DEPARTAMENTO DE
ELECTRÓNICA Y COMPUTADORES

UNIVERSIDAD DE CANTABRIA


Trabajo de Investigación

# RT-EP:
# PROTOCOLO DE COMUNICACIONES DE TIEMPO REAL SOBRE ETHERNET

José María Martínez Rodríguez
Septiembre - 2004

*A mis padres.*

*A mis hermanos.*

**AGRADECIMIENTOS**

A mi tutor, Michael González Harbour, por su constante interés e inestimable colaboración durante la realización de este Proyecto de investigación.

Al Departamento de Electrónica y Computadores en general, por el excelente trato recibido.

A mis padres y hermanos por su incondicional apoyo e interés.

A mis amigos por acompañarme, apoyarme y por esos tan buenos ratos que pasamos juntos y que hacen que todo sea más llevadero.

Muchas Gracias.

# Índice

# 1. Introducción

En este trabajo se presenta el desarrollo, implementación, modelado y métricas de un protocolo de tiempo real sobre Ethernet en un sistema operativo de tiempo real. Además también se muestra su implementación, modelado y métricas en un demostrador que es un brazo telemanipulado.

## 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 [42].

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. [24]. En la figura 1.1, y a modo de ilustración, ponemos un ejemplo de una red de área local en un 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 [39]. 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 [19].

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

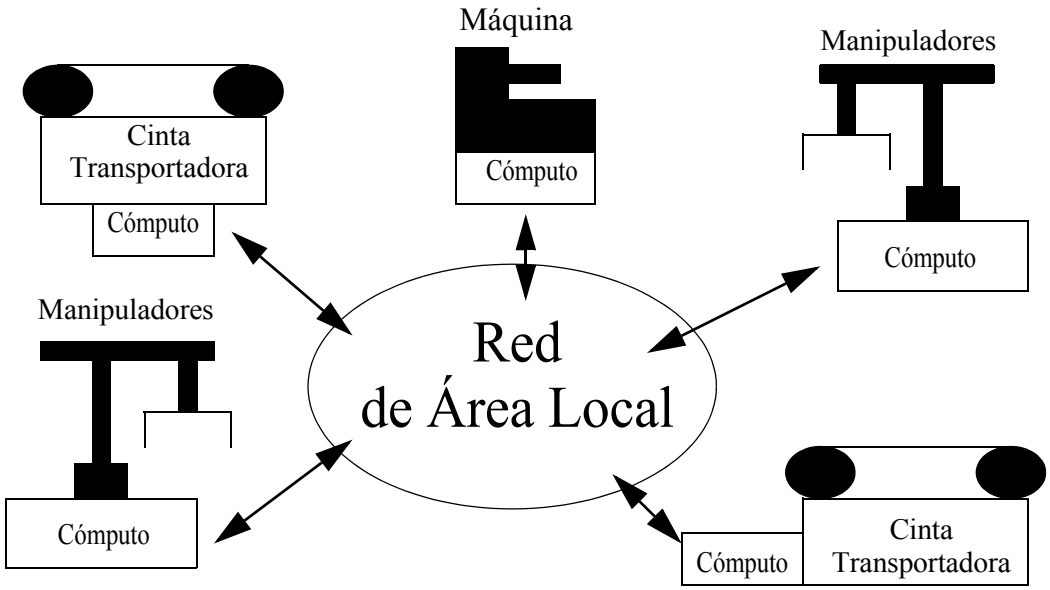



**Figura 1.1: Sistema de computo en entorno industrial**

(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 [43] 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 [43]. 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 [44].

- **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 y por último a 10 Gbps [15]. 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 [18], [21], [8] y [36]. Hacemos hincapié especial en los trabajos [18] y [21]:

- En [18] 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 [21] 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 [32] 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 [16]. 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 [38]. 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 [46]. 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 [25] 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 [20] 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 [23].

# 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 [33] 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 [34], no han sido diseñados siguiendo el modelo POSIX desde el principio sino mas bien solo ofrecen la interfaz externa. Otra alternativa como RT-Linux [47] que ofrece actualmente parte de la interfaz de Tiempo Real mínimo POSIX.13 no es apto para sistemas empotrados pequeños, aunque se está avanzando mucho en esta línea, 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 un sistema operativo de Tiempo Real.
- 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 sobre un sistema operativo de tiempo real concreto, MaRTE OS, para confirmar su validez.
- 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) [26].

Comparando RT-EP con Token bus [16], 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 iguales o superiores a los *fieldbuses* ya que el protocolo pretende ser una alternaniva a estos. Tomando CANbus como representante de los buses de campo y haciendo algunos números para estimar sus prestaciones, 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 47 bits de overhead (como mínimo). 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.

Es necesario aclarar que en [26] la implementación no se realizó sobre MaRTE debido a que todavía no estaban preparados los drivers de red para este sistema operativo. Se eligió GNU/Linux para su desarrollo ya que supone un entorno de programación afín. En este trabajo la implementación se ha realizado siguiendo las pautas exigidas por un sistema operativo de tiempo real y, en este caso, las caracteristicas de MaRTE OS.

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

## 1.5. Implementación en MaRTE OS

En estos momentos MaRTE ya está preparado para realizar la implementación del protocolo. Se ha optado por implementar el protocolo como una librería del sistema que ofrece una interfaz C y Ada a la aplicación.

Ha sido necesario desarrollar una serie de extensiones a MaRTE OS para que la implementación sea posible, las más significativas son:

- **Librería de acceso al PCI**: El PCI es el bus más utilizado actualmente en la arquitectura x86 para la comunicación con los periféricos. En el desarrollo de drivers para dispositivos con interfaz PCI es necesario el desarrollo de un mecanismo para poder acceder al dispositivo dentro del bus, concretamente al espacio de configuración del bus. Por ello se ha desarrollado en este trabajo una librería de acceso al PCI dándonos la posibilidad de realizar drivers para dispositivos PCI en MARTE OS.
- **Desarrollo drivers de red**: Debido a que el protocolo supone una modificación del control de acceso en Ethernet es necesario tener una red por debajo del protocolo. Este punto exige el desarrollo de unos drivers para

distintas tarjetas de red en MaRTE OS. En este trabajo se realizaron drivers para las tarjetas de red:

- **SiS 900 y SiS 7016**.
- **Intel EtherExpressPro100 y derivadas con chipset 82559 y 82562EM**

Conviene comentar que no sólo se han realizado la implementación de unos drivers de red para MaRTE OS sino que se ha proporcionado un *framework* que proporciona un acceso común a los drivers.

Con las características del protocolo y su implementación en un sistema operativo de Tiempo Real podremos utilizar la red para comunicaciones con requisitos de Tiempo Real.

# 1.6. Caracterización del protocolo

No sólo se ha realizado la implementación del protocolo en el sistema operativo MaRTE OS sino que además se ha caracterizado mediante un modelo de tiempo real y unas métricas de su funcionamiento sobre una arquitectura determinada.

La caracterización del protocolo se ha basado en su modelado con la herramienta MAST [10]. Los parámetros medidos en el protocolo corresponden con operaciones y atributos necesarios para poder ser analizado el protocolo junto con la aplicación que lo utilice. De esta manera, no sólo se han extraído de manera analítica los distintos parámetros necesarios (máximo bloqueo, overhead asociado a transmitir un mensaje etc...), sino que además estos parámetros se han medido para ratificar la validez de las formulas extraídas.

Además, debido a las características del protocolo es necesario proponer una extensión a los drivers de red que proporciona MAST para poder analizar de una manera más cómoda a la aplicación que utilice la red con RT-EP.

Otras métricas de utilidad para el implementador son las que se han extraído de la utilización de la CPU por el protocolo en función de un parámetro interno configurable y el ancho de banda (régimen binario) efectivo que nos proporciona el protocolo.

De todas las métricas efectuadas se proporcionan las formulas generales a las que están asociadas. De esta manera se posibilita realizar el diseño de la aplicación a priori con bastante exactitud.

# 1.7. Demostrador del protocolo

Con objetivo de demostrar la aplicación práctica de la tecnología desarrollada, se ha comprobado el funcionamiento del protocolo en un demostrador real. El demostrador consiste en un brazo telemanipulado y la implementación consistirá en un control distribuido para el brazo. Con un control distribuido sobre una red Ethernet y sobre un sistema operativo de tiempo real es posible confirmar la validez del protocolo y se facilita la transferencia de la tecnología a posibles empresas interesadas.

La implementación del demostrador consiste en dos CPUs, una conectada al brazo y que será la encargada de realizar operaciones sobre éste y la otra conectada a unos mandos que serán los encargados de interactuar con la persona que maneje el brazo. Ambas CPUs están conectadas a través de una red Ethernet. No sólo se ha realizado la

**Figura 1.2: Brazo Telemanipulado**

implementación sino que además se procedió a su modelado y a la extracción de unas métricas para demostrar la validez de utilización del protocolo.

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 implementación y evaluación del demostrador. Por último comentaremos los distintos trabajos futuros que pueden deribar de este trabajo.

# 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. Todas ellas 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.

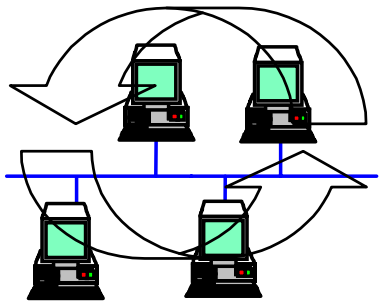


**Figura 2.1: Anillo lógico**

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.

.

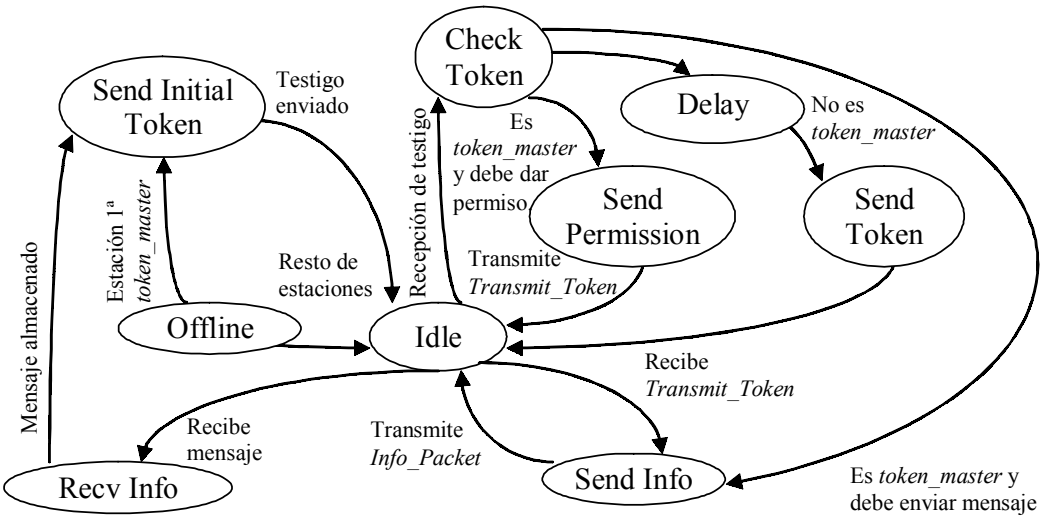



**Figura 2.2: Máquina de estados de RT-EP para cada estación**

- **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 | 46-1492 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 [15] 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 [35] 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 más 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.

De la misma manera, no se ha tenido en cuenta el caso denominado "*babling idiot*" que consiste en que una estación inunde el medio con ruido. Para resolver este caso sería necesario recurrir a la redundancia. Aunque también cabe destacar que debido a que el acceso al medio es controlado por el protocolo solo se podría dar por una aplicación que acceda directamente al medioo por un hardware defectuoso.

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

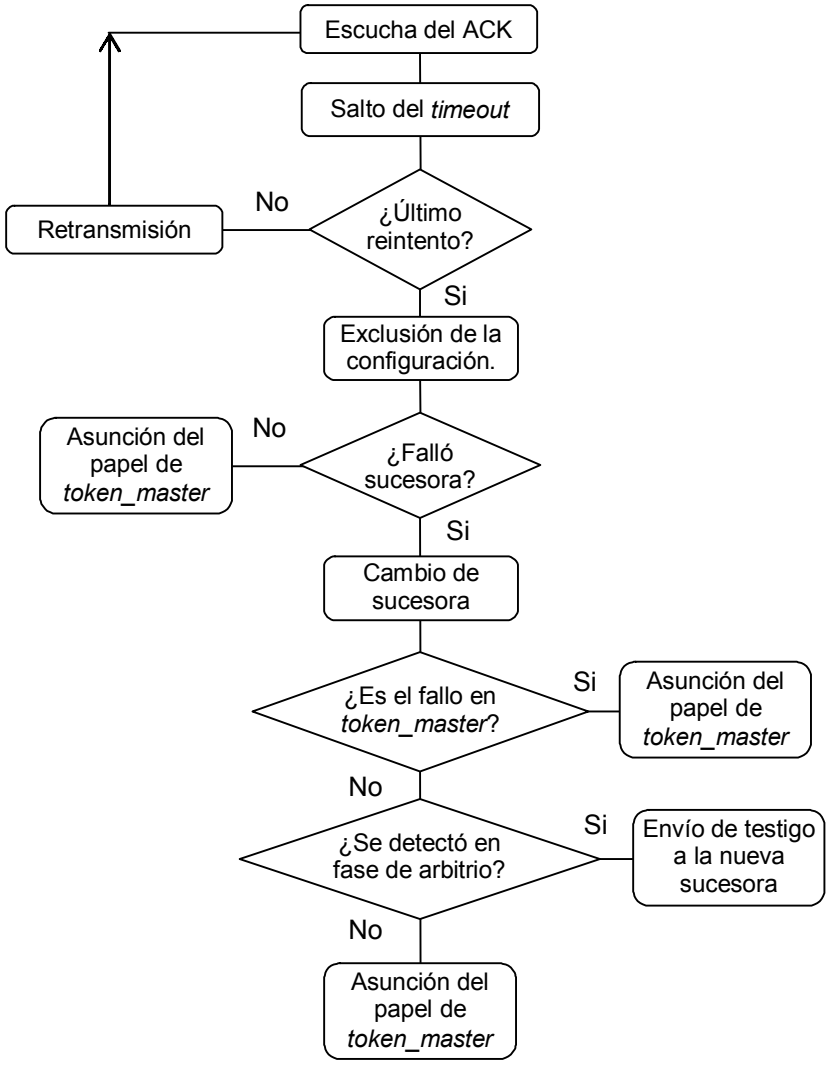



**Figura 2.6: Comportamiento de la estación una vez detectado fallo**

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.

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.

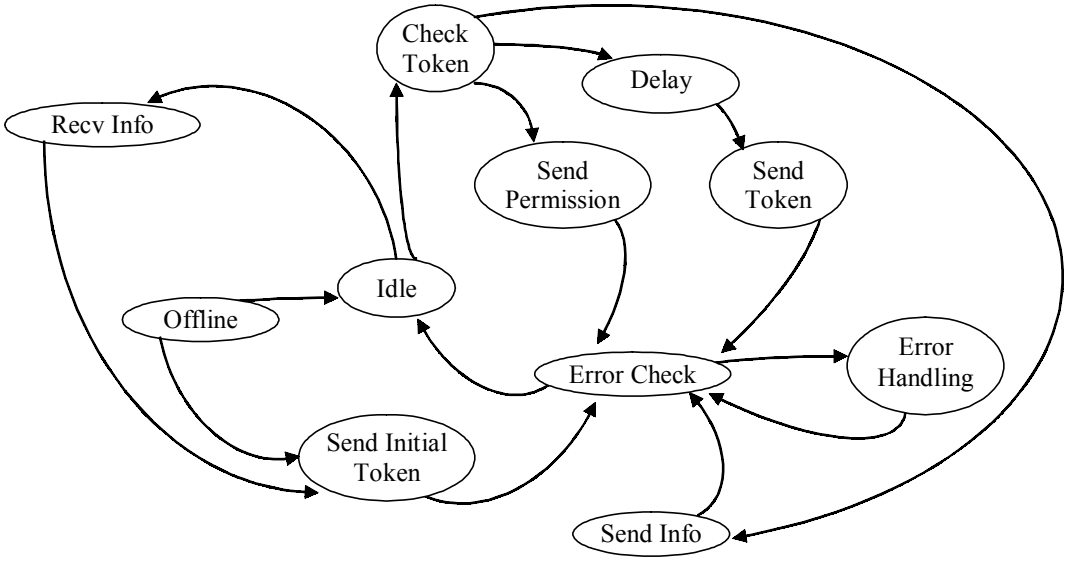



**Figura 2.7: Máquina de estados de RT-EP con detección de fallos**

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 ocurre

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. Extensiones a MaRTE OS

Debido a la ausencia de drivers de red en MaRTE OS ha sido preciso, en el proceso de implementación del protocolo, proporcionar varios drivers de red para poder utilizar el protocolo. Como se verá más adelante se ha intentado proporcionar un *framework* para los distintos drivers de red adaptándolos a las llamadas *create, open, read, write, close, ioctl*.

Así mismo fue necesario la creación de una librería de acceso al bus PCI para poder acceder a los recursos de los dispositivos PCI. El bus PCI es el más utilizado en las arquitecturas soportadas por MaRTE OS.

## 3.1. Librería de acceso al bus PCI

La arquitectura PCI [31] fue diseñada para sustituir al estándar ISA, y se desarrolló intentando conseguir:

- Un mejor rendimiento al transferir datos entre el computador y sus periféricos:
  - El bus PCI utiliza un reloj más rápido que el ISA.
  - Posee un bus de datos de 32 bits y la especificación incluye una extensión a 64 bits (aunque solo lo implementan las plataformas de 64 bits).
- Ser tan independiente de la plataforma como sea posible:
  - El bus PCI se utiliza intensamente en IA-32 (*Intel Architecture*), Alpha, PowerPC, SPARC64 e IA-64 además de alguna otra plataforma
- Facilitar la adición o sustracción de periféricos en el sistema:
  - Los dispositivos PCI no tienen *jumpers* (mecanismos estáticos de configuración *hardware*), a diferencia de la mayoría de periféricos, y son configurados automáticamente en el arranque del sistema.
  - El driver de dispositivo debe ser capaz de acceder a la información de configuración del dispositivo para completar la inicialización. Esto se realiza sin requerir ninguna prueba sobre el dispositivo. No necesitamos interrogar directamente al dispositivo intentando averiguar su configuración ya que podemos acceder a ésta directamente. Este comportamiento, además de ser más práctico, evita posibles grados de inestabilidad que pueden aparecer cuando se interroga al dispositivo.

Cada dispositivo PCI se localiza por:

- Un número de bus.
- Un número de dispositivo.
- Un número de función.

La especificación PCI permite al sistema alojar un máximo de 256 buses. Cada bus puede alojar un máximo de 32 dispositivos y cada dispositivo puede ser una tarjeta

multifunción, como una tarjeta de sonido con controladora ATAPI para CD-ROM, con un máximo de 8 funciones. Cada función por lo tanto puede ser identificada a nivel *hardware* por una dirección de 16 bits.

La circuiteria *hardware* de cada dispositivo contesta a peticiones pertenecientes a tres espacios de direcciones distintas:

- Localizaciones en memoria.
- Puertos de E/S.
- Registros de configuración.

Los dos primeros espacios de direcciones están compartidos por todos los dispositivos en el bus PCI. El espacio de configuración, por el contrario, se aprovecha del direccionamiento geográfico. Las transacciones de configuración direccionan un solo *slot* a la vez. Por lo tanto, no hay colisiones en el acceso a la configuración.

El espacio de configuración consiste en 256 bytes para cada función de dispositivo. La distribución de los registros en el espacio de configuración está estandarizada. Cuatro bytes del espacio de configuración almacenan un identificador de función único, y de esta manera el driver puede identificar a un dispositivo buscando el identificador específico para éste.

Cada dispositivo físico está direccionado geográficamente para recuperar la información de sus registros de configuración. La información en esos registros se puede utilizar para realizar un acceso I/O habitual sin la necesidad de más direccionamiento geográfico. Es decir, el *firmware* inicializa el *hardware* PCI en el arranque del sistema, mapeando cada región a una dirección distinta para evitar colisiones. Las direcciones a las que estas regiones están mapeadas se pueden leer en el espacio de configuración. Después de leer los registros de configuración, el *driver* puede acceder de manera segura al *hardware* sin necesidad de probarlo.

La librería se implementó en lenguaje C aunque se proporcionan los correspondientes *bindings* Ada para proporcionar una API Ada y poder realizar *drivers* que acceden a dispositivos PCI en Ada.

Para poder acceder al espacio de configuración y almacenar la configuración actual se define una estructura de datos de configuración PCI, llamada *pci_device*, que almacenará la información relevante de cada dispositivo.

La estructura *pci_device* es de la siguiente forma:

```
struct pci_device {
     uint32_t class;
     uint16_t vendor;
     uint16_t dev_id;
     struct region_map pci_region[6];
     unsigned char devfn;
     unsigned char bus;
     unsigned char irq;
     unsigned char pin;
};
```

Explicamos el significado de cada uno de los campos:

- *class*: Cada dispositivo pertenece a una clase. El registro de clase es un registro de 16 bits cuyos primeros 8 bits identifican la clase base o grupo. El resto

identifica al tipo de dispositivo. Por ejemplo *ethernet* y *token-ring* son dos clases que pertenecen al grupo *redes*, mientras las clases *serie* y *paralelo* pertenecen al grupo de *comunicaciones*. Algunos drivers pueden manejar distintos dispositivos similares, cada uno de ellos con un identificador distinto pero pertenecientes a la misma clase.

- *vendor*: Es un registro de 16 bits que identifica al fabricante del dispositivo. Existe un registro global de esos identificadores, mantenidos por el grupo *PCI Special Interest*, y los fabricantes deben solicitar un único número para asignar a sus dispositivos. En el archivo *pci_ids.h* se definen bastantes identificadores de fabricante al igual que de dispositivos.
- *dev_id*: Es otro registro de 16 bits que indica el identificador de dispositivo. Éste es asignado por el fabricante y no requiere de un registro oficial. Normalmente está emparejado con el identificador de fabricante formando un identificador de dispositivo único de 32 bits que se suele llamar *firma*. Un *driver* de dispositivo normalmente utiliza la *firma* para identificar el dispositivo. Los identificadores de fabricante y de dispositivo se pueden consultar en el manual del dispositivo.
- *pci_region[6]*: Un dispositivo PCI puede implementar hasta seis regiones de direcciones de I/O. Cada región puede estar ubicada en memoria o en el bus de I/O. La mayoría de los dispositivos implementan sus registros de I/O en regiones de memoria ya que generalmente es un aproximación más eficiente. Cada componente *pci_region* almacena en una estructura *region_map* la información referente a una región de memoria de un dispositivo. La estructura *region_map* tiene la siguiente estructura:

```
struct region_map {
  unsigned int   base_addr;
  unsigned char  space;
  uint32_t       size;
};
```

  - *base_addr*: Almacena la dirección del espacio genérico de direcciones de I/O. Puede ser mapeo de memoria o de puerto.
  - *space*: Indicará si el espacio genérico indicado por *base_addr* corresponde a un mapeo en memoria o en puerto. Los posibles valores devueltos son:
    - 'M': La región indica espacio de direcciones en memoria.
    - 'I': La región mapea espacio de direcciones de I/O.
    - 'N': Si la región no está mapeada.
  - *size*: Informa sobre el tamaño del área mapeada.
- *bus*: Corresponde al número de bus en el que se encuentra el dispositivo.
- *devfn*: Indica el número de dispositivo y número de función que identifica físicamente a un periférico PCI:
  - Los primeros 5 bits identifican el número de dispositivo.
  - Los últimos 3 bits corresponden al número de función.
- *irq*: Almacenará la línea de interrupción utilizada por el dispositivo. Un procesador con un número limitado de líneas de IRQ, como el x86, puede manejar muchas tarjetas de interfaz PCI, cada una de ellas con cuatro pines de

interrupción, debido a que la especificación de PCI requiere que las líneas de interrupción se puedan compartir.

- *pin*: Cada slot PCI tiene cuatro pines de interrupción y cada dispositivo puede utilizar una de ellas sin preocuparse de cómo esos pines son enrutados a la CPU. Ese enrutamiento es responsabilidad de la plataforma y se implementa fuera del bus PCI.

En la implementación real de la estructura *pci_device* existen otros campos. Éstos no serán comentados ya que sólo se han proporcionado para asegurar compatibilidad con las implementación de los drivers de red y no deben ser utilizados.

La librería proporciona la función *pci_find_device* que se encarga de buscar un dispositivo concreto y almacenar su configuración en una estructura *pci_device*:

```
int pci_find_device(unsigned short vendor,
     unsigned short device, struct pci_device *from,
     struct pci_device *found);
```

Proporcionando un fabricante (*vendor*) y un dispositivo (device) esta función buscará el dispositivo a lo largo de la jerarquía PCI. Si encuentra un dispositivo que reuna esa *firma,* guardará su configuración en la estructura *pci_device found*.

El argumento *from* se utiliza para poder localizar múltiples tarjetas con la misma *firma*, es decir, múltiples tarjetas idénticas. Este argumento deberá apuntar, a través de los campos *bus* y *devfn*, al último dispositivo encontrado. De ser así la búsqueda de dispositivos proseguirá a partir del último encontrado. Con este proceder podemos detectar, por ejemplo, varias tarjetas de red idénticas instaladas en el sistema. Para encontrar el primer dispositivo con una determinada *firma* el argumento *from* se debe especificar como NULL. Si no se encuentra ningún dispositivo con la *firma* requerida se retorna -1. Si se encuentra el dispositivo la función devuelve 0.

También existe un valor especial que identificará a cualquier dispositivo conectado al bus PCI. Ese valor es asignar 0 al argumento *vendor* y *device*. Este comportamiento es útil si se quiere escanear todos los dispositivos PCI conectados en el sistema.

De manera similar podemos encontrar dispositivos a partir de su clase con la función:

```
int pci_find_class(unsigned int class,
       struct pci_device *from,
       struct pci_device *found);
```

La utilización de esta función es similar a *pci_find_device*. Se le proporciona la clase en *class* y devuelve el primer dispositivo de la clase encontrado en *found*. Si queremos continuar con la búsqueda lo realizaremos utilizando el argumento *from* de manera análoga a *pci_find_device*.

Además de las funciones expuestas para localizar los recursos asociados a un dispositivo PCI. También proveemos unas funciones básicas para acceder al área de configuración de un dispositivo concreto para poder acceder directamente a sus registros. Para su correcta utilización se precisa un conocimiento del área de configuración del PCI.

Son funciones preparadas para leer y escribir los distintos registros de distinto tamaño del área de configuración:

```
int pci_read_config_byte(struct pci_device *dev,
       unsigned int where, uint8_t *value);

int pci_write_config_byte(struct pci_device *dev,
       unsigned int where, uint8_t value);

int pci_read_config_word(struct pci_device *dev,
       unsigned int where, uint16_t *value);

int pci_write_config_word(struct pci_device *dev,
       unsigned int where, uint16_t value);

int pci_read_config_dword(struct pci_device *dev,
       unsigned int where, uint32_t *value);

int pci_write_config_dword(struct pci_device *dev,
       unsigned int where, uint32_t value);
```

Todas las funciones reciben como argumento:

- *dev*: Una estructura *pci_device* con unos valores de *bus* y *devfn* identificando a un dispositivo PCI.
- *where*: La ubicación del registro de configuración sobre el que vamos a realizar la operación. Estos registros están definidos en el fichero de cabecera pci.h.
- *value*: El valor escrito o leido dependiendo de si la función es *pci_write_** o *pci_read_**.

Y devuelven 0 si no hay error.

También hemos provisto un *binding* Ada (*pci.ads*) para que pueda ser utilizado desde programas Ada. Las estructuras, funciones y comportamientos son equivalentes a las proporcionadas en C:

La estructura *pci_device* queda:

```
type Pci_Device is record
     Class       : Unsigned_32;
     Vendor      : Unsigned_16;
     Device_ID   : Unsigned_16;
     Pci_Region  : PCI_Region_Map;
     Device_Fn   : Unsigned_8;
     Pci_Bus     : Unsigned_8;
     Pci_Irq     : Unsigned_8;
     Pci_Pin     : Unsigned_8;
end record;
```

Siendo *PCI_Region_Map*:

```
type PCI_Region_Map is array (1 .. 6) of Region_Map;
```

Y la estructura correspondiente a cada región mapeada (*region_map*):

```
type Region_Map is record
     Base_Address : Unsigned_32;
     Space_Mapped : Character;
     Region_Size  : Unsigned_32;
end record;
```

Mientras que las funciones que proporcionan los recursos asociados a un dispositivo:

```
procedure Pci_Find_Device( Result : out Integer;
     Vendor : in Unsigned_16;
     Device : in Unsigned_16;
     From   : in out Pci_Device;
     Found  : in out Pci_Device);
```

En el *binding* Ada la función C *pci_find_device* pasa a ser un procedimiento Ada y el valor que retorna se almacena en el argumento *Result*. Este comportamiento es debido a que una función en Ada no admite parámetros de entrada / salida sino solamente de entrada.

De manera similar tenemos las funciones que nos permiten acceder a los registros del área de configuración de un dispositivo:

```
procedure Pci_Read_Config_Byte( Result : out Integer;
     Dev    : in out Pci_Device;
     Where  : in Unsigned_32;
     Value  : out Unsigned_8);

procedure Pci_Read_Config_Word( Result : out Integer;
     Dev    : in out Pci_Device;
     Where  : in Unsigned_32;
     Value  : out Unsigned_16);

procedure Pci_Read_Config_DWord( Result : out Integer;
     Dev    : in out Pci_Device;
     Where  : in Unsigned_32;
     Value  : out Unsigned_32);

procedure Pci_Write_Config_Byte( Result : out Integer;
     Dev    : in out Pci_Device;
     Where  : in Unsigned_32;
     Value  : out Unsigned_8);

procedure Pci_Write_Config_Word( Result : out Integer;
     Dev    : in out Pci_Device;
     Where  : in Unsigned_32;
     Value  : out Unsigned_16);

procedure Pci_Write_Config_DWord( Result : out Integer;
     Dev    : in out Pci_Device;
     Where  : in Unsigned_32;
     Value  : out Unsigned_32);
```

# 3.2. Drivers de red para MaRTE OS

Los drivers son módulos especiales de *software* que utiliza el sistema operativo para comunicarse con los dispositivos *hardware* conectados al sistema. La identificación de los drivers se realiza a través de dos números especiales:

- **Número Mayor**: Asociado a unidades funcionales separadas, la mayoría de las veces corresponde, aunque no es estrictamente necesario, a dispositivos físicos independientes. Esta unidad funcional suele tener acceso a sus propios registros, memoria de E/S y manejador de interrupciones.

- **Número Menor**: Utilizado para distinguir diferentes subunidades asociadas a un dispositivo mayor. Normalmente todos los dispositivos menores asociados al mismo dispositivo mayor comparten recursos, como puede ser registros de control o el manejador de interrupciones. También se puede utilizar el número menor para identificar diferentes funciones de un dispositivo mayor. En MaRTE OS utilizaremos el número menor para distinguir entre distintos dispositivos físicos del mismo tipo.

Para poder comunicarse con los drivers se definen una serie de funciones generales comunes a todos los dispositivos. Estas funciones poseen una especificación común pero una funcionalidad diferente. Las funciones presentes en MaRTE OS son:

- **Create**: Se invoca cuando se inicia el sistema, su finalidad es la de inicializar el dispositivo y todas las estructuras del driver.
- **Open**: Esta función devolverá un descriptor de fichero a un dispositivo. Además puede realizar alguna inicialización adicional.
- **Close**: Recibe un descriptor de fichero de un dispositivo abierto y libera todos los recursos sin utilizar asociados al dispositivo.
- **Read**: Lee datos del dispositivo.
- **Write**: Escribe datos al dispositivo.
- **Ioctl**: Con esta función podemos ajustar la configuración y parámetros de control del dispositivo además de chequear el estado del mismo.
- **Remove**: Esta función separa completamente el dispositivo del sistema.

Se configuran los dispositivos a través de un archivo especial (*k-devices_table.ads*) contenido en el directorio *kernel* de MaRTE. En ese archivo (ver la documentación de MaRTE OS), podemos especificar el número mayor, menor y los punteros asociados a las funciones genéricas sobre el dispositivo.

En la implementación actual se realizaron drivers para dos tipos de tarjetas de red:

- **SiS 900 y SiS 7016**: Es una solución Fast Ethernet 10/100 en un único chip que integra el control de acceso al medio (MAC) con la interfaz del bus PCI y es conforme con 802.3u. Los chips de SiS implementan por completo la interfaz del bus PCI version 2.1. Los descriptores de los paquetes y los datos se transfieren a través de canales DMA reduciendo la carga en la CPU y optimizando el uso del espacio de memoria.
- **Intel EtherExpressPro100 y derivadas con el chipset 82559 y 82562EM**: Un único chip que implementa el control de acceso al medio y la interfaz PCI. Del mismo modo que el chip SiS 900 es capaz de hacer uso del DMA para aumentar la eficiencia.

Para la realización de estos drivers se tomó como base de desarrollo los drivers de red generados en el proyecto Etherboot [12] que a su vez heredan los drivers de GNU/ Linux. La decisión de adaptar unos drivers ya existentes frente a la creación de unos propios nos proporciona el acceso, con menos esfuerzo, a un juego de drivers de tarjetas de red bastante amplio así como, al estar basado en un proyecto “vivo”, tendremos acceso a drivers de futuras tarjetas.

Debido a las particularidades de Etherboot así como a las exigencias de MaRTE OS, no se pueden utilizar los drivers en MaRTE “tal cual” sino que es preciso una adaptación

de los mismos. Con objeto de facilitar la futura integración de más drivers de Etherboot se decidió construir una capa de adaptación de los drivers en MaRTE OS. Además es preciso modificar los drivers de Etherboot para que soporten la recepción de paquetes por interrupción (nativamente lo hacen por *polling*) dotándoles de un manejador de interrupciones. También se debe proporcionar al driver la funcionalidad necesaria para poder operar en modo promiscuo.

El *framework* creado sobre los drivers de red nos proporciona una forma uniforme de acceso a la red desde MaRTE OS así como una serie de funciones homogéneas de configuración e información de la tarjeta de red.

La descripción básica de la arquitectura de red creada en MaRTE OS consiste en un *buffer* circular en memoria que es rellenado vía DMA por la tarjeta de red con las tramas válidas (aquellas que pasan el filtro de protocolo) recibidas. Cada trama recibida hace que se eleve un manejador de interrupción que chequea el protocolo de la trama y la copia al *buffer* circular si el protocolo pasa el filtro de protocolo previamente establecido (un protocolo 0 hace que sea recibida cualquier trama). La lectura del buffer por parte de la aplicación puede ser no bloqueante si así se configura. La operación write transmite directamente en la red.

Con este *framework* conseguimos una forma uniforme de acceso a la red al tener las llamadas *create, open, read, write, close, ioctl* y *remove*. Además poseemos una forma homogénea de configuración e información de la red al normalizar la llamada *ioctl* independientemente del driver que utilicemos.

El driver de red provee las siguientes funciones (especificadas en *ethernet.h*):

- **Create** (*eth_create*): Esta función escanea y configura las tarjetas de red PCI en el sistema e instala el manejador de interrupciones de la tarjeta. Devuelve 0 si no hay error.

```
ssize_t eth_create(int create_arg);
```

- **Open** (*eth_open*): Esta función "abre" el dispositivo y habilita el manejador de interrupciones. También realiza cierta inicialización adicional dentro del driver como habilitar los mecanismos de transmisión y recepción de la tarjeta de red.

```
ssize_t  eth_open(int file_descriptor,
       int file_access_mode);
```

- **Read** (*eth_read*): Recibe una trama Ethernet de la red. La trama es leída de un *buffer* circular que previamente ha sido rellenado, con una trama válida, por el manejador de interrupciones de la tarjeta de red.

```
ssize_t eth_read(int file_descriptor, void *msg,
       size_t len);
```

  La estructura de los datos recibidos corresponde con una trama Ethernet con la siguiente estructura:

```
{
     struct ethhdr eth_header;
     char *info;
}
```

  Donde la estructura *ethhdr* es la cabecera Ethernet definida en el fichero *if_ether.h* (48 bits correspondientes a la dirección MAC destino, 48 bits para la

estación origen y 16 bits para el protocolo). *info* es un puntero a un área de memoria reservada para almacenar la información a utilizar por la red (como máximo 1500 bytes). Es importante destacar que el protocolo de la trama Ethernet está expresado en el orden de bits de red (Big Enddian). Devuelve el número de bytes leidos. En caso de error devuelve -1.

- **Write** (*eth_write*): Manda una trama Ethernet a la red:

```
ssize_t eth_write(int file_descriptor, void *msg,
       size_t len);
```

  La estructura de *msg* es similar a la expuesta en la operación *read*. El argumento len debe contener el tamaño de la trama ethernet (cabecera + información). La cabecera Ethernet ocupa 14 bytes. El protocolo en la estructura ethhdr debe ser expresado en la ordenación de bits de red (Big Enddian).

- **Close** (*eth_close*): Esta función deshabilita las interrupciones y para los mecanismos de transmisión y recepción de paquetes.

```
ssize_t eth_close(int file_descriptor);
```

- **Ioctl** (*eth_ioctl*): Con los comandos adecuados se puede ajustar la configuración y parámetros de control del dispositivo:

```
ssize_t eth_ioctl(int file_descriptor, ssize_t request,
       void *priv);
```

  Los comandos que soporta este driver están especificados en *eth_ioctl.h* o en *ethernet_ioctl.ads* si se está utilizando el driver desde Ada:

  - **ETH_HWADDR**: Devolverá en priv la dirección MAC de la tarjeta Ethernet.
  - **ETH_BLOCKING_READ**: Cambiará la operación de lectura a bloqueante, es decir, que no retornará la llamada hasta que se reciba un dato. Este es el valor por defecto.
  - **ETH_NON_BLOCKIN_READ**: Hará que la operación de lectura sea no bloqueante, es decir, la operación lectura retornará después de ser invocada con o sin datos dependiendo de que estos estén disponibles.
  - **SET_PROTOCOL_FILTER**: Asignará el protocolo utilizado en el filtro de recepción. Por defecto, el driver sólo recibe paquetes RT-EP (protocolo 0x1000). Se puede configurar el driver para sólo recibir los paquetes del protocolo asignado en *priv* (*unsigned short* *) (IP, NETBIOS, etc.. para una ayuda con los números válidos de protocolo ver <*include/drivers/if_ether.h*>). Si se asigna 0 en *priv*, el driver recibirá todos los paquetes arrojados en la red.
  - **GET_PROTOCOL_FILTER**: Devuelve en *priv* (*unsigned short* *) el filtro de protocolo actual. Si el valor devuelto es 0 indica que el filtro de protocolo está deshabilitado.

Con el *framework* creado es posible acceder de manera uniforme a los drivers modificados proporcionados por Etherboot. Además de hacer más cómoda la incorporación de nuevos drivers de red en MaRTE OS.

# 4. Implementación de RT-EP

## 4.1. 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 [38]. 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.

La implementación sobre MaRTE OS ha sufrido una serie de cambios profundos y adaptaciones frente a la realizada sobre GNU/Linux presentada en [26]. Estas modificaciones obedecen a distintas adaptaciones específicas necesarias en MaRTE OS además de distintas mejoras y ampliaciones en las prestaciones de la implementación.

## 4.2. Arquitectura del software del protocolo

La funcionalidad y arquitectura del protocolo se muestran en la figura 4.1. 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 con una política

FIFO para los mensajes de la misma 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 la configuración extraída de un archivo de cabecera. 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 una serie de primitivas a la aplicación para realizar la comunicación:

- **Inicializar_Comunicación:** Esta llamada inicializa y configura la prioridad del *thread* de comunicaciones. 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.
- **Enviar_Información:** Se utiliza para poder mandar un mensaje de una determinada prioridad a un determinado canal de comunicaciones.
- **Recibir_Información:** Esta llamada se encarga de recibir un mensaje con su prioridad de un determinado canal de comunicaciones. Esta llamada es bloqueante.
- **Alguna_Información:** Nos indica si hay algún elemento en una determinada cola de recepción.

Debido a que la implementación del protocolo puede ser utilizada desde C o ADA, estas primitivas son ejecutadas gracias a una API para lenguajes C o ADA.

La comunicación con la red se hace a través del driver de red utilizando llamadas POSIX (*open, close, read, write*) como está detallado en 4.3, “Detalles de la implementación”.

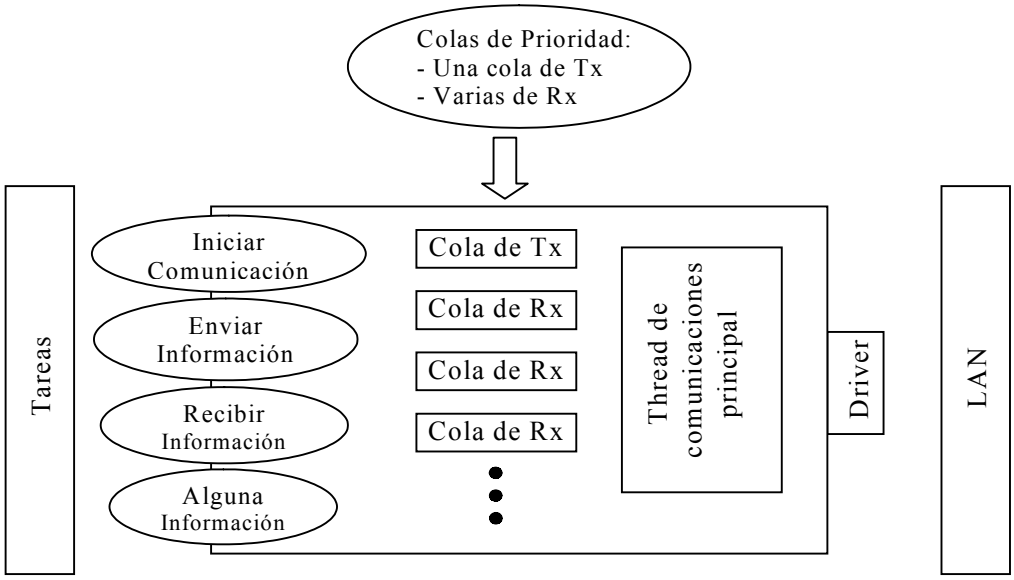



**Figura 4.1: Funcionalidad y arquitectura de RT-EP**

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 4.1, 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.

## 4.3. Detalles de la implementación

A continuación mostraremos una sucesión de apartados que describirán uno tras otro las distintas facetas de la implementación del protocolo en el sistema operativo de Tiempo Real MaRTE OS. Esta implementación no sólo es exclusiva de MaRTE OS sino que es extensible, con unos pequeños ajustes, a cualquier otro sistema operativo POSIX.

### 4.3.1. Comunicación con la red de 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.

En RT-EP sobre MaRTE OS no precisamos de ninguna abstracción externa que proporcione el punto final para la comunicación ya que es el propio protocolo el que se encarga de proporcionar el enlace con la aplicación. Además si se requiere una comunicación en tiempo real estricto es preciso que todas las aplicaciones que requieran una comunicación a través de la red lo hagan con RT-EP.

Para comunicarse con la red, RT-EP utiliza directamente el driver de red previamente instalado en MaRTE OS. El acceso al driver se realiza con llamadas *open, read, write, ioctl, close*. Con RT-EP 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).

Dada la naturaleza de MaRTE OS y la de las aplicaciones de Tiempo Real en la construcción de los drivers de red se ha establecido un filtro en la recepción de tramas. De manera que, estableciendo el filtro en el driver (mediante una llamada *ioctl*), únicamente recibiremos tramas Ethernet que pertenezcan a un protocolo determinado. Para poder recibir correctamente tramas de RT-EP en la red debemos establecer el filtro de protocolo en la red al valor de éste protocolo (actualmente 0x1000). Con este comportamiento se consigue reducir el overhead en la estación en el caso de recibir tramas que no son suyas. Conseguimos que las tramas sean descartadas a nivel de driver en vez de a nivel del protocolo aunque de llegar a la capa de RT-EP seria automáticamente descartado.

El protocolo también utiliza la llamada *ioctl* con el comando ETH_HWADDR 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 de la configuración como se expondrá más adelante.

Por defecto los drivers de red utilizan el modo promiscuo de la capa de enlace de manera que todas las estaciones reciben todas las tramas que circulan por la red y no sólo las que estén dirigidas a ellas. Se ha habilitado el modo promiscuo en el driver 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.

Antes de poder utilizar el dispositivo es necesario establecer una conexión entre el dispositivo y un descriptor que será utilizado por otras funciones de I/O para referirse al dispositivo. Esta conexión la realizamos con la función POSIX *open*:

```
int open(const char *path, int oflag, ... );
```

Si emplazamos en *path* la identificación del dispositivo de red en MaRTE (/dev/eth0 en esta implementación) y en las opciones de acceso a fichero (*oflags*) *O_RDWR* obtendremos un descriptor para poder utilizar el dispositivo de red en modo de lectura / escritura.

Una vez que se ha abierto el dispositivo con la función POSIX *open*, es posible recibir y enviar tramas mediante las funciones POSIX *read* y *write* respectivamente:

```
ssize_t read(int fildes, void *buf, size_t nbyte)
```

La función *read* intentará leer *nbyte* bytes del dispositivo apuntado por el descriptor *fildes* y coloca los datos leídos en el *buffer* apuntado por *buf*. Esta función devuelve el numero de bytes leídos. Utilizaremos esta función para leer las tramas de nivel de enlace arrojadas al medio.

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
```

La función *write* intentará escribir *nbyte* bytes del *buffer* apuntado por *buf* al dispositivo asociado por el descriptor *fildes*. Esta función devuelve el número de bytes escritos.

Corresponde al protocolo la construcción y manejo de tramas válidas de nivel de enlace y realizar las correspondientes operaciones para asegurar la comunicación entre estaciones.

### 4.3.2. 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 [13]. 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

En MaRTE OS la fuente de interrupciones para los temporizadores se encuentra implementado a través del contador 0 del PIT. Por lo tanto la resolución será la misma que la de los contadores del PIT, esto es 838.1 ns.

Para crear un temporizador usamos la siguiente operación:

```
int 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 envío de una señal.
- Crear y ejecutar un thread.
- Ninguna.

En RT-EP utilizaremos el envío de una señal para "despertar" a un thread que se encargará de 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.

### 4.3.3. 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 [37]. 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 *obra*. Podemos decir que un proceso es como una obra 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. En MaRTE OS no existen procesos ya que es un sistema operativo monoproceso. La programación de aplicaciones se debe realizar en base a *threads*.

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].

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

### 4.3.4. Sincronización de threads

Una tarea *cooperativa* [37] 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* [37].

Las alternativas que existen para la sincronización, en sistemas operativos POSIX, son [13]:

- **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. Éstos han sido la opción elegida para acceder a los recursos compartidos (las colas) en el protocolo al permitirnos implementar, de una manera muy cómoda, los mecanismos de sincronización de espera que necesitamos en las colas de prioridad.

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 de sincronización 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 [13]:

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

Pseudocódigo del *thread* que espera [13]:

```
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 [13]. 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 4.2. 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.

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

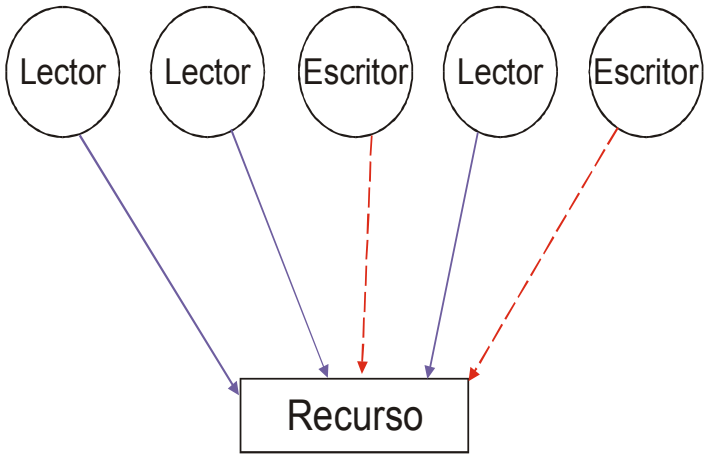



**Figura 4.2: Problema clásico lectores-escritores**

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 4.3.7.6, "prio_queue_monitor"

## 4.3.5. Estructura de datos: Colas de prioridad (Heaps) y FIFOs

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 4.3. 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.

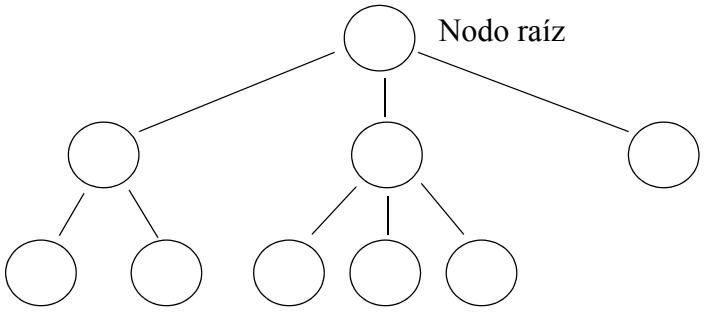



**Figura 4.3: Estructura de datos: Arbol**

Un tipo de árbol distinto al de la figura 4.3 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 *montículo* (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 *montículo binario*, figura 4.4, 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 montículo tiene una ordenación diferente a la del árbol binario tradicional. Sin embargo lo que sí 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 montículo siempre tiene la mayor prioridad de todo el árbol. Será ese nodo el primero que se extraiga del montículo.

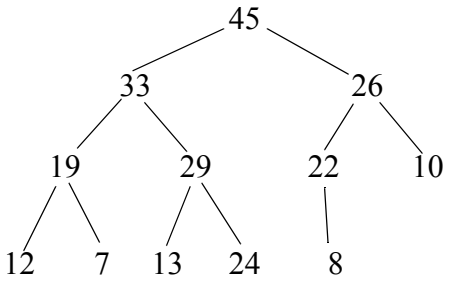



**Figura 4.4: Montículo binario**

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 montículo. 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 montículo. Este mecanismo de inserción queda representado en la figura 4.5 donde se inserta un nuevo elemento con prioridad 30.

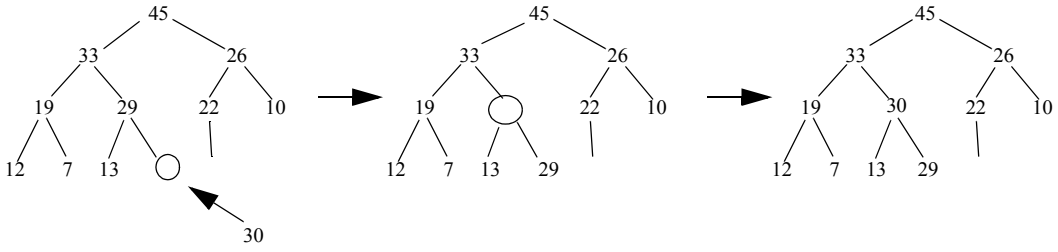



**Figura 4.5: Proceso a seguir para insertar un elemento**

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 montículo como se muestra en la figura 4.6.

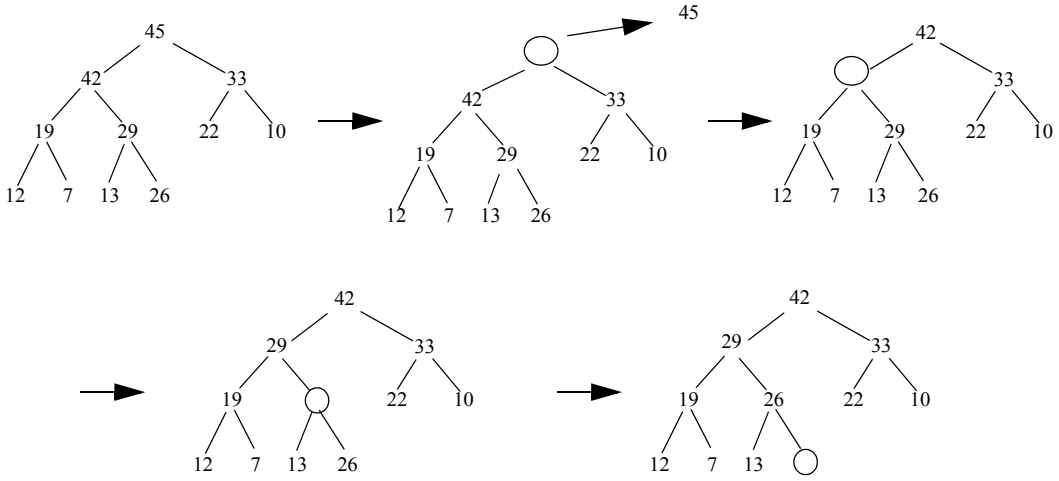



**Figura 4.6: Proceso a seguir para extraer un elemento**

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 montículo *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 4.7 mostramos cómo quedaría implementado en un array el montículo de la figura 4.4.

| **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 4.7: Representación del montículo 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)

Con la implementación de colas de prioridad obtenemos una ordenación de elementos por prioridad pero esto no es suficiente ya que las colas de prioridad no garantizan un comportamiento FIFO (First In First Out) para los elementos de la misma prioridad, es decir, queremos que el primer elemento de una prioridad determinadada introducido en la estructura de dados sea el primero en ser extraído de ésta. Para conseguir un comportamiento FIFO en los elementos de la misma prioridad se ha optado por añadir a la implementación de colas de prioridad una cola FIFO por cada prioridad del montículo.

La idea es que sólo se encolen como máximo un elemento por prioridad en la cola de prioridad y que el resto de los elementos de la misma prioridad, de existir, se añadan a la cola FIFO correspondiente. De esta manera conseguimos una estructura de datos en la que los elementos se almacenan por prioridad y además tenemos un comportamiento FIFO para los elementos de la misma prioridad.

La implementación de las colas FIFOs se han realizado mediante una lista enlazada. Tendremos un puntero al comienzo del la cola que determinara el elemento a extraer y otro puntero al final de la cola que determinará el lugar donde insertar el nuevo elemento.

Cuando intentemos insertar un elemento en la cola FIFO se comprobará si hay algún elemento ya almacenado en la cola, de existir, se añadirá al puntero que indica el final de la cola y de no existir haríamos que el puntero de la cabeza de la cola apunte al nuevo elemento. A la hora de extraer, comprobamos si existe algún elemento en la cola y, de existir, devolvemos el elemento referenciado por el puntero de la cabeza de la FIFO y actualizamos el puntero para que apunte al siguiente elemento de la cola.

En un análisis de costes se puede ver que el tiempo requerido para añadir o extraer un elemento de la cola es constante e independiente del número de elementos en la cola. El costo de esas operaciones es O(1).

En RT-EP se realizan operaciones de extracción, inserción y lectura del primer elemento de los datos almacenados, por lo tanto, tendremos un coste medio de O(log n) en el peor caso (extracción e inserción en la cola de prioridad) 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.

Como puede verse la inserción de la cola FIFO en la estructura de almacenamiento no ha repercutido, salvo constante, en el coste de las operaciones sobre los datos almacenados.

La implementación de este tipo de colas de prioridad y FIFO para la transmisión y recepción, se ha realizado en el módulo *prio_queue* que está descrito en el apartado 4.3.7.5.

### 4.3.6. Interfaces con el lenguaje Ada.

Ada proporciona una serie de facilidades que nos permiten utilizar desde Ada código realizado en otros lenguajes. Del mismo modo también nos proporciona las herramientas para utilizar recursos Ada desde otros lenguajes. Estas herramientas nos

permiten realizar una integración entre componentes escritos en Ada y otro lenguaje. Esta integración se realizará por medio de coberturas (*bindings*).

Existen dos tipos de coberturas (*bindings*):

- **Directo** (también llamado *fino*): Proporciona un mapeo uno-a-uno en Ada de la interfaz que proporciona el programa no-Ada. Son fáciles de crear y rápidas de entender si se entiende la interfaz del programa no-Ada. El inconveniente principal es que a menudo puede ser un poco engorroso trabajar con esta interfaz y normalmente no proveen la protección existente en las interfaces Ada. Por lo tanto es preferible siempre utilizar coberturas abstractas.
- **Abstracto** (también llamado *grueso*): Una cobertura abstracta proporciona una visión más Ada de la interfaz externa. Aunque este tipo de coberturas son mucho más cómodas para trabajar con ellas conlleva más tiempo y trabajo crearlas.

Ada95 proporciona un conjunto de paquetes y algunos *pragmas* especiales para proporcionar una interfaz con otros lenguajes de programación. Los pragmas mas útiles son:

- **pragma Import**: Importa un subprograma de otro lenguaje de programación a Ada. Es necesario utilizarlo si queremos invocar, por ejemplo, una función C.
- **pragma Import_Function**: Este pragma se utiliza junto con *pragma Import* para especificar información adicional para la función importada.
- **pragma Import_Procedure**: Como el *pragma Import_Function* pero proporciona opciones extras para procedimientos.
- **pragma Import_Valued_Procedure**: Este pragma es idéntico a *pragma Import_Procedure* excepto que proporciona un método para recuperar el valor retornado por una función. El propósito de este pragma es posibilitar la interfaz con funciones que devuelven parámetros en los argumentos (lo cual no está permitido en funciones Ada).
- **pragma Export**: Exporta un subprograma Ada a un lenguaje no-Ada. Por ejemplo, si queremos llamar desde C una función Ada tenemos que utilizar este pragma.
- **pragma Export_Function**: Este pragma se utiliza junto con *pragma Export* para especificar información adicional para la función exportada.
- **pragma Export_Procedure**: Como *pragma Export_Function* pero proporciona opciones extras para procedimientos.
- **pragma Convention**: Especifica que un tipo determinado debe utilizar los convenios de almacenamiento de un determinado lenguaje.
- **pragma Linker_Options**: Utilizado para especificar los parámetros de enlazado con el linker necesarios cuando se incluye una unidad de compilación.

- **Ejemplos**:
  - Importar una función C llamada *gettime* a una *Get_Time* en Ada:

```
pragma Import(C, gettime, "Get_Time");
```

  - Exportar una función Ada llamada Read_Sensor al compilador de COBOL:

```
pragma Export(COBOL, Read_Sensor);
```

- Leer y escribir la estructura State_Vector utilizando las convenciones de almacenamiento de Fortran:

```
pragma Convention(Fortran, State_Vector);
```

Existe un factor a tener en cuenta si el programa principal no es Ada, que es la elaboración. El subprograma principal debe asegurarse de que el entorno para la parte Ada está correctamente establecido. Esto se realiza automáticamente si el subprograma principal el Ada pero en el caso de otro lenguaje de programación es necesario asegurarse de que la parte Ada se inicializa antes de que sea invocada.

Normalmente los compiladores (GNAT en nuestro caso) proveen dos funciones Ada para manejar la parte Ada de un programa:

- **adainit**: Inicializará el entorno Ada.
- **adafinal**: Recupera los recursos capturados por la inicialización del entorno Ada una vez que se haya terminado su invocación.

Para poder utilizar correctamente desde un programa principal no-Ada funciones y procedimientos Ada debemos invocar la función *adainit* antes de cualquier invocación Ada y se llamará a *adafinal* cuando se finalice la utilización de la parte Ada.

Existe una serie de reglas generales de correspondencia Ada - C. Están basadas en el anexo B.3 del Manual de Referencia Ada [45]:

- Un procedimiento Ada corresponde a una función void de C.
- Una función Ada corresponde con una función C.
- Un array Ada corresponde a un puntero C al primer elemento.
- Los tipos escalares simples (*integer*, *float*, *access*, etc...) corresponden al tipo equivalente en C para representar a esos escalares.

Ada 95 proporciona un juego de paquetes predefinidos que facilitan la tarea de conectar con C. El paquete principal es “Interfaces.C”, que contiene las definiciones para los tipos C en Ada. Estos incluyen los tipos C *int*, *long*, *unsigned* y *double*. El tipo C *float* se llama “C_Float” en Ada por lo tanto no se confunde con el tipo *Float* en Ada. Los tipos *float* para C y Ada son probablemente idénticos pero esta afirmación puede no ser necesariamente cierta. El tipo “char_array” simboliza arrays de caracteres en C. Muchas funciones C asumen que un array de caracteres termina con el carácter especial *null* (\0 en C). Como los *strings* en Ada normalmente no terminan en *nul*, existen una serie de funciones llamadas *To_C* y *To_Ada* empleadas para convertir entre los *strings* Ada y los arrays de caracteres C.

Existen otros paquetes adicionales llamados “Interfaces.C.Strings” e “Interfaces.C.Pointers” que proporcionan tipos adicionales y operaciones sobre punteros y *strings* C. Por ejemplo, el paquete “Interfaces.C.Strings” define el tipo *chars_ptr* que corresponde con el tipo C *char* * cuando se utiliza para referenciar un *string* en C. El paquete también define:

- La constante *Null_Ptr* que corresponde en C con *(char*)NULL*.
- El procedimiento *Free* que corresponde con la función C *free()*.
- La función *Value* que toma como argumento un *char_ptr* y devuelve un string Ada. Esta función eleva la excepción *Dereference_Error* si se le pasa un puntero nulo.

Las estructuras en C corresponden con los *record* en Ada, el Manual de Referencia Ada aconseja, *pero no requiere*, que los *record* sean pasados a C como punteros al comienzo de la estructura C. Para los casos en los que la función C espera que se le pase una estructura por valor, se debería crear una nueva función C que convierta un puntero a la estructura y entonces se realice la llamada desde Ada. Sin embargo este comportamiento sobre las estructuras es una recomendación, el compilador Ada GNAT no sigue este consejo y en cambio transmite los records Ada por valor, es decir, una copia. Ambas aproximaciones son razonables pero desafortunadamente distintas. La aproximación más segura de pasar un record Ada es utilizar siempre un puntero al record, ya que al tratarse de un escalar nos aseguramos que se transmiten correctamente en todos los compiladores Ada.

Gracias a la interfaz con C podemos definir una *cobertura abstracta* Ada de RT-EP, que está programado en C. De esta manera podemos no sólo importar en Ada las funciones C del protocolo sino que podemos realizar una interfaz más acorde al lenguaje sobre el que se va a utilizar, aprovechando así las facilidades ofrecidas por el lenguaje Ada. Por ejemplo, se traducen los valores de error retornado por las funciones C a excepciones Ada, se ofrecen distinto tipo de funciones genéricas para mandar o recibir información, podemos sobrecargar las funciones envío y recepción, etc..

### 4.3.7. 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 4.8 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. Un módulo adicional del protocolo corresponde a la cobertura Ada del protocolo implementado en el paquete Rt_Ep_Protocol. 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:

- **rt_ep_protocol**: Interfaz Ada del protocolo
- **rt_comm:** Interfaz con la aplicación C.
- **prio_element:** Tipo de datos a transmitir.
- **prio_priority:** Tipo de la prioridad de los datos.
- **prio_queue:** Implementación de la cola de prioridades.
- **prio_queue_monitor:** Monitor de acceso a la cola de prioridades.
- **lnk_config:** Configuración del anillo.
- **ring_config**: Especificación del anillo lógico.
- **martelnk:** *Thread* de comunicaciones y tratamiento de fallos.
- **prot_types:** Configuración de parámetros del protocolo
- **eth_tools**: Define unas funciones de manejo de direcciones MAC.
- **rt_ep_errorn**: Definición de códigos de error del protocolo.

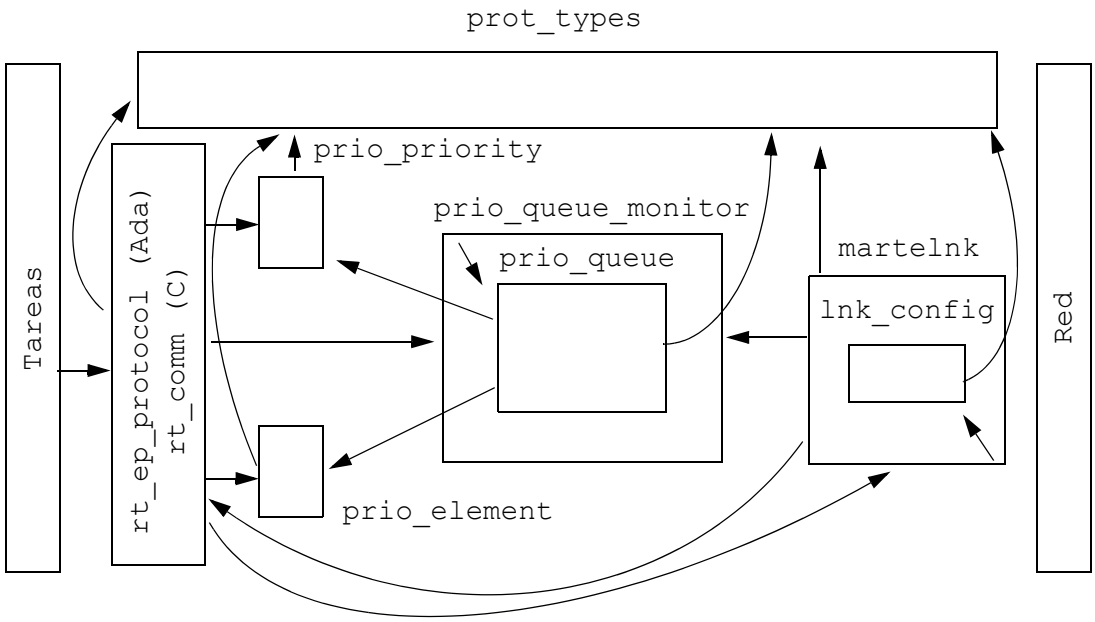



**Figura 4.8: Módulos y dependencias del protocolo**

### 4.3.7.1. rt_ep_protocol

En *rt_ep_protocol* es donde se define la interfaz Ada con la aplicación (ficheros *rt_ep_protocol.ads* y *rt_ep_protocol.adb*). Estas son las funciones que deberá utilizar la aplicación para poder comunicarse a través de la red:

- Función de inicialización del protocolo.
- Función que comprueba si existe algún elemento que recuperar de las colas de recepción.
- Funciones para mandar información a través de la red.
- Funciones para recibir información de la red.

Como se ha expuesto anteriormente, estas funciones son el resultado de realizar una cobertura abstracta sobre la interfaz C del protocolo (*rt_comm.h*). Tenemos por lo tanto distintos tipos de funciones que iremos exponiendo en este apartado.

Se definen en este módulo una serie de tipos y constantes necesarios para la correcta utilización del protocolo:

- **Max_Info_Size**: Constante que define la longitud del máximo paquete de información que se puede transmitir, actualmente 1492 bytes.
- **Ethernet_Address**: Tipo utilizado para alojar las direcciones MAC de 48 bits de la interfaz de red.
- **Rt_Ep_Priority**: Tipo utilizado para designar la prioridad del mensaje a enviar, actualmente en el rango 1..255.
- **Rt_Ep_Channel_Number**: Tipo que designa el canal hacia donde estamos transmitimos la información o sobre el canal en que la estamos recibiendo.
- **Rt_Ep_Packet_Size**: Tipo designado para contener el tamaño del paquete a transmitir o recibido.

También tenemos definidas una serie de excepciones que se elevarán en caso de error o reporte especial:

- **Station_Not_Valid**: Indica que la estación hacia la que estamos intentando transmitir, aunque inicialmente pertenecía al anillo, se ha excluido de la comunicación por no estar funcionando correctamente.
- **Station_Not_Found**: Designa que la dirección de la estación destinataria del mensaje no pertenece al anillo lógico, por tanto no puede intervenir en la comunicación.
- **Invalid_Channel**: Se eleva si intentamos utilizar una canal en la comunicación que no esta habilitado para ello.
- **Synch_Error**: Informa acerca de un error de sincronización en el objeto protegido. Es un error critico.
- **Creation_Error**: Indica que no ha sido capaz de crear las colas de recepción o la de transmisión. Al no poder inicializar las colas de prioridad, la comunicación no se puede llevar a cabo.
- **Unexpected_Error**: Si el software en alguna de sus comprobaciones encuentra algún error pero no es posible identificar su origen se elevará esta excepción informando a la aplicación de la situación de error.
- **Stream_Is_Full**: Indica que hemos sobrepasado el límite de información que la interfaz de streams es capaz de transmitir en una trama.
- **End_Of_Stream**: Si leyendo el stream llegamos al final de éste. Es decir, hemos llegado al final del *Stream'Length*.

Existe una función de inicialización del protocolo llamada ***Init_Comm*** que 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 de prioridad son globales al sistema y, por lo tanto, la función está programada para que sólo sea efectiva la primera vez que se invoque:

```
función Init_Comm (Prio_Ceiling : Integer)
       return Integer;
```

Esta función recibe como argumento la prioridad a la que queramos inicializar el *thread* de comunicaciones y a su vez el techo de prioridad de las colas de prioridad. Esta función devuelve 0 si la inicialización se establece correctamente. En caso de error puede elevar las excepciones:

- **Creation_Error**.
- **Unexpected_Error**.

Para comprobar si tenemos algún dato recibido en una determinada cola de recepción podemos utilizar la función ***Any_Info***.

```
function Any_Info
       (Channel_ID : in Rt_Ep_Channel_Number)
     return Boolean;
```

Esta función comprueba la cola de recepción especificada en *Channel_ID* y devuelve *True* en caso de existir algún elemento recibido esperando ser recogido, *False* en caso contrario. En caso de error eleva:

- **Invalid_Channel**.

- **Synch_Error**.
- **Unexpected_Error**.

En lo referente a la transmisión y recepción de datos sobre la red este módulo se divide en tres partes que constituyen tres métodos distintos de utilización del protocolo:

- **Unidades genéricas de transmisión y recepción**: Corresponden a dos procedimientos genéricos uno para transmitir y otro para recibir. Ambos procedimientos se pueden instanciar para manejar distintos tipos de datos para transmitir y recibir.
  - **Generic_Send_Info**: Se encarga de enviar el tipo *Data_Type*, que corresponde con el tipo instanciado dada la naturaleza genérica del procedimiento, a través de la red:

```
generic
     type Data_Type is private;
procedure Generic_Send_Info
     (Destination_Address : in out Ethernet_Address;
     Channel_ID : in Rt_Ep_Channel_Number;
     Data : in Data_Type;
     Size : in Rt_Ep_Packet_Size;
     Data_Priority : in Rt_Ep_Priority);
```

    En la invocación de este procedimiento para transmitir un tipo concreto se debe proporcionar como argumentos:
    - **Destination_Address**: La dirección MAC de la estación destino de la información.
    - **Channel_ID**: El identificador de canal que corresponde con la cola de recepción.
    - **Data**: Es el dato a transmitir del tipo con el que hemos instanciado el procedimiento genérico.
    - **Size**: Es el tamaño del dato a transmitir en bytes.
    - **Data_Priority**: Corresponde con la prioridad en la red de los datos a transmitir.

    En caso de error esta función eleva las excepciones:
    - **Station_Not_Valid**.
    - **Station_Not_Found**.
    - **Invalid_Channel**.
    - **Synch_Error**.
    - **Unexpected_Error**.
  - **Generic_Recv_Info**: Se encarga de recibir el tipo *Data_Type*, que corresponde con el tipo instanciado, a través de la red:

```
generic
     type Data_Type is private;
procedure Generic_Recv_Info
     (Source_Address : in out Ethernet_Address;
     Channel_ID : in Rt_Ep_Channel_Number;
     Data : out Data_Type;
```

```
        Size : out Rt_Ep_Packet_Size;
        Data_Priority : out Rt_Ep_Priority);
```

En este procedimiento tenemos como argumentos:

- **Source_Address**: La dirección MAC de la estación que envió la información.
- **Channel_ID**: El identificador de canal que corresponde con la cola de recepción de la que estamos recuperando la información.
- **Data**: Se completa con la información recibida.
- **Size**: Es el tamaño del dato que hemos recibido en bytes.
- **Data_Priority**: Corresponde con la prioridad en la red del dato recibido.

En caso de error esta función eleva las excepciones:

- **Invalid_Channel**.
- **Synch_Error**.
- **Unexpected_Error**.

- **Funciones de transmisión y recepción de streams**: Un *stream* es una secuencia de elementos compuestos por diferentes tipos proporcionando un acceso secuencial a éstos. Un tipo *stream* puede ser implementado en distintas maneras como puede ser un archivo secuencial, un *buffer* interno, o un canal de red. En nuestro caso implementamos un tipo *stream* privado *Rt_Ep_Streams_Type* como un *buffer* interno que será transmitido por la red:

```
type Rt_Ep_Streams_Type is
     new Ada.Streams.Root_Stream_Type
     with record
        Data:Stream_Element_Array (0 .. Max_Rt_Ep_Data);
        Read_Offset  : Stream_Element_Count := 0;
        Wrote_Offset : Stream_Element_Count := 0;
     end record;
```

La constante *Max_Rt_Ep_Data* fija el tamaño máximo de la información que se puede transmitir. Actualmente se iguala a *Max_Info_Size*.

Para insertar o extraer la información (tipos) del *stream* podemos utilizar las funciones atributo ‘Write o ‘Read. En esta parte del módulo se especifican las siguientes funciones:

- **Stream_Stub**: Se encarga de crear un puntero de un tipo *stream*:

```
procedure Stream_Stub
     (Rt_Ep_Stream : in out  Rt_Ep_Streams_Type;
     Rt_Ep_Stream_Ptr : out Rt_Ep_Streams_Type_Ac);
```

- **Reset_Read_Counter**: Esta función reseteará el contador de lectura del *stream* pasado como argumento, de manera que hace posible la lectura del mismo desde el principio:

```
procedure Reset_Read_Counter
       (Stream : in out Rt_Ep_Streams_Type);
```

- **Reset_Write_Counter**: Reseteará el contador de escritura a cero, así podremos escribir en el *stream* desde el comienzo:

```
procedure Reset_Write_Counter
        (Stream : in out Rt_Ep_Streams_Type);
```

- **Recv_Info**: Esta función se encargará de recibir el stream de la red:

```
procedure Recv_Info
        (Source_Address : in out Ethernet_Address;
        Channel_ID : in Rt_Ep_Channel_Number;
        Data : out Rt_Ep_Streams_Type;
        Size : out Rt_Ep_Packet_Size;
        Data_Priority : out Rt_Ep_Priority);
```

  La función leerá del canal *Channel_ID* proporcionado el elemento de mayor prioridad recibido de la estación *Source_Address* y lo almacenará en el *stream Data*. La longitud se guarda en *Size* (expresado en bytes) y la prioridad del *stream* se guardará en *Data_Priority*. Esta función actualiza el *offset* de escritura al tamaño del paquete recibido y el *offset* de lectura a 0 para que se pueda recuperar toda la información o seguir utilizando el *stream*.

  En caso de error eleva:

  - **Invalid_Channel**.
  - **Synch_Error**.
  - **Unexpected_Error**.

- **Send_Info**: Se encarga de mandar los datos a través de la red a la estación correspondiente:

```
procedure Send_Info
        (Destination_Address : in out Ethernet_Address;
        Channel_ID : in Rt_Ep_Channel_Number;
        Data : in out Rt_Ep_Streams_Type;
        Size : in Rt_Ep_Packet_Size;
        Data_Priority : in Rt_Ep_Priority);
```

  Se mandará *Size* bytes de la información contenida en *Data* con la prioridad *Data_Priority* al canal *Channel_ID* de la estación *Destination_Address* especificada. Esta función resetea los contadores de lectura y escritura del stream, así podrá ser utilizado de nuevo.

  En caso de error eleva:

  - **Station_Not_Valid**.
  - **Station_Not_Found**.
  - **Invalid_Channel**.
  - **Synch_Error**.
  - **Unexpected_Error**.

- **Read**: Función que sobrecarga la función atributo *'Read* con el stream que hemos definido:

```
procedure Read
        (Stream  : in out Rt_Ep_Streams_Type;
        Item :    out Ada.Streams.Stream_Element_Array;
        Last : out Ada.Streams.Stream_Element_Offset);
```

  En caso de error eleva:

- **End_Of_Stream**.

- **Write**: Función que sobrecarga la función atributo *'Write* con el Stream del protocolo.

```
procedure Write
       (Stream  : in out Rt_Ep_Streams_Type;
       Item : in Ada.Streams.Stream_Element_Array);
```

En caso de error eleva:

- **Stream_Is_Full**.

- **Funciones para la creación de stubs**: En esta parte se proporcionan distintas funciones basadas en *streams* que nos permitirán la creación de *stubs* entre llamadas dentro de una aplicación. Proporcionando las herramientas para la creación de aplicaciones distribuidas. Se define un tipo *stream* privado *Function_Stub_Type* que es el que se utilizará para la creación de stubs:

```
type Function_Stub_Type is
     new Ada.Streams.Root_Stream_Type
     with record
        Function_ID : Function_ID_Type;
        Data: Stream_Element_Array (0 .. Max_Data_Stub);
        Read_Offset  : Stream_Element_Count := 0;
        Wrote_Offset : Stream_Element_Count := 0;
     end record;
```

También se definen una serie de constantes:

- **Function_Stub_Overhead**: Constante que corresponde con el *overhead* en bytes añadido al *stream* por la utilización de *stubs*. Es útil para calcular el tamaño de la información recibida o el tamaño del paquete transmitido.
- **Max_Data_Stub**: Indica el tamaño de información, en bytes, máximo que se puede transmitir de una vez.

Además se definen las siguientes funciones:

- **Stream_Stub**: Esta función creará un puntero *Function_Stub_Type_Ac* del *stream Function_Stub_Type*:

```
procedure Stream_Stub
       (Function_Stub : in out  Function_Stub_Type;
       Function_Ptr : out Function_Stub_Type_Ac);
```

- **Put_Function_ID**: Esta función asignará un número de función al stream proporcionado. El número de función será el identificador del stub:

```
procedure Put_Function_ID
       (Stream : in out Function_Stub_Type;
       Function_ID : in Function_ID_Type);
```

- **Get_Function_ID**: Recupera el número de función guardado en el *stream* y que identifica al *stub*:

```
function Get_Function_ID
       (Stream : in Function_Stub_Type)
       return Function_ID_Type;
```

- **Reset_Write_Counter**: Pondrá a 0 el contador de lectura. Así podremos leer el *stream* desde el principio.

```
procedure Reset_Read_Counter
        (Stream : in out Function_Stub_Type);
```

- **Reset_Write_Counter**: Pondrá a cero el contador de escritura. Así podremos escribir en el *stream* desde el principio.

```
procedure Reset_Write_Counter
        (Stream : in out Function_Stub_Type);
```

- **Recv_Info**: Recibe un *stream* de la red:

```
procedure Recv_Info
        (Source_Address : in out Ethernet_Address;
        Channel_ID : in Rt_Ep_Channel_Number;
        Data : out Function_Stub_Type;
        Size : out Rt_Ep_Packet_Size;
        Data_Priority : out Rt_Ep_Priority);
```

  La función *Recv_Info* lee del canal *Channel_ID* el elemento *Data* de mayor prioridad enviado por *Source_Address* y de tamaño *Size* en bytes. La prioridad se almacén en *Data_Priority.* Esta función actualiza el *offset* de escritura al tamaño del paquete recibido y el *offset* de lectura a 0 para poder leer el stream desde el principio.

  En caso de error eleva:

  - **Invalid_Channel**.
  - **Synch_Error**.
  - **Unexpected_Error**.

- **Send_Info**: Se utiliza para mandar información por la red:

```
procedure Send_Info
        (Destination_Address : in out Ethernet_Address;
        Channel_ID : in Rt_Ep_Channel_Number;
        Data : in out Function_Stub_Type;
        Size : in Rt_Ep_Packet_Size;
        Data_Priority : in Rt_Ep_Priority);
```

  Se mandará la información *Data* de tamaño *Size* y con prioridad *Data_Priority* al canal *Channel_ID* de la dirección MAC *Destination_Address*. Esta función pone a 0 el contador de lectura y escritura, de esta manera podemos volver a utilizar el *stream*.

  En caso de error eleva:

  - **Station_Not_Valid**.
  - **Invalid_Channel**.
  - **Synch_Error**.
  - **Unexpected_Error**.

- **Read**: Función que sobrecarga la función atributo *'Read* con el stream que hemos definido:

```
procedure Read
        (Stream  : in out Function_Stub_Type;
        Item :    out Ada.Streams.Stream_Element_Array;
        Last : out Ada.Streams.Stream_Element_Offset);
```

En caso de error eleva:

- **End_Of_Stream**.

- **Write**: Función que sobrecarga la función atributo *'Write* con el Stream del protocolo.

```
procedure Write
       (Stream  : in out Function_Stub_Type;
       Item : in     Ada.Streams.Stream_Element_Array);
```

En caso de error eleva:

- **Stream_Is_Full**.

Como se ha descrito, este módulo nos ofrece una cobertura Ada de las funciones implementadas en el protocolo de manera que proporciona bastante flexibilidad para su utilización.

### 4.3.7.2. rt_comm

En *rt_comm* es donde se define la interfaz C con la aplicación. Estas son las funciones que deberá utilizar la aplicación escrita en C para poder comunicarse a través de la red. A continuación pasaremos a describir las funciones y los valores devueltos por éstas:

- **init_comm**: La función *init_comm* se utiliza para inicializar el protocolo:

```
int init_comm (int prio_ceiling);
```

  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*.

- **send_info**: Se utiliza para mandar datos a través de la red:

```
int send_info (char *addr_dest,
       unsigned short int channel_id,
       char *info, unsigned short int len,
       unsigned char priority);
```

  Recibe la información de la aplicación (**info*), su tamaño (*len*), la prioridad del mensaje en la red (*priority*), la estación destinataria de la información (la dirección MAC en **addr_dest*) y el canal destino (*channel_id*) y se encarga de encolar el mensaje en la cola de transmisión. El protocolo *copia* la información a transmitir a un espacio de memoria aparte, por tanto, no es necesario que la aplicación mantenga el mensaje en memoria hasta que se haya transmitido.

  La dirección *addr_dest* 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 (*char addr_eth[ETH_ALEN]*). 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.

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*, que en un principio formaba parte del anillo ha sido excluida por un mal funcionamiento de la misma 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 *len*, 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 *len*. Si se indicase mal ese campo se podrían dar mensajes incompletos si *len* 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.

- **recv_info**: Utilizada para recibir mensajes a través de la red:

```
int recv_info (char *addr_sourc,
       unsigned short int channel_id,
       char *recv_info, unsigned char *priority);
```

Esta función recibe el mensaje *recv_info* del canal especificado en *channel_id.* El mensaje corresponde al de mayor prioridad enviado a través de ese canal. La información de prioridad del mensaje es alojada en **priority* y la estación originaria del mensaje se indica en **addr_sourc*. 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.

- **Any_Info**: Nos indica si hay algún elemento en la cola de recepción indicada como argumento:

```
int any_info(unsigned int channel_id);
```

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.

### 4.3.7.3. prio_element

Este módulo está compuesto por el tipo base que almacena toda la información relacionada con el mensaje, *element_t*, y diversas funciones relacionadas con su utilización.

El tipo *element_t* contiene toda la información referida al mensaje y a la comunicación de este en la red, está definido como:

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

*element_t* es el tipo de datos que se utilizará internamente para transmitir el mensaje y la información asociada a éste. Para poder transmitir el mensaje se deberá rellenar la estructura convenientemente:

- **element_pointer:** Es un array que contendrá la información que la aplicación desea transmitir.
- **element_size:** Es un positivo (*unsigned short int*) que indica el tamaño de los datos apuntados por *element_pointer*. El valor máximo es MAX_INFO_LENGTH que actualmente 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 que corresponderá, dependiendo del contexto, a la dirección de la estación destino del mensaje o fuente de éste.
- **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.

Se define una función de comparación de tipos *element_t*:

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

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.

Para un uso eficiente de la memoria utilizada por el protocolo en sistemas como el que nos ocupa, en los que la utilización de memoria estática es no sólo preferible sino que en la mayoría de los casos imprescindible, se define en este módulo un mecanismo de reserva y liberación de memoria adecuada para el tipo *element_t*. Estas funciones operan sobre una lista enlazada de punteros, implementada por un array de tamaño MAX_MEMORY_ELEMENT_TYPE, a tipos *element_t* disponibles. Esta lista enlazada almacenará punteros hacia tipos *element_t* en desuso de manera que podrán ser utilizados más tarde.

Las funciones necesarias para utilizar este mecanismo:

- **element_monitor_init**: Con esta función inicializamos el monitor de acceso mutuamente exclusivo a la lista enlazada de elementos *element_t* e inicializamos algunas variables globales. Es necesario invocar esta función antes de la utilización del mecanismo de reserva de memoria de *element_t*. Esta llamada solo es efectiva la primera vez que se invoca:

  ```
  int element_monitor_init(int prio_ceiling);
  ```

  Recibe el techo de prioridad que queremos asignar al recurso compartido. Éste deberá ser al menos la prioridad de *thread* de mayor prioridad que utilice *element_t* mediante este mecanismo.

  Esta función devuelve:

  - **0** si no se produce ningún error.
  - **CREATION_ERROR** en caso de algún error en la creación del monitor de acceso.

- **alloc_element**: Esta función devuelve un puntero *element_t* a un área de memoria ya reservada:

  ```
  element_t *alloc_element();
  ```

  *alloc_element* comprueba si existen algún elemento en la lista enlazada para poder ser reutilizado, en caso de existir se devolverá el puntero y se eliminará de la lista enlazada. Si no existe ningún puntero disponible se reserva memoria con *malloc*.

  Esta función devuelve NULL en caso de error.

- **free_element**: "Libera" un puntero *element_t* que ya no está en uso:

  ```
  int free_element(element_t *E);
  ```

  El puntero *element_t* pasado como argumento se marca como no utilizado y pasa a formar parte de la lista enlazada dispuesto a ser utilizado en una posterior llamada a *alloc_element*.

  Esta función devuelve:

- **0** si no se produce ningún error.
- **MEM_LOST** si llenamos la lista con elementos vacíos, es decir, si intentamos liberar más de MAX_MEMORY_ELEMENT_TYPE elementos. En este caso estaríamos desaprovechando memoria ya que el puntero *element_t* "liberado" no podrá volverse utilizar al no poder guardar registro de él desperdiciando con esto memoria. Compete al implementador ajustar el parámetro correctamente para que esto no suceda.

### 4.3.7.4. prio_priority

En este módulo definimos el tipo prioridad:

```
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.

### 4.3.7.5. 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 4.3.5, se trata de colas de prioridades implementadas como montículos y un comportamiento FIFO para los elementos de la misma prioridad. A continuación expondremos los tipos de datos definidos para implementar las colas de transmisión y de recepción:

```
typedef struct cell_t_t {
     priority_t Pri;
     element_t *E;
     struct cell_t_t *next_cell;
} cell_t;

typedef struct {
     cell_t *first;
     cell_t *last;
     boolean_t dirty;
} FIFO_queue_t;

typedef struct queue_t_t {
     positive_t Length;
     positive_t queue_size;
     FIFO_queue_t FIFO_queue[PRIORITY_NUM];
     cell_t *Q;
} queue_t;
```

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 además de un puntero a otra celda del mismo tipo.

A continuación tenemos el tipo correspondiente a la cola FIFO, *FIFO_Queue_t*, que consiste en un puntero al comienzo de la cola FIFO destinado a extraer celdas y otro puntero al final de la cola para poder insertar celdas en la FIFO. Conviene destacar la definición del booleano *dirty* que nos ayudará a hacer más eficientes las inserciones en la

cola, ya que valdrá *True* si ya existe un elemento de la misma prioridad en el árbol binario lo que nos indicara que en ese momento hay que insertar los elementos de esa prioridad en la cola FIFO.

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. También están definidas tantas colas FIFOs como prioridades posibles en el sistema. Por lo tanto en esta implementación el montículo sólo almacenará como máximo un elemento de cada prioridad. El resto, de existir, irían a la cola FIFO para cada prioridad.

Las funciones definidas en este módulo:

- **init_queue**: para inicializar la cola:

```
void init_queue(queue_t *Q);
```

- **empty**: para indicar si una cola en particular está vacía. Devuelve *True* si lo está y *False* si no:

```
boolean empty(queue_t *Q);
```

- **queue_length**: nos devuelve la longitud de la cola.

```
positive_t queue_length(queue_t *Q);
```

- **enqueue**: se encarga de encolar un *element_t* y *priority_t* en la col:

```
int enqueue(element_t *E, priority_t P, queue_t *Q);
```

- **dequeue**: se encarga de sacar de la cola el elemento de mayor prioridad:

```
int dequeue(element_t **E, priority_t *P, queue_t *Q);
```

- **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.

```
int read_first(element_t **E,
       priority_t *P, queue_t *Q);
```

## 4.3.7.6. prio_queue_monitor

Tal y como se comentaba en el apartado 4.3.4, “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:

- **create_queue**: Utilizada para crear e inicializar las colas de prioridad:

```
int create_queue(int prio_ceiling, queue_t *Q,
       const int queue_num);
```

  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*. 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 4.3.4, "Sincronización de threads":

```
int queue_insert(element_t *E, priority_t P,
       queue_t *Q, const int queue_num);
```

  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.

- **queue_extract**: Se utiliza para extraer el elemento de mayor prioridad de la cola *queue_t* a través del monitor:

```
int queue_extract(element_t **E,priority_t *P,
       queue_t *Q, const int queue_num);
```

  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 4.3.4. 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.

- **read_first_in_queue**: Esta función no extrae ni encola ningún elemento de la cola, sino que su cometido es leer el elemento de mayor prioridad:

```
int read_first_in_queue(element_t **E,
       priority_t *P, queue_t *Q,const int queue_num);
```

  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.

- **is_queue_empty**: Nos indicará si la cola de prioridades está o no vacía:

```
int is_queue_empty(element_t *E,queue_t *Q,
       const int queue_num);
```

  Devuelve:

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

En este módulo 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.

### 4.3.7.7. lnk_config

Este módulo se encargará de obtener, de un fichero de cabecera, la configuración del anillo. De este modo la estación será capaz de situarse en el anillo y recuperarse de algún posible error debido al fallo de alguna estación. MaRTE OS, siguiendo las recomendaciones del subconjunto de Tiempo Real mínimo POSIX.13, no usa un sistema de ficheros, es por ello que la configuración en vez de proporcionarse como un archivo se haga como un fichero de cabecera.

Este módulo proporciona un tipo, *lnk_cell_t*, que será el utilizado para averiguar las estaciones que componen el anillo y si están o no operativas:

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

Como puede verse, cada estación se define en 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.

Las funciones necesarias para modificar o extraer información de la configuración del anillo son:

- **init_config**: Inicializa la estructura de datos y se almacenen en ella toda la configuración del anillo lógico:

  ```
  int init_config();
  ```

  Esta función se encarga de acceder a la variable global definida en el fichero de cabecera y extraer la configuración del anillo. El formato del fichero de cabecera se expondrá en 4.3.7.8, “ring_config”.

  Esta función devuelve:

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

- **read_station**: Se utiliza 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:

```
int read_station(lnk_cell_t *sta, positive_t node);
```

  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.

- **number_of_nodes**: Nos proporciona el número de estaciones que hay en el anillo lógico:

```
positive_t number_of_nodes();
```

  Devuelve el número de nodos (estaciones) existentes en la configuración.

- **check_station**: Se utiliza para comprobar si una estación está activa, es decir, si sigue perteneciendo al anillo:

```
int check_station(unsigned char *st_addr);
```

  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.

- **invalidated_station**: Comprueba si una estación está marcada como no valida:

```
int invalidated_station(unsigned char *st_addr);
```

  Debido a que *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.

- **inhibit_station**: Marca una estación como no válida:

```
int inhibit_station(unsigned char *st_addr);
```

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

La función 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 se incluye el archivo de cabecera que contiene la configuración del anillo.

### 4.3.7.8. ring_config

La configuración del anillo se realiza mediante el archivo de cabecera *ring_config.h*. En este archivo se definirán las direcciones MAC de las estaciones que conforman el anillo. Las direcciones se proporcionarán en formato texto, cada 8 bits de dirección en hexadecimal, dos puntos como se muestra a continuación:

```
00:4F:4C:02:F6:68
```

En este archivo de cabecera se define la constante que establecerá el anillo lógico:

```
static const char stations[][STATIONS_MAC_ASC_LEN]={
     {"00:08:9B:80:05:B2"},
     {"00:08:9B:80:09:60"},
};
```

La etiqueta STATIONS_MAC_ASC_LEN está definida como la longitud de una dirección MAC en formato texto, es decir, 18. Siguiendo la plantilla de dos estaciones apuntada anteriormente es posible conformar el anillo lógico para más estaciones. La estación colocada en primer lugar será la que desempeñe el papel de *token_master* en la primera vuelta del testigo.

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.

### 4.3.7.9. 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. Los paquetes del protocolo son: *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.

```
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;
```

  - *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:

```
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;
```

  - *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 se definen a través de *struct ethhdr* (en *if_ether.h*). 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.

- **martelnk**: Es la función que ejecutará el *thread* de comunicaciones:

```
int 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.

## 4.3.7.10. prot_types

Este módulo contiene multitud de parámetros de configuración del protocolo. Estos parámetros se desarrollarán en el apartado 4.3.8, “Configuración del protocolo”. Además también están presentes algunos tipos de datos y funciones que han sido usados a lo largo del protocolo:

- **boolean_t**: Tipo booleano.

```
typedef enum {false=0, true=1} boolean_t;
```

- **positive_t**: Un tipo de valores positivos de 16 bits:

```
typedef unsigned short int positive_t;
```

- **addrcmp**: Compara dos arrays que contienen direcciones MAC de estaciones:

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

  Recibe dos punteros a direcciones MAC y devuelve TRUE si son iguales y FALSE si son distintas.

- **addrcpy**: Copia direcciones MAC.

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

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.

### 4.3.7.11. eth_tools

En este módulo se implementan dos funciones definidas en *if_ether.h* que son utilizadas por el protocolo:

- **ether_ntoa**: Para convertir de direcciones MAC de 48 bits a ASCII:

```
char *ether_ntoa (const struct ether_addr *addr);
```

Esta función convierte la dirección MAC *addr* a un string en el formato estándar de cada byte en hexadecimal dos puntos. El *string* es devuelto es un *buffer* estático, por lo tanto sucesivas llamadas a *ether_ntoa* lo sobreescribirán.

- **ether_aton**: Convierte direcciones MAC en formato texto (cada byte en hexadecimal seguido de dos puntos) a el número de 48 bits que corresponde con la dirección MAC de una estación:

```
struct ether_addr *ether_aton (const char *asc);
```

El puntero devuelto apunta a una estructura estática en el interior del módulo, por lo tanto sucesivas llamadas sobreescribirán el resultado. En caso de error devuelve NULL.

### 4.3.7.12. rt_ep_errorn

En este archivo de cabecera se definen todos los códigos de error devueltos por el protocolo de manera que se evitan posibles confictos debidos al valor numérico de devuelven.

Actualmente está compuesto por:

- **CREATION_ERROR:** Indica que no ha sido capaz de crear las colas de recepción o la de transmisión. Al no poder inicializar las colas de prioridad, la comunicación no se puede llevar a cabo.
- **SYNCH_ERROR:** Informa acerca de un error de sincronización en el objeto protegido. Es un error critico.
- **UNEXPECTED_ERROR:** Si el software en alguna de sus comprobaciones encuentra algún error pero no es posible identificar su origen se elevará esta excepción informando a la aplicación de la situación de error.
- **STA_NOT_VALID:** Indica que la estación hacia la que estamos intentando transmitir, aunque inicialmente pertenecía al anillo, se ha excluido de la comunicación por no estar funcionando correctamente.
- **STA_NOT_FOUND:** Designa que la dirección de la estación destinataria del mensaje no pertenece al anillo lógico, por tanto no puede intervenir en la comunicación.

- **STATIONS_OVERFLOW:** Indica que el archivo de cabecera que contiene la configuración del anillo es incorrecto y hay descritas en él más estaciones de las que el protocolo tiene configuradas para manejar.
- **INFO_LENGTH_OVERFLOW.** Expresa que los datos a transmitir, indicados en el campo *len*, superan el tamaño del campo de datos del paquete de información de RT-EP.
- **INVALID_PRIORITY**.: Indica que 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:** Indica que intentamos utilizar un canal en la comunicación que no esta habilitado para ello.
- **NO_ELEMENTS:** Indica que no hay ningún elemento en la cola de recepción de un canal determinado.

### 4.3.8. 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 cabecera *prot_types.h* menos la configuración del anillo que, como ya se ha visto, se encuentra definido en el archivo *ring_config.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.

En la configuración del protocolo, a través de prot_types.h, en primer lugar tenemos definida la constante PRIORITY_NUM que informa acerca del número de prioridades existentes en el protocolo, en este caso 256:

```
#define PRIORITY_NUM 256
```

A continuación definimos el tamaño de las colas de prioridad. PRIO_INITIAL_SIZE corresponde con el número máximo de elementos que se podrán almacenar en cada montículo. Como se explicó en 4.3.7.5, “prio_queue” este valor es el número de prioridades en el sistema ya que de haber más de un mensaje por prioridad éste irá a las colas FIFO definidas para cada priorida:

```
#define PRIO_INITIAL_SIZE PRIORITY_NUM
```

Seguidamente, tenemos que definir el número máximo de canales que se utilicen en el protocolo. Normalmente corresponderá con el número de tareas o *threads* que quieran comunicarse a través de la red, a cada tarea o *thread* le corresponderá un canal de comunicaciones:

```
#define MAX_CHANNELS 10
```

El número de canales será, en este caso, desde el canal 1 hasta el canal 10 ambos incluidos. Este parámetro también determina el número de colas de recepción presentes en la estación, una por canal.

A continuación nos encontramos con la definición del número máximo de elementos *element_t* que pueden ser manejados por el protocolo. Este valor es muy importante ya que es el que se utilizará en la reserva / liberación de memoria que utiliza el protocolo para el tipo *element_t*:

```
#define MAX_MEMORY_ELEMENT_TYPE ((MAX_CHANNELS + 1) *
        PRIO_INITIAL_SIZE)
```

Más adelante definimos el número máximo de estaciones admitidas en la configuración del protocolo. Este parámetro se especifica de la siguiente manera:

```
#define MAX_STATIONS 100
```

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 DEVICE_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, si se excede este valor.

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 80000
```

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 $10^9$ 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*.

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 250000
#define MAX_RETRIES 1
```

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 $10^9$ 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' veces y al siguiente reintento se excluirá la estación en fallo del anillo lógico.

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_DISCARDED. 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_DISCARDED 5
```

- **CPU_OVERHEAD_ABS**: En este modo se accede a una visión global de las medidas tomadas con CPU_OVERHEAD_PROTOCOL_TIMES. A través de este modo obtenemos la utilización del protocolo. Como resultado comunica el tiempo total de la medida, el tiempo utilizado por el protocolo y la utilización en tanto por ciento. Para activar este modo también tiene que estar activado el modo de tiempos de protocolo (CPU_OVERHEAD_PROTOCOL_TIMES):

```
#define CPU_OVERHEAD_ABS
```

- **TIME_MEASURE**: Habilitando este modo habilitamos una funcionalidad de medición de tiempos externa. Esta funcionalidad es capaz de obtener mediciones de partes de código de interés y éstas son recuperables a fichero desde una máquina remota:

```
#define TIME_MEASURE
```

- **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, sea la *token_master*. Los tiempos mostrados son los relativos al peor caso, al mejor caso y a la media de todos los tiempos medidos descartando los primeros TOKEN_POINTS_DISCARDED

medidas. 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_POINTS_DISCARDED 1
```

El protocolo es bastante configurable en fase de compilación, requisito imprescindible en MaRTE OS que, siguiendo la norma POSIX.13 para sistemas mínimos de Tiempo Real [33], carece de ficheros.

# 5. Modelado MAST

## 5.1. 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) [14], 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, a su vez, 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.

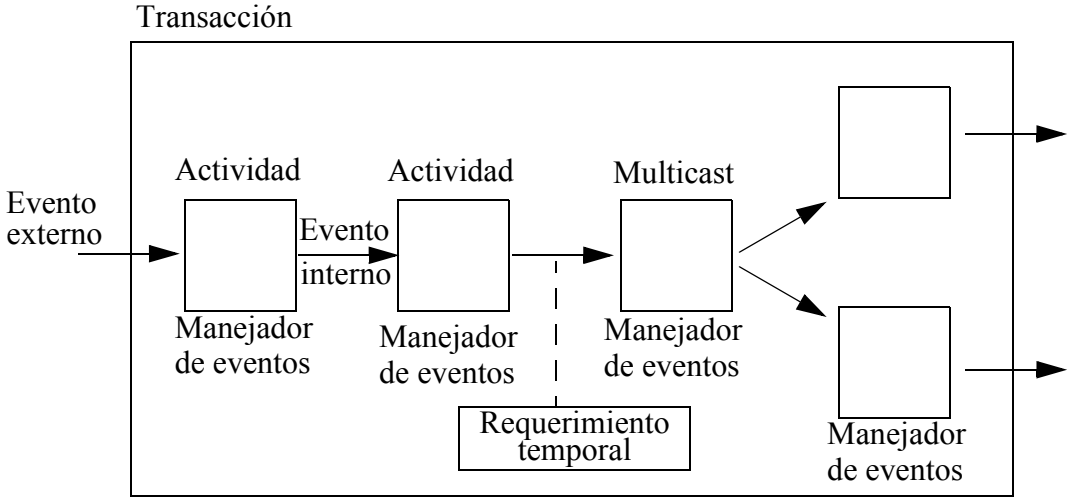



**Figura 5.1: Sistema de Tiempo Real compuesto de transacciones [10]**

La figura 5.1 [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 5.2 [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].

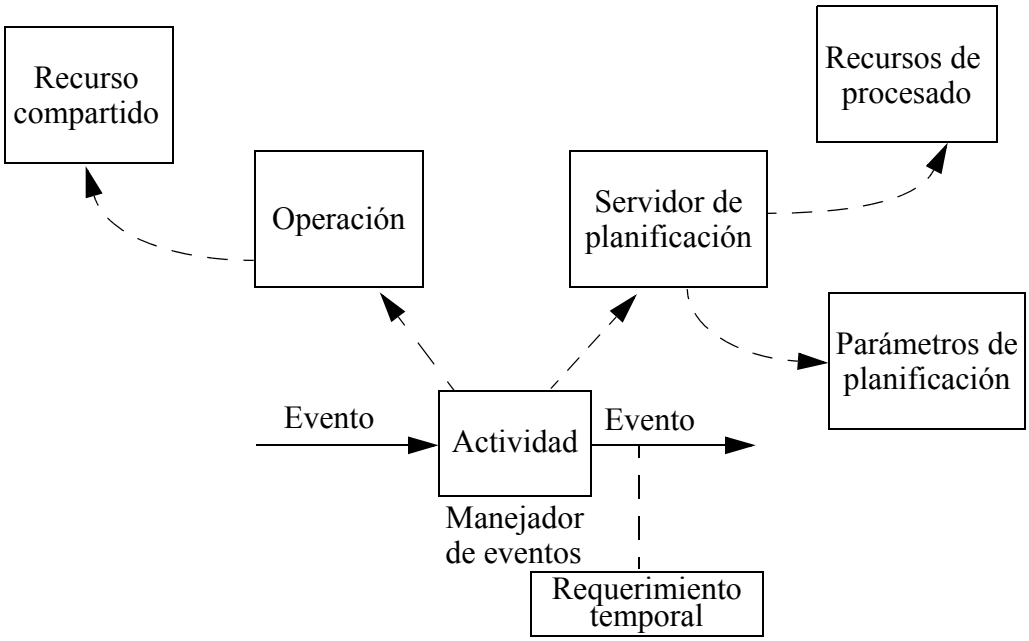



**Figura 5.2: Elementos que definen una actividad [10].**

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 [14]:

- **Recursos de procesado** (*Processing Resources*): Modelan la capacidad de procesado de un componente *hardware* que ejecuta alguna de las actividades del sistema modeladas. Existen dos clases:

- Procesador normal (*Regular Processor*): Es un procesador con los *overheads* básicos asociados a sus servicios de interrupcion *hardware*.
- Red de paquetes (*Packet Based Network*): Representa una red que utiliza un tipo de protocolo de tiempo real basado en paquetes no expulsables utilizados para mandar los mensajes. Esta clase tiene los siguientes atributos adicionales:
  - Tipo de transmisión (*Transmission Kind*): Indicará si la comunicación es Simplex, Half Duplex o Full Duplex.
  - Rendimiento (*Throughput*): Ancho de banda normalizado en bits por unidad de tiempo.
  - Máximo bloqueo (*Max Blocking*): El máximo bloqueo es causado por no poder expulsar los paquetes.
  - Tamaño máximo y mínimo de paquete (*Max Packet Size* y *Min Packet Size*): Describen la cantidad de de datos incluidos en un paquete, excluyendo la informacion de protocolo.
  - 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.

- **Planificadores** (*Schedulers*): Representan los objetos del sistema operativo que implementan las estrategias de planificación adecuadas para manejar la capacidad de procesado que se les ha asignado. Existen dos tipos de planificadores dependiendo de la fuente de la capacidad de procesado:
  - Planificador primario (*Primary Scheduler*): Representa el planificador base del sistema para el recurso de procesado asociado a éste. Esta clase tiene los siguientes atributos:
    - Politica de planificación (*Scheduling Policy*).
    - Anfitrion (*Host*): Identificador del recurso de procesado al que está asignado.
  - Planificador secundario (*Secondary Scheduler*): Es un planificador que sólo es capaz de administrar una fracción de la capacidad de procesado, que por turnos es proporcionada por un servidor de planificación determinado asociado a un planificador.
    - Politica de planificación (*Scheduling Policy*).
    - Servidor (*Server*): Identificador del servidor de planificación al que está asociado.
- **Política de planificación** (*Scheduling Policy*): Representa la estrategia que está implementada en el planificador para administrar la capacidad de procesado asignada a los sevidores de planificación asociados. Cada uno de estos servidores posee unos parámetros de planificación que describen los parámetros utilizados por el planificador. Tenemos las siguientes clases de políticas de planificación:
  - Prioridad fija (*Fixed Priority*): Representa una política de prioridades fijas. Posee los siguientes atributos:
    - *Context Switch Overheads* (*Worst, Average, Best*): Indican los overheads asocidados al cambio de contexto.
    - *Max Priority* y *Min Priority*: Indican el rango de prioridades válidos para las operaciónes sobre los servidores de planificación planificados con esta política.
  - EDF (*Earliest Deadline First*):
    - *Context Switch Overheads* (*Worst, Average, Best*): Indican los overheads asocidados al cambio de contexto.
  - Prioridad fija basada en paquetes (*Fixed Priority Packet Based*): Representa una política de prioridades fijas utilizada en una red de comunicaciones por paquetes. Los paquetes de la red se asumen que no son expulsables.
    - Overhead asociado al paquete (*Packet Overhead*): Este es el (peor, medio y mejor) *overhead* asociado a cada paquete producido por los mensajes de protocolo o cabeceras que deben ser mandados antes o después de cada paquete útil.

- *Max Priority* y *Min Priority*: Indican el rango de prioridades válidos para los mensajes mandados con esta política.

- **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]:
  - Parámetros de planificación de prioridades fijas (*Fixed Priority Scheduling Parameters*):
    - *Non Preemptible Fixed Priority Policy*
    - *Fixed Priority Policy*
    - *Interrupt Fixed Priority Policy*
    - *Polling Policy*
    - *Sporadic Server Policy*
  - Parámetros de planificación de EDF (*EDF Scheduling Parameters*):
    - *Earliest Deadline First Policy*
- **Parámetros de sincronización** (*Synchronization Parameters*): Estos parámetros están asociados a un servidor de planificación y especifican los parámetros utilizados por ese servidor cuando realiza un acceso mutuamente excluyente a un recurso compartido.
- **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*
  - *Stack Based Resource*: Basado en el protocolo Stack Resource Protocol (SRP)
- **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*)
  - Único (*Singular*)
  - Esporádico (*Sporadic*)
  - No acotado (*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.

## 5.2. 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.

### 5.2.1. 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 *Packet Based Network* y la política de planificación *Fixed Priority Packet Based* pertenecientes, respectivamente, al *Processing Resource* y del *Scheduler* primario 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 Interrupt Server*: Es el *Scheduling Server* que está ejecutando la rutina de interrupción que almacena cada trama recibida en la red independientemente de si se trata de un paquete de información o de arbitrio.
- *Packet ISR Operation (ISR)*: Corresponde con la operación ejecutada por el servidor de interrupción cada vez que se recibe una trama, independientemente de la naturaleza del paquete.
- *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.
- *Message_Partitioning*: Indica si el *driver* es capaz de particionar mensajes de gran longitud y reconstruirlos en el destino. Los posibles valores son Yes o No. En nuestro caso tendrá un valor de No.

- *RTA_Overhead_Model*: Es un atributo exclusivo de la herramienta de análisis del tiempo de respuesta en el peor caso. El valor determina el modelo de overherad que será utilizado para el driver. Los posibles valores son:
  - *Coupled*: En este modelo de *overhead* una operación de envio y recepción de mensaje se adjuntan a la transacción que causa la transmisión.
  - *Decoupled*: En este modelo, la operación de envío y recepción se ejecutan periódicamente con un periodo igual al tiempo de transmisión mínimo (o a algún otro tiempo dependiente del driver).

  Ambos modelos son pesimistas, pero el modelo *coupled* es más apropiado en sistemas con una poca utilización de la red, mientras que el modelo *decoupled* es más apropiado para sistemas con una gran utilización de la red.

*Packet Based 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 información, excluyendo cualquier información de protocolo. Corresponde con el tiempo que se tarda en enviar los bytes de información de un paquete del protocolo con el campo de información máximo y mínimo, que puede ser 1492 en el caso del tiempo máximo de transmisión o 46 en el caso del tiempo mínimo. Como vimos en el apartado 5.1, 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, 8 de información del protocolo, y 4 del campo FSC) en el caso de un tamaño de datos de 1492 bytes y de 72 bytes en el caso de un tamaño de datos de 46 bytes, podemos obtener:

  *Max Packet Tranmission Time*:

$$MaxPTT = \frac{1492 \cdot 8}{Rb}$$

  En *MaxPTT* se ha utilizado 1492 bytes que es el tamaño máximo de información que se puede transmitir en un paquete de protocolo. El *overhead* (34 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...)

- *Max Packet Size* y *Min Packet Size*: Según lo expuesto en el apartado anterior tenemos un valor de 1492 bytes para el paquete máximo y de 72 bytes para el paquete mínimo.
- *Throughput*: Corresponde con el regimen binario del medio.
- *Max Blocking*: El máximo bloqueo es causado por la no predecibilidad de los paquetes. En este protocolo se calcula de la siguiente manera:

$$(N) \cdot (MinPTT + ISR + TCO + TMO) + (N-1) \cdot TD + ISR + MaxPTT + \frac{34 \cdot 8}{R_b}$$

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 5.3:

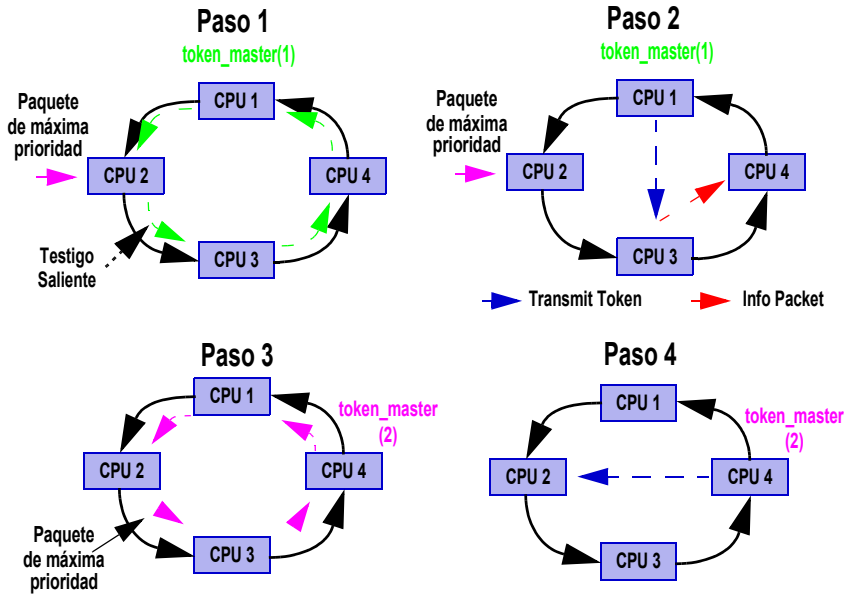



**Figura 5.3: Máximo bloqueo en RT-EP.**

- **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

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

- **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:

$$(MinPTT+ISR+TCO+TMO)+MaxPTT+ISR+\frac{34\cdot 8}{R_b}$$

- **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 5.3, 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.

*Planificador primario* con política *Fixed Priority Packet Based*: Seguidamente caracterizaremos este planificador asociado al recurso de procesado de la red. Se caracteriza por:

- *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:

$$(N+1)(MinPTT+ISR+TCO+TMO)+N\cdot TD+\frac{34\cdot 8}{R_b}$$

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 *Packet ISR Operation* (*ISR*), 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 Priority* y *Min Priority*: El protocolo posee de 255 prioridades, siendo la prioridad más baja la 1 y la más alta la 255.

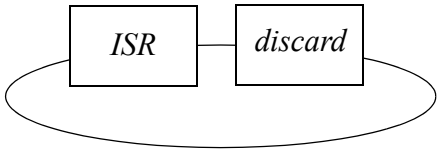
## 5.2.2. Caracterización interna del driver

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.

Conviene aclarar que todas las actividades son ejecutadas por el servidor de planificación *Packet Server* menos la actividad ISR que la ejecuta el servidor *Packet Interrupt Server*.

Las transacciones son debidas a:

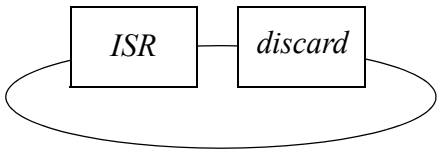
- **Testigos descartados:** Esta transacción está motivada por el hecho de utilizar el modo promiscuo en las estaciones. Todas las estaciones reciben, en todo momento, todas las tramas arrojadas al medio. Lo que conlleva a una actividad, ISR, debida a la recepción del paquete ejecutada por el servidor *Packet Interrupt Sertver* y a una actividad *discard*, ejecutada por el servidor *Packet Server*, 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 5.4, a la suma del mejor caso de la operación *Packet ISR Operation*, 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*.




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

**Figura 5.4: Transacción de descarte de testigos.**

- ***Transmit Token* 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 de *Packet ISR Operation*, 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 *Packet ISR Operation*, la operación *Token Check Operation*, más el mejor caso de las operaciones *Packet Send Operation* o *Token Manage Operation*. Este caso se encuentra incluido en la transacción correspondiente al caso del *transmit token siendo token_master.*
- **Info Packet recibido** (en el caso de que se transmita información): Esta transacción corresponde con el caso de que se reciba información transmitida por una estación. Distinguimos tres casos:
  - **La estación no es la destinataria del mensaje**: En este caso se descartará el paquete de informacion. Su periodo corresponderá con la suma del mejor caso de la operación *Packet ISR Operation*, 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 *Packet ISR Operation, Token Manage Operation* y *Token Check Operation*, más el tiempo de transmisión del paquete *Transmit Token* (*Min PTT*), más el tiempo de *Packet ISR Operation*, más el mejor caso de la operación *Token Check Operation* y *Packet Send Operation*. Este caso no puede ser simultáneo al testigo normal descartado.

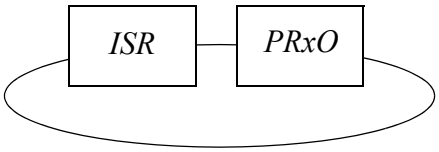



$T = ISR + CS + discard +$ Tiempo mínimo de rotación del testigo + $ISR + TMO + TCO + MinPTT + ISR + TCO + PSO$

**Figura 5.5: Transacción de descarte de información.**

  - **La estación es la destinataria del mensaje (Con RTA_Overhead_Model Decoupled):** En este caso el paquete se procesa en la estación destino. Su periodo corresponderá con la suma del mejor caso de la operación *Packet ISR Operation*, del cambio de contexto (*CS*) para la ejecución de la operación *Packet Receive Operation*, de la propia operación *Packet Receive Operation*, del tiempo mínimo de paso de testigo, más la suma del mejor caso de las operaciones *Packet ISR Operation, Token Manage Operation* y *Token Check Operation*, más el tiempo de transmisión del paquete *Transmit Token* (*Min PTT*), más el tiempo

de *Packet ISR Operation*, más el mejor caso de la operación *Token Check Operation* y *Packet Send Operation.*




$T = ISR + CS + PRxO +$ Tiempo mínimo de rotación del testigo + $ISR + TMO + TCO + MinPTT + ISR + TCO + PSO$

**Figura 5.6: Transacción de recepción de información *decoupled*.**

Debido a que el periodo en el que suceden ambos supuestos tienen que coincidir ya que son simultáneos en la red, tomamos como peor caso, en *Decoupled*, el de la transacción de recepción de información ya que incluye el efecto del descarte de información y es el que supone un mayor overhead en la estación, es más costoso procesar un paquete que descartarlo

- **La estación es la destinataria del mensaje (Con RTA_Overhead_Model Coupled):** En este caso, en vez de suponer la transacción de recepción periódica, se añade la actividad de recepción mostrada en la figura 5.7 a la transacción de recepción.

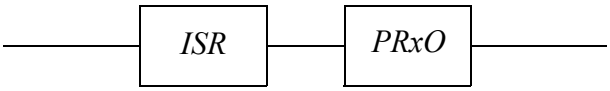
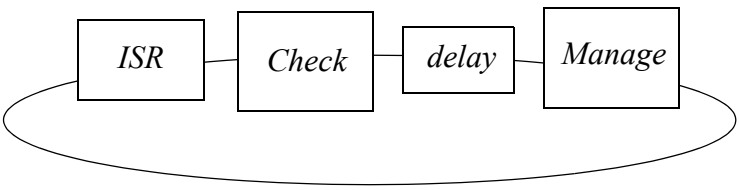



**Figura 5.7: Transacción de recepción de información *coupled*.**

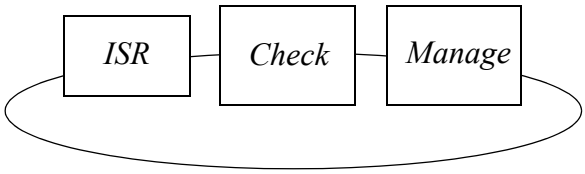
- **Paso del testigo** (sin intervenir en la comunicación): Esta situación está definida por aquellas estaciones que reciben, actualizan y mandan el testigo sin intervenir en la comunicación de información en ese paso de testigo. El "peor" caso ocurre cuando se transmiten testigos de prioridad 0. Esta transacción engloba al conjunto de actividades, ilustradas en la figura 5.8, 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 mejor caso de la operación *Packet ISR Operation*, más el tiempo del mejor caso de *Packet Check Operation*, más el delay configurado, más el mejor caso del *Tocken Manage Operation*, más el tiempo mínimo de rotación del testigo.




$T = ISR + TCO + delay + TMO +$ Tiempo mínimo de rotación del testigo

**Figura 5.8: Transacción del paso de testigo.**

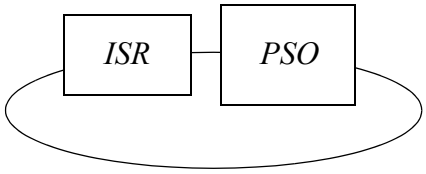
- **Envío *Transmit Token* siendo *token_master*** (sin intervenir en la comunicación): En este caso tenemos una transacción, ilustrada en la figura 5.9, que contendrá una actividad que se ejecutará cada vez que vuelva el testigo a la estación *token_master*. Su periodo será la suma del mejor caso de las operaciones *Packet ISR Operation, Token Check Operation* y *Token Manage Operation*, más el tiempo de transmisión del *Transmit Token* (*Min PTT*), más el tiempo de *Packet ISR Operation, Token Check Operation* y *Packet Send Operation*, más el tiempo de transmitir un mensaje de información de tamaño mínimo (*Min PTT*), más el tiempo de recepción del paquete de información, es decir, las operaciones *Packet ISR Operation* y *Packet Receive Operation,* más *el* tiempo mínimo de rotación del testigo. Todo esto suponiendo que volviese a ser la *token_master*.




$T = ISR + TCO + TMO + MinPTT + ISR + TCO + PSO + MinPTT + ISR + PRxO +$ Tiempo mínimo de rotación del testigo

**Figura 5.9: Transacción del *Transmit Token***

- **Envio de un Info Packet**: En este caso, al igual que en el caso del *info_packet* descartado, tenemos dos casos bien diferecciados:
  - **Caso desacoplado** (*decoupled*): Como se expone en la figura 5.10, tenemos una transacción que contendrá una actividad que se ejecutará cada vez que se transmita información al medio. Como peor caso suponemos que es la estación la que siempre está transmitiendo información. Su periodo será será la suma del mejor caso de las operaciones *Packet ISR Operation* y *Packet Send Operation*, más el tiempo de transmitir un mensaje de información de tamaño mínimo (*Min PTT*), más el tiempo de recepción del paquete de información, es decir, las operaciones *Packet ISR Operation* y *Packet Receive Operation,* más *el* tiempo mínimo de rotación del testigo, más el tiempo debido al efecto del *Transmit Token*, esto es, *Packet ISR Operation* más *Token Check Operation* más *Token Manage Operation*, más el tiempo de transmisión del *Transmit Token* (*MinPTT*).




$T = ISR + PSO + MinPTT + ISR + PRxO +$ Tiempo mínimo de rotación del testigo $+ ISR + TCO + TMO + MinPTT$

**Figura 5.10: Transacción del envío de un Info Packet *decoupled***

- **Caso acoplado (*coupled*)**: En este caso, en vez de suponer la transacción de transmisión periódica, se añade la actividad de transmisión mostrada en la figura 5.11 a la transacción de envío.

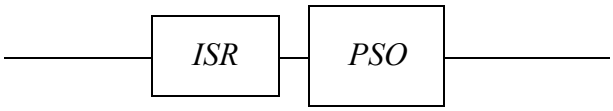



**Figura 5.11: Transacción del envío de un Info Packet *decoupled***

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

## 5.2.3. 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", sólo 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 las operaciones y atributos introducidos por el tratamiento de errores:

- *Packet Server*: Es el mismo que en el modelo sin fallos.
- *Packet Interrupt Server*: Permanece inalterado.
- *Packet ISR Operation (ISR)*: Es el mismo que en el caso sin error.
- *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* (*TD*): Permanece inalterado.
- *Packet Discard Operation* (*PDO*): Permanece inalterado.
- *Token Transmission Reties* (*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 Retries* (*PR*): Con este valor se indicará el número máximo de retransmisiones que se producirán en un fallo al transmitir un *Info Packet*.
- *Failure 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 el estado *Error_Check* y 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 el estado *Error_Check* y la otra parte del estado *Error_Handling* que se encarga de retransmitir el paquete de información (*Info Packet*).
- *Message_Partitioning*: Permanece inalterado.
- *RTA_Overhead_Model*: Es dependiente de la utilización de la red.

*Packet Based 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*:

$$MaxPTT = \frac{1492 \cdot 8}{Rb}$$

  *Min Packet transmisión Time*:

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

- *Max Packet Size* y *Min Packet Size*: Según lo expuesto en el apartado anterior tenemos un valor de 1492 bytes para el paquete máximo y de 72 bytes para el paquete mínimo.
- *Throughput*: Corresponde con el regimen binario del medio
- *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:

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

$$\left(MaxPTT + \frac{34 \cdot 8}{R_b} + PR \cdot (PRO + T)\right) +$$

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

  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.

*Planificador primario* con política *Fixed Priority Packet Based*: Seguidamente caracterizaremos este planificador asociado al recurso de procesado de la red. Se caracteriza por:

- *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 Retries* (*TR*). Por ello el *Packet Overhead* se calculará:

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

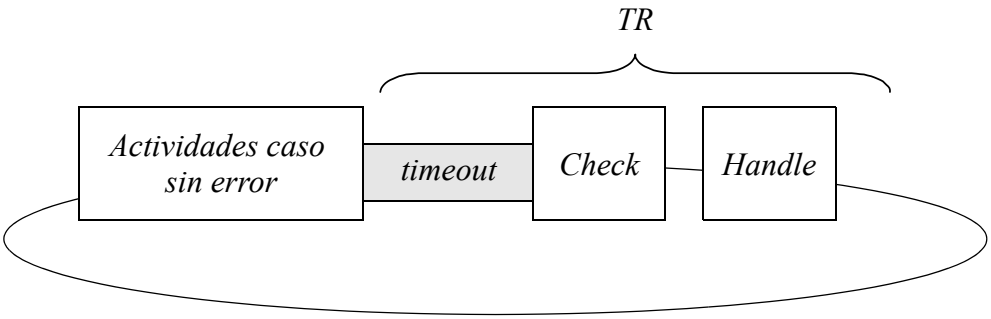
$$(TRO + T) \cdot TR + \frac{34 \cdot 8}{R_b}$$

  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 Priority* y *Min Priority*: El protocolo posee de 255 prioridades, siendo la prioridad más baja la 1 y la más alta la 255.

### 5.2.4. Caracterización interna del driver con tratamiento de errores

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 5.12. 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 Retries* (*TR*) veces hasta que no haya error. La duración corresponderá al tiempo de *timeout* más el peor caso de *Token Retransmission 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.




$$T = \text{caso sin error} + \text{TR} \cdot (T + TRO)$$

**Figura 5.12: Transacción en caso de error.**

De la misma manera, las operación de envío de información debe de estar seguida de un *timeout* y de un *Check* y *Handle.* La duración corresponderá al tiempo de *timeout* más el peor caso de Packet *Retransmission Operation*, que se ha dividido en la figura en dos partes como el caso anterior, todo ello *Packet Transmission Retries (PR)* veces:

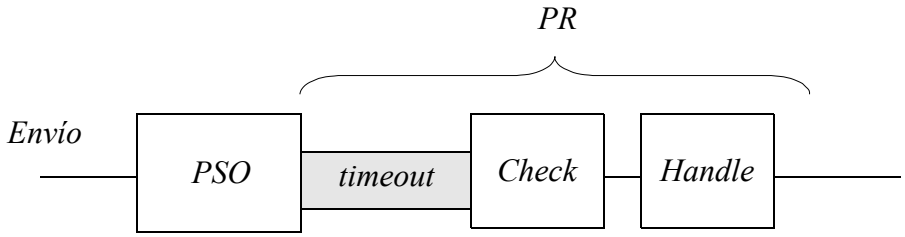



**Figura 5.13: Caso error en paquete de información.**

# 6. Métricas y modelo de RT-EP

## 6.1. 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 de la configuración del anillo lógico a través del modulo *lnk_config.h*. Lo único que hace es leer la configuración 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 y FIFO.
- **test_protocol**: Dependiendo de la opción -D que le pasemos a la hora de compilarlo (STATION_MASTER o STATION_SLAVE) compilará un ejecutable para la estación *master* o para la *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 (*slave*). 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 compilado con la opción STATION_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 el que esté con STATION_SLAVE las otra estación. Este programa es un ejemplo muy simple de la utilización del protocolo. Sirve para realizar comprobaciones y depuración.

- **test_ada_binding_master** y **test_ada_binding_slave**: Es el equivalente Ada de *test_protocol* utilizando la instanciación de las funciones genéricas del protocolo. La estación *master* manda un número que es recibido por la *slave* que contesta con otro numero todo ello en un lazo infinito. Este archivo es utilizado para depuración aunque es un ejemplo válido de la utilización de las unidades genéricas del protocolo en Ada.
- **test_rt_ep_streams** y **test_rt_ep_streams_slave**: Constituye una prueba análoga a *test_ada_binding_** sólo que utilizando la interfaz de streams del protocolo.

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, hay que destacar que se realizaron sobre una arquitectura de dos estaciones con procesadores *Pentium III* a 750 MHz corriendo MaRTE OS y conectadas a través de una red de 100 Mbps. Ahora sin más pasaremos a comentar el método y los tiempos medidos en los siguientes apartados.

## 6.2. Medida 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 creó el siguiente programa de prueba:

- **test_multi_thread:** Dependiendo del flag de compilación se obtienen distintos resultados:
  - **STATION_MASTER**: Consiste en cinco *threads* periódicos que se ejecutan con distinto periodo y mandan mensajes fijos a distintos canales de una estación. Debe ser ejecutado en la primera estación *token_master*. Se deberá especificar, en el código fuente, la dirección destino de los paquetes.
  - **STATION_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. Se deberá especificar la dirección destino en el fuente.

  Se pueden obtener ambos ejecutables de manera automática invocando *make test_multi.*

Este módulo 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. Como resultado nos muestra las medidas del peor caso, medio y mejor en la salida estándar. Una vez lanzados convenientemente los ejecutables en las máquinas correspondientes se procede a la medición. Cuando se haya procesado el número de tramas correspondiente a MAX_CPU_POINTS se mostrarán los tiempos medidos por la salida estándar. Por último solo indicar que se compilo tanto MaRTE Os como el protocolo con optimización para la arquitectura (i686) y con un *delay* de protocolo de 100 µs. Como resultado de las medidas obtenidas sobre la plataforma comentada anteriormente mostramos la figura 6.1.

| Estados | Peor caso (µs) | Mejor caso (µs) | Caso medio (µs) |
|---|---|---|---|
| *Idle State (+ Error_Check)* | 12.757 | 3.703 | 3.891 |
| *Send_Initial_Token* | 10.966 | 7.484 | 7.747 |
| *Check_Token* | 10.289 | 2.286 | 2.398 |
| *Send_Permission* | 6.61 | 5.830 | 6.032 |
| *Send_Token* | 20.605 | 9.729 | 10.395 |
| *Send_Info* | 19.813 | 14.184 | 15.486 |
| *Recv_Info* | 22.821 | 5.879 | 7.146 |

**Figura 6.1: 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.

Un overhead adicional que se produce en la estación, que aunque no pertenece directamente al protocolo si está asociado a éste, corresponde a la rutina de atención a la interrupción que se produce cuando se recibe una trama de la red. Esta rutina produce un *overhead* en la estación cada vez que se recibe una trama.

| Estado | Peor caso (µs) | Mejor caso (µs) | Caso medio (µs) |
|---|---|---|---|
| *ISR* | 6.48 | 2.50 | 3.74 |

**Figura 6.2: Overhead debido a la interrupción**

Con estos resultados se pueden calcular con exactitud los parámetros del modelo MAST del protocolo visto en el capitulo 5, “Modelado MAST”.

## 6.3. Medida 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 nuestro programa prueba creado con *make test_multi* compilado con el modo TOKEN_CHECKING_PRIORITY_TIMES del protocolo activado. Experimentalmente se ha obtenido el valor expuesto en la figura 6.3.

| Operación | Peor caso (μs) | Mejor caso (μs) | Caso medio (μs) |
|---|---|---|---|
| *Tiempo del paso de testigo* | 247.838 | 240.995 | 244.318 |

**Figura 6.3: Estimación del Packet Overhead**

Si aplicamos la fórmula obtenida en el modelo MAST del protocolo, sin fallos en la red, con los tiempos obtenidos del *overhead* de CPU nos saldría un *Packet Overhead* de unos **329.486 μseg**. La razón por la que de manera analítica nos aparece un valor más elevado que midiendo es debido a que en las mediciones no tenemos incluido el efecto del *transmit_token* ni el de transmisión de la cabecera de la trama de información que si están presentes en los cálculos formales.

Si añadimos esos dos factores a la medición realizada, es decir, el valor calculado formalmente de transmitir un *transmit_token* más lo que se tarda en transmitir la cabecera del paquete de información, tenemos que el overhead asociado al paquete seria de 305.72 μseg que ya es más cercano al calculado formalmente.

Es necesario destacar que el valor medido siempre será menor que el calculado ya que el valor calculado corresponde al peor caso de los overheads asociados a la transmisión de un paquete y en su calculo se tienen en cuenta el peor caso de todas las operaciones y parámetros que intervienen en el overhead, por lo tanto es muy dificil que en las mediciones nos encontremos con ese caso.

## 6.4. Régimen binario de RT-EP

Un parámetro bastante orientativo en las redes de comunicaciones es el régimen binario entendido como la cantidad de bits por segundo que se transmiten en un medio determinado. En nuestro caso, el protocolo trabaja sobre una red Ethernet con un régimen binario de 100 Mbps pero debido a las características de arbitrio sobre el medio que posee RT-EP este régimen binario “nativo” se ve degradado.

Para poder ofrecer un régimen binario efectivo, es decir, de información de aplicación, de RT-EP sobre una red Ethernet es necesario tener en cuenta el overhead que supone el protocolo sobre la red. Para caracterizar el régimen binario y como simplificación suponemos que es el régimen binario para un mensaje de la más alta prioridad, es decir, aquel que después del arbitrio le correspondería transmitir. También es necesario comentar que este régimen binario correspondería al "peor caso", es decir, es una cota inferior de la velocidad de transmisión.

Los datos numéricos que se ofrecen corresponden a una plataforma con dos estaciones (Pentium III a 750MHz), una red Ethernet de 100 Mbps y un delay ajustado a 80 μseg.

Es necesario separar un caso bastante común de utilización de la red que ofrece un mejor régimen binario que es el caso productor-consumidor en la red, este caso es especial ya que dado las características del protocolo no se produciría bloqueo en la transmisión de los mensajes. También este aplicable a los casos de transmisión y recepción sincronizada.

Por lo tanto el régimen binario nos queda:

- **Tx/Rx sincronizada**: En este caso no existe bloqueo por lo que el único factor que influye en el régimen binario es el overhead asociado a transmitir un mensaje:

$$R_{bT} = \frac{1492 \cdot 8}{PacketOverhead + MaxPTT}$$

  En nuestro caso, el régimen binario efectivo al transmitir información de aplicación en el caso de una transmisión/recepción sincronizada, libre de errores, es de **26.554 Mbps**.
- **Caso general**: En este caso, el de una transmisión en la red de forma totalmente asíncrona, nos encontraremos con casos de bloqueo que infuyen en el régimen binario. El máximo bloqueo para nuestro demostrador es de 879.794 μseg, este valor está calculado formalmente:

$$R_{bT} = \frac{1492 \cdot 8}{MaxBloqueo + PacketOverhead + MaxPTT}$$

  En nuestro caso, el régimen binario en caso de utilización normal de la red es de **8.979 Mbps.**

Como se puede observar y a pesar de que el *throughput* de la Ethernet nativa se ve degradada tenemos una red de comunicaciones de tiempo real muy rápida.

## 6.5. Utilización de RT-EP

Debido a que el protocolo es una modificación *software* del acceso al medio, resulta interesante averiguar cuál es la sobrecarga que añade el protocolo al procesador sobre el que se está ejecutando. Para el cálculo de la utilización, y como vimos en el apartado

4.3.8, "Configuración del protocolo", tenemos el modo de depuración CPU_OVERHEAD_ABS que se encarga de averiguar la ejecución del protocolo sobre un tiempo absoluto de ejecución. Este modo reporta el tiempo de ejecución total, el tiempo de ejecución del protocolo y la utilización del procesador en tanto por ciento. Como vimos, este modo actúa conjuntamente con el modo de medición de tiempos CPU_OVERHEAD_PROTOCOL_TIMES.

Un parámetro del protocolo introducido para controlar la utilización es el *delay* configurable en la transmisión del testigo. La razón por la que nos interesa controlar la utilización es debido a que no todas las aplicaciones tienen los mismos requerimientos de red, algunas pueden hacer un uso intensivo de la red y, por lo tanto, necesitan una pronta respuesta de ésta. Esta necesidad se traduce en una mayor utilización por parte del protocolo, sin embargo, otras pueden no hacer un uso tan intensivo y por lo tanto pueden ajustar la utilización del protocolo de acuerdo a sus necesidades.

| Delay (μs) | Utilización |
|---|---|
| 0 | 12 % |
| 1 | 12 % |
| 10 | 12 % |
| 30 | 7.8 % |
| 50 | 6.0 % |
| 100 | 3.8 % |
| 1.000 | 0.5 % |

**Figura 6.4: Utilización de RT-EP**

En la figura 6.4 se muestra como varía la utilización del protocolo en función del delay introducido. Como programa de prueba para las mediciones se ha empleado *test_multi_thread* que implementa cinco threads periódicos mandando datos, a distintas prioridades, a través de la red. Y el tiempo de ejecución ha sido el correspondiente a transmitir 50.000 tramas, ya sean testigos o información, a través de la red por parte del protocolo, es decir, asignando a MAX_CPU_POINTS el valor de 50.000.

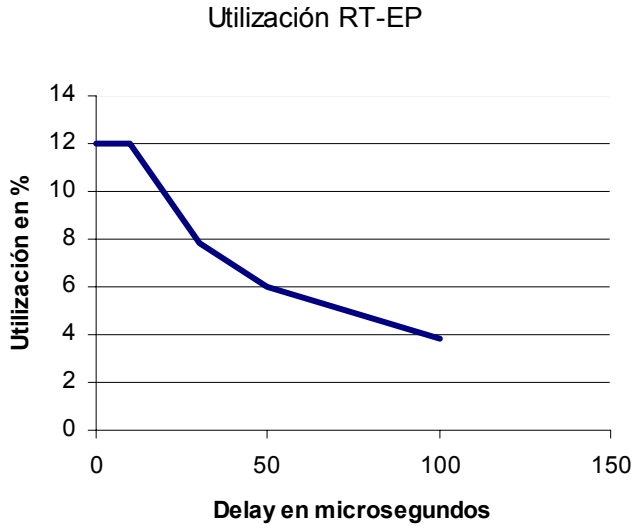



**Figura 6.5: Gráfica utilización de RT-EP.**

Como se puede observar de la figura 6.4, la franja de utilización óptima del protocolo es con un *delay* entre 30 y 100 µs. En esta franja se obtiene una utilización baja (3.8 % - 7.8 %) y una respuesta bastante rápida. Para una mejor visión de la influencia del *delay* en la utilización del protocolo se ha extraído una gráfica de la figura 6.4 que se expone en la figura 6.5.

# 6.6. RT-EP vs CAN bus

Un análisis interesante es ver como se desenvuelve RT-EP comparando sus números con los de un bus de Tiempo Real para situarnos es el contexto de las comunicaciones de Tiempo Real.

## 6.6.1. CAN bus.

CAN [4] es un bus de campo propuesto por Bosch para la fabricación de automoción y para el control de motores, es un sistema basado en bus serie adecuado para dispositivos de red "inteligentes" además de para sensores y actuadores dentro de un sistema.

Sus principales características son:

- 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.
- Es un bus de comunicaciones de alta integridad para aplicaciones de Tiempo Real.
- Posee un régimen binario de hasta **1 Mbps**. Este valor se aplica a redes de hasta **40 metros**, para distancias más grandes el régimen binario debe ser disminuido; en distancias de hasta 500 metros se puede conseguir una velocidad de 125 Kbps y para transmisiones hasta 1 km el régimen binario máximo permitido es de 50 Kbps. Una comparativa resumen entre la velocidad de CAN y la máxima distancia alcanzable la tenemos en la figura 6.6.

| Velocidad | Tiempo de bit | Longitud máxima |
|---|---|---|
| 1 Mbps | 1 µseg | 30 m |
| 800 Kbps | 1.25 µseg | 50 m |
| 500 Kbps | 2 µseg | 100 m |
| 250 Kbps | 4 µseg | 250 m |
| 125 Kbps | 8 µseg | 500 m |
| 50 Kbps | 20 µseg | 1000 m |
| 20 Kbps | 50 µseg | 2500 m |
| 10 Kbps | 100 µseg | 5000 m |

**Figura 6.6: Régimen Binario / Distancia en CAN**

- Es robusto frente a errores. La comunicación en CAN continuará a pesar de que:

    - Cualquiera de los dos cables se rompa.
    - Cualquier cable se cortocircuite a masa.
    - Cualquier cable se ponga a tensión.
- Originalmente desarrollado por Bosh para su utilización en automoción, actualmete se utiliza para multitud de automatización industrial y aplicaciones de control.
- Es un estándar internacional: ISO 11898.
- Se distinguen entre dos variantes de CAN, el definido en CAN 2.A o "CAN Standard" y el definido en CAN 2.B o "CAN Extendido". Los formatos de trama son análogos diferenciándose básicamente en el número de bits que se utiliza para el identificador de mensaje: 11 bits (2032 identificadores) diferentes en CAN Standard y 29 bits (536.870.912 identificadores) en CAN Extendido.
- Las tramas son de longitud reducida, presentado 8 bytes para los datos de la aplicación y 47 bits de overhead, de los cuales 34 bits se codifican con relleno de bits (si el transmisor encuentra cinco bits consecutivos de igual valor, inserta un bit complementario [4]). Por lo tanto, y segun lo expuesto en [40] y [41], tenemos un tamaño para la trama más larga de **130 bits** en el caso de CAN estándar y **154 bits** en el caso de can extendido.

### 6.6.2. Condiciones de la comparativa

Es necesario definir un entorno que nos sirva para realizar una comparativa más o menos efectiva de ambos protocolos. Para ello se define una serie de condiciones de transmisión haciendo hincapié sobre el protocolo en el que inciden estas condiciones y la justificación de ese parámetro:

- **Número de estaciones** (N): Se han realizado dos grupos:
    - Baja concentración de estaciones: 2 (demostrador), 5, 10 y 20 estaciones.
    - Alta concentración de estaciones: 30, 50, 75 estaciones.

  Este parámetro hace hincapié en la degradación que sufre RT-EP con el número de estaciones que intervienen en la comunicación. En CAN este valor, en principio, no es relevante.
- **Procesadores Pentium III a 750 MHz por cada estación**: Esta condición es un requisito imprescindible ya que las métricas del protocolo, que influyen en los tiempos de transmisión, se han realizado sobre esas plataformas. En el caso de CAN se realizaron las medidas de procesado en una máquina **Pentium III a 550 MHz**. Para la comparación se ha decidido normalizar los valores tomados en CAN por un factor que sea el cociente de los *ticks* por segundo de las dos máquinas. Así conseguimos, de una manera optimista, igualar la capacidad de procesado de ambas máquinas. Aunque cabe destacar que esa capacidad de procesado no depende únicamente de la velocidad del procesador. En la figura 6.7 se expone los *ticks* por segundos de ambas estaciones y el coeficiente aplicado. Los tiempos resultantes de las estaciónes serán los medidos divididos por el coeficiente.
- **Distancia de la comunicación**: Elegimos distancias que vayan acorde con el número de estaciones que se estén analizando e intentar que CAN transmita a

| Estacion | Ticks / seg | Coeficiente |
|---|---|---|
| Pentium III 750 MHz | 751743208 | 1 |
| Pentium III 550 MHz | 552379701 | 1.361 |

**Figura 6.7: Ticks por segundo y coeficiente normalizador**

la máxima velocidad ya que la distancia influye negativamente en el régimen binario de CAN. Para ello se definen varias distancias:

- **40 metros**: Distancia que corresponde con el mayor régimen binario de CAN (1Mbps). Esta distancia la utilizaremos para el caso de 2, 5, 10, 20 y 30 estaciones
- **50 metros**: El régimen binario en CAN a esta distancia es de 800 Kbps, suponemos que estamos en este caso cuando disponemos de 50 estaciones.
- **100 metros**: En este caso el régimen binario baja a 500 Kbps. Consideramos que están operando 75 estaciones.

RT-EP al utilizar Ethernet como red de comunicaciones su régimen binario no se ve degradado con la distancia. No incorporamos más distancia en la comparativa debido a que a más distancia CAN se degrada mucho y ya no es una solución a tener en cuenta.

- **Tamaño mensajes** (T): Se consideran que se transmiten distintos tipos de mensajes de longitudes distintas:
  - **8 bytes**: Mensajes de tamaño máximo para CAN.
  - **32 bytes**: Mensajes pequeños, CAN necesita realizar troceado de paquetes.

    En ambos caso, en RT-EP, el tamaño de la trama que se transmite es de 46 bytes debido al *zero-padding* que realiza Ethernet.
  - **64, 128, 256 bytes**: Tamaño medio de mensajes
  - **512 y 1024 bytes**: Mensaje de información grande.
- **Siempre el peor caso**: Es conveniente aclarar que en la comparativa siempre se ha considerado el peor caso, ya sea en los tiempos de protocolo, en la trama de CAN que se considera la máxima posible 130 bits considerando la codificación con relleno y con 8 bytes de información y en los tiempos de procesado.
- **Red libre de errores**: Se considera la red libre de errores y no se consideran retrasos de sincronismo en CAN, se supone un transmisor ideal.
- **Parámetros reales**: Con esta comparativa se pretende ofrecer una visión real del comportamiento de ambos protocolos. Para ello, y en el caso de RT-EP, queremos que se comporte de una manera real, por ello utilizaremos un *delay* en la configuración del protocolo de **30 μseg** que produce una utilización en los procesadores de la comparativa de un 7.8 %.
- **Utilización CAN 2.0A**: Utilizaremos el llamado CAN estándar ya que es más eficiente desde el punto de vista de régimen binario, posee menos overhead asociado al protocolo.
- **Utilizaremos Ethernet de 100 Mbps y 1000 Mbps**: Se trata de una comparativa teniendo en cuenta la tecnología actual, así que se ha considerado

los regímenes binarios implementados en las estaciones actuales que es Fast Ethernet (100 Mbps) y las velocidades que se empiezan a implementar que es la Gigabyte Ethenet (1000 Mbps).

- **Tiempo de procesado en CAN**: Para la medición, tanto en RT-EP como en CAN, se ha tenido en cuenta el tiempo de procesado en la estación necesario para transmitir el paquete de información, el tiempo de transmisión en el medio y el tiempo de procesado del paquete en la estación destinataria. En la figura 6.8 se muestran los tiempos correspondientes al procesado de transmisión y recepción en CAN. Entre parentesis se muestran los valores aplicados en los cálculos que son los resultantes de normalizar por el coeficiente.

| T. de procesado | Peor Caso (µs) | Mejor Caso (µs) | Caso Medio (µs) |
|---|---|---|---|
| Transmisión | 16.762 (12.315) | 15.923 (11.699) | 16.05 (11.793) |
| Recepción | 31.84 (23.394) | 31.00 (22.777) | 31.47 (23.122) |

**Figura 6.8: Tiempos de procesado de la información**

- **Troceado de paquetes**: Debido al tamaño de los mensajes de la comparativa, se hace necesario contabilizar el tiempo de procesado en CAN, cuyo campo de información es de 8 bytes, necesario para realizar ese des-ensamblado de paquetes en la transmisión y ensamblado en la recepción. La información necesaria para trocear y reconstruir los paquetes de tamaño superior a 8 bytes consume 1 byte del campo de información en la implementación actual de CAN, por lo tanto, a efectos de transmisión sólo se pueden transmitir 7 bytes. Es necesario resaltar que esta medida depende bastante en la implementación del troceo, por ello se ha elegido un mejor caso que sólo contempla la construcción del paquete fragmento, es decir, la cabecera y la copia de la parte del mensaje necesario. En la figura se muestra el coste de des-ensamblado y ensamblado de un paquete.

| Operación | Peor Caso (µs) | Mejor Caso (µs) | Caso Medio (µs) |
|---|---|---|---|
| Ensamblado | 7.543 (5.542) | 6.704 (4.925) | 6.732 (4.946) |
| Des-ensamblado | 5.867 (4.31) | 5.866 (4.31) | 5.866 (4.31) |

**Figura 6.9: Coste del troceo en paquetes**

## 6.6.3. Comparativa RT-EP vs CAN bus

Primeramente extraeremos las formulas generales del tiempo necesario para transmitir los mensajes de la comparativa.

- **CAN bus**: El tiempo total de transmisión es

$$\left\lceil \frac{T}{7} \right\rceil \cdot T_{Troceado} + \left\lceil \frac{T}{7} \right\rceil \cdot T_{PROCESADO-TX} + \left( \left\lceil \frac{T}{7} \right\rceil \cdot 130/R_b \right) + \left\lceil \frac{T}{7} \right\rceil \cdot T_{PROCESADO-RX} + \left\lceil \frac{T}{7} \right\rceil \cdot T_{Ensamblado} =$$
$$\left\lceil \frac{T}{7} \right\rceil \cdot \left( 4,31_{\mu s} + 12,315_{\mu s} + \frac{130}{R_b} + 23,394_{\mu s} + 5,542_{\mu s} \right)$$

No poseemos ninguna cifra para el tiempo de procesado del paquete en CAN por lo que utilizaremos el tiempo medido en el tratamiento de la rutina de interrupción del protocolo RT-EP de 6.48 µseg dicha rutina recibe un paquete en la red y lo almacena en memoria, en el caso de CAN se debería recibir el paquete de la red, ensamblarlo y almacenarlo en memoria. Por lo tanto el

tiempo de la rutina de interrupción sería una cota inferior del tiempo de procesado en CAN.

- **RT-EP**: El tiempo total de transmisión sería el peor caso del overhead de paquete más el tiempo que se tarda en procesar para el envío del mensaje más lo que se tarda en transmitir el mensaje, mas el tiempo que se tarda en procesar ese paquete de información. De la operación *Packet Receive Operation* se a sustraido el tiempo del estado *Send Initial Token* ya que ese tiempo está incluido en el overhead de paquete. La ecuación queda por tanto:

$$\left((N+1)(MinPTT+ISR+TCO+TMO)+N\cdot TD+\frac{34\cdot 8}{R_b}\right)+ISR+PSO+8\cdot\frac{T}{R_b}+ISR+PRxO=$$
$$(N+1)\left(\frac{72\cdot 8}{R_b}+\dot{6},48_{\mu s}+23,046_{\mu s}+20,605_{\mu s}\right)+N\cdot 30_{\mu s}+\frac{34\cdot 8}{R_b}+6,48_{\mu s}+$$
$$32,57_{\mu s}+\frac{8\cdot T}{R_b}+6,48_{\mu s}+35,578_{\mu s}$$

Una vez determinadas las formulas generales pasaremos a mostrar las comparativas (el tiempo en *milisegundos*):

- **Resultados dos estaciones**, con la velocidad de transmisión de CAN de 1 Mbps en una distancia menor de 40 metros. Los resultados se pueden observar en la figura 6.10:

| | **2 estaciones** | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.351 | 0.877 | 1.755 | 3.333 | 6.49 | 12.99 | 25.80 |
| RT-EP - 100 Mbps | 0.312 | 0.314 | 0.316 | 0.321 | 0.331 | 0.352 | 0.393 |
| RT-EP - 1000 Mbps | 0.293 | 0.293 | 0.294 | 0.294 | 0.295 | 0.297 | 0.301 |

**Figura 6.10: CAN vs RT-EP - 2 estaciones**

- **Resultados cinco estaciones**. Los resultados están expuestos en la figura 6.11, los tiempos para CAN son los mismos que en el caso de 2 estaciones ya que seguimos en una longitud de la red por debajo de 40 metros:

| | **5 estaciones** | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.351 | 0.877 | 1.755 | 3.333 | 6.49 | 12.99 | 25.80 |
| RT-EP - 100 Mbps | 0.569 | 0.571 | 0.574 | 0.579 | 0.589 | 0.610 | 0.651 |
| RT-EP - 1000 Mbps | 0.535 | 0.535 | 0.536 | 0.536 | 0.537 | 0.539 | 0.543 |

**Figura 6.11: CAN vs RT-EP - 5 estaciones**

- **Resultados diez estaciones**. De la misma manera que en el caso anterior los datos para CAN no varían y los del protocolo se ven ligeramente incrementados al aumentar el número de estaciones que intervienen en la comunicación. Los resultados están expuestos en la figura 6.12:
- **Resultados veinte estaciones**. En ese caso la velocidad de CAN sigue sin cambiar ya que seguimos por debajo de los 40 metros. Pero al existir un mayor número de estaciones, RT-EP, presenta unos tiempos de transmisión mayores como se puede observar en la figura 6.13:

| | 10 estaciones | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.351 | 0.877 | 1.755 | 3.333 | 6.49 | 12.99 | 25.80 |
| RT-EP - 100 Mbps | 0.999 | 1.001 | 1.003 | 1.008 | 1.019 | 1.039 | 1.080 |
| RT-EP - 1000 Mbps | 0.939 | 0.939 | 0.939 | 0.940 | 0.941 | 0.943 | 0.947 |

**Figura 6.12: CAN vs RT-EP - 10 estaciones**

| | 20 estaciones | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.351 | 0.877 | 1.755 | 3.333 | 6.49 | 12.99 | 25.80 |
| RT-EP - 100 Mbps | 1.858 | 1.860 | 1.862 | 1.867 | 1.878 | 1.898 | 1.939 |
| RT-EP - 1000 Mbps | 1.746 | 1.746 | 1.746 | 1.747 | 1.748 | 1.750 | 1.754 |

**Figura 6.13: CAN vs RT-EP - 20 estaciones**

- **Resultados treinta estaciones**. En ese caso la velocidad de CAN sigue sin cambiar ya que seguimos en una longitud de la red inferior a 40 metros. Los números del protocolo aumentan debido al aumento del número de estaciones. Los resultados se encuentran expuestos en la figura 6.14:

| | 30 estaciones | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.351 | 0.877 | 1.755 | 3.333 | 6.49 | 12.99 | 25.80 |
| RT-EP - 100 Mbps | 2.717 | 2.719 | 2.721 | 2.726 | 2.736 | 2.757 | 2.798 |
| RT-EP - 1000 Mbps | 2.553 | 2.553 | 2.553 | 2.554 | 2.555 | 2.557 | 2.561 |

**Figura 6.14: CAN vs RT-EP - 30 estaciones**

- **Resultados cincuenta estaciones**. En este caso la longitud de la red supera los 40 metros pero es inferior a 50 por lo que la velocidad binaria de CAN es de 800 Kbps. Además como aumentan el número de estaciones también lo hacen los tiempos asociados a RT-EP como puede verse en la figura 6.15:

| | 50 estaciones | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.416 | 1.04 | 2.08 | 3.953 | 7.698 | 15.396 | 30.584 |
| RT-EP - 100 Mbps | 4.434 | 4.436 | 4.439 | 4.444 | 4.454 | 4.475 | 4.516 |
| RT-EP - 1000 Mbps | 4.167 | 4.167 | 4.167 | 4.168 | 4.169 | 4.171 | 4.175 |

**Figura 6.15: CAN vs RT-EP - 50 estaciones**

- **Resultados setenta y cinco estaciones**. En este caso también cambia el régimen binario de CAN (500 Kbps) ya que la distancia es de 100 metros. Al aumentar el número de estaciones, los tiempos del protocolo también aumentan como lo refleja la figura 6.16:

Con esta comparación superficial, ya que son protocolos muy dispares, se puede comprobar que RT-EP ofrece mejores tiempos de transmisión para un número bajo o medio de estaciones y que éste se degrada a medida que aumentan las estaciones. Por el

| | 75 estaciones | | | | | | |
|---|---|---|---|---|---|---|---|
| Información (bytes) | **8** | **32** | **64** | **128** | **256** | **512** | **1024** |
| CAN bus | 0.611 | 1.527 | 3.055 | 5.805 | 11.305 | 22.611 | 44.917 |
| RT-EP - 100 Mbps | 6.582 | 6.584 | 6.586 | 6.591 | 6.602 | 6.622 | 6.663 |
| RT-EP - 1000 Mbps | 6.185 | 6.185 | 6.185 | 6.186 | 6.187 | 6.189 | 6.193 |

**Figura 6.16: CAN vs RT-EP - 75 estaciones**

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. Aunque para mensajes de tamaño medio o alto, RT-EP es superior, incluso con un número elevado de estaciones, esto es debido a que el troceo de mensajes incide seriamente en el rendimiento de CAN.

También cabe resaltar que, si se observan los valores con detenimiento, el hecho de utilizar una Ethernet con una velocidad binaria de 1000 Mbps, en contra de lo que cabría esperar a priori, no mejora significativamente los tiempos de transmisión. Esto es debido a que la parte que penaliza la transmisión cuando la velocidad del medio es mayor de 100 Mbps es la parte de procesado de los paquetes. Por tanto si queremos que los tiempos de RT-EP disminuyan deberemos utilizar estaciones más rápidas que bajen los tiempos de procesado.

## 6.7. Modelo de RT-EP

Como se vio en el apartado 5.2, “Modelo MAST de RT-EP”, se ha extraído un modelo de RT-EP competo que procedemos a su caracterización para realizar su análisis. Los tiempos correspondientes a las operaciones del protocolo deriban del apartado 6.2, “Medida de los Overhead de CPU”. Las expresiones “MAST” se complementan con las expresadas en el apartado 7.2, “Modelo MAST del BTM”, y quedan como sigue:

- **Processing Resource**: Utilizamos un *Packet Based Network* que representará a la red Ethernet con la adaptación de acceso al medio que supone RT-EP, proporcionando así una red de tiempo real. Esta directiva admite los siguientes atributos:
  - **Type**: Representa al tipo de recurso de procesado que estamos utilizando. En nuestro caso se trata de una *Packet_Based_Network* ya que se trata de una red.
  - **Name**: Identifica a la red que estamos definiendo.
  - **Throughput**: Corresponde con el ancho de banda normalizado en bits por unidad de tiempo. En nuestro demostrador hemos estado utilizando una red Ethernet con un régimen binario de 100 Mbps.
  - **Transmission**: Informa acerca de la capacidad del medio, *simplex*, *half-dupex* o *full-duplex*. A pesar de que la configuración a nivel físico sea de *full-duplex*, la utilización efectiva de la red debido a la lógica del protocolo se realiza de manera *half-duplex*.
  - **Max_Blocking**: Representa el bloqueo máximo que se puede experimentar antes de que se pueda transmitir un mensaje de alta prioridad. Este valor se puede determinar analíticamente como se expuso en el apartado 5.2.3, “Caracterización MAST de RT-EP con tratamiento

de errores". Teniendo en cuenta que sólo admitimos una retransmisión en caso de error, como se expondrá más adelante, y que el demostrador consiste en dos estaciones, obtenemos una valor para el máximo bloqueo de 879.794 μseg. Hay que tener en cuenta que en este cálculo se ha considerado un peor caso de una retransmisión de testigo y una retransmisión de paquete de información lo que supone una situación realmente pesimista ya que la probabilidad de error en bit de Ethernet es muy baja.

- **Max_Packet_Size** / **Min_Packet_Size**: Describen la cantidad de datos contenidos en el paquete de información, excluyendo cualquier información de protocolo. El tamaño mínimo corresponde a 72 bytes, este valor es debido a una restricción de la trama Ethernet, no es posible mandar una trama de tamaño menor a 46 bytes en el campo de información. El máximo corresponde a 1500 bytes.
- **Max_Packet_Transmission_Time** / **Min_Packet_Transmission_Time**: Representan otra posibilidad de expresar el tamaño máxima y mínimo del paquete pero expresado en tiempo en vez de en tamaño (bits):
  - *Max Packet Tranmission Time*:

$$MaxPTT = \frac{1492 \cdot 8}{Rb}$$

  - *Min Packet transmisión Time*:

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

- **List_of_Drivers**: Es una lista que referencia los drivers de red que contienen el overhead del procesador asociado con la transmisión de mensajes a través de la red.

Este *processing resource* quedara de la forma:

```
Processing_resource (
     Type                        => Packet_Based_Network,
     Name                        => RT_EP_network,
     Throughput                  => 100000000,
     Transmission                => Half_Duplex,
     Max_Blocking                => 879.794e-6,
     Max_Packet_Size             => 11936, -- 1492*8,
     Min_Packet_Size             => 576, --72*8
     List_of_Drivers             =>
                          (RTEP_Packet_Driver_Btmcont),
                           RTEP_Packet_Driver_Btmmmi));
```

Tenemos un driver por cada nodo que existe en la red, en nuestro caso dos drivers. Los atributos de los drivers deberían ir desglosados en la directiva *List of Drivers* pero para más claridad hemos decidido poner dos etiquetas y desglosar los drivers en un apartado. Para obtener la sintaxis MAST válida basta con sustituir la etiqueta por la agregación de todos los elemento MAST descritos en cada apartado.

- **RT-EP_Packet_Driver_Btmcont**: Procederemos a caracterizar y describir cada uno de los atributos de los que consta. La descripción sólo la realizaremos

para este *driver*, para el otro nodo pondremos sólamente el código MAST debido a que se trata del mismo driver. Para una descripción mas detallada de la sintaxis MAST véase [10]:

- **Type**: Identifica al tipo de driver que estamos utilizando, en nuestro caso *RTEP_Packet_Driver*.
- **Packet Server**: Corresponde con el *Scheduling Server* que está ejecutando el driver. Un servidor de planificación representa entidades planificables dentro de un recurso de procesado. En nuestro caso corresponde con el *thread* de comunicaciones que se encarga del arbitrio de acceso al medio. Tiene como atributos:
  - **Type**: Indica la clase a la que pertenece, actualmente solo existe una que es *Regular*.
  - **Name**: Es el identificador del servidor de planificación.
  - **Server_Sched_Parameters**: Representan distintos parámetros que necesita el planificador para aplicar la política de planificación apropiada.
  - **Scheduler**: Indica el planificador al que este objeto está asociado.

```
(Type                       => Regular,
Name                        => Communication_Thread,
Server_Sched_Parameters     => (
            Type            => Fixed_Priority_Policy,
            The_Priority    => 29,
            Preassigned     => Yes),
Scheduler                   => btmcont),
```

- **Packet Interrupt Server**: Corresponde con el servidor de planificación que está ejecutando la rutina de interrupción que atiende a la trama recibida por el medio.

```
(Type                       => Regular,
Name                        => Interrupt_handler,
Server_Sched_Parameters     => (
            Type            => Interrupt_FP_Policy,
            The_Priority    => 32,
            Preassigned     => Yes),
Scheduler                   => btmcont),
```

- **Packet Send Operation**: Corresponde con la operación que se ejecuta cada vez que se manda un paquete. Corresponde con los estados *Idle* y *Send_Info*.

```
(Type                       => Simple,
Name                        => Packet_Send,
Worst_Case_Execution_Time  => 32.39E-6,
Best_Case_Execution_Time   => 17.79E-6,
Avg_Case_Execution_Time    => 19.274E-6),
```

- **Packet Receive Operation**: Representa la operación que se ejecuta cada vez que se recibe un mensaje. Corresponde a los estados *Idle*, *Recv_Info* y *Send_Initial_Token*.

```
(Type                      => Simple,
Name                       => Packet_Receive,
Worst_Case_Execution_Time => 46.209E-6,
Best_Case_Execution_Time  => 16.975E-6,
Avg_Case_Execution_Time   => 18.684E-6),
```

- **Message_Partitioning**: Indica si el *driver* es capaz de particionar mensajes de gran longitud y reconstruirlos en el destino. Los posibles valores son Yes o No. En nuestro caso tendrá un valor de No.
- **RTA_Overhead_Model**: Es un atributo exclusivo de la herramienta de análisis del tiempo de respuesta en el peor caso. En nuestro caso tendrá un valor de *Coupled*.
- **Packet ISR Operation**: Es la operación ejecutada por el servidor de interrupción cada vez que se recibe una trama de la red.

```
(Type                      => Simple,
Name                       => Packet_ISR,
Worst_Case_Execution_Time => 6.44E-6,
Best_Case_Execution_Time  => 2.48E-6,
Avg_Case_Execution_Time   => 3.72E-6),
```

- **Number of Stations**: Identifica al número de estaciones (procesadores) conectados en red que como hemos visto se tratan de **dos** procesadores: *btmcont* y *btmmmi*.
- **Token Manage Operation**: Corresponde con la operación que ejecuta el *Packet Server* para mandar un mensaje. Corresponde al peor caso de los estados *Send_Token* o *Send_Permission*.En nuestro caso corresponderá a *Send_Token* que es una estado más pesado que *Send_Permission*.

```
(Type                      => Simple,
Name                       => Token_Manage,
Worst_Case_Execution_Time => 20.496E-6,
Best_Case_Execution_Time  => 9.677E-6,
Avg_Case_Execution_Time   => 10.340E-6),
```

- **Token Check Operation**: Es la operación que se ejecuta al recibir y chequear un paquete testigo. Corresponde a los estados *Idle* y *Check_Token*.

```
(Type                      => Simple,
Name                       => Token_Check,
Worst_Case_Execution_Time => 22.924E-6,
Best_Case_Execution_Time  => 14.963E-6,
Avg_Case_Execution_Time   => 18.059E-6),
```

- **Token Delay**: Es el retardo configurado en el protocolo para reducir el overhead de CPU. Éste se configuró a un valor de 80 µs para que la utilización de la CPU y respuesta de la red fuese óptima. Véase 6.5, “Utilización de RT-EP”.
- **Packet Discard Operation**: Esta operación se ejecuta al descartar tramas dirigidas a otra estación. Es un subconjunto del estado *Idle* por lo tanto se puede acotar por éste.

```
(Type                      => Simple,
Name                       => Packet_Discard,
Worst_Case_Execution_Time => 12.689E-6,
Best_Case_Execution_Time  => 3.683E-6,
Avg_Case_Execution_Time   => 3.870E-6),
```

- **Token Transmission Retries**: Indicará el número máximo de fallos, y sus retransmisiones, que permitiremos en cada vuelta de testigo. El valor asignado es **1** ya que es extraordinariamente dificil que haya dos fallos en el mismo paquete en situaciones normales.
- **Packet Transmission Retries**: Corresponde al número de fallos que permitimos al transmitirá un paquete de información. Por cada posible fallo se produce una retransmisión. Su valor es de **1**.
- **Failure Timeout**: Es el tiempo de timeout configurado en el protocolo. Para su cálculo tenemos que tener en cuenta dos posibles casos que son los más costosos en cuanto a consumo de tiempo:
    - *Transmisión de información*: En este caso el tiempo que estará escuchando la estación para ver si hay error es:

$$T = TtxInfo + ISR + Idle + RecvInfo + SendInitialToken + TtxTestigo + ISR + Idle$$

    - *Transmisión de un testigo*: En este caso, aunque el tiempo de transmisión es bastante menor, al ser tramas más pequeñas, tenemos que reflejar el efecto negativo en la red del *delay* introducido al mandar testigos. Por lo tanto el tiempo que debe esperar una estación para ver si se ha producido un fallo con un testigo es:

$$T = TtxTestigo + ISR + Idle + TokenCheck + delay + SendToken + TtxTestigo + ISR + Idle$$

  En nuestro caso y dada la configuración actual del protocolo tenemos que el peor caso es cuando estamos esperando el error de un paquete de información, 200.1 μseg frente a 160.888 μseg que supone esperar si se ha transmitido correctamente un testigo. Si el delay tuviese un valor más elevado entonces la situación se invertiría ya que solo influye en el caso de mandar testigo. Por lo tanto elegimos un valor para el timeout de 250 μseg, una corrección de un 25% sobre el valor calculado para estar seguros de estar contemplando el peor caso.
- **Token Retransmission Operation**: Representa la operación que se ejecuta cada vez que se retransmite un testigo. Esta operación se encuentra acotada por los estados *Idle* y *Send_Token*.

```
(Type                      => Simple,
Name                       => Token_Retransmission,
Worst_Case_Execution_Time => 33.185E-6,
Best_Case_Execution_Time  => 13.361E-6,
Avg_Case_Execution_Time   => 14.210E-6),
```

- **Packet Retransmission Operation**: Corresponde con la operación que se ejecuta cada vez que se retransmite un paquete de información. Esta operación está acotada por los estados *Idle* y *Send_Info*.

```
     (Type                      => Simple,
     Name                       => Packet_Retransmission,
     Worst_Case_Execution_Time  => 32.39E-6,
     Best_Case_Execution_Time   => 17.093E-6,
     Avg_Case_Execution_Time    => 19.274E-6)),
```

- **RT-EP Packet Driver_Btmmmi**:

```
(Type            => RTEP_Packet_Driver,

Packet_Server    =>
     (Type                      => Regular,
     Name                       => Communication_Thread,
     Server_Sched_Parameters    => (
                Type            => Fixed_Priority_Policy,
                The_Priority    => 29,
                Preassigned     => Yes),
     Scheduler                  => btmmmi),

Packet_Interrupt_Server    =>
     (Type                      => Regular,
     Name                       => Interrupt_handler,
     Server_Sched_Parameters    => (
                Type            => Interrupt_FP_Policy,
                The_Priority    => 32,
                Preassigned     => Yes),
     Scheduler                  => btmmmi),

Packet_Send_Operation      =>
     (Type                      => Simple,
     Name                       => Packet_Send,
     Worst_Case_Execution_Time  => ,
     Worst_Case_Execution_Time  => 32.57E-6,
     Best_Case_Execution_Time   => 17.887E-6,
     Avg_Case_Execution_Time    => 19.377E-6),

Packet_Receive_Operation   =>
     (Type                      => Simple,
     Name                       => Packet_Receive,
     Worst_Case_Execution_Time  => 46.544E-6,
     Best_Case_Execution_Time   => 17.066E-6,
     Avg_Case_Execution_Time    => 18.784E-6),

Message_Partitioning       => No,

RTA_Overhead_Model         => Coupled,

Number_Of_Stations         => 2,

Token_Delay                => 80.0E-6,

Failure_Timeout            => 250.0E-6,

Token_Transmission_Retries=> 1,

Packet_Transmission_Retries=>1,

Packet_ISR_Operation       =>
     (Type                      => Simple,
     Name                       => Packet_ISR,
     Worst_Case_Execution_Time  => 6.48E-6,
     Best_Case_Execution_Time   => 2.50E-6,
     Avg_Case_Execution_Time    => 3.74E-6),
```

```
Token_Check_Operation     =>
     (Type                      => Simple,
     Name                       => Token_Check,
     Worst_Case_Execution_Time  => 23.046E-6,
     Best_Case_Execution_Time   => 15.043E-6,
     Avg_Case_Execution_Time    => 18.155E-6),

Token_Manage_Operation         =>
     (Type                      => Simple,
     Name                       => Token_Manage,
     Worst_Case_Execution_Time  => 20.605E-6,
     Best_Case_Execution_Time   => 9.729E-6,
     Avg_Case_Execution_Time    => 10.395E-6),

Packet_Discard_Operation       =>
     (Type                      => Simple,
     Name                       => Packet_Discard,
     Worst_Case_Execution_Time  => 12.757E-6,
     Best_Case_Execution_Time   => 3.703E-6,
     Avg_Case_Execution_Time    => 3.891E-6),

Token_Retransmission_Operation =>
     (Type                      => Simple,
     Name                       => Token_Retransmission,
     Worst_Case_Execution_Time  => 33.362E-6,
     Best_Case_Execution_Time   => 13.432E-6,
     Avg_Case_Execution_Time    => 14.286E-6),

Packet_Retransmission_Operation=>
     (Type                      => Simple,
     Name                       => Packet_Retransmission,
     Worst_Case_Execution_Time  => 32.57E-6,
     Best_Case_Execution_Time   => 17.184E-6,
     Avg_Case_Execution_Time    => 19.377E-6))));
```

- **Scheduler**: Representa los objetos que implementan la estrategia de planificación adecuada para gestionar la capacidad de procesado asignada. Esta directiva admite los siguientes atributos:
  - **Type**: Expresa el tipo de planificador. Puede ser primario (Primary_Scheduler) en el caso de referirse a un planificador asociado a un recurso de procesado o secundario (Secondary_Scheduler) en el caso de un planificador que solo sea capaz de aprovechar una parte de la capacidad total de procesado.
  - **Name**: Corresponde con el identificador del Scheduler.
  - **Host**: Indica el recurso de procesado (*Processing Resource*) al que esta asociado el planificador.
  - **Policy**: Representa la estrategia básica que está implementada en el planificador para distribuir la capacidad de procesado asignada a los servidores de planificación (*Scheduling Servers*).

```
Scheduler (
     Type                  => Primary_Scheduler,
     Name                  => RT_EP_Network_Sched,
     Host                  => RT_EP_Network,
     Policy                =>
            (Type                    => FP_Packet_Based,
            Packet_Worst_Overhead    => 329.486e-6,
            Max_Priority             => 255,
            Min_Priority             => 1));
```

La justificación del valor empleado en el atributo *Packet_Worst_Overhead* se encuentra descrito en el apartado 6.3, “Medida del Packet Overhead.”.

En este capítulo hemos demostrado que es posible obtener un modelo y una métrica precisa del protocolo para poder, junto con el modelo de la aplicación, analizar el sistema en su totalidad.

# 7. Demostrador de RT-EP

## 7.1. Implementación en un demostrador

Una vez desarrollado, implementado y caracterizado el protocolo es importante su utilización en un demostrador para contrastar su validez en comunicaciones de tiempo real estricto. Se ha elegido como validador nuestra implementación de un brazo telemanipulado.

El brazo telemanipulado (BTM) es un brazo articulado antropomórfico que consta de seis articulaciones independientes movidas por servomotores. Su base está enclavada a la pieza donde se desea trabajar o a la superficie que le sostiene. El brazo está formado por varios cuerpos rígidos conectados en serie por articulaciones de tipo rotacional. Uno de los extremos de la cadena, su base, está fijo mientras que el otro extremo está libre y equipado con una herramienta especialmente diseñada para el trabajo a realizar.

Desde el punto de vista mecánico, como puede apreciarse en la figura 7.1, el brazo se puede descomponer en tres partes:

- *Cadena básica*: consta de tres articulaciones: la primera es la *base*, que se enclava a la pieza donde se desea trabajar o a la superficie que sostiene al robot; la segunda es el *hombro* y la tercera el *codo*, y se emplean para posicionar en el espacio a la muñeca.
- *Muñeca*: posee tres movimientos: giro, elevación y rotación, que se emplean para orientar el elemento terminal en el espacio.
- *Herramienta*: es el elemento terminal del robot y como tal lo vamos a considerar parte de él. En nuestro caso se trata de una herramienta de propósito especial que puede cambiar en cada aplicación.


**Figura 7.1: BTM**

El brazo puede hacer tareas en modo teleoperado, puede aprender y repetir trayectorias, y puede operar en modo programado mediante el uso de una interfaz de programación de robots. Puede operar tanto colocado sobre el suelo, como en posición invertida colgado del techo. La figura 7.2 muestra una visión global de la arquitectura utilizada en el demostrador.

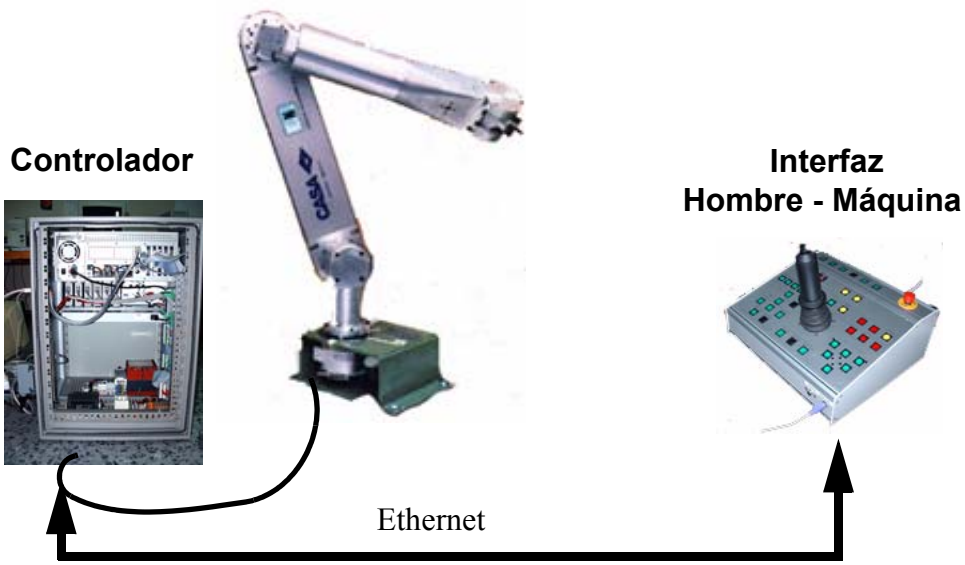



**Figura 7.2: Arquitectura del BTM**

La plataforma consta de dos estaciones:

- **Controlador del brazo**: Como puede verse en la figura 7.3, esta estación se encarga de comunicarse eléctricamente con el brazo articulado además de recabar información sobre el estado de éste e informar al software de su estado:

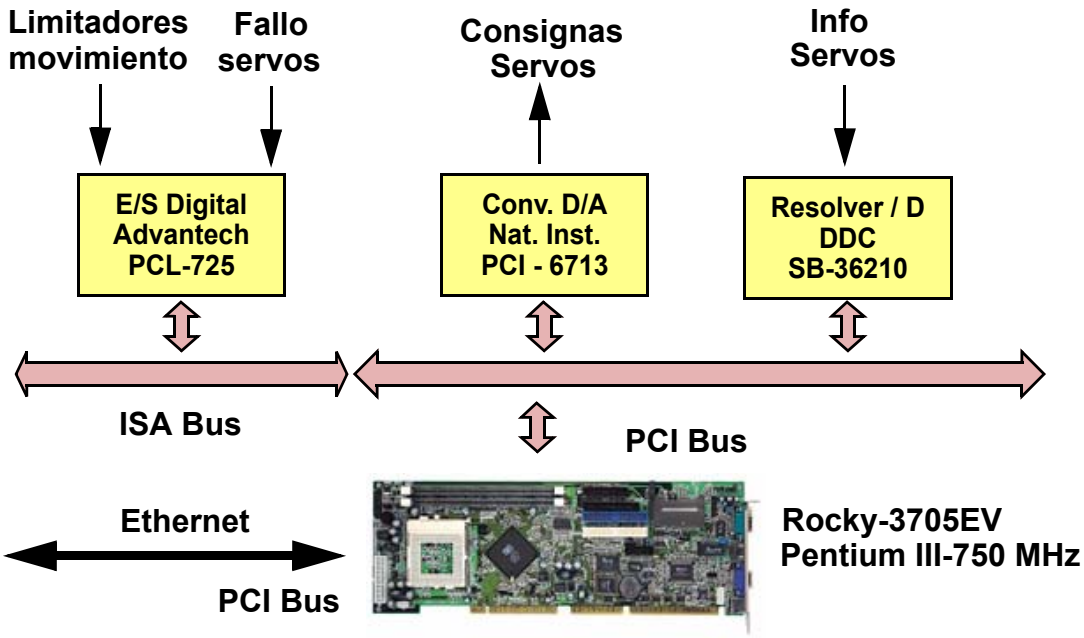



**Figura 7.3: Arquitectura controlador**

- Recupera mediante una tarjeta de Entrada / Salida digital los finales de carrera de los motores, las señales de fallos de los servos y la activación de la seta de emergencia además de habilitar los servos para su utilización.
- Con una tarjeta digital / analógica es capaz de asignar consigna a cada servo.
- Mediante una tarjeta resolver / digital puede averiguar el ángulo que forma el motor.
- Se encuentra conectado a una red Ethernet de 100 Mbps.

- **Interfaz hombre-máquina**: Se encarga de transmitir las órdenes del operador al brazo articulado. La arquitectura se muestra en la figura 7.4.Y está compuesto por:

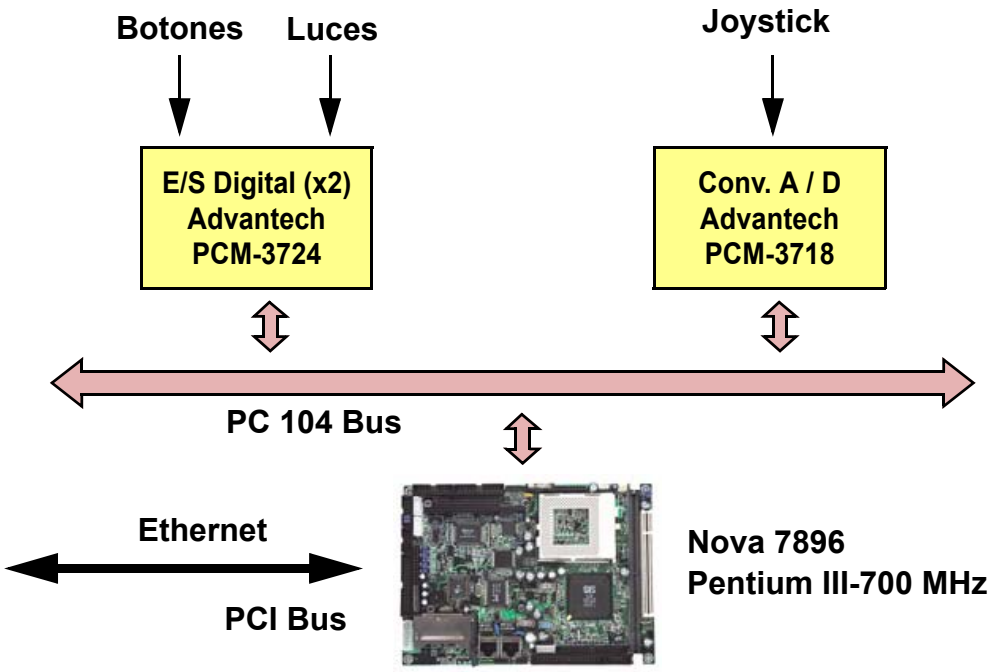



**Figura 7.4: Arquitectura interfaz hombre-máquina.**

- Conversor analógico / digital para poder recuperar la información suministrada a través del joystick.
- Dos tarjetas de entrada / salida digital para capturar las señales de los botones e informar con las luces.
- Conectada a una red Ethernet de 100 Mbps.

La figura 7.5 se muestra un esquema general de los objetos en que se ha dividido el software del controlador, así como las principales relaciones entre objetos. A continuación se describen brevemente cada uno de los objetos del primer nivel de descomposición del sistema:

- **Configuración**: En este objeto se describen todos los parámetros del sistema que son configurables. Aunque muchos de estos parámetros corresponden a otros objetos del sistema, tales como los mandos o el brazo, se ha preferido agruparlos todos en un mismo objeto con el fin de simplificar la labor de

configurar el brazo. Casi todos los objetos del sistema leen los parámetros configurables del objeto *Configuración*.

- **Mandos**: Este objeto engloba todos los elementos de la unidad de mandos del robot, en concreto el joystick, la botonera, las luces de la botonera, y la seta de emergencia. El objeto proporciona operaciones para leer el estado de los mandos, así como para poder encender o apagar las diferentes luces del cuadro de mandos.
- **Brazo**: Este objeto engloba todos los elementos del brazo telemanipulado. Los elementos se agrupan en dos tipos: sensores, cuyos valores se pueden leer, y actuadores, sobre los que se puede actuar. Los sensores son los topes de recorrido de cada eje y los sensores de posición (*resolvers*). Los actuadores son los que permiten especificar las consignas de intensidad de los servos, y el relé o relés que controlan la alimentación del brazo.
- **Gestor de Mandos:** Este objeto gestiona las ordenes que el usuario introduce a través de la unidad de mandos. En concreto determina el modo de funcionamiento del robot, gestiona los cambios de modo de funcionamiento, y almacena órdenes que el planificador de trayectorias debe ejecutar. También inicializa el control de servos, si el usuario lo solicita. Por último, gestiona las luces del cuadro de mandos, encendiéndolas y apagándolas en función del estado y de las alarmas existentes.
- **Planificador de Trayectorias**: este objeto se encarga de planificar las trayectorias que el robot debe de seguir. Como entradas, recibe el estado actual y las órdenes - que provienen del gestor de mandos - y los valores del joystick, y la posición actual del brazo. Como salida proporciona periódicamente puntos al controlador de servos, que se encargará de mover el brazo para ir alcanzando progresivamente esos puntos.
- **Control de Servos**: este objeto se encarga de ejecutar el algoritmo de control de servos. Como entradas recibe las posiciones actuales de los motores, así como el próximo punto a alcanzar, suministrado por el planificador de trayectorias. Genera como salida las consignas de intensidad de los motores.
- **Reportero**: este objeto se encarga de leer periódicamente el estado del sistema, y componer un mensaje que se envía a través de una línea serie. Este mensaje permite a un sistema externo conocer en cada momento el estado del sistema.
- **Alarmas**: este objeto se encarga de almacenar el estado de las alarmas del sistema. Otros objetos pueden modificar este estado, o leerlo.
- **Herramienta**. Define las operaciones asociadas a las herramientas que se pueden colocar en el robot. Actualmente sólo hay dos herramientas, una operación para conocer cuál de las dos está instalada, y operaciones para actuar sobre ellas.
- **PINROB**: Define una librería de control del robot en modo programado denominada PINROB, así como la aplicación que usa esa librería, y dos módulos para comunicación con el planificador de trayectorias: Movimiento programado y BTM Status.

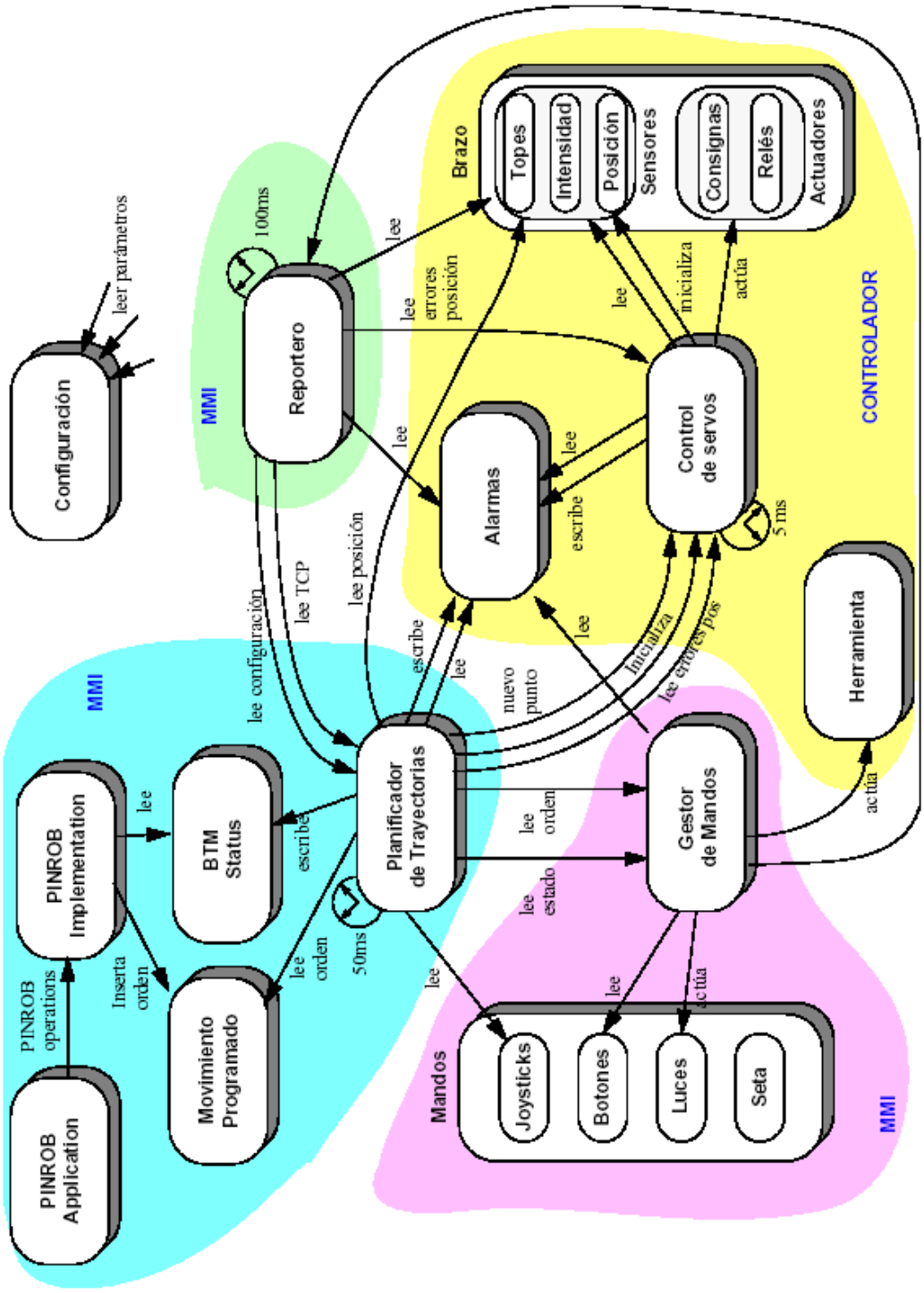



**Figura 7.5: Arquitectura software del controlador del BTM**

El software del controlador está diseñado para poder operar en un sistema distribuido. La figura 7.5 muestra las partes que podrían distribuirse en diferentes nudos del sistema distribuido.

Actualmente, la plataforma de ejecución sólamente dispone de dos estaciones conectados a través de una red ethernet, por lo que la asignación de módulos a estaciones es la siguiente:

- Módulos comunes replicados en todas las estaciones: Configuración.
- Módulos del controlador local: Alarmas, Control de servos, Brazo, y Herramienta.
- Módulos de la interfaz hombre-máquina (MMI): Reportero, Planificador de trayectorias. Módulos PINROB, Mandos, y Gestor de Mandos.

La tecnología que se utilizará para la distribución es la basada en el anexo de sistemas distribuidos del lenguaje Ada [45], utilizando la implementación propia del compilador *Gnat*, denominada Glade [29][30]. La implementación estándar de glade funciona sobre una red con protocolo TCP/IP, por lo que no ofrece funcionamiento de tiempo real.

Un proyecto dentro del departamento es el encargado de adaptar la implementación de glade para que opere con redes de tiempo real y para mejorar la gestión de las tareas [22]. Esta implementación se denomina RT-GLADE, y se basa en la utilización del protocolo de tiempo real sobre ethernet que versa este trabajo.

Constituye por tanto un demostrador ideal para el protocolo ya que se trata de una aplicación compleja completa, donde el más mínimo fallo o retraso es apreciable y además la comunicación de red en tiempo real es la piedra angular del demostrador.

Además de la implementación del demostrador y de la experimentación de su funcionamiento correcto se tomaron una serie de tiempos para corroborar el buen funcionamiento y se extrajo el modelo MAST del conjunto para su análisis.

# 7.2. Modelo MAST del BTM

En la medición de las operaciones que componen el BTM se utilizó una librería externa de nuestra creación, que nos proporciona el tiempo de ejecución de los fragmentos de código elegidos.

Para un mejor entendimiento del modelo MAST separaremos el fichero de descripción del BTM en distintos apartados, siendo el fichero final la adición de todos los apartados. Esta sección complementa al apartado 5.1, “Introducción a MAST”.

## 7.2.1. Identificando el modelo.

Lo primero que hay que especificar en un fichero MAST es la directiva que identifica al modelo. La directiva es la palabra reservada *Model* y como atributos tiene:

- **Model_Name**: El nombre que se quiera dar al modelo.
- **Model_Date**: La fecha de creación del modelo (AAAA-MM-DD).

Por tanto esta directiva quedará de la forma:

```
Model (
     Model_Name  => BTM,
     Model_Date  => 2004-07-25);
```

### 7.2.2. Recursos de procesado.

Corresponde con el tipo de procesador que se encargará de ejecutar el código de la aplicación. Se expresa mediante la directiva *Processing_Resource* y admite como argumentos:

- **Type**: Expresa el tipo de recurso de procesado que se trata.
- **Name**: Indica el nombre por el cual sera reconocido por la herramienta.
- **Max_Interrupt_Priority** / **Min_Interrupt_Priority**: Corresponden con el rango de prioridades válidas para las actividades planificadas por una rutina de interrupción.
- **ISR_Switch_Overheads (Worst, Average, Best)**: Expresa el coste del cambio de contexto.
- **Speed_Factor**: Indica el valor con el que se escalarán los tiempos para adecuarlos a la potencia del procesador.

Particularizando y al tener dos procesadores esta directiva nos quedará:

```
Processing_Resource (
     Type                 => Regular_Processor,
     Name                 => btmcont,
     Speed_Factor         => 1,
     Max_Interrupt_Priority=> 32,
     Min_Interrupt_Priority=> 32);

Processing_Resource (
     Type                 => Regular_Processor,
     Name                 => btmmmi,
     Speed_Factor         => 1,
     Max_Interrupt_Priority=> 32,
     Min_Interrupt_Priority=> 32);
```

### 7.2.3. Planificadores

Los planificadores (*Schedulers*) representan los objetos del sistema operativo que implementan las estrategias apropiadas de planificación necesarias para utilizar la capacidad de procesado que les ha sido asignada. Se expresan mediante la directiva *Schedulers* y admiten los siguientes atributos:

- **Type**: Indica si se trata del servidor base de planificación o de un servidor secundario.
- **Name**: Corresponde con el identificador del planificador.
- **Policy**: Define la política de planificación que implementa el planificador para distribuir la capacidad de procesado entre los servidores de planificación asociados a él.
- **Host** / **Server**: Dependiendo de que se trate de un planificador primario o secundario, hará referencia al recurso de procesado o al servidor de planificación asociado.

En el demostrador esta directiva quedará:

```
Scheduler (
     Type                  => Primary_Scheduler,
     Name                  => btmcont,
     Policy                =>
            (Type                    => Fixed_Priority,
             Max_Priority            => 30,
             Min_Priority            => 1),
     Host                  => btmcont);

Scheduler (
     Type                  => Primary_Scheduler,
     Name                  => btmmmi,
     Policy                =>
            (Type                    => Fixed_Priority,
             Max_Priority            => 30,
             Min_Priority            => 1),
     Host                  => btmmmi);
```

### 7.2.4. Servidores de planificación

Los servidores de planificación (Scheduling Servers) representan entidades planificables en los recursos de procesado. Se expresan mediante la directiva *Scheduling_Server* que admite como atributos:

- **Type**: Identifica el tipo de servidor de planificación que se trata.
- **Name**: Es el identificador del servidor de planificación en el modelo.
- **Server_Sched_Parameters**: Expresa distintos parámetros relacionados con el servidor.
- **Scheduler**: Indica el planificador sobre el que llevará a cabo su actividad.

En nuestro caso esta directiva quedara como sigue:

```
Scheduling_Server (
     Type                  => Regular,
     Name                  => Servo_Control,
     Server_Sched_Parameters=> (
            Type           => Fixed_Priority_Policy,
            The_Priority   => 22,
            Preassigned    => No ),
     Scheduler             => Btmcont );

Scheduling_Server (
     Type                  => Regular,
     Name                  => Trajectory_Planning,
     Server_Sched_Parameters=> (
            Type           => Fixed_Priority_Policy,
            The_Priority   => 20,
            Preassigned    => No ),
     Scheduler             => Btmmmi );

Scheduling_Server (
     Type                  => Regular,
     Name                  => Light_Manager,
     Server_Sched_Parameters=> (
            Type           => Fixed_Priority_Policy,
            The_Priority   => 18,
            Preassigned    => No ),
     Scheduler             => Btmmmi );
```

```
Scheduling_Server (
     Type                 => Regular,
     Name                 => Reporter,
     Server_Sched_Parameters=> (
            Type          => Fixed_Priority_Policy,
            The_Priority  => 10,
            Preassigned   => No ),
     Scheduler            => Btmmmi );

Scheduling_Server (
     Type                 => Regular,
     Name                 => Message_Logger,
     Server_Sched_Parameters=> (
            Type          => Fixed_Priority_Policy,
            The_Priority  => 5,
            Preassigned   => No ),
     Scheduler            => Btmmmi );
```

## 7.2.5. Recursos compartidos

Representan los recursos que se comparten entre distintas tareas y que deben ser usados de un modo mutuamente exclusivo. Se expresan mediante la directiva *Shared_Resource* que admite como atributos:

- **Type**: Identifica el tipo de recurso compartido que estamos utilizando.
- **Name**: Es el identificador del recurso compartido.
- **Ceiling**: Expresa el techo de prioridad utilizado para el recurso.
- **Preassigned**: Si el parámetro es asignado al valor “No” el techo de prioridad puede ser calculado por la herramienta. Si se precisa una asignación concreta debe ponerse a “Yes”.

En nuestra aplicación disponemos de los siguientes recursos compartidos:

```
Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Servo_Data,
     Ceiling     => 22,
     Preassigned => No);

Shared_Resource (
     type        => Immediate_Ceiling_Resource,
     Name        => Arm,
     Ceiling     => 22,
     Preassigned => No);

Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Alarms,
     Ceiling     => 22,
     Preassigned => No);

Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Position_Command,
     Ceiling     => 20,
     Preassigned => No);
```

```
Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Error_Log,
     Ceiling     => 20,
     Preassigned => No);

Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Joystick_Data,
     Ceiling     => 20,
     Preassigned => No);

Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Lights_Data,
     Ceiling     => 20,
     Preassigned => No);

Shared_Resource (
     Type        => Immediate_Ceiling_Resource,
     Name        => Button_Data,
     Ceiling     => 20,
     Preassigned => No);
```

### 7.2.6. Operaciones

Las operaciones representan una parte de código o un mensaje. Se expresan mediante la directiva *Operation* que admite como argumentos:

- **Type**: Identifica el tipo de operación. En nuestro caso las operaciones son simples y *enclosing*.
- **Name**: Expresa el identificador de la operación.
- **Worst_Case_Execution_Time**: Representa el peor caso del tiempo de ejecución de la operación.
- **Avg_Case_Execution_Time**: Indica el tiempo medio de ejecución de la operación.
- **Best_case_Execution_Time**: Expresa la ejecución más corta de la operación.
- **Shared_Resources_List**: Indica el recurso compartido al que la operación está ligada.

Las operaciones de nuestra aplicación con los tiempos expresados es segundos quedarían:

```
Operation(
     Type                      => Simple,
     Name                      => Lectura_Pos_Ejes,
     Worst_Case_Execution_Time => 5.6922e-05,
     Best_Case_Execution_Time  => 4.1244e-05,
     Avg_Case_Execution_Time   => 4.4181e-05,
     Shared_Resources_List     => (Arm) );
```

```
Operation(
     Type                       => Simple,
     Name                       => Lee_Posicion_Ejes,
     Worst_Case_Execution_Time  => 6.0138e-05,
     Best_Case_Execution_Time   => 4.4195e-05,
     Avg_Case_Execution_Time    => 4.7325e-05,
     Shared_Resources_List      => (Arm) );

Operation(
     Type                       => Simple,
     Name                       => Lee_Servos_OK,
     Worst_Case_Execution_Time  => 2.6092e-05,
     Best_Case_Execution_Time   => 1.1022e-05,
     Avg_Case_Execution_Time    => 1.3259e-05,
     Shared_Resources_List      => (Arm));

Operation(
     Type                       => Simple,
     Name                       => Lee_Topes_Inferiores,
     Worst_Case_Execution_Time  => 5.9553e-05,
     Best_Case_Execution_Time   => 4.254e-05,
     Avg_Case_Execution_Time    => 4.731e-05,
     Shared_Resources_List      => (Arm));

Operation(
     Type                       => Simple,
     Name                       => Lee_Topes_Superiores,
     Worst_Case_Execution_Time  => 6.0035e-05,
     Best_Case_Execution_Time   => 4.2858e-05,
     Avg_Case_Execution_Time    => 4.8314e-05,
     Shared_Resources_List      => (Arm));

Operation(
     Type                       => Simple,
     Name                       => Inicializa_Posicion,
     Worst_Case_Execution_Time  => 7.1767e-05,
     Shared_Resources_List      => (Arm));

Operation(
     Type                       => Simple,
     Name                       => Actua_Rele,
     Worst_Case_Execution_Time  => 1.6249e-05,
     Best_Case_Execution_Time   => 1.5007e-05,
     Shared_Resources_List      => (Arm));

Operation(
     Type                       => Simple,
     Name                       => Controla_Servos,
     Worst_Case_Execution_Time  => 4.7765e-05,
     Best_Case_Execution_Time   => 3.1555e-05,
     Avg_Case_Execution_Time    => 3.4216e-05,
     Shared_Resources_List      => (Arm));

Operation(
     Type                       => Simple,
     Name                       => Lee_Intensidad_Motores,
     Worst_Case_Execution_Time  => 6.0138e-05,
     Best_Case_Execution_Time   => 4.4195e-05,
     Avg_Case_Execution_Time    => 4.7325e-05,
     Shared_Resources_List      => (Arm));
```

```
Operation(
     Type                       => Simple,
     Name                       => Lee_Todas_Alarmas,
     Worst_Case_Execution_Time  => 1.2319e-05,
     Best_Case_Execution_Time   => 9.38e-07,
     Avg_Case_Execution_Time    => 1.143e-06,
     Shared_Resources_List      => (Alarms) );

Operation(
     Type                        => Simple,
     Name                        => Activa,
     Worst_Case_Execution_Time   => 2.132e-06,
     Best_Case_Execution_Time    => 1.084e-06,
     Shared_Resources_List       => (Alarms));

Operation(
     Type                        => Simple,
     Name                        => Desactiva,
     Worst_Case_Execution_Time   => 1.749e-06,
     Best_Case_Execution_Time    => 1.146e-06,
     Shared_Resources_List       => (Alarms));

Operation(
     Type                        => Simple,
     Name                        => Reconoce,
     Worst_Case_Execution_Time   => 1.890e-06,
     Best_Case_Execution_Time    => 1.465e-06,
     Shared_Resources_List       => (Alarms));

Operation(
     Type                        => Simple,
     Name                        => Inserta_Error,
     Worst_Case_Execution_Time   => 2.769e-06,
     Best_Case_Execution_Time    => 2.037e-06,
     Shared_Resources_List       => (Alarms));

Operation(
     Type                        => Simple,
     Name                        => Extrae_Error,
     Worst_Case_Execution_Time   => 3.948e-06,
     Best_Case_Execution_Time    => 1.77e-06,
     Avg_Case_Execution_Time     => 2.255e-06,
     Shared_Resources_List       => (Alarms));

Operation(
     Type                        => Simple,
     Name                        => Lee_Nuevo_Punto,
     Worst_Case_Execution_Time   => 1.1153e-05,
     Best_Case_Execution_Time    => 7.33e-07,
     Avg_Case_Execution_Time     => 9.92e-07,
     Shared_Resources_List       => (Servo_Data));

Operation(
     Type                        => Simple,
     Name                        => Escribe_Nuevo_Punto,
     Worst_Case_Execution_Time   => 1.1344e-05,
     Best_Case_Execution_Time    => 1.063e-06,
     Avg_Case_Execution_Time     => 1.366e-06,
     Shared_Resources_List       => (Servo_Data));
```

```
Operation(
     Type                        => Simple,
     Name                        => Lee_Errores_Posicion,
     Worst_Case_Execution_Time   => 1.831e-06,
     Best_Case_Execution_Time    => 1.192e-06,
     Avg_Case_Execution_Time     => 1.363e-06,
     Shared_Resources_List       => (Servo_Data));

Operation(
     Type                        => Simple,
     Name                        =>
                                Escribe_Errores_Posicion,
     Worst_Case_Execution_Time   => 1.1618e-05,
     Best_Case_Execution_Time    => 8.73e-07,
     Avg_Case_Execution_Time     => 9.94e-07,
     Shared_Resources_List       => (Servo_Data));

Operation(
     Type                        => Simple,
     Name                        => Lee_Pos_Inic,
     Worst_Case_Execution_Time   => 1.1933e-05,
     Best_Case_Execution_Time    => 7.73e-07,
     Avg_Case_Execution_Time     => 1.16e-06,
     Shared_Resources_List       => (Servo_Data));

Operation(
     Type                        => Simple,
     Name                        => Escribe_Pos_Inic,
     Worst_Case_Execution_Time   => 5.149e-06,
     Shared_Resources_List       => (Servo_Data));

Operation(
     Type                        => Simple,
     Name                        => Inserta_Elemento,
     Worst_Case_Execution_Time   => 1.646e-06,
     Best_Case_Execution_Time    => 1.105e-06,
     Shared_Resources_List       => (Error_Log));

Operation(
     Type                        => Simple,
     Name                        => Lee_Elemento,
     Worst_Case_Execution_Time   => 1.871e-06,
     Best_Case_Execution_Time    => 1.158e-06,
     Shared_Resources_List       => (Error_Log));

Operation(
     Type                        => Simple,
     Name                        => Lee,
     Worst_Case_Execution_Time   => 0.000119104,
     Best_Case_Execution_Time    => 9.308e-05,
     Avg_Case_Execution_Time     => 0.00010639,
     Shared_Resources_List       => (Joystick_Data));

Operation(
     Type                        => Simple,
     Name                        => Lee_Estado_Actual,
     Worst_Case_Execution_Time   => 9.213e-06,
     Best_Case_Execution_Time    => 7.519e-06,
     Avg_Case_Execution_Time     => 8.101e-06,
     Shared_Resources_List       => (Button_Data));
```

```
Operation(
     Type                        => Simple,
     Name                        => Lee_Todos,
     Worst_Case_Execution_Time   => 0.000258447,
     Best_Case_Execution_Time    => 0.000233221,
     Avg_Case_Execution_Time     => 0.000248938,
     Shared_Resources_List       => (Button_Data));

Operation(
     Type                        => Simple,
     Name                        => Detecta_Cambio,
     Worst_Case_Execution_Time   => 2.045e-05,
     Best_Case_Execution_Time    => 8.402e-06,
     Avg_Case_Execution_Time     => 1.0167e-05,
     Shared_Resources_List       => (Button_Data));

Operation(
     Type                        => Simple,
     Name                        => Detecta_Pulsacion,
     Worst_Case_Execution_Time   => 2.0752e-05,
     Best_Case_Execution_Time    => 8.825e-06,
     Avg_Case_Execution_Time     => 1.0222e-05,
     Shared_Resources_List       => (Button_Data));

Operation(
     Type                        => Simple,
     Name                        => Enciende,
     Worst_Case_Execution_Time   => 1.9376e-05,
     Best_Case_Execution_Time    => 9.569e-06,
     Shared_Resources_List       => (Lights_Data));

Operation(
     Type                        => Simple,
     Name                        => Apaga,
     Worst_Case_Execution_Time   => 2.1019e-05,
     Best_Case_Execution_Time    => 9.256e-06,
     Avg_Case_Execution_Time     => 1.078e-05,
     Shared_Resources_List       => (Lights_Data));

Operation(
     Type                        => Simple,
     Name                        => Enciende_Temporizada,
     Worst_Case_Execution_Time   => 1.3634e-05,
     Best_Case_Execution_Time    => 1.1668e-05,
     Shared_Resources_List       => (Lights_Data));

Operation(
     Type                        => Simple,
     Name                        => Enciende_con_Parpadeo,
     Worst_Case_Execution_Time   => 1.4541e-05,
     Best_Case_Execution_Time    => 1.3352e-05,
     Shared_Resources_List       => (Lights_Data));

Operation(
     Type                        => Simple,
     Name                        => Temporiza_Luces,
     Worst_Case_Execution_Time   => 3.089e-05,
     Best_Case_Execution_Time    => 3.569e-06,
     Avg_Case_Execution_Time     => 5.643e-06,
     Shared_Resources_List       => (Lights_Data));
```

```
Operation(
     Type                        => Simple,
     Name                        => Esta_Apagada,
     Worst_Case_Execution_Time   => 1.051e-05,
     Best_Case_Execution_Time    => 7.62e-07,
     Avg_Case_Execution_Time     => 9.14e-07,
     Shared_Resources_List       => (Lights_Data));

Operation(
     Type                        => Simple,
     Name                        => Lee_Consigna_Base,
     Worst_Case_Execution_Time   => 2.594e-06,
     Best_Case_Execution_Time    => 1.084e-06,
     Avg_Case_Execution_Time     => 1.52e-06,
     Shared_Resources_List       => (Position_Command));

Operation(
     Type                        => Simple,
     Name                        => Pon_Consigna_Base,
     Worst_Case_Execution_Time   => 2.91e-06,
     Best_Case_Execution_Time    => 1.15e-06,
     Avg_Case_Execution_Time     => 1.906e-06,
     Shared_Resources_List => (Position_Command));

Operation(
     Type                        => Enclosing,
     Name                        => Reportero,
     Worst_Case_Execution_Time   => 0.0204405,
     Best_Case_Execution_Time    => 0.0202658,
     Avg_Case_Execution_Time     => 0.0203834,
     Composite_Operation_List    => (Lee_Todas_Alarmas,
            Lee_Errores_Posicion, Lee_Todos, Lee,
            Lee_Intensidad_Motores, Lee_Posicion_Ejes,
            Lee_Topes_Inferiores, Lee_Topes_Superiores,
            Lee_Consigna_Base, Inserta_Error));

Operation(
     Type                        => Enclosing,
     Name                        => Control_servos,
     Worst_Case_Execution_Time   => 0.000463417,
     Best_Case_Execution_Time    => 0.000240053,
     Avg_Case_Execution_Time     => 0.000262042,
     Composite_Operation_List    => (Lee_Todas_Alarmas,
            Inicializa_Posicion, Actua_Rele,
            Lee_Posicion_Ejes, Escribe_Errores_Posicion,
            Desactiva, Activa, Inserta_Error,
            Lee_Nuevo_Punto, Controla_Servos));

Operation(
     Type                        => Enclosing,
     Name                        => Gestion_Luces,
     Worst_Case_Execution_Time   => 5.8307e-05,
     Best_Case_Execution_Time    => 2.4955e-05,
     Avg_Case_Execution_Time     => 3.206e-05,
     Composite_Operation_List    => (Temporiza_Luces));
```

```
Operation(
     Type                        => Enclosing,
     Name                        => Error_Logger,
     Worst_Case_Execution_Time   => 0.00170959,
     Best_Case_Execution_Time    => 0.00167901,
     Composite_Operation_List    => (Lee_Elemento));

Operation(
     Type                        => Enclosing,
     Name                   => Planificador_trayectorias,
     Worst_Case_Execution_Time   => 0.00377349,
     Best_Case_Execution_Time    => 0.00218197,
     Avg_Case_Execution_Time     => 0.00248435,
     Composite_Operation_List    => (Detecta_Pulsacion,
            Enciende, Apaga, Enciende_Temporizada,
            Enciende_con_Parpadeo, Temporiza_Luces,
            Esta_Apagada, Detecta_Cambio,
            Lee_Todas_Alarmas, Lee, Lee_Posicion_Ejes,
            Lee_Topes_Superiores, Lee_Topes_Inferiores,
            Activa, Inserta_Error, Lee_Errores_Posicion,
            Escribe_Pos_Inic, Lee_Todos, Desactiva,
            Reconoce, Notifica_Error, Lee_Estado_Actual,
            Inserta_Elemento));
```

### 7.2.7. Transacciones

Una transacción es un grafo de evento y manejadores de eventos que representan las actividades que son ejecutadas en un sistema que están relacionadas. Se expresan mediante la directiva *Transaction* que contiene los atributos:

- **Type**: Indica el tipo de transacción, por ahora solo existe la clase *Regular*.
- **Name**: Es el nombre con el que se identifica la transacción.
- **External_Events**: Es una lista de eventos externos que interaccionan con la transacción.
- **Internal_Events**: Consisten en una lista de eventos internos, con sus requisitos temporales si existiesen.
- **Event_Handlers**: Compuesto por una lista de manejadores de eventos.

En el caso del demostrador tendremos las siguientes transacciones:

```
Transaction(
     Type                 => Regular,
     name                 => Servo_Control,
     External_Events=> (
            (Type         => Periodic,
            Name          => E1,
            Period        => 5.0E-3)),
     Internal_Events      => (
            (Type         => Regular,
            Name          => O1,
            Timing_Requirements=> (
```

```
                    (Type              =>Hard_Global_Deadline,
                    Deadline           => 5E-3,
                    referenced_event   =>E1))),
          Event_Handlers=>(
                 (Type              => Activity,
                 Input_Event        => E1,
                 Output_event       => O1,
                 Activity_Operation => Control_Servos,
                 Activity_Server    => Servo_Control)));

     Transaction(
          Type                 => Enclosing,
          name                 => Light_Manager,
          External_Events=> (
                 (Type          => Periodic,
                 Name           => E2,
                 Period         => 100E-3)),
          Internal_Events       => (
                 (Type          => Regular,
                 Name           => O2,
                 Timing_Requirements=> (
                      Type               =>Hard_Global_Deadline,
                      Deadline           => 100E-3,
                      referenced_event   =>E2))),
          Event_Handlers=>(
                 (Type              => Activity,
                 Input_Event        => E2,
                 Output_event       => O2,
                 Activity_Operation => Gestion_Luces,
                 Activity_Server    => Light_Manager)));

     Transaction(
          Type                 => Enclosing,
          name                 => Message_Logger,
          External_Events=> (
                 (Type          => Unbounded,
                 Name           => E3,
                 Avg_Interarrival=> 1)),
          Internal_Events       => (
                 (Type          => Regular,
                 Name           => O3)),
          Event_Handlers=>(
                 (Type              => Activity,
                 Input_Event        => E3,
                 Output_event       => O3,
                 Activity_Operation => Error_Logger,
                 Activity_Server    => Message_Logger)));
```

La transacción correspondiente al módulo **reportero** es una transaccion distribuida por lo que es necesario especificar cada una de sus actividades en cada recurso de procesado (CPUs y red). Además es necesario especificar una serie de operaciones y servidores de planificación para completar la operación de cada actividad. El código correspondiente a cada operación de cada actividad se ha separado en secciones convenientemente indicadas en el nombre de cada operación y su servidor de planificación asociado. Un esquema general de como quedaría una transacción en caso de llamada remota a través de la red puede verse en la figura 7.6.

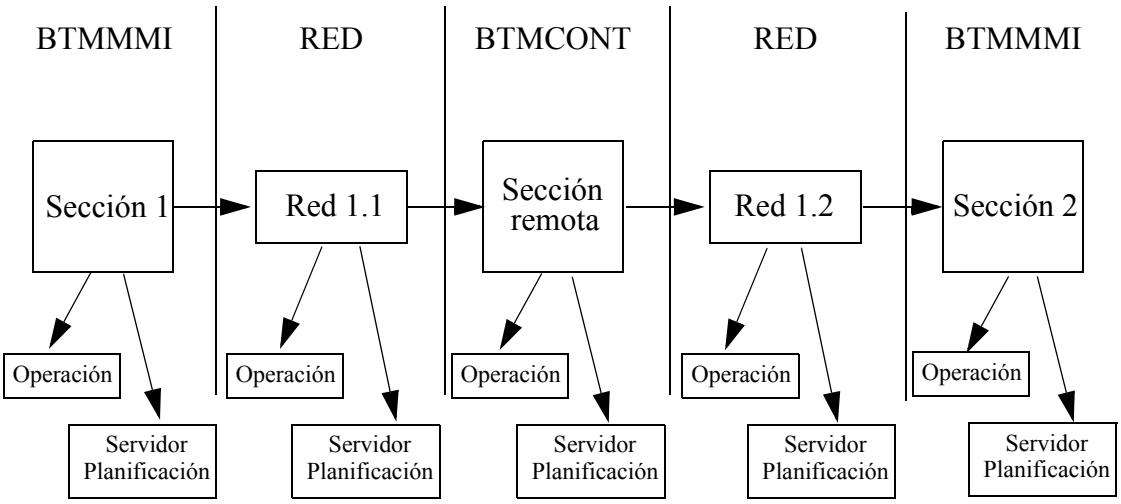



**Figura 7.6: Esquema transaccional de una llamada remota.**

Este módulo realiza seis llamadas remotas consecutivas al principio. La transacción queda por lo tanto:

- **Reportero sección 1:** Corresponde a la parte del código del reportero que se ejecuta en el recurso de procesado btmmmi. La actividad corresponde a la iniciada con el evento externo y finaliza al realizar la primera llamada remota.

```
Operation(
     Type                        => Simple,
     Name                        => Rep_Section_1,
     Worst_Case_Execution_Time => 5.0634e-5,
     Best_Case_Execution_Time  => 2.4614e-5,
     Avg_Case_Execution_Time   => 3.2174e-5);
```

  El servidor de planificación que le corresponderá es *Reporter*.

- **Red sección 1.1**: Corresponde con la actividad iniciada con la finalización de la primera sección de reportero y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont.

```
Operation(
     Type                          => Message_Transmission,
     Name                    => Alarmas_Lee_Todas_Alarmas,
     Max_Message_Size        => 1492,
     Avg_Message_Size        => 1492,
     Min_Message_size        => 1492);

Scheduling_Server (
     Type                    => Regular,
     Name                    => Reportero_RPC,
     Server_Sched_Parameters=> (
            Type             => Fixed_Priority_Policy,
            The_Priority     => 10,
            Preassigned      => No ),
     Scheduler               => RT_EP_Network_Sched );
```

- **Lee_Todas_Alarmas**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Todas_Alarmas* y con el servidor de planificación *Reportero_Remote*:

```
Scheduling_Server (
     Type                  => Regular,
     Name                  => Reportero_Remote,
     Server_Sched_Parameters=> (
            Type           => Fixed_Priority_Policy,
            The_Priority   => 10,
            Preassigned    => No ),
     Scheduler             => Btmcont );
```

- **Red sección 1.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la primera llamada remota, y finaliza cuando ha sido recibida correctamente por el reportero. El servidor de planificación es el mismo que en la sección 1.1 ya que el mensaje es de la misma prioridad y tipo, sólo cambia el origen y destino de éste.

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Alarmas_Lee_Todas_Alarmas_2,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Reportero sección 2:** La actividad se ve iniciada cuando se recibe la finalización de la primera llamada remota y finaliza cuando se inicia la segunda llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Rep_Section_2,
     Worst_Case_Execution_Time => 9.51e-7);
```

  El servidor de planificación que le corresponderá es *Reporter*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 2.1**: Corresponde con la actividad iniciada con la finalización de la segunda sección de reportero y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont.

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Servos_Lee_Errores_Posicion,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Lee_Errores_Posicion**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Errores_Posicion* y con el servidor de planificación *Reportero_Remote*.

- **Red sección 2.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la segunda llamada remota, y finaliza cuando ha sido recibida correctamente por el reportero. El servidor de planificación es el mismo que en la sección 2.1 ya que el mensaje es de la misma prioridad y tipo, solo cambia el origen y destino de éste.

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Servos_Lee_Errores_Posicion_2,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Reportero sección 3:** La actividad se ve iniciada cuando se recibe la finalización de la segunda llamada remota y finaliza cuando se inicia la tercera llamada remota.

```
Operation(
     Type                        => Simple,
     Name                        => Rep_Section_3,
     Worst_Case_Execution_Time => 9.51e-7);
```

  El servidor de planificación que le corresponderá es *Reporter*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 3.1**: Corresponde con la actividad iniciada con la finalización de la tercera sección de reportero y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont.

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Brazo_Lee_Intensidad_Motores,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Lee_Intensidad_Motores**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Intensidad_Motores* y con el servidor de planificación *Reportero_Remote*.

- **Red sección 3.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la tercera llamada remota, y finaliza cuando ha sido recibida correctamente por el reportero. El servidor de planificación es el mismo que en la sección 3.1.

```
Operation(
     Type                  => Message_Transmission,
     Name        => Brazo_Lee_Intensidad_Motores_2,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Reportero sección 4:** La actividad se ve iniciada cuando se recibe la finalización de la tercera llamada remota y finaliza cuando se inicia la cuarta llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Rep_Section_4,
     Worst_Case_Execution_Time => 9.51e-7);
```

  El servidor de planificación que le corresponderá es *Reporter*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 4.1**: Corresponde con la actividad iniciada con la finalización de la cuarta sección de reportero y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Brazo_Lee_Posicion_Ejes,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Lee_Posicion_Ejes**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Posicion_Ejes* y con el servidor de planificación *Reportero_Remote*.

- **Red sección 4.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la cuarta llamada remota, y finaliza cuando ha sido recibida correctamente por el reportero. El servidor de planificación es el mismo que en la sección 4.1.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Brazo_Lee_Posicion_Ejes_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

  El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Reportero sección 5:** La actividad se ve iniciada cuando se recibe la finalización de la cuarta llamada remota y finaliza cuando se inicia la quinta llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Rep_Section_5,
     Worst_Case_Execution_Time => 9.51e-7);
```

  El servidor de planificación que le corresponderá es *Reporter*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 5.1**: Corresponde con la actividad iniciada con la finalización de la quinta sección de reportero y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Brazo_Lee_Topes_Inferiores,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Lee_Topes_Inferiores**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Topes_Inferiores* y con el servidor de planificación *Reportero_Remote*.
- **Red sección 5.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la quinta llamada remota, y finaliza cuando ha sido recibida correctamente por el reportero. El servidor de planificación es el mismo que en la sección 5.1.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Brazo_Lee_Topes_Inferiores_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Reportero sección 6:** La actividad se ve iniciada cuando se recibe la finalización de la quinta llamada remota y finaliza cuando se inicia la sexta llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Rep_Section_6,
     Worst_Case_Execution_Time => 9.51e-7);
```

El servidor de planificación que le corresponderá es *Reporter*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 6.1**: Corresponde con la actividad iniciada con la finalización de la sexta sección de reportero y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Brazo_Lee_Topes_Superiores,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Lee_Topes_Superiores**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Topes_Superiores* y con el servidor de planificación *Reportero_Remote*.
- **Red sección 6.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la sexta llamada remota, y finaliza cuando ha sido recibida

correctamente por el reportero. El servidor de planificación es el mismo que en la sección 6.1.

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Brazo_Lee_Topes_Superiores_2,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

El servidor de planificación que le corresponderá será *Reportero_RPC*.

- **Reportero sección 7:** La actividad se ve iniciada cuando se recibe la finalización de la sexta y última llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Rep_Section_7,
     Worst_Case_Execution_Time  => 0.0182211,
     Best_Case_Execution_Time   => 0.0180699,
     Avg_Case_Execution_Time    => 0.0181547);
```

El servidor de planificación que le corresponderá es *Reporter*

Para ilustrar de una manera más gráfica la composición de esta transacción se expone la figura 7.7, en donde se muestra mediante un diagrama toda la transacción descrita anteriormente.

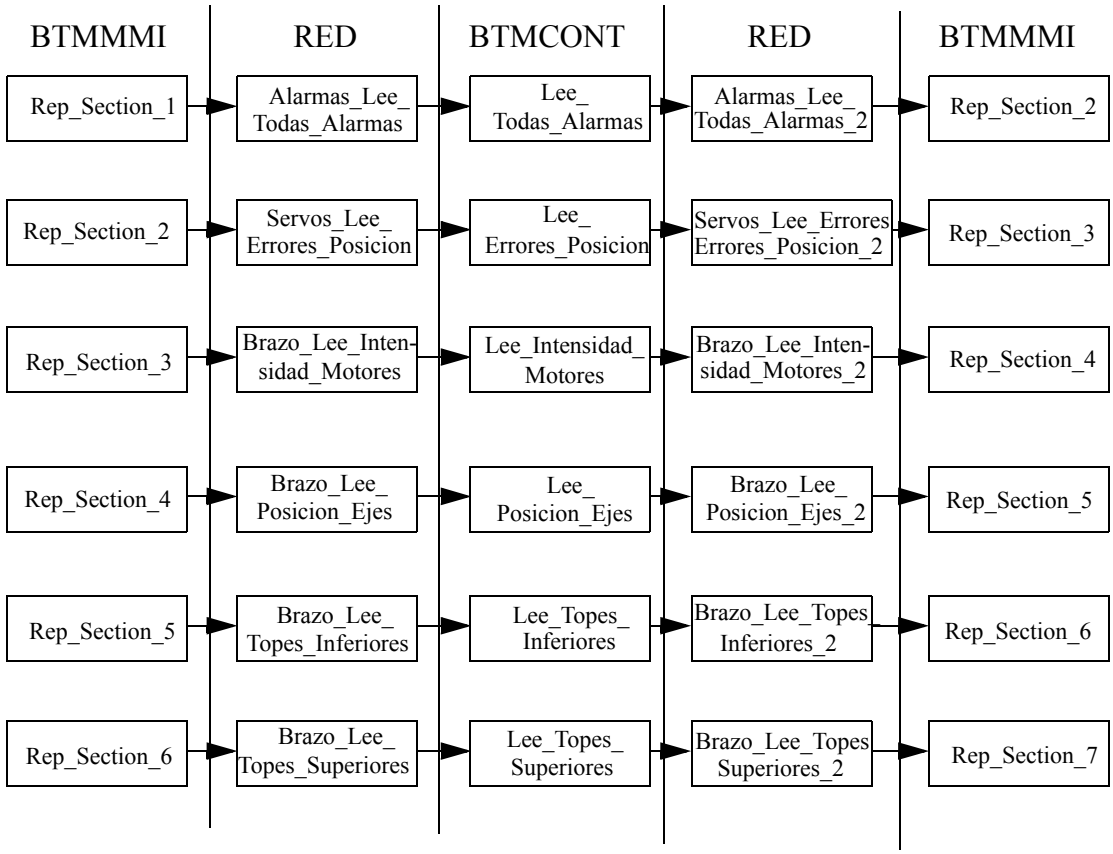



**Figura 7.7: Detalle de la transacción Reporter**

Segun todo lo expuesto la transacción nos quedara de la forma:

```
Transaction (
     Type                 => Regular,
     Name                 => Reporter,
     External_Events=> (
            (Type         => Periodic,
            Name          => In_reportero_01,
            Period        => 0.100)),
     Internal_Events=> (
            (Type         => Regular,
            Name          => Reportero_02),
            (Type         => Regular,
            Name          => Reportero_03),
            (Type         => Regular,
            Name          => Reportero_04),
            (Type         => Regular,
            Name          => Reportero_05),
            (Type         => Regular,
            Name          => Reportero_06),
            (Type         => Regular,
            Name          => Reportero_07),
            (Type         => Regular,
            Name          => Reportero_08),
            (Type         => Regular,
            Name          => Reportero_09),
            (Type         => Regular,
            Name          => Reportero_10),
            (Type         => Regular,
            Name          => Reportero_11),
            (Type         => Regular,
            Name          => Reportero_12),
            (Type         => Regular,
            Name          => Reportero_13),
            (Type         => Regular,
            Name          => Reportero_14),
            (Type         => Regular,
            Name          => Reportero_15),
            (Type         => Regular,
            Name          => Reportero_16),
            (Type         => Regular,
            Name          => Reportero_17),
            (Type         => Regular,
            Name          => Reportero_18),
            (Type         => Regular,
            Name          => Reportero_19),
            (Type         => Regular,
            Name          => Reportero_20),
            (Type         => Regular,
            Name          => Reportero_21),
            (Type         => Regular,
            Name          => Reportero_22),
            (Type         => Regular,
            Name          => Reportero_23),
            (Type         => Regular,
            Name          => Reportero_24),
            (Type         => Regular,
            Name          => Reportero_25),
            (Type         => Regular,
            Name          => Reportero_Out,
```

```
            Timing_Requirements => (
                 Type      => Hard_Global_Deadline,
                 Deadline  => 0.100,
                 Referenced_Event => In_Reportero_01)),
      Event_Handlers=>(
            (Type              => Activity,
            Input_Event        => In_Reportero_01,
            Output_event       => Reportero_02,
            Activity_Operation => Rep_Section_1,
            Activity_Server    => Reporter),
            (Type              => Activity,
            Input_Event        => Reportero_02,
            Output_event       => Reportero_03,
            Activity_Operation=> Alarmas_lee_Todas_Alarmas,
            Activity_Server    => Reportero_RPC),
            (Type              => Activity,
            Input_Event        => Reportero_03,
            Output_event       => Reportero_04,
            Activity_Operation=> Lee_Todas_Alarmas,
            Activity_Server    => Reportero_Remote),
            (Type              => Activity,
            Input_Event        => Reportero_04,
            Output_event       => Reportero_05,
            Activity_Operation =>
                        Alarmas_lee_Todas_Alarmas_2,
            Activity_Server    => Reportero_RPC),

            (Type              => Activity,
            Input_Event        => Reportero_05,
            Output_event       => Reportero_06,
            Activity_Operation => Rep_Section_2,
            Activity_Server    => Reporter),
            (Type              => Activity,
            Input_Event        => Reportero_06,
            Output_event       => Reportero_07,
            Activity_Operation =>
                        Servos_Lee_Errores_Posicion,
            Activity_Server    => Reportero_RPC),
            (Type              => Activity,
            Input_Event        => Reportero_07,
            Output_event       => Reportero_08,
            Activity_Operation=> Lee_Errores_Posicion,
            Activity_Server    => Reportero_Remote),
            (Type              => Activity,
            Input_Event        => Reportero_08,
            Output_event       => Reportero_09,
            Activity_Operation =>
                        Servos_Lee_Errores_Posicion_2,
            Activity_Server    => Reportero_RPC),

            (Type              => Activity,
            Input_Event        => Reportero_09,
            Output_event       => Reportero_10,
            Activity_Operation => Rep_Section_3,
            Activity_Server    => Reporter),
            (Type              => Activity,
            Input_Event        => Reportero_10,
            Output_event       => Reportero_11,
            Activity_Operation =>
```

```
                Brazo_Lee_Intensidad_Motores,
Activity_Server    => Reportero_RPC),
(Type              => Activity,
Input_Event        => Reportero_11,
Output_event       => Reportero_12,
Activity_Operation=> Lee_Intensidad_Motores,
Activity_Server    => Reportero_Remote),
(Type              => Activity,
Input_Event        => Reportero_12,
Output_event       => Reportero_13,
Activity_Operation =>
                Brazo_Lee_Intensidad_Motores_2,
Activity_Server    => Reportero_RPC),

(Type              => Activity,
Input_Event        => Reportero_13,
Output_event       => Reportero_14,
Activity_Operation => Rep_Section_4,
Activity_Server    => Reporter),
(Type              => Activity,
Input_Event        => Reportero_14,
Output_event       => Reportero_15,
Activity_Operation =>
                Brazo_Lee_Posicion_Ejes,
Activity_Server    => Reportero_RPC),
(Type              => Activity,
Input_Event        => Reportero_15,
Output_event       => Reportero_16,
Activity_Operation=> Lee_Posicion_Ejes,
Activity_Server    => Reportero_Remote),
(Type              => Activity,
Input_Event        => Reportero_16,
Output_event       => Reportero_17,
Activity_Operation =>
                Brazo_Lee_Posicion_ejes_2,
Activity_Server    => Reportero_RPC),

(Type              => Activity,
Input_Event        => Reportero_17,
Output_event       => Reportero_18,
Activity_Operation => Rep_Section_5,
Activity_Server    => Reporter),
(Type              => Activity,
Input_Event        => Reportero_18,
Output_event       => Reportero_19,
Activity_Operation =>
                Brazo_Lee_Topes_Inferiores,
Activity_Server    => Reportero_RPC),
(Type              => Activity,
Input_Event        => Reportero_19,
Output_event       => Reportero_20,
Activity_Operation=> Lee_Topes_Inferiores,
Activity_Server    => Reportero_Remote),
(Type              => Activity,
Input_Event        => Reportero_20,
Output_event       => Reportero_21,
Activity_Operation =>
                Brazo_Lee_Topes_Inferiores_2,
Activity_Server    => Reportero_RPC),
```

```
(Type              => Activity,
Input_Event        => Reportero_21,
Output_event       => Reportero_22,
Activity_Operation => Rep_Section_6,
Activity_Server    => Reporter),
(Type              => Activity,
Input_Event        => Reportero_22,
Output_event       => Reportero_23,
Activity_Operation =>
              Brazo_Lee_Topes_Superiores,
Activity_Server    => Reportero_RPC),
(Type              => Activity,
Input_Event        => Reportero_23,
Output_event       => Reportero_24,
Activity_Operation=> Lee_Topes_Superiores,
Activity_Server    => Reportero_Remote),
(Type              => Activity,
Input_Event        => Reportero_24,
Output_event       => Reportero_25,
Activity_Operation =>
              Brazo_Lee_Topes_Superiores_2,
Activity_Server    => Reportero_RPC),

(Type              => Activity,
Input_Event        => Reportero_25,
Output_event       => Reportero_Out,
Activity_Operation => Rep_Section_7,
Activity_Server    => Reporter)));
```

La transacción correspondiente al módulo **planificador_trayectorias** es una transacción distribuida, como lo era la del reportero, por lo que es necesario especificar cada una de sus actividades en cada recurso de procesado (CPUs y red). Además es necesario especificar una serie de operaciones y servidores de planificación para completar la operación de cada actividad. El código correspondiente a cada operación de cada actividad se ha separado en secciones convenientemente indicadas en el nombre de cada operación y su servidor de planificación asociado:

- **Planificador sección 1:** Corresponde a la parte del código del planificador que se ejecuta en el recurso de procesado btmmmi. La actividad corresponde a la iniciada con el evento externo y finaliza al realizar la primera llamada remota.

```
Operation(
     Type                      => Simple,
     Name                      => Traject_Section_1,
     Worst_Case_Execution_Time => 2.2044e-5,
     Best_Case_Execution_Time  => 7.019e-6,
     Avg_Case_Execution_Time   => 1.0198e-5);
```

  El servidor de planificación que le corresponderá es *Trajectory_Planning*.

- **Red sección 1.1**: Corresponde con la actividad iniciada con la finalización de la primera sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Lee_Todas_Alarmas* definida anteriormente en la transacción del modulo reportero.

```
Scheduling_Server (
     Type                  => Regular,
     Name                  => Planificador_RPC,
     Server_Sched_Parameters=> (
            Type           => Fixed_Priority_Policy,
            The_Priority   => 20,
            Preassigned    => No ),
     Scheduler             => RT_EP_Network_Sched );
```

- **Lee_Todas_Alarmas**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Todas_Alarmas* y con el servidor de planificación *Planificador_Remote*:

```
Scheduling_Server (
     Type                  => Regular,
     Name                  => Planificador_Remote,
     Server_Sched_Parameters=> (
            Type           => Fixed_Priority_Policy,
            The_Priority   => 20,
            Preassigned    => No ),
     Scheduler             => Btmcont );
```

- **Red sección 1.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la primera llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 1.1 ya que el mensaje es de la misma prioridad y tipo, solo cambia el origen y destino de éste y la operación corresponde con *Alarmas_Lee_Todas_Alarmas_2* definida anteriormente en la transacción del modulo reportero.

  El servidor de planificación que le corresponderá será *Planificador_RPC*.

- **Planificador sección 2:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la primera llamada remota y finaliza cuando se inicia la segunda llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_2,
     Worst_Case_Execution_Time  => 3.05e-4,
     Best_Case_Execution_Time   => 2.701e-4,
     Avg_Case_Execution_Time    => 2.788e-4);
```

  El servidor de planificación que le corresponderá es *Trajectory_Planning*.

- **Red sección 2.1**: Corresponde con la actividad iniciada con la finalización de la segunda sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación es Alarmas_Desactiva

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Alarmas_Desactiva,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

  El servidor de planificación que le corresponderá será *Planificador_RPC*

- **Desactiva**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Desactiva* y con el servidor de planificación *Planificador_Remote*.
- **Red sección 2.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la segunda llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 2.1 ya que el mensaje es de la misma prioridad y tipo, solo cambia el origen y destino de éste.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Alarmas_Desactiva_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

  El servidor de planificación que le corresponderá será *Planificador_RPC.*
- **Planificador sección 3:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la segunda llamada remota y finaliza cuando se inicia la tercera llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_3,
     Worst_Case_Execution_Time  => 9.51e-7);
```

  El servidor de planificación que le corresponderá es *Trajectory_Planning*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).
- **Red sección 3.1**: Corresponde con la actividad iniciada con la finalización de la tercera sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación es *Alarmas_Desactiva*. El servidor de planificación que le corresponderá será *Planificador_RPC*
- **Desactiva**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Desactiva* y con el servidor de planificación *Planificador_Remote*.
- **Red sección 3.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la tercera llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 3.1, *Planificador_RPC,* y la operación corresponde con *Alarmas_Desactiva_2*.
- **Planificador sección 4:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la tercera llamada remota y finaliza cuando se inicia la cuarta llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_4,
     Worst_Case_Execution_Time  => 9.51e-7);
```

El servidor de planificación que le corresponderá es *Trajectory_Planning*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 4.1**: Corresponde con la actividad iniciada con la finalización de la cuarta sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación es *Alarmas_Desactiva*. El servidor de planificación que le corresponderá será *Planificador_RPC*
- **Desactiva**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Desactiva* y con el servidor de planificación *Planificador_Remote*.
- **Red sección 4.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la cuarta llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 4.1, *Planificador_RPC,* y la operación corresponde con *Alarmas_Desactiva_2*.
- **Planificador sección 5:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la cuarta llamada remota y finaliza cuando se inicia la quinta llamada remota.

```
Operation(
     Type                        => Simple,
     Name                        => Traject_Section_5,
     Worst_Case_Execution_Time   => 2.0545e-5,
     Best_Case_Execution_Time    => 7.29e-7,
     Avg_Case_Execution_Time     => 7.46e-7);
```

El servidor de planificación que le corresponderá es *Trajectory_Planning*.

- **Red sección 5.1**: Corresponde con la actividad iniciada con la finalización de la quinta sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación es *Alarmas_Desactiva*. El servidor de planificación que le corresponderá será *Planificador_RPC*
- **Desactiva**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Desactiva* y con el servidor de planificación *Planificador_Remote*.
- **Red sección 5.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la quinta llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 5.1, *Planificador_RPC,* y la operación corresponde con *Alarmas_Desactiva_2*
- **Planificador sección 6:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la quinta llamada remota y finaliza cuando se inicia la sexta llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_6,
     Worst_Case_Execution_Time => 2.84e-6,
     Best_Case_Execution_Time   => 9.71e-7,
     Avg_Case_Execution_Time    => 1.406e-6);
```

El servidor de planificación que le corresponderá es *Trajectory_Planning*.

- **Red sección 6.1**: Corresponde con la actividad iniciada con la finalización de la sexta sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Extrae_Error*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Alarmas_Extrae_Error,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Extrae_Error**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Extrae_Error* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 6.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la sexta llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 6.1 y la operación corresponde con *Alarmas_Extrae_Error_2:*

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Alarmas_Extrae_Error_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 7:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la sexta llamada remota y finaliza cuando se inicia la séptima llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_7,
     Worst_Case_Execution_Time => 5.33974e-4,
     Best_Case_Execution_Time  => 1.543e-4,
     Avg_Case_Execution_Time   => 3.11451e-4);
```

El servidor de planificación que le corresponderá es *Trajectory_Planning*.

- **Red sección 7.1**: Corresponde con la actividad iniciada con la finalización de la séptima sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Herramientas_Actua_Auxiliar*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Herramientas_Actua_Auxiliar,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Actua_Auxiliar**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Actua_Auxiliar* y con el servidor de planificación *Planificador_Remote*:

```
Operation(
     Type                       => Simple,
     Name                       => Actua_Auxiliar,
     Worst_Case_Execution_Time  => 4.5236e-5,
     Best_Case_Execution_Time   => 2.123e-5,
     Avg_Case_Execution_Time    => 3.223e-5);
```

  No se poesía de herramienta auxiliar, así que se toma como cota las mediciones de la herramienta *garra*.

- **Red sección 7.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la séptima llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 7.1 y la operación corresponde con *Herramientas_Actua_Auxiliar_2:*

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Herramientas_Actua_Auxiliar_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 8:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la séptima llamada remota y finaliza cuando se inicia la octava llamada remota.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_8,
     Worst_Case_Execution_Time  => 9.51e-7);
```

  El servidor de planificación que le corresponderá es *Trajectory_Planning*. El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 8.1**: Corresponde con la actividad iniciada con la finalización de la octava sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Herramientas_Actua_Pinza*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Herramientas_Actua_Pinza,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Actua_Pinza**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Actua_Pinza* y con el servidor de planificación *Planificador_Remote*:

```
Operation(
     Type                       => Simple,
     Name                       => Actua_Pinza,
     Worst_Case_Execution_Time  => 4.5236e-5,
     Best_Case_Execution_Time   => 2.123e-5,
     Avg_Case_Execution_Time    => 3.223e-5);
```

- **Red sección 8.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la octava llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 8.1 y la operación corresponde con *Herramientas_Actua_Pinza_2:*

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Herramientas_Actua_Pinza_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 9:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la octava llamada remota y finaliza cuando se inicia la novena llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_9,
     Worst_Case_Execution_Time  => 1.302e-4,
     Best_Case_Execution_Time   => 1.003e-4,
     Avg_Case_Execution_Time    => 1.1155e-4);
```

- **Red sección 9.1**: Corresponde con la actividad iniciada con la finalización de la novena sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Lee_Todas_Alarmas*. El servidor de planificación corresponde con *Planificador_RPC*.
- **Lee_Todas_Alarmas**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Todas_alarmas* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 9.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la novena llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 9.1 y la operación corresponde con *Alarmas_Lee_Todas_Alarmas_2.*
- **Planificador sección 10:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la novena llamada remota y finaliza cuando se inicia la decima llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_10,
     Worst_Case_Execution_Time  => 7.2636e-5,
     Best_Case_Execution_Time   => 6.444e-6,
     Avg_Case_Execution_Time    => 2.7181e-5);
```

- **Red sección 10.1**: Corresponde con la actividad iniciada con la finalización de la decima sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Servos_Lee_Errores_Posicion*. El servidor de planificación corresponde con *Planificador_RPC*.
- **Lee_Errores_posicion**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Errores_Posicion* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 10.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la décima llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 10.1 y la operación ejectuada corresponde con *Servos_Lee_Errores_Posicion_2*.
- **Planificador sección 11:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la décima llamada remota y finaliza cuando se inicia la undecima llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_11,
     Worst_Case_Execution_Time  => 5.388e-5,
     Best_Case_Execution_Time   => 7.54e-7,
     Avg_Case_Execution_Time    => 5.308e-6);
```

- **Red sección 11.1**: Corresponde con la actividad iniciada con la finalización de la undécima sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Activa*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                  => Message_Transmission,
     Name                  => Alarmas_Activa,
     Max_Message_Size      => 1492,
     Avg_Message_Size      => 1492,
     Min_Message_size      => 1492);
```

- **Activa**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Activa* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 11.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la undécima llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 11.1 y la operación ejectuada corresponde con *Alarmas_Activa_2*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Alarmas_Activa_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 12:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la undécima llamada remota y finaliza cuando se inicia la duodécima llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_12,
     Worst_Case_Execution_Time  => 9.51e-7);
```

  El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 12.1**: Corresponde con la actividad iniciada con la finalización de la duodécima sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Inserta_Error*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation (
     Type                       => Message_Transmission,
     Name                       => Alarmas_Inserta_Error,
     Max_Message_Size           => 1492,
     Avg_Message_Size           => 1492,
     Min_Message_size           => 1492);
```

- **Inserta_Error**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Inserta_Error* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 12.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la duodécima llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 12.1 y la operación ejectuada corresponde con *Alarmas_Inserta_Error_2*.

```
Operation (
     Type                       => Message_Transmission,
     Name                       => Alarmas_Inserta_Error_2,
     Max_Message_Size           => 1492,
     Avg_Message_Size           => 1492,
     Min_Message_size           => 1492);
```

- **Planificador sección 13:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la duodécima llamada remota y finaliza cuando se inicia la decimotercera llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                        => Simple,
     Name                        => Traject_Section_13,
     Worst_Case_Execution_Time => 9.51e-7);
```

El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 13.1**: Corresponde con la actividad iniciada con la finalización de la decimotercera sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Activa*. El servidor de planificación corresponde con *Planificador_RPC*.
- **Activa**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Activa* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 13.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la decimotercera llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 13.1 y la operación ejectuada corresponde con *Alarmas_Activa_2*.
- **Planificador sección 14:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la decimotercera llamada remota y finaliza cuando se inicia la decimocuarta llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                        => Simple,
     Name                        => Traject_Section_14,
     Worst_Case_Execution_Time => 9.51e-7);
```

El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 14.1**: Corresponde con la actividad iniciada con la finalización de la decimocuarta sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Alarmas_Inserta_Error*. El servidor de planificación corresponde con *Planificador_RPC*.
- **Inserta_Error**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Inserta_Error* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 14.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la decimocuarta llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 14.1 y la operación ejectuada corresponde con *Inserta_Error_2*.
- **Planificador sección 15:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la decimocuarta llamada remota y finaliza cuando se inicia la decimoquinta llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_15,
     Worst_Case_Execution_Time  => 2.533e-6,
     Best_Case_Execution_Time   => 7.84e-7,
     Avg_Case_Execution_Time    => 1.166e-6);
```

- **Red sección 15.1**: Corresponde con la actividad iniciada con la finalización de la decimoquinta sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Servos_Lee_Topes_Superiores*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Servos_Lee_Topes_Superiores,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Lee_Topes_Superiores**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Topes_Superiores* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 15.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la decimoquinta llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 15.1 y la operación corresponde con *Servos_Lee_Topes_Superiores_2:*

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Servos_Lee_Topes_Superiores_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 16:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la decimoquinta llamada remota y finaliza cuando se inicia la decimoquinta llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_16,
     Worst_Case_Execution_Time  => 9.51e-7);
```

  El tiempo es una estimación de dos llamadas consecutivas, se ha tomado como cota el coste de una operación O(1).

- **Red sección 16.1**: Corresponde con la actividad iniciada con la finalización de la decimosexta sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Servos_Lee_Topes_Inferiores*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Servos_Lee_Topes_Inferiores,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Lee_Topes_Inferiores**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Lee_Topes_Inferiores* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 16.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la decimosexta llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 16.1 y la operación corresponde con *Servos_Lee_Topes_Inferiores_2:*

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Servos_Lee_Topes_Inferiores_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 17:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la decimosexta llamada remota y finaliza cuando se inicia la decimoséptima llamada remota. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_17,
     Worst_Case_Execution_Time  => 2.46e-6,
     Best_Case_Execution_Time   => 7.59e-7,
     Avg_Case_Execution_Time    => 9.11e-7);
```

- **Red sección 17.1**: Corresponde con la actividad iniciada con la finalización de la decimoséptima sección del planificador y finalizada cuando el mensaje se haya transmitido a la actividad del recurso de procesado del btmcont. La operación efectuada es *Servos_Escribe_Nuevo_Punto*. El servidor de planificación corresponde con *Planificador_RPC*.

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Servos_Escribe_Nuevo_Punto,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Nuevo_Punto**: Es la actividad que se ejecuta en el procesador remoto (btmcont) corresponde con la operación *Escribe_Nuevo_Punto* y con el servidor de planificación *Planificador_Remote*:
- **Red sección 17.2**: Corresponde con la actividad iniciada con el mensaje de retorno de la decimoséptima llamada remota, y finaliza cuando ha sido recibida correctamente por el planificador. El servidor de planificación es el mismo que en la sección 17.1 y la operación corresponde con *Servos_Escribe_Nuevo_Punto_2:*

```
Operation(
     Type                 => Message_Transmission,
     Name                 => Servos_Escribe_Nuevo_Punto_2,
     Max_Message_Size     => 1492,
     Avg_Message_Size     => 1492,
     Min_Message_size     => 1492);
```

- **Planificador sección 18:** La actividad en btmmmi que se ve iniciada cuando se recibe la finalización de la decimoséptima llamada remota y finaliza con el periodo de la tarea planificadora. El servidor de planificación que le corresponderá es *Trajectory_Planning*.

```
Operation(
     Type                       => Simple,
     Name                       => Traject_Section_18,
     Worst_Case_Execution_Time  => 1.0343e-5,
     Best_Case_Execution_Time   => 7.55e-6,
     Avg_Case_Execution_Time    => 8.538e-6);
```

Para ilustrar de una manera más gráfica la composición de esta transacción se expone la figura 7.8, en donde se muestra mediante un diagrama toda la transacción descrita anteriormente.

Según lo expuesto anteriormente la transacción correspondiente al módulo *planificador_trayectorias* es:

```
Transaction (
     Type                 => Regular,
     Name                 => Trajectory_Planner,
     External_Events=> (
            (Type         => Periodic,
            Name          => In_Trayectoria_01,
            Period        => 0.050)),
     Internal_Events=> (
            (Type         => Regular,
            Name          => Trayectoria_02),
            (Type         => Regular,
            Name          => Trayectoria_03),
            (Type         => Regular,
            Name          => Trayectoria_04),
            (Type         => Regular,
            Name          => Trayectoria_05),
            (Type         => Regular,
            Name          => Trayectoria_06),
            (Type         => Regular,
            Name          => Trayectoria_07),
            (Type         => Regular,
            Name          => Trayectoria_08),
            (Type         => Regular,
            Name          => Trayectoria_09),
            (Type         => Regular,
            Name          => Trayectoria_10),
            (Type         => Regular,
            Name          => Trayectoria_11),
            (Type         => Regular,
            Name          => Trayectoria_12),
            (Type         => Regular,
            Name          => Trayectoria_13),
```

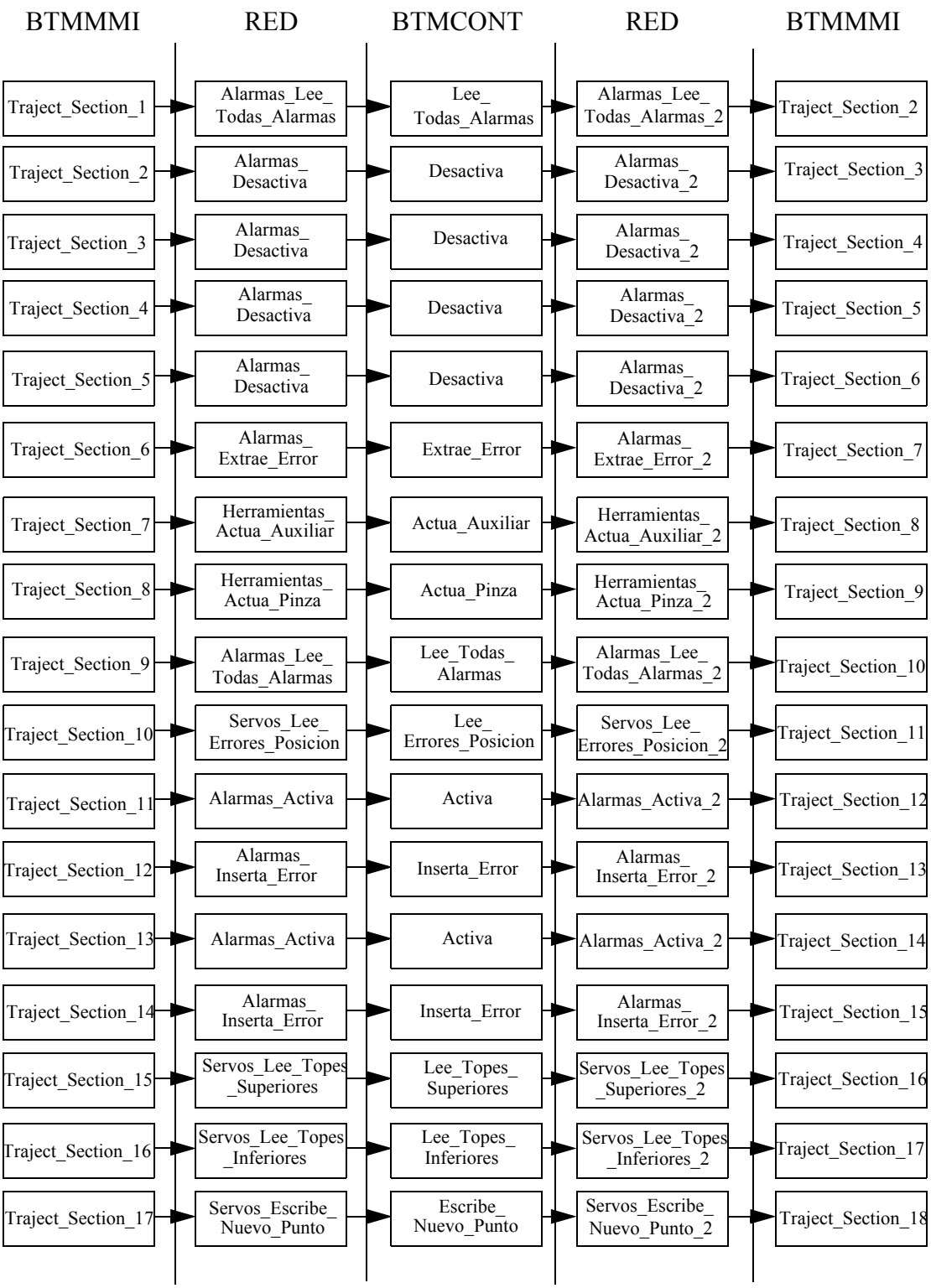



**Figura 7.8: Detalle de la transacción Trajectory_Planner**

```
(Type          => Regular,
Name           => Trayectoria_14),
(Type          => Regular,
Name           => Trayectoria_15),
(Type          => Regular,
Name           => Trayectoria_16),
(Type          => Regular,
Name           => Trayectoria_17),
(Type          => Regular,
Name           => Trayectoria_18),
(Type          => Regular,
Name           => Trayectoria_19),
(Type          => Regular,
Name           => Trayectoria_20),
(Type          => Regular,
Name           => Trayectoria_21),
(Type          => Regular,
Name           => Trayectoria_22),
(Type          => Regular,
Name           => Trayectoria_23),
(Type          => Regular,
Name           => Trayectoria_24),
(Type          => Regular,
Name           => Trayectoria_25),
(Type          => Regular,
Name           => Trayectoria_26),
(Type          => Regular,
Name           => Trayectoria_27),
(Type          => Regular,
Name           => Trayectoria_28),
(Type          => Regular,
Name           => Trayectoria_29),
(Type          => Regular,
Name           => Trayectoria_30),
(Type          => Regular,
Name           => Trayectoria_31),
(Type          => Regular,
Name           => Trayectoria_32),
(Type          => Regular,
Name           => Trayectoria_33),
(Type          => Regular,
Name           => Trayectoria_34),
(Type          => Regular,
Name           => Trayectoria_35),
(Type          => Regular,
Name           => Trayectoria_36),
(Type          => Regular,
Name           => Trayectoria_37),
(Type          => Regular,
Name           => Trayectoria_38),
(Type          => Regular,
Name           => Trayectoria_39),
(Type          => Regular,
Name           => Trayectoria_40),
(Type          => Regular,
Name           => Trayectoria_41),
(Type          => Regular,
Name           => Trayectoria_42),
```

```
(Type         => Regular,
Name          => Trayectoria_43),
(Type         => Regular,
Name          => Trayectoria_44),
(Type         => Regular,
Name          => Trayectoria_45),
(Type         => Regular,
Name          => Trayectoria_46),
(Type         => Regular,
Name          => Trayectoria_47),
(Type         => Regular,
Name          => Trayectoria_48),
(Type         => Regular,
Name          => Trayectoria_49),
(Type         => Regular,
Name          => Trayectoria_50),
(Type         => Regular,
Name          => Trayectoria_51),
(Type         => Regular,
Name          => Trayectoria_52),
(Type         => Regular,
Name          => Trayectoria_53),
(Type         => Regular,
Name          => Trayectoria_54),
(Type         => Regular,
Name          => Trayectoria_55),
(Type         => Regular,
Name          => Trayectoria_56),
(Type         => Regular,
Name          => Trayectoria_57),
(Type         => Regular,
Name          => Trayectoria_58),
(Type         => Regular,
Name          => Trayectoria_59),
(Type         => Regular,
Name          => Trayectoria_60),
(Type         => Regular,
Name          => Trayectoria_61),
(Type         => Regular,
Name          => Trayectoria_62),
(Type         => Regular,
Name          => Trayectoria_63),
(Type         => Regular,
Name          => Trayectoria_64),
(Type         => Regular,
Name          => Trayectoria_65),
(Type         => Regular,
Name          => Trayectoria_66),
(Type         => Regular,
Name          => Trayectoria_67),
(Type         => Regular,
Name          => Trayectoria_68),
(Type         => Regular,
Name          => Trayectoria_69),
(Type         => Regular,
Name          => Trayectoria_Out,
Timing_Requirements => (
     Type     => Hard_Global_Deadline,
```

```
                Deadline  => 0.050,
                Referenced_Event => In_Trayectoria_01))),
Event_Handlers=>(
       (Type              => Activity,
       Input_Event        => In_Trayectoria_01,
       Output_event       => Trayectoria_02,
       Activity_Operation => Traject_Section_1,
       Activity_Server    => Trajectory_Planning),
       (Type              => Activity,
       Input_Event        => Trayectoria_02,
       Output_event       => Trayectoria_03,
       Activity_Operation =>
                     Alarmas_Lee_Todas_Alarmas,
       Activity_Server    => Planificador_RPC),
       (Type              => Activity,
       Input_Event        => Trayectoria_03,
       Output_event       => Trayectoria_04,
       Activity_Operation => Lee_Todas_Alarmas,
       Activity_Server    => Planificador_Remote),
       (Type              => Activity,
       Input_Event        => Trayectoria_04,
       Output_event       => Trayectoria_05,
       Activity_Operation =>
                     Alarmas_Lee_Todas_Alarmas_2,
       Activity_Server    => Planificador_RPC),

       (Type              => Activity,
       Input_Event        => Trayectoria_05,
       Output_event       => Trayectoria_06,
       Activity_Operation => Traject_Section_2,
       Activity_Server    => Trajectory_Planning),
       (Type              => Activity,
       Input_Event        => Trayectoria_06,
       Output_event       => Trayectoria_07,
       Activity_Operation => Alarmas_Desactiva,
       Activity_Server    => Planificador_RPC),
       (Type              => Activity,
       Input_Event        => Trayectoria_07,
       Output_event       => Trayectoria_08,
       Activity_Operation => Desactiva,
       Activity_Server    => Planificador_Remote),
       (Type              => Activity,
       Input_Event        => Trayectoria_08,
       Output_event       => Trayectoria_09,
       Activity_Operation => Alarmas_Desactiva_2,
       Activity_Server    => Planificador_RPC),

       (Type              => Activity,
       Input_Event        => Trayectoria_09,
       Output_event       => Trayectoria_10,
       Activity_Operation => Traject_Section_3,
       Activity_Server    => Trajectory_Planning),
       (Type              => Activity,
       Input_Event        => Trayectoria_10,
       Output_event       => Trayectoria_11,
       Activity_Operation => Alarmas_Desactiva,
       Activity_Server    => Planificador_RPC),
       (Type              => Activity,
```

```
Input_Event        => Trayectoria_11,
Output_event       => Trayectoria_12,
Activity_Operation => Desactiva,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_12,
Output_event       => Trayectoria_13,
Activity_Operation => Alarmas_Desactiva_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_13,
Output_event       => Trayectoria_14,
Activity_Operation => Traject_Section_4,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_14,
Output_event       => Trayectoria_15,
Activity_Operation => Alarmas_Desactiva,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_15,
Output_event       => Trayectoria_16,
Activity_Operation => Desactiva,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_16,
Output_event       => Trayectoria_17,
Activity_Operation => Alarmas_Desactiva_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_17,
Output_event       => Trayectoria_18,
Activity_Operation => Traject_Section_5,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_18,
Output_event       => Trayectoria_19,
Activity_Operation => Alarmas_Desactiva,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_19,
Output_event       => Trayectoria_20,
Activity_Operation => Desactiva,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_20,
Output_event       => Trayectoria_21,
Activity_Operation => Alarmas_Desactiva_2,
Activity_Server    => Planificador_RPC),
```

```
(Type              => Activity,
Input_Event        => Trayectoria_21,
Output_event       => Trayectoria_22,
Activity_Operation => Traject_Section_6,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_22,
Output_event       => Trayectoria_23,
Activity_Operation => Alarmas_Extrae_Error,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_23,
Output_event       => Trayectoria_24,
Activity_Operation => Extrae_Error,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_24,
Output_event       => Trayectoria_25,
Activity_Operation => Alarmas_Extrae_Error_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_25,
Output_event       => Trayectoria_26,
Activity_Operation => Traject_Section_7,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_26,
Output_event       => Trayectoria_27,
Activity_Operation =>
              Herramienta_Actua_Auxiliar,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_27,
Output_event       => Trayectoria_28,
Activity_Operation => Actua_Auxiliar,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_28,
Output_event       => Trayectoria_29,
Activity_Operation =>
              Herramienta_Actua_Auxiliar_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_29,
Output_event       => Trayectoria_30,
Activity_Operation => Traject_Section_8,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_30,
Output_event       => Trayectoria_31,
Activity_Operation => Herramientas_Actua_Pinza,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_31,
```

```
Output_event       => Trayectoria_32,
Activity_Operation => Actua_Pinza,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_32,
Output_event       => Trayectoria_33,
Activity_Operation =>
              Herramientas_actua_Pinza_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_33,
Output_event       => Trayectoria_34,
Activity_Operation => Traject_Section_9,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_34,
Output_event       => Trayectoria_35,
Activity_Operation =>
              Alarmas_Lee_Todas_Alarmas,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_35,
Output_event       => Trayectoria_36,
Activity_Operation => Lee_Todas_Alarmas,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_36,
Output_event       => Trayectoria_37,
Activity_Operation =>
              Alarmas_Lee_Todas_Alarmas_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_37,
Output_event       => Trayectoria_38,
Activity_Operation => Traject_Section_10,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_38,
Output_event       => Trayectoria_39,
Activity_Operation =>
              Servos_Lee_Errores_Posicion,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_39,
Output_event       => Trayectoria_40,
Activity_Operation => Lee_Errores_Posicion,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_40,
Output_event       => Trayectoria_41,
Activity_Operation =>
              Servos_Lee_Errores_Posicion_2,
Activity_Server    => Planificador_RPC),
```

```
(Type              => Activity,
Input_Event        => Trayectoria_41,
Output_event       => Trayectoria_42,
Activity_Operation => Traject_Section_11,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_42,
Output_event       => Trayectoria_43,
Activity_Operation => Alarmas_Activa,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_43,
Output_event       => Trayectoria_44,
Activity_Operation => Activa,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_44,
Output_event       => Trayectoria_45,
Activity_Operation => Alarmas_Activa_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_45,
Output_event       => Trayectoria_46,
Activity_Operation => Traject_Section_12,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_46,
Output_event       => Trayectoria_47,
Activity_Operation => Alarmas_Inserta_Error,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_47,
Output_event       => Trayectoria_48,
Activity_Operation => Inserta_Error,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_48,
Output_event       => Trayectoria_49,
Activity_Operation => Alarmas_Inserta_Error_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_49,
Output_event       => Trayectoria_50,
Activity_Operation => Traject_Section_13,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_50,
Output_event       => Trayectoria_51,
Activity_Operation => Alarmas_Activa,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
```

```
Input_Event        => Trayectoria_51,
Output_event       => Trayectoria_52,
Activity_Operation => Activa,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_52,
Output_event       => Trayectoria_53,
Activity_Operation => Alarmas_Activa_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_53,
Output_event       => Trayectoria_54,
Activity_Operation => Traject_Section_14,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_54,
Output_event       => Trayectoria_55,
Activity_Operation => Alarmas_Inserta_Error,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_55,
Output_event       => Trayectoria_56,
Activity_Operation => Inserta_Error,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_56,
Output_event       => Trayectoria_57,
Activity_Operation => Alarmas_Inserta_Error_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_57,
Output_event       => Trayectoria_58,
Activity_Operation => Traject_Section_15,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_58,
Output_event       => Trayectoria_59,
Activity_Operation =>
              Servos_Lee_Topes_Superiores,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_59,
Output_event       => Trayectoria_60,
Activity_Operation => Lee_Topes_Superiores,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_60,
Output_event       => Trayectoria_61,
Activity_Operation =>
              Servos_Lee_Topes_Superiores_2,
Activity_Server    => Planificador_RPC),
```

```
(Type              => Activity,
Input_Event        => Trayectoria_61,
Output_event       => Trayectoria_62,
Activity_Operation => Traject_Section_16,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_62,
Output_event       => Trayectoria_63,
Activity_Operation =>
              Servos_Lee_Topes_Inferiores,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_63,
Output_event       => Trayectoria_64,
Activity_Operation => Lee_Topes_Inferiores,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_64,
Output_event       => Trayectoria_65,
Activity_Operation =>
              Servos_Lee_Topes_Inferiores_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_65,
Output_event       => Trayectoria_66,
Activity_Operation => Traject_Section_17,
Activity_Server    => Trajectory_Planning),
(Type              => Activity,
Input_Event        => Trayectoria_66,
Output_event       => Trayectoria_67,
Activity_Operation =>
              Servos_Escribe_Nuevo_Punto,
Activity_Server    => Planificador_RPC),
(Type              => Activity,
Input_Event        => Trayectoria_67,
Output_event       => Trayectoria_68,
Activity_Operation => Escribe_Nuevo_Punto,
Activity_Server    => Planificador_Remote),
(Type              => Activity,
Input_Event        => Trayectoria_68,
Output_event       => Trayectoria_69,
Activity_Operation =>
              Servos_Escribe_Nuevo_Punto_2,
Activity_Server    => Planificador_RPC),

(Type              => Activity,
Input_Event        => Trayectoria_69,
Output_event       => Trayectoria_Out,
Activity_Operation => Traject_Section_18,
Activity_Server    => Trajectory_Planning)));
```

Como se ha podido constatar a lo largo de este capítulo hemos obtenido un preciso modelo temporal tanto del protocolo como de la aplicación demostradora. Su cuantificación nos ha permitido realizar un análisis del conjunto para verificar su comportamiento de tiempo real.

Como mejora a la arquitectura distribuida se puede añadir una operación en un módulo aparte que se encarge de recoger todos los datos que necesita el módulo reportero para que cuando sea necesario pueda ser interrgado por éste.

# 8. 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.
- 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 temporal
- 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.
- Hemos caracterizado parámetros muy útiles para el implementador, como puede ser la utilización de CPU por parte del protocolo y el régimen binario efectivo del protocolo. Estos valores son configurables dentro del protocolo, con las limitaciones derivadas de su implementación.

- Se ha implementado y analizado una aplicación distribuida a modo de demostrador. Comprobando la validez del protocolo.
- 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.
- 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 porporcionado una interfaz del protocolo en C y Ada por lo que puede ser utilizado por distintas aplicaciones implementadas en dichos lenguajes.
- 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.
- Se ha realizado una comparativa con un bus de tiempo real en la que se pone de manifiesto que RT-EP ofrece mejores tiempos de transmisión para un número bajo o medio de estaciones y que sus prestaciones se degradan a medida que aumenta su número. 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. Aunque para mensajes de tamaño medio o alto, RT-EP es superior, incluso con un número elevado de estaciones, esto es debido a que el troceo de mensajes incide seriamente en el rendimiento de CAN.
- Es un protocolo cuyo rendimiento depende de la capacidad de procesado y de la velocidad de la red Ethernet. Ambos factores se ven favorecidos con el avance tecnológico lo que hace que, indirectamente, con el paso del tiempo, el protocolo obtenga mejores tiempos de respuesta.

# 9. Trabajo futuro

## 9.1. 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.

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

## 9.2. 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 16, 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 de la 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.

## 9.3. 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 16, 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.

La utilización de un puente software puede ser aplicada al caso de conectar una red RT-EP con otra red de tiempo real (CAN, PROFIBUS, etc..) o no tiempo real (Ethernet, X.25, etc..). Configuraríamos una estación con dos interfaces de red, para RT-EP y para la otra red, y de un software de ensamblado/desembamblado de paquetes que realize la labor de conexión entre ambas redes. Así es posible mandar mensajes de una red a otra conservando el comportamiento de RT-EP. La topología de la red propuesta se puede observar en la figura 9.1.

## 9.4. 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

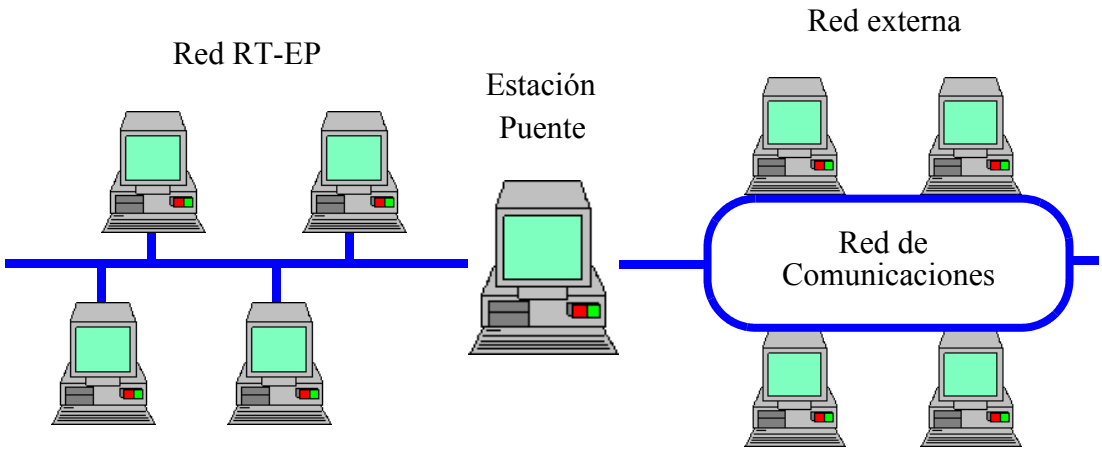



**Figura 9.1: Interconexión con red RT-EP**

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 [17]. 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 [17].

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 [17]. 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 apartado 9.3, “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 apartado 9.3, “Interconexión de redes.”

Como los fallos, en estos entornos, pueden ser más frecuentes, habría que aumentar las retransmisiones. Convendría además añadir la posibilidad de reincorporar a la comunicación una estación que según la lógica del protocolo hubiese fallado.

# 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] Etherboot Project. http://etherboot.sourceforge.net

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

[14] 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

[15] IEEE Std 802.3, 2002 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”

[16] 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”.

[17] 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”

[18] 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.

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

[20] 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

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

[22] López Campos, J. Gutiérrez, J.J. González Harbour, M. “The Chance for Ada to Support Distribution and Real-Time in Embedded Systems”. Intl. Conference on Reliable Software Technologies, Ada-Europe-2004, Palma de Mallorca, Spain, June 2004.

[23] 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

[24] 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

[25] 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.

[26] Martínez, J.M. González Harbour, M. “Diseño, implementación y modelado de un protocolo multipunto de comunicaciones para aplicaciones de tiempo real distribuidas y analizables sobre Ethernet. (RT-EP)”. Proyecto Fin de Carrera. Ingeniería de Telecomunicación. U. Cantabria. Septiembre 2002

[27] Martínez, J.M. González Harbour, M. Gutiérrez, J.J. "RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimun Real-Time POSIX Kernel". 2st Intl. Workshop on Real-Time LANs in the Internet Age. (RTLIA 2003). Porto, Portugal. Julio 2003.

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

[29] Pautet, L. Tardieu, S. "Inside the Distributed Systems Annex". Proceeding of the Intl. Conf. on Reliable Software Technologies, Ada-Europe'98, Uppsala, Sweden, in LNCS 1411, Springer, pp. 65-77, June 1998.

[30] Pautet, L. Tardieu, S. "GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems". Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'00), Newport Beach, USA, March 2000.

[31] PCI Special Interest Group. PCI Local Bus Specification, December 1995. Revision 2.2.

[32] 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.

[33] 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.

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

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

[36] Shimokawa, Y. Shiobara, Y. "Real-Time Ethernet for Industrial Applications" IECON'85, pp.829-834. 1985

[37] Silberschatz, A. Galvin, P. "Operating System Concepts" Addison-Wesley, 1998

[38] Stallings, W. "Local & Metropolitan Area Networks" Prentice Hall 5º Edicion 1990

[39] Tanenbaum, A.S. "Computers Networks". Prentice-Hall, Inc. 1997

[40] Tindell K., Hansson, H. Wellings, A. "Analysing Real-Time Communications: Controller Area Network (CAN)". Proceedings of the IEEE Real-Time Systems Symposium 1994, pp. 259-263

[41] Tindell K., Hansson, H. Wellings, A. "Calculating Controller Area Network (CAN) Message Response Time". Control Engineering Practice, Vol. 3, Nº 8, pp. 1163-1169

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

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

[44] 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

[45] Tucker Taft, S. Duff R.A. (Eds.). “Ada 95 Reference Manual. Language and Standard Libraries”. International Standard ISO/IEC 8652:1995(E), in LNCS 1246, Springer, 1997.

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

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