RT-EP: Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel

José María Martínez, Michael González Harbour and J. Javier Gutiérrez

Departamento de Electrónica y Computadores, Universidad de Cantabria, 39005-Santander, SPAIN
{martinjm,mgh,gutierjj}@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. This protocol allows the designer to model and analyze the real-time application using it, because it is based on fixed priorities and well-known schedulability analysis techniques can be applied. Furthermore, this protocol provides the applications the capacity of recovering from some fault conditions. It has been ported to an implementation of the Minimal Real-Time POSIX standard called MaRTE OS.

1. Introduction[1]

Ethernet is by far the most widely used local area networking (LAN) technology in the world today, although 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 [10], for example). Some of these approaches are the modification of the Medium Access Control [8], the addition of transmission control [7], a protocol using time-triggered traffic [4], or the usage of a switched Ethernet [9].

The objective of this work is to add a real-time communication network to MaRTE OS [3], which is a real-time kernel on which our research group has been working in the last few years. We want to achieve a relatively high speed mechanism for real-time communications at a low cost, while keeping the predictable timing behavior required in distributed hard real-time applications. The communications protocol proposed in this work is called RT-EP, and can be classified as an addition of a transmission control layer over Ethernet, since it is basically a token-passing protocol in a bus [2]. 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 [5], which did not take faults into account and which was tested under Linux. This paper discusses the fault recovery mechanism designed for this protocol. This extension can be modeled using the MAST [6] Real-time Modeling and Analysis Suite. We also discuss the implementation of RT-EP in MaRTE OS and we also show the overheads it introduces.

The paper is organized as follows. Section 2 introduces how the protocol works. In Section 3 we describe and discuss the fault conditions considered and how we have achieved the recovery method of the protocol. In Section 4 we explain the resulting frame formats of the protocol. Section 5 gives some details about the model describing the timing behavior of the implementation handling faults. In Section 6 we show the MaRTE implementation and provide some results with the overheads introduced by this protocol. Finally, Section 7 gives our conclusions.

2. Description of the Communication Protocol

RT-EP has been design to avoid collisions in the Ethernet media by the use of a token. Each station (processing node or CPU) has a transmission queue, which is a priority queue where all the packets to be transmitted are stored in priority order. Packet information size is limited to 1492 bytes and fragmentation of messages is not allow at this layer. Each station also has a set of reception queues that are also priority queues. 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. Each application thread should have its own reception queue attached. The application has to assign a number, the channel ID, to each application thread that requires communication through the protocol.

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC2002-04123-C03-02, and by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project)

The network is logically organized as a ring. Each station knows which other station is its predecessor and its successor, so 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. The network operates in two phases. The first phase corresponds to the priority arbitration, and the second phase to the transmission of an application message.

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. 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 at the *token_master* station, finishing the arbitration phase.

In the message-transmission phase the *token_master* station sends a message to the station with the highest priority message, which then sends the message. The receiving station becomes the new *token_master* station.

So far, the protocol is not fault tolerant. The loss of a token, for example, will cause the stop of the communication. An extended description of the preliminary protocol implementation can be found in [5]; here we have introduced some changes to deal with the recovery of faults.

3. Handling Faults in RT-EP

Since this protocol is designed for real networks it has to be able to deal with real fault situations that can occur in the life of a system.

We have considered three possible faults to be handled by the protocol:

- *Failure of a Station:* A reconfiguration of the ring is performed.
- *Loss of a packet:* A retransmission takes place.
- *Busy station* (a station that takes too long to respond): Duplicate packets are trashed.

The Real-Time behavior is guaranteed in case of the loss of a packet. The other faults are consequence of bad design or a hardware failure in the system.

The recovery method 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 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.

We can describe the full protocol implementation as a state machine for each station. There are only two additional states to be added for fault recovery, as shown in Figure 1:

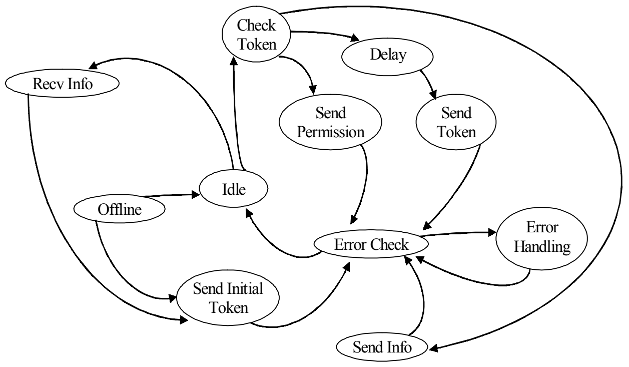

Figure 1RT-EP state machine including faults

- *Error Check*. The station starts listening to the media after transmitting a frame.
- *Error Handling*: In case no *acknowledge* is received, the station attempts to recover the communication.

4. RT-EP Frame Formats

The extension to the RT-EP protocol to handle faults makes it necessary to modify the packets; the token needs to carry more information about the state of the logical ring and a *sequence number* must be added to both bus arbitration (token) packets and information packets.

RT-EP packets are carried in the *Data* field of the Ethernet frame, that must be at least 46 bytes long [1]. Due to this restriction, even though our packets can be less than 46 bytes long, a 46 bytes data field will be built. Our protocol has two types of packets:

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

1 byte	1 byte	2 bytes	6 bytes	2 bytes	6 bytes	6 bytes	22 bytes
Packet Identifier	*Priority*	*Packet Number*	*Token Master Address*	*Failing Station*	*Failing Address*	*Station Address*	*Extra*

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:

Token (used in the arbitration phase to get the highest priority packet) or *Transmit Token* (it grants the destination station permission to transmit a message). The *Priority* indicates the highest priority element on the LAN at the rotation time. The *Packet Number* is used as a sequence number. *Token Master Address* will store the address of the current *token_master* station, this is needed to recover from a *token_master* failure. *Failing Station* and *Failing Address* are used to handle the failure of a station. 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 Address* stores the address of the station with the highest priority packet. Finally, the 22 *Extra* bytes are needed to be compliant with the Ethernet protocol.

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

1 byte	**1 byte**	**2 bytes**	**2 bytes**	**2 bytes**	**0-1492 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. 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. MAST Model of RT-EP with Error Handling

We can build a MAST [6] model to characterize RT-EP with fault recovery, so that the timing behavior of a distributed hard real-time application can be analyzed. The resulting model is similar to the one for same protocol without faults [5]. An *RT-EP Packet Driver* models the overhead of the protocol in a processor; its attributes are:

- *Packet Server* (the thread executing the driver).
- *Packet Send Operation* (*PSO*): code executed in the *Idle* state followed by the *Send_Info* state, and *Error_Check*.
- *Packet Receive Operation* (*PRxO*): code executed in the *Idle* state followed by the *Recv_Info* state, by the *Send_Initial_Token* state, and by the *Error_Check* state.
- *Number of Stations* (*N*).
- *Token Manage Operation* (*TMO*): It corresponds to TMO without errors plus the *Error_Check* state.
- *Token Check Operation* (*TCO*).
- *Token Delay Operation* (*TDO*).
- *Packet Discard* (*PD*).
- *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 of the protocol.
- *Token Retransmission Operation* (*TRO*): Time consumed in a token retransmission. It corresponds to the part of *Error_Handling* that is in charge of the token retransmission.
- *Packet Retransmission Operation* (*PRO*): Time consumed in an *Info_Packet* retransmission. It corresponds to the part of the *Error_Handling* state that is in charge of Packet retransmission.

And the attributes of the *Fixed_Priority_Network* resource for RT-EP are:

- *Max Packet Transmission Time (MaxPTT)* and *Min Packet Transmission Time (MinPTT)*.
- *Packet Overhead*. Considering retransmissions:

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

Max Blocking:

$$(N)(MinPTT+TCO+TMO)+((N-1)\cdot TD)+$$
$$MaxPTT+\frac{26\cdot 8}{R_b}+PR\cdot\left(PRO+T+MaxPTT+\frac{26\cdot 8}{R_b}\right)+$$
$$(MinPTT+TRO+T)\cdot TR$$

R_b is the binary rate.

6. Evaluation under MaRTE OS

This protocol has been ported to MaRTE OS [3]. The architecture is the same as in the protocol without fault recovery (Figure 2). The Linux “socket” that appears in the figure corresponds directly to the network driver in MaRTE OS.

Since MaRTE is a Real-Time OS we can have metrics of the worst, average, and best case response times. In [5], the metrics shown were only average cases, since the implementation was on GNU/Linux, which is not a Real-Time

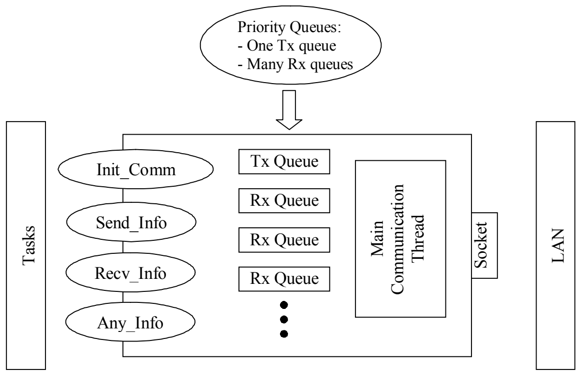

Figure 2Functionality and details of RT-EP

OS, and the worst case metrics were about three orders of magnitude higher than the average ones.

We have provided Ada and C interfaces to the protocol. The times has been measured in a minimum platform composed by two PCs (Pentium III 700 MHz) running MaRTE OS and connected by a 10 Mbps network. The application consisted of five threads in each PC sending each other average-size messages through five different channels.

RT-EP CPU Overheads	Worst (μs)	Best (μs)	Av (μs)
Idle State (+ Error_Check)	48.35	10.8	11.16
Send_Initial_Token	41.16	30.85	31.44
Check_Token	18.51	9.32	9.45
Send_Permission	25.93	24.40	24.75
Send_Token	41.39	24.30	24.74
Send_Info	41.63	37.44	38.36
Recv_Info	37.21	21.19	21.9

7. Conclusions

We have presented a method for dealing with fault situations that can occur in the context of the RT-EP protocol (loss of a packet, failure of a station, and delays caused by a busy station). A precise timing model of the extended protocol has been obtained, which enables us to perform a schedulability analysis of a distributed application using this protocol. With this full implementation (RT-EP + MaRTE OS) we can have a real-time communication system over Ethernet.

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

Future work plans for this protocol are to provide support for multicast traffic and wireless LANs.

REFERENCES

[1] 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”

[2] ANSI/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”.

[3] 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.

[4] 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.

[5] J.M. Martínez, M. González Harbour, and J.J. Gutiérrez. “A Multipoint Communication Protocol based on Ethernet for Analyzable Distributed Real-Time Applications”. Proceeding of the 1st International Workshop on Real-Time LANs in the Internet Age, RTLIA 2002, Vienna (Austria), June 2002.

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

[7] 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.

[8] 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.

[9] 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.

[10]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.