PROYECTO
SISTEMAS INFORMÁTICOS

HERRAMIENTA PARA SIMULAR Y EVALUAR LA INTERCONEXIÓN DE REDES CON IP
II – MOTOR DEL SIMULADOR

Alumno: José María Díaz Górriz
Tutores: D. Francisco A. Candelas Herías
D. Fernando Torres Medina

Dpto. de Física, Ingeniería de Sistemas y Teoría de la Señal
Escuela Politécnica Superior
Universidad de Alicante

Curso 2003/04

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE LISTADOS DE CÓDIGO

1. Introducción

El proyecto consiste en la creación de un sistema capaz de crear y posteriormente simular el comportamiento de redes IP. Debido a la complejidad del sistema a implementar se ha dividido en dos partes muy bien diferenciadas: una interfaz gráfica con la que crear fácilmente las redes y una interfaz de programación con la que simular todos los eventos que se puedan producir en una red IP. Esta documentación pertenece a la interfaz de programación.

Dada la gran cantidad de eventos y subsistemas que existen en una red IP, no es viable en un espacio de tiempo pequeño implementar un sistema que simule todo de forma exhaustiva. Después de un pequeño análisis previo se ha decidido implementar las siguientes funcionalidades:

- Pila de comunicaciones con los niveles de enlace y red.
- Protocolos de nivel de red soportados: ARP, IPv4 e ICMP.
- Equipos generadores de datos: Ordenadores y Enrutadores.
- Equipamiento de red: Redes Ethernet (en bus, hubs, switches, puentes) y Redes punto a punto.
- Simulación de errores.
- Facilidad de ampliación mediante la inclusión de nuevos tipos de red, protocolos y equipos en tiempo de ejecución sin necesidad de recompilar todo el sistema.
- El sistema debe poder ser ejecutado en distintos sistemas operativos: GNU/Linux, Windows, Solaris…

El lenguaje elegido para llevar a cabo el proyecto, es Java, ya que permite el desarrollo rápido de aplicaciones, basadas en la orientación a objetos, lo que nos permite conseguir una alta modularidad y facilidad de ampliación, además de ser totalmente multiplataforma.

2. Arquitectura del sistema

Una vez realizado un análisis en profundidad nos damos cuenta de que existen varias partes bien diferenciadas: la gestión de eventos, la pila de comunicaciones, el equipamiento de red y los equipos generadores de datos. La arquitectura es bastante simple, a la vez que muy efectiva y fácil de implementar.

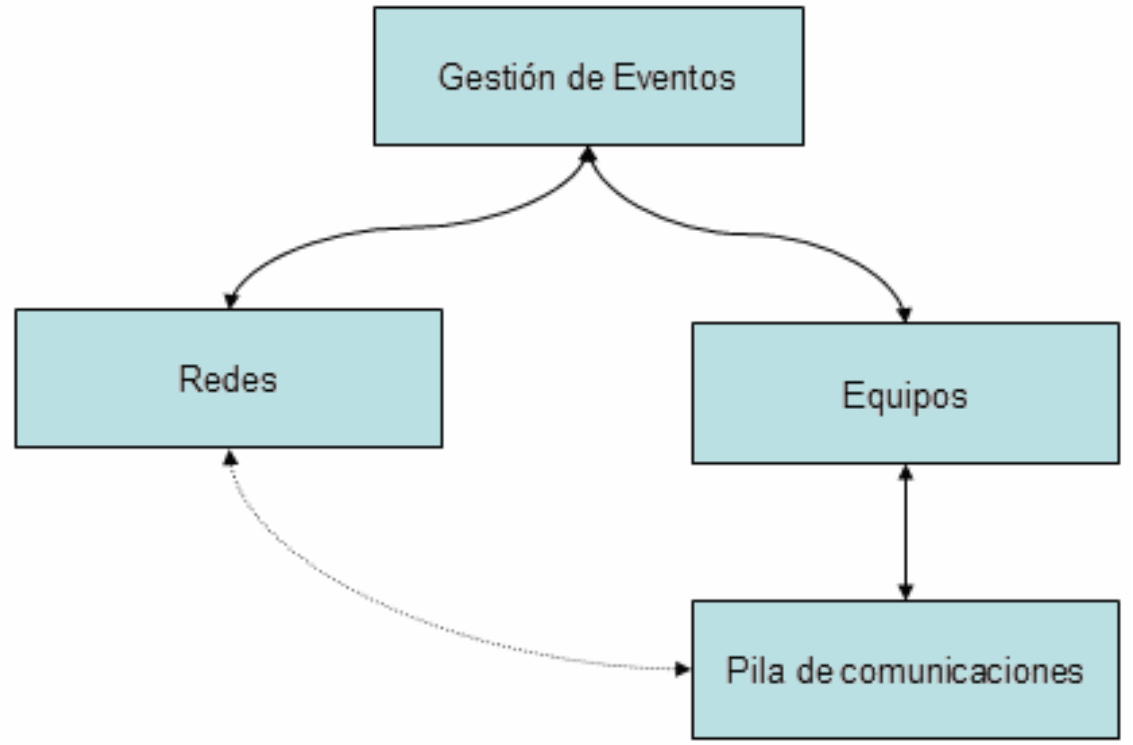

Figura 1. Esquema de bloques del sistema de simulación

El **sistema gestor de eventos** es el encargado de coordinar la evolución de cada uno de los demás componentes del sistema: las redes y los equipos. Lleva el control del tiempo, y va indicando en cada instante a cada componente del sistema que debe avanzar al siguiente estado. Cuando, en un determinado instante, en una red o en un equipo se produce un evento que sea de interés para el usuario, se registra para que pueda ser consultado con posterioridad. Cada uno de los eventos contiene información sobre el instante en que se produjo, el paquete de datos que provocó el evento, un mensaje descriptivo y un indicador de si el paquete de datos entraba, salía o provocó un error en la red o equipo. Estos eventos se registran en el componente que los genera. Se podría decir que el sistema gestor de eventos actúa de forma similar a un corazón en el sistema.

Las **redes** son los componentes encargados de hacer fluir entre los equipos los paquetes de datos que harán que el sistema evolucione de una u otra manera. Este comportamiento es el mismo que el de las redes reales. Cuando a una red le llega un paquete de datos, lo analiza, extrae la dirección de destino y de acuerdo con su comportamiento interno la manda a los equipos que corresponda.

Los **equipos** son, probablemente, uno de los componentes más importantes del sistema de simulación ya que son los que generan los paquetes de datos que provocarán que el sistema evolucione de una u otra forma. Además los equipos actúan como registros de eventos y tienen una pila de comunicaciones.

La **pila de comunicaciones** es el núcleo del sistema. Está formada por varios módulos, dependiendo del equipo en el que se incluya. Es la encargada de procesar los distintos paquetes de datos, mensajes, datagramas… y de enviarlos donde proceda o generar respuestas… También es la encargada de generar prácticamente todos los eventos: envío de datos, recepción de datos, errores…

2.1 El Sistema Gestor de Eventos

De los cuatro bloques de la figura 1, es el más fácil de implementar, aunque está distribuido. Tiene una parte centralizada formada por dos listas de componentes: una para las redes y otra para los equipos. De esta forma cuando hay que comunicarle a algún componente que debe cambiar de estado, se puede localizar fácilmente en una de estas dos listas. La parte distribuida esta implementada en cada uno de los demás componentes. Cuando el núcleo gestor de eventos da la orden a un componente para que cambie de estado, se pone en marcha una serie de procesos dentro del componente. Estos procesos dependen de cada componente pero, de forma genérica se puede decir que consiste en procesar las entradas, y procesar las salidas de paquetes de datos. A su vez, cuando se procesan las entradas (o las salidas) se envía la orden a cada nivel de la pila de comunicaciones (cuando exista) para que procese sus entradas (o salidas).

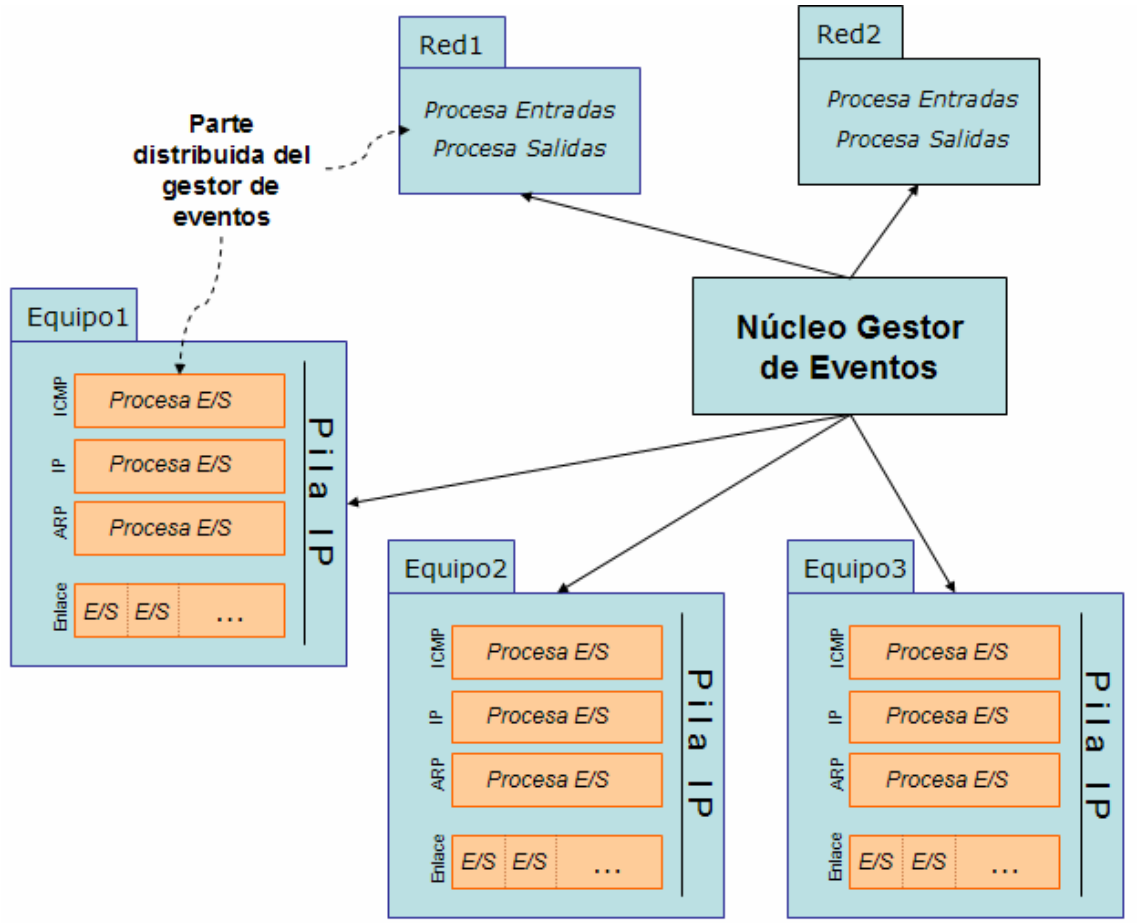

Figura 2. Esquema del funcionamiento distribuido del gestor de eventos

2.2 Las Redes

Las redes están formadas por cinco componentes: el **comportamiento**, el tipo de **direcciones** que usa, el **nivel de enlace**, el tipo de **interfaz** y el tipo de **trama de datos**. Mediante el componente que modela el comportamiento se consigue que sea fácilmente implementar nuevas redes, o introducir pequeñas modificaciones en las existentes (facilidad de ampliación). Para dotar al sistema de simulación del realismo necesario, es importante que las direcciones que se usen sean idénticas a las que utiliza ese tipo de red, de ahí que se haya decidido incluir un componente de este tipo. Otro aspecto muy importante es el nivel de enlace, que es dependiente de cada tipo de red, ya que no se comporta de igual forma el nivel de enlace Ethernet que el punto a punto, porque la información que se maneja es distinta. Esta información se define con el tipo de trama, que también se ha dotado de un componente de realismo importante. Finalmente, para hacer el sistema más parecido a la realidad se ha introducido el componente de interfaz, que podríamos equiparar a lo que son las tarjetas de red.

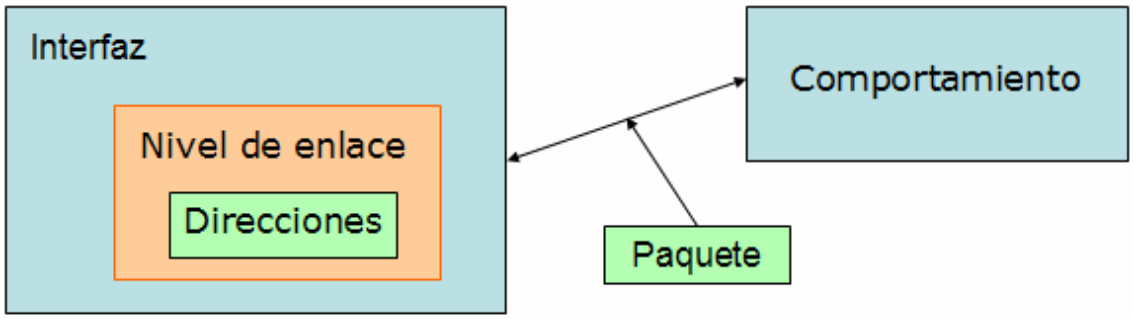

Figura 3. Esquema de bloques de los componentes de una red

La parte de la interfaz es la que se usaría dentro de los equipos cuando los quisiéramos conectar a una red con un determinado comportamiento. Por ejemplo: a una red de tipo Ethernet (comportamiento Ethernet) podremos conectar los equipos que dispongan de interfaces de tipo Ethernet, que serán las encargadas de gestionar la comunicación de la forma adecuada usando el nivel de enlace Ethernet que tienen incorporado y usando el direccionamiento correcto. El hecho de que el nivel de enlace esté asociado a la interfaz y no sea un bloque más como los que forman el resto de la pila de comunicaciones es por una cuestión de facilidad a la hora de usar la interfaz de programación además de que así resulta más intuitivo: se montaría una pila de comunicaciones con los niveles que queramos y después no tendríamos que preocuparnos de si el nivel de enlace es de tal o cual tipo o si hay varios niveles de enlace porque estamos conectados a varias redes… Sencillamente habría varias interfaces, se le indicaría al correspondiente nivel de enlace de la interfaz adecuada que hay que enviar tales datos y listo. La equivalencia con el mundo real sería que cuando queremos conectar un equipo a una red, solo tenemos que añadir la tarjeta (interfaz) indicada para el tipo de red al equipo e instalar el manejador apropiado (nivel de enlace). Aquí tenemos las dos cosas integradas en la interfaz, para facilitar su uso.

De este modo se facilita enormemente que cuando queramos añadir un tipo de red nueva al sistema, de forma dinámica, solo tengamos que cargar estos cinco componentes y todo funcionará correctamente, ya que los demás componentes que hacen uso de las redes y las interfaces utilizan una interfaz de programación común para todos ellas.

2.3 Los Equipos

Los equipos, comparados con las redes, son componentes bastante sencillos, ya que únicamente están formados por una pila de comunicaciones, generalmente distinta para cada tipo de equipo. Por ejemplo: un ordenador tiene una pila de comunicaciones con el nivel IP, y los módulos ARP y ICMP, pero un router, además configura la pila de forma que pueda usar el reenvío de datagramas en su nivel IP.

También tienen la funcionalidad de que se les pueden programar entradas y salidas de datos. Esto es útil tanto si se quiere programar un escenario previo al inicio de la simulación como si se reciben datos desde redes para darles entrada en el equipo sin tener que acceder a las estructuras de datos internas de los equipos. Con esto se facilita la simplicidad de uso por parte del usuario que quiera desarrollar nuevos componentes y que sea fácil añadir nuevos tipos de equipos en tiempo de ejecución al sistema.

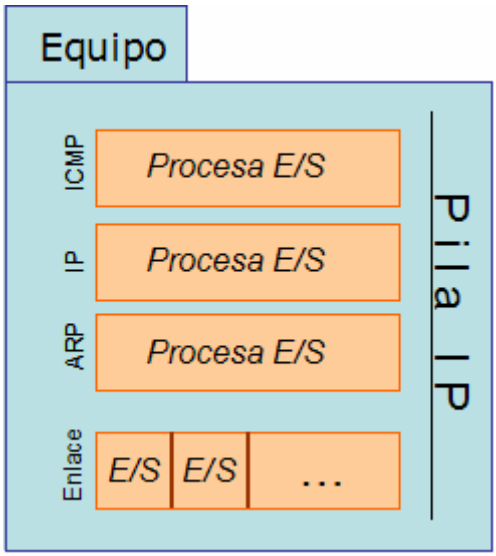

Figura 4. Esquema de una posible implementación de un equipo

2.4 La Pila de Comunicaciones

La pila de comunicaciones es el componente más complejo de todo el sistema de simulación, ya que es el encargado de gestionar todo el tráfico de información entre los equipos y las redes, así como de generar los eventos cuando corresponda. Se ha dividido la pila en niveles o módulos totalmente independientes desde el punto de vista de la implementación, aunque no así de su uso. Esto es comprensible ya que los módulos implementados están bastante relacionados entre si. Las opciones disponibles eran: implementar los componentes ARP, IPv4 e ICMP de forma monolítica, es decir, todo en un bloque, o implementarlos por separado definiendo una interfaz de uso entre ellos. Al final se optó por esta última opción porque permite un mejor mantenimiento del sistema y lo hace mucho más manejable por parte del usuario. La complejidad es superior debido a que un módulo no sabe a ciencia cierta si el módulo al que le debe enviar información existe o no. Para solventar esto se han creado mecanismos de enlace entre niveles para que sea fácil localizarlos dentro de la pila (si existen).

Cada uno de estos módulos dispone de:

- Una cola de entrada y otra de salida de datos. Cuando lo que se quiere es enviar hacia la red, o hacia los niveles inferiores alguna información, se usa la cola de salida y cuando se recibe algún dato desde la red o se quiere enviar algo hacia los niveles superiores se usa la cola de entrada.
- Una lista de niveles inferiores y otra de niveles superiores con los que está enlazado el modulo. De esta forma se puede tener que el modulo ARP tiene como niveles inferiores los distintos niveles de enlace, pero no tiene niveles superiores, o que el modulo IP tiene como nivel superior, al modulo ICMP y como niveles inferiores a los niveles de enlace y al modulo ARP.
- Una lista de errores que se deberán simular durante la evolución del sistema.
- Una lista de identificadores de niveles (ver más adelante la clase *LocalizadorRedes*).
- Un enlace al equipo propietario de la pila
- Una lista de propiedades del módulo, como pueden ser tiempos de retardo de procesamiento, *timeouts*, …

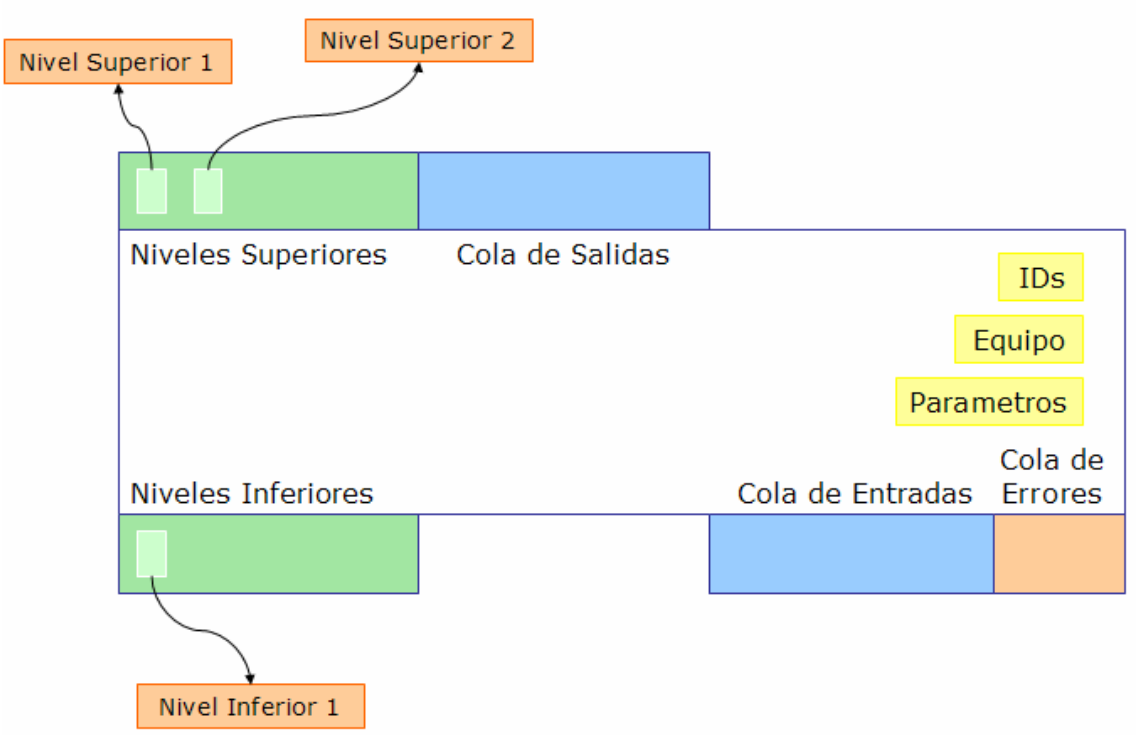

Figura 5. Estructuras de datos usadas por un módulo de la pila

En la siguiente figura se puede ver como quedarían las relaciones entre los niveles de una pila IP ya montada con los distintos enlaces entre los módulos, y donde estarían definidos los módulos:

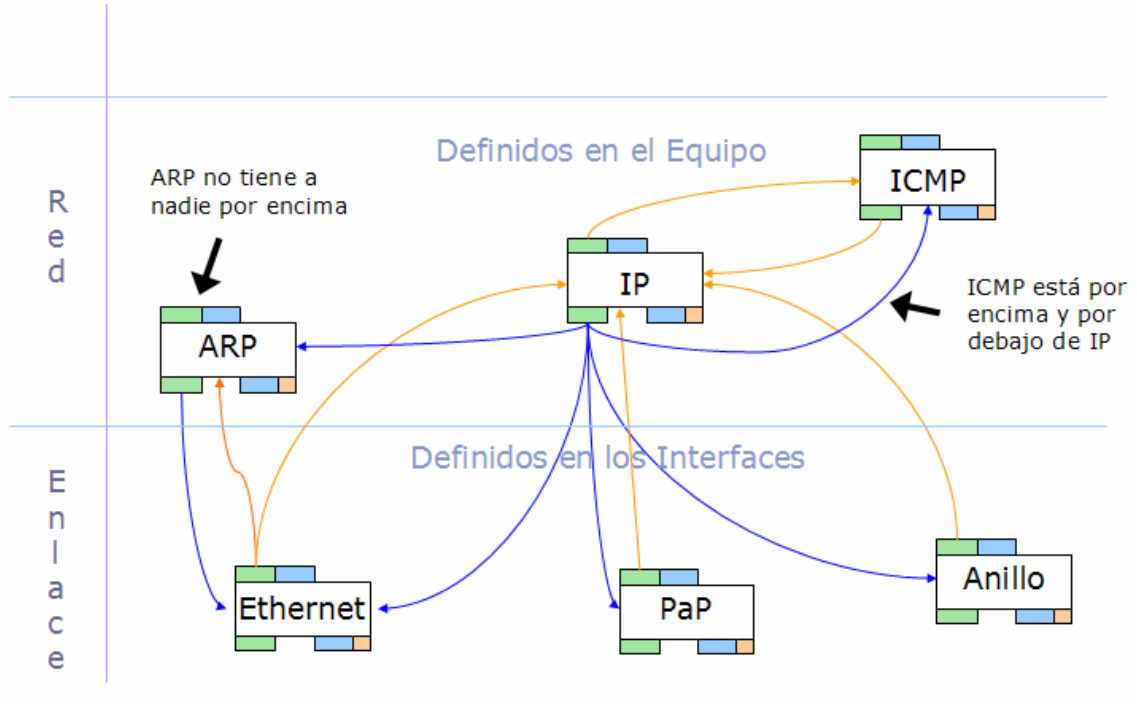

Figura 6. Ejemplo de pila IP con todas las conexiones entre módulos

Ante el esquema anterior, surge la duda de cómo es posible que recibiendo un modulo, un paquete de datos sepa a que nivel superior debe entregarlo. Este problema se soluciona mediante listas de identificadores. Estas listas contienen información que relaciona un determinado identificador de nivel superior en un campo de un paquete de datos con la información relativa al nivel inferior que debe procesar primero el paquete. Esto se traduce en que, por ejemplo, cuando a un nivel de enlace Ethernet le llega una trama, en cuyo campo de tipo de datos pone el valor 0x0800, se consulta la tabla de asociaciones buscando qué nivel es el que controla las tramas de tipo Ethernet/0x0800 y el sistema responde que el nivel buscado es el nivel IP; después el nivel de enlace Ethernet comprueba que tiene un nivel superior de tipo IP y le envía la carga de datos de la trama Ethernet.

El funcionamiento tiene dos casos especiales: cuando no existe un nivel superior adecuado, y cuando un determinado nivel esta por encima y por debajo de otro. En el primer caso, el comportamiento depende del nivel que tenga que pasar los datos: por ejemplo, si un nivel de enlace Ethernet tiene que enviar algo a un nivel superior que no existe, sencillamente descarta dicha información, puesto que no hay mecanismos de control de errores; en el caso de un nivel IP que no tuviese el nivel de destino superior requerido (por ejemplo TCP) el estándar dice que se debe enviar un mensaje ICMP de error al equipo que generó el datagrama. En el segundo caso, no existe ningún problema ya que es la gestión interna del nivel el que se encarga en cada momento de decidir si lo que busca es un nivel superior o un nivel inferior (utilizando las listas de niveles superiores e inferiores de que dispone, ya que esta información no está mezclada).

Otro aspecto que resulta interesante del esquema anterior es que unos módulos están definidos dentro del equipo y otros dentro de las interfaces del equipo. La forma en que un módulo, por ejemplo, el módulo IP, localiza a un determinado módulo de nivel de enlace es a través del la lista de interfaces del equipo al que pertenece. Imaginemos que el nivel IP, tras consultar su tabla de rutas decide que debe enviar un datagrama por la interfaz '*eth0*' del equipo: lo que hace es obtener del equipo el nivel

de enlace de dicha interfaz y pasarle el datagrama para que lo procese como una salida de datos.

2.5 Modularidad y Extensibilidad

Como se ha dicho anteriormente, uno de los requerimientos del sistema era que el sistema resultante debía ser muy fácilmente ampliable mediante la inclusión de nuevos tipos de redes, equipos y protocolos sin necesidad de tocar nada del código existente. Esto se ha conseguido mediante el uso de las capacidades de carga dinámica de clases de Java y con el uso de registros. Cuando tenemos un nuevo tipo de red, lo que hacemos es '*registrarlo*' como disponible mediante un '*Localizador de Redes*' que será el encargado de cargar las clases correspondientes en memoria cuando se necesiten. A la hora de crear una red de un tipo no conocido a priori o añadida al sistema con posterioridad a su desarrollo, se le dice al localizador que se cree una red de tal tipo, dando el nombre de la red (por ejemplo: '*Ethernet*') y el localizador se encarga de buscar las clases y cargarlas en memoria de forma dinámica, en tiempo de ejecución. De forma similar, existe un '*Localizador de Equipos*' para cargar los equipos. Esta forma de trabajar permite tratar el sistema de simulación como si fuera una especie de '*mecano*' en el que si hacen falta determinadas piezas, se usan y listo, y si no, pues no se usan. También permite desarrollar módulos para el simulador que ni siquiera estaban presentes en los requerimientos iniciales, como por ejemplo los módulos TCP y UDP para la pila de comunicaciones. Los módulos de código '*externo*' que se han desarrollado son las redes Ethernet y Punto a Punto, los Ordenadores y los Routers.

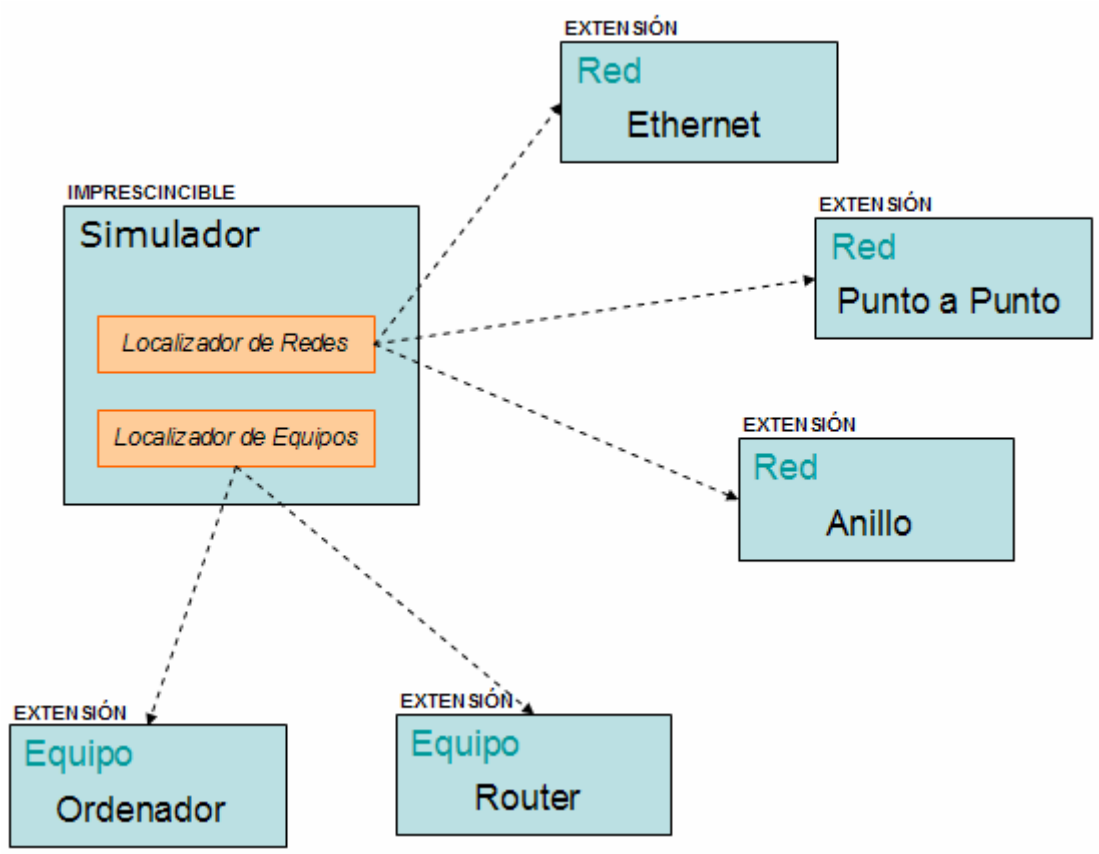

Figura 7. Carga dinámica de redes y equipos en el sistema de simulación

3. Implementación

La implementación de la arquitectura descrita en el punto anterior se ha dividido en varios paquetes de código Java y en clases siguiendo un esquema similar al propuesto anteriormente. El esquema de paquetes que se ha usado es el siguiente:

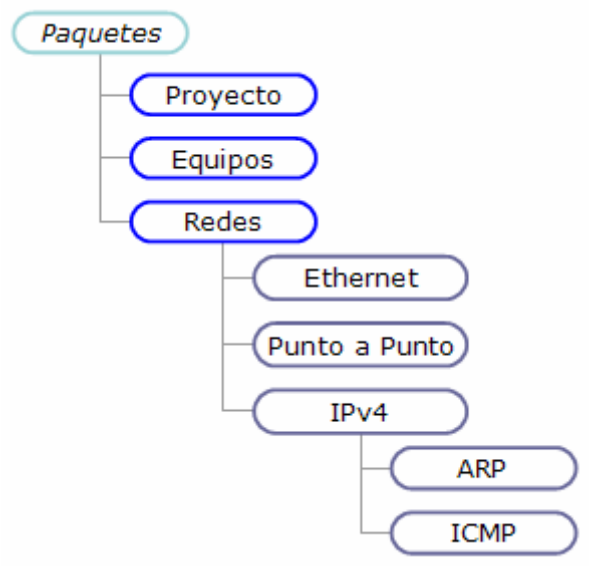

Figura 8. Organización del código en paquetes

3.1 Paquete Proyecto

Este paquete contiene las clases más genéricas de todo el sistema, que son las siguientes:

Nombre de la clase	Descripción
Evento	Es la estructura de datos que se utiliza en el simulador para registrar cualquier evento de importancia que se produzca durante la simulación. Almacena información sobre cuando se produjo el evento, si los datos eran de entrada o de salida, el paquete de datos que provocó el evento…
Objeto	Es la clase base de las redes y los equipos. Define la interfaz común que usará el gestor de eventos a la hora de comunicarle a los componentes de la simulación que deben cambiar de estado. También almacena la información relativa a las interfaces, eventos, nombre…
Simulador	Es el núcleo del gestor de eventos. Permite añadir redes y equipos a la simulación, así como poner en marcha la simulación.
Parametro	Dado que los equipos y las redes pueden tener muchas características, se ha hecho necesario crear

	una estructura básica que almacene el nombre, el valor y una descripción de cada característica o parámetro.
ListaParametros	Para agrupar todos los parámetros que pueden tener los equipos y las redes tenemos esta lista de parámetros.

Tabla 1. Clases del paquete Proyecto.

3.1.1 Clase Evento

El registro de la información relativa a los eventos que se producen en el transcurso de la simulación se lleva a cabo mediante objetos de esta clase, que no es más que una estructura de datos en la que se guarda la siguiente información:

Atributo de la clase	Descripción
`int instante`	Instante de tiempo en que se ha producido el evento durante la simulación
`String mensaje`	Mensaje descriptivo del evento. Sirve para que el usuario sea capaz de distinguir claramente entre unos eventos y otros.
`Buffer paquete`	Paquete de datos que ha provocado el evento. Muy útil si se quiere analizar la información que contiene. Su información se puede analizar a nivel de bit.
`char indicador`	Dependiendo de su valor, nos indica si el paquete de datos estaba siendo recibido por el sistema (‘*R*’), si estaba siendo enviado (‘*E*’), si se produjo algún error durante su procesamiento (‘*X*’), o si se transmitía por una red (‘T’).

Tabla 2. Información asociada a un evento.

Los eventos son registrados en los componentes (redes o equipos) relacionados con ellos. Si se quiere saber qué eventos están asociados a un determinado componente, hay que consultar su lista de eventos (ver atributo *eventos* de la clase Objeto) mediante los métodos que proporciona la clase base Objeto.

3.1.2 Clase Objeto

Esta clase sirve de clase base para las redes y los equipos. Implementa la funcionalidad común a todos los componentes, como son el registro de eventos, la gestión de interfaces y el registro de parámetros de funcionamiento. Para almacenar esta información se utilizan los siguientes atributos:

Atributo de la clase	Descripción
`Vector Interfaces`	Lista de interfaces que relacionados con el componente. En los equipos serán los interfaces que poseen y en las redes serán los interfaces conectados a ellas.
`Vector eventos`	Lista de eventos asociados al componente.
`ListaParametros parametros`	Lista de parámetros de funcionamiento del componente o descriptivos del mismo.

Tabla 3. Información común a todos los componentes.

Para el registro de eventos tenemos los siguientes métodos:

Método	Descripción
`NuevoEvento`	Añade un nuevo evento al registro de eventos del componente, especificando la información descrita en la tabla 2.
`getEvento`	Recupera el i-ésimo evento del registro
`NumEventos`	Devuelve el número de eventos registrados

Tabla 4. Métodos usados en el registro de eventos.

Para la gestión de las interfaces tenemos:

Método	Descripción
`NumInterfaces`	Devuelve el número de interfaces asociadas al componente.
`getInterfaz`	Recupera el i-ésimo Interfaz de la lista, o el que posea el nombre especificado (método sobrecargado)

Tabla 5. Métodos usados en la gestión de interfaces.

Y para la gestión de parámetros tenemos:

Método	Descripción
`getParametros`	Devuelve la lista de parámetros del componente
`getNombre`	Devuelve el nombre del componente.
`setNombre`	Cambia el nombre del componente.

Tabla 6. Métodos de registro de parámetros de funcionamiento.

Además de la lista de parámetros que es especifica de cada instancia de cada componente, existe una lista de características que es global a todos los componentes del mismo tipo y que permite especificar cosas como el nombre genérico del componente, el tipo de dibujo que se usará (bus, anillo, bloque…) y si se trata de una red, si se permite que se conecte a otras redes o a equipos, el tipo de trama/direcciones/interfaces que usa, etc.… Esta lista de características puede ser consultada desde cualquier parte de la aplicación que use la librería mediante el método *getCaracteristicas*.

Existen varios métodos abstractos definidos en esta clase que todos los componentes deben implementar con el fin de definir una interfaz de programación común a todos ellos. Entre ellos hay dos muy importantes:

Método abstracto	Descripción
`Procesar`	Procesa los paquetes de datos programados para un determinado instante.
`Pendientes`	Comprueba si en el componente queda algún paquete de datos pendiente de ser procesado, y devuelve '*true*' en ese caso.

Tabla 7. Métodos abstractos que deben definir los componentes.

3.1.3 Clase Parametro

Los parámetros de funcionamiento de los componentes se definen mediante objetos de esta clase. Un parámetro está formado por cuatro elementos:

Atributo	Descripción
`String descripcion`	Es una descripción del parámetro, sirve para que el usuario sepa en tiempo de ejecución cual es la finalidad del parámetro por si tiene que especificarse mediante un cuadro de texto o similar.
`String clase`	Clase Java asociada al valor del parámetro
`String nombre`	Nombre del parámetro
`String valor`	Valor del parámetro, debe ser un objeto de la clase especificada por el atributo '*clase*'

Tabla 8. Elementos de un 'parámetro' de funcionamiento.

Se dispone de varios métodos de tipo get/set para acceder y/o modificar esta información, son los siguientes:

Método	Descripción
`get`	Devuelve el valor del parámetro
`set`	Modifica el valor del parámetro
`getNombre`	Devuelve el nombre
`getDescripcion`	Devuelve la descripción

Tabla 9. Métodos de la clase Parametro.

3.1.4 Clase ListaParametros

Para encapsular el funcionamiento y los datos relativos a los parámetros se ha implementado esta clase. Se trata de una lista de parámetros con una serie de métodos para manipular cada uno de forma sencilla.

Atributo de la clase	Descripción
`Vector parametros`	Lista de parámetros de funcionamiento del componente o descriptivos del mismo.

Tabla 10. Atributo de la clase ListaParametros

Los métodos implementados para manipular los parámetros son los siguientes:

Método	Descripción
`get`	Devuelve el valor del i-ésimo parámetro o del que tenga el nombre especificado (método sobrecargado).
`add`	Añade un parámetro a la lista.
`size`	Devuelve el tamaño de la lista de parámetros.
`setValor`	Cambia el valor del parámetro que tenga el nombre especificado.
`getValor`	Devuelve el valor del parámetro que tenga el nombre especificado.

Tabla 11. Métodos de la clase ListaParametros.

3.1.5 Clase Simulador

Esta clase implementa la parte centralizada del gestor de eventos definido en el punto 2.1 anteriormente. Toda simulación debe contar con un objeto de este tipo y solo con uno. Está formado por cuatro elementos y por varios métodos usados para acceder a ellos:

Atributo de la clase	Descripción
`Vector ListaEquipos`	Lista de equipo que participan en la simulación.
`Vector ListaRedes`	Lista de redes que participan en la simulación.
`int instante_actual`	Instante de tiempo actual en que se encuentra la simulación, comenzando por 0.
`int numPasosMax`	Número máximo de pasos que se realizará durante la simulación (es lo mismo que el instante de tiempo en que deberá acabar la simulación)

Tabla 12. Atributos de la clase Simulador.

Método	Descripción
`NuevoEquipo`	Añade un equipo a la simulación.
`getEquipo`	Obtiene el i-ésimo equipo de la simulación.
`NuevaRed`	Añade una red a la simulación
`getRed`	Obtiene la i-ésima red de la simulación.
`NumeroMaximoDePasos`	Ajusta el número máximo de pasos que se realizará durante la simulación.
`SimularUnPaso`	Este método es el encargado de hacer que todos los componentes de la simulación vayan cambiando.

Tabla 13. Métodos de la clase Simulador.

El método *SimularUnPaso* es el núcleo de la parte centralizada de la gestión de eventos, su funcionamiento es el siguiente se puede ver en el siguiente cuadro. Básicamente consiste en recorrer la lista de equipos y después la de redes invocando a los métodos Procesar de cada uno de ellos para que cambien de estado (pasos 1 y 2), después comprobar si queda algún paquete de datos pendiente de ser procesado para detener o no la simulación (pasos 3 y 4), y al final comprobar si se ha llegado al número máximo de pasos programado para finalizar la simulación (paso 5).

```
public boolean SimularUnPaso()
{
   boolean se_puede_continuar=false;

   // 1. Comprobamos si algún equipo tiene algo que procesar en
   //    el instante de tiempo actual
   for(int i=0;i<ListaEquipos.size();i++)
   {
       Equipo e=(Equipo)ListaEquipos.get(i);
       e.Procesar(instante_actual);
   }

   // 2. Comprobamos si alguna red tiene que enviar alguna trama
   for(int i=0;i<ListaRedes.size();i++)
   {
       Red r=(Red)ListaRedes.get(i);
       r.Procesar(instante_actual);
   }

   // 3. Comprobamos si queda algún paquete por procesar en los
   //    equipos o en las redes
   for(int i=0;!se_puede_continuar && i<ListaEquipos.size();i++)
   {
       Equipo e=(Equipo)ListaEquipos.get(i);
       if(e.Pendientes()>0)
           se_puede_continuar=true;
   }
   for(int i=0;!se_puede_continuar && i<ListaRedes.size();i++)
   {
       Red r=(Red)ListaRedes.get(i);
       if(r.Pendientes()>0)
           se_puede_continuar=true;
   }

   // 4. Pasamos al siguiente instante de tiempo
   instante_actual++;

   // 5. Devolvemos el estado del simulador
   if(se_puede_continuar && instante_actual==numPasosMax)
   {
        se_puede_continuar=false; //salida forzada
   }
   return(se_puede_continuar);
}
```

Código 1. Núcleo gestor de eventos. Método Simulador.SimularUnPaso

3.2 Paquete Equipos

En este paquete encontramos todas las clases relacionadas con el comportamiento de los distintos equipos, la clase base *Equipo*, de la cual derivan todos, la clase *LocalizadorEquipos* que es la encargada de cargar en tiempo de ejecución las clases asociadas a los equipos, y las clases asociadas a los equipos, como son *Ordenador* y *Router*.

3.2.1 Clase Equipo

Un equipo es un componente capaz de generar y enviar tramas de datos, que además posee una tabla de rutas y define una serie de métodos que forman la interfaz de programación común a todos los equipos. También, como se veía en la *tabla 5*, define un comportamiento a la hora de añadir un interfaz a la lista de interfaces.

Por tanto tenemos que los atributos específicos de un Equipo son:

Atributo de la clase	Descripción
`TablaDeRutas tablaDeRutas`	Tabla de rutas propia de un equipo que trabaja a nivel IP. Se inicializa con la del módulo IP, en realidad no es más que un enlace a la del módulo IP, pero así se facilita mucho su manejo y se evita que el usuario pueda acceder al módulo directamente.

Tabla 14. Atributos de la clase Equipo.

La interfaz de programación que debe definir cada uno de los componentes que deriven de esta clase es la definida en la siguiente tabla. El hecho de que estos métodos deban ser definidos en las clases derivadas de Equipo es porque, ni todos los equipos tienen que tener el mismo número de niveles en la pila, ni tienen por qué comportarse de la misma forma, ni permitir que se configuren igual. Mediante estos métodos es posible crear equipos con distintos comportamientos, con gran facilidad.

Método	Descripción
`ProgramarEntrada`	Simula la entrada de un determinado dato en el equipo, por el nivel más bajo de la pila de comunicaciones.
`ProgramarSalida`	Simula la salida (envío) de un dato en el equipo en el nivel que corresponda de la pila de comunicaciones. Por ejemplo: si el dato es una petición ARP se programará como salida en el modulo ARP.
`SimularError`	Activa o desactiva la simulación de un determinado tipo de error en el módulo de la pila que se especifique.

`ConfiguraPila`	Controla el funcionamiento de la pila de comunicaciones, y de cada uno de sus módulos.

Tabla 15. Métodos abstractos de la clase Equipo.

Como se veía en la *tabla 5*, los equipos deben definir un comportamiento a la hora de añadir una interfaz a su lista de interfaces ya que no es lo mismo añadir una interfaz a una red que a un equipo. El comportamiento de los equipos en esta situación es comprobar que la interfaz está conectada a una red, crear el nivel de enlace asociado a la interfaz y crear el enlace de la interfaz con el equipo. Una vez hecho esto se comprueba que la interfaz se puede conectar a la red y se crea el enlace entre la red y el equipo con la interfaz. Como se puede ver existe un enlace en la interfaz que apunta a la red y otro en la red que apunta a la interfaz y al equipo, esto es debido a que el nivel de enlace de la interfaz tiene que enviar las tramas de datos a la red y es conveniente que tenga un acceso al objeto Red correspondiente, algo similar sucede al revés, cuando la red tiene que enviar tramas a los equipos, es conveniente que tenga enlaces con los equipos y con sus interfaces con el fin de saber las direcciones de destino y poder programar las entradas en los equipos correspondientes.

3.2.2 Clase LocalizadorEquipos

Esta clase implementa el cargador de equipos en tiempo de ejecución. Cuando se dispone en el sistema de una nueva clase de equipo es necesario registrarla en este cargador para que se pueda cargar en tiempo de ejecución. Este registro se lleva a cabo mediante el método *Registrar*, al cual se le pasa el nombre de la clase asociada al nuevo equipo. Cuando se quiere crear un objeto equipo de un tipo conocido por el cargador se llama al método *New* pasándole el tipo de equipo y el nombre que se le quiere dar a dicho equipo.

Atributo de la clase	Descripción
`Vector Clases`	Vector en el que se guardará el nombre de todas las clases que el cargador sea capaz de utilizar para crear equipos.

Tabla 16. Atributos de la clase LocalizadorEquipos.

Método	Descripción
`Registrar`	Comprueba que la clase se puede cargar y la añade a la lista de clases disponibles.
`NumEquipos`	Devuelve el número de equipos registrados.
`New`	Devuelve un objeto del tipo equipo especificado.

Tabla 17. Métodos de la clase LocalizadorEquipos.

3.2.3 Clase Ordenador

Un ordenador es un tipo de equipo que posee una pila de comunicaciones formada por los módulos de nivel de enlace correspondientes, un módulo ARP, un módulo IP y otro ICMP. Estos módulos están enlazados entre sí de la siguiente forma:

```
public Ordenador()
{
    // 1. Definimos los niveles de la pila
    moduloARP=new ModuloARP(this);
    moduloICMP=new ModuloICMP(this);
    nivelIPv4=new NivelIPv4(this,moduloARP,moduloICMP);
    nivelIPv4.IPForwarding(false);

    // 2. Interconectamos los niveles
    moduloICMP.setNivelInferior(nivelIPv4);
    nivelIPv4.setNivelInferior(moduloARP);
    nivelIPv4.setNivelSuperior(moduloICMP);
    nivelIPv4.IPForwarding(false);

    // Los niveles de enlace son gestionados por la tabla de rutas
    // del nivel IP y por los interfaces.

    // 3. Enlazamos la tabla de rutas
    tablaDeRutas=nivelIPv4.tablaDeRutas;
}
```

Código 2. Creación y enlazado de módulos de la pila IP (ver figura 6).

La forma de crear los niveles se puede ver en el punto 1, los constructores no tienen un prototipo predefinido, pero se les debe pasar el enlace al equipo para que tengan accesible la información relativa a los demás niveles. Como se puede ver en el punto 2, la forma de interconectar módulos de la pila de comunicaciones es mediante las llamadas a los métodos *setNivelInferior* y *setNivelSuperior* de cada módulo, pasándoles el módulo con el que se quiere enlazar. Estos métodos hacen uso de la lista de identificadores que se menciona en la *figura 5*. Finalmente, como se anunciaba en la *tabla 14*, la tabla de rutas de los equipos (en este caso el ordenador), no es más que un enlace a la tabla de rutas del nivel IP, de este modo conseguimos que sea visible (accesible) para el usuario a través del objeto equipo, pero que los módulos de la pila sean invisibles (inaccesibles).

Como en un ordenador tenemos niveles de la pila por encima de los niveles de enlace, es necesario que cuando se añada un nivel de enlace, se creen las asociaciones necesarias con los niveles superiores, es decir con los niveles ARP e IP. Para conseguirlo solo tenemos que introducir una pequeña ampliación al método *setInterfaz* heredado de la clase *Equipo*:

```
public void setInterfaz(Interfaz interfaz)
{
    super.setInterfaz(interfaz);

    // 1. Enlazamos el nivel de enlace
    interfaz.getNivelEnlace().setNivelSuperior(moduloARP);
    interfaz.getNivelEnlace().setNivelSuperior(nivelIPv4);
}
```

Código 3. Enlazado de un nivel de enlace con los niveles superiores.

Y ¿cómo saben esos niveles que tienen por debajo un nuevo nivel de enlace? Pues lo saben a través de la tabla de rutas y del enlace al equipo. Por ejemplo: el módulo IP obtiene la interfaz a partir de la tabla de rutas y a partir de la interfaz obtiene el nivel de enlace (ver *figura 3*); y el módulo ARP lo obtiene de la información que le pasa el módulo IP cuando hay que hacer una petición (el módulo IP le dice a través de que interfaz tiene que enviar la petición y el módulo ARP se la pasa al nivel de enlace de dicha interfaz). El hecho de que la asociación de los niveles de enlace se realice de forma distinta a la que aparece en el listado de *código 2*, es debido a la arquitectura del sistema, y que a priori un equipo no sabe que módulos de nivel de enlace tiene (por que no tiene interfaces) como se puede observar en la *figura 4* y en la *figura 6*.

Debido a que cada tipo de equipo puede tener un número variable de módulos en su pila de comunicaciones, y que puede gestionarlos de formas muy diferentes, es necesaria una implementación personalizada de los métodos abstractos descritos en la *tabla 15*. En el caso de la clase Ordenador esta implementación es muy sencilla como se muestra en los listados siguientes, y no consiste más que en recorrer los elementos de la pila ejecutando las operaciones adecuadas:

```
public void Procesar(int instante)
{
    // 1. Comprobamos si hay algo que procesar en el modulo ICMP
    moduloICMP.Procesar(instante);

    // 2. Comprobamos si hay algo en el nivel IPv4
    nivelIPv4.Procesar(instante);

    // 3. Comprobamos si el modulo ARP tiene que enviar peticiones
    moduloARP.Procesar(instante);

    // 4. Comprobamos si hay algo que procesar en los niv. de enlace
    for(int i=0;i<NumInterfaces();i++)
        getInterfaz(i).getNivelEnlace().Procesar(instante);
}
```

Código 4. Orden de procesado de la información en un Ordenador.

Como se puede ver, el hecho de haber definido una interfaz de programación común para todos los módulos, facilita enormemente la posterior interpretación del código.

Un aspecto muy importante desde el punto de vista de los demás componentes que participan en la simulación es cómo se le hace llegar un determinado paquete de datos al equipo y cómo se hace para que el equipo envíe un dato al exterior. La entrada de datos desde la red se lleva a cabo mediante el método *ProgramarEntrada*, el cual pone en la cola de entrada del nivel de enlace adecuado el dato especificado (ver *figura 5*) y la salida de datos se realiza mediante el método *ProgramarSalida*, que coloca en la cola de salida del nivel correspondiente el dato especificado. A la hora de programar una salida hay que tener en cuenta el tipo de dato que estamos programando porque, por ejemplo, si queremos enviar un mensaje ICMP lo tendremos que programar en el modulo ICMP y no en los demás; estas distinciones son fáciles de hacer mediante la identificación dinámica de tipos de Java:

```
public void ProgramarEntrada(Dato dato)
{
    if(dato!=null && dato.interfaz!=null)
       dato.interfaz.getNivelEnlace().ProgramarEntrada(dato);
}
```

Código 5. Entrada de datos en un Ordenador.

```
public void ProgramarSalida(Dato dato)
{
    if(dato==null) return;

    // 1. Paquete ARP
    if(dato.paquete instanceof PaqueteARP)
        if(moduloARP.ProgramarSalida(dato))
           NuevoEvento('E',dato.instante,dato.paquete,
                       "Envio de datos programado en ARP");

    // 2. Mensaje ICMP
    else if(dato.paquete instanceof MensajeICMP)
        if(moduloICMP.ProgramarSalida(dato))
            NuevoEvento('E',dato.instante,dato.paquete,
                        "Envio de datos programado en ICMP");

    // 3. Otros (DatagramaIPv4 y otros tipos de paquete)
    else
        if(nivelIPv4.ProgramarSalida(dato))
            NuevoEvento('E',dato.instante,dato.paquete,
                        "Envio de datos programado en IP");
}
```

Código 6. Salida de datos de un ordenador.

Cuando el núcleo gestor de eventos (ver *código 1*) necesita comprobar si un ordenador tiene algo pendiente de ser procesado utiliza el método *Pendientes*, heredado de la clase *Objeto*. Como en un ordenador tenemos tres módulos en el nivel de red (IP, ICMP y ARP), más un número variable de módulos de nivel de enlace, el número de paquetes de datos pendientes de ser procesados será la suma de los paquetes pendientes en cada uno de los módulos:

```
public int Pendientes()
{
    int pendientes=0;

    // 1. Comprobamos si hay algo que procesar en los niveles
    //    de enlace
    for(int i=0;i<NumInterfaces();i++)
        pendientes+=getInterfaz(i).getNivelEnlace().Pendientes();

    // 2. Devolvemos el numero de paquetes que nos han quedado
    //    por procesar
    pendientes+=moduloARP.Pendientes() + moduloICMP.Pendientes() +
                nivelIPv4.Pendientes();
    return(pendientes);
}
```

Código 7. Cálculo del número de paquetes pendientes de procesar.

La simulación de errores se lleva a cabo mediante la activación de flags en los distintos módulos de la pila, pero para evitar que desde fuera de un objeto Equipo se pueda acceder a ellos, se hace necesario utilizar el método *SimularError*, al cual se le pasa la información relativa al nivel, al tipo de error (flag) y un valor que indica si se activa o se desactiva la simulación de ese tipo de error (ver *código 8*). Los tipos de errores son propios de cada módulo de la pila, y se llaman de forma distinta, por ejemplo: para el nivel IP existe un flan llamado *NO_ENVIAR_1* que cuando esta activo provoca que no se envíe el primer fragmento de los datagramas fragmentados. El listado completo de flags se puede consultar más adelante.

Finalmente tenemos la configuración de la pila de comunicaciones. Para ajustar los parámetros de funcionamiento de cada módulo (y evitar el acceso a los módulos desde fuera del objeto Equipo) tenemos dos métodos, uno para configurar los módulos de enlace y otro para los módulos del nivel de red. La única diferencia es que para configurar los de nivel de enlace, usamos el nombre de la interfaz que tienen asociada y para configurar los de nivel de red usamos uno identificadores internos (constantes) que posee la clase Equipo (ver *código 9* y *código 10*).

Se puede observar, que la interfaz de programación usada en los listados de código del 4 al 7, es muy homogénea ya que todos los métodos que tienen un mismo funcionamiento (aunque en lugares diferentes) se llaman de igual forma. Así conseguimos que el usuario se acostumbre más rápida y fácilmente a programar con este sistema de simulación.

```
public boolean SimularError(int nivelID,String flag,boolean activar)
{
    boolean correcto=true; // de momento todo va bien

    switch(nivelID)
    {
        case Equipo.kARP:
        {
            correcto=moduloARP.SimularError(flag,activar);
            break;
        }

        case Equipo.kIPv4:
        {
            correcto=nivelIPv4.SimularError(flag,activar);
            break;
        }

        case Equipo.kICMP:
        {
            correcto=moduloICMP.SimularError(flag,activar);
            break;
        }

        default:
        {
            correcto=false;  // nivel no permitido (desconocido)
        }
    }

    return(correcto);
}
```

Código 8. Simulación de errores en los niveles IP, ICMP y ARP.

```
public void ConfiguraPila(String nInterfaz,String param,Object valor)
{
    Interfaz interfaz=this.getInterfaz(nInterfaz);
    if(interfaz!=null)
        interfaz.getNivelEnlace().parametros.setValor(param,valor);
}
```

Código 9. Configuración de los módulos de enlace de la pila.

```
public void ConfiguraPila(int nivelID, String param, Object valor)
{
    switch(nivelID)
    {
        case Equipo.kARP:
        {
            moduloARP.parametros.setValor(param,valor);
            break;
        }

        case Equipo.kIPv4:
        {
            nivelIPv4.parametros.setValor(param,valor);
            break;
        }

        case Equipo.kICMP:
        {
            moduloICMP.parametros.setValor(param,valor);
            break;
        }
    }
}
```

Código 10. Configuración de los módulos del nivel de red de la pila.

3.2.4 Clase Router

Un equipo de tipo Router es prácticamente lo mismo que un equipo de tipo Ordenador, solo que con una pequeña diferencia: puede redirigir el tráfico IP si no va dirigido a él, es decir, permite 'IP Forwarding'. El 'IP Forwarding' es un parámetro de funcionamiento del módulo IP de la pila de comunicaciones que por defecto está desactivado, pero que, para los routers, se activa en su constructor mediante las siguientes líneas de código:

```
...
    // 4. Activamos el IP Forwarding
    nivelIPv4.IPForwarding(true);
...
```

Código 11. Activación del IP Forwarding en un Router

El resto del funcionamiento de un router, es idéntico al de un ordenador.

3.3 Paquete Redes

En este paquete se encuentran todas las clases relacionadas con las distintas topologías y clases de redes que se pueden usar durante la simulación, así como la implementación de los distintos módulos que componen la pila de comunicaciones. La estructura interna del paquete es la que se muestra en la siguiente figura:

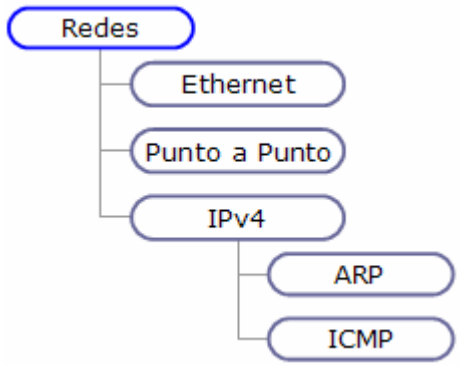

Figura 9. Estructura interna del paquete Redes.

Cada tipo de red tiene su propio subpaquete, totalmente independiente del resto, aumentando así la modularidad del sistema, pero manteniendo un cierto orden dentro de la estructura del código. Cada tipo de red se compone obligatoriamente de una serie de clases, cada una correspondiente a cada uno de los cinco módulos que se definían anteriormente (ver punto 2.2), y cada módulo de la pila de comunicaciones también esta formado por varias partes.

3.3.1 Clases base y clases utilidad

Dentro de los cinco módulos que debe implementar cada tipo de red, existe una gran parte de funcionalidad que es común, sea cual sea el tipo de red, como es la gestión a nivel de bit/byte de las tramas de datos o que las direcciones estén formadas por una serie de bytes… Debido a esto, se ha decidido crear una serie de clases (*Red*, *Nivel*, *Direccion*, *Interfaz* y *Buffer*) que servirán de base para la implementación de redes y que además definirán una interfaz de programación común. También se han creado un par de clases (*Dato* e *IDNivel*) que se usarán como apoyo para el correcto funcionamiento de las cinco clases anteriores.

3.3.1.1 Clase base Buffer

La clase Buffer sirve como punto de partida para implementar distintos tipos de paquetes de datos, ya que implementa los métodos necesarios para manipular la información a nivel de bits, bytes y nibbles (4 bits). Tiene métodos para recuperar y para modificar el contenido de un buffer, del tipo get/set. Una característica importante es que dentro de un buffer el primer byte es el 0 y dentro de un byte el bit de mayor peso es el 7. A continuación se muestra un resumen de los atributos y los métodos más importantes de esta clase:

Atributo de la clase	Descripción
`int[] Datos`	Vector de elementos que simulan ser los bytes del paquete de datos.
`int longitud`	Número de bytes del vector *Datos*

Tabla 18. Atributos de la clase Buffer.

Método	Descripción
`Buffer`	Constructor. Se puede crear un objeto Buffer vacío o basándose en otro Buffer, en una cadena de texto, o con una longitud determinada.
`getbit/setbit`	Recupera o modifica el bit especificado de un determinado byte.
`getbyte/setbyte`	Recupera o modifica el byte especificado.
`getByteH/setByteH`	Recupera o modifica la parte alta de un byte.
`getByteL/setByteL`	Recupera o modifica la parte baja de un byte.
`Tam`	Devuelve el tamaño del buffer
`Contenido`	Devuelve un mensaje con la información relativa al buffer (dividida en campos).

Tabla 19. Métodos de la clase Buffer.

3.3.1.2 Clase base Direccion

En esta clase se basan todas las clases *DireccionXXX* de los distintos tipos de redes. En ella se encuentra la implementación de los métodos para acceder, modificar y comparar la dirección a nivel de byte. Los atributos y los métodos más importantes son los siguientes:

Atributo de la clase	Descripción
`int[] bytes`	Vector que contiene los distintos bytes que forman la dirección.
`int longitud`	Número de bytes que forman la dirección.
`int tipo`	Tipo de dirección. Es una constante asociada a cada tipo de red.

Tabla 20. Atributos de la clase Direccion.

Método	Descripción
`Buffer`	Constructor. Se puede crear un objeto *Direccion* basándose en otra dirección o dando la longitud y el tipo.
`getbits`	Devuelve la dirección en forma de cadena binaria.
`getbyte/setbyte`	Recupera o modifica el byte especificado.
`Longitud`	Devuelve el número de bytes que forman la dirección.
`tipo`	Devuelve el identificador de tipo de dirección.
`Tam`	Devuelve el tamaño del buffer
`equals`	Método comparador de direcciones.

Tabla 21. Métodos de la clase Direccion.

3.3.1.3 Clase base Interfaz

Las interfaces son el mecanismo por el cual un equipo se conecta a una red. Cada conexión de un equipo a una red se distingue por su dirección IP y su correspondiente máscara, además cada interfaz tiene un nombre (por ejemplo: '*eth0*'). Como se veía anteriormente (ver punto 2.2), asociado a cada tipo de interfaz tenemos el módulo de nivel de enlace encargado de comunicarse con la red. De todo esto tenemos que una interfaz debe estar compuesta, en cualquier caso por:

Atributo de la clase	Descripción
`Red red`	Red a la que estará conectada la interfaz.
`Nivel nivelEnlace`	Módulo de la pila que se usará como nivel de enlace con la red.
`ListaParametros parámetros`	Lista de parámetros de funcionamiento de la interfaz, como pueden ser, la dirección IP, la máscara y el nombre de la interfaz.

Tabla 22. Atributos de la clase Interfaz.

Para tratar con esta información se han creado una serie de métodos que evitan tener que acceder a la lista de parámetros cada vez que se quiera acceder la IP, a la máscara y al nombre de la interfaz (métodos de acceso rápido, por llamarlos de alguna forma), y que son del tipo *getXXX*. También se han definido métodos de este tipo para consultar el nivel de enlace y la red.

Además de estos métodos, todos los tipos de interfaces que se implementen deben definir un método que sea capaz de crear el nivel de enlace apropiado para gestionar la conexión con la red.

Método	Descripción
`Conectar`	Conecta la interfaz a una red de tipo compatible, es decir, del mismo paquete que la interfaz.
`getRed`	Devuelve la red a la que está conectada la interfaz.
`getNivelEnlace`	Devuelve el nivel de enlace asociado a la interfaz.
`getNombre`	Devuelve en nombre de la interfaz.
`getDirFisica`	Devuelve la dirección de nivel de enlace, que es un parámetro que se define en las clases derivadas.
`getClaseRed`	Devuelve el paquete que contiene las clases de red que son compatibles con esta interfaz.
`getIP`	Devuelve la IP de la interfaz.
`getMascara`	Devuelve la máscara asociada a la IP.
`CreaNivelEnlace`	Crea el módulo que gestionará el nivel de enlace y lo asociará al equipo. (método abstracto)

Tabla 23. Métodos de la clase Interfaz.

3.3.1.4 Clase utilidad Dato

Los objetos de la clase *Dato* son los que se utilizan durante la simulación para pasar la información relevante entre los equipos y las redes. Contiene información sobre el instante de tiempo en que deberá ser procesado, el paquete de datos asociado, si se puede fragmentar o no… Es un elemento crítico del sistema de simulación ya que es la base del intercambio de información entre los componentes. Su contenido debe interpretarse siempre de acuerdo a la situación en que se esté, ya que no todos sus atributos tienen sentido en todas las situaciones. A continuación se muestra una tabla con los distintos atributos que tiene un objeto de la clase *Dato* junto con una descripción detallada.

Algo a tener en cuenta sobre esta clase, es que es la que utilizará el usuario de la API de simulación para programar los envíos (y/o recepciones) de datos antes de iniciar la simulación, por tanto deberá poner un especial cuidado en comprender para que se usa cada campo, con el fin de obtener los resultados esperados. Si no se usa correctamente se pueden producir eventos incomprensibles durante el proceso de simulación que pueden ser extremadamente difíciles de analizar mediante un proceso de depuración 'línea a línea' del código, aunque se han implementado varias medidas que facilitan el procedimiento (aunque no se garantiza nada).

Atributo de la clase	Descripción
`int instante`	Instante de tiempo en que el equipo o red que posea el *Dato* deberá proceder a su procesamiento.
`Buffer paquete`	Paquete de datos que será objeto de análisis durante el procesamiento.
`int protocolo`	En el caso de que el paquete sea de entrada y se deba pasar a un nivel superior, este campo sirve para distinguir a ese nivel superior de otros. Por ejemplo, IP a UDP o TCP, este atributo identificaría si el paquete iría destinado al nivel UDP o al TCP. Si el paquete es de salida y el nivel inferior necesita un identificador de tipo o algo parecido, se le pasa a través de este atributo (debe haberse registrado mediante el método setID() del nivel).
`Direccion direccion`	Dirección de destino al que debe enviarse el paquete.
`Interfaz interfaz`	Interfaz por donde entró el paquete al equipo (o por la que salió de él).
`Red red`	Ultima red por la que circuló el paquete. Muy útil en el funcionamiento de los puentes, por ejemplo.
`bolean fragmentable`	Flag que indica si el paquete de datos se puede dividir en otros más pequeños para ser enviado. Útil si se quiere simular la fragmentación y el reensamblado de datagramas a nivel IP.

Tabla 24. Atributos de la clase Dato.

Al tratarse de una estructura de datos, en la que muchos de sus atributos cambiarán durante el transcurso de la simulación, no se han implementado métodos consultores ni modificadores (get/set), porque se ha optado por hacer todos los atributos visibles desde fuera de la clase. En cambio, se han implementado varios tipos de constructores para facilitar la creación de este tipo de objeto, que es muy usado durante el proceso de simulación.

El hecho de que todos los atributos sean de tipo público (visibilidad pública) implica que se debe tener especial cuidado cuando se maneje este tipo de objeto en los equipos o redes nuevas que se implementes, así como en los módulos de la pila de comunicaciones, ya que no se comprueba, automáticamente (en las modificaciones) que los valores nuevos para los atributos sean coherentes.

También es importante no modificar el contenido del atributo *paquete* ya que no se utiliza copia alguna del pasado a los constructores, se utiliza directamente ese. El usuario es el encargado de hacer las copias si procede.

3.3.1.5 Clase utilidad LocalizadorRedes

Esta clase es la equivalente a *LocalizadorEquipos*, pero para las clases relacionadas con las redes y la interfaces. De forma similar, cuando tenemos un nuevo tipo de red en el sistema de simulación, debemos registrarla en el localizador, para que pueda ser cargada posteriormente en tiempo de ejecución. También permite la creación de los distintos tipos de interfaces.

Atributo de la clase	Descripción
`Vector Clases`	Vector de clases asociadas a las redes (clases que definen el comportamiento de las redes)
`Vector IDs`	Vector de identificadores de los módulos de la pila de comunicaciones.

Tabla 25. Atributos de la clase LocalizadorRedes.

Método	Descripción
`Buffer`	Constructor. Se puede crear un objeto Buffer vacío o basándose en otro Buffer, en una cadena de texto, o con una longitud determinada.
`getbit/setbit`	Recupera o modifica el bit especificado de un determinado byte.
`getbyte/setbyte`	Recupera o modifica el byte especificado.
`getByteH/setByteH`	Recupera o modifica la parte alta de un byte.
`getByteL/setByteL`	Recupera o modifica la parte baja de un byte.
`Tam`	Devuelve el tamaño del buffer
`Contenido`	Devuelve un mensaje con la información relativa al buffer (dividida en campos).

Tabla 26. Métodos de la clase LocalizadorRedes.

Además sirve como gestor de identificadores para los módulos de la pila de comunicaciones, es decir, cuando un módulo de la pila, por ejemplo, el módulo de nivel de enlace Ethernet, quiere saber a que módulo tiene que enviarle los datos que le han llegado en una trama que posee el campo *type* con el valor *0x0800*, analiza la lista de identificadores que posee la clase *LocalizadorRedes*, obtiene que el módulo destino es aquel cuyo identificador es *ipv4*, y busca un nivel superior suyo que posea dicho identificador. El registro de estos identificadores da igual donde se realice dentro del código, aunque se ha decidido hacerlo en las clases superiores, es decir, en el ejemplo anterior, sería el nivel IP el encargado de registrar para cada posible nivel inferior, el identificador que usa ese nivel inferior para referirse a IP (ver código. Esto puede parecer un problema si se introducen nuevos módulos y hay que registrar nuevos niveles inferiores, pero no es así, ya que el registro de estos identificadores se

puede hacer en cualquier para, eso sí, siempre antes de iniciar una simulación, ya que si no es así, las conexiones entre niveles de la pila no funcionarán adecuadamente.

```
static
{
    // El id en IPv4 para ethernet es 0x0800...
    LocalizadorRedes.Registrar("ipv4","ethernet",0x0800);
    LocalizadorRedes.Registrar("ipv4","puntoapunto",0x0001);
    LocalizadorRedes.Registrar("ipv4","anillo",0xFFFF);
    LocalizadorRedes.Registrar("ipv4","ipv4",0x0000);
}
```

Código 12. Inicializador de la clase NivelIPv4.

Del listado de código anterior podemos ver que el registro de nuevos identificadores es muy sencillo. El hecho de que el código se encuentre en el inicializador de la clase hace que no se produzcan múltiples registros para los mismos identificadores.

3.3.1.6 Clase utilidad IDNivel

Los identificadores de los que habla el punto anterior están formados por tres elementos:

Atributo de la clase	Descripción
`String nivel`	Nivel al que se van a referir los niveles inferiores con el *codigo*.
`String nivel_inferior`	Nivel que usará el *codigo* para referirse al *nivel*
`int codigo`	Código numérico que usará en *nivel_inferior* cuando quiera hacer referencia al *nivel*.

Tabla 27. Atributos de la clase IDNivel.

Al tratarse de una estructura de datos, solo tiene un constructor, con el que se inicializan los tres atributos (de visibilidad pública) de la tabla anterior.

3.3.1.7 Clase base Red

Las redes, junto con los equipos, son los componentes básicos de cualquier simulación, y que el usuario de la API usará para montar el sistema. Está formado por cinco componentes:

Atributo de la clase	Descripción
`Vector ListaEquipos`	Lista de objetos de tipo Equipo conectados a la red.
`Vector ListaRedes`	Lista de objetos de tipo Red conectados a la red (si se permite la conexión con otras redes),
`int MTU`	MTU de la red.
`Vector colaTramas`	Lista de tramas que se transmitirán por la red.
`Vector Interfaces`	*Lista de interfaces con las que los equipos se conectan a la red. Atributo heredado de la clase Objeto.*

Tabla 28. Atributos de la clase Red.

La lista de equipos se utiliza cuando la red necesita enviar una trama de datos a los equipos (a uno o varios), para programarles las entradas. La lista de redes se usa cuando hay que retransmitir tramas entre redes. La MTU es un atributo importante de una red, y que se puede cambiar en tiempo de ejecución para hacer pruebas de fragmentación de paquetes. Esta funcionalidad, de cambiar la MTU de una red, se ha incluido sobretodo con fines didácticos. La cola de tramas es la estructura de datos donde se almacenan las tramas que envían los equipos a la red o que retransmiten las otras redes.

A la hora de implementar nuevos tipos de red y nuevos tipos de equipos, hay que conocer bien como funciona la clase *Red*. Existe un método que es el que deben usar los equipos para enviar tramas a la red y otro distinto que deben usar las otras redes para retransmitir tramas a la red. Los métodos de la clase son los siguientes:

Método	Descripción
`Enviar`	Envía una trama a la red. Es usado por los equipos cuando quieren enviar datos a la red. Es similar al método *ProgramarEntrada* de los equipos.
`Retransmitir`	Envía una trama a otra red. Lo usan otras redes cuando quieren enviar algo a esta red. Similar al método *ProgramarEntrada* de los equipos.
`NumEquiposConectados`	Devuelve el número de equipos conectados a la red.
`getMTU/setMTU`	Recupera y modifica la MTU de la red.
`Conectar`	Conecta la red a otra red. Por defecto no se permiten conexiones directas entre redes y este método lanza una excepción. En el caso de que se puedan conectar directamente dos redes, se deberá redefinir este método como se hace en la clase *HubEthernet*. (Asociado al método *ConectarA*).

`ConectarA`	Conecta la red a un equipo por la interfaz especificada del equipo. Este método se usa exclusivamente en el método *setInterfaz* de la clase *Equipo* para crear el enlace al equipo en la lista de equipos de la red de forma automática. *No usar en otro caso.*
`ConectarA`	Conecta la red a otra red. La usan exclusivamente otras redes cuando quieren conectarse a esta red. *No usar en otro caso.*
`SoportaARP`	Devuelve *cierto*, ya que por defecto todas las redes soportan ARP. En el caso de redes que no lo soporten como las redes punto a punto, hay que redefinir este método para que devuelva el valor *falso*, y el módulo IP de la pila de comunicaciones no envíe peticiones ARP.

Tabla 29. Métodos de la clase Red.

La forma de conectar redes con redes no es demasiado clara, por el hecho de que se ha intentado evitar que se conecte una red de tipo *Ethernet* por ejemplo, con otra de tipo *PuntoAPunto* (cosa que no se ha conseguido plenamente, aunque si se usa la API correctamente no tiene por qué suceder algo así). En el caso extremo de que se llegasen a conectar redes de tipos incompatibles, no pasaría nada, porque una red solo procesa las tramas de tipo conocido y el resto las ignora. Para permitir que una red se pueda conectar a otra lo que se hace es redefinir el método *Conectar*, que a su vez utilizará el método *ConectarA* para realizar la conexión entre las redes en los dos sentidos, es decir, conectar la red 1 con la red 2, y la red 2 con la red 1, con solo una llamada al método *Conectar* de una de las redes. Este automatismo en las conexiones tiene un precio: la complejidad interna de las clases aumenta, pero también tiene una ventaja muy importante: a la hora de utilizar la API para simular, el montaje de las redes se simplifica mucho y se reduce la posibilidad de error (conexiones olvidadas por ejemplo).

Cuando la conexión se quiere realizar entre un equipo y una red, lo único que hay que hacer es conectar la interfaz a la red y luego añadir la interfaz al equipo. Cuando se llama al método *setInterfaz* de la clase *Equipo*, se llama al método *ConectarA* de la clase red y se añade el equipo a la lista de equipos de la red de forma transparente para el usuario.

3.3.1.8 Clase Nivel

Esta es la clase de la que derivan todos los módulos de la pila de comunicaciones, e implementa la parte de simulación de errores, gestión de enlaces entre niveles y la programación de datos como entradas y salidas. También define una interfaz común. El esquema básico de un módulo de la pila se puede ver en la *figura 5*, y las estructuras de datos que se han usado para su implementación se pueden ver en la siguiente tabla:

Atributo de la clase	Descripción
`Vector colaSalida`	Vector en el que se almacenarán todos los *datos* (objetos de la clase *Dato*) que deban ser enviados a un nivel inferior o a una red (si es el caso de un nivel de enlace).
`Vector colaEntrada`	Vector en el que se guardaran los *datos* que lleguen al nivel desde un nivel inferior o desde la red
`Equipo equipo`	Enlace al equipo poseedor de la pila de comunicaciones a la que pertenece el nivel.
`Vector nivelesInferiores`	Vector con todos los niveles inferiores a este nivel.
`Vector nivelesSuperiores`	Vector con todos los niveles superiores a este nivel.
`Vector Errores`	Lista con los identificadores de los errores que se deberán simular.
`ListaParametros parámetros`	Parámetros de funcionamiento del nivel, como puede ser el retardo de procesamiento de un paquete…

Tabla 30. Atributos de la clase Nivel.

La gestión de errores a simular se lleva a cabo usando el vector *Errores* y dos métodos con el mismo nombre: *SimularError*, uno para consultar si esta activada la simulación de un determinado error, y otro para activar o desactivar dicha simulación.

La gestión de los enlaces entre los distintos niveles es algo compleja, ya que incluye toda la gestión de los identificadores de protocolos (ver clase *IDNivel* más arriba). Desde el punto de vista del programador de nuevos niveles existen cuatro métodos importantes: *getNivelSuperior* / *getNivelInferior* que a partir de un identificador de modulo y protocolo devuelven un objeto de tipo *Nivel*, y *setNivelSuperior* / *setNivelInferior* que sirven para crear los enlaces de un nivel con sus niveles superiores e inferiores respectivamente. Si profundizamos más en la implementación de estos métodos descubrimos tres métodos que trabajan directamente con objetos de la clase *IDNivel* e identificadores de niveles: *Soporta*, para ver si un determinado módulo soporta un protocolo, *getID* que devuelve el identificador de un módulo e *ID* que devuelve el identificador propio del módulo. Estos tres últimos métodos están muy relacionados con el registro de identificadores que se comentaba en la descripción de la clase *LocalizadorRedes*.

Además de estos métodos, existen otros cuatro que son de obligada implementación para todas las clases derivadas de la clase Nivel, y que son las que realmente implementan el comportamiento de un módulo de la pila de comunicaciones, son: *Procesar*, *Pendientes*, *ComprobarSalida* y *ComprobarEntrada*. Los dos primeros son idénticos a los que debían implementar las clases derivadas de *Equipo* y *Red*, y su finalidad es la misma; tienen los mismos nombres para conseguir una interfaz de programación lo más homogénea y fácil de recordar posible. Los dos últimos sirven para comprobar que los datos de salida y entrada están bien formados.

Método	Descripción
`setNivelInferior / getNivelInferior`	Enlaza (o recupera) un nivel inferior.
`setNivelSuperior / getNivelSuperior`	Enlaza (o recupera) un nivel superior.
`ID`	Devuelve el identificador del módulo.
`getID`	Devuelve el identificador con el que otro módulo (parámetro) se referirá a este.
`Soporta`	Comprueba si el módulo soporta un determinado protocolo.
`SimularError`	Activa/Desactiva la simulación de un determinado error en el módulo o devuelve si un error se va a simular o no (*método sobrecargado*)
`ComprobarEntrada`	Comprueba que un dato de entrada es correcto.
`ComprobarSalida`	Comprueba que un dato de salida es correcto.
`Procesar`	Procesa todos los paquetes de datos que estén programados para un determinado instante.
`Pendientes`	Devuelve el número de paquetes que están pendientes de ser procesados.

Tabla 31. Métodos de la clase Nivel.

3.3.2 Clases asociadas a los módulos de la pila de comunicaciones

Dentro del paquete Redes también se encuentran los subpaquetes que implementan cada uno de los módulos de la pila de comunicaciones. Para crear un módulo solo estamos obligados a crear una clase derivada de la clase abstracta Nivel, descrita en el punto anterior, pero con el fin de conseguir un código estructurado y coherente se recomienda implementar una clase derivada de *Nivel*, que implemente el comportamiento del módulo de la pila, una clase que implemente el paquete de datos que use el módulo en sus comunicaciones y una clase que agrupe los flags asociados a los errores que se puedan simular. Tomando como ejemplo los módulos IP, ARP e ICMP que se han implementado tenemos que para IP estas clases son *NivelIPv4*, *DatagramaIPv4* y *ErroresIPv4*; para ARP son *ModuloARP*, *PaqueteARP* (con sus derivadas *PeticionARP* y *RespuestaARP*) y *ErroresARP*; y finalmente para ICMP son *ModuloICMP*, *MensajeICMP* y *ErroresICMP*. La implementación es muy similar en todas ellas, salvando las diferencias entre cada uno de los distintos módulos. El caso del comportamiento del módulo IPv4 (clase *NivelIPv4*) es algo distinto debido a la gran complejidad del protocolo IP, y de describirá brevemente a continuación. Además de estas clases, en cada módulo pueden ser necesarias otras más como la clase *TablaDeRutas* en IP, o la clase *CacheARP* en ARP, no hay ningún problema en

implementar cuantas clases sean necesarias dentro de cada subpaquete asociado a cada módulo.

3.3.2.1 Clase ErroresICMP

Esta clase contiene dos listas: una de los nombres de los flags asociados a los errores ICMP que se podrán simular, y otra con una descripción de cada uno de ellos. Estos flags se utilizarán en la clase que implemente el comportamiento del módulo de la pila para saber cuando hay que simular los errores y cuando no.

Su estructura es muy simple:

```
public class ErroresICMP
{
    /**
     * Flags asociados a los errores que se pueden simular
     */
    public static String[] flags={
        "IGNORAR_ICMP_3_2",
        "IGNORAR_ICMP_REPLY"
    };


    /**
     * Descripcion de los errores que se pueden simular
     */
    public static String descripcion[]={
        "Ignora los mensajes ICMP Protocolo Inalcanzable",
        "Ignora los mensajes ICMP Echo Reply"
    };
}
```

Código 13. Implementación de la clase ErroresICMP

3.3.2.2 Clase PaqueteARP

En esta clase se implementa el tipo de paquete de datos que utiliza el módulo ARP para enviar y recibir información. Es una implementación directa de la RFC 826, lo cual es un ejemplo del realismo del que se ha querido dotar a la simulación. En dicho RFC se define que un paquete ARP debe tener la siguiente estructura:

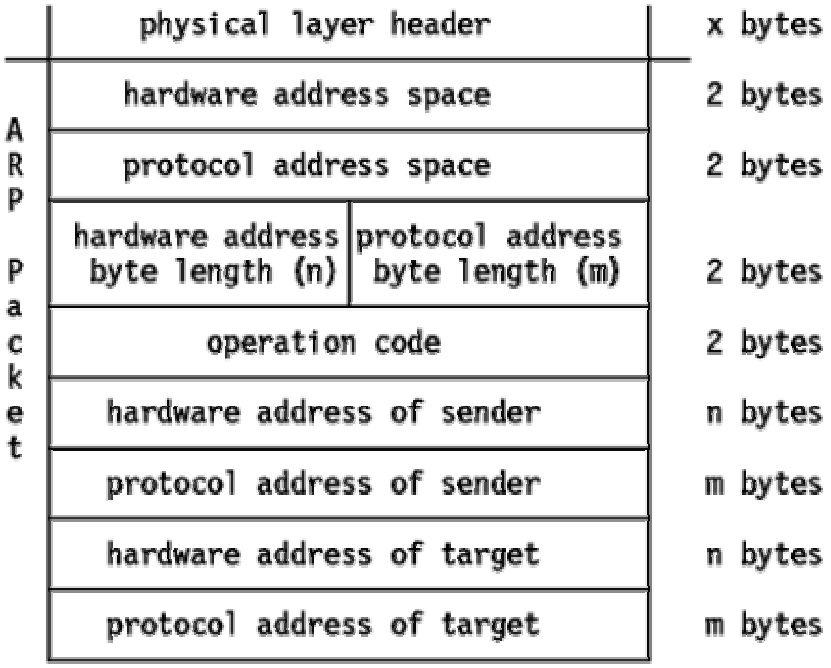

Figura 10. Estructura de los paquetes ARP.

En la implementación existen métodos del tipo get/set para cada uno de los distintos campos definidos en la *figura 10*. Estos métodos se basan en los que se definen en la clase base *Buffer* y que permiten acceder y modificar a nivel de bit (o byte) el contenido del paquete. Un ejemplo lo tenemos en el siguiente listado:

```
public void setHWAddressSpace(int tipo)
{
   // 0. Comprobacion
   if(tipo<0 || tipo>65535)
      throw new IllegalArgumentException("Espacio de direcciones
                                          hardware invalido");

   setByte(0,tipo>>8);
   setByte(1,tipo&0x00FF);
}


public int getHWAddressSpace()
{
   return((getByte(0)<<8)+getByte(1));
}
```

Código 14. Ejemplo de los métodos de la clase PaqueteARP

3.3.2.3 Clase NivelIPv4

Esta es la clase donde se implementa todo lo asociado con el módulo IP de la pila de comunicaciones y el comportamiento del protocolo IPv4. Como ya se ha comentado anteriormente, no se ha implementado la parte de IP que trata las 'opciones IP' porque complicaba demasiado el sistema, aunque es posible hacerlo.

Como en un módulo real, existen enlaces directos a los módulos ICMP y ARP, una lista en la que almacenamos los datagramas que hay que enviar pero están a la espera de recibir respuestas ARP y otra donde se almacenan los fragmentos de los datagrama que van siendo reensamblados.

Atributo de la clase	Descripción
`TablaDeRutas tablaDeRutas`	Tabla de rutas que se usará para el encaminamiento de los datagramas.
`Vector Reensamblados`	En este vector se almacenarán los datos asociados al reensamblaje de los datagramas y los fragmentos de los mismos.
`ModuloARP moduloARP`	Módulo ARP que usará IP para resolver las direcciones.
`ModuloICMP moduloICMP`	Módulo ICMP que usará IP para enviar alertas y mensajes de error.
`Vector enEspera`	Los datagramas que tengan que esperar una respuesta del módulo ARP se guardarán temporalmente en este vector.
`int instanteActual`	Instante de tiempo actual dentro de la simulación.

Tabla 31. Atributos de la clase NivelIPv4.

Además de estos atributos, el módulo IP tiene una serie de parámetros de funcionamiento que permiten ajustar algunos tiempos de respuesta y retardos. Sus valores por defecto se ajustan en el constructor de la clase. Son lo siguientes:

Parámetro	Descripción
`ARP Timeout`	Tiempo que debe transcurrir desde que se envía una petición ARP hasta que se envíe un mensaje de error de ARP Timeout porque no se reciba la correspondiente respuesta ARP. Su valor por defecto es 30 unidades de tiempo del simulador.
`IP Reassembly Timeout`	Tiempo que transcurre desde que se recibe un fragmento de un datagrama nuevo, hasta que se envía un mensaje de error ICMP porque no se han recibido más fragmentos. Valor por defecto: 150 unidades de tiempo del simulador.

`IP Forwarding`	Permite la redirección del tráfico IP cuando no va dirigido a esté equipo. Su valor por defecto es: false (deshabilitado).
`Retardo`	Retardo de funcionamiento de IP, es decir tiempo que pasa desde que se empieza a procesar un datagrama y se envía a otro nivel. Este tiempo puede variar, en función del proceso que se realice (peticiones ARP, reensamblaje…), pero su valor por defecto es 1 unidad de tiempo. *Todos los módulos deberían definir este parámetro.*

Tabla 32. Parámetros de funcionamiento del Nivel IPv4.

El esquema, bastante simplificado, del funcionamiento del módulo se puede ver en la siguiente figura:

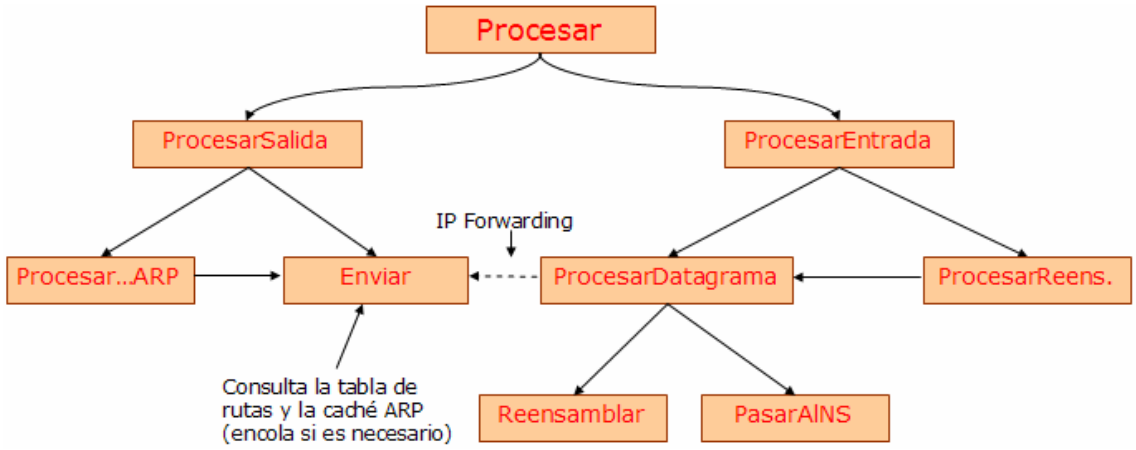

Figura 11. Esquema del funcionamiento del módulo IP.

Cuando hay programada una salida (envío de información hacia la red) se llama al método *ProcesarSalida* que a su vez obtiene la información de los datagramas que hay que enviar y se la pasa al método *Enviar* que es el encargado de hacer que lleguen a su destino, después de esto también se comprueba si existen datagramas que estén a la espera de alguna respuesta ARP que ya se haya recibido y también se envían (método *ProcesarEnEsperaARP*).

Cuando lo que hay programado es una entrada de datos (recepción de un datagrama), se llama al método *ProcesarEntrada* que obtiene la información de los datagramas y se los pasa al método *ProcesarDatagrama* para que realice las operaciones pertinentes (pasarlo a un nivel superior, reenviarlo, …), además comprueba el estado de la lista de datagramas que están siendo reensamblados (método *ProcesarReensamblados*) por si hay alguno que se haya completado y se deba enviar al método *ProcesarDatagrama*. Cuando se recibe un datagrama puede darse el caso de que sea un fragmento y se deba pasar al método *Reensamblar* para que controle el reensamblaje o que se deba pasar al nivel superior correspondiente mediante una llamada al método *PasarAlNivelSuperior*.

El núcleo del comportamiento es el que se implementa en los métodos *Enviar*, *ProcesarDatagrama* y los auxiliares que utilicen, ya que es ahí donde se desencadena todas las acciones que realiza un módulo IP. En el código fuente están documentados todos los algoritmos que se han utilizado en el módulo, además se pueden consultar el RFC 791 sobre IP, y el 792 sobre ICMP, ya que en ellos se basa esta implementación.

3.3.3 Clases asociadas a las redes

Algunos ejemplos significativos de la implementación de los dos tipos de redes que se han creado se muestran en los siguientes apartados.

3.3.3.1 Redes Ethernet

El tipo de red Ethernet que se ha implementado es el Ethernet DIX, que es más fácil de implementar que el aparecido posteriormente y conocido como IEEE 802.3. El esquema de clases, basándonos en la arquitectura mostrada en la *figura 3* es la siguiente:

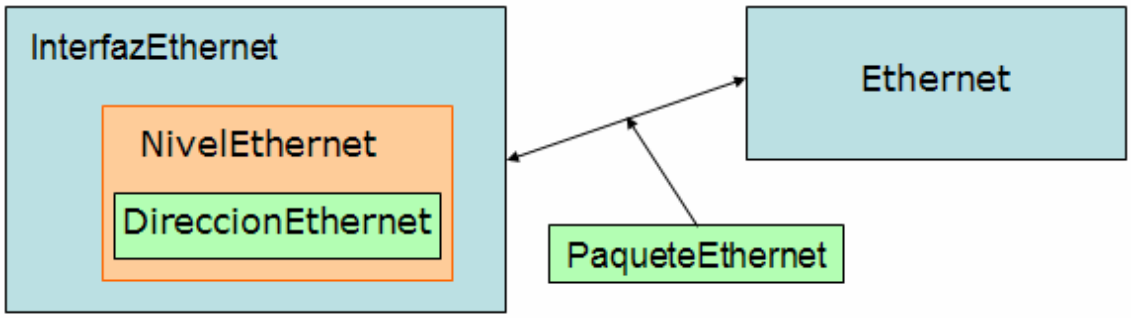

Figura 12. División en clases de la implementación de las redes Ethernet

El nombre de estas clases puede ser cualquiera, excepto el nombre de la clase que implementa el comportamiento de la red, que debe ser idéntico al nombre del subpaquete dentro del paquete *Redes*: en este caso como estamos dentro del paquete *Ethernet*, la clase que implementa el comportamiento se debe llamar *Ethernet*. Esto se debe a que es posible cargar estos componentes de forma dinámica (mediante la clase *LocalizadorRedes*).

En el método principal, *Procesar*, lo que se hace es comprobar que objetos de tipo *Dato*, de *colaTramas*, están programados para ser procesados en el instante actual; en ese caso se eliminan de *colaTramas*, y se envían a todos los equipos de *ListaEquipos* (menos al equipo que envió el dato a la red) y a todas las redes de *ListaRedes*. Este es el comportamiento típico de la redes de difusión.

Además se implementa el método *Pendientes*, que devuelve el número de datos pendientes de ser procesados y el método *Caracteristicas* que devuelve la lista de características generales de las redes Ethernet.

Como los métodos *Enviar* y *Retransmitir* no hacen mas que añadir los datos en *colaTramas* no es necesario implementarlos ya que los genéricos implementados en la clase *Red* nos sirven, pero en el caso de que se necesitase algún procesamiento adicional, como sucede en la clase *SwitchEthernet*, donde es necesario llevar un

control de los puertos por los que se retransmiten las tramas, habría que reimplementarlos.

3.3.3.2 Redes PuntoAPunto

Las redes de tipo punto a punto que se han implementado no equivalen a las redes PPP debido a la gran complejidad del protocolo PPP. Este es un tipo de red que permite conectar dos equipos de forma fácil y con un protocolo de nivel de enlace trivial. Se puede ver este tipo de red como un ejemplo básico de implementación de redes para el sistema de simulación.

El esquema de clases desarrollado es el siguiente:

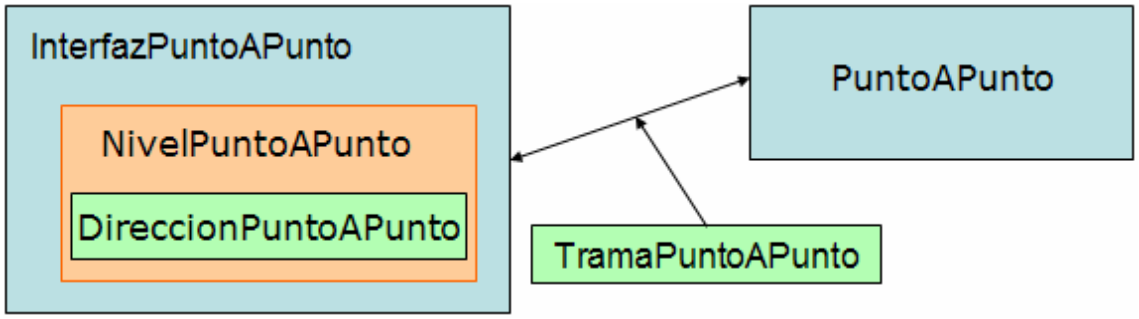

Figura 13. División en clases de la implementación de las redes PaP

En este tipo de red se envían los datos directamente al equipo destino (es decir, el equipo que no envía la trama) y solo se permiten dos equipos conectados a la red, si se diera el caso de que se añadiesen más no se enviaría ninguna trama a ningún equipo (la red dejaría de funcionar).

Un dato importante de este tipo de redes es que no usan ARP y para ello se redefine el método *SoportaARP* para que devuelva el valor *false*, y que el módulo IP no use ARP sobre este tipo de red.

3.4 Esqueletos de código

Los siguientes esqueletos de código que pueden servir de guía para extender la funcionalidad del simulador con nuevos tipos de equipos, redes o módulos de la pila de comunicaciones. También se incluyen los ficheros en formato Java para facilitar, en lo posible, el proceso de implementación.

3.4.1 Nuevos equipos

A continuación se muestra el código base para la implementación de nuevos equipos para ser usados en el simulador:

```
package Equipo;

import Redes.*;
import Redes.IPv4.*;
import Redes.IPv4.ICMP.*;
import Redes.IPv4.ARP.*;
import Proyecto.*;

public class NuevoEquipo extends Equipo
{
    private static ListaParametros caracteristicas;

    static { caracteristicas=new ListaParametros(); ... }

    public NuevoEquipo() { }

    public void setInterfaz(Interfaz interfaz) { }

    public void Procesar(int instante) { }

    public int Pendientes() { }

    public void ProgramarEntrada(Dato dato) { }

    public void ProgramarSalida(Dato dato) { }

    public boolean SimularError(int nivelID,String flag,boolean act) { }

    public void ConfiguraPila(int nivelID,String param,Object valor) { }

    public void ConfiguraPila(String nomIfaz,String param,Object val) { }

    public ListaParametros Caracteristicas() { return(caracteristicas); }
}
```

Código 15. Esqueleto para la implementación de equipos.

Al menos se deben implementar esos métodos ya que forman la interfaz de programación común a todos los equipos. Además, casi todos ellos, son de implementación obligatoria ya que provienen de las clases bases abstractas (*Equipo* y *Objeto*).

3.4.2 Nuevos Interfaces

Los nuevos interfaces se crearán dentro del paquete asociado a la red que pertenezcan.

```
package Redes.NuevaRed;

import Equipos.Equipo;
import Redes.*;
import Proyecto.Parametro;

public class InterfazNuevaRed extends Interfaz
{
    public InterfazNuevaRed(String nombre, String ip, String mascara,
              String dirEnlace) throws IllegalArgumentException { }

    public void CreaNivelEnlace(Equipo equipo) { }
}
```

Código 16. Esqueleto para la implementación de interfaces.

3.4.3 Nuevas Direcciones

Las direcciones, como mínimo deben implementar el constructor a partir de una cadena de texto, el constructor de copia y el método para devolver la dirección en formato de cadena de texto (recomendable, pero no imprescindible).

```
package Redes.NuevaRed;

import Redes.Direccion;

public class DireccionNuevaRed extends Direccion
{
    public DireccionNuevaRed(String direccion)
        throws IllegalArgumentException { }

    public DireccionNuevaRed(Direccion direccion)
        throws IllegalArgumentException { }

    public String toString() { }
}
```

Código 17. Esqueleto para la implementación de direcciones.

3.4.4 Nuevos Niveles de la pila de comunicaciones

Los métodos *ProcesarSalida* y *ProcesarEntrada* del siguiente listado no son estrictamente necesarios aunque si muy recomendables ya que aclaran mucho el funcionamiento del módulo.

```
package Redes.NuevaRed;

import Redes.*;
import Equipos.*;
import Proyecto.*;

public class NivelNuevaRed extends Nivel
{
    public NivelNuevaRed(Equipo equipo, Red red,
                         DireccionNuevaRed direccion) { }

    public void Procesar(int instante) { }

    public int Pendientes() { }

    private void ProcesarSalida(int instante) { }

    private void ProcesarEntrada(int instante) { }

    public String ID() { }

    public boolean ComprobarEntrada(Dato dato) { }

    public boolean ComprobarSalida(Dato dato) { }
}
```

Código 18. Esqueleto para la implementación de módulos de la pila.

3.4.5 Nuevas Redes

La clases que implementen el comportamiento de las redes son un componente crítico ya que en su inicializador estático definen cuales son las clases asociadas a cada uno de los demás elementos de la red mediante la lista de características de la red. Como ya se ha visto anteriormente, el método *SoportaARP* solo se debe redefinir si el tipo de red no soporta el protocolo ARP, como sucedía con las redes *PuntoAPunto*.

```
package Redes.NuevaRed;

import Redes.*;
import Equipos.Equipo;
import Proyecto.ListaParametros;
import Proyecto.Parametro;

public class NuevaRed extends Red
{
    static { }

    public NuevaRed() { }

    public void Procesar(int instante) { }

    public int Pendientes() { }

    public ListaParametros Caracteristicas() { }

    public SoportaARP() { }
}
```

Código 19. Esqueleto para la implementación de redes.

3.4.6 Nuevos tipos de tramas

La implementación de los nuevos tipos de tramas no está sujeta a restricciones, solo que debe definir el constructor y el constructor de copia, aunque se recomienda definir métodos para acceder y modificar la información de cada uno de los campos en que esté definida la trama:

```
package Redes.NuevaRed;

import Redes.*;

public class TramaNuevaRed extends Buffer
{
    public TramaNuevaRed(DireccionNuevaRed origen,
                         DireccionNuevaRed destino, Buffer p, int tipo)
                         throws IllegalArgumentExceptio { }

    public TramaNuevaRed(Buffer trama) { super(trama); }
}
```

Código 20. Esqueleto para la implementación de tramas.

4. Ejemplos de utilización

El uso de la API de simulación es bastante simple ya que se reduce a:

- Crear las redes.
- Crear los equipos sus interfaces, así como configurar sus tablas de rutas.
- Añadir los equipos y las redes al simulador.
- Crear los datos y programarlos en los instantes que se deseen.
- Ajustar la simulación de errores en cada equipo.
- Iniciar la simulación.
- Observar la lista de eventos de cada equipo y red.

A continuación se describen cada uno de los puntos anteriores y en el fichero Ejemplos.java se pueden encontrar más de 30 ejemplos más.

4.1 Creación de redes

Se lleva a cabo mediante llamadas al método *New* de la clase *LocalizadorRedes*, especificando el nombre completo (con el nombre del paquete) de la clase que implementa el comportamiento de la red junto al nombre que se le desee dar a dicha red. También se puede crear llamando directamente al constructor de la clase en cuestión, pero al hacerlo así, nos obligamos a '*import*'ar las clases y paquetes asociados a la red. Algunos ejemplos son los siguientes:

```
Red ethernet1=LocalizadorRedes.New("Ethernet.Ethernet","ethernet1");
Red ethernet2=LocalizadorRedes.New("Ethernet.Ethernet","ethernet2");
Red switch=LocalizadorRedes.New("Ethernet.SwitchEthernet","switch");

Red mired=new PuntoAPunto();
mired.setNombre("mired");
mired.setMTU(5000); // ¡muy importante no omitir esta llamada!
```

Código 21. Ejemplo de creación de redes.

4.2 Creación de los equipos y las interfaces

De forma similar a lo que sucede con las redes, los equipos pueden ser creados llamando al método *New* de la clase *LocalizadorEquipos*, especificando el nombre de la clase (no hace falta el paquete porque es el mismo para todos (paquete *Equipos*)) y el nombre del equipo. También se pueden crear llamando directamente al constructor de la clase.

Los interfaces se crean con llamadas al método *NewInterfaz* de la clase *LocalizadorEquipos*, pasándole toda la información de configuración, esto es: nombre de la interfaz, dirección IP y máscara, dirección de nivel de enlace y nombre completo

de la clase que define el comportamiento de la red a la que queremos conectar la interfaz.

Una vez que tenemos el equipo, la red y la interfaz, podemos conectar la interfaz a la red correspondiente y después (muy importante que sea **después**) añadir la interfaz al equipo. Por la forma en que se inicializan las estructuras internas de los tres objetos implicados en la conexión es muy importante que se conecte la interfaz primero a la red y después se añada al equipo, ya que al conectar la interfaz a la red se crea un puntero interno en la interfaz, hacia la red, que después será usado por el equipo para hacer una serie de comprobaciones para crear y enlazar correctamente el nivel de enlace asociado a la interfaz.

Finalmente añadir entradas a la tabla de rutas es tan simple como se puede ver en el listado de código siguiente:

```
// Ordenador PC1
Equipo pc1=LocalizadorEquipos.New("Ordenador","PC1");
interfaz=LocalizadorRedes.NewInterfaz("eth0", "10.1.0.3", "255.255.0.0",
                                      "00:00:00:00:00:01",
                                      "Ethernet.Ethernet");
pc1.setInterfaz(interfaz);
pc1.tablaDeRutas.Anadir("10.1.0.0","255.255.0.0","255.255.255.255","eth0");
pc1.tablaDeRutas.Anadir("0.0.0.0","255.255.255.255","10.1.0.10","eth0");

// Router R1
Equipo R1=LocalizadorEquipos.New("Router","R1");
interfaz=LocalizadorRedes.NewInterfaz("eth1", "10.1.0.10", "255.255.0.0",
                                      "00:00:00:00:00:f0",
                                      "Ethernet.Ethernet");
interfaz.Conectar(ethernet1);
R1.setInterfaz(interfaz);
interfaz=LocalizadorRedes.NewInterfaz("eth0","10.2.0.10", "255.255.0.0",
                                      "00:00:00:00:00:f1",
                                      "Ethernet.Ethernet");
interfaz.Conectar(ethernet2);
R1.setInterfaz(interfaz);
R1.tablaDeRutas.Anadir("10.1.0.0","255.0.0.0","255.255.255.255","eth1");
R1.tablaDeRutas.Anadir("10.2.0.0","255.255.0.0","255.255.255.255","eth0");
```

Código 22. Ejemplo de creación de interfaces y equipos.

El primer elemento de la tabla de rutas, es el **destino**, que puede ser una dirección IP, o si se trata de la ruta por defecto, se puede señalar con “0.0.0.0” o con “por defecto”. El segundo es la **máscara** asociada a la dirección de destino, que, en el caso de que la ruta sea la ruta por defecto, su valor puede ser cualquiera (se usa en cualquier caso la máscara “255.255.255.255”). El tercero es el **gateway** (o puerta de enlace) que puede ser una dirección IP si la ruta es indirecta, o los valores “directa”, “0.0.0.0” o “255.255.255.255” si la ruta es directa. El último parámetro es el **nombre de la interfaz** por la que se enviarán los datos dirigidos al destino indicado. Este último parámetro también puede ser un objeto de tipo *DireccionIPv4* que sea la dirección IP de la interfaz que se quiera usar.

4.3 Añadir los equipos y las redes al simulador

Para que se vaya indicando a cada componente de la simulación que debe ir cambiado de estado es necesario que exista un objeto que vaya marcando el ritmo, este objeto es de la clase Simulador, y a el hay que indicarle que componentes forman el sistema a simulador: si se trata de una red se le indicará mediante el método NuevoEquipo y si es una red, se le indicará mediante el método NuevaRed:

```
Simulador simulador=new Simulador();
simulador.MaximoNumeroDePasos(4000);
simulador.NuevoEquipo(pc1);
simulador.NuevoEquipo(pc2);
simulador.NuevoEquipo(pc3);
simulador.NuevoEquipo(router1);
simulador.NuevaRed(ethernet1);
simulador.NuevaRed(ethernet2);
```

Código 23. Ejemplo para añadir componentes a la simulación.

El número máximo de pasos se utiliza por si se crean bucles en las redes y hay siempre algún paquete de datos circulando por el sistema, para detener la simulación al cabo de un tiempo determinado.

4.4 Creación de los datos a enviar y su programación

Para desencadenar una secuencia de eventos es necesario que algunos paquetes de datos circulen por las redes y para ello hay que crearlos y programarlos en algún equipo o red. Como existen distintos tipos de paquetes de datos que se pueden crear el método puede cambiar ligeramente, pero la idea siempre es la misma. Por ejemplo: para enviar un paquete de 200 bytes con datos arbitrarios (como si se enviase algo por un socket) se haría de la siguiente forma:

```
Buffer buffer=new Buffer(200);
for(int k=0;k<buffer.Tam();k++)
    buffer.setByte(k,k%255);
Dato dato=new Dato(0,buffer);
dato.direccion=new DireccionIPv4("10.2.0.1");
dato.protocolo=0;
dato.fragmentable=true;
```

Código 24. Creación de un paquete de datos arbitrario

Como se puede ver es necesario crear un objeto de tipo Buffer que servirá de base a otro objeto, esta vez de tipo Dato que será lo que se programe en algún equipo o red. Además se especifica la dirección IP del destino, y el protocolo de destino. El

protocolo de destino sirve para el módulo IP del equipo de destino sepa distinguir a cual de sus módulos superiores debe entregar los datos cuando los reciba (los 200 bytes). También se puede especificar si IP podrá dividir los datos en varios fragmentos a la hora de enviarlos. El valor 0 en el constructor Dato, nos indica que el dato se programará para ser procesado en el instante 0.

Un ejemplo algo más sofisticado de creación de un paquete de datos es el siguiente, que consiste en la creación de un mensaje ICMP Echo con una pequeña carga de datos:

```
// Preparamos la carga de datos del mensaje ICMP Echo
Buffer buffer=new Buffer(100);
for(int i=0;i<buffer.Tam();i++)
    buffer.setByte(i,65+i);

// Montamos el mensaje ICMP Echo
MensajeICMP echo=new MensajeICMP(buffer,8,0);

Dato dato=new Dato(0,echo);
dato.direccion=new DireccionIPv4("192.168.0.2");
dato.fragmentable=true;

pc1.ProgramarSalida(dato);
```

Código 24. Creación de un mensaje ICMP Echo y su programación.

La última línea le indica al equipo pc1 que tiene algo programado como una salida de datos.

4.5 Simulación de errores

Como punto previo al lanzamiento de la simulación se deben ajustar los tipos de errores que se van a simular, y los equipos en los que se simularán. En el caso anterior, en el que el protocolo de destino es 0, se producirá un error cuando el módulo IP del destino intente entregar los datos a un nivel superior que no exista. Si queremos evitar el envío del mensaje ICMP 'Protocolo inalcanzable' lo haríamos de la siguiente forma (suponiendo que lo hacemos para dos equipos):

```
pc1.SimularError(Equipo.kICMP,"IGNORAR_ICMP_3_2",true);
pc2.SimularError(Equipo.kICMP,"IGNORAR_ICMP_3_2",true);
```

Código 25. Simulación de errores.

La lista completa de errores se puede obtener de las clases *ErroresXXX* mencionadas anteriormente.

4.6 Inicio de la simulación

Cuando todo está listo, lo único que nos queda es lanzar la simulación y ver como evoluciona el sistema. La forma más fácil es haciendo que se ejecuten todos los pasos y ver por la salida estándar los eventos producidos. Para ello lo que hay que hacer es ejecutar el siguiente fragmento de código:

```
while(simulador.SimularUnPaso());
```

Código 26. Ejecución de la simulación hasta el final.

4.7 Comprobación de las listas de eventos producidos

Si el acceso a la salida estándar no está disponible, se pueden consultar los eventos de cada componente de la simulación llamando repetidas veces al método *getEvento()* que tienen todos los equipos y redes y analizándolo posteriormente.

Para obtener los equipos y las redes se pueden usar los métodos *getEquipo()* y *getRed()* del objeto de la clase *Simulador* que ha realizado la simulación.

5. Conclusiones y posibles mejoras

El objetivo principal que se perseguía con el desarrollo de una API de programación que se pudiese utilizar para simular el comportamiento de redes IP se ha conseguido en gran medida, aunque existe una limitación importante: el número de componentes que se pueden simular y que no se ha implementado la parte de opciones IP en el módulo IP, lo cual ampliaría enormemente el número de escenarios que se podrían simular.

Otro apartado importante, es que se ha logrado desarrollar una API que es muy fácilmente utilizable por el usuario final, y que también es muy fácil de ampliar con nuevos módulos de comunicaciones, redes y equipos. Con esto tenemos un sistema básico que debería ser ampliado en un futuro (cercano) para conseguir una herramienta muy útil en entornos educativos donde los usuarios deseen comprender cual es el funcionamiento de las redes IP. El resultado de cada simulación, en forma de eventos, describe con gran detalle *cuando* se produce algún hecho remarcable en la simulación, *dónde* ocurre, y *qué* lo provoca, de modo que se ha obtenido un sistema de simulación con gran realismo.

Debido a la magnitud del proyecto, se han quedado en el tintero muchas cosas, como pueden ser la implementación de más tipos de mensajes ICMP (el Redirect, por ejemplo), varios tipos de puentes, NAT en routers, redes en anillo, implementación de los estándares del IEEE (Ethernet, TokenBus y TokenRing), incluso es posible, debido a la arquitectura abierta y modular del sistema, la implementación de módulos capaces de simular los niveles físicos de los distintos tipos de redes existentes (retardos, colisiones de tramas, distintos tipos de codificación de tramas…).

Desde estas líneas me gustaría dedicar unas palabras de agradecimiento a todos los colaboradores del Proyecto Eclipse (http://www.eclipse.org) y de la Apache Software Foundation (http://www.apache.org) por facilitarme enormemente la tarea de desarrollar un sistema multiplataforma en Java bajo el entorno GNU/Linux, y a todos aquellos que comparten sus conocimientos libremente a través de Internet, ya sea publicando gratuitamente manuales, documentos, tutoriales, opiniones, experiencias…

6. Bibliografía

[1] RFC 791, Internet Protocol. DARPA, 1981

[2] RFC 792, Internet Control Message Protocol. J. Postel, 1981

[3] RFC 826, An Ethernet Address Resolution Protocol, David C. Plummer, 1982

[4] RFC 1180, A TCP/IP Tutorial. T. Socolofsky & C. Kale, 1991

[5] RFC 950, Internet Standard Subnetting Procedure. J. Mogul & J. Postel, 1985

[6] RFC 790, Assigned Numbers, J. Postel, 1981

[7] RFC 1700, Assigned Numbers, J. Reynolds & J. Postel, 1991

[8] TCP IP Illustrated Vol 1: The Protocols, W. Richard Stevens, 1994

[9] TCP IP Illustrated Vol 2: The Implementation, W. Richard Stevens, 1995

[10] Tutorial y descripción técnica de TCP/IP,
http://ditec.um.es/laso/docs/tut-tcpip/

[11] Prácticas de Redes, F. Ortiz, F. Candelas, J. Pomares, P. Gil, L. Crespo, '02

[12] Teach Yourself TCP/IP in 24 Hours (SAMS), Joe Casad, 2003

[13] Documentación de la API de Java, Sun Microsystems Inc.

[14] Internet Protocols Handbook (Coriolis), Dave Roberts, 1996

[15] Internetworking with TCP/IP, Douglas Comer, 2000

… y muchos otros, anónimos.