ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

DISEÑO, IMPLEMENTACIÓN Y MODELADO DE UN PROTOCOLO MULTIPUNTO DE COMUNICACIONES PARA APLICACIONES DE TIEMPO REAL DISTRIBUIDAS Y ANALIZABLES SOBRE ETHERNET. (RT-EP)

Para acceder al Título de

INGENIERO DE TELECOMUNICACIÓN

José María Martínez Rodríguez

Septiembre - 2002

Índice

1. Introducción a las redes de TR 1

1.1. Comunicaciones de Tiempo Real 1
1.2. Ethernet en Tiempo Real 4
1.3. Antecedentes en la adaptación de Ethernet para TR 5
1.3.1. Modificación del Control de Acceso al Medio (MAC). 5
1.3.2. Añadir un controlador de la transmisión sobre Ethernet 6
1.3.3. Contención de tráfico. 7
1.3.4. Switched Ethernet. 7
1.4. Objetivos de este proyecto 8

2. Descripción del protocolo 11

2.1. Descripción general 11
2.2. RT-EP como una máquina de estados 12
2.3. Formato de los paquetes RT-EP 14
2.4. Tratamiento de errores en RT-EP 17
2.4.1. Descripción General. 17
2.4.2. Máquina de estados de RT-EP contemplando fallos. 19

3. Implementación de RT-EP 21

3.1. Introducción 21
3.2. Arquitectura del software del protocolo 22
3.3. Detalles de la implementación 23
3.3.1. Sockets en RT-EP 24
3.3.2. Temporizadores en RT-EP 27
3.3.3. Procesos ligeros o Threads 28
3.3.4. Sincronización de threads 29
3.3.5. Estructura de datos: Colas de prioridad (Heaps) 32
3.3.6. Módulos del protocolo 35
3.3.6.1. prio_element 35
3.3.6.2. prio_priority 37
3.3.6.3. rt_comm 37

3.3.6.4. prio_queue 39
3.3.6.5. prio_queue_monitor 40
3.3.6.6. lnk_config 41
3.3.6.7. martelnk 43
3.3.6.8. prot_types 45
3.3.7. Configuración del protocolo 45

4. Modelado MAST 51

4.1. Introducción a MAST 51
4.2. Modelo MAST de RT-EP 55
4.2.1. Caracterización MAST de RT-EP sin tratamiento de errores 55
4.2.2. Caracterización MAST de RT-EP con tratamiento de errores 61

5. Medidas de prestaciones 64

5.1. Introducción a las pruebas sobre RT-EP 64
5.2. Cálculo de los *Overhead* de CPU 65
5.3. Cálculo del *Packet Overhead*. 67
5.4. Cálculo del *Max Blocking*. 67

6. Conclusiones 70

7. Trabajo futuro 73

7.1. Introducción 73
7.2. Implementación de RT-EP en MaRTE 73
7.3. Tráfico *Multicast*. 73
7.4. Interconexión de redes 75
7.5. Implementación en Wireless LAN 77

Bibliografía y referencias 79

Índice de figuras 82

1. Introducción a las redes de TR

1.1. Comunicaciones de Tiempo Real

Antes de comenzar con el desarrollo del trabajo, es necesario introducir una serie de conceptos importantes en el mundo del tiempo real. De este modo dejaremos claras las motivaciones de este trabajo. Primeramente procederemos a definir un sistema de tiempo real:

> *Un sistema de tiempo real es un sistema de procesamiento de información el cual tiene que responder a estímulos de entrada generados externamente en un intervalo finito y específico.*

De la definición es inmediato concluir que en un sistema de tiempo real las respuestas correctas dependen no sólo de los resultados sino también del instante en que se generan y que el hecho de no responder a tiempo es tan malo para el sistema como una mala respuesta.

En este tipo de sistemas, las tareas deben ser activadas y terminadas antes de su plazo (*deadline*). Desde un punto de vista de pérdida de plazo global tenemos dos clasificaciones:

- **Tiempo real no estricto (*soft real-time*):** Cuando la perdida de un deadline o de un número de ellos no es muy importante, aunque el sistema debe cumplir, en promedio, un cierto porcentaje de veces con su plazo.
- **Tiempo real estricto (*hard real-time*):** Cuando la pérdida de un deadline es crítica y provoca el fallo del sistema.

Una característica distintiva de un sistema en tiempo real es la *predecibilidad*. Esta peculiaridad está asociada a la posibilidad de demostrar o comprobar a priori que los requerimientos temporales de un sistema se cumplen en cualquier circunstancia. Como consecuencia, la predecibilidad implica:

- Una cuidadosa planificación de tareas y recursos.
- Cumplimiento predecible de requisitos temporales.
- Anticipación a fallos, y sus requerimientos temporales.
- Consideraciones de sobrecargas: degradación controlada.
- Consideraciones de elementos de impredecibilidad.
- Dotar al sistema con capacidades de monitorización y control de tiempos (hardware, software, sistema operativo, lenguaje, líneas y protocolos de comunicaciones).

A vista de estas consideraciones podemos afirmar que el éxito de usar un sistema distribuido para una aplicación de Tiempo Real reside en la ejecución oportuna de tareas que normalmente residirán en distintos nodos y comunicarlas para que sean capaces de

desarrollar su labor. Es, por tanto, difícil o imposible asegurar resultados temporalmente acotados en sistemas sin una red de comunicaciones que tenga capacidades de Tiempo Real. Estas redes tienen que ser capaces de respetar las restricciones temporales de los distintos tipos de tareas que se puedan presentar.

A continuación enumeramos distintas características que pueden estar asociadas a las tareas:

Por su plazo pueden ser de:

- **Tiempo Crítico:** Las tareas deben completarse antes de su plazo de respuesta.
- **Acríticas:** Las tareas deben completarse tan pronto como sea posible.

En base a su comportamiento en el tiempo tenemos:

- **Tareas Periódicas:** Corresponden con la reiniciación periódica de tareas, cada instancia debe completarse antes de su plazo.
- **Tareas Aperiódicas:** Se activan una sola vez o a intervalos irregulares.

Las aplicaciones en entornos de control de procesos y fabricación industrial presentan unas restricciones de Tiempo Real. No sólo hay que procesar la información a tiempo sino también los resultados deben generarse en su plazo.

Durante las dos últimas décadas se han desarrollado una serie de redes de comunicaciones de propósito específico para que proporcionen la calidad de servicio adecuada a nivel de campo (interconexión de sensores, actuadores y controladores). Estas redes, generalmente llamadas buses de campo (*fieldbuses*), están adaptadas para soportar frecuentes intercambios de pequeñas cantidades de tráfico respetando plazos y prioridades [37].

Las redes de área local (LAN) son las más usadas con aplicaciones distribuidas. Estas redes están limitadas geográficamente a un edificio o un grupo de éstos, un barco, un avión, un coche etc. [25]. En la figura 1.1, y a modo de ilustración, ponemos un ejemplo de una red de área local en un entorno industrial.

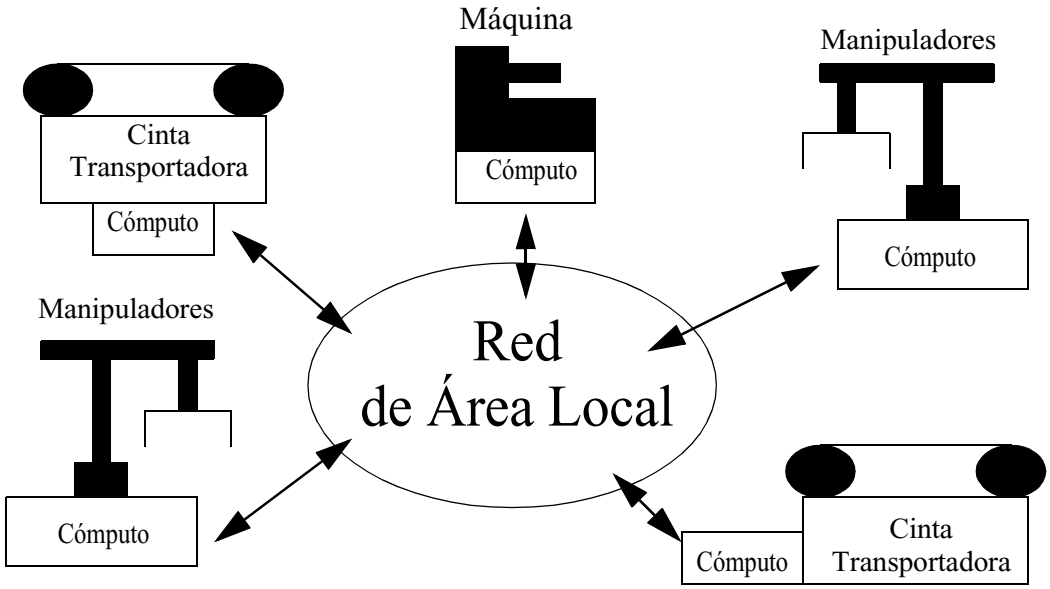


Figura 1.1: Sistema de computo en entorno industrial

Debido a que la mayoría de las aplicaciones distribuidas no tienen requisitos de Tiempo Real estricto, las redes están diseñadas de acuerdo a otras prioridades. En redes sin restricciones de Tiempo Real el diseño se centra en maximizar el *throughput* del mensaje y minimizar el retardo medio [36]. En cambio, una red diseñada para soportar tráfico de Tiempo Real debe considerar las restricciones temporales de cada mensaje. Se favorece la predicibilidad frente al *throughput* medio ya que la mayor consideración de diseño debe ser asegurar que se respetan los plazos de cada mensaje [20].

A continuación describimos algunos de los *fieldbuses* más usados actualmente:

- **PROFIBUS (Process Field Bus)[5]:** Es la propuesta alemana para un bus de campo. La estructura del protocolo sigue el modelo reducido ISO/OSI (basado en las capas 1,2 y 7). La capa de enlace define un modelo lógico de nodos maestros y esclavos. Está dividida en dos subcapas, una de control de acceso al medio y otra que proporciona una interfaz a niveles superiores con servicios de transmisión síncronos / asíncronos. Básicamente es un protocolo de paso de testigo que como característica importante destaca que carece del reparto síncrono del ancho de banda. Aun así puede tener capacidades de Tiempo Real [38] dependiendo del perfil del tráfico. Implementa dos categorías de mensajes: de alta y baja prioridad. Estas dos categorías de mensajes usan dos colas de salida independientes. En el peor caso, Profibus transmite un único mensaje de alta prioridad en la posesión del testigo [38]. Profibus presenta tres interfaces de usuario:
  - **PROFIBUS FMS** (Fieldbus Message Specification) que ofrece servicio de usuario estructurado para la comunicación abierta en pequeñas células. En esta configuración, lo principal es el voluminoso intercambio de información y no el tiempo de respuesta de los mismos.
  - **PROFIBUS DP** (Decentral Periphery) es la interfaz de usuario para el acoplamiento de dispositivos de campo. Utilizada para aplicaciones de bajo costo en redes sensor/actuador. Es especial para transmisión de mensajes cortos transferidos a alta velocidad.
  - **PROFIBUS PA** (Process Automation) que se utiliza para la automatización de procesos en recintos expuestos al peligro de explosiones (áreas clasificadas).
- **WorldFIP/FIP (Factory Instrumentation Protocol)[5]:** Es una propuesta franco-italiana para un bus de campo. La estructura sigue el modelo reducido ISO/OSI con distinto control de acceso. Proporciona comunicación periódica y aperiódica. FIP funciona como una base de datos distribuida donde todos los datos son recogidos y encaminados por un solo 'distribuidor', el acceso al medio es centralizado. Este protocolo se suele utilizar para conectar PCs, sensores y actuadores en plantas de energía, automoción etc. Una evolución del FIP es el WordFIP el cual añade mejoras sobre el primero. WorldFIP presenta dos tipos de servicios básicos de transmisión: el intercambio de variables y el intercambio de mensajes. Solo los servicios de intercambio de variables son relevantes para el tráfico de tiempo real. WorldFIP puede asegurar comportamiento de tiempo real para tráfico periódico de forma sencilla. En el caso de tráfico no periódico se debe efectuar un análisis más complejo [39].

- **P-NET (Process Network) [5]:** Es un estándar "multi-maestro" basado en el esquema de paso de testigo virtual. En P-NET todas las comunicaciones están basadas en el principio de mensaje cíclico, es decir, una estación maestra manda peticiones a una estación esclava y ésta contesta inmediatamente. El estándar P-NET permite a cada estación maestra que procese al menos un mensaje por visita de testigo. No esta muy estudiado para ofrecer capacidades de Tiempo Real.
- **CAN bus (Controller Area Network)[4]:** Es un bus de campo propuesto por Bosch para la fabricación de automoción y para el control de motores. CAN sigue el modelo reducido de ISO/OSI con una topología de bus, en un medio de par trenzado y con un modelo de comunicación cliente/ servidor. Es un protocolo basado en prioridades que evitan las colisiones en el bus mediante CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) como método de acceso al medio. La resolución de las colisiones es no destructiva en el sentido de que siempre se transmitirá un mensaje.
- **Interbus-S[6]:** Es un sistema de comunicaciones serie digital propuesto por Phoneix Contact. Es un protocolo de paso de testigo. Provee conexión para actuadores y sensores con una topología de árbol. Se interconectan con un bus primario (llamado REMOTE) y varios buses secundarios (llamados LOCAL). Interbus-S está recomendado para sistemas con un gran número de unidades sin prioridades.

1.2. Ethernet en Tiempo Real

Ethernet es, con diferencia, la tecnología LAN más usada hoy en día. Fue inventada por Bob Metcalfe en 1973 en el centro de desarrollo de Xerox en California para conectar un PC a una impresora láser. Posteriormente fue estandarizada como IEEE 802.3 con pequeñas diferencias frente a la especificación original. Su velocidad de transmisión ha pasado de los originales 2.94 Mbps a 10 Mbps, luego a 100 Mbps, 1 Gbps [14] y recientemente 10 Gbps (*www.ieee.org*). A pesar de los grandes cambios en Ethernet en estos años dos propiedades fundamentales se han mantenido inalterables:

- Un solo dominio de colisión. Las tramas son transmitidas a un medio común para todos los nodos; así las NICs (Network Interface Card) que escuchan el medio pueden recibirlas.
- El algoritmo de arbitraje. Denominado CSMA/CD (Carrier Sense Multiple Access with Collision Detection). De acuerdo con este mecanismo una NIC que quiera transmitir deberá esperar a que el bus esté libre, es decir, que no haya nadie trasmitiendo (esa es la parte CS del acrónimo). Cuando eso ocurre comienza la transmisión al medio (esa es la parte MA ya que el medio es compartido por todas las NICs). Si dos NICs comienzan a transmitir a la vez se produce una *colisión.* Si se produce es detectada (parte CD) y todos los nodos que estaban trasmitiendo se detienen y esperan un tiempo aleatorio antes de volver a transmitir. Éste proceso continuará hasta 15 reintentos.

Existe una serie de pros y contras a la hora de usar Ethernet a nivel de campo [9]. Los principales argumentos a favor son:

- Es una tecnología barata debido a la producción en masa.
- La integración con Internet es fácil.
- Su velocidad de transmisión ha ido incrementando con el tiempo y seguirá aumentando.
- El ancho de banda disponible actualmente en los buses de campo tradicionales es insuficiente para soportar las tendencias recientes como el uso multimedia a nivel de campo.
- Es un protocolo muy popular; muchos técnicos están familiarizados con él.
- Existe una amplia gama de herramientas software y hardware de testeo.
- Es una tecnología madura, bien especificada y abierta. Existen muchos fabricantes sin problemas de incompatibilidad.

En cambio la tecnología Ethernet no cumple con algunos requerimientos fundamentales que se esperan de un protocolo que funcione a nivel de campo:

- No garantiza un tiempo de transmisión acotado.
- No indica el límite temporal de los mensajes.
- No soporta distinto tipo de tráfico (periódico, esporádico etc.)

1.3. Antecedentes en la adaptación de Ethernet para TR

Se han hecho mucho intentos para adaptar Ethernet a un comportamiento de Tiempo Real. En esta sección mostramos una clasificación y caracterización de los trabajos más relevantes en este campo.

1.3.1. Modificación del Control de Acceso al Medio (MAC).

Esta aproximación consiste en modificar la capa MAC de Ethernet de manera que se consigan tiempos de acceso al bus acotados. Trabajos a este respecto están reflejados en [19], [22], [8] y [33]. Hacemos hincapié especial en los trabajos [19] y [22]:

- En [19] se propone e implementa una nueva arquitectura de red combinando el método de acceso token-bus (IEEE 802.4) y la capa física de IEEE 802.3. De esta manera consigue una red token-bus usando tecnología Ethernet. Implementando esta técnica se consiguen regímenes binarios de 5 Mbps sobre un medio Ethernet de 10 Mbps.
- En [22] la solución (CSMA/DCR) consiste en crear una jerarquía de prioridades en forma de árbol binario. Si se produce una colisión, los nodos de menor prioridad no reintentan la trasmisión voluntariamente. En cambio, los de mayor prioridad sí reintentan la trasmisión. Este proceso es repetido hasta que ocurra una trasmisión correcta. Un gran inconveniente de esta solución es que en los sistemas de Tiempo Real es más importante asignar prioridades a los mensajes, y no a los nodos, ya que en un mismo nodo pueden coexistir mensajes muy urgentes con otros sin urgencia.

Estas soluciones tienen dos principales inconvenientes:

- Modifican el *hardware* o el *firmware* del dispositivo y, por lo tanto, ya no es una solución económica o al menos no tan económica como Ethernet.

- El tiempo de transmisión del peor caso (*worst-case*), que es el principal factor considerado cuando se diseñan sistemas de Tiempo Real, puede ser de varios órdenes de magnitud superior que el tiempo medio de trasmisión. Esto fuerza a cualquier tipo de análisis a ser muy pesimista lo que conlleva a la infra-utilización del ancho de banda.

1.3.2. Añadir un controlador de la transmisión sobre Ethernet

Si se añade una capa por encima de Ethernet que intente controlar los instantes de las trasmisiones de los mensajes, se consigue un número acotado de colisiones o su completa desaparición. Las distintas aproximaciones en esta sección se pueden clasificar en:

- **Maestro/Esclavo:** En este caso, todos los nodos en el sistema trasmiten mensajes sólo cuando reciben un mensaje especial por parte de un nodo señalizado como Maestro. Esta propuesta es ineficiente para el tráfico asíncrono ya que el nodo maestro debe darse cuenta de la petición de transmisión antes de interrogar al nodo que quiere transmitir. Una solución a este problema es el protocolo propuesto por [29] que se basa en una estructura de control de transmisión maestro / esclavo combinado con una política de planificación centralizada. Aun así no soluciona el problema de una importante inversión de prioridad, esto es, que un mensaje de alta prioridad esté esperando hasta que se transmita uno de baja prioridad cuando lo deseable en este tipo de sistemas es lo contrario.
- **Paso de testigo:** Este método consiste en que un testigo circule a través de todas los nodos. Sólo el nodo que posea el 'token' o testigo puede transmitir. El tiempo de posesión del testigo está acotado. Esta solución es similar al método de acceso del IEEE 802.4 token-bus [15]. Este método no es muy eficiente debido al ancho de banda usado por el testigo, aunque obtiene muy buenos resultados cuando la carga de tráfico es elevada y su *throughput* es elevado debido a la ausencia de colisiones [35]. Introduce una larga latencia a las tareas periódicas debido a las variaciones en el tiempo de posesión del testigo. Las pérdidas de testigo, generalmente, introducen largos períodos de inaccesibilidad en el bus. Además también se pueden producir grandes inversiones de prioridad.
- **Virtual Timed-Token:** Esta técnica está basada en el paso de testigo y es la base para el protocolo RETHER [40]. Es capaz de manejar tráfico de Tiempo Real y tráfico normal. Operando en tiempo real divide la red en dos grupos: uno de tiempo real (RT) y otro de 'no tiempo real' (NRT). Los mensajes de Tiempo Real son considerados como periódicos y el acceso al canal por ambos tipos de tráfico está regulado por un testigo que se transmite por los nodos RT y si sobra tiempo en el ciclo de circulación del testigo, visita también el grupo NRT. Sus inconvenientes son similares a el método maestro/esclavo.
- **TDMA:** En este caso los nodos transmiten en unos determinados instantes temporales previamente asignados y de manera cíclica. Esta aproximación requiere una precisa sincronización entre todos los nodos. Cualquier cambio en la temporalidad de los mensajes ha de hacerse globalmente. Un punto a favor es que es bastante eficiente desde el punto de vista de utilización del ancho de banda.

- **Protocolo de tiempo virtual (Virtual Time Protocol):** Estos protocolos [26] intentan reducir el número de colisiones en el bus mientras que ofrecen la flexibilidad de ofrecer distintas políticas de planificación. Cuando un nodo quiere transmitir un mensaje espera una cantidad de tiempo desde el momento en el que el bus está inactivo. Este tiempo depende de la política de planificación elegida. Cuando expira, y si el bus sigue estando inactivo, comienza a transmitir. Si ocurre una colisión los nodos involucrados pueden, o bien volver a retransmitir el mensaje esperando un tiempo de probabilidad p o bien volver a esperar otra cantidad de tiempo similar. Una gran pega de este método es que el rendimiento depende en gran medida del tiempo que esperen las estaciones antes de poder transmitir el mensaje. Esto conduce a colisiones si el tiempo es muy corto o a largos períodos de inactividad en el bus si el tiempo es muy largo.

1.3.3. Contención de tráfico.

Contrariamente al control de transmisión, esta técnica se basa en que si la utilización del bus se mantiene baja, entonces la probabilidad de colisiones también se mantiene baja aunque no cero. Por lo tanto, si la carga media de la red se mantiene por debajo de un umbral y se evitan las ráfagas de tráfico, se puede obtener una determinada probabilidad de colisión. Una implementación de esta técnica se puede ver en [21] en donde se propone una capa de adaptación entre la capa de transporte (TCP/UDP) y Ethernet. En esta capa el tráfico de tiempo real es atendido bajo demanda mientras que el tráfico de tiempo no real es considerado a ráfagas y es filtrado. Un gran inconveniente de esta solución es que se trata de un método estadístico por lo que no garantiza a priori que un mensaje sea entregado antes de un determinado periodo.

1.3.4. Switched Ethernet.

El uso de switches se ha hecho muy popular recientemente. Los switches proveen un dominio de colisión exclusivo para cada uno de sus puertos ya que no existe conexión directa entre ellos. Cuando un nodo transmite un mensaje, éste es recibido por el switch y puesto en los bufferes correspondientes a los puertos donde están conectados los receptores del mensaje. Si varios mensajes son transmitidos a la vez, o en un corto intervalo de tiempo, a un mismo puerto son almacenados en el buffer del puerto y transmitidos secuencialmente. Acerca de la política de planificación sobre los mensajes que esperan en un determinado puerto, algunos switches implementan el estándar IEEE 802.1D que especifica un campo de prioridad en la trama de Ethernet y es posible disponer de 8 colas en cada puerto (implementa 8 prioridades). El uso de un *switch* en una red Ethernet no es suficiente para hacerla de Tiempo Real. Por ejemplo, los bufferes se pueden desbordar y perderse los mensajes si se transmiten ráfagas de datos a un mismo puerto. Otro problema es que en una *switched Ethernet* no se pueden mandar mensajes a direcciones *multicast*. En *Shared-Ethernet*, para dar soporte a este tipo de tráfico, cada NIC define una tabla local con las direcciones *multicast* relativas a los datos que debería recibir. El *switch* no tiene constancia de estas tablas así que no las puede aplicar. Esta situación es bastante común en modelos productor/consumidores (como un sensor). Como conclusión diremos que un *switch* hace Ethernet determinista solo con cargas de tráfico controladas. Aunque también hay que indicar que este es un campo todavía muy abierto a la investigación.

También se han hecho intentos de modificar el hardware del *switch* básico para que, añadiéndole más bufferes e inteligencia, sea capaz de ser totalmente determinista [24].

1.4. Objetivos de este proyecto

Una vez que hemos repasado las comunicaciones de Tiempo Real y los antecedentes del uso de Ethernet como *fieldbus*, pasaremos a comentar los objetivos de este trabajo.

El Departamento de Electrónica y Computadores ha estado trabajando en los últimos años en la creación y desarrollo del sistema operativo MaRTE OS [1] (Minimal Real-Time Operating System for Embedded Applications) que es un núcleo de Tiempo Real para aplicaciones empotradas que sigue las recomendaciones de Tiempo Real mínimo de la especificación POSIX.13 [30] y que provee las interfaces POSIX para los lenguajes C y Ada.

Los sistemas operativos POSIX actuales son comerciales y no proporcionan su código fuente y las implementaciones de código abierto existentes actualmente, como RTEMS [31], no han sido diseñados siguiendo el modelo POSIX desde el principio sino mas bien solo ofrecen la interfaz externa. Otra alternativa como RT-Linux [41] que ofrece actualmente parte de la interfaz de Tiempo Real mínimo POSIX.13 no es apto para sistemas empotrados pequeños ya que requiere el apoyo del sistema operativo Linux completo.

Debido a estas razones el grupo de investigación del departamento decidió diseñar e implementar un kernel de Tiempo Real para aplicaciones empotradas que pudiera ser utilizado en distintas plataformas, incluyendo microcontroladores, y que siguiese el subconjunto POSIX.13 de Tiempo Real mínimo. Este kernel, llamado MaRTE OS, se puede utilizar para el desarrollo de aplicaciones de Tiempo Real, como controladores de robots, y también como una herramienta de investigación de sistemas operativos y de mecanismos de planificación. Además, es una prueba de que un sistema operativo POSIX puede ser implementado en Ada, proporcionando una interfaz de lenguaje C y permitiendo que aplicaciones *multithread* en C se ejecuten sobre él [1].

MaRTE permite el desarrollo cruzado de aplicaciones de Tiempo Real en C y Ada utilizando los compiladores GNU *gnat* y *gcc*. La mayor parte del código está escrito en Ada con algunas partes en C y en ensamblador. Las aplicaciones *multithread* en C pueden utilizar el kernel de Ada a través de una interfaz de lenguaje C que se ha creado sobre el kernel.

El objetivo general del presente trabajo es diseñar e implementar un protocolo de comunicaciones de Tiempo Real para MaRTE OS.

El uso de este kernel está destinado para sistemas empotrados industriales como pueden ser sistemas de adquisición de datos y controladores de robots. Por tanto, es necesario que tanto el kernel como la red de comunicaciones cumplan con una serie de requerimientos exigidos por este tipo de ámbitos:

- Las aplicaciones serán estáticas, es decir, el número de *threads* y de recursos del sistema son conocidos en tiempo de compilación. Esto hace que los recursos, como pueden ser *threads*, *mutexes*, *timers*, etc., sean preasignados en tiempo de configuración lo que hará que se ahorre mucho

tiempo cuando la aplicación requiera la creación de alguno de estos objetos.

- El ámbito de la red de comunicaciones será una red de área local (LAN) ya que estará limitada geográficamente, como mucho, a un edificio o a un grupo de estos.
- Esta red constará de un número conocido y no muy extenso de nodos o estaciones, se pueden limitar a uno o varios centenares.
- La red de comunicaciones debe ofrecer un comportamiento de Tiempo Real. Esto significa que debemos ser capaces de acotar temporalmente cualquier respuesta de ésta.

Una vez que se ha expuesto el ámbito del trabajo y sus requisitos procederemos a comentar los objetivos concretos de este trabajo:

- Diseñar un protocolo de comunicaciones de Tiempo Real basado en prioridades para MaRTE.
- Conseguir una red de comunicaciones de alta velocidad a un bajo coste manteniendo el comportamiento predictivo requerido en las aplicaciones de Tiempo Real.
- Implementar el protocolo en GNU/Linux de manera que sea fácilmente portable a MaRTE.
- Modelar el protocolo para que se pueda analizar con la aplicación.
- Evaluar el funcionamiento del protocolo de acuerdo a su modelo.

Ethernet es una red rápida (cada vez más) y barata, pero como ya se ha comentado con anterioridad no garantiza un comportamiento de Tiempo Real debido a que el mecanismo de arbitraje no es determinista (CSMA/CD). En este trabajo diseñamos e implementamos un protocolo de paso de testigo sobre Ethernet basado en prioridades fijas con lo que conseguimos predecibilidad sobre Ethernet sin ningún cambio adicional sobre el hardware. Llamamos a este protocolo RT-EP (Real Time Ethernet Protocol) [27].

Comparando RT-EP con Token bus [15], protocolo que podemos considerar como una referencia en cuanto a la técnica de paso de testigo en bus, podemos afirmar que:

- RT-EP aporta una nueva visión del paso de testigo que soluciona sus grandes problemas clásicos ya que reduce la larga latencia en los mensajes de alta prioridad, aumenta el procesado de los mensajes asíncronos y reduce considerablemente la inversión de prioridad que se da en el paso de testigo.
- Implementa más prioridades a los mensajes, 255 frente a 4. Token bus implementa 4 subestaciones (con prioridades 0, 2, 4, 6) dentro de la propia estación para almacenar los mensajes a transmitir / recibir.
- RT-EP es un protocolo estático lo que le hace muy indicado para este tipo de sistemas empotrados frente al dinamismo que presenta Token bus que pueden dar paso a instantes de colisiones controladas o pérdida de testigo además de retrasos, también controlados, en el paso de testigo.
- La perdida de testigo no introduce un largo periodo de inaccesibilidad en el bus como en Token bus

Se pretende conseguir un ancho de banda efectivo y latencia similar a CANbus para mensajes de tamaño medio, unos 100-200 bytes, que son los que se necesitan en este tipo de sistemas. Haciendo algunos números para estimar las prestaciones de CANbus, tenemos que la máxima velocidad que se obtiene, hasta una distancia de 40 metros, es de 1 Mbps. Los datos son troceados en paquetes de 8 bytes y encapsulados en una trama con 46 bits de overhead. Ethernet, en cambio, tiene como mínimo una velocidad de 10 Mbps, como máximo de 10 Gbps (por el momento), y que, dependiendo del medio, podemos llegar hasta una distancia de dos kilómetros por segmento y un campo de datos de hasta 1.500 bytes. Con RT-EP se reduce drásticamente el rendimiento de Ethernet por el proceso del paso de testigo. Pero aún así y como se expondrá más adelante supone una alternativa más que válida para CANbus y otros *fieldbuses* de Tiempo Real.

La metodología utilizada en la implementación de RT-EP está basada en *sockets* sobre Linux. Se pretende implementar el protocolo en el espacio de usuario a una prioridad mayor que el resto de los *threads* de aplicación ejecutados en el sistema. Debido a la necesidad de trabajar directamente sobre las tramas Ethernet, se ha utilizado una familia especial de *sockets* denominada *packet sockets* que es capaz de recibir y transmitir directamente tramas, es decir, opera a nivel de enlace. Dentro de esta familia se ha elegido el tipo RAW para poder tener un control total sobre la cabecera de las tramas. Se pretende que la implementación del protocolo sea lo más directa posible para conseguir una mayor eficiencia.

Es necesario aclarar que la implementación no se ha realizado sobre MaRTE debido a que todavía no están preparados los drivers de red para este sistema operativo. Se ha elegido GNU/Linux para su desarrollo ya que supone un entorno de programación afín. Cuando se dote a MaRTE de unos drivers de red y de una mínima interfaz de *sockets* será posible utilizar la implementación actual de RT-EP sin apenas modificaciones en su código ya que, a la hora de su desarrollo, se ha prestado especial atención a esta futura transición.

También es importante señalar que RT-EP no se trata de un protocolo específico para MaRTE sino que serviría, con las modificaciones adecuadas en el código, para cualquier sistema operativo de Tiempo Real que cumpla con los requerimientos de los sistemas empotrados industriales descritos anteriormente.

En esta memoria primero veremos el diseño del protocolo dando paso a su implementación, seguida del modelado del mismo. Luego se expondrán las medidas de prestaciones que darán paso a las conclusiones del trabajo. Para finalizar se expondrá el trabajo futuro que puede derivar de este proyecto.

2. Descripción del protocolo

2.1. Descripción general

RT-EP ha sido diseñado para evitar las colisiones en Ethernet por medio del uso de un testigo que arbitra el acceso al medio compartido. Implementa una capa de adaptación por encima de la capa de acceso al medio en Ethernet que comunica directamente con la aplicación. Cada estación (ya sea un nodo o CPU) posee una cola de transmisión y varias de recepción. Estas colas son colas de prioridad donde se almacenan, en orden descendente de prioridad, todos los mensajes que se van a transmitir y los que se reciben. En el caso de tener mensajes de la misma prioridad se almacenan con un orden FIFO (First in First Out). El número de colas de recepción puede ser configurado a priori dependiendo del número de *threads* (o tareas) en el sistema que necesiten recibir mensajes de la red. Cada *thread* debe tener su propia cola de recepción. La aplicación debe asignar un canal, denominado *channel_id,* a cada *thread* que requiera comunicación a través de la red. Este *channel_id* se utiliza para identificar los extremos de la comunicación (las tareas o *threads*), es decir, proporciona el punto de acceso a la aplicación.

La red está organizada en un anillo lógico sobre un bus como se muestra en la figura 2.1. Existe un número fijo de estaciones que serán las que formen la red durante la toda la vida del sistema. La configuración de las estaciones se hace sobre su dirección MAC de 48 bits. Cada estación conoce quién es su sucesora, información suficiente para construir el anillo. Además todas las estaciones tienen acceso a la información de configuración del anillo.

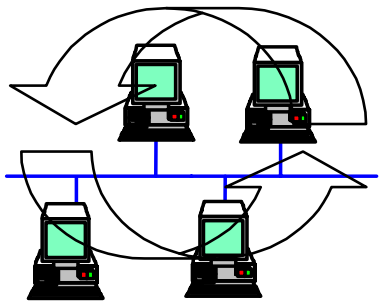


Figura 2.1: Anillo lógico

El protocolo funciona circulando un testigo a través del anillo. Este testigo contiene la dirección de la estación que tiene el paquete de mayor prioridad en la red en ese momento. También almacena la prioridad de ese paquete. Existe una estación, llamada *token_master,* que crea y controla la circulación del testigo. La asignación de esta estación es dinámica. Otra característica importante del protocolo es que, por sencillez, no admite fragmentación de mensajes, por lo tanto, es la aplicación la responsable de fragmentar los

posibles mensajes de tamaño superior al máximo permitido. Podemos dividir el comportamiento de la red en dos fases, la primera corresponde a la fase de arbitrio de prioridad y la segunda corresponde a la transmisión del mensaje:

Fase de arbitrio de prioridad:

- La estación con el papel de *token_master* se encarga de crear un testigo con la información de prioridad de su cola de transmisión y lo manda a su sucesora. El testigo viajará por todo el anillo visitando todas las estaciones.
- Cada estación comprueba la información de prioridad contenida en él. Si tiene algún mensaje de mayor prioridad en su cola de transmisión que la contenida en el testigo, actualiza el testigo con dicha información.
- Se manda el testigo a la estación sucesora en el anillo.
- Se prosigue con el proceso hasta que el testigo llega a la estación *token_master*. En este momento la estación *token_master* conoce quién contiene el mensaje de mayor prioridad de toda la red.

Fase de transmisión del mensaje:

- La estación *token_master* manda un mensaje especial, *Transmit_Token*, a la estación que tiene, según la información del testigo, el mensaje de mayor prioridad de la red en ese momento. Este mensaje especial da permiso a la estación para que pueda transmitir el mensaje de mayor prioridad encolado.
- La estación que recibe el mensaje especial puede transmitir y envía el paquete de información, *Info_Packet*, de mayor prioridad de su cola de transmisión a la destinataria.
- La estación destinataria del mensaje se convierte en la nueva *token_master* que se encargará de crear otro testigo y de circularlo.

Como se ha comentado anteriormente no se permite la fragmentación a este nivel. El mensaje máximo permitido está en *1492 bytes* como se ve en 2.3, “Formato de los paquetes RT-EP”.

El protocolo ofrece 256 prioridades. La prioridad 0, la más baja, se reserva para uso interno del protocolo y por lo tanto proporciona 255 prioridades para los mensajes de la aplicación.

Conviene señalar que, para un mejor entendimiento del funcionamiento del protocolo, realizaremos su descripción en dos fases: la primera sin considerar el tratamiento de los posibles errores que se puedan producir y la segunda describiendo el protocolo junto con el tratamiento de errores.

2.2. RT-EP como una máquina de estados

RT-EP puede describirse como una máquina de estados para cada estación. De esta manera podremos entender de una forma más sencilla la funcionalidad del protocolo. La figura 2.2 nos describe la máquina de estados y sus transiciones que serán comentadas seguidamente:

- **Offline:** Corresponde al estado inicial de cada estación. Cada una lee la configuración del anillo y realiza los distintos ajustes que la permitirán mantener una comunicación a través del anillo. Inicialmente la primera estación del anillo es configurada como *token_master* que será la encargada de crear y distribuir el primer testigo. Ésta primera estación realizará la transición al estado *Send_Initial_Token* y las demás alcanzarán el estado *Idle*.
- **Idle:** La estación permanece escuchando el medio en busca de cualquier trama. Descarta, silenciosamente, aquellas tramas que no vayan dirigidas a ella. Cuando recibe una trama destinada a ella, la chequea para determinar el tipo de paquete que contiene. Si es un paquete de información la estación cambia al estado *Recv_Info*. Si es un paquete de testigo se puede cambiar a dos estados:
  - *Send_Info* si se recibe un testigo especial que es el que otorga a la estación permiso para transmitir información.
  - *Check_Token* que ocurre cuando se recibe un testigo normal.

.

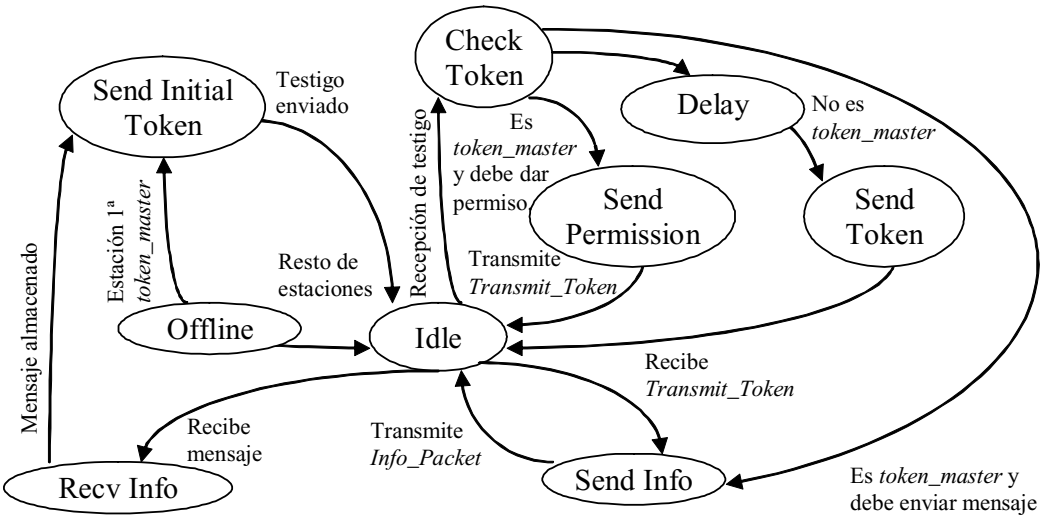


Figura 2.2: Máquina de estados de RT-EP para cada estación

- **Send_Initial_Token:** La estación que alcanza este estado pasa a ser la *token_master*. En este estado se confecciona un testigo en base a la información en la cola de transmisión propia y se manda a la estación sucesora en el anillo lógico. Una vez enviado, la estación pasa al estado *Idle*.
- **Check_Token:** Aquí la información que transporta el testigo recibido es comparada con la información del elemento de mayor prioridad de la cola de transmisión de la estación. Si la estación no es la *token_master* o la información de prioridad corresponde a la prioridad reservada 0, la estación alcanza el estado *Send_Token*. Si la estación es la *token_master* y además el campo *Station Address* del testigo contiene su dirección, es

decir, que la estación *token_master* tiene el mensaje de mayor prioridad para ser transmitido, conmuta al estado *Send_Info*. En caso contrario la estación se sitúa en el estado *Send_Permission*.

- **Send_Token:** En este estado, el testigo, después de ser actualizado, es mandado a la estación sucesora. A continuación la estación se traslada al estado *Idle*.
- **Delay:** Este estado únicamente introduce un retraso. Está motivado porque durante la fase de arbitrio de prioridad, incluso si no existe ningún mensaje que transmitir en la red, siempre se transmite un testigo de prioridad 0. Cuando una estación no tiene ningún mensaje que transmitir asume que tiene en cola un mensaje, que nunca llegará a transmitirse, de prioridad 0. Así logramos que en cuanto una estación tenga algún mensaje que transmitir sea negociada su transmisión lo antes posible. Conseguir este corto tiempo de respuesta conlleva un *overhead* de CPU ya que las estaciones están continuamente procesando testigos. Resulta conveniente, dependiendo de la aplicación y sus restricciones temporales, definir un tiempo en el que la estación "se duerma" mientras procesa el testigo para rebajar ese overhead de CPU. Solamente hay que tomar esta decisión en el paso de testigo ya que este *delay* es innecesario y puede llegar a ser perjudicial en el procesado del *Transmit Token* y el paquete de información (*Info Packet*). Este retardo es configurable en el protocolo como se verá más adelante.
- **Send_Permission:** En este estado la estación *token_master* pierde esta característica y construye un testigo especial llamado *Transmit_Token*. Este testigo es enviado a la estación que tiene el mensaje de mayor prioridad para darle permiso de transmisión. Después pasa al estado *Idle*.
- **Send_Info:** La estación que tiene permiso para transmitir lee la información del mensaje de mayor prioridad de la cola de transmisión, construye el paquete de información y lo transmite. Una vez transmitido el mensaje la estación alcanza el estado *Idle*.
- **Recv_Info:** La información recibida se escribe en la cola apropiada de recepción de la estación. A continuación alcanza el estado *Send_Initial_Token* y se convierte en la nueva *token_master*, que crea y hace circular un nuevo testigo.

2.3. Formato de los paquetes RT-EP

RT-EP se implementa como una capa de adaptación por encima de la capa de acceso al medio de Ethernet. Por lo tanto, los paquetes usados en este protocolo van encapsulados en el campo de datos de la trama Ethernet que tiene la siguiente estructura [7]:

8 bytes	**6 bytes**	**6 bytes**	**2 bytes**	**46-1500 bytes**	**4 bytes**
Preamble	*Destination Address*	*Source Address*	*Type*	*Data*	*Frame Check Sequence*

Figura 2.3: Formato de la trama Ethernet

Pasaremos a comentar brevemente cada uno de los campos. Para más información conviene recurrir a la bibliografía [7]:

- **Preamble:** Preámbulo, es un campo de 8 bytes necesario en las interfaces Ethernet de 10 Mbps para la sincronización de los datos entrantes. El preámbulo se mantiene en los sistemas *Fast Ethernet* y *Gigabit Ethernet* para proveer compatibilidad con la trama Ethernet original. Sin embargo estos sistemas no utilizan el preámbulo como medio para detectar datos en el canal sino que usan mecanismos más complicados de sincronización.
- **Destination Address:** Es una dirección de 48 bits que se suele llamar dirección hardware, física o también se puede llamar dirección MAC. Indica la dirección de la estación destino de la trama.
- **Source Address:** Al igual que con el campo anterior, este campo indica la dirección de la estación que origina la trama. Esta información no es interpretada por el protocolo de acceso al medio de Ethernet. Se provee para su posible uso en protocolos superiores.
- **Type:** Este campo identifica el protocolo de nivel superior que se transporta en el campo de datos. En el estándar IEEE 802.3 [14] se contempla que este campo pueda ser, en vez de un identificador del protocolo, un campo que identifica la longitud de la información que contiene la trama. Para poder distinguirlos, por norma, ningún protocolo puede tener un identificador decimal menor que 1500. Así si en el campo tipo vemos un valor numérico inferior a 1500 se tratará de la longitud de datos transportados por la trama. Si no, se referirá al protocolo de nivel superior.
- **Data:** Aquí se alojan todos los datos de niveles superiores.
- **Frame Check Sequence:** Este último campo es la secuencia de verificación de trama, también llamado chequeo de redundancia cíclica. El valor de 32 bits alojado en este campo se utiliza para determinar si la trama recibida es o no correcta.

En el campo tipo usamos un valor de *0x1000* para RT-EP, valor que no está asignado a ningún protocolo conocido. Usamos este número sin asignar para implementar el protocolo. Más adelante en caso de necesidad puede ser cambiado por si el protocolo se registrase.

Los paquetes del protocolo, como se ha comentado con anterioridad, se transportan en el campo de datos de la trama Ethernet. Este campo de datos tiene la restricción de 46 bytes como mínimo hasta un tope de 1500 bytes. Esta limitación se debe a que se necesita un tamaño mínimo en la trama Ethernet para que, de producirse, pueda ser correctamente detectada una colisión en toda la extensión de la red. Debido a esta restricción aunque algunos de nuestros paquetes sean menores de 46 bytes, como puede ser el testigo, un campo de datos de 46 bytes será construido por Ethernet, rellenando los bytes que falten.

A continuación mostraremos y comentaremos los dos tipos de paquetes utilizados en RT-EP. Antes de proceder a su descripción es necesario comentar que para el pleno entendimiento de todos los campos es necesario el apartado 2.4, “Tratamiento de errores en RT-EP”, donde se explican los métodos utilizados para la detección y corrección de los posibles errores contemplados en este protocolo.

Los paquetes del protocolo RT-EP son los siguientes:

Token Packet: Es el paquete utilizado para transmitir el testigo y tiene la estructura que se comenta a continuación:

1 byte	1 byte	2 bytes	6 bytes	2 bytes	6 bytes	6 bytes	22 bytes
Packet Identifier	*Priority*	*Packet Number*	*Token_Master Address*	*Failure Station*	*Failure Address*	*Station Address*	*Extra*

Figura 2.4: Formato del paquete de testigo

- **Packet Identifier:** Este campo de un solo byte, repetido también en el paquete de información, se utiliza para identificar el tipo de paquete. En el caso del testigo puede tomar los valores de:
  - **0x54** ('T') - Identifica el paquete como testigo normal. Este testigo es el empleado en determinar el mensaje de mayor prioridad de la red.
  - **0x50** ('P') - Identifica a un *Transmit Token* que, como ya se mencionó con anterioridad, es el testigo especial que otorga a una estación el derecho a transmitir.
- **Priority:** Campo de 1 byte que se utiliza para almacenar la prioridad del mensaje, que ocupa el primer lugar de la cola de transmisión, almacenado en la estación indicada en el campo *Station Address*. Un byte nos da 256 prioridades distintas, pero como la prioridad 0 esta reservada para uso interno del protocolo tenemos un total de 255 prioridades. Estudios teóricos [32] muestran que un número de prioridades de 255 es adecuado incluso para sistemas con una gran cantidad de tareas y plazos.
- **Packet Number:** Este campo de dos bytes sirve para proveer a los paquetes de un número de secuencia para poder descartar los paquetes duplicados. Para que funcione correctamente el número de secuencia, al ser un campo de 2 bytes, el número de estaciones configuradas en el anillo tiene que ser menor que el número máximo que se puede representar con 2 bytes, esto es 65535.
- **Token_Master Address:** Aquí se almacenará la dirección MAC de 48 bits correspondiente a la estación que ocupa el rol de *token_master* en la actual vuelta del testigo.
- **Failure Station:** Este entero de 2 bytes se activará para indicar que ha fallado alguna estación en el anillo, como se verá en el apartado 2.4, "Tratamiento de errores en RT-EP". Para el uso que se da a este campo con 1 byte (o menos) hubiese sido suficiente, pero debido a motivos de eficiencia de alineamiento de memoria y posibles futuros usos, se ha decidido usar 2 bytes.
- **Failure Address:** Aquí se almacenará, en caso de producirse fallo, la dirección MAC de la estación que ha fallado en una vuelta del testigo.
- **Station Address:** En este campo se guarda la dirección MAC de la estación que tiene el mensaje de la prioridad marcada por el campo de prioridad.
- **Extra:** Estos son los bytes de relleno necesarios para que el campo de datos de la trama Ethernet sea de 46 bytes que es el mínimo que se va a poder transmitir.

Info Packet: Es el paquete que RT-EP utiliza para transmitir los datos de la aplicación y tiene la estructura que sigue:

1 byte	1 byte	2 bytes	2 bytes	2 bytes	0-1492 bytes
Packet Identifier	*Priority*	*Packet Number*	*Channel ID*	*Info Length*	*Info*

Figura 2.5: Formato del paquete de información

- **Packet Identifier:** Este campo de 1 byte, como ya se ha comentado con anterioridad, identifica el tipo de paquete que en este caso tomará un valor de **0x49** ('I').
- **Priority:** Aquí, en 1 byte, se almacena la prioridad del mensaje transportado en el campo *Info*.
- **Packet Number:** Este campo de 2 bytes almacena el número de secuencia del paquete.
- **Channel ID:** Estos 2 bytes identifican el extremo destino de la comunicación, es decir, un punto de acceso creado para la recepción de mensajes en el nodo destino.
- **Info Length:** Aquí, en 2 bytes, se refleja la longitud del mensaje transportado en el campo *Info*.
- **Info:** Este último campo es donde se guardará el mensaje de la aplicación. El tamaño máximo del mensaje es de 1492 bytes. No existe mínimo aunque si el mensaje es menor de 38 bytes se efectuará un relleno de hasta esa cantidad para poder cumplir con el requisito de 46 bytes en el campo de información de la trama Ethernet.

2.4. Tratamiento de errores en RT-EP

2.4.1. Descripción General.

Se han tenido en cuenta los siguientes errores, como posibles, en el funcionamiento del protocolo:

- **Pérdida de un paquete:** Ya sea testigo o paquete de información, en este caso se producirá una retransmisión del paquete.
- **Fallo de una estación, que deja de responder:** En este caso se realizará una reconfiguración del anillo excluyendo a las estaciones en fallo.
- **Estación ocupada:** Una estación 'ocupada' es una estación que tarda más de lo normal en responder, lo que generará un paquete duplicado que deberá ser descartado.

Los dos últimos fallos, el caso de una estación ocupada y el de fallo de una estación, se consideran de bajísima probabilidad en un sistema de Tiempo Real bien diseñado, por lo que, aunque el protocolo sea capaz de recuperarse de esos errores, no se tendrán en cuenta para el comportamiento determinista del sistema, ya que podrían producirse colisiones. En cambio la perdida de un paquete produce una recuperación determinista y rápida.

No se ha tenido en cuenta el caso de dos o mas testigos circulando ya que, debido a la lógica del protocolo y a las condiciones tan controladas con que cuenta este tipo de sistemas de Tiempo Real, es una situación que no se puede (debe) dar.

Tampoco se ha tenido en cuenta el caso de fallo simultáneo de dos estaciones consecutivas, que podría producir, en el caso de estar involucradas en el paso de testigo en ese momento, el fallo total del protocolo. Este fallo es de bajísima probabilidad en el correcto funcionamiento de un sistema de Tiempo Real.

El método que se ha utilizado para solucionar estos posibles fallos es simple. Cada estación, una vez transmitido un paquete, escucha el medio durante un tiempo (*timeout*) configurable a priori. Si al cabo de ese tiempo no escucha ninguna trama válida en el medio, lo que supondría una confirmación (*acknowledge*) del último paquete enviado, supone que el último paquete enviado se perdió y necesita ser retransmitido. Realmente han podido ocurrir dos fenómenos:

- Que realmente se haya perdido (o corrompido) el paquete.
- Que la estación destino estuviera ocupada y no haya podido procesar todavía el paquete entregado.

De cualquier forma, la estación que estaba escuchando al ver que transcurrido ese *timeout* no ha habido una trama válida en el medio, retransmite el último paquete. Si la estación no hubiese recibido el primer paquete, aceptará éste como válido. Si lo que ocurría es que estaba ocupada, entonces descartará este último paquete mediante comprobación del número de secuencia. El número de secuencia es el campo *Packet Number* del paquete RT-EP. Toda estación descartará cada paquete que tenga un número de secuencia idéntico al último paquete procesado.

El proceso de reintento de la transmisión mediante el *timeout* se hace un número finito y configurable de veces. Transcurrido ese número de reintentos se asumirá que la estación no responde y se procederá a su exclusión del anillo lógico. La estación que detecta el fallo excluye a la que ha fallado de su configuración local del anillo y ya no admite más mensajes de la aplicación destinados a esa estación que ha fallado. Si la estación en fallo es su sucesora, modifica ésta por la siguiente en el anillo. Si la estación en fallo se ha detectado no en el circular del testigo sino a la hora de mandar la información o el testigo *Transmit Token*, descarta el paquete que ya no se puede entregar, asume el rol de *token_master* y comienza circulando un nuevo testigo. Un problema que se puede plantear es que la estación en fallo sea la *token_master*, cuya dirección está indicada en el campo pertinente del testigo. En este caso la estación que detectó el fallo pasa a asumir el papel de *token_master*. Así circulará el testigo o mandará el paquete *Transmit Token* según corresponda. Se ha añadido la figura 2.6 para ilustrar este proceso:

Una vez que una estación ha cambiado su configuración local deberá indicar a las demás estaciones que ha habido una que ha fallado la cual deberá ser excluida de las configuraciones locales de cada estación. De esta manera no admitirán ningún mensaje destinado a la estación en fallo. Esto lo hace cambiando el valor del campo *Failure Station* en el paquete de testigo de 0 a 1. Así comunica a las demás estaciones que la indicada por el campo *Failure Address* debe descartarse de la configuración local de cada estación. Esta etapa de descarte de estación se realiza durante el paso de testigo. Cuando el testigo vuelve a la estación que detectó el fallo vuelve a poner el campo *Failure Station* de 1 a 0 otra vez, pudiéndose comunicar nuevos fallos de equipos.

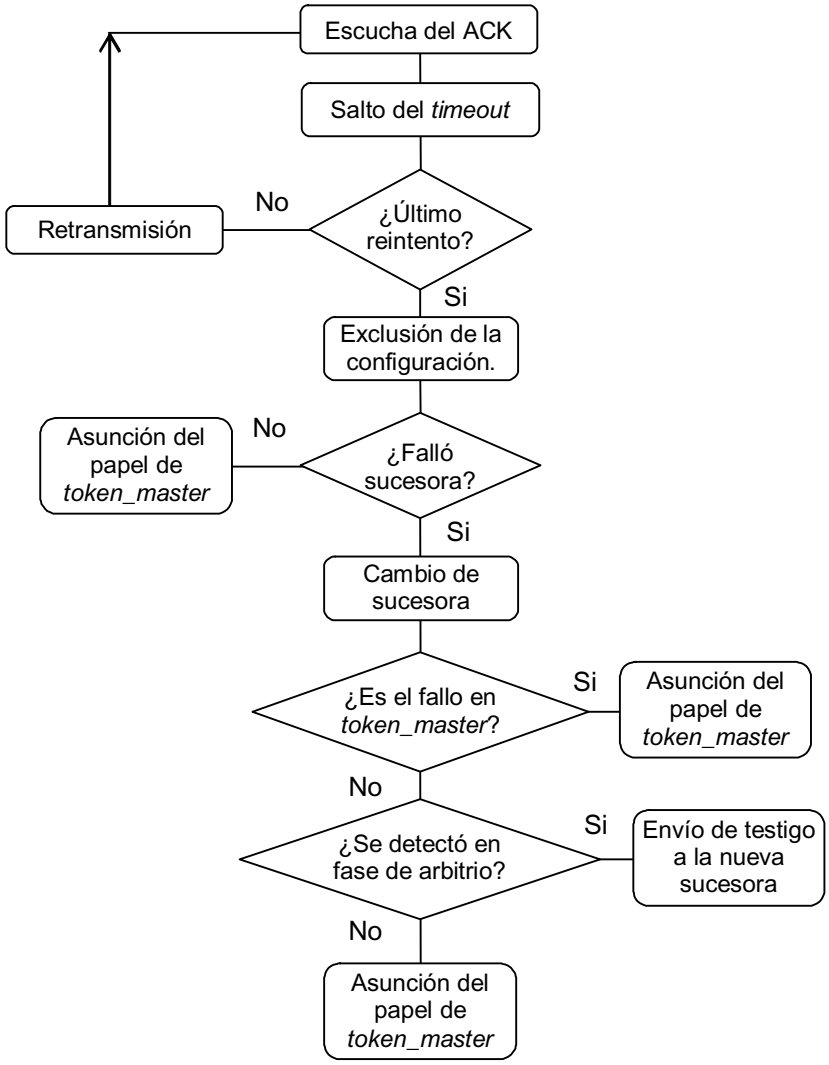


Figura 2.6: Comportamiento de la estación una vez detectado fallo

Una consideración especial la reciben aquellos mensajes destinados a una estación en fallo que ya estén encolados en las colas de prioridad. En este caso no se pueden rechazar y por tanto se intentarán transmitir, pero en la primera retransmisión (ya que si la estación no funciona no responderá a un supuesto paquete de información destinada a ella) se detecta que es una estación que ya había fallado antes y se ignora la acción. La razón para hacer la comprobación en la primera retransmisión se debe a que la condición de fallo de estación no es la habitual, sino más bien es una situación extrema. Así que se prefiere gastar un *timeout* de tiempo frente al *overhead* de CPU que supondría estar comprobando el destino de todos los mensajes salientes.

2.4.2. Máquina de estados de RT-EP contemplando fallos.

Al igual que en el apartado 2.2, "RT-EP como una máquina de estados", podemos describir el protocolo en su totalidad (incluyendo fallos) como una máquina de estados para cada estación, mostrada en la figura 2.7. Sólo tenemos que añadir un estado transitorio, *Error Check*, que realizará las operaciones de escucha comentadas anteriormente. Cada vez que una estación transmita un paquete, accederá a ese estado en

el cual se comprobará si el paquete ha sido entregado correctamente. Si se presenta un fallo daremos paso al estado *Error Handling* y si no volveremos al estado *Idle* como ya se hacía en la descripción del protocolo sin errores.

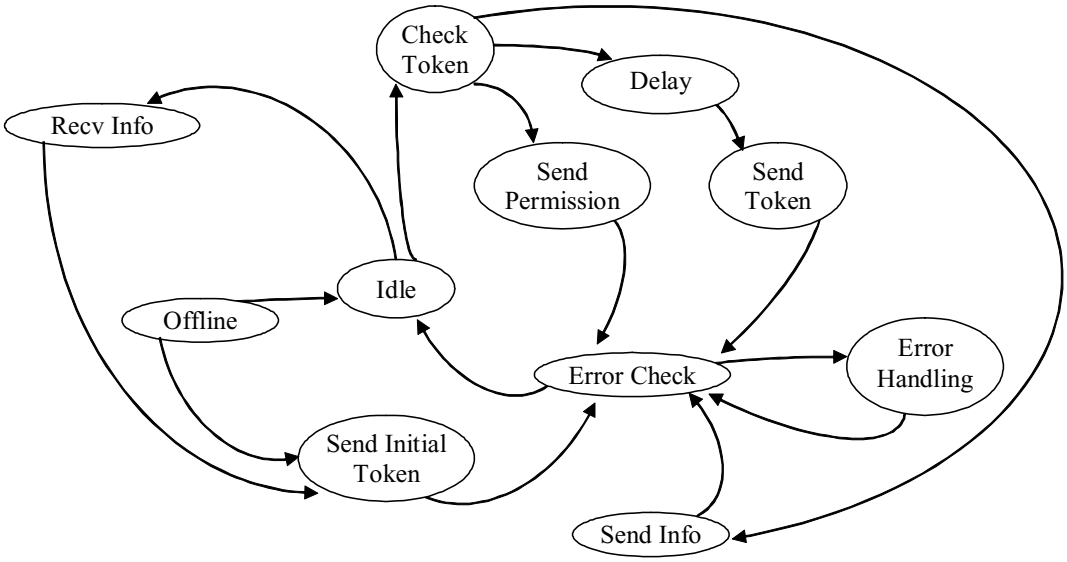


Figura 2.7: Máquina de estados de RT-EP con detección de fallos

Este nuevo estado *Error Check* se encargará de recuperarse del fallo en función del último estado presente en la estación que detectó el error y volverá a hacer las mismas acciones que se realizaron en ese último estado. Si se trata del último reintento, entonces se procederá a realizar una acción de lectura y paso de testigo.

Como resumen de las acciones que se toman en ese nuevo estado *Error Handling* exponemos la siguiente figura 2.8.

Último estado	Primeros reintentos	Último reintento
Send Initial Token	Retransmisión del testigo	Reconfiguración & Nuevo testigo
Send Token	Retransmisión del testigo	Reconfiguración & Nuevo Testigo, *Transmit token o Info*
Send Info	Retransmisión de la información	Reconfiguración & Nuevo testigo
Send Permission	Retransmisión del *Transmit Token*	Reconfiguración & Nuevo testigo

Figura 2.8: Resumen del funcionamiento del estado *Error Handling*

Como se puede apreciar, todas las acciones que derivan del último reintento suponen la expulsión de la estación del anillo, la creación y la circulación de un nuevo testigo menos en el caso de *Send Token*. En este caso existe una excepción que es cuando la estación que falla es la sucesora de la que detectó el fallo y además es la *token_master*. La estación que detecta el fallo asume ese papel y transmite el paquete *Transmit Token* a la estación que deba transmitir o transmite la información en el caso que le tocase a ella.

El método anteriormente expuesto hace que el protocolo se recupere del fallo más crítico: el fallo de una de las estaciones.

3. Implementación de RT-EP

3.1. Introducción

Los protocolos definidos específicamente para que transmitan información sobre una red de área local tienen que solventar los problemas concernientes a la transmisión de bloques de datos sobre la red [35]. Desde un punto de vista OSI de abajo a arriba para realizar una comunicación a través de una LAN necesitamos una capa física que nos proporcione:

- Codificación / decodificación de señales
- Sincronización de tramas
- Recepción y transmisión de bits.

Además para podernos comunicar en una LAN necesitamos una capa de enlace que se encargue de:

- Proveer unos puntos de enlace con la aplicación (o capas superiores).
- En transmisión, ensamblar datos en tramas con la dirección destino y con campos de detección de errores
- En recepción, desensamblar las tramas, realizar reconocimiento de dirección destino y detección de errores.
- Arbitrar el acceso al medio.

En Ethernet, entendida como capa física y capa MAC únicamente, tenemos todas las funciones de la capa física y las de la capa de enlace menos la primera función.

RT-EP añade una capa de adaptación a la capa de enlace que utiliza y sobrecarga las funciones de ésta además de proporcionar un punto de acceso a la aplicación por medio de canales de comunicación (*channel_id*). Proporciona así una comunicación fiable a través de una LAN. Es responsabilidad de la aplicación asignar estos identificadores de canal a cada *thread* o tarea.

El diseño e implementación de este protocolo en MaRTE, como se expuso en el apartado 1.4, se puede dividir en tres fases como se refleja en la figura 3.1. Las dos primeras fases son objeto de este trabajo. La tercera es muy similar a la segunda ya que solo cambia el sistema operativo sobre el que se “ejecuta”. En todo momento se ha tenido siempre en cuenta, a la hora de la programación, que el objetivo final era MaRTE así que su implementación en ese sistema operativo será bastante inmediata.

Primeramente RT-EP se implementó sobre datagramas UDP como se muestra en el primer paso de la figura 3.1. Esta implementación respondía más bien a la comprobación del arbitrio de acceso al medio. Una vez comprobado el funcionamiento del paso de testigo se procedió a implementar el protocolo sobre Ethernet como era el objetivo. No se comentará más sobre la implementación del protocolo sobre UDP ya que sólamente se

realizó en forma de experimento. La verdadera implementación del protocolo se realizó sobre la capa de enlace de Ethernet. El sistema operativo utilizado es GNU/Linux y el lenguaje de programación C. Los motivos han sido porque GNU/Linux es un sistema abierto, bien conocido, fácilmente configurable con muchas facilidades para el soporte de redes y nuevos protocolos. La elección de C por su parte vino motivada por la necesidad de utilizar interfaz de *sockets* y su integración en drivers.

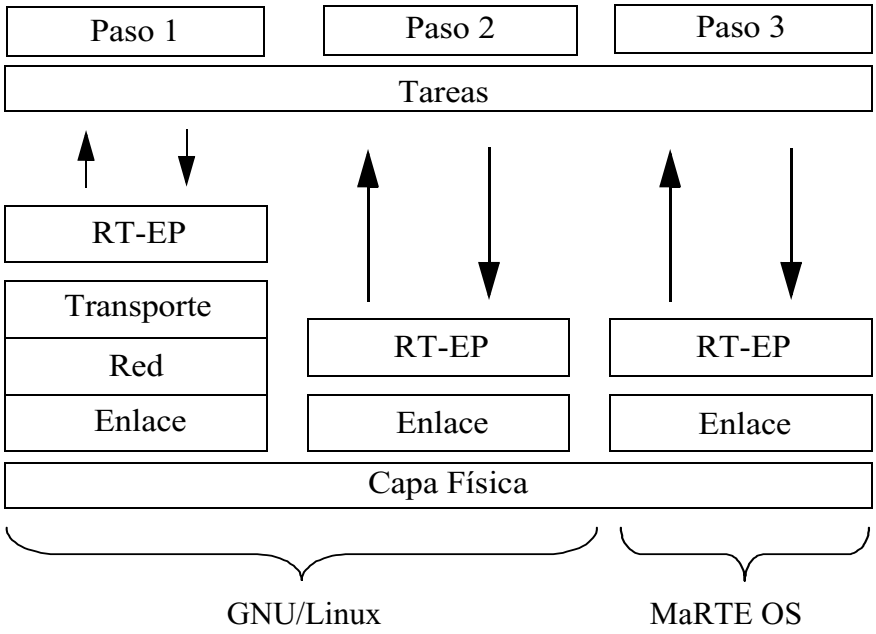


Figura 3.1: Fases en el desarrollo del protocolo

3.2. Arquitectura del software del protocolo

La funcionalidad y arquitectura del protocolo se muestran en la figura 3.2. Como se puede observar el protocolo consta de varias colas o canales de recepción, una por cada tarea o *thread* que requiera comunicarse a través de la red, y una cola de transmisión. Además posee una interfaz con la aplicación. Estas colas son de prioridad. En la cola de transmisión se almacenarán todos los mensajes que se quieran transmitir con su prioridad y el canal al cual tengan que ser entregados. El canal de comunicación coincide con el número de cola de recepción en el nodo destino. Existe también un *thread* de comunicaciones que se encarga de configurar el anillo y el acceso a la red a través de un archivo de configuración. Este *thread* lee la información de la cola de transmisión y la entrega a la red; también lee información de la red y la guarda en la cola de recepción correspondiente al canal del mensaje recibido.

Este protocolo ofrece tres funciones y un tipo de datos a la aplicación:

- **init_comm:** Esta función inicializa y configura la prioridad del *thread* de comunicaciones. Esta función, aunque se puede llamar repetidas veces, sólo funciona con la primera llamada en la que quedará fijado el techo de prioridad del *thread* de comunicaciones y quedará configurado el acceso a la red.

- **send_info:** Se utiliza para poder mandar un mensaje de una determinada prioridad a un determinado canal de comunicaciones.
- **recv_info:** Esta función se encarga de recibir un mensaje con su prioridad de un determinado canal de comunicaciones.
- **any_info:** Nos indica si hay algún elemento en una determinada cola de recepción.
- **element_t:** Este tipo de datos, que se describirá a fondo más adelante, es el tipo de mensaje que la aplicación deberá pasar a las funciones para mandar y recibir información de otras aplicaciones en red. La aplicación deberá proporcionar, además del mensaje a entregar, la dirección MAC de la estación destino y el canal de comunicaciones que desea utilizar.

La comunicación con la red se hace a través de la interfaz *sockets* que será brevemente descrita en 3.3, "Detalles de la implementación".

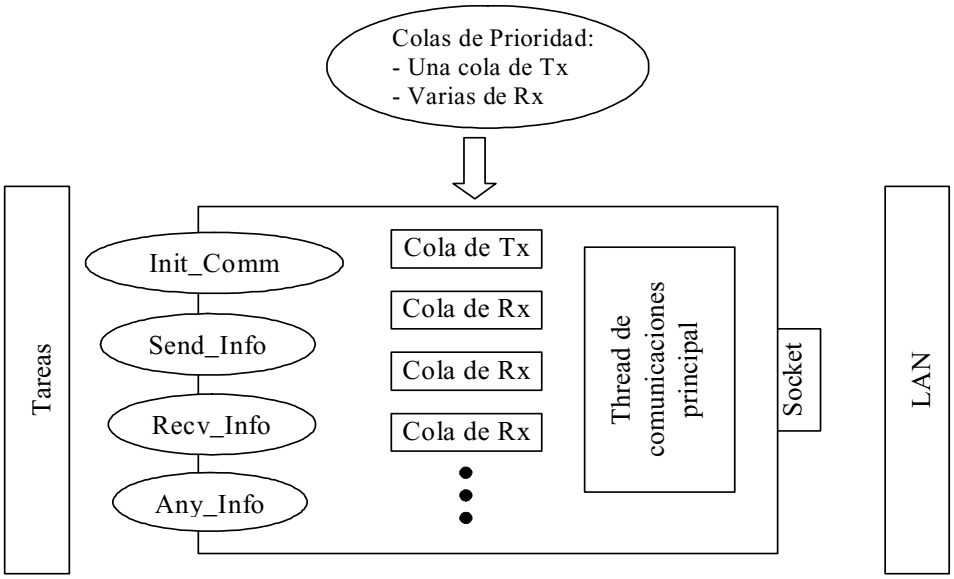


Figura 3.2: Funcionalidad y arquitectura de RT-EP

Para los sistemas de Tiempo Real, conocer el número de tareas que se van a ejecutar a priori es un requisito. Por ello, como puede verse en la figura 3.2, el número de colas de recepción, es decir, el número de tareas que deseen comunicarse a través de la red, ha de ser conocido a priori e indicado al protocolo en su fase de configuración.

3.3. Detalles de la implementación

A continuación mostraremos una sucesión de apartados que describirán uno tras otro las distintas facetas de la implementación del protocolo. Para un mayor entendimiento de las facetas utilizadas en toda esta sección recomendamos "The GNU C Library Reference Manual" [23] que es una gran referencia de C (aparte del estándar [17]) usando el compilador *gcc* en entornos Linux.

3.3.1. Sockets en RT-EP

La base para la E/S de red en los sistemas UNIX / Linux se centra en la abstracción conocida como *socket*. Podemos considerar el socket como una generalización del mecanismo de acceso a archivos de UNIX que proporciona un punto final para la comunicación. Al igual que con el acceso a archivos, la aplicación requiere crear un *socket* cuando necesita acceder a un dispositivo de E/S de red. El sistema operativo devuelve un entero que la aplicación utiliza para hacer referencia al *socket* creado. La principal diferencia entre un descriptor de archivo y un descriptor de *socket* está en que en el caso de un descriptor de archivo el S.O. enlaza ese descriptor con un archivo o un dispositivo específico, mientras que un descriptor de *socket* es creado sin estar enlazado a una dirección de destino específica. La aplicación puede elegir abastecer una dirección de destino cada vez que usa un *socket* (comunicación no orientada a conexión) o por el contrario enlazar la dirección del *socket* a una dirección de destino y así evadir la especificación de destino repetidamente (comunicación orientada a conexión).

Cada vez que tenga sentido, los *sockets* hacen operaciones exactamente iguales a los archivos o dispositivos de UNIX, de manera que pueden utilizarse con las funciones tradicionales de *read - write*.

Los *sockets* son muy utilizados para realizar conexiones TCP o UDP, pero también admiten otro tipo de conexiones. En nuestro caso usamos una opción especial de la interfaz socket que son los *packet sockets*. Estos *sockets* han sido introducidos por el kernel 2.x[1] de Linux y se utilizan para enviar y recibir tramas a nivel del driver del dispositivo (capa de enlace). Permiten al usuario implementar módulos de protocolos en el espacio de usuario por encima de la capa física. Se necesitan permisos de usuario con identificador 0 (root) o con capacidad de administrador de red (CAP_NET_ADMIN). La creación de un *packet socket*, incluyendo sus respectivos *includes*, se realiza de la siguiente manera:

```
#include <sys/socket.h>
#include <features.h>/* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>/* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>/* The L2 protocols */
#endif
packet_socket = socket(PF_PACKET, int socket_type , int protocol);
```

En el argumento *socket_type* puede indicarse:

- SOCK_RAW para acceder a tramas con la cabecera de nivel de enlace (en realidad, en Ethernet, a los campos de dirección y tipo)
- SOCK_DGRAM para recibir únicamente el campo de datos de la trama Ethernet.

El argumento *protocol* es para indicar qué protocolos vamos a utilizar a través de ese *socket*. En nuestro caso usaremos, como *socket_type*, SOCK_RAW para poder

1. Realmente han sido fruto de una evolución desde el kernel 2.0. Los packet sockets actuales son válidos a partir del kernel 2.4

acceder y tener control sobre la trama Ethernet. Y como protocolo, como ya hemos indicado con anterioridad, hemos usado un número de protocolo no utilizado 0x1000; así sólo recibiremos tramas de nuestro protocolo a través del *socket*.

Los paquetes SOCK_RAW se transmiten y reciben a través del driver de dispositivo sin ningún cambio en la trama. Cuando se recibe una trama, la dirección también es devuelta en el formato de la estructura *sockaddr_ll*. Esta estructura nos permite acceder a la dirección física de forma independiente del dispositivo. Este tipo de estructuras *sockaddr* son las utilizadas comúnmente con los *sockets* para conseguir la independencia del dispositivo. Aun así nosotros no las usaremos ya que lo que intentamos conseguir es acceder lo más cerca posible a la trama Ethernet. La ventaja de utilizar este tipo de *sockets* es que podemos mandar y recibir tramas directamente del driver de red sin que sea modificado ya que somos nosotros los que construimos parte de la trama (el CRC lo calcula el driver de red, y el preámbulo es añadido por el dispositivo). Otra ventaja que tenemos es que su implementación en MaRTE u otros sistemas operativos de Tiempo Real será más sencilla al no requerir una implementación de *sockets* completa.

Por omisión, todas las tramas del protocolo serán pasadas al *packet socket*, vengan de la interfaz que vengan. Si tenemos 10 tarjetas Ethernet conectadas a un sistema, todos las tramas dirigidas a cualquiera de esas tarjetas serán recibidas por el *socket*. Para solo recibir las tramas de una determinada interfaz como puede ser *eth0*, es decir, la primera tarjeta Ethernet conectada a nuestro sistema, debemos utilizar la llamada *bind* especificando la dirección de la tarjeta y la interfaz de ésta en una estructura *sockaddr_ll*. De esta manera se consigue enlazar el *packet socket* a una determinada interfaz. Para este propósito, y según las páginas de manual de Linux, para conseguir el enlazado deseado es suficiente especificar, en los campos de la estructura *sockaddr_sll*, el protocolo *sll_protocol* (htons(0x1000)) y la interfaz mediante *sll_ifindex* (la primera interfaz Ethernet coincide con el índice 2; explicaremos cómo obtener el índice de una interfaz más adelante). Pero en realidad esa llamada no funcionará correctamente si no especificamos la familia de *sockets* sobre la que operamos con *sll_family* (AF_PACKET al tratarse de *packet sockets*).

Una interfaz se identifica por su nombre (*eth0*, *eth1* etc.) y un número que es el que usa el S.O. para identificarla. La estructura *struct if_nameindex* es la que identifica una interfaz. Contiene un puntero a carácter, el nombre de la interfaz y un entero que es el número asignado a esa interfaz. Para averiguar las interfaces que poseemos en el sistema podemos hacer uso de la llamada *struct if_nameindex * if nameindex (void)* que devuelve un array de estructuras *if_nameindex*. Cada elemento del array será una interfaz presente en el sistema. En nuestro caso, como ya tenemos un *socket* abierto, debemos indicarle que sólo debe escuchar de una determinada interfaz que, como ya se comentó anteriormente, se especifica con el campo *sll_ifindex* de la estructura *sockaddr_sll* en la llamada a *bind*.

Para conseguir el índice debemos rellenar el campo *ifr_name* de la estructura *struct ifteq* con el nombre de la interfaz cuyo índice queramos conocer, en nuestro caso *eth0*. Como ya se ha dicho con anterioridad esto es configurable en el protocolo debido a que no siempre tenemos la necesidad de utilizar la misma interfaz Ethernet. La estructura *struct ifreq* es la que se utiliza para configurar los dispositivos de red a través de la llamada *ioctl.* Ésta realiza una operación genérica de E/S sobre descriptores de ficheros (hemos comentado que un *socket* se puede considerar como un descriptor de fichero). Además se pueden utilizar macros para acceder a cierta información del dispositivo, realizando así una llamada sobre el descriptor del *socket*, con la petición SIOGIFINDEX y con la

estructura *ifreq* con su campo *ifr_name* indicando el nombre de la interfaz. Devuelve en el campo *ifr_index* de *ifreq* el número de la interfaz. Una vez conseguida la interfaz ya podemos realizar la llamada a *bind* y hacer que el *socket* sólo reciba las tramas de una determinada interfaz.

Además de utilizar la llamada *ioctl* para conseguir el número de la interfaz también la usamos con la petición SIOCGIFHWADDR para conocer la dirección MAC de la interfaz. Esta dirección se necesita para que la estación pueda localizar su posición en el anillo a partir del fichero de configuración como se expondrá más adelante.

Los *packet sockets* se pueden utilizar para configurar los modos *multicast* y *promiscuo* de la capa de enlace. El modo *multicast* es para activar el empleo de direcciones *multicast*. Una dirección *multicast*, a diferencia de una *broadcast*, es capaz de mandar una trama a un grupo de estaciones configuradas para recibir tramas de una determinada dirección *multicast*. Es decir, no todas las estaciones del medio las reciben como suyas, como pasaría en un *broadcast*, sino solo aquellas que formen parte de la dirección *multicast*. El modo promiscuo es un modo especial en el que el *socket* recibe no sólo las tramas dirigidas a él, sino también todas las tramas válidas arrojadas al medio. En esta implementación hemos habilitado el modo promiscuo para que todas las estaciones sean capaces de recibir todas las tramas transmitidas al medio, este comportamiento es imprescindible para el tratamiento de errores en el protocolo. Para que una estación compruebe el correcto funcionamiento de su sucesora deberá escuchar una trama válida en el medio enviada por la sucesora, trama que no va dirigida a ella. Se deja el modo *multicast* para futuras modificaciones del protocolo. Para activar dicho modo usamos la llamada *setsockopt*. Esta llamada nos permite manipular distintas opciones asociadas a un *socket*. Otra de las opciones, que a primera vista podríamos haber utilizado, es SO_BINDTODEVICE que permite enlazar un *socket* a una interfaz. En nuestro caso, no fue posible utilizar esta opción. Tuvimos que realizar esta acción, como se ha visto con anterioridad, mediante una llamada a *ioctl*. No se pudo emplear *setsockopt*, como hubiese sido deseable, debido a que esta opción no está implementada aún (en el kernel de Linux 2.4.x) para *packet sockets*.

Una vez que se ha enlazado el socket a una interfaz podemos usar las llamadas *sendto* para enviar y *recvfrom* para recibir. Como ya se ha comentado con anterioridad no vamos a usar la estructura *sockaddr_sll* ya que vamos a construir directamente las tramas Ethernet:

```
int sendto(int s , const void * msg , size_t len , int flags ,
              const struct sockaddr * to , socklen_t tolen);
```

Situamos en *s* el descriptor del *socket*, en **msg* el puntero al mensaje que ya incluirá la cabecera Ethernet y en *len* la longitud del mensaje. Todos los demás campos se podrán a 0 (*flags y tolen*) o a NULL (*to*).

```
int recvfrom(int s , void * buf , size_t len , int flags ,
              struct sockaddr * from , socklen_t * fromlen);
```

De forma análoga a *sendto*, emplazamos en *s* el descriptor del *socket*, en **buf* indicamos el comienzo del *buffer* de recepción y en *len* la longitud del *buffer*. Todos los demás campos se podrán a 0 (*flags y fromlen*) o a NULL (*from*).

3.3.2. Temporizadores en RT-EP

Como se ha comentado en el apartado 2.4, “Tratamiento de errores en RT-EP”, el método que usa este protocolo para detectar fallo es el uso de un *timeout*. Este *timeout* se implementa mediante el uso de un temporizador. Un temporizador es un objeto que puede notificar a una tarea si ha transcurrido un cierto intervalo de tiempo o si se ha alcanzado una hora determinada [12]. Cada temporizador esta asociado a un reloj:

Reloj del sistema:

- Mide los segundos transcurridos desde las 0 horas, 0 minutos, 0 segundos del 1 de enero de 1970 UTC (Coordinated Universal Time).
- Se utiliza para marcar las horas de creación de directorios, archivos etc.
- Es global al sistema.

Reloj de Tiempo Real:

- También mide el tiempo UTC.
- Se utiliza para *timeouts* y temporizadores.
- Puede coincidir o no con el reloj del sistema.
- Es global al sistema.
- Resolución máxima: 20 ms
- Resolución mínima: 1 ns

Para crear un temporizador usamos la siguiente operación:

```
timer_create(clockid_t clockid, struct sigevent *evp,
             timer_t *timerid);
```

De esta manera creamos un temporizador asociado al reloj *clockid* que, en nuestro caso, será CLOCK_REALTIME. En *evp* indicamos la notificación que queremos recibir. Ésta puede ser:

- El envio de una señal.
- Crear y ejecutar un thread.
- Ninguna.

En RT-EP utilizaremos el manejador de la señal enviada por el temporizador para realizar el control de errores en el protocolo.

En *timerid* se devuelve el identificador del temporizador. El temporizador siempre se crea desarmado, esto es, no está activo. Para armar un temporizador debemos utilizar la llamada:

```
int timer_settime(timer_t timerid, int flags,
             const struct itimerspec *value,
             struct itimerspec *ovalue);
```

La estructura itimerspec consta de:

- **struct timespec it_interval:** Indicamos el periodo del *timer*, es decir, cada cuánto tiempo queremos que se repita.
- **struct timespec it_value:** Indicamos el tiempo de expiración. Si vale 0 el temporizador se desarma y en caso contrario pasa a tomar su valor

En RT-EP armaremos el temporizador cuando le toque a la estación detectar un posible fallo en la red y se desarmará cuando éste no se haya producido. Si se produce algún error se resuelve y se rearma el temporizador.

En el campo *flag* especificamos si queremos que la primera expiración sea en un tiempo absoluto determinado por *value.it_value*, en cuyo caso valdrá TIMER_ABSTIME. Si queremos que la expiración ocurra cuando pase un intervalo de tiempo igual a *value.it_value*, nuestro caso, el campo *flags* sera 0.

Si *ovalue* no es NULL devuelve el valor anterior del temporizador.

El *timeout* implementado en el protocolo es configurable a priori ya que dependiendo del tiempo de respuesta de las estaciones convendrá ponerlo a un valor o a otro.

3.3.3. Procesos ligeros o Threads

Un *thread*, también llamado proceso ligero, es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio en la pila [34]. Un *thread* comparte, con sus iguales, la sección de código, sección de datos y recursos del sistema operativo como archivos abiertos y señales, lo que se denomina colectivamente como una *tarea*. Podemos decir que un proceso es como una tarea con un solo *thread*. Debido a que son "procesos ligeros", la conmutación de la CPU entre threads y su creación es poco costosa en comparación con el cambio de contexto entre procesos.

Los *threads* pueden estar en uno de los siguientes estados: listo, bloqueado, en ejecución o terminado. Al igual que los procesos, los *threads* comparten la CPU, y solo hay un hilo en ejecución en un instante dado. Un *thread* dentro de un proceso se ejecuta secuencialmente y cada hilo tiene su propia pila y contador de programa. Un *thread* puede crear threads hijos, pero éstos no serán independientes entre sí ya que todos los *threads* pueden acceder a todas las direcciones de la tarea (un *thread* puede leer la pila de cualquier otro o escribir sobre ella).

Los *threads* tienen dos estados posibles para controlar la devolución de recursos al sistema:

- **Independiente (*detached*):** Cuando termina el *thread* devuelve al sistema los recursos utilizados.
- **Sincronizado (*joinable*):** Cuando el thread termina mantiene sus recursos que se liberan cuando otro *thread* llama a *pthread_join()*.

En nuestro caso daría igual como devolviese el *thread* de comunicaciones los recursos ya que nunca termina su ejecución en la vida del sistema.

Para crear un *thread* es necesario definir sus atributos en un objeto especial *pthread_attr_t*. En este objeto especificaremos:

- El tamaño mínimo de la pila.
- Su dirección.
- El control de devolución de recursos.

La llamada de creación del thread se realiza:

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
              void *(*start_routine)(void *), void *arg);
```

Donde:

- *thread* es el identificador (tid) del thread.
- *attr* es el conjunto de atributos que queremos que contenga el thread.
- *start_routine* es un puntero donde se comenzará a ejecutar el thread que será la dirección de una función.
- *arg* es un puntero al parámetro que será pasado a la rutina donde comenzará el *thread*.

Para la terminación del *thread* usamos:

```
void pthread_exit (void *value_ptr);
```

Esta función termina el *thread* si es sincronizado, y hace que el valor apuntado por *value_ptr* esté disponible para una operación *join*. La llamada para esperar a la terminación de un *thread* cuyo estado es sincronizado (*joinable*) es:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

También se puede cambiar el estado del *thread* a independiente (*detached*) mediante la instrucción:

```
int pthread_detach (pthread_t thread);
```

Para más información sobre *threads* se recomienda consultar [28].

El protocolo RT-EP está implementado en un *thread* de comunicaciones que se ejecutará de acuerdo a una prioridad establecida sobre MaRTE.

3.3.4. Sincronización de threads

Una tarea *cooperativa* [34] es la que puede afectar o ser afectada por las demás tareas que se ejecutan en el sistema. El acceso concurrente a datos compartidos puede dar pie a inconsistencia de los mismos. Para que esto no ocurra es necesario un mecanismo que asegure la ejecución ordenada de procesos cooperativos. En general, los procesos cooperantes deben sincronizarse siempre que intenten usar recursos compartidos por varios de ellos, tales como estructuras de datos o dispositivos físicos comunes. A ese segmento de código que puede modificar los recursos compartidos se denomina *sección critica* [34].

Las alternativas que existen para la sincronización, en sistemas operativos POSIX, son [12]:

- **Semáforos contadores:** Se trata de un recurso compartido en forma de un entero no negativo. Se utilizan tanto para sincronización de acceso mutuamente exclusivo, como de espera. Si el valor del semáforo es 0, no está disponible. Existen un par de operaciones que se realizan sobre los semáforos:
  - *Wait:* Si el valor del semáforo es 0 la tarea se añade a una cola de espera. Si es mayor que 0 la tarea se ejecuta y se decrementa el valor del contador.
  - *Signal:* Si hay tareas esperando, se elimina a una de la cola y se activa. Si no, se incrementa el valor del contador.

- **Mutex:** Es un método de sincronización en el que múltiples *threads* acceden de forma mutuamente exclusiva a un recurso compartido. Solo aquél que tenga posesión del *mutex* podrá ejecutarse. Las operaciones que se realizan sobre un *mutex* son:
  - *Lock:* Si el *mutex* está libre, el *thread* lo toma y se convierte en propietario. Si no estuviese libre, el *thread* se suspende y se añade a una cola.
  - *Unlock:* Si hay *threads* esperando en la cola, se activa uno de ellos y se convierte en el nuevo propietario del *mutex*. Si no hubiese ninguno en la cola, el *mutex* quedaría libre. Solo aquellos threads que sean propietarios de un *mutex* pueden efectuar esta operación.
- **Variable Condicional:** Es un método de sincronización que permite a un *thread* suspenderse hasta que otro lo reactiva y se cumple una condición (o predicativo lógico). Las operaciones que se realizan sobre las variables condicionales son:
  - **Wait** (a una condición): Se suspende la tarea hasta que otro *thread* señaliza esta condición. Entonces se comprueba un predicado lógico y se repite la espera si el predicado es falso.
  - **Signal** (una condición): Se reactiva uno o más de los *threads* suspendidos en espera de esa condición.
  - **Broadcast** (de una condición): Se reactivan todos los *threads* suspendidos en espera de esa condición.

Nos centraremos en la utilización de *mutex* y variables condicionales ya que ha sido la opción elegida para acceder a los recursos compartidos (las colas) en el protocolo.

Antes de crear un *mutex* o una variable condicional hay que definir sus atributos en un objeto especial: *pthread_mutexattr_t* y *pthread_condattr_t* respectivamente. En el caso de los *mutex* podemos indicar:

- Si pueden o no compartirse entre procesos (compartido o privado).
- El protocolo utilizado (con o sin herencia de prioridad o con protección de prioridad)
- El techo de prioridad que es un valor entero que se puede modificar en tiempo de ejecución.

En las variables condicionales sólo podemos indicar si se pueden compartir entre procesos.

En RT-EP se crearon los *mutexes* con un protocolo de protección de prioridad y estableciendo un techo de prioridad en su creación. El techo de prioridad es un argumento de entrada a la función *init_comm* que inicia la comunicación en la estación.

A continuación mostraremos una serie de llamadas utilizadas para el manejo de los mutex y las variables condicionales. Para una mayor descripción de estos se recomienda consultar [28]:

- Inicializar un *mutex*:

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

- Destruir un *mutex*:

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

- Tomar un *mutex* y suspenderse si no esta libre:

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

- Liberar un *mutex*:

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

- Inicializar una variable condicional:

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

- Destruir una variable condicional:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Señalizar:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Broadcast:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Esperar:

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

- Espera con *timeout*:

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

En el caso de la variable condicional, la condición se representa como un predicado booleano protegiendo su evaluación con un *mutex*.

A continuación expondremos el pseudocódigo correspondiente a este tipo de señalización:

Pseudocódigo del *thread* que señaliza [12]:

```
pthread_mutex_lock(un_mutex);
     predicate=TRUE;
     pthread_cond_signal(cond);
pthread_mutex_unlock(un_mutex);
```

Pseudocódigo del *thread* que espera [12]:

```
pthread_mutex_lock(un_mutex);
     while(predicate == FALSE){
        pthread_cond_wait(cond, un_mutex);
     }
pthread_mutex_unlock(un_mutex);
```

Para realizar una sincronización de espera sobre un objeto determinado es preferible crear un monitor [12]. Éste tendrá operaciones que encapsulan el uso de las primitivas de sincronización y poseerá una interfaz separada del cuerpo. De esta manera evitaremos errores en el manejo del *mutex* y de la variable condicional, que podrían llegar a bloquear la aplicación.

En el acceso a las colas de prioridades, tenemos el problema clásico de *lectores - escritores* reflejado en la figura 3.3. Para solucionarlo implementamos un *monitor de espera* para sincronizar el acceso al objeto compartido que son las colas de prioridad del protocolo. Hemos definido un monitor con operaciones de *creación, extracción* y *encolado*. Como parece lógico pensar, la operación de extracción utiliza sincronización de espera. Si la cola esta vacía, la llamada se bloquea hasta que se introduzca algún elemento en la cola.

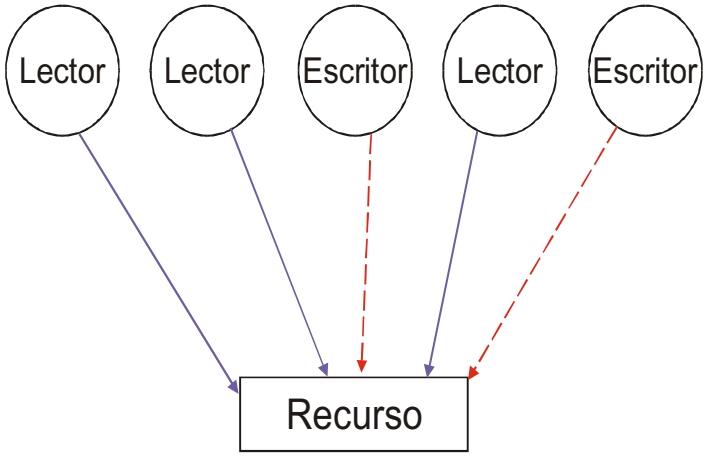


Figura 3.3: Problema clásico lectores-escritores

Aplicando el pseudocódigo expuesto anteriormente parece lógico que la operación de encolado debe ser la acción que se encarga de señalizar. La de extracción ha de ser la que espera a que haya algún elemento en la cola, y el predicado "que haya algún elemento en la cola" será la condición a esperar. Evidentemente la operación de creación se encargará de crear y configurar los *mutex* y variables condicionales. Cuando intentemos acceder a la cola de prioridades sólo lo podremos hacer si no está siendo usada por otro *thread* y, además, cuando intentemos extraer un elemento de la cola, si está vacía, esperaremos (liberando momentáneamente el *mutex*) hasta que algún *thread* la llene y nos despierte mediante *pthread_cond_signal*.

En este protocolo tenemos un *mutex* y una variable condicional por cada cola, de manera que se sincroniza el acceso a las distintas colas, bien por parte del *thread* de comunicaciones, o bien por parte de la aplicación.

La exposición del monitor utilizado por este protocolo se encontrará descrita en el apartado 3.3.6.5, "prio_queue_monitor"

3.3.5. Estructura de datos: Colas de prioridad (Heaps)

Un árbol es una estructura de datos jerarquizada [2]. Está constituido por una serie de elementos llamados nodos. Cada nodo tiene a un padre, menos un nodo especial llamado nodo raíz del cual cuelgan todos los demás, y pueden o no tener nodos hijos. Esta estructura esta expuesta en la figura 3.4. La ordenación de los nodos es importante en el árbol. Se realiza ordenando los hijos de un mismo nodo de izquierda a derecha. Este orden se puede extender a los nodos no hermanos de forma que si un nodo está a la izquierda de otro nodo todos sus hijos también quedan a la izquierda de su descendencia.

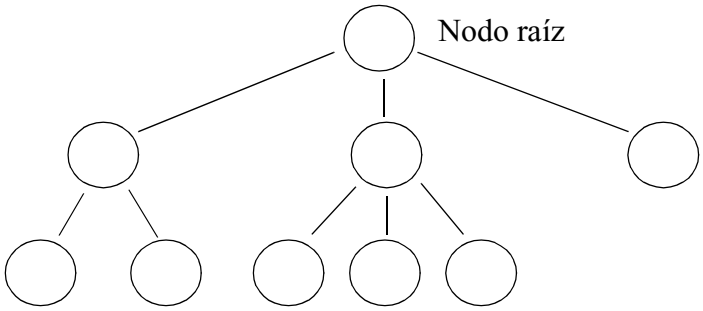


Figura 3.4: Estructura de datos: Arbol

Un tipo de árbol distinto al de la figura 3.4 es el *árbol binario*. Un árbol binario es un árbol en el que cada nodo puede tener como máximo dos hijos: uno derecho y otro izquierdo. Esta estructura tiene como ventaja que la búsqueda de elementos se realiza de forma muy eficiente si el árbol tiene sus elementos ordenados, O(log n) en caso promedio. El método de ordenación consiste en colocar a todos los descendientes de un nodo que sean menores que él a la izquierda y a los que sean mayores a su derecha.

La idea de un *heap* [2], o cola de prioridades, es conseguir una cola ordenada de acuerdo a la prioridad de cada elemento. A diferencia de un árbol binario, un *heap binario*, figura 3.5, es una estructura de árbol más simple que tiene las dos propiedades siguientes:

- La prioridad de cada nodo es mayor o igual que la de sus dos hijos
- No existen huecos en el árbol, se rellena de arriba a abajo y de izquierda a derecha , no nos podemos saltar ningún nodo.

Como se puede ver, una sub-rama a la izquierda no necesariamente contiene prioridades menores que una a la derecha. Esto es debido a que un *heap* tiene una ordenación diferente a la del árbol binario tradicional. Sin embargo lo que si se puede afirmar es que el nodo raíz de cada sub-árbol tiene una prioridad que es mayor o igual que cualquier nodo que cuelga de él. El nodo raíz del *heap* siempre tiene la mayor prioridad de todo el árbol. Será ese nodo el primero que se extraiga del *heap*.

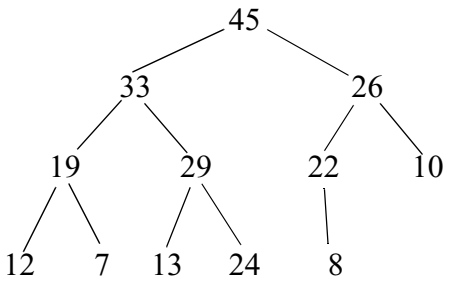


Figura 3.5: Heap binario

Para mantener la propiedad comentada anteriormente de que no debían existir huecos, a la hora de insertar un elemento se deberá crear un nodo al final del nivel más bajo del *heap*. Si el nuevo elemento tiene una prioridad mayor que la del padre deberá sustituirle ocupando el ex-padre el lugar que ocupaba el hijo. Se realizará esta operación de sustitución de padres a lo largo del árbol hasta que algún padre sea mayor que el nuevo nodo. De esta forma se conservará la estructura del *heap*. Este mecanismo de inserción queda representado en la figura 3.6 donde se inserta un nuevo elemento con prioridad 30.

En el caso de la extracción del nodo de mayor prioridad, se crea un hueco en el nodo raíz. Debemos trasladar el hueco a la parte más baja y más a la derecha del *heap* como se muestra en la figura 3.7.

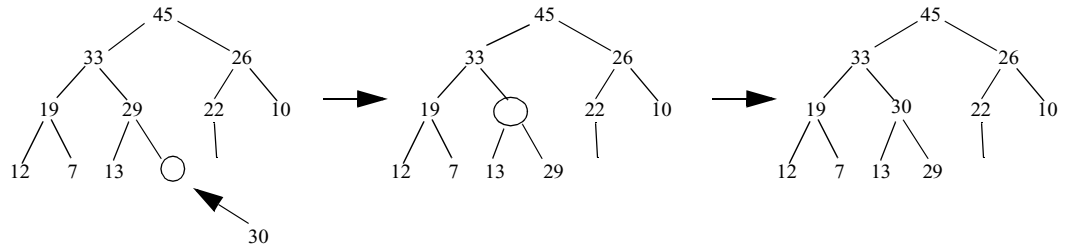


Figura 3.6: Proceso a seguir para insertar un elemento

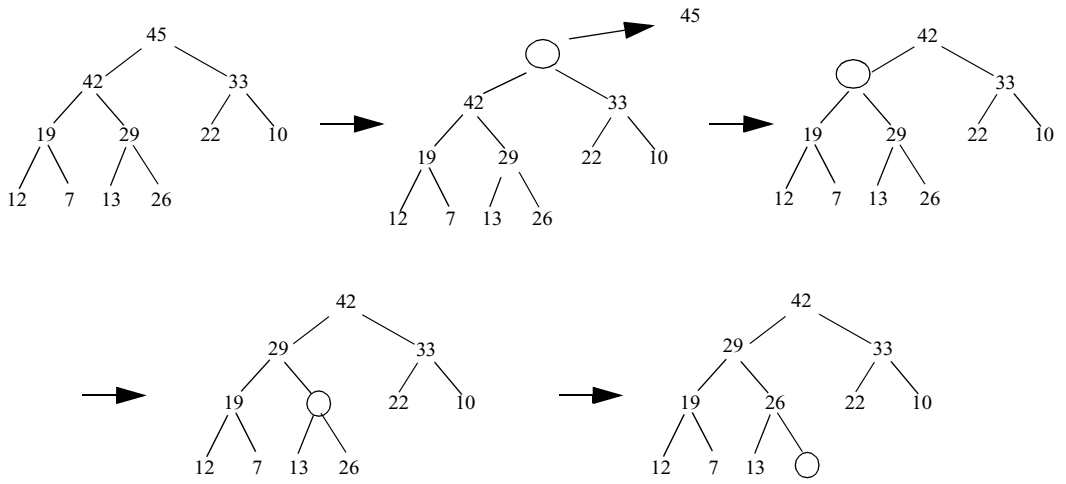


Figura 3.7: Proceso a seguir para extraer un elemento

Primeramente procederemos a comparar el último elemento, en nuestro caso el 26, con los dos hijos que colgaban del antiguo nodo raíz. De ser mayor o igual, este nodo sustituiría al antiguo nodo raíz. De no ser así, el mayor de los hijos del nodo raíz pasará a ser el nuevo nodo raíz y, de esta manera, hemos desplazado el hueco un nivel. De forma análoga, como el último nodo, el 26, no es mayor o igual que los nuevos hijos del hueco, 19 y 29, ponemos al mayor de éstos en el sitio del hueco. Así hemos desplazado el hueco otro nivel. Finalmente en la última comparación vemos que 26 es mayor que 13. Por tanto ocupa el lugar del hueco.

Un faceta interesante del *heap binario* es que se puede implementar como un array. Al no tener huecos, podemos colocar los nodos en un array de forma que podamos utilizar formulación aritmética para encontrar el hijo de cualquier nodo [2]. El array se crea colocando el nodo raíz como primer elemento del array y, a continuación, el siguiente nivel de izquierda a derecha. Se sigue este proceso hasta que se termina el árbol. En la figura 3.8 mostramos cómo quedaría implementado en un array el *heap* de la figura 3.5.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
45	33	26	19	29	22	10	12	7	13	24	8

Figura 3.8: Representación del *heap* como un array

Ahora podemos hacer uso de la siguiente propiedad: dado un elemento del array *i*, sus dos hijos estarán en las posiciones *2i* y *2i+1*. De esta manera podemos navegar por el array como si fuese un árbol. Como se ve el array comienza en 1. Si estamos implementando este método en algún lenguaje, como es el caso de C, donde la numeración del primer elemento del array es 0, deberemos dejar el primer elemento vacío.

Esta implementación tiene los siguientes costes medios:

- Inserción: O(log n)
- Extracción del nodo de mayor prioridad: O(log n)
- Extracción del nodo de menor prioridad: O(n)
- Lectura del nodo de mayor prioridad: O(1)
- Borrado: O(n)

En RT-EP se realizan operaciones de extracción, inserción y lectura del primer elemento de la cola de prioridad, por lo tanto, tendremos de un coste medio de O(log n) en el peor caso (extracción, inserción) y O(1) en el caso de la lectura del primer elemento de la cola. Este último coste está asociado a la operación realizada sobre las colas en proceso del paso de testigo.

La implementación de este tipo de colas de prioridad para la transmisión y recepción, se ha realizado en el módulo *prio_queue* que está descrito en el apartado 3.3.6.4.

3.3.6. Módulos del protocolo

En este apartado describiremos los distintos módulos en los que se divide el software del protocolo como puede verse en la figura 3.9 en la que están dibujados los módulos según su cercanía a la aplicación o a la red. Además también está indicado si dependen unos de otros. Se puede observar que el módulo *prot_types* abarca toda la extensión ya que además de contener parámetros importantes para la configuración del protocolo, contiene tipos y funciones de uso en todos los módulos. Para un mejor entendimiento de la figura y del protocolo pasaremos a enumerar primero y, en los distintos subapartados, a describir cada uno de los módulos:

- **prio_element:** Tipo de datos a transmitir.
- **prio_priority:** Tipo de la prioridad de los datos.
- **rt_comm:** Interfaz con la aplicación.
- **prio_queue:** Implementación de la cola de prioridades.
- **prio_queue_monitor:** Monitor de acceso a la cola de prioridades.
- **lnk_config:** Configuración del anillo.
- **martelnk:** *Thread* de comunicaciones.
- **prot_types:** Configuración de parámetros del protocolo.

3.3.6.1. prio_element

Este módulo está compuesto por un tipo de datos y una función de comparación de elementos:

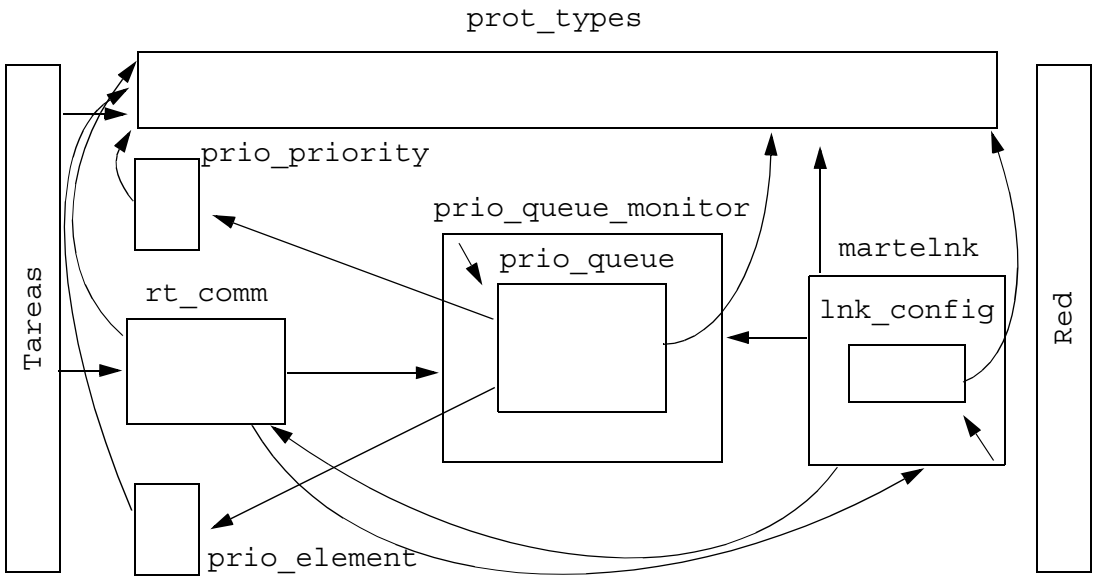


Figura 3.9: Módulos y dependencias del protocolo

```
typedef struct {
     char *element_pointer;
     positive_t element_size;
     unsigned char addr_dest[ETH_ALEN];
     positive_t channel_id;
} element_t;

boolean equal_element(element_t str1, element_t str2);
```

element_t es el tipo de datos que se utilizará para transmitir el mensaje de la aplicación. Ésta deberá rellenar esta estructura convenientemente para poder transmitir el mensaje:

- **element_pointer:** Es un puntero al comienzo de los datos que la aplicación desea transmitir.
- **element_size:** Es un positivo (*unsigned*) que indica el tamaño de los datos apuntados por *element_pointer*. El valor máximo es MAX_INFO_LENGTH que toma un valor de 1492 bytes. Indica el tamaño de los datos que se van a transmitir.
- **addr_dest:** Ésta es la dirección de 48 bits destino del mensaje que se quiere enviar. Se debe introducir por orden, un byte de la dirección (por parejas de dos números si la dirección viene dada en hexadecimal, que es lo común) en cada campo del array. La constante que define el tamaño del array, ETH_ALEN, toma un valor de 6. Este tamaño es el necesario para guardar los 48 bits de la dirección MAC.
- **channel_id:** Es un positivo que identifica el canal destino del mensaje, es decir, a la tarea destino del mensaje. No se puede usar el valor 0 ya que está reservado para uso interno del protocolo. Los valores válidos para este campo son los comprendidos entre el 1 hasta MAX_CHANNELS, ambos inclusive. El parámetro de MAX_CHANNEL está definido en el módulo de configuración del protocolo *prot_types.h*. Como ya se ha

expresado con anterioridad, es la aplicación la encargada de asignar estos canales a los *threads* que deseen comunicarse a través de la red.

La función *equal_element* devuelve *true* si los datos apuntados por el elemento *element_pointer* de str1 y str2 son iguales y *false* si no lo son

3.3.6.2. prio_priority

En este módulo definimos el tipo prioridad y una función de comparación de prioridades:

```
typedef unsigned char priority_t;
```

El tipo *priority_t* es un carácter sin signo, un byte. Esto quiere decir que disponemos de 255 prioridades ya que como se comentó con anterioridad la prioridad 0 está reservada para un uso interno del protocolo. 255 representa la máxima prioridad y 1 la mínima.

3.3.6.3. rt_comm

En *rt_comm* es donde se define la interfaz con la aplicación. Estas son las funciones que deberá utilizar la aplicación para poder comunicarse a través de la red. Las funciones usan el tipo *element_t* y el tipo *priority_t* definidos anteriormente. Por lo tanto la aplicación deberá usarlos adecuadamente. A continuación pasaremos a describir las funciones y los valores devueltos por éstas:

```
int init_comm(int prio_ceiling);
int send_info(element_t E, priority_t P);
int recv_info(element_t *E, priority_t *P);
int any_info(element_t *E);
```

La función *init_comm* se utiliza para inicializar el protocolo. Esta función se encarga de configurar y lanzar el *thread* de comunicaciones además de inicializar la cola de transmisión y las de recepción. Tanto el *thread* como las colas son globales al sistema y, por lo tanto, la función está programada para que sólo sea efectiva la primera vez que se la invoque. La función devuelve:

- **0** si la función ha terminado correctamente, es decir, no ha habido error.
- **CREATION_ERROR** si ha habido algún problema a la hora de crear las colas de transmisión y recepción.
- **UNEXPECTED_ERROR** en caso de algún error al crear el *thread*.

Para mandar datos a través de la red tenemos la función *send_info* que recibe la información de la aplicación en forma del tipo *element_t* y *priority_t* y se encarga de encolarla en la cola de transmisión. Los tipos *element_t* y *priority_t* deberán ser correctamente completados como se indicó en los correspondientes apartados. El protocolo no copia la información a transmitir a un espacio de memoria aparte, tan solo realiza una copia del tipo *element_t*. Por tanto, es responsabilidad de la aplicación el mantener el mensaje en memoria hasta que se haya transmitido. La función devuelve:

- **0** si no ha habido error.
- **STA_NOT_VALID** si la estación destino de la información, indicada por el campo *addr_dest* de *element_t* que en un principio formaba parte del anillo, ha sido excluida por un mal funcionamiento y ya no se le puede enviar información

- **STA_NOT_FOUND** si la estación, indicada por *addr_dest*, no es una estación a la cual se la pueda mandar datos ya que nunca ha formado parte del anillo. También se utiliza este mismo error para indicar una estación incompleta o no válida.
- **INFO_LENGTH_OVERFLOW** si los datos a transmitir, indicados en el campo *element_size*, superan el tamaño del campo de datos del paquete de información de RT-EP. Este protocolo no admite fragmentación de mensajes por lo tanto es responsabilidad de la aplicación hacer la fragmentación si desea mandar un mensaje de longitud superior a MAX_INFO_LENGTH. Otro apunte importante es que sólo se transmitirán los datos de longitud indicados en *element_size*. Si se indicase mal ese campo se podrían dar mensajes incompletos si *element_size* es menor que el volumen de datos que queramos transmitir, o con ruido si es mayor.
- **INVALID_PRIORITY** si hemos intentado encolar un dato con una prioridad incorrecta. De momento solo se contempla el 0 como prioridad no válida. La prioridad 0 se reserva para uso interno del protocolo.
- **INVALID_CHANNEL** si hemos utilizado algún canal de comunicaciones no válido, esto es, que el canal elegido haya sido el 0 o un canal superior a MAX_CHANNELS.
- **SYNCH_ERROR** si se ha cometido algún error en la sincronización de acceso al recurso compartido que es la cola de prioridades.
- **UNEXPECTED_ERROR** en el caso de tener un error no identificado.

Para recibir mensajes a través de la red usamos la función *recv_info*. Esta función recibe el elemento *element_t* donde debe especificarse el canal del cual queramos recibir los datos y nos devuelve en ese mismo *element_t*, debidamente completado, el mensaje de mayor prioridad enviado a través de ese canal. También devuelve la prioridad del elemento. El área de memoria que ocupa el mensaje es reservado por el protocolo y es responsabilidad de la aplicación liberarla, mediante la función *free*, cuando ya no sea utilizada. La llamada a *recv_info* es bloqueante. Esta función devuelve:

- **0** si no ha habido error y se han recibido correctamente los datos.
- **INVALID_CHANNEL** si el canal del que queremos recibir datos no es valido, esto es, que el canal elegido haya sido el 0 o un canal superior a MAX_CHANNELS.
- **SYNCH_ERROR** si se ha cometido algún error en la sincronización de acceso al recurso compartido que es la cola de prioridades.
- **UNEXPECTED_ERROR** en el caso de tener un error no identificado.

Por último tenemos una función, *any_info*, que nos indica si hay algún elemento en la cola de recepción indicada por el tipo *element_t*. Esta función devuelve:

- **0** si hay algún elemento esperando ser recogido.
- **NO_ELEMENTS** si no hay ninguno.
- **INVALID_CHANNEL** en caso de canal inválido.
- **SYNCH_ERROR** si se ha cometido algún error en la sincronización.
- **UNEXPECTED_ERROR** en el caso de tener un error no identificado.

También se definen las variables globales que serán las colas de transmisión y recepción utilizadas por el protocolo:

```
queue_t Q_tx,Q_rx[MAX_CHANNELS+1];
```

Son variables globales de tipo *queue_t*, que está definido en el módulo *prio_queue.h*. Éstas serán las colas de transmisión y de recepción de la estación. El número de colas de recepción, *Q_rx*, depende del parámetro MAX_CHANNELS de la configuración del protocolo que indica el número de *threads* de aplicación que en la vida del sistema van a comunicarse a través de la red simultáneamente en un nodo determinado. El protocolo se reserva el uso de la cola de recepción 0 para poder implementar con el mismo monitor las colas de prioridad tanto de recepción como de transmisión, manteniendo estas colas claramente diferenciadas en distintas variables para su mejor identificación en el código.

3.3.6.4. prio_queue

Este módulo implementa las estructuras de datos utilizadas para almacenar los elementos y prioridades a transmitir y recibir. Que según lo expresado en el apartado 3.3.5, se trata de colas de prioridades implementadas como *heaps*. A continuación expondremos los dos tipos de datos definidos para implementar las colas y las funciones utilizadas.

```
typedef struct {
     priority_t Pri;
     element_t E;
}cell_t;

typedef struct{
     positive_t Length;
     positive_t queue_size;
     cell_t *Q;
} queue_t;

void init_queue(queue_t *Q);
boolean empty(queue_t *Q);
positive_t queue_length(queue_t *Q);
int enqueue(element_t E, priority_t P, queue_t *Q);
int dequeue(element_t *E, priority_t *P, queue_t *Q);
boolean dequeue_middle(element_t *E, priority_t *P, queue_t *Q);
int read_first(element_t *E, priority_t *P, queue_t *Q);
boolean set_prio(element_t E, priority_t P,queue_t *Q);
```

Como se puede observar se ha definido un tipo, *cell_t*, para cada celda de la cola de prioridades. Este tipo contiene el *element_t* y *priority_t* que se van a transmitir / recibir. Seguidamente podemos ver el tipo para la cola, *queue_t*, que consta además del puntero que será el comienzo de la cola, de dos positivos de longitud: *Length* y *queue_size*. *Length* indica la longitud actual de la cola de prioridades y *queue_size* es un parámetro que indica el número de posiciones de memoria que están reservadas para su uso. Este último campo es imprescindible ya que al ser un tipo de datos de tamaño dinámico debemos tener algún medio que nos indique que debemos reservar más memoria para la cola si ésta desborda el tamaño asignado en un principio.

A continuación podemos ver el juego de funciones:

- *init_queue* para inicializar la cola

- *empty* para indicar si está vacía. Devuelve *true* si lo está y *false* si no.
- *queue_length* nos devuelve la longitud de la cola.
- *enqueue* se encarga de encolar un *element_t* y *priority_t* en la cola.
- *dequeue* se encarga de sacar de la cola el elemento de mayor prioridad.
- *dequeue_middle* se encarga de quitar de la cola un determinado *element_t*. En caso de no encontrarlo en la cola devuelve *false*.
- *read_first* se encarga de leer (no extraer) el elemento de mayor prioridad de la cola. En caso de estar la cola vacía, devuelve QUEUE_EMPTY.
- *set_prio* se encarga de buscar un elemento con una determinada prioridad. Si lo encuentra devuelve *true*. Si encuentra el elemento pero con distinta prioridad, se la cambia y devuelve *true*. Si no encuentra el elemento devuelve *false*.

3.3.6.5. prio_queue_monitor

Tal y como se comentaba en el apartado 3.3.4, "Sincronización de threads", este módulo es la implementación del monitor de espera que accede al recurso compartido que son las colas de prioridad. Para ello se han definido unas funciones de creación, encolado, extracción y lectura del primer elemento de la cola:

```
int create_queue(int prio_ceiling, queue_t *Q,
              const int queue_num);
int queue_insert(element_t E, priority_t P,queue_t *Q,
              const int queue_num);
int queue_extract(element_t *E,priority_t *P,queue_t *Q,
              const int queue_num);
int read_first_in_queue(element_t *E, priority_t *P,
              queue_t *Q,const int queue_num);
int is_queue_empty(element_t *E,queue_t *Q,const int queue_num)
```

Para crear las colas de prioridad a través del monitor debemos llamar a la función *create_queue*. Esta función debe llamarse antes de poder usar las colas ya que se encarga no sólo de inicializar los *mutex* y variables condicionales, sino que también hace la llamada pertinente para inicializar la cola *Q*. En esta función, debido a las diferencias en cuanto a la planificabilidad entre MaRTE y Linux, se han incluido unas directivas para que cuando se compile en Linux no se use, por ejemplo, *prio_ceiling* que es el techo de prioridad al que asociamos el *mutex* y que en Linux no esta implementado. Es importante comentar el uso de *queue_num*. Gracias a este entero conseguimos, con las mismas funciones, acceder y manejar distintas colas con distintos *mutex* y variables condicionales sin más que especificar con este entero el número de la cola a la cual accedemos. El valor de *queue_num* corresponde con el *channel_id* de la tarea. La función *create_queue* devuelve:

- **0** si no ha habido error.
- **CREATION_ERROR** si ha ocurrido algún error en la creación de los mutex o variables condicionales.

queue_insert es el acceso a través del monitor a la función que inserta un elemento con una prioridad en una cola *queue_t*. El proceso que sigue es como el caso del "thread que señaliza" en el apartado 3.3.4, "Sincronización de threads". La función devuelve:

- **0** si no ha habido error.

- **SYNCH_ERROR** en caso de producirse un error en la sincronización con los *mutex* o variables condicionales.

La función *queue_extract* es utilizada para extraer el elemento de mayor prioridad de la cola *queue_t* a través del monitor. Devuelve el elemento y la prioridad. Al igual que *queue_insert*, esta función toma su correspondencia con el caso del "thread que espera" del apartado 3.3.4. Por lo tanto, como se veía en el pseudocódigo, esta llamada se bloquea hasta que haya algún dato que retirar en la cola. La función devuelve:

- **0** si no ha habido error.
- **SYNCH_ERROR** en el caso de producirse un error en la sincronización.

Hemos definido en el monitor una función, *read_first_in_queue*, que realmente no extrae ni encola ningún elemento de la cola, sino que su cometido es leer el elemento de mayor prioridad. Usamos esta función para conseguir la información de prioridad para añadirla al testigo en la negociación de prioridades en la red sin desencolar ningún elemento. *read_first_in_queue* devuelve:

- **0** si no ha habido error.
- **QUEUE_EMPTY** en el caso de que no haya ningún elemento en la cola.
- **SYNCH_ERROR** si ha habido algún error de sincronización.

Por último se ha definido la función *is_queue_empty* que nos indicará si la cola de prioridades está o no vacía. Devuelve:

- **0** si hay algún elemento.
- **1** si la cola está vacía.
- **SYNCH_ERROR** en el caso de producirse un error en la sincronización.

3.3.6.6. lnk_config

Este módulo se encargará de leer, de un archivo, la configuración del anillo y guardarlo en una variable global a la estación. De este modo la estación será capaz de situarse en el anillo y recuperarse de algún posible error debido al fallo de alguna estación. Conviene aclarar que en Linux la configuración del anillo se obtiene a partir de un archivo, pero MaRTE, siguiendo las recomendaciones del subconjunto de Tiempo Real mínimo POSIX.13, no usa un sistema de ficheros. Es por ello que la configuración, en el caso de MaRTE, se provee directamente en el módulo *lnk_config* creando explícitamente la variable global de tipo *st_array* con la configuración del anillo lógico.

Este módulo proporciona los tipos que se encargaran de almacenar la información del anillo y las funciones necesarias para modificar o extraer información de la configuración del anillo:

```
typedef struct node {
     unsigned char addr_dest[ETH_ALEN];
     boolean valid_station;
} lnk_cell_t;

typedef struct {
     lnk_cell_t stations_array[MAX_STATIONS];
     positive_t elements;
} st_array;
```

```
int init_config();
int read_station(lnk_cell_t *sta, positive_t node);
positive_t number_of_nodes();
int check_station(unsigned char *st_addr);
int is_station_valid(unsigned char *st_addr);
int inhibit_station(unsigned char *st_addr);
```

Como puede verse, cada estación ocupa una celda, *lnk_cell_t*, en la que se guarda información de la dirección MAC de la estación, *addr_dest*, y un booleano, *valid_station*, que indicará si la estación está operativa en el anillo o no. Esta unidad básica forma parte de una array de celdas *stations_array*, cuyo tamaño es el número máximo de estaciones permitidas en el sistema. También tenemos la variable *elements* que indica el número de estaciones que forman el anillo lógico. Estas dos variables forman el tipo *st_array* que guardará toda la configuración de la red. Además del tipo de datos hemos definido una serie de funciones para su manejo.

Para que se inicialice la estructura de datos y se almacenen en ella toda la configuración del anillo lógico es preciso llamar a la función *init_config* que se encargará de abrir el archivo de configuración, leerlo y guardar los datos en la estructura global. En el caso de MaRTE, esta función rellena la variable global con los datos suministrados en el propio fuente. Esta es la única modificación que habría que hacer a este módulo para salvar la carencia de ficheros de MaRTE. El formato del archivo se describirá en el apartado 3.3.7, “Configuración del protocolo”. Esta función devuelve:

- **0** en caso de no haber habido error y haber rellenado correctamente la variable global.
- **STATIONS_OVERFLOW** si el archivo de configuración es incorrecto y hay descritas en él más estaciones de las que el protocolo tiene configuradas para manejar.
- **FILE_ERROR** en caso de no poder encontrar o abrir el archivo de configuración.

Utilizamos la función *read_station* para localizar una estación en la configuración. Ésta será la función que nos permita ir recorriendo toda la estructura que guarda la configuración del anillo lógico. Provee en el argumento *sta* la estación situada en la posición indicada por *node*. La primera estación es el nodo 0. La función *read_station* devuelve:

- **0** si ha podido leer la estación y emplazarla en *sta*.
- **STA_NOT_FOUND** en caso de que la estación indicada por *node* no exista debido a que la posición indicada sobrepasa la cantidad de estaciones indicadas en la configuración.
- **STA_NOT_VALID** en el caso de que una estación, que pertenezca al anillo lógico, esté marcada como inválida debido a que se ha detectado su fallo. Este valor es indicativo, es decir, guardará en *sta* la información de la estación inválida además de devolvernos este valor.

Para averiguar el número de estaciones que hay en el anillo lógico usamos la función *number_nodes* que devuelve el número de nodos (estaciones) existentes en la configuración.

Para comprobar si una estación está activa, es decir, que sigue perteneciendo al anillo, usamos la función *check_station*. Recibe un array apuntando a una dirección MAC

de 48 bits y comprueba si la estación pertenece al anillo y si está activa. Esta función es muy útil para chequear la dirección destino de un mensaje, es decir, ver si es un destino válido dentro del anillo. Esto evitará esperas innecesarias. Devuelve:

- **0** si la estación apuntada por *st_addr* pertenece al anillo lógico y está activa.
- **STA_NOT_VALID** en el caso de que la estación, aunque pertenezca al anillo lógico, esté marcada como no válida y por lo tanto no esté operativa para procesar mensajes.
- **STA_NOT_FOUND** si la estación no se encuentra en la configuración del anillo.

Debido a que el *check_station* revisa toda la configuración hasta encontrar la estación *st_addr* y comprueba si es o no valida, puede llegar a ser un sobre-esfuerzo en aquellas situaciones en las que no es necesario comprobar si una determinada estación es válida sino simplemente si no está marcada como no válida. Esta sutil diferencia nos ahorra gasto computacional ya que se realizan muchas menos comparaciones. Por eso se ha definido la función *invalidated_station*, que sólo comprueba si entre las estaciones marcadas como inválidas está la estación *st_addr*. Esta función devuelve:

- **0** si la estación no se encuentra entre las estaciones marcadas como no válidas.
- **STA_NOT_VALID** en el caso de que la estación se encuentre entre las estaciones marcadas como no válidas.

Una de las bases de configuración del anillo lógico es marcar las estaciones no válidas. Esto se consigue con la función *inhibit_station* que se encargará de marcar como no válida a la estación *st_addr*. Invalidar una estación previamente invalidada no constituye error. La función *inhibit_station* devuelve:

- **0** si ha marcado la estación, *st_addr*, como no válida sin error.
- **STA_NOT_FOUND** si la estación especificada por *st_addr* no se encuentra en el anillo lógico.

En este módulo también se especifica la localización y nombre del archivo de configuración.

3.3.6.7. martelnk

Este módulo es el más importante, ya que realiza la implementación del protocolo propiamente dicha. En este módulo definimos los paquetes de comunicación de RT-EP y el *thread* de comunicaciones:

```
typedef struct {
     struct ethhdr eth_header;
     unsigned char packet_id;
     priority_t priority;
     unsigned short int packet_number;
     positive_t channel_id;
     positive_t length_info;
} info_packet_t;

typedef struct {
     struct ethhdr eth_header;
     unsigned char packet_id;
```

```
        priority_t priority;
        unsigned short int packet_number;
        unsigned char token_master_addr[ETH_ALEN];
        positive_t failure_station;
        unsigned char failure_station_addr[ETH_ALEN];
        unsigned char addr_prior[ETH_ALEN];
    } token_packet_t;

    int martelnk();
```

Tenemos definidos los paquetes del protocolo: *info_packet_t* y *token_packet_t.* La composición de las estructuras corresponde con los formatos y tamaños de las tramas definidas en el apartado 2.3, “Formato de los paquetes RT-EP”, más la cabecera Ethernet:

- **info_packet_t:** Aquí se muestra sólo la cabecera del paquete de información, los datos van añadidos al final de esta cabecera.
    - *eth_header*: Cabecera Ethernet.
    - *packet_id*: Es el identificador de paquete, *Packet Identifier.*
    - *priority*: La prioridad del mensaje que se transporta. Está asociada al campo *Priority* del paquete.
    - *packet_number*: Es el número de secuencia *Packet Number.*
    - *channel_id*: es el identificador de canal, *Channel Identifier*, que vincula los extremos de la comunicación
    - *length_info*: Indica la longitud de datos transmitidos; corresponde con el campo *Info Length* del paquete de información.
- **token_packet_t:** En este tipo definimos los distintos campos que utilizaremos cuando manejemos el testigo.
    - *eth_header*: Cabecera Ethernet.
    - *packet_id*: Corresponde con el identificador de paquete, *Packet Identifier.*
    - *priority*: Es la prioridad del mensaje de mayor prioridad de la estación *addr_prior.* Está asociado al campo *Priority* del paquete de testigo.
    - *packet_number*: Es el número de secuencia, *Packet Number.*
    - *token_master_addr*: Se asocia al campo *Token_Master Address* que alberga la dirección de la estación *token_master* en ese momento.
    - *failure_station*: Este campo es la implementación del campo del mismo nombre *Failure Station* del paquete de testigo que indica si ha habido algún fallo en alguna máquina.
    - *failure_station_addr*: Se corresponde con el campo *Failure Address* que almacenará la dirección de la estación en fallo.
    - *addr_prior*: Se asocia al campo *Station Address* que alberga la dirección MAC de la estación que posee en ese momento el mensaje de mayor prioridad a transmitir.

Llamamos cabecera Ethernet a los campos dirección destino, dirección fuente y el tipo de trama de Ethernet. Estos campos, en Linux, se definen a través de *struct ethhdr.* Esta estructura tiene tres campos:

- *h_dest* para indicar la dirección MAC de la estación destino.
- *h_source* que indica la dirección origen de la trama.

- *h_proto* que señala el protocolo utilizado.

Es el protocolo el que se encarga de generar y manejar directamente las tramas que se van a transmitir / recibir. Esta característica hace que el protocolo esté más cerca del driver del dispositivo y así conseguimos aumentar la eficiencia y la flexibilidad de éste al tener un mayor control directo sobre la trama enviada.

La función que ejecutará el *thread* de comunicaciones es *martelnk*. Esta función es la que se encarga de realizar todo el trabajo de:

- Configuración de las estaciones.
- Acceso a la red.
- Entrega de datos a la aplicación destinataria de los mensajes almacenados en la cola de transmisión.
- Recogida de los paquetes de la red y su entrega a la cola de recepción adecuada para que la aplicación destinataria pueda recogerlas.

En caso de existir algún error en la configuración de las estaciones que conforman el anillo lógico, el thread termina prematuramente devolviendo UNEXPECTED_ERROR.

3.3.6.8. prot_types

Este módulo contiene multitud de parámetros de configuración del protocolo. Estos parámetros se desarrollarán en el apartado 3.3.7, "Configuración del protocolo". Además también están presentes algunos tipos de datos y funciones que han sido usados a lo largo del protocolo:

```
typedef char boolean;
typedef unsigned short int positive_t;

int addrcmp(unsigned char *addr1, unsigned char *addr2,
              const int len);
int addrcpy(unsigned char *addr1, unsigned char *addr2,
              const int len);
```

Hemos definido el tipo booleano, *boolean*, de 1 byte. Además el tipo positivo, *positive_t*, de 2 bytes como un entero "corto" sin signo. También tenemos dos funciones *addrcmp* y *addrcpy*, que se utilizan para comparar y copiar los arrays que contienen las direcciones MAC de las estaciones. Estas funciones, básicamente, lo único que hacen es comparar y copiar, respectivamente, los arrays elemento a elemento. Se han realizado varias pruebas comprobando que es mucho más rápido computacionalmente que utilizar las funciones de la librería estándar *memcmp* y *memcpy*. Las funciones reciben dos punteros que indican el comienzo de las direcciones. El argumento *addr1* identifica el destino y *addr2* la fuente. También hay que especificar el tamaño de las estructuras en *len*. En el protocolo se ha utilizado la constante ETH_ALEN para definir el tamaño que tiene un valor de 6. La función *addrcmp* devuelve TRUE si son iguales y FALSE si no lo son.

3.3.7. Configuración del protocolo

En este apartado se describirán los parámetros utilizados para la configuración del protocolo. Todos estos parámetros se alojan en el fichero de cabeceras *prot_types.h.* Estos

parámetros están definidos como directivas al preprocesador mediante cláusulas *#define*. Se han seguido dos políticas en el uso de estas cláusulas:

- La directiva vista como una definición de una constante de configuración.
- El que esté o no definido un determinado parámetro, sin que tenga que tener asociado un valor necesariamente, implicará adoptar una política u otra.

Un apunte importante de la configuración es que el protocolo no exige que todas las estaciones tengan la misma configuración. Esa es una libertad que se ha dejado al desarrollador. Aunque es recomendable poner la misma configuración en todas las máquinas no es imprescindible salvo en el caso de la información de configuración del anillo lógico.

El primer parámetro especificado en el fichero *prot_types.h* es una pareja de cláusulas y una de ellas necesariamente deberá estar comentada:

```
#define LINUX_STATE
/* #define MARTE_STATE */
```

Con ellas indicamos el entorno para el que compilaremos el programa. Son necesarias debido a que GNU/Linux carece de atributos de prioridad que en MaRTE, por ejemplo, son imprescindibles. Así si estamos compilando en un entorno GNU/Linux al definir LINUX_STATE obviamos las líneas de código que hacen referencia al manejo de prioridades en la CPU que se tendrán en cuenta cuando se defina MARTE_STATE.

A continuación podemos especificar el nombre del fichero del que el protocolo tomará la configuración del anillo. Como ya hemos comentado con anterioridad, a la hora de la implementación en MaRTE no se podrá utilizar esta definición. Tendremos que introducir la configuración del anillo en el fichero de cabecera. De todas formas, la configuración consistirá en las direcciones MAC de las estaciones que componen el anillo y su orden. Especificamos el archivo y su localización por medio de:

```
#define FILE_NAME_CONFIG "./lnk.conf"
```

El nombre del archivo con su ruta debe ser puesto entre comillas. En este caso se ha utilizado el archivo *lnk.conf* del directorio donde se lanza el *thread*, aunque lo ideal será poner una ruta global al sistema como */etc/net/lnk.conf*. La configuración deberá darse por medio de un archivo de texto plano (*plain text*) con una enumeración de las direcciones MAC que conforman el anillo lógico y su orden en éste. La estación colocada en primer lugar será la que desempeñe el papel de *token_master* en la primera vuelta del testigo. Las direcciones MAC irán colocadas en forma de una dirección por cada fila, cada 8 bits de dirección dos puntos como se muestra a continuación:

```
00:4F:4C:02:F6:68
00:4F:49:04:7D:8D
```

Es importante cerciorarse de que las estaciones que conforman el anillo estén correctamente especificadas ya que la lectura del anillo lógico es una parte crítica del funcionamiento del protocolo. Una estación mal especificada conllevará la no incorporación de ésta en el anillo lógico.

Más adelante encontramos la forma de definir el número máximo de estaciones admitidas en la configuración del protocolo. Este parámetro, actualmente, sólo se utiliza

a modo de comprobación de la integridad del archivo de configuración del anillo lógico. Este parámetro se especifica de la siguiente manera:

```
#define MAX_STATIONS 100
```

El número de canales de comunicación que deseamos abrir en la estación quedará definido mediante:

```
#define MAX_CHANNELS 5
```

El número de canales será, en este caso, desde el canal 1 hasta el canal 5 ambos incluidos. Este parámetro también determina el número de colas de recepción presentes en la estación.

También podemos especificar la interfaz de red, en caso de poseer más de una tarjeta Ethernet en el sistema. Así especificamos la interfaz sobre la que queremos que "funcione" el protocolo. Esta definición se realiza:

```
#define IFR_NAME "eth0"
```

El *estilo* empleado para la definición de la interfaz de red es el mismo que se utiliza en GNU/Linux. Así *eth0* se refiere a la primera tarjeta Ethernet del sistema, *eth1* se referirá a la segunda etc.

Una constante de posible utilidad en el programa de usuario es MAX_INFO_LENGTH que indica el tamaño máximo de datos de usuario permitido en el protocolo:

```
#define MAX_INFO_LENGTH 1492
```

El protocolo no admite mensajes superiores a este tamaño, la función *send_info* devolverá error.

Otro parámetro de la configuración del protocolo es el identificador utilizado en la trama Ethernet al que, como ya se expuso con anterioridad, se ha asignado un valor correspondiente a un protocolo desconocido 0x1000:

```
#define RT_EP_PROTOCOL 0x1000
```

Un parámetro importante en la configuración del protocolo es STATE_SLEEP_TIME. La justificación de este parámetro, como ya se explicó con anterioridad, es debida al *overhead* introducido por la transmisión del testigo de prioridad 0. Conseguimos que una estación se "duerma" mediante el uso de un *nanosleep* previo al procesado del testigo entrante. Esta llamada se configura con los parámetros:

```
#define STATE_SLEEP_TIME_S 0
#define STATE_SLEEP_TIME_NS 0
```

STATE_SLEEP_TIME_S define los segundos que se dormirá la estación y STATE_SLEEP_TIME_NS define los nanosegundos en caso de requerir una mayor precisión. El tiempo total que se dormirá la estación será la suma de ambos parámetros. Una restricción que tiene STATE_SLEEP_TIME_NS es que no se puede poner una cantidad mayor o igual que 1e9 nanosegundos ya que esa cantidad supone un segundo y debe especificarse en el parámetro que indica los segundos. En el caso de que ambos parámetros estén igualados a 0 se inhibe la llamada a *nanosleep*.

Una consideración importante que no podemos pasar por alto es que, en el caso de GNU/Linux, la implementación actual de *nanosleep* está basada en el mecanismo del

timer del *kernel*, que tiene una resolución de 1/HZ (que corresponde con 10 ms en la implementación de Linux sobre máquinas i386 y de 1 ms sobre máquinas Alpha). Por lo tanto la llamada a *nanosleep* actúa, como mínimo, el tiempo especificado en el argumento. Sin embargo la activación de la tarea o *thread* puede demorarse hasta 10 ms más que el tiempo especificado en *nanosleep*. La implementación de esta llamada en MaRTE, o cualquier otro sistema operativo de Tiempo Real, carece de esta "característica" ya que en este tipo de sistemas se emplea una política de planificación de Tiempo Real. Así podemos usar *nanosleep* con la precisión de varios nanosegundos proporcionada por los relojes hardware de este tipo de sistemas.

Otro parámetro de vital importancia para el correcto funcionamiento del protocolo es la especificación del *timeout* y del número de reintentos que se harán en el caso de detectar error. Recordemos que el *timeout* es el tiempo que estará una estación, que ha transmitido un paquete, escuchando el medio en espera de una respuesta a su transmisión. La correcta asignación de tiempo es imprescindible para que el protocolo pueda recuperarse adecuadamente de errores en las tramas o en las estaciones. Estos parámetros se asignan:

```
#define LISTEN_TIMEOUT_S 0
#define LISTEN_TIMEOUT_NS 900000
#define MAX_RETRIES 3
```

LISTEN_TIMEOUT_S y LISTEN_TIMEOUT_NS especifican el *timeout* activo en cada estación en segundos y nanosegundos respectivamente. De forma análoga a *nanosleep*, LISTEN_TIMEOUT_NS tiene una restricción de 1e9 nanosegundos. MAX_RETRIES especifica el número de reintentos en los que se considerará que la estación ha fallado. Es decir, se reintentará la transmisión 'MAX_RETRIES - 1' veces y al siguiente reintento se excluirá la estación en fallo del anillo lógico. El *timeout* de la estación tiene una resolución de 10 ms en GNU/Linux sobre máquinas i386 (y de 1ms en Alpha) debido a los problemas comentados anteriormente en el caso de *nanosleep*.

Seguidamente expondremos unos parámetros que hacen que el protocolo alcance distintos modos de depuración (*debug*). Para activar un modo basta con quitar comentarios a los parámetros y para desactivarlos, añadirlos. Se usan los comentarios estándar de C. Este modo puede ser útil para, experimentalmente, chequear o ajustar tiempos de parámetros del protocolo:

- **FAULT_HANDLING_DEBUG_MODE**: Quitando los comentarios a este parámetro, el protocolo entra en un modo de comprobación de recuperación de errores. Es decir, el protocolo realiza una pausa de duración igual al tiempo definido en el *timeout* de la estación cada vez que ésta procese un número de paquetes especificados en el parámetro FAULT_PACKET. Se consigue que salten los *timeouts* de las estaciones cada FAULT_PACKET transmitidos y se intente recuperar del fallo que supone tener una estación ocupada. Los parámetros se definen:

  ```
  #define FAULT_HANDLING_DEBUG_MODE
  #define FAULT_PACKET 3000
  ```

- **MARTELNK_DEBUG_MODE**: Con este parámetro sin comentar entramos en un modo de depuración del *thread* de comunicaciones, que se traduce en un modo *verbose*, es decir, recibimos información en la salida estándar sobre los estados que atraviesa la máquina, paquetes que recibe, tratamiento de errores etc. Se define mediante:

```
#define MARTELNK_DEBUG_MODE
```

- **MONITOR_DEBUG_MODE**: En este modo recibimos en la salida estándar distintas indicaciones sobre el funcionamiento del monitor de acceso al recurso compartido que son las colas de prioridad. Está definido como:

```
#define MONITOR_DEBUG_MODE
```

- **LNK_CONFIG_DEBUG_MODE**: Aquí configuramos la estación para que nos indique, a través de la salida estándar, los pormenores de la lectura y configuración del anillo lógico. Se define:

```
#define LNK_CONFIG_DEBUG_MODE
```

- **RT_COMM_DEBUG_MODE:** Con este parámetro podemos recibir a través de la salida estándar información sobre cuándo y cómo se utilizan las funciones de la interfaz con la aplicación.

```
#define RT_COMM_DEBUG_MODE
```

- **CPU_OVERHEAD_PROTOCOL_TIMES**: Con este parámetro accedemos a un modo en el que podemos medir los overheads de CPU asociados a los distintos estados del protocolo durante las primeras MAX_CPU_POINTS tramas. En caso de necesitar un "tiempo de calentamiento" en el sistema donde sea deseable descartar los primeros tiempos medidos se define el parámetro CPU_POINTS_DISCARTED. Tiene dos modos de medida dependiendo de si el parámetro CPU_FILE_OUTPUT está sin comentar o no. Si no está comentado se generará una salida al fichero especificado (solo en Linux) indicando todos los tiempos medidos de cada estado. En MaRTE, lógicamente, no se puede usar esta opción. En este modo se descartarán los tiempos de las primeras CPU_POINTS_DISCARTED tramas y no se contarán a efectos de cálculo de overheads. Si CPU_FILE_OUTPUT está comentado se informará a través de la salida estándar de los overheads de CPU cuando se hayan procesado las tramas indicadas descartando las especificadas por los parámetros correspondientes. En este caso, los tiempos mostrados son los relativos al peor caso, al mejor caso y a la media de todos los tiempos medidos. También se muestra información acerca del número de tiempos parciales que se han tomado en los distintos estados, para así comprobar la validez de los datos medidos y ajustar MAX_CPU_POINTS en caso de no tener suficientes muestras en algún estado. Una vez informado de los overheads, el protocolo deja de actuar parando la comunicación. Estos parámetros se definen como:

```
#define CPU_OVERHEAD_PROTOCOL_TIMES
#define MAX_CPU_POINTS 50000
#define CPU_POINTS_DISCARTED 5
#define CPU_FILE_OUTPUT "CPU_Overhead_times"
```

- **TOKEN_CHECKING_PRIORITY_TIMES**: En este modo se accede a la medición del tiempo que se tarda en realizar el paso de testigo durante una vuelta completa al anillo lógico. Se mide el tiempo que transcurre desde que la estación *token_master* está en el estado *Send Initial Token* hasta que llega al estado *Check Token*. Toma tiempos durante los MAX_TOKEN_POINTS primeras veces en que la estación, que no sea la

primera en el anillo, es la *token_master* y, de estar sin comentar, aloja el resultado de los tiempos parciales en el archivo especificado. Si estuviese comentado, los tiempos mostrados son los relativos al peor caso, al mejor caso y a la media de todos los tiempos medidos. Cuando se han acabado de efectuar las medidas, el protocolo se para. Este modo de medición se indica con:

```
#define TOKEN_CHECKING_PRIORITY_TIMES
#define MAX_TOKEN_POINTS 500
#define TOKEN_FILE_OUTPUT "TOKEN_Overhead_times"
```

El protocolo es bastante configurable en fase de compilación, requisito imprescindible en MaRTE que, siguiendo la norma POSIX.13 para sistemas mínimos de Tiempo Real [30], carece de ficheros. Con una pequeña modificación en el módulo de configuración del anillo podemos conseguir que tampoco precise un archivo.

4. Modelado MAST

4.1. Introducción a MAST

En el apartado 1.4, “Objetivos de este proyecto”, ya se expresó que una faceta importante en el protocolo es su modelado para que pueda ser analizado con la aplicación. Esta sección muestra la información del modelado de RT-EP de acuerdo a MAST (Modeling and Analysis Suite for Real-Time Applications) [13], que es un proyecto que está siendo desarrollado por el Departamento de Electrónica y Computadores de la Universidad de Cantabria.

MAST es un conjunto de herramientas de código abierto que permite el modelado y el análisis temporal de aplicaciones de Tiempo Real, incluyendo el análisis de planificabilidad para la comprobación de los requerimientos temporales estrictos [10]. El modelo MAST puede ser usado [11]:

- En un entorno de diseño UML (Unified Modeling Language) para diseñar aplicaciones de Tiempo Real.
- Para representar todo el comportamiento y requerimientos de Tiempo Real junto con la información de diseño.
- Para permitir un análisis de planificabilidad.

Un sistema de tiempo real se modela como un conjunto de *transacciones*. Cada transacción se activa por uno o más eventos externos y representa un conjunto de *actividades* que se ejecutan en el sistema. Las actividades generan eventos que son internos a la transacción y que, por turnos, activarán otras actividades. En el modelo existen estructuras especiales manejadoras de eventos para utilizarlos de distintas maneras. Los eventos internos tienen requerimientos temporales asociados a ellos.

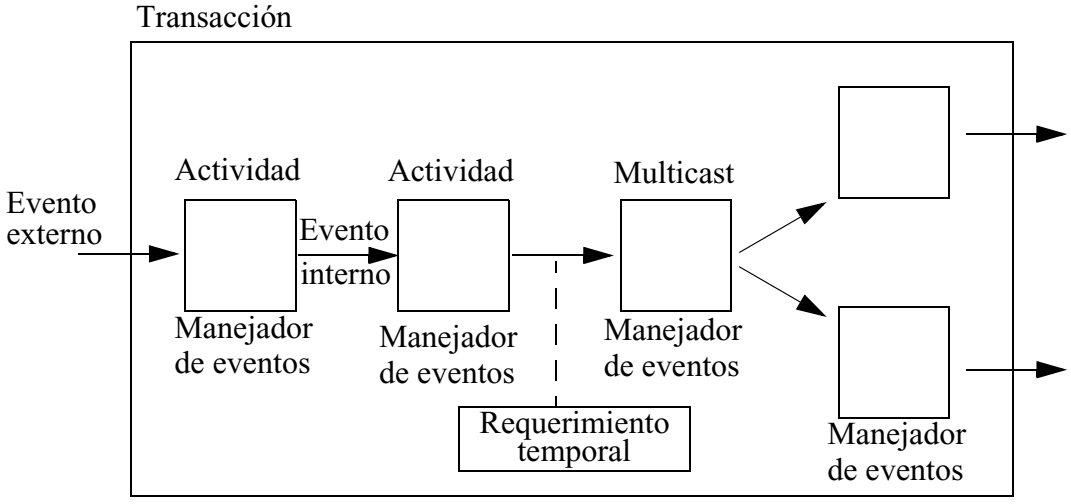


Figura 4.1: Sistema de Tiempo Real compuesto de transacciones [10]

La figura 4.1 [10] muestra un ejemplo de una transacción de un sistema. Esta transacción está activada por un solo evento externo. Después de haber ejecutado dos actividades, un manejador de eventos *multicast* genera dos eventos que activan a las dos últimas actividades en paralelo.

Los elementos que definen una actividad están descritos en la figura 4.2 [10]. Podemos comprobar que cada actividad está activada por un evento *entrante*, y genera un evento *saliente* cuando se haya completado. Si se necesita generar algún evento intermedio, la actividad será dividida en partes. Cada actividad ejecuta una *Operación* que representa un trozo de código para ejecutar en un procesador o un mensaje para ser mandado por la red. Una operación puede tener una lista de *Recursos compartidos* que precisa utilizar de un modo mutuamente exclusivo. La actividad es ejecutada por un *Servidor de planificación*, que representa una entidad planificable en el *Recurso de procesado* al cual se asigna (un procesador o una red). Por ejemplo, el modelo de un servidor de planificación es una tarea. Una tarea puede ser la responsable de ejecutar distintas actividades (procedimientos). El servidor de planificación se asigna a un objeto de *parámetros de planificación* que contiene la información de la política y parámetros de planificación utilizados [10].

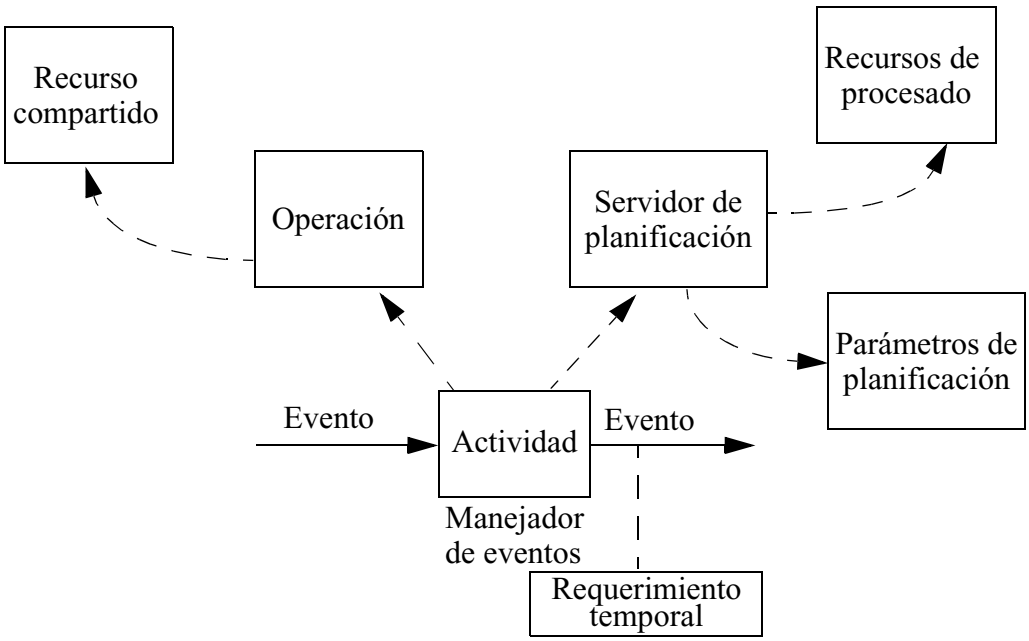


Figura 4.2: Elementos que definen una actividad [10].

A continuación se enumeran los elementos de los que consta MAST y, entre paréntesis, se expone la nomenclatura utilizada por el modelo MAST [10]. Debido a que este trabajo no versa sobre MAST, sólo se describirán brevemente los distintos elementos dando alguna explicación adicional en aquellos elementos y atributos que precisemos para el modelado el protocolo. Para una completa especificación de los elementos y del modelo MAST remitimos al lector a [10], [11] y [13]:

- **Recursos de procesado** (*Processing Resources*): Existen dos clases:
  - Procesador de prioridades fijas (*Fixed Priority Processor*): Es un procesador con una política de planificación basada en prioridades fijas.

- Red de prioridades fijas (*Fixed Priority Network*): Representa una red que utiliza un protocolo basado en prioridades fijas para mandar mensajes a la red. Esta clase tiene los siguientes atributos adicionales al *Fixed Priority Processor*:
  - Overhead asociado al paquete (*Packet Overhead*): Este es el (peor, medio y mejor) *overhead* asociado a cada paquete producido por los mensajes de protocolo que deben ser mandados antes o después de cada paquete útil.
  - Tipo de transmisión (*Transmission Kind*): Indicará si la comunicación es Simplex, Half Duplex o Full Duplex.
  - Máximo bloqueo (*Max Blocking*): El máximo bloqueo es causado por la no predecibilidad de los paquetes.
  - Tiempo de transmisión máximo y mínimo de paquete (*Max Packet Transmission Time* y *Min Packet Transmission Time*): El tiempo máximo se utiliza para el cálculo del modelo de overhead de la red. El tiempo mínimo representa el periodo más corto de los *overheads* asociados a la transmisión de cada paquete y, por tanto, tiene un gran impacto en el overhead que causan los controladores de red (*network drivers*) en los procesadores por el uso de ésta.
  - Lista de controladores (*List of Drivers*): Una lista de referencias a los controladores de red, que contienen el modelo de overhead del procesador asociado con la transmisión de mensajes a través de la red.

- **Timer del sistema** (*System Timers*): Representa los distintos *overheads* asociados con el modo en que el sistema maneja los eventos temporizados. Existen dos clases:
  - Reloj de alarma (*Alarm Clock*): Representa a un sistema cuyos eventos temporizados son activados por una interrupción del temporizador hardware.
  - Reloj periódico (*Ticker*): Representa a un sistema que tiene un reloj periódico, como puede ser una interrupción periódica que llega al sistema.
- **Controlador de red** (*Network Drivers*): Representa las operaciones ejecutadas en un procesador como consecuencia de la transmisión o recepción de un paquete a través de la red. Existen dos tipos:
  - Controlador de paquetes (*Packet Driver*): Representa un controlador que es activado en la transmisión o recepción de los mensajes.
  - Controlador carácter-paquete (*Character Packet Driver*): Es una especialización del *packet driver* en el que hay un *overhead* adicional asociado a la transmisión de cada carácter como ocurre en algunas líneas serie.
- **Parámetros de planificación** (*Scheduling Parameters*): Representan las políticas de planificación de prioridades fijas y sus parámetros asociados.

Existen las siguientes clases que sólo se enunciarán. Para su descripción consultar [10]:

- *Non Preemptible Fixed Priority Scheduler*
- *Fixed Priority Scheduler*
- *Interrupt Fixed Priority Scheduler*
- *Polling Scheduler*
- *Sporadic Server Scheduler*

- **Servidores de planificación** (*Scheduling Servers*): Representan entidades planificables en los recursos de procesado (*Processing Resource*)
- **Recursos compartidos** (*Shared Resources*): Representan los recursos que se comparten entre distintas tareas y que deben ser usadas de un modo mutuamente exclusivo. Sólo se permiten los protocolos que evitan las inversiones de prioridad sin límite. Existen dos clases:
  - *Immediate Ceiling Resource*
  - *Priority Inheritance Resource*
- **Operaciones** (*Operations*): Representan una parte de código o un mensaje. Existen tres clases:
  - Simple (*Simple*)
  - Compuesta (*Composite*)
  - Englobante (*Enclosing*)
- **Eventos** (*Events*): Los eventos pueden ser externos o internos y representan canales de cadenas de eventos a través de los cuales se generan las instancias de eventos. Entre los externos tenemos las siguientes clases:
  - Periódico (*Periodic*)
  - Unitario (*Singular*)
  - Esporádico (*Sporadic*)
  - Sin límite (*Unbounded*)
  - A ráfagas (*Bursty*)
- **Requerimientos temporales** (*Timing Requirements*): Representan los requerimientos impuestos en el instante de la generación del evento interno asociado. Existen de distintos tipos:
  - *Deadlines*
  - *Max Output Jitter Requirement*
  - *Max Miss Ratio*
- **Manejadores de eventos** (*Event Handlers*): Los manejadores de eventos representan las acciones que son activadas por la llegada de uno o más eventos. Existen distintos tipos de manejadores:
  - *Activity*
  - *System Timed Activity*
  - *Concentrator*
  - *Barrier*

- *Delivery Server*
- *Query Server*
- *Multicast*
- *Rate Divisor*
- *Delay*
- *Offset*

- **Transacciones** (*Transactions*): Una transacción es un conjunto de manejadores y eventos que representan las actividades interrelacionadas ejecutadas en el sistema.

4.2. Modelo MAST de RT-EP

RT-EP es un protocolo de red que está basado en prioridades fijas y, como hemos visto en el apartado anterior, MAST incluye el modelo de una red de prioridades fijas como una clase especial de un *Processing Resource*. El modelo de la red encapsula la información relevante para que se pueda emplear el análisis de planificabilidad. Esta información, como se vio en el apartado anterior, incluye parámetros específicos de red además de parámetros sobre las actividades creadas por el procesador para manejar los paquetes que están agrupados en los *Network Drivers*.

De esta manera podemos usar el modelo MAST para caracterizar RT-EP. Lo lograremos obteniendo los valores adecuados para los parámetros de *Processing Resource* y *Network Driver*. Para obtener una completa descripción del modelo del protocolo debemos extender MAST añadiendo un nuevo *Network Driver* (basado en el *Packet Driver*). Este *driver* incluirá la información sobre las operaciones de mandar y recibir paquetes realizadas por el *thread* principal de comunicaciones, sobre el propio *thread* y sobre las operaciones del protocolo para manejar y pasar el testigo. La información del *Network Driver* para el modelo de RT-EP (que llamaremos *RT-EP Packet Driver*) la describimos en los siguientes sub-apartados con la notación MAST.

4.2.1. Caracterización MAST de RT-EP sin tratamiento de errores

En este apartado describiremos el *RT-EP Packet Driver* definido para el protocolo sin tratamiento de errores, así como los atributos que caracterizan la clase *Fixed Priority Network* del *Processing Resource* del modelo MAST. También se describirá el modelo interno del *driver*.

RT-EP Packet Driver: Es, como el *Character Packet Driver*, una especialización del *Packet Driver* en el que existe un *overhead* adicional asociado al paso de testigo. Está caracterizado por los siguientes atributos:

- *Packet Server*: Es el *Scheduling Server* que está ejecutando el *driver*, que es el *thread* principal de comunicaciones. Está caracterizado por los *Scheduling Parameters*, destacándose la prioridad, y por el procesador que lo ejecuta. La prioridad de este *thread* debe ser más alta que la de los *threads* de la aplicación.
- *Packet Send Operation* (*PSO*): Es la operación que se ejecuta cada vez que se manda un paquete. Corresponde a la actividad ejecutada al pasar del estado *Idle* a *Send_Info*

- *Packet Receive Operation* (*PRxO*): Es la operación que se ejecuta cada vez que se recibe un paquete. Corresponde a la actividad ejecutada al pasar del estado *Idle* a *Recv_Info* y *Send_Initial_Token*.
- *Number of Stations* (*N*): Corresponde al número de estaciones o procesadores conectados a través de la red. Este atributo se usará para evaluar el tiempo consumido en la fase de arbitraje.
- *Token Manage Operation* (*TMO*): Es la operación que se ejecuta para mandar el testigo en los estados *Send_Token* o *Send_Permission*. Se tendrá en cuenta el tiempo del peor caso de estas operaciones.
- *Token Check Operation* (*TCO*): Es la operación que se ejecuta para recibir y chequear un paquete de testigo. Corresponde al estado *Idle* seguido de *Check_Token*.
- *Token Delay* (*TD*): Es un retardo que se introduce en el procesamiento del testigo para así poder reducir el *overhead* de CPU.
- *Packet Discard Operation* (*PDO*): Es la operación que se ejecuta cuando se reciben, debido al modo promiscuo, y descartan tramas dirigidas a otro destino.

Fixed Priority Network: A continuación caracterizaremos este *Processing Resource* con sus atributos especiales:

- *Max Packet Transmission Time* y *Min Packet Transmission Time*: El tiempo máximo y mínimo de transmisión de paquetes incluye el tiempo que se tarda en enviar los bytes de la trama Ethernet con el campo de información máximo y mínimo, que puede ser 1500 en el caso del tiempo máximo de transmisión o 46 en el caso del tiempo mínimo. Como vimos en el apartado 4.1, el tiempo mínimo representa el periodo más corto de los *overheads* asociados a la transmisión de cada paquete. Por lo tanto tiene un gran impacto en los *overheads* que causan los *Network Drivers* al procesador cuando usan la red. El tiempo mínimo corresponde con el tiempo que se tarda en enviar un paquete de testigo o el *Transmit Token*, ya que como se expresó con anterioridad, en el apartado 2.3, “Formato de los paquetes RT-EP”, el campo de información mínimo de una trama Ethernet es de 46 bytes. Aunque los paquetes del protocolo ocupen menos siempre será esa cantidad de información la que se transmitirá. Sabiendo que el número máximo de bytes que se transmiten son 1526 (8 bytes de preámbulo, 6 de dirección MAC destino, 6 de dirección MAC origen, 2 del campo de tipo y 4 del campo FSC) en el caso de un tamaño de datos de 1500 bytes y de 72 bytes en el caso de un tamaño de datos de 46 bytes, podemos obtener:

  Max Packet Tranmission Time:

$$MaxPTT = \frac{1500 \cdot 8}{Rb}$$

  En *MaxPTT* se ha utilizado 1500 bytes que es el tamaño máximo de información que se puede transmitir en una trama. El *overhead* (26 bytes) se tendrá en cuenta en el *Packet Overhead*.

Min Packet transmisión Time:

$$MinPTT = \frac{72 \cdot 8}{Rb}$$

donde *Rb* es el régimen binario del medio (10 Mbps, 100 Mbps, 1 Gbps, 10 Gbps etc...)

- *Packet Overhead* (Peor caso: *PWO*. Medio: *PAO*. Mejor caso: *PBO*): Este es el *overhead* asociado a mandar cada paquete. Teniendo en cuenta la negociación y control que se precisa antes de mandar un paquete de información, el *Packet Overhead* se calcula:

$$(N+1)(MinPTT + TCO + TMO) + N \cdot TD + \frac{26 \cdot 8}{R_b}$$

que corresponde al tiempo que se tarda en enviar un número de testigos igual al número de estaciones, *N*, realizando una completa circulación del testigo más un *Transmit Token*. El tiempo para mandar un testigo se calcula como la suma del tiempo de transmisión mínimo (*MinPTT*), el tiempo *Token Check Operation* (*TCO*) y el tiempo *Token Manage Operation* (*TMO*). A esa cantidad hay que sumarla el retraso *TD* asociado a los *N* testigos enviados. Por último se ha añadido el *overhead* asociado a la transmisión de la cabecera de la trama. Nos interesa el *PWO*, que es el peor caso. Se calcula teniendo en cuenta los peores casos de *TCO* y de *TMO*.

- *Max Blocking*: El máximo bloqueo es causado por la no predecibilidad de los paquetes. En este protocolo se calcula de la siguiente manera:

$$(N) \cdot (MinPTT + TCO + TMO) + (N-1) \cdot TD + MaxPTT + \frac{26 \cdot 8}{R_b}$$

Esta fórmula representa el máximo bloqueo que se produce en la red. Para un mayor entendimiento del máximo bloqueo que puede producirse en la red, se desglosará en una serie de cuatro pasos como se expone en la figura 4.3:

  - **Paso 1:** La *CPU 1* es la *token_master* y comienza el arbitrio de prioridad creando y mandando un testigo. En el momento en que la *CPU 2* ha recibido y actualizado el testigo con la información del elemento de mayor prioridad de su cola de transmisión, recibe por parte de algún *thread* el mensaje para transmitir con la mayor prioridad en toda la red. Este mensaje debería ser el transmitido y no lo será ya que en la información del testigo, que introdujo la *CPU 2*, está la prioridad del mensaje que antes era el de mayor prioridad. El testigo sigue circulando hasta que llega a la *CPU 1* que es la *token_master*. Este paso aporta a la fórmula del bloqueo los términos

$$(N-1) \cdot (MinPTT + TCO + TMO) + (N-1) \cdot TD$$

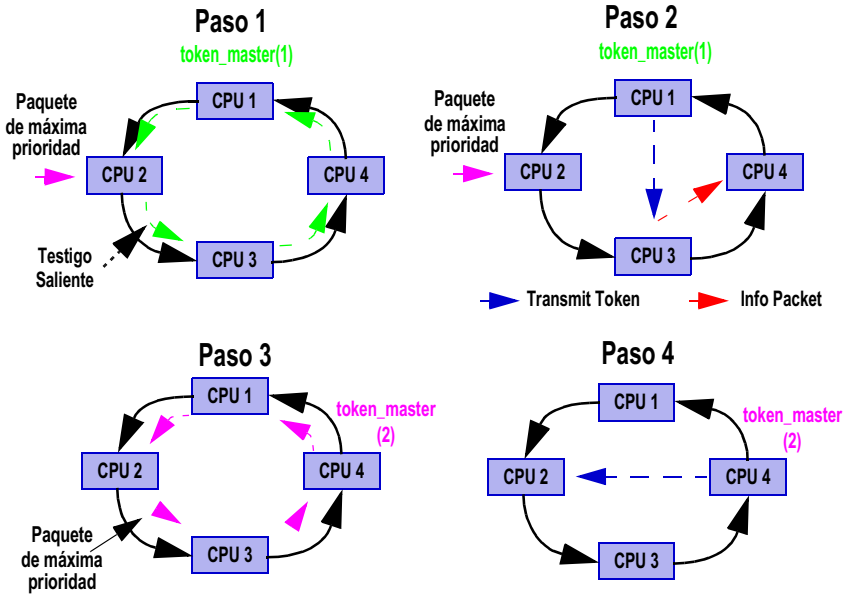


Figura 4.3: Máximo bloqueo en RT-EP.

- **Paso 2**: La *CPU 1* mira en la información de prioridad del testigo y otorga a la *CPU 3* el derecho a transmitir ya que tenía la mayor prioridad en el arbitrio. La *CPU 3* transmite su información, por ejemplo, a la *CPU 4*. Esta información será del tamaño máximo permitido por *Info Packet*, ya que debemos situarnos en el peor caso. La *CPU 4* pasa a ser la nueva *token_master*. Este paso aporta al bloqueo los siguientes términos:

$$(MinPTT + TCO + TMO) + MaxPTT + \frac{26 \cdot 8}{R_b}$$

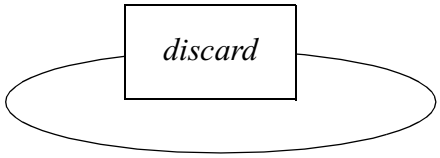
- **Paso 3**: La *CPU 4* crea y distribuye un nuevo testigo, donde ya irá la información del mensaje de mayor prioridad en bloqueo de la *CPU 2*. Este tiempo, y el del paso 4, ya se incluye en el *overhead* del paquete (*Packet Overhead*).
- **Paso 4**: El testigo llega a la *CPU 4*, la *token_master*. Y ésta, observando la información que alberga el testigo, otorga a la *CPU 2* el derecho a transmitir el mensaje, que era el que sufría el bloqueo.

Como ha quedado expuesto, el mensaje de máxima prioridad deberá esperar el tiempo correspondiente al *PWO* más el tiempo *Max PTT* que será el que necesitaremos para transmitir un mensaje de máxima longitud antes de empezar la negociación del nuevo mensaje en la cola. Sumando este valor al del tiempo de la nueva negociación obtendremos el valor del máximo bloqueo. En la figura 4.3, la estación *CPU 1* es la *token_master* cuando se produce el bloqueo. Después del primer arbitrio de prioridad, cuando todavía no se ha contemplado el mensaje de mayor prioridad de la red, se vio que era la *CPU 3* la que debía mandar mensaje a la *CPU 4*. Por ello esta última ha pasado a ser la nueva *token_master* que otorgará a la

CPU 2 el permiso para transmitir, después de otro arbitrio, deshaciendo así el bloqueo. Al resultado obtenido hay que sumarle el retraso asociado a los *N-1* testigos involucrados en el máximo bloqueo.

Caracterización interna del driver: Este modelo se centrará en la actividad del *driver*. A continuación expondremos distintas transacciones que se producen, en cada estación, al utilizar RT-EP como medio de comunicación. Éstas son debidas a:

- **Testigos descartados:** Esta transacción está motivada por el hecho de utilizar el modo promiscuo en las estaciones. Todas las estaciones reciben, en todo momento, todas las tramas arrojadas al medio. Lo que conlleva a una actividad, *discard*, en cada estación que se encargue de descartar todas las tramas que no vayan dirigidas a ella. Debido al método del protocolo, esta actividad se ejecutará periódicamente y su periodo corresponderá, como se ilustra en la figura 4.4, a la suma del mejor caso del cambio de contexto (*CS*) para la ejecución de la actividad *discard*, del mejor caso de la propia actividad *discard*, del tiempo de transmisión del testigo (*MinPTT*), del *delay* programado en el paso de cada testigo y la suma del mejor caso de las operaciones *Token Manage Operation* y *Token Check Operation*.



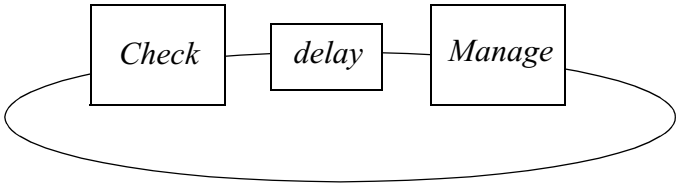
$$T = CS + discard + MinPTT + delay + TMO + TCO$$

Figura 4.4: Transacción de descarte de testigos.

- ***Transmit Token* o *info packet* descartado** (siempre que haya algo que transmitir en la red): De la misma manera que el caso anterior esta transacción corresponderá con el hecho de descartar el testigo especial transmitido o el paquete de información en el único caso en el que no se transmite ese testigo especial, que es cuando a la estación que le corresponde transmitir es la *token_master*, en ese caso se pasa directamente a transmitir el mensaje. Su periodo corresponderá en el "peor" caso a la suma del mejor caso del cambio de contexto (*CS*) para la ejecución de la actividad *discard*, del mejor caso de la propia actividad *discard*, al tiempo mínimo de rotación de testigo, más el mejor caso de la operación *Token Check Operation*, más el mejor caso de las operaciones *Packet Send Operation* o *Token Manage Operation*. Este caso no puede ser simultáneo al testigo normal descartado y tiene un periodo mayor que éste, por lo que el *overhead* estaría ya incluido en el caso anterior.
- **Info Packet descartado** (en el caso de que se transmita información): Esta transacción corresponde con el caso de que se transmita información que no vaya dirigida a la estación, por lo que se descartará. Su periodo corresponderá a la suma del cambio de contexto (*CS*) para la ejecución de la actividad *discard*, de la propia actividad *discard*, del tiempo mínimo de paso de testigo, más la suma del mejor caso de las operaciones *Token Manage Operation* y *Token Check Operation*, más el tiempo de

transmisión del paquete *Transmit Token* (*Min PTT*), más el mejor caso de la operación *Packet Send Operation*. Este caso no puede ser simultáneo al testigo normal descartado y tiene un periodo mayor que éste, por lo que el *overhead* estaría ya incluido en el caso del testigo descartado.

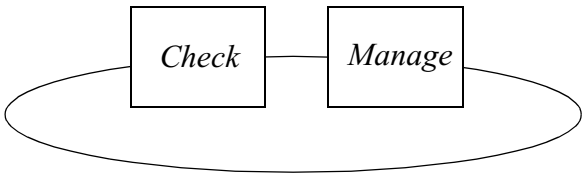
- **Paso del testigo** (sin intervenir en la comunicación): Esta situación está definida por aquellas estaciones que reciben, actualizan y mandan el testigo sin intervenir en la comunicación de información en ese paso de testigo. El "peor" caso ocurre cuando se transmiten testigos de prioridad 0. Esta transacción engloba al conjunto de actividades, ilustradas en la figura 4.5, que se encargarán de chequear y volver a mandar el testigo en la red y el *delay* programado en el paso de testigo. Su periodo corresponderá al tiempo mínimo de rotación del testigo.



T = Tiempo mínimo de rotación del testigo

Figura 4.5: Transacción del paso de testigo.

- ***Transmit Token* siendo *token_master*** (sin intervenir en la comunicación): En este caso tenemos una transacción, ilustrada en la figura 4.6, que contendrá una actividad que se ejecutará cada vez que vuelva el testigo a la estación *token_master*. Su periodo será la suma del mejor caso de las operaciones *Token Check Operation* y *Token Manage Operation*, más el tiempo de transmisión del *Transmit Token* (*Min PTT*), más el tiempo de *Packet Send Operation*, más el tiempo de transmitir un mensaje de tamaño mínimo (*Min PTT*), más el tiempo mínimo de rotación del testigo. Todo esto suponiendo que volviese a ser la *token_master*.



$T = TCO + TMO + 2 \cdot MinPTT + PSO +$
Tiempo mínimo de rotación del testigo

Figura 4.6: Transacción del *Transmit Token*

- **Estación transmisora o receptora de información**: Estos casos se añaden a las operaciones de envío y recepción respectivamente.

De esta manera el protocolo RT-EP se modela con MAST sin tener en cuenta errores en el medio o en estaciones.

4.2.2. Caracterización MAST de RT-EP con tratamiento de errores

En el apartado anterior vimos cómo podíamos modelar RT-EP con MAST para su posterior análisis con la aplicación. Podríamos calificar al modelado anterior como un modelado optimista, ya que no tiene en cuenta en su análisis posibles pérdidas de testigos o datos debido al ruido o fallos en el sistema. Para modelar RT-EP, teniendo en cuenta los fallos, se hará referencia y se ampliarán atributos y parámetros definidos anteriormente. Como ya se indicó en el apartado 2.4, "Tratamiento de errores en RT-EP", solo vamos a considerar en este estudio el caso de pérdida de paquetes como error a tener en cuenta; no se tratarán ni el caso de una estación ocupada ni el caso de fallo de una o más estaciones.

Debemos modificar el *RT-EP Packet Driver* añadiendo dos nuevas operaciones y dos atributos. También es necesario recalcular los valores del *Fixed Priority Network*:

RT-EP Packet Driver: Modifica algunas las operaciones del modelo sin fallos y añade dos operaciones y dos parámetros:

- *Packet Server*: Es el mismo que en el modelo sin fallos.
- *Packet Send Operation* (*PSO*): Ahora corresponderá a los estados *Idle* seguido de *Send_Info* y de *Error_Check*.
- *Packet Receive Operation* (*PRxO*): También cambia un poco correspondiendo ahora a los estados *Idle* seguido de *Recv_Info*, *Send_Initial_Token* y *Error_Check*.
- *Number of Stations* (*N*): Permanece inalterado.
- *Token Manage Operation* (*TMO*): También ahora tenemos en cuenta el peor caso de *Send_Token* o de *Send_Permission*, pero al peor de los dos le sumamos el estado *Error_Check*.
- *Token Check Operation* (*TCO*): Permanece inalterado.
- *Token Delay Operation* (*TDO*): Permanece inalterado.
- *Packet Discard* (*PD*): Permanece inalterado.
- *Token Transmission Retry* (*TR*): Aquí se indicará el número máximo de fallos (y sus correspondientes retransmisiones) que permitiremos en cada vuelta del testigo en la fase de arbitrio de prioridad.
- *Packet Tranmission Retry* (*PR*): Con este valor se indicará el número máximo de retransmisiones que se producirán en un fallo al transmitir un *Info Packet*.
- *Timeout* (*T*): Corresponde al tiempo de *timeout* configurado en el protocolo.
- *Token Retransmission Operation* (*TRO*): Es la operación que se ejecuta cada vez que se retransmite un testigo ya sea un *Transmit Token* o un testigo normal. Corresponde con la parte del estado *Error_Handling* que se encarga de retransmitir el testigo.
- *Packet Retransmission Operation* (*PRO*): Esta otra operación se ejecuta cada vez que se retransmita un paquete de información. Corresponde con

la otra parte del estado *Error_Handling* que se encarga de retransmitir el paquete de información (*Info Packet*).

Fixed Priority_Network: Este Processing Resource hereda y adapta las características del modelo sin fallos:

- *Max Packet Transmission Time* y *Min Packet Transmission Time*: Permanecen inalterados ya que las retransmisiones no afectan a este valor:

  Max Packet Tranmission Time:

$$MaxPTT = \frac{1500 \cdot 8}{Rb}$$

  Min Packet transmisión Time:

$$MinPTT = \frac{72 \cdot 8}{Rb}$$

- *Packet Overhead* (*PWO*,*PAO*,*PBO*): Ahora tenemos en cuenta los reintentos que afectarán a todo el conjunto, ya que en el modelo de fallos suponemos que todas las estaciones fallan de acuerdo a la información proporcionada por *Token Transmission Retry* (R_i). Por ello el *Packet Overhead* se calculará:

$$(N+1)(MinPTT + TCO + TMO) + (N \cdot TD) +$$

$$(MinPTT + TRO + T) \cdot TR + \frac{26 \cdot 8}{R_b}$$

  Como vemos el fallo se ha incluido como la suma de diferentes reintentos en cada estación. Una aproximación a este *Packet Overhead* es suponer que en una vuelta del testigo, como máximo (dada la probabilidad de error en una LAN), se producirá un fallo en un solo paquete y por lo tanto se efectuará una sola retransmisión. De considerar esta alternativa, se reduciría considerablemente el *Packet Overhead*. Pero hay que tener en cuenta que estaríamos excluyendo al peor caso de nuestro modelo, un caso que se puede producir aunque sea muy improbable en un funcionamiento normal de la red.

- *Max Blocking*: En este caso, con fallos, el máximo bloqueo que sufrirá un mensaje teniendo en cuenta el fallo o pérdida de los distintos paquetes que se generan y sus correspondientes retransmisiones se puede calcular como:

$$(N)(MinPTT + TCO + TMO) + ((N-1) \cdot TD) +$$

$$\left(MaxPTT + \frac{26 \cdot 8}{R_b} + PR \cdot \left(PRO + T + MaxPTT + \frac{26 \cdot 8}{R_b}\right)\right) +$$

$$(MinPTT + TRO + T) \cdot TR$$

Como puede verse se considera también la pérdida del paquete de información. De la misma manera que en el caso anterior se podría hacer la aproximación de una sola pérdida y, por consiguiente, una sola retransmisión. Aunque con esta consideración, como vimos antes, se excluye el peor caso posible.

Caracterización interna del driver: En esta caracterización se modificarán las transacciones "paso de testigo" y "*Transmit Token*", descritas en el caso sin error, para incluir los efectos causados. Se añaden a las actividades correspondientes dos actividades más, previa espera de un *timeout*, como se ilustra en la figura 4.7. La identificación del error producido conlleva a la actividad *Check* y su recuperación a la actividad *Handle*. Estas actividades junto con el tiempo de espera se repetirán *Token Transmission Retry* (*TR*) veces hasta que no haya error. La duración de la operación corresponderá al tiempo de *timeout* más el peor caso de *Token Retry Operation*, que se ha dividido en la figura en dos partes: *check* y *handle*. Se deberá añadir el tiempo correspondiente a estas actividades a los periodos calculados en el caso sin error.

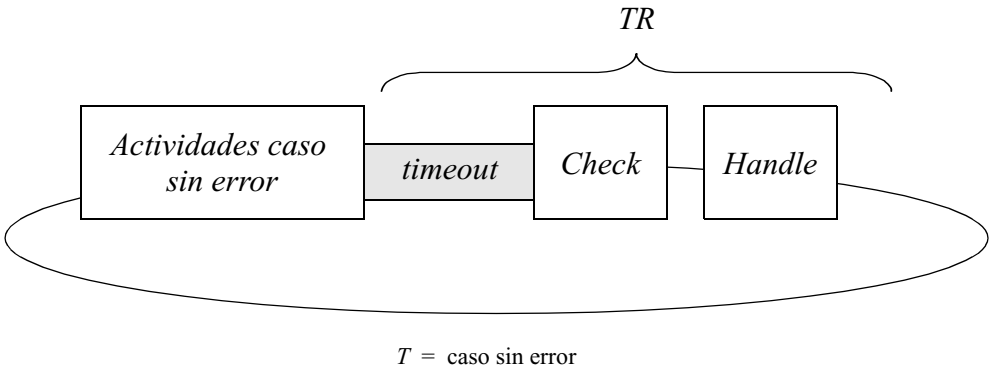


Figura 4.7: Transacción en caso de error.

5. Medidas de prestaciones

5.1. Introducción a las pruebas sobre RT-EP

En este capítulo comentaremos las pruebas más relevantes que se han efectuado al protocolo así como las medidas de tiempos más significativos, que es la mejor caracterización de un protocolo de Tiempo Real.

Primeramente comentar que todos los programas de test se han incluido en el *Makefile* del proyecto del protocolo. Así estarán accesibles por el implementador en caso de necesitar medir o probar alguna implementación de RT-EP, como puede ser por ejemplo, el establecimiento del mejor tiempo de *timeout* en función de la red presente. Serán muy útiles en combinación con los distintos modos de depuración del protocolo. Por último, indicar que se distinguen estos programas de prueba por empezar su nombre por *test_*.

También es necesario indicar que se analizó RT-EP con un analizador de protocolos para comprobar su correcto funcionamiento en cuanto a tramas transmitidas en el medio. El analizador de protocolos por software que se utilizó fue *Ethereal*.

Ethereal es un analizador de protocolos gratuito que está disponible en *http://www.ethereal.com* en su versión para entornos UNIX y Windows. Permite examinar y capturar en un archivo el tráfico existente en la red. Además de dar una completa información de los protocolos encapsulados en las tramas capturadas en el medio, posee una gran capacidad de filtrado para un análisis más concienzudo.

Pasaremos a una enumeración y descripción de los programas generales de prueba que se han efectuado para, posteriormente, indicar en los sucesivos apartados los programas de prueba más específicos.

- **test_config**: Programa que comprueba la integridad del archivo de configuración. Lo único que hace es leer el archivo de configuración, que no tiene por que ser necesariamente *lnk.config* sino el que esté configurado, y escribir las estaciones por la salida estándar indicando si ha tenido alguna dificultad a la hora de leerlo.
- **test_monitor**: Programa de prueba que comprueba la cola de prioridades accediendo a través del monitor.
- **test_queue**: Realiza una prueba sobre la cola de prioridades.
- **test_times**: Realiza una comprobación en la que se puede observar la ventaja de utilizar las funciones *addrcmp* y *addrcpy* definidas en el módulo *prot_types* frente a *memcmp* y *memcpy.*
- **test_mono_thread_master** y **test_mono_thread_slave**: Se trata de un *thread* periódico que se dedica a mandar un mensaje a través de un canal que es recibido por otra máquina y contesta con un mensaje fijo de la

máxima prioridad. Las direcciones de las máquinas deberán ser adecuadamente asignadas en los parámetros correspondientes. Están declaradas mediante la cláusula *#define*, en cada uno de los fuentes. El archivo *test_mono_thread_master* deberá ejecutarse en la estación que figure como primera *token_master* en el archivo de configuración, es decir, la que ocupe el primer lugar. Y *test_mono_thread_slave* en todas las demás estaciones. Este programa muestra, por cada paquete transmitido, el tiempo que se tarda en transmitir un paquete y recibir contestación.

Se recomienda el uso de estos programas de test en combinación de los distintos modos de depuración para comprobar el protocolo antes de realizar cualquier medición temporal de éste.

Acerca de las mediciones, diremos que como este protocolo de momento sólo está implementado en Linux, la cuantía de las mediciones obtenidas no son de mucha relevancia ya que GNU/Linux no es un sistema operativo de Tiempo Real y por tanto, sus respuestas no están acotadas en el tiempo. Por ello no podemos tener en cuenta ni el peor ni el mejor tiempo en las mediciones. Debido a este problema nos hemos quedado con el tiempo medio de todas las mediciones para formarnos una idea de la magnitud de los tiempos presentes en el sistema. Éste es un problema del sistema operativo y no de la configuración o programas de *test* del protocolo para medir tiempos. Una vez que el protocolo se implemente en MaRTE conseguiremos mejores y peores tiempos exactos sin necesidad de efectuar cambios en el código fuente de los programas de *test*. También hay que comentar que los tiempos se han medido en una red compuesta de *cinco ordenadores Pentium II a 266 MHz conectados en red a una velocidad binaria de 10 Mbps*. Conviene resaltar que se han hecho todas las pruebas del protocolo con el *delay* puesto a 0 para poder medir los tiempos con mejor precisión, ya que en GNU/Linux el *delay* tiene escasa resolución (± 10 ms). Ahora sin más pasaremos a comentar el método y los tiempos medidos en los siguientes apartados.

5.2. Cálculo de los *Overhead* de CPU

Para realizar el cálculo del *Overhead* de CPU asociado a los distintos estados del protocolo, pudiendo proceder a la caracterización de su modelo MAST, se crearon los siguientes módulos:

- **test_multi_thread_master**: Consiste en cinco *threads* periódicos que se ejecutan con distinto periodo y mandan mensajes fijos a distintos canales de una estación que tiene otros cinco *threads* que reciben esa información y devuelven un mensaje constante. Debe ser ejecutado en la primera estación *token_master*. De la misma manera que en el caso de *test_mono_thread_master*, se deberá especificar, en el código fuente, la dirección destino de los paquetes.
- **test_multi_thread_slave**: Son cinco threads que reciben a través de un canal un mensaje y lo contestan con otro mensaje fijo. Se ejecutará en el resto de las estaciones. De forma análoga a *test_multi_thread_master* se deberá especificar la dirección destino en el fuente.

Como vemos, el que hace todo el trabajo en este caso es el módulo *test_multi_thread_master* que se integra junto con los modos de depuración del protocolo, de forma que si no está definido ningún modo de depuración, muestra en la salida estándar

los paquetes enviados y recibidos y el tiempo que han tardado. Pero sin embargo, si está definido el modo de depuración de medir o bien el paso de testigo o bien los *overheads* de CPU, ya no se realiza ninguna salida estándar para que no se vean afectados los tiempos medidos.

Para poder medir los *overheads* de CPU deberemos situar al protocolo en el modo CPU_OVERHEAD_PROTOCOL_TIMES, para de este modo acceder a la auto-medición de los distintos estados. Si comentamos el parámetro CPU_FILE_OUTPUT en *prot_types.h* nos muestra las medidas del peor caso, medio y mejor en la salida estándar. Una vez realizado este cambio en la configuración deberemos compilar los módulos *test_multi_thread_master* y *test_multi_thread_slave* y lanzarlos en las máquinas convenientes. Cuando se haya procesado el número de tramas correspondiente a MAX_CPU_POINTS se mostrarán los tiempos medidos por la salida estándar. Debido a lo comentado anteriormente sólo expondremos los tiempos medios medidos en la figura 5.1.

Operaciones	Tiempo (μs)
Idle State (+ Error_Check)	8.73
Send_Initial_Token	51.02
Check_Token	2.03
Send_Permission	64.29
Send_Token	42.77
Send_Info	59.98
Recv_Info	70.25

Figura 5.1: Overheads de CPU de RT-EP

El protocolo, por exactitud en el cálculo, no reporta medidas del estado *Error_Check* en solitario*,* sino que la medida que da el protocolo del estado *Idle* ya contiene al estado *Error_Check.* Esto es debido a que siempre va acompañado de éste en todos los cálculos. Así se ha adoptado la decisión de medirlos conjuntamente frente a añadir más *overhead* de medida en el estado *Error_Check*. También es importante comentar que, como es normal, el protocolo no da ningún valor para el estado *Error_Handling*. Esto es debido a que la situación normal de funcionamiento no es una situación de error. Por tanto en situaciones normales no es posible medir el valor de este estado. Además, para el modelo, necesitamos medidas del estado desglosadas en la parte de retransmisión de testigo y retransmisión de información. Una cota para estos tiempos la obtenemos de la siguiente forma siguiendo la notación del modelo MAST:

- *Token Retransmission Operation* (*TRO*): Tendrá como cota superior el *overhead* asociado a *Check_Token* más el peor caso de los estados *Send_Token* o *Send_Permission* más el *timeout* del protocolo.
- *Packet Retransmission Operation* (*PRO*): Tendrá como cota superior el *overhead* asociado al estado *Send_Info* más el *timeout* del protocolo.

Con estos resultados se pueden calcular los parámetros del modelo MAST del protocolo visto en el capitulo 4, “Modelado MAST”. Aunque, como ya hemos expresado con anterioridad, no serán valores exactos al no estar medidos sobre un sistema operativo de Tiempo Real como es MaRTE.

5.3. Cálculo del *Packet Overhead*.

Adicionalmente al cálculo formal del atributo *Packet Overhead* a partir de los tiempos del protocolo, se ha definido un modo de depuración en el protocolo para su cálculo experimental. Este modo se puede utilizar para obtener una estimación de este valor. El tiempo medido será una aproximación del *overhead* deseado porque, en realidad, al medir en el modo TOKEN_CHECKING_PRIORITY_TIMES lo que hacemos es obtener el tiempo que tarda el testigo en dar una vuelta completa al anillo. Por lo tanto, para obtener el valor del *Packet Overhead* nos falta el tiempo que se tarda en mandar el paquete adicional *Transmit_Token*. No podemos medir de forma directa este último paquete ya que para hacerlo necesitaríamos que los relojes de la estación que transmite el paquete y la que lo recibe estuviesen sincronizados, situación que no es posible.

Por tanto para tener una aproximación experimental de este valor, ejecutamos la pareja *test_multi_thread_master* y *test_multi_thread_slave* una vez que hayan sido compilados con el modo TOKEN_CHECKING_PRIORITY_TIMES del protocolo activado. Experimentalmente se ha obtenido el valor expuesto en la figura 5.2.

Operación	Tempo (μs)
Tiempo del paso de testigo	1058

Figura 5.2: Estimación del Packet Overhead para cinco estaciones.

Si aplicamos la fórmula obtenida en el modelo MAST del protocolo con los tiempos obtenidos del *overhead* de CPU nos saldría un *Packet Overhead* de unos 800 us. Entonces, si hemos comentado que en esta medición experimental nos falta el tiempo de procesar y enviar el *Transmit Token*, ¿cómo es posible que nos salga un tiempo mayor, 1058 us, si hemos tenido en cuenta un caso menos? Esta posible incongruencia tiene una fácil explicación y es que los tiempos medidos en este apartado corresponden a testigos con información de prioridad, esto es, testigos que se han creado accediendo a la cola de prioridades realizando una comprobación en el campo correspondiente del testigo recibido y mandándolo a la estación sucesora. En cambio en los valores de los *overheads* de CPU que se han utilizado para el cálculo del *Packet Overhead* se ha tomado la media de todos los testigos enviados, que en su mayoría son testigos que, poseyendo una prioridad cero, circulan en todo momento esperando encontrar alguna estación que tenga algún mensaje que transmitir. El procesado de estos testigos es más rápido que en el caso de tener algún mensaje que transmitir y, por tanto, al tomar el valor medio ocurre esta singularidad. Los valores que se deberían tener en cuenta, en el caso de haberlo ejecutado en MaRTE o en algún otro sistema operativo de Tiempo Real, son los del peor caso. De haberlo hecho nos habría dado un valor superior al del peor caso del tiempo del paso de testigo calculado en este apartado. Aun así, si estamos tratando con tiempos medios (que no es el tiempo que deberíamos tener en cuenta), es preferible quedarnos con el valor calculado en este apartado, frente a los 800 us del apartado anterior, ya que el cálculo que efectúa siempre es con los testigos que requieren un mayor *overhead*.

5.4. Cálculo del *Max Blocking*.

Se ha creado otra pareja análoga a *test_multi_thread_master / _slave* que se han denominado *test_packets_master* y *test_packets_slave*. Se encargarán de medir el bloqueo producido por la acción del protocolo. Si su configuración es adecuada se puede medir experimentalmente el bloqueo máximo producido. Aun así, para el cálculo exacto

del bloqueo máximo deberemos recurrir a la expresión obtenida en el modelo MAST del protocolo. Procederemos a la explicación del funcionamiento de este módulo ya que conseguir medir este tiempo requiere un poco de planificación:

- *test_packets_slave*: Este programa se encarga de lanzar dos *threads* que poseen dos acciones básicas: recibir información cada uno de un canal y transmitir la misma información recibida de vuelta por los mismos canales a la estación configurada como destino.
- *test_packets_master*: En este programa se define un *thread* periódico, de periodo largo que intentará transmitir mensajes de tamaño mínimo pero de una prioridad grande.

Ahora que conocemos el funcionamiento debemos configurar adecuadamente los módulos para conseguir medir el bloqueo. Retomando la figura 4.3, página 58, y con la ayuda de la figura 5.3, podemos aclarar como debe ser la configuración. El módulo *test_packets_master* debe ejecutarse en la estación que sea la primera *token_master* que en nuestro caso será la *CPU 2* por lo tanto, y para que el bloqueo se produzca como en la figura 4.3, se debe considerar la primera rotación del anillo como un tiempo de calentamiento. Una vez completada esa primera rotación, la *CPU 1* pasará a ser la *token_master* reproduciendo la situación de la figura anteriormente citada. En el resto de las estaciones se ejecutará *test_packets_slave*. El módulo *test_packets_master* señalará como dirección destino de los paquetes de alta prioridad y mínimo tamaño la *CPU 4*. Se configurará la *CPU 4* para que devuelva todo paquete recibido, con la misma prioridad, a la *CPU 2*, la *CPU 1* para que sólo reciba paquetes y la *CPU 3* para que sólo mande paquetes de tamaño máximo y prioridad mínima a la *CPU 1*. De esta manera se consigue que la *CPU 1* sea siempre la *token_master*, hasta que se transmita el mensaje de alta prioridad por parte de la *CPU 2*, ya que siempre está recibiendo información. Se consigue que *test_packets_slave* se dedique sólo a transmitir o sólo a recibir comentando, antes de compilar en las respectivas estaciones, la función de recibir o enviar en el código fuente.

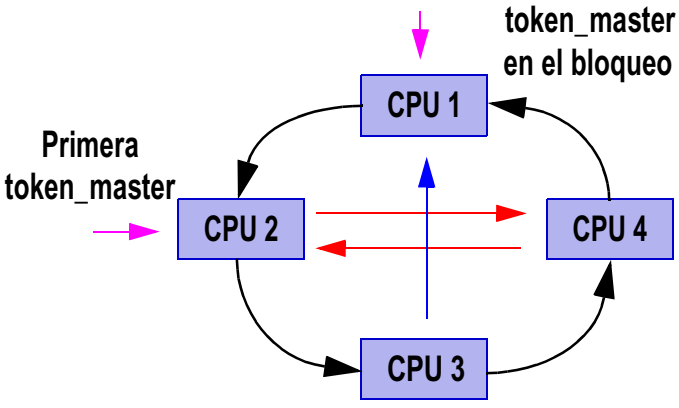


Figura 5.3: Composición del anillo para el máximo bloqueo

La *CPU 4* pasará a ser la nueva *token_master* cuando la *CPU 2* la transmita información. Ésta devolverá un mensaje con la misma prioridad a su destino, la *CPU 2*, para que podamos contabilizar cuánto ha tardado en transmitirse ese mensaje (tiempo desde que mandamos un paquete hasta que lo recibamos dividido entre dos). Seguidamente al no haber ningún mensaje de alta prioridad que transmitir se seguirán

transmitiendo los paquetes desde la *CPU 3* a la *CPU 1* volviendo ésta última a ser la *token_master* repitiéndose la situación inicial.

De esta manera el bloqueo del peor caso medido corresponderá a una cota superior del bloqueo máximo que se producirá en el sistema siempre que tomemos un número elevado de medidas. Volvemos a decir que el bloqueo medido es una estimación experimental y que el verdadero bloqueo se calcula con la expresión del modelo, aunque el bloqueo medido estará próximo a éste.

Otra aclaración importante es que se ha medido el bloqueo como el tiempo que se tarda en mandar un mensaje de alta prioridad cuando la red está siendo intensamente ocupada por mensajes de baja prioridad. Este valor incluye más pasos que cualquier caso de bloqueo ya que incluye la parte de transmisión del mensaje de alta prioridad. Por eso mismo, el tiempo del peor caso se puede tomar como una cota superior del *Max Blocking*.

El tiempo medido se muestra a continuación y puede servir para darnos una idea del bloqueo medio sufrido por un mensaje de alta prioridad:

Operación	Tempo (μs)
Tiempo para transmitir un mensaje de alta prioridad	1667

Figura 5.4: Bloqueo medio para cinco estaciones.

6. Conclusiones

En este capítulo expondremos una serie de conclusiones que se derivan del trabajo expuesto en los capítulos anteriores.

A lo largo del trabajo hemos podido examinar una serie de decisiones de diseño e implementación de un protocolo de Tiempo Real cuyo propósito es lograr que el medio Ethernet sea determinista para la aplicación. A continuación expondremos cuáles son los principales logros de RT-EP:

- Hemos presentado una implementación software de un protocolo de comunicaciones multipunto para aplicaciones de Tiempo Real sobre Ethernet, que no requiere ninguna modificación del hardware Ethernet existente.
- Se aporta una nueva visión del paso de testigo sobre un bus, en la que las estaciones no transmiten información de forma más o menos periódica debido al paso de testigo sino dependiendo de la prioridad de los mensajes que almacene. Es la prioridad de los mensajes almacenados y no el testigo lo que da permiso para transmitir. Se solucionan así los problemas clásicos que el paso de testigo presenta en las redes de Tiempo Real debido a que se reduce la larga latencia en los mensajes de alta prioridad. Se aumenta el procesado de los mensajes asíncronos y se reduce considerablemente la inversión de prioridad que se da en el paso de testigo.
- Evita totalmente las colisiones con lo que conseguimos predecibilidad sobre Ethernet sin ningún cambio adicional sobre el hardware.
- El protocolo está basado en prioridades fijas y por ello podemos usar herramientas para el análisis de planificabilidad de prioridades fijas.
- Se ha obtenido un preciso modelo temporal del protocolo, el cual nos permite realizar un análisis de planificabilidad de una aplicación distribuida usando este protocolo.
- Es un protocolo con detección de errores que además hereda todas las características de enlace de Ethernet. Además aplica mecanismos de recuperación ante los errores de: pérdida de un paquete, fallo de una estación o el caso de una estación que tarda en responder. Sólo se respeta el comportamiento de Tiempo Real para la pérdida de paquete; en los otros dos casos el protocolo continua funcionando después de un breve espacio de tiempo.
- Se ha conseguido una red de comunicaciones de Tiempo Real bastante rápida, cada vez más ya que está en constante desarrollo. Actualmente Ethernet alcanza unos regímenes de 10 Gbps y se está trabajando en el estándar de los 100 Gbps. Además es muy popular y económica.
  Retomando los datos de CANbus y Ethernet expuestos en el apartado 1.4,

"Objetivos de este proyecto", se puede calcular, a modo de comparación, el tiempo empleado en transmitir dos mensajes de 200 y 1000 bytes (que son valores habituales en aplicaciones de control industrial) a una distancia de 30 metros (despreciamos el tiempo de propagación), por ejemplo.
En CAN el mensaje se troceará en paquetes de 8 bytes con 46 bits de overhead por cada paquete. El régimen binario es de 1 Mbps (a una distancia menor de 40 metros). Podemos calcular el tiempo total de transmisión, suponiendo que durante todo el tiempo que dura la transmisión es el mensaje de mayor prioridad a transmitir:

- **Caso 200 bytes de información:** En este caso tenemos que transmitir 25 paquetes con un *overhead* de 46 bits por paquete lo que nos hace transmitir 25 paquetes de 110 bits. Por lo tanto, nos da un tiempo total de transmisión de:

$$(25 \cdot 110 / 10^6) + 25 \cdot T_{PROCESADO} = 2,75_{ms} + 25 \cdot T_{PROCESADO}$$

- **Caso 1000 bytes de información:** Aquí tenemos que transmitir 125 paquetes de 110 bits cada uno:

$$(125 \cdot 110 / 10^6) + 125 \cdot T_{PROCESADO} = 13,75_{ms} + 125 \cdot T_{PROCESADO}$$

En el caso de RT-EP el tiempo de transmisión depende del número de estaciones involucradas en la comunicación, suponemos un número medio de éstas, es decir, unas 20 estaciones, y un *delay* configurado a 0. Por lo tanto:

- **Caso 200 bytes de información:** El *overhead* corresponde a una cabecera de 26 bytes de Ethernet y 8 del *Info Packet* de RT-EP, previa circulación del testigo que supone 72 bytes por las 20 estaciones a una velocidad de 10 Mbps, es:

$$8 \cdot \frac{(200 + 26 + 8)}{10^7} + \frac{(20 + 1) \cdot 72 \cdot 8}{10^7} + 21 \cdot T_{PROCESADO} = 1,39_{ms} + 21 \cdot T_{PROCESADO}$$

- **Caso 1000 bytes de información:** Aquí el *overhead* es el mismo que en el caso anterior ya que se manda toda la información en un solo paquete contenido en una sola trama (el límite del protocolo es de 1492 bytes):

$$8 \cdot \frac{(1000 + 26 + 8)}{10^7} + \frac{(20 + 1) \cdot 72 \cdot 8}{10^7} + 21 \cdot T_{PROCESADO} = 2,03_{ms} + 21 \cdot T_{PROCESADO}$$

Se propone la figura 6.1 como muestra orientativa del tiempo que se tarda en CAN y en las distintas alternativas, económicamente atractivas (10 y 100 Mbps), de Ethernet en transmitir los ejemplos vistos anteriormente pero variando el número de estaciones. El tiempo está expresado en milisegundos. Como aproximación del tiempo de procesado para RT-EP se han utilizado los valores medios calculados en la figura 5.1, página 66, de modo que el tiempo de procesado corresponderá al tiempo del estado *Idle* + *Check Token* + *Send Permission* (75,05 us) en el caso del testigo

Transmit Token y al tiempo de los estados *Idle* + *Check Token* + *Send Token* (53,53 us) en el caso de los testigos normales. Para CAN se han supuesto tiempos de procesado nulos ya que no se han podido medir.

	10 estaciones		**20 estaciones**		**50 estaciones**	
Información (bytes)	**200**	**1000**	**200**	**1000**	**200**	**1000**
CAN bus	2.75	13.75	2.75	13.75	2.75	13.75
RT-EP - 10 Mbps	1.43	2.07	2.54	3.17	5.87	6.51
RT-EP - 100 Mbps	0.69	0.75	1.28	1.34	3.06	3.12

Figura 6.1: Comparativa CAN vs RT-EP

Con esta comparación superficial, ya que son protocolos muy dispares, se puede comprobar que RT-EP ofrece mejores tiempos de transmisión para un número medio de estaciones y que éste se degrada a medida que aumentan las estaciones. Por el contrario CAN ofrece las mismas prestaciones con independencia del número de estaciones, pero el tamaño del mensaje enviado incide seriamente en el tiempo de transmisión. También cabe resaltar que 100 Mbps en Ethernet, actualmente, es una opción económica. Su uso con RT-EP hace que se obtengan mejores resultados.

- No es un protocolo propietario de MaRTE, sino que valdría para cualquier otro sistema operativo de Tiempo Real ya que la información que hay que suministrar al protocolo para su correcto funcionamiento no es ninguna otra que no se deba tener en cuenta antes de montar un sistema de Tiempo Real estricto.
- La implementación se ha cuidado para que, en la medida de lo posible, se pueda controlar el nivel de enlace usando funciones POSIX para configurar las estaciones, mandar, recibir paquetes, etc. De esta forma conseguimos que sea bastante portable a nivel del código fuente.
- Se ha proporcionado un juego de programas de prueba y de medición para que actúen como ayuda a la hora de configurar los parámetros del protocolo sobre una red concreta y sobre los parámetros de su modelado.

7. Trabajo futuro

7.1. Introducción

En este capítulo expondremos distintos trabajos y criterios que pueden emplearse en un futuro para hacer más flexible y extenso el uso del protocolo. Además de una descripción de estos, se propondrán distintas alternativas para su implementación.

Aun así, hay que dejar constancia que el trabajo futuro más importante será la implementación de RT-EP en MaRTE.

También se propone la posibilidad de portar el protocolo a entornos *Wireless LAN* que pueden ser muy útiles en el ámbito industrial.

7.2. Implementación de RT-EP en MaRTE

Como se ha venido expresando a lo largo del trabajo, se ha realizado un esfuerzo por conseguir que la implementación de RT-EP en MaRTE sea sencilla. Para ello, y por la manera en que el protocolo usa la red, no haría falta una implementación completa de *sockets* debido a que el único protocolo que accederá a la red será RT-EP y es él mismo el que ya proporciona el medio de iniciar una comunicación de red mediante *init_comm* y un punto de acceso a la aplicación. Sólo hace falta que el S.O. proporcione un mecanismo para poder recibir a través de una interfaz (un sustituto de *socket* y *bind*), funciones para enviar (un sustituto de *sendto*) y recibir a través de esa interfaz (un sustituto de *recvfrom*). MaRTE tampoco necesitaría procesar las tramas recibidas ya que tal como las reciba el driver de red pueden ser pasadas al protocolo.

También se necesitara reemplazar las llamadas a *ioctl* por las correspondientes en MaRTE.

Por tanto se propone una implementación del protocolo de manera que forme parte del driver de red. Se conseguirá así que la parte de arbitrio del protocolo (ver *Main Communication Thread* de la figura 3.2, página 23) esté próxima a las tramas recibidas/ enviadas por el driver mientras que ofrece una interfaz de comunicaciones a la aplicación.

7.3. Tráfico *Multicast.*

La primera y quizá más importante ampliación corresponderá a permitir, en el protocolo, tráfico *broadcast* y sobretodo tráfico *multicast* tanto a nivel de enlace como a nivel de aplicación. En un medio compartido como es Ethernet, en donde una trama transmitida por una estación es escuchada por todas, se definen, además de las direcciones individuales de las estaciones, dos tipos de direcciones adicionales que son las de *broadcast* y las de *multicast*:

- **Broadcast**: Es una dirección especial que corresponde con FF:FF:FF:FF:FF:FF y se utiliza para enviar una misma trama a todos los dispositivos conectados a la red. Esto permite a la estación emisora direccionar todos los demás dispositivos de una sola vez en lugar de enviar un paquete a cada puesto destinatario.
- **Multicast**: Se utiliza un *multicast*, a diferencia de *broadcast*, para enviar una misma trama a un grupo definido de dispositivos conectados a la red. Esto permite a la estación emisora direccionar un grupo de dispositivos de una sola vez en lugar de enviar un paquete a cada puesto destinatario. Una dirección multicast es del tipo 01:xx:xx:xx:xx:xx, es decir, es como una dirección MAC normal solo que tiene el bit menos significativo del octeto más significativo puesto a 1.

Más interesante que el tráfico *broadcast*, es el tráfico *multicast* ya que es de gran importancia en situaciones productor / consumidores. Se reduce enormemente el número de tramas arrojadas al medio.

Además del tráfico *multicast* en el nivel de enlace, también sería muy interesante implementar *multicast* a nivel de aplicación, es decir, que también se puedan mandar mensajes a múltiples *threads* dentro de la misma estación con un solo paquete de información. De esta manera, por ejemplo, enviando un solo paquete de información podemos hacer que llegue a múltiples estaciones y a múltiples *threads* dentro de las estaciones. Para lograr esto necesitamos dos niveles de *multicast*, uno en el nivel de enlace definido anteriormente y otro a nivel de aplicación.

Con la situación actual del protocolo, sólo es posible mandar mensajes a las direcciones configuradas en el anillo lógico. Si se consiguiera mandar un mensaje a una dirección *broadcast*, presentaría un error en la lógica actual. Según la lógica del protocolo, la estación que recibe el mensaje pasa a ser la nueva *token_master*. Esto aplicado al caso de *broadcast* o *multicast* daría una situación de múltiples estaciones con el papel de *token_master* lo cual es una situación de error.

Una posible solución para dar soporte *multicast* al protocolo (consideraremos el *broadcast* como un caso especial del *multicast*), es que, una vez definidas las direcciones *multicast* con el grupo de estaciones que queramos incluir (ya que Ethernet permite direcciones *multicast*), utilizaremos el campo *Source Address*, figura 2.3, página 14, de la trama Ethernet para suministrar la dirección de la estación que deba asumir el papel de *token_master* dentro del grupo *multicast*. Este campo no es utilizado por el protocolo Ethernet, sino que se provee para dar información adicional a los niveles superiores [7].

Otra posible solución, y más directa que la anterior, podría ser que suministrásemos los grupos *multicast* dentro del archivo de configuración del anillo lógico. Así, todas las estaciones conocerían su posición dentro del grupo o grupos *multicast* a los que está asociada. Por ejemplo, sería la primera estación del grupo *multicast* la única que se encargaría de asumir el papel de *token_master* una vez que se detectase que la dirección destino de la trama es la dirección *multicast* a la que pertenece la estación en primer lugar. Esta última solución parece la más sencilla de implementar y es la que se propone como solución para RT-EP.

En cuanto al *multicast* en el nivel de aplicación, se pueden definir una serie de *canales de grupos*. Estos canales harán referencia al grupo de *threads* destino. Estos grupos se configurarían a priori y serían conocidos por la estación. Se deberá modificar el

formato de la trama de información para que añada un campo de dos bytes *Channel Group,* por ejemplo, que indique el grupo *multicast* al que va dirigido el paquete. El *thread* de comunicaciones internamente se encargará de hacer efectiva la correspondencia del grupo con los *threads* y hacer llegar el mensaje a sus respectivas colas de recepción.

7.4. Interconexión de redes.

Siempre puede ser necesario aumentar la dimensión de una red de área local. Un problema al añadir distintos enlaces a la LAN es que podemos sacrificar la respuesta temporal de nuestro sistema. Es por ello que se proponen unas opciones para poder aumentar la longitud de nuestra LAN:

- **Repetidores**: Los repetidores realizan la interconexión a nivel físico. Su función consiste en amplificar y regenerar la señal, compensando la atenuación y distorsión debidas a la propagación por el medio físico. Son, por consiguiente, transparentes al subnivel MAC y a niveles superiores. También estarían incluidos en este ámbito los concentradores o *hubs*. Éstos disponen de un bus posterior, denominado plano posterior (backplane) que hace las funciones de medio compartido. Se pueden colocar en topología de árbol conformando un único medio. Las características más significativas de los repetidores, también extensibles a los concentradores, son:
    - Permiten incrementar la longitud de la red.
    - Operan con cualquier tipo de protocolo, ya que sólo trabajan con señales físicas.
    - No procesan tramas por lo que el retardo es mínimo.
    - Son de bajo coste, debido a su simplicidad.
    - El número total de repetidores que se pueden incorporar en una red está limitado por la longitud máxima de la misma debido a la arquitectura que puede llegar hasta varios kilómetros en Ethernet dependiendo de la tecnología utilizada.
    - No aislan tráfico, es decir, el ancho de banda del medio está compartido por todas las estaciones, independientemente de la sección de la red en que estén ubicadas, es decir, comparten el mismo dominio de colisión.
- **Puntes (Bridges)**: Los puentes son elementos que operan a nivel de enlace. En consecuencia son más complejos que los repetidores o *hubs* y son naturalmente más costosos. Sus características más significativas son:
    - Permiten aislar tráfico entre segmentos de red.
    - Operan transparentemente al nivel de red y superiores.
    - No hay limitación conceptual para el número de puentes en una red.
    - Procesan las tramas, lo que aumenta el retardo.
    - Utilizan algoritmos de encaminamiento, que pueden generar tráfico adicional en la red dependiendo del algoritmo utilizado. Si es encaminamiento transparente no genera tramas adicionales pero

en cambio, si es encaminamiento fuente, o algún derivado de éste, sí las genera

- Filtran las tramas por dirección física y por protocolo.

La incorporación de repetidores o *hubs* para alargar la extensión de la red, como opera a nivel físico, no introducirá modificaciones en RT-EP porque se mantiene el único dominio de colisión. Pero como se ha visto con anterioridad tiene una limitación de longitud.

El problema del uso de puentes radica, además de introducir retardos al procesar las tramas, en que aislan los dominios de colisión. Esto puede ser un problema en la implementación actual del protocolo ya que si se manda una trama a una dirección MAC de una estación conectada a otro puente, la estación origen nunca podrá saber si la ha recibido correctamente ya que no escuchará a la estación destino transmitir una trama válida, lo que suponía una confirmación al paquete transmitido. Esta situación la hará pensar que ha habido error y reintentará la transmisión.

Es posible efectuar una modificación al protocolo para conseguir que se pueda utilizar con puentes. Esta modificación consiste en que, cuando una estación haya recibido correctamente una trama, mande un paquete de confirmación (ACK) a la dirección de *broadcast* que es recibida por todas las estaciones en el medio. Con esta situación deberemos tener en cuenta que tener en cuenta, en el modelado, el tiempo de procesado añadido por los puentes y el *overhead* que supondrá el nuevo paquete de confirmación a cada paquete enviado. Esta solución, además de añadir más *overhead* al protocolo nos plantea un problema que es la detección de la pérdida de una trama de confirmación. Habría que idear algún método para su detección.

Otra posible solución es modificar los campos de dirección destino y origen en la trama Ethernet, figura 2.3, página 14, al transmitir el paquete de testigo, el *transmit token* y el paquete de información. En la transmisión de los paquetes del protocolo se deberá utilizar como dirección destino la dirección de *broadcast*, así el paquete llegará a todas las estaciones independientemente del dominio en el que estén. El campo de dirección de origen deberá rellenarse con la dirección de la estación destinataria de la trama. Como ya se expresó anteriormente, este campo no es utilizado por el protocolo Ethernet y lo podemos utilizar sin problemas para nuestro propósito. De esta manera todas las estaciones recibirán todas las tramas y mediante una comprobación del campo origen sabrán si está destinadas a ellas o no. Este método es compatible con el tráfico *broadcast* y *multicast*. Se pueden emplazar estas direcciones en el campo origen y es la estación que recibe la trama la que decide, a nivel del protocolo, si debe o no aceptarla. Esta sería la mejor solución para el caso de múltiples enlaces ya que no añade *overhead* adicional al protocolo como la propuesta anterior y se pueden detectar errores sin añadir ninguna modificación a los estados correspondientes. El inconveniente principal de esta solución es que, al procesar parcialmente todas las tramas transmitidas en el medio, se aumenta el *overhead* de CPU. Además, no se aísla el tráfico que es lo que se desea con el uso de puentes, con este método se conseguiría un efecto similar al del repetidor sin la limitación en longitud.

Por último, en el caso de querer aislar segmentos de red utilizando RT-EP, se propone crear nuestro propio puente que consistiría en una estación operando con MaRTE OS con dos interfaces de red, cada una conectada a cada segmento y que esa estación implementase las funciones de un puente respetando el protocolo de paso de testigo en ambos segmentos. Cada segmento poseerá su propio arbitrio, de necesitar transmitir algún

mensaje entre los dos segmentos se realizará en dos fases (aunque será transparente para la estación transmisora), una consistirá en transmitir de la estación origen al *bridge* y la segunda del *bridge* a la estación destino. En ambas fases se someterá el mensaje al arbitrio de cada segmento. El puente conocerá las estaciones conectadas a cada segmento para poder realizar su labor. Con la incorporación de este puente se deberá redefinir el modelado MAST del protocolo.

7.5. Implementación en Wireless LAN

Como su nombre indica una Wireless LAN es una red de área local "sin cables". Su utilización está motivada por el incremento de estaciones móviles, como puede ser un robot, que no tienen un emplazamiento fijo y que están en movimiento.

Existen tres tipos distintos de medio físico [3]:

- **Infrarrojos**: Los sistemas infrarrojos son simples en diseño y por lo tanto baratos. Los sistemas IR sólo detectan la amplitud de la señal de manera que se reducen en gran medida las interferencias. Estos sistemas presentan un gran ancho de banda y por ello pueden alcanzar velocidades mayores que cualquier otro sistema. El gran inconveniente es que, una gran interferencia (por el sol o luces fluorescentes) puede derivar en la inutilización de la red.
- **Microondas**: Los sistemas en microondas operan a una baja potencia. La gran ventaja es el gran *througthput* que tiene.
- **Radiofrecuencia**: Es el más popular, usa espectro ensanchado para la comunicación, y así consigue una gran inmunidad ante las interferencias, pero obtiene los peores regímenes binarios en comparación con IR o microondas.

Wireless LAN utiliza un formato de tramas específico distinto al de Ethernet como se puede consultar en el estándar [16]. Para asegurar la coexistencia de LANs mixtas, existe un dispositivo especial llamado *Portal*. Un portal por lo tanto es la integración lógica entre LANs cableadas y 802.11 (Wireless LAN). Por lo tanto funciona como un puente entre redes inalámbricas y cableadas.

En cuanto al acceso al medio, Wireless LAN utiliza un acceso de contienda pero no CSMA/CD como en Ethernet ya que resultaría inapropiado para este tipo de redes en donde la recuperación de una colisión es muy costosa debido a los problemas en su detección. Por ello se utiliza CSMA/CA (Carrier Sense Multiple Access with Collision Avoidence) cuyo máximo logro estriba en que intenta evitar que se produzcan colisiones, aunque éstas se producen con baja probabilidad. CSMA/CA funciona de la siguiente manera:

- La estación escucha el medio antes de transmitir.
- Si alguien está transmitiendo, espera un periodo de tiempo y lo vuelve a intentar.
- Si no hay nadie transmitiendo en el medio, manda un mensaje corto llamado *Ready To Send* (RTS). Este mensaje contiene la dirección de la estación destino y la duración de la transmisión. Así las otras estaciones ya saben cuánto tienen que esperar hasta intentar volver a transmitir.

- La estación destinataria envía un mensaje corto llamado *Clear To Send* (CTS) indicando a la estación que ya puede transmitir.
- Cada paquete tiene su confirmación. Si no se recibe alguna confirmación, la subcapa MAC retransmite los datos.

Para mas información sobre Wireless LAN se recomienda consultar el estándar [16].

En cuanto a la implementación de RT-EP sobre Wireless LAN hay que decir que sólo habría que efectuar modificaciones menores en el manejo de tramas, ya que son distintas [16]. Por lo demás la implementación sería bastante directa. Además como el protocolo evita totalmente las colisiones y es fiable en cuanto a errores o tramas perdidas, conseguiría una gran eficiencia.

Esta solución es factible si las estaciones están relativamente cerca y conforman un único dominio de colisión, es decir, todas las estaciones escuchan el mismo medio. Éste no será el caso normal ya que según vayamos alejando las estaciones vamos necesitando el uso de *access points*, *bridges* o *antenas* que harán que se formen distintos dominios de colisión. De la misma manera, otro problema que se puede plantear en RT-EP es el caso de cuando no tengamos una red Wireless pura, es decir, tengamos parte Wireless y parte Ethernet. En este sentido, el problema se presenta porque, como se ha expresado con anterioridad, no tenemos un solo dominio de colisión ya que existe una especie de puente que es el Portal que se encargará de adaptar los dos sistemas. Una solución posible puede ser la misma que la propuesta en el 7.4, “Interconexión de redes.”, en donde se proponía la confirmación mediante un ACK a la dirección de *broadcast* de cada trama recibida o bien el envío de todas las tramas del protocolo a la dirección de *broadcast*. Se recomienda la implementación de esta última opción en el protocolo para afrontar esta situación en entornos Wireless.

Otra posible solución será la creación de los puentes RT-EP comentados en el 7.4, “Interconexión de redes.”

Bibliografía y referencias

[1] Aldea, M. González, M. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, Lecture Notes in Computer Science. LNCS 2043. Mayo 2001

[2] Amsbury, W. "Data Structures. From Arrays to Priority Queues" Wadsworth, 1985.

[3] Balakrishna, S. “Network Topologies in Wireless LAN” IFSM 652 Diciembre 1995

[4] Bosch, Postfach 50, D-700 Stuttgart 1. “CAN Specification, version 2.0 edition”. 1991

[5] CENELEC. EN 50170, “General Purpose Field Communication System, Vol. 1/3 (P-NET), Vol. 2/3 (Profibus), Vol. 3/3 (FIP)”. Diciembre 1996

[6] CENELEC. EN 50254, “Interbus-S, Sensor and Actuator Network for Industrial Control Systems”

[7] Charles E. Spurgeon. “Ethernet: The definitive Guide”. O’Reilly Associates, Inc. 2000.

[8] Court, R. “Real-Time Ethernet”. Computer Communications, 15 pp.198-201. Abril 1992

[9] Decotignie, J-D. “A Perspective on Ethernet as a Fieldbus” FeT’01, 4th Int. Conf. on Fieldbus Systems and their Applications. Nancy, France. Noviembre 2001.

[10] Drake, J.M. González Harbour, M. Gutierrez, J.J. and Palencia, J.C. “Description of the MAST Model” http://mast.unican.es/

[11] Drake, J.M. González Harbour, M. Gutierrez, J.J. and Palencia, J.C. “UML-MAST Metamodel, specification, example of application and source code” http://mast.unican.es/umlmast

[12] González Harbour, M. “POSIX de tiempo real” Grupo de computadores y tiempo real. Dpto. de electrónica y computadores. Universidad de Cantabria. Octubre 2001.

[13] González Harbour, M. Gutiérrez, J.J. Palencia, J.C. and Drake, J.M. “MAST: Modeling and Analysis Suite for Real-Time Applications”. Proceedings of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001

[14] IEEE Std 802.3, 2000 Edition: “IEEE Standard for Information technology--Telecommunications and information exchange between systems--Local and metropolitan area networks--Common specifications--Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications”

[15] IEEE Std 802.4-1990. “IEEE Standard for Information technology--Telecommunications and information exchange between systems--Local and metropolitan area networks--Common specifications--Part 4: Token-Passing Bus Access Method and Physical Layer Specifications”.

[16] IEEE Std 802.11-2001 “IEEE Standard for Information technology--Telecommunications and information exchange between systems--Local and metropolitan area networks--Common specifications--Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications”

[17] ISO/IEC 9899 (1999). ISO/IEC Standard 9899:1999. “Programming languages -- C”

[18] ISO/IEC 9945-1 (1996). ISO/IEC Standard 9945-1:1996. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]. Institute of Electrical and electronic Engineers.

[19] Jae-Young Lee, Hong-ju Moon, Sang Yong Moon, Wook Hyun Kwon, Sung Woo Lee, and Ik Soo Park. “Token-Passing bus access method on the IEEE 802.3 physical layer for distributed control networks”. Distributed Computer Control Systems 1998 (DCCS'98), Proceedings volume from the 15th IFAC Workshop. Elsevier Science, Kidlington, UK, pp. 31-36, 1999.

[20] Kurose, J.F. , Schwartz, M. y Yemini, Y. “Multiple-access protocols and time constrained communication” Computing Surveys, 16(1), pp.43-70, Marzo 1984

[21] Kweon, S-K. y Shin, K.G. “Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing” IEEE Real-Time Technology and Applications Symp, pp.90-100. Washington DC, USA Junio 2000

[22] LeLan, G. Rivierre, N. “Real-Time Communications over Broadcast Networks: the CSMA-DCR and the DOD-CSMA-CD Protocols. INRIA Report RR1863. 1993

[23] Loosemore, S. et al. “The GNU C Library Reference Manual” Draft Edition 0.09. Agosto 1999 (http://www.gnu.org)

[24] Löser, J. Härtig, H. “Real-Time on Ethernet using off-the-shelf Hardware”. Proceedings of the 1st Intl. Workshop on Real-Time LANs in the Internet Age Viena, Austria, Junio 2002

[25] Malcolm, N., Zhao, W. “Advances in hard real-time communication with local area networks”. Local Computer Networks, 1992. Proceedings., 17th Conference on. pp: 548 -557. 1992

[26] Malcolm, N. y Zhao, W. "Hard Real-Time Communication in Multiple-Access Networks ", Real-Time Systems, Vol 8, No 1, Enero 1995, pp. 35-78.

[27] Martínez, J.M. González Harbour, M. Gutiérrez, J.J. “A Multipoint Communication Protocol based on Ethernet for Analyzable Distributed Real-Time Applications” 1st Intl. Workshop on Real-Time LANs in the Internet Age. (RTLIA 2002) Satellite Event to 14th Euromicro Conference on Real-Time Systems. Vienna, Austria June 2002

[28] Nichols, B. Buttlar, D. y Proulx Farrell, J. "Pthreads Programming". O’Reilly & Associates, Inc. 1996.

[29] Pedreiras, P. Almeida, L. Gar, P. “The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency”. Proceedings of the 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, Junio 2002.

[30] POSIX.13 (1998). IEEE Std. 1003.13-1998. Infomation Technology - Standarized Application Environment Profile- POSIX Realtime Applications Support (AEP). The Institute of Electrical and Electronics Engineers.

[31] “Real-Time Executive for Multiprocessor Systems: Reference Manual”. U.S. Army Missile Command, redstone Arsenal, Alabama, USA, January 1996.

[32] Sha, L. Rajkumor, R. Lehoczky, J.P. “Real-Time Computing with IEE Futurebus+” IEE Micro 11 (Junio 1991).

[33] Shimokawa, Y. Shiobara, Y. “Real-Time Ethernet for Industrial Applications” IECON’85, pp.829-834. 1985

[34] Silberschatz, A. Galvin, P. “Operating System Concepts” Addison-Wesley, 1998

[35] Stallings, W. “Local & Metropolitan Area Networks” Prentice Hall 5º Edicion 1990

[36] Tanenbaum, A.S. “Computers Networks”. Prentice-Hall, Inc. 1997

[37] Tomesse, J-P. “Fieldbus and Interoperability”. Control Engineering Practice, 7(1), pp.81-94. 1999

[38] Tovar, E. Vasques, F. “Real-time fieldbus communications using Profibus networks”. Industrial Electronics, IEEE Transactions on. Vol: 46 6, pp.1241 -1251. Dic. 1999

[39] Tovar, E. Vasques, F. “Using WorldFIP networks to support periodic and sporadic real-time traffic”. Industrial Electronics Society, 1999. IECON '99 Proceedings. The 25th Annual Conference of the IEEE. Vol: 3 , pp.1216 -1221 vol.3. Año 1999

[40] Venkatramani, C. Chiueh, T. “Supporting Real-Time Traffic on Ethernet” IEEE Real-Time System Symposium. San Juan. Puerto Rico. Diciembre 1994

[41] Yodaiken V., “An RT-Linux Manifesto”. Proceedings of the 5th Linux Expo, Raleigh, North Carolina, USA, May 1999.

Índice de figuras

1.1.Sistema de computo en entorno industrial 2
2.1.Anillo lógico 11
2.2.Máquina de estados de RT-EP para cada estación 13
2.3.Formato de la trama Ethernet 14
2.4.Formato del paquete de testigo 16
2.5.Formato del paquete de información 17
2.6.Comportamiento de la estación una vez detectado fallo 19
2.7.Máquina de estados de RT-EP con detección de fallos 20
2.8.Resumen del funcionamiento del estado Error Handling 20
3.1.Fases en el desarrollo del protocolo 22
3.2.Funcionalidad y arquitectura de RT-EP 23
3.3.Problema clásico lectores-escritores 32
3.4.Estructura de datos: Arbol 33
3.5.Heap binario 33
3.6.Proceso a seguir para insertar un elemento 34
3.7.Proceso a seguir para extraer un elemento 34
3.8.Representación del *heap* como un array 34
3.9.Módulos y dependencias del protocolo 36
4.1.Sistema de Tiempo Real compuesto de transacciones [10] 51
4.2.Elementos que definen una actividad [10]. 52
4.3.Máximo bloqueo en RT-EP. 58
4.4.Transacción de descarte de testigos. 59
4.5.Transacción del paso de testigo. 60
4.6.Transacción del *Transmit Token* *60*
4.7.Transacción en caso de error. 63
5.1.Overheads de CPU de RT-EP 66
5.2.Estimación del Packet Overhead para cinco estaciones. 67
5.3.Composición del anillo para el máximo bloqueo 68
5.4.Bloqueo medio para cinco estaciones 69
6.1.Comparativa CAN vs RT-EP 72