UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

DISEÑO E IMPLANTACION DE UNA INTERFAZ DE RED ORIENTADA A OBJETOS

T E S I S

QUE PARA OBTENER EL TITULO DE:

M A T E M A T I C O

P R E S E N T A :

FRANCISCO JAVIER HERRERA CUADRA

MEXICO, D. F. 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

M. EN C. VIRGINIA ABRIN BATULE
Jefe de la División de Estudios Profesionales
Facultad de Ciencias
Presente

Los abajo firmantes, comunicamos a Usted, que habiendo revisado el trabajo de Tesis que realiz(ó)ron el pasante(s) Francisco Javier Herrera Cuadra

con número de cuenta 8855945-0 con el Título:

Diseño e Implantación de una Interfaz de Red Orientada a Objetos

Otorgamos nuestro Voto Aprobatorio y consideramos que a la brevedad deberá presentar su Examen Profesional para obtener el título de Matemático

GRADO	NOMBRE(S)	APELLIDOS COMPLETOS	FIRMA
Mat.	José Galaviz	Casas	
Director de Tesis			
Mat.	Monica Leñero	Padierna	
Dra.	Hanna	Oktaba	
M. en C.	Guadalupe Ibargüengoitia	González	
Suplente			
Mat.	Ana Luisa Solís	González-Cosío	
Suplente			

AGRADECIMIENTOS

" En la vastitud del espacio
y en la eternidad del tiempo,
mi alegría es compartir con ustedes
un planeta y una época ... "

Mi papá (quién más me ha apoyado y puesto ejemplos): *¿ A qué hora llegas ?*; **mi mamá** (con quién más medito y filosofo): *No te asustes si lloro !!!*; **mi chaparra** (lo mejor que me ha pasado en la vida): *Chaparro, ponte desodorante !!!*; **Juan Arturo** (el más bien intencionado): *Oye... ¿ las mujeres de más de 50 años pueden caminar ?*; **Mauricio** (el tobillitos de popote): *¿ En donde y a qué hora es el tinaco ?*;**Omar** (ese eterno impredecible): *¿ Qué..., se trata de pedalearle como BESTIAS ó qué ?*; **Armando** (mi hermano alpineador): *No se como decirlo Güero ... ¿ me prestas tus Camalots ?*; **Hans** (el más sentimental de todos): *te lo juro que ésta niña sí es diferente !!!*; **Jaime** (quién me educó bastante): *Tú ... tranquilo !!!*; **Daniel**: *¿ A qué hora jalamos ?*; **Rubén y Ricardo**: cuantas experiencias compartidas !!!; y así la lista puede seguir y seguir mencionando amigos (Españoles, Franceses, etc), abuelos, tios, primos, cuñados, suegros, compañeros, etc, etc.

Quiero agradecer a mis sinodales, en especial a José que es una de las pocas personas que conozco que tiene la capacidad de explicar Computación de manera ENTENDIBLE !!! y a Mónica por esos ratos que no me despegaba de ella para que revisara el trabajo. Agradezco también al Dr. Alberto Barajas quién alguna vez me dió un consejo para que sucediera todo esto.

En especial agradezco a DIOS por la oportunidad de vivir !!!

INDICE

PROLOGO 1

CAPITULO I .- INTRODUCCION A LAS REDES DE COMPUTADORAS 2

1.1.- Historia de las redes 2
1.2.- Clasificación de las redes 3
1.3.- La " *Arquitectura de Red* " 8
1.4.- La Red *Ethernet* 9
1.5.- La tarjeta de Red y los manejadores de Clarkson 12
Referencias 13

CAPITULO II .- DISEÑO ORIENTADO A OBJETOS 14

2.1.- Complejidad 14
2.1.1.- Atributos de la complejidad 14
2.1.2.- Aspectos de la complejidad 15
2.2.- Evolución y descripción del diseño Orientado a Objetos 17
2.2.1.- Definiciones de Programación, Diseño y Análisis Orientado a Objetos 20
2.2.2.- Elementos del modelo de Objetos 22
2.3.- Objetos y Clases 26
2.3.1.- Definición de Objeto y características 26
2.3.2.- Definición de Clase y características 26
2.3.3.- Relaciones entre Objetos 27
2.3.4.- Relaciones entre Clases 27
2.4.- El método del Diseño Orientado a Objetos 29
2.4.1.- Notación 29
2.4.2.- El proceso de diseño 32
2.4.3.- Beneficios y desventajas del Diseño Orientado a Objetos 34
Referencias 35

CAPITULO III .- DISEÑO E IMPLANTACION DE LA INTERFAZ DE RED 36

3.1.- Características de la interfaz de Red 36
3.2.- Aplicación del método e implantación 36

CAPITULO IV .- APLICACIONES DE LA INTERFAZ 50

4.1.- Programa de recepción de paquetes (RECIBE) 50
4.2.- Programa de envío de paquetes (ENVIA) 51
4.3.- Monitoreador de red (MONRED) 53
4.3.1.- Diseño e implantación 57
Referencias 65

CONCLUSIONES 66

BIBLIOGRAFIA 67

APENDICE 68

PROLOGO

El presente trabajo de tesis ha sido creado con la finalidad de diseñar e implantar una interfaz Orientada a Objetos para red *Ethernet*. El método a seguir es el descrito por Grady Booch en su libro " *Object Oriented Design*, The Benjamin / Cummings Publishing Company Inc, 1991 ".

El propósito de crearla en el paradigma Orientado a Objetos es que se tuviera una interfaz que más que utilizar algoritmos, contuviera entes que representen a los objetos reales existentes en una red *Ethernet*.

Llevar a cabo un desarrollo de esta manera no solo proporciona una interfaz implantada desde otro punto de vista y en la cual las aplicaciones que se monten tengan la filosofía Orientada a Objetos, sino que también pone a prueba a las otras interfaces desarrolladas con la filosofía tradicional, pudiendo comparar de cierta manera en cuáles aspectos es mejor la filosofía tradicional de Descomposición por Algoritmos, y en cuáles resulta mejor la de Descomposición por Objetos.

La tesis está dividida en cuatro capítulos. El Capítulo I da una breve introducción a las redes de computadoras poniendo especial énfasis en *Ethernet*; éste concluye con una breve explicación acerca de las tarjetas de red que hay, así como de los manejadores de tarjeta de Clarkson. El Capítulo II es un esbozo general del método de diseño Orientado a Objetos propuesto por Booch. El Capítulo III presenta los pasos más importantes durante el diseño e implantación de la interfaz de red. El Capítulo IV describe algunas de las aplicaciones que fueron hechas utilizando a la interfaz; esto último con el propósito de mostrar que la interfaz funciona correctamente y crear algunas aplicaciones de red que puedan ser utilizadas por el público en general; la más importante de estas aplicaciones es un monitoreador para redes *Ethernet* llamado MONRED, actualmente funcionando en el Centro de Cómputo de la Facultad de Ciencias.

CAPITULO I

INTRODUCCION A LAS REDES DE COMPUTADORAS.

1.1.- HISTORIA DE LAS REDES.

A lo largo de la historia, el hombre ha tenido la necesidad de manejar y guardar información. Con el surgimiento de los equipos de cómputo esta tarea se vió facilitada en forma sin precedente.

Hasta antes de la década de los 60´s los primeros equipos de cómputo eran grandes máquinas controladas por bulbos, las cuales sólo podian ser manejadas por verdaderos expertos y requerían de grandes sistemas de mantenimiento (enfriamiento, cuartos especiales, etc). Debido a ésto, sólo lugares muy especializados y con mucho dinero, podian contar con estos equipos. Además, las máquinas sólo podían ser manejadas por un usuario a la vez y toda la información estaba almacenada en un sólo lugar.

En la década de los 60´s con el desarrollo de los transistores se empezaron a fabricar sistemas mucho menos costosos y en los cuales había una gran computadora central, a la que se le conectaban varias terminales. Esto hizo posible que varios usuarios tuvieran acceso a la computadora de forma simultánea y comenzara a notarse la gran ventaja de que los recursos se compartieran, es decir, que todos tuvieran acceso a los programas almacenados en una sola computadora.

Sin embargo, estos equipos seguian siendo muy grandes y costosos. En el momento que surgen los circuitos integrados se genera un enorme auge por el desarrollo de sistemas mucho más pequeños y mucho menos costosos, llegando en la decada de los 80's a lo que se conoce hoy en día como PC's. Con el paso del tiempo todo el mundo pudo tener un sistema hasta en su propia casa.

Esto tuvo algunos bemoles, como el hecho de que cada sistema era independiente y por lo tanto debía tener su propio *software*, así que los recursos volvieron a ser no compartidos. Un ejemplo del gran problema que representa ésto es lo que ocurriría en las compañías que manejan una gran base de datos, la cual debe ser actualizada constantemente de modo que muchos puedan estar al tanto de todas las operaciones. Esto hace necesario que los recursos se compartan, implicando que los equipos deban estar comunicados entre sí.

Con el tiempo se hizo más urgente la necesidad de que los sistemas pudieran comunicarse y de este modo comenzaron a desarrollarse los sistemas de redes que se conocen actualmente.

1.2.- CLASIFICACION DE LAS REDES.

Las redes de computadoras pueden clasificarse de acuerdo a varios criterios :

1) Arquitectura del medio de comunicación .-

Según este criterio existen 2 clases de redes [Tanenbaum]:

- Redes de comunicación punto a punto .-

Esto es, donde cada segmento del medio de comunicación conecta a 2 anfitriones o nodos de la red. Aquí la comunicación entre 2 nodos se hace directamente si es que ambos estan conectados e indirectamente si es que se necesita pasar por otros nodos para llegar de uno a otro.

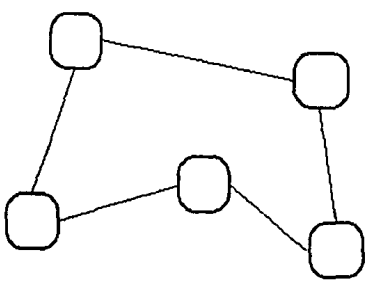

FIGURA 1.1 .- Comunicación punto a punto.

- Redes de comunicación por difusión .-

En este tipo de redes existe un único canal de comunicación, el cual es compartido por todos los nodos, es decir, si una máquina transmite un mensaje éste es recibido por todas las demás. Si una máquina recibe un mensaje que va dirigido a ella entonces lo procesará, en caso contrario simplemente lo recibirá y lo ignorará.

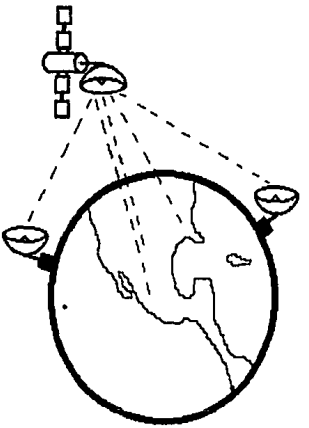

FIGURA 1.2 .- Comunicación por difusión.

2) Tamaño de la red .-

Esta clasificación está basada en la distancia máxima entre 2 nodos. De este modo se tienen :

Redes de Cobertura Local (LAN's)

Redes de Cobertura Extensa (WAN's)

Redes de Cobertura Metropolitana (MAN's)

No existe un patrón fijo que determine si una red es LAN o WAN, pero los criterios más usados fluctúan entre 1 y 10 Km, así que las MAN's fluctúan también dentro de ese rango. Así por ejemplo Tanenbaum propone un límite de 10 Km entre nodos para que la red sea LAN [Tanenbaum], mientras que Halsall propone 1 Km como máximo [Halsall].

La velocidad y confiabilidad de una red dependen de la cobertura de la misma, por ejemplo, en el caso de las LAN's, la velocidad fluctua entre 4 Mb/s y 2 Gb/s, con un promedio de errores de 1 bit erróneo por cada 1 x 10 E8 bits transmitidos; aunque actualmente la fibra óptica está revolucionando estas medidas, ya que es mucho más rápida y practicamente no tiene errores.

3) Topología de la red .-

Según este criterio, el cual más bien es aplicable a LAN's que a WAN's (ya que estas últimas son por lo general tan grandes, que se vuelven practicamente irregulares) existen las siguientes configuraciones [Black]:

- *Bus* o Canal :

Todos los nodos son conectados a un *Bus* común de transmisión de datos. Esta era la más popular hasta hace un par de años (Figura 1.3).

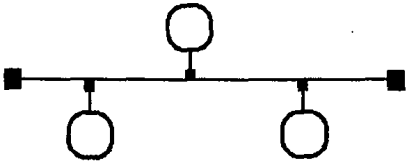

FIGURA 1.3 .- Topología de *Bus* o Canal.

- Estrella :

Está diseñada de manera que todos los nodos son conectados a un nodo central, implicando que si un mensaje se transmite de un nodo a otro, este pasará por el nodo central (Figura 1.4).

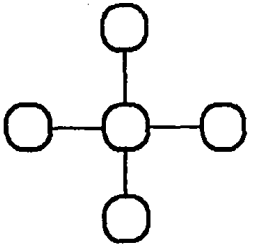

FIGURA 1.4 .- Topología de Estrella.

- Ciclo :

Es un *Bus* que tiene unidos sus extremos y donde en algunas ocasiones existe un nodo que funge como controlador (Figura 1.5).

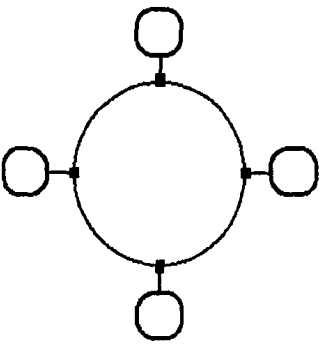

FIGURA 1.5 .- Topología de Ciclo.

- Anillo :

Esta configuración es punto a punto (Figura 1.6). Tiene la gran desventaja de que si un nodo falla la red falla.

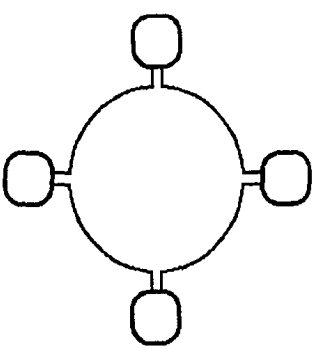

FIGURA 1.6 .- Topología de Anillo.

- Anillo 2 :

Es un anillo convencional que tiene un relevador central (Figura 1.7). Aquí si un nodo falla -a excepción del central- los mensajes son redirigidos por este último a su destino.

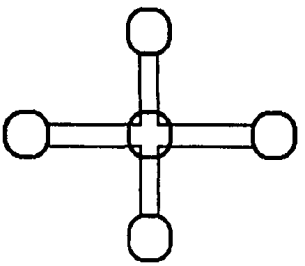

FIGURA 1.7 .- Topología de Anillo 2.

- Arbol :

En esta configuración existen nodos llamados *padres* y nodos llamados *hijos*. De un nodo *padre* se pueden derivar varios nodos *hijos*, pero cada *hijo* solo se conecta a su nodo *padre* y a los nodos *hijos* que él derive, es decir se forma una estructura jerárquica (Figura 1.8).

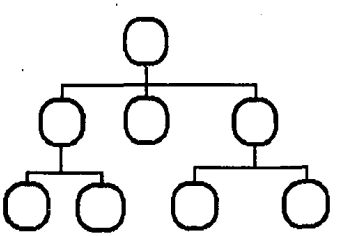

FIGURA 1.8 .- Topología de Arbol.

- Irregular :

Como su nombre lo indica no tiene forma definida. Por lo general las WAN's son redes de este tipo (Figura 1.9).

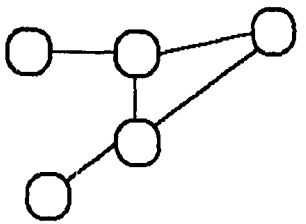

FIGURA 1.9 .- Topología Irregular.

1.3.- LA " *ARQUITECTURA DE RED* ".

Las redes de computadoras deben proveer a sus usuarios con servicios de comunicación, estos servicios son implantados de diversas maneras, pero de modo que el proceso de comunicación se efectúe de forma transparente.

Las redes han adquirido una estructura constituída por capas o niveles, donde el nivel superior es el nivel que se comunica directamente con el usuario, mientras que el nivel inferior es el que se comunica con el medio físico de transmisión de la información. Todos estos niveles se comunican con sus niveles contiguos; de esta forma la comunicación se establece pasando por cada uno de ellos hasta llegar al medio físico, de donde el nodo destino tomará los datos, para después procesarlos a través de todos los niveles de menor a mayor.

La comunicación entre niveles se hace a través de llamadas a rutinas y paso de parámetros. A todo este conjunto de niveles es a lo que se le llama *Arquitectura de Red.*

Existen actualmente varios arquitecturas de red, entre las más importantes destaca la arquitectura de ARPANET. Por otro lado, la ISO (*International Organization for Standarization*) creó un modelo para las arquitecturas de red llamado OSI (*Open Systems Interconexion*, Figura 1.10) el cual se ha adoptado como un estándar en cuanto a arquitecturas de red [Black].

CAPA	FUNCION
APLICACION	Da a los usuarios acceso al ambiente de red.
PRESENTACION	Da a la capa de aplicación medios para asegurar la integridad del significado de la información.
SESION	Provee de organización y sincronización en el intercambio de la información.
TRANSPORTE	Da servicios que aseguran la confiabilidad de la transferencia de información entre puntos terminales o interlocutores.
RED	Hace confiable la transferencia de información punto a punto. Brinda servicios para el ruteo de los paquetes.
ENLACE DE DATOS	Da los mecanismos de transferencia, con detección de errores. Agrupa la información en paquetes.
FISICA	Realiza transmisiones de cadenas de bits, a través de las interconexiones físicas de los sistemas.

FIGURA 1.10 .- Modelo de Aquitectura OSI.

1.4.- LA RED *ETHERNET.*

Ethernet es una red local (LAN) con topología de *Bus* en la cual no existen jerarquías entre los nodos, es decir, todos tienen los mismos derechos, y no existe algún nodo que cumpla con funciones de controlador.

En una configuración estándar de *Ethernet*, el segmento del *Bus* puede llegar a medir hasta 500 metros. Es posible unir hasta 5 segmentos usando repetidores [Black].

Cada nodo es conectado al *Bus* por medio de un cable, el cual se conecta a un dispositivo llamado *transceiver* que se encuentra conectado al *Bus* de comunicación. La velocidad estándar de *Ethernet* es 10 Mb/s aunque puede variar en otras configuraciones.

En la versión original, *Ethernet* usa como *Bus* de transmisión cable coaxial de 1.02 cm de diámetro, mientras que en otras versiones usa cable más delgado de aproximademente 0.5 cm. En un segmento de cable grueso es posible tener más de 100 nodos, siempre y cuando éstos se encuentren al menos a 2.5 metros. de distancia uno del otro. El cable que va de los nodos a los *transceivers* puede llegar a medir hasta 50 metros.

Al momento de la transmisión la señal viaja por el cable hasta llegar a algún extremo, donde deben colocarse terminadores, los cuales simplemente provocan que la señal no se refleje y regrese causando interferencia.

Los bits a lo largo del *Bus*, son representados por variaciones del voltaje; por ejemplo un cambio de -0.85 V a +0.85 V representa un 1, mientras que el cambio opuesto representa un 0, estas variaciones son hechas cada 100 nanosegundos, por lo que cada 100 nanosegundos se transmite un bit.

En *Ethernet* la comunicación se efectúa transmitiendo paquetes de bits. La estructura de estos es la siguiente [Tanenbaum] :

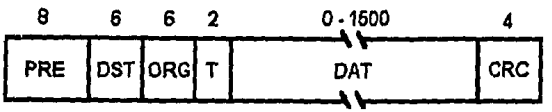

FIGURA 1.11 .- Formato de los paquetes de *Ethernet*.

Donde :

PRE .- Es el preámbulo, esto es, una secuencia de 64 bits que comienza con 1's y 0's alternados y al final dos 1's seguidos. Esto es con fines de sincronización.

DST, ORG .- Las direcciones en *Ethernet* son números de 48 bits. Cada tarjeta tiene asociada una dirección única en el mundo que por lo general se almacena en ROM, así que DST y ORG guardan las direcciones destino y fuente respectivamente.

T .- Es un número de 2 bytes que identifica cual de los protocolos que se montan sobre *Ethernet* fue el que envió el paquete.

DAT .- Este es el campo de datos, que puede ser desde 0 hasta 1500 bytes.

CRC .- Es el Código Cíclico de Redundancia, el cuál sirve para la detección de errores después de la transmisión, ya que al recibir un paquete, se extrae la cadena de bits desde el inicio del mismo hasta el final del campo de datos y se calcula nuevamente el CRC, el cual se compara con el recibido, deduciendo así si el paquete sufrió alteraciones durante la transmisión. Este código se obtiene de extraer los coeficientes de la división polinomial :

$$P(x) / G(x)$$

donde :

$$P(x) = x^{32} D(x)$$

y $D(x)$ es el polinomio resultante de considerar como coeficientes a la cadena de bits formada desde el preámbulo hasta el final del campo de datos; es decir que este polinomio tendrá un grado igual al número de bits considerado en esa cadena. Por otro lado $G(x)$ es el polinomio :

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Es fácil notar que dado que en *Ethernet* todos los nodos tienen los mismos derechos, se puede dar el caso en que haya 2 nodos que comiencen a transmitir simultáneamente, originándose una *colisión* de paquetes en el medio de comunicación. Para evitar esto, existe un mecanismo llamado CSMA/CD (*Carrier Sense, Multiple Access / Collision Detection*) mediante el cual es posible que los nodos detecten cuando existe una colisión. En caso de existir colisión la transmisión es abortada y al cabo de un tiempo se vuelve a intentar transmitir "procurando" no comenzar simultáneamente con otro nodo. Esto es, todos los nodos escuchan el canal, y si hay alguien hablando (transmitiendo) ningún otro habla (transmite); por otro lado, si no hay alguien hablando y dos nodos comienzan a hablar al mismo tiempo, esta situación es detectada y se transmite una ráfaga de ruido por la red, enseguida los nodos implicados se callan momentáneamente, esperando un tiempo aleatorio para comenzar a hablar de nuevo. Este algoritmo para el acceso al medio es mostrado en la Figura 1.12 [Galavíz].

```
1   S=51.2 μsegs.
2   col = 0
3   SI ( hay paquete que transmitir ) ENTONCES
4         MIENTRAS ( esté ocupado el canal )
5               espera
6         transmisión durante tiempo S
7         SI ( no colisión ) ENTONCES
8               continua transmisión
9         DE OTRO MODO
10              col = col + 1
11              SI ( col = 15 ) ENTONCES
12                    ve a 18
13              transmisión de ráfaga de ruido
14              r = 2^min(col, 10) - 1
15              elige aleatoriamente k ∈ {0,1,...,r}
16              espera tiempo t = S x k
17              ve a 4
18  FIN
```

FIGURA 1.12 .- Algoritmo para la detección de colisiones (CSMA/CD).

1.5.- LA TARJETA DE RED Y LOS MANEJADORES DE CLARKSON.

El control de la comunicación entre la computadora y el medio lo efectúa una tarjeta electrónica llamada tarjeta de red. Existen tarjetas de diversas marcas. Todas ellas se conectan a la red utilizando *transceivers* o directamente en caso de que el segmento bien sea de cable delgado o par trenzado; generalmente tienen dos puertos de salida, los cuales se usan dependiendo de como se conecte.

Las tarjetas de red son capaces de generar interrupciones al procesador, por lo que al momento de su instalación deberá asignárseles un nivel de interrupción, conocido como IRQ.

A las interrupciones del procesador generalmente se les asignan rutinas de servicio, esto es, rutinas que son llamadas cada vez que la interrupción se genera. Los manejadores de Clarkson son pequeños programas que sirven para manejar las distintas tarjetas de red que hay, con la caracteristica de que todos presentan la misma interfaz, así que una aplicación que se apega a esa interfaz no tendra que preocuparse de ser utilizada en otra máquina con tarjeta distinta.

En general, las aplicaciones de red traen interconstruidos sus propios manejadores de tarjeta, los cuales se adueñan de ésta y no permiten que otra aplicación la use, causando que si se quiere cambiar de aplicación tenga que reinicializarse el sistema. Los manejadores de Clarkson no tienen ese problema ya que como se había dicho, si una aplicación se apega a su interfaz no habrá necesidad de reinicializar el sistema al cambiarse de aplicación.

Los manejadores de Clarkson son instalados al momento de inicio del sistema. Estos se colocan como una rutina de servicio de interrupción, por lo que cada vez que se genere la misma los manejadores de Clarkson entrarán en acción.

Los manejadores cuentan a su vez con diversos servicios, tal como lo hace la interrupción 21H del sistema operativo MS-DOS. Es decir, si se quiere un determinado servicio, solo hay que generar la interrupción asociada a los manejadores, pasando los argumentos necesarios en los registros del procesador. Es de esta forma como pueden ejecutarse rutinas de envío, recepción, obtención de la dirección de la tarjeta, etc.

REFERENCIAS.

- [Black]
Black, Uyless. " *Redes de Computadoras* ". Macrobit RA-MA, 1990.

- [Galavíz]
Galavíz Casas, José. " *Diseño de Módulos para la Experimentación con Redes de Computadoras* ". Tesis de Licenciatura, 1993.

- [Halsall]
Halsall, Fred. " *Data Comunications, Computer Networks and OSI* ". Addison Wesley, 1992.

- [Tanenbaum]
Tanenbaum, Andrew S. " *Computer Networks* ". Prentice Hall, 1991.

CAPITULO II

DISEÑO ORIENTADO A OBJETOS.

2.1.- COMPLEJIDAD.

El mundo está lleno de cosas complejas, son muchos los objetos que realizan comportamientos complicados (una bacteria reproduciéndose, un motor en movimiento, un ser humano aprendiendo a leer, etc), el *software* no está exento de ésto. De aquí que es de vital importancia el estudio de la complejidad de los problemas y gracias a él se han logrado grandes avances en muchos campos, en especial en el desarrollo de sistemas de *software*.

2.1.1.- ATRIBUTOS DE LA COMPLEJIDAD.

Existen cinco atributos que son característicos de la complejidad:

1) La complejidad toma la forma de una jerarquía, es decir, está compuesta de partes interrelacionadas las cuales a su vez tienen sus propias subpartes interrelacionadas, continuando ésto hasta que se llega a un nivel donde existen sólo partes de complejidad elemental.

2) La determinación de cuales son las partes más elementales de algo muy complejo, está sujeta al criterio de quien observa. Es decir, algo que posee complejidad elemental para un observador, puede resultar algo muy complejo para otro.

3) Las relaciones internas de las partes de algo muy complejo, son distintas de las relaciones entre las partes mismas. Así que es posible estudiar por un lado a cada una de las partes aisladamente y por otro a las relaciones entre las mismas.

4) La jerarquía de la complejidad en problemas muy grandes usualmente está compuesta sólo por unas cuantas partes, las cuales se combinan de múltiples formas.

5) Todo problema muy complejo que se ha resuelto bien, es debido invariablemente a que fue derivado de otro menos complejo que se resolvió bien.

2.1.2.- ASPECTOS DE LA COMPLEJIDAD.

En el afán de desarrollar métodos que resuelvan fácil y ordenadamente problemas muy complejos, surgen 3 aspectos fundamentales : la descomposición del problema en subproblemas, la abstracción del mismo y la jerarquización de las partes.

A) Descomposición de un problema en subproblemas :

Ya bien lo dice el refran " Divide et Impera". Y el estudio de la complejidad no ha sido la excepción. De la imperiosa necesidad de la descomposición de un problema han surgido 2 grandes tendencias, la descomposición por algoritmos, y la descomposición orientada a objetos.

La primera es la descomposición del problema en pequeñas tareas, donde cada una puede ejecutarse una o muchas veces a lo largo de la solución, dependiendo de como se requiera.

En cambio la segunda se refiere a la identificación de los objetos del problema y su comportamiento, para así crear entes que los representen y se comporten como tales; es decir, entes que no sólo posean los atributos de los objetos reales que modelan, sino que también tengan un comportamiento propio igual a estos últimos.

Pero la pregunta es clara : ¿ Cuál es la mejor forma de descomponer un problema ?. Esta no es una pregunta fácil de responder, pero lo que si está claro es que no es posible construir un sistema de ambas formas a la vez, ya que ambas son completamente ortogonales una con respecto de la otra.

La experiencia ha demostrado que es mejor si el primer intento es orientado a objetos, ya que esta es una forma más natural de descomponer un problema [Booch]. Es más fácil visualizar a los objetos y su comportamiento en el problema, que descomponer a éste simplemente en distintas tareas. En particular pienso que el análisis por objetos, es una forma más natural de descomponer un problema, pero es claro que hay que aprender a hacerlo.

Tomemos como ejemplo el caso de los programas de computadoras. Para todos aquellos que han sido enseñados a programar por algoritmos, el método Orientado a Objetos constituye un cambio radical en la forma de ver las cosas. En los primeros intentos de programar con esta nueva filosofía se tiende a seguir utilizando muchos conceptos de la descomposición algoritmica. Además, como se verá más adelante, existen ciertas cosas que para un método resultan triviales y que al cambiar al otro se convierten en algo impreciso y difícil de implantar (al menos al principio).

Además de ser una forma más natural de ver a la complejidad de un problema, el método de descomposición de objetos tiene otras ventajas, como por ejemplo el hecho de que las soluciones resulten más pequeñas y fáciles de entender, además de que son más facilmente modificables posteriormente y por lo tanto, puedan evolucionar de forma mucho más sencilla.

B) Abstracción del problema :

Esto se refiere a la forma en como los elementos de un problema son modelados, viendo primeramente sólo las características que más nos interesan, para luego ir descendiendo paso a paso hasta abarcarlas todas, consiguiendo así una modelación adecuada del problema.

Es claro que lo que cada persona abstrae en un paso puede ser distinto, y por lo tanto al final las implantaciones pueden resultar diferentes; aunque en el fondo las soluciones orientadas o objetos tienden a ser parecidas, ya que en este método se trata de modelar objetos "reales", los cuales por lo general son facilmente distinguibles y tienen comportamientos muy especificos.

En mi opinión, el hecho de que las soluciones sean parecidas es una gran ventaja, por que no sólo ayuda a que el entendimiento de una solución por alguien ajeno sea mucho más sencillo, sino que además es mucho más facil solucionar algo en equipo, puesto que todos visualizan los mismos objetos, los cuales marcan su propio comportamiento.

C) Jerarquización de las partes :

La jerarquización surge de manera natural en la descomposición o análisis Orientado a Objetos. Primeramente se hará una distinción entre dos conceptos fundamentales: clase y objeto. Una clase es una definición genérica, una descripción de las características y el comportamiento de un conjunto de entes, los objetos. Estos últimos son instancias, casos particulares de una clase, individuos con una identidad propia y que son representantes de alguna clase.

Por ejemplo, si se deseara modelar un ser humano en particular, digamos Juan, primero se haría una clase que modelara a todos los seres vivos, donde se definen todas las características de los seres vivos, luego se construiría a partir de esta una segunda clase que abarcara a todos los animales, donde se definen todas las funciones que tienen éstos, después se derivaría una tercera clase que abarcara a los mamíferos, con las características que comparten los mamíferos, y por último, se derivaría la clase ser humano, donde se definen todas las características de los seres humanos; al término de ésto, se podría crear un objeto de la clase ser humano llamado Juan, el cual sería de la clase ser humano, así como un mamífero, un animal y un ser vivo y por lo tanto, Juan tendría todas las características definidas en esas clases.

Es debido a esta forma de definir las cosas por lo que se dice que la complejidad es jerárquica, ya que se forma una Jerarquía de clases.

2.2.- EVOLUCION Y DESCRIPCION DEL DISEÑO ORIENTADO A OBJETOS.

Debido al gran crecimiento en la complejidad de los problemas que el hombre ha tenido que resolver, la investigación en el campo de los métodos de resolución ha aumentado notablemente.

En el caso de los lenguajes de programación, éstos han evolucionado significativamente y gracias a ésto se han podido atacar problemas antes inabordables. Se ha pasado de lenguajes que le decían a la computadora que hacer, a lenguajes que describen las abstracciones claves de los problemas.

Así pues se tienen:

A) Lenguajes de primera generación (1954-1958)

Estos lenguajes fueron usados principalmente en aplicaciones científicas y de ingeniería [Booch]; casi solamente manejaban conceptos matemáticos. Entre ellos destacan :

- FORTRAN I
- ALGOL 58
- Flowmatic
- IPL V

En estos lenguajes las subrutinas usaban datos compartidos en memoria, por lo que las dependencias de datos eran muy fuertes y un error en una parte del programa tenía efectos devastadores en todo el sistema.

Cualquier cambio en el *software* resultaba difícil de llevar a cabo ya que la integridad del programa podía verse afectada. La topología típica de un programa era :

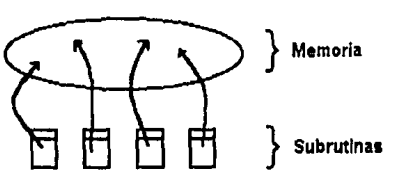

FIGURA 2.1 .- Topología de los programas de la primera generación.

B) Lenguajes de segunda generación (1959-1961)

En esta etapa el énfasis era sobre abstracciones algoritmicas. Fue aquí que las máquinas comenzaron a ser mucho más potentes. Los principales lenguajes eran :

- FORTRAN	Subrutinas, compilación por separado
- ALGOL 60	Estructura en bloques, tipos de datos
- COBOL	Descripción de datos, manejo de archivos
- Lisp	Procesamiento de listas, apuntadores

En estos lenguajes se consideran por primera vez a las subrutinas como mecanismos de abstracción. Se utilizaron por primera vez el paso de parámetros y la programación estructurada, consiguiendo grandes avances teóricos con respecto al alcance y visibilidad de variables así como de las estructuras de control. Pero estos lenguajes seguían teniendo muchas de las inconveniencias de los de primera generación. La topología típica de un programa era algo como :

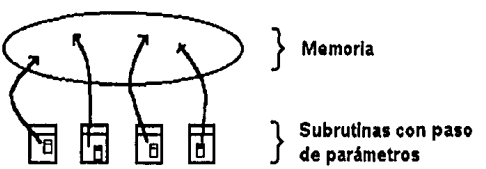

FIGURA 2.2 .- Topología de los programas de la segunda generación.

C) Lenguajes de tercera generación (1962-1970)

Para fines de los años 60's con el advenimiento de los transistores y más tarde de los circuitos integrados, el costo en los equipos de cómputo decayó dramáticamente y entonces nuevos problemas podían ser atacados, pero éstos requerían de la utilización de nuevos tipos de datos así que los lenguajes tuvieron que introducir los conceptos de abstracción de datos. Algunos de estos lenguajes fueron :

- PL/1	FORTRAN + ALGOL + COBOL
- ALGOL 68	Sucesor de ALGOL 60
- Pascal	Sucesor de ALGOL 60
- Simula	Clases y abstracción de datos

La topología de los programas seguía siendo la misma que en la segunda generación, sólo que los problemas eran mucho más grandes, así que ya no podía haber un solo

programador y empezaron a hacerse programas en equipo. Esto llevó a la creación de módulos que se compilaban por separado para luego ser unidos, tal como se ve en la Figura 2.3.

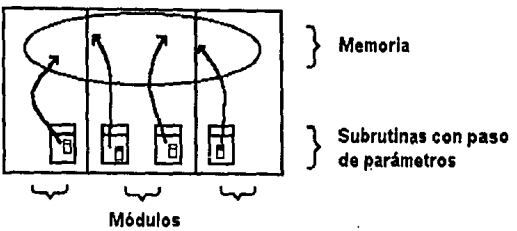

FIGURA 2.3 .- Topología de los programas de la tercera generación.

D) Espacio Generacional (1970-1980)

Los años setenta estuvieron llenos de actividad en lo referente a la investigación en lenguajes de programación, resultando la creación de cerca de un par de miles de lenguajes y dialectos. Pocos de estos lenguajes sobrevivieron, aunque los conceptos que ellos introdujeron sirvieron de base para muchos de los lenguajes actuales, como por ejemplo :

- Ada	Sucesor de ALGOL 68 y Pascal
- CLOS	Evolucionado de Lisp
- C++	Derivado de C y Simula (C desarrollado en AT&T en 1970)

Nuestro gran interes aqui es de los lenguajes que llamaremos "Orientados a Objetos" y "Basados en Objetos", donde la topología de un programa consta de módulos que representan una colección lógica de clases y objetos, en vez de subrutinas como en los otros lenguajes. Por decirlo de otro modo " Si los procedimientos y funciones son los verbos y las estructuras de datos son los sustantivos, un programa basado en algorítmos es organizado alrededor de los verbos, mientras que un programa orientado a objetos está organizado alrededor de los sustantivos " [Booch]. De este modo la topología de un programa orientado a objetos es algo como :

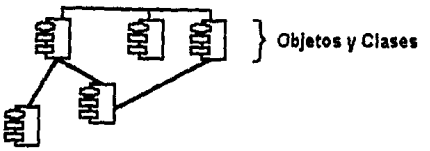

FIGURA 2.4 .- Topología de los programas Orientados a Objetos.

Al tratarse de programas bastante grandes orientados a objetos, encontramos grandes colecciones o *clusters* unas encima de otras, donde cada una contiene a su vez colecciones de clases. Tal como se ve en la Figura 2.5.

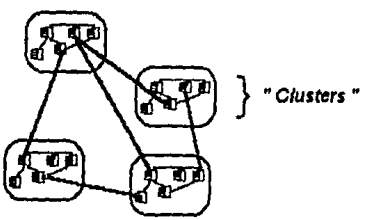

FIGURA 2.5 .- Topología de programas muy grandes Orientados a Objetos.

2.2.1.- DEFINICIONES DE PROGRAMACION, DISEÑO Y ANALISIS ORIENTADO A OBJETOS.

El modelo de objetos ha demostrado ser hasta ahora un concepto unificante en la Computación, ya que ha sido aplicado no solo a lenguajes de programación sino al diseño de bases de datos, interfaces, y hasta para el desarrollo de arquitecturas computacionales.

El modelo de objetos no representa un paso revolucionario, por el contrario, es un paso evolutivo ya que no rompe con los avances del pasado sino que permite construir sobre lo ya existente.

El modelo de objetos se deriva de muchas fuentes, es por eso que se han generado algunas confusiones con respecto a la terminología. En un esfuerzo por estandarizar los términos se han hecho las siguientes definiciones:

1) Programación Orientada a Objetos (OOP) :

Método de implantación en el cual los programas están organizados en colecciones cooperativas de objetos, cada uno de los cuales es instancia de una clase que a su vez proviene de una jerarquía de clases; la jerarquía se establece mediante relaciones de herencia. [Booch]

En esta definición propuesta por Booch cabe resaltar tres aspectos : 1) Son objetos y no algoritmos los que constituyen los bloques principales de un programa, es decir que el método de descomposición del problema es por objetos y no por algoritmos; 2) Cada objeto es la instancia de alguna clase, es decir, el programa trabaja con instancias particulares de clases; 3) Las clases están relacionadas entre sí por una jerarquía hereditaria, esto es que algunas clases heredan sus características y a otras le son heredadas las características de las clases de las que se derivan.

Puede decirse que cualquier programa que no posea estas tres características no es orientado a objetos, y específicamente, si las relaciones entre las clases no son por una jerarquía de herencia entonces decimos que es un programa que "usa tipos de datos abstractos".

Ahora podemos identificar cuales lenguajes son orientados a objetos y cuales no, ya que un lenguaje que soporta un estilo de programación debe proveer facilidades para que se construyan de modo relativamente sencillo, programas apegados al mismo. Así por ejemplo tratar de construir un programa orientado a objetos usando Ensamblador o Pascal teóricamente es posible, pero requiere de un tremendo esfuerzo, por lo que Pascal y Ensamblador no son considerados lenguajes orientados a objetos.

Se considera que un lenguaje orientado a objetos debe satisfacer lo siguiente [Booch]:

- Soporte para la creación de objetos que son abstracciones de datos, los cuales presentan una interfaz al usuario y contienen una parte escondida para este.

- Los objetos deben ser instancias de alguna clase.

- Las clases heredan las propiedades de las clases superiores.

Este último punto es interesante ya que si un lenguaje no soporta que las relaciones entre clases sean por herencia, entonces se dice que es un lenguaje "Basado en Objetos", y no "Orientado a Objetos". Así, C++, Smalltalk, y Object Pascal son Orientados a Objetos, mientras que Ada es Basado en Objetos.

2) Diseño Orientado a Objetos (OOD) :

Es el método de diseño basado en la descomposición del problema mediante objetos. La representación del mismo se hace mediante una notación que describa los modelos lógicos, físicos, estáticos y dinámicos del problema a solucionar. [Booch]

De lo anterior surgen dos aspectos importantes : 1) la descomposición del problema es mediante la filosofía de objetos; 2) a lo largo del diseño se usa una notación especial para expresar a los modelos lógicos y físicos del problema.

3) Análisis Orientado a Objetos (OOA) :

Para la creación de objetos y clases el método Orientado a Objetos se basa en un examen de las palabras que describen el problema a solucionar. [Booch]

El análisis comienza haciendo una descripción breve y precisa del problema. En esta se identificarán tanto a los sustantivos como a los verbos para luego escribir con ellos dos listas; la primera de los candidatos a clases (sustantivos) y la segunda de los candidatos a métodos u operaciones sobre las instancias de las clases (verbos).

El método de objetos primero utiliza OOA para identificar a los elementos del problema, después utiliza OOD para la construcción del modelo teórico de la solución y finalmente utiliza OOP para la implantación de la misma.

2.2.2.- ELEMENTOS DEL MODELO DE OBJETOS.

Dentro del modelo de objetos existen cuatro "grandes" elementos : Abstracción, Encapsulación, Modularidad y Herencia; por "grandes" entendemos que un modelo que no los posea todos simplemente no es orientado a objetos. De igual modo existen tres "pequeños" elementos : Tipos, Concurrencia, y Persistencia; por "pequeños" entendemos que son útiles para el modelo, pero no son esenciales.

- Abstracción

Como ya vimos, la abstracción es uno de los principales medios con los que el hombre representa a la complejidad y dentro de el modelo de objetos es un elemento esencial.

La abstracción denota las principales características de un objeto. Estas lo diferencian de los demás objetos, marcando límites conceptuales de lo que el objeto es. [Booch]

Como se ve lo más importante de la abstracción es que captura lo principal de un objeto, es decir, pone especial atención en las características de los objetos que son relevantes en el contexto del modelo.

Una abstracción debe poner especial énfasis a las características externas de un objeto, separándolas de las características internas. Gracias a esto, las características propias de un objeto quedan ocultas y la comunicación de este con el mundo exterior se establece mediante su interfaz.

Como ejemplo observemos un coche: podemos Encenderlo, Apagarlo o Conducirlo, así que estas son algunas características mediante las cuales el usuario de un coche se comunica o "usa" al coche, por lo tanto hemos abstraído el hecho de que todo coche se debe Encender, Apagar y Manejar; hasta aquí la forma como esto es hecho todavía no nos interesa.

- Encapsulación

Esta es la manera como lo interno de un objeto se esconde al exterior, es decir es el proceso mediante el cual los detalles no esenciales de un objeto son ocultados al mundo exterior . [Booch]

Abtracción y Encapsulación son conceptos complementarios, ya que uno se encarga de las características exteriores de un objeto, y el otro de los detalles propios de este.

Consideremos nuevamente un coche, este posee partes propias como son motor, ruedas y frenos, aunque la forma como el motor funciona, las ruedas "ruedan" y los frenos "frenan" no tiene importancia, ya que mientras ellas funcionen bien el coche se podrá Encender, Apagar y Manejar. La encapsulación se encargará de definir e implantar al motor, ruedas y frenos.

- Modularidad

Ya hemos visto que en el análisis de la complejidad, el hecho de descomponer un problema en partes es de suma importancia. El objetivo de la modularidad es identificar partes esenciales de un problema para luego poder partirlo. Esto generalmente se hace cuando los problemas son bastante grandes.

La modularidad es la propiedad que un sistema posee al haber sido partido en pedazos o "módulos" que se relacionan entre sí [Booch]. Esto resulta imprescindible en todo problema de tamaño considerable.

Supongamos que tenemos el problema del transporte. Una buena forma de "modularizar" al problema sería por ejemplo dividirlo en :

- Transportes terrestres
- Transportes aereos
- Transportes acuáticos

Nuestros coches estarían dentro del módulo de Transportes terrestres, junto con muchas otras clases de objetos (camiones, metro, trenes, etc).

- Herencia

Hemos usado hasta ahora tres elementos : Abstracción, Encapsulación y Modularidad, sin embargo, para completar un modelo esto no es suficiente ya que al abstraer propiedades, resulta que algunos objetos se relacionan con otros mediante las relaciones de herencia; por tanto, es necesario identificar esas relaciones y clasificarlas.

Volviendo al problema del transporte, notamos que dentro de los Transportes terrestres existen muchas subdivisiones, una de ellas por ejemplo es el tipo de combustible que utilizan; de este modo un camión, un coche y una motocicleta son objetos que se derivan de los transportes de gasolina, por tanto, todos ellos heredan las propiedades de los objetos que usan gasolina.

Es posible también que un objeto no se derive de una sola clase sino que sea producto de varias a la vez, haciéndolo que herede las propiedades de todas ellas. A esto se le conoce como " Herencia Multiple ".

- Tipos

El concepto de tipo se deriva del estudio de los tipos de datos abstractos; en general un *tipo* es la caracterización precisa de las propiedades estructurales y funcionales que una colección de entidades comparte [Booch]. En el método de objetos, Booch llama *tipo* a la liga existente entre un objeto y la clase a la que pertenece, la cual provoca que un objeto de una clase pueda a veces si y a veces no substituir a otro de otra clase; simplemente se refiere a que un objeto de una clase posee todas las características de ella y de las clases de las que se deriva, así que en un momento dado un objeto, aunque no sea una instancia directa de una clase, puede representar a la misma. Por ejemplo un Tsuru es una instancia de la clase Coche, pero dado que un Coche es un vehiculo terrestre, entonces un Tsuru puede también representar a un Transporte terrestre. A esta liga es a la que Booch llama tipo.

- Concurrencia

En ciertos problemas es necesario que puedan darse eventos al mismo tiempo, esto ha significado que los sistemas de cómputo deban manejar la concurrencia o al menos simularla eficientemente. De este modo, el modelo de objetos se preocupa por tratar de que en los programas puedan darse eventos que puedan ser sincronizados, inclusive llegando a establecer concurrencia en sistemas con varios procesadores y simulándola en sistemas con un solo procesador.

El método de objetos es bastante claro a este respecto; define la concurrencia como la propiedad que distingue a los objetos que están activos de los que están inactivos.

- Persistencia

Este término se refiere al tiempo de vida de un objeto. De acuerdo a los tiempos, los objetos se pueden clasificar en :

- Objetos que existen a lo largo de la evaluación de expresiones.

- Objetos que existen como variables locales en procedimientos y funciones.

- Objetos globales en un programa así como aquellos que existen en el *heap*.

- Objetos que existen durante varias ejecuciones de un programa.

- Objetos que existen a lo largo de varias versiones de un programa.

- Objetos que existen más allá de la vida de un programa.

Los lenguajes tradicionales solo hacen énfasis en los primeros tres tipos de objetos, mientras que los lenguajes Orientados a Objetos poseen mecanismos para definir objetos de esas seis categorías.

Dentro de el método de objetos se define la *persistencia* como: la propiedad de los objetos a través de la cual trascienden en tiempo y espacio a la vida de los programas [Booch].

2.3.- OBJETOS Y CLASES.

2.3.1.- DEFINICION DE OBJETO Y CARACTERISTICAS.

Hasta ahora se ha hablado bastante del método de objetos, pero nunca se ha dicho qué es realmente un objeto.

Se ha dicho informalmente que un objeto es algo tangible que presenta comportamiento y limites bien definidos, que está tratando de modelar algo "real" y por lo tanto existe en el tiempo y espacio.

Un objeto representa un "algo" individual bien identificado el cual puede ser o no real, teniendo un papel bien específico en el contexto en el que existe. [Booch]

De todo esto es necesario extraer lo fundamental, por eso adoptamos la siguiente definición :

Un objeto tiene estados, comportamiento e identidad; la estructura y comportamiento de objetos similares está definida por la clase común a la que pertenecen, de este modo, objeto e instancia significan lo mismo [Booch].

Una vez dada esta definición es conveniente analizarla un poco. Cuando hablamos de que un objeto tiene estado, nos referimos al hecho de que un objeto engloba a los valores que posee en un momento dado, además de las características estáticas que ya poseía. Es decir, en el tiempo un objeto puede presentar distintos estados dependiendo de los valores de sus atributos en ese momento.

Al hablar de comportamiento, nos referimos a la forma como un objeto actúa cuando su estado cambia. De esta forma, visualizamos a los objetos como seres capaces de reaccionar a ciertos eventos.

Al decir que un objeto posee identidad, nos referimos a la propiedad de que este pueda ser distinguible de otros objetos similares a él. Por ejemplo, cuando creamos a un objeto por lo general se le da un nombre, el cual lo identificará de los demás (esto es parte de su identidad); además, si este objeto cambia de estado no es otro objeto, sigue conservando su identidad.

2.3.2.- DEFINICION DE CLASE Y CARACTERISTICAS.

También se ha hablado del concepto de clase; pero en realidad no se ha dicho qué es; utilizaremos de aquí en adelante la siguiente definición:

Una *clase* es un conjunto de objetos que comparten estructura y comportamiento [Booch]. De este modo, un objeto resulta ser solo una instancia de alguna clase.

En una clase existen dos partes fundamentales, la primera es su interfaz, la cual captura el comportamiento de esta con el mundo exterior, escondiendo su estructura y comportamientos internos. La segunda es la implantación, la cual guarda la parte interna de su estructura y comportamiento, aquí se encierran el cómo y por qué un objeto se comporta de una cierta manera.

Dentro de la interfaz pueden identificarse tres partes :

- Parte pública : todas las declaraciones hechas aquí forman parte de la interfaz y los usuarios de la clase tienen acceso a ellas.

- Parte protegida : aquí las declaraciones sólo son visibles a las clases derivadas de ésta.

- Parte privada : las declaraciones hechas aquí sólo son visibles a la clase misma.

2.3.3.- RELACIONES ENTRE OBJETOS.

Para los objetos es posible relacionarse entre sí. Existen dos tipos básicos de relaciones: de uso y de contención.

En la primera dos objetos se relacionan entre sí mandándose mensajes para poder activar algún proceso o para establecer alguna relación de uso entre los dos, en cuyo caso hay tres posibilidades : la primera en la que un objeto usa a otro, no siendo éste usado por ningun otro, diciéndose que este objeto es un actor; la segunda donde el objeto sea usado por otros, pero éste no usa a ningún otro, diciéndose que éste es un servidor; o la tercera y última en donde el objeto usa y es usado, diciéndose entonces que el objeto es un agente.

El segundo tipo de relaciones son relaciones de contención, donde un objeto contiene a otros y éste los usa a lo largo de su vida.

2.3.4.- RELACIONES ENTRE CLASES.

También se ha hablado de que existen relaciones entre las clases; pero ahora se establecerá una clasificación entre ellas. Existen básicamente 4 relaciones entre clases.

La primera es la relación de herencia en la que una clase hereda la estructura y el comportamiento de las clases de las cuales se derivó. La segunda relación es la que se llama relación de uso; queriendo decir que una clase usa dentro de sí, a alguna o algunas instancias de otras clases, las cuales interactúan en estructura y comportamiento con la clase que las instanció. La tercera relación se llama relación de

instancia y se refiere al tipo de relación que se establece cuando se define una clase que al ser instanciada, contendrá a un mismo tipo de objetos, donde el tipo se define al momento de la instanciación. La cuarta y última relación se llama relación con metaclases. Este tipo de relación surge cuando al instanciar una clase X, esta contendrá clases y no objetos, es decir, la clase X es una metaclase de sus miembros.

Además de estas cuatro relaciones entre clases es posible observar que existe una relación entre objetos y clases. Cada objeto es una instancia de una clase y cada clase puede contener cero o más instancias de otras clases.

2.4.- EL METODO DEL DISEÑO ORIENTADO A OBJETOS.

2.4.1.- NOTACION.

El método de objetos trabaja con aspectos que en ocasiones resultan invisibles al observador, tales como las relaciones de las que ya hemos hablado o como el hecho de que los objetos son instancias de clases que interactúan unas con otras. Esto hace notar que la representación de lo anterior requiere del uso de una buena notación.

Aquí se describirá la notación propuesta por Booch para el método de objetos. Se presentarán los símbolos y formatos usados para clases unicamente; la notación de Booch abarca también símbolos y relaciones entre objetos y módulos, pero estos no serán presentados dado que no se utilizaron en el desarrollo del trabajo de tesis.

Existen cuatro aspectos de importancia a considerar en la implantación de un proyecto: 1) qué clases existen y cómo se relacionan; 2) qué mecanismos son usados para regular la colaboración entre objetos; 3) qué lugar es el más conveniente para declarar objetos y clases, y 4) en cuál procesador se instalará la implantación para poder hacer un buen esquema de sincronización de procesos.

Estos aspectos pueden ser modelados usando diagramas y formatos. Dentro de los diagramas existen cuatro que son estáticos y dos que son dinámicos. En el caso de los formatos estos son llenados con datos que sirven para complementar a los diagramas.

A) Diagramas de Clases.

Estos diagramas sirven para mostrar la existencia de las clases y sus relaciones con el sistema. Los tres aspectos más importantes de un diagrama de clase son : i) la clase misma, ii) las relaciones, y iii) las utilerías.

i) Las clases son generalmente representadas por bolas amorfas (Figura 2.6).

FIGURA 2.6 .- Símbolo para Clase.

ii) Para las relaciones entre clases se utilizan los símbolos de la Figura 2.7. Aquí se puede ver que son representadas relaciones de herencia, uso, instanciación, y

metaclase; así por ejemplo si la interfaz de una clase A usa los recursos de una clase B, entonces se dibuja una doble línea entre las dos y se pondrá un círculo no relleno del lado de la clase A.

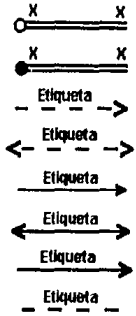

FIGURA 2.7 .- Símbolos para las relaciones entre Clases y cardinalidad.

En ciertas circunstancias resulta bastante valioso mostrar la cardinalidad entre las clases que se relacionan, por ejemplo, si entre dos clases A y B se coloca un 1 cerca de A y un + cerca de B, se interpretará que por cada instancia de A se tendrá una o más instancias de B y que por cada instancia de B existe sólo una instancia de A (Figura 2.7 y Figura 2.8). Una relación donde A instancia a B es indicada por una línea punteada con una flecha apuntando hacia B.

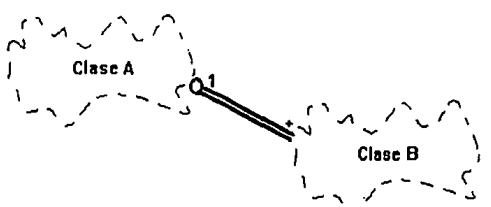

FIGURA 2.8 .- Ejemplo de relación de instanciación entre clases.

iii) El símbolo representativo en el caso de las utilerías, es como el de una clase sencilla pero con el borde sombreado para diferenciarlo. Este símbolo representa a uno o varios subprogramas libres.

Existen agrupaciones de clases llamadas categorías, las cuales sirven para organizar a las distintas clases que conforman una aplicación. Esto generalmente sucede en sistemas muy grandes. El símbolo usado es mostrado en la Figura 2.9.

Dentro de estas categorías existen tanto clases que son visibles desde el exterior como otras que no lo son, e incluso existen clases que son traidas de partes visibles de otras categorías. Estas relaciones son expresadas mediante los símbolos de la Figura 2.9.

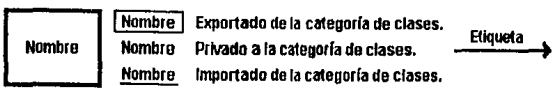

FIGURA 2.9 .- Símbolos para categorías de Clases y visibilidad.

Para los diagramas de clases hay dos tipos de formatos a utilizar: el de clase y el de operación (Figura 2.10).

Nombre : Identificador.
Documentación : texto.
Visibilidad : exportada / privada / importada.
Parámetros : lista de parámetros.
Interfaz / Implantación
Uso : Lista de nombres de Clases.
Lista de Campos.
Lista de Operaciones.

Documentación : texto.
Categoría : texto.
Calificador : texto.
Parámetros formales : lista de parámetros.
Resultado : Nombre de la Clase.
Precondiciones : Diagrama de Objeto.
Acción : Diagrama de Objeto.
Postcondiciones : Diagrama de Objeto.
Excepciones : lista de declaraciones.
Concurrencia : secuencial / protegida / concurrente / múltiple.
Tiempo : texto.
Lugar : texto.

FIGURA 2.10 .- Formatos para Utilización de Clases y operaciones.

B) Diagramas de Transición de Estados.

Son usados para describir el comportamiento dinámico asociado a ciertas clases. Estos muestran el estado en el tiempo de una clase, así como las acciones que resultan de un cambio de estado. El círculo de la Figura 2.11 representa un estado.

Las transiciones de estado se representan por una flecha, la cual va del símbolo de estado inicial al símbolo de estado final.

Típicamente a un objeto no se le asocian estados de inicio y fin por el contrario cuando éste es creado, parte de un estado dependiente del medio que lo rodea.

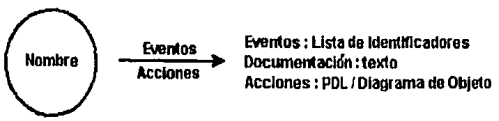

FIGURA 2.11 .- Símbolos para los diagramas de transición de estados.

El formato utilizado para los cambios de estado se muestra en la Figura 2.11.

2.4.2.- EL PROCESO DE DISEÑO.

El método de diseño Orientado a Objetos no comienza en un punto y termina en otro, por el contrario, es un método iterativo y evolutivo. Comienza con la identificación de los elementos principales del problema y va construyéndose paso a paso la solución de modo que en algún punto esta funcione; aunque esto último no necesariamente marca el final del método, ya que es posible seguir haciendo mejoras utilizando las técnicas de los pasos anteriores.

Hay cuatro pasos principales en el método y en cada uno de ellos existe algo qué hacer; por consiguiente un resultado es esperado al termino del mismo. A continuación describimos estos pasos.

1) Identificación de Clases y Objetos.

En este paso existen dos actividades principales: establecer las abstracciones claves del problema e inventar los mecanismos más importantes con los que los objetos actuarán entre sí para desarrollar una tarea.

En este paso obtendremos una lista de nombres de objetos y clases identificados a través de las abstracciones hechas. Resulta de utilidad el que se construyan algunos diagramas de objetos y clases con el fin de mostrar de una forma general como éstos interactúan.

2) Identificación de la Semántica de Clases y Objetos.

Este paso requiere que se establezca el significado de los objetos y clases creadas en el paso anterior. Aquí se deberá describir los comportamientos de clases y objetos desde un punto de vista exterior, es decir, como serán las interfaces de las clases y qué podremos hacer con instancias de las mismas; así mismo se deben describir las cosas que un objeto podrá hacer con otro.

Cabe mencionar que el proceso puede resultar iterativo ya que al fijar el comportamineto de un objeto, éste puede alterar el comportamiento de otro; en este caso Booch recomienda que se hagan escritos donde se describa de forma precisa qué es lo que cada objeto podrá hacer. Es en este momento cuando se deben documentar los significados tanto estáticos como dinámicos, de todas las abstracciones hechas. Esta documentación se hace mediante los diagramas, gráficas y formatos que ya se han mencionado.

3) Identificación de las Relaciones entre Clases y Objetos.

Este paso es en gran medida una extensión del anterior. Aqui es dónde se establece de forma precisa cómo las clases y objetos interactuarán; por tanto, aquí es dónde se especifican las relaciones entre objetos y clases (herencias, usos, instanciaciones, etc). Primeramente se recomienda encontrar patrones entre las clases para que una vez hecho esto se puedan simplificar al máximo las estructuras de las mismas.

Enseguida es de suma importancia el hecho de que se establezca de forma precisa el como las clases y los objetos se verán entre sí. Una buena forma de hacer esto es tomar por pares a los objetos e irse preguntando si esos dos se relacionan en algo; si la respuesta es afirmativa entonces se establecen dos puntos : 1) que tipo de relación es y 2) qué mensajes serán los que se enviarán el uno al otro. Cabe notar que es posible que en este paso sea necesario modificar algunas de las definiciones de las clases, implicando entonces que se vuelva a pasos anteriores. Es por este tipo de situaciones que el proceso de diseño orientado a objetos es un proceso iterativo.

4) Implantación de Clases y Objetos.

Aunque este es el cuarto paso no necesariamente es el último. Aquí es dónde por primera vez vemos el interior de las clases, ya que debemos definir e implantar el como es que las clases funcionarán. Igualmente aquí es dónde se debe separar al programa en módulos si es que esto es necesario.

Puede darse el caso de que al estar implantando la estructura interna de alguna clase, sea necesario regresar pasos atrás y redefinir el concepto; este es otro punto más, donde el proceso se vuelve iterativo y evolutivo.

Una vez terminado estos cuatro pasos existe ya una implantación que posee un cierto grado de abstracción, de este modo al ir pasando por aquí más veces, ésta se va puliendo y perfeccionando hasta que en algún momento se considere con un nivel suficientemente bueno como para ser utilizada.

2.4.3.- BENEFICIOS Y DESVENTAJAS DEL DISEÑO ORIENTADO A OBJETOS.

- Beneficios.

El diseño orientado a objetos no es un concepto nuevo y aunque tampoco es muy antiguo, todavía no se considera como un concepto maduro. Quizá se deba al hecho de que la mayor parte de la gente está acostumbrada a los estilos de programación tradicionales [Booch]. Se espera que esto cambie con el tiempo.

Hasta ahora no hay mucho con que argumentar el hecho de que esta pueda ser una técnica sumamente buena, lo que sí es claro es que los problemas que ya se han resuelto de esta forma han funcionado perfectamente, así es que quizá es solo cuestión de tiempo para que más y más problemas sean atacados de esta forma.

En cuestiones teóricas, algunos de los principales beneficios de esta filosofía ya los hemos descrito. En general se tiene que : promueve la reutilización de distintos componentes, hace que los sistemas sean fácilmente evolucionables, utiliza el método más natural de pensar del ser humano y promueve la utilización de todos los recursos que tienen los lenguajes orientados a objetos.

Existen estudios con resultados positivos acerca de el hecho de que el código orientado a objetos ocupa mucho menos espacio; en el caso de esta tesis pudo observarse este hecho ya que la sola interfaz en C++ ocupa casi la mitad de espacio que la desarrollada en C.

- Desventajas.

Hay dos tipos de desventajas principales: 1) en el rendimiento de un programa y 2) en los costos adicionales que una implantación con esta filosofía requiere.

1) Con respecto al rendimiento de un programa se tienen varios puntos importantes. Uno de ellos es el hecho de que mandar mensajes de un objeto a otro ocupa tiempo y algo de espacio.

Otro de ellos es que muchas funciones y procedimientos acceden a datos que son privados, por lo que si tenemos una jeraquía de clases muy grande, acceder a algún dato muy por debajo en la jerarquía puede resultar en una gran sucesión de llamadas a funciones, lo cual implica por supuesto mucho tiempo. Es claro que esto puede evitarse desencapsulando los datos o implantando las funciones en línea, pero ésto puede provocar la pérdida de ciertas características en el diseño implantado; se verá un pequeño ejemplo en el capítulo III.

Otro hecho es que la mayor parte de los compiladores producen código objeto distribuído por segmentos(generalmente un archivo estará en un sólo segmento), sin embargo, en el código orientado a objetos no existe una referencia tan precisa, ya que

muchas veces las clases están definidas en archivos separados, así que dado que la llamada a una función miembro de una clase puede invocar muchas llamadas a funciones de otras clases, la ejecución de esta hace referencia a código distribuído en varios segmentos, afectando directamente el tiempo de ejecución. Esto puede evitarse simplemente tratando de definir a las clases en los mismos archivos [Booch].

Otro problema es la creación y destrucción dinámica de objetos a lo largo de la ejecución de un programa, ya que reservar y utilizar espacio en el *heap*, toma mucho más tiempo que reservar y utilizar variables locales o en el *stack*. Los lenguajes orientados a objetos utilizan en gran medida el *heap*, por lo que el rendimiento en tiempo puede verse afectado.

2) Con respecto a los costos adicionales la principal desventaja es el hecho de que actualmente la mayoría de la gente está acostumbrada a las otras filosofías, por tanto, el desarrollar una implantación orientada a objetos además de requerir de el *software* necesario, requiere de capacitar al equipo de desarrollo (y ésto no es algo que se haga en 3 días), así que ésto puede tomar tiempo.

Como se ha podido ver hasta aquí, el código orientado a objetos tiende a ocupar mucho menos espacio, aunque muchas veces no resulta muy bueno en cuánto a tiempos de ejecución. Por tanto en aplicaciones donde el tiempo es importante puede ser que este método no sea muy buena opción (al menos para los principiantes).

REFERENCIAS.

[Booch]
- Booch, Grady. " *Object Oriented Design* ". The Benjamin / Cummings Publishing Company, Inc., 1991.

CAPITULO III

DISEÑO E IMPLANTACION DE LA INTERFAZ DE RED.

3.1.- CARACTERISTICAS DE LA INTERFAZ DE RED.

En este capítulo se procederá a diseñar e implantar una interfaz para redes *Ethernet* apoyada en los manejadores de Clarkson. La filosofía del diseño será siguiendo el método Orientado a Objetos.

La interfaz de red será implantada usando el compilador de Borland C++ versión 3.1, y usando los manejadores de Tarjetas de red elaborados en la Universidad de Clarkson. Las pruebas serán en máquinas PC 386 y 486, con sistemas operativos MS-DOS en sus distintas versiones, utilizando la red *Ethernet* del Instituto de Matemáticas de la Universidad Nacional Autónoma de México. La interfaz podrá utilizarse en máquinas PC con sistema operativo MS-DOS las cuales tengan instalado tanto Tarjeta para Red como a los manejadores de Clarkson.

3.2.- APLICACION DEL METODO E IMPLANTACION.

Como ya se ha dicho el método de objetos consta de 4 pasos los cuales no necesariamente constituyen un principio y un fin, así que se irá tratando de seguir la secuencia en la medida de lo posible.

Primeramente es necesario describir el problema :

"Implantar una interfaz de red *Ethernet* al nivel de la capa de enlace de datos (Figura 1.10) del modelo OSI, que proporcione servicios de red tales como envío y recepción de paquetes".

De lo anterior es posible identificar a los sustantivos : interfaz y paquetes, así como también a los verbos : proporcionar servicios de red, envío y recepción.

De los sustantivos se nota la necesidad de crear clases que representen tanto a los paquetes como a la interfaz. Estas las llamaremos por ahora " *Paquete* " e "*Interfaz*". De los verbos se nota la necesidad de que existan rutinas para servicios de red tales como de envío y recepción de paquetes.

Se ha llegado a la conclusión que dado que los paquetes son enviados y recibidos, las rutinas de envío y recepción de paquetes deben pertenecer a la clase *Interfaz*; así que por homogeneidad, las demás rutinas de servicio de red también pertenecerán a la clase *Interfaz*. Esta última es la que ejecuta acciones sobre los paquetes, por lo tanto

será tal que al instanciarse proporcionará objetos que brindarán servicios de red, los cuales manejarán instancias de la clase *Paquete*, por lo que habrá una relación de uso entre las clases *Interfaz* y *Paquete*.

Por lo pronto se tiene hasta una ahora una situación como la que se ilustra en la Figura 3.1.

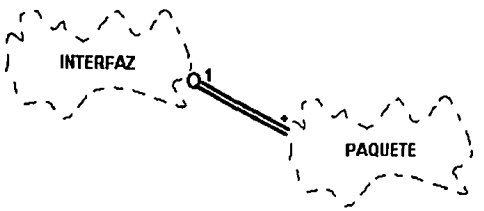

FIGURA 3.1 .- Diagrama temporal de Clases *Interfaz* y *Paquete*.

Una vez establecido lo anterior se nota claramente que al instanciar *Interfaz* debe poderse enviar y recibir paquetes, así que deberá proveerse alguna forma para saber qué tipo de paquetes son los que se quieren recibir, ya que como se vió en el capítulo I los paquetes poseen campos para las direcciones origen y destino. Con lo anterior debe poderse identificar cuáles paquetes están dirigidos a nuestra máquina y cuales no y entonces decidir cuáles son los que se recibirán. Como consecuencia de esto la interfaz debe de ser capaz de saber la dirección de la máquina en donde ella se encuentra (Direccion_local).

Otro punto interesante es el hecho de que los paquetes tienen un formato(ya que estos provienen de una red *Ethernet*), esto nos sugiere que la clase *Paquete* deba tener un formato parecido, es decir que sea compatible con el formato de los paquetes de *Ethernet*; por lo tanto de aquí en adelante la clase será llamada "*Paquethe*" que quiere decir PAQUete de *ETHErnet*.

Aquí se puede observar que dado que la interfaz proporciona los servicios de red al usuario, estos servicios deberán ser parte de la interfaz de la clase; por lo tanto, serán servicios que estarán en la parte publica de la definición de la misma. Se pasará ahora a analizar más detalladamente los servicios de red.

El servicio de envío de paquetes al que llamaremos Envia_paq, será un servicio que como su nombre lo indica enviará paquetes a través de la red, es decir, mandará instancias de la clase *Paquethe*. Hay que recordar que la interfaz estará

comunicándose con los manejadores de Clarkson, así que se deberá decir a éstos cuál es la ubicación en memoria de los *bytes* que se quieren enviar.

Esto nos plantea dos situaciones: o la rutina de envío debe saber donde es que la clase *Paquethe* tiene los datos o que la clase *Paquethe* tenga alguna rutina que nos brinde esa información. Esto quizá sea más conveniente decidirlo cuando se hayan analizado más servicios de red.

Otro de los servicios de red que se necesitan es el de recepción de paquetes, al que llamaremos Recibe_paq. En este servicio existe una comunicación directa de los manejadores con la interfaz, puesto que son estos los que vacían la información que llega en el lugar que se les haya dicho; además la rutina de recepción debe proporcionar un *Paquethe* con los datos que llegaron. Cabe señalar aquí que los paquetes son recibidos por la interfaz como respuesta a una interrupción del procesador, esto hace que no sea posible saber en que momento está arribando un mensaje; por lo tanto debe implantarse algun sistema que tanto almacene los *Paquethe's* como los proporcione en el momento que se le diga. Esto se implantó creando una clase llamada *Cola*, la que como su nombre lo indica, es la tradicional estructura de datos pero vista con el enfoque de objetos; ésta tendrá un lugar de almacenamiento de *Paquethe's* y servicios tales como meter a la cola (Mete), sacar de la cola (Saca) y llevar un conteo de los elementos que hay (Cuantos_hay). De este modo Recibe_paq deberá invocar el servicio que saca de la cola para obtener el primer *Paquethe* que llegó de la red.

Hay que notar que para cada interfaz es necesario solo una *Cola*, por lo que se define un miembro de *Interfaz* llamado cola, el cual es una instancia de *Cola* que proporciona servicios de red entre otros.

Hasta aquí se tienen definidas 3 clases distintas : *Paquethe*, *Cola*, e *Interfaz*; hasta este momento las relaciones entre ellas se muestran en la Figura 3.2.

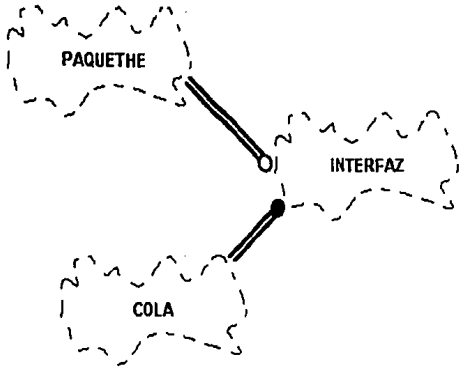

FIGURA 3.2 .- Diagrama temporal de Clases *Interfaz*, *Cola* y *Paquethe*.

Hasta aquí se han definido los siguientes mecanismos : Recibe_paq, Envía_paq, Direccion_local, Mete, Saca, y Cuantos_hay. En la Figura 3.3 se muestran algunos de los formatos de estas funciones.

Nombre : Recibe_paq. **Documentación : Proporciona el primer Paquethe de la Cola.** **Parámetros : paq - Paquethe.** **Resultado : entero.** **Acciones : Colocar en paq, el Paquethe que está en el primer Elemento de la Cola.**	**Nombre : Mete.** **Documentación : Coloca en el último lugar de la Cola, al Elemento recién recibido.** **Parámetros : El - Elemento.** **Resultado : entero.** **Acciones : Colocar una copia de El en el último lugar de la Cola.**

FIGURA 3.3 .- Formatos de las funciones Recibe_paq y Mete.

Se comenzará con la implantación de las clases. Es posible que al hacerlo surjan mejoras a lo ya definido.

Primeramente se implantará la clase *Paquethe*. Un paquete de *Ethernet* como ya se describió en el Capítulo I, tiene dos campos de direcciones de 6 *bytes*, un campo de tipo de 2 *bytes*, y un campo de datos de 1500 *bytes*. Las direcciones podrían ser implantadas con dos arreglos de caracteres de tamaño 6, aunque cabe decir que en lo futuro será necesario comparar direcciones, asignar una dirección a otra y construirlas de distintas maneras, por lo que se ha decidido que las direcciones de *Ethernet* constituyan una clase, la cual se llamará *Direccion_ether*.

La clase *Direccion_ether* contendrá un campo de datos de tamaño 6 *bytes* donde se almacenará la dirección, tendrá a su disposición funciones especiales que permitirán construir direcciones de varias maneras, asignar direcciones y compararlas.

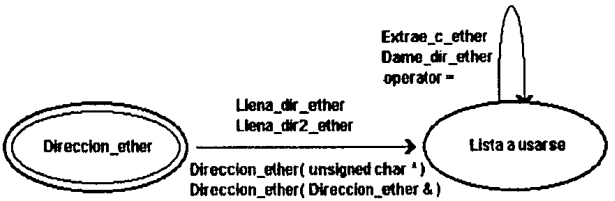

FIGURA 3.4 .- Diagrama de estados de la clase *Direccion_ether.*

La forma que tendrá esta clase se ilustra en la Figura 3.5.

```
class Direccion_ether
{
        protected:
                unsigned char dir_ether[ 6 ];                   // Almacena la
                                                                dirección
        public:
                Direccion_ether();                              // Constructor
                Direccion_ether( unsigned char * );             // Constructor
                Direccion_ether( Direccion_ether & );           // Constructor
                ~Direccion_ether(){ }                           // Destructor
                void Llena_dir_ether ( unsigned char * );       // Llena la
                                                                dirección
                void Llena_dir2_ether ( int i, unsigned char c )// Pone un
                        { dir_ether[ i ] = c; }                 caracter
                unsigned char Extrae_c_ether ( int i )          // Extrae un
                        { return dir_ether[ i ]; }              caracter
                unsigned char *Dame_dir_ether ( void )          // Da acceso la
                        { return dir_ether; }                   dirección
                void operator =( const Direccion_ether & );     // Asigna
                                                                Direccion_ether's
                friend int operator ==( Direccion_ether &,      // Compara
                                        Direccion_ether & );    Direccion_ether's

};
```

FIGURA 3.5 .- Implantación de la clase *Direccion_ether.*

De lo anterior se concluye que dado que un *Paquethe* posee dos direcciones (origen y destino) esta clase contendrá dos instancias de la clase *Direccion_ether*, las cuales se llamarán " org" y " dst " respectivamente.

El tipo de *Paquethe* será guardado en un entero sin signo llamado " tipo ". De igual modo los datos serán almacenados en un buffer de caracteres sin signo llamado "datos". Además de los datos un *Paquethe* tendrá funciones para ser construído de varias maneras, así como funciones para accesar la información, ya que esta quedará encapsulada en la parte privada de la clase.

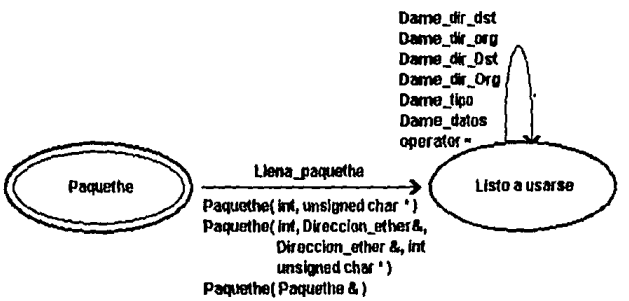

FIGURA 3.6 .- Diagrama de estados de la clase *Paquethe*.

La implantación de esta clase quedará así :

```
class Paquethe
{

        protected:
                Direccion_ether dst;                            // Dirección destino
                Direccion_ether org;                            // Dirección origen
                unsigned int  tipo;                             // Tipo del paquete
                unsigned char datos[ LDETHER ];                 // Datos del paquete

        public:
                Paquethe();                                     // Constructor
                Paquethe( int, unsigned char * );               // Constructor
                Paquethe( int, Direccion_ether &,
                        Direccion_ether &, int, unsigned char * );  // Constructor
```

```
Paquethe( Paquethe & );                    // Constructor
~Paquethe(){}                              // Destructor
void Llena_paquethe( Paquethe * );         // Llena el paquete
unsigned char *Dame_dir_dst ( void )       // Da acceso a la
        { return dst.Dame_dir_ether(); }   dirección destino
unsigned char *Dame_dir_org ( void )       // Da acceso a la
        { return org.Dame_dir_ether(); }   dirección origen
Direccion_ether Dame_dir_Dst ( void );     // Da una instacia
                                           de Direccion_ether
Direccion_ether Dame_dir_Org ( void );     // Da una instancia
                                           de Direccion_ether
unsigned int Dame_tipo ( void ){ return tipo; }  // Da el tipo
unsigned char *Dame_datos ( void )         // Da acceso a los
        { return datos; }                  datos
void operator =( const Paquethe &paq );    // Asigna Paquethe's
};
```

FIGURA 3.7 .- Implantación de la clase *Paquethe*.

Una vez definida *Paquethe* cabe decir que los manejadores de Clarkson no solo proporcionan el paquete obtenido de la red, sino además proporcionan el número de caracteres que contiene éste en el campo de datos. De esta forma es posible mejorar el rendimiento tanto al meter como al sacar *Paquethe's* de la *Cola*, por lo que se ha decidido que exista una nueva clase llamada *Paquethe_ext* (*Paquethe extendido*), la cual es simplemente un *Paquethe*, pero con un campo extra que contiene el número de caracteres recibidos en el campo de datos. Se modificará *Cola* para que en vez de contener *Paquethe*'s contenga *Paquethe_ext*'dos.

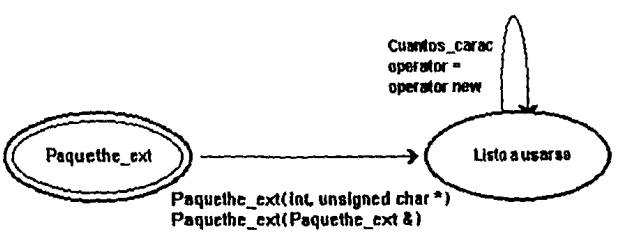

FIGURA 3.8 .- Diagrama de estados de la clase *Paquethe_ext*.

La clase *Paquethe_ext* es solo una ampliación de *Paquethe*, por lo que su implantación queda como :

```
class Paquethe_ext : public Paquethe
{
        int num_carac;                          // Guarda el número de
                                                caracteres recibidos
    public:
        Paquethe_ext();                         // Constructor
        Paquethe_ext( int, unsigned char * );   // Constructor
        Paquethe_ext( Paquethe_ext & );         // Constructor
        ~Paquethe_ext(){};                      // Destructor
        int Cuantos_carac ( void )              // Da el número de
            { return num_carac; }               caracteres recibidos
                                                en el paquete
        void operator =( const Paquethe_ext & ); // Asigna Paquethe_ext'dos
        void *operator new( size_t, void *p )   // Operador new para
            { return p; }                       Paquethe_ext'dos
};
```

FIGURA 3.9 .- Implantación de la clase *Paquethe_ext.*

Ahora si es posible hacer la implantación definitiva de *Cola.* Esta clase es simplemente una estructura de datos cola (FIFO) donde lo que se encola son *Paquethe_ext'*dos. Se define un arreglo de *Paquethe_ext'*dos llamado datos donde los *Paquethe_ext'*dos serán guardados. *Cola* posee tres enteros que controlan la entrada y salida de *Paquethe_ext'*dos, es decir quién es el primero (*primero*), quién es el último (*ultimo*), y cuántos *Paquethe_ext'*dos hay en la cola (*num_elem*).

Con respecto a las funciones miembro estas serán como ya se había dicho : Mete, Saca y Cuantos_hay.

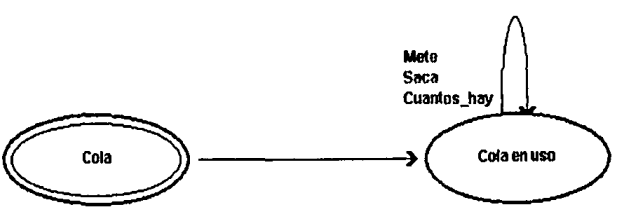

FIGURA 3.10 .- Diagrama de estados de la clase *Cola.*

La implantación quedará :

```
class Cola
{
        Paquethe_ext datos[ MAXCOLA ];      // Arreglo de Paquethe_ext' dos
        int primero,ultimo,num_elem;        // Control de Cola

    public:
        Cola();                             // Constructor
        ~Cola(){}                           // Destructor
        int Mete( Paquethe_ext * );         // Mete a la Cola
        int Saca( Paquethe_ext * );         // Saca de la Cola
        int Cuantos_hay ( void )            // Da el número de Paquethe_ext'dos en la Cola
            { return num_elem; }
};
```

FIGURA 3.11 .- Implantación de la clase *Cola*.

Ahora por fin se implantará la clase *Interfaz*. Como se ha visto *Interfaz* es una *Cola* con algunas cosas más. Una de esas cosas más surge del hecho de que cada vez que la Tarjeta recibe un paquete, los manejadores requieren de un buffer de caracteres dónde depositar los datos; además, es necesario " limpiar " este buffer antes de recibir los datos, por lo que se ha decidido que exista una clase llamada *Buffer*, la cual es simplemente un buffer de caracteres que posee una rutina de limpieza (Limpia_buffer) y una rutina de acceso a los datos (Dame_buffer).

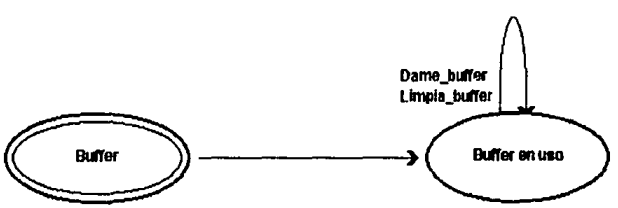

FIGURA 3.12 .- Diagrama de estados de la clase *Buffer*.

La implantación de esta clase es sencilla :

```
class Buffer
{
        unsigned char buffer_recepcion[ LTETHER ];      // Guarda los bytes
                                                        provenientes de la
                                                        red
    public:
        Buffer();                                       // Constructor
        ~Buffer(){ }                                    // Destructor
        unsigned char *Dame_buffer( void )              // Da acceso a los
            { return buffer_recepcion; }                datos
        void Limpia_buffer( void );                     // Pone en ceros
                                                        a Buffer
};
```

FIGURA 3.13 .- Implantación de la clase *Buffer.*

Una vez implantada la clase *Buffer* se ha decidido que la clase *Interfaz* contenga una instancia de *Buffer* llamada *buffer,* ya que toda *Interfaz* necesita de un buffer de recepción.

Hasta aquí ya se ha hablado de algunos de los servicios que esta clase debe proporcionar, pero ahora se hablará en especial de una de las rutinas que posee *Interfaz*, la cual puso de cabeza a todo el desarrollo. Esta rutina es llamada *Recepcion.*

Debido a que la interfaz de red utiliza a los manejadores de Clarkson, es necesario considerar varios puntos. Los manejadores reciben junto con la ayuda de la Tarjeta de red, paquetes provenientes del cable; una vez que los *bytes* son recibidos, los manejadores necesitan saber una sola cosa: ¿ QUE HACEMOS CON LOS DATOS ?, pues bien, al inicio de toda *Interfaz* es necesario que ella establezca una relación con los manejadores, es decir, que la interfaz les informe de su presencia y de este modo los manejadores le den un número de identificación; como parte de este protocolo de inicio, la interfaz está obligada a proporcionar la dirección de la rutina que se ejecutará cada vez que llegue un paquete a la Tarjeta.

Este es un punto importante, ya que hasta la fecha no se ha encontrado la manera de saber como localizar en memoria el segmento y el *offset* (i.e. la dirección física) de una función miembro de una clase; ninguno de los libros consultados de C++ dice como hacerlo, ni siquiera hacen mención de si esto es posible. Por mi parte pienso que quizá sea posible, pero considero que no aparece en ningún libro, dado que C++ organiza la información de forma muy distinta a los lenguajes tradicionales y eso generalmente no se menciona en los libros.

Ahora bien, lo anterior se soluciona si se declara la rutina de recepción como rutina miembro " estática ". La desventaja de ello es que sólo puede haber una instancia de *Interfaz* en una máquina. Esto no representa gran problema, ya que todas las aplicaciones que se han desarrollado hasta ahora (y por lo general cualquier aplicación) necesitan una sola instancia de *Interfaz*. La gran ventaja de declararla así, es que la interfaz no perderá elegancia y se puede dar la dirección de la rutina *Recepcion* sin complicaciones.

Una vez establecido el protocolo de inicio y cada vez que la Tarjeta reciba un paquete, los manejadores de Clarkson ejecutarán 2 veces a *Recepcion*: una con la finalidad de que se les proporcione la dirección del buffer de recepción, y otra con la finalidad de colocar los datos en él; es aquí en donde se debe definir un miembro de *Interfaz* llamado " *espacio_temp* ". Este es un miembro que posee espacio suficiente para que se coloque en él una instancia de *Paquethe_ext*, la cual fue creada con los datos recibidos que se almacenaron en *buffer*. Esto se hace así ya que en la *Cola* se deben almacenar *Paquethe_ext*'dos, por lo que a partir de los datos de *buffer* se debe construir una instancia de *Paquethe_ext* para luego almacenarla en la *Cola*.

La razón de existir de *espacio_temp* es que cuando los manejadores ejecutan la rutina *Recepcion*, no es posible almacenar nada en el *heap*, por lo que no se pueden hacer instancias de *Paquethe_ext*'dos cuyo lugar de almacenamiento se defina en ese momento; es decir sólo es posible crear instancias de *Paquethe_ext*'dos en lugares previamente definidos en memoria.

Una vez terminada de almacenar en la *Cola* la nueva instancia de *Paquethe_ext*, el *Buffer* es limpiado para que esté listo para recibir el siguiente paquete. Aquí termina la rutina *Recepcion*.

Existe otro detalle en la implantación de *Interfaz*. La rutina *Recepcion* debe conocer cuál *Interfaz* es la que la está usando, por lo tanto, debe saber en cuál *buffer* vaciará los datos, así como en cuál *cola* meterá el nuevo *Paquethe_ext*. El problema aquí es que la rutina *Recepcion* es estática y declarada con el modificador " *interrupt* ", por lo que no puede accesar a ningun miembro de *Interfaz* y no pueden dársele parámetros adicionales a los que marca el compilador (los registros del procesador).

Para solucionar esto se define un miembro estático de *Interfaz*, el cual es un apuntador a una *Interfaz* (A_I). Este es inicializado al momento de la construcción de *Interfaz* y se coloca referenciando a la *Interfaz* que se está creando, provocando de esta forma que la rutina de recepción pueda utilizar tanto al *buffer* como a la *cola* de esa última instancia. A continuación mostramos la implantación de la rutina de Recepcion :

```
void interrupt Interfaz::Recepcion ( unsigned bp, unsigned di, unsigned si,
                                     unsigned ds, unsigned es, unsigned dx,
                                     unsigned cx, unsigned bx  unsigned ax,
                                     unsigned ip, unsigned cs, unsigned flags )
{
      static int bytes_recibidos;
```

```
        switch ( ax )
        {
                case 0 : if ( Interfaz::A_I->cola.Cuantos_hay() < MAXCOLA )
                        {
                                es = FP_SEG( Interfaz::A_I->buffer.Dame_buffer() );
                                di = FP_OFF( Interfaz::A_I->buffer.Dame_buffer() );
                                bytes_recibidos = cx - 14;
                        }
                        else es = di = 0;
                        break;

                case 1 : Interfaz::A_I->cola.Mete( new( Interfaz::A_I->espacio_temp )
                                Paquethe_ext( bytes_recibidos, Interfaz::A_I->
                                buffer.Dame_buffer() ) );
                        Interfaz::A_I->buffer.Limpia_buffer();
                        break;

                default : break;
        }
}
```

FIGURA 3.14 .- Implantación de la rutina *Recepcion*.

Enseguida se muestra el diagrama de estados de la clase *Interfaz*:

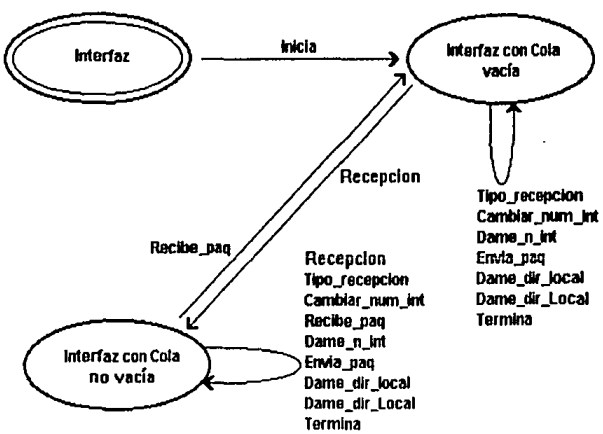

FIGURA 3.15 .- Diagrama de estados de la clase *Interfaz*.

La implantación final de la clase *Interfaz* es :

```
class Interfaz
{
        int num_man;                        // Identificación dada por los
                                            manejadores
        int num_int;                        // Interrupción de los manejadores
        Direccion_ether dir_local;          // Dirección de la tarjeta
        Cola cola;
        Buffer buffer;
        static Interfaz *A_I;
        unsigned char espacio_temp[ sizeof( Paquethe_ext ) ];
        static void interrupt Recepcion ( unsigned,unsigned,unsigned,
                                          unsigned,unsigned,unsigned,
                                          unsigned,unsigned,unsigned,
                                          unsigned,unsigned,unsigned );

    public:

        Interfaz(){ num_int = 0x60; Interfaz::A_I = this; }   // Constructor
        ~Interfaz(){ }                                        // Destructor
        void Cambiar_num_int ( int num_i )                    // Cambia la
                { num_int = num_i; }                          interrupción
        int Dame_n_int ( void )                               // Da la interrupción
                { return num_int; }                           actual
        int Inicia ( unsigned int );                          // Inicia Interfaz
        int Recibe_paq ( Paquethe *, int * );                 // Recibe paquete
        int Envia_paq ( Paquethe *, int );                    // Envia paquete
        int Tipo_recepcion ( int );                           // Cambia el tipo de
                                                              recepción
        unsigned char *Dame_dir_local ( void ){ return        // Da acceso a la
                dir_local.Dame_dir_ether(); }                 dirección local
        Direccion_ether Dame_dir_Local ( void );              // Da una instancia
                                                              de Direccion_ether
        int Termina ( void );                                 // Finaliza Interfaz
};
```

FIGURA 3.16 .- Implantación de la clase *Interfaz*.

Hasta aquí se ha descrito de forma lo más clara posible el desarrollo para la implantación de *Interfaz*. En realidad el proceso no fue exactamente igual al que aquí se mostró, ya que como se comprenderá, se hicieron multiples "brincos" entre todos los pasos, los cuales por razones obvias serían imposible describirlos completamente (IMAGINENSE !!!).

A continuación se muestran los Diagramas de Clases y de Transición de Estados con la implantación ya terminada.

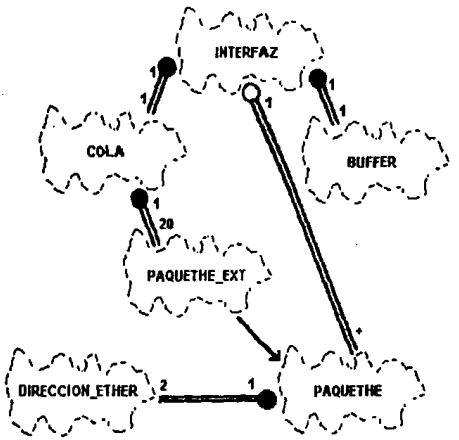

FIGURA 3.17 .- Diagrama final de las Clases.

CAPITULO IV

APLICACIONES DE LA INTERFAZ.

Como parte final del desarrollo de tesis se construyeron distintas aplicaciones con el objeto de probar y utilizar a la interfaz de red; aquí se mencionarán tres de ellas.

4.1.- PROGRAMA DE RECEPCION DE PAQUETES (RECIBE).

El objetivo de RECIBE es el de capturar a todos los paquetes que pasan a través de la red y desplegar en pantalla tanto el tipo como las direcciones fuente y destino de los mismos.

RECIBE fue bastante sencillo de implantar, se proporcionó una instancia de *Interfaz* (int_recibe) y se inicializó de modo que capturara a todos los paquetes de la red (int_recibe.Inicia(CUALQUIERA)), es decir, sin importar la dirección destino de los mismos. Después se construyó una función llamada Escribe_paquete cuyo único objetivo es desplegar en pantalla el tipo y las direcciones fuente y destino del paquete que se le proporcione. De este modo Int_recibe captura todos los paquetes de la red y se los da a Escribe_paquete, la cual se encarga de desplegarlos en pantalla.

RECIBE resultó ser un programa de gran utilidad para el desarrollo de otras aplicaciones, ya que era posible saber de forma muy fácil y rápida si un determinado paquete viajaba por la red. Por ejemplo en el desarrollo de el analizador de tráfico que más tarde veremos(MONRED), existen paquetes que son enviados a través de la red, los cuales generan paquetes respuesta por parte de las máquinas receptoras; usando RECIBE era muy fácil detectar si los paquetes que el analizador de tráfico generaba eran efectivamente enviados a través de la red, así como también saber si existían respuestas a los mismos.

El cuerpo principal de este programa es relativamente sencillo y se muestra a continuación:

```
main( int argc, char *argv[ ] )
{
        int num_carac;
        char *ap_ul;
        Interfaz int_recibe;

  if ( argc == 2 )
    int_recibe.Cambiar_num_int( (int) (strtol(argv[1],&ap_ul,16)) );
  else if ( argc > 2 )
```

```
  printf("\n ERROR : demasiados argumentos. ");
 int_recibe.Inicia( CUALQUIERA );
 clrscr();
 printf("\n Este es un programa de prueba para recibir paquethes");
 printf("\n \n La interrupción del manejador es : %04X ",int_recibe.Dame_n_int() );
 printf("\n El número de mi tarjeta es : ");
 for ( int i=0; i<6; i++ )
 {
  printf("%02X",(int_recibe.Dame_dir_local())[i]);
  if (i!=5) putchar(':');
 }

 do
 {
  if ( int_recibe.Recibe_paq( &paq, &num_carac ) )
    Escribe_paq( paq, num_carac );
 }
 while( !kbhit() );
 int_recibe.Termina();

 return 0;

}
```

FIGURA 4.1 .- Cuerpo principal de RECIBE.

Cabe mencionar que RECIBE no sólo sirvió para el desarrollo de aplicaciones montadas sobre la interfaz (*Interfaz*), sino que también fue utilizado de manera muy efectiva durante el desarrollo de la misma. Por ejemplo, la rutina de recepción de paquetes de *Interfaz* (Recibe_paq) no funcionó correctamente en el primer intento, fue entonces que conjuntamente con otra aplicación que más tarde se verá (ENVIA), resultó muy fácil detectar si Recibe_paq funcionaba correctamente.

4.2.- PROGRAMA DE ENVIO DE PAQUETES (ENVIA).

De manera semejante a RECIBE, ENVIA tiene como finalidad el envío de paquetes a través de la red. Estos paquetes son enviados en difusión (es decir con dirección destino FF:FF:FF:FF:FF:FF y por lo tanto son recibidos por todas las máquinas) y con el tipo que se les proporcione.

ENVIA resultó también muy fácil de implantar. Se proporcionó una instancia de *Interfaz* (int_envia) y un *buffer* (arreglo de caracteres) donde se colocan los datos; después se crea una instancia de *Paquethe* utilizando al *buffer* con los datos en él y se inicializa Int_envia de cualquier forma que se quiera.

Una vez creado todo esto comienza un ciclo donde cada vez que se presione cualquier tecla (excepto ESC) un paquete con los datos del *buffer* sera enviado. Enseguida mostramos el cuerpo principal de ENVIA:

```
main (int argc, char *argv[ ])
{
        unsigned char buffer_aux[ LTETHER ];
        char ch;
        unsigned int tipo;
        Interfaz int_envia;
        char *ap_ul;

 if ( int_envia.Inicia( CUALQUIERA ) )
 {
  printf("NO JALO");
  return 1;
 }
 if ( argc == 2 )
  int_envia.Cambiar_num_int( (int) (strtol(argv[1],&ap_ul,16)) );
 else if ( argc > 2 )
  printf("ERROR : Demasiados argumentos. ");
 clrscr();
 printf ("\n Este programa envia paquetes a través de la red ");
 printf ("\n \n Interrupción del manejador: %04X \n",int_envia.Dame_n_int());
 for( int i=0; i<6; i++ )
  buffer_aux[ i ] = 0xFF;

 printf("\nDame el tipo de paquetes a enviar( IP=0X0800, ARP=0X0806,
        LOOPBACK=0X9000): ");
 scanf("%X",&tipo);
 for( i=0; i<LDETHER; i++ )
  buffer_aux[ i+14 ]='A';

 Direccion_ether d_org( int_envia.Dame_dir_local() );
 Direccion_ether d_dst( buffer_aux );
 Paquethe paq( LDETHER, d_org, d_dst, tipo, buffer_aux );

 printf("\n\nDirección origen: ");
 for( i=0; i<6; i++ )
 {
  printf("%02X",(paq.Dame_dir_org())[ i ]);
  if (i!=5)  putchar (':');
 }
 printf("    Dirección destino: ");
 for( i=0; i<6; i++ )
 {
  printf("%02X",(paq.Dame_dir_dst())[ i ]);
```

```
  if (i!=5)  putchar (':');
 }
 printf("\n\n\nPresione <ESC> para terminar");
 printf("\n\nSe enviarán paquetes tipo %04X con cada teclazo . . . ",
 paq.Dame_tipo());

 do
 {
  int_envia.Envia_paq( &paq,LDETHER );
  ch = getch ();
 }
 while( ch!=27 );
 int_envia.Termina();

}
```

FIGURA 4.2 .- Cuerpo principal de ENVIA.

Con esto se ve que ENVIA es también bastante sencillo y como ya se dijo resultó de gran importancia para el desarrollo de otras aplicaciones más complejas; incluso sirvió para el desarrollo de *Interfaz* misma.

Utilizando estos programas resultó muy fácil hacer pruebas de la red, bastaba poner a ENVIA en una máquina y a RECIBE en otra que estuviera cercana (a 50 cm de distancia) y entonces se podía ver si lo que enviaba efectivamente se enviaba y de modo correcto.

4.3.- MONITOREADOR DE RED (MONRED).

La aplicación más importante que se desarrolló con *Interfaz* fue un MONitoreador de RED (MONRED), que como su nombre lo indica tiene como objetivo el monitoreo de una red *Ethernet* de forma agradable para el usuario.

MONRED se puede dividir en dos partes fundamentales: 1) el monitoreo de las máquinas definidas en el archivo "hosts.txt" y 2) la recepción de los paquetes que circulan por la red. De este modo el usuario está enterado al mismo tiempo tanto de los paquetes que circulan por la red, como del estado en que se encuentran las máquinas que se monitorean.

1) Monitoreo de Máquinas.

El monitoreo de máquinas tiene como objetivo el estar al tanto periodicamente del estado en que se encuentran las máquinas que se monitorean, es decir, si están

funcionando correctamente en lo que a la red se refiere. Esto es hecho mediante el envio y recepción de paquetes especiales de los protocolos IP y ARP de internet [Comer].

Al inicio del programa son leídos 3 archivos de configuración. El primero es un archivo llamado " hosts.txt " el cual contiene las direcciones IP y los nombres de las máquinas a monitorear. La lectura de este archivo produce la creación de una lista ligada de *Host's* con la cual se realizará el monitoreo. El segundo archivo llamado "anfitrio.txt" contiene la dirección IP y el nombre de la máquina que ejecuta al monitoreador; este sirve para la creación de una instancia de *Host* que representará a la máquina fuente. El tercer archivo llamado " monred.cfg " contiene los parámetros necesarios para la ejecución de MONRED. Cabe aquí notar que al inicio de MONRED no se tienen las direcciones *ethernet* de las máquinas de la lista, es por esto que es necesario utilizar el protocolo ARP para la obtención de las mismas.

De este modo el primer mensaje que se envía a una máquina es un mensaje de resolución de dirección IP (petición de ARP) [Comer]. En caso de obtener respuesta la dirección *ethernet* de esa máquina se actualiza en la lista, y de ahí en adelante los siguientes mensajes enviados serán simplemente mensajes de petición de eco-respuesta llamados normalmente "Ping's" (petición de eco IP-ICMP). En caso de no obtener respuesta los siguientes mensajes enviados a esa máquina seguirán siendo como el primero hasta que la respuesta adecuada se obtenga.

Existe un contador de espera para las respuestas de los mensajes enviados, el cual se va decrementando con el tiempo y sirve para determinar cuando una máquina se deje de monitorear y haya que pasar a la siguiente de la lista. En caso de obtener respuesta por parte de la máquina monitoreada, el tiempo de espera es respetado y se pasa a la siguiente una vez que termine este. A continuación mostramos a las dos rutinas que mandan paquetes de monitoreo:

```
void Ping_arp( void )
{
        int i;

        for ( i=0; i<LTETHER; i++ )
                buf_auxiliar[ i ] = '\x00';

        /* ---- DATOS ETHERNET ---- */

        for ( i=0; i<6; i++ )
                buf_auxiliar[ i ] = '\xFF';              // DIRECCION BROADCAST

        /* ------- DATOS ARP -------- */

        buf_auxiliar[ 14 ] = '\x00';                     // HARDWARE
        buf_auxiliar[ 15 ] = '\x01';                     //      TYPE
        buf_auxiliar[ 16 ] = '\x08';                     // PROTOCOL
        buf_auxiliar[ 17 ] = '\x00';                     //      TYPE
```

```
        buf_auxiliar[ 18 ] = '\x06';                    // HLEN
        buf_auxiliar[ 19 ] = '\x04';                    // PLEN
        buf_auxiliar[ 20 ] = PREGUNTA_ARP >> 8;         // OPER
        buf_auxiliar[ 21 ] = PREGUNTA_ARP;              //      ATION
        for ( i=0; i<6; i++ )
                buf_auxiliar[ 22+i ] = (anfitrion.Dame_dir_e())[ i ];
        for ( i=0; i<6; i++ )
        {
                buf_auxiliar[ 22+i ] = (anfitrion.Dame_dir_i())[ i ];
                buf_auxiliar[ 22+i ] = (siguiente_host->Dame_dir_i())[ i ];
        }

        Direccion_ether dir_org( anfitrion.Dame_dir_e() );
        Direccion_ether dir_dst( buf_auxiliar );

        Paquethe paq( 42, dir_org, dir_dst, 0X0806, buf_auxiliar+14 );
        int_monitor.Envia_paq( &paq, 42 );

}
```

FIGURA 4.3 .- Rutina de envio de paquetes de petición de ARP.

```
void Ping_icmp( void )
{
        unsigned int ip_checksum, icmp_checksum;
        int i;

        if ( identificacion == 0xFFFFFFFF )
                identificacion = 0x00000000;
        identificacion++;
        for ( i=0; i<LTETHER; i++ )
                buf_auxiliar[ i ] = '\x00';


        /* -------  DATOS IP  -------- */

        buf_auxiliar[ 14 ] = '\x45';                    // VERS Y HLEN
        buf_auxiliar[ 15 ] = '\x00';                    // SERVICE TYPE
        buf_auxiliar[ 16 ] = '\x00';                    // TOTAL
        buf_auxiliar[ 17 ] = '\x1E';                    // LENGHT
        buf_auxiliar[ 18 ] = identificacion >> 8;       // IDENT
        buf_auxiliar[ 19 ] = identificacion;            //      IFIER
        buf_auxiliar[ 20 ] = '\x00';                    // FLAGS Y OFF
        buf_auxiliar[ 21 ] = '\x00';                    //           SET
        buf_auxiliar[ 22 ] = '\xFF';                    // TIME TO LIVE
        buf_auxiliar[ 23 ] = P_ICMP;                    // PROTOCOL
```

```
for ( i=0; i<4; i++ )
{
        buf_auxiliar[ 26+i ] = (anfitrion.Dame_dir_i())[ i ];
        buf_auxiliar[ 30+i ] = (siguiente_host->Dame_dir_i())[ i ];
}
ip_checksum = Checksum( buf_auxiliar+14, 20 );
buf_auxiliar[ 24 ] = ip_checksum >> 8;      // IP_HEADER
buf_auxiliar[ 25 ] = ip_checksum;           //       CHECKSUM

/* ------  DATOS ICMP  ------ */

buf_auxiliar[ 34 ] = PREGUNTA_ICMP;         // TYPE
buf_auxiliar[ 35 ] = '\x00';                // CODE

buf_auxiliar[ 38 ] = identificacion >> 8;   // IDENT
buf_auxiliar[ 39 ] = identificacion;        //       IFIER
buf_auxiliar[ 40 ] = identificacion >> 8;   // SEQUENCE
buf_auxiliar[ 41 ] = identificacion;        //       NUMBER
for ( i=0; i<2; i++ )
        buf_auxiliar[ 42+i ] = '\x06';      // OPTIONAL DATA

icmp_checksum = Checksum( buf_auxiliar+34, 10 );
buf_auxiliar[ 36 ] = icmp_checksum >> 8;    // ICMP
buf_auxiliar[ 37 ] = icmp_checksum;         //       CHECKSUM

Direccion_ether dir_org( anfitrion.Dame_dir_e() );
Direccion_ether dir_dst( siguiente_host->Dame_dir_e() );

Paquethe paq( 44, dir_org, dir_dst, 0X0800, buf_auxiliar+14 );
int_monitor.Envia_paq( &paq, 44 );

}
```

FIGURA 4.4 .- Rutina de envio de paquetes de eco-respuesta de ICMP

El control de los tiempos de envio de mensajes así como el orden en que se monitorean las máquinas lo tiene una rutina llamada Monitorea.

Este monitoreo se lleva a cabo automáticamente durante la ejecución del programa y no es visible al usuario, así que la forma de ver si las máquinas han respondido o no es presionando la tecla "l", la cual nos da un listado de los resultados hasta ese momento.

Presionando la tecla "e" es posible ver dos pantallas de estadísticas. La primera muestra cuántos paquetes han sido tanto enviados como recibidos (los paquetes enviados son los de monitoreo) y la segunda muestra estadísticas referentes a los paquetes recibidos.

2) Recepción de los paquetes de la Red.

La recepción de paquetes de la red tiene dos finalidades: 1) el análisis de los paquetes para la obtención de estadísticas y 2) la posible recepción de las respuestas generadas por los paquetes enviados para monitoreo. Cabe decir aquí que las respuestas a los paquetes enviados también son contabilizadas y tomadas en cuenta para las estadísticas. Este proceso se lleva a cabo por una rutina llamada Analiza_despliega, la cual es invocada por una rutina de control llamada Monitorea.

El análisis y las estadísticas sirven para dar información acerca de cuántos paquetes han circulado en la red, así como el tamaño promedio de los mismos durante un lapso de tiempo. De igual modo sirven para identificar que protocolos han generado los paquetes recibidos. Es posible ver las estadísticas en todo momento con solo presionar la tecla "e". Las pantallas mostradas darán toda la información acumulada hasta entonces.

En el caso del tamaño promedio de los paquetes recibidos, no es posible promediar el tamaño de una cantidad arbitraria de paquetes, así que este se calcula con base en un lapso de tiempo determinado por la máquina.

Es posible observar la gráfica de tráfico de paquetes con solo presionar la tecla "g". La gráfica muestra un desplegado de el número de paquetes recibidos a lo largo del tiempo.

La gráfica, así como las estadísticas y los resultados de monitoreo, son construidos a partir de los datos correspondientes al momento de invocación, por lo tanto, las pantallas que se muestran en distintas ocasiones generalmente serán diferentes. Lo anterior se debe a que existe una constante actualización de los datos.

Presionando la tecla "ESC" el programa terminará su ejecución mostrando tanto las estadísticas como la gráfica finales.

4.3.1.- DISEÑO E IMPLANTACION.

El diseño de MONRED se desarrolló Orientado a Objetos y este fue el motivo por el que la implatación fue hecha en C++.

El problema a resolver era el siguiente:

" Diseñar un programa que utilizando a *Interfaz* monitoree máquinas o "*hosts*" conectados a una red *Ethernet* desde otro "*host*" de la misma, así como también formule estadísticas y gráficas acerca de los paquetes que circulan por esta. "

De lo anterior se identifican cuatro componentes del problema: "*hosts*", "estadísticas", "gráficas" y "paquetes". En el caso de los paquetes ya se tiene dentro de *Interfaz* a la clase que los representa, así que es necesaria la creación de por lo menos otras tres clases.

La primera de ellas es una clase que represente a los *hosts* de la red, la cual debe tener una dirección *ethernet*, una dirección ip y un nombre. Por facilidad y completez en el diseño primero se creó la clase *Direccion_ip* (Figura 4.5), la cual es una completa analogía de *Direccion_ether*. Esta tiene la finalidad de poder hacer instancias de direcciones IP y tratarlas como entes autónomos.

```
class Direccion_ip
{
        protected:

                unsigned char dir_ip[ 4 ];

        public:

                Direccion_ip();
                Direccion_ip( unsigned char * );
                Direccion_ip( Direccion_ip & );
                ~Direccion_ip(){ }
                void Llena_dir_ip ( unsigned char * );
                void Llena_dir2_ip ( int i, unsigned char c ){ dir_ip[ i ] = c; }
                unsigned char Extrae_c_ip ( int i ){ return dir_ip[ i ]; }
                unsigned char *Dame_dir_ip ( void ){ return dir_ip; }
                void operator =( const Direccion_ip & );
                friend int operator ==( Direccion_ip &, Direccion_ip & );
};
```

FIGURA 4.5 .- Implantación de la clase *Direccion_ip*.

Enseguida fue creada la clase *Host* (Figura 4.6) que se derivó de las clases *Direccion_ether* y *Direccion_ip*. Le fueron agregados un nombre y dos campos para el almacenamiento de los resultados del monitoreo. Esta clase resultó muy conveniente no solo para los *hosts* a monitorear, sino también para representar a la máquina donde MONRED se ejecute. Es decir que *Host* como su nombre lo indica puede representar a un *host* de la red y guardar los resultados de su correspondiente monitoreo.

```
class Host : public Direccion_ether, public Direccion_ip
{
        unsigned char nombre[ 30 ];
```

```
public:

        unsigned char respuesta_ping;
        unsigned long barrido_num;
        struct time hora_ult_ping;
        Host *a_siguiente;
        Host();
        Host( unsigned char * );
        Host( Direccion_ether &, Direccion_ip &, unsigned char * );
        ~Host(){ }
        void Llena_dir_e( unsigned char * );
        void Llena_dir_i( unsigned char * );
        void Llena_nombre( unsigned char * );
        unsigned char *Dame_dir_e ( void ){ return Dame_dir_ether(); }
        unsigned char *Dame_dir_i ( void ){ return Dame_dir_ip(); }
        Direccion_ether Dame_dir_Ether ( void );
        Direccion_ip Dame_dir_Ip ( void );
        unsigned char *Dame_nombre ( void ){ return nombre; }
        friend int operator ==( Host &, Host & );
};
```

FIGURA 4.6 .- Implantación de la clase *Host*.

Hasta aquí era posible guardar los resultados del monitoreo de cada *host*, pero no se tenía la forma de agruparlos y ordenarlos. La siguiente clase llamada *Lista* (Figura 4.7), es una lista ligada de instancias de *Host* la que sirve para tener un orden en el proceso de monitoreo.

```
class Lista
{
        public:

                Host *a_host;

                Lista(){ a_host = NULL; }
                ~Lista(){ }
                void Inserta_host( Host * );
                void Arma_lista( void );
                Host *Dame_host_ip( Direccion_ip & );
                Host *Dame_host_ether( Direccion_ether & );
                char Da_listado( void );
};
```

FIGURA 4.7 .- Implantación de la clase *Lista*.

En este punto era posible representar a los *hosts* a monitorear, pero faltaba la forma de representar a las estadísticas y gráficas generadas con los resultados; esto provocó la creación de otras dos clases *Estadisticas* y *Grafica*. El objetivo de ellas es que tanto las estadísticas como las gráficas sean independientes del código principal del programa y tengan comportamiento autosuficiente.

La clase *Estadisticas* (Figura 4.8) guarda los conteos tanto para el cálculo del tamaño promedio de los paquetes recibidos, como para el cálculo de estadísticas. Estos cálculos son llevados a cabo por la función de despliegue de estadísticas (Escribe_estadisticas) en el momento necesario. Los conteos de paquetes tanto enviados como recibidos son guardados en una estructura llamada datos.

```
class Estadisticas
{
        unsigned long suma;
        unsigned long paq_recibidos;
        struct time tiempo1, tiempo2, tiempo_aux;

        public:

                Datos datos;
                Datos datos_aux;
                Estadisticas();
                ~Estadisticas(){ }
                void Limpia_datos( Datos * );
                void Copia_datos( void );
                char Escribe_estadisticas( int, char * [ ], char );
                void Actualiza_tam_promedio( int );
                long Dame_tam_promedio( char );
};
```

FIGURA 4.8 .- Implantación de la clase *Estadisticas*.

La intención de las gráficas es mostrar al usuario el flujo de recepción de paquetes a lo largo del tiempo. El proceso de graficación resultó ser un poco más complejo que el proceso de estadísticas, así que con fines de hacerlo más práctico se creó una clase auxiliar llamada *Datos_grafica* (Figura 4.9), la cual como su nombre lo indica contiene los datos necesarios para que en cualquier momento se pueda crear una gráfica. Es decir que en intervalos de tiempo que se establecen al inicio del programa, se van creando instancias de una estructura llamada Paqtiempo, las cuales contienen los datos necesarios para el desplegado de una parte de la gráfica.

```
class Datos_grafica
{
        unsigned long num_datos;
        unsigned long num_paq_penultimo;
        unsigned long max_elementos;
        Paqtiempo *primero;
        Paqtiempo *ultimo;

        public:
                Datos_grafica();
                ~Datos_grafica();
                void Inserta_paqtiempo( Estadisticas & );
                int Dame_num_datos( void ){ return num_datos; }
                int Dame_max_elementos( void ){ return max_elementos; }
                Paqtiempo *Dame_primero( void ){ return primero; }
};
```

FIGURA 4.9 .- Implantación de la clase *Datos_grafica*.

La clase *Datos_grafica* recaba la información, mientras que la clase *Grafica* (Figura 4.10) solo se dedica al proceso de control y desplegado de la misma.

```
class Grafica
{
        char etapa;
        char buf_aux[ 3 ];
        int gdriver;
        int gmode;
        int gerr;
        Datos_grafica datos_grafica;
        char *tam_intervalo;

        public:

                Grafica();
                ~Grafica(){}
                void Llena_tam_intervalo ( int );
                void Reestablece_texto( void );
                void Termina_intervalo( Estadisticas &est );
                void Inicia_grafica ( void );
                int Pos_x( int, int );
                int Pos_y( unsigned long, unsigned long );
                void Traza_ejes( int, unsigned long );
                void Despliega_grafica( char );
                void Construye_grafica( char );
};
```

FIGURA 4.10 .- Implantación de la clase *Grafica*.

Con estas clases la tarea de implantación de MONRED resultó mucho más sencilla y eficiente. Enseguida se muestra el Diagrama final de clases de MONRED.

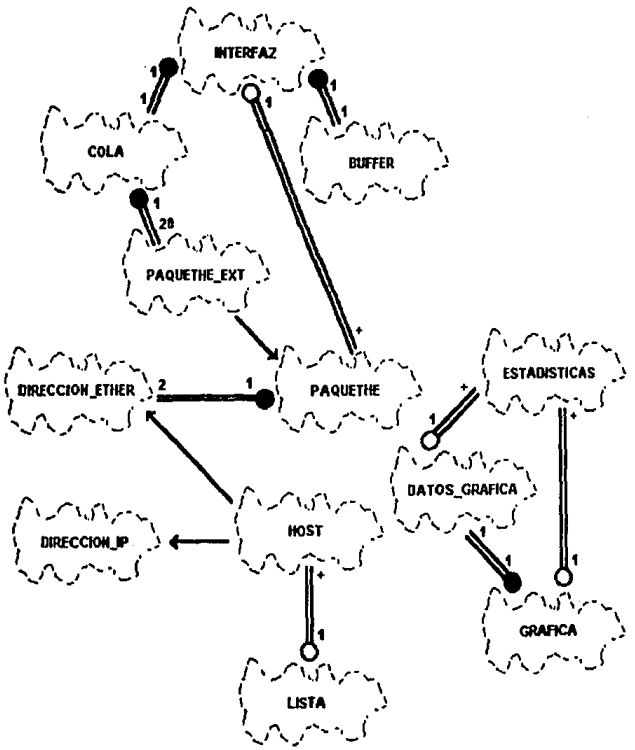

FIGURA 4.11 .- Diagrama de clases de MONRED.

En el cuerpo principal del programa las variables globales son:

```
unsigned char buf_auxiliar[ LTETHER ];
unsigned int identificacion = 0X0000;
```

```
int interv_paqtiempos;
int interv_respuesta;
int interv_no_monitorear;
Interfaz int_monitor;
Host anfitrion;
Lista lista;
Estadisticas estadisticas;
Grafica grafica;
Host *siguiente_host;
Host *host_monitoreado;
```

FIGURA 4.12 .- Variables globales de MONRED.

Las variables globales más importantes son: buf_auxiliar, que sirve para el envio de los paquetes de monitoreo; int_monitor, que es la instancia de *Interfaz* que establece la comunicación con los manejadores de Clarkson; lista, que es la instancia de *Lista* encargada de representar la lista con los *hosts* a monitorear; anfitrion, que es la representación de la máquina donde se ejecuta MONRED; estadisticas, que es la instancia de *Estadisticas* que las manejará; grafica, que es la instancia de *Grafica* encargada del manejo gráfico; y por último siguiente_host y *host* monitoreado, que sirven para el control de turnos de las máquinas a monitorear.

Las funciones principales de MONRED pueden clasificarse en 3 grupos principales:

1) Funciones de incio:

```
void Presenta( void );
void Lee_archivos( int argc, char *argv[ ] );
int Inicia( void );
```

Todas ellas con la finalidad de iniciar MONRED.

2) Funciones para el monitoreo:

```
void Monitorea( void );
int Checksum( void *a_buffer, int num_bytes );
void Ping_arp( void );
void Ping_icmp( void );
void Analiza_despliega( int );
```

Checksum, Ping_arp y Ping_icmp se encargan de llenar y mandar los paquetes de monitoreo; Analiza_despliega checa por las posibles respuestas a los mismos y Monitorea controla tanto al monitoreo como a los despligues estadísticos y gráficos que se requieran.

3) Funciones estadísticas:

```
void Analiza_despliega( int );
```

Aunque "estadisticas" se encarga por sí misma del desplegado de éstas, Analiza_despliega se encarga de actualizarlas, además de que muestra en pantalla los paquetes que se reciben en ese momento.

El cuerpo principal de MONRED resulta bastante sencillo de implantar y es mostrado a continuación:

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

        Presenta();
        Inicia( argc, argv );
        Monitorea();
        estadisticas.Escribe_estadisticas( argc, argv, FINAL );
        grafica.Construye_grafica( FINAL );

        return 0;

}
```

FIGURA 4.13 .- Cuerpo principal de MONRED.

A continuación se muestra el Diagrama de estados de MONRED.

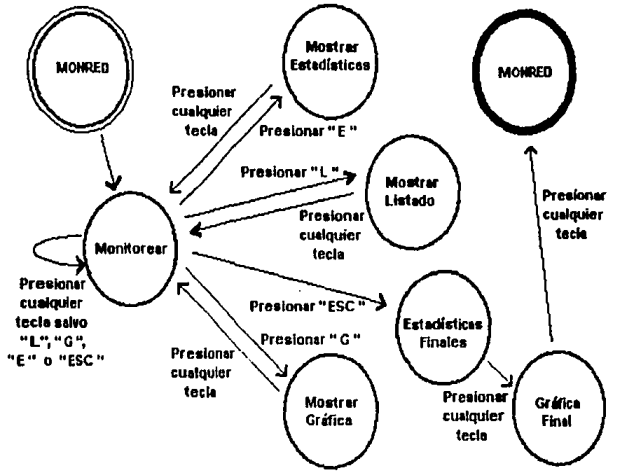

FIGURA 4.14 .- Diagrama de estados de MONRED.

Hasta aquí se han descrito tres de las principales aplicaciones que se han implantado sobre *Interfaz*, dejando la posibilidad para que futuros usuarios hagan uso de la misma y creen nuevas aplicaciones segun las necesidades.

Con estas aplicaciones se ha demostrado que el método Orientado a Objetos hace que el diseño de aplicaciones sea más parecido a la forma natural de pensar y por tanto el implantar cosas complejas que utilizan a otras, resulta mucho más fácil de visualizar, además que tanto la escritura como entendimiento del código resultan mucho más sencillos.

REFERENCIAS.

- [Comer]
Comer, Douglas E; Stevens, David L. " *Internetworking with TCP / IP* ". Volume I. Prentice Hall,*1988.

CONCLUSIONES

Como parte de las conclusiones finales quiero mencionar algunos de los problemas y ventajas que resultaron en el desarrollo de la interfaz utilizando el Método de Objetos.

Los problemas que se presentaron fueron de dos clases: 1) problemas de consistencia al utilizarse el método y 2) problemas por parte del compilador de C++.

Concluyo que el método de objetos nos da una muy buena base para la creación de diseños teóricos aunque no siempre prácticos de implantar. Por lo general el diseño resulta muy bonito, cómodo y fácil de mejorar, pero en el momento de la implantación no resulta lo mismo, ya que hubo varias veces que se diseñaba algo y resultaba imposible traducirlo a C++ o hacerlo funcionar en el marco de un compilador particular. Pienso que esto se debe a las carencias que presentan los compiladores de C++. En conclusión, en un diseño Orientado a Objetos algunas veces habrá que hacer cambios que alterarán un poco su filosofía con tal de que el programa funcione.

Me atrevo a decir que las aplicaciones Orientadas a Objetos resultan más faciles de usar que sus similares tradicionales, especialmente para aquellas personas que no son expertas programando (el código se comporta como se piensa). Por tanto considero que desarrollar Orientado a Objetos es mejor si se quieren producir aplicaciones que no necesitan de un alto rendimiento de tiempo y que serán utilizadas por muchas personas que desarrollarán sobre ellas.

La interfaz aquí realizada resulta muy comoda de utilizar en la creación de aplicaciones de red. El código generado por aplicaciones que la utilicen tiende a ocupar menos espacio que el generado por aplicaciones similares hechas en C. Por lo general las aplicaciones que la usan no son tan rápidas como aquellas implantadas de forma convencional.

La interfaz aquí presentada cumple con los objetivos propuestos de esta tesis, además de que puede ser utilizada por otros usuarios para la futura creación de aplicaciones de red orientadas a Objetos; por lo tanto considero que el trabajo aquí desarrollado pone a prueba el Método de Objetos y posee una utilidad práctica.

BIBLIOGRAFIA

- Booch, Grady. " *Object Oriented Design* ". The Benjamin / Cummings Publishing Company, Inc., 1991.

- Comer, Douglas E; Stevens, David L. " *Internetworking with TCP / IP* ". Volume I. Prentice Hall, 1988.

- Dewhurst, Stephen C; Stark, Kathy T. " *Programming in C++* ". Prentice Hall, 1989.

- Galavíz Casas, José. " *Diseño de Módulos para la Experimentación con Redes de Computadoras* ". Tesis de Licenciatura, 1993.

- *Internet Protocol.* RFC 791. Defense Advanced Research Projects Agency, 1981.

- Kernighan, Brian W; Ritchie, Dennis M. " *El lenguaje de Programación C* ". Prentice Hall Hispanoamericana, Segunda Edición, 1991.

- *PC / TCP Packet Driver Specification.* Revision 1.09. FTP Software, Inc., 1989.

- Plummer, David C. " *An Ethernet Address Resolution Protocol* ". RFC 826, 1982.

- Postel, J. " *Internet Control Message Protocol* ". RFC 792, 1981.

- Socolofsky, T. " *A TCP / IP tutorial* ". RFC 1180, 1991.

- Stevens, W Richard. " *UNIX Network Programming* ". Prentice Hall, 1990.

- Swan, Tom. " *Mastering Borland C++* ". Sams Publishing, 1992.

- Tanenbaum, Andrew S. " *Redes de Ordenadores* ". Prentice Hall. Segunda Edición, 1991.

- Thorne, Michael. " *Computer Organization and Assembly Language Programming* ". The Benjamin / Cummings Publishing Company, Inc. Second Edition, 1990.

- Stevens, W Richard. " *UNIX Network Programming* ". Prentice Hall, 1990.

APENDICE

En este apéndice se muestran los formatos de las tres principales clases desarrolladas en el trabajo de tesis. Estas clases son: *Direccion_ether*, *Paquethe* e *Interfaz*.

1) Formatos de Clases:

DIRECCION_ETHER

Nombre:	Direccion_ether.
Documentación:	Sirve para representar las direcciones de las tarjetas de las redes *Ethernet*.
Visibilidad:	Exportada.
Cardinalidad:	n.
Clases que usa:	Ninguna clase.
Operaciones y campos de acceso público:	Direccion_ether, Direccion_ether(unsigned char *), Direccion_ether(Direccion_ether &), Llena_dir_ether, Llena_dir2_ether, Extrae_c_ether, Dame_dir_ether, operador =.
Operaciones y campos de acceso privado:	dir_ether.
Diagrama de estados:	Figura 3.4.
Complejidad de espacio:	6 *bytes*.
Persistencia:	Transitoria.

PAQUETHE

Nombre:	Paquethe.
Documentación:	Sirve para representar a los paquetes de las redes *Ethernet*.
Visibilidad:	Exportada.
Cardinalidad:	n.
Clases que usa:	Direccion_ether.
Operaciones y campos de acceso público:	Paquethe, Paquethe(int, unsigned char *), Paquethe(int, Direccion_ether &, Direccion_ether &, int, unsigned char *), Paquethe(Paquethe &), Llena_paquethe, Dame_dir_dst, Dame_dir_org, Dame_dir_Dst, Dame_dir_Org, Dame_tipo, Dame_datos, operador =.
Operaciones y campos de acceso privado:	dst, org, tipo, datos.
Diagrama de estados:	Figura 3.6.
Complejidad de espacio:	1514 *bytes*.
Persistencia:	Transitoria.

INTERFAZ

Nombre:	Interfaz.
Documentación:	Sirve como interfaz de comunicación entre los manejadores de Clarkson y los programas de aplicación en redes *Ethernet*.
Visibilidad:	Exportada.
Cardinalidad:	1.
Clases que usa:	Direccion_ether, Paquethe, Paquethe_ext, Cola, Buffer.
Operaciones y campos de acceso público:	Interfaz, Cambiar_num_int, Dame_n_int, Inicia, Recibe_paq, Envia_paq, Tipo_recepcion, Dame_dir_local, Dame_dir_Local, Termina.
Operaciones y campos de acceso privado:	num_man, num_int, dir_local, cola, buffer, A_I, Recepcion.
Diagrama de estados:	Figura 3.15.
Complejidad de espacio:	33370 *bytes*.
Persistencia:	Transitoria.

2) Formatos de Operaciones:

DIRECCION_ETHER

Nombre:	Direccion_ether.
Documentación:	Crea instancias da la clase Direccion_ether.
Categoría:	Constructora.
Acción:	Crea una instancia de Direccion_ether inicializando la dirección en cero.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	Una instancia de Direccion_ether.
Postcondiciones:	La dirección resultante es cero.

DIRECCION_ETHER

Nombre:	Direccion_ether(unsigned char *).
Documentación:	Crea instancias de la clase Direccion_ether.
Categoría:	Constructora.
Acción:	Crea una instancia de Direccion_ether inicializando la dirección con los datos apuntados por el parámetro.
Parámetros formales:	unsigned char *.
Precondiciones:	El parámetro debe apuntar a una dirección de *Ethernet*.
Resultados:	Una instancia de Direccion_ether.
Postcondiciones:	Ninguna.

DIRECCION_ETHER

Nombre:	Direccion_ether(Direccion_ether &).
Documentación:	Crea instancias de la clase Direccion_ether.
Categoría:	Constructora.
Acción:	Crea una instancia de Direccion_ether inicializando la dirección con los datos referenciados por el parámetro.
Parámetros formales:	Direccion_ether &.
Precondiciones:	El parámetro debe ser una referencia a una instancia existente de Direccion_ether.
Resultados:	Una instancia de Direccion_ether.
Postcondiciones:	La dirección resultante es igual a la del parámetro.

DIRECCION_ETHER

Nombre:	Llena_dir_ether.
Documentación:	Modifica la dirección de la instancia invocadora.
Categoría:	Modificadora.
Acción:	Modifica la dirección de la instancia invocadora utilizando el parámetro.
Parámetros formales:	unsigned char *.
Precondiciones:	El parámetro debe apuntar a una dirección de *Ethernet*.
Resultados:	Ninguno.
Postcondiciones:	Ninguna.

DIRECCION_ETHER

Nombre:	Llena_dir2_ether.
Documentación:	Modifica un caracter de la dirección almacenada en la instancia invocadora.
Categoría:	Modificadora.
Acción:	Modifica el caracter que se le diga de la instancia invocadora.
Parámetros formales:	int, unsigned char.
Precondiciones:	El entero debe ser un número de 0 a 5 y el caracter del parámetro debe ser un número hexadecimal.
Resultados:	Ninguno.
Postcondiciones:	Ninguna.

DIRECCION_ETHER

Nombre:	Extrae_c_ether.
Documentación:	Extrae un caracter de la dirección almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Extrae el caracter que se le diga de la instancia invocadora.
Parámetros formales:	int.
Precondiciones:	El parámetro debe ser un número de 0 a 5.
Resultados:	unsigned char.
Postcondiciones:	El caracter regresado es un número hexadecimal.

DIRECCION_ETHER

Nombre:	Dame_dir_ether.
Documentación:	Regresa un apuntador a la dirección almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa un apuntador a la dirección almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	unsigned char *.
Postcondiciones:	Ninguna.

DIRECCION_ETHER

Nombre:	operador =.
Documentación:	Asigna una Direccion_ether a otra.
Categoría:	Modificadora.
Acción:	Copia la Direccion_ether fuente en la Direccion_ether destino.
Parámetros formales:	Direccion_ether &.
Precondiciones:	Ninguna.
Resultados:	Ninguno.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Paquethe.
Documentación:	Crea instancias de la clase Paquethe.
Categoría:	Constructora.
Acción:	Crea una instancia de Paquethe inicializando sus campos en cero.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	Una instancia de Paquethe.
Postcondiciones:	Los campos resultantes son cero.

PAQUETHE

Nombre:	Paquethe(int, unsigned char *).
Documentación:	Crea instancias de la clase Paquethe.
Categoría:	Constructora.
Acción:	Crea una instancia de Paquethe iniciálizandolo con los datos apuntados por el segundo parámetro.
Parámetros formales:	int, unsigned char *.
Precondiciones:	El entero debe ser el número de datos que contiene el paquete.
Resultados:	Una instancia de Paquethe.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Paquethe(int, Direccion_ether &, Direccion_ether &, int, unsigned char *).
Documentación:	Crea instancias de la clase Paquethe.
Categoría:	Constructora.
Acción:	Crea una instancia de Paquethe inicializándolo con los datos de los parámetros.
Parámetros formales:	int, Direccion_ether &, Direccion_ether &, int, unsigned char *.
Precondiciones:	El primer parámetro es el número de datos que contiene el paquete, el segundo es la dirección origen, el tercero es la dirección destino, el cuarto es el tipo de paquete de *Ethernet*, y el quinto son los datos del paquete.
Resultados:	Una instancia de Paquethe.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Paquethe(Paquethe &).
Documentación:	Crea instancias de la clase Paquethe.
Categoría:	Constructora.
Acción:	Crea una instancia de Paquethe iniciálizandolo con los datos referenciados por el parámetro.
Parámetros formales:	Paquethe &.
Precondiciones:	El parámetro refiere a un Paquethe existente.
Resultados:	Una instancia de Paquethe.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Llena_paquethe.
Documentación:	Modifica el contenido de la instancia invocadora.
Categoría:	Modificadora.
Acción:	Modifica el contenido de la instancia invocadora copiando los datos del Paquethe apuntado por el parámetro.
Parámetros formales:	Paquethe *.
Precondiciones:	El parámetro apunta a un Paquethe existente.
Resultados:	Ninguno.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Dame_dir_dst.
Documentación:	Regresa un apuntador a la dirección destino almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa un apuntador a la dirección destino almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	unsigned char *.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Dame_dir_org.
Documentación:	Regresa un apuntador a la dirección origen almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa un apuntador a la dirección origen almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	unsigned char *.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Dame_dir_Dst.
Documentación:	Regresa una instancia de Direccion_ether inicializada con la dirección destino almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa una instancia de Direccion_ether inicializada con la dirección destino almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	Direccion_ether.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Dame_dir_Org.
Documentación:	Regresa una instancia de Direccion_ether inicializada con la dirección origen almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa una instancia de Direccion_ether inicializada con la dirección origen almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	Direccion_ether.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Dame_tipo.
Documentación:	Regresa el tipo almacenado en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa el tipo almacenado en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	unsigned int.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	Dame_datos.
Documentación:	Regresa un apuntador al campo de datos de la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa un apuntador al campo de datos de la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	unsigned char *.
Postcondiciones:	Ninguna.

PAQUETHE

Nombre:	operador =.
Documentación:	Asigna un Paquethe a otro.
Categoría:	Modificadora.
Acción:	Copia el Paquethe fuente al Paquethe destino.
Parámetros formales:	Paquethe &.
Precondiciones:	Ninguna.
Resultados:	Ninguno.
Postcondiciones:	Ninguna.

INTERFAZ

Nombre:	Interfaz.
Documentación:	Crea instancias de la clase Interfaz.
Categoría:	Constructora.
Acción:	Crea una instancia de Interfaz inicializando num_int=0x60, A_I=this.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	Una instancia de Interfaz.
Postcondiciones:	num_int se inicia con 0x60.

INTERFAZ

Nombre:	Cambiar_num_int.
Documentación:	Modifica el valor de num_int de la instancia invocadora.
Categoría:	Modificadora.
Acción:	Modifica el valor de num_int de la instancia invocadora utilizando el parámetro.
Parámetros formales:	int.
Precondiciones:	El entero debe ser un número de interrupción (0-255).
Resultados:	Ninguno.
Postcondiciones:	Ninguna.

INTERFAZ

Nombre:	Dame_n_int.
Documentación:	Proporciona el valor de num_int de la instancia invocadora.
Categoría:	Observadora.
Acción:	Proporciona el valor de num_int de la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	int.
Postcondiciones:	El entero regresado es un número de interrupción (0-255).

INTERFAZ

Nombre:	Inicia.
Documentación:	Inicia la comunicación de la instancia invocadora con los Manejadores de Clarkson.
Categoría:	Modificadora.
Acción:	Inicia la comunicación de la instancia invocadora con los Manejadores de Clarkson.
Parámetros formales:	unsigned int.
Precondiciones:	El parámetro es el tipo de paquetes que se quieren poder recibir.
Resultados:	int.
Postcondiciones:	El valor regresado es EXITO_FUNC si hay éxito y el código de error si no.

INTERFAZ

Nombre:	Recibe_paq.
Documentación:	Recibe un paquete proveniente de la red.
Categoría:	Modificadora.
Acción:	Recibe un paquete proveniente de la red.
Parámetros formales:	Paquethe *, int *.
Precondiciones:	El primer parámetro es la instancia de Paquethe donde se almacenará el paquete recibido, el segundo es el entero donde se almacenará el número de caracteres recibidos.
Resultados:	int.
Postcondiciones:	El valor regresado es 1 si hay éxito y 0 si no.

INTERFAZ

Nombre:	Envia_paq.
Documentación:	Envía un paquete hacia la red.
Categoría:	Modificadora.
Acción:	Envía un paquete hacia la red.
Parámetros formales:	Paquethe *, int.
Precondiciones:	El primer parámetro es la instancia de Paquethe que contiene el paquete a enviar, el segundo es el número de caracteres que serán enviados.
Resultados:	int.
Postcondiciones:	El valor regresado es EXITO_FUNC si hay éxito y ERROR_FUNC si no.

INTERFAZ

Nombre:	Tipo_recepción.
Documentación:	Cambia el tipo de paquetes que se quieren poder recibir en la instancia invocadora.
Categoría:	Modificadora.
Acción:	Cambia el tipo de paquetes que se quieren poder recibir en la instancia invocadora.
Parámetros formales:	int.
Precondiciones:	El parámetro es el tipo de paquetes que se desea recibir.
Resultados:	int.
Postcondiciones:	El valor regresado es EXITO_FUNC si hay éxito y el código de error si no.

INTERFAZ

Nombre:	Dame_dir_local.
Documentación:	Regresa un apuntador a la dirección local almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa un apuntador a la dirección local almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	unsigned char *.
Postcondiciones:	El resultado apunta a la dirección de la tarjeta de *Ethernet*.

INTERFAZ

Nombre:	Dame_dir_Local.
Documentación:	Regresa una instancia de Direccion_ether inicializada con la dirección local almacenada en la instancia invocadora.
Categoría:	Observadora.
Acción:	Regresa una instancia de Direccion_ether inicializada con la dirección local almacenada en la instancia invocadora.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	Direccion_ether.
Postcondiciones:	Ninguna.

INTERFAZ

Nombre:	Termina.
Documentación:	Termina la comunicación de la instancia invocadora con los Manejadores de Clarkson.
Categoría:	Modificadora.
Acción:	Termina la comunicación de la instancia invocadora con los Manejadores de Clarkson.
Parámetros formales:	Ninguno.
Precondiciones:	Ninguna.
Resultados:	int.
Postcondiciones:	El valor regresado es EXITO_FUNC si hay éxito y el código de error si no.