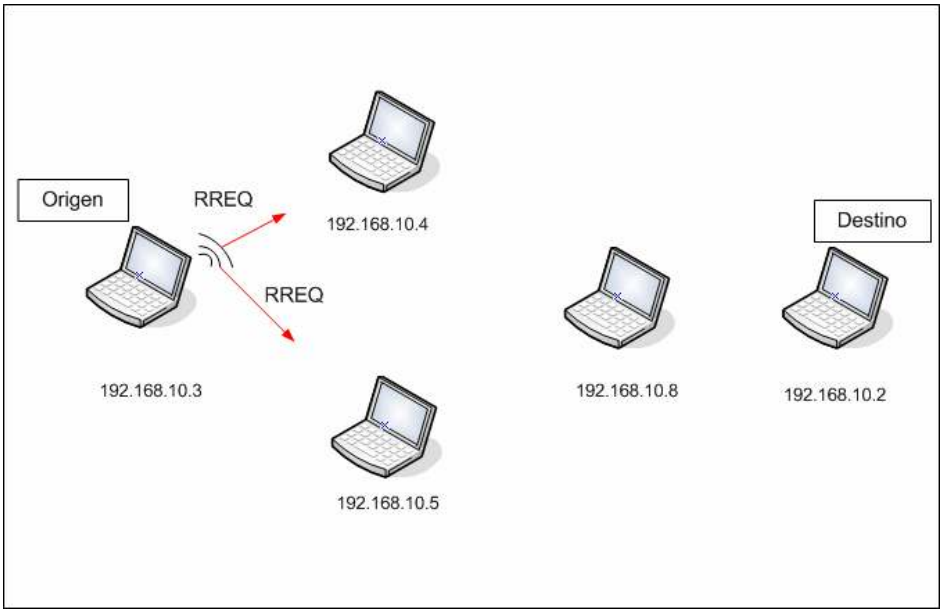


Fig. 2.1 Transmisión de paquetes RREQ en modo Broadcast.

- Cualquier nodo que conozca una ruta hacia el destino solicitado (incluido el propio destino) puede contestar enviando un RREP. Esto se puede observar en la figura 2.2.

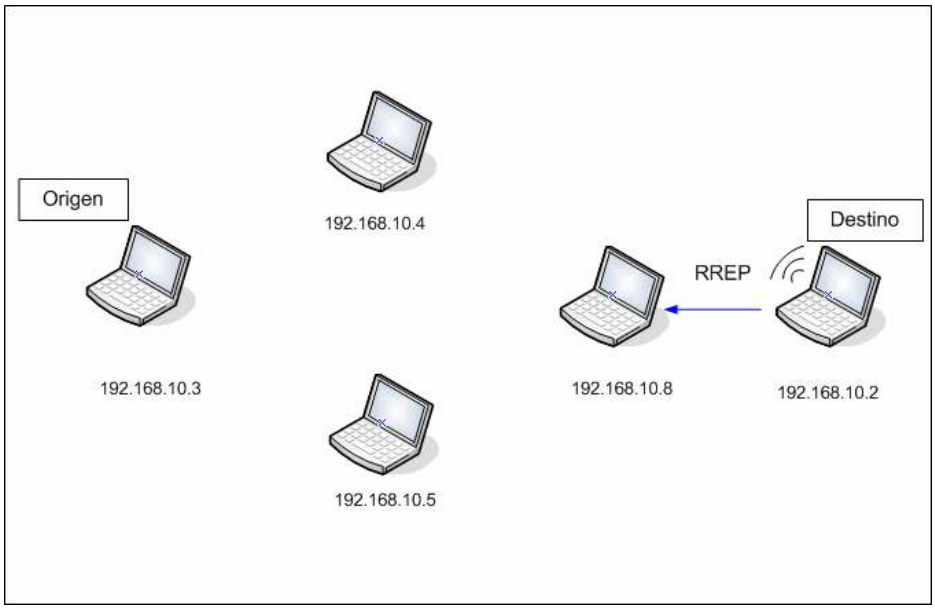


Fig. 2.2 Transmisión de un RREP de un nodo que conoce la ruta.

- Ésta información viaja de vuelta hasta el nodo que originó el RREQ y sirve para actualizar las rutas de los nodos que lo necesiten. Ver figura 2.3.

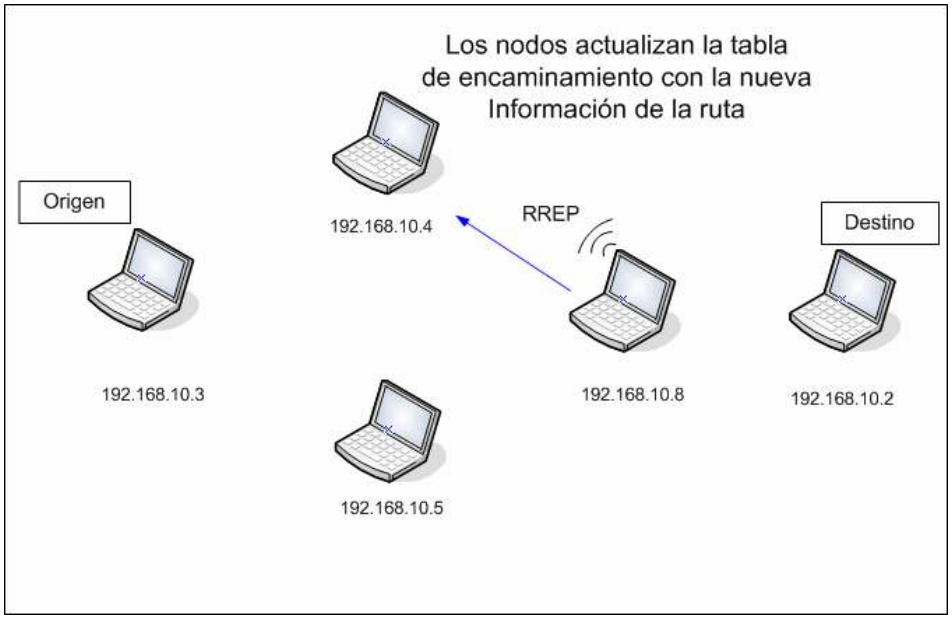


Fig. 2.3 Actualización de los nodos intermedios.

- La información recibida por el nodo destino del RREP se almacena en su tabla de encaminamiento como muestra la figura 2.4.

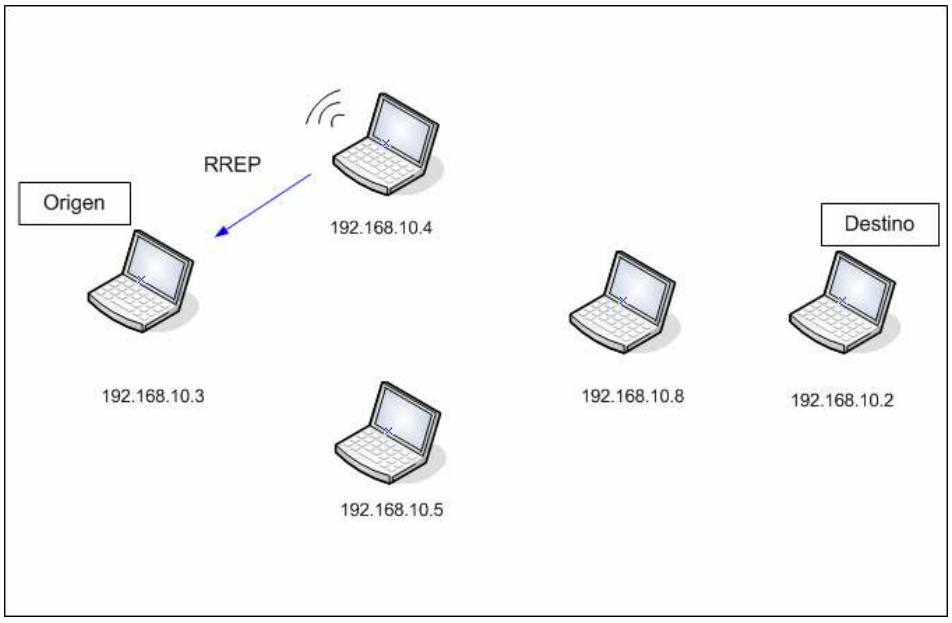


Fig. 2.4 Actualización del nodo origen del paquete RREQ.

Ahora, el nodo ya podría encaminar su paquete de datos, pues ya conoce un camino hacia su destino.

Cuando un nodo desea enviar datos a otro, primero chequea si tiene alguna entrada en su *caché* de rutas para dicho destino. Si tiene alguna entrada activa, encamina los datos por el vecino que le indica la tabla. Sin embargo, si el origen no dispone de una entrada activa, bien porque es la primera vez que se va a comunicar con él, o bien porque el plazo para ese destino ha expirado (al comprobar el campo *lifetime* y la fecha de última modificación), se inicia un descubrimiento de ruta. Para ello se debe crear un paquete RREQ que contiene información relativa al nodo destino e información propia. Cada paquete RREQ es identificado unívocamente con un identificador propio. Éste identificador se incrementa cada vez que se genere un nuevo RREQ y lo utilizan los nodos intermedios, para saber si deben retransmitir el paquete o, por el contrario, descartarlo porque ya lo retransmitieron con anterioridad. Dichos nodos, aún no siendo los destinatarios del RREQ, si mantienen una entrada para ese destino en su tabla de encaminamiento, contestarían al origen para evitar la propagación innecesaria de RREQ a través de la red. Incluso teniendo alguna entrada activa, es necesario que se cumpla que dicha ruta es más actual que la última ruta recibida por el emisor del RREQ. Aquí es donde entran en juego los números de secuencia. Cuando un nodo reenvía un RREQ, añade una ruta inversa en su tabla, que apunta al origen del RREQ (se supone enlaces simétricos). Si éste paquete llega al destinatario, éste devolverá un RREP al origen, a través del camino inverso por el que le llegó la petición.

Cuando se establece una ruta entre dos nodos, la ruta se considera válida durante un periodo de tiempo. Esto es debido a que los nodos son móviles y un camino que antes era óptimo, pasado un tiempo puede que no sea válido. Para defenderse de estas situaciones, AODV utiliza el mantenimiento de rutas. Si el nodo origen de un envío se mueve (y altera la topología de la red), él debe reiniciar un nuevo descubrimiento de ruta hacia el destino. Sin embargo, si ha sido el nodo destino de los datos el que se ha movido o algún nodo intermedio, y hay algún mensaje dirigido hacia él, un mensaje especial de error en ruta (*RERR*) será enviado al nodo que originó el envío, por el nodo que advierta el cambio en la topología de la red. Es importante resaltar que no todos los cambios de los nodos ocasionan operaciones en el protocolo, recordemos que AODV encamina bajo demanda. Todos los nodos por los que atraviese el paquete RERR, cancelarán las rutas que pasaran por el nodo que se ha vuelto inaccesible. En el momento que el paquete RERR llegue a su destino, éste puede decidir dar por terminado el envío o iniciar un nuevo RREQ si aún necesitase establecer la comunicación.

Es preciso mantener información actualizada de quiénes son los vecinos de cada nodo cada cierto tiempo. Cada vez que un nodo recibe un paquete de algún vecino, la entrada para ese vecino en la tabla de rutas se refresca, pues se sabe con seguridad que sigue en su lugar. Si no hubiera entrada todavía para el vecino, se crearía una nueva en la tabla de encaminamiento. Además, cada cierto intervalo de tiempo, se mandan paquetes *HELLO* a los vecinos para informarles que el propio nodo sigue activo. Ésta información es usada por los

vecinos para actualizar los temporizadores asociados a dicho nodo o en su defecto, para deshabilitar las entradas que se encaminen por el nodo que no responde.

2.3. La implementación original: AODV-UU

La implementación del protocolo AODV realizada por la Universidad de Upsala (AODV-UU) [2] fue la elegida en el trabajo de final de carrera anterior [1]. La implementación AODV-UU es un módulo que ha sido diseñado de tal forma que se puede instalar tanto en el sistema operativo Linux, como en el simulador Network Simulator (NS-2). El trabajo anterior, modificó éste módulo para que pudiera transportar información de la QoS requerida por las aplicaciones. Una vez modificado, se dieron cuenta de que no podían realizar la instalación por incompatibilidad, debido a que el modulo soportaba multi-camino, en cambio, el Kernel sin ninguna modificación no soporta multi-camino.

Las características más destacadas de esta implementación son:

- Funciona sobre Kernels de Linux 2.4.x y superiores y en el simulador NS-2.

- Permite crosscompilaciones en procesadores ARM para dispositivos del tipo iPAQ y Zaurus

- Cumple con el RFC3561.

- Trabaja como una aplicación de usuario, sin necesidad de modificar el Kernel de Linux. Esto es una ventaja para el estudio del protocolo porque permite monitorizar en todo momento las acciones que está llevando a cabo.

- Es sencillo de compilar, instalar y hacer funcionar.

- Acepta múltiples interfaces de red.

- Acepta las funcionalidades de Gateway aunque no estén especificadas en el RFC 3561(está en modo experimental).

- La implementación funciona correctamente sobre IPv6

- No soporta varias rutas con diferente QoS hacia un mismo destino (multi-camino).

2.4. La implementación modificada: AODV-QoS

El trabajo anterior [1] definió en detalle el módulo AODV-UU modificado para que pudiera transportar información de QoS. En ese trabajo surgió la necesidad de modificar el Kernel de Linux para que el protocolo obtuviera importantes mejoras.

El Kernel de Linux sólo permite tener guardada una ruta hacía un destino. Es decir, la tabla de encaminamiento sólo permite una ruta hacia una dirección IP destino. El trabajo anterior, destacó que la característica que diferenciaba su protocolo de los encontrados en la literatura es que el protocolo diseñado podría soportar varias rutas hacía un destino con diferentes requerimientos de QoS. Para obtener tal característica era necesaria la modificación del Kernel de Linux además de la del módulo AODV-UU.

Éste trabajo se plantea como objetivo la modificación del tratamiento de las tablas de encaminamiento del Kernel para soportar la transmisión de información por diferentes rutas dependiendo de la QoS para un mismo destino. Para ello se habrá que hacer un estudio detallado del Kernel y ver todas las posibilidades para la modificación. A más a más, también se tendrá que tener un conocimiento de cómo se comunica el módulo AODV con el Kernel para entender como éste añade las rutas en la tabla. El siguiente esquema da una idea general de lo que se pretende realizar:

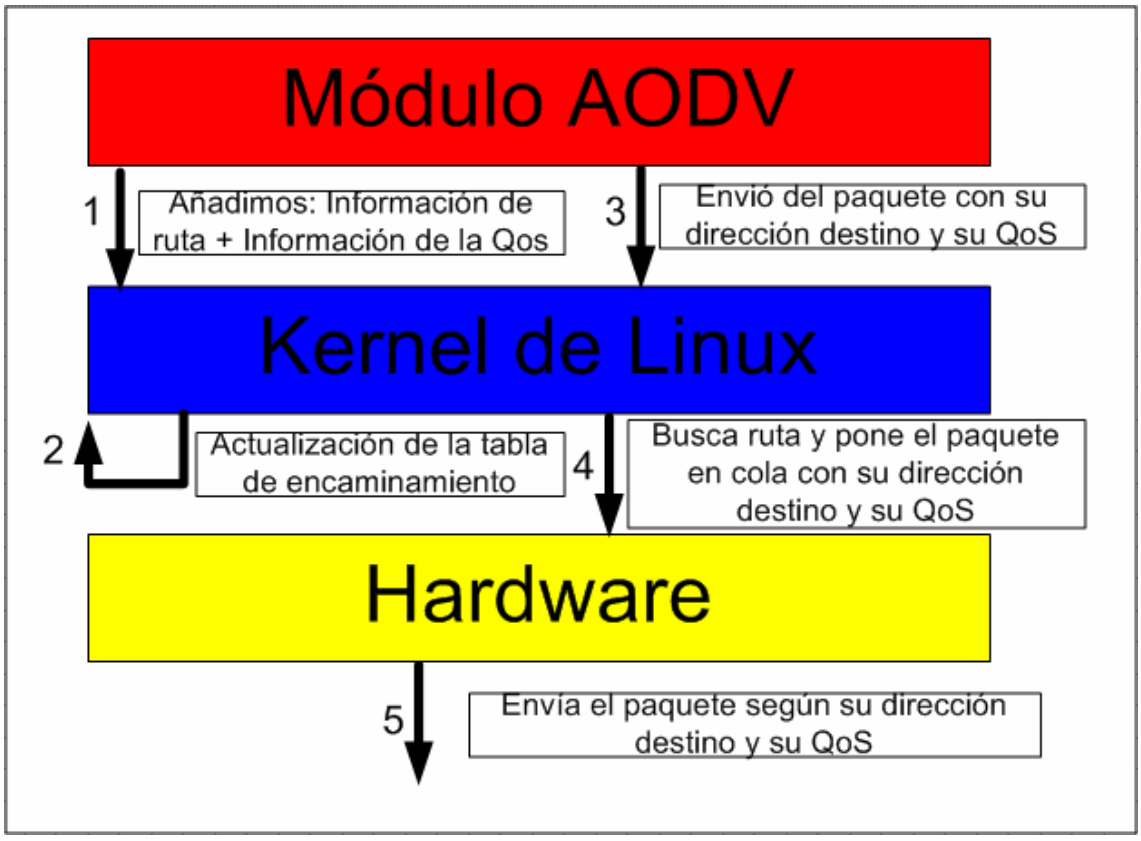


Fig. 2.5 Visión general del proyecto.

Como podemos observar en la figura 2.5, lo que se pretende hacer es que cuando el módulo sepa una ruta con su QoS, se la notifique al Kernel (paso 1) y éste la añada a su tabla de encaminamiento (paso 2). Así, cuando reciba un paquete tanto desde el módulo (paso 3) como desde el exterior, el Kernel búsque la ruta con la dirección destino y la Qos y añada el paquete a la cola de la tarjeta de red (paso 4), y ésta la transmita (paso 5).

Le hemos llamado AODV-QoS debido a que es la implementación de la universidad de Uppsala modificada para ofrecer QoS. Se mantienen todas las características anteriores excepto la característica que dice que no hace falta modificar el Kernel. En éste caso, si que hace falta modificar el Kernel debido a que se le tiene que pasar un parámetro que el Kernel sin modificar no tiene. Nosotros hemos utilizado el Kernel 2.6.9 para no tener ningún problema con la instalación de la implementación ni problemas con la instalación de los drivers de las tarjetas inalámbricas.

Capitulo 3. Kernel de Linux

3.1 ¿Qué es el Kernel de Linux?

El Kernel (también conocido como núcleo) es la parte fundamental de un sistema operativo. Es el software responsable de facilitar a los distintos programas el acceso seguro al hardware del ordenador, es decir, el encargado de gestionar recursos a través de servicios de llamada al sistema. Como hay muchos programas y el acceso al hardware es limitado, el núcleo también se encarga de decidir qué programa podrá hacer uso de un dispositivo de hardware y durante cuánto tiempo, lo que se conoce como multiplexado. Acceder al hardware directamente puede ser realmente complejo, por lo que los núcleos suelen implementar una serie de abstracciones del hardware. Esto permite esconder la complejidad, y proporciona una interfaz limpia y uniforme al hardware subyacente, lo que facilita su uso para el programador.

Hay cuatro grandes tipos de núcleos:

- Los núcleos monolíticos facilitan abstracciones del hardware subyacente realmente potentes y variadas.

- Los micronúcleos (o microKernel) proporcionan un pequeño conjunto de abstracciones simples del hardware, y usan las aplicaciones llamadas servidores para ofrecer mayor funcionalidad.

- Los híbridos (micronúcleos modificados) son muy parecidos a los micronúcleos puros, excepto porque incluyen código adicional en el espacio de núcleo para que se ejecute más rápidamente.

- Los exonúcleos no facilitan ninguna abstracción, pero permiten el uso de bibliotecas que proporcionan mayor funcionalidad gracias al acceso directo o casi directo al hardware.

3.1.1 Micronúcleos

El enfoque del micronúcleo consiste en definir una abstracción muy simple sobre el hardware, con un conjunto de primitivas o llamadas al sistema que implementan servicios del sistema operativo mínimos, como la gestión de hilos, el espacio de direccionamiento y la comunicación entre procesos.

El objetivo principal es la separación de la implementación de los servicios básicos y de la política de funcionamiento del sistema. Por ejemplo, el proceso de bloqueo de E/S se puede implementar con un servidor en espacio de usuario ejecutándose encima del micronúcleo. Estos servidores de usuario, usados para gestionar las partes de alto nivel del sistema, son muy modulares y simplifican la estructura y diseño del núcleo. Si falla uno de estos servidores, no se colgará el sistema entero, y se podrá reiniciar éste módulo independientemente del resto.

3.1.2 Núcleos monolíticos en contraposición a micronúcleos

Frecuentemente se prefieren los núcleos monolíticos frente a los micronúcleos debido al menor nivel de complejidad que comporta el tratar con todo el código de control del sistema en un solo espacio de direccionamiento. A principios de los años 1990, los núcleos monolíticos se consideraban obsoletos. El diseño de Linux como un núcleo monolítico en lugar de como un micronúcleo fue el tema de una famosa disputa entre Linus Torvalds y Andrew Tanenbaum (Ver [9]).

Los núcleos monolíticos suelen ser más fáciles de diseñar correctamente, y por lo tanto pueden crecer más rápidamente que un sistema basado en micronúcleo, pero hay casos de éxito en ambos bandos. Los micronúcleos suelen usarse en robótica embebida, ya que la mayoría de los componentes del sistema operativo residen en su propio espacio de memoria privado y protegido. Esto no sería posible con los núcleos monolíticos, ni siquiera con los modernos que permiten cargar módulos.

3.1.3 Núcleos híbridos (micronúcleos modificados)

Los núcleos híbridos fundamentalmente son micronúcleos que tienen algo de código “no esencial” en espacio de núcleo para que éste se ejecute más rápido de lo que lo haría si estuviera en espacio de usuario. Éste fue un compromiso que muchos desarrolladores de los primeros sistemas operativos con arquitectura basada en micronúcleo adoptaron antes que se demostrara que

los micronúcleos pueden tener muy buen rendimiento. La mayoría de sistemas operativos modernos pertenecen a esta categoría, siendo el más popular Microsoft Windows. XNU, el núcleo de Mac OS X, también es un micronúcleo modificado, debido a la inclusión de código del núcleo de FreeBSD en el núcleo basado en Mach. DragonFly BSD es el primer sistema BSD que adopta una arquitectura de núcleo híbrido sin basarse en Mach.
Hay gente que confunde el término «núcleo híbrido» con los núcleos monolíticos que pueden cargar módulos después del arranque, lo que es un error. «Híbrido» implica que el núcleo en cuestión usa conceptos de arquitectura o mecanismos tanto del diseño monolítico como del micronúcleo, específicamente el paso de mensajes y la migración de código «no esencial» hacia el espacio de usuario, pero manteniendo cierto código «no esencial» en el propio núcleo por razones de rendimiento.

3.1.4 Exonúcleos

Los exonúcleos, también conocidos como sistemas operativos verticalmente estructurados, representan una aproximación radicalmente nueva al diseño de sistemas operativos.

La idea es permitir que el desarrollador tome todas las decisiones relativas al rendimiento del hardware. Los exonúcleos son extremadamente pequeños, ya que limitan expresamente su funcionalidad a la protección y el multiplexado de los recursos.

Los diseños de núcleos clásicos (tanto el monolítico como el micronúcleo) abstraen el hardware, escondiendo los recursos bajo una capa de abstracción del hardware, o detrás de los controladores de dispositivo. En los sistemas clásicos, si se asigna memoria física, nadie puede estar seguro de cuál es su localización real, por ejemplo.

La finalidad de un exonúcleo es permitir a una aplicación que solicite una región específica de la memoria, un bloque de disco concreto, etc., y simplemente asegurarse que los recursos pedidos están disponibles, y que el programa tiene derecho a acceder a ellos.

Debido a que el exonúcleo sólo proporciona una interfaz al hardware de muy bajo nivel, careciendo de todas las funcionalidades de alto nivel de otros sistemas operativos, éste es complementado por una «biblioteca de sistema operativo». Ésta biblioteca se comunica con el exonúcleo, y facilita a los programadores de aplicaciones las funcionalidades que son comunes en otros sistemas operativos.

Algunas de las implicaciones teóricas de un sistema exonúcleo son que es posible tener distintos tipos de sistemas operativos (p.e. Windows, Unix)

ejecutándose en un solo exonúcleo, y que los desarrolladores pueden elegir prescindir de o incrementar funcionalidades por motivos de rendimiento.

Actualmente, los diseños exonúcleo están fundamentalmente en fase de estudio y no se usan en ningún sistema popular.

3.1.5 Código de red en el Kernel

La figura 3.1 nos muestra los directorios en los cuales se encuentra el código de red dentro del Kernel. La mayor parte del código esta situado en *net/ipv4*. El resto de código relevante se puede encontrar en *net/core* y *en net/sched*. Los archivos header (todo aquel archivo utilizado para definir estructuras y funciones, resumiendo, archivos con extensión *.h) se puede encontrar en *include/linux* y en *include/net*.

Todos los archivos include's específicos para IP se encuentran en *include/net*. En cambio, aquellos archivos que son relacionados con una parte general del Kernel o que son específicos de Linux se encuentran en *include/Linux*.

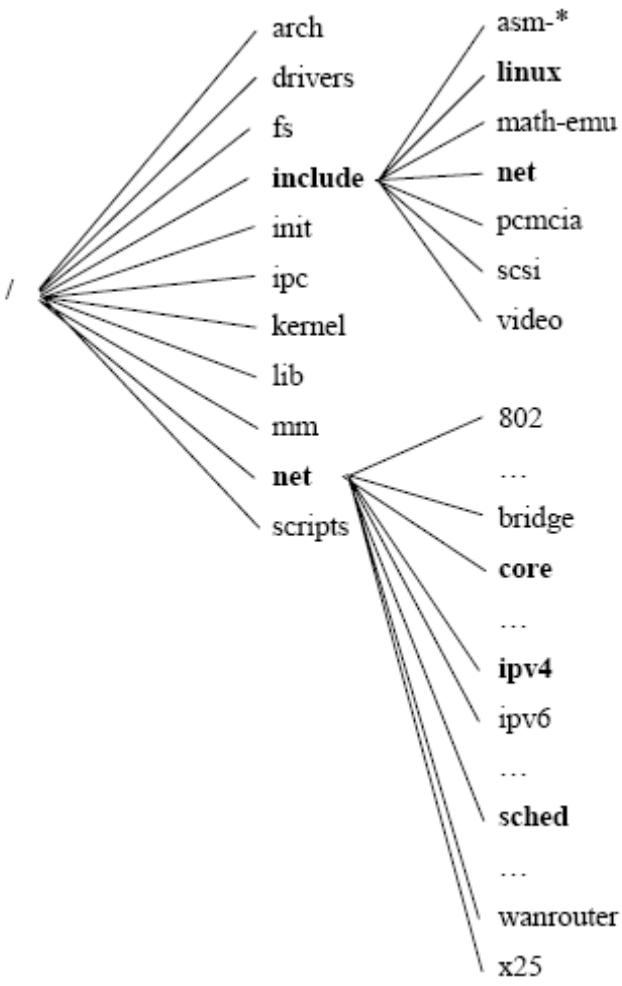


Fig. 3.1 Situación del código de red en la estructura del Kernel.

3.2 Estructura general de datos

La parte del Kernel destinada a red utiliza principalmente dos estructuras de datos: una para guardar el estado de la conexión llamada *sock* (de "socket"), y otra para guardar el estado de los paquetes de entrada y salida llamada *sk_buff* (para "socket buffer"). A continuación, son explicadas estas dos estructuras con detalle.

3.2.1 Socket Buffer (sk_buff)

La estructura de datos *sk_buff* esta definida en *include/linux/skbuff.h*.

Cuando un paquete es procesado por el Kernel, llegando del espacio de usuarios o de la tarjeta de red, se crea una de estas estructuras y el paquete es archivado en la estructura *sk_buff*. Un ejemplo de esta estructura se muestra en la Fig 3.2. En el código de red, cada función se invoca con un *sk_buff* (la variable normalmente se llama *skb*) pasada como parámetro.

Los dos primeros campos de la estructura son punteros hacia el siguiente y anterior *sk_buff* en la lista enlazada (los paquetes son guardados frecuentemente en listas enlazadas o en colas).

En el momento de llegada del paquete, éste es almacenado con un *timestamp* llamado *stamp*. El campo *dev* guarda el dispositivo por donde ha llegado el paquete, si el paquete es de entrada, el campo *dev* es actualizado y si el dispositivo es utilizado para la transmisión el campo *dev* es actualizado con el nuevo valor. Todo esto se puede observar en la figura 3.2

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff      *next;
    struct sk_buff      *prev;

    struct sk_buff_head *list;
    struct sock     *sk;
    struct timeval      stamp;
    struct net_device   *dev;
    struct net_device   *input_dev;
    struct net_device   *real_dev;
```

Fig. 3.2 Ejemplo de una parte de la estructura sk_buff.

Después de esta parte de la estructura, nos aparecen unas uniones para las capas de enlace, de red y de transporte. Un ejemplo se muestra en la Fig. 3.3. Como éste proyecto pretende centrarse en la parte de red, solo comentaremos la unión destinada a la capa de red.

```
union {
    struct iphdr    *iph;
    struct ipv6hdr  *ipv6h;
    struct arphdr   *arph;
    unsigned char   *raw;
} nh;
```

Fig. 3.3 Ejemplo de unión del sk_buff.

En la Fig. 3.4 podemos observar que existen diferentes variables, son utilizadas para almacenar la cabecera IP (dependiendo de la que lleve), debido a que el

socket buffer lo que intenta es separar la información real de la información de cabeceras. Para las capas de transporte y de enlace es similar.

Otra información sobre el paquete tal como la longitud, la longitud de los datos, el checksum, el tipo de paquete, etc es almacenado en lo que queda de estructura, tal como muestra la imagen.

```
    char          cb[48];             This field is not defined!
    unsigned int  len;                /* Length of actual data */
    unsigned int  data_len;           JP: This field is not defined!
    unsigned int  csum;               /* Checksum */
    unsigned char       __unused,     /* Dead field, may be reused */
                  cloned,             /* head may be cloned (check refcnt
to be sure) */
                  pkt_type,           /* Packet class */
                  ip_summed;          /* Driver fed us an IP checksum */
    __u32         priority;           /* Packet queueing priority */
    atomic_t      users;        /* User count - see datagram.c,tcp.c */
    unsigned short      protocol;     /* Packet protocol from driver */
    unsigned short      security;     /* Security level of packet */
    unsigned int  truesize;           /* Buffer size */
    unsigned char *head;              /* Head of buffer */
    unsigned char *data;              /* Data head pointer*/
    unsigned char *tail;              /* Tail pointer */
    unsigned char       *end;         /* End pointer */
```

Fig. 3.4 Ejemplo de la última parte de la estructura sk_buff.

3.2.2 Socket

La estructura de datos *sock* guarda los datos de una conexión TCP o de una conexión virtual UDP. Un ejemplo se muestra en la Fig. 3.5. Cuando se crea el socket en el espacio de usuario, una estructura *sock* es almacenada en el Kernel.

Los primeros campos de la estructura contienen las direcciones origen y destino, y el par de puertos (puerto origen y destino) de la conexión.

```
struct sock {
        /* Socket demultiplex comparisons on incoming packets. */
        __u32               daddr;        /* Foreign IPv4 addr             */
        __u32               rcv_saddr;    /* Bound local IPv4 addr         */
        __u16               dport;        /* Destination port              */
        unsigned short      num;          /* Local port                    */
        int                 bound_dev_if; /* Bound device index if != 0    */
```

Fig. 3.5 Ejemplo del inicio de la estructura sock.

Entre muchos otros campos, la estructura de *sock* contiene información específica del protocolo que utiliza. Estos campos contienen la información de estado sobre cada capa. Como podemos observar en la Fig. 3.6 existen varias variables para protocolo el protocolo IP (el trabajo se centra en éste nivel).

```
        union {
                struct ipv6_pinfo   af_inet6;
        } net_pinfo;

        union {
                struct tcp_opt              af_tcp;
                struct raw_opt              tp_raw4;
                struct raw6_opt             tp_raw;
                struct spx_opt              af_spx;
        } tp_pinfo;

};
```

Fig. 3.6 Ejemplo de las variables de la estructura sock.

3.3 Inicialización del módulo IP

El módulo IP es inicializado cuando *inet_init()* llama a la función *ip_init()*. Ésta función (*ip_init()*), cuyo código se muestra en la Fig. 3.7, está definida en *net/ipv4/ip_output.c* y realiza llamadas a tres funciones importantes. Una es la llamada a *dev_add_pack()*, que se utiliza para registrar paquetes Ethernet en la capa de red y recibe el controlador y así hace que los paquetes IP sean genéricos. La segunda llamada es a *ip_rt_init()*. Ésta función tiene la misión de inicializar la tabla caché y la tabla FIB dentro del Kernel. En ésta sección no explicamos las tablas debido a que posteriormente en el apartado 3.5 estas tablas se describen en detalle. Y para acabar, encontramos la función *inet_initpeers()* que inicializa la estructura AVL que es utilizada para guardar los puntos de conexión IP, es decir, hosts que actualmente esta cambiándose paquetes con nuestro host.

```
void __init ip_init(void)
{
    dev_add_pack(&ip_packet_type);

    ip_rt_init();
    inet_initpeers();

#if defined(CONFIG_IP_MULTICAST) && defined(CONFIG_PROC_FS)
    igmp_mc_proc_init();
#endif
}
```

Fig. 3.7 Código de la función ip_init().

Ahora explicaremos con detalle la función *ip_rt_init()*, debido a que es la que crea las tablas de encaminamiento y éste proyecto esta centrado en el estudio de cómo utilizar el código de encaminamiento para añadirle calidad de servicio (QoS) para proveer a la red la posibilidad de tener multi-camino para un mismo destino.

La función *ip_rt_init()* esta definida en *net/ipv4/route.c* y se muestra en la Fig. 3.8. Ésta función como ya hemos comentado se utiliza para inicializar las tablas de encaminamiento.

Linux mantiene guardadas en una tabla de hash[1] unas cuantas direcciones destino que están actualmente en uso o que han sido recientemente utilizadas.

Cada dirección IP destino activa y diferente, es descrita por una instancia de *struct rtable*. Ésta estructura es definida en *incluye/net/route.h* y más adelante será descrita con detalle (apartado 3.5 Tablas de encaminamiento).

Una vez descrita la función con más detalle, vamos analizar el código de la función. Primero, podemos observar como crea la tabla caché con una variable *struct rtable* ya almacenada.

[1] Una tabla de hash es utilizada para acelerar el proceso de búsqueda de un registro de información según una clave.

```
int __init ip_rt_init(void)
{
    int i, order, goal, rc = 0;

    rt_hash_rnd = (int) ((num_physpages ^ (num_physpages>>8)) ^
                (jiffies ^ (jiffies >> 7)));

#ifdef CONFIG_NET_CLS_ROUTE
    for (order = 0;
         (PAGE_SIZE << order) < 256 * sizeof(struct ip_rt_acct) * NR_CPUS; order++)
        /* NOTHING */;
    ip_rt_acct = (struct ip_rt_acct *)__get_free_pages(GFP_KERNEL, order);
    if (!ip_rt_acct)
        panic("IP: failed to allocate ip_rt_acct\n");
    memset(ip_rt_acct, 0, PAGE_SIZE << order);
#endif

    ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
                           sizeof(struct rtable),
                           0, SLAB_HWCACHE_ALIGN,
                           NULL, NULL);

    if (!ipv4_dst_ops.kmem_cachep)
        panic("IP: failed to allocate ip_dst_cache\n");
```

Fig. 3.8 Creación de una tabla caché.

Si no pudiera crear la tabla, surgiría un error. Una vez creada la tabla, intenta almacenarla. El tamaño de la tabla en páginas es siempre potencia de 2. El mejor tamaño esta determinado por el tamaño de memoria (variable *num_physpages*), pero el algoritmo de almacenamiento intenta guardarlo todo en una sola página cuando el almacenamiento en varias páginas no funciona. A continuación, en la Fig. 3.9 podemos ver el código encargado de toda esta operación.

```
    goal = num_physpages >> (26 - PAGE_SHIFT);
    if (rhash_entries)
        goal = (rhash_entries * sizeof(struct rt_hash_bucket)) >> PAGE_SHIFT;
    for (order = 0; (1UL << order) < goal; order++)
        /* NOTHING */;

    do {
        rt_hash_mask = (1UL << order) * PAGE_SIZE /
            sizeof(struct rt_hash_bucket);
        while (rt_hash_mask & (rt_hash_mask - 1))
            rt_hash_mask--;
        rt_hash_table = (struct rt_hash_bucket *)
            __get_free_pages(GFP_ATOMIC, order);
    } while (rt_hash_table == NULL && --order > 0);
```

Fig. 3.9 Comprobación del número de páginas en la tabla caché.

Los valores de umbral utilizados para detectar que una tabla es demasiado grande están establecidos, como podemos ver en la imagen 3.9. Típicamente,

el prefijo *gc* representa el termino garbage collection o “recolector de basura”. En cualquier caso la longitud de la tabla está limitada por el número de ranuras o slots en la tabla de hash, y el tamaño máximo de la tabla esta limitado por 16 veces el número de slots de la tabla de hash. A continuación, en la Fig. 3.10 aparece el código de la explicación.

```
ipv4_dst_ops.gc_thresh = (rt_hash_mask + 1);
ip_rt_max_size = (rt_hash_mask + 1) * 16;
```

Fig. 3.10 Definición del umbral y el número de slots.

Después de crear la tabla caché, se llama a la función *devinet_init()*. Ésta función, como observamos en la Fig. 3.11, inicializa los elementos específicos de IP de la capa física.

```
devinet_init();
ip_fib_init();

init_timer(&rt_flush_timer);
rt_flush_timer.function = rt_run_flush;
init_timer(&rt_periodic_timer);
rt_periodic_timer.function = rt_check_expire;
init_timer(&rt_secret_timer);
rt_secret_timer.function = rt_secret_rebuild;

/* All the timers, started at system startup tend
   to synchronize. Perturb it a bit.
 */
rt_periodic_timer.expires = jiffies + net_random() % ip_rt_gc_interval +
                ip_rt_gc_interval;
add_timer(&rt_periodic_timer);
```

Fig. 3.11 Código de inicialización.

Como podemos observar en la imagen, también se llama a la función *ip_fib_init()*.

La función *ip_fib_init()* es utilizada para la inicialización de la tabla FIB. Ésta función esta definida en *net/ipv4/fib_frontend.c*. Con una configuración estándar, esta función referencia a dos variables globales que son *struct fib_table *local_table* y *struct fib_table *main_table*. Estos punteros son áreas de almacenamiento dinámicas dentro de la memoria del Kernel, que contiene un tamaño fijo para la estructura fib_table seguido de una tabla de hash con entradas simples para cada uno de los posibles números de bits en la mascara de red. El contenido de la *main_table* representa las direcciones IP remotas definidas en tabla de encaminamiento y que pueden verse en */proc/net/route*.

El contenido de la *local_table* refleja las direcciones IP que existen en el mismo host.

3.4 Comunicación entre el módulo AODV-UU y el Kernel

A continuación se describe como se realiza la comunicación entre el módulo AODV-UU y el Kernel de Linux. El estudio del módulo AODV-UU nos permitirá conocer cuales son las interacciones entre el módulo y el Kernel. De éste modo, averiguaremos las modificaciones necesarias en el código del Kernel para llevar a cabo la implementación del AODV-QoS.

La implementación del módulo AODV-UU se puede dividir en varias partes: la lógica de la implementación, las funciones de envío y recepción de paquetes y por último la comunicación con el Kernel. La comunicación con el Kernel se realiza mediante la función *IOCTL()*, la cual se encuentra en el fichero del módulo AODV-UU llamado *k_route.c*. La figura 3.12 muestra el uso de la función IOCTL en el código de la implementación.

```
/* Send message by ioctl(). */
ret = ioctl(sock, SIOCADDRT, &rte);
```

Fig. 3.12 Llamada a la función IOCTL().

La función *IOCTL()* (abreviatura de **i**nput **o**utput **c**on**t**ro**l**) es la manera que tiene Linux de representar los dispositivos físicos mediante ficheros de dispositivos. La mayoría de los dispositivos físicos se utilizan para salida y para entrada, por lo tanto tiene que haber algún mecanismo para que los controladores de dispositivos que están en el núcleo obtengan la salida a enviar al dispositivo desde los procesos. Esto se hace abriendo el fichero del dispositivo para salida y escribiendo en él, igual que se escribe en un fichero. Es así como el núcleo tiene para cada dispositivo sus propias órdenes ioctl, que pueden leer ioctl's (para enviar información desde un proceso al núcleo), escribir ioctl's (para devolver información a un proceso), ambas o ninguna. La función se llama con tres parámetros: el descriptor del fichero del dispositivo apropiado, el número de ioctl, y un parámetro, que es de tipo long y al que le puedes hacer una conversión (cast) para usarlo para pasar la información necesaria. En nuestro caso, para añadir una ruta en la tabla de encaminamiento hemos de utilizar los siguientes parámetros, tal y como se muestra en la Fig. 3.12:

- **Descriptor del socket:** es el descriptor del dispositivo que queremos utilizar.

- **SIOCADDRT:** esto indica que queremos añadir la información del tercer parámetro en la tabla de encaminamiento. Si utilizáramos SIOCDELRT, indicaríamos que queremos eliminar de la tabla de encaminamiento la información que coincida con el tercer parámetro.

- **rtentry:** La estructura de rtentry nos permite añadir toda la información necesaria para hacer un encaminamiento, así cuando añadimos una ruta se copiará toda la información a esta estructura, es decir, es la estructura que permite coger llamadas de inserción/eliminación de rutas desde fuera del Kernel. La siguiente figura muestra la estructura de rtentry.

```
struct rtentry
{
    unsigned long   rt_pad1;
    struct sockaddr rt_dst;      /* target address        */
    struct sockaddr rt_gateway; /* gateway addr (RTF_GATEWAY)    */
    struct sockaddr rt_genmask; /* target network mask (IP) */
    unsigned short  rt_flags;
    short       rt_pad2;
    unsigned long   rt_pad3;
    void        *rt_pad4;
    short       rt_metric;  /* +1 for binary compatibility! */
    char __user *rt_dev;    /* forcing the device at add    */
    unsigned long   rt_mtu;     /* per route MTU/Window     */
#ifndef __KERNEL__
#define rt_mss  rt_mtu          /* Compatibility :-(            */
#endif
    unsigned long   rt_window;  /* Window clamping      */
    unsigned short  rt_irtt;    /* Initial RTT          */
};
```

Fig. 3.13 Estructura de rtentry.

La estructura de *rtentry* mostrada en la Fig. 3.13 contiene toda la información necesaria para una ruta como pueden ser la dirección destino (*rt_dst*), la máscara de red (*rt_genmask*) o la puerta de enlace (*rt_gateway*).

Una vez llamada la función *IOCTL()*, esta llama a *ip_rt_ioctl()* que se encarga de llamar a *fib_convert_rtentry()* entre otras operaciones tal y como se muestra en la Fig. 3.14.

```
err = fib_convert_rtentry(cmd, &req.nlh, &req.rtm, &rta, &r);
```

Fig. 3.14 Llamada a fib_convert_rtentry desde ip_rt_ioctl().

Ésta función (*fib_convert_rtentry()*) convierte la estructura *rtentry* en otras tres estructuras. Estas estructuras son las que más tarde las funciones que se utilizan para la tabla FIB utilizarán.

La primera estructura que encontramos es la de *kern_rta*. Ésta estructura contiene toda la información necesaria para más tarde realizar la conexión. Podemos ver la estructura en la Fig. 3.15.

```
struct kern_rta {
    void        *rta_dst;
    void        *rta_src;
    int     *rta_iif;
    int     *rta_oif;
    void        *rta_gw;
    u32     *rta_priority;
    void        *rta_prefsrc;
    struct rtattr   *rta_mx;
    struct rtattr   *rta_mp;
    unsigned char   *rta_protoinfo;
    u32     *rta_flow;
    struct rta_cacheinfo *rta_ci;
    struct rta_session *rta_sess;
};
```

Fig. 3.15 Estructura de kern_rta.

Como podemos observar en la Fig. 3.15, esta estructura guarda la dirección destino, la dirección origen, la interfaz de entrada y de salida, la dirección de la puerta de enlace, el protocolo utilizado, entre otros elementos.

Otra estructura que se utiliza para la conversión, es la estructura *nlmsghdr* (*include/net/netlink.h*). Ésta estructura también contiene información relativa a la conexión. Su estructura se muestra en la figura 3.16.

```
struct nlmsghdr
{
    __u32       nlmsg_len;  /* Length of message including header */
    __u16       nlmsg_type; /* Message content */
    __u16       nlmsg_flags;    /* Additional flags */
    __u32       nlmsg_seq;  /* Sequence number */
    __u32       nlmsg_pid;  /* Sending process PID */
};
```

Fig. 3.16 Estructura de nlmsghdr.

Como podemos ver en la Fig. 3.16, esta estructura mantiene el número de secuencia de los paquetes (nlmsg_seq), el número de identificación del proceso que ha creado la información, flags que se necesitaran, etc.

Otra estructura que utilizan las funciones de FIB es la *rtmsg* (include/Linux/rtnetlink.h). Ésta estructura sirve para guardar información referente a la administración de tablas dentro del Kernel.

```
struct rtmsg
{
    unsigned char       rtm_family;
    unsigned char       rtm_dst_len;
    unsigned char       rtm_src_len;
    unsigned char       rtm_tos;

    unsigned char       rtm_table;  /* Routing table id */
    unsigned char       rtm_protocol;   /* Routing protocol; see below  */
    unsigned char       rtm_scope;  /* See below */
    unsigned char       rtm_type;   /* See below    */

    unsigned        rtm_flags;
};
```

Fig. 3.17 Estructura de rtmsg.

Como se puede observar en la Fig. 3.17, las variables que forman la estructura *rtmsg* son: una variable para el almacenamiento del identificador de tablas de encaminamiento, una variable para almacenar el protocolo a utilizar, entre otros parámetros menos importantes.

Una vez tenemos la *rtentry* convertida en las tres estructuras anteriores, se procede a comprobar que es lo que se desea hacer: añadir ruta en la tabla o eliminarla.

Si lo que se quiere hacer es añadir una ruta, se llama a la función *fn_hash_insert*, tal como muestra la figura 3.18.

```
static int
fn_hash_insert(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
           struct nlmsghdr *n, struct netlink_skb_parms *req)
```

Fig. 3.18 Llamada a la función fn_hash_insert().

Ésta función comprueba que todos los tamaños de variables sean correctos y busca en la tabla FIB una entrada con la información recibida. Si encuentra una entrada la actualiza, si no la encontrase crearía una nueva entrada.

Si por lo contrario, se quiere eliminar una ruta, se llama a la función *fn_hash_delete*.

```
static int
fn_hash_delete(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
           struct nlmsghdr *n, struct netlink_skb_parms *req)
```

Fig. 3.19 Llamada a la función fn_hash_delete().

Ésta función, mostrada en la figura 3.19, comprueba que todas las variables sean correctas. Después, busca en la tabla FIB la entrada a eliminar. En caso de encontrarla la elimina, sino devuelve un error.

Una vez llegados a éste punto, ya tenemos la información añadida a la tabla FIB o eliminada de la tabla. La figura 3.20 resume todos los pasos necesarios en éste proceso.

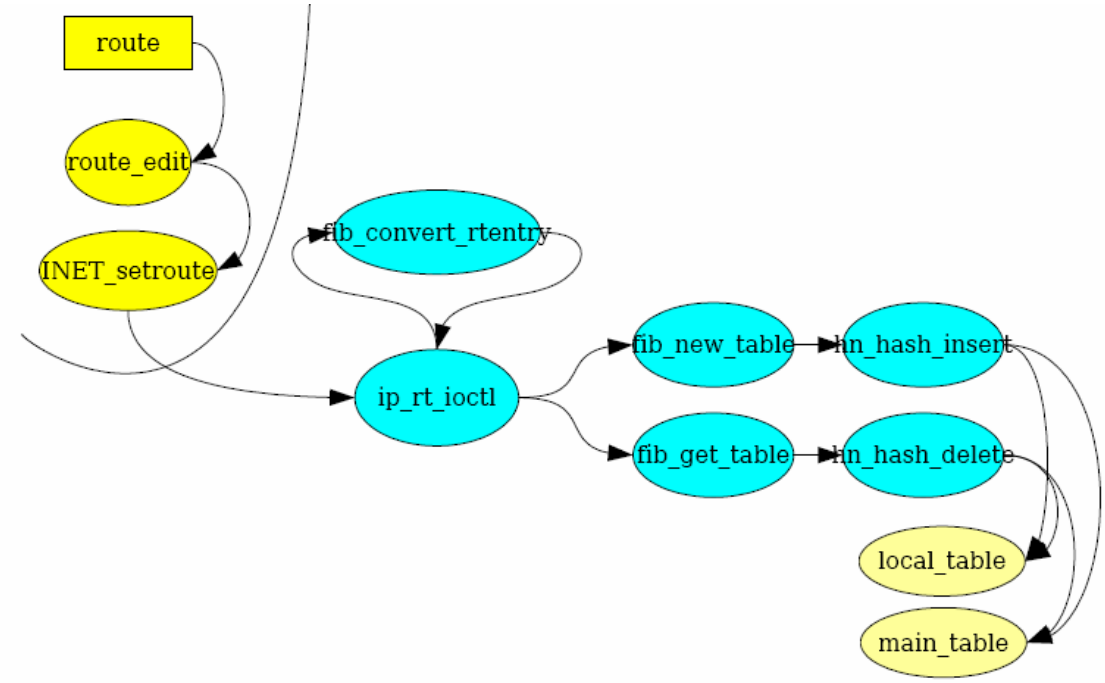


Fig. 3.20 Resumen del proceso de añadir/eliminar ruta.

3.5 Tablas de encaminamiento del Kernel

En éste apartado se pretende explicar las diferentes tablas de encaminamiento que el Kernel de Linux dispone para hacer el encaminamiento. Se detallan las diferentes tablas a continuación.

3.5.1 Neighbor table

La *neighbor table* o también llamada “tabla de vecinos” contiene información de direcciones para hosts que están directamente conectados a nuestro host. La información que incluye son: el dispositivo al que esta conectado el vecino y el protocolo que utiliza para el intercambio de información. Linux utiliza el protocolo ARP (Address Resolution Protocol) para mantener y actualizar esta tabla. La tabla es dinámica y sus entradas son añadidas cuando son necesarias, pero si no son utilizadas en un cierto tiempo estas entradas son eliminadas.

3.5.2 Forwarding Information Base table

La tabla Forwarding Information Base (FIB) es la estructura más importante para el encaminamiento en el Kernel. Ésta contiene toda la información de encaminamiento necesaria para buscar cualquier dirección IP a partir de su mascara de red. Una función de la capa de red entra en la tabla con la dirección destino del paquete en el momento de encaminar y va comparando esta con mascaras de red más especificas para ir buscando la entrada en la tabla. Si utilizando mascaras de red especificas no encuentra la entrada, entonces se sube a un nivel más general de mascara de red y compara direcciones IP’s. Una vez encontrada la entrada, la capa de red copia la dirección destino en otra tabla llamada tabla caché y envía el paquete.

Las variables globales que hace de punto de acceso a la tabla FIB son *struct fib_table *local_table, *main_table*. Estas contienen una función de salto de tablas que nos permiten entrar en un punto concreto en la tabla de hash y acceder a la información de zona que contienen.

La estructura de la *fib_table* (puntero hacía *fn_hash*) incluye punteros hacia estructuras de zona (*fn_zone*) con un prefijo limitado (de 0 a 32 bits). Todas las entradas de la tabla de encaminamiento con el mismo prefijo son almacenadas en la misma zona. Además, cada *fn_zone* utiliza una tabla de hash adicional para guardar las entradas individuales, estas son representadas como estructuras llamadas nodos (*fib_node*). La función de hash utilizada para estas entradas es la utilización del prefijo de red de entrada. Si existen unas cuantas entradas a la tabla de encaminamiento con el mismo valor de hash, entonces se crea una lista enlazada con los nodos existentes. No toda la información es guardada en el *fib_node*, sino que existe una estructura que se utiliza para almacenar información común a las entradas. Ésta estructura recibe el nombre de *fib_info*.

Dentro de estas estructuras, existe la estructura de tabla de mascaras de red (*fn_hash*). Ésta estructura contiene punteros hacia zonas individuales, organizadas por mascara de red. Cada zona corresponde a una única máscara de red.

```
struct fn_hash {
    struct fn_zone  *fn_zones[33];
    struct fn_zone  *fn_zone_list;
};
```

Fig. 3.21 Estructura de fn_hash.

Como podemos ver en la Fig. 3.21, podemos acceder a las diferentes zonas mediante la variable **fn_zones[33]* [2].

Después, nos encontramos con la estructura de zona de red (*fn_zone*) tal y como muestra la figura 3.22. Ésta tabla contiene información de hash y punteros hacia tablas de hash de los nodos. Estos nodos tienen su propia estructura (*fib_node*) y contienen la información para una dirección IP y un puntero hacia una estructura llamada *fib_info*. La *fib_info* contiene la información sobre protocolo y hardware que los nodos utilizaran para enviar la información.

```
struct fn_zone {
    struct fn_zone      *fz_next;   /* Next not empty zone  */
    struct hlist_head   *fz_hash;   /* Hash table pointer   */
    int             fz_nent;    /* Number of entries    */

    int             fz_divisor; /* Hash divisor     */
    u32             fz_hashmask;    /* (fz_divisor - 1) */
#define FZ_HASHMASK(fz)     ((fz)->fz_hashmask)

    int             fz_order;   /* Zone order       */
    u32             fz_mask;
#define FZ_MASK(fz)     ((fz)->fz_mask)
};
```

Fig. 3.22 Estructura de fn_zone.

La estructura *fn_zone* es la encargada de manejar todas las entradas con la misma longitud de prefijo que es usado como valor de hash en la tabla de hash de la FIB. El tamaño del prefijo es guardado en la variable *fz_order* de la estructura *fn_zone* y en *fz_mask* se guarda la máscara de red.

La tabla de hash de la estructura *fn_zone* consiste en un array de *fib_node*'s, que son referenciados a partir del valor del puntero *fz_hash*. El tamaño de esta array lo podemos encontrar en la variable *fz_divisor*. La estructura *fn_zone* existe para un determinado valor de prefijo, solo si esta estructura contiene

[2] Esta es de tamaño 33 debido a que la primera zona es la 0 y la última sería la 32 (todos los bits de la máscara de red a 1).

entradas para ese valor de prefijo. Todas las *fn_zone* se encuentran situadas en una lista enlazada ordenadas de forma descendiente por el prefijo.

La estructura *fib_node*, representada en la Fig. 3.23, representa una entrada simple en una tabla de encaminamiento. Cada nodo tiene una estructura llamaba *fn_key*, que contiene la dirección de destino (mapeada según el prefijo de la zona) y el campo de la cabecera IP Type Of Service (ToS), ya que estos valores son los utilizados para hacer la búsqueda dentro de la tabla FIB. Toda la información adicional en la entrada de la tabla de encaminamiento no es requerida por lo que se guarda en la estructura *fib_info*.

```
struct fib_node {
    struct hlist_node   fn_hash;
    struct list_head    fn_alias;
    u32         fn_key;
};
```

Fig. 3.23 Estructura de fib_node.

La estructura *fib_info* (Fig. 3.24) es la única estructura que no contiene una referencia anterior sobre su posición. Todas las *fib_info* están estructuradas en una doble lista enlazada, que utilizan los punteros *fib_next* y *fib_prev* para desplazarse sobre esta. En esta estructura se suele almacenar el tipo de protocolo a utilizar y la interfaz de salida del paquete, entre otra información.

```
struct fib_info
{
    struct fib_info     *fib_next;
    struct fib_info     *fib_prev;
    int         fib_treeref;
    atomic_t        fib_clntref;
    int         fib_dead;
    unsigned        fib_flags;
    int         fib_protocol;
    u32         fib_prefsrc;
    u32         fib_priority;
    unsigned        fib_metrics[RTAX_MAX];
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMSS-1]
    int         fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    int         fib_power;
#endif
    struct fib_nh       fib_nh[0];
#define fib_dev     fib_nh[0].nh_dev
};
```

Fig. 3.24 Estructura de fib_info.

En la siguiente figura, Fig. 3.25, se muestra un esquema de la estructura de la tabla FIB para su mejor comprensión.

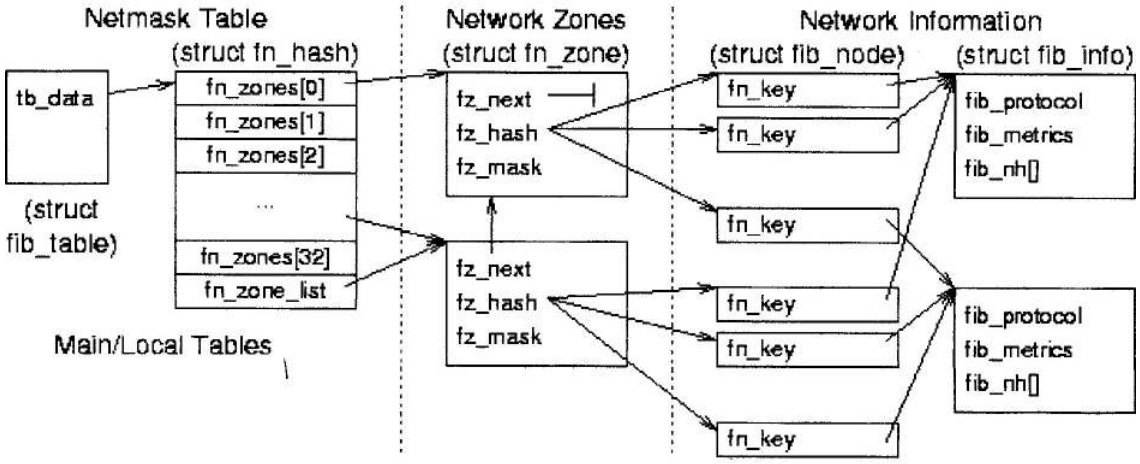


Fig. 3.25 Visión general de la tabla FIB.

3.5.3 Cache table

De cara a disminuir el tiempo de búsqueda de una dirección IP en la tabla FIB, el Kernel de Linux tiene una cache table (tabla caché) donde guarda los resultados de las búsquedas de encaminamiento más frecuentes.

Para el encaminamiento de un paquete, la primera operación que se hace es consultar la tabla caché. Si esta búsqueda no tiene éxito, se procede a buscar en la tabla FIB. Si se ha tenido que consultar la tabla FIB, el resultado de esta creará una nueva entrada en la tabla caché.

La tabla caché esta basada en una estructura simple de datos. Ésta formada por una tabla de hash en la cual cada posición es el inicio a una lista enlazada, es decir, cada posición de la lista enlazada tendrá el mismo valor en la tabla de hash. Para hacer el cálculo del valor de hash, éste se basa en la dirección IP origen y destino y también el valor de ToS. Cada posición de una lista enlazada contiene toda la información para el encaminamiento de un paquete.

La tabla de hash es un array que recibe el nombre de *rt_hash_bucket*. Cada posición de esta array tiene un chain (la lista enlazada), y cada posición de la chain tiene una estructura *rtable* que representa las entradas en caché donde se guarda la información. A continuación, la Fig. 3.26, muestra un gráfico conceptual que hace más comprensible la estructura.

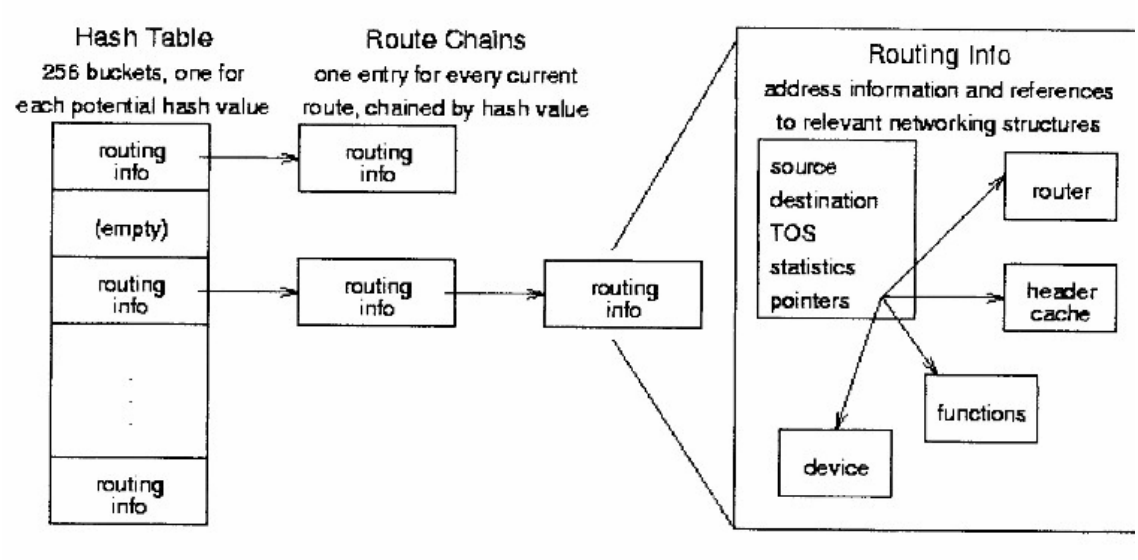


Fig. 3.26 Visión general de la tabla caché.

Como hemos comentado antes en éste mismo apartado, cada posición individual de la lista enlazada contiene la información de encaminamiento de una ruta. Cada posición individual de la lista enlazada es una estructura del tipo *rtable*. La Fig. 3.27 muestra la estructura *rtable*.

```
struct rtable
{
    union
    {
        struct dst_entry    dst;
        struct rtable       *rt_next;
    } u;

    struct in_device    *idev;

    unsigned        rt_flags;
    unsigned        rt_type;

    __u32           rt_dst; /* Path destination */
    __u32           rt_src; /* Path source      */
    int         rt_iif;

    /* Info on neighbour */
    __u32           rt_gateway;

    /* Cache lookup keys */
    struct flowi        fl;

    /* Miscellaneous cached information */
    __u32           rt_spec_dst; /* RFC1122 specific destination */
    struct inet_peer    *peer; /* long-living peer info */
};
```

Fig. 3.27 Estructura de rtable.

La Fig. 3.27 muestra que la estructura *rtable* almacena la información de encaminamiento. Se puede ver una unión entra la dirección IP destino y el siguiente punto en la lista enlazada, esto es así para hacer más rápida la búsqueda de ruta, ya que solo comprueba la dirección IP destino y sino es la buena continua en la siguiente posición de la lista enlazada. A parte de esta información, también podemos encontrar otras variables útiles para el encaminamiento como puede ser la puerta de enlace o la interfaz de salida.

3.5 Procesamiento de paquetes de entrada

Éste apartado pretende mostrar como funciona el código del Kernel para recibir un paquete. Ésta explicación describe el proceso completo de recepción[3].

Cuando recibimos un paquete, la primera función que trata el paquete a nivel IP es la función *ip_rcv()* (*net/ipv4/ip_input.c*). La Fig. 3.28 muestra la cabecera de esta función.

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
```

Fig. 3.28 Función ip_rcv().

Ésta función tiene el objetivo de validar la cabecera IP. En esta validación se tiene en cuenta que la longitud del datagrama sea la correcta, que la versión del protocolo IP sea la versión 4 y que el checksum sea el correcto, sino el paquete es descartado. En la figura 3.29 podemos ver el código que realiza éstas comprobaciones.

[3] Trabajos futuros podrán consultar estas secciones como manual de código.

```
/*
 *  RFC1122: 3.1.2.2 MUST silently discard any IP frame that fails the checksum.
 *
 *  Is the datagram acceptable?
 *
 *  1.  Length at least the size of an ip header
 *  2.  Version of 4
 *  3.  Checksums correctly. [Speed optimisation for later, skip loopback checksums]
 *  4.  Doesn't have a bogus length
 */

if (iph->ihl < 5 || iph->version != 4)
    goto inhdr_error;

if (!pskb_may_pull(skb, iph->ihl*4))
    goto inhdr_error;

iph = skb->nh.iph;

if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
    goto inhdr_error;

{
    __u32 len = ntohs(iph->tot_len);
    if (skb->len < len || len < (iph->ihl<<2))
        goto inhdr_error;
```

Fig. 3.29 Código de validación de un datagrama IP en ip_rcv().

Cuando se han comprobado todos los campos, se pasa el paquete a la función *ip_rcv_finish()*. La figura 3.30 muestra la llamada desde *ip_rcv()* a la función *ip_rcv_finish()*.

```
return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
               ip_rcv_finish);
```

Fig. 3.30 Llamada a ip_rcv_finish() en ip_rcv().

La función *ip_rcv_finish()* esta definida en *net/ipv4/ip_input.c*. El primer objetivo de esta función es llamar a *ip_route_input(),* como se muestra en la Fig. 3.31, para que determine la siguiente función para controlar el sk_buff, para compilar cualquier opción de la cabecera IP en el controlador de buffer de sk_buff's.

```
if (skb->dst == NULL) {
    if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
        goto drop;
}
```

Fig. 3.31 Llamada a ip_route_input().

El *sk_buff* (socket buffer), como ya hemos comentado anteriormente en el apartado 3.2.1, es la manera que tiene el Kernel de disponer de toda la información de red sin necesidad de saber a que capa pertenece, es decir en el socket buffer podemos encontrar información desde la capa de enlace hasta la capa de transporte. Ésta información es necesaria para las diferentes funciones que posee el Kernel para el control de red.

El valor de *skb->dst* es normalmente nulo si el paquete ha sido recibido desde fuera. Por éste motivo se llama a i*p_route_input()* para poner en *skb->dst* el destino del paquete. Ésta parte del código se muestra en la Fig. 3.31.

La función *ip_route_input()* busca una ruta en la tabla de encaminamiento caché. Si no encuentra la ruta para éste paquete, llama a la función *ip_route_input_slow()*, mostrada en la Fig. 3.32.

```
return ip_route_input_slow(skb, daddr, saddr, tos, dev);
```

Fig. 3.32 Llamada a ip_route_input_slow().

La función *ip_route_input_slow()* realiza una búsqueda más intensa consultando la tabla FIB, que como ya se ha comentado en el apartado 3.5.2, es la tabla donde hay toda la información de encaminamiento que posee el sistema.

Después de hallar la ruta, el Kernel debe decidir si ese paquete se tiene que reenviar o si el paquete es para el propio host. En caso que sea para el propio host, se llama a la función *ip_local_delivery()* que es la encargada de reensamblar el paquete IP original (en caso que que sea fragmentado) y de leer la cabecera del paquete para saber que protocolo de transporte utiliza y pasar el paquete a un nivel superior. Si el paquete se ha de encaminar, se llama a la función *ip_forward()*

A continuación, la Fig. 3.33 muestra un esquema de todo el proceso de recepción de un paquete IP (a nivel de red):

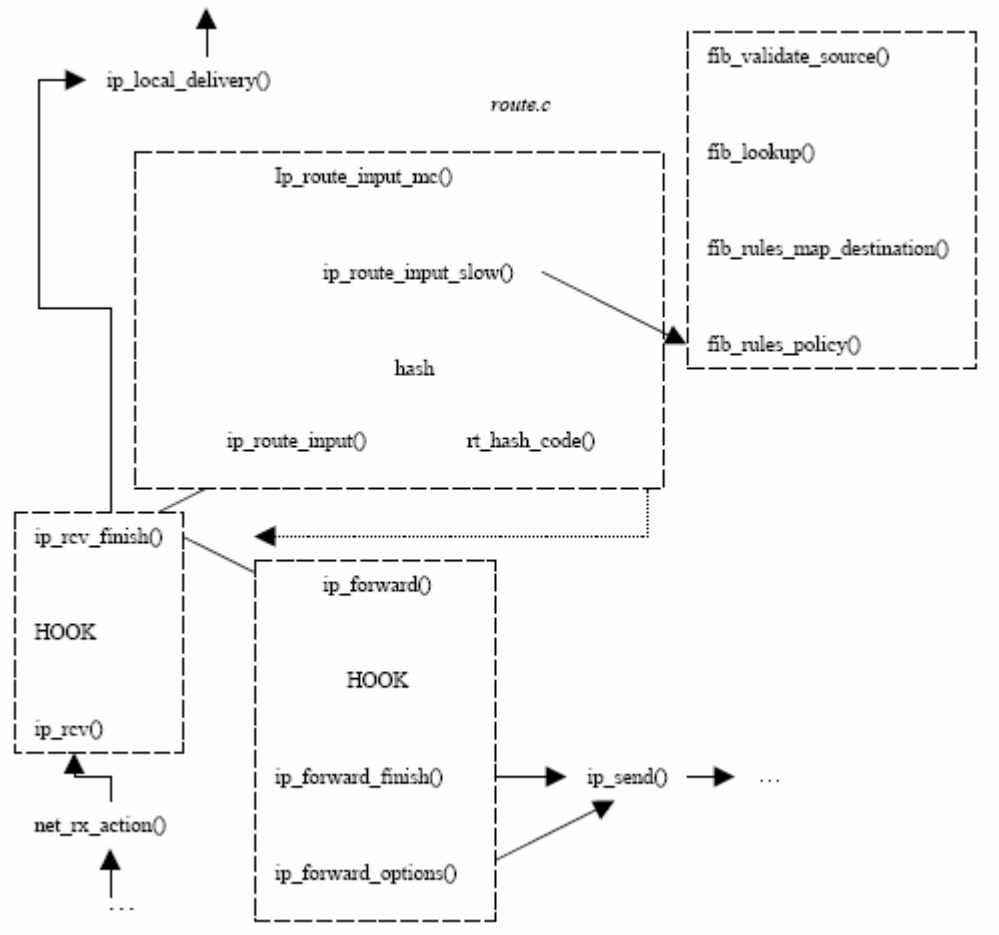


Fig. 3.33 Visión general de recepción de un paquete IP a nivel de red.

3.6 Procesamiento de paquete de salida

En éste apartado se pretende realizar el mismo análisis que en el apartado anterior, pero para un paquete que sale de nuestro host.

La primera función que encontramos (*ip_queue_xmit()*) realiza la función de añadir a la cabecera IP toda la información de opciones útiles para el encaminamiento. Después, *ip_queue_xmit2()* comprueba que sea todo correcto.

Entonces se llama a la función *ip_output()*, mostrada en la Fig. 3.34. Ésta función comprueba el tamaño del paquete IP y si el tamaño excede la MTU, llama a una función (*ip_fragment()*) que se encarga de fragmentarlo y al final se acaba llamando a *ip_finish_output()*, mostrada en el Fig. 3.34.

```
int ip_output(struct sk_buff **pskb)
{
    struct sk_buff *skb = *pskb;

    IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);

    if ((skb->len > dst_pmtu(skb->dst) || skb_shinfo(skb)->frag_list) &&
        !skb_shinfo(skb)->tso_size)
        return ip_fragment(skb, ip_finish_output);
    else
        return ip_finish_output(skb);
}
```

Fig. 3.34 Código de la función ip_output().

La función *ip_finish_output()* se encarga de definir la interfaz de salida y el protocolo de nivel de enlace en el socket buffer. Como último, llama a *ip_finish_output2()*. Esto se puede observar en la figura 3.35.

```
int ip_finish_output(struct sk_buff *skb)
{
    struct net_device *dev = skb->dst->dev;

    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);

    return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
                   ip_finish_output2);
}
```

Fig. 3.35 Código de la función ip_finish_output().

Lo único que hace *ip_finish_output2()* (llamada por *ip_finish_output()* como podemos ver en la Fig. 3.35) es reservar espacio en el buffer de la tarjeta.

Cada uno de los paquetes contiene un campo *dst*, que almacena la dirección IP destino para ser utilizada en el encaminamiento. El campo *dst* provee una función de salida. Para los paquetes IP es la función *dev_queue_xmit()*.

El Kernel posee unas cuantas disciplinas de cola. La cola por defecto que utiliza es la cola *FIFO* (First In First Out) con una longitud por defecto de 100 paquetes.

Para cada paquete que se va a transmitir desde la capa de red (IP), se llama a la función *dev_queue_xmit()*. Ésta pone los paquetes IP en la cola de *qdisc* (herramienta que gestiona las colas de paquetes en Linux) asociada con la interfaz de salida (es determinada por el encaminamiento anterior). Entonces, si el dispositivo en nuestro caso la tarjeta de red no esta parada (podría estar

parada por ejemplo por fallo en el enlace), todo los paquetes que haya en *qdisc* serán tratados por *qdisc_restart()*

Después, se llama a la función *hard_start_xmit()*. Ésta función esta implementada en el código del driver. El descriptor del paquete se sitúa en el *tx_ring* y el driver decide la NIC que tiene algún paquete para enviar.

Una vez que la tarjeta de red ha enviado el paquete o un grupo de paquetes, esta comunica a la CPU que los paquetes han sido enviados. La CPU utiliza esta información para poner nuevos paquetes en la cola y el planificador (*scheduler*) libera la memoria utilizada para esos paquetes. Ésta comunicación entre la tarjeta de red y la CPU depende de la tarjeta y del driver utilizado. Todo éste proceso se muestra en la Fig. 3.36.

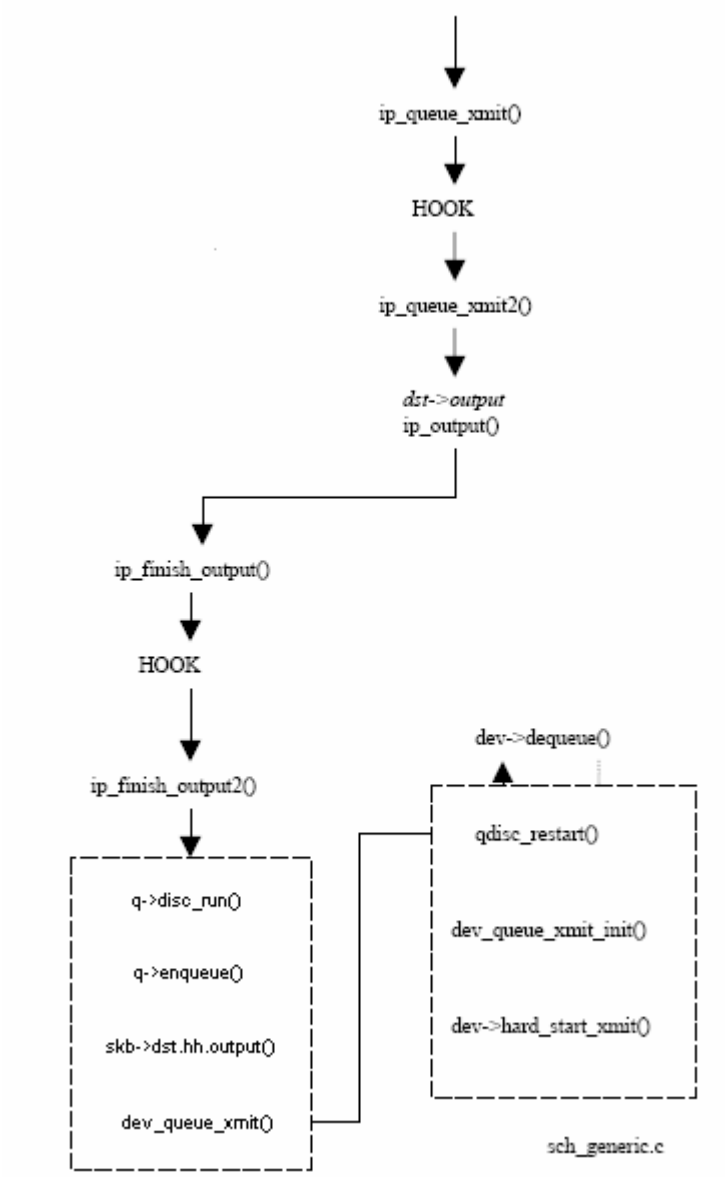


Fig. 3.36 Visión general del proceso de un paquete de salida.

CAPITULO 4. Modificaciones realizadas

4.1. Posibles opciones de modificación

En éste proyecto como en muchos otros, existen varias posibilidades para llegar a un mismo resultado, en nuestro caso el encaminamiento según la QoS de los paquetes. En esta parte, se enumeraran la diferentes opciones que se tenían y que dificultades tenían.

La primera posibilidad de la que se disponía era la de utilizar el software Iptables [12] que viene instalado por defecto en muchas distribuciones de Linux y el espacio reservado para ToS en los paquetes IP utilizarlo para poner el valor de QoS. La idea era programar Iptables para enviar los paquetes con una cierta ToS por una ruta y con otra ToS por otra ruta. Ésta opción no era buena para nuestro proyecto, debido a que se trata de una red móvil en la cual los dispositivos pueden ir cambiando de lugar y por la cual cosa ir cambiando la QoS. Además, queríamos que la asignación de ruta fuese dinámica y no estática como sucedería con esta opción.

Una segunda opción fue estudiar el Kernel para ver su estructura, entenderla e intentar añadir una variable de QoS en las estructuras de las tablas de encaminamiento y con lo que ello conlleva modificar las funciones de tratamiento de la información, para que encaminara según la QoS. Ésta opción parecía buena, pero se necesitaba un alto conocimiento del Kernel de Linux y al no existir ningún *debugger* (al menos no he encontrado ninguno) era una tarea complicada, porque el mínimo error haría que todo el sistema fallara.

En el transcurso del estudio de la segunda opción, se propuso realizar el encaminamiento de paquetes a partir de un campo ya existente en las tablas de encaminamiento del Kernel. Está fue la opción definitiva y la que se eligió como la más idonea para la realización de éste trabajo. Básicamente la idea es que el módulo AODV-QoS comunicaría al Kernel cuando tuviera una nueva ruta con un valor de QoS mediante el transferencia de éste valor a una variable de ToS que las tablas del Kernel ya poseen. Y a la hora de encaminar a ni vel de Kernel éste miraría tanto la dirección destino como la ToS del paquete para elegir una ruta.

Ésta es la opción elegida y se explica con más detalle en la siguiente sección.

4.2. Desarrollo de la opción elegida

Como ya hemos comentado en la sección 4.1, esta opción surgió en el transcurso de la investigación del Kernel de Linux. Pudimos comprobar mediante literatur [11] que los Kernel 2.4 tenían una opción de encaminamiento por ToS, que según el valor de éste elegía una ruta u otra.

El problema que teníamos es que teníamos que utilizar una versión de Kernel más nueva por la distribución de Linux que utilizábamos y que en los Kernels más nuevos no venía esta opción.

La solución, fue mirar el código del Kernel como si llegara un paquete para ver el funcionamiento del Kernel en el tratamiento de un paquete, y comprobamos que la búsqueda de una ruta[4] en la tabla FIB (ver sección 3.5.2) se realizaba a partir del valor del campo ToS del paquete. Para el caso de la búsqueda de una ruta en la tabla cache (ver sección 3.5.3), éste problema también lo teníamos solucionado, debido a que al crear la *key* de búsqueda de la tabla utilizaba la ToS (entre otros valores) para el cálculo de ésta.

El segundo paso fue ver si para una misma ruta podíamos añadir en la tabla de encaminamiento más de una ruta hacia el mismo destino pero con la variable QoS diferente.

A partir de un estudio exhaustivo del Kernel y de las diferentes tablas de encaminamiento, se observó que la variable ToS se utiliza (a partir de una función de *hash* junto con otros parámetros) para elegir la posición en la tabla cache. Por lo tanto, en caso de utlizar el campo ToS para almacenar la variable QoS, para diferentes valores de QoS existirán diferentes valores de ToS, es decir, existirán diferentes posiciones y diferentes entradas en las tablas de encaminamiento cache y FIB.

Una vez realizado éste estudio del Kernel, se inicio la tarea de modificar primero la implementación. Al principio, cuando la implementación (AODV-QoS) realizaba la llamada a la función para añadir una ruta a nivel de usuario, si que se tenía en cuenta la QoS, pero esta misma función cuando realizaba la llamada a la función de la implementación que añadía la ruta en el Kernel, esta no tenía en cuenta el valor de QoS. Visto esto, el primer paso fue la modificación de la estructura rtentry, que como ya hemos comentado con anterioridad, esta es la estructura que nos permite añadir rutas des del exterior. La modificación se desarollo añadiendo una variable en la estructura rtentry para guardar la variable que nos que nos transmitiría la implementación AODV-QoS. La modificación quedaría tal y como muestra la figura 4.1:

[4] La zona la busca mediante la máscara de red, pero para los nodos utiliza la dirección IP destino y el valor de ToS.

```
struct rtentry
{
    unsigned long   rt_pad1;
    struct sockaddr rt_dst;      /* target address        */
    struct sockaddr rt_gateway; /* gateway addr (RTF_GATEWAY)    */
    struct sockaddr rt_genmask; /* target network mask (IP) */
    unsigned char rt_tos;
    unsigned short  rt_flags;
    short       rt_pad2;
    unsigned long   rt_pad3;
    void        *rt_pad4;
    short       rt_metric;  /* +1 for binary compatibility! */
    char __user *rt_dev;    /* forcing the device at add    */
    unsigned long   rt_mtu;     /* per route MTU/Window     */
#ifndef __KERNEL__
#define rt_mss  rt_mtu          /* Compatibility :-(            */
#endif
    unsigned long   rt_window;  /* Window clamping      */
    unsigned short  rt_irtt;    /* Initial RTT          */
};
```

Fig. 4.1 Estructura rtentry modificada.

Una vez modificada la estructura de rtentry, el siguiente paso era modificar la implementación de AODV-QoS, para que la función que añade la ruta al Kernel pudíera también añadir la QoS en esa ruta,es decir, modificar la cabecera de la función junto con el archivo .h que define la función. La figura 4.2 nos muestra la cabecera modificada.

```
int k_add_rte(struct in_addr dst, struct in_addr gw, struct in_addr nm,
          short int hcnt, unsigned int ifindex, u_int8_t qos)
```

Fig. 4.2 Cabecera de la función k_add_rte() modificada.

Una vez realizada la modificación anterior, únicamente no quedaría añadir la modificación para añadir esta variable a la variable de la tabla *rtentry*. La figura 4.3 muestra como se realizo.

```
rte.rt_tos = qos;
```

Fig. 4.3 Añadimos la QoS en la tabla rtentry.

A nivel de implementación ya no hacia falta realizar ninguna modificación más, ya que como veremos en la figura 4.4, el próximo paso es llamar a *IOCTL()* para empezar a añadir la ruta en el Kernel.

```
/* Send message by ioctl(). */
ret = ioctl(sock, SIOCADDRT, &rte);
```

Fig. 4.4 Llamada a la función IOCTL().

Una vez modificada la parte de nivel alto, había que empezar a pensar en como modificar el Kernel en su parte más interior. Observando la función de insertar una ruta en la tabla FIB, pude ver como la ToS era transmitida de una de las tres estructuras que salen de la función *fib_convert_rtentry()* a una variable llamada tos, tal y como muestra la figura 4.5.

```
static int
fn_hash_insert(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
           struct nlmsghdr *n, struct netlink_skb_parms *req)
{
    struct fn_hash *table = (struct fn_hash *) tb->tb_data;
    struct fib_node *new_f, *f;
    struct fib_alias *fa, *new_fa;
    struct fn_zone *fz;
    struct fib_info *fi;
    int z = r->rtm_dst_len;
    int type = r->rtm_type;
    u8 tos = r->rtm_tos;
    u32 key;
    int err;
```

Fig. 4.5 Definición del valor del ToS.

Entonces, la idea fue estudiar la función *fib_convert_rtentry()* para observar de donde se cogía el valor de ToS que más tarde se utilizaría para coger el valor de QoS. Por sorpresa, ninguna variable le pasaba ningún valor ha éste parámetro. Entonces, la solución fue añadir una líneas ha esta función que se muestran en la figura 4.6.

```
if(r->rt_tos == NULL){
                rtm->rtm_tos = 0;
}else{
        rtm->rtm_tos = r->rt_tos;
}
```

Fig. 4.6 Código añadido en la función fib_convert_rtentry().

Como podemos ver en la imagen, son las líneas añadidas a la función. Se tiene un control de rt_tos debido a que otros programas o implementaciones no utilizaran éste valor entonces el valor de QoS es el más bajo. En cambio si la llamada es de la implementación de AODV-QoS, éste valor no será nulo y se le pasará bien. Esto se ha realizado para intentar dejar el Kernel lo más funcional

posible, es decir, no hacer solamente un Kernel para AODV-QoS sinó un Kernel genérico.

4.3. Resultados

Los resultados no eran del todo los que esperábamos, pero por una parte se entiende, ya que modificar el núcleo de un sistema es muy difícil y al no tener ningún programa para hacer un debug pues la tarea llega a un nivel de complicación elevado.

La finalidad del proyecto era modificar el Kernel y hacer alguna prueba sobre una red real. La modificación ha sido realiza, pero durante las pruebas he tenido problemas.

Conseguí hacer funcionar el AODV-QoS con el Kernel modificado haciendo ping's de una máquina a otra, íncluso conseguí una conexión entre dos máquinas mediante SSH, pero no acabava de enviar bien la información.

Uno de los posibles problemas puede ser que la implementación modificada (AODV-QoS) no marque los paquetes, es decir, que no ponga en el campo ToS el valor de la QoS, con lo cual siempre enviaríamos paquetes con el campo ToS a cero con lo cual no podríamos tener la opción de multicamino. Otro posible problema sería en el momento que hemos modificado el Kernel, que como hemos visto en el apartado anterior, la función que añade una ruta al Kernel cogía el valor de ToS de una variable (*r->rtm_tos*). Ésta línea de código la podemos observar en la Fig. 4.7.

```
static int
fn_hash_insert(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
           struct nlmsghdr *n, struct netlink_skb_parms *req)
{
    struct fn_hash *table = (struct fn_hash *) tb->tb_data;
    struct fib_node *new_f, *f;
    struct fib_alias *fa, *new_fa;
    struct fn_zone *fz;
    struct fib_info *fi;
    int z = r->rtm_dst_len;
    int type = r->rtm_type;
    u8 tos = r->rtm_tos;
    u32 key;
    int err;
```

Fig. 4.7 Localización de un posible problema.

Como vemos en la figura 4.7, es ahí donde se pasa el valor de la ToS, pero en la función anterior (*fib_rtentry_convert()*) en la cual se pasaba de una estructura rtentry a tres estructuras que más adelante se utilizaban para añadir la ruta, no hemos encontrado ninguna línea de código que cogiera el valor de la ToS y lo

añadiera a la variable r->rtm_tos. Podemos intuir que puede estar el problema por éste sector del código, que íncluso siendo el esquema correcto para añadir la ruta, quizas el valor de la ToS no se adquiera por esta banda.

Despues de encontrar posible problemas, dejamos para proyectos posteriores investigar sobre como solucionar estos dos posibles problemas, es decir, implementar un marcador de paquetes para marcar los paquetes con la ToS que a su vez será la QoS e investigar de donde consigue la variable r->rtm_tos el valor de la ToS.

Conclusiones

A primera vista cuando te plantean un problema o como es el caso, un tema para un proyecto final de carrera, uno tiene la sensación que quizás no sea una tarea difícil. Pero, poco a poco a medida que vas desarrollando el trabajo las dificultades aumentan y te das cuenta de lo complejo que puede llegar a ser el trabajo.

La información sobre la parte de red del Kernel es casi nula, y la poca que hay es para el Kernel 2.4, lo cual ha hecho que entender el Kernel haya sido toda una odisea. Luego esta el tema de modificar el Kernel a ciegas, es decir, estamos acostumbrados a modificar o escribir código mediante un programa que te va diciendo los fallos o que cuando ejecutas puedes ir haciéndolo paso a paso para ver donde falla. En cambio, con el Kernel esta comodidad no existe, ya que no hay ningún programa que haga todo lo comentado anteriormente.

Después, aparece el tema de instalar el Kernel tarea que parecía fácil, porque estamos acostumbrados a instalar software de tipo Microsoft que es hacer un par de clicks y ya esta instalado. Instalar un Kernel si es la primera vez que lo haces es un poco complicado, debido a que tienes muchas opciones en el menú de configuración y para elegir bien las opciones, has de entender bien el funcionamiento de un ordenador.

Pero gracias a todas estas complicaciones, he podido desarrollar mi capacidad de investigación y aprender más sobre Linux, ese sistema que tan versátil es y tanto miedo tenemos de utilizar por su complejidad, pero que si haces un buen estudio sobre lo que quieres hacer sobre el no esta complicado.

Para finalizar, ha sido un proyecto muy interesante y me ha dado la oportunidad de superarme dentro de un ámbito que era casi totalmente desconocido por mí.

Impacto Medioambiental

Las redes inalámbricas, en especial las Ad-hoc, tienen un bajo impacto medioambiental del tipo visual o estético, gracias a que no es necesario instalar ningún tipo de cable ni punto de acceso.

Por otra parte, las comunicaciones inalámbricas se basan en la utilización de las ondas electromagnéticas para transportar la información. Ésta radiación comporta un fuerte impacto ambiental sobre todo para las personas, ya que en determinadas situaciones puede provocar un efecto nocivo en la salud de las personas y animales.

En el entorno laboral, doméstico y sobretodo en el urbano es muy difícil encontrar un espacio libre de emisiones radioeléctricas. En principio el espectro electromagnético está regulado por las administraciones públicas pero en algunas ocasiones si la radiación no está controlada puede provocar efectos térmicos a las personas y ruido electrónico que afecta al sistema inmunológico y neuronal y puede provocar problemas como dolor de cabeza, a parte de crear interferencias en diferentes máquinas o electrodomésticos.

Las redes Ad-hoc pueden conseguir importantes avances con muy poco impacto medioambiental negativo. Por ejemplo las operaciones de emergencia en catástrofes naturales (terremotos, inundaciones), zonas sin infraestructuras (zonas en guerra, pobreza) o las redes de sensores, las cuales disponen de dispositivos muy pequeños, baratos y con batería de larga duración que se pueden usar por ejemplo para detectar en bosques posibles incendios y así reaccionar rápidamente.

Bibliografía

[1] R. Galera y E. Gispert. TFC. “Banco de pruebas para protocolos de encaminamiento en redes móviles Ad-hoc”

[2] E. Nordstrom, H. Lundgren, Implementación de AODV-UU (Universidad de Uppsala). http://user.it.uu.se/~henrikl/aodv/.

[3] Inicialización de TCP/IP :

http://www.rabbitsemiconductor.com/documentation/docs/manuals/TCPIP/UsersManualV1/dcrtcp.html

[4] C. Perkins, E.M.Belding-Royer, S.Das, ”Ad hoc On-Demand Distance Vector (AODV) Routing” RFC 3561, 2003

[5] Generador de tráfico Iperf, http://dast.nlanr.net/Projects/Iperf/

[6] D. Remondo, “Tutorial on Wireless Ad-hoc Networks”, *Het-Nets’04: Second International Working Conference in Performance Modelling and Evaluation of Heterogeneous Networks*, 26-28 July.

[7] A. Tanenbaum : “Redes de computadoras", Prentice Hall, 1997.

[8] B. Hubert: “Linux 2.4 Advanced Routing HOWTO”

[9] Debate entre A. Tanenbaum y L. Torvals sobre el Kernel: http://groups.google.com/group/comp.os.minix/browse_frm/thread/c25870d7a41696d2/f447530d082cd95d#f447530d082cd95d

[10] Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stac:

http://www.cs.unh.edu/cnrg/gherrin/linux-net.html

[11] S. Maxwell: “Linux Core Kernel Commentary”. Coriolis Group Books cop. 2001. 2ª Edición.

[12] Web donde descargar el Iptables e información sobre él:

http://www.netfilter.org

Anexos

A. Versiones del Kernel

Existen dos versiones del Kernel:

- *Versión de producción:* Ésta es la versión estable hasta el momento. Ésta versión es el resultado final de las versiones de desarrollo o experimentales. Cuando el equipo de desarrollo del Kernel experimental, decide que ha conseguido un Kernel estable y con la suficiente calidad, se lanza una nueva versión de producción o estable. Ésta versión es la que se debería utilizar para un uso normal del sistema, ya que son las versiones consideradas más estables y libres de fallos en el momento de su lanzamiento.

- *Versión de desarrollo:* Ésta versión es experimental y es la que utilizan los desarrolladores para programar, comprobar y verificar nuevas características, correcciones, etc. Estos núcleos suelen ser inestables y no se deberían usar, a no ser que sepas lo que haces.

B. Interpretación de los números de versión

Las versiones del Kernel se numeran con 3 números, de la siguiente forma:

XX.YY.ZZ

XX: Indica la serie principal del Kernel. Hasta el momento solo existen la 1 y 2. Éste número cambia cuando la manera de funcionamiento del Kernel ha sufrido un cambio muy importante.

YY: Indica si la versión es de desarrollo o de producción. Un número impar, significa que es de desarrollo, uno par, que es de producción.

ZZ: Indica nuevas versiones dentro de una versión, en las que lo único que se ha modificado, son fallos de programación /bugs.

Unos ejemplos nos ayudaran a entenderlo mejor:

- Ejemplo 1:

  Versión del Kernel 2.0.0: Kernel de la serie 2 (XX=2), versión de producción 0 (YY=0 par), primera versión de 2.0 (ZZ=0).

- Ejemplo 2:

  Versión del Kernel 2.0.1: Kernel de la serie 2, versión 0, en el que se han corregido errores de programación presentes en la versión 2.0.0 (ZZ=1)

- Ejemplo 3:

  Versión del Kernel 2.1.10: versión 10 del Kernel de desarrollo 2.1.

C. Donde conseguir el Kernel

El Kernel se puede bajar de un gran número de servidores en Internet. Lo más normal es descargarse el Kernel de la web oficial del Kernel:

http://www.kernel.org

En éste enlace tienes la lista oficial de servidores espejos, de donde es posible bajarse cualquier versión del Kernel (ultima y antiguas) y en el subdirectorio /Kernel puedes obtener las ultimas versiones

D. Configuración e instalación de un nuevo Kernel

Éste es uno de los temas que asustan a los nuevos usuarios de Linux. Si que hay que tener claro una serie de cosas antes de ponernos en la instalación, para asi evitar problemas. A continuación, se detallan los pasos para configurar e instalar un nuevo Kernel.

- Hay que bajar la última version del Kernel. Donde la podemos encontrar? En www.kernel.org hay un link a un servidor FTP con todas las versiones del Kernel.

  Si va a instalar un Kernel de las últimas series de producción, se ha de tener en cuenta que algunas distribuciones (si son antiguas) pueden no estar preparadas para hacer uso de estas series. Si vuestra distribución no viene preparada para soportar los últimos Kernels, se tiene que actualizar una serie de paquetes/programas antes de instalar el nuevo Kernel o actualizar a una distribución que los soporte. Se tiene que tener claro las opciones que tenemos que configurar, para poder utilizar el hardware de nuestro sistema, asi como las caracteristicas que queremos utilizar. Por ejemplo, si no utilizamos un dispositivo SCSI, no tenemos que configurar nada en el apartado SCSI de nuestro Kernel. Asi nos ahorramos espacio y tiempo.

- Entrar como root:

```
su root
```

- Copiar el archivo del Kernel al directorio */usr/src/*:

  cp linux-xx.xx.xx.tar.bz2 /usr/src/

- Descomprimir y desempaquetar el Kernel:

  tar -xvzpf linux-xx.yy.zz.tar.gz

  El archivo `linux-xx.yy.zz.tar` se desempaquetara en el directorio `/usr/src/linux`. Si ya existe un directorio llamado linux en el sistema, hay que renombralo, p.ej: `mv linux linux-old`. En algunas distribuciones, `linux` es un enlace simbolico a linux-xx.yy.zz, se ha de borrar éste enlace simbolico. Es importante que no exista ningun directorio/enlace simbolico llamado linux, antes de desempaquetar la nueva version.

- Entrar en */usr/src/linux*:

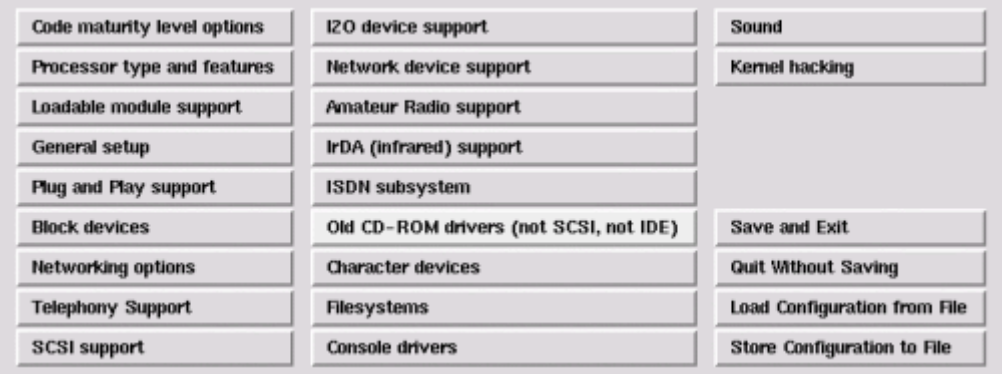
  cd /usr/src/linux

- Configurar el Kernel, esto se puede hacer de tres maneras diferentes:

  make config (modo texto)

  make menuconfig (modo texto con menus)

  make xconfig (X-windows version)



La mejor manera de configurar el Kernel es utilizando el último comando o el segundo, debido a que el primero requiere un nivel de experiencia elevado.

Ahora se configura las opciones que debe tener el nuevo Kernel. Una vez terminada la configuracion, se ha de grabar los cambios y salir del programa de configuracion.

- Una vez terminado el proceso de configuracion, tenemos que compilar nuestro nuevo nucleo. Para ello hay que hacer lo siguiente:

  make dep

  make clean

  make bzImage

- Si en el proceso de configuracion, elegimos alguna opcion como modulo, tendremos que compilar/instalar dichos modulos:

  make modules

  make modules_install

  No olvidar ejecutar como root el comando depmod -a la primera vez que se arranque el nuevo Kernel, para computar las dependencias entre modulos.

- Ya tenemos el Kernel y los modulos compilados, ahora tenemos que instalarlo. Casi todo el mundo utiliza LILO para arrancar el sistema, por ello se explica como instalarlo utilizando LILO.

  Todavia estamos en */usr/src/linux* se ha de ejecutar el comando make install , esto copiara el Kernel que acabamos de crear, a el directorio /boot de nuestro sistema, con el nombre vmlinuz., o como un enlace simbolico vmlinuz -> vmlinuz-xx.yy.zz.

Ahora tenemos que configurar LILO para que reconozca el nuevo Kernel. Tendremos que modificar el fichero /etc/lilo.conf
Un ejemplo del fichero /etc/lilo.conf antes de modificarlo:

```
boot=/dev/hda
prompt
timeout=50
image=/boot/vmlinuz-2.0.34
label=linux
root=/dev/hda1
read-only
```

Despues de la modificacion para que reconozca nuestro nuevo Kernel al arrancar:

```
boot=/dev/hda
prompt
timeout=50
image=/boot/vmlinuz
label=nuevokernel
root=/dev/hda1
read-only
image=/boot/vmlinuz-2.0.34
label=linux
  root=/dev/hda1
  read-only
```

- Ahora solo tenemos que ejecutar el comando /sbin/lilo y arrancar el sistema de nuevo. Si tuviesemos algun problema con el nuevo Kernel, siempre podriamos arrancar con el antiguo escribiendo linux <ENTER> cuando arrancamos y nos sale en pantalla lilo. De esta manera podemos entrar y ver que es lo que ha fallado.

E. Instalación del Controlador para las tarjetas Ipw2200BG

Para instalar las tarjetas Wireless Ipw2200BG integradas en los portátiles se deben seguir los siguientes pasos:

- Crear la carpeta /usr/lib/hotplug/firmware

- Descomprimir el firmware descargado de la pagina Web con la instrucción tar –zxvf ipw2200.fw2.0.tgz
- Movemos los archivos a la ruta que antes hemos creado con mv *.fw /usr/lib/hotplug/firmware

- Descargamos el driver de la tarjeta

- Descomprimimos el controlador con: tar –zxvf ipw 2200.0.15.tgz

- Vamos a la carpeta descomprimida: cd ipw2200-0.15

- Instalamos el controlador con la instrucción: make

- Los cargamos y arrancamos haciendo: ./load

F. Configuración de las tarjetas inalámbricas para trabajar en modo Ad-Hoc

Con las siguientes líneas conseguimos configurar la tarjeta de red en modo Ad-Hoc:

Ifconfig eth0 down
Ifconfig eth1 up
Ifconfig eth1 192.168.10.5 netmask 255.255.255.0
Iwconfig eth1 ESSID "aodv"
Iwconfig eth1 channel 4
Iwconfig eth1 mode Ad-hoc

En estas líneas podemos ver como primero desactivamos la eth0 (que es la tarjeta de red ethernet), activamos eth1 que es la tarjeta inalámbrica, configuramos la IP para esta tarjeta y finalmente la configuramos en la ESSID "aodv" en modo Ad-hoc en el canal 4.

G. Instalación y funcionamiento de la implementación de AODV-QoS: aodvd.

Éste programa es el que implementa el protocolo AODV con QoS y el funcionamiento es el mismo que el de la implementación AODV-UU.

G.1. Instalación

Para instalar la implementación tenemos que realizar los pasos siguientes:

- Desde la consola vamos al directorio en el que tenemos el programa.

- Lo descomprimimos

$: tar –zxvf aodv-uu-0.8.1.tar.gz

- Entramos en el directorio que se acaba de crear (/aodv-uu-0.8.1/)

- Finalmente lo instalamos con los comandos

$: make
$: make install

G.2. Utilización

Para arrancar la implementación hay que entrar en la consola de Linux y ejecutar la siguiente instrucción.

$: aodvd –l –r 3 –o

Aodvd es el nombre del programa y lo demás son las distintas opciones que hemos configurado. En éste caso estamos indicando que haga log de los mensajes que genera el programa (-l), que haga log de la tabla de encaminamiento cada 3 segundos (-r 3) y que utilice la opción optimized-hellos (-o).

Además de estas opciones se pueden utilizar otras. Las opciones posibles son:

-d: Modo daemon. El programa se ejecuta en segundo plano, de forma transparente para el usuario. Los mensajes generados por el programa no se muestran en la consola.

-g: Forzar flag Gratuitous. Fuerza a que todos los paquetes RREQ lleven el flag G activo.

-h: Ayuda. Muestra por pantalla las diferentes opciones que se pueden configurar.

-i (interfaz): Indicamos por cual de las interfaces inalámbricas se tiene que arrancar aodvd. Por defecto se arranca en la primera que esté activa.

-l: Guarda el log de los mensajes del programa en el archivo /var/log/aodvd.log

-o (Optimized-hellos): Los nodos envían paquetes HELLO sólo cuando son parte de una ruta activa. Por defecto envían paquetes HELLO durante todo el tiempo que la aplicación está activa.

-r (seg): Guarda un log de la tabla de encaminamiento cada seg segundos en el archivo /var/log/aodvd.rtlog.

-n (nhellos): Indica el número de hellos que un nodo debe recibir de un vecino para considerarlo un vecino. Por defecto éste número es 3.

-w: Modo Gateway. Es una funcionalidad experimental. Permite que se puedan enviar paquetes hacia un nodo que hace de Gateway dentro de la red Ad-hoc.

-D: Desactivación del temporizador Wait-on-reeboot. Por defecto éste temporizador está activo durante 15 segundos.

-L: Local Repair. Activa la funcionalidad de local repair por la que un nodo intermedio intenta reparar una ruta hacia el destino si cae.

-R (rate-limit): Limita el número de paquetes RREQ y RREP transmitidos en un periodo de tiempo para no colapsar la red con paquetes de control. Por defecto está activo.