CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL IPN

Departamento de Ingenieria Eléctrica.
Sección de Computación.

Protocolos de Comunicación para la Computación Distribuida

Tesis que presenta el **ING. ALEXANDRO HERNANDEZ LICEAGA**
para obtener el grado de

MAESTRO EN CIENCIAS

en la especialidad de:
INGENIERIA ELECTRICA CON OPCION EN COMPUTACION

Trabajo dirigido por el Dr. Jan Janecek Hyan.

México, D.F., Junio de 1990.

"No necesito ser perdonado por amarte tanto".
Silvia, este trabajo es para ti.

A mis padres, Salvador y Margarita,
por todo cuanto han hecho por mí.
Por darme amor y libertad, respeto y protección.

A mi hermana Marisol,
un ejemplo de dedicación al trabajo,
alguien que siempre ha creido en mí.

A mi hermano Salvador,
amigo de verdad y confidente en noches de insomnio.

Agradecimientos

Deseo hacer patente mi agradecimiento a las siguientes personas:

Al Dr. Jan Janecek, quien me ofreció realizar el presente trabajo. Su guía, consejos y observaciones son invaluables. Mi deuda para con él excede con mucho los aspectos meramente académicos.

Al Dr. Armando Maldonado Talamantes y al M. en C. Carlos E. Hirsch Gabievich, por su paciencia en la revisión del presente trabajo y sus valiosas observaciones.

A todos los profesores que ha contribuido a mi formación. En particular, a la planta docente del CINVESTAV.

A mis compañeros de este Centro de Estudios. Conocer a tantas personas con visiones tan diferentes ha sido una experiencia sumamente enriquecedora.

Al personal del Departamento de Ingeniería Eléctrica de la Universidad Autónoma Metropolitana-Iztapalapa, en particular al Ing. Francisco J. Olvera Hernández, por ofrecerme la oportunidad de colaborar con ellos.

A la Lic. Graciela Román Alonso, por sus palabras de apoyo en momentos de desaliento.

Para con todos ellos no tengo sino una infinita gratitud.

Alexandro Hernández Liceaga.

Indice

I. Introducción

I.1 Objetivos.

1. Proponer alternativas para protocolos de comunicación orientados al control automático de procesos industriales que permitan una mayor seguridad y eficiencia en sistemas distribuidos de este tipo.

2. Evaluar los protocolos propuestos.

I.2 Arquitectura OSI de ISO.

Al hablar de una arquitectura, nos referimos a la apariencia que presenta un sistema hacia el exterior. Al hablar de una arquitectura de redes de computadoras aludimos a la forma en que se organizan las diferentes tareas que se realizan en un sistema de este tipo.

A medida que la complejidad de un sistema aumenta, se hace necesario dividir un problema complejo en partes manejables. Es por eso (y por otras razones que veremos más adelante) que un modelo monolítico de una red de computadoras se vuelve inoperante, y se hace necesario presentar un modelo manejable.

La arquitectura OSI (*Open System Interconnection:* Interconexión de Sistemas Abiertos) fue propuesta por la ISO (*International Standards Organization:* Organización Internacional de Normas). En esta arquitectura nos encontramos con una estructura jerarquizada de niveles (figura 1.1).

Las principales razones que llevaron al establecimiento de las 7 capas son [TAN82]:

1. Una capa representa un cierto nivel de abstracción.
2. Una capa realiza una función bien definida.
3. La intención es crear normas internacionales para cada capa.
4. Las interferencias entre las capas deben minimizar el flujo de información entre ellas.
5. El número de capas será suficientemente grande para distinguir entre funciones y suficientemente pequeño para que sea manejable.

De esta manera, se propuso esta arquitectura, que actualmente es de uso ampliamente extendido.

Dentro de esta arquitectura, los 3 niveles más bajos corresponden a funciones directamente relacionados con la red, mientras los niveles superiores corresponden a funciones de aplicación.

Las funciones de los 7 diferentes niveles son las siguientes:

Nivel 1. (Nivel Físico). Encargado de la transmisión de bits sobre el canal de comunicación.

Nivel 2. (Nivel de Enlace de Datos). Mostrar a las capas superiores de la arquitectura un canal de comunicación libre de errores.

Nivel 3. (Nivel de Red). Enrutamiento de paquetes de información dentro de la red.

Nivel 4. (Nivel de Transporte). Proporcionar a los niveles superiores un servicio independiente de red.

Nivel 5. (Nivel de Sesión). Establecer una sesión de comunicación entre 2 procesos de aplicación.

Nivel 6. (Nivel de Presentación). Permite el intercambio de información con diferentes formatos.

Nivel 7. (Nivel de Aplicación). Nivel reservado al desarrollo de aplicaciones para satisfacer necesidades específicas del usuario.

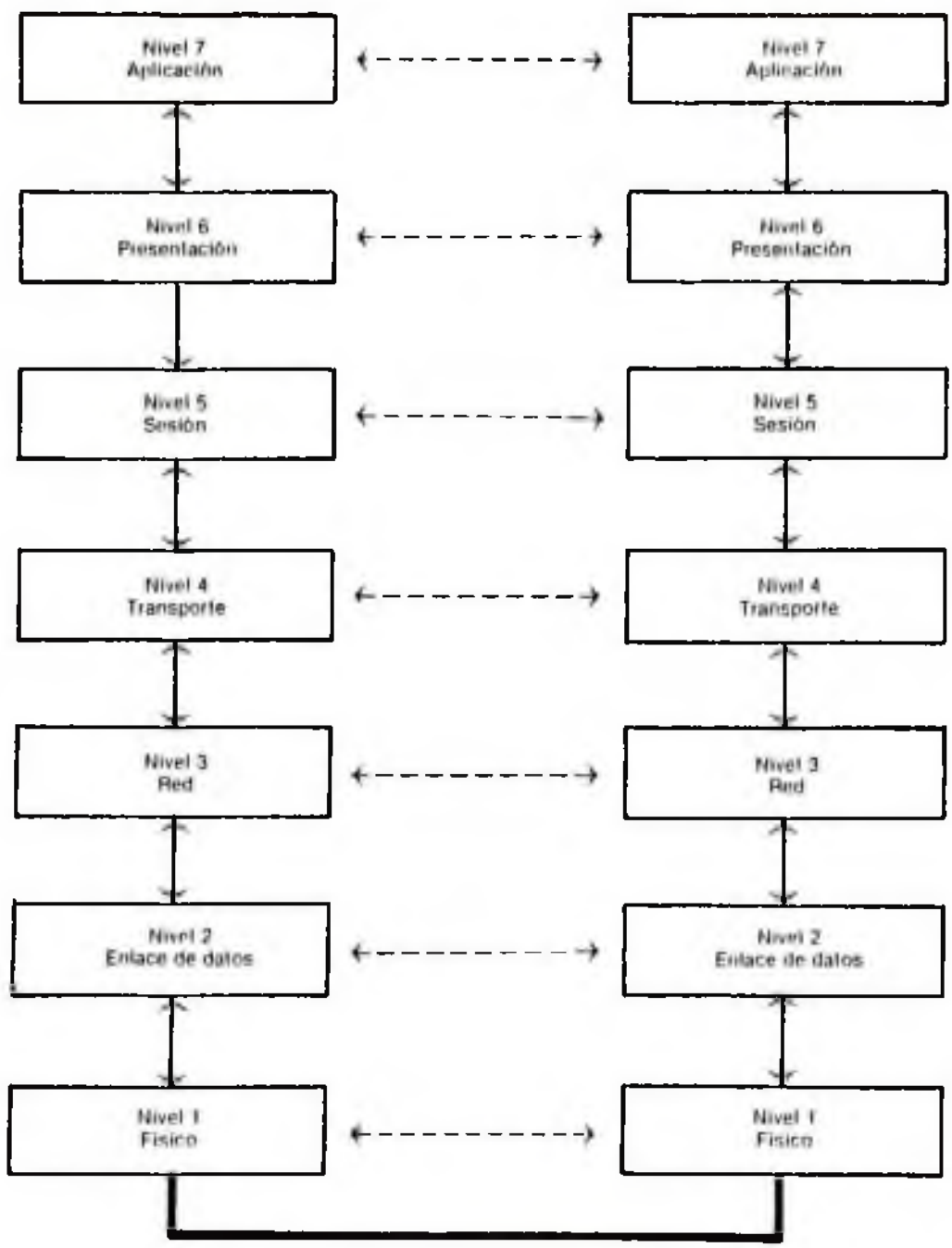

Figura 1.1 Arquitectura OSI de ISO.

Para una discusión amplia sobre los diferentes niveles de esta arquitectura puede consultarse [TAN82], [YAK83], [HUT88].

I.3 Nivel de enlace de datos en redes locales.

En el caso de las redes locales la división de niveles no corresponde exactamente a la de la arquitectura OSI. La figura 1.2 muestra la forma en que se organiza una red local. Las funciones que corresponden al nivel de enlace de datos en este modelo corresponden a funciones de los niveles 1, 2 y 3 de la arquitectura OSI.

El nivel de enlace de datos tiene dos funciones. Debido a esto, este nivel a su vez se divide en 2 subcapas. La subcapa de control de acceso al medio (MAC, por *Medium Access Control*), permite que diferentes estaciones puedan compartir el mismo medio de comunicación. Esta subcapa es dependiente del medio utilizado, y sus funciones corresponderían a las que define el nivel 3 de OSI. Por otra parte, tenemos la subcapa de control de enlace lógico (LLC, por *Logical Link Control*), que define los tipos de servicio que proporciona la capa de enlace de datos. En particular, se refiere a tener un servicio para comunicación sin conexión virtual (por datagramas) o un servicio orientado a comunicaciones punto a punto. Esta subcapa es independiente del medio. Las funciones de esta capa corresponden a las del nivel 2 y parte del nivel 1 de la arquitectura OSI.

Figura 1.2 Arquitectura para redes locales.

Los protocolos de nivel 2 que se describen en la siguiente sección corresponden a la subcapa MAC. Un protocolo de la subcapa LLC orientado a conexiones punto a punto es HDLC. Descripciones de HDLC y protocolos similares se pueden encontrar en [YAC83], [INT85].

I.4 Descripción operativa de protocolos de nivel 2.

Los protocolos que se mencionarán a continuación son utilizados en redes locales. Las redes locales tienen como característica principal estar limitadas a, cuando más, algunos miles de metros. Pensando en una red local ubicada en un ambiente industrial, es más o menos evidente que el número de estaciones que se encuentran en una red de este tipo también es limitado.

Otra consideración que es necesario hacer es acerca de la topología de la red que se usa para estos protocolos. En lo que sigue, consideraremos que la topología de la red es de tipo *bus*.

I.4.1 CSMA/CD.

CSMA/CD (*Carrier Sense Multiple Access with Collision Detection:* Acceso Múltiple por Sensado de Portadora con Detección de Colisiones) es un protocolo de acceso a medio bastante popular, como consecuencia de su implantación en la red Ethernet.

CSMA/CD (*Carrier Sense Multiple Access with Collision Detection:* Acceso Múltiple por Sensado de Portadora con Detección de Colisiones) es un protocolo de acceso al medio bastante popular, como consecuencia de su implantación en la red Ethernet.

i) Un paquete se transmite sin problemas.
ii) Se intentaron transmitir 2 o más paquetes al mismo tiempo, perdiéndose ambos (colisión).
iii) El canal está inactivo.

La detección de la portadora consiste en determinar si en cierto momento la red se encuentra libre. Una estación cualquiera no podrá transmitir a menos que el canal de comunicación se encuen-

tre libre. Con este fin estará "escuchando" la red hasta que no detecte actividad en la misma. Si ya no detecta actividad en la red, entonces podrá enviar su información al canal sin mayor trámite, con la esperanza de que no ocurra ningún error.

Este mecanismo seria muy simple si no fuera porque puede ocurrir que dos o más estaciones deseen transmitir y al sensar la portadora noten que hay actividad en la red, por lo tanto inhibiendo su transmisión. En el momento que la red se encuentre libre, las estaciones detectarán ese hecho *al mismo tiempo*, y enviarán su paquete a la red, lo que provocará que los datos enviados se revuelvan. Esta situación es conocida como colisión.

Una vez que se ha detectado la existencia de la colisión (lo que se puede ver al percatarse de que lo enviado no corresponde a lo recibido) es necesario detener el envio del paquete que se está enviando y reintentar la transmisión.

Un nuevo problema se ve venir. Si todas las estaciones reintentaran la transmisión, es inminente una nueva colisión, lo que provocará nuevos intentos de transmisión por parte de las estaciones, lo que ocasionará nuevas colisiones, y lo que significará el bloqueo de la red. Para evitar esta situación tan poco alagueña CSMA/CD tiene un mecanismo para la resolución de las colisiones.

Tal mecanismo consiste simplemente en esperar un intervalo *aleatorio* de tiempo, y después revisar nuevamente si el canal está desocupado. Esto garantizará que al menos la estación que esperó el intervalo más corto de tiempo tendrá acceso a la red.

En [FRA82] se incluye una descripción completa de Ethernet, donde se puede ver con detalle el mecanismo CSMA/CD.

En la figura 1.3 se muestra un diagrama de tiempo en donde se ilustra el mecanismo de operación anteriormente descrito. Se tienen 4 estaciones. La nomenclatura es la siguiente:

C_s : Detección de actividad en la red.
M : Mensaje de información.
C : Colisión de mensajes.
t_{al} : Tiempo de espera aleatorio.

En t_1 la estación 1 sensa el canal y al no detectar actividad comienza a transmitir su mensaje. En t_2 la estación 3 sensa el canal y detecta que la estación 1 está transmitiendo, así que se espera a que ésta desocupe el canal para iniciar su transmisión. En t_3 la estación 4 sensa el canal y lo detecta ocupado. Lo mismo le ocurre a la estación 2 en t_4. En t_5 ambas detectan el canal desocupado e inician la transmisión, lo que provoca una colisión. Ambas abortan su transmisión y esperan algún tiempo más para volverla a intentar. La estación 2 vuelve a sensar el canal en t_6, lo detecta ocupado y se espera hasta que la estación 1 deja de transmitir. Entonces puede transmitir correctamente su mensaje. En cambio, la estación 4 vuelve a intentar la transmisión en t_7, vuelve a colisionar. ¿Cuántas veces colisionará el mensaje transmitido por una estación antes de poder ser transmitido correctamente? Esto es algo que no está determinado, por lo que el tiempo máximo de acceso de un mensaje a la red tampoco está determinado.

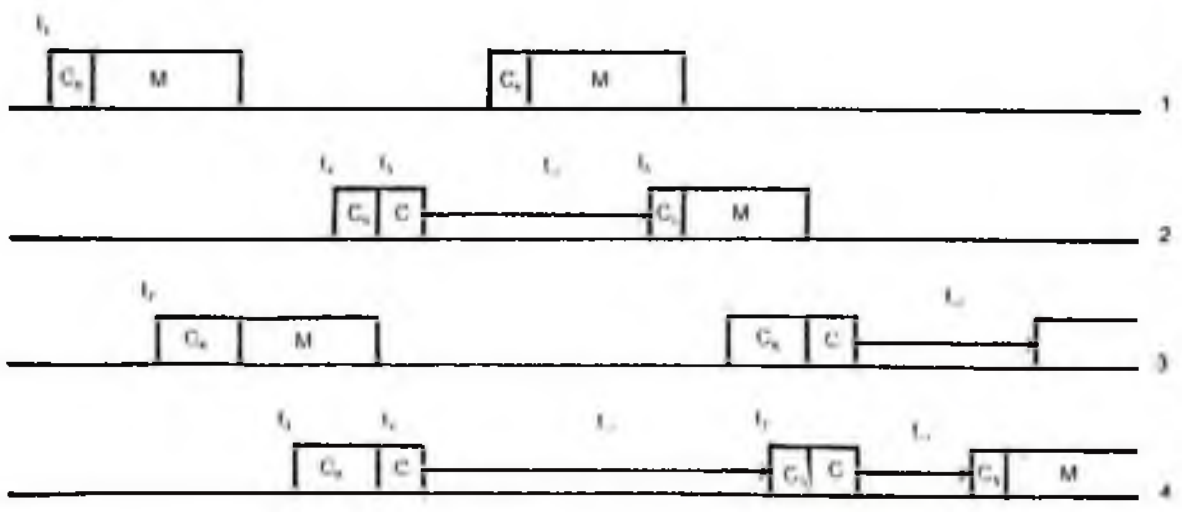

Figura 1.3 Mecanismo de acceso CSMA/CD.

I.4.2 Protocolos por sondeo.

Un protocolo por sondeo (*polling*) es un protocolo centralizado, en el sentido de que es necesario tener una estación maestra que habilite la comunicación de las restantes estaciones. Esto equivale a asumir el control absoluto de la red.

El mecanismo de operación es simple. En la red existe una estación maestra y varias estaciones subordinadas. Una estación subordinada no puede enviar nada a la red a menos que la estación maestra se lo permita.

Para lograr que todas las estaciones puedan transmitir, la estación maestra constantemente está sondeando (es decir, enviando mensajes de habilitación) a todas las subordinadas.

El tiempo máximo de acceso a la red con un protocolo por sondeo viene dado por:

$$t_{máx} = N(t_{pol} + t_{msg})$$

donde:

- N número de estaciones en la red.
- t_{pol} tiempo de envio del mensaje de sondeo.
- t_{msg} tiempo de envio de la trama más larga posible.

La figura 1.4 muestra de manera gráfica el mecanismo de operación de un protocolo por sondeo. La nomenclatura para tal figura es la siguiente:

Pi : Mensaje de sondeo (*polling*) a la estación 1.
M : Mensaje de información.
$t_{mák}$: Tiempo máximo para que una estación envíe un mensaje de información.

En el tiempo t_1 la estación 1 (que funciona como directora del tráfico en la red) envia un mensaje de sondeo a la estación 2. Esta tiene un mensaje que enviar y lo puede hacer en t_2. Lo mismo se repite para las estaciones 3 y 4.

Veamos ahora qué sucede en t_3. En este momento, la estación maestra envia un mensaje de sondeo a la estación 2, pero ésta no tiene ningún mensaje que transmitir, por lo que no contesta. Después de pasado cierto tiempo, la estación maestra se da cuenta que el mensaje de sondeo recién enviado no será contestado, y entonces envia la sonda a la siguiente estación, que es la estación 3.

Esto ilustra las particularidades de este protocolo de comunicación.

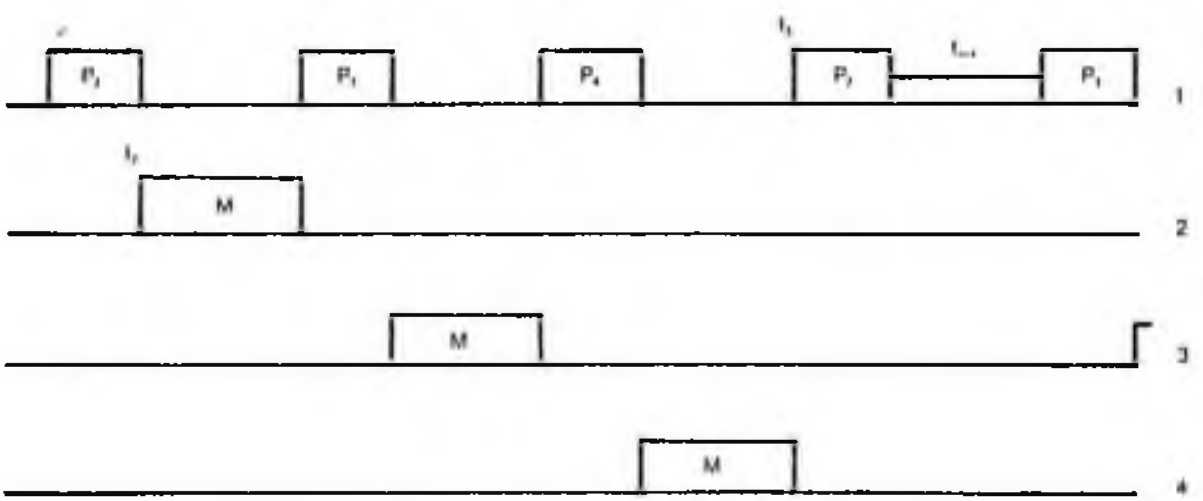

Figura 1.4 Mecanismo de acceso por sondeo.

I.4.3. Anillos Lógicos.

Los anillos lógicos pueden verse como implantaciones de *token ring* en una topologia de *bus*. También pueden ser vistos como una implantación descentralizada de un protocolo por sondeo.

En el inicio de operación de una red con este protocolo una estación y sólo una estación tiene acceso a la red. Si tiene algún paquete para transmitir, lo envia. Si no es asi, o si la transmisión ha

concluido, envía un *token* (que aquí tiene el significado de estafeta) a la estación que le sucede lógicamente. Ahora esta estación puede transmitir, y entonces deberá enviar el *token* a la estación que le sucede. De esta manera se llega a la última estación, la cual podrá transmitir y posteriormente deberá enviar el *token* a la primera estación, con lo cual se forma el anillo lógico deseado.

El tiempo máximo de acceso a la red puede considerarse igual al que se tiene para un protocolo por sondeo, sustituyendo el valor de t_{pol} por t_{tok}.

Una descripción detallada de un anillo lógico se incluye en [ISA85], en donde se propone este método de acceso a la red como un *standard* para una red de control industrial.

La figura 1.5 muestra un diagrama de tiempo que ilustra este mecanismo.

La nomenclatura es la siguiente:

M: Mensaje de información.
T: Mensaje de habilitación a la siguiente estación en el anillo (estafeta o *token*).

En t_1 la estación 1 envía su mensaje a la red. Una vez que terminó de enviar ese mensaje, envía la estafeta a la siguiente estación del anillo (en t_2). Cuando la estación 2 recibe la estafeta puede empezar a transmitir su mensaje, que es lo que ocurre en t_3. Lo anterior se repite para todas las estaciones que forman parte del anillo. Una vez que la última estación ha enviado su mensaje, envía la estafeta a la primera estación del anillo (en t_4), con lo que éste se cierra.

Nótese que si una estación no tiene ningún mensaje a enviar simplemente envía la estafeta, tal y como ocurre en t_5.

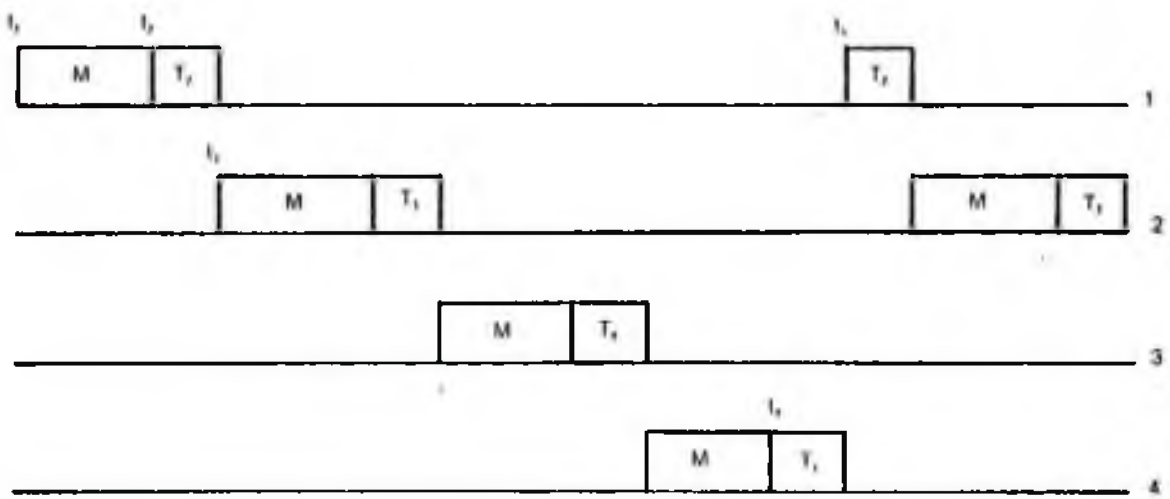

Figura 1.5 Mecanismo de acceso por anillo lógico.

1.5 Justificación del proyecto.

Los protocolos presentados en la sección anterior tienen algunas limitaciones. Algunas de estas limitaciones constituyen serios problemas cuando se desea trabajar en ambientes industriales, en los cuales es importante asegurar el acceso a la red en un tiempo definido. Otras limitaciones se refieren a la eficiencia de los protocolos en determinadas situaciones. A continuación se enlistan las limitaciones de los protocolos mencionados.

CSMA/CD.

— Por el carácter aleatorio de la resolución de las colisiones, este protocolo no garantiza un tiempo máximo de acceso a la red.

— En condiciones de tráfico intenso de mensajes el número de colisiones tiende a aumentar, con lo que la utilización efectiva de la red disminuye dramáticamente.

Protocolos por sondeo:

— Por el carácter centralizado de estos protocolos, su confiabilidad resulta baja, pues una vez que la estación encargada de realizar el sondeo queda fuera de operación, la red completa queda inutilizada.
— La utilización efectiva de la red es baja en condiciones de tráfico ligero.
— Si bien el tiempo máximo de acceso a la red está acotado, también lo está el tiempo mínimo, lo que resulta poco ventajoso en condiciones de tráfico ligero.

Anillos lógicos.

— En caso de pérdida del token, la recuperación del orden en el anillo se convierte en un serio problema.
— La utilización efectiva de la red es baja en condiciones de tráfico ligero.
— Si bien el tiempo máximo de acceso a la red está acotado, también lo está el tiempo mínimo, lo que resulta poco ventajoso en condiciones de tráfico ligero.

Por las razones anteriores, resulta atractivo proponer alternativas a los protocolos establecidos, buscando subsanar algunas de sus limitaciones.
Se plantean las siguientes hipótesis:

i) Un protocolo de detección de portadora con resolución determinística de las colisiones. Los objetivos de este protocolo son asegurar el tiempo máximo de acceso a la red para un protocolo del tipo CSMA y permitir una mejor utilización del ancho de banda de la red.
ii) Un protocolo por sondeo evitando la centralización de la estación habilitadora. Los objetivos de este protocolo son dar confiabilidad a un protocolo por sondeo evitando los complicados mecanismos de recuperación propios de un anillo lógico.

En la siguiente sección se introducen las características operativas de los protocolos propuestos.

I.6 Descripción operativa de los protocolos propuestos.

I.6.1 CSMA/CDR.

CSMA/CDR (por las siglas de *Carrier Sense Multiple Access with Collision Deterministic Resolution*: Acceso múltiple con detección de portadora con resolución determinística de las colisiones) es un esfuerzo por que el tiempo máximo de acceso esté definido en una red con detección de portadora. Existen antecedentes de este tipo de protocolos. Alallah [ATA88] propone el mecanismo básico de operación (del cual hablaremos más adelante), aunque no presenta evaluaciones de la red. Crane *et al* [CRA89] implantaron un esquema en el cual se utiliza un anillo lógico montado sobre una red Ethernet, lo cual evita las colisiones. Por lo tanto, su trabajo no se puede considerar como una implantación de CSMA/CDR (si bien ese no es su objetivo).
El mecanismo básico de operación parte de una idea simple. Mientras no existan colisiones, el protocolo funciona exactamente como CSMA/CD. En el momento que ocurre una colisión, todas las estaciones (incluso aquellas que no intervinieron en ella) entran en una fase de resolución, en la cual la red se comporta como un anillo lógico. De esta manera, se garantiza un tiempo máximo de acceso a la red, que se puede calcular por:

$$t_{max} = t_{wait} + t_{col} + N(t_{tok} + t_{msg})$$

donde:

- t_{wait} tiempo de espera mientras otra estación ocupa la red, que es menor o igual al tiempo en que se transmite la trama más larga posible.
- t_{col} tiempo de detección de la colisión.
- N número de estaciones en la red.
- t_{tok} tiempo de envío del *token*.
- t_{msg} tiempo de envío de la trama más larga posible.

Además de proporcionar determinismo, se puede ver de una manera intuitiva que este mecanismo tiende a autorregularse, de manera que en condiciones de carga ligera en la red se comporte

como el protocolo CSMA/CD, mientras que en condiciones de tráfico intenso tenderá a actuar como un anillo lógico. Dicho en otras palabras, se espera que la utilización efectiva de la red sea buena independientemente de la carga en ella.

La figura 1.6 muestra un diagrama de tiempo en el cual se muestra la operación del protocolo. La nomenclatura para esta figura es la siguiente:

C_s : Detección de actividad en la red.
M : Mensaje de información.
C : Colisión.
J : Aviso a todas las estaciones que existió una colisión.
T : Estafeta de una estación a otra.

En t_1 la estación sensa la actividad en el canal, se da cuenta que está ocupado y se espera a transmitir su mensaje. Lo mismo le ocurre a la estación 4 en t_2. Al terminar de transmitir su mensaje la estación 1, ocurre que las estaciones 2 y 4 detectan libre el canal (en t_3), lo que provoca una colisión. En t_4 envían un mensaje a todas las estaciones para indicarles que existió una colisión y en t_5 todas las estaciones entran en un esquema de anillo lógico. Cuando la última estación ha transmitido su mensaje, envía un aviso a todas las estaciones (en t_6) de que la resolución de las colisiones ha concluido.

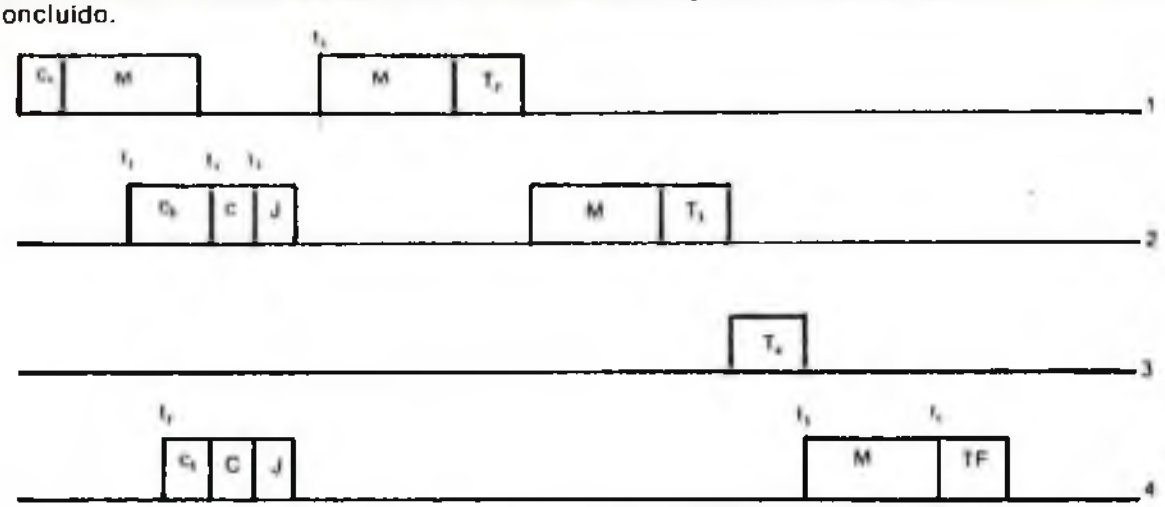

Figura 1.6 Mecanismo de acceso CSMA/CDR.

1.6.2 Sondeo descentralizado.

La naturaleza misma de un protocolo por sondeo es centralizada. El esquema de anillo lógico puede verse como una manera de efectuar sondeo descentralizado (donde el *token* actuaría como la sonda).

El mecanismo aquí propuesto también es simple y se basa en lo siguiente: mientras la estación encargada de hacer *polling* (maestra) funciona adecuadamente, el protocolo funciona exactamente como lo haría un protocolo completamente centralizado. En el momento en que la estación maestra deje de enviar tramas de habilitación (por cualquier causa), la actividad en la red se verá interrumpida. Las estaciones subordinadas podrán percatarse de ésto e inicializarán temporizadores. Una vez que el temporizador alcance su cuenta, entonces la subordinada asumirá que la maestra falló y ella tomará el papel de reclora de la red. Para evitar que dos o más estaciones subordinadas tomen el control de la red al mismo tiempo, el valor del temporizador será función de la dirección de la estación.

La figura 1.7 muestra la forma en que trabaja este protocolo.
La nomenclatura para esta figura es la siguiente:

P : Mensaje de sondeo (*polling*).
M : Mensaje de información.
t_{con}: Tiempo máximo de espera antes de tomar el control de la red.

En t_1 la estación maestra envía un mensaje de sondeo a la estación 3, así que ésta inicia en t_2 la transmisión de un mensaje de información. En t_3 ocurre algo con la estación maestra de manera que

ya no envía más mensajes de sondeo. Así, en t_4 todas las estaciones arrancan sus temporizadores que indican que la red lleva un tiempo desocupada. En t_5 se vence el temporizador correspondiente a la estación 2 (el valor de los temporizadores debe ser diferente para las diferentes estaciones de la red, pues de lo contrario 2 o más estaciones podrían tomar el control de la red simultáneamente), y es ahora esta estación la que se encarga de enviar los mensajes de sondeo al resto de las estaciones.

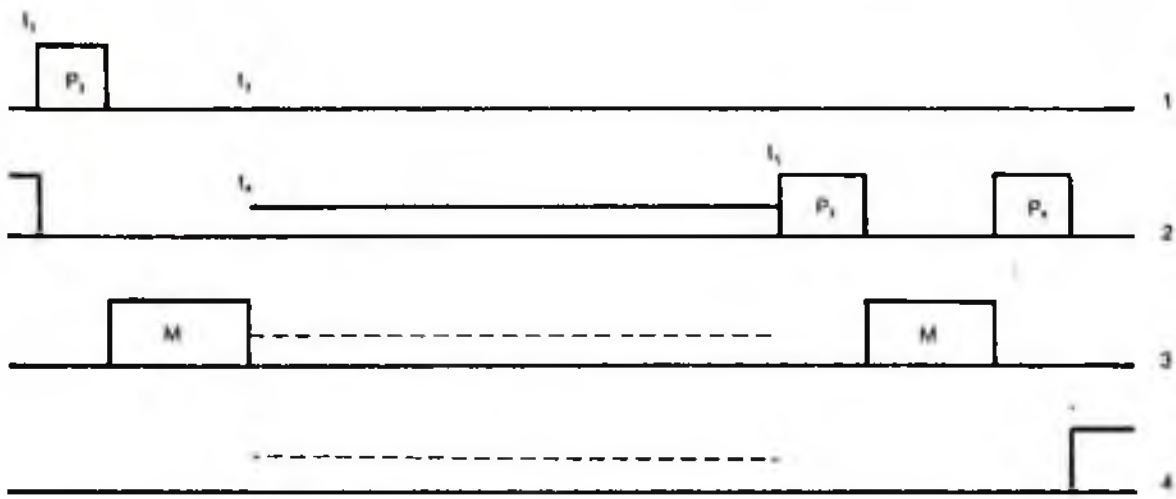

Figura 1.7 Mecanismo de acceso por sondeo descentralizado.

II. Ambiente de Trabajo

II.1 La Red.

Para este trabajo se utiliza un modelo de red (no una red real) que se compone de estaciones de trabajo y un medio de comunicación.

II.1.1 Las estaciones de trabajo.

Las estaciones de trabajo utilizadas son máquinas PC con un *Asynchronous Communication Adapter* (Adaptador para Comunicaciones Asincronas, en adelante referenciado como ACA), que puede operar en modo de lazo de corriente o como un interfaz RS-232C. En cualquier caso, la transmisión-recepción de datos se realiza en forma serie. Estas máquinas vienen normalmente configuradas para trabajar como un interfaz tipo RS-232C.

El interfaz RS-232C define las funciones de las señales necesarias para la transmisión y recepción de datos, las caracteristicas mecánicas de los conectores a utilizarse, y las caracteristicas eléctricas de las señales definidas. Una descripción del interfaz RS-232C se encuentra en [SCH87].

Un diagrama a bloques del ACA (adaptado de [IBM83]) se muestra en la figura 2.1.

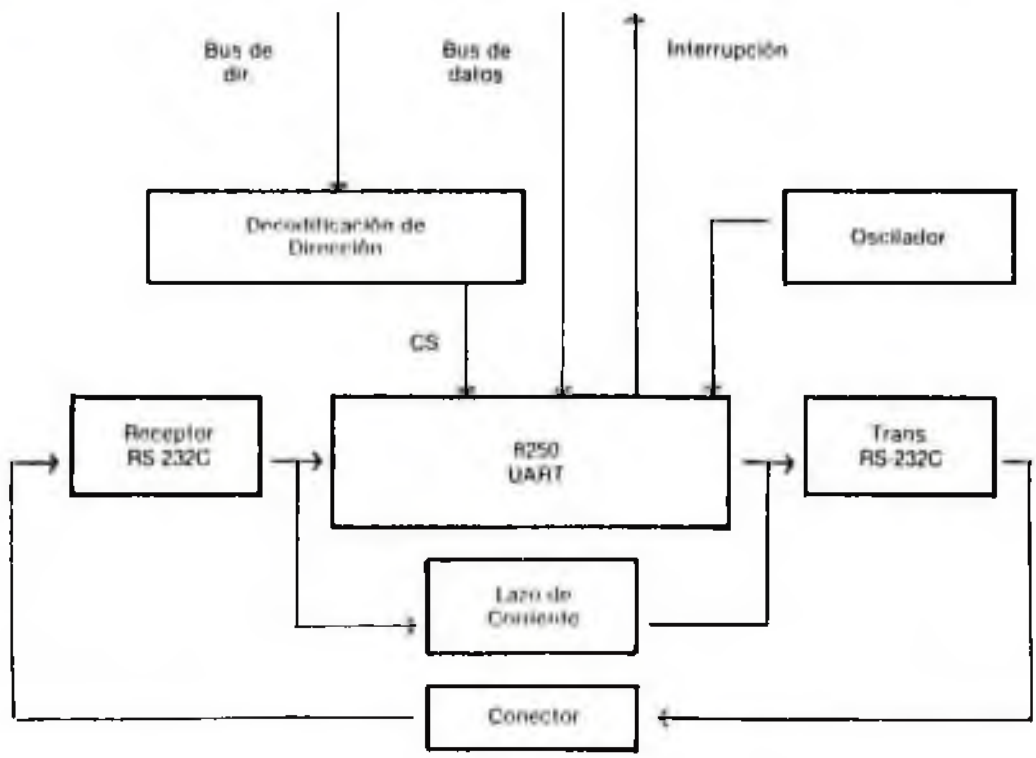

Figura 2.1 Diagrama a bloques del ACA

Los bloques Receptor RS-232C y Transmisor RS-232C tienen como función adaptar los niveles de voltaje manejados internamente por la PC y aquellos definidos en el estándar.

Quien se encarga de recibir/enviar bits en forma serial de/a los manejadores RS-232C es el UART 8250 ([IBM83], [WD]). Este circuito de comunicación tiene algunas características que inciden sobre la implantación realizada. Algunas de ellas son las siguientes:

— La transmisión y recepción se controla por caracteres. Esto tiene 2 implicaciones:

i) La recepción y transmisión de tramas completas no es apoyada por la circuiteria. Es una tarea de la que se debe encargar alguna parte del programa de implantación.
ii) No es posible utilizar la técnica de inserción de bits (*bit stfuling*) para enviar caracteres de fin de trama, lo que quiere decir que no existe transparencia en el envio de datos. En otras palabras, el caracter que se utilice para indicar el fin de trama no deberá ser usado en algún otro campo de la misma.

— La velocidad de la red esta bastante limitada. El estándar RS-232C especifica que la velocidad de transmisión debe ser menor de 20k bps. La velocidad usada en la presente implantación es de 9600 bps, una velocidad bastante común cuando se utiliza este estándar.
— La distancia máxima entre las estaciones debe ser menor de 15 metros.

Si bien estas características no corresponden a las de una red local (con velocidades de transmisión mayores a 1M bps, distancias entre estaciones de hasta varios kilómetros), hay algunas ventajas en usar este tipo de estaciones (al menos para tener funcionando una versión de los protocolos propuestos):

— Son bastante comunes.
— Hay una gran cantidad de *software* de apoyo (compiladores, núcleos de paralelismo, emuladores de terminal, etc.) para la construcción de los protocolos.
— Existe abundante información acerca de las PC's a nivel de circulteria.

Las desventajas son obvias, y tienen que ver de manera muy importante con lo inadecuado que resulta el sistema operativo DOS para soportar este tipo de aplicaciones (lo que obliga al uso de un núcleo de paralelismo).

Queda claro que el uso de tales estaciones sólo puede justificarse desde el punto de vista experimental, y de ninguna manera se recomienda su uso tal cual en una red local (diferente seria el caso en el que se incluyera una tarjeta que apoyara el interfaz de nivel físico para una red local).

II.1.2 El medio de comunicación.

Los protocolos que se proponen presuponen que el medio de comunicación es compartido por todas las estaciones (*broadcast*). Como la comunicación se realiza a través de un cable, nos referiremos a este tipo de estructura de la red como *bus* compartido. La figura 2.2 muestra esta estructura.

Figura 2.2 Estructura tipo *bus* compartido.

El interfaz RS 232C define una conexión para los datos que se transmiten (*pin* 2) y otra para los datos que se reciben (*pin* 3). Como es necesario que el canal sea compartido, es necesario que se utilice un solo cable para la transmisión y la recepción. Para lograr lo anterior se utiliza MILAN (*Modem Interface Local Area Network*), en la cual se conectan las patitas de transmisión y de recepción por medio de un diodo (cuyo cátodo se conecta a la patita de recepción). El uso del diodo evita que señales provenientes de otras estaciones lleguen a la patita de transmisión, donde podrían provocar un corto circuito, pero permite que lo que es transmitido también sea "escuchado" por la misma es-

tación (lo cual es particularmente importante en los protocolos de acceso alealorio, pues la propiedad de "escuchar" aquello que se transmite permite detectar las colisiones). Sólo se necesita un cable extra para conectar la tierra (*pin* 7) de todas las estaciones.

La forma de conexión se muestra en la figura 2.3.

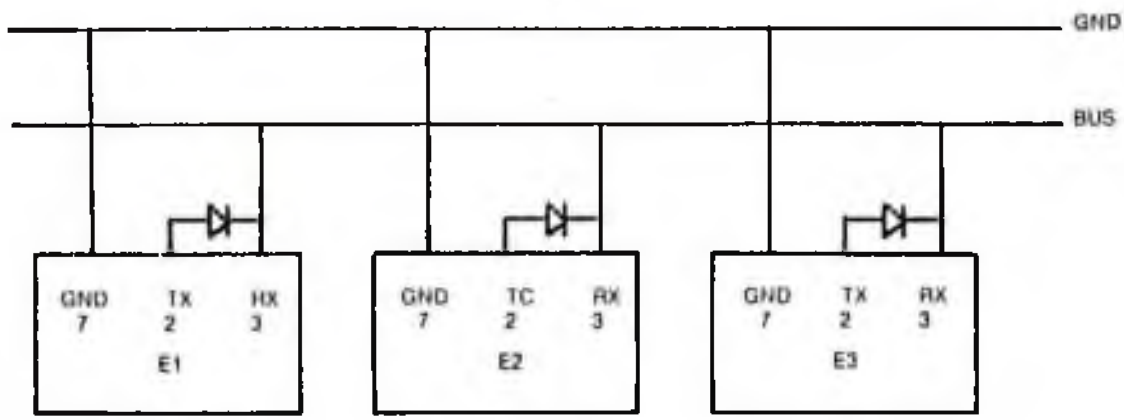

Figura 2.3 Conexiones para una red MILAN de 3 estaciones.

II.2 El núcleo de concurrencia.

El protocolo que se propone requiere manejar varios eventos de manera simultánea (notablemente, la recepción y la transmisión). Son varias las opciones que se pueden tomar cuando se requiere paralelismo (lenguaje, sistema operativo, etc.). Cuando se tiene un sistema operativo que no permite paralelismo (como en este caso), una solución puede ser utilizar un núcleo de concurrencia (o *kernel*), que permite la realización de varias tareas "al mismo tiempo" mediante la creación de entidades independientes llamadas procesos. De esta manera es posible utilizar un lenguaje de programación secuencial apoyado por una libreria para hacer uso de las funciones del núcleo. Esta fue la solución adoptada.

El lenguaje de programación sencuencial utilizado es C.

El núcleo de concurrencia usado fue hecho por el Ing. David Solis Pacheco, estudiante de la Sección de Computación del CIEA. Pudo haberse utilizado el núcleo realizado por el M. en C. Andrés Vega. Si bien el segundo *kernel* es más completo y resultaba más confiable, la decisión de adoptar el primero se debió a su menor tamaño (lo cual permitió que los archivos ejecutables fueran bastante cortos), a su simplicidad, y a que experiencias anteriores con el *kernel* de Vega dieron algunos problemas cuando la velocidad de transmisión era de 9600 bps.

El núcleo utilizado hace cambio de contexto por rebanada de tiempo y también después de llamar a alguna primitiva que implique una posible modificación de la cola de procesos listos.

A continuación se describen las funciones de las primitivas importantes que incluye originalmente el *kenel* usado. También veremos algunas modificaciones y adiciones que fue necesario hacerle al núcleo para que cumpliera con los requerimientos de los protocolos, en particular con respecto al uso de temporizadores. Los puntos marcados con una arroba son adiciones al *kernel* original.

Primitiva:
    initkernel
Declaración:
    procedure initkernel (void);
Función:
    Inicializa colas de procesos.
    Arranca proceso para hacer cambio de contexto por tiempo.
    Arranca proceso ocioso (*iddle*).
    Inicializa cola de temporizadores. (@)

Primitiva:
    crealeprocess

Declaración:
process_rec_ptr createprocess (offset prog, word memreq, char *name);
Función:
Crea un proceso y lo lanza a competir por el procesador.
Entradas:
prog: Apuntador corto al código del proceso.
memreq: Tamaño del *stack* particular del proceso.
name: Apuntador al nombre del proceso (@)
Salidas:
Apuntador a la estructura que define al proceso.

Primitiva:
initsem
Declaración:
semaphore_ptr initsem (int initval);
Función:
Crea un semáforo y le asigna un valor inicial.
Entradas:
initval: Valor inicial.
Salidas:
Apuntador a la estructura que define al semáforo.

Primitiva:
wait
Declaración:
procedure wait (semaphore_ptr sem);
Función:
Si el semáforo tiene un valor mayor que cero, lo decrementa.
En caso contrario:
— Retira el procesador al proceso que invoco a wait.
— Pone al proceso en la cola de los que esperan al semáforo.
— Hace cambio de contexto.
Entradas:
sem: Apuntador al semáforo sobre el que aplica el wait.

Primitiva:
signalns (@)
Declaración:
procedure signalns (semaphore_ptr sem);
Función:
Si no hay procesos esperando en el semáforo, lo incrementa.
En caso contrario:
Saca al primer proceso de la cola del semáforo y lo pone en la cola de procesos listos.
No hace cambio de contexto.
Entradas:
sem: Apuntador al semáforo sobre el que aplica el signalns.
Observaciones:
La razón para evitar el cambio de contexto en esta rutina es evitar la interferencia con los cambios de contexto realizados por el proceso reloj.

Primitiva:
waittime
Declaración:
procedure waittime (word t);
Función:
El proceso que la invoca queda fuera de competencia del procesador t *ticks* de reloj.
Entradas:
t: número de *ticks* que "duerme" el proceso invocante.

Primitiva:
create_timer

Declaración:
timer_ptr create_timer (word count, offset prog, char id);
Función:
Crea un temporizador y lo deja listo para usarse
Entradas:
count: Número de *ticks* que tarda en entrar la rutina que atiende al temporizador a partir de que éste arrancó.
prog: Apuntador a la rutina que atiende al temporizador.
id: Identificación del timer.
Salidas:
Apuntador a la estructura que define al temporizador.

Primitiva:
start_timer
Declaración:
procedure start_timer (timer_ptr timer);
Función:
Pone un temporizador en la cola de temporizadores activos.
Entradas:
timer: Apuntador al temporizador a ser insertado en la cola de temporizadores activos.

Primitiva:
stop_timer
Declaración:
procedure stop_timer (timer_ptr timer).
Función:
Saca a un temporizador de la cola de temporizadores activos, aunque no haya cumplido su cuenta.
Entradas:
timer: Apuntador al temporizador a ser excluido de la cola de temporizadores activos.

III. Implantación de los Protocolos

III.1 Generalidades.

La sección III.1 describe tópicos comunes a la implantación de ambos protocolos.

III.1.1 Definiciones comunes a ambos protocolos.

Se incluyen los siguientes *headers*:

```
#include <stdlib.h>
#include <string.h>
#include "cls.h"
#include "kernel.h"
```

El único que vale la pena comentar es kernel.h. En el capitulo anterior vimos las funciones básicas que proporciona el núcleo de concurrencia utilizado. La forma de llamado a estas funciones se incluye en este header, así como la definición de las estructuras para los procesos, los semáforos y los temporizadores.

Se definen los siguientes tipos:

```
#define byte unsigned char
#define word unsigned int
```

Se definen valores TRUE y FALSE:

```
#define TRUE            1
#define FALSE           0
```

Se definen los siguientes valores, utiles para la programación del circuito 8250 (ver III 1.2):

```
#define LCRW            0x9B    /* LCR word: 8 bits, 1 stop bit, paridad par
#define SPD_L           0x0C    /* Parte baja de factor de velocidad  */
#define P_SPDL          0x3F8
#define SPD_H           0x00    /* Parte alta de factor de velocidad  */
#define P_SPDH          0x3F9   /* 9600 bps  */
#define SET_IO          0x1B    /* habilita I/O  */
#define INT_DR          0x05    /* habilita interrupción  */
#define P_INT3259       0x21    /* puerto del 8259  */
#define P_DATA          0x3F8   /* puerto de Rx/Tx de datos.  */
#define P_IER           0x3F9   /* Interrupt Enable Register  */
#define P_IIR           0x3FA   /* Interrupt Id Register  */
#define P_LCR           0x3FB   /* Line Control Register  */
#define P_MODEM         0x3FC   /* Modem Control Register  */
#define P_LSR           0x3FD   /* Line Status Register  */
#define P_MSR           0x3FE   /* Modem Status Register  */
```

Se define un valor para la bandera de fin de trama (ver III.1.3):

```
#define END_FRAME          0x7A
```

Se definen abreviaciones para:

— Habilitar interrupción por transmisión de caracter (buffer de transmisión vacío).

```
#define TRANSMIT      outportb(P_IER, inportb(P_IER): 0x2)
```

— Deshabilitar interrupción por recepción de caracter.

```
#define NOTRANSMIT      outportb(P_IER, inportb(P_IER)&0x1d)
```

— Probar si la interrupción del 8250 fue por transmisión.

```
#define INTTX status & 0x2
```

— Probar si la interrupción del 8250 fue por recepción.

```
#define INTRX status & 0x4
```

— Probar si la interrupción del 8250 fue por detección de error.

```
#define RECERR (status & 0x6) = = 0x6
```

— Determinar si hay interrupciones del 8250 pendientes de atender.

```
#define PEND status & 0x1
```

Se define tamaño máximo de información a ser transmitido en una trama.

```
#define INFOSIZE          255
```

Se define tamaño del encabezado de la trama.

```
#define HEADSIZE          3
```

III.1.2 Programación del 8250.

El circuito 8250 fue programado para transmitir a una velocidad de 9600 bps, un tamaño de caracter a enviar y recibir de 8 bits, un bit de parada, bit de paridad par, interrupciones por recepción habilitadas. Además de programar al 8250 hay que programar al 8259 para habilitar las interrupciones generadas por el 8250. A continuación se incluye el código de esta rutina. Las constantes aquí usadas ya fueron definidas con anterioridad.

```
/*************************************************************************
init_uart:   Inicializa 8250. Habilita interrupciones por Rx.
*************************************************************************/
procedure init_uart (void)
{
outportb(P_INT8259, inportb(P_INT8250) & 0xEF);
outportb(P_MODEM, inportb(P_MODEM) | 0x08);
outportb(P_LCR, LCRW);
outportb(P_SPDL, SPD_L);                    /*  Velocidad de transmisión: 9600  */
outportb(P_SPDH, SPD_H);
outportb(P_LCR, SET_IO);
outportb(P_IEP, INT_DR);
inportb(P_DATA);                            /*  Limpia registro de datos  */
}
```

III.1.3 Estructura de la trama.

En un principio se planteó la posibilidad de utilizar una estructura de trama similar a la del protocolo HDLC. Pero surgieron algunas dificultades.

— Deseábamos incluir direcciones fuente y destino.
— No contamos con *bit stuffing* para poder generar las banderas de inicio y fin de trama.
— No contamos con ninguna ayuda por hardware para poder generar y verificar el CRC.

El primero de los problemas se soluciona simplemente incluyendo campos para ambas direcciones.

Para el segundo problema, decidimos sacrificar la transparencia de datos y queda definido un fin de trama (0x7A) que no debe ser usado como valor dentro de algún otro campo de la trama o dentro de la información a transmitirse. Aunque esto permitiría tramas de tamaño arbitrario, limitamos el tamaño de la información que puede transmitirse en una sola trama a 255 bytes.

El tercer problema (cálculo por *software* del CRC) se pasó por alto.

Así, la estructura de la trama tiene la forma de la figura 3.1.

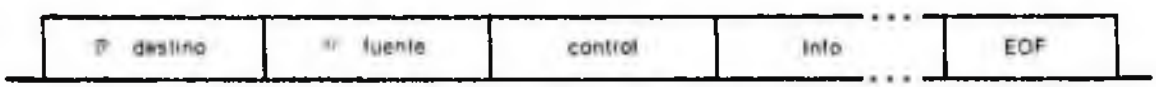

Figura 3.1 Estructura de la trama.

Y la declaración de la estructura tipo trama quedó como sigue:

```
typedef struct frame {

                byte        dest_addr;
                byte        srce_addr;
                byte        control;
                byte        *info;

} frame;
```

El fin de trama será insertado y detectado por las rutinas correspondientes, por lo que no es necesario incluir ese campo dentro de la estructura.

III.1.4 Descripción de los interfaces.

Los interfaces entre los procesos de usuario y los procesos de nivel 2 tienen la misma forma tanto para el protocolo por sondeo descentralizado como para el protocolo CSMA/CDR. La idea es que un mismo programa de usuario pueda correr usando los diferentes protocolos. Precisamente ese es uno de los objetivos de la arquitectura OSI de ISO, hacer independientes los diferentes niveles entre sí.

Los interfaces están dados como se muestra a continuación:

Para la transmisión:

Llamado a la rutina TransmitFrame, que tiene la siguiente forma:

```
TransmitFrame(byte dest, byte source, byte type, byte *data).
```

en donde los parámetros de entrada son:

dest: dirección destino de la trama a enviarse.
source: dirección origen de la trama a enviarse.
type: tipo de trama a enviarse. Actualmente sólo puede ser una trama de información, pero se considera que pueden haber otros tipos de mensajes, como comentaremos posteriormente.
data: apuntador a la cadena de bytes que se desean enviar.

La rutina TransmitFrame en realidad simplemente forma los datos en la trama de envío, despierta al proceso encargado de hacer la transmisión en el nivel 2 de la arquitectura OSI y espera a que la labor de ese proceso concluya. Por esta última causa es que la rutina TransmitFrame es síncrona (de acuerdo con la terminología de Sloman), pues se continúa con la ejecución del proceso de usuario hasta que no se satisface la petición hecha a la rutina.

A continuación se incluye el código para esta rutina. Consúltense las siguientes secciones para una mejor comprensión del lugar de esta rutina en el contexto de cada protocolo en particular.

```
/*****************************************************************************
        TransmitFrame:  Rutina para el envío de información por la red.
*****************************************************************************/

TransmitFrame(byte dest, byte source, byte, type, byte *data)
{
        s->dest_addr = dest;            /*  Forma la trama de envío  */
        s->srce_addr = source;
        s->control = type;
        s->info = data;
        signalns(HstRdySem);            /*  Activa proceso TxN2  */
        wait(ProcMess);                 /*  Espera a que TxN2 termine  */
}
```

Para la recepción:

Llamado a la rutina ReceiveFrame, que tiene la siguiente forma:

```
frame *ReceiveFrame( )
```

La rutina no tiene parámetros de entrada, pero regresa un apuntador a una estructura tipo trama. Este apuntador indica en dónde se encuentra la trama que acaba de ser recibida. Una vez que el usuario conoce tal dirección puede extraer toda la información que necesite de la estructura misma.

El código de ReceiveFrame se incluye a continuación. Simplemente espera a que alguna trama esté lista para el usuario y regresa esa trama. La rutina también es síncrona en el sentido mencionado para TransmitFrame.

```
/*****************************************************************************
        ReceiveFrame:  Se encarga de tomar una trama que corresponda a la estación local.
*****************************************************************************/
frame   *ReceiveFrame ( )
{
    wait(FrmRdy);
    return(r);
}
```

III.1.5 Programas de usuario.

Para poder probar el correcto funcionamiento de los protocolos se crearon algunos procesos de prueba para transmisión y para recepción. El primero de ellos tiene como función generar datos para ser transmitidos. El segundo recibe datos y los pone en pantalla.

Cabe aclarar que los llamados procesos de usuario no son de ninguna manera normas de niveles superiores de la red. En un sentido estricto, son simples programas de prueba, y no satisfacen ninguna necesidad práctica de algún usuario.

Tenemos en ambos protocolos los mismos programas de usuario. En cada caso tenemos dos procesos de usuario: uno para transmisión y otro para recepción. Describiremos primero el proceso para transmisión, llamado UserTx.

Este proceso se sincroniza con main( ) mediante los semáforos EnableHost (que enciende main para despertar a UserTx) y ProcMess2 (que enciende UserTx para despertar a main).

Lo primero que hace UserTx es dejar en un arreglo una serie de número y letras.

A continuación entramos en un ciclo (sin fin) en el cual:

- Esperamos a que main despierte a este proceso.
- Se toma del arreglo uno de sus elementos (escogido en forma aleatoria).

— Se llena el arreglo del usuario para transmitir sus datos con el elemento escogido en forma aleatoria.
— Se llama a la rutina TransmitFrame para que envíe los datos a la red.
— Se despierta al proceso main.

A continuación se incluye el código para este proceso.

```
/*****************************************************************************
UserTx:  Proceso de usuario para transmitir datos. Obtiene un dato en forma aleatoria, pone en pantalla dirección origen, dirección destino y dato a transmitir, y llama a la rutina TransmitFrame para enviarlo a la red
*****************************************************************************/

procedure UserTx( )
{
    byte number;
    byte i;

    strcpy(char_buff, "01234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    while(flag) {
        wait(EnableHost);
        number = random(36);
        for (i = 0; i < MSG_LEN; i++)
            user_buff[i] = char_buff[number];
        if (tx_line > 14) tx_line = 0;
        gotoxy(10, 7 + tx_line++);
        printf("%d  %d  %c", remote_addr, local_addr, user_buff[0]);
        TransmitFrame(remote_addr, local_addr, INFO_MSG, user_buff);
        if (tx_line > 14) tx_line = 0;
        gotoxy(10, 7 + tx_line++);
        printf("Procesado          ");
        signalns(ProcMess2);
    }
}
```

Si bien el proceso main también es un proceso de usuario, las rutinas a las que llama varían para los diferentes protocolos, por lo que será tratado en las secciones correspondientes a la descripción de los procesos para cada uno de los protocolos.

El proceso de usuario para recepción de datos es aún más simple. Este proceso se limita a llamar a ReceiveFrame. Una vez que ReceiveFrame le regresa una trama, llama a ProcRxFrame (rutina encargada de procesar la trama recién recibida), que en este caso se limita a poner en pantalla información acerca de la trama que llegó.

A continuación se incluye el código de UserRx y ProcRxFrame.

```
/*****************************************************************************
ProcRxFrame:  Rutina encargada de procesar una trama recibida por el usuario. Pone dirección destino, fuente, tipo de trama y el primer caracter de la información recibida en pantalla.
*****************************************************************************/

ProcRxFrame(frame *frmptr)
{
    if (rx_line > 14 ) rx_line = 0;
    gotoxy(45, 7 + (rx_line++));
    printf(c"%d %d %d %c",frmptr->dest_addr, frmptr->srce_addr, frmptr->control, *(frmptr->info));
}
```

UserRx: Proceso de usuario para recepción de datos.

```
procedure UserRx(void)
{
    frame *trama;

    while(flag) {
        trama = ReceiveFrame( );
        ProcRxFrame(trama);
    }
}
```

III.2 Sondeo Descentralizado.

III.2.1 Descripción.

En la sección 1.6.2 se incluye una descripción verbal del protocolo por sondeo descentralizado. También se incluye un diagrama de tiempo que muestra la forma en que se espera que se realice el acceso a la red para varias estaciones. La figura 3.2 muestra un diagrama (tipo red de Petri) de como se comportan la transmisión y la recepción en una estación que se atenga a este protocolo.

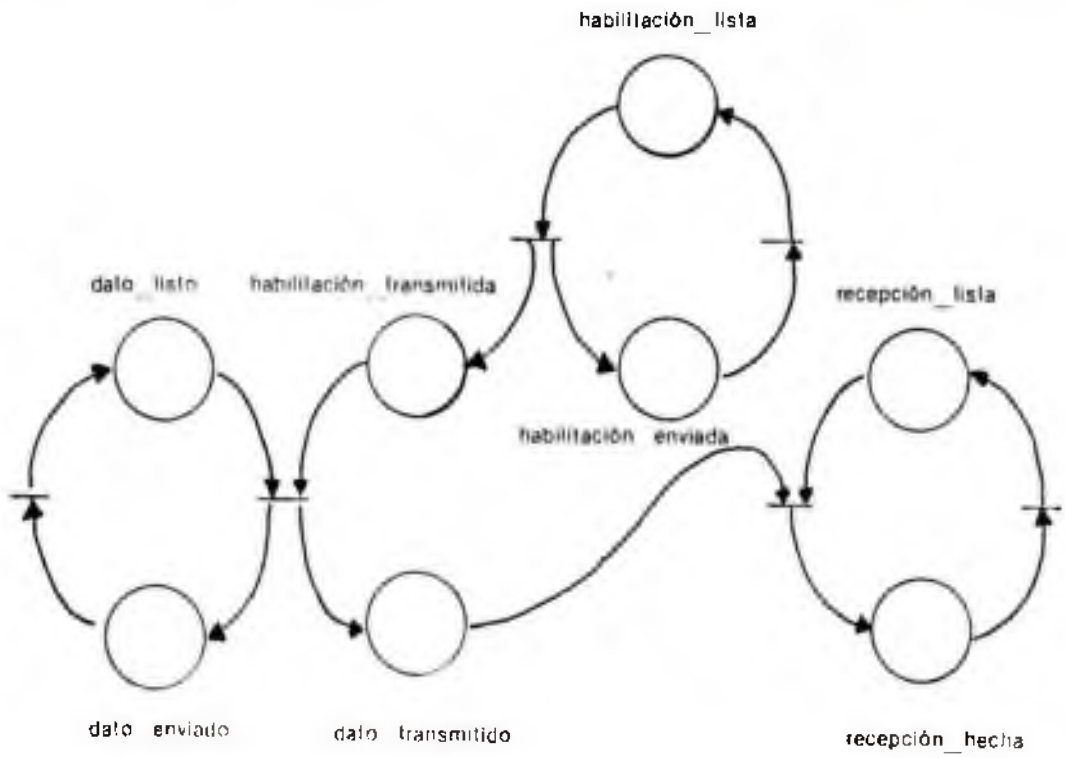

Figura 3.2 Descripción del protocolo por sondeo descentralizado.

Del lado derecho de la figura tenemos la parte que se encarga de la transmisión. En el estado dato_listo hay un dato a ser transmitido por el usuario. El dato no será transmitido hasta que una habilitación sea transmitida (habilitación_transmitida, estado al cual se llega después de estar en habilitación_lista). A continuación ocurre lo siguiente:

i) Los datos llegan a la estación destino (recepción hecha).
ii) La parte transmisora se encuentra en posibilidad de transmitir más datos (dato_enviado).

III.2.2 Estructura de los Procesos.

La figura 3.3 muestra de una manera esquemática cuáles son los procesos necesarios para la implantación del protocolo y la manera en que estos interactúan entre sí.

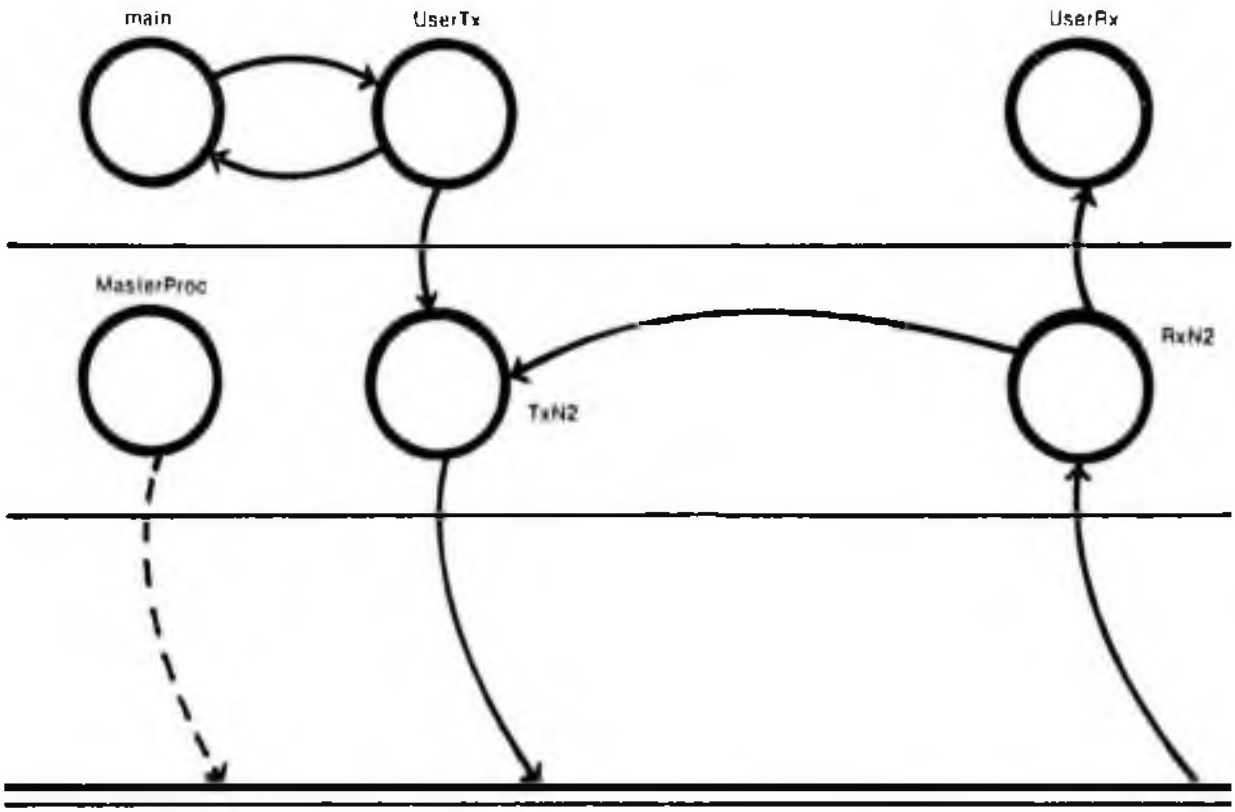

Figura 3.3 Estructura de los procesos para el protocolo por sondeo descentralizado.

Existen 2 procesos correspondientes a funciones del usuario. Uno está encargado de la transmisión (UserTx) y otro de la recepción (UserRx) de datos. Estos procesos no son parte del protocolo, sino que constituyen funciones del usuario (o en todo caso, funciones de niveles superiores de la arquitectura OSI de ISO). Estos procesos son los encargados de proporcionar los datos a transmitir al nivel 2 (en el caso de UserTx) o de procesar los datos recibidos del nivel 2 (en el caso de UserRx).

¿Cómo es que el proceso UseTx genera datos? Eso no es de importancia para el protocolo (aunque en la siguientes sección explicaremos cómo se hizo para nuestra aplicación). El protocolo tampoco tiene que ver con la forma en que UserRx procese los datos recibidos.

Los procesos que tienen que ver con las funciones del protocolo son TxN2 (proceso transmisor del nivel 2), RxN2 (proceso receptor del nivel 2) y MasterProc (proceso encargado de generar las tramas de sondeo).

La descripción de estos tres procesos se hace de manera detallada en la siguiente sección. Por lo pronto, describiremos de una forma global la función de estos procesos y la forma en que interactúan entre sí.

RxN2 recibe tramas completas (el puerto RS-232C recibe bytes, no tramas. Formar tramas completas es función de la rutina que recibe los bytes, como se verá adelante). Esas tramas pueden ser:

— Tramas para otra estación, en cuyo caso se desechan.
— Tramas de información para la estación, en cuyo caso se envían al proceso de usuario.
— Tramas de habilitación para la estación, en cuyo caso se le hace saber al proceso TxN2.

TxN2 recibe datos del proceso UserTx y los envía a la red cuando haya recibido permiso para enviarlos a la red, es decir, hasta que RxN2 detecte una trama de habilitación para la estación.

MasterProc tiene como función hacer el sondeo a cada una de las estaciones que se incluyen en la red para que puedan transmitir. Como es lógico, sólo una de todas las estaciones de la red puede tener esta función. Por lo tanto, en el momento de configurar la red y las estaciones que formen parte de la misma, se debe determinar cuál será la estación maestra. Sólo esa estación tendrá corriendo a este proceso. El resto de las estaciones lo tendrá bloqueado.

MasterProc enviará las tramas de habilitación con una dirección fuente diferente de cualquier otra dirección en la red. Así, MasterProc es un proceso "transparente" aún para la estación en la que se encuentra corriendo. Esta característica es la que permite que MasterProc pueda correr en cualquier estación dada la eventualidad de que la estación configurada inicialmente como Maestra quede fuera de funcionamiento.

III.2.3 Descripción del programa.

III.2.3.1 Definiciones.

Las definiciones exclusivas de este protocolo son las siguientes:
Se definen valores para el tipo de estación. Pueden ser MASTER o SLAVE.

```
#define MASTER          1
#define SLAVE           2
```

Se definen los tipos de mensajes disponibles. La actual implantación sólo utiliza EN_SLAVE e INFO_MSG.

```
#define EN_SLAVE            71    /*  Habilita esclavo para Tx    */
#define INFO_MSG            73    /*  Mensaje de información      */
```

NO_MSG_TO_SEND se utilizaría en caso de que una estación haya recibido un mensaje de habilitación pero no tenga datos a transmitir. Enviar un mensaje para indicar que no hay datos para transmitir no es un acto de mera cortesía, sino que permite agilizar la operación de la red, puesto que entonces la estación maestra podrá habilitar rápidamente a una estación diferente.

```
#define NO_MSG_TO_SEND          72    /*  Avisa que no tiene info a Tx  */
```

Un mensaje tipo REQ_CONTROL sería utilizado por alguna estación subordinada para solicitar el control de la red. Este tipo de mensaje tendrá como destino al proceso maestro, el cual podría rechazar la solicitud (simplemente ignorándola) o bien podría aceptarla (en cuyo caso deberá enviar una trama de tipo ACK_CONTROL a la estación solicitante del control). Este tipo de mecanismo de transferencia de control no está implantado actualmente en el protocolo. Quedarían por definir las situaciones en que una estación maestra acepta o rechaza ceder el control.

```
#define REQ_CONTROL             74  /*  Solicita ser maestro        */
#define ACK_CONTROL             75  /*  Otorga maestría a un esclavo  */
```

Se define la dirección origen de las tramas de habilitación. Esta dirección es independiente de la dirección de la estación en la que se encuentre corriendo el proceso MasterProc.

```
#define MASTER_ADDRESS          125.
```

III.2.3.2 Variables globales.

Se definen las siguientes variables:

local_addr: tipo byte, es utilizada para guardar la dirección de la estación. Se le asigna un valor durante la inicialización, y ya no es modificada posteriormente.

remote_addr: tipo byte, se usa para indicar la dirección de la estación a la que se le envían las tramas. Como no tiene que ver con el nivel 2 de la arquitectura OSI, eventualmente puede ser modificada por los programas de usuario.

MSG_LEN: tipo byte, se usa para indicar la longitud del mensaje a ser transmitido.

net_stat: tipo byte, indica el status de una estación (MASTER o SLAVE).

destino: tipo byte, es una variable que va modificando MasterProc y que indica la dirección de la estación a la cual se le enviará a continuación una trama de habilitación.

flag: tipo int, variable que al hacerse falsa, termina todos los procesos.

DataReady: tipo byte, variable que impide que se acepte una trama de habilitación si no hay datos listos. En otras palabras, si DataReady es FALSE, una trama de habilitación que llegue es desechada.

rxframe, r, txframe, s: estructuras tipo trama (rxframe y txframe) y apuntadores a estructuras tipo trama (r y s). rxframe y txframe asignan espacio físico de memoria para las tramas de recepción y transmisión; r y s son apuntadores a la localidad de memoria en donde inician las estructuras anteriores.

char_buff[40]: arreglo de 40 bytes que tiene diferentes caracteres de los cuales se escogerá uno (de manera aleatoria) para que sea transmitida una trama que incluya como datos únicamente el caracter escogido.

user_buff[INFOSIZE]: arreglo de 255 bytes que almacena los datos que el usuario desea enviar.

buff_out: buffer de salida (arreglo del tamaño máximo de una trama) para enviar tramas. De este buffer son tomados directamente los datos (por la rutina de atención a interrupciones por transmisión) para ser mandados al canal de comunicación.

buff_out_index: tipo int, índice del buffer de salida.

buff_in: buffer para almacenar los caracteres recibidos por el puerto RS-232C, hasta completar una trama. Del mismo tamaño que buff_out.

buff_in_index: tipo int, índice del buffer de entrada.

tx_line, rx_line: tipo int, número de línea en pantalla en donde aparecerá el siguiente mensaje de impresión.

III.2.3.3 Semáforos.

Los semáforos que se utilizan en este protocolo son los siguientes:

Nombre	Valor Inicial	Función
HstRdySem	0	Indica que el usuario tiene listo un dato para ser transmitido.
FrmArrSem	0	Una trama completa está lista para el proceso de recepción de nivel 2.
FrmRdy	0	Hay una trama para el usuario.
EnableHost y ProcMess2	0 0	El programa main en realidad también es un proceso. Se sincroniza con UserTx mediante estos 2 semáforos.
EnablerFrame	0	Se recibió una trama de habilitación.
ProcMess	0	Ha terminado de ejecutarse un llamado a TransmitFrame.
SendEnable	0 ó 1	Controla la entrada al proceso MasterProc.
FreeChan	1	Impide que MasterProc y TxN2 envíen datos al mismo tiempo. Protege al *buffer* de salida.

Obsérvese que el semáforo SendEnable puede tener como valor inicial 0 ó 1. El valor que tenga dependerá de si la estación es SLAVE (0) ó MASTER (1).

III.2.3.4 Temporizadores.

Para este protocolo son necesarios dos temporizadores.

El primero de ellos se llama NoReq y es utilizada por el proceso MasterProc para continuar con su rol de sondeo cuando una estación no envió nada a la red después de recibir una trama de habilitación. Si después de que se envió el mensaje de sondeo a una estación transcurrieron 15 ticks de reloj, el temporizador se cumple y llama a la rutina timeNoReq, que desbloquea al proceso MasterProc. La declaración de este temporizador se encuentra a continuación, así como el código de la rutina que se llama al vencimiento del mismo.

declaración:

```
NoReq = create_timer(15, (offset)timeNoReq, '1');
```

código de timeNoReq:

```
void timeNoReq( )  /*  No contestó la estación a la que se habilitó  */
{
        signalns(SendEnable);  /*  Desbloquea a MasterProc  */
}
```

El segundo de los temporizadores es usado por todas las estaciones subordinadas para detectar el tiempo de inactividad en la red, y recibe el nombre de TakeControl. Este temporizador es detenido cada vez que se recibe un *byte* de la red y arrancado inmediatamente después. De esta manera, si se cumple el temporizador quiere decir que una estación detectó inactividad en la red por un periodo excesivamente prolongado, por lo que asume que la estación que tiene corriendo a MasterProc tuvo algún problema, por lo que puede tomar el control de la red. Para evitar que 2 o más estaciones tomen el control de la red al mismo tiempo (lo que sería un error), se hace que este temporizador dependa de la dirección de la estación.

A continuación se incluye la declaración de este temporizador y la rutina que lo atiende.

declaración:

```
TakeControl = create_timer(500 + (local_addr * 2),
                           (offset)timeTakeControl, '2');
```

código de timeTakeControl:

```
void timeTakeControl( )  /*  La red se cayó. Tomar control de la red  */
{
   buff_in_index = 0;          /*  desecha tramas incompletas  */
   signalns(SendEnable);       /*  Arranca MasterProc  */
}
```

III.2.3.5 Procesos.

Se tienen 5 procesos, como se ve en la figura 3.3. Estos procesos son:

UserTx, proceso encargado de la transmisión de datos del usuario.

UserRx, proceso encargado de la recepción de datos por parte del usuario.

TxN2, proceso encargado de la transmisión de datos en el nivel de enlace de datos

RxN2, proceso encargado de la recepción de tramas en el nivel de enlace de datos

MasterProc, proceso encargado de generar las tramas de habilitación para todos las estaciones

Además la rutina main funciona como un proceso de usuario que está sincronizándose con el proceso UserTx, como ya se explicó en la sección III.1.5.

Los procesos UserTx y UserRx ya han sido explicados en la sección III.1.5, por lo que nos concentraremos en la descripción de main, TxN2, RxN2 y MasterProc.

III.2.3.5.1 Proceso main.

El proceso main tiene las siguientes funciones:

— Asigna a los apuntadores r y s las direcciones de las tramas rxframe y txframe, respectivamente.

— Llama a la función OpenStation, que se encarga de inicializar el núcleo de concurrencia, crear al resto de los procesos, crear e inicializar semáforos (excepto SendEnable, cuyo valor se determina al decidir si la estación es MASTER o SLAVE, y ProcMess2, que es un semáforo de uso exclusivo del usuario), crear los temporizadores, inicializar 8250, salvar vectores de interrupción (interrupción del reloj, interrupciones del 8250 e interrupción por control break para finalizar programas) y asignar los nuevos vectores de interrupción (el nuevo vector de interrupción para el reloj es asignado por initkernel( )).

— Crea e inicializa el semáforo ProcMess2.

— Pone en pantalla una ventana para imprimir mensajes de transmisión y de recepción (presentacion( )).

— Llama a init_station( ) para dar los valores de la dirección local, de la dirección de la estación destino de los mensajes, la longitud de los mensajes a ser enviados y el status de la estación (MASTER o SLAVE). En este último caso se define el valor para el semáforo SendEnable.

— Entra en un ciclo infinito en el cual despierta al proceso UserTx (haciendo signalns sobre el semáforo EnableHost) y se bloquea hasta que tal proceso termine de enviar un dato (haciendo wait sobre el semáforo ProcMass2).

A continuación se incluye el código para las rutinas main( ), init_station( ) y OpenStation( ).

```
/*****************************************************************************
OpenStation:  Crea procesos, semáforos y temporizadores a ser usados por la estación. Salva y
              asigna vectores de interrupción.
*****************************************************************************/


procedure OpenStation( )
{
    time_int = getvect(0x08);
    initkernel ( );
        /* Crea procesos */
    p0 = createprocess ((offset) TxN2, 1024, "TxN2");
    p1 = createprocess ((offset) RxN2, 1024, "RxN2");
    p3 = createprocess ((offset) MasterProc, 1024, "Master");
    p4 = createprocess ((offset) UserRx, 1024, "UserRx");
        /* Crea e inicializa semáforos */
    HstRdySem    = initsem(0);
    EnableHost   = initsem(0);
    EnablerFrame = initsem(0);
    ProcMess     = initsem(0);
    FreeChan     = initsem(1);
    FrmArrSem    = initsem(0);
    FrmRdy       = initsem(0);
        /* Crea temporizadores */
    NoReq = create_timer(15, (offset)timeNoReq, '1');
    TakeControl = create_timer(500 + local_addr, (offset)timeTakeControl, '2');

    init_uart ( );          /* Inicializa 8250 */
        /* Salva y asigna vectores de interrupción */
    ctrl_break = getvect (0x1B);
    setvect (0x0C, int_UART);
    if (net_stat == SLAVE) start_timer(TakeControl);
}
```

```
/*****************************************************************************
                    init_station:  Inicializa una estación.
*****************************************************************************/

init_station(byte *l_addr, byte *r_addr, byte *len, byte *stat)
{
    char c;

    gotoxy(5,23);
    printf("Dar la dirección de esta máquina:   ");
    *l_addr = getint( );
    gotoxy(5,23);
    printf("Dar dirección de estación remota:   ");
    *r_addr = getint( );
    gotoxy(5,23);
    printf("                                    ");
    gotoxy(5,23);
    printf("Longitud del Mensaje:   ");
   *len = getint( );
    gotoxy(5,23);
    printf("Esta máquina será MASTER o SLAVE (M, S)?   ");
    c = getch( );
    printf("%c", c);
    if (c == 'M' | c == 'm')  {
        *stat = MASTER;
        SendEnable = initsem(1);
    }
    else {
    *stat = SLAVE;
    SendEnable = initsem(0);
    }
}


/*****************************************************************************
main:  Programa principal que se convierte en un proceso de usuario que inicializa una estación y
       se sincroniza con UserTx.
*****************************************************************************/

main( )
{

    r = &rxframe;
    s = &txframe;
    clrscr( );
    OpenStation( );
    ProcMess2 = initsem(0);
    presentación( );
    init_station(&local_addr, &remote_addr, &MSG_LEN, &net_stat);
    gotoxy(5,23);
    printf("Local: %d   Remota: %d   Lenght: %d   Status: %d", local_addr, remote_addr,
        MSG_LEN, net_stat);
    getch( );
    while(flag)  {
        signalns(EnableHost);
        wait(ProcMess2);
    }
}
```

III.2.3.5.2 Proceso TxN2.

El proceso TxN2 se queda en un ciclo infinito durante el cual realiza lo siguiente:

— Se queda bloqueado esperando que lo despierte la rutina TransmitFrame, lo cual quiere decir que hay datos en espera de ser transmitidos.
— Indica que un dato está listo para ser transmitido (DataReady), y que se está esperando una trama de habilitación.
— Una vez recibida la trama de habilitación, apaga la bandera de dato listo a ser transmitido.
— Espera a que el canal de transmisión esté libre (en el caso de que en esta misma estación esté corriendo el proceso MasterProc).
— Forma los datos de la trama en el buffer de salida (llamando a la función FormFrame).
— Envía la trama a la red llamando a la función SendFrame.
— Desocupa el canal de transmisión.
— Indica a la rutina TransmitFrame que el dato que llegó ya fue transmitido, desbloqueándola.

A continuación se muestra el código del proceso TxN2:

```
/*************************************************************************
TxN2:  Proceso encargado de transmitir los datos recibidos del usuario.
*************************************************************************/

procedure TxN2( )
{
    while(flag) {
        wait(HstRdySem);            /*  Espera a tener un dato a enviar  */
        DataReady = TRUE;
        wait(EnableFrame);          /*  Espera a recibir trama de habilitación  */
        DataReady = FALSE;
        wait(FreeChan);
        FormFrame(s-> dest_addr, s->srce_addr, s->control, s->info, MSG_LEN)
        SendFrame( );               /*  Envía la trama a la red  */
        signalns(FreeChan);
        signalns(ProcMess);
    }
}
```

La rutina FormFrame forma la trama de transmisión en el buffer de salida. Recibe como parámetros la dirección origen del mensaje, la dirección destino, el tipo de mensaje, un apuntador al lugar donde se encuentran los datos a ser transmitidos, y la longitud del mensaje.

El código de esta rutina es el siguiente:

```
/*************************************************************************
                        FormFrame:  Rutina para formación de la trama.
*************************************************************************/

FormFrame(byte dest, byte source, byte type, byte *data, byte len)
{
    byte i = HEADSIZE;

    buff_out[0] = dest;
    buff_out[1] = source;
    buff_out[2] = type;
    while (i < len)
        buff_out[i++] = *data++;
    buff_out[i++] = '\0';
    buff_out[i] = END_FRAME;
}
```

La rutina SendFrame simplemente pone el índice del buffer de salida en cero, espera a que el registro de transmisión de datos esté vacío y habilita la interrupción por transmisión (registro de transmisión de datos vacío). La rutina de atención a las interrupciones del 8250 se encarga (en la parte de atención a las interrupciones por transmisión) de vaciar el buffer de salida por el canal de comunicación (ver sección III.2.3.5.5).

El código de la rutina SendFrame se muestra a continuación:

```
/*****************************************************************************
                    SendFrame:  Envía una trama por el puerto serie.
*****************************************************************************/

void SendFrame(void)
{
    buff_out_index = 0;
    while((inportb(P_LSR) & 0x40) != 0x40);
    TRANSMIT;
}
```

III.2.3.5.3 Proceso RxN2.

El proceso Rxn2 es un ciclo infinito en el cual se realiza lo siguiente:

— Se queda bloqueado esperando que una trama completa llegue. La rutina de atención a las interrupciones del 8250 en la parte que atiende a las interrupciones por recepción se encarga de ir formando la trama recibida y de hacer un singnals al semáforo FrmArrSem una vez que la trama está completa. Ver sección III.2.3.5.5.

— Llama a la rutina getframe, para poner en una estructura tipo trama (apuntada por r) la trama recibida.

— Revisa si la trama es para esta estación. Si es así, revisa si se trata de una trama de habilitación o de una trama de información. En el primer caso, se lo hace saber a TxN2 (revisando si hay datos listos a ser transmitidos) haciendo un signalns sobre el semáforo EnablerFrame. En el segundo caso, se lo hace saber al proceso UserRx (rutina ReceiveFrame) haciendo un signal sobre el semáforo FrmRdy.

— Si la estación tiene corriendo al proceso MasterProc, y la trama recibida no es una trama de habilitación, entonces quiere decir que la trama de habilitación si fue contestada, en cuyo caso el temporizador NoReq se detiene y además se desbloquea al proceso MasterProc (haciendo un signal sobre el semáforo SendEnable) para que continúe enviando tramas de habilitación.

Los códigos del proceso RxN2 y de la rutina get frame se muestran a continuación:

```
/*****************************************************************************
RxN2:  Proceso que recibe tramas completas y envia la información correspondiente al proceso de
       recepción del usuario.
*****************************************************************************/

procedure RxN2( )
{
    while(flag) {
        wait (FrmArrSem);
        get_frame(r);
        gotoxy(1, 1); printf("T");
        if(r->dest_addr == local_addr) {
        if(r->srce_addr != MASTER_ADDRESS)
            signalns(FrmRdy);
        if(r->control == EN_SLAVE && DataReady)
            signalns(EnablerFrame);
        }
        if(net_stat == MASTER && r->srce_addr != MASTER_ADDRESS) {
            stop_timer(NoReq);
            signalns(SendEnable);
        }
    }
}
```

```
/*****************************************************************************
        get_frame: rutina encargada de obtener la trama recién obtenida.
*****************************************************************************/

get_frame(frameptr)
frame *frameptr;
{

        frameptr->dest_addr = buff_in[0];
        frameptr->srce_addr = buff_in[1];
        frameptr->control = buff_in[2];
        strcpy(frameptr->info, &buff_in[3]);
}
```

III.2.3.5.4 Proceso MasterProc.

El proceso MasterProc sólo se encuentra corriendo en una de las estaciones de la red. Es un ciclo infinito durante el cual se realiza lo siguiente:

— Se queda bloqueado esperando que el semáforo SendEnable se encienda. Esto ocurrirá si:

i) La estación es asignada como MASTER (al inicio).
ii) La estación tiene el status de MASTER y se recibió una trama que contesta a una trama de sondeo.
iii) La estación tiene el status de MASTER y no se recibió una trama de respuesta a un sondeo y el temporizador NoReq se venció.
iv) La estación no es MASTER y no se detecto actividad en la red por un tiempo mayor al de duración del temporizador TakeControl.

— Determina la siguiente estación a la que se enviará una trama de sondeo (modificando la variable destino).
— Espera a que el canal de transmisión esté libre y lo toma.
— Forma la trama de habilitación a ser enviada en el *buffer* de salida.
— Envía la trama de habilitación.
— Libera el canal de transmisión.
— Arranca el temporizador de espera de respuesta a una trama de habilitación (NoReq).

El código correspondiente a MasterProc se muestra a continuación:

```
/*****************************************************************************
MasterProc:  Proceso encargado de generar las tramas de habilitación para las estaciones subordinadas.
*****************************************************************************/

procedure MasterProc( )
{
    while(flag) {
        wait(SendEnable);
        if (destino++ = MAXADDR)
            destino = MINADDR;
        else
            destino += STEPADDR;
        wait(FreeChan);
        FormFrame(destino, MASTER_ADDRESS, EN_SLAVE, "", 5);
        SendFrame( );
        signians(FreeChan);
        start_time(NoReq);
    }
}
```

III.2.3.5.5 Rutina de atención a las interrupciones del 8250.

Esta rutina atiende los casos en los que se tiene interrupción por transmisión e interrupción por recepción. Leyendo la palabra del IIR (*Interrupt Identification Register*) se puede determinar de qué tipo de interrupción se trató.

En el caso de las interrupciones por recepción se realiza lo siguiente:

— Detiene el temporizador que indica inactividad en la red.
— Llama a la rutina form_frame para guardar el dato recibido en el *buffer* de entrada
— Si la estación no es MÁSTER, arranca el temporizador para detectar inactividad en la red.

En el caso de las interrupciones por transmisión, se realiza lo siguiente:

— Si el *byte* que se sacó la vez anterior es fin de trama, suspende la transmisión. En caso contrario, toma el siguiente *byte* del *buffer* de salida y lo envía a la red.

El código de esta rutina se muestra a continuación:

```
/*************************************************************************
intUART:   Rutina de atención a interrupciones por transmisión y recepción.
*************************************************************************/
void interrupt int_UART ( )
{
        int status;

        status = inportb(P_IIR);
        if (INTRX) {                            /*  Recepción  */
            stop_timer(TakeControl);
            form_frame( );
            if (net_stat == SLAVE) start_timer(TakeControl);
        }
        else
            if (INTTX) {                        /*  Transmisión  */
                if (buff_outbuff_out_index-1] == END_FRAME)
                    NOTRANSMIT;
                else
                    outportb(P_DATA, buff_out[buff_out_index++]);
            }
        outportb(0x20, 0x20);
}
```

La rutina form_frame hace lo siguiente:

— Toma el dato recibido.
— Si el dato no es fin de trama, lo pone en el *buffer* de entrada.
— Si el dato fue fin de trama, pone el índice del *buffer* de entrada nuevamente en cero y enciende el semáforo FrmArrSem.

El código de esta rutina se incluye a continuación:

```
/*************************************************************************
        form_frame:   Forma los caracteres recibidos en la trama de entrada.
*************************************************************************/
form_frame( )
{
    byte dato_rx;

    dato_rx = inportb(P_DATA);
    if (dato_rx != END_FRAME)
        buff_in[buff_in_index++] = dato_rx;
    else{
        buff_in_index = 0;
        signalna(FrmArrSem);
    }
{
```

III.3 CSMA/CDR.

III.3.1 Descripción.

Ya hemos dado una breve descripción verbal del protocolo CSMA/CDR en la sección I.6.1. También hemos mostrado un diagrama de tiempo en el que se puede apreciar el mecanismo básico de operación de este protocolo. Ahora pasaremos a describir con la ayuda de un diagrama (en forma análoga a como lo hicimos en la sección III.2.1) la forma en la que se procede para la transmisión y la recepción del nivel de enlace en este protocolo.

La figura 3.4 muestra el diagrama con el cual nos auxiliaremos para hacer nuestra descripción.

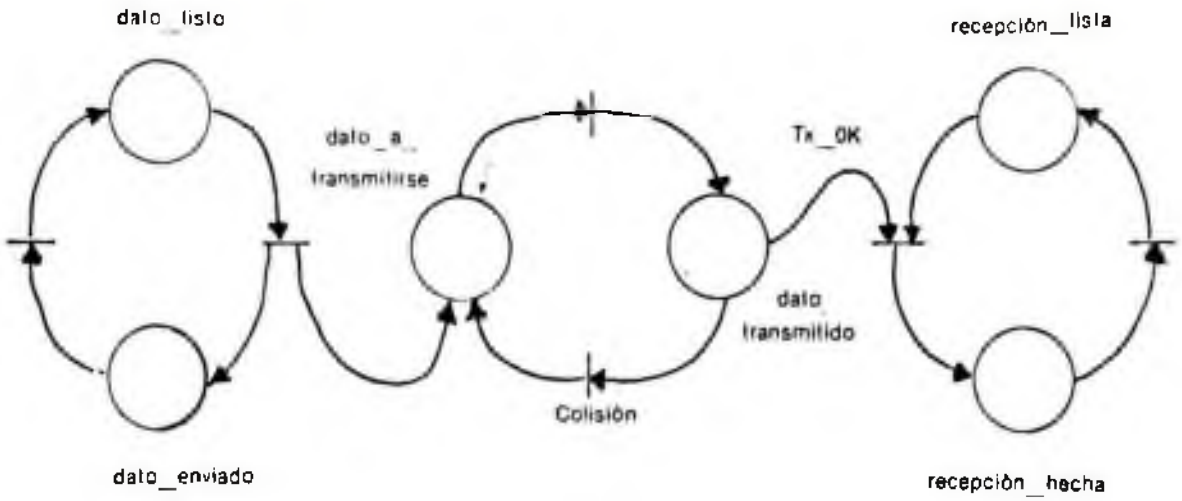

Figura 3.4 Descripción del protocolo CSMA/CDR.

En el estado dato_listo nos encontramos preparados para intentar el envio de un dato a la red. El estado recepción_lista indica que se puede recibir algún dato de la red. Cuando el dato está listo pasaremos a los estados dato_enviado y dato_a_transmitirse. El primero implica que estamos en condición de intentar la transmisión de otro dato. Del segundo podemos pasar a dato_transmitido, cuyas transiciones pueden ser una colisión o una transmisión correcta (tx0K). En el caso de una colisión se intentará de nuevo la transmisión. En el caso de una transmisión correcta se cumplirá una recepción.

La forma como se reintenta la transmisión una vez que ocurre una colisión no se muestra en el anterior diagrama. Pero en la figura 3.5 se muestra un autómata que muestra la operación de la máquina transmisora al presentarse una colisión.

En este caso se consideran acciones diferentes para diferentes tipos de estaciones. Una estación podrá ser inicial si es la primera en el anillo lógico que se forma para resolver las colisiones. Final si es la última estación del anillo. Cualquier otra estación será intermedia.

Una estación inicial podrá enviar sus datos (si es que tiene) y posteriormente transferirá la estafeta (*token*) a la siguiente estación que le sigue en el anillo. Después de esto quedará bloqueada hasta que no reciba una trama de fin de resolución de la colisión.

Una estación intermedia esperará para enviar sus datos hasta que reciba la estafeta de parte de la estación que le antecede en el anillo lógico. Una vez recibida la estafeta podrá enviar una trama con los datos a transmitir (si los hay), y posteriormente deberá enviar la estafeta a la estación que le sigue en el anillo. Por ultimo, quedará bloqueada hasta que no reciba la trama de fin de resolución de la colisión.

Una estación final esperará a recibir la estafeta, podrá enviar sus datos y finalmente deberá enviar una trama de fin de resolución de la colisión.

Aunque este fue el esquema adoptado en la implantación de los protocolos, presenta la grave desventaja de que una falla en cualquier estación que forme el anillo hará fallar la red. Un esquema semejante sería manejar la resolución de las colisiones evitando el envio de estafetas. En su lugar se utilizarían temporizadores (con valores diferentes para cada estación). Así, cada estación enviaría sus datos únicamente al vencimiento de su temporizador. La idea es reservar slots de tiempo para

cada estación exclusivamente durante la resolución de las colisiones. En este caso sería necesario incluir otro temporizador (que tuviera como cuenta el tiempo máximo de resolución de las colisiones) para finalizar la resolución de las colisiones

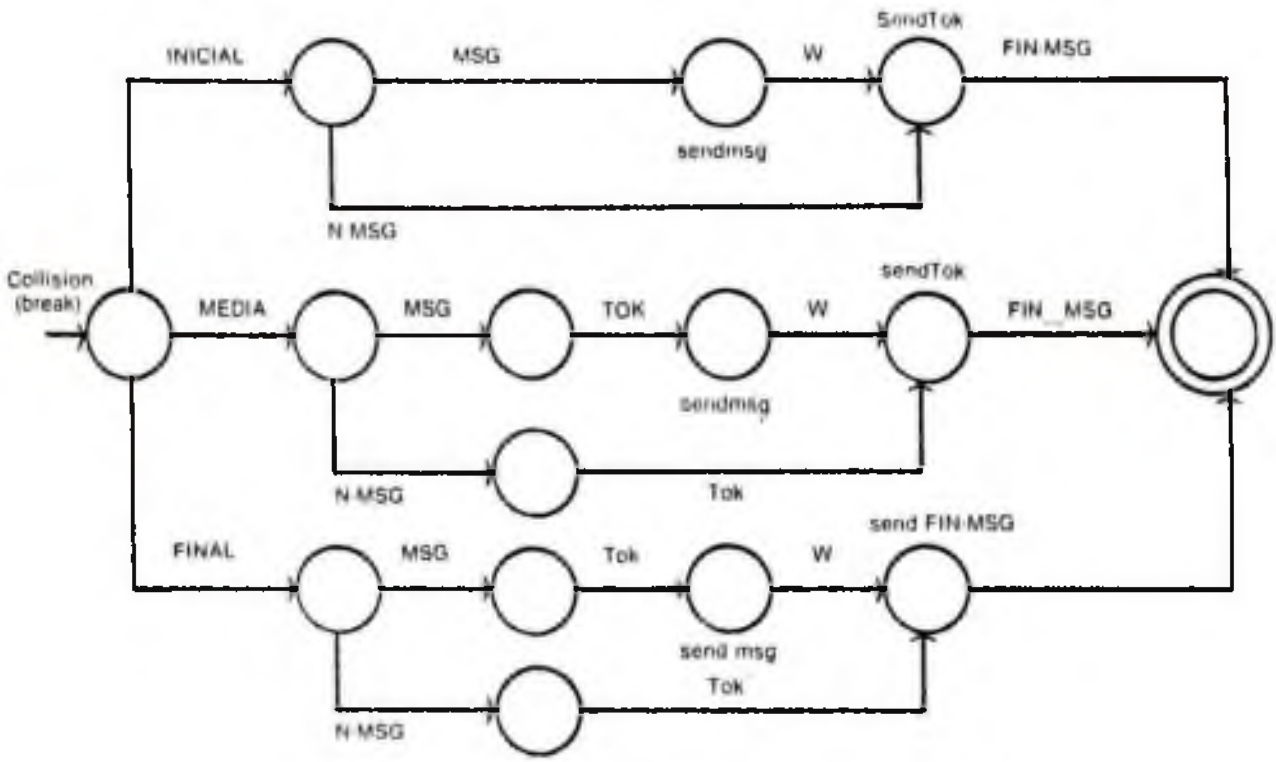

Figura 3.5 Resolución de las colisiones.

III.3.2 Estructura de los procesos.

La figura 3.6 muestra un diagrama de la estructura de los procesos que intervienen para la implantación de este protocolo.

El proceso main y UserTx se sincronizan, y corresponden a la parte de la transmisión en los niveles superiores de la arquitectura OSI.

UserRx es el proceso correspondiente a la recepción para los niveles superiores.

El proceso TxN2 recibe datos a enviar de parte del proceso de transmisión del usuario, y se encarga de llevarlos a la red.

El proceso RxN2 recibe tramas completas que lleva al proceso UserTx. Sin embargo, en el nivel 1 puede detectarse una colisión, que activará al proceso Contention. Si eso ocurre, RxN2 no sólo enviará tramas recibidas al usuario, sino que también enviará las tramas correspondientes a las estafetas al proceso Contention.

Una descripción detallada de los procesos anteriores se encuentra en las siguientes secciones.

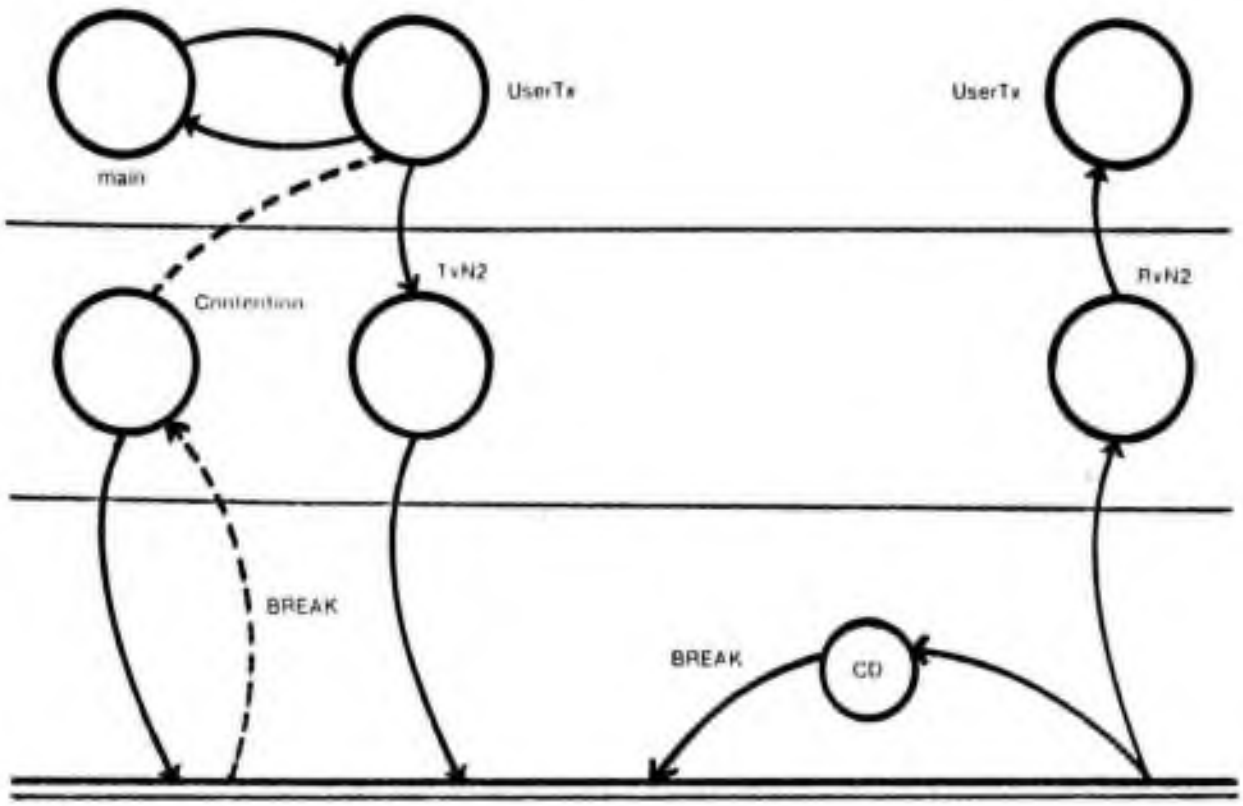

Figura 3.6 Estructura de los procesos para el protocolo CSMA/CDR.

III.3.3 Descripción del programa.

III.3.3.1 Definiciones.

Las definiciones exclusivas de este protocolo tienen que ver con el manejo de una señal de *break* en la línea de transmisión. La señal de *break* es utilizada para advertir a todas las estaciones (y no solo a las directamente involucradas) que hubo una colisión. El uso de esta señal de *break* es análogo a la trama de *jamming* que se usa en Ethernet para indicar la ocurrencia de una colisión. La señal de *break* se envía por la misma línea de transmisión (es decir, no requiere un cable separado para enviarse).

En la implantación actual es conveniente usar la señal de *break* porque el 8250 puede ser interrumpido por una señal de este tipo. La atención a esta interrupción pone en marcha el mecanismo de resolución de las colisiones en todas las estaciones.

Antes de continuar, expliquemos brevemente la forma en que se detectan las colisiones y cómo es que la señal de *break* auxilia en esta situación.

Cuando una estación está transmitiendo, al mismo tiempo "escucha" lo que ocurre en la red (es decir, recibe lo que envía). Si la estación detecta que lo recibido no coincide con lo transmitido, evidentemente ha ocurrido una colisión. Toda estación involucrada en una colisión detectara la misma de esta manera. Pero ¿de qué manera se enteran las demás estaciones que hubo una colisión? (esto es necesario porque en la resolución de las colisiones se involucran todas las estaciones).

Una señal de *break* consiste en tener en un cero lógico la línea de transmisión por un tiempo mayor al necesario para transmitir un *byte* (incluyendo sus *bits* de inicio, de parada y de paridad). La señal de *break* se puede detectar fácilmente, pues genera una interrupción al recibirse.

Las estaciones involucradas en la colisión abortan su actual transmisión y generan una señal de *break*, que detectarán todas las estaciones. La recepción de la señal de *break* activará al proceso encargado de resolver las colisiones.

Las definiciones para el *break* son:

— Para detección de error (la detección de la señal de *break* es considerada como un error).

```
#define RECERR      (status & 0x06) = = 0x06
```

— Para detección de la interrupción por *break*.

```
#define BREAK_INT      ((inportb(P_LSR) & 0x10) = = 0x10)
```

— Para inicio de *break* (en este caso la salida de transmisión del 8250 queda en un nivel de *spacing* de entre 3 y 15 volts, sin importar lo que se transmita.

```
#define INIT_BREAK      outportb(P_LCR, inportb(P_LCR) : 0x40)
```

— Para detener la generación de la señal de *break*.

```
#define STOP_BREAK      outportb(P_LCR, inportb(P_LCR) & 0xBF)
```

III.3.3.2 Variables globales.

Para este protocolo se usan las siguientes variables, que ya fueron descritas en la sección III.2.3.2:

local_addr, remote_addr, MSG_LEN, rxframe, r, txframe, s, char_buff, user_buff, buff_out, buff_out_index, buff_in, buff_in_index, tx_line y rx_line.

También se definen las siguientes variables:

next_addr: tipo *byte*, dirección de la próxima estación para la configuración del anillo lógico.
position: tipo byte, lugar de la estación dentro del anillo lógico (inicial, intermedia, final).
transmiting: tipo byte, bandera para indicar que la estación está transmitiendo una trama en ese momento.
CarrierSense: tipo byte, bandera para indicar que se está detectando alguna actividad en la red. Se enciende mientras se estén recibiendo datos y no se detecte el fin de trama.
collision: tipo byte, bandera para indicar la existencia de una colisión de datos
Content: tipo byte, bandera para indicar que actualmente se está resolviendo una colisión.
count_coll: tipo word, contador del número de colisiones detectadas. .
MSG: tipo byte, bandera para indicar que se tiene un dato en espera de ser transmitido.

III.3.3.3 Semáforos.

Para este protocolo se utilizan los siguientes semáforos:

Nombre	Valor Inicial	Función
HstRdySem	0	Indica que el usuario tiene listo un dato para ser transmitido.
FrmArrSem	0	Una trama completa está lista para el proceso de recepción de nivel 2.
FrmRdy	0	Hay una trama para el usuario.
EnableHost y ProcMess2	0 0	El programa main en realidad también es un proceso. Se sincroniza con UserTx mediante estos 2 semáforos.
ProcMess	0	Ha terminado de ejecutarse un llamado a Transmit Frame.
CollSem	0	Indica que una colisión ha ocurrido. Desbloquea Contention.
ContSem	0	Se ha terminado de resolver una colisión.
TOKEN_Sem	0	Se ha recibido la estaleta para esta estación.
FINAL_MSG_Sem	0	Se ha terminado de resolver una colisión.

III.3.3.4 Procesos.

Al igual que en el protocolo anterior, son 5 los procesos usados para la implantación, además de main.

Las funciones de los procesos UserTx, UserRx, TxN2 y RxN2 ya fueron descritas (brevemente) en la sección III.2.3.5. El único proceso nuevo es Contention.

La función de Contention es resolver las colisiones existentes de una manera determinística.

En las siguientes secciones se incluye una descripción detallada de los procesos main, TxN2 RxN2 y Contention para el protocolo CSMA/CDR

III.3.3.4.1 Proceso main.

El proceso main es casi igual al del protocolo por sondeo descentralizado, excepto que la rutina init_station y la rutina OpenStation presentan algunas pequeñas diferencias.

Por principio de cuentas, en este caso no es necesario diferenciar entre estaciones MASTER y SLAVE. En este protocolo todas las estaciones tienen la misma jerarquía.

En cambio, si es necesario diferenciar la posición de la estación dentro del anillo lógico (inicial, media, final), lo cual se hace dentro de esta rutina.

Además, para el caso de estaciones tipo inicial y media, es necesario indicar cuál es la siguiente estación en el anillo lógico.

En cuanto a la rutina OpenStation, las diferencias son exclusivamente en cuanto a los semáforos que se inicializan y en cuanto a que en esta implantación del protocolo no se utilizan temporizadores.

A continuación se incluye el código de OpenStation e init_station para este protocolo:

```
/*****************************************************************************
                        init_station:  Inicializa una estación.
*****************************************************************************/

init_station(byte *l_addr, byte *r_addr, byte *len)

{
    char c;

    gotoxy(5,23);
    printf("Dar la dirección de esta máquina:    ");
    *l_addr = getint( );
    gotoxy(5,23);
    printf("Dar dirección de estación remota:    ");
    *r_addr = getint( );
    gotoxy(5,23);
    printf("                                     ");
    gotoxy(5,23);
    printf("Longitud del Mensaje:   ");
    *len = getint( );
    gotoxy(5,23);
    printf("Position en el anillo (I, M, F) :    ");
    position = getch( );
    if (position != 'F' && position != 'f') {
        gotoxy(5,23);
        printf("Dirección de próxima estación : ");
        next_addr = getint( );
    }
}

/*****************************************************************************
OpenStation:  Crea procesos, semáforos y temporizadores a ser usados por la estación. Salva y asig-
              na vectores de interrupción.
*****************************************************************************/

procedure OpenStation( )
{
    time_int = getvect(0x08);
    initkernel ( );
    p0 = createprocess ((offset) TxN2, 1024, "TxN2");
    p1 = createprocess ((offset) RxN2, 1024, "RxN2");
    p2 = createprocess ((offset) UserTx, 1024, "UserTx");
    p3 = createprocess ((offset) UserRx, 1024, "UserRx");
    p4 = createprocess ((offset) Contention, 1024, "Cont");

HstRdySem        = initsem(0);
EnableHost       = initsem(0);
ProcMess         = initsem(0);
FrmArrSem        = initsem(0);
FrmRdy           = initsem(0);
CollSem          = initsem(0);
ContSem          = initsem(0);
FINAL_MSG_Sem    = initsem(0);
TOKEN_Sem        = initsem(0);

    init_uart ( );
    ctrl_break = getvect (0x1B);
    intr_4 = getvect (0x0C);
    setvect (0x0C, int_UART);
    setvect (0x1B, breaker);
}
```

III.3.3.4.2 Proceso TxN2.

El proceso TxN2 es un ciclo infinito durante el cual:

— Se queda bloqueado hasta que haya sido realizado un llamado a TransmitFrame.
— Se indica mediante MSG que hay un mensaje esperando ser enviado a la red.
— Si se está resolviendo una colisión, evita el envío del mensaje hasta que aquella haya sido resuelta.
— Forma la trama en el *buffer* de salida.
— Espera a detectar ausencia de actividad en la red para poder transmitir (*Carrier Sense*).
— Si hubo una colisión y ya se está resolviendo, espera a que termine de resolverse (en este punto, el proceso Contention ya sabrá que esta estación tiene un dato a enviar y este proceso lo hará).
— En caso contrario intentará enviar la trama a la red sin más trámite. Si después de intentar enviar la trama a la red se detectó que hubo colisión, entonces iniciará la transmisión de un *break* (para avisar a todas las estaciones que hubo una colisión) y esperará a que el proceso Contention envíe la trama a la red.

— Al terminar, indicará que ya no hay un mensaje a transmitir (MSG = FALSED) y le hará saber a la rutina TransmitFrame que su petición de servicio ya fue atendida (mediante un signalns al semáforo ProcMess).

El código de este proceso se muestra a continuación:

```
/*************************************************************************
        TxN2:  Proceso encargado de transmitir los datos recibidos del usuario.
*************************************************************************/

procedure TxN2( )
{
    while(flag) {
        wait(HstRdySem);                    /*  Espera a tener un dato a enviar  */
        MSG = TRUE;
        while(Content);
        FormFrame(s->dest_addr, s->srce_addr, s->control, s->info);
        do {
            collision = FALSE;
            waittime(1);
        } while (CarrierSense);             /*  Espera a que el canal esté libre  */
        waittime(10);
        if (!Content) {
            transmiting = TRUE;
            SendFrame( );                   /*  Envia la trama a la red  */
            if (collision) {
                collision = FALSE;
                INIT_BREAK;
                wait(ContSem);
            }
        }
        else {
            wait(ContSem);
        }
        MSG = FALSE;
        signalns(ProcMess);
    }
}
```

Para este protocolo la transmisión no se hace por interrupciones, sino por *polling*. Por lo tanto la rutina SendFrame es un poco diferente y además se utiliza una rutina SendByte, que envía los caracteres al canal de comunicación. El código de ambas rutinas se muestra enseguida:

```
/*****************************************************************************
                SendFrame:  Envia información de estación local a otra estación.
*****************************************************************************/

SendFrame(void)
{
    buff_out_index = 0;
    buff_in_index = 0;
    do {
        SendByte(buff_out[buff_out_index]);
    } while(buff_out[buff_out_index++] != END_FRAME && (transmiting));
    transmiting = FALSE;
}

/*****************************************************************************
                    SendByte:  Envia un byte por el puerto de datos del 8250.
*****************************************************************************/

SendByte(byte car)
{

    while (!(inportb(0x3FD) & 0x20)); /*  Espera a poder transmitir datos*/
    outportb(P_DATA, car);
}
```

III.3.3.4.3 Proceso RxN2.

El proceso RxN2 es un ciclo infinito durante el cual se ejecutan las siguientes acciones:

— Espera a que le llegue una trama completa.
— La trama que llegó en el *buffer* de entrada se transfiere a una estructura tipo trama.
— Si la trama que llegó es una estafeta correspondiente a esta estación, se enciende el semáforo TOKEN_Sem.
— Si la trama que llegó es un mensaje de fin de resolución de colisión, se enciende el semáforo FINAL_MSG_Sem.
— Si llegó una trama dirigida a la estación, se le pasa al usuario (labor que hará ReceiveFrame) al encenderse el semáforo FrmRdy.

El código de este proceso viene a continuación:

```
/*****************************************************************************
RxN2:   Proceso que recibe tramas completas y envia la información correspondiente al proceso de
        recepción del usuario.
*****************************************************************************/

procedure RxN2( )
{
    while(llag) {
        wait (FrmArrSem);
        get_frame(r);
        gotoxy(1, 1); printf("T");
        if (r->control = = TOKEN && r->dest_addr = = local_addr)
            signalns(TOKEN_Sem);
        if (r->control = = FINAL_MSG)
            signalns(FINAL_MSG_Sem);
        if (r->dest_addr = = local_addr)
            signalns(FrmRdy);
    }
}
```

III.3.3.4.4 Proceso Contention.

El proceso Contention es el encargado de resolver las colisiones. Es un ciclo infinito en el cual:

— Se espera a que se indique la existencia de una colisión mediante el semáforo CollSem
— Se hacen algunos ajustes en las variables para permitir la resolución sin problema de las colisiones.
— Dependiendo de la posición de la estación en la red se efectúan las siguientes tareas.

i) Inicial: se envían los datos (si los hay), se envía estafeta a la siguiente estación y se espera a la llegada del mensaje de fin de resolución de las colisiones.
ii) Media: se espera la llegada de la estafeta correspondiente a la estación y se hace lo mismo que para una estación Inicial.
iii) Final: se espera la llegada de la estafeta, se envían los datos (si los hay) a la red y se envía un mensaje de fin de resolución de las colisiones.

El código de este proceso se muestra a continuación:

```
/*****************************************************************************
                    Contention:  Proceso encargado de procesar colisiones.
*****************************************************************************/
procedure Contention( )
{
    while(flag) {
        waitt(CollSem);
        waittime(10);
        count_coll1 + +;
        transmiting = FALSE;
        CarrierSense = FALSE;
        buff_in_index = 0;
        switch(position) {
            case 'I':
            case 'i': if (MSG) {
                        transmiting = TRUE;
                        SendFrame( );
                        waittime(1);
                    }
                    SendTok(TOKEN);
                    wait(FINAL_MSG_Sem);
                    break;
            case 'm':
            case 'M': wait(TOKEN_Sem);
                    if (MSG) {
                        transmiting = TRUE;
                        SendFrame( );
                        waittime(1);
                    }
                    SendTok(TOKEN);
                    wait(FINAL_MSG_Sem);
                    break;
            case 'f':
            case 'F': wait(TOKEN_Sem);
                    if (MSG) {
                        transmiting = TRUE;
                        SendFrame( );
                        waittime(1);
                    }
                    SendTok(FINAL_MSG);
                    break;
            default: break;
        }
```

```
        Content = FALSE;
        signalns(ContSem);
    }
}
```

En este proceso se utiliza la función SendTok, cuyo único parámetro es el tipo de mensaje enviado (TOKEN o FINAL_MSG). Es obvio que para estos mensajes no podemos variar la estación destino, pues el anillo lógico no es flexible (en esta implantación) una vez que ha sido configurado.

El código correspondientes a SendTok se incluye a continuación:

```
/*****************************************************************************
SendTok:  Envia token (cuando hubo colisión y la red tome forma de anillo lógico).
*****************************************************************************/

SendTok(byte type)
{

    buff_in_index = 0;
    SendByte(next_addr);
    SendByte(local_addr);
    SendByte(type);
    SendByte(END_FRAME);
    buff_in_index = 0;
}
```

III.3.3.4.5 Rutina de atención a las interrupciones del 8250.

Esta rutina atiende los casos de interrupciones por recepción y por llegada de una señal de *break*.

En el caso de las interrupciones por recepción de caracteres esta rutina hace lo siguiente:

— Si no se está transmitiendo, quiere decir que el dato recibido lo envio otra estación, por lo que se enciende la bandera CarrierSense.

— Se lee el dato recibido.

— Se llama a la rutina form_frame, que se encarga no sólo de formar la trama, sino de detectar una colisión en un dato transmitido por esta estación.

Para las interrupciones por *break* se hace lo siguiente:

— Se detecta si la interrupción recibida fue debida a algún error (el *break* es uno de ellos).

— Si el error recibido fue un *break* se suspende la transmisión de este (se espera que el *break* recibido haya sido el mismo que se envió, aunque esto no fuese asi, no afecta suspender la transmisión del *break*)

— Se indica que hubo una colisión y se despierta al proceso Contention al hacer un signalns sobre el semáforo CollSem.

El código de la rutina de interrupción y de la rutina form_frame se muestran enseguida:

```
/*****************************************************************************
                    intUART:  Rutina de atención a interrupciones del 8250.
*****************************************************************************/
```

```
void interrupt Int_UART ( )
{

    byte status;
    byte dato_rx;

    do {
        status = inportb(P_IIR);
        if (RECERR) {                                   /*  Error  */
            if (BREAK_INT) {
                STOP_BREAK;
                if (!Content) {
                    Content = TRUE;
                    signalns(CollSem);

            }
        }
        else
        if (INTRX) {                                    /*  Recepción  */
            if (!transmiting) CarrierSense = TRUE;
            dato_rx = inportb(P_DATA);
            form_frame (dato_rx);
        }
    } while (PEND);
    outportb(0x20, 0x20);
}

/*****************************************************************************
   form_frame:  Forma los caracteres recibidos en la trama de entrada y detecta colisiones.
*****************************************************************************/

form_frame(byte dato_rx)
{

    if (dato_rx ! = END_FRAME) {
        buff_in[buff_in_index + + ] = dato_rx;
        if (transmiting &&
            (buff_in[buff_in_index-1] ! = buff_out[buff_in_index-1]))
            transmiting = FALSE;
            collision = TRUE;
            CarrierSense = FALSE;
            buff_in_index = 0;
        }
    }
    else {
        CarrierSense = FALSE;
        buff_in_index = 0;
        signalns(FrmArrSem);
    }
}
```

La rutina form_frame hace lo siguiente:

— Si el dato recibido fue fin de trama se lo hace saber a RxN2 mediante el semáforo FrmArrSem.
— En caso contrario:
  — Forma el dato recibido en el *buffer* de recepción.
  — Si estamos transmitiendo y el caracter recibido no coincide con el caracter que se acaba de mandar, se aborta la transmisión de la trama y se indica que hubo una colisión. Quien se dará cuenta de esos será el proceso TxN2, que a su vez mandará una señal de *break* que eventualmente despertará al proceso Contention.

IV. Pruebas de los Protocolos.

IV.1 Evaluaciones teóricas.

Hasta este momento nos hemos concentrado exclusivamente en los aspectos operativos de los protocolos propuestos. Llega el momento de hacer algunas evaluaciones del comportamiento esperado de los protocolos propuestos, con el objeto de establecer algunas conclusiones acerca de lo que puede esperarse de ellos.

Existen en la literatura algunas evaluaciones hechas a protocolos de acceso controlado (como son los protocolos por sondeo y los anillos lógicos) y a protocolos de acceso aleatorio. Algunas de esas evaluaciones son utilizadas a continuación para establecer algún tipo de parámetro de comparación con los resultados que posteriormente se presentan.

IV.1.1 Evaluación teórica del protocolo por sondeo descentralizado.

La evaluación para este tipo de protocolo es la que corresponde a cualquier protocolo por sondeo, incluido el caso de que los anillos lógicos. En la introducción estimamos de manera deductiva el tiempo máximo de acceso a la red. Ahora nos interesa ver el tiempo promedio de acceso a la red y el aprovechamiento del ancho de banda de la red. Schwartz [SCHW87] incluye un análisis para protocolos por *polling* del que se toman a continuación algunos resultados.

A continuación vemos la forma de calcular el tiempo de acceso en función de los parámetros de la red.

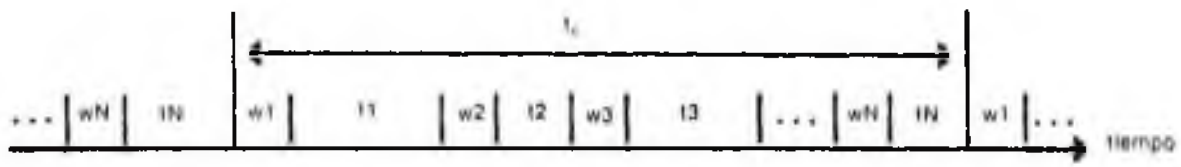

Figura 4.1 Tiempo de sondeo para todas las estaciones.

Consideremos primero el tiempo requerido para hacer una vez el sondeo para N estaciones. Llamemos a este tiempo t_c. Observando la figura 4.1 vemos que aparecen alternamente un tiempo requerido para hacer el sondeo (llamado w_i) y un tiempo para la transmisión de los datos (llamado t_i). De aquí resulta evidente que:

$$t_c = \sum_{i=1}^{N} w_i + \sum_{i=1}^{N} t_i \qquad (4.1)$$

Podemos considerar que el primer término de la ecuación es constante para cada ciclo de sondeo, y lo llamaremos L. El segundo término es diferente para cada ciclo. No nos interesa el tiempo

de un ciclo en particular, sino el tiempo promedio, por lo que será necesario reflejar este hecho en la ecuación.

$$t_c = L + \sum_{i=1}^{N} t_i \qquad (4.2)$$

Ahora supondremos que los *buffers* (que almacenan los mensajes a ser transmitidos) de todas las estaciones son infinitos, y que a cada estación llegan λ_i paquetes/seg., con una longitud promedio de l bits de datos, con l' bits extras (debidos a los campos de la trama que no son datos, lo que se conoce como *overhead*). Digamos, además que la capacidad de la red (su velocidad en bits por segundo) es C. Cuando llega la señal de sondeo para transmitir hay $\lambda_i t_c$ paquetes para ser transmitidos, pues son los paquetes que se acumularon desde la anterior señal de sondeo hasta la actual (esto implica que al llegar la señal de sondeo se envían a la red todos los paquetes acumulados). Tenemos entonces que el tiempo promedio que tardarán en transmitirse esos paquetes será:

$$t_i = \lambda_i t_c (l + l')/C = t_c \varrho_i \qquad (4.3)$$

En donde ϱ_i es la intensidad de tráfico provocada por la estación i. Sustituyendo (4.3) en (4.2) se tiene que:

$$t_c = L / (1-\varrho) \qquad (4.4)$$

En donde $\varrho = \sum_{i=1}^{N} \varrho_i$, y es el tráfico producido por todas las estaciones. Esto constituye una

buena medida para evaluar el aprovechamiento de la red

Hasta este momento hemos seguido el análisis hecho por Schawartz. Recordemos ahora cuáles son los parámetros que nos interesa medir. Primero nos interesa calcular ϱ, para evaluar el aprovechamiento de la red. De la expresión (4.4) podemos despejar ϱ, que es:

$$\varrho = 1 - (L/t_c) \qquad (4.5)$$

¿Cuál es el máximo aprovechamiento que podemos tener en la red? ϱ es una cantidad positiva, por lo que $0 <= L/t_c <= 1$. Para maximizar ϱ, necesitamos minimizar L/t_c. L es un valor constante, como ya hemos visto, por lo que para minimizar esa expresión, es necesario maximizar t_c. Así:

$$\varrho_{max} = 1 - (L/t_{c_{max}}) \qquad (4.6)$$

Es posible ver [de (4.2)] que:

$$t_{c_{max}} = L + \sum_{i=1}^{N} t_{i_{max}} \qquad (4.7)$$

Para maximizar t_i, observamos (4.3) y notamos que l' y C son constantes. Supongamos que cada vez que se recibe un mensaje de sondeo el nivel de enlace de datos sólo envía un paquete a la red. Esta suposición es válida para la implantación realizada, pues la llamada a la rutina TransmitFrame(...) hace un wait sobre el semáforo ProcMess (ver capítulo III), lo que por una parte indica que se está

usando un *buffer* de tamaño 1, y que el nivel de enlace de datos no acepta más que un mensaje de los niveles superiores por cada ciclo de sondeo. Eso hace que el producto $\lambda_i t_c$ sea, en este caso, 1. En general, $\lambda_i t_c$ tendrá como valor máximo el tamaño del *buffer* del nivel 2. Por lo tanto, para maximizar t_c nos basta con maximizar l, que es la longitud máxima de una trama. Así, sustituyendo (4.7) (con las consideraciones hechas anteriormente) en (4.6) tenemos que una expresión para calcular ϱ_{max} es:

$$\varrho_{max} = 1 - \frac{L}{L + Nt_{i_{max}}} = 1 - \frac{L}{L + N\frac{l_{max} + l'}{C}} \qquad (4.8)$$

Consideremos ahora que los mensajes de sondeo no llevan datos, (a diferencia del protocolo HDLC, en donde los mensajes de control también pueden llevar datos). Eso hace que:

$$L = N\frac{l'}{C} \qquad (4.9)$$

Y sustituyendo (4.9) en (4.8) y simplificando se obtiene:

$$\varrho_{max} = 1 - \frac{l'}{2l' + l_{max}} \qquad (4.10)$$

Si $K << l'$ la expresión anterior es válida para cualquier ϱ dada la longitud máxima del mensaje. La figura 4.2 es una gráfica de ϱ contra l, para $l > 320$ *bits* ($l' = 32$ *bits*). La grálica nos será útil para compararla con los resultados experimentales obtenidos.

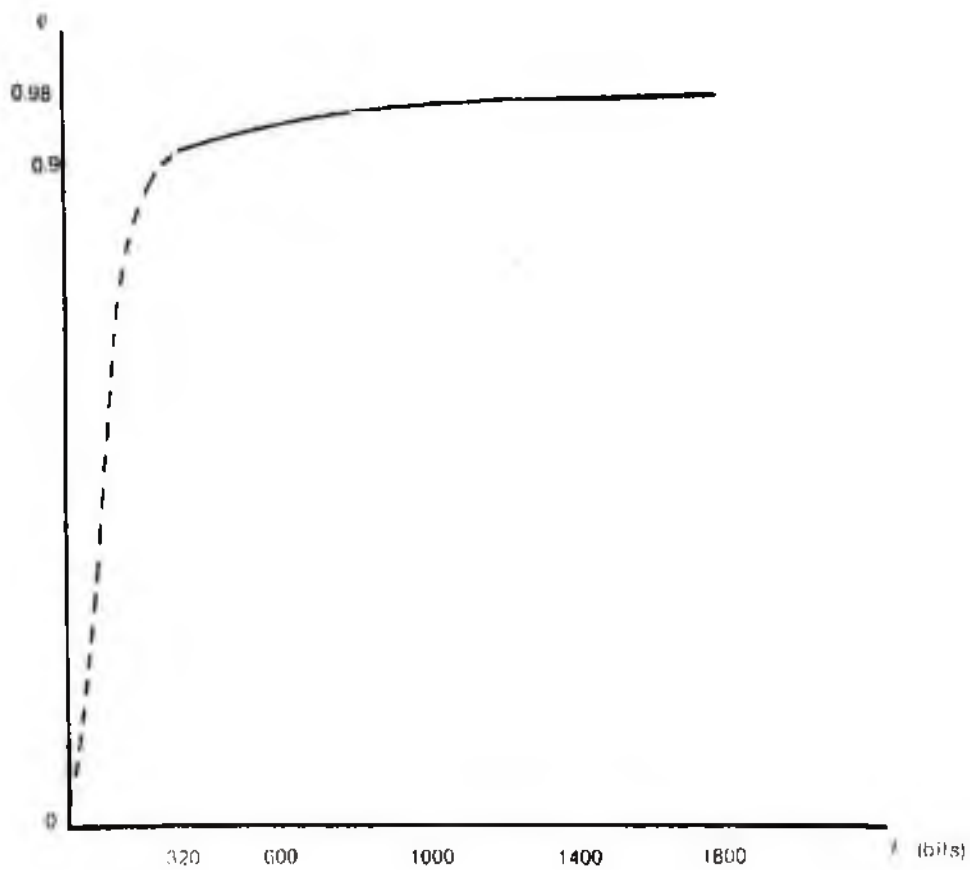

Figura 4.2 Aprovechamiento de la red vs longitud del mensaje.

Si tenemos que el máximo número de bytes que se pueden transmitir en un mensaje es 255 (l_{max}), y que el número de bytes de *overhead* en un mensaje es de 4 (l'), entonces vemos que el aprovechamiento máximo de la red es:

$$\varrho_{max} = 0.98$$

Lo que quiere decir que en el mejor de los casos la red se utiliza para transmitir datos en un 98%.

Esta estimación se vuelve poco realista si consideramos que en estos cálculos se están pasando por alto 3 factores importantes:

— El espaciamiento entre tramas. Antes de lanzar un mensaje a la red se programó un retardo de 5 *ticks* para dar espacio entre tramas.
— Los bits de inicio, de parada y de paridad de cada uno de los *bytes* transmitidos, que no llevan información.
— El tiempo que lleva la ejecución de los procesos que se encargan tanto de enviar tramas de habilitación (MasterProc) como de enviar datos.

El primero de los factores es conocido (5 *ticks* son 0.277 seg.). El segundo puede ser calculado a partir de los parámetros de la red. El último es difícil de evaluar.

Considerando lo anterior, vemos que podemos hacer una aproximación más realista del aprovechamiento efectivo de la red si dividimos el tiempo que toma transmitir *bits* efectivos de información en un ciclo entre el tiempo que lleva un ciclo completo. Esto es:

$$\varrho = \frac{N(l_{max}/C)}{N(l_{max}/C) + 2N(l'/C) + N(l_{ex}/C) + t_{inter}}$$

donde:

$N(l_{max}/C)$ es el tiempo correspondiente a los bits efectivos transmitidos en un ciclo de sondeo.

$2N(l'/C)$ es el tiempo correspondiente a los bits transmitidos en los mensajes de sondeo y los bits de las tramas de información que corresponden a campos de dirección, de control y de fin de trama.

$N(l_{ex}/C)$ es el tiempo correspondiente a los bits de inicio, de paridad y de parada.

t_{inter} es el tiempo de espacios entre tramas.

Tomando N = 3 (que es el número de estaciones que se utilizaron para hacer las evaluaciones experimentales) tenemos que el valor resultante para ϱ es:

$$\varrho = 0.54$$

La figura 4.2bis muestra la gráfica resultante para el aprovechamiento de la red contra la longitud de los mensajes con las consideraciones hechas anteriormente.

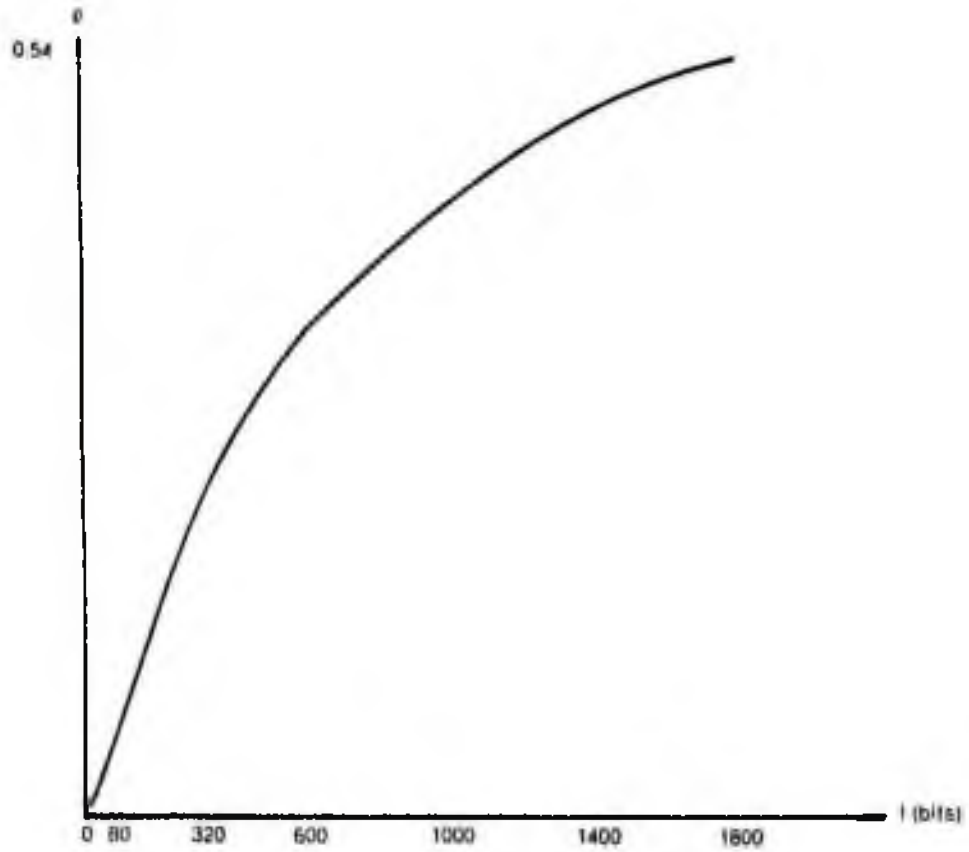

Figura 4.2bis Aprovechamiento de la red vs. longitud del mensaje: una aproximación más realista.

En cuanto al tiempo máximo de acceso, de manera intuitiva podemos observar que este sería el tiempo máximo de un ciclo de sondeo ($t_{c_{máx}}$), el cual viene dado [de (4.4)] por:

$$t_{c_{máx}} = L / (1 - \rho_{máx}) \qquad (4.11)$$

Es interesante observar la forma que presenta la curva obtenida de graficar la ecuación anterior. A continuación se muestra:

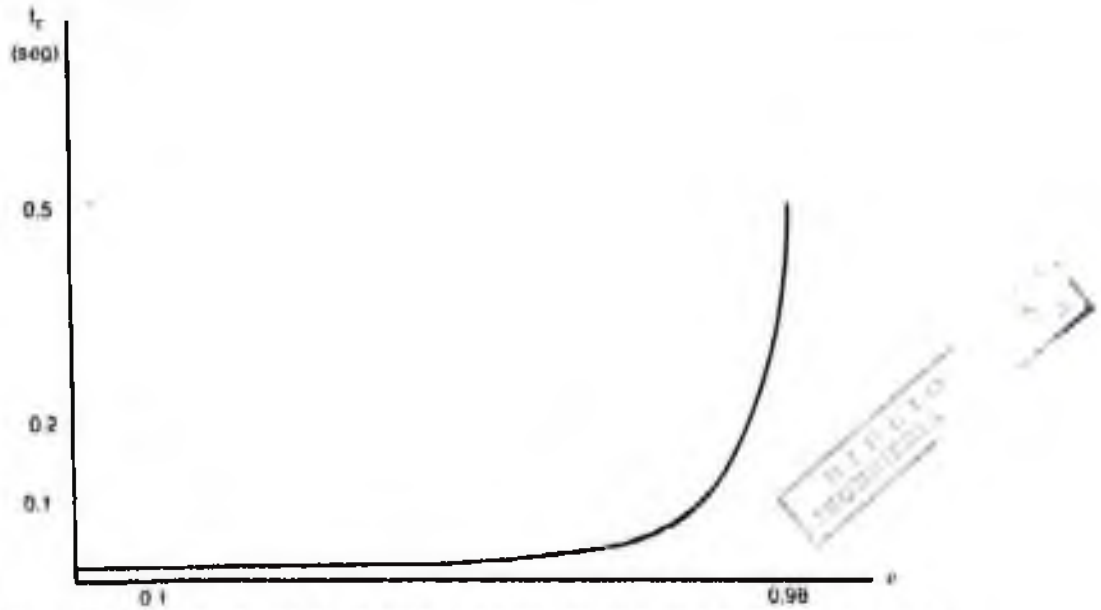

Figura 4.3 Tiempo máximo de un ciclo de sondeo contra aprovechamiento de la red.

Continuando con nuestro análisis, tenemos que sustituyendo (4.9) en (4.11) y simplificando se obtiene:

$$t_{c_{máx}} = \frac{Nl' / C}{(1 - \varrho_{máx})} \qquad (4.12)$$

Que es función del número de estaciones activas en la red. Con los datos utilizados para calcular $\varrho_{máx}$ tenemos que:

$$t_{c_{máx}} = (0.22)N \text{ seg.}$$

IV.1.1 Evaluación teórica del protocolo CSMA/CDR.

El aprovechamiento máximo de la red se da en condiciones de tráfico intenso. En tales condiciones podemos considerar que las colisiones van a ocurrir para cada trama que se desee enviar a la red, por lo que el cálculo de aprovechamiento máximo de la red para este protocolo será muy semejante al calculado anteriormente, con la salvedad que hay que considerar los tiempos de sensado de la portadora, de transmisión incompleta de la trama y el tiempo de detección de colisión. Los dos primeros tiempos mencionados no pueden ser mayores (cada uno) que el tiempo que tarda en ser transmitida una trama completa con la máxima longitud de bytes posibles. El segundo puede considerarse muy pequeño con respecto a los 2 primeros puesto que es el tiempo en que tarda un byte completo en transmitirse como una señal de *break*. Claramente, el tiempo requerido para transmitir un byte es mucho menor que el tiempo requerido para transmitir 255. Considerando lo anterior, la ecuación (4.2) quedará:

$$t_t = L + \sum_{i=1}^{N} t_i + t_{col} + t_c \qquad (4.13)$$

donde t_{col} es el tiempo máximo que dura una colisión, y es igual a ($l_{máx}$ = l')/C.

Y siguiendo un procedimiento semejante al descrito para el protocolo por sondeo descentralizado tendremos una expresión para $\varrho_{máx}$ como la siguiente:

$$\varrho_{máx} = 1 - \frac{L + t_{col} + t_{wait}}{L + Nt_{wait}} = \frac{Nl' + 2(l' + l_{máx})}{N(2l' + l_{máx})} \qquad (4.14)$$

Lo que para N<<2 tiende al valor obtenido para el protocolo anterior.

La figura 4.4 muestra una gráfica de aprovechamiento máximo de la red contra el número de estaciones.

En cuanto al tiempo máximo de acceso a la red, vemos que si se evalúa (4.13) con los datos manejados hasta el momento, el resultado será:

$$t_{c_{máx}} = (0.22)N + 0.42 \text{ seg.}$$

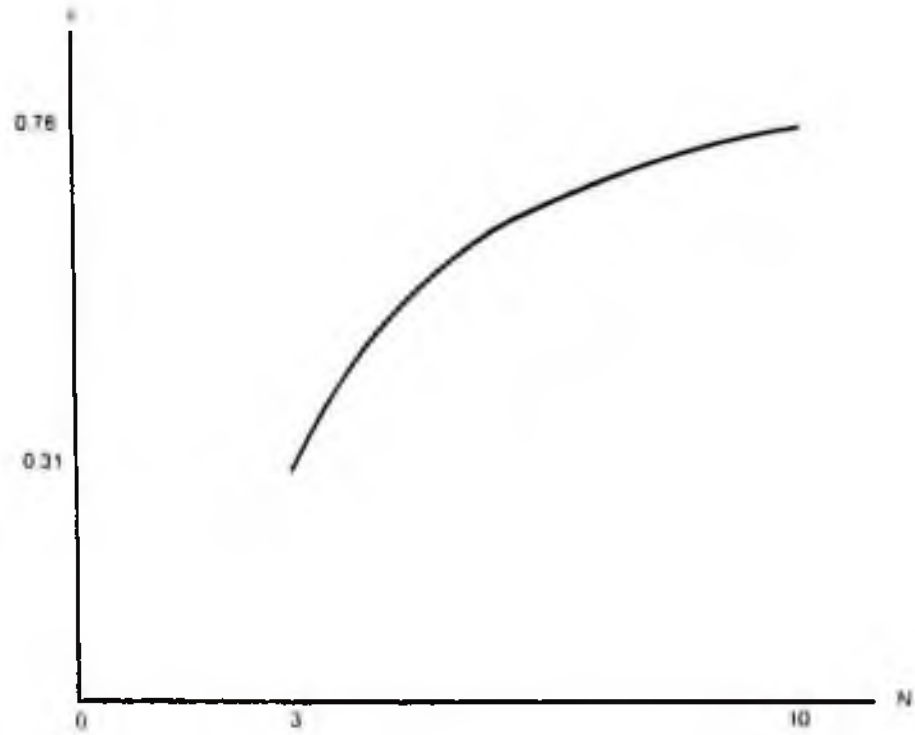

Figura 4.4 Máximo aprovechamiento de la red contra número de estaciones.

IV.2 Evaluaciones experimentales.

Son dos los parámetros que deseamos medir en la red. La manera de medirlos experimentalmente fue la siguiente:

Para el tiempo de acceso a la red se arrancó un temporizador justo antes de la llamada a TransmitFrame(. . .) en el proceso de usuario UserTx y se detectó cuanto tiempo habia transcurrido desde entonces hasta el fin de ejecución del servicio de transmisión de la información a la red (recuérdese que TransmitFrame es sincrona, lo cual permitió medir el tiempo de acceso de esta manera). En este punto fue necesario incluir en el núcleo de concurrencia una rutina llamada get_time( ), para averiguar cuál era el tiempo transcurrido.

Para el aprovechamiento de la red, la medición se llevó a cabo de manera indirecta y consistió en contar el número de tramas de información recibidas durante cierto tiempo. Como era conocido el número de *bytes* enviados en cada trama (tamaño de mensaje), se podia saber el número de *bits* efectivos de información transmitidos en un segundo. Dividiendo estos entre la velocidad de la red (capacidad), se encuentra una razón del aprovechamiento de la red. En este caso se utilizó otro temporizador para poder acotar el lapso de tiempo durante el cual se contaron las tramas recibidas.

IV.2.1 Resultados experimentales para el protocolo por sondeo descentralizado.

La siguiente tabla muestra los resultados experimentales obtenidos para este protocolo. El número de estaciones utilizadas fue de 3, efectuando sondeo sólo para las tres estaciones conectadas a la red. Las 2 últimas columnas muestran los resultados de mayor interés (aprovechamiento de la red y tiempo de acceso). La abreviación beps significa *bits* efectivos por segundo. Las unidades de tiempo de las mediciones originales eran *ticks* (1 *ticks* = 1/18 seg.). En la tabla aparecen convertidas a segundos.

Notamos que el aprovechamiento de la red está en niveles muy por abajo de los esperados. Esto se debe en parte a que existen varios factores que no son considerados en el cálculo teórico y que en definitiva incluyen en los resultados obtenidos. Tales factores son:

— El espaciamiento entre tramas. Antes de lanzar un mensaje a la red se programó un retardo de 5 *ticks* para dar espacio entre tramas.

— Los bits de inicio, de parada y de paridad de cada uno de los *bytes* transmitidos.

— El tiempo que lleva la ejecución de los proceso que se encargan tanto de enviar tramas de habilitación (MasterProc) como de enviar datos.

Longitud bytes	bits	# tramas/tiempo cada 10 s.	cada s.	beps	beps/ 9600	tacc
10	80	16.5	1.65	132	0.014	1.84
30	240	15.5	1.55	372	0.038	1.94
70	560	14	1.4	784	0.082	2.11
110	880	13.5	1.35	1188	0.124	2.22
150	1200	12.5	1.25	1500	0.156	2.44
200	1600	11.5	1.15	1840	0.192	2.55
250	2000	10.5	1.05	2100	0.219	2.77

Más interesante es ver la gráfica que se obtiene de los datos experimentales para beps/9600 (aprovechamiento de la red) contra las longitudes de mensaje. La figura 4.5 muestra tal gráfica. Compárese contra la de la figura 4.2. Podemos observar el mismo comportamiento de la curva, si bien los valores son bastante inferiores.

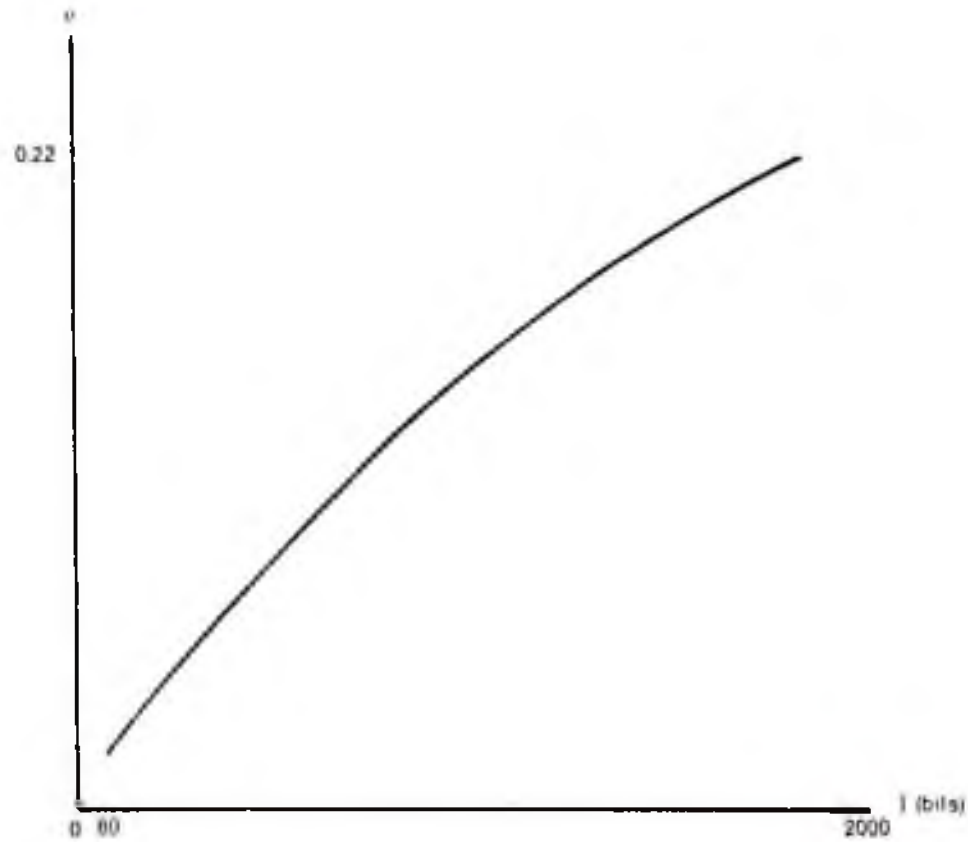

Figura 4.5 Aprovechamiento de la red contra longitud de mensaje (experimental por sondeo)

IV.2.2 Resultados experimentales para el protocolo CSMA(CDR.

Para las mediciones de este protocolo se tomaron 2 casos. El primero con una sola estación transmitiendo a la red (con lo que se evitan completamente las colisiones) y el segundo forzando colisiones entre 2 estaciones (inicial y final cuando se forma el anillo lógico) mediante el recurso de eliminar del programa la parte de sensado de la portadora.

Para el primer experimento (una sola estación sin colisiones) se esperaba que el aprovechamiento de la red se aproxima a los niveles máximos. La figura 4.6 muestra que no es así. Esto se debe en parte a que no todos los *bits* transmitidos por cada *byte* son útiles (bit de inicio, bit de paridad, bit de parada y a que los programas toman algún tiempo en ejecutarse (las salidas a pantalla afectan el tiempo de ejecución de manera notoria). Este experimento justifica de manera clara el porqué los resultados prácticos se encuentran por debajo de los resultados teóricos.

Tabla para una sola estación sin colisiones.

Longitud bytes	bits	# tramas/tiempo cada 10 s	cada s.	beps	beps/ 9600	Iacc
10	80	269	26.9	2152	0.20	0.3
30	240	163	16.3	3912	0.40	0.7
70	560	92	9 2	5152	0.53	1.5
110	880	63	6.3	5544	0.57	2.3
150	1200	48	4.8	5760	0.60	3.1
200	1600	37	3.7	5920	0.61	4.2
250	2000	30	3.0	6000	0.625	5 5

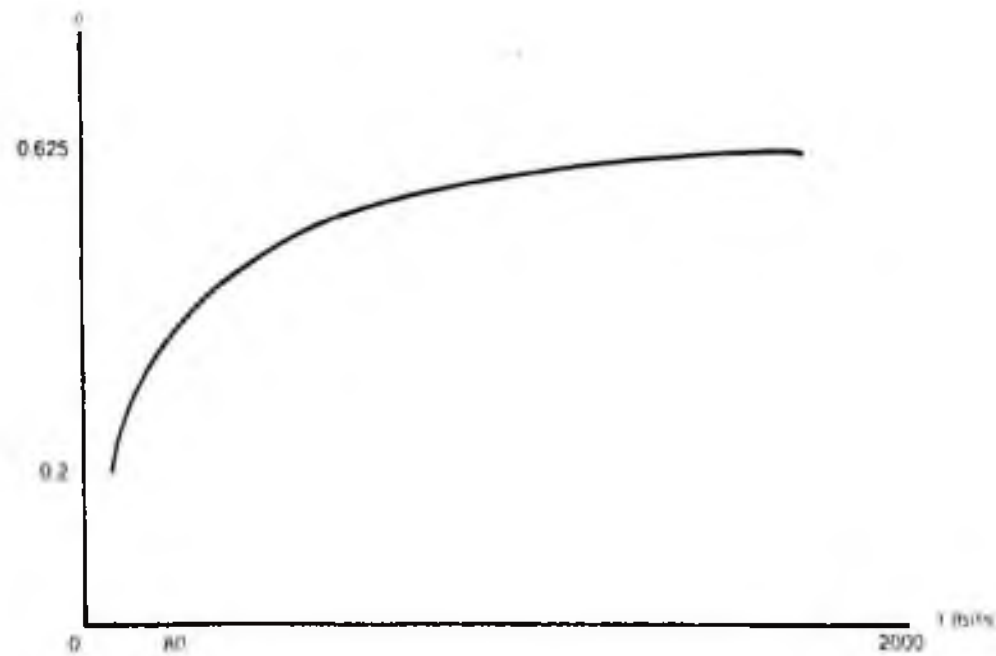

Figura 4.6 Aprovechamiento de la red contra longitud de mensaje (experimental CSMA/CDR sin colisiones).

Para el segundo experimento (2 estaciones colisionando en cada trama que envían) no fue posible lograr que las estaciones colisionarán con cada mensaje, pues ambas estaciones se sincronizaban de tal manera que no chocaban sus mensajes.

Para poder obtener alguna evaluación un retraso en el envío de cada *byte* (con dos ciclos for anidados que duraban aproximadamente un *tick*). Los resultados que se muestran de manera gráfica en la figura 4.7 son el resultado de mediciones hechas durante intervalos muy largos de tiempo, para poder obtener algún promedio de tramas transmitidas. Observamos en este caso que el aprovechamiento de la red tiende muy rápidamente a 1. Esto es explicable porque el tiempo que requiere para enviar los *bits* que no contienen datos se hace despreciable (recuérdese que el retraso se introdujo por cada *byte* enviado, no por cada *bit*) y porque el tiempo de ejecución de los programas también se hace muy pequeño.

En este caso se consideró que la velocidad de la red es de 18 bps (puesto que cada *byte* tarda en enviarse aproximadamente un *tick* por segundo).

Tabla para 2 estaciones colisionando constantemente (protocolo con retraso).

Longitud bytes	bits	#tramas/tiempo cada 100 s.	cada s.	beps	beps/ 18	tacc
10	80	16	0.16	12.8	0.71	1.5
30	240	6	0.06	14.4	0.8	3.1
70	560	2.7	0.027	15.1	0.84	6.1
110	880	1.9	0.019	16.7	0.92	9.2
150	1200	1.4	0.014	16.8	0.93	12.2
200	1600	1.1	0.011	17.6	0.97	16.1
250	2000	0.9	0.009	18	1.0	20

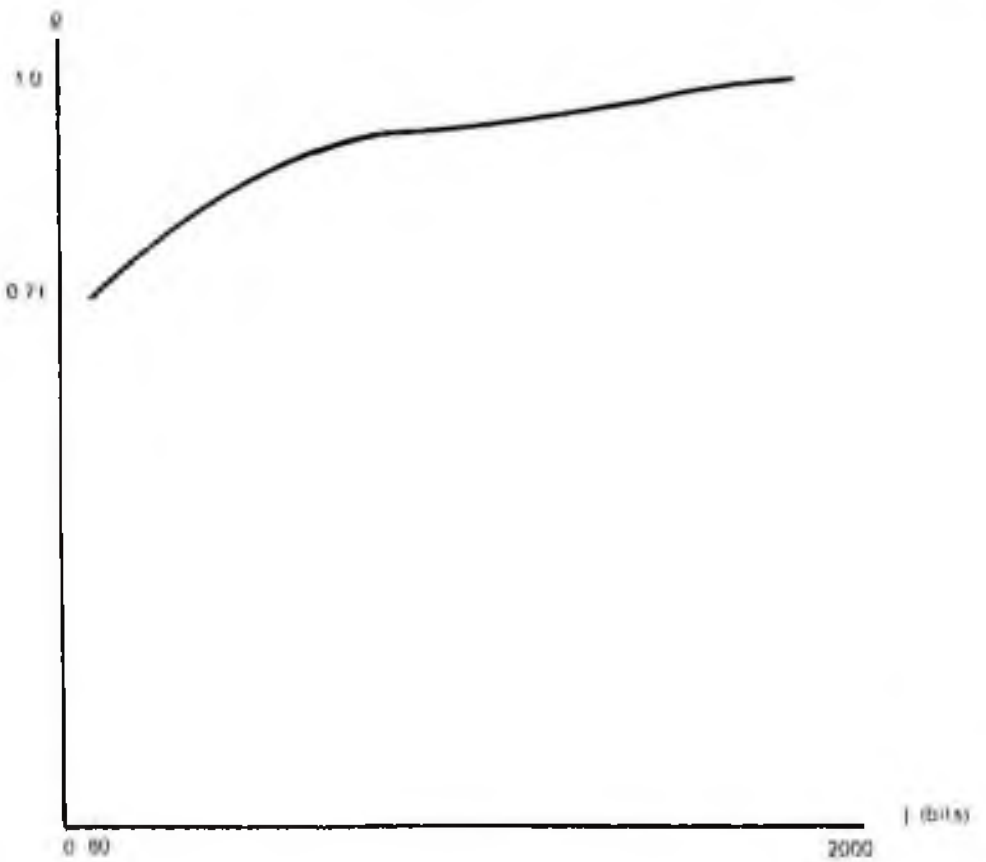

Figura 4.7 Aprovechamiento de la red contra longitud de mensaje (experimental CSMA/CDR con colisiones).

V. Conclusiones

Durante el desarrollo del presente trabajo hemos hecho algunos comentarios que pueden considerarse como aspectos concluyentes de lo expuesto, en particular durante el capitulo anterior. En esta sección puntualizaremos algunos aspectos.

— La implantación realizada es un intento por demostrar la operatividad de los protocolos propuestos. No se puede pretender que este trabajo quede como un *standad*, puesto que muchas de las caracteristicas de este trabajo (por ejemplo la estructura de la trama) lo hacen inadecuado para funcionar en aplicaciones reales. Me parece que seria muy conveniente intentar la implantación de los protocolos aquí propuestos en otros ambientes. Podria sugerirse un proyecto de laboratorio que consistiera en adaptar lo aqui planteado al sistema distribuido que realiza el Ing. Alejandro Tinoco (estudiante de este Centro).

— La modelación de los protocolos no está formalizada. Se demostró la viabilidad del protocolo, pero no se ha demostrado formalmente la corrección de este modelo.

— El protocolo por sondeo descentralizado es mucho más simple que el CSMA/CDR, tanto desde el punto de vista de realización como desde el punto de vista de operación. Tal simplicidad es muy probable que repercutiese favorablemente en aspectos económicos (horas/hombre de desarrollo, de depuración, de mantenimiento; en general, costo del *software*) una vez que se plantease la posibilidad de incluirlo en algun trabajo de indole práctica. Aunque existe circuiteria (ver, por ejemplo, [ATA88]) que soporta el protocolo CSMA/CDR, me parece que el protocolo por sondeo descentralizado puede realizarse a un menor costo. Esta es una apreciación basada exclusivamente en el tiempo que me llevó realizar cada protocolo, y de ninguna manera es una conclusión terminante.

— La confiabilidad del protocolo por sondeo descentralizado me parece mayor que la del protocolo CSMA/CDR, toda vez que el primero puede continuar trabajando si la estación maestra falla, pero el segundo no tiene un mecanismo de recuperación para el caso en que falle una estación que forme parte del anillo. Se puede agregar (se debe agregar en un caso real) tal mecanismo, a costa de una mayor complejidad.

— El tiempo máximo de acceso a la red es menor para el caso del protocolo por sondeo descentralizado. Si no existen colisiones, el tiempo promedio de acceso a la red será menor para el protocolo CSMA/CDR.

— El aprovechamiento de la red depende del número de estaciones que se encuentren en la red y de la longitud de los mensajes manejados. En condiciones de tráfico intenso (muchas estaciones transmitiendo mensajes largos), el aprovechamiento de la red será ligeramente mejor para el protocolo por sondeo descentralizado.

— Los protocolos de acceso al medio por anillos lógicos surgen como una versión descentralizada de un protocolo por sondeo. Con lo expuesto aquí, cabe pensar en la posibilidad de usar un protocolo por sondeo descentralizado como una alternativa ventajosa frente a los anillos lógicos, en cuanto a sencillez y en cuanto a confiabilidad.

Referencias

[ATA88] Atallah, Daif N.: "Peer to peer protocol facilitates real time communications", EDN, Agosto 1988, pp. 179-186, 1988.

[CRA89] Crane, Stephen; Tangney, Brendan y Moreau, John: "Enforcing Determinism in a CSMA/CD Local Area Network", Microprocessing and Microprogramming vol. 26, pp. 205-211, 1989.

[FRA81] Franta, W. R. y Chlamtac, Imrich: "Local Network", D.C. Heath, U.S.A., 1981

[HUT88] Hutchinson, David: "Local Area Networks Architectures", Addison-Wesley, U.K., 1989.

[IBM83] IBM: "IBM PC Technical Reference", IBM, U.S.A., 1983.

[INT85] Intel: "Microsystem Components Handbook", Intel, U.S.A., 1985.

[ISA85] Instrument Society of America: "PROWAY LAN Industrial Data Highway", U.S.A., 1985.

[SCH87] Schwartz, Mischa: "Telecommunication Networks. Protocols, Modeling and Analysis", Addison-Wesley, U.S.A., 1987.

[TAN81] Tanenbaum, Andrew S.: "Computer Networks", Prentice Hall, U.S.A., 1981.

[WD] Western Digital: "WD8250 Asynchronous Communications Element", WD, U.S.A., 1981.

[YAK83] Yakubaitis, Eduard: "Network Architectures for Distributed Computing", Allerton Press, U.S.A., 1983.

El jurado designado por la Sección de Computación del Departamento de Ingenieria Eléctrica del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, aprobó esta tesis el 22 de Junio de 1990.

Dr. Jan Janecek Hyan

Dr. Armando Maldonado Talamantes

M. en C. Carlos E. Hirsch Ganievich