Algoritmo de encaminamiento para los RCX de los Legos Mindstorm

Universidad Rey Juan Carlos
5º Ingeniería Informática

Diego Chaparro González
José Pelegrín López
Raúl Rodríguez Aparicio
{dchaparr, jpelegri, rrodrigu}@gsyc.escet.urjc.es

27 de junio de 2003

Índice general

Índice de figuras

Capítulo 1

Introducción

Los dispositivos electrónicos móviles dotados de microprocesadores se están poniéndo cada vez más de moda hoy en día. Las *PDAs* o agendas electrónicas se están convirtiendo en un elemento tan habitual como los teléfonos móviles, y, por otro lado, los juguetes controlados por pequeños procesadores capaces de realizar tareas sencillas (y a veces no tan sencillas), están empezando a cambiar la industría dedicada a la comercialización de este tipo de entretenimientos.

Otro aspecto de actualidad relacionado con los temas anteriores es la posibilidad de montar *redes sobre la marcha* usando dispositivos móviles con capacidades de comunicación. Ya existen soluciones que permiten de forma sencilla y rápida establecer pequeñas *redes ad-hoc* de ordenadores portátiles que permitan la comunicación entre todos los dispositivos que conforman la red.

El objetivo que se pretende en esta práctica es combinar estos elementos (dispositivos móviles como una *PDA* y robots de *Lego*) de manera que una serie de robots formen una pequeña *red ad-hoc* y desde una *PDA* se les pueda perdir información relativa acerca de su estado (por ejemplo, nivel de luz de un robot determinado). Un PC hará de intermediario entre la agenda electrónica y la red de robots.

Capítulo 2

Encaminamiento *Ad-Hoc*

2.1. Protocolos de encaminamiento en redes *ad hoc*

La siguiente información ha sido extraida del documento *A Review of Current Routing Protocols for Ad hoc Mobile Wireless Networks*[1] y del documento *Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks*[2].

Los protocolos de encaminamiento *Ad hoc* se suelen clasificar en dos clases:

- Dirigidos por tablas de encaminamiento, como por ejemplo *AODV*, *DSDV* y *WRP*.
- Por demanda iniciada en origen, como los protocolos *DSR*, *LMR*, *TORA* ó *ABR*.

Los protocolos de encaminamiento dirigidos por tablas de encaminamiento tratan de mantener la información de encaminamiento en estado consistente y actualizada de cada nodo de la red. Este tipo de protocolos requiere que cada nodo mantenga una o más tablas en donde tengan almacenada la información de encaminamiento, y deben responder a los cambios en la topología de la red propagando actualizaciones a través de toda la red para mantener consistente la visión de la misma. Los aspectos en los que se diferencian unos protocolos de otros en este tipo de protocolos es en el número de tablas de encaminamiento necesarias y los métodos utilizados para anunciar un cambio en la red.

Tres ejemplos de protocolos dirigidos por tablas de encamiento son *Ad Hoc On-Demand Distance Vector Routing* (*AODV*), *The Destination-Sequenced Distance -Vector Routing Protocol* (*DSDV*) y *The Wireless Routing Protocol* (*WRP*).

Por otro lado, los protocolos por demanda iniciada en origen, crean rutas únicamente cuando un nodo desea enviar un paquete. Así, cuando un nodo emisor necesita saber la ruta hacia un nodo destino, inicia un proceso de descubrimiento de ruta (*Route Discovery*) dentro de la red para averiguar la ruta. Este proceso finaliza al encontrar la ruta hacia el nodo deseado o bien cuando

[1] http://www-users.cs.umn.edu/ hngo/WirelessNetworking/MANET/Adhoc_Review.pdf
[2] http://www.iprg.nokia.com/ charliep/txt/infocom00/aodv/main.ps

todas las posibles rutas hacia él han sido examinadas. Una vez que se ha establecido una ruta, es mantenida en una cache hasta que o bien el nodo destino pasa a ser inalcanzable o bien la ruta en cache expira al no haber sido utilizada recientemente.

Ejemplos de este tipo de protocolos son *Dynamic Source Routing* (*DSR*), *Temporary Ordered Routing Algorithm* (*TORA*) y *Associative-Based Routing* (*ABR*).

2.2. Comparativa entre *DSR* y *AODV*

Ambos protocolos, *DSR* y *AODV*, tienen en común que son protocolos de naturaleza reactiva (*reactive*) - ambos inician las actividades de encaminamiento bajo demanda. En cuanto a sus diferencias más significativas, *DSR* usa encaminamiento en origen, mientras que *AODV* utiliza encaminamiento dirigido por tablas y números de secuencia en los nodos destino.

2.2.1. *Dynamic Source Routing - DSR*

La característica clave en *DSR* es el uso de *encaminamiento en origen*, esto es, el emisor conoce la ruta completa salto-a-salto (*hop-by-hop*) hacia el nodo destino. Estas rutas se almacenan en la *Route Cache* y los paquetes de datos llevan la ruta completa hasta el destino en una cabecera del propio paquete.

Cuando un nodo en la red *Ad hoc* intenta enviar un paquete de datos a un nodo destino para el cual no conoce todavía su ruta, utiliza el procedimiento de *Route Discovery* para determinar dinámicamente cual es la ruta hasta dicho nodo.

El mecanismo de *Route Discovery* funciona inundando la red con paquetes de petición de ruta (*Route Request packets - RREQ*), que son reenviados de unos nodos a otros hasta que alcance el nodo destino de la petición o se halle una ruta hacia dicho nodo en alguna de las *Route Caches* de los nodos intermedios a los que va llegando la petición.

El nodo destino (o el que tiene en su *Route Cache* la entrada para ese nodo destino) entonces envía una respuesta al nodo que realizó la petición (*Route Reply - RREP*) con la ruta solicitada.

Si cualquier enlace de la ruta en origen se cae, se le notifica al nodo origen usando el paquete de error (*Route Error - RERR*). En el nodo origen, se eliminará dicho enlace de su *Route Cache*. Si aún es necesaria esa ruta, se tendrá que iniciar otra vez el procedimiento de descubrimiento de ruta.

DSR es muy agresivo en cuanto al uso del encaminamiento en origen y de *Route Caches*. No son necesarios mecanismos especiales para la detección de bucles. Además cualquier nodo intermedio en la ruta desde el nodo origen hacia el destino actualiza la información de su *Route Cache* para posibles usos en el futuro.

2.2.2. *Ad Hoc On-Demand Distance Vector Routing - AODV*

AODV adpota un mecanismo muy diferente para mantener la información de encaminamiento. Utiliza las tablas de encaminamiento tradicionales, con una entrada por cada destino. En *DSR*, en cambio, se pueden tener múltiples entradas en la *Route Cache* por cada nodo destino.

Sin encaminamiento en origen, *AODV* utiliza las entradas de la tabla de encaminamiento para propagar las *Route Reply* hacia el origen y, los paquetes de datos para enviarlos a los destinos.

AODV utiliza números de secuencia mantenidos en cada nodo destino para determinar el grado de validez de la información de encaminamiento y para prevenir la existencia de bucles al encaminar. Estos números de secuencia viajan en todos los paquetes de encaminamiento.

Una característica importante de *AODV* es el mantenimiento de estados basados en timers (*timer-based states*) en cuanto a las entradas de la tabla de encaminamiento se refiere. Una entrada de la tabla de encaminamiento expira si no ha sido utilizada recientemente.

También se mantiene un conjunto de nodos predecesores por cada entrada en la tabla de encaminamiento que denotan el conjunto de nodos vecinos que usan la entrada de la tabla para encaminar los paquetes de datos. A estos nodos se le comunica mediante paquetes *Route Error* cuando el enlace del próximo salto está caído. Cada nodo predecesor, reenvía el *Route Error* a su propio conjunto de nodos predecesores, eliminando así todas las rutas existentes que utilizan el enlace caído.

2.2.3. Crítica de *DSR* y *AODV*

Ambos protocolos descubren rutas solo ante la presencia de paquetes de datos que necesian ser encaminados. El descubrimiento de rutas en ambos se basa en ciclos de consulta y respuestas, y en información de encaminamiento almacenada en forma de tablas de encaminamiento (*AODV*) o de *Route Caches* (*DSR*).

DSR tiene acceso a una mayor cantidad de información de encaminamiento que *AODV*. Por ejemplo, en *DSR*, utilizando un ciclo simple de *consulta-respuesta*, el origen puede aprender las rutas hacia los nodos intermedios que conforman la ruta hacia el nodo destino. Además, cada nodo intermedio puede aprender las rutas hacia cada nodo de la propia ruta.

Por otro lado, al contestar *DSR* a todas las peticiones de búsqueda de nodos destino, permite al origen aprender rutas alternativas hacia el destino, que serán muy útiles en caso de que la ruta primaria (la más corta) falle. En *AODV*, en cambio, sólo se contesta a la primera petición. El resto son ignoradas.

El mecanismo de eliminación de rutas usando paquetes *Route Error* es más conservativo en *AODV*. A través de la lista de nodos predecesores, los paquetes de error alcanzan a todos los nodos usando el enlace que ha fallado o como parte de la ruta hacia un nodo destino. En *DSR*, en cambio, los nodos que se hallan en del enlace caído hacia el nodo destino, no son informados rápidamente del error. Los nodos entre el origen y el enlace que ha fallado sí se enteran de inmediato.

Capítulo 3

Lego Mindstorms

Los robots *Legos Mindstorms* siguen la metodología de la firma creadora de estos robots, *Lego*. Los típicos juegos de construcción de *Lego* se han extendido hasta el mundo de los robots en los que a partir de un pequeño procesador (un *Hitachi H8300* incrustado en el *RCX* o "ladrillo"), unos cuantos motores y una serie de sensores (de contacto, de luz o de rotación, entre otros), se pueden diseñar y construir robots de diversos tipos y funcionalidades.

El sistema operativo que traen estos robots *Lego Mindstorms* está considerablemente limitado a la hora de progamar sobre ellos aplicaciones con cierto grado de complejidad. Por ello, se ha desarrollado un sistema operativo, denominado *LegOS*, independiente del sistema operativo que traen los *Lego Mindstorms*, que entre sus caracterísitcas principales destacan:

- Capacidades para ejecutar varias tareas
- Mecanismos de ahorro de energía
- Gestión dinámica de memoria
- Uso de semáforos
- Acceso a los cada uno de los elementos de los robots:
  - *Display*
  - Botones del *RCX*
  - Motores
  - Sensores
  - ***Comunicación por Infrarrojos***

Precisamente el último punto, la *comunicación por infrarrojos*, va a permitir a los robots ejecutar un protocolo de *encaminamiento ad-hoc* simplificado de forma que desde cualquier robot se puedan establecer comunicaciones con cualquier otro robot dentro de la *red ad-hoc*.

3.1. Esquema de direccionamiento en *LegOS*

El siguiente artículo[1], enviado por Martin Cornelius[2], explica todo lo relevante acerca de direccionamiento en *LegOS*:

- **Addressing**:
  - *How does the addressing scheme work?*
    It's quite simple: Currently, there are two kinds of LNP-Packets: the first one is called the integrity packet

```
+--+---+-----------------------+---+
|F0|LEN|        IDATA          |CHK|
+--+---+-----------------------+---+

F0    : identifies an integrity packet
LEN   : length of IDATA section, 0 to 255
IDATA : payload data
CHK   : checksum
```

    because LEN is one byte, DATA can be at most 255 byte, so the overall maximum length of an LNP Packet ist 258 bytes. The Integrity packet does not carry any addressing information, you could consider it as the LNP broadcast mechanism.
    The second kind of packet is the addressing packet. Actually, it is an integrity packet that encapsulates payload data and addressing information in the IDATA section:

```
+--+---+----+---+--------------+---+
|F1|LEN|DEST|SRC|   ADATA      |CHK|
+--+---+----+---+--------------+---+
       |                       |
       |<-      IDATA        ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data (1 to 253 bytes)
```

    Generally, LNP-addressing packets give you functionality quite similar to UDP: they are not guaranteed to arrive, but if they arrive, they will contain no errors. An addressing packet is always send to a

[1] http://news.lugnet.com/robotics/rcx/legos/?n=788
[2] cornelius@csd.de

specific port on a specific host. If the host has started a server on this port (an addressing-handler in LNP-idiom), and the packet arrives without errors, the packet is passed to the handler, otherwise (no handler or errors), the packet is silently discarded.

DEST and SRC carry the destination address respectively the originating address of the packet. Both are 1 byte, which is split into a host and a port section. The macro CONF_LNP_HOSTMASK (config.h) determines, which individual bits of this byte are carrying host address information, and which carry port information. This gives you the option to configure the addressing scheme according to your needs. E.g, if you had only 2 hosts (a PC and a RCX), but wanted to have 128 distinguishable ports on each host, you could set CONF_LNP_HOSTMASK to 0x80. Actually, it is not neccessary to use bit-schemes like 10000000, 11000000, 11100000, ... for CONF_LNP_HOSTMASK, but it is wise to do so!, because otherwise precious RAM will be wasted inside LNP.

Anyways, for most stituations the default CONF_LNP_HOSTMASK of 0xF0 will be a good choice. With this setup, the upper 4 bit of DEST and SRC are the host address, and the lower 4 bit denote the port. Thus, this allows you to assign 16 different hostaddresses, and have 16 different ports on each host.

- *Can a RCX address more than one host (via its Tower), and more than one program on a particular host?*

  As said above, with default CONF_LNP_HOSTMASK you can address up to 15 remote Hosts, and 16 ports on each of this hosts. Several ports can be opened in one application or in several applications, it's just a matter of taste.

- *How does the RCX side set the source & destination addresses?*

  The source-hostaddress for the RCX is set when compiling legOS by macro CONF_LNP_HOSTADDR (boot/config.h). By default, it is 0x00. If you want another hostaddress for your RCX, you have to change this macro and recompile legOS. Remember, the upper four bits form the hostaddress! Thus, to yield hostaddress 1 you would have to specify 0x10, NOT 0x01. Thus, to have several RCX's have different hostaddresses, you must compile abd doenload legOS several times, changing the CONF_LNP_HOSTADDR in between.

  The destination address (HOST/PORT) is specified as parameter to lnp-addressing_write(). You specify the destination for every packet sent in the call of this function.

- *Can a Host address more than one RCX?*

  Same as above. The semantics of lnp_addressing_write(), lnp_addressing_set_handler(), etc. is exactly the same on PC and RCX.

- *Can the host set its source address? If so, how?*

  Before you continue, i'd encourage you to open file liblnp.h from the lnpd tarball and look at the prototype for lnp_init(). As described in the header file, the hostaddress and mask may be set in the call to lnp_init(). If you use 0, the defaults, means 0x80 and 0xF0 are used.

With liblnp, you can set an arbitrary hostaddress and hostmask inside each application you run. If you use the same hostaddress in 2 or more applications running concurrently, and have setup a handler for the same port in this applications, every incoming packet for this host/port will be delivered to each application, means, the packet is multiplied (lnpd realises this).

IMHO, normally you don't want this. I suggest to use 1 hostaddress for every PC involved in communications (there may be several, lnpd is fully networked), and use different port numbers in different applications running on the same PC. However, it's perfectly legal to use different hostaddresses in different applications running on the same PC. In contrast, it's NOT OK to have different HOSTMASKs. Therefore, the default hostmask used by liblnp is 0xF0, the same as the default for the RCX. If you want to use another HOSTMASK, you have to do this on all RCX's (recompile legOS) and all PC-applications that are involved (set in lnp_init).

- **Transmission**:
  - *Is there a way to send reliable, sequenced data, like TCP does?*

    Currently not, but i'm nearly finished with an implementation of this (it quickly gets more complicated than one might think at the first glance). The RCX-side is already done, but i have only one RCX, so i can't test it. Because i'm quite busy at the moment, it may take about 4 weeks before the PC side counterpart will be available and tested.
- **Host Programming**:
  - *Is there a way to write simple programs on the host side that don't need to go through the lnpd demon?*

    Actually, the development goal of lnpd was to make application programming simple ;-).

    If you want to write applications that don't rely on lnpd, you have to do all the tower related stuff (configure tty, wakeup, send keepalive), and packet processing inside your application - an example of how to do this is the 'original' dll that comes with legOS. Of course, one could wrap all this into a library. Once i find the time, i'll try to pick up the various fragments of code from my trash-folder and build that beast, but that might take some time...

Más información se puede encontrar en el *HowTo*[3] de *LegOS*.

[3] http://legOS.sourceforge.net/HOWTO/HOWTO.ps

Capítulo 4

Propuesta de práctica

4.1. Propuesta de protocolo de encaminamiento

El objetivo principal es implementar una versión simplificada del protocolo *DSR* (*The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks*[1]) utilizando como terminales los *Legos Mindstorm*.

El protocolo *DSR* se compone de dos mecanismos que funcionan conjuntamente para permitir el descubrimiento y mantenimiento de las rutas en origen en las redes *ad hoc*:

- *Route Discovery*. Este mecanismo permite a un nodo S(ource) que quiere enviar un paquete a un nodo D(estiny) obtener la ruta en origen para enviar el paquete. Únicamente se usa cuando S intenta enviar un paquete a D y no conoce la ruta hasta él.

- *Route Maintenance*. Mediante este mecanismo, un nodo S puede detectar, mientras utiliza una ruta en origen para D, si la topología de la red ha cambiado de forma que la ruta que utilizaba para dirigir paquetes a D ya no es válida debido a que un enlace de la ruta ha dejado de funcionar. Cuando *Route Maintenance* indica que una ruta en origen ha dejado de existir, S puede intentar utilizar otro ruta hacia D o bien usar *Route Discovery* de nuevo para encontrar una nueva ruta para D. *Route Maintenance* solo se usa cuando S está enviando paquetes a D.

[1] draft-ietf-manet-dsr-05.txt

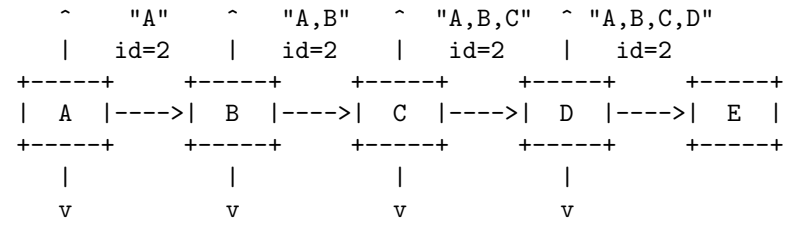
4.1.1. Descripción del protocolo

Route Discovery

Al enviar un nodo origen un nuevo paquete a un nodo destino, el nodo origen debe indicar en la cabecera del paquete la ruta en origen que debe seguir el paquete para que llegue al nodo destino, indicando la secuencia de saltos que debe dar el paquete en su camino hacia el destino.

Si el nodo emisor consulta su *Route Cache* y encuentra una entrada con el nodo receptor, obtendrá la ruta en origen que debe seguir el paquete hasta el destino. Si por el contrario no existe esa entrada, usará el mecanismo de *Route Discovery* para encontrar dinámicamente una nueva ruta hasta el nodo destino.

En el siguiente ejemplo, el nodo A trata de descubrir una ruta para dirigir paquetes al nodo E. El mecanismo de *Route Discovery* iniciado por el nodo A es el siguiente:



El nodo A transmite una *Route Request* a través de un paquete *broadcast* que es recibido por (aproximiadamente) todos los nodos que están dentro del rango de transmisión de A (en el ejemplo, el nodo B). Cada *Route Request* identifica el nodo origen y destino de la ruta a descubrir, y contiene un identificador de petición único ("2" en el ejemplo) determinado por el nodo emisor de la petición. Cada *Route Request* también contiene una lista de registros con las direcciones de cada nodo intermedio a través de los cuales esta copia particular de *Route Request* ha atravesado. Esta lista esta inicialmente vacía. En el ejemplo, inicialmente solo contiene el nodo A.

Cuando la *Route Request* llega a otro nodo (en el ejemplo, al nodo B), si dicho nodo es el nodo destino de la petición, devuelve un *Route Reply* al emisor del *Route Discovery*, devolviéndole una copia del registro de nodos que conforman la ruta hasta él; cuando el emisor recibe la *Route Reply*, almacena la ruta en su *Route Cache* para utilizarla más tarde cuando quiera enviar paquetes al nodo destino.

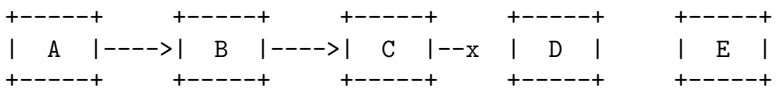
Si por el contrario el nodo receptor de la *Route Request* ha "visto" otro mensaje de *Route Request* del nodo emisor con el mismo identificador de petición y de dirección destino, o si la dirección del propio nodo está ya en la lista de nodos de la *Route Request*, entonces el nodo descarta la petición. En cualquier otro caso (el nodo receptor ni es el destino de la *Route Request*, ni ha visto antes otro mensaje con el mismo identificador de petición y de dirección destino, ni el propio nodo está ya en la lista de nodos), el nodo añade su dirección a la lista y propaga la *Route Request* transmitiéndola a través de *broadcast* (con el mismo identificador de petición). En el ejemplo, el nodo B envía un paquete *broadcast* con la *Route Request* y es recibido por el nodo C; a continuación el nodo C y

el *D*, hasta que una copia de la *Route Request* llega al nodo destino *E*, el cual contestará al nodo *A* con el camino completo hasta él. Para hacerle llegar la respuesta utilizará el camino recién creado en la solicitud del nodo *A*.

Al iniciarse la *Route Discovery*, el nodo emisor guarda una copia del paquete original en un *buffer* local denominado *Send Buffer*. El *Send Buffer* contiene una copia de cada paquete que no puede ser transmitido por el nodo emisor debido a que aún no tiene la ruta en encaminamiento hacia el nodo destino del paquete. Cada paquete en el *Send Buffer* tiene asociado la hora en la que fue introducido en el *buffer*, siendo descartado después de un cierto período de tiempo.

Route Maintenance

Al originarse o encaminar un paquete utilizando una ruta en origen, cada nodo que transmite el paquete es responsable de la confirmación de que el paquete ha sido recibido por el seguiente salto de la ruta en origen. En el ejemplo siguiente, el nodo *A* desea enviar un paquete al nodo *E* utilizando ruta en origen a través de los nodos intermedios *B*, *C* y *D*.



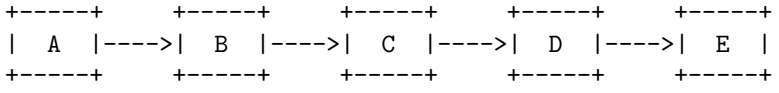
En este caso, el nodo *A* es responsable de la recepción del paquete en el nodo *B*, el nodo *B* es responsable de la entrega del paquete en el nodo *C*, el nodo *C* de la recepción en el nodo *D*, y el nodo *D* es responsable de la recepción final en el nodo destino *E*.

Si no se recibe confirmación de que el paquete ha sido recibido en el nodo siguiente de la ruta, tras un número máximo de intentos, el nodo que es responsable de la entrega del paquete en el siguiente nodo debería devolver un *Route Error* al nodo emisor del paquete indicando el enlace por el cual no ha podido ser encaminado el paquete. En el ejemplo anterior, el nodo *C* no puede entregar el paquete en el siguiente nodo, el *D*, y devuelve un *Route Error* a *A* indicando que el enlace entre *C* y *D* está caído. El nodo *A* eliminará de su *Route Cache* el enlace para ese nodo.

Actualización de *Route Caches* en nodos intermedios

Un nodo que está en la ruta en origen de un paquete, puede añadir información de encaminamiento al encaminar los paquetes hacia el nodo destino a su popia *Route Cache* de cara a poder utilizarla en futuros envíos.

Por ejemplo, el nodo *A* utiliza una ruta en origen para comunicarse con *E*.

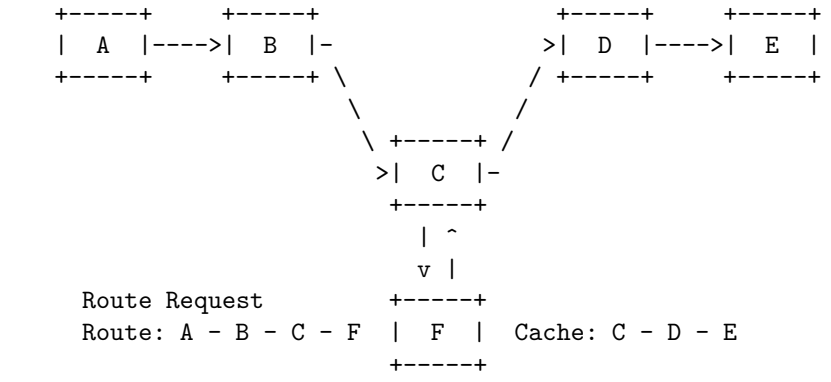


Como el nodo *C* encamina el paquete en la ruta desde *A* hasta *E*, puede actualizar su cache con la información de las rutas hacia *A* y hacia *E*.

Respuestas a las *Route Requests*

Cuando un nodo recibe un *Route Request* para la cual él no es el destino de la misma, busca en su *Route Cache* a ver si existe una ruta para el nodo destino de la petición. Si la encuentra, el nodo generalmente devuelve un *Route Reply* al emisor en vez de continuar enviando la solicitud hasta el nodo destino. En la *Route Reply*, este nodo actualiza la lista de nodos que tiene que seguir el paquete en su camino hasta el destino, concatenando los que ya traía con los de la ruta que tenía en su cache.

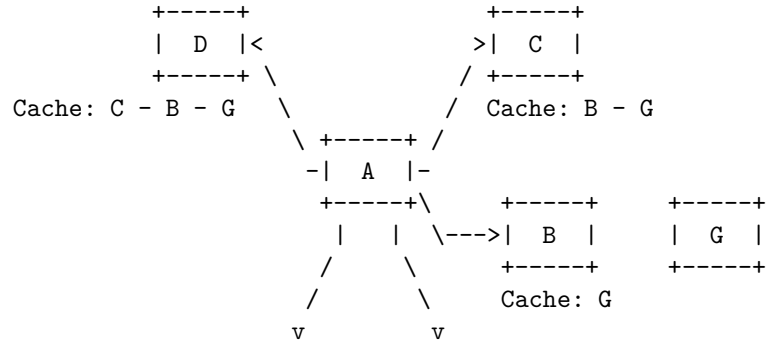
Sin embargo, antes de transmitir la *Route Reply*, el nodo debe verificar que la lista resultante a enviar en la *Route Reply* no contenga nodos duplicados. Por ejemplo, la siguiente figura ilustra una situación en la que la *Route Request* para el destino E ha sido recibida por el nodo F, y el nodo F ya tiene en su *Route Cache* una ruta de él mismo hasta el nodo E:

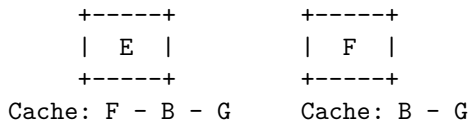


La concatenación de la lista de nodos que traía la *Route Reply* con la ruta que el nodo F tenía en su *Route Cache* produce un nodo duplicado al pasar de C a F y de F a C nuevamente.

Evitar tormentas de *Route Replies*

La siguiente figura muestra una situación en la que los nodos B, C, D, E y F reciben una *Route Request* de A con destino G, y cada uno tiene en su *Route Cache* una ruta hacia el nodo destino mencionado:





Cada nodo debería retrasar el envío de su propia *Route Reply* durante un pequeño período, mientras escucha a ver si el nodo emisor comienza a utilizar una ruta más corta primero.

4.1.2. Estructuras de datos necesarias

Se ha de usar una estructura de datos que modele la *Route Cache*. Para ello, se podría utilizar un array de *strings* en el que cada índice del array representaría el nodo al que se desea llegar, y en el *string* se almacenaría el conjunto de nodos por los que debe pasar el paquete a encaminar. Cada letra del *string* se corresponde con la posición que ocupa dicha letra en el abecedario: la "a" corresponde al robot número 1, la "b" al 2, ... Esto se podría implementar con una función que devuleva el código ASCII de la letra restando el valor 64 que se corresponde con el valor ASCII anterior a la "a", la primera de las letras:

```
número de robot = char (letra) - 64
```

Para conocer la letra correspondiente a un robot se utilizaría la función inversa siguiente:

```
letra de robot = char (número + 64)
```

Un ejemplo de *Route Cache* sería el siguiente:

1 (="a")	"bfe"
2 (="b")	-
3 (="c")	"bdg"
4 (="d")	"c"
5 (="e")	"a"
6 (="f")	"abcdeghi"
7 (="g")	-
8 (="h")	-
9 (="i")	"aceg"
10 (="j")	-

La tabla representaría la *Route Cache* de un nodo en una red *ad hoc* compuesta por 10 nodos. Por cada uno de los 10 nodos existe un *string* con la ruta hacia el nodo que hace de índice de la tabla. Así, para ir al nodo "6" desde el nodo actual, la ruta a seguir es (si el nodo al que pertenece la tabla es el nodo "7" por ejemplo) "7-1-2-3-4-5-8-9-10-6".

El tamaño máximo de un *string* en una tabla será igual al número de nodos que forman la red menos 2 (todos los nodos de la red excepto el orgien y el destino de la ruta). En el peor de los casos, los mensajes de cada nodo tienen que pasar todos los demás nodos de la red, por lo que el tamaño máximo en memoria de la tabla sería:

```
Tamaño = Número de nodos * (Número de nodos - 2)
```

En el ejemplo, con $N = 10$, el tamaño máximo será de

$$\text{Tamaño} = 10 * (10 - 8) = 20 \text{ bytes}$$

La forma, pues, de declarar el array de *strings* que contendrá cada nodo para mantener la *Route Cache* tendrá un aspecto similar a este:

```
RouteCache : array [1..N] of string[N - 2]
```

4.1.3. Formato de los mensajes

El *LegOS Network Protocol* proporciona dos tipos de mensajes:

- *Paquetes de integridad.*

```
+--+---+-----------------------+---+
|F0|LEN|        IDATA          |CHK|
+--+---+-----------------------+---+

F0    : identifies an integrity packet
LEN   : length of IDATA section, 0 to 255
IDATA : payload data
CHK   : checksum
```

  No llevan ningún tipo de información acerca de direccionamiento. Se pueden considerar como el mecanismo de *broadcast* del *LNP*.

- *Paquetes de direccionamiento.*

```
+--+---+----+---+--------------+---+
|F1|LEN|DEST|SRC|   ADATA      |CHK|
+--+---+----+---+--------------+---+
       |                       |
       |<-      IDATA        ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data (1 to 253 bytes)
```

  En ellos se especifica la dirección origen y destino del paquete, con los datos correspondientes.

Mensajes del protocolo *DSR*

De acuerdo con los mecanismos del protocolo *DSR* (*Route Discovery* y *Route Maintenance*), con los dos tipos de paquetes comentados anteriormente, se satisfacen perfectamente las necesidades para la implementación del protocolo.

El mecanismo de *Route Discovery* permite a un nodo S(ource) que quiere enviar un paquete a un nodo D(estiny), obtener la ruta en origen que le permite enviar el paquete. Únicamente se usa cuando S intenta enviar un paquete a D y no conoce la ruta hasta él. Para llevar a cabo tal acción, el nodo emisor envía un paquete *broadcast* de *Route Request* que será recibido por todos los nodos que están dentro de su radio. Por tanto, este paquete *broadcast* de *Route Request* se implementará utilizando el *Paquete de integridad* que proporciona *LNP*. En el

campo de datos de dicho paquete se especificará el nodo D(estiny) del cual se desea conocer la ruta hasta él.

Los otros tipos de paquetes necesarios para la implementación del protocolo *DSR* son:

- *Route Reply*. Es el paquete que le devuelven a un nodo S(ource) que ha enviado previamente una petición de *Route Request* y en el que se le informa de la ruta a seguir para encaminar paquetes hasta el nodo D(estiny) al cual hizo la solicitud en el *Route Request*.
- *Route Error*. Es el paquete que se devuelve cuando un nodo no puede entregar un paquete al siguiente nodo de la ruta.
- *Route ACK*. Es el paquete de confirmación que un nodo le envía a su predecesor en la ruta de encaminamiento para indicarle que ha recibido su paquete correctamente.

Por tanto se tienen 3 tipos de paquetes. Con lo cual, con 2 bits se pueden codificar estos tipos:

Tipo de paquete	Identificador de paquete	Descripción
1	*RREP*	*Route Reply*
2	*RERR*	*Route Error*
3	*RACK*	*Route ACK*

Los campos `SRC` y `DST` de los *Paquetes de integridad* tienen tamaño de 1 byte. Esos 8 bits se dividen en una parte para indicar la dirección de la máquina y otra parte para indicar el puerto. La macro `CONF_LNP_HOSTMASK`, definida en el fichero `config.h`, determina qué parte de esos 8 bits están dedicados a llevar la información de la dirección de la máquina y qué parte están dedicados a la parte del puerto.

En nuestra implementación utilizaremos 6 bits para la parte de dirección y 2 para el puerto.

De los 6 bits de mayor peso, los 2 primeros se reservan para posibles ampliaciones del protocolo relacionadas con *multicast*. Los otros 4 se corresponderían con el número de máquinas direciconables. De esa forma se podrán direccionar hasta 16 máquinas, más que suficientes para probar los objetivos propuestos.

La solución podría adaptarse y configurar la macro `CONF_LNP_MASK` de tal forma que los 7 primeros bits fueran la dirección de las máquinas, y el bit restante el puerto (con un puerto para la recepción de los datos sobra). Así, se podrían tener hasta 128 *hosts* participando en el protocolo. Sin embargo, tan alto número de robots aumentaría considerablemente el tamaño de la memoria utilizada y de los paquetes que se envían. En el peor de los casos, cuando un paquete ha de atravesar los 127 nodos de la red, deberá llevar esos 127 números de máquina en la parte de datos del paquete. Esto haría crecer el tamaño de los paquetes con el consecuetne aumento del tiempo que tardan los robots en enviar y recibir los datos a través de los infrarrojos.

De igual forma, la estructura de datos en la que se almacenan las rutas que indican cómo "alcanzar" a cada nodo de la red, también se verían incrementadas considerablemente. En el caso de utilizar 16 *hosts* y en la peor de las situaciones, cada nodo tendría una entrada para cada uno de los restantes 15 nodos por

los cuales tienen que pasar los paquetes. Esto serían 15 x 15 = 225 bytes en memoria. Si fueran 128 máquinas, el tamaño de la estructura en el peor de los casos sería 127 x 127 = 16129 bytes. A primera vista no parecen tamaños muy significativos, pero si tenemos en cuenta que los robots poseen 32K de memoria, de los cuales 18K aproximadamente los consume el sistema operativo *LegOS*, no queda demasiado espacio disponible para derrocharlo.

Mensajes de información

El programa que se ejecuta en el PC va a ser el encargado de desencadenar las peticiones. Al mandar un paquete de direccionamiento que no es un paquete del protocolo *DSR* (byte 1 de la sección `AADATA` de los paquetes de direccionamiento con el valor "2", como se explica más adelante), en estas pruebas, el robot recibirá una información y actuará en consecuencia.

Como ejemplo, hemos hecho que se haga una petición para que en el robot haya movimiento hacia la derecha o la izquierda.

Detalles sobre la implementación

A continuación se detallan los distintos tipos de mensajes que se manejarán:

- Paquetes del protocolo *DSR*:
  - *Route Request*
  - *Route Reply*
  - *Route Error*
  - *Route ACK*
- Paquetes de información

A la hora de implementar estos paquetes con los tipos de paquetes que proporciona el protocolo *LNP* , se utilizarán:

- Paquetes de integridad
  - *Route Request*
- Paquetes de direccionamiento
  - *Route Reply*
  - *Route Error*
  - *Route ACK*
  - Paquetes de información

El formato de los paquetes de direccionamiento quedará definido de la siguiente forma, partiendo de la estructura de los mensajes de integridad:

```
+--+---+----+---+--------------+---+
|F1|LEN|DEST|SRC|   ADATA      |CHK|
+--+---+----+---+--------------+---+
       |                       |
```

```
            |<-      IDATA        ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data (1 to 253 bytes)
```

En el primer byte[2] del campo `ADATA` (byte 1), se especificará si el paquete es un paquete del protocolo *DSR* ("1") o de información ("2").

Si se trata de un paquete del protocolo *DSR*, el siguiente byte (byte 2) contedrá uno de estos tres valores:

- 1 - *Route Reply - RREP*
- 2 - *Route Error - REER*
- 3 - *Route ACK - RACK*

Si por el contrario es un paquete de información, el valor de los bytes siguientes (a partir del byte 2) se organizarán de la siguiente forma:

```
cadena_con_ruta_a_seguir_por_paquete + ‘‘-’’ + info
```

Por ejemplo:

```
ABCDE-12
```

Esto indicaría que el paquete es un paquete con la información *"12"* y que tiene que seguir la ruta *A-B-C-D-E*.

En los paquetes del protocolo *DSR*, el resto de bytes (del 3 al 253) del campo `ADATA` se utilizarán (si es el caso) para almacenar información específica (no de tipo de paquete).

En los paquetes de información, apartir del símbolo de guión ("-") vendrá la información que devuelve el robot al cual se le solicitó.

Por tanto, el esquema general de los paquetes de direccionamiento será el siguiente:

```
+--+---+----+---+-+-+-------------+---+
|F1|LEN|DEST|SRC|T|C|  ADATA      |CHK|
+--+---+----+---+-+-+-------------+---+
       |                           |
       |<-      IDATA           ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
```

[2]Se podría trabajar a nivel de bits pero es más sencillo hacerlo a nivel de bytes y el coste no es demasiado grande

```
DEST   : destination address
SRC    : source address
ADATA  : payload data
T      : tipo de paquete de datos (‘‘1’’|‘‘2’’) - byte 1 de ADATA
C      : clase de paquete - byte 2 de ADATA:
           + T = ‘‘1’’
             - C (‘‘1’’|‘‘2’’|‘‘3’’)
           + T = ‘‘2’’
```

Route Request

El formato de los mensajes de *Route Request* empleados en el mecanismo de *Route Discovery* será:

```
+--+---+----------------------+---+
|F0|LEN|        IDATA         |CHK|
+--+---+----------------------+---+

F0    : identifies an integrity packet
LEN   : length of IDATA section, 0 to 255
IDATA : payload data
CHK   : checksum
```

En el campo `IDATA`, cada nodo que reciba este tipo de mensaje incorporará a la cadena con los nodos por los que ya pasado su identificador (la letra que le identifica).

El nodo que manda el mensaje, en la parte `IDATA` incorporará únicamente su identificador. Los nodos que lo reciben, añadirán al final del *string* (conocido a partir del campo `LEN`) su identificador.

Route Reply

El formato de los mensajes de *Route Reply* será:

```
+--+---+----+---+-+-+-------------+---+
|F1|LEN|DEST|SRC|1|1|  ADATA      |CHK|
+--+---+----+---+-+-+-------------+---+
       |                          |
       |<-      IDATA           ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data
T      : tipo de paquete de datos = ‘‘1’’ - byte 1 de ADATA
C      : clase de paquete = ‘‘1’’ - byte 2 de ADATA:
```

A partir del byte 3 del campo `ADATA` se indicará la ruta completa para alcanzar al nodo destino solicitado por el receptor del mensaje.

Route Error

El formato de los mensajes de *Route Error* será:

```
+--+---+----+---+-+-+-------------+---+
|F1|LEN|DEST|SRC|1|2|  ADATA      |CHK|
+--+---+----+---+-+-+-------------+---+
       |                           |
       |<-      IDATA           ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data
T      : tipo de paquete de datos = ‘‘1’’ - byte 1 de ADATA
C      : clase de paquete = ‘‘2’’ - byte 2 de ADATA:
```

En el byte 3 del campo `ADATA` se indicará el nodo en el cual se produjo el error.

Route ACK

El formato de los mensajes de *Route ACK* será:

```
+--+---+----+---+-+-+-------------+---+
|F1|LEN|DEST|SRC|1|3|  ADATA      |CHK|
+--+---+----+---+-+-+-------------+---+
       |                           |
       |<-      IDATA           ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data
T      : tipo de paquete de datos = ‘‘1’’ - byte 1 de ADATA
C      : clase de paquete = ‘‘3’’ - byte 2 de ADATA:
```

No se ha de especificar nada a partir del byte 3 del campo `ADATA`. Únicamente se informa que el mensaje ha sido entregado correctamente.

Paquetes de información

El formato de los mensajes con información será:

```
+--+---+----+---+-+-------------+---+
|F1|LEN|DEST|SRC|2|  ADATA      |CHK|
+--+---+----+---+-+-------------+---+
       |                         |
       |<-      IDATA         ->|


F1     : identifies an addressing packet
LEN    : length of IDATA section ( 1 to 255 bytes )
DEST   : destination address
SRC    : source address
ADATA  : payload data
T      : tipo de paquete de datos = ‘‘2’’ - byte 1 de ADATA
```

Del byte 2 en adelante, vendrá la información de la siguiente forma, como se ha comentado anteriormente:

```
cadena_con_ruta_a_seguir_por_paquete + ‘‘-’’ + info
```

Por ejemplo:

```
ABCDE-12
```

Esto indicaría que el paquete es un paquete con la información *"12"* y que tiene que seguir la ruta *A-B-C-D-E*.

4.2. PC

El PC va a ser el encargado de interactuar a través de infrarrojos con la *red ad hoc* que formen los *Lego Mindstorms*. El PC va formar parte de la red, siendo un nodo *"especial"* encargado de iniciar las peticiones al resto de nodos.

Las interacción del PC con alguno de los robots de la red *ad hoc* se realizarán a través de *HTTP*, por lo que el programa que se ejecute en el PC será un servidor que responda a peticiones *HTTP*.

Las peticiones que el servidor puede recibir tienen este estilo:

```
GET /info.html?nodo=H&action=luz HTTP/1.0
```

En el PC por tanto vamos a tener los siguientes elementos:

- Servidor HTTP.
- Nivel DSR.
- Demonio LNP.

4.2.1. Servidor HTTP.

Para el servidor HTTP, hemos escogido Python, en su versión 2.1, como lenguaje de implementación, debido a la facilidades que este lenguaje ofrece por medio de su enorme variedad de módulos.

La forma de arrancar el servidor es la siguiente:

```
./sioux.py 8000
```

Lo que hace que el servidor escuche las peticiones en el puerto 8000.

Para implementar el servidor HTTP, utilizamos el módulo *Basic HTTP Server (*`BaseHTTPServer`*)*, que ofrece dos clases `BaseHTTPRequestHandler` y `HTTPServer` que nos dan la implementación requerida. Aunque en realidad sólo ofrece la *infraestructura* que necesitamos, ya que este módulo, requiere que se le defina una clase manejador para las peticiones.

En nuestro caso la clase que maneja las peticiones la denominamos `RCXHTTPHandler` con dos métodos, `do_GET` y `do_HEAD` que servirán las páginas.

`do_HEAD` Este método se encarga de procesar la petición en concreto tiene las siguientes responsabilidades:

- Recoger los parámetros, si hubiere, pasados en la petición. En nuestro caso se recogerá los parámetros de `nodo` y `action` que se pasan cuando se quiere establecer comunicación con un nodo.
- Controlar si la página pedida existe.
- Insertar cabeceras de respuesta.

`do_GET` Lo primero que `do_GET` hace es comprobar la cabecera pedida con una llamada a `do_HEAD`. Una vez inspeccionada la petición, procedemos al servicio. En nuestro caso, sólo damos dos posibilidades, una página inicial, `/index.html` ó `/`, o la petición de información `/info.html`. Desde la página inicial se construye la petición hacia el nodo en la forma descrita anteriormente.

El desencadenante de las peticiones se hace aquí mediante la llamada:

```
RCX.Get(self.dicParams["nodo"][0], self.dicParams[.ªction"][0])
```

que se encuentra en el módulo `RCX` que implementa el protocolo *DSR* y que pasamos a explicar en la siguiente sección.

4.2.2. Nivel DSR.

El nivel DSR en la parte del PC se ha implementado como un módulo para Python escrito en C. Para la implementación del nivel DSR, son necesarios los ficheros `RCXmodule.c`, que hace de interfaz para Python, es decir implementa la llamada `RCX.Get()` y el fichero `dsr.c`, que hace del PC un nodo más en la red ad-hoc.

En cuanto a la compilación, Python ofrece dos formas de incluir el módulo, una estática incluyendo el módulo dentro del interpréte, generando así un nuevo intérprete y otra dinámica, es decir, generando una librería dinámica que será utilizada cuando el intérprete genere las llamadas a nuestro módulo. Nosotros hemos escogido esta segunda forma por ser más independiente.

Python da un `Makefile` genérico para hacer esto, en el cual nosotros hemos debido incluir lo siguiente para compilar nuestra librería:

```
./RCXmodule.o: $(srcdir)/./RCXmodule.c;
        $(CC) $(CCSHARED) $(CFLAGS) -I/usr/local/lnp/lnpd+liblnp/liblnp \
        -c $(srcdir)/./RCXmodule.c -o ./RCXmodule.o
./dsr.o: $(srcdir)/./dsr.c;
        $(CC) $(CCSHARED) $(CFLAGS) -I/usr/local/lnp/lnpd+liblnp/liblnp \
        -c $(srcdir)/./dsr.c -o ./dsr.o
./RCXmodule$(SO):  ./RCXmodule.o ./dsr.o;
        $(LDSHARED)  ./RCXmodule.o ./dsr.o -L/usr/local/lnp/lnpd+liblnp/liblnp \
        -llnp  -o ./RCXmodule$(SO)
```

`RCXmodule.c` Python es un lenguaje muy versátil y la facilidad con que se puede extender con módulos escritos en otros lenguajes, como C y C++, es una prueba palpable de ello.

A continuación paso a describir los elementos utilizados para la extensión ofrecidos por `Python.h`.

- `static PyObject *RCXError;`
  Define una nueva excepción que será utilizada en caso de error específico del módulo.
- `static PyObject *RCX_Get(self, args);`
  Es nuestra función que implementará la llamada `RCX.Get()`. En ella hacemos la llamada `send(nodo[0], action)` que envía un paquete hacia el nodo solicitado.

- `static PyMethodDef RCXMethods[]`
  Define los métodos que exporta el módulo, en nuestro caso `Get()`.

- `void initRCX();`
  Función que es llamada cuando se inicializa el módulo en el *script* Python, esto ocurre cuando se hace `import` del módulo. En ella colocamos la llamada a la inicialización del protocolo *DSR* `init_dsr()`.

`dsr.c` Estas funciones son básicamente iguales que las utilizadas para implementar *DSR* en el lado del RCX, excepto por dos detalles, las funciones de entrada y salida, y los *threads* que reciben los paquetes, que en el lado del PC se utiliza POSIX threads, en concreto dos, uno que se encarga de la recepción de RREQ y otro para el resto de paquetes.

4.2.3. Demonio LNP.

En el PC debe haber un demonio de *LegOS Network Protocol* que se esté comunicando con los RCX. Para ello utilizamos `lnpd` que tan sólo ejecutándolo lo tendremos activo escuchando y enviando peticiones.

Tan sólo un pequeño detalle, y es que como se puede ver en el `Makefile` el módulo `RCX` está enlazado dinámicamente con librerías dinámicas de lnpd, y estas deben estar visibles en el para el intérprete. Esto se consigue incluyendo el directorio donde se encuentran las librerías dinámicas que lnpd ofrece. En nuestro caso `export LD_LIBRARY_PATH=/usr/local/lnp/lnpd+liblnp/liblnp` ha sido suficiente.

4.3. RCX

El comportamiento de cada robot *Lego Mindstorm* integrante de la red *ad hoc* se reduce al intercambio de información entre cada robot y el resto de componentes de la red (el resto de robots o el PC).

Las misiones de cada robot son:

- Mantener / Atender los requerimientos del protocolo de encaminamiento de redes *ad hoc DSR*.

- Atender las peticiones de solicitud de información de alguno de sus sensores.

En cuanto al primer punto, *mantener y atender los requerimientos del protocolo DSR*, se engloban las funciones relacionadas directamente con el protocolo de encaminamiento *DSR*:

- Mecanismo de *Route Discovery* a través de mensajes de *Route Request*

- Mensajes de

  - *Route Reply*
  - *Route ACK*
  - *Route Error*

- Mantenimiento de la *Route Cache*

Sobre *atender peticiones de solicitud de información*, cada robot debe poder responder a las solicitudes sobre el estado de alguno de sus sensores.

Así pues, cada robot tendrá un *thread* encargado de gestionar la recepción de mensajes y que actuará en consecuencia dependiendo de si son mensajes del protocolo *DSR* o de solicitud de información.

4.3.1. *LegOS Network Protocol*

Las funciones básicas de cada robot relacionadas con el protocolo *LNP* son las presentadas en el apartado anterior:

```
function crear_paquete
  (in origen  : string;
   in destino : string;
   in tipo    : string;
   in clase   : string;
   in datos   : string) return packet;

function envia_paquete
  (in paquete : packet)
```

Interacciones con el protocolo *DSR*

Existen varias circustancias por las que un robot ha de realizar operaciones relacionadas con el protocolo de encaminamiento *DSR*:

1. Desea enviar un paquete con información a un nodo de la red.

   Si conoce la ruta del nodo al que desea enviar el paquete, el robot se limitará a construir el paquete y mandárselo. Para ello se utilizarán las funciones mencionadas en el apartado anterior, `crear_paquete` y `envia_paquete`.

   Si no tiene la ruta, tendrá que poner en marcha el mecanismo de *Route Discovery* enviando un mensaje de *Route Request*.

2. Es el nodo destino de un *Route Request*.

   El nodo ha de enviar la respuesta al nodo origen de la petición con un mensaje de *Route Reply* y actualizar su *Route Cache*.

3. Es un nodo intermedio en la ruta para encaminar un paquete.

   En este caso ha de realizar dos tareas:

   - Asegurarse que el paquete es entregado en el siguiente nodo de la ruta.

     Si la entrega se realiza satisfactoriamente, se envía un mensaje de asentimiento (*Route ACK*) al nodo que le envío el paquete en el salto anterior de la ruta.

     Si por el contrario no se puede entregar el paquete, se envía un mensaje de error *Route Error* al nodo origen del paquete.
   - Actualizar su *Route Cache*

     Si en el paquete el nodo puede aprender nuevas rutas hacia nuevos nodos, debe actualizar su *Route Cache*

Petición del estado de sus sensores

Cuando un nodo recibe la petición de información acerca de un sensor, ha de obtener la información relativa al sensor y enviarla de vuelta al nodo. Para ello ha de utilizar las funciones de `crear_paquete` y `envia_paquete` comentadas anteriormente.

Capítulo 5

Maqueta de pruebas

Dados una serie de Legos Mindstorm (5 o 6 aproximadamente) separados por distancias considerablemente significativas, se han de poder comunicar todos con todos usando la versión simplificada del protocolo *DSR*.

La siguiente descripción ilustra un posible escenario en el que se desarrollarían las pruebas y el flujo de control que seguiría cada elemento:

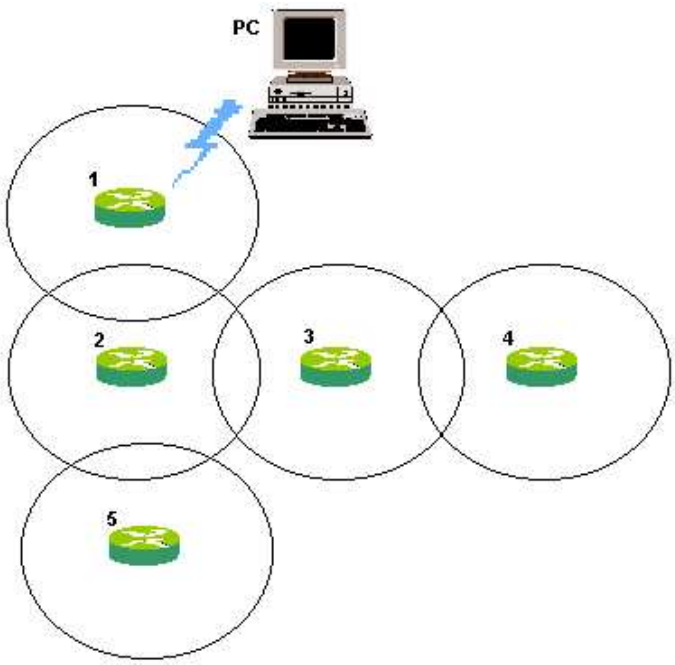


Figura 5.1: Situación inicial

Inicialmente, cada robot está identificado con un número que se utilizará en los *paquetes de direccionamiento* en los campos de dirección origen y de dirección destino (a modo de dirección *IP*).

El *PC* conoce a priori cuántos nodos pueden formar la red *Ad Hoc*, de

forma que en cualquier momento, desde la *Palm* se puede solicitar el estado de cualquiera de esos robots.

Cada robot posee un radio de acción limitado, por lo que mediante los mecanismos de encaminamiento *Ad Hoc* de la versión simplicada del protocolo *DSR*, se puede alcanzar cualquier nodo (siempre que éstos esten disponibles).

Al comenzar, todos los robots tienen sus *Route Caches* vacías. La *Palm* solicita, por ejemplo, el estado del robot número *5*. Esta petición le llega al *PC* que se la hace llegar al robot número *1* que es el que hace de enlace entre la red *Ad Hoc* y el *PC*.

El primer paso para el robot *1* es consultar su *Route Cache*. Como no tiene ninguna entrada para el nodo *5* solicitado, inicia el mencanismo de *Route Discovery* y envía un mensaje de *broadcast* con el *paquete de integridad* que proporciona el *LegOS Network Protocol* (el *Route Request Packet*). En la parte de datos indicará, aparte del identificador de solicitud y el nodo destino, la lista de nodos por los que ha pasado el mensaje y que inicialmente solo contendrá un elemento, el *1* correspondiente al nodo solicitante de la ruta.

Este paquete es recibido por el nodo *2*, que al no ser el nodo destino solicitado retransmitirá el paquete. De esta forma, con sucesivas retransmisiones, el paquete se difundirá por la red, y cada nodo que no sea el nodo destino de la solicitud, se añadirá a la lista de nodos y retransmitirá el paquete.

De esta forma, la retransmisión de la solicitud del robot *2*, alcanzará al robot *5*. Éste, al ser el nodo destino de la solicitud enviará un *Route Reply* al nodo *1* utilizando la ruta recién creada para hacerle llegar el paquete. Por tanto, en la parte de datos se indicará el identificador de la solucitud a la que se contesta, la ruta que ha de seguir y la información (intensidad de la luz, temperatura, ...) solicitada.

La nueva ruta creada (`1-2-5`) será utilizada por los nodos que la componen y através de los cuales pasará el *Route Reply Packet*, para actualizar sus *Route Caches*. De esta forma, se tienen las siguientes entradas en cada uno de estos nodos:

- *Robot #1*:

Nodo destino	*Ruta*
2	1
5	2

- *Robot #2*:

Nodo destino	*Ruta*
1	2
5	2

- *Robot #5*:

Nodo destino	*Ruta*
1	2
2	5

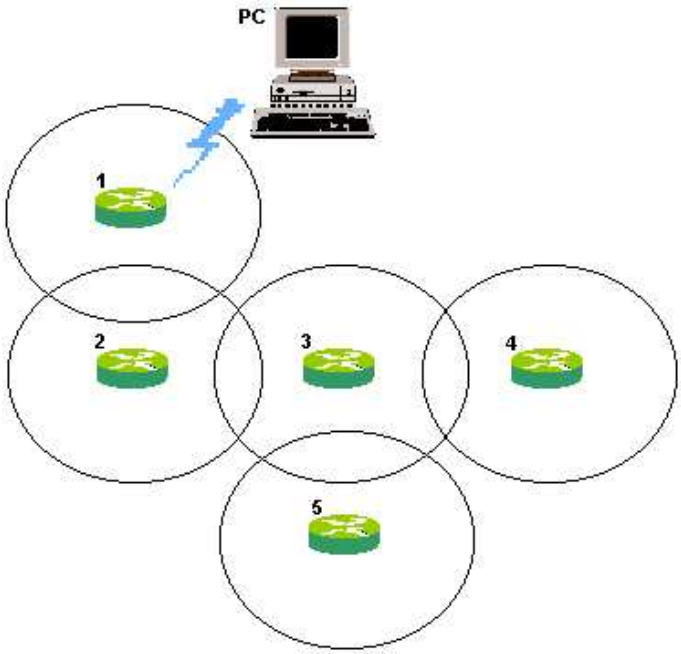


Figura 5.2: Situación 2

En la figura 5.2, el robot *5* ha cambiado de situación y ya no está bajo el radio de acción del robot *2*.

Nuevamente se desea obtener información del robot *5*. El nodo *1* consulta su *Route Cache* y ve que para enviar un paquete al nodo *5* tiene que pasar por el nodo *2*, por lo que envía un paquete dirigido al nodo siguiente en la lista de nodos a atravesar (*2*) para hacerle llegar la solicitud de información junto con dicha lista (*1-2-5*).

Al llegar el mensaje al robot *2*, este lo envía al siguiente nodo que es a la vez el nodo destino, el robot *5*. Sin embargo, dicho robot ya no recibe el mensaje porque ya no está en el radio de acción del nodo *2*, por lo que tras no una serie de reintentos del nodo *2* de hacerle llegar el mensaje y ver que no obtiene el *Route ACK Packet* correspondiente (que informa de que el paquete ha sido entregado correctamente), envía un *Route Error Packet* al nodo origen, el *1*, para indicarle que el nodo *5* ya no es alcanzable desde el nodo *2*.

Al recibir el *Route Error* el nodo *1*, iniciará nuevamente el proceso de *Route Discovery*, siguiendo los mismos pasos que para la figura 5.1.

En este caso, el robot *3* es el nuevo (y único) enlace hacia el nodo destino *5*. Las *Route Caches* de los nodos son nuevamente actualizadas quedando como sigue:

- *Robot #1*:

Nodo destino	*Ruta*
2	1
3	2
5	2-3

- *Robot #2*:

Nodo destino	*Ruta*
1	2
3	2
5	3

- *Robot #3*:

Nodo destino	*Ruta*
1	2
2	3
5	3

- *Robot #5*:

Nodo destino	*Ruta*
1	3-2
2	3
3	5

Otra posible situación representaría la que se muestra en la figura 5.3: Las *Route Caches* quedarían de la siguiente manera:

- *Robot #1*:

Nodo destino	*Ruta*
2	1
3	2
4	2-3
5	2-3-4

- *Robot #2*:

Nodo destino	*Ruta*
1	2
3	2
4	3
5	3-4

- *Robot #3*:

Nodo destino	*Ruta*
1	2
2	3
4	3
5	4

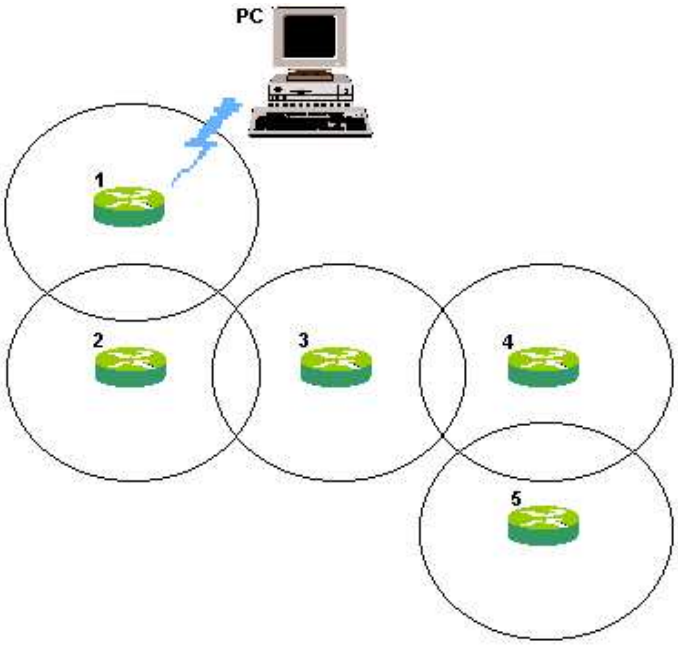


Figura 5.3: Situación 3

- *Robot #4*:

Nodo destino	*Ruta*
1	3-2
2	3
3	4
5	4

- *Robot #5*:

Nodo destino	*Ruta*
1	4-3-2
2	4-3
3	4
4	5

Capítulo 6

Detalles de implementación

6.1. Pruebas con el protocolo *LNP*

En esta sección se hace una introducción al uso del *LegOS Network Protocol* mediante ejemplos.

6.1.1. Envío de paquetes entre dos *RCX*

En este ejemplo se muestra cómo enviar paquetes entre dos *RCX*. Uno de ellos envía paquetes contínuamente, y el otro tiene un manejador y está esperando a recibir mensajes. Cada vez que recibe un mensaje muestra un texto en la pantalla del *RCX*:

Programa que envía:

```
#include <conio.h>
#include <unistd.h>
#include <lnp.h>
#include <lnp-logical.h>

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

  int length = 3;
  char *packet;

  while (1) {
    packet = "aaa";
    cputs("Antes de enviar");
    // send packet
    lnp_integrity_write(packet,length);
    cputs("Despues de enviar");
    sleep (1);
  }
}
```

Fichero fuente

Programa que recibe:

```
#include <dsound.h>
#include <conio.h>
#include <unistd.h>
#include <lnp.h>
#include <lnp-logical.h>

// create a handler function that will receive broadcast packets:
void my_integrity_handler(const unsigned char *data, unsigned char len)
{
  cputs ("RECIBIDO");
}

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

  // activate the handler
  lnp_integrity_set_handler(my_integrity_handler);

  return 0;
}
```

Fichero fuente

6.1.2. Envío de paquetes entre el PC y un *RCX*

En este ejemplo se arrancan varias tareas tanto en el PC como en el *RCX* que mandan mensajes contínuamente, y otra tarea que se encarga de recibir los mensajes. En la parte del PC se utiliza el demonio `lnpd` que es el encargado de enviar los mensajes por el puerto serie al que está conectada la torre de infrarrojos.

Programa en el PC:

```
//
// kleines Testprogram für linux lnp
//

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>

#include "liblnp.h"

#define MY_PORT_1  7
#define MY_PORT_2  8

#define DEST_HOST 0
#define DEST_PORT 2
```

```
#define DEST_ADDR ( DEST_HOST << 4 | DEST_PORT )
#define LEN 253

void addr_handler_1
(const unsigned char* data,unsigned char length, unsigned char src)
{
    char pbuf[100];

    sprintf(pbuf,">> Source:%2X ",(unsigned)src);
    sprintf(pbuf,"Length:%u",(unsigned)length);
    sprintf(pbuf,"PacketNo:%u <<\n"(unsigned)data[0]);

    write(STDERR_FILENO,pbuf,strlen(pbuf));
}

void addr_handler_2
(const unsigned char* data,unsigned char length, unsigned char src)
{
    char pbuf[100];

    sprintf(pbuf,">> Source:%2X ",(unsigned)src);
    sprintf(pbuf,"Length:%u",(unsigned)length);
    sprintf(pbuf,"PacketNo:%u <<\n"(unsigned)data[0]);


    write(STDERR_FILENO,pbuf,strlen(pbuf));
}

void int_handler(const unsigned char* data,unsigned char length)
{
    char pbuf[100];

    sprintf(pbuf,">> Integrity Length:%u",(unsigned)length);
    sprintf(pbuf,"PacketNo:%u <<\n",(unsigned)data[0]);

    write(STDERR_FILENO,pbuf,strlen(pbuf));
}

int main(int argc, char *argv[])
{
    char data[253];
    int i;
    lnp_tx_result result;
    unsigned char len;
    int count = 0;

    for (i=0;i<sizeof(data);i++) data[i] = i;

    if ( lnp_init(0,0,0,0,0) )
    {
```

```
        perror("lnp_init");
        exit(1);
    }

    else fprintf(stderr,"init OK\n");

    lnp_addressing_set_handler (MY_PORT_1, addr_handler_1);
    lnp_addressing_set_handler (MY_PORT_2, addr_handler_2);
    lnp_integrity_set_handler (int_handler);

    while (1)
    {
        //sleep(1000);
        //continue;
            len = LEN; //random() % 252 + 1;
                result = lnp_addressing_write(data,len ,DEST_ADDR,MY_PORT_1);
                switch (result)
                {
                case TX_SUCCESS:
                        printf("Tansmitted %d : %d\n",len,count++);
                        break;
                case TX_FAILURE:
                        printf("Collision\n");
                        break;
                default:
                        perror("Transmit error");
                        exit(1);
        }
        }

    return 0;
}
```

Programa en el RCX:

```
//
// send & receive lnp Packets
//


// terminate this program with the PRGM-Button !!!

#include <lnp.h>
#include <conio.h>
#include <sys/irq.h>
#include <sys/h8.h>
#include <tm.h>
```

```
#include <dkey.h>
#include <unistd.h>
#include <lnp-logical.h>
#include <sys/lnp-logical.h>

#define MY_PORT 2

#define DEST_HOST 0x8
#define DEST_PORT 0x7
#define DEST_PORT_2 0x8
#define DEST_ADDR ( DEST_HOST << 4 | DEST_PORT )
#define DEST_ADDR_2 ( DEST_HOST << 4 | DEST_PORT_2 )
#define LEN_1 100
#define LEN_2 77
#define LEN_3 13

static unsigned char rx_count;
static unsigned char tx_count_1,tx_count_2,tx_count_3;
static int done,sender_1_done,sender_2_done,sender_3_done;

wakeup_t prgmKeyCheck(wakeup_t data)
{
    return dkey == KEY_PRGM;
}

wakeup_t countchanged(wakeup_t data)
{
        wakeup_t value = 0;
        static unsigned char oldrx,oldtx;

        if ( (tx_count_1 + tx_count_2 + tx_count_3) != oldtx )
        {
                oldtx = tx_count_1 + tx_count_2 + tx_count_3;
                value = 1;
        }
        if (rx_count != oldrx)
        {
                oldrx = rx_count;
                value = 1;
        }
        return value;
}

wakeup_t countOrKey(wakeup_t dummy)
{
        return ( countchanged(0) || prgmKeyCheck(0) );
}

wakeup_t childTest(wakeup_t dummy)
{
```

```
        return ( sender_1_done && sender_2_done && sender_3_done );
}

void show_counters(void)
{
    cputw( ( ( tx_count_1 + tx_count_2 + tx_count_3) << 8) | rx_count );
}

void packet_handler
(const unsigned char* data,unsigned char length, unsigned char src)
{
        ++rx_count;
}

int sender_1(int argc, char *argv[])
{
    unsigned char data[LEN_1];
    int i;
    unsigned char len = LEN_1;
    signed char result;

    for (i=0;i<sizeof(data);i++) data[i] = i;

    while(!done)
    {
        data[0]=tx_count_1;
        result = lnp_integrity_write(data,len);
        if (result == TX_IDLE) ++tx_count_1;
        msleep(10);
    }
    sender_1_done = 1;
    return 0;
}

int sender_2(int argc, char *argv[])
{
    unsigned char data[LEN_2];
    int i;
    unsigned char len = LEN_2;
    signed char result;

    for (i=0;i<sizeof(data);i++) data[i] = i;

    while(!done)
    {
        data[0]=tx_count_2;
        result = lnp_addressing_write(data,len ,DEST_ADDR,MY_PORT);
        if (result == TX_IDLE) ++tx_count_2;
        msleep(10);
    }
```

```
        sender_2_done = 1;
    return 0;
}

int sender_3(int argc, char *argv[])
{
    unsigned char data[LEN_3];
    int i;
    unsigned char len = LEN_3;
    signed char result;

    for (i=0;i<sizeof(data);i++) data[i] = i;

    while(!done)
    {
        data[0]=tx_count_3;
        result = lnp_addressing_write(data,len ,DEST_ADDR_2,MY_PORT);
        if (result == TX_IDLE) ++tx_count_3;
        msleep(10);
    }
        sender_3_done = 1;
    return 0;
}

int main(int argc, char *argv[])
{
    lnp_logical_range(0);
    lcd_clear();
    cputs("wait");

    lnp_addressing_set_handler(MY_PORT, packet_handler);
    show_counters();

        execi(sender_1,0,NULL,PRIO_NORMAL,DEFAULT_STACK_SIZE);
        execi(sender_2,0,NULL,PRIO_NORMAL,DEFAULT_STACK_SIZE);
        execi(sender_3,0,NULL,PRIO_NORMAL,DEFAULT_STACK_SIZE);

        while ( 1 )
        {
                wait_event(countOrKey,0);
                show_counters();
        if (prgmKeyCheck(0)) break;
    }

    done = 1;

    wait_event(childTest,0);
    lnp_addressing_set_handler(MY_PORT,LNP_DUMMY_ADDRESSING);
    cputs("done");
    return 0;
```

```
}
```

Y este es un ejemplo de los mensajes que se muestran en el PC, se muestran los mensajes que se envían y los que se reciben. También se muestran las colisiones :

```
root@Chapi:/usr/local/lnp/lnpd/lnpd+liblnp/applications# ./lnptest
init OK
Tansmitted 253 : 0
Tansmitted 253 : 1
Tansmitted 253 : 2
>> Integrity Length:100 PacketNo:0 <<
Tansmitted 253 : 3
>> Source: 2 Length:77 PacketNo:0 <<
Tansmitted 253 : 4
>> Source: 2 Length:13 PacketNo:0 <<
Tansmitted 253 : 5
>> Integrity Length:100 PacketNo:1 <<
Tansmitted 253 : 6
>> Source: 2 Length:77 PacketNo:1 <<
Tansmitted 253 : 7
>> Source: 2 Length:13 PacketNo:1 <<
Tansmitted 253 : 8
>> Integrity Length:100 PacketNo:2 <<
Tansmitted 253 : 9
>> Source: 2 Length:77 PacketNo:2 <<
Tansmitted 253 : 10
>> Source: 2 Length:13 PacketNo:2 <<
Tansmitted 253 : 11
>> Integrity Length:100 PacketNo:3 <<
Tansmitted 253 : 12
>> Source: 2 Length:77 PacketNo:3 <<
Tansmitted 253 : 13
>> Source: 2 Length:13 PacketNo:3 <<
Tansmitted 253 : 14
>> Integrity Length:100 PacketNo:4 <<
Tansmitted 253 : 15
>> Source: 2 Length:77 PacketNo:4 <<
Tansmitted 253 : 16
Collision
Tansmitted 253 : 17
>> Integrity Length:100 PacketNo:5 <<
Tansmitted 253 : 18
>> Source: 2 Length:77 PacketNo:5 <<
Tansmitted 253 : 19
>> Source: 2 Length:13 PacketNo:4 <<
Tansmitted 253 : 20
>> Integrity Length:100 PacketNo:6 <<
```

```
Tansmitted 253 : 21
>> Source: 2 Length:77 PacketNo:6 <<
Tansmitted 253 : 22
>> Source: 2 Length:13 PacketNo:5 <<
Tansmitted 253 : 23
>> Integrity Length:100 PacketNo:7 <<
Tansmitted 253 : 24
Collision
Tansmitted 253 : 25
>> Source: 2 Length:13 PacketNo:6 <<
Tansmitted 253 : 26
>> Integrity Length:100 PacketNo:8 <<
Tansmitted 253 : 27
```

6.2. Implementación del protocolo *DSR* simplificado

A continuación se comentan las líneas principales del código para implementar la versión simplificada del protocolo *DSR* para los *Legos MindStorm*.

6.2.1. Fichero `dsr.h`

En este fichero se incluyen los prototipos de las principales funciones del protocolo. La que pone en funcionamiento el protocolo es la función `init_dsr (int tipo, int dir)`, a la cual se ha de pasar como parámetros dos enteros, uno para indicar en dónde se ejecuta el programa que utiliza el protocolo (*"1"* para indicar que se ejecuta en el *PC* y *"2"* para indicar que es sobre uno de los *Legos MindStorm*) y otro para indicar la dirección que identificará al nodo dentro del sistema, como se muestra a continuación:

```
// Iniciar el protocolo para poder enviar/recibir datos
// Se le pasa el tipo de máquina en la que se usará:
// 1-> PC    2-> Legos
void init_dsr (int tipo, int dir);
```

Las servicios que proporcionará el protocolo serán básicamente dos, una para enviar paquetes y otra para recibirlos:

```
// Función utilizada para mandar paquetes de datos.
int send (unsigned char dir, char *data);

// Función utilizada para recibir paquetes de datos.
int recv (unsigned char *dir, char *data);
```

Para detener el protocolo se utiliza la función `void exit_dsr (void)`:

```
// Terminar el protocolo
void exit_dsr (void);
```

6.2.2. Fichero `dsr.c`

Fichero principal en el cual se desarrolla completamente el protocolo. Las secciones principales del fichero son:

Delaración de constantes

Se declaran los valores que permiten adpatar el protocolo a las necesidades del usuario. Así, por ejemplo, para definir el número de nodos que habrá en la *red ad-hoc* se utiliza la constante `NUM_HOSTS`:

```
// Número de nodos que habrá en la red
#define NUM_HOSTS 10
```

El número de envíos de paquetes que determinan cuando caducarán las rutas en las *Route Caches* de los nodos está determinado por la siguiente constante:

```
// Variable que servirá para calcular el tiempo que durarán
// las rutas en la cache. Cada vez que haga un envío se
// incrementará, y cuando llegue a un umbral
// se borrarán las cachés.
#define MAX_SENDS 3
```

Declaración de variables

Se declaran las variables que se utilizarán en las distintas funciones. Entre las más importantes, destacan:

- Estructura de datos para las *Route Caches*.

  Las *Route Caches* se modelan como un array de cadenas de caracteres. La longitud de estas cadenas de caracteres está determinada por el número de nodos que hay en la red:

  ```
  // Tipo ruta
  typedef char t_path[NUM_HOSTS];

  // Tipo Route cache
  typedef t_path t_route_cache[NUM_HOSTS];

  // Cache para guardar las rutas
  t_route_cache route_cache;
  ```

- *Cadena con la ruta obtenida tras un Route Request.*

  Tras iniciar un *Route Discovery* mediante un paquete *Route Request*, la cadena con la ruta hasta el nodo destino es almacenada en la variable `route`:

  ```
  // Ruta obtenida del rreq
  char *route;
  ```

- *Variable para controlar el último id de Route Request recibido.*

  Cada paquete *Route Request* lleva un identificador que permite a los nodos distinguir cuando procesan por primera vez un *Route Request* o reciben (debido a que estos paquetes se manda por inundación a toda la red) una petición de *Route Request* que ya ha procesado con anterioridad.

  ```
  // Lista para guardar el último id_rreq recibido de
  // cada nodo
  char *last_id_rreq;
  ```

  Basta con tener una variable que controle el último *id* de *Route Request* recibido porque en la *red ad-hoc* únicamente va a ver un único *Route Request* circulando al mismo tiempo.

- *Variables asociadas a las tareas de recepción de paquetes.*

  Para manejar los dos tipos de paquetes existentes (paquetes de *broadcast* para las *Route Request* y paquetes de direccionamiento *unicast* para el resto), son necesarias dos tareas que informen sobre cuándo se ha recibido un paquete y los datos que contiene dicho paquete.

  Para el manejo de los paquetes de *Route Request*, se declaran las siguientes variables:

  ```
  // Variables globales para la tarea que procesa
  // los rreq
  int recv_rreq = 0;
  char *data_rreq;
  unsigned char leng_rreq;
  ```

  Cuando se recibe un paquete de este tipo, la variable `recv_rreq` se pone a *"1"* y se guarda la información y la longitud del paquete en `data_rreq` y `leng_rreq` respectivamente.

  De forma análoga. para los paquetes de direccionamiento, se tiene:

  ```
  // Variables globales para la tarea que procesa
  // mensajes con dirección
  int recv_add = 0;
  char *data_recv;
  unsigned char leng_recv;
  unsigned char src_recv;
  ```

- *Variable para comprobar la recepción de ACKs.*

  Cada nodo intermedio en la ruta para la entrega de un paquete de datos es responsable de la entrega del paquete en el nodo siguiente de la ruta.

  La variable `recv_ack` se encarga de indicar si se ha recibido o no el *ACK* procedente del siguiente nodo en la ruta:

  ```
  // Variable para comprobar si recibo un ack
  int recv_ack = 0;
  ```

Funciones

Las principales funciones utilizadas son las siguientes:

- `init_dsr`. *Puesta en marcha del protocolo.*

  El objetivo de esta función es inicializar, en caso de que el nodo en que se ejecuta el programa sea un *RCX*, el *Legos Network Protocol* (con la función `lnp_init`):

```
// Inicializo lnp
if (tipo = 1)
  if ( lnp_init(0,0,dir ,0x03,0) )
  {
    exit(1);
  }
```

  A continuación se establecen los manejadores de los tipos de paquetes que tratará el programa

```
// Inicializo manejador de paquetes broadcast
lnp_integrity_set_handler (recv_route_req);

// Inicializo el manejador para paquetes con dirección
lnp_addressing_set_handler (PORT, recv_address);
```

  y se lanzan las tareas encargadas de realizar las operaciones oportunas al recibirse uno de estos paquetes:

```
// Ejecuto las dos tareas que procesarán los
// paquetes recibidos
execi (test_recv_route_req, 0, NULL, PRIO_NORMAL, DEFAULT_STACK_SIZE);
execi (test_recv, 0, NULL, PRIO_NORMAL, DEFAULT_STACK_SIZE);
```

- `test_recv_route_req`. *Procesamiento de los paquetes de Route Request.*

  Esta es la tarea encargada de la recepción y procesamiento de los paquetes de *Route Request*. Cada 0.1sg comprueba la variable `recv_rreq` que indica si se ha recibido un paquete de *broadcast* (*Route Request*) en el manejador correspondiente:

```
while (!recv_rreq && !done){
  msleep(10);
}
```

  Cuando se ha recibido un paquete de *Route Request*, lo que hay que comprobar es si el nodo que recibe el paquete es el el nodo por el cual se pregunta en el *Route Request*:

```
if (((char)get_host((unsigned)ADDRESS)) !=
     (char)(data_rreq[((int)leng_rreq)-1]))
```

Si es el nodo por el cual se pregunta, se envía la respuesta al nodo que originó la petición:

```
// Las direcciones son iguales.
car = data_rreq[(char)((int)leng_rreq)-2];
packet[0] = '1';
// Tipo de mensaje: route_rep
packet[1] = '1';
packet[2] = '\0';
// Y después concateno la lista de nodos
concat (packet, data_rreq+1);
//cputc ((strlen(packet)+48), 4);

lnp_addressing_write (packet, strlen(packet), get_dir(car), PORT);
cputs ("SRREP");
```

Si no, se reenvía el *Route Request* para que se inunde toda la red:

```
// Reenvío el route_req

car = data_rreq[leng_rreq-1];

data = malloc (leng_rreq+1);
memcpy (data, data_rreq, leng_rreq);
data[leng_rreq] = '\0';
cputs ("RRREQ");

// Espero un tiempo aleatorio para enviar
// No debe ser demasiado bajo, porque se reenvían
// los rreq antes de que llegue el rrep y se
// hacen un lío
sleep (random() % 4);
send_route_req (data);
```

Esta función `test_recv_route_req` está condicionada por la tarea `recv_route_req` que se encarga (además de recuperar la información que viene en el paquete e introducirla dentro de la variable textttdata_rreq) de poner a *"1"* la variable `recv_rreq` que le indica a esta función `test_recv_route_req` que se ha recibido un nuevo *Route Request*:

```
// Manejador para recibir paquetes broadcast
void
recv_route_req
   (const unsigned char *d, unsigned char leng) {

   //sem_wait (&sem_rec_rreq);
   recv_rreq = 1;
   data_rreq = malloc (leng+1);
   memcpy (data_rreq, d, leng);
```

```
    data_rreq[(int)leng] = '\0';
    leng_rreq = leng;
}
```

- `test_recv`. *Procesamiento de los paquetes de direcciones.*

  Esta tarea se encarga de llevar a cabo las operaciones pertinentes al recibirse un paquete *unicast* (un *Route Reply*, un *ACK* o un paquete de datos con información).

  Al igual que en la función `test_recv_route_req`, se comprueba cada0.1sg el estado de la variable `recv_add` que marca cuando se ha recibido un nuevo paquete de direccionamiento:

```
while (!recv_add && !done) {
  msleep(10);
}
```

  Cuando se recibe un paquete, la información del mismo se almacena en la variable `data_rcv`. Esto, al igual que poner el valor de `recv_add` a *"1"* se realiza en la tarea `recv_address`:

```
// Manejador para recibir paquetes con direcciones
void
recv_address
  (const unsigned char* data, unsigned char leng, unsigned char src)
{
  recv_add = 1;

  data_recv = malloc (leng+1);
  memcpy (data_recv, data, leng);
  data_recv[(int)leng] = '\0';

  leng_recv = leng;

  src_recv = src;
  lcd_clear();
  cputs ("RDIR");
}
```

  Dependiendo del valor del primer byte que venga en la parte de datos del paquete, se realizarán unas u otras operaciones. Así, si el valor es *"1"*, indica que se trata de un paquete del protocolo *DSR* (si es un *"2"* significa que es un paquete de datos normal):

```
switch (data_recv[0]) {
  case '1': // Paquete del protocolo DSR
  ...
  case '2': // Paquete de datos
  ...
}
```

Si es un paquete del protocolo *DSR*, hay que saber que clase de paquete es:

```
switch (data_recv[0]) {
  case '1': // Paquete del protocolo DSR
    switch (data_recv[1]) {
      case '1': // Route Reply
      ...
      case '2': // Route Error
      ...
      case '3': // Route ACK
      ...
    }
}
```

- *Route Reply.*
  Cuando es un *Route Reply*, se comprueba si el nodo que está procesando el paquete es el que originó la petición. Si es así, el mecanismo de *Route Discovery* ha finalizado y ya se tiene la ruta hacia el nodo por el cual se preguntó. Se almacena la ruta en cache:

  ```
  if (data_recv[2] == ((char)get_host((unsigned char) ADDRESS))) {
    // Soy el que inicio el route request.
    // Debo guardar la ruta
    strcpy (route_cache[((int)data_recv[leng_recv-1])-64], data_recv+2);
    //cputs (data_recv);
    cputs (route_cache[((int)data_recv[leng_recv-1])-64]);

    //route = malloc (sizeof(char) * (strlen(data_recv)-2));
    memcpy (route, data_recv+2, leng_recv-2);
  }
  ```

  Si no es el nodo que originó el *Route Request*, entonces es un nodo que está en la ruta para devolver la respuesta al nodo origen. Tiene que encaminar el paquete hacia el siguiente nodo en la lista:

  ```
  car = (char) get_host((unsigned)ADDRESS);
  if (prev= prev_node (data_recv, car)) {
    cputs ("RRREP");
    lnp_addressing_write (data_recv, strlen(data_recv), get_dir(prev), PORT);
  }
  ```

- *Route Error.*
  Si el paquete es un paquete de *Route Error* y el nodo que lo recibe es el destinatario del paquete, entonces dicho nodo tiene que borrar de su *Route Cache* la ruta para el cual había enviado un paquete:

  ```
  // Soy yo el destinatario del paquete
  // Borrar la ruta del nodo origen
  cputs ("RRERROR");
  route_cache[((int)data_recv[2])-64][0] = '\0';
  ```

En caso de que no sea el destinatario del paquete, entonces es un nodo que está en la ruta hacia el destinatario final del paquete y ha de reenviarlo al siguiente nodo en la lista:

```
// No soy yo el destinatario, se lo envio al siguiente en la lista
next = next_node (data_recv, ((char)get_host((unsigned char) ADDRESS)));
lnp_addressing_write (data_recv, strlen(data_recv), get_dir(next), PORT);
```

- *Route ACK.*
  El último tipo de paquete del protocolo *DSR* que puede recibir del nodo es un paquete de asentimiento, es decir, que el paquete que el propio nodo había enviado instantes antes ha sido recibido correctamente en el destinatario (el siguiente nodo de la ruta):

  ```
  recv_ack = 1;
  ```

  Al poner este valor a esa variable `recv_ack`, la función `send_packet` sabe que el paquete que acaba de enviar ha sido correctamente entregado. Si no ocurriera eso (que se recibiera asentimiento del paquete recién enviado), entonces se borraría la ruta correspondiente de la *Route Cache*:

  ```
  // Manda un paquete y recibe la respuesta
  int
  send_packet (char *data) {
    ...
    next = next_node (packet, (char)get_host((unsigned)ADDRESS));
    lnp_addressing_write (packet, strlen(packet), get_dir(next), PORT);

    sleep (1);
    if (!recv_ack) {
      cputs ("RERROR");
      route_cache[((int)route[strlen(route)-1])-64][0] = '\0';
    }
    recv_ack=0;
    ...
  }
  ```

Si el paquete recibido no es paquete del protocolo *DSR*, entonces es un paquete de datos:

```
switch (data_recv[0]) {
  ...
  case '2': // Paquete de datos
  ...
}
```

Al recibir un paquete de datos, la primera acción a realizar es enviar el asentimiento al nodo que envío el paquete:

```
packet[0] = '1';
packet[1] = '3';
packet[2] = '\0';
```

```
cputs ("SACK");
prev = prev_node (data_recv, (char) get_host((unsigned) ADDRESS));
// Envío el ack al nodo que me envía el paquete
lnp_addressing_write (packet, strlen(packet), get_dir(prev), PORT);
```

Una vez realizada esta operación, los casos que se pueden dar ante un paquete de datos son los siguientes:

1. *El paquete de datos va dirigido al nodo que lo recibe*
   Se guarda la información recibida para procesarla según las necesidades:

```
if (data_recv[ind(data_recv,'-')-1] == ((char)get_host((unsigned char) ADDRESS))) {
  // El paquete va destinado a mi
  // Guardo los datos
  //sem_wait (&sem_rec_datos);
  recv_packet = 1;
  dir_packet = data_recv[1];
  data_packet = malloc (sizeof(char)*(leng_recv-(ind(data_recv, '-'))));
  //memcpy (data_packet, data_recv+(ind(data_recv,'-'))+1, (leng_recv-(ind(data_recv, '-')
  strcpy (data_packet, data_recv+(ind(data_recv,'-'))+1);
  data_packet[strlen(data_packet)]='\0';
  lcd_clear();
  cputs (data_packet);
}
```

2. *El nodo que recibe el paquete es un intermediario en la ruta hacia el nodo destino del paquete.*
   En este caso se ha de reenviar el paquete al siguiente nodo de la lista:

```
// Reenvío el paquete al siguiente en la lista
next = next_node (data_recv, (char)get_host((unsigned char) ADDRESS));
lnp_addressing_write (data_recv, strlen(data_recv), get_dir(next), PORT);
```

- `send`. *Envío de paquetes de datos.*

  Mediante esta función, se envían los datos pasados como parámetro de la función a la dirección especificada (también pasada como parámetro):

```
int
send (unsigned char dir, char *data)
```

  El procedimiento que sigue a la hora de enviar un paquete es el siguiente: si el nodo que desea enviar el paquete no tiene en su *Route Cache* la ruta para hacerle llegar el paquete al nodo, entonces se ha de poner en marcha el mecanismo de *Route Discovery* para hallarla:

```
if (route_cache[dir-64][0] != '\0'){
  cputs ("CACHE");
  strcpy (route, route_cache[dir-64]);
} else
  send_route_req (d);
```

Una vez que se disponga de la ruta hacia el nodo (se ha recibido el paquete de *Route Reply* y el manejador correspondiente a dejado la ruta buscada en la variable `route`), únicamente se ha de invocar a la función `send_packet` que se encarga del envío de paquetes:

```
if (route[0] != 0) {
  // Mando paquete
  send_packet(data);
  exit = 1;
} else
  sleep (num_send);
```

La función `send_packet` la única misión que tiene es hacer llegar el paquete con los datos al nodo indicado y esperar el asentimiento del nodo destino:

```
// Manda un paquete y recibe la respuesta
int
send_packet (char *data) {
  ...
  next = next_node (packet, (char)get_host((unsigned)ADDRESS));
  lnp_addressing_write (packet, strlen(packet), get_dir(next), PORT);

  sleep (1);
  if (!recv_ack) {
    cputs ("RERROR");
    route_cache[((int)route[strlen(route)-1])-64][0] = '\0';
   }
   recv_ack=0;
   ...
}
```

- `recv`. *Recepción de paquetes de datos.*

  Esta función se encarga de la recepción y procesamiento de los paquetes de datos:

```
// Función para recibir paquetes de datos.
int
recv (unsigned char *dir, char *data) {

  while (!recv_packet)
  msleep (10);

  //data = malloc (sizeof(char) * (strlen(data_packet)+1));
  memcpy (data, data_packet, strlen(data_packet));
  data[strlen(data_packet)] = '\0';
  *dir = dir_packet;

  lcd_clear();
  cputs (data_packet);

  recv_packet = 0;
  //sem_post (&sem_rec_datos);
```

```
  return 0;
}
```

- `exit_dsr`. *Finalización del protocolo.*

  Se pone a *"1"* la variable encargada de indicar que se desea para el protocolo:

```
// Finalizar el protocolo
void
exit_dsr() {

  done = 1;
  lnp_addressing_set_handler(PORT,LNP_DUMMY_ADDRESSING);
}
```

6.2.3. Fichero `pru_dsr.c`

El siguiente ejemplo muestra como enviar y recibir paquetes usando el protocolo *DSR*:

```
#include <unistd.h>
#include <conio.h>
#include <dsr.h>


#define DIR 0x1

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

  unsigned char tmp;
  char * recibido;
  char * data;

  init_dsr(0, DIR);

  cputs ("ANTES");
  msleep(10);

  recibido = malloc (sizeof(char)*250);

  if (DIR == 0x1){

    //ENVIA
    data = malloc (sizeof(char)*5);
    strcpy (data, "papa");
    tmp = 'C';
    send (tmp, data);

    strcpy (data, "pepe");
```

```
    send (tmp, data);

    strcpy (data, "pipi");
    send (tmp, data);

    strcpy (data, "popo");
    send (tmp, data);

    strcpy (data, "pupu");
    send (tmp, data);

  } else if (DIR == 0x03) {

    //RECIBE
    while (1){
      recv (&tmp, recibido);
    }

  } else

      while (1){
        msleep (10);
      }

  free(recibido);
  free(data);
  exit_dsr();

  return 0;
}
```

El programa está pensado para probarlo en una maqueta compuesta de varios nodos en el que el nodo con la *dirección* "1" es el emisor y envía cadenas de caracteres al nodo "C" (que equivale a la dirección "3").

La constante `DIR` determina la *dirección* del nodo dentro de la red:

```
#define DIR 0x1
```

Si el nodo que ejecuta el código es el emisor (dirección "1"), entrará por la siguiente rama del `if`:

```
  if (DIR == 0x1){
  //ENVIA
    data = malloc (sizeof(char)*5);
    strcpy (data, "papa");
    tmp = 'C';
    send (tmp, data);

    strcpy (data, "pepe");
    send (tmp, data);
```

```
    strcpy (data, "pipi");
    send (tmp, data);

    strcpy (data, "popo");
    send (tmp, data);

    strcpy (data, "pupu");
    send (tmp, data);
 }
```

Si es el nodo destino (el nodo "C"), entonces se quedará a la espera de recibir paquetes:

```
    else if (DIR == 0x03) {
    //RECIBE
    while (1){
      recv (&tmp, recibido);
    }
```

Cualquier otro nodo componente de la red se queda esperando eventos del protocolo:

```
    else
      while (1){
        msleep (10);
      }
```

6.3. Código final

En este apartado se incluye el código fuente final para poner en marcha la práctica propuesta:

6.3.1. Ficheros en el PC

Estos son los ficheros y scripts necesarios para poner en marcha la parte del PC. Basta con descomprimir todos los ficheros (fuentes.tar.gz) en un directorio, y ejecutar `make` para compilarlos.

Para arrancar el servidor, ejecutar `./sioux.py 8080` desde la linea de comandos.

Fichero `sioux.py`

```
#!/usr/bin/python
# $Id: encaminamiento_ad-hoc.tex,v 1.15 2002/02/18 11:07:35 jpl Exp $

from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer

import sys
import string
import RCX
```

```
import cgi
import os

html_pages_dir = "html"

def main():

    if len(sys.argv) < 2:
        print "Usage: " + sys.argv[0] + " <puerto>"
        print
        sys.exit(1)
    else:
        port = string.atoi(sys.argv[1])
        httpd = HTTPServer( ('localhost', port), RCXHTTPHandler)
        print "Sirviendo en puerto %s..." % port
        httpd.serve_forever()

class RCXHTTPHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.do_HEAD()

        if self.response == 404:
            self.wfile.write("404 That action does not exist")
            return

        if self.action == "/" or self.action == "/index.html":
            f = open(html_pages_dir + "/index.html", "r")
        elif self.action == "/info.html":
            f = open(html_pages_dir + "/index.html", "r")
            RCX.Get(self.dicParams["nodo"][0], self.dicParams["action"][0])

        self.wfile.write(f.read())
        f.close()

    def do_HEAD(self):
        if '?' in self.path:
            self.action, params = string.split(self.path, '?')
        else:
            self.action, params = self.path, ''

        self.dicParams = cgi.parse_qs(params)

        if self.action[1:] in os.listdir(html_pages_dir) or self.action == "/":
            self.response = 200
        else:
            self.response = 404

        self.send_response(self.response)
        self.send_header("Content-type", "text/html")
```

```
        self.end_headers()

main()
```

Fichero fuente

Fichero `RCXmodule.c`

```
#include <python2.1/Python.h>
#include "dsr.h"

static PyObject *RCXError;

static PyObject *RCX_Get(self, args);

static PyMethodDef RCXMethods[] = {
  {"Get", RCX_Get, METH_VARARGS},
  {NULL , NULL}
};

static PyObject*
RCX_Get(self, args)
     PyObject *self;
     PyObject *args;
{

  char* nodo;
  char* action;

  PyArg_ParseTuple(args, "ss", &nodo, &action);
  printf("%s -- %s\n", nodo, action);

  send(nodo[0], action);
  return Py_BuildValue("i", 0);
}

void
initRCX()
{
  PyObject *m, *d;

  init_dsr(1, 0x01);

  m = Py_InitModule("RCX", RCXMethods);
  d = PyModule_GetDict(m);
  RCXError = PyErr_NewException("RCX.error", NULL, NULL);
  PyDict_SetItemString(d, "error", RCXError);
}
```

Fichero fuente

Fichero `dsr.h`

```
//#include <lnp.h>
#include "liblnp.h"
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>


#define NUM_HOSTS 10
#define MULTICAST 0x0
#define PORT 0x0
#define MAX_SENDS 3

typedef char   t_path[NUM_HOSTS];
typedef t_path t_route_cache[NUM_HOSTS];

void init_dsr(int tipo, int dir);
void exit_dsr(void);
int send (unsigned char dir, char *data);
int recv (unsigned char dir, char *data);
```

Fichero fuente

Fichero `dsr.c`

```
#include "dsr.h"

char*         last_id_rreq;
t_route_cache route_cache;
int           host;
int           address;
int           recv_rreq = 0;
char*         data_rreq;
unsigned char leng_rreq;
int           recv_add = 0;
char*         data_recv;
unsigned char leng_recv;
unsigned char src_recv;
int           done = 0;
char*         route;
int           recv_packet = 0;
char*         data_packet;
char          dir_packet;
int           recv_ack = 0;
char          id_rreq = 20;
int           num_sends = 0;

int
get_host (unsigned char add )
```

```
{
  return ((add & 0xFC) >> 2) + 64;
}

unsigned char
get_dir (int host)
{
  return ((unsigned char)((host -64) << 2));
}

unsigned char
prev_node (const unsigned char* data, char host)
{
  unsigned char prev;
  int i;

  i= 0;

  do {
    i++;
  } while ((data[i] != host) && (i<=strlen(data)));

  if (i>strlen(data))
    prev = 0;
  else
    prev = data[i-1];

  return (prev);
}

unsigned char
next_node (const unsigned char* data, char host) {

  unsigned char next;
  int i;

  i=0;

  do {
    i++;
  } while ((data[i] !=host) && (i<= strlen(data)));

  if (i> strlen(data))
    next = 0;
  else
    next = data[i+1];

  return (next);
}
```

```
int
ind (char *data, char car) {
  int resul;
  int i=0;
  int exit=0;

  while (i<=strlen(data) && !exit) {
    if (car == data[i])
      exit=1;
    else
      i++;
  }
  if (exit)
    resul = i;
  else
    resul = -1;

  return resul;

}

void
concat (unsigned char *dest, unsigned char *src) {
  int num, i;

  num= strlen(dest);
  for (i=num; i<(num+strlen(src)); i++){
    dest[i] = src[i-num];
  }
  dest[i] = '\0';
}

int
send_route_req(char* add)
{
  char* the_address;
  char car;

  car = add[strlen(add) - 1];

  the_address = malloc(strlen(add) - 1);
  memcpy(the_address, add, strlen(add));

  the_address[strlen(add) - 1] = (char)get_host((unsigned)address);
  the_address[strlen(add)] = car;
  the_address[strlen(add) + 1] = '\0';

  lnp_integrity_write(the_address, strlen(the_address));

  printf("SRREQ: %s\n", the_address);
```

```
  return 0;
}

void
recv_route_req(const unsigned char* d, unsigned char leng)
{
  recv_rreq = 1;
  data_rreq = malloc(leng + 1);
  memcpy(data_rreq, d, leng);
  data_rreq[(int)leng] = '\0';

  printf("RRREQ\n");
}

void*
test_recv_route_req(void *args)
{
  unsigned char car;
  unsigned char packet[255];
  char* data;

  while(!done) {
    while(!recv_rreq && !done)
      usleep(10);

    if (!done) {
      if ((data_rreq[0] > last_id_rreq[((int)data_rreq[1]) - 65]) &&
          (data_rreq[1] != (char)get_host((unsigned)address))) {
        last_id_rreq[((int)data_rreq[1]) - 65] = data_rreq[0];
        if (((char)get_host((unsigned)address)) != (char)(data_rreq[((int)leng_rreq) - 1])
          car = data_rreq[leng_rreq - 1];
          data = malloc(leng_rreq + 1);
          memcpy(data, data_rreq, leng_rreq);
          data[leng_rreq] = '\0';
          printf("RREQ\n");

          sleep(random() % 4);
          send_route_req(data);
        } else {
          car = data_rreq[(char)((int)leng_rreq) - 2];
          packet[0] = '1';
          packet[1] = '1';
          packet[2] = '\0';
          concat(packet, data_rreq + 1);

          lnp_addressing_write(packet, strlen(packet), get_dir(car), PORT);
          printf("SRREP\n");
        }
      } else
        printf("IDRREQ\n");
```

```
      recv_rreq = 0;
    }
  }

  return 0;
}

void
recv_address(const unsigned char* data, unsigned char leng, unsigned char src)
{
  recv_add = 1;

  data_recv = malloc(leng + 1);
  memcpy(data_recv, data, leng);
  data_recv[(int)leng] = '\0';

  leng_recv = leng;

  src_recv = src;
  printf("RDIR\n");
}

void*
test_recv(void *args)
{
  unsigned char prev = 0;
  unsigned char next;
  unsigned char packet[255];
  char car;

  while (!done) {
    while (!recv_add &&  !done)
      usleep(10);

    if (!done) {
      switch(data_recv[0]) {
      case '1':
        switch(data_recv[1]) {
        case '1':
          if (data_recv[2] == ((char)get_host((unsigned char) address))) {
            strcpy(route_cache[((int)data_recv[leng_recv - 1]) - 64], data_recv + 2);
            printf("GUARDA CACHE[%d]: %s\n", ((int)data_recv[leng_recv - 1]) - 64, route_c
            memcpy(route, data_recv + 2, leng_recv - 2);
          } else {
            car = (char)get_host((unsigned)address);
            if (prev == prev_node(data_recv, car)) {
              printf("RRREP\n");
              lnp_addressing_write(data_recv, strlen(data_recv), get_dir(prev), PORT);
              printf("ERR RECIBIR\n");
```

```
            }
          }
          break;
        case '2':
          if (data_recv[leng_recv - 1] == ((char)get_host((unsigned char)address))) {
            printf("RRERROR\n");
            route_cache[((int)data_recv[2]) - 64][0] = '\0';
          } else {
            next = next_node(data_recv, ((char)get_host((unsigned char)address)));
            lnp_addressing_write(data_recv, strlen(data_recv), get_dir(next), PORT);
          }
          printf("ERROR\n");
          break;
        case '3': // Route ACK
          recv_ack = 1;
          printf ("RECV ACK\n");
          break;
        }
        break;
      case '2':
        packet[0] = '1';
        packet[1] = '3';
        packet[2] = '\0';
        printf("SACK\n");
        prev = prev_node(data_recv, (char)get_host((unsigned)address));
        lnp_addressing_write(packet, strlen(packet), get_dir(prev), PORT);

        if (data_recv[ind(data_recv, '-') - 1] == ((char)get_host((unsigned char)address))
          recv_packet = 1;
          dir_packet = data_recv[1];
          data_packet = malloc(sizeof(char)*(leng_recv - (ind(data_recv, '-'))));
          strcpy(data_packet, data_recv + (ind(data_recv, '-')) + 1);
          data_packet[strlen(data_packet)] = '\0';
          printf("%s\n", data_packet);
        } else {
          next = next_node(data_recv, (char)get_host((unsigned char)address));
          lnp_addressing_write(data_recv, strlen(data_recv), get_dir(next), PORT);
          recv_ack = 0;
          printf("REENVIO\n");
        }
        break;
      default:
        printf("ERR RECIBIR\n");
      }
      recv_add = 0;
    }
  }

  return 0;
}
```

```
void
init_dsr (int tipo, int dir)
{
  int i;
  pthread_t route_req;
  pthread_t recv;

  last_id_rreq = malloc(sizeof(char) * (NUM_HOSTS + 1));

  memset(last_id_rreq, 0, NUM_HOSTS + 1);

  for (i = 0; i < NUM_HOSTS; i++)
    route_cache[i][0] = '\0';

  host = dir;
  address = (MULTICAST << 6 | (host << 2)) | PORT;

  printf ("Yo soy: %d\n", address);

  if (tipo == 1)
    if (lnp_init(0, 0, address, 0xfc, 0))
    //if (lnp_init(0, 0, 0, 0, 0))
      exit(1);

  srandom(address);

  lnp_integrity_set_handler(recv_route_req);
  lnp_addressing_set_handler(PORT, recv_address);

  pthread_create(&route_req, NULL, test_recv_route_req, NULL);
  pthread_create(&recv, NULL, test_recv, NULL);
}

int
send_packet(char* data)
{
  int i;
  char packet[250];
  unsigned char next;
  unsigned int tiempo_restante;
   num_sends++;
   if (num_sends > MAX_SENDS) {
     for(i = 0; i < NUM_HOSTS; i++)
       route_cache[i][0] = '\0';
     num_sends = 0;
   }

  packet[0] = '2';
  packet[1] = '\0';
```

```
  concat(packet, route);
  packet[strlen(route) + 1] = '-';
  packet[strlen(route) + 2] = '\0';
  concat(packet, data);
  printf("%s\n", packet);

  next = next_node(packet, (char)get_host((unsigned)address));
  lnp_addressing_write(packet, strlen(packet), get_dir(next), PORT);

  tiempo_restante = sleep(1);
  while (tiempo_restante && !recv_ack)
    tiempo_restante = sleep (tiempo_restante);
  if (!recv_ack) {
    printf("RERROR\n");
    route_cache[((int)route[strlen(route) - 1]) - 64][0] = '\0';
  }

  recv_ack = 0;

  for (i = 0; i < (strlen(route)); i++)
    route[i] = 0;
  route[strlen(route)] = '\0';

  return 0;
}

int
send(unsigned char dir, char* data)
{
  int i;
  char* d;
  int num_send = 4;
  int exit = 0;
  unsigned int tiempo_restante;

  route = malloc(sizeof(char)*(NUM_HOSTS + 1));
  for (i = 0; i < NUM_HOSTS; i++)
    route[i] = 0;
  route[NUM_HOSTS] = '\0';

  if (id_rreq > 250)
    id_rreq = 20;
  else
    id_rreq++;

  d = malloc(2 * sizeof(char));

  d[0] = id_rreq;
  d[1] = (char)dir;
```

```
  d[2] = '\0';


  printf("CACHE[%d]: %s\n", dir - 64, route_cache[dir - 64]);
  while ((num_send < 9) && (!exit)) {
    if (route_cache[dir - 64][0] != '\0') {
      printf ("CACHE\n");
      strcpy(route, route_cache[dir - 64]);
    } else
      send_route_req(d);

    id_rreq++;
    d[0] = id_rreq;
    num_send = num_send * 2;

    printf("ROUTE: %s\n", route);
    if (route[0] != 0) {
      send_packet(data);
      exit = 1;
    } else {
      printf("A DORMIR\n");
      tiempo_restante = sleep(num_send);
      while (tiempo_restante && route[0] == 0)
        tiempo_restante = sleep(tiempo_restante);
    }
  }

  printf("MANDAO\n");

  return 0;
}

void
exit_dsr()
{
  done = 1;
}
```

Fichero fuente

Fichero `index.html`

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>Encaminamiento Ad-Hoc para Legos MindStorm</title>
  </head>

  <body bgcolor="#ffffff">
    <br>
```

```
    <center>
       <h1>
          Bienvenido a Encaminamiento Ad-Hoc para Legos MindStorm
       </h1>
    </center><hr><br>

<form method="get" action="info.html" enctype="text/plain">
    <center>
      <table border="3">
        <tr>
          <td>
            <table cellspacing="20">
              <tr>
                <td>
                  <center>
                  Selecciona el nodo del cual quieres obtener información:
                  <select name="nodo">
                    <option>B
                    <option>C
                    <option>D
                    <option>E
                    <option>F
                    <option>G
                    <option>H
                    <option>I
                    <option>J
                    <option>K
                  </select>
                  </center>
                </td>
              </tr>
              <tr>
                <td>
                  <center>
                  Selecciona la acción en el nodo:
                  <select name="action">
                    <option>DER
                    <option>IZQ
                    <option>DEL
                    <option>ATR
                  </select>
                  </center>
                </td>
              </tr>
              <tr>
                <td>
                  <center>
                    <input value="Enviar petición" type="submit">
                  </center>
                </td>
```

```
            </tr>
          </table>
        </td>
      </tr>
    </table>
  </center>
</form>


  <br><hr>
  <address>
    <a href="mailto:dchaparro@acm.org">Diego Chaparro</a>  
    <a href="mailto:jpelegri@gsyc.escet.urjc.es">Jose Pelegrín</a>  
    <a href="mailto:rrodrigu@gsyc.escet.urjc.es">Raúl Rodríguez</a>  
  </address>
<!-- Created: Sun Oct 28 13:20:08 CET 2001 -->
<!-- hhmts start -->
Last modified: Sat Feb  9 22:28:53 CET 2002
<!-- hhmts end -->
</body>
</html>
```

Fichero fuente

Ficheros para la compilación

- Makefile
- Makefile.pre
- Makefile.pre.in
- Setup
- Setup.in
- sedscript
- config.c

6.3.2. Ficheros en los *Legos*

Esto son los ficheros necesarios para ejecutar en cada *Lego MindStorm* la aplicación sobre el *DSR*. Descomprime los ficheros fuente (fuentes-rcx.tar.gz) en un directorio y ejecuta `make`.

A continuación sigue el procedimiento normal para descargar programas en los *RCX* con el comando `dll`.

Fichero `dsr.h`

```
#include <lnp.h>
#include <stdlib.h>
#include <string.h>
```

```
// PRINCIPALES FUNCIONES DEL DSR



// Función utilizada para mandar paquetes de datos.
int send (unsigned char dir, char *data);

// Función utilizada para recibir paquetes de datos.
int recv (unsigned char *dir, char *data);

// Iniciar el protocolo para poder enviar/recibir datos
// Se le pasa el tipo de máquina en la que se usará:
// 1-> PC    2-> Legos
void init_dsr (int tipo, int dir);

// Terminar el protocolo
void exit_dsr (void);
```

dsr.h

Fichero `dsr.c`

```
#include <dsr.h>
#include <dsr_aux.h>
#include <conio.h>
#include <unistd.h>
#include <semaphore.h>
#include <stdlib.h>


// Número de nodos que habrá en la red
#define NUM_HOSTS 10

// DIRECCION DE CADA NODO
// Dos bits para multicast (0, 1, 2, 3)
#define MULTICAST 0x0
// Cuatros bits para dirección de máquina (0, 1, 2, ..., 14, 15)
// Haremos que empiece por 1, para algunos controles de error en funciones.
//#define HOST 0x1
// Dos bits para el puerto (0, 1, 2, 3)
#define PORT 0x0
// Dirección
//#define ADDRESS ( MULTICAST << 6 | ( HOST << 2)) | PORT

int ADDRESS;
int HOST;

// Variable para terminar la ejecución de tareas, handlers...
int done = 0;
```

```
// Tipo ruta
typedef char t_path[NUM_HOSTS];

// Tipo Route cache
typedef t_path t_route_cache[NUM_HOSTS];

// Cache para guardar las rutas
t_route_cache route_cache;

// Variable que servirá para calcular el tiempo que durarán las rutas en la cache
// Cada vez que haga un envío se incrementará, y cuando llegue a un umbral
// se borrarán las cachés.
int num_sends = 0;
#define MAX_SENDS 3

// Ruta obtenida del rreq
char *route;
// Semáforo para cuando envío un paquete de datos.
// Para que no se modifique la ruta desde que recibo el rrep
// hasta que lo envío.
sem_t sem_env_datos;

// Lista para guardar el último id_rreq recibido de cada nodo
char *last_id_rreq;

// Identificador de los rreq
// Valores válidos 20-250 para facilitar la impresión en pantalla
char id_rreq = 20;

// Variables globales para la tarea que procesa los rreq
int recv_rreq = 0;
char *data_rreq;
unsigned char leng_rreq;
// Semáforo para sincronizar el manejador que recibe el rreq con la
// tarea que lo procesa
sem_t sem_rec_rreq;

// Variables globales para la tarea que procesa mensajes con dirección
int recv_add = 0;
char *data_recv;
unsigned char leng_recv;
unsigned char src_recv;
// Semáforo para sincronizar el manejador que recibe paquetes con dirección
// con la tarea que los procesa
sem_t sem_rec_dir;

// Variable global para saber si me ha llegado un mensaje de datos
int recv_packet = 0;
char *data_packet;
```

```
char dir_packet;
// Semáforo para cuando recibo un paquete de datos.
sem_t sem_rec_datos;

// Variable para comprobar si recibo un ack
int recv_ack = 0;

// Mandar route request para saber la ruta hacia un nodo.
int
send_route_req ( char *add) {

  char *address;
  char car;

  lcd_clear();

  car = add[strlen(add)-1];

  address = malloc (strlen(add)+1);
  memcpy (address, add, strlen(add));

  // Meto mi dirección en la penúltima posición

  address[strlen(add)-1] = (char)get_host((unsigned)ADDRESS);
  address[strlen(add)] = car;
  address[strlen(add)+1] = '\0';

  // Mando el rreq
  lnp_integrity_write (address, strlen(address));

  cputs ("Srreq");
  return 0;
}


// Tarea que comprueba si se reciben paquetes con direcciones

int
test_recv (int argc, char *argv[]){
  unsigned char prev, next;
  unsigned char packet[255];
  int result, j, i;
  char car;

  while (!done) {
    while (!recv_add && !done) {
      msleep(10);
    }
    if (!done) {
      lcd_clear();
```

```
switch (data_recv[0]) {

  case '1': // Paquete del protocolo DSR
    switch (data_recv[1]) {
      case '1': // Route Reply
        if (data_recv[2] == ((char)get_host((unsigned char) ADDRESS))) {
          // Soy el que inicio el route request.
          // Debo guardar la ruta
          strcpy (route_cache[((int)data_recv[leng_recv-1])-64], data_recv+2);
          //cputs (data_recv);
          cputs (route_cache[((int)data_recv[leng_recv-1])-64]);

          //route = malloc (sizeof(char) * (strlen(data_recv)-2));
          memcpy (route, data_recv+2, leng_recv-2);
        } else {
          car = (char) get_host((unsigned)ADDRESS);
          if (prev= prev_node (data_recv, car)) {
            cputs ("RRREP");

            lnp_addressing_write (data_recv, strlen(data_recv), get_dir(prev), PORT)
          } else
            // El nodo actual no está en la lista
            cputs ("ERR RECIBIR");
        }
        break;
      case '2': // Route Error

        if (data_recv[leng_recv-1] == ((char)get_host((unsigned char) ADDRESS))){
          // Soy yo el destinatario del paquete
          // Borrar la ruta del nodo origen
          cputs ("RRERROR");
          route_cache[((int)data_recv[2])-64][0] = '\0';
        } else {
          // No soy yo el destinatario, se lo envio al siguiente en la lista
          next = next_node (data_recv, ((char)get_host((unsigned char) ADDRESS)));
          lnp_addressing_write (data_recv, strlen(data_recv), get_dir(next), PORT);
        }

        cputs ("ERROR");
        break;
      case '3': // Route ACK
        recv_ack = 1;
        cputs ("ACK");
        break;
    }
    break;
  case '2': // Paquete de datos

    packet[0] = '1';
    packet[1] = '3';
```

```
        packet[2] = '\0';
        cputs ("SACK");
        prev = prev_node (data_recv, (char) get_host((unsigned) ADDRESS));
        // Envío el ack al nodo que me envía el paquete
        lnp_addressing_write (packet, strlen(packet), get_dir(prev), PORT);

        if (data_recv[ind(data_recv,'-')-1] == ((char)get_host((unsigned char) ADDRESS))
          // El paquete va destinado a mi
          // Guardo los datos
          //sem_wait (&sem_rec_datos);
          recv_packet = 1;
          dir_packet = data_recv[1];
          data_packet = malloc (sizeof(char)*(leng_recv-(ind(data_recv, '-'))));
          //memcpy (data_packet, data_recv+(ind(data_recv,'-'))+1, (leng_recv-(ind(data_
          strcpy (data_packet, data_recv+(ind(data_recv,'-'))+1);
          data_packet[strlen(data_packet)]='\0';
          lcd_clear();
          cputs (data_packet);
        } else {
          // Reenvío el paquete al siguiente en la lista
          next = next_node (data_recv, (char)get_host((unsigned char) ADDRESS));
          lnp_addressing_write (data_recv, strlen(data_recv), get_dir(next), PORT);

          // Este sería el control para el envío de RERR, pero no es posible hacerlo aqu
          // porque esta misma tarea es la encargada de recibir los ACK, y no puede hace
          // dos cosas a la vez :-((
          //sleep (2);
          //if ((!recv_ack) ) { //Debo enviar un RERROR
          //  packet[0] = '1';
          //  packet[1] = '2';
          //  j = 2;
          //  for (i=ind(data_recv,'-')-1; i>0; i--)
          //  {
          //    packet[j] = data_recv[i];
          //    j++;
          // }
          //  packet[j] = '\0';
          //  next = next_node (packet, (char)get_host((unsigned)ADDRESS));
          //  lnp_addressing_write (packet, strlen(packet), get_dir(next), PORT);
          //  lcd_clear();
          //  cputs ("SRERROR");
          //  sleep (10);
          //}
          recv_ack=0;

          cputs ("REENVIO");
        }
        //cputs ("DATOS");
        break;
      default:
```

```
          cputs ("ERR RECIBIR");
      }
      recv_add = 0;
      //sem_post (&sem_rec_dir);
    }
  }
  return 0;
}

// Manejador para recibir paquetes con direcciones
void
recv_address (const unsigned char* data, unsigned char leng, unsigned char src)
{
  //sem_wait (&sem_rec_dir);
  recv_add = 1;

  data_recv = malloc (leng+1);
  memcpy (data_recv, data, leng);
  data_recv[(int)leng] = '\0';

  leng_recv = leng;

  src_recv = src;
  lcd_clear();
  cputs ("RDIR");
}

// Tarea que comprueba si se reciben paquetes broadcast, y hace su trabajo...
int
test_recv_route_req (int argc, char *argv[]){

  unsigned char car;
  unsigned char packet[255];
  signed char result;
  char *data;

  while (!done) {
    while (!recv_rreq && !done){
      msleep(10);
    }
    if (!done) {
      // Para evitar mandar los rreq dos veces
      if ((data_rreq[0] > last_id_rreq[((int)data_rreq[1])-65]) && (data_rreq[1] != (char)
        // Incremento el número de id_rreq para el nodo origen
        last_id_rreq[((int)data_rreq[1])-65] = data_rreq[0];
        if (((char)get_host((unsigned)ADDRESS)) != (char)(data_rreq[((int)leng_rreq)-1]))
          // Reenvío el route_req

          car = data_rreq[leng_rreq-1];
```

```
          data = malloc (leng_rreq+1);
          memcpy (data, data_rreq, leng_rreq);
          data[leng_rreq] = '\0';
          cputs ("RRREQ");

          // Espero un tiempo aleatorio para enviar
          // No debe ser demasiado bajo, porque se reenvían los rreq antes de que llegue e
          // y se hacen un lío
          sleep (random() % 4);
          send_route_req (data);

        } else {
          // Las direcciones son iguales.
          car = data_rreq[(char)((int)leng_rreq)-2];
          packet[0] = '1';
          // Tipo de mensaje: route_rep
          packet[1] = '1';
          packet[2] = '\0';
          // Y después concateno la lista de nodos
          concat (packet, data_rreq+1);
          //cputc ((strlen(packet)+48), 4);

          lnp_addressing_write (packet, strlen(packet), get_dir(car), PORT);
          cputs ("SRREP");
          // Modificaciónnnnnn
          cputc (car, 4);
        }
      } else
        cputs ("idrreq");
      recv_rreq = 0;
      //sem_post (&sem_rec_rreq);
    }
  }
return 0;
}

// Manejador para recibir paquetes broadcast
void
recv_route_req (const unsigned char *d, unsigned char leng) {

  //sem_wait (&sem_rec_rreq);
  recv_rreq = 1;
  data_rreq = malloc (leng+1);
  memcpy (data_rreq, d, leng);
  data_rreq[(int)leng] = '\0';
  leng_rreq = leng;
}

// Manda un paquete y recibe la respuesta
int
```

```
send_packet (char *data) {

  int i;
  char packet[250];
  unsigned char next;

  // incremento la variable que servirá para borrar las caches
  num_sends++;
  if (num_sends > MAX_SENDS){
    // Inicializo la cache de rutas
    for (i=0; i<NUM_HOSTS; i++)
      route_cache[i][0]='\0';
    num_sends = 0;
  }

  packet[0] = '2';
  packet[1] = '\0';

  concat (packet, route);
  packet [strlen(route)+1]= '-';
  packet [strlen(route)+2]= '\0';
  concat (packet, data);
  //cputs (packet);

  next = next_node (packet, (char)get_host((unsigned)ADDRESS));
  lnp_addressing_write (packet, strlen(packet), get_dir(next), PORT);

  sleep (1);
  if (!recv_ack) {
    cputs ("RERROR");
    route_cache[((int)route[strlen(route)-1])-64][0] = '\0';
  }
  recv_ack=0;

  for (i=0; i<(strlen(route)); i++)
    route[i] = 0;
  route[strlen(route)]= '\0';

  return 0;
}

// Función utilizada para mandar paquetes de datos.
int
send (unsigned char dir, char *data) {

  int i;
  char *d;
  int num_send = 4;
  int exit = 0;
```

```
  route = malloc (sizeof(char)*(NUM_HOSTS+1));
  for (i=0; i<NUM_HOSTS; i++)
    route[i] = 0;
  route[NUM_HOSTS] = '\0';

  if (id_rreq >250)
    id_rreq = 20;
  else
    id_rreq++;

  d = malloc (3*sizeof(char));

  d[0]= id_rreq;
  d[1]= (char) dir;
  d[2]= '\0';

  // Pongo el semáforo para que no se modifique la ruta que recibo
  //sem_wait(&sem_env_datos);
  while ((num_send<17) && (!exit)){

    if (route_cache[dir-64][0] != '\0'){
      cputs ("CACHE");
      strcpy (route, route_cache[dir-64]);
    } else
      send_route_req (d);
    id_rreq++;
    d[0]= id_rreq;
    num_send = num_send * 2;
    if (route[0] != 0) {
      // Mando paquete
      send_packet(data);
      exit = 1;
    } else
      sleep (num_send);
  }
  // Libero el semáforo porque ya he enviado el paquete, y ya puedo recibir otro rreq
  //sem_post (&sem_env_datos);

  cputs ("MANDADO");
  //if (!exit)
    // Error, no se ha podido enviar

  return 0;
}

// Función para recibir paquetes de datos.
int
recv (unsigned char *dir, char *data) {

  while (!recv_packet)
```

```
    msleep (10);

  //data = malloc (sizeof(char) * (strlen(data_packet)+1));
  memcpy (data, data_packet, strlen(data_packet));
  data[strlen(data_packet)] = '\0';
  *dir = dir_packet;

  lcd_clear();
  cputs (data_packet);

  recv_packet = 0;
  //sem_post (&sem_rec_datos);

  return 0;
}

// Iniciar el protocolo para poder enviar/recibir datos
void
init_dsr (int tipo, int dir) {

  int i;

  // Inicializacion de semaforos
  //sem_init (&sem_env_datos, 0, 1);
  //sem_init (&sem_rec_rreq, 0, 1);
  //sem_init (&sem_rec_dir, 0, 1);
  //sem_init (&sem_rec_datos, 0, 1);

  // Inicializo la lista de últimos identificadores de paquetes rreq recibidos
  last_id_rreq = malloc (sizeof(char)*(NUM_HOSTS+1));
  for (i=0; i<=NUM_HOSTS; i++)
    last_id_rreq[i] = 0;
  last_id_rreq[NUM_HOSTS] = '\0';

  // Inicializo la cache de rutas
  for (i=0; i<NUM_HOSTS; i++)
    route_cache[i][0]='\0';


  // Inicializo lnp
  if (tipo = 1)
    if ( lnp_init(0,0,dir ,0x03,0) )
    {
//      perror("lnp_init");
      exit(1);
    }

  HOST = dir;

  ADDRESS = (MULTICAST << 6 | (HOST <<2)) | PORT;
```

```
  // Inicializo los número aleatorios,
  // Uso seed= ADDRESS para que cada nodo se inicialice con un número distinto
  srandom (ADDRESS);


  // Inicializo manejador de paquetes broadcast
  lnp_integrity_set_handler (recv_route_req);

  // Inicializo el manejador para paquetes con dirección
  lnp_addressing_set_handler (PORT, recv_address);

  // Ejecuto las dos tareas que procesarán los paquetes recibidos
  execi (test_recv_route_req, 0, NULL, PRIO_NORMAL, DEFAULT_STACK_SIZE);
  execi (test_recv, 0, NULL, PRIO_NORMAL, DEFAULT_STACK_SIZE);
}

// Finalizar el protocolo
void
exit_dsr() {

  done = 1;
  lnp_addressing_set_handler(PORT,LNP_DUMMY_ADDRESSING);
}
```

dsr.c

Fichero `dsr_aux.h`

```
// ******************************************************************************
// ******************** FUNCIONES AUXILIARES ************************************
// ******************************************************************************

// Concatena varios strings. No valen caracteres.
void
concat (unsigned char *dest, unsigned char *src) {
  int num, i;

  num= strlen(dest);
  for (i=num; i<(num+strlen(src)); i++){
    dest[i] = src[i-num];
  }
  dest[i] = '\0';
}

// Devuelve el nombre del host (la letra)
int
get_host (unsigned char add ) {
  // Le sumo 64 para que empiecen en la A, B, ...
```

```
  //return ((add & 0x3C) >> 2 ) + 64;
  return ((add & 0xFC) >> 2) + 64;
}

// Devuelve la dirección del host a partir del nombre
unsigned char
get_dir (int host) {
        return ((unsigned char)((host -64) << 2));
}

// Se le pasa una cadena de nodos, y devuelve el anterior al nodo actual.
// Si el nodo que procesa la siguiente cadena: ABCDE es C, esta función debe
// devolver el nodo B
unsigned char
prev_node (const unsigned char* data, char host) {

  unsigned char prev;
  int i;

  i= 0;

  do {
    i++;
  } while ((data[i] != host) && (i<=strlen(data)));

  if (i>strlen(data))
    prev = 0;
  else
    prev = data[i-1];

  return (prev);
}

// Se le pasa una cadena de nodos, y devuelve el siguiente al actual.
unsigned char
next_node (const unsigned char* data, char host) {

  unsigned char next;
  int i;

  i=0;

  do {
    i++;
  } while ((data[i] !=host) && (i<= strlen(data)));

  if (i> strlen(data))
    next = 0;
  else
    next = data[i+1];
```

```
  return (next);
}

int
ind (char *data, char car) {
  int resul;
  int i=0;
  int exit=0;

  while (i<=strlen(data) && !exit) {
    if (car == data[i])
      exit=1;
    else
      i++;
  }
  if (exit)
    resul = i;
  else
    resul = -1;

  return resul;

}

// ********************************************************************************************
```

dsr_aux.c

Fichero `pru_dsr.c`

```
#include <unistd.h>
#include <conio.h>
#include <dsr.h>
#include <dsound.h>

#define DIR 0x5

static const note_t devil[] = {
  { PITCH_G4, QUARTER },
  { PITCH_G4, QUARTER },
  { PITCH_G4, QUARTER },
  { PITCH_G4, QUARTER },
  { PITCH_G4, HALF },
  { PITCH_G4, HALF },

  { PITCH_G4, HALF },
  { PITCH_G4, HALF },
  { PITCH_G4, HALF },
  { PITCH_G4, HALF },
```

```
  { PITCH_F4, QUARTER },
  { PITCH_F4, QUARTER },
  { PITCH_F4, QUARTER },
  { PITCH_F4, QUARTER },
  { PITCH_F4, HALF },

  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_E4, QUARTER },
  { PITCH_F4, HALF },
  { PITCH_F4, HALF },

  { PITCH_E4, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_PAUSE, HALF },
  { PITCH_END, 0 }
};



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

  unsigned char tmp;
  char * recibido;
  char * data;

  dsound_set_duration(40);

  init_dsr(0, DIR);

  cputs ("ANTES");
  msleep(10);

  recibido = malloc (sizeof(char)*250);

  if (DIR == 0x1){
  //ENVIA
    data = malloc (sizeof(char)*5);
    strcpy (data, "papa");
    tmp = 'B';
    send (tmp, data);
    strcpy (data, "pepe");
    tmp = 'C';
    send (tmp, data);
    strcpy (data, "pipi");
    tmp = 'D';
    send (tmp, data);
```

```
    strcpy (data, "popo");
    tmp = 'E';
    send (tmp, data);
    strcpy (data, "pupu");
    tmp = 'B';
    send (tmp, data);
    //recv (&tmp, recibido);
  } else {
    //RECIBE
    while (1){
      recv (&tmp, recibido);
      dsound_play(devil);
      //wait_event(dsound_finished,0);
      //recv (&tmp, recibido);
      //free (data);
      //data = malloc (sizeof(char)*5);
      //strcpy (data, "HIJO");

      //send (tmp, data);
    }
  }// else
   //   while (1){
   //     msleep (10);
   //   }


  //lcd_clear();
  //cputs (recibido);

  free(recibido);
  free(data);
  exit_dsr();
//  while (1) {
//        cputs ("BIEN");
//  }


  return 0;
}
```

pru_dsr.c

Ficheros para la compilación

- Makefile

Capítulo 7

Otros comentarios

La gestión de errores no ha sido implementada completamente por falta de tiempo. Simplemente se comprueba si ha habido un cambio de ruta en el primer salto al enviar un paquete, es decir, si el nodo A quiere enviar un paquete al nodo C por la ruta ABC, entonces A envía el paquete de datos a B, pero si B no responde con un ACK, A se da cuenta, borra su ruta e inicia un descubrimiento de ruta. Esta misma acción debería realizarse en el resto de saltos por los que pasa el paquete de datos, pero todavía no está implementado.

Para evitar problemas con rutas que cambian y que los nodos no se den cuenta, hemos puesto un contador, y cada 4 paquetes que manda un nodo borra su caché de rutas.