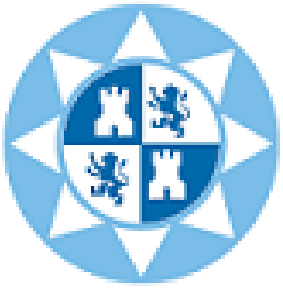


[ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN]

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Mecanismo para el acceso público a servidores con direccionamiento privado.

Proyecto Fin de Carrera

Autor: José María Vidal Fidel
Director: José María Malgosa Sanahuja
Titulación: Ingeniería Técnica de Telecomunicación esp. Telemática
Departamento: TIC

Cartagena, Diciembre de 2011

Contenido

1. Introducción

1.1 Planteamiento general del problema

El objetivo principal del proyecto es dar accesibilidad desde Internet a servidores con direccionamiento privado. Una de las razones por las que se suele utilizar direccionamiento privado es por la escasez de direcciones IPv4 públicas, que se han agotado progresivamente debido al crecimiento exponencial de Internet. Además, para que una red privada tenga acceso a otras redes necesita algún tipo de mecanismo, y el más utilizado es el llamado NAT (Network Address Traslation). Este mecanismo de traducción de direcciones de red es utilizado por *routers* IP para intercambiar paquetes entre dos redes con direccionamiento incompatible. Su uso más común es permitir utilizar direcciones privadas para acceder a Internet. Principalmente se distinguen dos modalidades a la hora de establecer una NAT: la DNAT (Destination NAT) y la SNAT (Source NAT).

- **DNAT**: mediante esta técnica se modifica la dirección de destino del paquete antes de tomar la decisión de encaminamiento, de esta manera se pueden desviar paquetes a diferentes destinos privados aunque contemos solamente con una dirección pública, generalmente por redirección de puertos de la IP pública hacia direcciones IP de la red privada. Se suele utilizar para publicar un servicio situado en una red privada en una dirección IP de acceso público.

- **SNAT**: esta técnica permite modificar la dirección de origen del paquete justo antes de devolverlo a la red, en la interfaz de salida. Un caso especial de SNAT es el *masquerading* (enmascaramiento), que sólo debe ser utilizado cuando las direcciones IP son asignadas de forma dinámica. En este caso, no es necesario escribir la dirección de origen de forma explícita, ya que por defecto se utiliza la dirección origen de la interfaz por la que el paquete sale.

Conocidos estos mecanismos, se podría pensar una posible solución utilizando DNAT para ofrecer servicios ubicados en redes privadas, con ayuda de un servidor auxiliar que indicase la dirección y puerto de los servicios públicos. Sin embargo, esta solución sería demasiado compleja para un usuario normal y corriente.

1.2 Soluciones propuestas:

Las soluciones que existen para poder acceder desde Internet a servidores situados en el hogar (es decir, con direccionamiento privado) son:

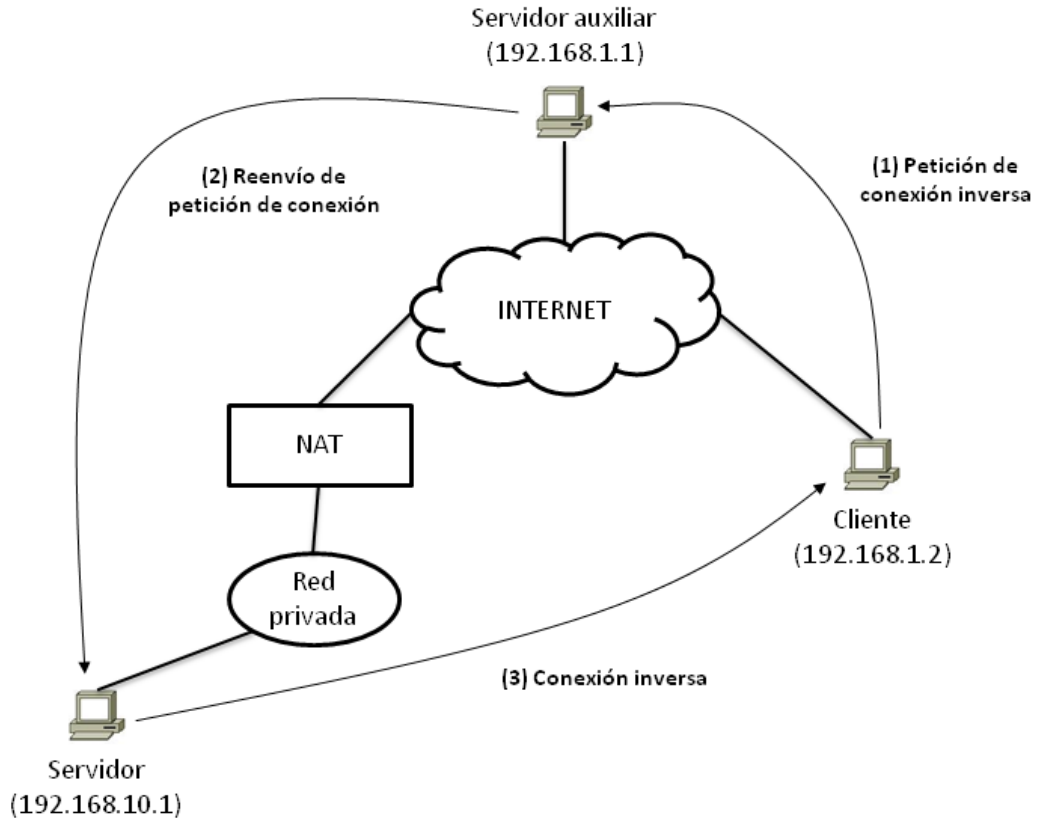
- IPv6: descartada. En cualquier caso, se pospone su implementación al momento en que los operadores apuesten decididamente por esta solución.

- Connection reversal junto con SNAT modificado: esta última es la que se ha elegido por su simplicidad y viabilidad de implementación. A continuación se detalla dicha técnica.

Un servidor que se encuentra tas una NAT ofrece distintos tipos de servicios (web, ftp, etc.). Los clientes sólo pueden acceder a estos servicios mientras el mecanismo implementado por la NAT sea DNAT y exista un servidor auxiliar que informe de los puertos de los servicios. El objetivo es conseguir que los clientes utilicen los servicios sin que el servidor se encuentre tras un DNAT. Para ello, se utilizará un servidor auxiliar que hará de intermediario. La interactuación de este servidor auxiliar con el resto de servidores y clientes es la siguiente:

- Por un lado, los servidores se conectarán y se mantendrán conectados a este servidor auxiliar. Además, quedarán registrados en una base de datos, a la espera de peticiones de clientes.
- En el caso de los clientes, realizarán una petición al servidor auxiliar indicando el servicio que desean utilizar. De este modo, el auxiliar le comunicará al servidor correspondiente que un cliente desea utilizar sus servicios. Por tanto este servidor se encargará de "abrir un camino" en la NAT con los datos de dicho cliente (IP y puerto requerido) para que sea posible el acceso a los servicios.

El método para realizar este tipo de comunicación es conocido como *Connection Reversal*.
En el siguiente esquema gráfico se ilustra el mecanismo (las direcciones IP son un ejemplo).



Descripción del escenario

El cliente quiere utilizar un servicio del servidor principal. Como no puede conectarse directamente dado que el servidor se encuentra tras una SNAT, envía petición de servicio al servidor auxiliar.

El servidor auxiliar conoce la dirección del servidor principal, por lo que se encarga de retransmitir la solicitud del cliente al servidor principal, proporcionándole la dirección IP origen del cliente y el puerto por el que quiere comunicarse.

El servidor principal envía un mensaje de respuesta al cliente, abriendo un camino en la SNAT por donde el cliente puede comunicarse con él y utilizar su servicio.

2. Implementación

2.1 Descripción de cada uno de los elementos.

- Servidor auxiliar.

El servidor auxiliar es el gran pilar del sistema. La primera acción que realiza tras su ejecución va relacionada con la gestión de su propia base de datos (mysql). En primer lugar, conecta con

el servidor mysql que ha de estar instalado en el sistema que se ejecuta, y tras esta conexión, lo primero que comprueba es si existe una base de datos creada anteriormente. La primera ejecución de este servidor en un sistema creará la base de datos. Seguidamente comprobará la existencia previa de una tabla donde se almacenarán los servidores conectados, como en el caso anterior, si no existe será creada. El siguiente paso es la creación y configuración del socket para permanecer a la escucha y recibir conexiones (tanto de servidores como de clientes). Inmediatamente, tras la conexión de un servidor o un cliente, distingue de qué tipo se trata, puesto que cada uno se identifica enviando un *string* diferenciado. A partir de este punto, el servidor auxiliar actuará completamente diferente en función del tipo de conexión recibida: un cliente o un servidor.

- Caso de recepción de un servidor:

Tras inicializar la variable de condición del hilo (más adelante se utilizará para despertar el hilo) y almacenar sus datos, se crea un hilo en base a una función genérica para los servidores y se continua con la ejecución. El siguiente paso es extraer los datos que nos envía el servidor para guardarlos posteriormente en la base de datos. A partir de este momento, se llama a la variable de condición para esperar la llegada de algún cliente que despierte al hilo. Cuando un cliente requiere los servicios de este servidor, el hilo captura los datos del cliente y los envía al servidor principal que va a ser el que se encargue de crear un camino en la NAT con el cliente.

- Caso de recepción de un cliente:

Como ocurría en el caso de la recepción de un servidor, cuando llega un cliente, lo primero es almacenar sus datos y posteriormente crear un hilo en base a una función genérica para los clientes. Aquí también se extraen los datos enviados por el cliente, pero en este caso no va a ser para almacenarlos; estos datos contienen la petición del cliente, que puede ser de tres tipos. El primero es el caso en el que no se indica ni el nombre de un servidor ni el tipo de servicio que le interesa, por tanto el servidor auxiliar le responde con un listado de los servidores publicables registrados en la base de datos. El segundo tipo es parecido al anterior, pero en este caso ha indicado un tipo en concreto de servicio, por tanto el cliente recibirá un listado filtrado por ese tipo de servicio y que a su vez sea publicable. El tercer caso es el más importante; el cliente envía el nombre del servidor con el que quiere contactar y en caso de que exista en la base de datos, se almacena la dirección IP origen y puerto por el que quiere comunicar en variables compartidas y protegidas. Acto seguido, se despierta al hilo del servidor requerido por el cliente para que recoja los datos compartidos.

- Servidor

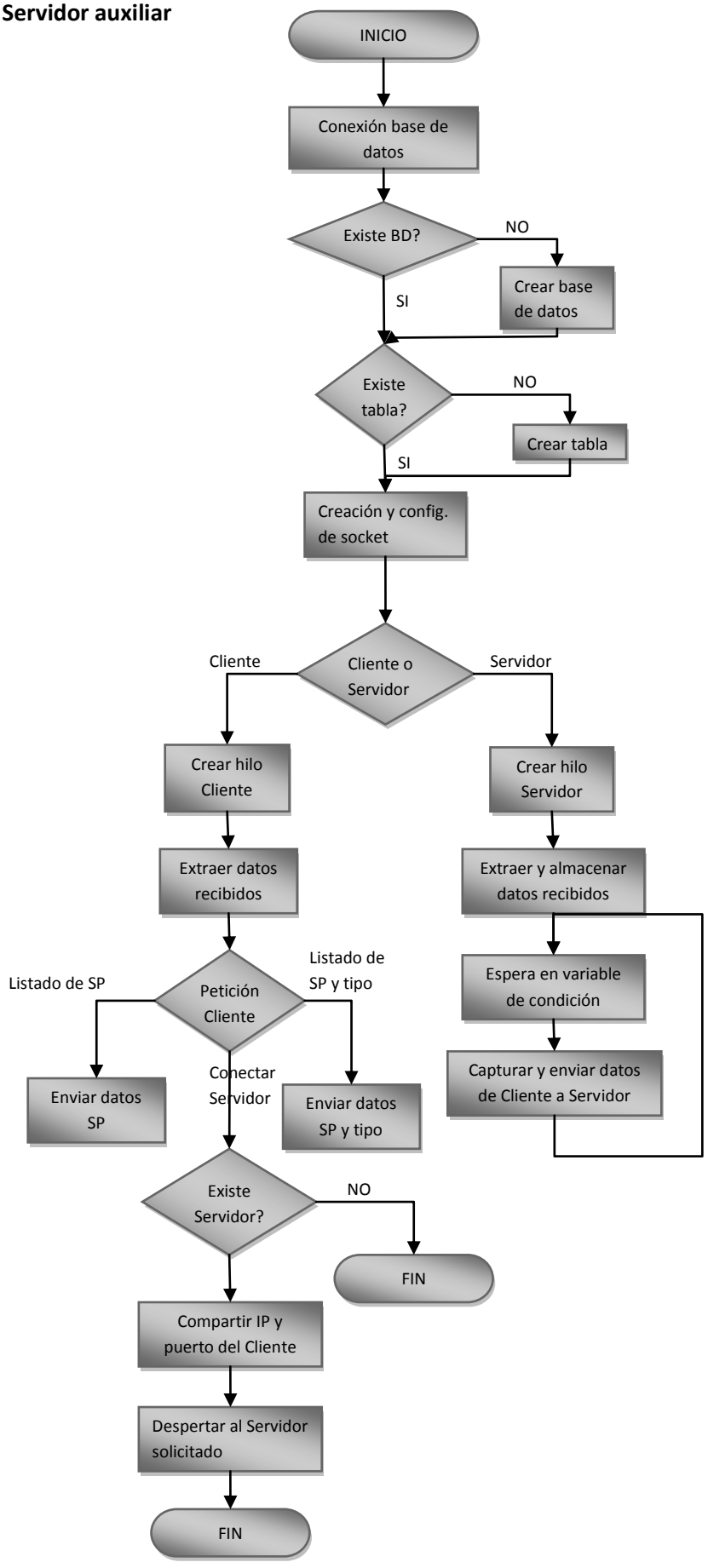
Lo primero que realiza la parte de código del servidor es la creación y configuración del socket para conectarse al servidor auxiliar. Inmediatamente después, envía los datos previamente introducidos por línea de comando en la ejecución del programa. Tras este envío pueden darse dos situaciones: que el nombre del servidor ya esté siendo utilizado, por tanto habría que ejecutar de nuevo el programa proporcionando otro nombre, o bien, que el nombre del servidor no exista y el programa continúe su ejecución. En el caso de que el servidor no exista, el programa entra en un bucle, esperando recibir datos de algún cliente enviados por el servidor auxiliar. Tras la petición de un cliente, se recibe su dirección IP y el puerto por el que quiere comunicarse. Estos datos van a quedar almacenados para ser utilizados en la creación del paquete *raw*. El siguiente paso es la creación de un socket *raw*. Seguidamente se procede a rellenar las cabeceras IP y TCP del paquete que va a ser enviado, lo que incluye un cálculo de *checksum* para cada cabecera. Por último, se envía el paquete *raw* construido.

- Cliente

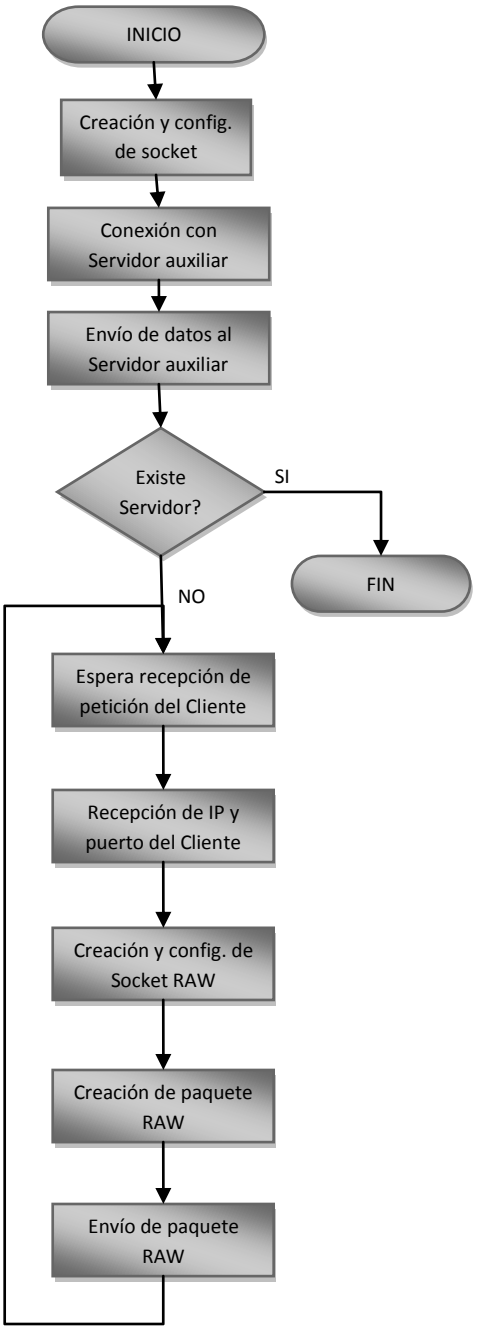
Como ocurría con el servidor, el primer paso del código del cliente es conectarse con el servidor auxiliar. Inmediatamente después, se envían los datos de petición y pueden darse tres casos. Si en la petición no se indica ni tipo de servicio ni servidor, recibirá una lista de los servidores publicables. Si se indica el tipo pero no se indica el servidor, recibirá una lista de los servidores publicables del tipo solicitado. Si se indica el nombre del servidor con el que se quiere contactar, también se envía el puerto por el que se desea comunicar y se pueden recibir dos tipos de respuesta. Que el servidor no se encuentre en la base de datos, por tanto no podrá haber comunicación o bien que el servidor esté registrado en la base de datos, de modo que se producen las gestiones necesarias (mediante los hilos) para que el servidor auxiliar contacte con el servidor y se consiga el objetivo.

2.2 Diagramas de bloque

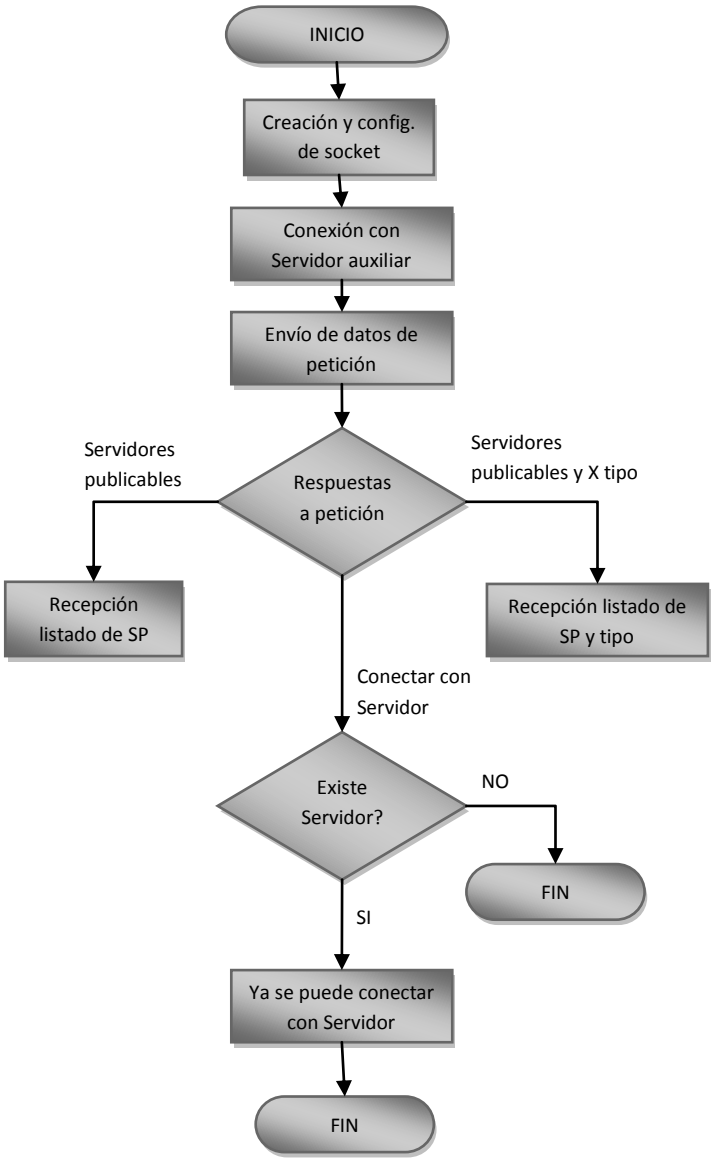
Servidor auxiliar



Servidor


INICIO
Creación y config. de socket
Conexión con Servidor auxiliar
Envío de datos al Servidor auxiliar
Existe Servidor?
SI
FIN
NO
Espera recepción de petición del Cliente
Recepción de IP y puerto del Cliente
Creación y config. de Socket RAW
Creación de paquete RAW
Envío de paquete RAW

Cliente


INICIO
Creación y config. de socket
Conexión con Servidor auxiliar
Envío de datos de petición
Respuestas a petición
Servidores publicables
Servidores publicables y X tipo
Recepción listado de SP
Recepción listado de SP y tipo
Conectar con Servidor
Existe Servidor?
NO
FIN
SI
Ya se puede conectar con Servidor
FIN

2.3 Generalidades de los POSIX Threads.

- Introducción.

La librería *pthreads* es una librería que cumple los estándares POSIX y que permite trabajar con distintos hilos de ejecución (*threads*) al mismo tiempo.
La diferencia entre un hilo y un proceso es que los procesos no comparten memoria entre sí, a no ser que se haya declarado explícitamente usando alguno de los mecanismos de IPC (*InterProcess Communication*) de Unix, mientras que los hilo sí que comparten totalmente la memoria entre ellos.
Ya que pthreads es una librería POSIX, se pueden portar los programas hechos con ella a cualquier sistema operativo POSIX que soporte hilos.

- Compilar un programa que utiliza hilos.

Para crear programas que hagan uso de la librería pthreads se necesita, en primer lugar, la librería en sí. Se puede encontrar en la mayoría de distribuciones linux, y seguramente se instale al mismo tiempo que los paquetes incluidos para el desarrollo de aplicaciones. Una vez instalada la librería, se compila el programa incluyendo la librería. La forma más usual de hacer esto es, en el caso del compilador GNU gcc con el comando:

```
gcc programa_con_pthreads.c -o programa_con_pthreads –lpthread
```

- Crear y manipular hilos.

Para crear un hilo hay que llamar a la función *pthread_create()***.** Su prototipo es el siguiente:

```
int pthread_create (pthread_t * hilo, pthread_attr_t *attr, void *
(*funcion)(void *), void *arg);
```

• **hilo**: Es una variable de tipo *pthread_t* que contendrá los datos del hilo y que servirá para identificarlo cuando interese hacer llamadas a la librería para llevar a cabo alguna acción sobre él.

• **attr**: Es un parámetro de tipo *pthread_attr_t* que se debe inicializar previamente con los atributos que deba tener el hilo. Lo recomendable es pasarle **NULL** a la función, de esta forma se le asignará al hilo unos atributos por defecto.
• **funcion**: Aquí hay que poner la dirección de la función que ejecutará el hilo, la cual debe devolver un puntero genérico *(void *)* como resultado, y debe tener como único parámetro otro puntero genérico. La ventaja de que estos dos punteros sean genéricos es que se puede devolver cualquier cosa mediante los *castings* de tipos necesarios.
Para pasar o devolver más de un parámetro a la vez, se puede crear una estructura y meter allí dentro todo lo que se necesite. Luego se pasa la dirección de esta estructura como único parámetro.
• **arg**: Es un puntero al parámetro que se le pasará a la función. Puede ser **NULL** si no hay que pasarle nada a la función.

Para finalizar la ejecución de un hilo, él mismo debe llamar a la función *pthread_exit()* cuyo prototipo es el siguiente:

```
void pthread_exit (void *datos);
```

• **datos**: Es un puntero genérico a los datos que se quieren devolver como resultado. Estos datos serán recogidos más tarde llamando a la función *pthread_join()* con el identificador del hilo deseado.

Los datos devueltos por *pthread_exit()* son recogidos con *pthread_join()*, que espera a que termine la ejecución del hilo para hacerlo. Su prototipo es el siguiente:

```
int pthread_join (pthread_t id_hilo, void **datos);
```

• **id_hilo**: Es el identificador del hilo del cual se quieren recoger datos, y es el mismo que se obtuvo al crearlo con *pthread_create()*.
• **datos**: Es un puntero a puntero que apunta (valga la redundancia) al resultado devuelto por el hilo indicado en *id_hilo*.

Si el hilo creado no devuelve ningún resultado (como es el caso de este proyecto) o no interesa recogerlo, hay que llamar a la función *pthread_detach()* para indicar a la librería pthread que no guarde ningún resultado del hilo indicado. Esto es importante puesto que esta llamada permitirá liberar los recursos reservados para dicho hilo. Su prototipo es el siguiente:

```
int pthread_detach (pthread_t id_hilo);
```

• **id_hilo**: Es el identificador del hilo.

Existe una función llamada *pthread_self()* que sirve para que cada hilo pueda obtener su propio identificador. Tiene el siguiente prototipo:

```
pthread_t pthread_self(void);
```

Basta con crear una variable de tipo *pthread_t* y asignarle el valor devuelto por la llamada a esta función, de esta forma un hilo puede liberar por sí mismo sus recursos llamando a *pthread_detach()*.

- Problemas de concurrencia y mecanismos para solucionarlos.

Puesto que muchas variables son globales a todos los hilos, es decir, la memoria es compartida, hay que mantener un riguroso control en el acceso a estas. Si no se lleva a cabo este procedimiento, puede que el valor de alguna variable no sea el esperado, lo que conlleva a posibles *bugs* bastante difíciles de detectar. Para una correcta sincronización entre hilos se dispone de dos mecanismos, los *mutex* y las *variables de condición*.

- **Exclusión mutua o MUTEX**

Mediante los *mutex* se preservan regiones críticas y se obtiene acceso exclusivo a los recursos, es decir, se trata de indicar que una región particular de código sólo puede ser ejecutada por un determinado hilo al mismo tiempo.

Lo primero que hay que hacer es declarar una variable global de tipo *pthread_mutex_t* que será utilizada por los distintos hilos que compartan memoria. Esta variable ha de ser inicializada antes de que la utilice algún hilo. Para ello existen dos formas.
La primera de ellas es mediante el inicializador estático **PTHREAD_MUTEX_INITIALIZER**, que debemos asignar a la variable *mutex* de la siguiente manera:

```
pthread_mutex_t  var_mutex = PTHREAD_MUTEX_INITIALIZER;
```

La otra forma de inicializar la variable es mediante la función *pthread_mutex_init()*, que como diferencia permite definir atributos propios del *mutex*, frente al inicializador estático que inicializa el *mutex* con los atributos por defecto. Su prototipo es el siguiente:

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Para inicializar el *mutex* con los atributos por defecto hay que pasar **NULL** como segundo argumento a la función. En este proyecto los atributos se han establecido por defecto.

Una vez inicializada la variable, el método para acceder a las variables compartidas es bloqueando y desbloqueando el *mutex*. Para ello se dispone de dos funciones, *pthread_mutex_lock()* y *pthread_mutex_unlock()*, para bloquear y desbloquear respectivamente. Sus prototipos son los siguientes:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

El funcionamiento es el siguiente: el hilo que desee utilizar o modificar variables globales, llama a la función *pthread_mutex_lock()* pasándole la variable *mutex* previamente inicializada, seguidamente debe ir el fragmento de código que utiliza las variables globales. Tras esto, el mismo hilo debe liberar el *mutex* llamando a la función *pthread_mutex_unlock()*, de forma que otro hilo podrá utilizar el *mutex* y acceder a esas variables compartidas.

- **Variables de condición**

Mediante este mecanismo, es posible suspender un hilo hasta que sea necesario despertarlo (desde otro hilo). Una variable de condición está siempre asociada con un *mutex* particular y con los datos protegidos por el *mutex*. El funcionamiento básico es el siguiente: el hilo que ha de ser suspendido llama a una función de espera y se queda bloqueado. Por otra parte, un hilo que quiera compartir algo con el hilo bloqueado, en un momento determinado debe llamar a otra

función que envía una señal al hilo suspendido y lo despierta, de modo que toma el bloqueo del *mutex* mientras realiza sus tareas y lo libera al terminar.

Estas variables de condición son de tipo *pthread_cond_t* y el primer paso para utilizarlas, al igual que ocurre con los *mutex*, es la inicialización. También existen dos métodos, el inicializador estático y la función de inicialización. El primero se declara de la siguiente manera:

```
pthread_cond_t  v_cond = PTHREAD_COND_INITIALIZER;
```

Y mediante la función de inicialización:

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Como segundo argumento se debe pasar siempre **NULL** salvo que se deseen cambiar los atributos por defecto.

Mediante la función *pthread_cond_wait()* y una variable de condición, se suspende el hilo a la espera de ser despertado. Paralelamente, otro hilo ha de llamar a *pthread_cond_signal()* utilizando la variable condición que se le pasó a la función de espera, de modo que el hilo bloqueado despierta y continua su ejecución. Los prototipos de estas funciones son los siguientes:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
```

- **cond**: variable de condición previamente inicializada.
- **mutex**: variable *mutex* utilizada para proteger la memoria compartida.

2.4 Los POSIX Threads en este proyecto.

Surgen por la necesidad de comunicar procesos de forma sencilla y eficiente. Se han estudiado los diferentes métodos de comunicación entre procesos (IPC), tales como tuberías, colas de

mensajes, semáforos y memoria compartida, frente a las características de la programación mediante hilos, llegando a ciertas conclusiones. La opción de programar la concurrencia con procesos (mediante *fork()*) obliga a utilizar los métodos de IPC, donde es posible que tenga que intervenir el núcleo y esto supone un punto negativo desde el punto de vista de la eficiencia. En cuanto a la facilidad de uso, los hilos, concretamente la librería de POSIX threads, son indiscutiblemente más cómodos de utilizar, comparten el mismo entorno por naturaleza y además, existen unas funciones definidas para comunicarlos garantizando la consistencia. Por cuestiones de rendimiento y ante la posibilidad de reusabilidad y expansión del código, hay que tener en cuenta que el coste de crear un proceso es mucho mayor que el de crear un hilo, por tanto, programando mediante hilos el rendimiento es muy superior frente a la programación con procesos.

En este proyecto, los hilos se utilizan sólo en la parte del servidor auxiliar y son de dos tipos, uno de ellos para atender a los clientes y otro para atender y registrar a los servidores, que quedarán conectados a la espera de clientes. Los prototipos de las funciones de estos hilos son las siguientes:

```
void *f_cliente(thr_data *arg);
void *f_servidor(thr_data *arg);
```

Se ha definido una estructura llamada *thr_data* para poder pasarle a estas funciones los datos recibidos de clientes y servidores. Esta estructura almacena sus datos, su identificador de socket y su dirección IP.

El hilo de los servidores se encarga de recoger la información del servidor que se ha conectado, introducirlo en la base de datos y mantenerse bloqueado mediante una variable condición a la espera de peticiones de clientes. Cuando tiene lugar una llegada, el hilo despierta y continúa su ejecución, captura los datos del cliente que se encuentran en variables compartidas, los envía al servidor en cuestión y vuelve a bloquearse a la espera de otra petición (bucle infinito). Por otra parte, el hilo de los clientes recibe también los datos del cliente y los interpreta. Si el cliente no indica nombre del servidor, se le enviará un listado de los servidores. En caso de que indique el servidor con el que quiere conectar, se buscará en la base de datos y si existe, se obtendrá su variable de condición y se enviará una señal para despertar al hilo que lo gestiona. Este hilo morirá tras finalizar su tarea. Cada hilo se encarga de liberar los recursos por sí mismo, obteniendo su propio identificador de hilo y pasándoselo a la función *pthread_detach(),* de modo que no ocupen memoria de forma innecesaria.

En cuanto a los *mutex*, se han definido dos: uno para el acceso a la base de datos y otro para el acceso a las variables compartidas. Se ha mantenido la disciplina de bloquearlos y desbloquearlos en cada acceso, en ambos hilos.

2.5 Los socket RAW

Un socket *raw* es un tipo de socket especial que permite el acceso directo al nivel de red (IP) sin necesidad de utilizar un protocolo de transporte. Estos socket tienen multitud de usos, pero en este proyecto se le ha dado concretamente el de crear un paquete "a medida", es decir, un paquete con las cabeceras IP y TCP configuradas manualmente.

Hay que cuidar ciertos detalles a la hora de programar un socket *raw*. El programa que utilice un socket *raw* ha de ser ejecutado con permisos de administrador. El valor de la suma de verificación (*checksum*) de la cabecera IP y de la cabecera TCP/UDP deben ser calculados correctamente, en caso contrario, el paquete probablemente será descartado en su destino.

Detalles de implementación:

Para crear un socket *raw* hay que llamar a la función socket pasándole los parámetros adecuados:

```
socket_raw = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

Igual que ocurre en los sockets TCP y UDP, hay que emplear una estructura de tipo *sockaddr_in* y rellenarla, con la diferencia de que el valor del campo *sin_port* no es necesario indicarlo, se pone a 0, puesto que en socket *raw* no tiene porqué utilizarse la capa de transporte.
Para rellenar la cabecera IP, se declara una variable puntero de tipo estructura *iphdr* que apunte al principio del paquete a enviar. Aquí es donde se colocará las IP origen y destino del paquete.

En cuanto a la cabecera TCP, se declara también una variable puntero, pero en este caso de tipo estructura *tcphdr* que apunte a continuación de la cabecera IP dentro del paquete que se va a enviar. Los campos más destacados a rellenar son los del puerto origen y puerto destino.

Antes de enviar el paquete hay que hacer el cálculo del *checksum* de la cabecera IP y de la cabecera TCP. Es importante poner estos valores a "0" antes de realizar los cálculos.

Por último, hay que enviar el paquete y para ello se utiliza la conocida función *sendto()*.

2.6 Checksum de las cabeceras IP y TCP.

Para estos cálculos de *checksum* se utilizarán dos funciones distintas, una se encargará de calcular el *checksum* de la cabecera IP y la otra hará lo mismo para la cabecera TCP.

La función que calcula el *checksum* de la cabecera IP es la siguiente:

```
unsigned short checksum(unsigned short *ptr, int len)
{
    int sum = 0;
    unsigned short answer = 0;
    unsigned short *w = ptr;
    int nleft = len;

    while(nleft > 1){
        sum += *w++;
        nleft -= 2;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return(answer);
}
```

Lo que hay que pasar a la función como primer argumento es la estructura de tipo *iphdr* que apunta a la cabecera IP del paquete. El segundo argumento a pasar es la longitud de la cabecera expresada en bytes. Una forma de obtener este segundo argumento sería multiplicar el campo longitud de la cabecera IP (*ihl)* por 4 para pasar de palabras de 32bits a bytes.

Por otra parte, la función que calcula el *checksum* de la cabecera TCP:

```
unsigned short tcp_sum_calc(unsigned short len_tcp, unsigned short
src_addr[],unsigned short dest_addr[], unsigned short buff[])
```

```
{
    unsigned char prot_tcp=6;
    unsigned long sum;
    int nleft;
    unsigned short *w;

    sum = 0;
    nleft = len_tcp;
    w=buff;

    /* calculate the checksum for the tcp header and payload */
    while(nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }

    /* if nleft is 1 there ist still on byte left. We add a padding
byte (0xFF) to build a 16bit word */
    if(nleft>0)
    {
      /* sum += *w&0xFF; */
              sum += *w&ntohs(0xFF00);   /* Thanks to Dalton */
    }

    /* add the pseudo header */
    sum += src_addr[0];
    sum += src_addr[1];
    sum += dest_addr[0];
    sum += dest_addr[1];
    sum += htons(len_tcp);
    sum += htons(prot_tcp);

    // keep only the last 16 bits of the 32 bit calculated sum and add
the carries
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);

    // Take the one's complement of sum
    sum = ~sum;
```

```
return ((unsigned short) sum);
}
```

El primer argumento ha de ser la longitud de la cabecera TCP más la longitud de los datos del paquete. Una forma de calcularlo sería la longitud total del paquete menos la longitud de la cabecera IP. El segundo y tercer argumento son las direcciones IP origen y destino respectivamente. Una forma de obtenerlas sería de la variable estructura de tipo *iphdr* que apunta a la cabecera IP, utilizando los campos *saddr* para la IP origen y *daddr* para la IP destino. Por último, el cuarto argumento *buff* debe contener la cabecera TCP y los datos del paquete. Para ello, utilizamos como argumento la estructura de tipo *tcphdr* que apunta a la cabecera TCP, porque lo que hay tras la cabecera son los datos del paquete.

2.7 El navegador web DILLO y sus modificaciones.

Dillo es un navegador web muy sencillo, escrito en C y C++, basado en la biblioteca de interfaz gráfica FLTK (Fast Light Toolkit). Se caracteriza por su velocidad y su pequeño tamaño (unos 350KB), por ello es especialmente interesante para computadores con pocos recursos.

Lo que llevó a escoger Dillo como navegador para efectuar las modificaciones, además del buen funcionamiento del mismo, fue su sencillez y claridad a nivel de código, completamente escrito en C y C++.

El objetivo es conseguir que el navegador utilice un puerto origen determinado en sus conexiones, en lugar de utilizar uno aleatorio como hace por defecto. Para ello se han realizado las siguientes modificaciones en el código original de la versión 2.2 de Dillo:

- Modificación para insertar el puerto por línea de comandos.

Las líneas de código que se han añadido o modificado se encuentran en negrita y la ruta del archivo modificado es la siguiente: **dillo-2.2/src/dillo.cc**

```
typedef enum {
   DILLO_CLI_NONE          = 0,
   DILLO_CLI_XID           = 1 << 0,
   DILLO_CLI_FULLWINDOW    = 1 << 1,
   DILLO_CLI_HELP          = 1 << 2,
   DILLO_CLI_VERSION       = 1 << 3,
   DILLO_CLI_LOCAL         = 1 << 4,
```

```
   DILLO_CLI_GEOMETRY      = 1 << 5,
   DILLO_CLI_PORT        = 1 << 6,
   DILLO_CLI_ERROR         = 1 << 15,
} OptID;

static const CLI_options Options[] = {
   {"-f", "--fullwindow", 0, DILLO_CLI_FULLWINDOW,
    "  -f, --fullwindow      Start in full window mode: hide address bar,\n"
    "                         navigation buttons, menu, and status bar."},
   {"-g", "-geometry",    1, DILLO_CLI_GEOMETRY,
    "  -g, -geometry GEO      Set initial window position where GEO is\n"
    "                         WxH[{+-}X{+-}Y]"},
   {"-h", "--help",       0, DILLO_CLI_HELP,
    "  -h, --help             Display this help text and exit."},
   {"-p", "--port",     1, DILLO_CLI_PORT,
    "  -p, --port PORT      Use this port as source port on requests."},
   {"-l", "--local",      0, DILLO_CLI_LOCAL,
    "  -l, --local            Don't load images for these URL(s)."},
   {"-v", "--version",    0, DILLO_CLI_VERSION,
    "  -v, --version          Display version info and exit."},
   {"-x", "--xid",        1, DILLO_CLI_XID,
    "  -x, --xid XID          Open first Dillo window in an existing\n"
    "                         window whose window ID is XID."},
   {NULL, NULL, 0, DILLO_CLI_NONE, NULL}
};

//Declaración de variable antes del main() para usarla como external en otros archivos
del propio programa.

char *puerto_en;

//Dentro del "switch (opt_id)" añadido el caso del puerto

      case DILLO_CLI_PORT:
      {
       puerto_en = opt_argv[0];
       break;
      }
```

- Modificación para utilizar el puerto capturado por argumento en línea de comandos y fijarlo mediante "**bind()**" en la petición de web del cliente.

Ruta del archivo modificado: **dillo-2.2/src/IO/http.c**

```
#include <strings.h>            // para usar la función bzero()
```

```
extern char *puerto_en;    // variable externa

/* Forzado de puerto mediante bind() */

int portint;
portint = strtol(puerto_en,NULL,10);

struct sockaddr_in cliport;

bzero(&cliport, sizeof(cliport));
cliport.sin_family = AF_INET;
cliport.sin_port=htons(portint);
cliport.sin_addr.s_addr=htonl(INADDR_ANY);

bind(S->SockFD, (struct sockaddr *) &cliport, sizeof(cliport));

/* Fin de forzado de puerto mediante bind()*/
```

Una vez añadidas estas modificaciones, es posible compilar e instalar el programa. Hay que tener en cuenta que Dillo está basado en la librería FLTK, por tanto es necesario tenerla instalada previamente. Para utilizar Dillo empleando un puerto origen concreto hay que llamar al programa de la siguiente manera: “dillo –p puerto”.

3. Netfilter

3.1 Dificultades con la NAT de Netfilter.

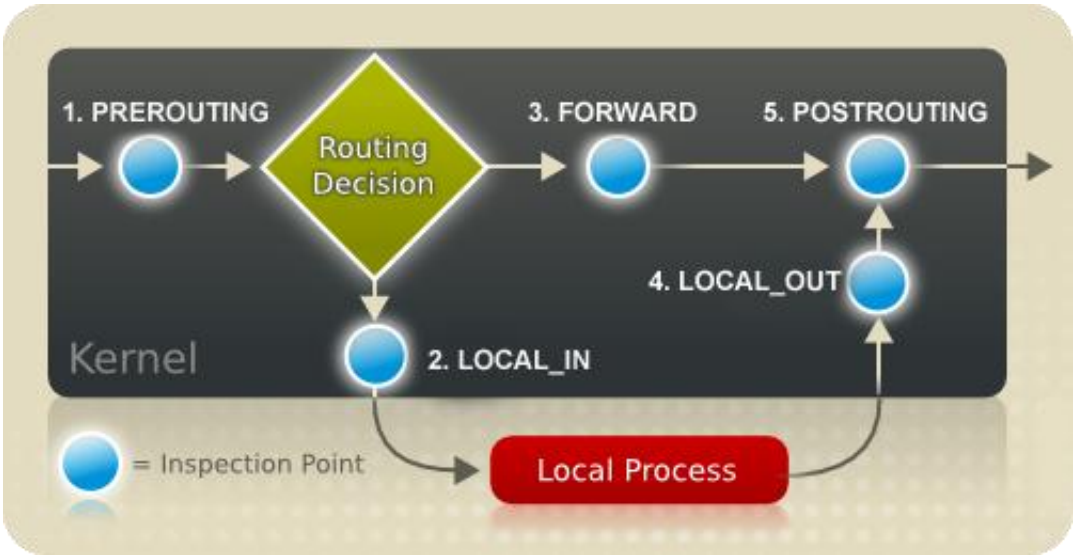
La NAT depende del seguimiento de conexiones (*connection tracking*) que realiza Netfilter. Este seguimiento permite al núcleo llevar la cuenta de todas las conexiones o sesiones lógicas de red y de este modo relacionar todos los paquetes que pueden llegar a formar parte de esa conexión. Esta característica de la NAT provoca que no se registre una conexión TCP si no está completada, por tanto no sirve para realizar el propósito de este proyecto.

La solución a este inconveniente ha sido implementar una SNAT sencilla sin seguimiento de conexión. Para ello se han estudiado generalidades sobre Netfilter, generalidades sobre los módulos del *kernel* en Linux y los módulos específicos para Netfilter.

3.2 Generalidades sobre Netfilter.

Netfilter es un *framework* que se encuentra integrado en el *kernel* de Linux y que permite interceptar y manipular paquetes. Proporciona diferentes funcionalidades de red tales como filtrado de paquetes, traducción de direcciones de red (NAT) y seguimiento de conexiones. Estos procedimientos funcionan utilizando unos ganchos que se encuentran en el *kernel* de Linux. Esto permite programar módulos para el *kernel* con funciones que se registran en alguno de estos ganchos y que son llamadas cuando ocurre algún evento específico de la red. Un ejemplo básico de estos eventos sería la recepción de un paquete.

Netfilter define cinco ganchos (*hooks*) en IPv4:



1. **NF_INET_PRE_ROUTING**: antes de las decisiones de enrutamiento.
2. **NF_INET_LOCAL_IN**: para paquetes destinados al propio host.
3. **NF_INET_FORWARD**: para paquetes destinados a otra interfaz.
4. **NF_INET_LOCAL_OUT**: para paquetes generados por procesos locales.
5. **NF_INET_POST_ROUTING**: justo antes de salir a la red.

Netfilter también define cinco valores de retorno para usar en las funciones registradas:

- **NF_DROP**: descarta el paquete, es eliminado.
- **NF_ACCEPT**: acepta el paquete (continua su camino).
- **NF_STOLEN**: la función se hace cargo del paquete, no continua el recorrido.
- **NF_QUEUE**: encola el paquete.

- **NF_REPEAT**: ejecuta la función de nuevo.

El registro de una función en uno de estos ganchos es un proceso simple que gira en torno a la estructura *nf_hook_ops*, la cual está definida de la siguiente manera:

```
struct nf_hook_ops {
struct list_head list;

/* User fills in from here down. */
nf_hookfn *hook;
int pf;
int hooknum;
/* Hooks are ordered in ascending priority. */
int priority;
};
```

Los campos más interesantes son:

- **hook**: es un puntero a la función que será llamada por el gancho.
- **pf**: especifica una familia de protocolos.
- **hooknum**: especifica el gancho en el cual se va a registrar la función.
- **priority**: especifica la prioridad en el orden de ejecución.

Lo primero que hay que declarar es estructura que se usará para registrar la función:

```
static struct nf_hook_ops nfho;
```

Un ejemplo de función que se encarga de descartar todos los paquetes tiene la siguiente estructura:

```
unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb,
const struct net_device *in, const struct net_device *out, int
(*okfn)(struct sk_buff *))
{
return NF_DROP; /* Drop ALL packets */
```

```
}
```

Un ejemplo de rutina de inicialización, donde se rellena la estructura y se registra en el gancho mediante la función *nf_register_hook():*

```
int init_module()
{
nfho.hook = hook_func; /* Handler function */
nfho.hooknum = NF_INET_PRE_ROUTING; /* First hook for IPv4 */
nfho.pf = PF_INET;
nfho.priority = NF_IP_PRI_FIRST; /* Make our function first */

nf_register_hook(&nfho);

return 0;
}
```

La rutina de limpieza es muy sencilla, simplemente elimina el registro que se hizo de la función en el gancho mediante *nf_unregister_hook()*:

```
void cleanup_module()
{
nf_unregister_hook(&nfho);
}
```

3.3 Generalidades sobre un módulo del kernel (2.6.x).

Es importante tener en cuenta que para versiones anteriores al *kernel* 2.6.x, algunos procedimientos que se van a describir a continuación se realizan de forma diferente o simplemente no son necesarios. También hay que tener instaladas las herramientas de compilar, utilidades de módulos, etc.

Lo primero que hay que tener descargado en el PC donde se va a compilar el módulo son las fuentes del *kernel* (*kernel soruce*). Si la distribución tiene algún gestor de descarga automatizado, lo más probable es que guarde la descarga en la ruta *"/usr/src/"*. Es

imprescindible que la versión del *kernel* ejecutado coincida exactamente con la versión de las fuentes del *kernel*, en caso contrario probablemente surjan errores.
El siguiente paso es configurar el kernel, para ello hay que ir a la ruta donde se descargaron las fuentes *("/usr/src/Linux/"*) y escribir alguna de las opciones: "*make oldconfig*" o "*make menuconfig*". Tras este proceso de configuración, se debe haber generado un fichero "*.config*". Con todo esto ya se puede empezar a escribir y compilar un módulo.

Un módulo funciona básicamente con dos funciones, una de inicialización y otra de salida/limpieza. Además, también existen dos formas de codificar estas dos funciones.

La primera forma supone escribir las funciones directamente, con la correspondiente estructura.

- Función de inicialización:

```
int init_module()
{ /* Aquí hay que codificar lo que tiene que hacer el módulo */
return 0;
}
```

Es importante que esta función termine devolviendo el valor 0.

- Función de salida/limpieza:

```
void cleanup_module()
{
/*Aquí hay que codificar la salida de la función. Por ejemplo, si hubo algún tipo de registro en la inicialización, aquí debe ir la función que elimina ese registro */
}
```

La segunda forma se basa en construir las funciones primero y posteriormente llamarlas. Un ejemplo completo del clásico "Hello world" para ilustrarlo:

```
#include <linux/init.h>
#include <linux/module.h>
```

```
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
printk(KERN_ALERT "Hello, world\n");
return 0;
}

static void hello_exit(void)
{
printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

La función pasada a *module_init()* será la que se llama al cargar el módulo y la que se le pasa a *module_exit()*, la llamada al quitar el módulo. Como se puede observar, la forma de imprimir mensajes o información para *debugar* es mediante la función *printk()*. La cadena "KERN_ALTERT" indica la prioridad del mensaje a mostrar. Estos mensajes se generan en el fichero "*/var/log/messages"*.

Para compilar el módulo se utiliza un "*Makefile*" que debe tener una estructura como la siguiente:

```
obj-m := modulo.o

all:
	make -C /usr/src/linux/ M=`pwd` modules
```

En la primera línea, "modulo.o" debe llevar el mismo nombre que el del fichero fuente de extensión ".c".
En la última línea hay que indicar el directorio donde se encuentran las fuentes del kernel.

Una vez compilado con éxito, se debe haber generado un fichero con el mismo nombre que el fichero fuente y extensión ".ko". En el ejemplo, "modulo.ko".

Los comandos para cargar y descargar módulos son *insmod* y *rmmod* respectivamente. Estos comandos han de usarse con permisos de administrador (*root*). Para comprobar que el módulo ha sido cargado correctamente se puede utilizar el comando *lsmod* , que muestra un listado de los módulos cargados en ese instante.

3.4 Modulos específicos para Netfilter.

Para construir una NAT sencilla que no tenga seguimiento de conexión ha sido necesario programar dos módulos, uno que cambia la dirección IP origen del paquete generado dentro de la red privada por la dirección IP de la interfaz pública de la NAT y otro que deshace el cambio de los paquetes recibidos, cambiando en el campo dirección IP destino la dirección IP de la interfaz pública de la NAT por la dirección IP privada del servidor. Esto ha sido diseñado en forma de dos módulos por cuestiones de funcionamiento de los ganchos de Netfilter.

El primer módulo es el que cambiará la dirección IP origen del paquete generado dentro de la red privada por la dirección IP de la interfaz de salida de la NAT, es decir, el paquete será generado por el servidor con su dirección IP origen y en la salida de la NAT tendrá la dirección IP de su interfaz pública. Esto se realiza en el gancho **NF_INET_POST_ROUTING**, puesto que el cambio se realiza justo antes de salir de la NAT.
Como se estudió en las generalidades de Netfilter, a nivel de programación, el módulo consta de una función que se encarga de aceptar todos los paquetes, excepto aquellos que son TCP y cuya dirección IP origen sea la del servidor. A estos paquetes se les asigna la dirección IP de la interfaz pública de la NAT y acto seguido se recalcula el *checksum* de las cabeceras IP y TCP, para posteriormente seguir su camino. Todas las funciones que se utilicen en un módulo para el *kernel* deben estar definidas en la "*Linux Kernel API (LKA)*", en caso contrario deben de ser definidas en el propio módulo, como es el caso de las funciones que realizan el cálculo de los *checksum*. Otros elementos claves del módulo son:

```
static struct nf_hook_ops nfho; //Declaración de la estructura para
asignar la función al gancho

int init_module(){
  nfho.hook = hook_func;
  nfho.hooknum = NF_INET_POST_ROUTING;
  nfho.pf = PF_INET;
```

```
  nfho.priority = NF_IP_PRI_FIRST;

  nf_register_hook(&nfho);

  return 0;
}

void cleanup_module(){
  nf_unregister_hook(&nfho);
}
```

En la función *init_module()* es donde se especifica el nombre de la función a registrar "*hook_func*" y el gancho donde va a ser registrada, en este caso **NF_INET_POST_ROUTING**. Tras esto se realiza la llamada a *nf_register_hook()* que hace efectivo ese registro.
El módulo de salida/limpieza se encarga simplemente de eliminar ese registro de la función al gancho.

El otro módulo se encarga de deshacer la traducción de dirección IP en los paquetes que regresan, es decir, modifica la dirección IP destino de los paquetes con destino a la interfaz pública de la NAT por la dirección IP del servidor. Esto se realiza en el gancho **NF_INET_PRE_ROUTING**, puesto que el cambio ha de realizarse justo tras recibir el paquete, antes de tomar decisiones de encaminamiento.
La programación de este módulo es prácticamente idéntica al anterior, así que sólo se expondrán las diferencias:

- La función sólo difiere en que en este caso modificará la dirección IP de los paquetes TCP que tengan como dirección IP destino la interfaz pública de la NAT, cambiándola por la dirección IP del servidor.

- En la función de inicialización, el gancho que se asigna a la estructura es el **NF_INET_PRE_ROUTING**. El resto del proceso es idéntico.

Anexo I: código completo de toda la aplicación.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include <linux/if_ether.h>
#include <unistd.h>
#include <string.h>
#include <getopt.h>
#include <mysql/mysql.h>
#include <mysql/errmsg.h>
#include <mysql/mysqld_error.h>
#include <pthread.h>

#define BACKLOG 5
#define MAX_SERVERS 50

//Defino estructura
typedef struct {
  int socket;
  char *buff;
  char *ip;
} thr_data;

//Funciones definidas
int ExisteDB(MYSQL *myData, char *db);
int ExisteTabla(MYSQL *myData, char *db, char *tabla);
char *ip_local();
unsigned short tcp_sum_calc(unsigned short len_tcp, unsigned short
src_addr[],unsigned short dest_addr[], unsigned short buff[]);
unsigned short checksum(unsigned short *ptr, int len);
void *f_cliente(thr_data *arg);
void *f_servidor(thr_data *arg);


//Variables
```

```
int mister = 1;
int ind = 0;
char *ip_cli;
char *port_cli;

thr_data data;
pthread_mutex_t var_mutex = PTHREAD_MUTEX_INITIALIZER, var_mutex_db =
PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t var_cond[MAX_SERVERS];


//Variables para el manejo de la base de datos
MYSQL *conexion;
MYSQL_RES *R;
MYSQL_ROW COL;

char *servidor = "localhost";
char *usuario = "root";
char *clave = "root";
char *db = "NATEO";
char *tabla = "CONECTADOS";
char *plantilla = "INSERT INTO CONECTADOS (cos, tipo, nombre, publicable,
idserv, puerto) VALUES (\'%s\',\'%s\',\'%s\',\'%s\', \'%s\',\'%s\')";
char *plantilla2 = "id: %s | CoS: %s | Tipo: %s | Nombre: %s | Publicable: %s
| IDServ: %s | Puerto: %s\n";
char *plantilla3 = "SELECT * FROM CONECTADOS WHERE nombre = \'%s\'";
char *plantilla4 = "SELECT * FROM CONECTADOS WHERE tipo = \'%s\' AND
publicable = 'SP'";


int main (int argc, char **argv){

int c,i,wr_e, sock_cli, sock_serv, sock_raw, error_c;
int len_ip, len_port, len_sport, server = 0, client = 0, portint, sportint;
socklen_t longitud;
char *dir_ip=NULL, *datos=NULL, *buff=NULL, *aux=NULL, *aux2=NULL,
*text=NULL, packet[40];
char respuesta[2], *buff2=NULL,*ip, *port=NULL, *s_port=NULL, *port_cli=NULL,
*dat_cli;
```

```
struct iphdr *ip_raw = (struct iphdr *) packet;
struct sockaddr_in serv_aux, serv_pri, dir_cli, dir_raw;

pthread_t id_thread;


  while ((c = getopt (argc, argv, "c:s:d:p:a")) != -1)
       switch (c){
              case 'c':
                     dir_ip = optarg;

                     sock_cli = socket (AF_INET, SOCK_STREAM, 0);
                     bzero(&serv_aux, sizeof(serv_aux));
                     serv_aux.sin_family=AF_INET;
                     serv_aux.sin_port=htons(9999);
                     serv_aux.sin_addr.s_addr=inet_addr(dir_ip);

                     error_c=connect (sock_cli, (struct sockaddr *) &serv_aux,
sizeof(serv_aux));
                     if (error_c==-1) {
                       printf("ERROR AL CONECTAR\n");
                       exit(0);
                     }

                     client = 1;



                     break;

              case 's':

                     dir_ip = optarg;

                     //Conexion con el servidor auxiliar
                     sock_serv = socket (AF_INET, SOCK_STREAM, 0);

                     bzero(&serv_aux, sizeof(serv_aux));
                     serv_aux.sin_family=AF_INET;
                     serv_aux.sin_port=htons(9999);
                     serv_aux.sin_addr.s_addr=inet_addr(dir_ip);
```

```
                    error_c=connect (sock_serv, (struct sockaddr *) &serv_aux,
sizeof(serv_aux));
                    if (error_c==-1) {
                      printf("ERROR AL CONECTAR\n");
                      exit(0);
                    }

                    server = 1;



                    break;

              case 'd':
                    datos = optarg;
                    break;

              case 'p':
                    port_cli = optarg;
                    break;

              case 'a':

                    conexion = mysql_init(NULL);
                    //conexion con el servidor
                    mysql_real_connect(conexion,  servidor,  usuario,  clave,
NULL, MYSQL_PORT, NULL, 0);
                    mysql_query(conexion, "DROP DATABASE NATEO");
                    //comprobacion de existencia de DB, si no existe, crear
                    if(ExisteDB(conexion,db) == 1){
                      printf("Existe la base de datos, voy a usarla\n");
                      mysql_query(conexion, "use NATEO");
                    }
                    else{
                      printf("No existe la base de datos, voy a crearla y a
usarla\n");
                      mysql_query(conexion, "CREATE DATABASE NATEO");
                      mysql_query(conexion, "use NATEO");
                    }
```

```
                    //comprobacion de existencia de tabla, si no existe, crear
                    if(ExisteTabla(conexion,db,tabla) == 1){
                      printf("Existe la tabla\n");
                    }
                    else{
                      printf("No existe la tabla, voy a crearla\n");
                      mysql_query(conexion, "CREATE TABLE CONECTADOS (id INT
NOT NULL AUTO_INCREMENT, cos VARCHAR(10), tipo VARCHAR(20), nombre
VARCHAR(64), publicable VARCHAR(10), idserv INT, puerto INT, PRIMARY KEY
(id)) ENGINE = InnoDB");
                    }
                    //Fragmento de codigo necesario
                    mysql_query(conexion, "SELECT * FROM CONECTADOS");
                    R = mysql_use_result(conexion);
                    while ((COL = mysql_fetch_row(R)) != NULL);


                    //Creacion y config. de socket para recibir conexiones
(servidores o clientes)
                    sock_serv = socket (AF_INET, SOCK_STREAM, 0);

                    bzero(&serv_pri, sizeof(serv_pri));
                    serv_pri.sin_family=AF_INET;
                    serv_pri.sin_port=htons(9999);
                    serv_pri.sin_addr.s_addr=htonl(INADDR_ANY);

                    bind(sock_serv,    (struct    sockaddr    *)    &serv_pri,
sizeof(serv_pri));
                    listen(sock_serv, BACKLOG);

                    longitud=sizeof(dir_cli);

                    while(1){
                    sock_cli=accept(sock_serv, (struct sockaddr *) &dir_cli,
&longitud);
                    if (sock_cli==-1){
                      printf("ERROR EN EL ACCEPT\n");
                      exit(1);
                    }
                    printf("Acabo de aceptar una peticion\n");
```

```
                    buff = (char *)malloc(50);
                    wr_e = read(sock_cli, buff, 50);
                    printf("El valor de buff: %s \n", buff);

                    if (strncmp(buff, "C",1) == 0){
                      //se trata de un cliente
                      data.socket = sock_cli;
                      data.buff = (char *)malloc(50);
                      strcpy(data.buff, buff);
                      free(buff);
                      data.ip = inet_ntoa(dir_cli.sin_addr);
                      pthread_create(&id_thread,  NULL,  (void  *)&f_cliente,
(void *)&data);
                    }
                    else if (strncmp(buff, "S", 1) == 0){
                      //se trata de un servidor
                      pthread_mutex_lock(&var_mutex);
                      pthread_cond_init(&var_cond[ind], NULL);
                      pthread_mutex_unlock(&var_mutex);
                      data.socket = sock_cli;
                      data.buff = (char *)malloc(50);
                      strcpy (data.buff, buff);
                      free(buff);
                      data.ip =inet_ntoa(dir_cli.sin_addr);
                      pthread_create(&id_thread,  NULL,  (void  *)&f_servidor,
(void *)&data);
                    }

                    }
                    //cierre de conexion
                    mysql_close(conexion);
                    break;
          }

          if (datos != NULL){
          if (server == 1){ //Codigo para el servidor

            wr_e = write(sock_serv, datos, strlen(datos)+1);

            text = (char *)malloc(100);
```

```
wr_e = read(sock_serv, text, 100);
if (strncmp(text, "ERROR", 5) == 0){ //nombre de servidor ya existe
  printf("%s", text);
  free(text);
  close(sock_serv);
  exit(1);
}
printf("%s", text);
free(text);



while(1){

    dat_cli = (char *)malloc(30);
    wr_e = read(sock_serv, dat_cli, 30);
    wr_e = write(sock_serv, "OK\0", 3);

    printf("Estoy en el servidor, datos cliente: %s\n", dat_cli);
    len_ip = strcspn (dat_cli, ":");
    ip = (char *)malloc(len_ip+1);
    strncpy(ip, dat_cli, len_ip);
    printf("Contenido de IP: %s\n", ip);
    //extraccion del puerto
    aux = strchr(dat_cli,':');
    len_port = strcspn (aux+1, ":");
    port = (char *)malloc(len_port+1);
    strncpy(port, aux+1, len_port);
    printf("Contenido de port: %s\n", port);
    portint = strtol(port,NULL,10);

    aux2 = strchr(aux+1,':');
    len_sport = strcspn (aux2+1, "\0");
    s_port = (char *)malloc(len_sport+1);
    strncpy(s_port, aux2+1, len_sport+1);
    printf("Contenido s_port: %s\n", s_port);
    sportint = strtol(s_port, NULL, 10);

    free(s_port);
    free(port);
```

```
            free(dat_cli);


            if ((sock_raw = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
              printf("Error en la funcion socket() \n");
              exit(1);
            }

            dir_raw.sin_family = AF_INET;
            dir_raw.sin_port = 0; /* not needed in SOCK_RAW */
            dir_raw.sin_addr.s_addr=inet_addr(ip);
            memset(dir_raw.sin_zero, 0, sizeof(dir_raw.sin_zero));

            ip_raw->ihl = 5;
            ip_raw->version = 4;
            ip_raw->tos = 0;
            ip_raw->tot_len = htons(sizeof (struct iphdr) + sizeof (struct
tcphdr));    /* 16 byte value */
            ip_raw->frag_off = 0;           /* no fragment */
            ip_raw->ttl = 64;               /* default value */
            ip_raw->protocol = IPPROTO_TCP; /* protocol at L4 */
            ip_raw->check = 0;              /* not needed in iphdr */
            ip_raw->saddr = inet_addr(ip_local());
            ip_raw->daddr = dir_raw.sin_addr.s_addr;

            //Cabecera TCP

            struct tcphdr *tcph = (struct tcphdr *) (packet + ip_raw-
>ihl*4);

            tcph->source = htons(sportint);
            tcph->dest = htons(portint);
            tcph->seq = htons(1);
            tcph->ack_seq = 0x00000000;
            tcph->res1 = 0;
            tcph->doff = 5;
            tcph->window = htons(32767);
            tcph->check = 0;
            tcph->urg_ptr = 0;
```

```
            ip_raw->check   =   (unsigned   short)checksum((unsigned   short
*)ip_raw, ip_raw->ihl*4);

            tcph->check  =  (unsigned  short)  tcp_sum_calc((unsigned  short)
(20), (unsigned short *) &ip_raw->saddr, (unsigned short *) &ip_raw->daddr,
(unsigned short *) tcph);


            if (sendto(sock_raw, (char *)packet, sizeof(packet), 0, (struct
sockaddr *)&dir_raw, (socklen_t)sizeof(dir_raw)) < 0)
               perror("packet send error:");

            close (sock_raw);



            free(ip);

          }

      }
      else if (client == 1){ // Codigo para cliente

        if (port_cli != NULL){
        wr_e = write(sock_cli, datos, strlen(datos)+1);


        wr_e = read (sock_cli, respuesta, 2);

        if (strncmp(respuesta, "1", 1) == 0){
          //vamos a intentar conectar con el servidor y enviamos el puerto
          wr_e = write(sock_cli, port_cli, strlen(port_cli)+1);
          //recibimos respuesta
          text = (char *)malloc(100);
          wr_e = read (sock_cli,text, 100);
          printf("%s\n", text);
          free(text);

        }
        else if (strncmp(respuesta, "2", 1) == 0){
```

```
          buff2 = (char *) malloc(10000);
          wr_e = read(sock_cli,buff2, 10000);

          printf("%s\n", buff2);

          free(buff2);

        }
        close(sock_cli);
        exit(1);
        }
        else{
          printf("Llamada incorrecta al programa.\n");
          printf("La forma correcta es..\n");
          printf("Cliente: ./caller -c IP_SERV_AUX -d DATOS -p PUERTO\n");
        }
      }
      }
      else {
       printf("Llamada incorrecta al programa.\n");
       printf("La forma correcta es..\n");
       printf("Cliente: ./caller -c IP_SERV_AUX -d DATOS -p PUERTO \n");
       printf("Servidor: ./caller -s IP_SERV_AUX -d DATOS \n");
      }


  exit(0);

}

int ExisteDB(MYSQL *myData, char *db){
   // Conectar a base de datos.
   if(mysql_select_db(myData, db)) {
      if(ER_BAD_DB_ERROR == mysql_errno(myData)) return -1;
   }
   return 1;
}

int ExisteTabla(MYSQL *myData, char *db, char *tabla){
   char *consulta;
```

```
   char *plantilla = "SELECT * FROM %s.%s";
   int valorret = 1;

   consulta = (char *) malloc(strlen(db)+strlen(tabla)+strlen(plantilla)-1);
   sprintf(consulta, plantilla, db, tabla);
   if(mysql_query(myData, consulta)) {
      if(ER_NO_SUCH_TABLE == mysql_errno(myData)) valorret = -1;
   }
   free(consulta);
   return valorret;
}

void *f_cliente(thr_data *arg){
  printf("Soy un cliente\n");
  int socket = (int)arg->socket;
  char *buff = (char *)arg->buff;
  char *ip = (char *)malloc(20);
  strcpy(ip, (char *)arg->ip);
  printf("Cliente buffer: %s\n", buff);

  pthread_t id_thr;
  id_thr = pthread_self();
  pthread_detach(id_thr);


  int i, wr_e, len_cos, len_tipo, len_nom;
  char   *aux=NULL,   *aux2=NULL,   *r_port,   *consulta,   *mensaje,   *fila,
*m_id=NULL, *idserv=NULL, *s_port;


  //extraccion de cos
  len_cos = strcspn (buff, ":");
  printf("len_cos: %d\n", len_cos);
  char cos[len_cos+1];
  for (i = 0; i < len_cos+1; i++){
    cos[i] = 0;
  }
  strncpy(cos, buff, len_cos);
  printf("Contenido cos: %s\n", cos);
```

```
//extraccion de tipo
aux = strchr(buff,':');
if (aux != NULL){
  len_tipo = strcspn (aux+1, ":");
  char tipo[len_tipo+1];
  for (i = 0; i < len_tipo+1; i++){
    tipo[i] = 0;
  }
  strncpy(tipo, aux+1, len_tipo);
  printf("Contenido tipo: %s\n", tipo);

  //extraccion de nombre
  aux2 = strchr(aux+1,':');
  if (aux2 != NULL){
    wr_e = write (socket, "1\0",2);
    len_nom = strcspn (aux2+1, ":");
    char nombre[len_nom+1];
    for (i = 0; i < len_nom+1; i++){
     nombre[i] = 0;
    }
    strncpy(nombre, aux2+1, len_nom);
    printf("Contenido nombre: %s\n", nombre);

    r_port = (char *) malloc(10);
    wr_e = read (socket, r_port, 10);
    printf("Quiere conectar por el puerto: %s\n", r_port);

    consulta = (char *) malloc (strlen(plantilla3)+strlen(nombre)-1);
    sprintf(consulta, plantilla3, nombre);

    pthread_mutex_lock(&var_mutex_db);
    mysql_query(conexion, consulta);

    R = mysql_use_result(conexion);
    while ((COL = mysql_fetch_row(R)) != NULL){
     idserv = COL[5];
     s_port = COL[6];
    }
    pthread_mutex_unlock(&var_mutex_db);
    free(consulta);
```

```
      if (idserv == NULL){
       printf("Error,  el  nombre  introducido  no  existe  en  la  base  de
datos\n");
       m_id = "ERROR: el nombre introducido no existe en la base de datos\n";
       wr_e = write (socket, m_id, strlen(m_id)+1);
      }
      else{
       printf("Valor de la ID del servidor escogido: %s\n", idserv);
       m_id = "OK, ahora puede conectar con el servidor escogido\n";
       wr_e = write (socket, m_id, strlen(m_id)+1);
       //inserto IP y puerto en variables compartidas y protegidas
       int ids = atoi (idserv);
       pthread_mutex_lock(&var_mutex);
       ip_cli = (char *)malloc(20);
       strcpy(ip_cli, ip);
       port_cli = (char *)malloc(10);
       strcpy(port_cli, r_port);
       mister = 0;
       pthread_cond_signal(&var_cond[ids]);
       pthread_mutex_unlock(&var_mutex);
       free(ip);
       free(r_port);
       printf("He despertado al servidor dormido\n");

      }

    }
    else{
    //utilizar tipo y SP para obtener y enviar un listado al cliente
      wr_e = write (socket, "2\0",2);
      mensaje = (char *) malloc(10000);
      strcpy(mensaje, "\n");
      consulta = (char *) malloc (strlen(plantilla4)+strlen(tipo)-1);
      sprintf(consulta, plantilla4, tipo);
      pthread_mutex_lock(&var_mutex_db);
      mysql_query(conexion, consulta);

      R = mysql_use_result(conexion);
      while ((COL = mysql_fetch_row(R)) != NULL){
```

```
        fila                    =                    (char                    *)
malloc(strlen(plantilla2)+strlen(COL[0])+strlen(COL[1])+strlen(COL[2])+strlen
(COL[3])+strlen(COL[4])+strlen(COL[5])+strlen(COL[6])-1);
        sprintf(fila, plantilla2, COL[0], COL[1], COL[2], COL[3], COL[4],
COL[5], COL[6]);
        strcat (mensaje, fila);
        free (fila);
      }
      pthread_mutex_unlock(&var_mutex_db);
      free(consulta);
      wr_e = write (socket, mensaje, strlen(mensaje)+1);
      free(mensaje);
      printf("Enviada la lista completa\n");
    }
  }
  else{
    //envio listado de los servidores si publicables
     wr_e = write (socket, "2\0", 2);
     printf("Envio listado de servicios publicos\n");
     mensaje = (char *) malloc(10000);
     strcpy(mensaje, "\n");
     pthread_mutex_lock(&var_mutex_db);
     mysql_query(conexion, "SELECT * FROM CONECTADOS WHERE publicable =
'SP'");

     R = mysql_use_result(conexion);
     while ((COL = mysql_fetch_row(R)) != NULL){
        fila                    =                    (char                    *)
malloc(strlen(plantilla2)+strlen(COL[0])+strlen(COL[1])+strlen(COL[2])+strlen
(COL[3])+strlen(COL[4])+strlen(COL[5])+strlen(COL[6])-1);
        sprintf(fila, plantilla2, COL[0], COL[1], COL[2], COL[3], COL[4],
COL[5], COL[6]);
        strcat (mensaje, fila);
        free (fila);
     }
     pthread_mutex_unlock(&var_mutex_db);
     wr_e = write (socket, mensaje, strlen(mensaje)+1);
     free(mensaje);
     printf("Enviada la lista completa\n");
  }
```

```
  free(buff);
  close(socket);
  pthread_exit(0);
}

void *f_servidor(thr_data *arg){
  int i, myID, wr_e, len_cos, len_tipo, len_nom, len_pub, len_port;
  char *aux = NULL, *aux2 = NULL, *aux3=NULL, *aux4=NULL, *consulta, *idserv
= NULL, *msg=NULL, *puerto=NULL, numID[2], *dat_serv;

  pthread_mutex_lock(&var_mutex);
  myID = ind;
  ind++;
  pthread_mutex_unlock(&var_mutex);
  printf("Soy un servidor\n");
  int socket = (int)arg->socket;
  char *buff = (char *)arg->buff;
  char *ip = (char *)arg->ip;
  printf("Servidor buffer: %s\n", buff);

  printf("Mi IP es: %s\n", ip);

  pthread_t id_thr;
  id_thr = pthread_self();
  pthread_detach(id_thr);

  //extraccion de cos
  len_cos = strcspn (buff, ":");

  char cos[len_cos+1];
  for (i = 0; i < len_cos+1; i++){
  cos[i] = 0;
  }
  strncpy(cos, buff, len_cos);
  printf("Contenido cos: %s\n", cos);

  //extraccion de tipo
  aux = strchr(buff,':');
  if (aux == NULL){
```

```
  printf("Llamada incorrecta del programa servidor\n");
  pthread_mutex_lock(&var_mutex);
  ind--;
  pthread_mutex_unlock(&var_mutex);
  free(buff);
  close(socket);
  pthread_exit(0);
}
len_tipo = strcspn (aux+1, ":");
char tipo[len_tipo+1];
for (i = 0; i < len_tipo+1; i++){
  tipo[i] = 0;
}
strncpy(tipo, aux+1, len_tipo);
printf("Contenido tipo: %s\n", tipo);

//extraccion de nombre
aux2 = strchr(aux+1,':');
if (aux2 == NULL){
  printf("Llamada incorrecta del programa servidor\n");
  pthread_mutex_lock(&var_mutex);
  ind--;
  pthread_mutex_unlock(&var_mutex);
  free(buff);
  close(socket);
  pthread_exit(0);
}
len_nom = strcspn (aux2+1, ":");
char nombre[len_nom+1];
for (i = 0; i < len_nom+1; i++){
  nombre[i] = 0;
}
strncpy(nombre, aux2+1, len_nom);
printf("Contenido nombre: %s\n", nombre);

consulta = (char *) malloc (strlen(plantilla3)+strlen(nombre)-1);
sprintf(consulta, plantilla3, nombre);
pthread_mutex_lock(&var_mutex_db);
mysql_query(conexion, consulta);
```

```
R = mysql_use_result(conexion);
while ((COL = mysql_fetch_row(R)) != NULL){
  idserv = COL[5];
}
pthread_mutex_unlock(&var_mutex_db);
free(consulta);

if (idserv != NULL){
  msg = "ERROR: el nombre del servidor ya existe\n";
  wr_e = write (socket, msg, strlen(msg)+1);
  printf("Mensaje: %s", msg);
  pthread_mutex_lock(&var_mutex);
  ind--;
  pthread_mutex_unlock(&var_mutex);
  free(buff);
  close(socket);
  pthread_exit(0);
}

//extraccion de publicable
aux3 = strchr(aux2+1,':');
if (aux3 == NULL){
  printf("Llamada incorrecta del programa servidor\n");
  pthread_mutex_lock(&var_mutex);
  ind--;
  pthread_mutex_unlock(&var_mutex);
  free(buff);
  close(socket);
  pthread_exit(0);
}
len_pub = strcspn (aux3+1, ":");
char publicable[len_pub+1];
for (i = 0; i < len_pub+1; i++){
  publicable[i] = 0;
}
strncpy(publicable, aux3+1, len_pub);
printf("Contenido publicable: %s\n", publicable);

//extraccion del puerto
aux4 = strchr (aux3+1, ':');
```

```
  if (aux4 != NULL){
  len_port = strcspn (aux4+1, "\0");
  puerto = (char *)malloc(len_port+1);
  strncpy(puerto, aux4+1, len_port+1);
  }
  else {
    puerto = (char *)malloc(3);
    strcpy(puerto, "80");
  }

  printf("Contenido puerto: %s\n", puerto);

  msg = "Datos introducidos correctamente\n";
  wr_e = write (socket, msg, strlen(msg)+1);
  //insertar datos
  sprintf(numID, "%d", myID);
  printf("Procedo a insertar datos\n");
  consulta                    =                    (char                    *)
malloc(strlen(plantilla)+strlen(cos)+strlen(tipo)+strlen(nombre)+strlen(publi
cable)+strlen(numID)+strlen(puerto)-1);
  sprintf(consulta, plantilla, cos, tipo, nombre, publicable, numID, puerto);

  pthread_mutex_lock(&var_mutex_db);
  mysql_query(conexion, consulta);
  pthread_mutex_unlock(&var_mutex_db);
  free(consulta);

  free(buff);
  //llamada a var. condicion para esperar clientes
  while(1){
    printf("Estoy en el while infinito\n");
    pthread_mutex_lock(&var_mutex);
  while (mister == 1)
    pthread_cond_wait(&var_cond[myID], &var_mutex);
  printf("Un cliente me ha despertado. Mi ID es: %d\n", myID);
  printf("IP del cliente: %s\n", ip_cli);
  printf("Puerto del cliente: %s\n", port_cli);
  dat_serv = (char *)malloc(30);
  strcpy(dat_serv, ip_cli);
  strcat (dat_serv, ":");
```

```
  strcat (dat_serv, port_cli);
  strcat (dat_serv, ":");
  strcat (dat_serv, puerto);
  printf("Los datos que voy a enviar: %s\n", dat_serv);
  free(ip_cli);
  free(port_cli);
  wr_e = write(socket, dat_serv, strlen(dat_serv)+1);
  char *resp = (char *)malloc(3);
  wr_e = read (socket, resp, 3);
  if (strncmp (resp, "OK", 2) != 0){
    printf("El servidor no se encuentra accesible\n");
    free(dat_serv);
    free(puerto);
    free(resp);
    close(socket);
    pthread_exit(0);
  }
  mister = 1;
  pthread_mutex_unlock(&var_mutex);
  free(resp);
  free(dat_serv);

  }
  free(puerto);
  close(socket);
  pthread_exit(0);
}

unsigned short tcp_sum_calc(unsigned short len_tcp, unsigned short
src_addr[],unsigned short dest_addr[], unsigned short buff[])
{
    unsigned char prot_tcp=6;
    unsigned long sum;
    int nleft;
    unsigned short *w;

    sum = 0;
    nleft = len_tcp;
    w=buff;
```

```
    /* calculate the checksum for the tcp header and payload */
    while(nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }

    /* if nleft is 1 there ist still on byte left. We add a padding byte
(0xFF) to build a 16bit word */
    if(nleft>0)
    {
      /* sum += *w&0xFF; */
             sum += *w&ntohs(0xFF00);   /* Thanks to Dalton */
    }

    /* add the pseudo header */
    sum += src_addr[0];
    sum += src_addr[1];
    sum += dest_addr[0];
    sum += dest_addr[1];
    sum += htons(len_tcp);
    sum += htons(prot_tcp);

    // keep only the last 16 bits of the 32 bit calculated sum and add the
carries
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);

    // Take the one's complement of sum
    sum = ~sum;

return ((unsigned short) sum);
}


unsigned short checksum(unsigned short *ptr, int len)
{
    int sum = 0;
    unsigned short answer = 0;
    unsigned short *w = ptr;
```

```
    int nleft = len;

    while(nleft > 1){
        sum += *w++;
        nleft -= 2;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return(answer);
}

char *ip_local() {
    struct sockaddr_in host;
    char nombre[255], *ip;

    gethostname(nombre, 255);
    host.sin_addr = * (struct in_addr*) gethostbyname(nombre)->h_addr;
    ip = inet_ntoa(host.sin_addr);

    return ip;
}
```

Anexo II: código completo de los módulos.

- Módulo registrado en POST_ROUTING

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/types.h>
#include <linux/in.h>
```

```
/* For IP first_socketer */
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>


unsigned short tcp_sum_calc(unsigned short len_tcp, unsigned short src_addr[],unsigned short dest_addr[], unsigned short buff[])
{
    unsigned char prot_tcp=6;
    unsigned long sum;
    int nleft;
    unsigned short *w;

    sum = 0;
    nleft = len_tcp;
    w=buff;

    /* calculate the checksum for the tcp header and payload */
    while(nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }

    /* if nleft is 1 there ist still on byte left. We add a padding byte (0xFF) to build a 16bit word */
    if(nleft>0)
    {
      /* sum += *w&0xFF; */
            sum += *w&ntohs(0xFF00);   /* Thanks to Dalton */
```

```
    }

    /* add the pseudo header */
    sum += src_addr[0];
    sum += src_addr[1];
    sum += dest_addr[0];
    sum += dest_addr[1];
    sum += htons(len_tcp);
    sum += htons(prot_tcp);

    // keep only the last 16 bits of the 32 bit calculated sum and add the
carries
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);

    // Take the one's complement of sum
    sum = ~sum;

return ((unsigned short) sum);
}


unsigned short checksum(unsigned short *ptr, int len)
{
    int sum = 0;
    unsigned short answer = 0;
    unsigned short *w = ptr;
    int nleft = len;

    while(nleft > 1){
```

```
        sum += *w++;
        nleft -= 2;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return(answer);
}

unsigned int inet_addr(char *str) {
  int a,b,c,d;
  char arr[4];
  sscanf(str,"%d.%d.%d.%d",&a,&b,&c,&d);
  arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d;
  return *(unsigned int*)arr;
}

static struct nf_hook_ops nfho;

unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb, const
struct net_device *in, const struct net_device *out, int (*okfn)(struct
sk_buff *)){

  struct sk_buff *sb = skb;
  struct iphdr *iph;
  struct tcphdr *tcph;

iph = (struct iphdr *)skb_network_header(skb);
tcph = (struct tcphdr *)(skb_transport_header(skb) + 20);
```

```
printk("Received packet from source address: %d.%d.%d.%d!\n",NIPQUAD(iph->saddr));

//Comprobaciones iniciales para descartar punteros nulos o paquetes distintos a TCP
if (!sb) return NF_ACCEPT;
if (!skb_network_header(skb)) return NF_ACCEPT;
if (iph->protocol != IPPROTO_TCP) {
  printk("NO soy paquete TCP\n");

  return NF_ACCEPT;
}


if (iph->saddr == inet_addr("192.168.10.1")){
  //funciona con POST_ROUTING
  iph->saddr = inet_addr("192.168.1.254");
  printk("IP Cambiada: %d.%d.%d.%d!\n",NIPQUAD(iph->saddr));
  iph->check = 0;
  iph->check = (unsigned short)checksum((unsigned short *)iph , iph->ihl*4);

  tcph->check = 0;
  tcph->check = (unsigned short)tcp_sum_calc((unsigned short)(skb->len - 20), (unsigned short *) &iph->saddr, (unsigned short *) &iph->daddr, (unsigned short *) tcph);
  printk("Longitud de los datos: %d\n", skb->len);

  return NF_ACCEPT;
}
```

```
 // printk("SI soy paquete TCP\n");
  return NF_ACCEPT;
}

int init_module(){
  nfho.hook = hook_func;
  nfho.hooknum = NF_INET_POST_ROUTING;
  nfho.pf = PF_INET;
  nfho.priority = NF_IP_PRI_FIRST;

  nf_register_hook(&nfho);

  return 0;
}

void cleanup_module(){
  nf_unregister_hook(&nfho);
}
```

- Módulo registrado en PRE_ROUTING

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/types.h>
#include <linux/in.h>
/* For IP first_socketer */
#include <linux/netfilter.h>
```

```
#include <linux/netfilter_ipv4.h>

unsigned short tcp_sum_calc(unsigned short len_tcp, unsigned short src_addr[],unsigned short dest_addr[], unsigned short buff[])
{
    unsigned char prot_tcp=6;
    unsigned long sum;
    int nleft;
    unsigned short *w;

    sum = 0;
    nleft = len_tcp;
    w=buff;

    /* calculate the checksum for the tcp header and payload */
    while(nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }

    /* if nleft is 1 there ist still on byte left. We add a padding byte (0xFF) to build a 16bit word */
    if(nleft>0)
    {
      /* sum += *w&0xFF; */
            sum += *w&ntohs(0xFF00);   /* Thanks to Dalton */
    }

    /* add the pseudo header */
```

```
    sum += src_addr[0];
    sum += src_addr[1];
    sum += dest_addr[0];
    sum += dest_addr[1];
    sum += htons(len_tcp);
    sum += htons(prot_tcp);

    // keep only the last 16 bits of the 32 bit calculated sum and add the
carries
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);

    // Take the one's complement of sum
    sum = ~sum;

return ((unsigned short) sum);
}


unsigned short checksum(unsigned short *ptr, int len)
{
    int sum = 0;
    unsigned short answer = 0;
    unsigned short *w = ptr;
    int nleft = len;

    while(nleft > 1){
        sum += *w++;
        nleft -= 2;
    }
```

```
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return(answer);
}



unsigned int inet_addr(char *str) {
  int a,b,c,d;
  char arr[4];
  sscanf(str,"%d.%d.%d.%d",&a,&b,&c,&d);
  arr[0] = a; arr[1] = b; arr[2] = c; arr[3] = d;
  return *(unsigned int*)arr;
}



static struct nf_hook_ops nfho;

unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in, const struct net_device *out, int (*okfn)(struct sk_buff *)){

  struct sk_buff *sb = skb;
  struct iphdr *iph;
  struct tcphdr *tcph;

iph = (struct iphdr *)skb_network_header(skb);
```

```
tcph = (struct tcphdr *)(skb_transport_header(skb) + 20);

printk("Received packet from source address: %d.%d.%d.%d!\n",NIPQUAD(iph->saddr));

//Comprobaciones iniciales para descartar punteros nulos o paquetes distintos a TCP
if (!sb) return NF_ACCEPT;
if (!skb_network_header(skb)) return NF_ACCEPT;
if (iph->protocol != IPPROTO_TCP) {
  printk("NO soy paquete TCP\n");

  return NF_ACCEPT;
}

if (iph->daddr == inet_addr("192.168.1.254")){
  //funciona con PRE_ROUTING
  iph->daddr = inet_addr("192.168.10.1");
  iph->check = 0;
  iph->check = (unsigned short)checksum((unsigned short *)iph , iph->ihl*4);

  tcph->check = 0;
  tcph->check = (unsigned short)tcp_sum_calc((unsigned short)(skb->len - 20), (unsigned short *) &iph->saddr, (unsigned short *) &iph->daddr, (unsigned short *) tcph);
  printk("Longitud de los datos: %d\n", skb->len);

  return NF_ACCEPT;
}
```

```
 // printk("SI soy paquete TCP\n");
  return NF_ACCEPT;
}

int init_module(){
  nfho.hook = hook_func;
  nfho.hooknum = NF_INET_PRE_ROUTING;
  nfho.pf = PF_INET;
  nfho.priority = NF_IP_PRI_FIRST;

  nf_register_hook(&nfho);

  return 0;
}

void cleanup_module(){
  nf_unregister_hook(&nfho);
}
```

Bibliografía

- Benvenuti, Cristian. *Understanding Linux network internals*. Sebastopol, Calif. ; Farnham: O'Reilly, 2005, c2006. ISBN 0596002556
- Rubini, Alexandro. *Linux device drivers*. Oram, Andy; Siever, Ellen (ed. lit.). Cambridge: O'Reilly, c1998. ISBN 1565922921
- Love, Robert. *Linux kernel development*. Indianapolis, Ind.: Novell Press, c2005. ISBN 0672327201
- Nichols, Bradford; Buttlar, Dick; Farrell, Jackie. *Pthreads programming*, Cambridge: O'Reilly, 1996. ISBN 1565921151