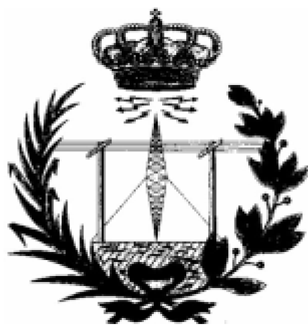
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Implementación del protocolo W2LAN como modulo kernel



Autor	Youssef Katfi
E-mail del Autor	katfiyoussef@hotmail.com
Director(es)	Francesc Burrull i Mestres
E-mail del Director	francesc.burrull@upct.es
Título del PFC	Implementación del protocolo W2LAN
Descriptores	W2lan, Kernel 2.6, Modulo kernel.

Resumen

Partiendo del modelo SDL (Specification and Description Language) del protocolo de comunicaciones W2lan se pretende integrar este ultimo como un modulo de kernel al subsistema de red de Linux/Unix. El protocolo W2lan transforma desde el punto de vista de capas superiores, una red Manet 802.11 en una Lan_Ethernet.

Titulación	Ingeniería Técnica de Telecomunicación, especialidad en Telemática
Intensificación	
Departamento	Departamento de Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	10/01/12

Índice:

Introducción.

Objetivos.

1. Presentación.

1.1 Desafíos y problemas de las redes MANET.

1.2 El estándar IEEE 802.11.

1.2.1 La capa física IEEE 802.11.

1.2.2 La capa de control de acceso al medio MAC.

1.2.2.1 El protocolo CSMA/CA.

1.2.2.2 CSMA/CA con tramas RTS/CTS.

1.3 El encaminamiento en las redes MANET.

2. El protocolo W2LAN.

2.1 Especificación del protocolo W2LAN.

2.1.1 Servicios.

2.1.2 Suposiciones.

2.1.3 Vocabulario.

2.1.4 Formato.

3. Implementación de las funcionalidades de red en el núcleo Linux.

3.1 Técnicas de implementación en el núcleo de Linux.

3.1.1 El paralelismo en Linux.

3.1.2 Concurrencia y sincronización.

3.1.3 Las listas enlazadas en el Kernel de Linux.

3.1.4 Los temporizadores.

3.1.5 Métodos de interacción con un núcleo en ejecución.

3.1.6 Gestión de la memoria.

3.1.7 Los módulos.

3.2 El subsistema de red en Linux.

3.2.1 La interfaz net_device.

3.2.2 La estructura qdisc o disciplinas de cola.

3.2.3 La estructura socket buffer (sk_buff).

3.2.4 El itinerario de un paquete.

3.2.5 Gestión de los protocolos de red.

4. Implementación del protocolo W2LAN en Linux.

4.1 Organización del modulo W2LAN.

4.2 Recepción de los datos.

4.3 Salida de los datos.

Conclusiones y líneas futuras.

bibliografía.

Introducción.

Las redes AD-HOC o MANET "Mobile ad hoc Networks", surgieron por la necesidad de conectar un conjunto de nodos móviles entre-si. Al contrario de las redes convencionales y por tratarse de redes de múltiples saltos,los equipos o nodos que forman parte de ella deben ser capaces de funcionar simultáneamente como "clientes" o "servidores" , sin la necesidad de utilizar una infraestructura fija o celular.

Durante los últimos años,se han realizado grandes esfuerzos en la investigación de las redes AD-HOC. Sin embargo, uno de los temas centrales ha sido el encaminamiento de datos,debido a que los protocolos de las redes fijas no se adaptan a este tipo de redes cuya topologia varia frecuentemente. El IETF (Internet Engineering Task Force) ha estandarizado varios protocolos de encaminamiento entre los cuales AODV [2] y OLSR[3] son los más conocidos dentro de los protocolos de encaminamientos reactivos y pro-activos.

El presente PFC se enfoca en la presentación y la implementacion del protocolo de encaminamiento W2LAN " Wireless to LAN ", que si hubiera de clasificarse, podemos considerarlo en la categoría de los protocolos reactivos a pesar de que este ultimo opera sin memoria de rutas ni de posiciones. El objetivo de W2LAN es la transformación de una red MANET desde las capas superiores a una LAN ETHERNET, resolviendo así el problema de la visibilidad parcial.

Objetivos.

El protocolo W2LAN es una nueva solución, todavía en fase de desarrollo. A día de hoy existe una versión sobre la plataforma Linux programada a nivel de usuario con las librerías Pcap. El objetivo inicial de este trabajo era la modificación del código existente,sin embargo, para dar suporte a aplicaciones de red y su integración a la pila de protocolos la programación a nivel del núcleo del sistema se hace necesaria.

En esta memoria, propondremos un prototipo de implementacion del protocolo W2LAN para Linux según la especificación descrita en[1], con el propósito de identificar las dificultades relacionadas a la programación o modificación en el núcleo de un sistema operativo y las soluciones optadas para resolver los problemas de dicha implementacion.

El resultado final es una versión operativa de W2LAN,no obstante,algunas definiciones del protocolo han evolucionado a lo largo de este trabajo pero manteniendo la idea base del funcionamiento del protocolo W2LAN.

Capitulo 1.
Presentación.

Las redes AD-HOC son sistemas distribuidos formados por un conjunto de dispositivos móviles y dinámicos que comunican entre ellos sin ningún tipo de infraestructura previa desplegada. De esta forma,los propios nodos adoptan funciones para la configuración, encaminamiento y mantenimiento de la red.

El origen de las redes AD-HOC viene por el interés del ejercito estadounidense , por el despliegue de una infraestructura de telecomunicaciones de forma rápida en medios hostiles e irregulares. En el año 1979, el DARPA patrocinó el programa de investigación PRNET(PacketsRadio Network) que trataba particularmente el problema del encaminamiento y el acceso al medio en las redes múltiples saltos por radiofrecuencia.

Durante los primeros años posteriores a su aparición ,la investigación en redes AD-HOC ha permanecido en ámbitos militares. Sin embargo, la aparición de dispositivos móviles equipados con puertos infrarrojos y radiofrecuencia puso la tecnología PRNET a la disposición del gran publico.

En los años noventa,el IEEE(Institute of Electrical and Electronics Engineers) adopto el termino redes AD-HOC para el estándar IEEE 802.11 de las WLAN y creo el grupo MANET dentro del IETF (Internet Engineering Task Force) que tiene por misión la estandarización de los protocolos de encaminamiento en la redes AH-HOC.

1.1 Desafíos y problemas de las redes MANET.

Las características principales de las redes MANET,tanto el comportamiento dinámico o el medio inalambrico para la transmisión de los datos hacen que estas redes presenten multitud de problemas que hay que considerar a la hora de su implantación.

Podemos destacar una serie de limitaciones que reducen su rendimiento :

- ***Topología variable :*** *La naturaleza de nodos móviles,provoca el cambio frecuente de la topologia de la red y por consecuencia,la dinámica construcción o destrucción de los enlaces entre los dispositivos.*

- ***Limitaciones en el ancho de banda:*** *El medio de propagación en la redes MANET no es constante en el tiempo debido a las atenuaciones e interferencias de las señales electromagnéticas. Una consecuencia de la variación del canal en el tiempo y de la longitud de la señal es la elevada tasa de error de estas redes respecto a las redes cableadas. Hay que añadir también ,el modo de operación unidireccional de los enlaces y el problema de los nodos ocultos e expuestos.*

- ***Consumo de energía:*** *El ahorro de energía en este tipo de redes es un aspecto clave, ya que algunos nodos o todos tienen baterías como fuente de alimentación.*

- ***Seguridad:*** *Las redes inalambricas y especialmente la redes MANET son vulnerables a los ataques tanto pasivos como activos. Al ser un medio compartido al que cualquiera puede tener acceso,hay que cuidar la confidencialidad de los datos.*

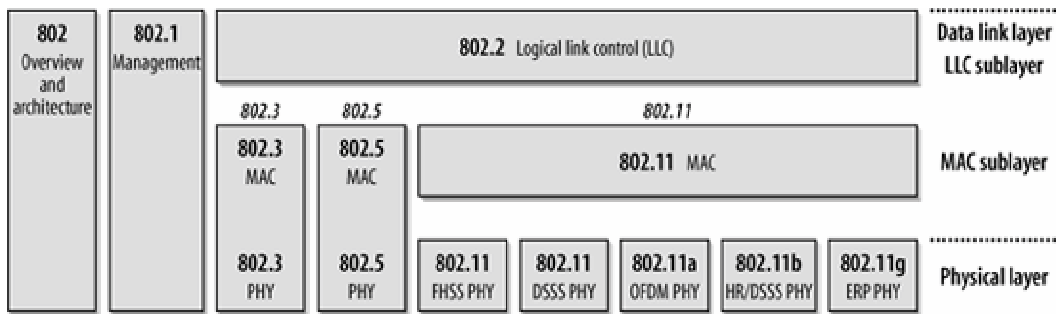
1.2 El estándar IEEE 802.11.

Las especificaciones ieee 802.11 definen las normas de funcionamiento de las redes de área local inalambricas "WLAN",operan en los niveles inferiores del modelo OSI abarcando las capas física e enlace de datos.

En la capa física,distinguimos los diferentes estándares que se encargan de la codificación y transmisión de la señal por el medio ,mientras la capa de datos esta compuesta por dos subcapas:

- ***Capa de Control Lógico de Enlace (LLC):*** *maneja las comunicaciones entre los nodos sobre una sola conexión en una red. Fue creada con el propósito de proporcionar servicios a las capas superiores independientemente de las tecnologías existentes. El LLC se define en la especificación de ieee 802.2 y soporta servicios en modo conexión y no orientado a conexión.*

- ***Capa de Control de Acceso al Medio (MAC):*** *es el mecanismo encargado de controlar el acceso de cada estación al canal compartido. Dispone de dos funciones de coordinación de acceso, la función DCF " Distributed Coordination Function" que utiliza el protocolo "CSMA/CA" como método de acceso fundamental y la función PCF " Point Coordination Function" utilizada en redes con infraestructura . Ambas pueden coexistir alternativamente utilizando diferentes intervalos de tiempo.*



1.2.1 La capa física 802.11 :

La capa física IEEE 802.11 ofrece dos técnicas para la transmisión en frecuencias de radio y una especificación para transmisiones infrarrojas.

El salto de frecuencia (Frecuency Hoping Spread Spectrum ,FHSS) *es la forma mas simple de modulación del espectro ensanchado(Spread Spectrum,SS) que consiste en aumentar el ancho de banda de la señal y reducir su potencia de pico. La información se emite en una frecuencia distinta durante un intervalo muy corto de tiempo. Esta técnica consigue una alta inmunidad a las interferencias y al ruido ambiente,sobre todo cuando usa patrones aleatorios de salto de frecuencia. Operan en la banda ISM de 2.4 Ghz y su velocidad máxima no alcanza los 2 Mbps.*

Secuencia directa (Direct Secuence Spread Spectrum,DSSS) *consiste en codificar cada uno de los bits de información con una secuencia de bits conocida como "Señal de Chip". Esta señal permite al receptor identificar los datos de un determinado emisor y que varios sistemas puedan funcionar en paralelo.*

A lo largo de los años,se han definido nuevas normas y técnicas de modulación que usan el mismo protocolo y pueden alcanzar en teoría hasta 54 Mbps . Los mas populares son aquellos definidos por las normas a,b y g a enmiendas del estándar original.

La siguiente tabla nos indica un resumen de las versiones mas empleadas del estándar IEEE 802.11.

Versión	Fecha	Modulación	Frec.(Ghz)	V.max(Mbps)
802.11 a	1999	OFDM	5	54
802.11 b	1999	CCK/DSSS	2.4	11
802.11 g	2003	OFDM/DSSS	2.4	54

1.2.2 La capa de Control de Acceso al Medio MAC:

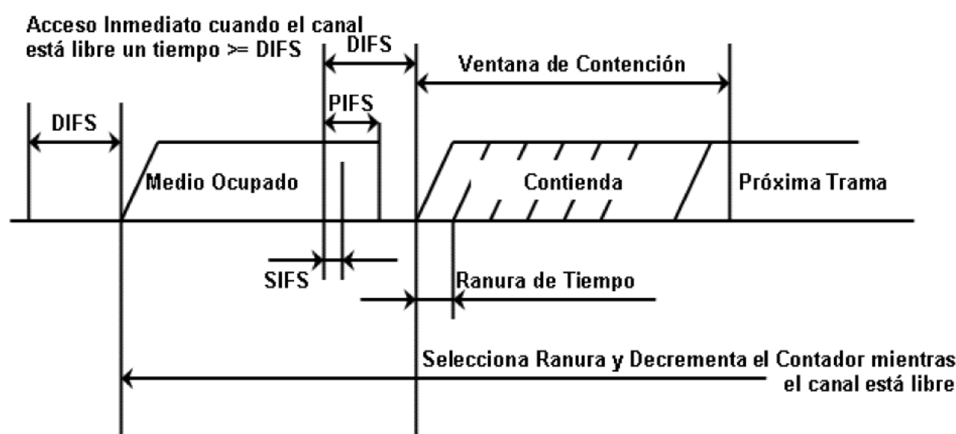
Ademas de las funciones realizadas generalmente por la capa MAC ,cabe destacar que el estándar IEEE 801.1 ofrece otros servicios que normalmente se asignan a protocolos superiores,tal como la segmentación , retransmisiones de los paquetes y la notificación de recepción ACK.

En el modo AD-HOC,se utiliza la función de coordinación DCF, que corresponde a un método de acceso bastante similar al utilizado en la redes convencionales. La función DCF fue diseñada para transportar datos de manera asíncrona donde todos los usuario tienen la misma posibilidad para acceder al medio. En el estándar IEEE 802.11,la función DCF se basa en el protocolo CSMA/CA.

1.2.2.1 El protocolo CSMA/CA:

El protocolo CSMA (Carrier Sense Multiple Access) utilizado por Ethernet,forma parte de los protocolos de competición. Cuando un nodo A quiere transmitir datos por la red,tiene que comprobar si el medio esta libre,en cuyo caso emitirá su paquete.
En el caso contrario,debe esperar el tiempo que la transmisión finalice mas un tiempo de espera aleatoria para luego volver a ejecutar el procedimiento. Así que, si un nodo B quiere también transmitir, es probable que el tiempo de espera sea diferente al nodo A .
Esta técnica sola no resuelve el problema de colisiones,ya que B puede considerar el medio como libre mientras que hay una transmisión en curso que todavía no le ha llegado.
Para reducir esta probabilidad , el IEEE 802.11 utiliza el mecanismo de Evasión de Colisiones CA (Collision Avoidance).

Como parte del mecanismo CA, antes de comenzar a transmitir se realiza un procedimiento de contienda que se muestra en la figura .



Si el canal esta ocupado,el nodo se mantiene escuchando hasta que este se quede libre

por un tiempo DIFS mas el tiempo de la Ventana de Contención (CW) cuyo valor viene determinado por un numero entero de ranuras de tiempo . Esta ventana depende del estado de las retransmisiones del paquete actual,por cada transmisión fallida se dobla el tamaño fijando el estándar un valor máximo de 1023 (Cwmax).

Finalizada la emisión,la estación debe esperar una confirmación ACK durante un intervalo de tiempo SIFS(Short Interframe Space). Transcurrido ese tiempo, si no se recibe una respuesta, se puede considerar que hubo errores en la transmisión o que se ha producido una colisión. En todo caso, la estación origen debe iniciar de nuevo el algoritmo de contención.

Conociendo las características de ubicación de cada nodo en la red,los protocolos de detección de portadora se encuentran frente a dos tipos de estaciones,los nodos ocultos e expuesto.

A - Los nodos ocultos:

La situación de una estación oculta ocurre cuando el nodo esta dentro del alcance de un nodo receptor y fuera del alcance del nodo emisor. En la figura[], los nodos A y C quieren transmitir simultáneamente datos al nodo B. Cuando A empieza a transmitir datos a B y el nodo C a su vez sondea el medio para iniciar su transmisión a B,este ultimo supone que el medio esta libre puesto que no esta en el área de cubertura de A. Sin embargo, para B hay una colisión ya que recibe datos de A y B al mismo tiempo. En este caso, se dice que el nodo C esta oculto al nodo A.

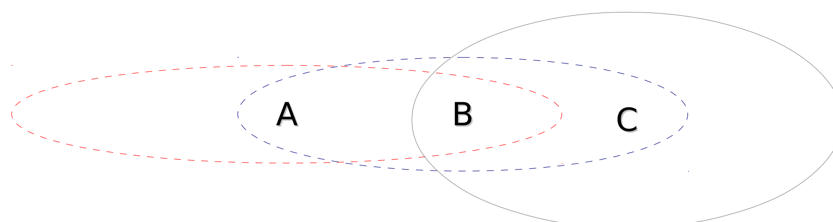


Figura 1. ejemplo de la situación de un nodo oculto.

B – Los nodos expuestos:

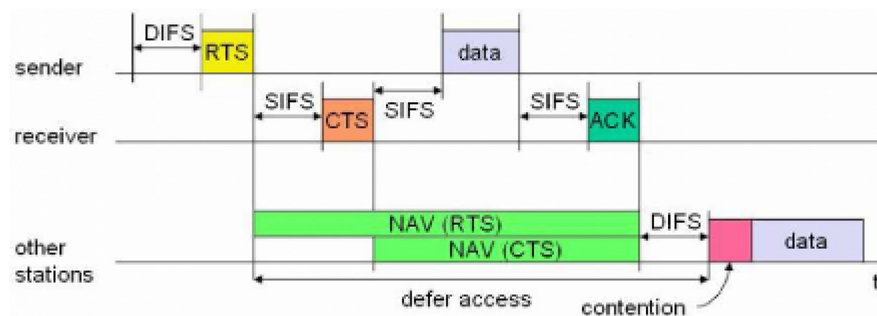
En el caso de los nodos expuestos, el nodo esta dentro del alcance de un nodo emisor y fuera del alcance del nodo receptor. En la figura anterior, consideremos el caso en que el nodo B intenta transmitir datos al nodo A, como el nodo C esta en el alcance del nodo B supone que el medio esta ocupado y por lo tanto, el problema de los nodos expuestos reside en que el nodo C puede tener datos a enviar paralelamente a otros nodos que no están en el alcance de los nodos A y B. Se considera en este caso que el nodo C esta expuesto a B.

Con el fin de reducir el problema de los nodos ocultos e expuestos,el IEEE 802.11 utiliza una extensión del protocolo CSMA/CA con el mecanismo de sensado de portadora virtual (VCS).

1.2.2.2 CSMA/CA con tramas RTS(Request To Send)/CTS(Clear To Send):

En este método de control,se enviá previamente una trama RTS, notificando que hay datos pendientes a transmitir . El nodo destino responde con una trama CTS que confirma el inicio de envío.

Las tramas RTS/CTS incluyen el periodo total de la transmisión,es decir, el tiempo necesario para la emisión de la trama de datos y el ACK. Este información, puede ser leída por cualquier nodo que esta en el rango de cobertura del origen o del destino,por lo cual actualiza su Vector de Asignación de Red (NAV) que provee un mecanismo CSMA virtual.



1.3 El encaminamiento en las redes MANET.

Como ya hemos señalado anteriormente, las MANET presentan una serie de limitaciones o problemas y son redes de múltiples saltos, esto es, la posibilidad de que un nodo quisiera comunicarse con otro que no esta en su rango de comunicación. Los protocolos de encaminamiento son aquellos que encuentran la ruta que debe seguir los datos desde la fuente hasta su destino. Estos protocolos trabajan a nivel de red y son totalmente independientes de las capas inferiores. El encaminamiento IP permite particularmente una interconectividad fácil entre otros tipo de redes o materiales.

En el modo AD-HOC, no existe una única estrategia de encaminamiento posible o valida sino que cada uno de los distintos enfoques realizados hasta ahora puede resultar especialmente adecuado para un tipo de escenario en concreto. Según la forma de creación y mantenimiento de los enlaces los protocolos de encaminamiento en redes MANET se clasifican en tres categorías:

A -Protocolos proactivos:

Los protocolos proactivos buscan obtener información actualizada de la topologia de la red para poder proporcionar una respuesta inmediata a un pedido de ruta, eso significa que todos los nodos conocen un camino hacia todos los destinos. La ventaja principal de este tipo de protocolos es tener una ruta inmediata cuando es requerida por las aplicaciones,esto nos permite por ejemplo seleccionar rutas con menos saltos, mas ancho de banda o incluso aquellas con el menor tiempo de retardo, pero el intercambio periódico de información de enrutamiento genera trafico adicional y consumo en los recursos de los nodos.

B -Protocolos reactivos:

El principio de los protocolos reactivos es no hacer nada mientras que ninguna aplicación solicite explicitamente el envió de un paquete hacia un nodo distante. Esto permite ahorrar tanto el uso del ancho de banda o el consumo de la energía. Cuando se debe enviar datos,los protocolos reactivos buscaran un camino hasta el destino. Una vez encontrado,actualizan sus tablas de enrutamiento y es entonces cuando es posible la comunicación con dicho nodo. En general, esta búsqueda implica alguna clase de inundación de la red con la petición de la ruta.

C -Protocolos híbridos:

Los protocolos híbridos combinan el principio de los protocolos reactivos y proactivos.
La búsqueda de los nodos vecinos se hace de una manera proactiva hasta una cierta distancia o numero de saltos, fuera de este rango se hace de manera reactiva.
En función del tipo de trafico y de las rutas solicitadas,estos protocolos combinan también las desventajas de los dos métodos: el intercambio de paquetes de control y la inundación de la red en el caso de un nodo lejano.

Capitulo 2.
El protocolo W2LAN.

W2LAN es un protocolo que transforma desde el punto de vista de capas superiores una red 802.11x MANET en una red LAN Ethernet. Éste surgió para dar suporte al protocolo MCDP-LAN "Multimedia content Distribution Protocol over LAN" encargado de la distribución de contenidos multimedia en un entorno SOHO"Smalle Office/Smalle Home", es decir mediante una red LAN. Así usando el protocolo W2LAN cualquier par de terminales inalambricos que pertenecen a la misma LAN se verán entre ellos "visibilidad total" siempre que al menos exista una ruta entre ellos.

2.1 Especificación del protocolo W2LAN.

2.1.1 Servicios.

El servicio que el protocolo W2LAN ofrece es dotar de visibilidad total a los nodos pertenecientes a una red Ad-Hoc determinada. Cuando un nodo tiene una trama Ethernet preparada para transmitir, en lugar de transmitir la trama directamente, ésta es entregada a la capa W2LAN, que de inmediato iniciará una nueva conversación W2LAN. De manera complementaria, cuando un nodo ha recibido una conversación completa, su capa W2LAN entregará la trama Ethernet asociada a dicha conversación igual que lo haría un dispositivo Ethernet normal.

2.1.2 Suposiciones.

El protocolo W2LAN se basa en 2 suposiciones. La primera es que opera sobre dispositivos wireless Ethernet en modo Ad-Hoc. Esto significa que los nodos deben de tener al menos un dispositivo wireless configurado correctamente, tal como se haría en una red Ad-Hoc sin W2LAN. La segunda suposición es que debe existir al menos una ruta posible entre cualquier par de nodos. El protocolo sigue funcionando en redes fragmentadas, tratándolas como redes aisladas. Ahora bien, se sobreentiende que no es el caso deseado, ya que el hecho de usar W2LAN lleva implícito que se está intentando solucionar un problema de visibilidad parcial.

2.1.3 Vocabulario.

Para describir el vocabulario es suficiente con desglosar las transacciones. Una transacción W2LAN consiste en una tripleta {anuncio, solicitud, datos} que transportará tramas Ethernet a los distintos nodos vecinos interesados.

El vocabulario del protocolo W2LAN necesario para construir transacciones W2LAN será:

*- **Anuncio:** El mensaje anuncio es utilizado cuando un nodo tiene una nueva trama a transmitir, ya sea proveniente de capas superiores o proveniente de una conversación acabada de recibir que debe ser retransmitida. Este mensaje lleva asociado la dirección MAC origen del nodo, un identificador (ID) de conversación, la cabecera y los seis primeros bytes del campo datos de la trama Ethernet que se está distribuyendo.*

*- **Solicitud:** El mensaje solicitud se utiliza la primera vez que un nodo recibe un anuncio de una conversación nueva. Este mensaje lleva asociado la dirección MAC destino del nodo (o de manera equivalente, la dirección MAC origen del nodo que mandó el correspondiente anuncio) y el identificador de conversación correspondiente.*

*- **Datos:** El mensaje datos se utiliza si un nodo que ha anunciado previamente una conversación recibe alguna solicitud de la conversación anunciada. Este mensaje lleva asociado el correspondiente identificador de conversación y el campo de datos de la trama Ethernet (a excepción de los seis bytes que se enviaron junto la trama de anuncio) que se está distribuyendo. El formato de cada mensaje se ha diseñado para que sea compatible con Ethernet, teniendo en cuenta en todo momento la simplicidad. Así, cada mensaje encaja en una trama Ethernet, evitando fragmentar los mensajes.*

2.1.4 Formato.

El formato de cada mensaje se ha diseñado para que sea compatible con Ethernet, teniendo en cuento en todo momento la simplicidad. Así, cada mensaje encaja en un trama Ethernet, evitando fragmentar los mensajes.

Los campos de cada mensaje son :

-Announce (34 Bytes) *:*

- *Campos de la cabecera Ethernet:*

  - Dirección destino (6 bytes). En las tramas announce siempre es la dirección de broadcast FF:FF:FF:FF:FF:FF.

  -Dirección origen (6 bytes). Corresponde con la dirección MAC del nodo que transmite.

  -Tipo de datos (2 bytes). Se utiliza el valor 0x6901 para identificar una trama announce.

- *Campos del cuerpo Ethernet:*

  - Conversation ID (6 bytes). Es un identificador único,que es compartido por toda trama perteneciente a una conversación determinada.

  -Dirección MAC destino(6 bytes) .Corresponde con la dirección MAC destino de la trama Ethernet que se esta esparciendo.

  -Dirección MAC origen(6 bytes) . Corresponde con la dirección MAC origen de la trama Ethernet que se esta esparciendo.

  -Tipo de datos (2 bytes). Corresponde con el campo de tipo de datos de la trama Ethernet que se esta esparciendo.

-Request (20 Bytes) *:*

- *Campos de la cabecera Ethernet:*

  - Dirección destino (6 bytes). En las tramas request corresponde con la dirección MAC del nodo que anteriormente anuncio la conversación que se esta llevando a cabo.

  -Dirección origen (6 bytes). Corresponde con la dirección MAC del nodo que transmite.

  -Tipo de datos (2 bytes). Se utiliza el valor 0x6902 para identificar una trama request.

- *Campos del cuerpo Ethernet:*

  - Conversation ID (6 bytes). Es un identificador único,que es compartido por toda trama perteneciente a una conversación determinada.

-Data(igual tamaño que el campo de datos Ethernet original mas el convID) *:*

- *Campos de la cabecera Ethernet:*

  - Dirección destino (6 bytes). En las tramas data siempre es la dirección de broadcast FF:FF:FF:FF:FF:FF.

  -Dirección origen (6 bytes). Corresponde con la dirección MAC del nodo que transmite.

  -Tipo de datos (2 bytes). Se utiliza el valor 0x6903 para identificar una trama data.

- *Campos del cuerpo Ethernet:*

  - Conversation ID (6 bytes). Es un identificador único,que es compartido por toda trama perteneciente a una conversación determinada.

  - Campo de datos de la trama Ethernet (payload). Corresponde exactamente con el campo de datos de la trama Ethernet que se esta distribuyendo.

Anuncio:

FF:FF:FF:FF:FF:FF	MAC orig	6901	convID	MAC destino	MAC origen	ETH. T

Solicitud.

MAC dest(*)	MAC orig	6902	convID

Datos.

FF:FF:FF:FF:FF:FF	MAC orig	6903	convID	Datos Ethernet

(*) MAC dest coincide con MAC orig de la trama Anuncio

Capitulo 3.

Implementacion de las funcionalidades de red en el núcleo Linux.

En este capitulo, estudiaremos la arquitectura de red en el núcleo de Linux y nos centraremos particularmente en las partes que debemos modificar para dar suporte al protocolo W2LAN. Vamos a empezar por introducir diversos elementos específicos a la programación del núcleo después veremos mas detalladamente el funcionamiento de la arquitectura de red.

3.1 Técnicas de implementacion en el núcleo de Linux.

En esta sección ,vamos a comentar brevemente algunas técnicas de uso frecuente en el núcleo de Linux que no estamos acostumbrados de manejar en el contexto de usuario. El objetivo de esta parte es ofrecer los elementos necesarios para la comprensión del código descrito en el siguiente capitulo, para mas información podemos encontrar buena documentación que introduce a la programación del núcleo [2].

3.1.1 El paralelismo en Linux.

El núcleo de Linux esta concebido para funcionar tanto en equipos con un solo procesador UP(Uni-Processor) como en equipos con varios procesadores SMP (Symmetric Multi-

Processors).
A partir de las versiones 2.6, el núcleo se ha convertido en preemptible ,eso quiere decir que un equipo UP puede adoptar un comportamiento similar a los SMP.

Para tener una idea de las consecuencias del paralelismo,debemos tener conocimiento de las diferentes formas de actividades existente en el núcleo y su manera de ser interrumpida.

Distinguimos:

Las interrupciones hardware (Hard IRQ) : *son ejecutadas como su nombre lo indica a petición del hardware. Deben ser tratadas de forma rápida y no pueden ser interrumpidas. El código de la rutina que trata este tipo de interrupciones tiene que ser muy corto, dada su prioridad elevada el sistema se bloquea durante su ejecución.*

Las interrupciones software (Soft IRQ) : *son tareas de ejecución diferida que no necesitan bloquear el sistema mientras que están invocados por este ultimo. Una soft IRQ puede ser interrumpida por una Hard IRQ durante su ejecución.*

Un ejemplo en el caso de las redes es la llegada de una trama a un dispositivo de red que activa una interrupción hardware encargada de hacer solo el trabajo urgente que consiste en copiar la trama en memoria y activar a su vez la interrupción software NET_RX. En cuanto sea posible se ejecuta la tarea asociada de manera que la trama pueda atravesar parte o toda la pila IP/TCP hasta llegar a un proceso del espacio usuario.

La interrupción NET_RX gestiona a la vez la recepción ,el encaminamiento y el envió de un paquete a través de otra interfaz. Mientras que la interrupción NET_TX sera activa cuando haya una trama en la cola de espera lista para el envió.

3.1.2 Concurrencia y sincronizacion.

En todo sistema que cuenta con acceso concurrente a la memoria se debe asegurar la consistencia de los datos. A causa de la grand variedades de situaciones que podemos encontrar en el núcleo de Linux,existe también una grand variedades de cerrojos.

Semáforos (include/asm/arch/semaphore.h): *permiten el acceso de un numero limitado de tareas a una zona en concreto del código. En la practica, queremos que este numero se traduce a una sola tarea que es el caso de los Mutex.*
Los Mutex se declaran y se utilizan de la siguiente forma:

```
/* Declaraciones */
  void sema_init(struct semaphore *sem,int val);
  DECLARE_MUTEX(cerrojo);
  DECLARE_MUTEX_LOCKED(cerrojo);

  if (down(&cerrojo))  return -EINTR;
  if (down_interruptible(&cerrojo))  return -EINTR;
  /*Zona critica*/
  ….....
 __función critica();
  up(&cerrojo);
```

Si un proceso A ejecutado en el contexto de usuario logra obtener el cerrojo mediante la funcion down() o down_interruptible() ,obtiene también el acceso a la sección critica del código y todo otro proceso que intenta a su vez obtener el cerrojo sera dormido hasta que el primero haya terminado (up).
Los semáforos son los únicos en pasar al estado "sleeping" y por lo tanto su uso se limita al contexto usuario.
La función down_interruptible() permite ademas de reclamar el cerrojo,la recepción de señales mientras que el proceso se encuentra dormido.

Spinlocks (include/linux/spinlock.h): *Los semáforos por su limitación al contexto de usuario son muy poco utilizados. En la mayoría de los casos, los datos compartidos son gestionados por medio de las interrupciones software, o a veces combinándolas con el contexto usuario.*

Los Spinlocks se declaran y se utilizan de la siguiente forma:

```
/* Declaraciones y inicializacion */
  spinlock_t  MyLock;
  .........
  spinlock_lock_init(&MyLock);


/* Acceso a la sección critica */
  spin_lock(&MyLock);
  …......
 spin_unlock(&MyLock);
```

spin_lock() es una macro, que es transformada según si trabajamos en modo UP, SMP o *preemptible. En el modo SMP, el paralelismo es real y la macro spin_lock() hace un test continuo al cerrojo hasta que este se encuentre libre "Busy waiting", mientras que en el modo preemptible se utiliza el pseudoparalelismo. Finalmente en el modo UP las interrupciones software están por defecto desactivadas.*
El uso elegante del pre-procesador nos permite realizar código portable suponiendo que siempre trabajamos sobre SMP. Las macro permiten la ejecución de este código en en maquinas UP.

En el caso de una estructura de datos manipulada tanto dentro de una interrupción software así como en el contexto usuario utilizamos entonces la siguiente variante de los Spinlocks para evitar el "deadlock":

```
/* Acceso a la sección critica */
 spin_lock_bh(&MyLock);
 …....
 spin_unlock_bh(&MyLock);
```

s*pin_lock_bh() desactiva las interrupciones softwares en el procesador local únicamente para no bloquear en el contexto usuario.*

Rwlocks (include/linux/spinlock.h): *son una variante de lectura/escritura de los spinlocks que permiten tener varios lectores simultáneamente pero con un único escritor.*

```
/* Declaraciones y inicializacion */
   rwlock_t  MyLock;
   .........
   rwlock_t _init(&MyLock);


/* Modificación la sección critica */
   write_lock(&MyLock);
   …......
  write_unlock(&MyLock);

/* Acceso a la sección critica */
  read_lock(&MyLock);
  …......
  read_unlock(&MyLock);
```

Al igual que los spinlocks,la variante _bh existe también para los rwlocks ya que el problema es idéntico al caso anterior.

Operaciones atómicas (include/asm/arch/atomic.h): *nos permiten proteger un numero entero de 32 bits, generalmente es un contador al que accedemos y modificamos mediante operaciones de tipo "test and set" atómicas. Este mecanismo se puede implementar con los spinlocks, pero las operaciones atómicas ofrecen una grand claridad del código y mejor provecho de las posibilidades ofrecidas por el procesador. El siguiente ejemplo vemos la equivalencia conceptual de los dos métodos:*

atomic_t numero;
/* inicializacion */
atomic_set(&numero,10);
int x = atomic_inc_return(&numero);
	/* inicializacion */
	int numero = 10;
	spin_lock(&Cerrojo);
	numero++;
	spin_unlock(&Cerrojo);

Una utilización frecuente de esta interfaz en el núcleo son los contadores de referencia que veremos a continuación.

El contador de referencia: *es una técnica clásica que nos permite asegurarnos que nadie posee un puntero hacia un objeto antes de suprimirlo. En Linux esta técnica consiste en poner un contador "***refcnt (referencia del contador)***" en la estructura que queremos proteger, esta variable atómica es incrementada cada vez que se crea una nueva referencia del objeto y decrementada cuando es liberado. Los datos son totalmente eliminados de la memoria cuando refcnt vale 0.*

Un ejemplo del mecanismo del contador de referencia se puede implementar de la siguiente forma:

```
struct  data{
/* Campo de datos */
    …...
   atomic_t refcnt;
};

struct data get_data(){
   struct data *p;
   spin_lock(&cerrojo);
   .......
  /* incrementamos  el contador */
  atomic_inc(&p->refcnt);
  spin_unlock(&cerrojo);
  return p;
}

 void use_data(){
 struct data *p = get_data();
  /* uso de datos * /
    …....
  if(atomic_dec_and_test(&p->refcnt) kfree(p);
}


void delete_data(struct data *p){
  spin_lock(&cerrojo);
  if(atomic_dec_and_test(&p->refcnt) kfree(p);
  spin_unlock(&cerrojo);
}
```

La función get_data() es la responsable de devolver un puntero del objeto y incrementar el contador de referencia. En la función que utiliza el puntero,una vez que hayamos terminado se decrementa el refcnt y estaremos frente a 2 posibles situaciones:

- *Si el objeto no ha sido eliminado mediante la función delete_data(), entonces refcnt vale generalmente 1 (salvo si otras instancias tienen el puntero del objeto) y la memoria no es liberada.*
- *En el caso si se llama a la función de supresión delete_data() y el objeto esta siendo usado, el refcnt es decrementado a uno y la ultima instancia que llama a la función use_data() liberara la memoria.*

Dado que esta técnica se utiliza ampliamente en todo el núcleo de Linux, una estructura se propuso para unificar la gestión del contador de referencia y también aumentar la claridad del código y disminuir los errores. Se trata de una interfaz simple disponible en ***include/linux/kref.h .***

El ejemplo anterior se convierte en :

```
struct  data{
/* Campo de datos */
…...
struct  kref kref ;
};
/* inicializacion */
kref_init(&data->kref);


struct data get_data(){
  struct data *p;
  spin_lock(&cerrojo);
   .......
  /* incrementamos  el contador */
  kref_get(&p->refcnt);
  spin_unlock(&cerrojo);
  return p;
}

void use_data(){
struct data *p = get_data();
  /* uso de datos * /
  …....
  kref_put(&p->kref, data_release);
  spin_unlock(&cerrojo);
}

void delete_data(struct data *p){
  spin_lock(&cerrojo);
  kref_put(&p->kref, data_release);
  spin_unlock(&cerrojo);
}

void data_release(){

struct data *p = container_of(kref, struct data,kref);
kfree(p);
}
```

kref_put recibe como argumento la función de supresión del contexto ,cada vez que llamamos la función es muy probable eliminar los datos de la memoria si el contador vale 0.

3.1.3 Las listas enlazadas en el kernel de Linux .

Las listas enlazadas son estructuras de datos básicas empleadas en el interior del núcleo de Linux. Aunque su uso es frecuente en el mundo de la programación,su implementacion en el núcleo es muy particular. La definición genérica de estas listas viene en el fichero ***include /linux/list.h:***

```
 19struct list_head {
 20        struct list_head *next, *prev;
 21};
```

Como vemos a primera vista,la estructura de la lista carece de los campos de datos porque a diferencia de las listas enlazadas que utilizamos habitualmente son los datos quienes llevan la estructura de la lista en este caso. En el siguiente ejemplo imaginamos unas estructuras de datos del mismo tipo T que queremos enlazar entre si, para ello,definimos un campo list_head dentro de nuestra estructura T como viene a continuación:

```
struct T{
 int num;
 ...
 struct list_head lista_T;
 };
```

Para iniciar una lista vaciá disponemos de muchas macros que nos ofrece la interfaz ***list.h*** *como LIST_HEAD_INIT(name) o bien LIST_HEAD(name) que crean una variable list_head de nombre name y los punteros *next y *prev apuntando a ellos mismos.*

La figura siguiente ilustra un ejemplo de una lista de dos elementos.

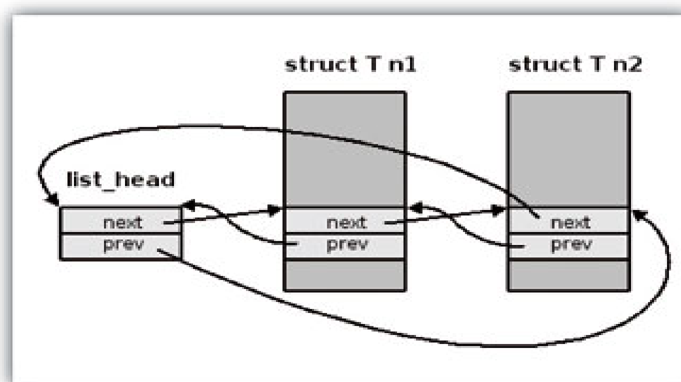


fig : Lista doblemente enlazada de 2 elementos.

*El enlace de las estructuras T se hace gracias al elemento lista_T, pero el problema que surge es saber como encontrar la estructura de datos cuando estamos recorriendo la lista. La solución del problema es un truco que consiste en sustraer a la dirección de lista_T su offset dentro de la estructura de datos, de esta manera obtendremos la dirección del comienzo de la estructura T. Para recuperar el offset disponemos de la siguiente macro definida en **include/linux/stddef.h:***

```
 24#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

*Para el compilador GCC, el numero 0 es considerado como la dirección de una estructura del tipo TYPE. La macro devuelve entonces el offset que tendrá el miembro MEMBER puesto que la dirección de la estructura es nula. A continuación vemos la implementacion de la macro definida en **include/linux/kernel.h** que devuelve la dirección de una estructura a partir del tipo, nombre del miembro que contiene el nodo list_head y su dirección:*

```
715 /**
716  * container_of -cast a member of a structure out to the containing structure
717  * @ptr:  the pointer to the member.
718  * @type: the type of the container struct this isembedded in.
719  * @member:     the name of the member within the struct.
720  *
721  */
722  #define container_of(ptr, type, member) ({                    \
723        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
724        (type *)( (char *)__mptr - offsetof(type,member) );})
```

*Sin embargo, la interfaz list.h nos ofrece la función **list_entry** que corresponde exactamente a container_of:*

```
 342/**
 343 * list_entry - get the struct for this entry
 344 * @ptr:        the &struct list_head pointer.
 345 * @type:       the type of the struct this is embedded in.
 346 * @member:     the name of the list_struct within the struct.
 347 */
 348#define list_entry(ptr, type, member) \
 349        container_of(ptr, type, member)
```

Entre las funciones ofrecidas,la función list_add nos permite insertar un nuevo elemento al principio de nuestra lista.

```
 64static inline void list_add(struct list_head *new, struct list_head *head)
 65{
 66        __list_add(new, head, head->next);
 67}
```

La funcion list_for_each_entry nos permite iterar de manera simple dentro de una lista.

```
414/**
415 * list_for_each_entry -      iterate over list of given type
416 * @pos:       the type * to use as a loop cursor.
417 * @head:      the head for your list.
418 * @member:    the name of the list_struct within the struct.
419 */
420#define list_for_each_entry(pos, head, member)                \
421      for (pos = list_entry((head)->next, typeof(*pos), member); \
422          prefetch(pos->member.next), &pos->member != (head);  \
423          pos = list_entry(pos->member.next, typeof(*pos), member))
```

Volviendo al ejemplo anterior de una lista doblemente enlazada de dos elementos,una posible solución seria:

```
/**
 *    Ejemplo de una lista de 2 nodos.
 */

struct T{
 int num;
 …...
 struct list_head lista_T;
 };

struct T *n1;
struct T *n2;
struct T *ptr;


struct list_head lista_cab;   /* Cabecera de la lista */

INIT_LIST_HEAD(&lista_cab);  /* iniciamos la lista */

n1 = kmalloc (sizeof(struct T));
n1->num = 20;
n2 = kmalloc (sizeof(struct T));
n2->num = 10;

list_add (&n2->lista_T,&lista_cab);
list_add (&n1->lista_T,&lista_cab);



int contador = 0;
list_for_each_entry(ptr,&lista_cab,lista_T){
    contador++;
    printk("Nodo: %d ------> Numero = %d \n",contador,ptr->num);
}
```

El resultado de la iteracion seria : Nodo: 1 -------> Numero = 20
Nodo: 2 -------> Numero = 10

3.1.4 Los temporizadores:

*La función principal de un temporizador es la ejecución de una tarea en un instante preciso. En el fichero **include/linux/timer.h** tenemos la definición de la estructura timer_list y otras funciones para la inicializacion,modificación o la eliminación de un temporizador.*

Entre los campos de la estructura timer_list ,la variable expires especifica el instante en el cual la función diferida sera ejecutada. La variable global Jiffies entera de 32 bits representa el numero de "ticks" transcurridos desde el arranque del sistema y es incrementada a cada interrupción de tiempo, hay HZ interrupciones de tiempo en el intervalo de un segundo. La frecuencia HZ "Tike_Rate" es configurada en el inicio del sistema según un valor estático que depende de la arquitectura de la maquina, en la versión genérica este valor vale 100.

El campo "function" es un puntero a la función diferida, mientras que data es una variable de tipo unsigned long que por su tamaño se usa como argumento de la función diferida para almacenar un puntero a una estructura completa, con un simple "Casting" recuperamos la estructura dentro de la rutina del temporizador.

A continuación un ejemplo de un temporizador que se ejecuta en 5 segundos:

```
 struct  datos{
 /* Campo de datos */
 …...
 };


 struct func_diferida (unsigned long data){
     struct datos *p = (struct datos *)data;
     /* código de ejecución diferida */
    …...
   }
int tiempo = 5 ; /* 5 segundos */
struct timer_list temporizador;
/* Inicializacion */
init_timer(&temporizador);

temporizador.expires  = jiffies + tiempo *HZ;
temporizador.function = &func_diferida;
temporizador.data     = (unsigned long)datos;

/* Activación del Timer */
add_timer(&temporizador);
```

3.1.5 Métodos de interacción con un núcleo en ejecución :

la interacción con el núcleo en ejecución nos permite extraer información sobre las diferentes actividades del sistema mediante los archivos de registros o como se conocen comúnmente los archivos de log. En la carpeta ***/var/log*** *encontramos en general casi todos los archivos de registros del sistema,sin embargo,muchas aplicaciones crean sus propios archivos fuera de esta carpeta. Ademas de monitorizar el sistema, existen también otros medios para modificar dinamicamente su comportamiento.*

Uso de printk () (include/linux/kernel.h): *es la variante de printf para el núcleo Linux,que ofrece entre sus características la posibilidad de clasificar los mensajes que muestra por su prioridad o criticidad mediante la variable* ***loglevel.***
El nivel de depuración es especificado al principio de la cadena de caracteres como se muestra en el siguiente ejemplo:

printk(<0> " Mensaje Urgente");
printk(<7> "depuración ");

Pero es recomendable utilizar las macros KERN_XXX para una mayor claridad del código.

printk(KERN_EMERG " Mensaje Urgente");
printk(KERN_DEBUG " depuración ");

En todo caso, los mensajes del núcleo son tratados mediante los deamons ***klogd*** *y* ***syslogd*** *que según su configuración los enviaran a uno o mas archivos de registro.*

El sistema de ficheros virtual procfs (include/linux/proc_fs.h): *es una capa de abstracción y unificación que nos permite obtener el "estado" de las diferentes partes del sistema, como por ejemplo la visualización de las tablas ARP en el caso de las redes. Este sistema esta montado en* ***/proc,*** *que es un repertorio virtual dado que los datos no están realmente guardados en ningún medio de almacenamiento y los ficheros se cargan dinamicamente en la memoria cada vez que se acceda a ellos.*
A continuación, vemos como el protocolo ARP crea sus tablas en ***net/ipv4/arp.c.***

```
1404 static const struct seq_operations arp_seq_ops = {
1405         .start  = arp_seq_start,
1406         .next   = neigh_seq_next,
1407         .stop   = neigh_seq_stop,
1408         .show   = arp_seq_show,
1409 };
1410
1411 static int arp_seq_open(struct inode *inode, struct file *file)
1412 {
1413         return seq_open_net(inode, file, &arp_seq_ops,
1414                             sizeof(struct neigh_seq_state));
1415 }
```

```
1417 static const struct file_operations arp_seq_fops = {
1418        .owner          = THIS_MODULE,
1419        .open           = arp_seq_open,
1420        .read           = seq_read,
1421        .llseek         = seq_lseek,
1422        .release        = seq_release_net,
1423 };
```

```
1426 static int __net_init arp_net_init(struct net *net)
1427 {
1428        if (!proc_net_fops_create(net, "arp", S_IRUGO, &arp_seq_fops))
1429                return -ENOMEM;
1430        return 0;
1431 }
```

la función proc_net_fops_create() crea un nueva entrada en el repertorio ***/proc/net/*** *con el nombre arp y los permisos de solo lectura . La estructura file_operations contiene como campos los punteros de las funciones para la manipulación del fichero. Esta estructura en realidad esta mas grande de lo que aparece pero gracias a una extensión especifica al compilador los demás campos se inicializan por defecto con el valor NULL. En este caso,arp define solamente las funciones "open","read","llseek" y "release".*

Finalmente la estructura seq_operations es una interfaz de 4 métodos que sirve para poder establecer una posición o moverse dentro de la secuencia gracias a los método start y next. Una posición en la secuencia representa habitualmente una posición dentro de una lista de estructuras tal como una lista enlazada o una tabla hash. Los datos se transmiten al fichero mediante el método show.

Aunque este mecanismo no esta implementado en esta versión de W2LAN,se puede usar como futuro trabajo para determinar por ejemplo los terminales conectados en cada nodo.

La interfaz sysctl (include/linux/sysctl.h) : *nos ofrece la posibilidad de modificar dinamicamente el comportamiento del sistema. El conjunto de la variables modificables o consultables son accesibles en el repertorio* ***/proc/sys*** *que forma parte del sistema de ficheros procfs.*

Un ejemplo de red del uso de la interfaz sysctl es de la activación del redireccionamiento de paquetes ip mediante el comando sysctl:

sysctl -w net.ipv4.ip_forward = 1
o
echo 1 > /proc/sys/net/ipv4/ip_forward

su implementación en el núcleo de linux viene en el fichero ***net/ipv4/devinet.c*** *donde tenemos la inicializacion de la estructura* ***ctl_table :***

```
1508 static struct ctl_table ctl_forward_entry[] = {
1509         {
1510             .procname      = "ip_forward",
1511             .data          = &ipv4_devconf.data[
1512                                 IPV4_DEVCONF_FORWARDING - 1],
1513             .maxlen        = sizeof(int),
1514             .mode          = 0644,
1515             .proc_handler  = devinet_sysctl_forward,
1516             .extra1        = &ipv4_devconf,
1517             .extra2        = &init_net,
1518         },
1519         { },
1520 };

1522 static __net_initdata struct ctl_path net_ipv4_path[] = {
1523         { .procname = "net", },
1524         { .procname = "ipv4", },
1525         { },
1526 };
```

los parámetros de la estructura son los siguientes:

- procname : *el nombre del fichero dentro del repertorio **/proc/sys/net/ipv4***

- data : *la dirección del dato mediante la interface sysctl.*

- maxlen : *tamaño máximo permitido a partir de la dirección **data.** Es una forma de controlar el acceso al núcleo a partir del espacio usuario.*

- mode : *los permisos de acceso al archivo, en este caso con los derechos de lectura y escritura.*

- proc_handler : *es el puntero de la función que hay llamar por cada lectura o escritura del fichero. Existen funciones predefinidas en el fichero **sysctl.h** para modificar diferente tipos de datos,como por ejemplo, la función **proc_dointvec** en el caso de un entero.*

- extra1,2 : *son punteros adicionales utilizados por algunas rutinas del controlador proc.*

*La entrada "ip_forward" es registrada gracias a la función **register_net_sysctl_table(),** que básicamente y sin entrar mucho en detalles, añade un nuevo espacio de nombre "namespace" y la estructura ctl_table en la lista interna de sysctl.*

3.1.6 Gestión de la memoria :

la gestión de la memoria en el modo kernel no difiere sustancialmente de la manera en que se realiza en el espacio usuario o mediante las librerías estándares (desde la perspectiva de un programador de sistemas): asignación , liberación de la memoria, copia entre el espacio de direcciones del núcleo y del usuario y finalmente el uso de la memoria cache.

- Asignación de la memoria mediante la función kmalloc(size,priority):

La función kmalloc() intenta reservar una zona de memoria conexa de size octetos en el espacio de direccionamiento del núcleo. Algunos octetos suplementarios son reservados a causa de la memoria cache que veremos mas adelante.
El argumento "priority" permite indicar las opciones de asignación de memoria que se definen en ***linux/gfp.h.***

- GFP_USER : *para asignar memoria en el espacio usuario.*

- GFP_KERNEL : *para asignar memoria en el espacio kernel, la función que hace esta asignación debe ser reentrante y puede ser interrumpida a lo largo del proceso.*

- GFP_ATOMIC : *es análoga a GFP_KERNEL salvo que la asignación de la memoria se atribuye de manera automática, es decir, no puede ser interrumpida. Se usa en el caso de obtener memoria desde una interrupción o timer .*

- GFP_DMA : *para un acceso directo a la memoria.*

Existe también la función vmalloc() que trabaja de forma similar a kmalloc(), salvo que asigna memoria que solamente es contigua virtualmente.
La liberación de la memoria se realiza a través de la llamada a la función kfree().

- Memoria cache (slab alloctor):

la asignación de la memoria mediante la llamada al sistema kmalloc() puede demorar un tiempo. Si se espera que una zona de memoria del mismo tamaño sera necesaria en breve, conviene en lugar de liberar la zona de memoria con la llamada del sistema kfree(),guardar los datos en una lista para su utilización bajo demanda.
Este modo de operación esta implementado en Linux por medio del "slab allocator".

*La función **kmem_cache_create()** devuelve un descriptor de cache del tipo **kmem_cache** para un tipo de datos en particular y esta definida en **mm/slab.c** :*

```
2166 struct kmem_cache *
2167 kmem_cache_create (const char *name, size_t size, size_t align,
2168         unsigned long flags, void (*ctor)(void *))
```

describimos a continuación los argumentos de esta función :

- name :** nombre que identifica la cache en el directorio **/proc/slabinfo

***- size :** el tamaño de los objetos o datos creados.*

***- align :** la alineación de los datos en la memoria. Este parámetro toma un valor nulo la mayor parte del tiempo.*

***- flags :** si la opción CONFIG_SLAB_DEBUG esta activada podemos eligir algunos parámetros adicionales de depuración o bien, especificar la forma de almacenamiento de los datos.*
Por ejemplo la macro :

- *SLAB_CACHE_DMA : permite un acceso directo a la memoria.*
- *SLAB_HWCACHE_ALIGN : para alinearse en el primer nivel L1 de la memoria cache del procesador "cpu".*

***- ctor :** es un puntero al constructor del objeto si esta definido,sino toma el valor null.*

*Una vez creada la cache podemos asignar los objetos gracias a **kmem_cache_alloc(),** esta función si encuentra un puntero libre lo devuelve inmediatamente . Cuando la cache esta llena, la función reserva memoria entonces gracias a kmalloc().*
*La liberación de los objetos se realiza mediante la función **kmem_cache_free().***

3.1.7 Los módulos :

Linux es un sistema operativo a capas que no emplea un micro-núcleo. Dado que las comunicaciones entre las extensiones de un sistema de este tipo son muy lentas en el tiempo, Linux utiliza la noción de los módulos,es decir, porciones de código de sistema que se pueden insertar o eliminar cuando el usuario o algún dispositivo lo solicita.

Hay diferentes modos de funcionamiento o tipos de módulos según el dispositivo a controlar, módulos en modo carácter, bloque o red. Los 2 primeros trabajan sobre un tipo de ficheros accesibles o bien en modo bloque como los dispositivos de almacenamiento de datos,o bien en modo carácter como por ejemplo los puertos series.

los módulos poseen dos funciones imprescindibles para la carga y descarga en memoria, ***module_init()*** *y* ***module_exit()****. Estas funciones se encuentran definidas en* ***linux/module.h*** *junto a otras que sirven para manejar información adicional de los módulos y incluso pasar argumentos a la hora de cargarlos en memoria como veremos en el siguiente ejemplo.*

```
MODULE_DESCRIPTION("Descripción del modulo"); /*opcional*/
MODULE_AUTHOR("el autor");
MODULE_VERSION("versión del modulo");
MODULE_LICENSE("tipo de licencia");

static int variable;
 module_param_int(variable,variable,sizeof(variable),0644);

static int __init  func_inicio(void)
 {
  if(!variable) {….}
  ….
 }

static void __exit  func_salida(void)
 {
   /*liberación de recursos*/
   ….
 }
module_init(func_inicio);
module_exit(func_salida);
```

los módulos se cargan mediante el comando ***insmod*** *o* ***modprobe****, en nuestro ejemplo, hemos definido una variable de tipo entero que pasamos como argumento a la hora de insertar el modulo:*

insmod ejemplo.ko variable = 2;

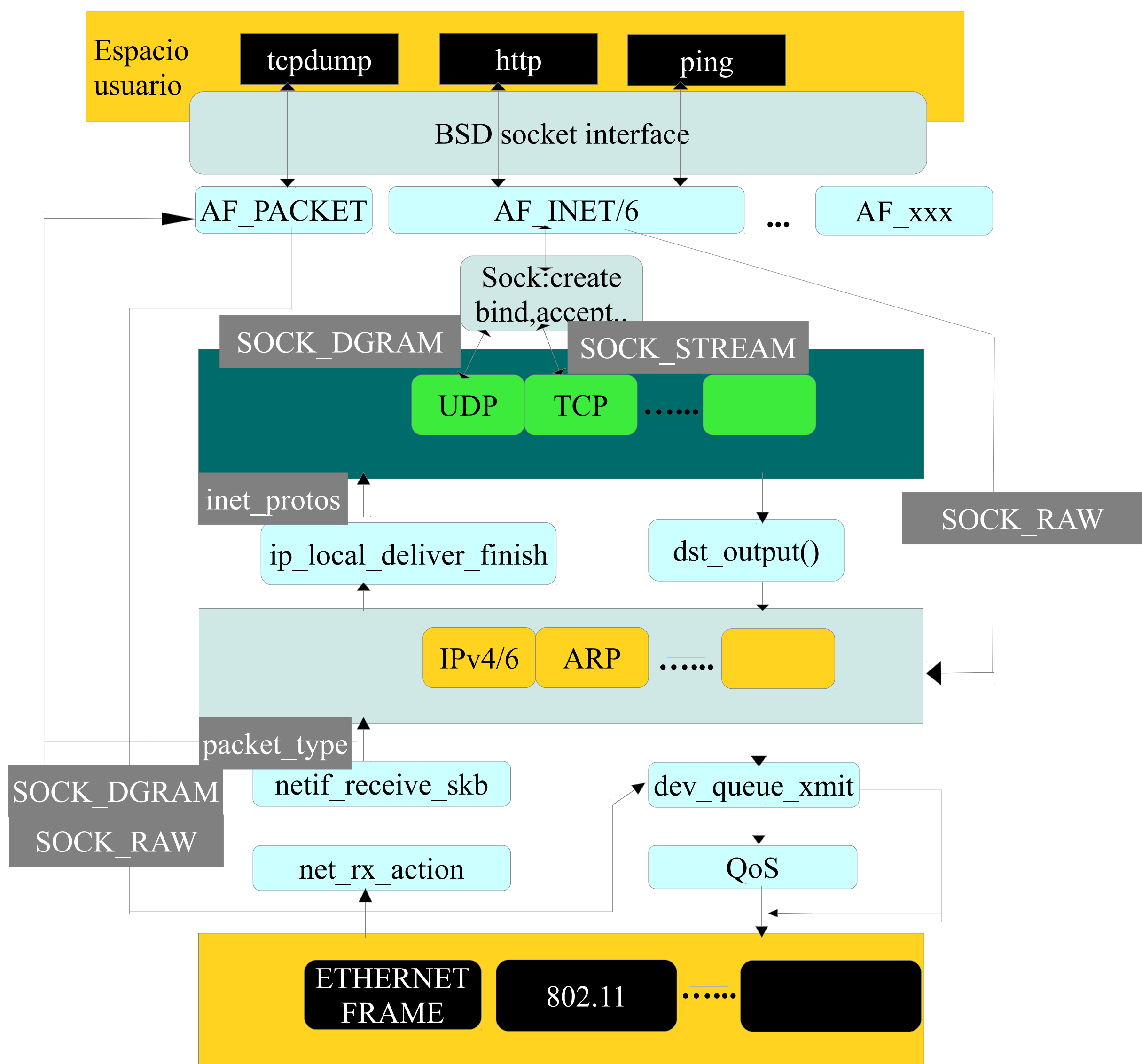
mediante ***rmmod*** *se descarga un modulo de la memoria.*

3.2 *El subsistema* de red en Linux.

La organización de la pila de red en Linux gira alrededor de 3 elementos principales :

- *Las interfaces Sockets.*
- *Los protocolos de red.*
- *Los periféricos de comunicación.*

la siguiente figura muestra un resumen del subsistema de red :



Como puede verse en la figura,en la capa mas alta, los Socket BSD constituyen la primera interfaz entre los procesos usuario y los servicios de red. Estos sockets suportan diferentes tipos de comunicación como secuencia, datagrama o paquetes.

Por otro lado,los sockets INET proporcionan una capa intermediaria entre los sockets BSD y los protocolos de transporte. En esta capa, se definen las principales primitivas de creación, conexión o asignación de la familia de direcciones que se podrían dar a un socket entre otras y los diferentes protocolos suportados para este dominio.

Por debajo de los sockets, se encuentra la capa de transporte,ademas de la capa de red y la de enlaces de datos donde opera el protocolo W2LAN. Grand parte del mecanismo de red en el kernel se hace en el contexto de dos interrupciones softirq NET_RX_SOFTIRQ y NET_TX_SOFTIRQ que se registran en la inicialización y serán planificadas cada vez que se reciba o se transmita una trama. En esta sección, vamos a presentar algunas interfaces y estructuras que intervienen en la capas del modelo de red donde opera el protocolo W2LAN.

El código kernel de la pila de red se ubica en un gran numero de subdirectorios del directorio ***linux/net.*** *Vemos a continuación algunos de sus repertorios :*

-- ./core : contiene el código común para el acceso a los periféricos de red,socket,etc

– ./ipv4 : contiene la implementacion del protocolo IP versión 4.

– ./packet : la implementacion de los sockets raw o brutos.

–./mac80211 : implementacion del estándar IEEE 802.11.

–./netfilter : implementacion de netfilter para interceptar los paquetes y su manipulación.

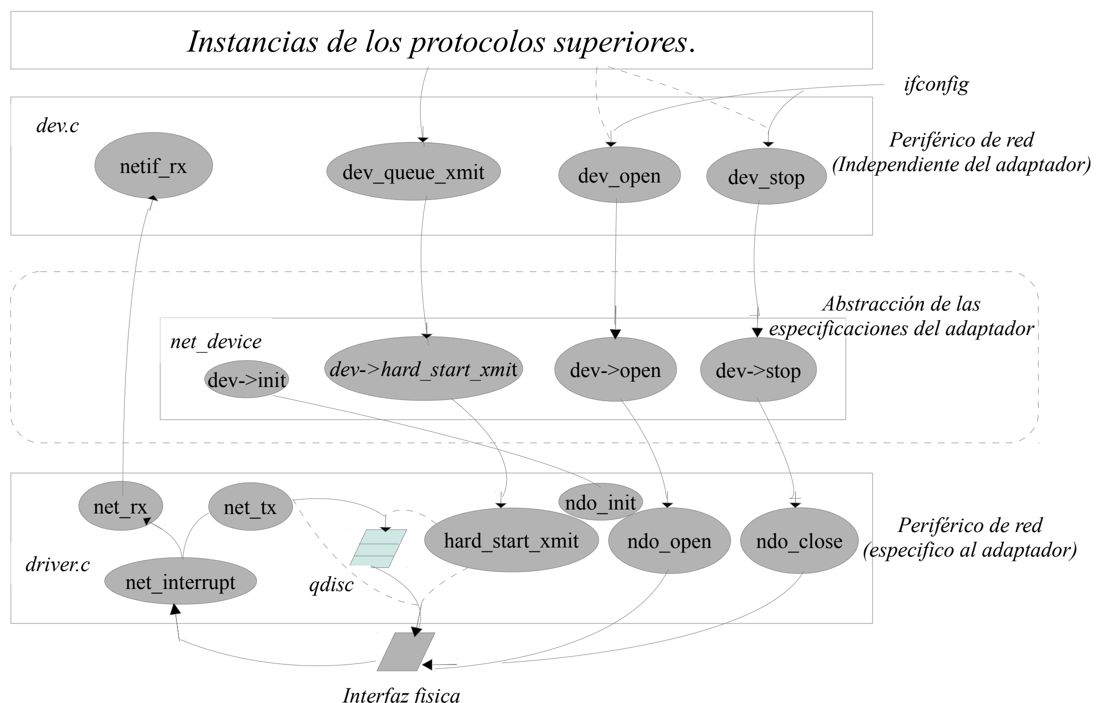
–./sched : contiene la implementacion de las diferentes políticas de gestión de las colas de espera.

3.2.1 La interfaz net_device :

*Los periféricos de red constituyen la tercera categoría de los adaptadores de linux. A diferencia de los periféricos de bloque o carácter, los adaptadores de red no poseen una entrada en el sistema de ficheros "el repertorio /dev" y tampoco se pueden realizar las operaciones de lectura y escritura simples,pero en cambio se pueden configurar desde el espacio de usuario mediante las herramientas **ifconfig** o similares. La interfaz de red puede ser activada en el arranque del sistema o cargarse en ejecución como modulo de kernel.*

*La estructura net_device es la base de todos los periféricos de red en el núcleo de Linux. Contiene información no solo del material de la interfaz (interrupción,puertos..etc), pero igualmente mantiene algunos parámetros de configuración relativos a los protocolos superiores. Esta definida en el fichero **include/linux/netdevice.h** y se puede considerar como una clase con atributos y métodos en el sentido de la programación orientada a objetos pero implementada en C, de hecho grand parte del subsistema de red emplea este mecanismo. los campos de esta estructura pueden clasificarse en tres categorías: campos generales del adaptador de red, campos específicos del hardware y otros relacionados con los protocolos de red. Ademas ofrece métodos para la configuración y manipulación de la interfaz.*

La siguiente figura muestra una visión general de algunos métodos ofrecidos para manipular una interfaz de red.



*la estructura net_device forma una interfaz general entre las instancias de los protocolos superiores y el adaptador físico. El concepto del puntero de una función es empleado de nuevo,donde la interfaz de las funciones de configuración y manipulación especifica de cada tarjeta de red están agrupadas en las estructuras **net_device_ops** e **ethtool_ops**, esta ultima opera solo si la interfaz es de tipo Ethernet. Como puede verse en la figura, la interfaz net_device_ops ofrece algunas funciones básicas como init, open o close que se pueden llamar desde una llamada al sistema o por el propio kernel . Existen también funciones para configurar diversos parámetros de la tarjeta como (Direccion MAC, MTU, modo de funcionamient etc..) y otras llamadas por los protocolos superiores como el método hard_start_xmit para enviar paquetes vía el suporte físico.*

3.2.2 La estructura qdisc o disciplinas de cola:

Las disciplinas de cola son algoritmos que se encargan de gestionar las colas en todo el proceso de ingreso y salida de los paquetes sobre un dispositivo de red. Estas son las base sobre la que se fundamenta la gestión de trafico en linux. Cada dispositivo tiene asociado un proceso de ingreso y de egreso y funcionan generalmente según la política FIFO. Pero es posible difinir varias colas en la misma interfaz y utilizarlas con diferentes disciplinas.

*Existen dos grupos de disciplinas de colas, unas con clase y las otras sin clase. Aquellas con clases permiten al usuario crear subdivisiones para realizar diferenciación en el tratamiento del trafico. La estructura de colas se llama **qdisc** y esta definida en **include /net/ sch_generic .h.***

*Linux suporta varias disciplinas y sus implementaciones se encuentran definidas **net/sched.** algunas son las siguientes:*

- *CBQ: Class based queue.*
- *TBF: Token Bucket Filter*
- *TEQL : Priority Traffic Equalizer*
- *GRED : Generalized Random Early Detection.*
- *FIFO : First In First Out.*
- *ATM : Asinchronous Transfer Mode.*
- *RED : Random Early Detection.*
- *DSMARK : Diff-serv Mark.*
- *HTB : Hierarchical Token Bucket.*

también se ofrecen las funciones que soportan las diferentes disciplinas de colas y están agrupadas en la estructura Qdisc_ops y son:

```
108struct Qdisc_ops {
109        struct Qdisc_ops        *next;
110        const struct Qdisc_class_ops    *cl_ops;
111        char                    id[IFNAMSIZ];
112        int                     priv_size;
113
114        int                     (*enqueue)(struct sk_buff *, struct Qdisc *);
115        struct sk_buff *        (*dequeue)(struct Qdisc *);
116        struct sk_buff *        (*peek)(struct Qdisc *);
117        unsigned int            (*drop)(struct Qdisc *);
118
119        int                     (*init)(struct Qdisc *, struct nlattr *arg);
120        void                    (*reset)(struct Qdisc *);
121        void                    (*destroy)(struct Qdisc *);
122        int                     (*change)(struct Qdisc *, struct nlattr *arg);
123        void                    (*attach)(struct Qdisc *);
124
125        int             (*dump)(struct Qdisc *, struct sk_buff *);
126        int             (*dump_stats)(struct Qdisc *, struct gnet_dump *);
127
128        struct module           *owner;
129};
```

- *Enqueue : Permite encolar los paquetes a la disciplina de cola.*
- *Dequeue : Desencola los paquetes para que sean enviados.*
- *Peek : Reencola un paquete para que sea transmitido,esta función se utiliza en el caso de que al ser desencolado el paquete este no es transmitido por alguna razón, se vuelve a encolar a la misma posición en la cola.*
- *Drop : descarta los paquetes de la cola.*
- *Init : inicializa y configura los parametros de una disciplina de cola cuando es creada.*
- *Reset: Reinicializa la disciplina de cola a su estado inicial.*
- *Destroy : Usada para remover una disciplina de cola.*
- *Dump : Empleada para realizar diagnostico de los datos de una disciplina de cola.*

*las colas qdisc se registran en el espacio kernel mediante la función **register_qdisc()** y toma como argumento la estructura qdisc_ops haciendo coincidir los diversos eventos que pueden ocurrir en la qdisc con las funciones definidas.*

3.2.3 La estructura socket buffer (sk_buff) :

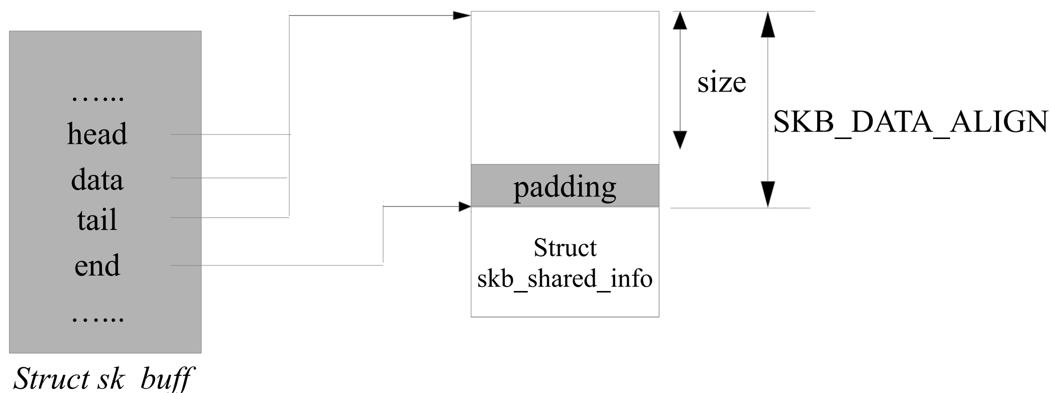
Es la estructura central de los datos utilizada por todas las capas del subsistema de red para gestionar los paquetes. Contiene información tanto de gestión de control como varios punteros al contenido de la trama que cambian conforme pasamos de una capa a otra. Si son paquetes salientes se alocan en el dispositivo de red y si en cambio son entrantes se alocan en la capa socket.

La definición de la estructura sk_buff se encuentra en ***linux/skbuff.h.*** *La siguiente figura muestra algunos campos de la estructura sk_buff y su relación con una trama:*

Estos punteros representan los limites de una trama en la memoria, el kernel ofrece muchas funciones para la manipulación de estos punteros pero en primer lugar vamos a ver la función de asignación de memoria para una estructura del tipo sk_buff .

- La función alloc_skb :

Es la función principal de asignación de la memoria y esta definida en ***net/core/skbuff.c*** *Esta función reserva dos zonas de memoria, una para la estructura sk_buff y la otra para el buffer de datos utilizando el mismo mecanismo de la memoria cache descrito anteriormente. Idealmente en esta fase, es deseable predecir la cantidad de memoria que sera utilizada por el paquete, se utiliza la macro SKB_DATA_ALIGN(size) para forzar el alineamiento. También se inicializan los punteros head, data y tail con la primera dirección de la zona de datos y el puntero end con la ultima dirección como se muestra en la siguiente figura.*



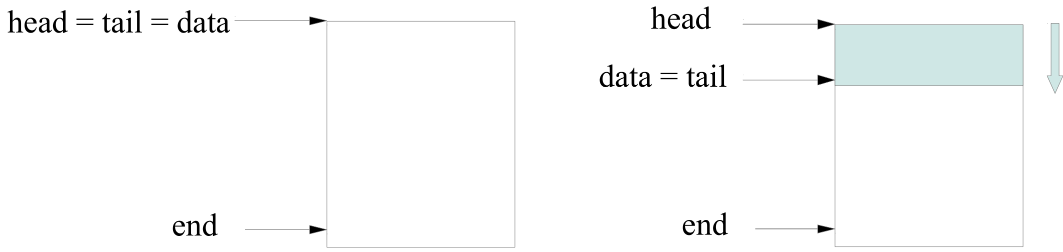
la estructura skb_shared_info colocada justo después del puntero end se utiliza principalmente para manejar fragmentos IP.

*Existe también la función **dev_alloc_skb** utilizada por los controladores de dispositivos y que se puede llamar desde una rutina que se ejecuta en el contexto de una interrupción.*

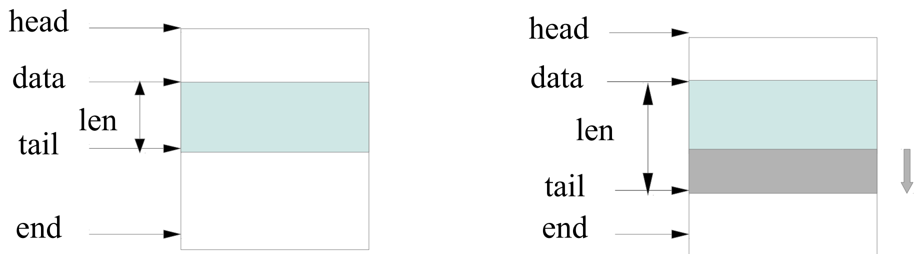
- Manipulación del buffer de datos:

Un vez la memoria es asignada, se tiene que manipular la zona de datos para construir un paquete. Puesto que en cada capa es susceptible añadir datos tanto al principio como al final de un paquete,la zona de datos se divide en tres zonas headroom,data y finalmente tailroom como hemos vemos en la figura . Por la misma razón, la estructura sk_buff contiene lo punteros transport_header, network_header y mac_header apuntando a las diferentes cabeceras del paquete.

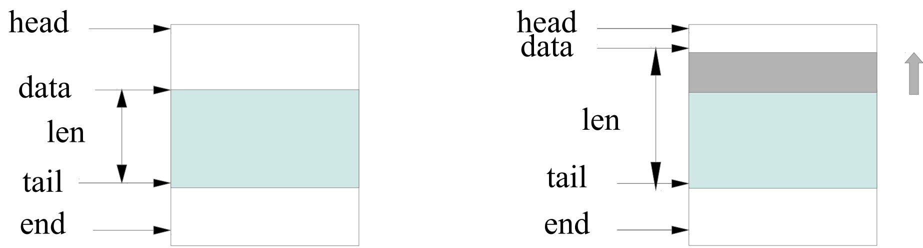
*La interfaz **include/linux/skbuff.h** nos ofrece varias funciones para manipular estos punteros y poder ajustar los campos de la estructura de manera adecuada. la siguiente figura muestra las principales funciones para mover los punteros head,data,tail y end.*



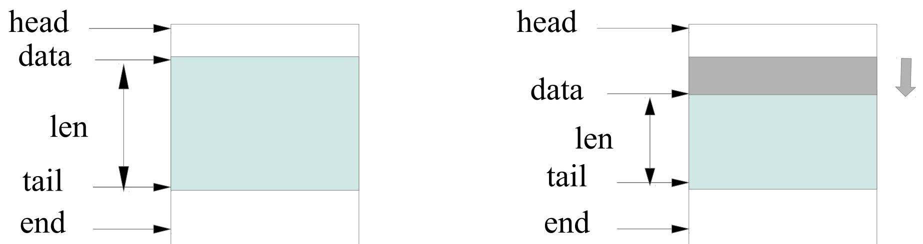
(a) skb_reserve().



(b) skb_put().



(c) skb_push().



(d) skb_pull().

*La función **skb_reserve()** asigna memoria a la zona headroom desplazando los punteros data y tail hacia abajo. Esta operación se realiza sobre un paquete vació y la asignación se hace de forma definitiva sobre todo, si escribimos datos en esta zona de memoria. Si en algún momento se requiere mas espacio en la zona headroom después de llamar a la función skb_reserve(), tenemos que volver a crear un buffer mas grande por medio de la función **skb_realloc_headroom()**.*

Para escribir en la zona de datos, debemos asignar espacio moviendo los punteros data o tail hacia arriba o abajo mediante las funciones skb_push() y skb_put() respectivamente. Es aceptable asignar mas memoria de lo necesario para la zona de datos de un paquete o para la cabecera head, pero en cambio cuando se usan las funciones push y put debemos proporcionar la longitud exacta de la zona necesaria.

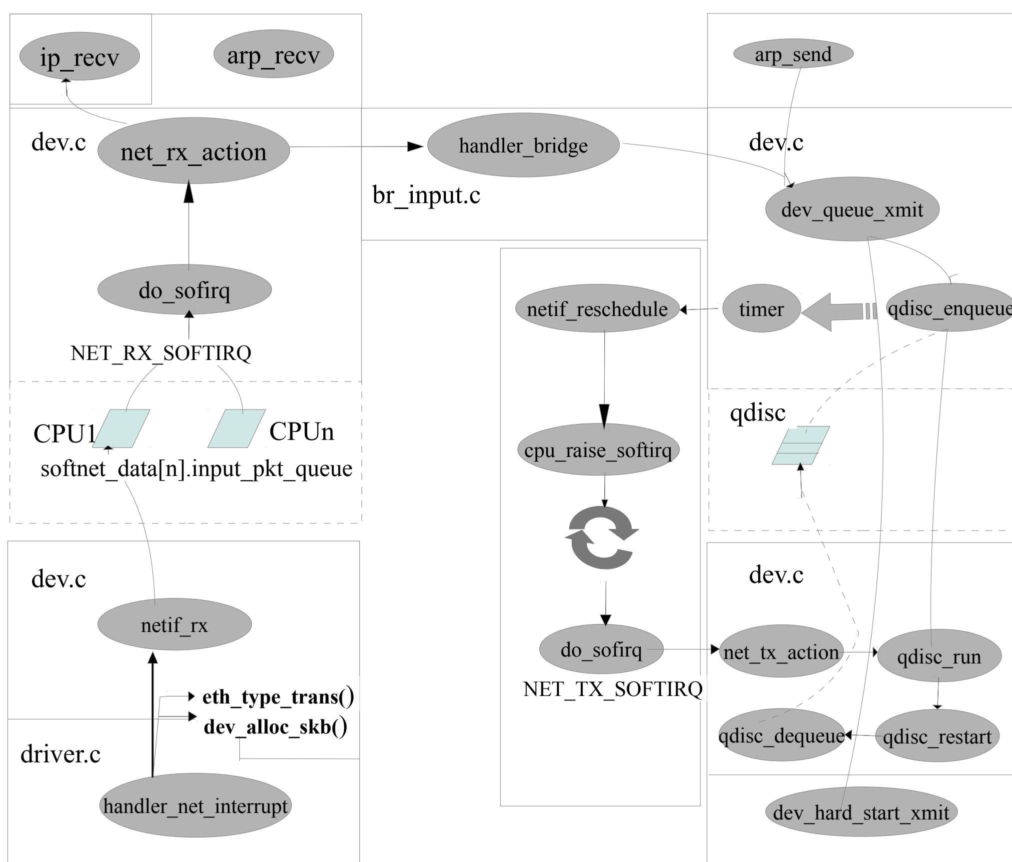
La variable len, es un campo de la estructura sk_buff que contiene la longitud actual de la zona de datos. Esta variable cambia conforme en que nivel se encuentra la trama en la pila de los protocolos y sirve para determinar cuantos bytes hay que enviar a la interfaz de red.

*Finalmente, las funciones **skb_reset_network_header(), skb_reset_mac_header(skb) y skb_reset_transport_header()** sirven para inicializar los punteros network_header transport_header y mac_header respectivamente.*

3.2.4 El itinerario de un paquete :

El itinerario de todo paquete enviado o recibido pasa por el intermedio de un suporte físico. Un adaptador de red es la interfaz final entre las instancias de los protocolos y el suporte de red. En esta sección, vamos a describir como se realiza la transición y las diferentes formas de actividades entre los dos niveles mas bajos de la pila de protocolos, las capas de red y enlace.

La siguiente figura muestra los elementos que intervienen en el proceso:



Recepción de una trama :

Cuando la tarjeta de red recibe una trama correctamente,lanza una interrupción para informar al núcleo de su llegada. La rutina de tratamiento de interrupción representa por lo tanto la primera actividad del núcleo que se encarga del tratamiento del paquete entrante.

*Una vez verificada la trama y identificada la causa de la interrupción como recepción ya que se utiliza el mismo numero de interrupción tanto para la recepción como para el envió, el procesamiento del paquete sigue en la función **net_rx()**. Se asigna un descriptor de socket gracias a la función **dev_alloc_skb()** y se copia el paquete desde el adaptador de red a la zona de datos. El puntero skb->dev apunta entonces a la interfaz de red y el tipo de datos de la capa 2 es establecido gracias a la función **eth_type_trans().** Por lo tanto, el proceso de demultiplexaje de la capa LLC tiene lugar aquí y sera presentado en la sección 3.2.3. Para otras tecnologías MAC como (Token Ring,FDDI) existen otras funciones similares.*

*La función **netif_rx()** termina el procesamiento de la interrupción hardware marcando en primer lugar la hora actual de la llegada y colocando el descriptor de socket en la cola de recepción de la CPU que ha tratado la interrupción en la estructura **softnet_data**.*
*El itinerario de un paquete termina por el momento en una cola de espera reservada a los paquetes entrantes **softnet_data [n].input_pkt_queue**. La rutina de tratamiento de una interrupción hardware como hemos comentado antes permite solo ejecutar las operaciones obligatorias para no mantener el núcleo bloqueado durante mucho tiempo. El proceso sigue ahora gracias a la interrupción software y precisamente en el caso de la recepción con la interrupción NET_RX_SOFTIRQ.*

*El tratamiento de la interrupción HARD_IRQ se termina y el núcleo continua el procesamiento de la tareas ejecutadas anteriormente. Si después de un tiempo, el administrador de tareas es llamado de nuevo,ejecuta la rutina de tratamiento de la interrupción llamando la función **net_rx_action().** Esta función retira los paquetes uno a uno de la cola de recepción de la CPU en curso para su procesamiento por las instancias de los protocolos superiores.*

Transmisión de una trama :

El proceso de transmisión de un paquete puede, como muestra la figura,tener lugar vía diferentes actividades del núcleo. Podemos distinguir dos procesos de emisión:

- *El proceso normal de emisión,durante el cual una actividad intenta reorganizar un paquete listo para su expedición en una cola de espera de salida (qdisc). Eso significa que el proceso de emisión se realiza o bien en el contexto de una interrupción NET_TX_SOFTIRQ, o bien después de una llamada al sistema.*

- *El segundo tipo de emisión es puesto en marcha solo por medio de la interrupción NET_TX_SOFTIRQ principalmente cuando los paquetes deben,por alguna razón, ser enviados fuera del proceso normal. Eso ocurre cuando se requiere por ejemplo, diferir el envió para controlar el flujo de transmisión o cuando no hay suficiente memoria en la interfaz de red.*

la función principal utilizada por las instancias de los protocolos superiores para emitir un paquete en la forma de un descriptor de socket es ***dev_queue_xmit(skb).*** *La interfaz de red es indicada gracias al puntero skb->dev de la estructura sk_buff.*
El descriptor de socket es reorganizado en la cola de salida del periférico de red. Esta operación se realiza mediante el método ***dev->qdisc->enqueue()*** *.*

una vez reorganizado el paquete en la cola según el método qdisc elegido, se invoca el método ***qdisc_run()*** *que a su vez llama a la función* ***qdisc_restart()*** *que se encarga de recuperar y emitir el próximo paquete como se muestra en la figura. Los paquetes se recuperan de la cola mediante el método* ***dev->qdisc->dequeue(),*** *en caso de éxito son enviados a la interfaz de red por medio del método* ***dev->hard_start_xmit()****.*

En el caso particular de un periférico de red sin ningún método de gestión de colas,los paquetes son directamente enviados mediante ***dev->hard_start_xmit()****. Generalmente se trata de los periféricos de red lógicos "loopback" o de puente "bridge".*

La interrupción NET_TX_SOFTIRQ se considera como un segundo itinerario de un paquete saliente. La función ***netif_reschedule()*** *es llamada cada vez que un descriptor de socket skb no ha sido enviado por el proceso normal. La rutina* ***net_tx_action()*** *es la responsable de tratar la interrupción y su función consiste en llamar al método qdisc_restart() para volver a iniciar la transmisión al periférico de red.*

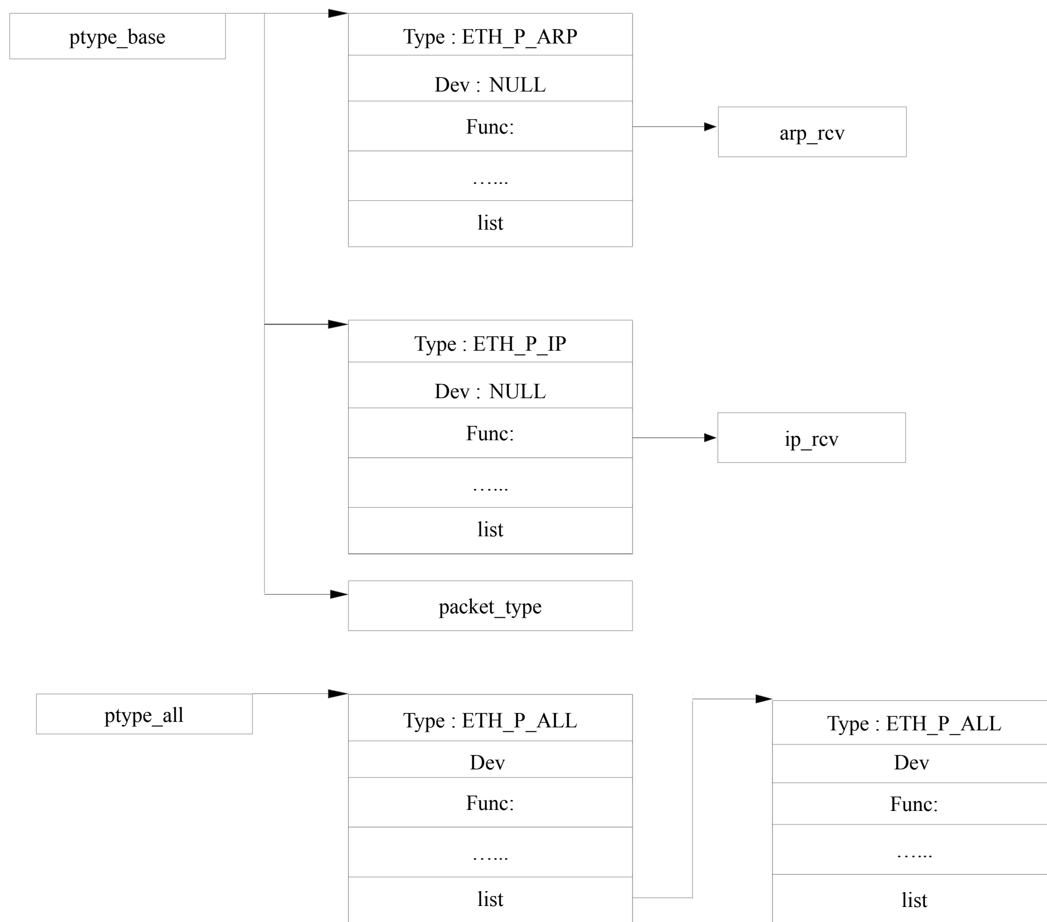
3.2.5 Gestión de los protocolos de red :

En la sección anterior vimos el itinerario de un paquete entre el adaptador de red y las interfaces de las instancias de los protocolos superiores. Lo siguiente sera precisamente estudiar esta interfaz y ver como se añaden nuevos protocolos de la capa de red.
En regla general, solo los protocolos de esta capa pueden operar en este nivel, pero en el núcleo de Linux distinguimos dos tipos de protocolos de la capa 3 :

-- protocolos que reciben todos los paquetes que llegan por una interfaz.

– protocolos que reciben solo los paquetes con el tipo del protocolo correspondiente.

Los dos tipos de protocolos se manipulan en dos listas de datos diferentes ***ptype_base*** *y* ***ptype_all*** *como muestra claramente la siguiente figura .*



*El tipo del paquete es una instancia de la estructura **packet_type** definida en **include/linux/netdevice.h.***

```
1203struct packet_type {
1204        __be16                  type;/* This is really htons(ether_type). */
1205        struct net_device       *dev;/* NULL is wildcarded here          */
1206        int                     (*func) (struct sk_buff *,
1207                                         struct net_device *,
1208                                         struct packet_type *,
1209                                         struct net_device *);
1210        struct sk_buff          *(*gso_segment)(struct sk_buff *skb,
1211                                                int features);
1212        int                     (*gso_send_check)(struct sk_buff *skb);
1213        struct sk_buff          **(*gro_receive)(struct sk_buff **head,
1214                                                 struct sk_buff *skb);
1215        int                     (*gro_complete)(struct sk_buff *skb);
1216        void                    *af_packet_priv;
1217        struct list_head        list;
1218};
```

Los parámetros siguientes son necesarios para la definición de un protocolo en la estructura packet_type:

– *type : Este campo indica la identificación del protocolo. Estas constantes están definidas en **include/linux/if_ether.h**. Si indicamos el tipo **ETH_P_ALL** el protocolo se registra en la lista ptype_all y recibe todos los paquetes entrantes. En cambio los protocolos de la pila de red se registran en la lista ptype_base. Por ejemplo, el tipo ETH_P_ARP sirve para identificar el protocolo ARP.*

– *dev : puede apuntar a un adaptador de red, en esta caso todos los paquetes que lleguen por esta interfaz serán directamente transmitidos al protocolo. Cuando dev vale null, que corresponde al caso normal, el periférico de red no interviene en la elección del protocolo.*

– *func: Es la rutina de tratamiento del protocolo encargada de recibir un paquete de la interfaz de enlace. Realmente el procesamiento del protocolo empieza en esta función.*

– *af_packet_priv : este puntero puedes servir para guardar datos privados del protocolo.*

– *gso (Generic segmentation offload)gro(Generic Receive offload) _function : si la la tarjeta de red lo permite, las operaciones de la segmentacion y el calculo del checksum se efectúan en el propio procesador de la tarjeta. Esto permite ahorra en la carga de la CPU del sistema y disminuir las comunicaciones en el bus PCI .*

*Los dos métodos siguientes están disponible para enlazar o desenlazar los protocolos e/o la estructura packet_type a la pila de red , **dev_add_pack()** y **dev_remove_pack()**.*

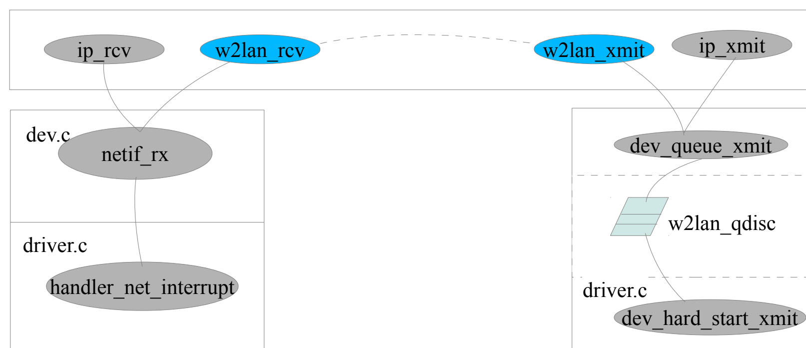
La identificación de un protocolo se realiza en la capa de encale y precisamente en la función eth_type_trans(), como hemos visto en la sección anterior. Ademas,esta función se considera como el segundo elemento importante de la implementacion de la capa LLC, ya que, en esta fase se determina el tipo del paquete (unicast,multicast,broadcast) y se verifica el destino del paquete.

Capitulo 4.

Implementacion del protocolo W2LAN en Linux.

El protocolo w2lan esta integrado al susbsistema de red de linux como modulo kernel. La elección de programar un modulo kernel viene por la facilidad que ofrece el sistema linux para añadir nuevas funcionalidades en tiempo de ejecución sin la necesidad de recompilar todo el núcleo.

En el siguiente capitulo,vamos a describir como se ha llevado a cabo la implementacion asociándola con lo visto anteriormente. Se presenta la siguiente figura, que muestra de manera esquematica el modo de funcionamiento del protocolo w2lan para poder comprender mejor el código que se expondrá a continuación.



Básicamente, el terminal que ejecuta el protocolo w2lan debe capturar todos los paquetes entrantes y salientes, por lo cual el protocolo w2lan se registra en la capa de red y mas concretamente en la lista ptype_all para la recepción. En el caso de los mensajes salientes, se ha definido una disciplina de cola w2lan_qdisc que se enlaza con la interfaz de red. Otra solución que ofrece Linux para controlar o manipular paquetes tanto salientes como entrantes en cualquier nivel de red es mediante la interfaz NETFILTER, pero la condición de capturar todos los tipos de paquetes salientes nos llevo a utilizar la interfaz qdisc.

3.1 Organización del modulo W2LAN.

El modulo w2lan esta organizado en cuatro archivos .c con sus respectivas cabeceras .h como sigue :

- *w2lan.c*
- *w2lan_input.c*
- *w2lan_output.c*
- *w2lan_list.c*

El archivo w2lan.c es el punto de entrada del modulo,implementa las funciones init y exit cada vez que se carga y descarga en memoria,pero antes vamos a ver la estructura de datos y las diferentes variables utilizada por el protocolo definidas en el fichero w2lan.h

```
#ifndef W2LAN_H
#define W2LAN_H

#define DRIVER_AUTHOR             "Katfi Youssef <Katfiyoussef@hotmail.com>, " \
                  "francesc burrull <francesc.burrull@upct.es>" \
#define DRIVER_DESC               "w2lan:Protocol that transforms a 802.11 mobile \
 ad-hoc network (MANET) into an ethernet LAN."

#ifndef SOURCE_VERSION
#define SOURCE_VERSION "2012.0.1"
#endif

/*Id protocol Announce */
#define W2LAN_A  0X6901
/*Id protocol Request  */
#define W2LAN_R  0X6902
/*Id protocol Data     */
#define W2LAN_D  0X6903

/********************/
#define MaxConvID        10

#define W2LAN_ETH_ALEN            6
#define W2LAN_ETH_HLEN            14

#define W2LAN_CONVID_LEN      6
#define W2LAN_ANNOUNCE_LEN    20
#define W2LAN_REQUEST_LEN     6

/*          */
#define w2_flag               0x69
#define proto_flag            0x06

#define SEND_ANNOUNCE             0x01
#define SEND_REQUEST              0x02
#define SEND_DATA             0x03
```

```
#define W2_LIST_RX                    0
#define W2_LIST_TX                    1

#define W2_EXPIRY_TIME      (10 * HZ)

#define W2LAN_MODE_ALWAYS      0
#define W2LAN_MODE_PASSIVE     1
#define W2LAN_MODE_REACTIVE    2

struct w2lan_convID
{
        unsigned char convID[W2LAN_CONVID_LEN];
        unsigned char orig_eth[W2LAN_ETH_HLEN];
        struct list_head list;
        atomic_t refcnt;
};

struct w2lan_data
{
        unsigned char convID[W2LAN_CONVID_LEN];
        unsigned char orig_eth[W2LAN_ETH_HLEN];
        unsigned char *data;
        int len;
        int TX;
        int forward;
        int pending;
        int request;
        atomic_t refcnt;
        struct list_head list;
        struct timer_list timer;
};
#endif
```

w2lan_data *es la estructura principal de datos que manipula el modulo w2lan. Es utilizada para guardar tanto las conversaciones recibidas como enviadas. Los campos de esta estructura son lo siguientes :*

- *convID : identificador de la conversación de 6 bytes.*
- *orig_eth: cabecera original ethernet del paquete que sirve para reconstruir la trama cuando llega a su destino.*
- *Data: zona de memoria donde se guardan los campos de datos de la trama.*
- *Len : longitud del campo data de una trama en bytes.*
- *Tx : esta variable toma el valor 1 en caso de transmisión y el valor 0 en la recepción.*
- *Forward: indica si la conversación tiene ser encaminada o ha llegado a su destino.*
- *Pending : se usa en el caso de la recepción de una trama announce ,toma el valor 1 si recibe los datos de una conversación iniciada. En caso contrario vale 0.*
- *request : numero de tramas request recibidas después de enviar una trama announce.*
- *refnct : contador de referencia.*
- *List : la estructura w2lan_data puede ser enlazada a una lista enlazada de recepción o transmisión.*
- *Timer : es el tiempo que se mantiene una conversación en memoria.*

El modulo funciona según 3 tipos de modos passive,active o always, que se le pasa a la variable mode a la hora de cargar el modulo en memoria, ademas se crea una entrada w2lan_time en el repertorio net/w2lan para modificar el tiempo del temporizador en tiempo de ejecución.

```
#define MAX_STRING_LEN  10

static char mode[MAX_STRING_LEN];

module_param_string(mode,mode,sizeof(mode),0644);

static struct ctl_table w2lan_table[] = {
    {
            .procname       = "w2lan_time_set",
            .data           = &w2lan_time,
            .maxlen         = sizeof(int),
            .mode           = 0644,
            .proc_handler   = proc_dointvec
    },
    {},
};

static struct ctl_path w2lan_path[] = {
              { .procname = "net", },
              { .procname = "w2lan", },
      { }
};

static struct ctl_table_header *w2lan_header;

static int __init w2lan_init(void)
{
    int err = -ENODEV;
    struct ctl_table *table;


    err = init_w2lan_output();
    if(err<0)    return err;

    init_w2lan_input();

    err =  list_init();

    if(err<0)
    {
            exit_w2lan_input();
            exit_w2lan_output();
            return  err;
    }

    w2lan_header = register_sysctl_paths(w2lan_path, w2lan_table);

    if (w2lan_header == NULL) return -1;

    if(strncmp(mode, "always"  ,6)==0)
                        w2lan_mode = W2LAN_MODE_ALWAYS;

    if(strncmp(mode, "passive" ,7)==0)
                        w2lan_mode = W2LAN_MODE_PASSIVE;

    if(strncmp(mode, "reactive", 8)==0)
                        w2lan_mode = W2LAN_MODE_REACTIVE;

    return  0;
}
```

```
static void __exit w2lan_exit(void)
{
   unregister_net_sysctl_table(w2lan_header);
   exit_w2lan_output();
   exit_w2lan_input();
   purge_w2lan_list();
}

module_init(w2lan_init);
module_exit(w2lan_exit);
```

la función init_w2lan_input registra el protocolo w2lan en la lista de los protocolos de red, mientras que la función init_w2lan_output crea la disciplina de colas w2lan_queue. Estas funciones junto a las funciones exit_w2lan_output y exit_w2lan_input están definidas en los archivos w2lan_input.c y w2lan_output.c que veremos en la próxima sección.

el protocolo w2lan maneja 3 listas diferentes, listas de las conversaciones recibidas RX y transmitidas TX, ademas de una lista que registra todas las conversaciones que hay en la red almacenando el convID y la cabecera ethernet para evitar la duplicación de una trama. En el fichero w2lan_list.c tenemos definidas las diferentes cabeceras de las listas como sigue:

```
static struct list_head w2_head_list_TX;
static struct list_head w2_head_list_RX;
static struct list_head w2_head_list_convID;
```

La estructura w2lan_data se enlaza con las cabeceras w2_head_list_TX o w2_head_list_RX según si estamos enviando o recibiendo y se utiliza el concepto de memoria cache ya que la estructura w2lan_data se usa frecuentemente ,mientras que la estructura w2lan_convID se enlaza con w2_head_list_convID y tiene un tamaño máximo de MaxConvID. Cuando se supera este valor se sobrescribe en la posición de la conversación mas antigua.

```
static struct kmem_cache *w2_slab;

static int  w2_slab_init(void)
{
       w2_slab = kmem_cache_create("w2lan_slab",
                               sizeof(struct w2lan_data),
                                0, 0, NULL);
       if (w2_slab == NULL)
              return -ENOMEM;
       return 0;
}
int list_init()
{
      int ret;
      ret = w2_slab_init();
      if(ret!=0) return ret;
      INIT_LIST_HEAD(&w2_head_list_TX);
      INIT_LIST_HEAD(&w2_head_list_RX);
      INIT_LIST_HEAD(&w2_head_list_convID);
      return ret;
}
```

Las funciones para obtener memoria y inicializar algunos campos de la estructura w2lan_data serian :

```
struct w2lan_data *w2lan_alloc()
{
        struct w2lan_data *w2=NULL;

        w2 = kmem_cache_alloc(w2_slab,GFP_ATOMIC);
        return w2;
}

void set_w2lan_data(struct w2lan_data *w2,int type)
{
        struct list_head *l_head;
        spinlock_t *lock;

        atomic_set(&w2->refcnt,1);
        init_timer(&w2->timer);
        w2->timer.expires  = jiffies + w2lan_time;
        w2->timer.function  = &w2_timer_expiry;
        w2->timer.data  = (unsigned long)w2;
        add_timer(&w2->timer);

        if(type == W2_LIST_RX)
        {
                w2->TX = 0;
                lock = &w2_lock_RX;
                l_head = &w2_head_list_RX;
        }
        else // W2_LIST_TX
        {
                w2->TX = 1;
                lock = &w2_lock_TX;
                l_head = &w2_head_list_TX;
        }
        spin_lock(lock);
        list_add_tail(&w2->list,l_head);
        spin_unlock(lock);
}
```

Cuando se expira el temporizador se ejecuta la función w2_timer_expiry como se muestra a continuación:

```
static void w2_timer_expiry(unsigned long param)
{
        struct w2lan_data *w2 = (struct w2lan_data *)param;

        if(w2)
        {
                if(!w2->TX)
                {
                    del_foo(w2,W2_LIST_TX);
                    put_foo(w2);
                }
                else if(w2->TX == 0 && w2->forward == 0 )
                {
                    del_foo(w2,W2_LIST_RX);
                    put_foo(w2);
                }
                else
                {
                   w2->TX = 1;
                   w2->forward = 0;
                   w2->timer.expires  = jiffies + w2lan_time;
                   add_timer(&w2->timer);
                }
        }
}
```

Las funciones para iterar, añadir o eliminar un elemento de la lista de TX o RX serian :

```
struct w2lan_data *get_data(unsigned char *convID,int type)
{
        int count = 0;
        spinlock_t *lock;
        struct w2lan_data *w2=NULL;
        struct list_head *l_head;

    if(type == W2_LIST_RX)
        {
            lock = &w2_lock_RX;
            l_head = &w2_head_list_RX;
        }
    else
        {
            lock = &w2_lock_TX;
            l_head = &w2_head_list_TX;
    }


    spin_lock(lock);

        list_for_each_entry(w2,l_head,list){
            if (compare_ether_addr(w2->convID,convID) == 0 ){
            count++;
            atomic_inc(&w2->refcnt);
            break;
          }
        }
        spin_unlock(lock);

        if(count==0)
            w2=NULL;
        return w2;
}

void w2lan_list_add_TX(struct w2lan_data *w2)
{
        spin_lock(&w2_lock_TX);
        list_add_tail(&w2->list,&w2_head_list_TX);
        spin_unlock(&w2_lock_TX);
}


void del_foo(struct w2lan_data *w2,int type)
{
    spinlock_t *lock;

    if(type == W2_LIST_RX)     lock = &w2_lock_RX;
    else lock = &w2_lock_TX;

    spin_lock(lock);
    list_del(&w2->list);
    spin_unlock(lock);
}


void put_foo(struct w2lan_data *w2)
{
    if (atomic_dec_and_test(&w2->refcnt))
    {
        if(w2->data!=NULL) kfree(w2->data);
        kmem_cache_free(w2_slab, w2);
    }
}
```

finalmente la función w2lan_list_for_each es llamada cuando se descarga el modulo de la memoria y permite liberar los recursos de manera segura.Si un temporizador esta en ejecución a la hora de llamar a la función del_timer el flujo del programa es redirigido a la etiqueta retry para evitar una situación de bloqueo.

```
void w2lan_list_for_each_safe(struct list_head *l_head,spinlock_t *lock)
{

        struct list_head *p,*n;
        struct w2lan_data *w2=NULL;

retry:
        spin_lock(lock);
        list_for_each_safe(p,n,l_head) {
                w2 = list_entry(p, struct w2lan_data, list);

                if(!del_timer(&w2->timer)){
                        spin_unlock(lock);
                        goto retry;
                 }
                 list_del(&w2->list);
                 kmem_cache_free(w2_slab, w2);
        }
        spin_unlock(lock);
}
```

Por otro lado, la función is_duplicate permite manejar las listas de todas las conversaciones que pasan por la red.

```
int is_duplicate(struct w2lan_hdr_rcv *p)
{
        struct w2lan_convID *convID_list;
        spin_lock(&w2_lock_convID);

        list_for_each_entry(convID_list,&w2_head_list_convID,list){

                if (compare_ether_addr(convID_list->convID,p->convID) == 0 ){
                        printk("Duplicado \n ");
                        spin_unlock(&w2_lock_convID);
                        return 1;
                }
        }
        convID_list = NULL;

        if(list_empty(&w2_head_list_convID) || count_convID_list < MaxConvID  )
        {
                convID_list = kmalloc (sizeof(struct w2lan_convID),GFP_ATOMIC);
                if(convID_list != NULL){
                        memcpy(convID_list->convID,p->convID,W2LAN_CONVID_LEN);
                        memcpy(convID_list->orig_eth,p->orig_eth,W2LAN_ETH_HLEN);
                        list_add_tail(&convID_list->list,&w2_head_list_convID);
                        count_convID_list++;
                }
        }
        else
        {
     convID_list = list_first_entry(&w2_head_list_convID,struct   w2lan_convID,list);
                        memcpy(convID_list->convID,p->convID,W2LAN_CONVID_LEN);
                        memcpy(convID_list->orig_eth,p->orig_eth,W2LAN_ETH_HLEN);
                list_del(&convID_list->list);
                list_add_tail(&convID_list->list,&w2_head_list_convID);
        }
        spin_unlock(&w2_lock_convID);
       return 0;
}
```

3.2 Recepción de los Datos.

Como hemos comentado anteriormente, el protocolo w2lan se registra en la capa de red y precisamente en la lista ptype_all ya que se utilizan 3 tipos de identificadores de protocolo Announce, Request y Data. Otra alternativa para registrar el protocolo en la lista base de los protocolos de red ptype_base, seria unificando los identificadores y definiendo un nuevo campo en la zona de datos.

```
static struct packet_type w2lan_packet_type = {
                        .type =htons(ETH_P_ALL),// __constant_htons(W2LAN),
                        .dev = NULL,
                        .func = w2lan_recv,
 };

void init_w2lan_input(void)
{
    dev_add_pack(&w2lan_packet_type);
}

void exit_w2lan_input(void)
{
    dev_remove_pack(&w2lan_packet_type);
}
```

La función w2lan_recv() es el punto de comienzo del protocolo. El paquete es tratado según su tipo mediante las funciones process_Announe, process_Request o process_Data. En caso de recibir un trama con un identificador diferente y según el modo de funcionamiento del modulo se realizaran las siguientes operaciones:

- *Modo Reactivo : se genera una anuncio mediante la función generate_annouce solo en el caso de recibir antes alguna trama del tipo w2lan. Se debe cumplir que la trama recibida tenga como origen la interfaz donde esta enlazada la cola w2lan_queue y que la variable activewindow vale 1.*
- *Modo Always : en este modo se genera automáticamente un anuncio al llegar cualquier trama que no fuese w2lan.*
- *Modo Passive: en este caso, no se efectuaría ninguna acción*

```
static int w2lan_recv(struct sk_buff *skb, struct net_device *dv, struct packet_type *pt,struct net_device *org_dev)
{
            struct w2lan_hdr_rcv *w2_hdr=NULL;
            struct sk_buff *skb2 = NULL;

            skb2 = skb_copy(skb,GFP_ATOMIC);
            kfree_skb(skb);

            if(skb2==NULL) goto out;

            w2_hdr = (struct w2lan_hdr_rcv *)eth_hdr(skb2);
            if(compare_ether_addr(w2_hdr->eth.h_source,dv->dev_addr) == 0 ||
                                    compare_ether_addr(&w2_hdr->orig_eth[6],dv->dev_addr) == 0)           goto out_free;
```

```
		switch (ntohs(w2_hdr->eth.h_proto))
		{

			case W2LAN_A:
			if(is_duplicate(w2_hdr)== 0) process_Announce(skb2,w2_hdr); /**/
			break;

			case W2LAN_R:
			process_Request(skb2,w2_hdr);
			break;

		case W2LAN_D:
			process_Data(skb2,w2_hdr);
			break;

			default:
			if(w2lan_mode != W2LAN_MODE_PASSIVE && dv == w2lan_dev){

				if(w2lan_mode == W2LAN_MODE_REACTIVE && activewindow != 1)
					goto out;
				skb = w2lan_generate_announce(skb2);
				dev_queue_xmit(skb);
				goto out;

			}
			/*passive mode: nada*/;
		}
out_free:
		kfree_skb(skb2);
		return 0;
out:
		return 0;
}
```

Una vez que se identifica el tipo de la trama nos encontraremos frente a una de la siguientes situaciones:

- *process_Announce.*
- *process_Request.*
- *process_Data.*

Process_Announce:

Antes de llamar la función process_Announce, se verifica mediante la función is_duplicate si ya hemos tenido esta conversación, en caso contrario, comprobamos de nuevo en la lista de recepción. Si es una nueva conversación se procede a copiar el convID , la cabecera ethernet y se enviá un mensaje request al emisor.
Este mecanismo de control es debido a que la lista de las conversaciones tiene una memoria limita y puede ser que ya no tiene dicha convID guardada.

```
void process_Announce(struct sk_buff *skb,struct w2lan_hdr_rcv *head)
{
		struct w2lan_data *w2=NULL;
		activewindow = 1;

		w2 = get_data(head->convID,W2_LIST_RX);

		if(w2==NULL)		/*Nueva Conversacion,crear y enviar un mensaje Request*/
		{
		  w2 = w2lan_alloc();
		  if(w2 != NULL )
		  {
			memcpy(w2->orig_eth,head->orig_eth,W2LAN_ETH_HLEN);
			memcpy(w2->convID,head->convID,W2LAN_CONVID_LEN );
			w2->TX = 0;
			w2->pending=0;
			w2->data=NULL;
			w2lan_send_frame(w2,head->eth.h_source,SEND_REQUEST);
			set_w2lan_data(w2,W2_LIST_RX);
		  }
		}
		else put_foo(w2);

}
```

Process_Request:

Cuando llega una trama request, se verifica en la lista de transmisión si tenemos dicha conversación. En caso afirmativo, se enviá directamente la trama data y se incremente el contador de request. El uso de la variable request es debido a que se puede dar el caso de tener mas de dos nodos vecinos que envían un trama request de la misma conversación, para evitar congestionar la red con mensajes duplicados ya que la trama Data es de tipo broadcast.

Hay que señalar que en las especificaciones del protocolo los datos se envían cuando expira el temporizador, pero en esta versión se ha optado enviar directamente la trama de datos justo después de recibir un request, ya que mucho protocolos tienen un tiempo limitado de espera.

```
void process_Request(struct sk_buff *skb,struct w2lan_hdr_rcv *head)
{
		struct w2lan_data *w2=NULL;

		w2 = get_data(head->convID,W2_LIST_TX);
		if(w2!=NULL)
		{
				if(w2->data!=NULL && w2->request == 0)
				{
						w2lan_send_frame(w2,NULL,SEND_DATA);
						w2->request ++;
						put_foo(w2);
				}
		}
}
```

Process_Data:

Cuando se recibe una trama Data y después de verificar el convID en la lista de recepción se comprueba la cabecera original ethernet y se dan los siguientes casos :

- *Si la trama es destina a dicho nodo o es de tipo broadcast y esta pendiente de recibir los datos se procede a la reconstrucción del paquete que se pasa a los niveles superiores mediante la función w2_netif_rx.*
- *En el caso contrario, se elimina la conversación de la lista de recepción y se mueve a la lista de TX activando la opción forward. Luego se procede a enviar una trama announce y se vuelve a activar el temporizador en la función w2_timer_expiry.*

```
void process_Data(struct sk_buff *skb,struct w2lan_hdr_rcv *head)
{
            struct w2lan_data *w2=NULL;
            struct ethhdr *eth =NULL;

            skb_reset_network_header(skb);

            w2 = get_data(head->convID,W2_LIST_RX);

            if(w2!=NULL)
            {
              eth=(struct ethhdr *)w2->orig_eth;

              if (compare_ether_addr(eth->h_dest, skb->dev->dev_addr) != 0 ||
          is_broadcast_ether_addr(eth->h_dest))
                {
                  w2->forward = 1;
                        del_foo(w2,W2_LIST_RX);
                        w2lan_list_add_TX(w2);

                        w2->len = skb->len - W2LAN_REQUEST_LEN;

                        if(w2->len > 0)
                           w2->data = kmalloc(w2->len,GFP_ATOMIC);
                        if(w2->data==NULL)
                        {
                            w2->forward = 0;
                            goto end;
                        }
                        memcpy(w2->data,&skb->data[W2LAN_REQUEST_LEN],w2->len);
                        w2lan_send_frame(w2,NULL,SEND_ANNOUNCE);
              }

              if ((compare_ether_addr(eth->h_dest, skb->dev->dev_addr) == 0 ||
          is_broadcast_ether_addr(eth->h_dest)) &&
                  (compare_ether_addr(eth->h_source, skb->dev->dev_addr) != 0))
                 {
                  if(w2->pending == 0)
                        {
                                    skb_push(skb,W2LAN_ETH_HLEN);
                                    memcpy(skb->data,w2->orig_eth,W2LAN_ETH_HLEN);
                                    w2_netif_rx(skb);
                                    w2->pending = 1;
                        }
                 }
end:
              put_foo(w2);
            }
}
```

3.3 Salida de los Datos.

La salida de cualquier paquete pasa por la disciplina de cola w2lan_qdisc. Las operaciones de cola de espera se definen en el archivo w2lan_output.c junto a otras funciones que sirven para construcción y el envió de las tramas w2lan.

```
struct Qdisc_ops w2lan_qdisc_ops __read_mostly = {

    .id        =   "w2lan",
    .enqueue   =   w2lan_enqueue,
    .dequeue   =   qdisc_dequeue_head,
    .peek      =   qdisc_peek_head,
    .drop      =   qdisc_queue_drop,
    .init      =   w2lan_qdisc_init,
    .reset     =   qdisc_reset_queue,
    .owner              =   THIS_MODULE,

 };
EXPORT_SYMBOL(w2lan_qdisc_ops);

int init_w2lan_output(void)
{
    int ret;
    ret = register_qdisc(&w2lan_qdisc_ops);
    return ret;
}

void exit_w2lan_output(void)
{
    unregister_qdisc(&w2lan_qdisc_ops);
}
```

si nos fijamos en la estructura w2lan_qdisc_ops, se han definido dos funciones propias del protocolo: w2lan_enqueue y w2lan_qdisc_init, las demás funciones son genéricas definidas en el archivo include/net/sch_generic. El objetivo principal de la cola w2lan_qdisc es la filtración los paquetes w2lan.

```
static int w2lan_enqueue(struct sk_buff *skb, struct Qdisc *sch)
{
            struct net_device *dev = qdisc_dev(sch);

            unsigned char *p;
            p = &skb->data[12];

            if((*p ^ w2_flag))
             skb = w2lan_generate_announce(skb);

            if(skb == NULL)       {
                        sch->qstats.drops++;
                        return NET_XMIT_DROP;
            }
      if (sch->q.qlen < dev->tx_queue_len) {
                         __skb_queue_tail(&sch->q, skb);
             sch->bstats.bytes += qdisc_pkt_len(skb);
             sch->bstats.packets++;
             return 0;
       }
            return NET_XMIT_DROP;
}
```

la función w2lan_qdisc_init simplemente incializa la variable w2lan_dev con la interfaz de red elegida.

```
static int w2lan_qdisc_init(struct Qdisc *sch, struct nlattr *opt)
 {
       w2lan_dev = qdisc_dev(sch);
       return 0;
 }
```

por otro lado la función w2_skb_create nos devuelve un puntero a una trama w2lan creada según su tipo Announce,Request o Data.

```
struct sk_buff *w2_skb_create(struct w2lan_data *w2,unsigned char *daddr,int size,int type){

                unsigned char *p;
                struct sk_buff *skb;
                struct ethhdr *eth;


                skb = alloc_skb(size + LL_ALLOCATED_SPACE(w2lan_dev),GFP_ATOMIC)
                if(skb == NULL)
                        return skb;

                skb_reserve(skb,LL_RESERVED_SPACE(w2lan_dev));

                skb_reset_network_header(skb);
                p =(unsigned char*)skb_put(skb,size);

                memcpy(p,w2->convID,W2LAN_CONVID_LEN);

                if(type == SEND_DATA)
                         memcpy(&p[6],w2->data,w2->len);

                if(type == SEND_ANNOUNCE)
                         memcpy(&p[6],w2->orig_eth,W2LAN_ETH_HLEN);

                eth = (struct ethhdr*)skb_push(skb,W2LAN_ETH_HLEN);
                skb_reset_mac_header(skb);

                if(type == SEND_ANNOUNCE && daddr == NULL)
                {
                        memcpy(eth->h_dest,broadcast,W2LAN_ETH_ALEN);
                        memcpy(eth->h_source,w2lan_dev->dev_addr,W2LAN_ETH_ALEN);
                        eth->h_proto = htons(W2LAN_A);
                }
                else if (type == SEND_REQUEST)
                {
                        memcpy(eth->h_dest,daddr,W2LAN_ETH_ALEN);
                        memcpy(eth->h_source,w2lan_dev->dev_addr,W2LAN_ETH_ALEN);
                        eth->h_proto = htons(W2LAN_R);
                }
                else
                {
                        memcpy(eth->h_dest,broadcast,W2LAN_ETH_ALEN);
                        memcpy(eth->h_source,w2lan_dev->dev_addr,W2LAN_ETH_ALEN);
                        eth->h_proto = htons(W2LAN_D);
                }

                skb->len = size + W2LAN_ETH_HLEN;
                skb->dev = w2lan_dev;
                return skb;
}
```

Finalmente la función generate_announce es llamada cada vez que se interceptan paquetes salientes que no son del tipo w2lan o cuando el modulo opera en el modo Reactivo o Always.

```
struct sk_buff *w2lan_generate_announce(struct sk_buff *skb)
{
	struct w2lan_data *w2=NULL;
	struct w2lan_hdr_rcv *w2_hdr=NULL;

	w2 = w2lan_alloc();
	if(w2 == NULL )
		return NULL;

	get_random_bytes(w2->convID,W2LAN_CONVID_LEN);

	memcpy(&w2->orig_eth,&skb->data[0],W2LAN_ETH_HLEN);

	w2->len = skb->len - W2LAN_ETH_HLEN;
	if(w2->len > 0)
		w2->data = kmalloc(w2->len,GFP_ATOMIC);
	 if(w2->data==NULL){
	  put_foo(w2);
	  kfree_skb(skb);
	  return NULL;
	}

	memcpy(&w2->data[0],&skb->data[W2LAN_ETH_HLEN],w2->len);
	kfree_skb(skb);
	skb = w2_skb_create(w2,NULL,W2LAN_ANNOUNCE_LEN,SEND_ANNOUNCE);
	set_w2lan_data(w2,W2_LIST_TX);
	w2_hdr = (struct w2lan_hdr_rcv *)eth_hdr(skb);
	is_duplicate(w2_hdr);

	return skb;
}
```

3.4 Escenario de pruebas.

3.4.1 Plataforma de desarrollo.

El desarrollo del código se ha llevado a cabo sobre la plataforma VirtualBox operando con un kernel versión 2.6.35. El uso de un entorno virtualizado ofrece muchas ventajas y comodidad a la hora de probar, modificar o programar un sistema operativo y incluso para su depuración. También ofrece la posibilidad de experimentar las funcionalidades de red usando una sola maquina. Sin embargo, para la simulación de redes inalambricas existen otras herramientas como ns-2, opnet, omnet++ , etc.

la depuración en el kernel se puede realizar de distintas maneras según la necesidad que tenemos. Existen diferentes categorías y son :

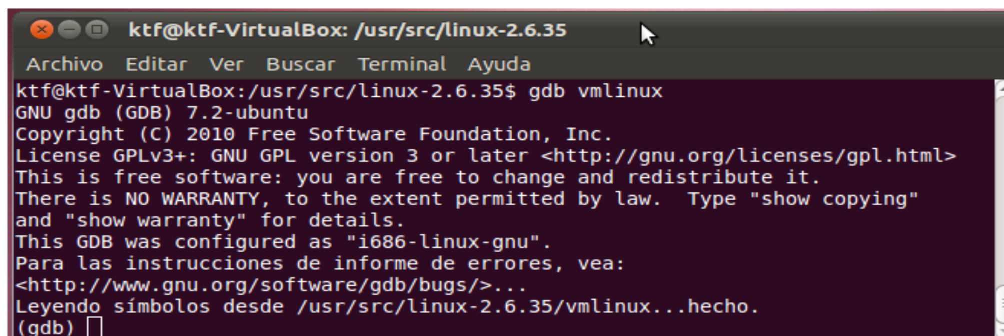
- *El análisis post-mortem (dump).*
- *Las herramientas de rastreo(printk,Ftrace).*
- *Las herramientas de perfilado (profiling).*
- *La depuración (kgdb).*

La depuración con kgdb necesita al menos 2 equipos. Un equipo "host" donde se realiza el desarrollo y la compilación del código y un equipo "target" que ejecuta una imagen del sistema modificado. La comunicación entre ellos se realiza mediante el puerto serie.
En el caso de los módulos cargables, se añade la dirección donde esta ubicado en memoria como veremos a continuación.

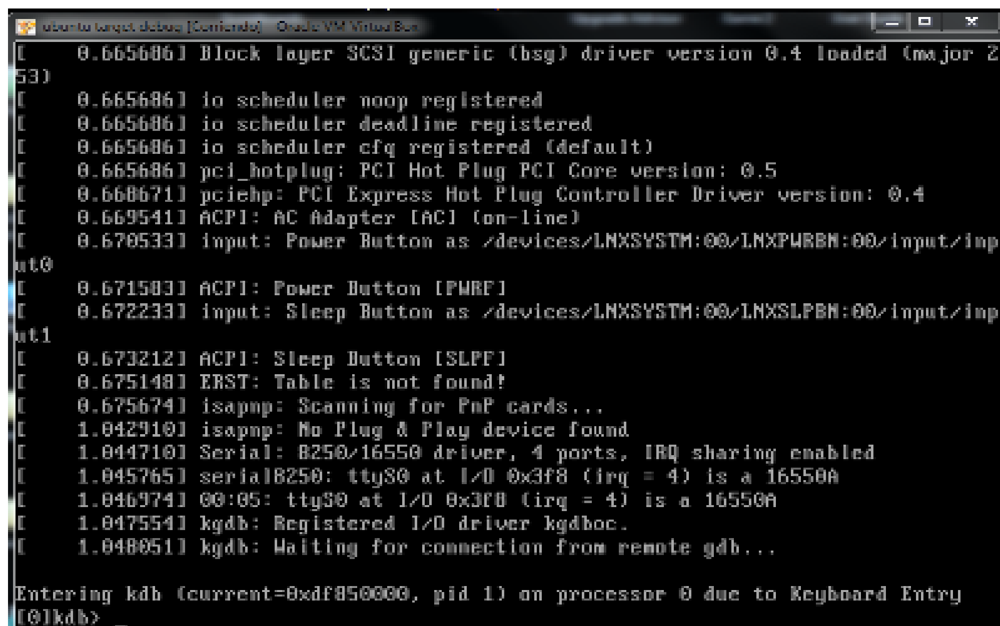
Para la puesta en marcha del depurador kgdb se debe compilar el núcleo activando las opciones de depuración en el menu de configuración "kernel hacking" y crear una nueva entrada en el gestor de arranque Grub.

Los pasos para la depuración son los siguientes :

- *En el directorio donde se ubica la imagen generada,iniciamos el depurador mediante el comando gdb vmlinux .*



- *Arrancamos la maquina target que se quedara a la espera de una conexión.*



- *En el host,ejecutamos el comando gdb> target remote /dev/ttyS[nº de puerto] ,y gdb>continue o cont que devuelve el control a la maquina target.*

```
ktf@ktf-VirtualBox: /usr/src/linux-2.6.35
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
^CQuit
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
aviso: Respuesta remota inválida: ?
aviso: Respuesta remota inválida:
^C^CInterrupted while waiting for the program.
Give up (and stop debugging it)? (y o n) y
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
kgdb_breakpoint (new_dbg_io_ops=0xc07d47c0) at kernel/debug/debug_core.c:967
967             wmb(); /* Sync point after breakpoint */
(gdb)
```

- *Cargamos el modulo en la maquina target y recuperamos las direcciones de las secciones .text,.data,.bss y .rodata que corresponden a diferentes elementos de un programa ejecutable o de un modulo. Esta información se encuentra en el fichero virtual /sys/module/w2lan/sections/.text y se obtiene mediante un cat. Para devolver el control a la maquina host se utiliza la siguiente combinación de teclas Ctrl+print+g.*

- *Finalmente de nuevo en el host, cargamos la dirección del modulo en kgdb mediante el comando gdb>add-symbol-file /directorio de los archivos fuentes del modulo/ w2lan.ko at [dirección del modulo] y ya podemos depurar el modulo con gdb como si fuera una aplicación de usuario.*

3.4.2 Configuración.

Para poder realizar las pruebas de funcionamiento del protocolo tenemos que montar una red ad-hoc de 3 nodos como mínimo. Para ello, utilizaremos las herramientas ifconfig e iwconfig para configurar la interfaz de red:

- *iwconfig [interface] mode ad-hoc*
- *iwconfig [interface] essid [nombre de la red].*
- *Ifconfig [interface] [dirección IP].*

el siguiente script bach facilita la tarea de configurar la red,cargar o descargar el modulo.

```
#!/bin/bash
dev
mode

clear
while [ 1 ]; do

echo -n "W2lan_config$"$'\n'
echo -n $'\t' "1-> Configurar la interfaz de red."$'\n'
echo -n $'\t' "2-> insertar el modulo."$'\n'
echo -n $'\t' "3-> descargar el modulo."$'\n'
echo -n $'\t' "q-> salir."$'\n'
echo -n "eliga una opcion:"
read reps

case $reps in
  1 )
    echo -n "introduce el nombre de la interfaz : "
    read dev
    if ifconfig $dev down ; then
    sleep 2
    iwconfig $dev mode ad-hoc
    sleep 2
    ifconfig $dev up
    echo -n "introduce el nombre de la red : "
    read name
    iwconfig $dev essid $name
    echo -n "introduce una @ip : "
    read ip
    ifconfig $dev $ip
    else echo "error : "
    fi
    echo ;;
  2 )
    if $dev 2>>/dev/null ; then
    echo -n "vuelve a introducir el nombre de la interfaz de red:"
    read dev
    fi
    echo  "elige el modo de funcionamiento : "$'\n'
    echo  $'\t\t\t' "1-> Always."$'\n'
    echo  $'\t\t\t' "2-> Reactive."$'\n'
    echo  $'\t\t\t' "3-> Passive."$'\n'
    echo -n "$:"
    read reps
    case $reps in
    1 ) mode="always";;
    2 ) mode="reactive";;
    3 ) mode="passive "
        break;;
     * )
    mode="passive";;
    esac
    insmod w2lan.ko mode==$mode
    tc qdisc add dev $dev root w2lan;;
  3 )
    if $dev 2>>/dev/null ; then
    echo -n "vuelve a introducir el nombre de la interfaz de red:"
    read dev
    fi
    tc qdisc del dev $dev root w2lan
    rmmod w2lan.ko;;
  q | "exit" )
    echo quit.
    break;;
  * )
              echo "";;
esac
done
exit 0
```

Conclusiones y líneas futuras:

Todos los objetivos planteados en este proyecto fin de carrerqa se han alcanzado satisfactoriamente.

El objetivo principal de este proyecto fin de carrera se ha completado con éxito, consiguiendo integrar el protocolo W2lan a la pila de red. Se ha mantenido la mayor parte de las especificaciones definidas por el director del proyecto, sin embargo, en esta versión la utilización de los temporizadores se ha limitado solo a controlar el tiempo de vida de una conversación en curso ya que la solución elegida fue transmitir las tramas Data justo después de la recepción de la primera trama Request. Esto nos permitió ademas de reducir el tiempo de vida de las conversaciones la utilización de w2lan como suporte a diferentes protocolos ya establecidos que requieren un tiempo menor de respuesta.

Otro de los objetivos alcanzado fue la implementación de W2lan en dispositivos reales y las pruebas realizadas en una red ah-hoc que presenta el problema del nodo oculto consiguiendo una visibilidad total entre todos los dispositivos.

Por otro lado, las dificultades encontradas durante la desarrollo del protocolo eran principalmente el entorno de programación que implica la utilización de varios equipos y la puesta en marcha del escenario de pruebas ya que se trata de redes de radiofrecuencia .

Como línea futura está el desarrollo del propio código. La solución elegida en este PCF representa una posible implementación de W2lan en Linux ya que el uso de las colas qdisc como mecanismo para interceptar las tramas salientes es un método poco ortodoxo. Se podría usar la interfaz Netfilter que ofrece varios hooks o ganchos, o bien implementar el protocolo como un periférico de red virtual mediante la interfaz net_device desviando y controlando de esta manera todo el trafico.

Bibliografía:

- [1] *Francesc Burrull i Mestres, ―Contribución a la Evaluación y Diseño de Protocolos Broadcast para Redes LAN Ethernet y MANET‖, Tesis Doctoral, Junio 2005.*

- [2]*Linux Kernel Development (2nd Edition).*