RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet

José María Martínez and Michael González Harbour

Departamento de Electrónica y Computadores
Universidad de Cantabria, 39005 - Santander, SPAIN
{martinjm,mgh}@unican.es

Abstract. This paper presents the design and implementation of RT-EP (Real-Time Ethernet Protocol), which is a software-based token-passing Ethernet protocol for multipoint communications in real-time applications, that does not require any modification to existing Ethernet hardware. The protocol allows a fixed priority to be assigned to each message, and consequently well-known schedulability analysis techniques can be applied. A precise model of its timing behavior has been obtained. Furthermore, this protocol provides the ability of recovering from some fault conditions. It has been ported to an implementation of the Minimal Real-Time POSIX standard called MaRTE OS [10], and is being used to support real-time communications in an implementation of Ada's Distributed Systems Annex (RT-GLADE). It has been successfully used to implement a distributed controlled for an industrial robot.

Keywords: Real-Time, Embedded Systems, Networks, Ethernet, Distributed Systems, Ada 95, Modelling, Schedulability.

1 Introduction[1]

Ethernet is by far the most widely used local area networking (LAN) technology in the world today, even though it has unpredictable transmission times because it uses a non-deterministic arbitration mechanism (CSMA/CD). Several approaches and techniques have been used to make Ethernet deterministic in order to take advantage of its low cost and higher speeds than those of real-time field buses available today (like the CAN bus [9], for example). Some of these approaches are the modification of the Medium Access Control [7], the addition of transmission control [2], a protocol using time-triggered traffic [14], or the usage of a switched Ethernet [3].

The objective of this work is to achieve a relatively high speed mechanism for real-time communications at a low cost, while keeping the predictable timing behaviour required in distributed hard real-time applications. The communication protocol proposed in this work is called RT-EP (Real-Time Ethernet Protocol), and can be classified as an addition of a transmission control layer over Ethernet, since it is basically a token-passing arbitration in a bus [6], capable of transmitting fixed-priority messages.

---

1. This work has been funded in part by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) of the Spanish Government under grant number TIC2002-04123-C03-02 (TRECOM), and by the IST Programme of the European Commission under project IST-2001-34820 (FIRST).

It provides a Real-Time Ethernet communication without modifying the existing hardware, and has been designed to avoid collisions in Ethernet media.

In a previous work we presented a preliminary version of RT-EP [11] in which we discussed the fault-recovery mechanism and the way in which the protocol can be modelled using the MAST [12] Real-time Modelling and Analysis Suite. We also discussed the implementation, in C language, of RT-EP in MaRTE OS [10] and we also measured the overheads it introduces.

In this paper we present a complete description of the protocol and we propose the Ada language for its implementation, to ensure a smooth integration with DSA middleware such as RT-GLADE [8], or Polyorb [15] in the future. By porting the protocol to Ada we can also take advantage of the reliability and robustness that the Ada language provides, and we open the door to a possible future extension to support real-time distribution in the Ravescar profile [1]. We also have done a clean-up of the protocol, we have generalized the architecture to make it independent of the underlying communications network, and we have developed a more precise real-time model to be used with MAST, with measurements performed on a MaRTE OS platform. The RT-EP protocol is being used to support real-time communications in an implementation of Ada's Distributed Systems Annex (DSA) called RT-GLADE. It has also been successfully used to implement a distributed controller for an industrial robot.

The paper is organized as follows. Section 2 introduces how the protocol works. In Section 3 we describe the state diagram that controls the behaviour of the protocol. In Section 4 we explain the layout of the packets used in the protocol. Section 5 discusses some details of the current implementation. Section 6 gives some details about the MAST model that describes the timing behaviour of the protocol. In Section 7 we provide some results of the metrics taken in a particular platform. Section 8 briefly describes the robot system that has been implemented with RT-EP. Finally, Section 9 gives our conclusions.

2 Description of the Communication Protocol

RT-EP has been designed to avoid collisions in the Ethernet media by the use of a token. It implements an adaptation layer over the Medium Access Control in Ethernet. The protocol is used to transmit messages that have a fixed priority assigned to them. It works by dividing the message transmission process into two phases: a priority arbitration phase in which the highest priority message to be transmitted is determined, and the transmission of the message itself. With this mechanism, the protocol does not require any clock synchronization mechanism to keep a synchronized notion of time in all the nodes.

Messages size is limited by the parameter RT-EP_MTU (RT-EP Maximum Transmission Unit) which depends on the MTU of the underlying protocol (1492 bytes for Ethernet). Message packets are non preemptible, and therefore there is a bounded blocking time that has to be taken into account during the analysis. We do not provide fragmentation of messages at this layer, although it would be easy to add. In that case

large messages would be partitioned into several packets, and preemption would be possible up to the packet-size limit.

Each station (processing node or CPU) has a transmission priority queue, in which all the packets to be transmitted are stored in priority order. Each station also has a set of reception priority queues that provide the equivalent of virtual separate channels for sending messages to the desired recipient. Packets with the same priority are stored in FIFO order. The number of reception queues can be configured depending on the number of application threads (or tasks) running in the system and requiring reception of messages. The common usage model is that each application task has its own channel or reception queue reserved for itself, and in that way the sender can send messages addressed to that specific task, through the desired channel. Channels are identified with a number called the channel ID.

The network is organized as a logical ring over a bus, as shown in Fig. 1. The ring is configured statically, and every station has access to the full ring configuration, including topological information with the successor of every station, so that the logical ring can be built. The protocol works by rotating a token in this logical ring. The token holds information about the station having the highest priority packet to be transmitted and its priority value.

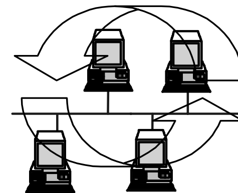

Fig. 1 Logical ring

For the transmission of one message, an arbitrary station is designated as the *token_master*. During the priority-arbitration phase the token travels through the whole ring, visiting all the nodes, in a type of packet called a *Regular Token*. Each station checks the information in the token to determine if one of its own packets has a priority higher than the priority carried by the token. In that case, it changes the highest priority station and associated priority in the token information; otherwise the token is left unchanged. Then, the token is sent to the successor station. This process is followed until the token arrives back at the *token_master* station, finishing the arbitration phase.

In the message-transmission phase the *token_master* station sends a packet with a token of a special type called a *Transmit Token*, addressed to the station with the highest priority message, which then sends its message in a packet of the type *Info Packet*. The receiving station becomes the new *token_master* station.

The actual behaviour of the protocol is a bit more complex because it is designed to tolerate some faults. Otherwise, the loss of a token, for example, would cause communications to be stopped. We have considered three possible faults to be handled by the protocol:

- *Failure of a Station:* A reconfiguration of the ring is performed to leave the failing station out of the ring.
- *Loss of a packet:* A retransmission takes place after a configurable timeout.
- *Busy station* (a station that takes too long to respond): A retransmission will occur, but if it was not caused by a packet loss a duplicate packet would be generated. We need to trash such duplicate packets.

The real-time behaviour is only guaranteed in case of the loss of a packet. A busy station error is a consequence of bad system design, and should be detected and corrected at system design time. The failure of a station is a major fault that would require fault-tolerant techniques with redundant software and/or hardware, which are outside the scope of our work. Therefore, we provide some level of recovery from this error, but real-time response is not guaranteed during the recovery process.

The recovery method for these errors is based on simultaneous listening to the media by all the stations, in a promiscuous mode. Each station, after sending a packet, listens to the media for an *acknowledge*, which is the correct transmission of the next frame by the receiving station. If no acknowledge is received after some specified *timeout*, the station assumes that the packet is lost and retransmits it. The station repeats this process until an acknowledge is received or a specified number or retransmissions is produced. In the latter case the receiving station is considered as a *failing station* and will be excluded from the logical ring. Because retransmission opens the door to duplicate packets if a station does not respond in time, a *sequence number* is used to discard duplicates at the receiving end.

3 RT-EP as a State Machine

RT-EP can be described as a state machine for each station, in order to understand its functionality and to identify operations involved in the timing model. Fig. 2 shows the states and the transitions among them using UML notation [13], which is based on Harel's statecharts [4]. The description of the different states is as follows:

- *Offline*. It is the initial state reached during configuration time. Each station reads the configuration describing the logical ring and gets configured as one of its stations. The station configured as the initial *token_master* (*isTokenMaster* = True) performs a *Send_Initial_Token* action and then enter the *Error_Check* state. The others are set to the *Idle* state.
- *Idle*. In this state, the station listens for the arrival of any packet, discarding possible duplicates. In addition, due to the promiscuous mode (every station listens to every packet), it discards all the packets not specifically addressed to this station. When a non-discarded packet is received, a check is made to determine its type: if it is an *Info_Packet* (*Message Reception* event) the station stores the message and performs a *Send_Initial_Token* action; if it is a *Token Packet* (either a *Token Reception* or a *Transmit Token Reception* event), three different actions can be performed: *Send_Info* (if a *Transmit Token* is received or if the *token_master* receives a regular token and it has the highest priority message to transmit), *Send_Token* (if a regular token is received and the station is not the current *token_master*), or *Send_Permission* (when the *token_master* receives a regular token and is not the station with the highest priority message).
- *Error Check*. The station starts listening to the media after transmitting a frame. If a correct packet is detected the station switches to the *Idle* state. If not, after a configurable timeout, the station tries to recover from the error by resending the last packet. If the number of resent packets exceeds a configurable maximum, the

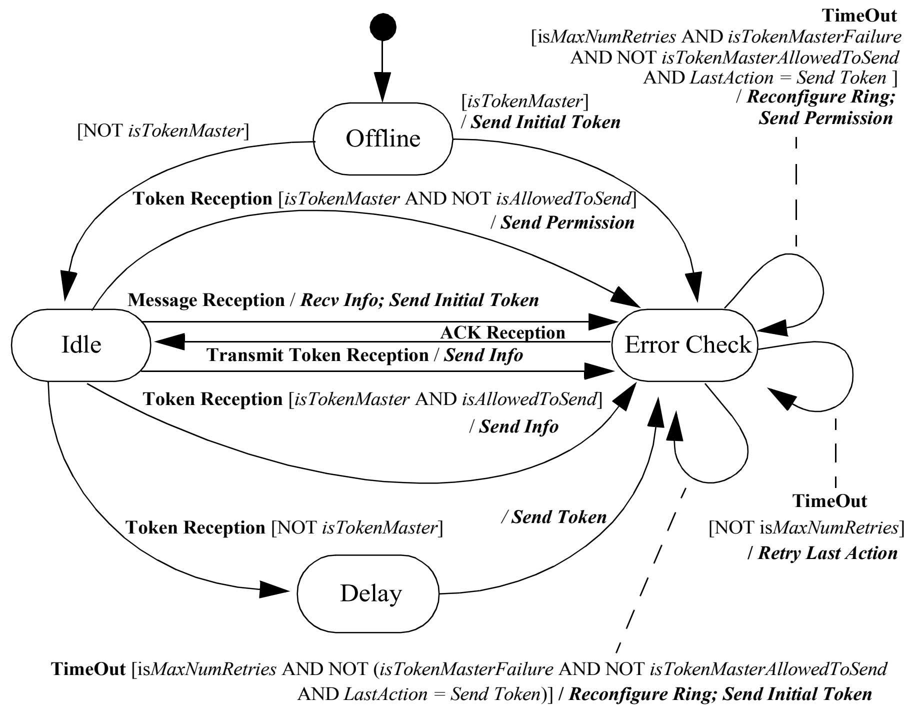

Fig. 2 RT-EP state machine

successor station is considered to have failed. The failing station identifier is transmitted in the token and is erased from the ring in all the stations. No subsequent info message transmission is allowed for that station

- *Delay*. This state represents a delay when sending the token. It is introduced to reduce the overhead incurred by the execution of the protocol operations in the different stations. Its interval is a configurable parameter inside the protocol, with which we can trade message latency against processor overhead.

The actions that appear in the state machine are:

- *Send_Initial_Token*. The station performing this action becomes the current *token_master*. A token is sent to the successor station with the highest of the priorities of the messages pending in the transmission queue, if any. Then, the station switches to the *Error_Check* state.
- *Send_Token*. The station compares the priority of the token with the highest priority element on its transmission queue, updates the token with its own priority if it is higher, and sends the token to next station. Then, it switches to the *Error_Check* state.

- *Send_Permission*. The *token_master* role is lost and a *Transmit Token* is built and sent to the *Station Address* recorded in the last received token, which is the station with the highest priority message. Then, the *Error_Check* state is reached.
- *Send_Info*. This is the action performed when a station has the highest priority packet in the ring and is allowed to transmit it. Then it switches to the *Error_Check* state.
- *Recv_Info*. The information received inside the *Info_Packet* is written into the appropriate reception queue and the station performs a *Send_Initial_Token* action, becoming the new *token_master*.

4 RT-EP Packets

RT-EP packets are carried inside the *Data* field of the lower-layer protocol. We are currently using Ethernet for the lower layer, but it can be easily replaced by another protocol. Ethernet II has the following structure [5]:

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

The *Type* field identifies what type of high-level network protocol is being carried in the data field. We use a value of 0x1000 for the *Type* field, which represents an unused number protocol that could be changed if the protocol is registered in the future.

The maximum message size of Ethernet is 1500 bytes and therefore the RT-EP_MTU parameter is 1492 bytes (Ethernet MTU minus the RT-EP info packet header). The protocol packets are carried inside the *Data* field of the Ethernet frame, which must be at least 46 bytes long. Due to this restriction, if the packet to be sent is less than 46 bytes long, it is padded with zeros to build a 46 bytes packet. Our protocol has two types of packets:

- *Token Packet*: it is used to transmit the tokens and has the following structure:

1 byte	1 byte	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes
Packet Identifier	*Priority*	*Packet Number*	*Token Master ID*	*Failing Station*	*Failing Station ID*	*Station ID*

The *Packet Identifier* field is present also in the *Info Packet* and is used to identify the type of the packet. It can hold two different values for this type of packet:

- *Regular Token*, used in the arbitration phase to find the highest priority packet
- *Transmit Token*, which grants permission to transmit a message

The *Priority* indicates the highest priority element in the ring, found during the rotation. The *Packet Number* is used as a sequence number to eliminate duplicate packets. Each station transmits the received *Packet Number* incremented by one.

Token Master ID stores the identification of the current *token_master* station, which is needed to recover from a *token_master* failure. *Failing Station* indicates whether there is a failing station or not, and *Failing Station ID* specifies which station, if any. By including this information in the token it is possible for the stations in the ring to remove the failing station from their local configuration, discarding any further messages to this station. The *Station ID* stores the station with the highest priority packet.

- *Info Packet*: it is used to transmit data and has the following structure:

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

The *Packet Identifier* has a value that identifies it as an *Info Packet*. The *Priority* field holds the priority of the packet being transmitted. As well as in the *Token Packet*, *Packet Number* is the sequence number. The *Channel ID* is used to identify the destination queue in the destination station. The *Info Length* is the size of the data stored in the *Info* field. In Ethernet, if the information to be transmitted is less than 38 bytes long, padding is performed in order to get the 46 data bytes required in an Ethernet frame.

5 RT-EP Implementation

The functionality and architecture of the communication protocol are shown in Fig. 3. This protocol offers four primitives to any application or middleware using the network: *Init_Comm* (to initialize the network), *Send_Info* (to send a message), *Receive_Info* (to receive a message), and *Try_Receive_Info* (a non-blocking version of receive message). To send a message it is necessary to encapsulate the information in a message type, which is used both for transmission and reception. This message type contains the destination station address, the destination channel ID, the priority of the message and the application data.

Inside RT-EP there is only one task, the *Main Communications Task,* that is responsible of reading the packets from the transmission queue and of writing the received packets into the reception queues. Usually this task executes at a priority level higher than that of the application tasks, and its overhead is modelled as extra load using the model described in Section 6. In addition, there is an interrupt service routine (ISR) that handles incoming packets, and awakens the internal task if necessary.

We have attempted to make the protocol as system independent as possible. To achieve this we have made separate modules that deal with the system, the Ethernet frames and the network drivers. If someone needs to port the protocol to another communications subsystem (for instance, on top of UDP), only the modules under the adaptation layer need to be changed. The protocol has a clear interface with functions to identify the stations and to construct the frame that is going to be sent through the network.

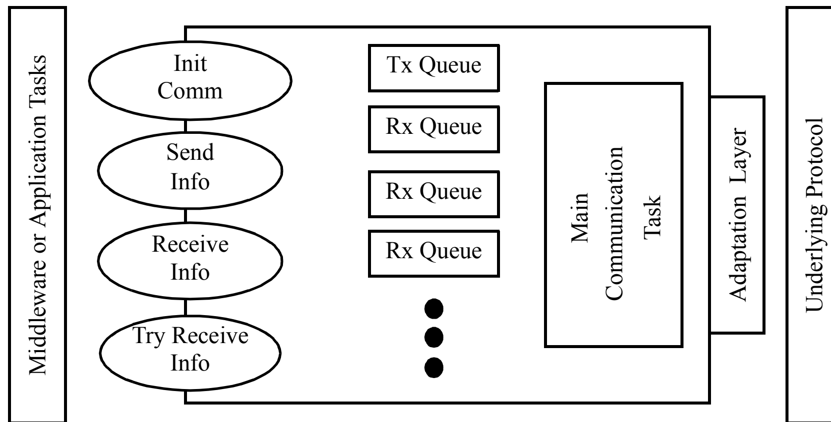

Fig. 3 Functionality and architecture of RT-EP

We have provided Ada and C interfaces to the protocol. We will now show the Ada interfaces for the application and the adaptation layer.

5.1. Ada Application Interface

The main primitives to be used by an Ada application or middleware are the following:

- *Init_Comm*: When initializing the protocol, the Main Communication Task is started. The priority of this special task has to be configured at compilation time and it has to be higher than that of the tasks that are communicating through the network:

```
procedure Init_Comm;
```

- *Send_Info*: To implement this primitive two operations are provided:
  - Generic *Send_Info*:

```
generic
      type Data_Type is private;
procedure Generic_Send_Info
      (Destination_Station_ID  : in Station_ID;
       Channel_ID              : in Channel;
       Data                    : in Data_Type;
       Data_Priority           : in Priority);
```

  - Stream-based *Send_Info*:

```
procedure Send_Info
      (Destination_Station_ID  : in Station_ID;
       Channel_ID              : in Channel;
       Data                    : in Stream_Element_Array;
       Data_Priority           : in Priority);
```

- *Receive_Info*: In the same way as in *Send_Info*, we provide two sets of operations: generic, and stream-based. To save space we only show the blocking receive procedures, but similar non-blocking procedures exist.

  - Generic *Receive_Info* (blocking):

```
generic
      type Data_type is private;
procedure Generic_Recv_Info
      (Source_Address           : out Station_ID;
       Channel_ID               : in Channel;
       Data                     : out Data_Type;
       Data_Priority            : out Priority);
```

  - Stream-based *Receive_Info* (blocking):

```
procedure Recv_Info
      (Source_Address           : out Station_ID;
       Channel_ID               : in Channel;
       Data                     : out Stream_Element_Array;
       Last                     : out Stream_Element_Offset;
       Data_Priority            : out Priority);
```

5.2. Adaptation layer.

This API must be implemented for the specific underlying protocol:

- **Set_Promiscuous_Mode**: Sets the promiscuous mode in the system. The protocol has to be able to receive all the frames transmitted in the media.

```
procedure Set_Promiscuous_Mode;
```

- **Open_RTEP_Comm**: Makes the required initialization to be able to read/ write from/to the media. Further calls to this function will have no effect.

```
procedure Open_RTEP_Comm;
```

- **Close_RTEP_Comm**: Closes the media.

```
procedure Close_RTEP_Comm;
```

- **Send_RTEP_Packet**: Sends the `RT_EP_Packet` to the `Dest_Station`

```
procedure Send_RTEP_Packet
      (Dest_Station             : in Station_ID;
       RT_EP_Packet             : in Stream_Element_Array);
```

- **Recv_RTEP_Packet**: Returns a packet addressed to `Dest_Station`.

```
procedure Recv_RTEP_Packet
      (Dest_Station             : out Station_ID;
       Source_Station           : out Station_ID;
       RT_EP_Packet             : out Stream_Element_Array;
       Last                     : out Stream_Element_Offset);
```

6 MAST Model of RT-EP

This subsection describes the modelling information of RT-EP according to MAST (Modelling and Analysis Suite for Real-Time Applications) [12]. This methodology provides an open-source set of tools that enables engineers developing real-time applications to check the timing behaviour of their application, including schedulability analysis for checking hard timing requirements.

We have built a MAST model to characterize a communications system based on RT-EP, so that the timing behaviour of the distributed hard real-time application can be analysed. The model is divided into three elements: network drivers, the network itself, and the network scheduler.

6.1. Network drivers

In MAST, the overhead of the protocol in the different processors in the ring is modelled by a set of network drivers, one for each processor. The drivers previously available in MAST were not appropriate for modelling the complexity of the RT-EP protocol, and therefore we had to create a new driver called the *RTEP_Packet_Driver*. Its most important attributes are the following:

- *Packet Server*: the task executing the driver in the corresponding processor.
- *Packet Interrupt Server*: It represents the interrupt routine that stores the frames received from the net.
- *Packet ISR Operation* (*ISR*): Code executed by the *Packet Interrupt Server*.
- *Packet Send Operation* (*PSO*): Code executed for the *Send_Info* action.
- *Packet Receive Operation* (*PRxO*): Code executed for the *Recv_Info* and *Send_Initial_Token* actions.
- *Number of Stations* (*N*).
- *Token Manage Operation* (*TMO*): Maximum of the times required to send the token in the *Send_Token* or the *Send_Permission* actions.
- *Token Check Operation* (*TCO*). Code executed in the *Idle* state upon the *Token_Reception* event, that determines the following state.
- *Token Delay* (*TD*). The configurable time of the *Delay* state.
- *Packet Discard Operation* (*PDO*). Code executed when packets addressed to other stations are received, because of the network promiscuous mode used.
- *Token Transmission Retry* (*TR*): Maximum number of faults (and their retransmissions) that we allow in each token arbitration.
- *Packet Transmission Retry* (*PR*): Maximum number of retransmissions when transmitting an *Info Packet*.
- *Timeout* (*T*): Timeout used to detect a message transmission failure.

- *Token Retransmission Operation* (*TRO*): Time consumed in a token retransmission. It corresponds to *Retry_Last_Action* in the *Error_Check* state, when the retransmitted packet is a token.
- *Packet Retransmission Operation* (*PRO*): Time consumed in an *Info_Packet* retransmission. It corresponds to *Retry_Last_Action* in the *Error_Check* state, when an info packet is retransmitted.

6.2. Network

For modelling the ethernet we use a model already built in MAST that is called the *Packet_Based_Network*. Its most important attributes for modelling the timing behaviour are the following:

- *Throughput* (*Rb*): It is the bit rate of the media.
- *Max Packet Transmission Time (MaxPTT):* $MaxPTT = (1492 \cdot 8)/Rb$
- *Min Packet Transmission Time (MinPTT).* $MinPTT = (72 \cdot 8)/Rb$
- *Max Packet Size* and *Min Packet Size*: They are 1492 and 72 respectively.
- *Max Blocking*: The maximum blocking caused by the non preemptability of message packets. In RT-EP, it is calculated as:

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

$$\left(PSO + ISR + PRxO + MaxPTT + \frac{34 \cdot 8}{R_b} + PR \cdot (PRO + T)\right) + (TRO + T) \cdot TR$$

  it represents a complete rotation of the token (*N-1* tokens sent), plus the blocking effect caused by the transmission of non preemptible packet considering the possible faults in the info packet (*PR*(PRO+T)*), and also in the tokens (*TR*(TRO+T)*).

6.3. Scheduler

Each network mush have a primary scheduler for its messages with a scheduling policy. For the latter we use the *FP_Packet_Based* policy that models fixed priority messages composed of non-preemptible packets. Its main attribute is:

- *Packet Overhead*. This is the overhead caused by the protocol information that needs to be sent before or after each packet. It is calculated as:

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

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

which is the time spent to send a number of tokens equal to the number of stations, *N*, performing a complete circulation of the *Token*, plus one *Transmit Token*. The time needed to send a token is calculated as the sum of the *Min Packet Transmission Time*, *Min_PTT*, the time to get the packet from the media (*ISR*), the time of the *Token Check Operation*, and the time of the *Token Manage Operation.*

We also take into account the number of token retries (*TR*) and the cost of the associated *Token Retransmit Operation* (*TRO*) and the Timeouts (*T*). We also have to add the time to send the protocol bytes.

7 Evaluation under MaRTE OS

We have obtained metrics of the worst, average, and best case response times of the different operations executed in the RT-EP protocol, measured in a minimum platform composed of two industrial PCs (Pentium III 750 MHz) running MaRTE OS and connected by a 100 Mbps Ethernet network. The application consisted of five tasks in each PC sending each other average-size messages through five different channels. The results are shown in Table 1.

Table 1. Measured execution times for the RTEP driver operations

RTEP driver operation	Worst (μs)	Best (μs)	Av (μs)
Packet ISR (ISR)	6.48	2.50	3.74
Packet Send (PSO)	60.39	47.98	49.72
Packet Receive (PRxO)	93.13	76.12	77.30
Token Manage (TMO)	41.86	34.70	35.10
Token Check (TCO)	15.65	8.673	9.515
Packet Discard (PDO)	6.169	1.545	1.685
Token Retransmit (TRO)	48.03	36.25	36.79
Packet Retransmit (PRO)	60.38	47.98	49.72

It is also interesting to show the parameters of the network overhead model, including the time it takes to perform a token rotation, the *Packet Overhead* parameter, and the *Max Blocking* parameter (see Table 2). The token rotation was measured with a protocol delay of 100 μs. The other two parameters were calculated using the data from Table 1 in the corresponding formulas.

Table 2. Main Parameters of the Network Overhead Model

Operation	Worst case (μs)	Best case (μs)	Avg. case (μs)
Token Rotation Time	297.56	287.01	288.83
Packet Overhead	411.97	357.62	365.06
Max Blocking	521.58	451.95	461.07

Due to the priority arbitration prior to sending the packet, the effective throughput of the media gets degraded. In order to calculate which is the effective bit rate achieved with the protocol we have extracted two different cases:

- *Synchronized Tx/Rx*: A "synchronized" communication occurs when the stations agree to transmit or receive by any method, producer-consumer, stop and wait, etc. In this case we don't have blocking and the attribute that influences the bit rate is the *Packet Overhead*:

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

  In this case we have achieved an effective bit rate of 22.464 Mbps

- *General case*: The transmission of a packet in the network is totally asynchronous. In this case we have to consider the maximum blocking (521.58 μs with the measured values):

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

  With this consideration the effective bit rate is 11.336 Mbps.

Another important factor of the protocol is how the delay introduced in the configuration influences the load introduced by the protocol. With the delay attribute we can control the protocol CPU utilization, as shown in Table 3.

Table 3. Driver overhead as a function of the protocol delay

Delay (μs)	CPU Utilization
0	12 %
1	12 %
10	12 %
30	7.8 %
50	6.0 %
100	3.8 %
1.000	0.5 %

As it can be seen from the table, the optimum values from the utilization point of view, are with a delay value between 30 and 100 μs. In this band we achieve a low utilization with a quick response from the protocol.

8 RT-EP demonstrator

Once the protocol is designed, implemented and characterized, it is important to use it in a demonstrator in order to confirm its validity in hard real time communications. The chosen demonstrator is an industrial robot arm for maintenance in radioactive environments. A diagram of this platform is shown in Fig. 3.

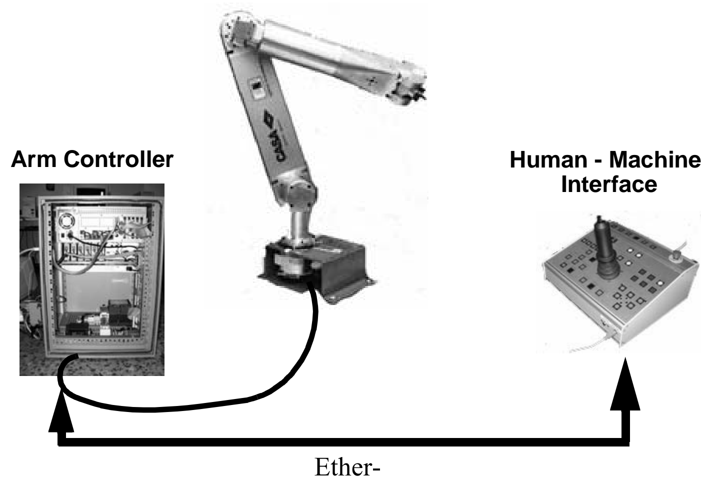

Fig. 3 Demonstrator platform

The control software is written in Ada and the distribution is done through RT-GLADE [8] implemented on top of RT-EP. Through this demonstrator we have confirmed the validity of the protocol.

9 Conclusions

We have presented an implementation of a software-based token-passing Ethernet protocol for multipoint communications in real-time applications, that does not require any modification to existing Ethernet hardware, and does not require any clock synchronization mechanism. The protocol is based on fixed priorities and thus common tools for fixed priority schedulability analysis can be used to analyse the timing behaviour of applications using it. For this purpose, a precise timing model of the protocol has been obtained.

In order to avoid collisions in the ethernet media the protocol uses a token passing mechanism that causes the utilization of the network to be rather low compared to the usual utilization level in standard ethernet. However, the bit rate obtained is still larger than in most field busses, so compared to them we can get the same predictability level at a very low cost and with high performance. The overhead caused by the network driver can be traded against the latency of message passing in the network by means of

the protocol delay parameter. Consequently, RT-EP has proven to be an excellent choice for real-time communications for systems that do not have a large number of stations.

Although the current implementation is in MaRTE OS, it is important to say that the protocol is suitable for other Real-Time OSs and underlying protocols.

References

[1] Alan Burns. “The Ravenscar Profile”. Department of Computer Science, University of York, UK

[2] Chiueh Tzi-Cker and C. Venkatramani. “Fault handling mechanisms in the RETHER protocol”. Symposium on Fault-Tolerant Systems, Pacific Rim International, pp. 153-159, 1997.

[3] Choi Baek-Young, Song Sejun, N. Birch, and Huang Jim. “Probabilistic approach to switched Ethernet for real-time control applications”. Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications, pp. 384-388, 2000.

[4] Harel David, Politi Michael. “Modeling reactive systems with statecharts: the statemate approach“. McGraw-Hill, 1998

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

[6] 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”.

[7] 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.

[8] J. López Campos, J. J. Gutiérrez, and M. González Harbour, “The Chance for Ada to Support Distribution and Real-Time in Embedded Systems“. 9th Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2004.

[9] K. Tindell, A. Burns, and A.J. Wellings, “Calculating Controller Area Network (CAN) Message Response Times”. Proceedings of the 1994 IFAC Workshop on Distributed Computer Control Systems (DCCS), Toledo, Spain, 1994.

[10] M. Aldea and M. González. “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, May, 2001

[11] Martínez, J.M. González Harbour, M. and Gutiérrez, J.J. "RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel". Proceedings of the 2nd International Workshop on Real-Time LANs in the Internet Age, RTLIA 2003, Porto (Portugal), July 2003.

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

[13] Object Management Group (OMG). Unified Modeling Language (UML). http://www.uml.org

[14] Paulo Pedreiras, Luis Almeida, Paolo Gar. "The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency". Proceedings of the 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, June 2002.

[15] Thomas Vergnaud, Jérome Hugues, Laurent Pautet, and Fabrice Kordon "PolyORB: A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications". Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, June 14-18, 2004, in LNCS 3063, Springer Varlag.