AN ETHERNET LAYER FOR SUPPORTING ENHANCED REAL-TIME COMMUNICATION SERVICES

Tullio Facchinetti * **Gianluca Franchino** *
Paulo Pedreiras ** **Ricardo Marau** **

* *University of Pavia*
** *University of Aveiro*

Abstract:
Real-time communication often requires the support of periodic and aperiodic message transmission, precise control of the message transmission instants, message prioritization, bounded latency for processing the incoming messages, etc. Standard Ethernet drivers do not natively support those features, since they are intended for generic data communication aiming to improve average throughput and performance.
This paper presents a 2-tier solution that natively takes into account the real-time requirements above described, presenting to the application an hardware independent interface that extends the services supported by standard Ethernet layers to include autonomous periodic transmission with selectable scheduling policy, priority-driven message transmission queues and prioritized message delivery on reception. These services simplify the application development and improve the real-time communication performance. The paper includes a set of experimental results that show the effectiveness of the solution, namely concerning the support for periodic message transmission.

Keywords:
Real-time Communication, Ethernet, Distributed Systems, Fieldbus, Real-time Systems.

1. INTRODUCTION

In the last few years the popularity of Ethernet technologies as a suitable solution for supporting real-time communication has grown. Ethernet, however, has been originally developed for general-purpose data communication, which exhibits distinct, and sometimes conflicting, requirements with respect to real-time applications. In the former case average throughput and overhead minimization are typically the main objectives, while in the latter case predictability, bounded access to the transmission medium and precise control of the transmission instants are the key issues.

This situation fostered, in the last years, the development of several protocols able to deliver real-time communication services on top of Ethernet-based networks. The methods used by these protocols range from modifications to the Medium Access Control (MAC) layer to the addition of sub-layers over Ethernet's Data Link Layer (DLL) to control the transmission instants of messages and therefore avoid collisions. More recently, with the advent of Switched Ethernet, which inherently avoids collisions, a new set of work concern-

ing the ability of this topology to support time-constrained communication is being carried out. A survey of these approaches can be found in (Pedreiras and Almeida, 2005; Decotignie, 2005).

While some of the above referred protocols tamper with the standard Ethernet protocol, thus requiring special-purpose Ethernet devices, others may be implemented either on top of Common-Of-The-Shelf (COTS) hardware, typically in the form of custom drivers, or in hardware. While hardware solutions have the potential to achieve better performance, software-based solutions are frequently preferred because they are less expensive to deploy. Some real-time protocols (e.g. Ethernet Powerlink (EPSG, 2005)) specify both types of implementations, with different performance levels.

Developing custom drivers is an expensive task due to the huge amount of Network Interface Cards (NICs) already available on the market and to the rate at which new ones are released. For this reason some real-time operating systems rely on generic NIC drivers, e.g. imported from Linux, with small adaptations. This is the case of the current S.Ha.R.K. real-time kernel network layer (Gai *et al.*, 2001), where Linux drivers are imported through a compatibility layer requiring an ad-hoc interrupt management (Facchinetti *et al.*, 2005). It is also the case of RTNet, the Hard Real-Time Networking for Real-Time Linux (Kiszka *et al.*, 2005), developed for the Xenomai and RTAI real-time Linux extensions. However, using standard existing Ethernet drivers frequently does not allow obtaining adequate guarantees for the real-time communication. For this reason in some cases (e.g. RTNet) the drivers are customized, with critical parts rewritten. A different approach has been adopted in the case of the REDD, RTLinux Ethernet Device Drivers (*REDD homepage*, n.d.), which includes native device drivers as well as device drivers imported from the Etherboot project (*EtherBoot homepage*, n.d.). In this case, besides a better real-time performance when compared with standard drivers, the interface is extended to include a standard POSIX API (open, close, read, write and ioctl calls) made available for the real-time tasks.

Despite the enhancements described in the previous paragraph, in terms of performance or interface, the services delivered by generic drivers do not suit well the requirements of real-time applications due to the lack of support for key real-time functionalities like message prioritization, periodic message transmission and message scheduling. These services can be implemented more easily and efficiently at a lower-level, and can be used directly by real-time applications or to facilitate the deployment and enhance the performance of real-time Ethernet protocols.

This paper presents a 2-tier enhanced Ethernet layer that extends the features of standard Ethernet drivers, to facilitate the development of real-time services on top of the proposed infrastrucure. The enhanced layer provides:

(1) a general infrastructure to support the implementation of real-time communication protocols and applications;
(2) a modular organization to allow the interchange of different scheduling policies;
(3) packet scheduling management transparent to the application;
(4) tight control on the packet transmission and reception instants.

A set of low-level and integrated services providing basic functionalities simplifies the development of real-time distributed applications as well as complex protocol stacks. The architecture proposed in this paper supports different packet scheduling policies through a transparent management of queues with different priorities for real-time traffic, both periodic and aperiodic, and regular non-real-time traffic. Moving the communication management to the lowest possible level improves the control over several key aspects like time-stamping (hardware time-stamping is actually quite uncommon) and transmission instant control. Furthermore, it also allows a better integration of the communication layer with the operative system, since the interference to the scheduling of the real-time application tasks can be analyzed and accounted with available methods (Jeffay and Stone, 1993).

The rest of the paper is organized as follows: Section 2 describes the driver architecture; Section 3 shows an example protocol implemented on top of the enhanced driver and Section 4 reports the experimental results obtained from such implementation. Finally, the conclusions are stated in Section 5.

2. THE ARCHITECTURE

The 2-tier architecture, depicted in Figure 1, is organized in two complementary layers. The higher layer includes the interface with the application level, buffer management and packet management (scheduling, classification and dispatching).The lower layer is the hardware-dependent component of the infrastructure, being responsible for carrying out the actual packet transmission and reception. Both the layers are thought to be modular, allowing the interchange of different modules, and supporting different NICs and communication policies.

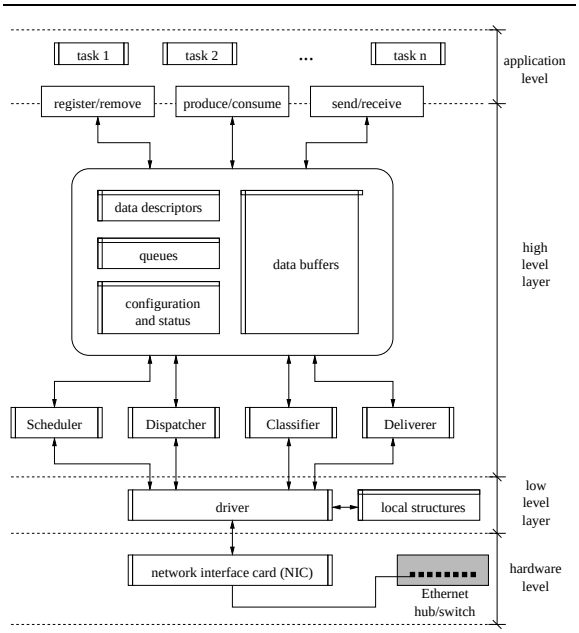

Fig. 1. The communication infrastructure scheme.

2.1 High-level layer overview

The high-level layer (Figure 1) provides to the application a set of primitives that allow to register and unregister classes of messages, to define the priorities of different packet streams, and to send/receive messages using the two most frequently adopted semantics: state-based (produce/consume) and event-based (send/receive).

The communication management requires a set of structures to store data and meta-data, namely:

- data buffers, containing the message data;
- data descriptors, storing message attributes like message IDs and communication status, data length, and callback functions associated with specific events (transmission, reception, errors, etc.);
- queues, to manage the message transmission from the data buffers and the buffer update after a packet reception.

The fundamental components of the higher level layer are the *Packet Scheduler*, the *Packet Dispatcher*, the *Packet Classifier* and the *Packet Deliverer* (Figure 2). While the Packet Scheduler and the Packet Dispatcher manage the message transmission, the Packet Classifier and the Packet Deliverer handle the message reception.

The job of the Packet Scheduler consists in sorting the packets that refer to a given queue. Distinct Packet Schedulers can be associated to distinct queues, allowing different policies for the transmission of distinct message streams. For example, a queue may contain messages sorted according with the arrival time (FIFO) while the messages of another queue may be sorted based on their absolute deadlines or, for periodic streams, according to the periods. The Packet Dispatcher is activated upon the occurrence of relevant transmission events, namely the completion of a previous transmission request, an explicit request for the transmission of an event message or, finally, the expiration of an internal timer resulting in the activation of a new instance of a periodic state message. When activated, this component scans the transmission queues, according to their priority, and triggers the message transmissions when a non-empty queue is found.

The Packet Classifier acts upon packet receptions, performing the packet classification and moving it to the queue associated to the packet class. The availability of multiple queues for the packet reception allows an out of the order management of the incoming messages that can be sent to the application tasks according to their priority instead of their arrival order. A Packet Deliverer, that is associated to each reception queue, is in charge to activate the application tasks after a message reception. If needed, the Dispatcher and the Deliverer may also perform some kind of traffic shaping to limit the load on the system given by the communication traffic.

2.2 Low-level layer overview

The lower level interacts directly with the NIC's hardware. This layer is organized into modules, each module supporting one specific network card. The modules may be obtained by modifying standard Linux network drivers, rewriting critical sections such as the ones related with message transmission, message reception and interrupt handling, in an approach similar to RTNet (Kiszka *et al.*, 2005).

The interrupt handling is the core element for achieving adequate real-time performance. It is also important because it represents the link between the lower and the higher level layers. To show the interaction between the two layers, performed through the *Interrupt Service Routine* (ISR), it is interesting to describe the most important details of an ISR.

Figure 3 shows the structure of a generic ISR. Some NICs do not generate all the events depicted in Figure 3. However, the absence of some class of interrupts only impacts on the performance of the system, not affecting the interaction between the two layers of the enhanced driver.

The ISR is called whenever an interrupt is triggered by the NIC. Referring to Figure 3, each branch manages a specific communication event, signaled by a specific interrupt source, and results in the activation of a callback task. The callback tasks signal the interrupt at the higher layer, trig-

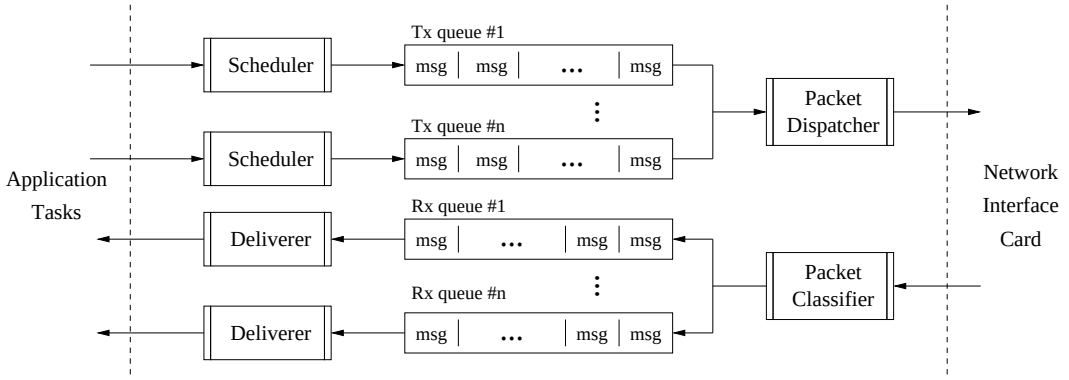

Fig. 2. Queues management scheme at the high level layer.

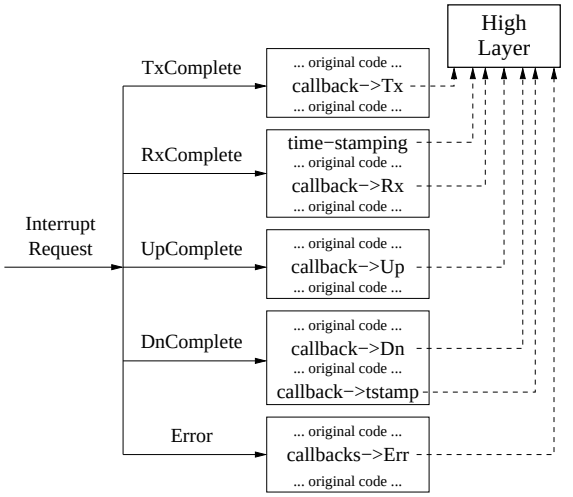

Fig. 3. Structure of the interrupt handler.

gering the adequate actions, e.g. queue, message or buffer updates.

The most common events that generate interrupt requests are[1] :

- *TxComplete*, after the completion of a packet transmission;
- *RxComplete*, after the reception of a packet within the NIC's memory;
- *UpComplete*, after copying the message from the NIC's memory to the host PC memory;
- *DnComplete*, after copying the message from the host PC memory to the NIC's memory;
- errors.

For example, when an *UpComplete* event is generated, the interrupt handler executes the code inherited from the original driver and activates the callback routine for the associated upper layer mechanism, which in this case is the Packet Classifier. The activation of this task ultimately results in the update of the reception queue associated with the received message and the activation of the associated application-defined reception callback task.

For many communication schemes, precise time-stamping of message transmission and reception is crucial. Whenever hardware time-stamping is not available, it is desirable to obtain a timestamp at the lowest possible level, avoiding the jitter induced by the execution of several protocol-stack related management functions. For this reason, the Ethernet layer herein proposed natively provides timestamp information at the low level. During message reception, the timestamps are acquired when *RxComplete* interrupts are generated. For message transmissions the timestamp is generated just before the low-level transmission request. Not all NICs are able to generate all the interrupt events listed above. In these cases the interaction scheme between higher and lower layers does not change appreciably but the performance of some services may be degraded. E.g., some NICs are not able to generate *RxComplete/UpComplete* pairs of events, generating instead a single interrupt when the message is uploaded to the host memory. In this case low-level timestamp is less accurate since it includes also the additional jitter and latency associated with the upload operation.

3. IMPLEMENTATION

The enhanced driver has been implemented under the S.Ha.R.K. real-time kernel (Gai *et al.*, 2001). This kernel already includes a network library, which is adapted from the Linux drivers using a layer of glue code and specific techniques for the interrupt management (Facchinetti *et al.*, 2005). The layered organization of the enhanced driver fits well with the modular organization of the S.Ha.R.K. operating system, where both scheduling algorithm for the application tasks and resource sharing policies are highly configurable and interchangeable.

[1] Even though the naming convention varies among different NICs, the associated actions remain generally the same.

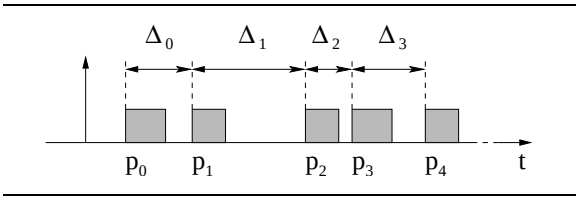

Fig. 4. Measurements performed for the received packets.

3.1 Example of transmission protocol

To assess the correctness and performance of the implementation, a MAC level protocol, similar to the Implicit EDF proposed by Caccamo et al. in (Caccamo *et al.*, 2002) for wireless sensor networks, was deployed on top of the enhanced Ethernet driver. Implicit EDF uses a slotted TDMA medium access scheme where a replicated Earliest Deadline First (Liu and Layland, 1973) scheduler rules the access to the communication channel in each slot: only one node can broadcast a message in a given slot. It allows the broadcasting of multiple streams of periodic messages and eliminates all the broadcasting collisions, which are one of the main sources of communication unpredictability. As for any TDMA protocol, the performance of Implicit EDF is severely affected by the packet transmission timeliness and clock synchronization accuracy, which constraints the guard-bands and thus the protocol efficiency. The enhanced features of the Ethernet layer herein described address both these issues, providing autonomous message triggering, improving the packet transmission timeliness, and improved time-stamping accuracy, allowing more precise clock synchronization.

The Implicit EDF protocol has been implemented by triggering the Packet Scheduler task at every slot. The Scheduler inserts new packets into the ready queue and the Packet Dispatcher fetches the ready packet to be sent in the current slot.

4. EXPERIMENTAL RESULTS

The performance evaluation is based on the infrastructure described in Section 3: it adopts the Implicit EDF scheduling policy to guarantee the communication real-time constraints. The measurements have been performed on two different experimental setups:

(1) PCs equipped with Intel 550 MHz Pentium III-MMX processors;
(2) PCs equipped with Intel 1.7 GHz Pentium IV processors;

Both the setups use 3Com 3C905CX-TX-M PCI Ethernet cards with the transmission speed set to 10 Mbit/sec.

All the machine processors are loaded with hard real-time tasks for a total utilization of about 80%, with $45 \sim 50$ tasks having a period of 50ms and a computation time varying from 700 to $900\mu s$.

Table 1. Parameters used for the experimental tests and worst-case jitter.

	Slot size [μs]	Payload [bytes]	Jitter 1 [μs]	Jitter 2 [μs]
Test 1	500	200	36	45
Test 2	500	50	41	46
Test 3	500	10	36	45
Test 4	1000	1000	36	44
Test 5	1000	100	36	47
Test 6	10000	1000	33	42
Test 7	10000	10	35	37

The senders generated up to eight periodic streams with different periods and packet lengths. Table 1 shows the packet periods and payloads considered for the experimental tests. They are set to send exactly one packet in every slot, with identical payload for each packet. For example, during Test 1 a message with a payload of 200 bytes is sent every $500\mu s$. The network was composed only by nodes executing the Implicit EDF protocol, resulting in the absence of message collisions.

The measurements were performed both at the packet transmission and reception instants. The measured transmission latency at the sender nodes, i.e., the difference between the desired transmission instant and the actual transmission timestamp, was around 1-2μs for all the tests run on both experimental setups. This result was achieved in presence of relatively high processor utilizations and is independent from the transmission periods and payload size.

The relative reception jitter (Δ_i, Figure 4) was computed by the receiver nodes by calculating the difference between the arrival time of consecutive incoming packets, p_i and p_{i+1}. The last two columns of Table 1 report the reception jitter variation ($max(\Delta_i) - min(\Delta_i)$ for every i) for both the experimental setups. This value is considerably higher than the variation of the transmission latency because it accumulates the effects of the jitter induced by the Ethernet switch and packet reception at the consumer node. It is interesting to note that the reception jitter variation is also independent of the transmission periods and payload size.

Figure 5 shows the histogram of the relative reception jitter for both setups. For setup 1 and for all the tests except Test 7, from 50% to 75% of the messages have experienced a jitter lower than $4\mu s$. Furthermore, over 99% of the messages have a jitter below 16 μs. The performance of the setup 2 is considerably worse, with more than 50% of the messages experiencing a jitter greater than $16\mu s$. Notice that the X axis of Figure 5 is truncated to

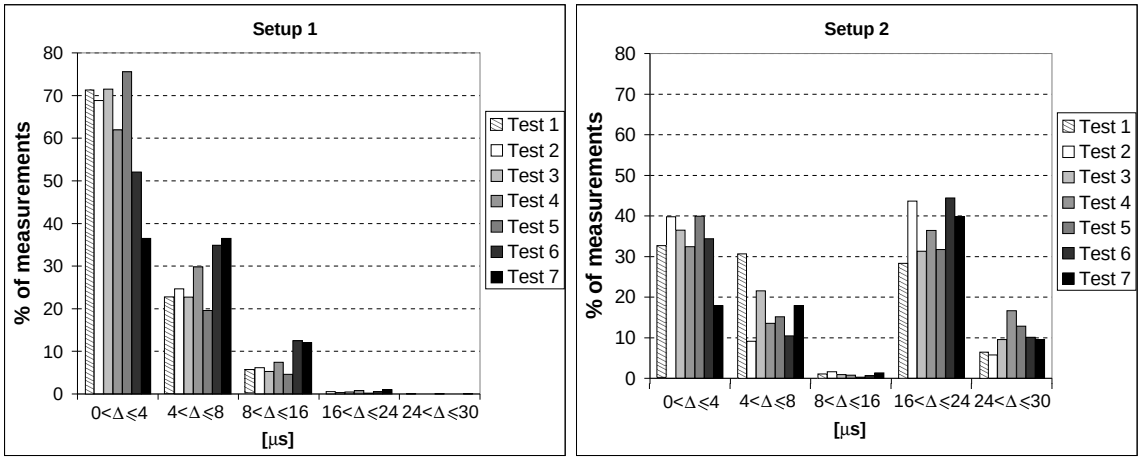

Fig. 5. Message reception jitter histogram for setups 1 and 2 respectively.

$30\mu s$ due to the very limited number of packets (between 5 and 30 in several millions) that had a jitter greater than $30\mu s$.

The results displayed in Figure 5 show a relative independence of the jitter from both the frequency of the message generation and the packet size, since all the tests have similar distributions of the jitter values for a given setup. On the other hand, comparing the performance of the two setups, the slower systems turn out to have lower jitter. This fact may be explained considering the difference among processors and motherboards for the two architectures. Moreover, the extremely low number of messages with a jitter between 8 and $16\mu s$ for setup 2 may confirm the dependence of the results on the machine architecture.

5. CONCLUSIONS

This paper described a 2-tier enhanced real-time Ethernet layer, which extends the services supported by standard drivers by including natively autonomous periodic transmission with selectable scheduling policy, priority-driven message transmission queues, and prioritized message delivery on reception. These services simplify the application development and improve the real-time communication performance as well as the system predictability, since the overhead and the interference to the real-time application tasks can be taken into account.

The enhanced driver was implemented as a part of the S.Ha.R.K. real-time kernel. The experimental results have shown that the driver provides good level of control of the message transmission instants and a relatively low message reception jitter. Furthermore, these figures are independent of the application CPU load and from the traffic characteristics, namely period and payload size.

REFERENCES

Caccamo, M., L. Y. Zhang, L. Sha and G. Buttazzo (2002). An implicit prioritized access protocol for wireless sensor networks. In: *Proceedings of the IEEE Real-Time Systems Symposium.*

Decotignie, J.-D. (2005). Ethernet based real-time and industrial communications. In: *Proceedings of the IEEE.* Vol. 93. pp. 1102–17.

EPSG (2005). Ethernet powerlink protocol. www.ethernet-powerlink.org.

EtherBoot homepage (n.d.). http://etherboot.sourceforge.net.

Facchinetti, T., G. Buttazzo, M. Marinoni and G. Guidi (2005). Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In: *Proceedings of the IEEE Euromicro Conference on Real-Time Systems.*

Gai, P., L. Abeni, M. Giorgi and G. Buttazzo (2001). A new kernel approach for modular real-time systems development. pp. 199–206.

Jeffay, K. and D. L. Stone (1993). Accounting for interrupt handling costs in dynamic priority task systems. In: *Proceedings 14th IEEE Real-Time Systems Symposium.* pp. 212–221.

Kiszka, J., B. Wagner, Y. Zhang and J. Broenink (2005). RTnet - a flexible hard real-time networking framework. In: *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation.*

Liu, C. L. and J. W. Layland (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery.*

Pedreiras, P. and L. Almeida (2005). *The Industrial Communication Systems Handbook.* Chap. Approaches to Enforce Real-Time Behavior in Ethernet. CRC Press.

REDD homepage (n.d.). http://redd.sourceforge.net.