Temporal behavior of Ethernet communications: Impact of the operating system and protocol stack

Pedro Silva*+ Luís Almeida*+ Mário Sousa* Ricardo Marau*
+Instituto de Telecomunicações, Porto
*DEEC - Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, 4200-465 Porto PORTUGAL
{psns, lda, msousa, marau}@fe.up.pt

***Abstract* – Ethernet is currently the most widely used networking technology, spanning across many application domains including embedded systems. In this particular case, Ethernet is even used in many time-critical applications in which the delay induced by communication must be short and bounded. It is thus very important to understand the entire transmission process and assess its temporal behavior. There are a number of aspects to consider, including the network protocol, network topology, network elements and end devices. This paper aims at assessing the impact of the operating system and its protocol stack implementation in the end devices on the network temporal behavior. We studied four operating systems, namely a standard Ubuntu distribution with and without a real-time kernel patch, an embedded stripped down version of Linux and QNX Neutrino, and two hardware platforms, namely ordinary PCs and a single board computer based on an AVR32 CPU. We measured the Round Trip Delay (RTD) using RAW, UDP and TCP sockets to interface the protocol stack. We verified that on high computing power platforms the difference between the sockets is small but still significant in resource-constrained platforms. On the other hand, full featured general OSs present rather large worst-case delays. These can be reduced using real-time patches for those OSs, RTOSs, or even removing unnecessary modules, services and particularly, data intensive device drivers. We believe this study can be helpful for system designers as well as for teaching networks courses in embedded systems.**

Distributed embedded systems; protocol stack; operating systems; education; latency; socket interface

I. INTRODUCTION

Ethernet is currently the most widely used computer network technology. Mainly due to its relatively low cost, both in terms of hardware, software and even training of designers, installers and maintainers, and its high bandwidth and flexibility of use, it became the de facto standard for computer networks in many environments, not only in the office/domestic environment but also in large enterprises, industrial automation and even medium to large embedded systems [1].

However, the use of Ethernet in applications with real-time constraints requires a careful assessment of its temporal behavior in terms of end-to-end transmissions between end nodes. This includes assessing the performance of the network itself in terms of throughput and transmission delays as well as of the network interfaces at the end nodes. Much effort has been devoted to analyze and control the performance of the network itself but minor attention has been given to the impact of the end nodes due to the operating system and protocol stack implementation. This is probably due to the typical use, in real-time applications, of special real-time protocols that have very light application layers that provide direct access to the data link layer implemented inside the network controller. However, with complex applications, complex operating systems must be used, which manage many devices and execute many tasks concurrently. Similarly, more abstract network interfaces that include network and transport layers functionality, e.g., logic and hierarchical addressing, automatic fragmentation/re-assembly and end-to-end transmission control, become highly attractive. As a result, the end nodes operating system and protocol stack gain a stronger impact in the end-to-end transmission latencies, impact that must be bounded and properly analyzed.

Nevertheless, certain operating systems, such as general purpose ones, and protocol stacks, such as TCP/UDP/IP, present strong limitations in delivering hard real-time performance levels. For example, it might be impossible to control transmission instants with a precision better than a few tens of microseconds. This, in turn, might prevent the efficient use of software-implemented Ethernet-based real-time protocols. The delivery of information is equally affected with a reduced temporal precision, with extra delay and jitter.

On the other hand, the actual latencies imposed by the OS and protocol stack are technology dependent. This means that it is important to carry out an actual assessment to determine their magnitude. With such information, a designer might do a more informed trade-off between the extra latencies imposed by a more elaborated protocol stack such as TCP/UDP/IP and the higher abstraction level and flexibility of use that it provides, or the trade-off between using a general purpose operating system with all its support to hardware and development, and a real-time operating system with its improved temporal determinism but more limited hardware support.

This paper presents such an experimental assessment, focusing on the latencies induced by the pair operating system / protocol stack. It includes four operating systems (standard Linux, Linux with a real-time patch, embedded Linux and QNX Neutrino), two hardware platforms (common PCs and an AVR32 single board computer), three protocol stacks (TCP/IP, UDP/IP and Ethernet device driver) and considers two interconnection topologies (back-to-back connection and through a switch). The assessment is done measuring round-

trip delays. The work was carried out in the scope of the iLAND ARTEMIS project that aims at developing a real-time service-oriented middleware for dynamically reconfigurable systems.

The rest of the paper is organized as follows, the next section discusses the sources of the impact of the end nodes in the communication latencies, namely the operating systems and the protocol stacks. Then, Section 3 describes the experimental test bed, while Section 4 presents and discusses all the round-trip measurements. Section 5 concludes the paper.

II. IMPACT OF END NODES

In a networked system, the communication between two or more end nodes takes a certain time that is dictated by several components, such as the length of the communication channel and the speed of information propagation, transmission rate of information symbols and length of information units, medium access protocol, latencies in network elements and routing policies and the network service time and buffer management policies in the end nodes, involving the network device driver and protocol stack layers. In this paper we show experiments to measure the whole end-to-end delay but, by using a simplified network layout and varying the operating system, hardware platform and protocol stack interface we focus on the impact of these aspects in the communication latency.

A. Operating system

The most important network-related feature that an operating system (OS) provides in the end nodes is the interface to the network, which includes the device driver for the specific network interface card (NIC) and the support of the network protocol stack. The temporal behavior of both components depends on the OS architecture and its ability to isolate the interference from other subsystems running on the same node.

General purpose OSs, such as Linux, are normally designed to improve throughput and average responsiveness, exhibiting a number of architectural features that degrade their temporal predictability and their real-time performance, namely execution of asynchronous interrupts, non-preemption, poor timer resolution, frequently in the order of 10ms, spin-locks typically used to deliberately postpone interrupts, as well as the use of best-effort schedulers.

To overcome these issues, several Linux-oriented real-time OSs (RTOSs), such as RTLinux or RTAI, were developed which reduce or even eliminate these sources of unpredictability, normally trapping the system asynchronous (non-predictable) interrupts and executing them at a task level according to a preemptive scheduler. These are, however, specialized real-time kernels that execute Linux as a non-real-time thread. However, in recent years, the Linux community started to pay attention to timing issues in order to improve the responsiveness and predictability of the Linux kernel, pushing patches to the main distribution kernel that introduce preemptive execution, high resolution timers, non-blocking spin-lock calls (whenever a higher priority execution is issued) and true priority-based scheduling. These real-time patches are quite recent and to the best of our knowledge there is still a lack of assessment of their impact.

OSs following a micro-kernel architecture have a reduced kernel size thus with reduced non-preemptive sections and potential for lower blockings and improved responsiveness. Concerning the network case, the whole protocol stack is executed as a task or set of tasks in the system, consistently scheduled with the remaining ones according to the application requirements. One example is QNX Neutrino, which is an RTOS. Linux is, conversely, an example of a monolithic OS, in which many services are included inside the kernel. Therefore, in this case, the more services are included in the kernel, the higher the potential for longer non-preemptive sections and stronger kernel impact on the system temporal behavior. In this aspect, embedded versions of monolithic OSs that are stripped down so as to include the minimum functionality can present substantial improvements in the predictability of the system temporal behavior.

B. Protocol stack

A protocol stack or communications stack is a particular hardware and/or software implementation of a computer networking protocol suite. The traditional packet-based network architecture assumes that communication functions are organized into nested levels of abstraction called protocol layers [2].

In practical implementations protocol stacks are often divided into three major sections - media, transport, and applications. A particular operating system running on a specific platform will often have some of the protocol layers implemented in hardware, other layers implemented in software inside the operating system kernel, and yet other layers implemented in software outside the operating system, i.e., in user space.

The most commonly found division of the protocol layers between the three implementation types is to have the media section implemented in hardware, the transport layers implemented inside the operating system kernel, and the application section in user space. This architecture often results in two well-defined interfaces: one between the media and transport layers, and one between the transport layers and applications.

The OSI model [3], on its top three layers, defines how the applications within the end stations will format the data being exchanged and how the interaction between them is managed, and are usually implemented in user space. The bottom four layers of this model (the media and transport sections) define how data is transmitted end-to-end, and are often implemented in hardware (media and part of the data link layers) and inside the operating system (remaining part of the data link, network and transport layers).

As mentioned in [4] there are a few protocols that are usually called the “core” of the suite, because they are responsible for the basic operation of a very wide range of higher layer protocols and applications: the Internet Protocol (IP), Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). IP is a network layer protocol that provides a logical network interface independent of the media. TCP is connection-oriented protocol operating over IP, providing reliable, ordered, data transmission using byte streaming

semantics. UDP is a simpler message-based connectionless protocol also operating over IP and offering unreliable, un-ordered data packet transmission [5]. This set of protocols is typically referred to as the TCP/IP protocol stack.

Concerning the temporal impact of the TCP/IP stack, it is important to realize that TCP and UDP represent two different trade-offs with respect to the reliability versus timeliness conflict. While TCP favors the former, including a complex automatic retransmission procedure that ensures a reliable and orderly delivery, UDP favors the latter, not imposing any extra delays beyond those associated to the protocol overhead. In case of a clean error-free channel, we expect TCP and UDP to perform similarly, with just an extra overhead in TCP due to its higher internal complexity.

In any case, common TCP/IP stacks tends to use significant amounts of memory for the multiple simultaneous sessions they can support as well as for the automatic fragmentation/ reassembly and retransmissions management. These lead to a substantially unpredictable temporal behavior and to memory and processing requirements that are incompatible with low resource platforms such as 8/16-bit microcontrollers. For these cases, there are stripped down implementations of the TCP/IP stack, such as lwIP and µIP [6], which constrain the number of simultaneous connections and carry out efficient buffers management, e.g., with zero-copy techniques, that result in a significant improvement in the temporal behavior and resource requirements.

Finally, beneath the TCP/IP stack a mandatory layer interfaces the media and transport sections, which is the network driver. This is a device driver, thus residing in the OS kernel space, which manages the communication between the network interface card and the upper protocol stack. It provides direct access to the network data link layer, which is typically implemented in hardware by the network controller.

The way the device driver is designed, particularly with respect to buffer management policies, has a direct impact on its temporal behavior. However, it provides the simplest communication interface in the end nodes, with the lowest abstraction level and consequently the lowest overhead.

C. *The socket interface*

Application and higher layer protocol designers frequently access the network services using the TCP/IP stack directly through an operating system specific API (Application Programming Interface). Even though there are currently many operating systems, most of them use what is known as the socket interface (or a derivative thereof, e.g., Winsock) for this programming interface, which has therefore become a de facto standard [7].

Also known as Berkeley sockets, since this API was originally developed in the University of California at Berkeley, the socket interface allows applications to access communications end-points with similar semantics to a file handle, through which an application may read and write data. The socket interface is generic, and may be used for very distinct protocol implementations with very different quality of service. Supported QoS classes are the reliable stream-oriented service (an unbounded sequence of bytes), unreliable datagram service, reliable sequenced packet service, and raw protocols which directly access the underlying network data link layer. Each of these QoS classes is commonly provided by a distinct protocol of a family. For the TCP/IP family, the stream class QoS is provided by TCP, while the unreliable datagram is provided by UDP. The raw sockets allow sending and receiving data link layer packets directly, which corresponds to bypassing the TCP/IP protocol stack and providing direct access to the network driver.

Therefore, the socket interface seems to be particularly adequate to measure the impact of different protocol stack interfaces, particularly between TCP, UDP and RAW sockets.

Finally, there are also other kinds of sockets, some of them enhanced towards a specific goal. For example, the work in [8], MuniSocket, proposes an expandable high bandwidth solution with fault tolerance and load balancing for transmitting large messages over multiple networks. A specific socket interface based on UDP provides parallel message fragmentation, transmission and reconstruction in a transparent way. However, such enhanced socket versions are out of the scope of the study presented in this paper.

III. EXPERIMENTAL TESTBED

As mentioned before, our purpose is to assess the impact of the operating system and protocol stack in the end-to-end delays of Ethernet communication. The impact of different protocol stacks was assessed using sockets of different types, namely TCP and UDP for the two upper layer protocols of the TCP/IP stack and RAW for Linux and BPF for QNX Neutrino for direct access to the network driver.

These experiments were repeated under four operating systems: Ubuntu standard distribution, Ubuntu with real-time kernel patches, an embedded Linux running the 2.6.24 kernel and the QNX Neutrino RTOS.

All but the embedded Linux OSs were running on two standard Personal Computers - HP xw4600 WorkStation QuadCore (one @2.66Ghz and other @2.50Ghz) and a Gigabit Ethernet network interface. The embedded Linux OS was running on a single board computer, an ICnova board with a 140Mhz AP7000 CPU (AVR32 architecture) and an 100Mbit/s Ethernet interface.

To reduce the impact of the network itself, we used two very simple interconnection topologies with no extra load beyond the RTD measurements:

(i). A direct cross-over connection over the Ethernet interface

(ii). Connection through two 100Mbit/s or 1Gbit/s ports of a layer 2 Ethernet switch (Allied Telesis AT-8000S).

However, given the consistency of the results achieved in both cases, in this paper we focus on those obtained with configuration (i), only, despite the plots containing both.

We measured the Round Trip Delay (RTD),as depicted in Fig.1, when transmitting packets with a data payload varying from 46 to 1500 bytes (the minimum and maximum payload on an Ethernet packet), and also with other intermediate values of 64, 128, 256, 512 and 1024 bytes. Note the timestamps were extracted just before sending the packet and right after receiving it, thus including the whole round trip process.

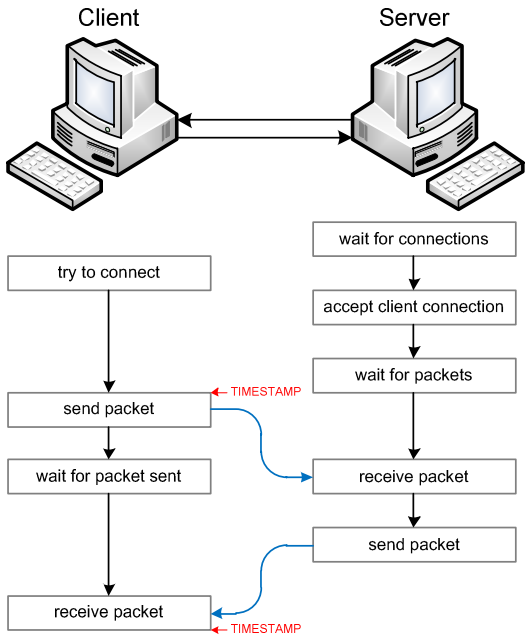

Fig.1 – Application software for measuring the round-trip delay

IV. ROUND TRIP DELAY RESULTS

The Round-Trip Delay (RTD) appears in the literature as the time between sending a message and receiving it back. However, there are two common definitions depending on whether the initial transmission time is included or not, i.e., whether the initial timestamp is extracted before or after sending the packet. In this work we consider it extracted before, thus our RTD measurements include the whole process as depicted in Fig.1, and it can be decomposed in several components as shown in Fig.2.

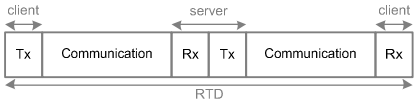

Fig.2 – RTD components – Direct connection

In the case of using a direct connection, i.e., sender and receiver back-to-back, the RTD can be expressed by (1) where TX is the sending time, i.e., the time to prepare the packet and send it to the Physical Layer (PHY), Tcom is the transmission time on the wire and RX is the reception time, i.e., the time it takes since the packet arrives at the receiver PHY and is made available to the receiver application.

$$RTD_{direct} = 2 \times (T_x + T_{com.} + R_x) \quad (1)$$

In this case, the time taken by the protocol stack on each side is given by (2).

$$T_{Protocol\ Stack} = T_x + R_x \quad (2)$$

Replacing (2) in (1) we can infer the time taken inside the protocol stack from the measured RTD as in (3) where TotalFrameSize represents the total length of the frame in bits including all PHY overhead, namely interframe space, preamble and start-of-frame and minimum bits per frame.

$$T_{Protocol\ Stack} = \frac{RTD_{direct}}{2} - \frac{\text{Total frame size}}{\text{Transm. rate}} \quad (3)$$

The experimental procedure to measure the RTD used 10000 repetitions for each payload value and for each configuration. Then we computed the average and the standard deviation considering the values below the 99.9 percentile, only, in order to avoid the impact of spurious spikes that could possibly occur. Nevertheless, such spikes were considered for determining the maximum and minimum values.

In the plots that follow, the results obtained with a direct connection are represented by squares and with a connection through the switch are represented by dots. The RAW sockets are represented by dashed lines, UDP sockets by dotted lines and a continuous line is used for TCP sockets. The tables contain the direct connection data, only.

A. *Standard Ubuntu distribution*

Table 1 and Fig.3 show the case of Ubuntu with a standard distribution (no RT features). As expected, the RTD increases with the payload but such increase is only noticeable after 512B. This is expected since at 1Gbit/s, for processing overhead reasons, the PHY always sends at least 512B per frame, independently of the payload. Consequently, the resulting RTD becomes insensitive to the payload for values less than 512B.

TABLE 1 – RTD ON UBUNTU STANDARD – RAW, UDP AND TCP SOCKETS – 1 GBIT/S

Payload [bytes]	Connected Directly					
	RAW		UDP		TCP	
	Mean [µs]	*STD [µs]*	*Mean [µs]*	*STD [µs]*	*Mean [µs]*	*STD [µs]*
46	101,0	4,1	109,2	3,6	113,8	3,7
64	102,2	5,2	108,5	4,3	113,7	3,8
128	101,5	6,7	109,9	4,1	112,9	4,5
256	105,3	5,8	111,7	3,7	115,6	3,7
512	109,3	4,2	112,4	4,3	117,2	4,3
1024	134,1	5,8	140,6	4,0	150,1	4,7
1500	152,9	4,1	172,1	4,0	172,8	3,3

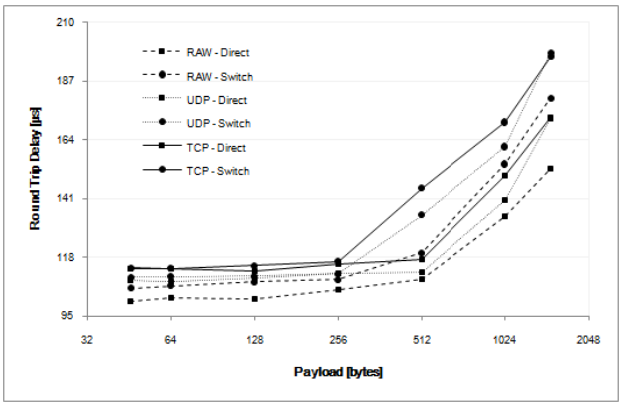

Fig.3 – RTD on Ubuntu standard – RAW, UDP and TCP sockets

Another interesting observation is the small RTD difference between RAW, UDP and TCP sockets. This means that the reduction achieved by using RAW sockets might not pay off the increased management complexity that it incurs and it may be better to use UDP, which already includes a network layer and abstracts away the details of the physical network such as physical addresses. The standard deviation is relatively low, around 4% of the average values, meaning that large deviations are infrequent. Moreover, the standard deviation does not seem to vary with the payload. This behavior was observed for all the cases.

However, the maximum values observed go up to 537µs, detected with TCP with 128B packets, representing 3.8 times the respective average value. Also RAW and UDP sockets exhibited spikes that went up to 2.4 times the respective average values. This shows that the worst-case behavior is poor and might create problems for time-critical applications, even when using RAW or UDP sockets. Moreover, note that TCP sockets can always exhibit large delay variations due to its more complex internal control flow, retransmissions and concurrent connections, independently of the OS, as it will be seen later on. This is the reason why TCP is normally not recommended for time critical applications.

B. *Ubuntu with RT patched kernel*

Table 2 and Fig.4 shows the values achieved with Ubuntu but with the kernel patched with the RT features, namely preemption and high resolution timers.

TABLE 2 – RTD ON UBUNTU RT KERNEL – RAW, UDP AND TCP SOCKETS – 1 GBIT/S

Payload [bytes]	Connected Directly					
	RAW		UDP		TCP	
	Mean [µs]	*STD [µs]*	*Mean [µs]*	*STD [µs]*	*Mean [µs]*	*STD [µs]*
46	98,8	3,0	103,8	2,2	107,7	2,5
64	99,3	3,0	104,0	2,2	108,0	2,2
128	101,7	2,4	104,2	2,2	108,6	2,3
256	103,0	2,2	105,5	2,3	109,3	2,9
512	104,6	2,2	107,4	3,7	123,2	10,4
1024	132,4	3,4	140,1	4,2	147,4	3,4
1500	149,2	3,5	168,8	3,6	171,3	4,5

The average behavior is very close to the previous case with a negligible reduction in the average delays. However, there is a noticeable reduction in the standard deviation, around 2% of average values, meaning that the delays are more steady and closer to average. This improvement becomes evident in the maximum values detected. In fact, the maximum delay observed was now 241µs with UDP and 1500B packets, representing a spike of 42% above the average value. Again note the higher standard deviation with TCP at 512B caused by more irregular delays associated to the more complex internal management structure.

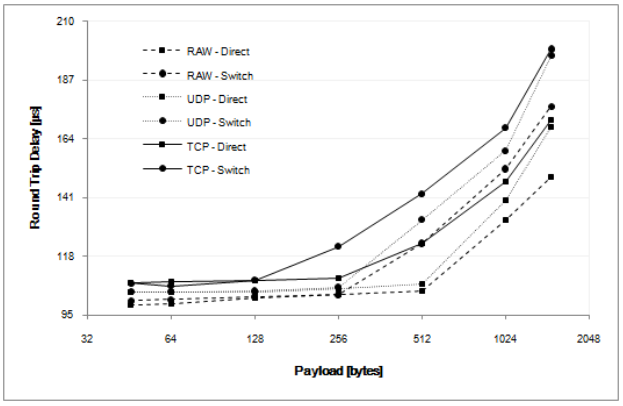

Fig.4 – RTD on Ubuntu with RT kernel – RAW, UDP and TCP sockets

C. *Embedded Linux platform*

Table 3 and Fig.5 show the case of the embedded Linux running on a platform with substantially less computing resources. Moreover, the bit rate is now 100Mbit/s. Together, these effects are clearly visible on the obtained results, with substantially longer RTD values than in the previous cases. In this case we can distinguish clearly the difference between the three types of sockets. The difference, now, between RAW and UDP sockets, as well as between these and TCP sockets, is significant. Consequently, for delay-sensitive applications, it might now be worth considering the use of RAW sockets which grant a direct access to the device driver leading to a much smaller overhead than IP based protocols. On the other hand, the extra weight of TCP's internal mechanisms to provide reliability and ordered delivery, detecting and correcting problems of lost packets, duplicated packets, or packets delivered out of order, is clearly visible and significant.

Note that this platform uses 100Mbit/s only, which results in a smoother increase of the RTD with the payload since the actual physical frame also increases linearly with the payload.

The standard deviation is larger in absolute terms when compared with the previous case but smaller in percentage of the average value, varying from 4% down to less than 1%. In fact, the measurements are relatively steady, which results from the stripped down structure of the Linux installation with less modules and services, particularly without data storage devices and graphical display, even without real-time patches. This is also clear in the maximum values observed, namely 1642µs with TCP and 1500B packets, representing 50% above the corresponding average value.

TABLE 3 – RTD ON LINUX EMBEDDED – RAW, UDP AND TCP SOCKETS – 1 GBIT/S

Payload [bytes]	Connected Directly					
	RAW		UDP		TCP	
	Mean [µs]	*STD [µs]*	*Mean [µs]*	*STD [µs]*	*Mean [µs]*	*STD [µs]*
46	214,1	8,8	378,3	7,7	514,9	8,5
64	221,6	8,3	379,3	10,1	519,7	5,9
128	239,3	9,4	412,6	21,0	544,7	6,2
256	281,5	9,7	439,9	8,2	585,2	6,5
512	360,2	8,0	527,6	8,5	688,5	23,5
1024	520,0	9,1	690,0	7,6	850,8	6,6
1500	678,6	8,1	1047,6	8,0	1094,1	10,3

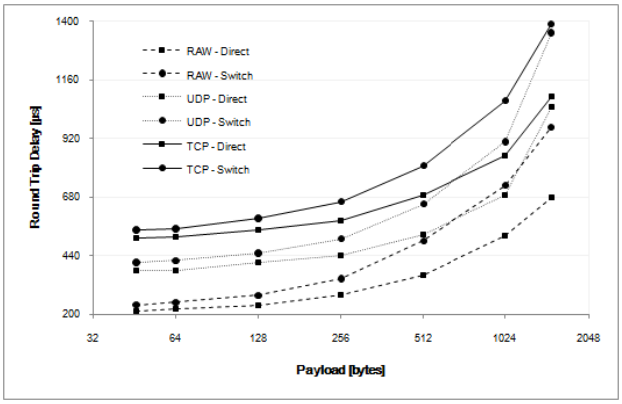

Fig.5 – RTD on Linux Embedded – RAW, UDP and TCP sockets

D. QNX Neutrino

The case of the QNX Neutrino OS is shown in Table 4 and Fig.6. This RTOS uses an enhanced IP stack that reduces the extra cost with respect to the BPF interface (equivalent to the RAW socket). Consequently, the obtained average values are very close in all cases. Note that these experiments are again carried out with the PC platforms and at 1Gbit/s. The average RTD values are comparable to those of the Ubuntu cases. However, beyond the shorter difference between the three interfaces, there is now a more perceptible increase in the RTD for the payloads between 46B and 512B, which is due solely to the protocol stack since the physical transmission is always done for 512B.

TABLE 4 – RTD ON QNX NEUTRINO OS – BPF, UDP AND TCP SOCKETS – 1 GBIT/S

Payload [bytes]	Connected Directly					
	BPF		UDP		TCP	
	Mean [µs]	*STD [µs]*	*Mean [µs]*	*STD [µs]*	*Mean [µs]*	*STD [µs]*
46	101,2	9,4	102,5	10,0	105,6	10,1
64	103,9	10,7	103,9	10,0	106,2	10,5
128	106,1	9,5	108,1	9,4	109,1	9,8
256	113,6	9,5	113,8	10,3	117,1	8,5
512	124,4	10,0	119,3	8,6	121,2	9,0
1024	142,0	9,9	138,1	8,6	139,3	9,4
1500	159,1	10,2	162,7	11,7	167,8	12,9

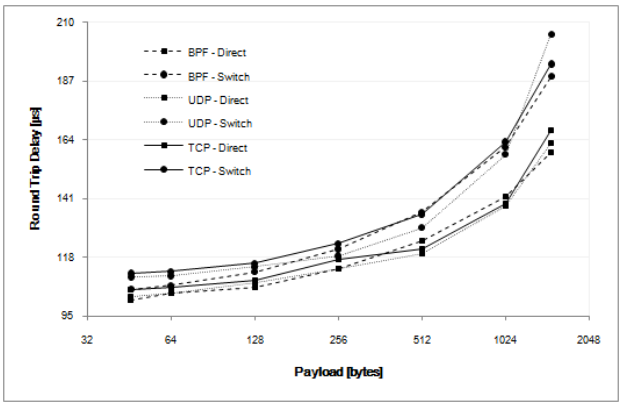

Fig.6 – RTD on QNX Neutrino OS – BPF, UDP and TCP sockets

Surprisingly, the standard deviation is the highest, relative to the average values (around 9%). This is due to frequent small variations around the average value, which might also be related to the discrete temporal resolution this kernel uses. The different patterns of RTD among the three cases running on PCs, namely Ubuntu standard and with real-time patches, and QNX Neutrino, are clear in Fig. 7.

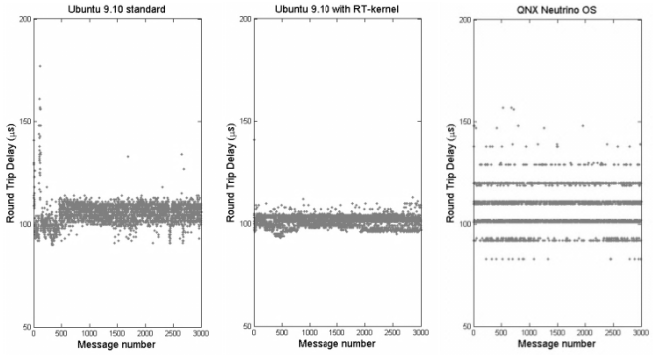

Fig. 7 – Different RTD patterns obtained with the three OSs on PCs.

Concerning the maximum values observed, they are less frequent but the magnitude is similar to the case of Ubuntu with RT patches. The absolute maximum delay that was measured was 296µs with UDP and 1500B packets, representing an overshoot of 81% with respect to the corresponding average value.

E. Protocol stack execution time

Using expression (3), we also inferred the time consumed by the protocol stack in each case (Fig. 8). At 1Gbit/s, for the Ubuntu and QNX experiments, the values were all very similar, roughly between 50µs and 85µs, growing with the payload, which is expected since larger payloads involve the transfer of larger amounts of data. At 100Mbit/s, with the slower platform running embedded Linux, the execution times are longer and exhibit a stronger difference for the different payloads and for the IP versus RAW sockets, which is compatible with the higher computational cost of handling larger frames and particularly within the IP stack.

However, perhaps more important than the absolute protocol stack execution times shown in Fig. 8 is the ratio with respect to the time it takes to send one packet of the respective

size. This ratio is shown in Fig. 9 where we can see that the protocol stack execution time is around one order of magnitude higher than the time needed to transmit one packet. Generally, this ratio decreases for growing packet sizes (at 100Mbit/s, and 1Gbit/s for packets above 512B), which means that the overhead grows less than linearly with this variable.

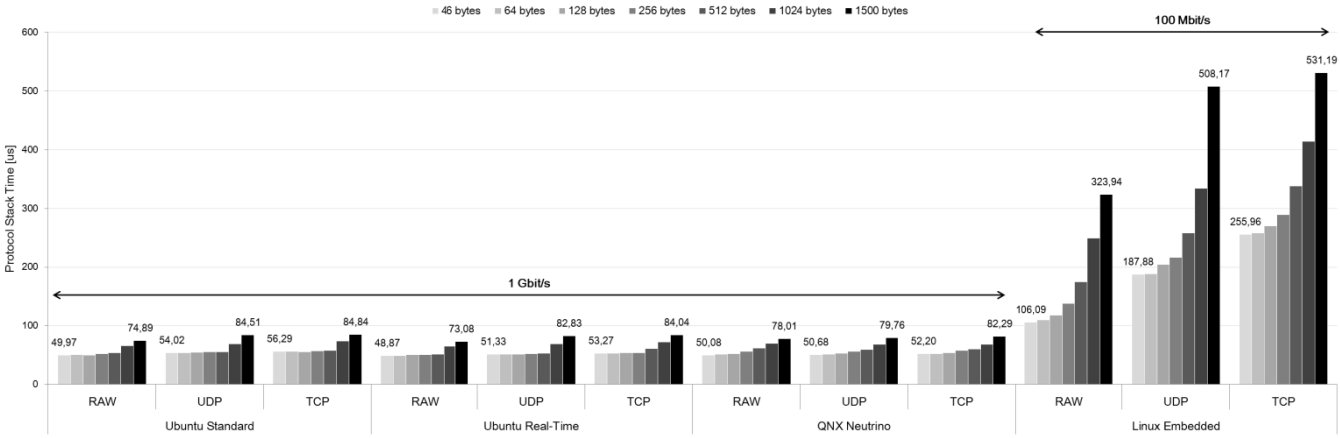

Fig. 8 – Time consumed by the protocol stack for each OS, type of socket used and payload length

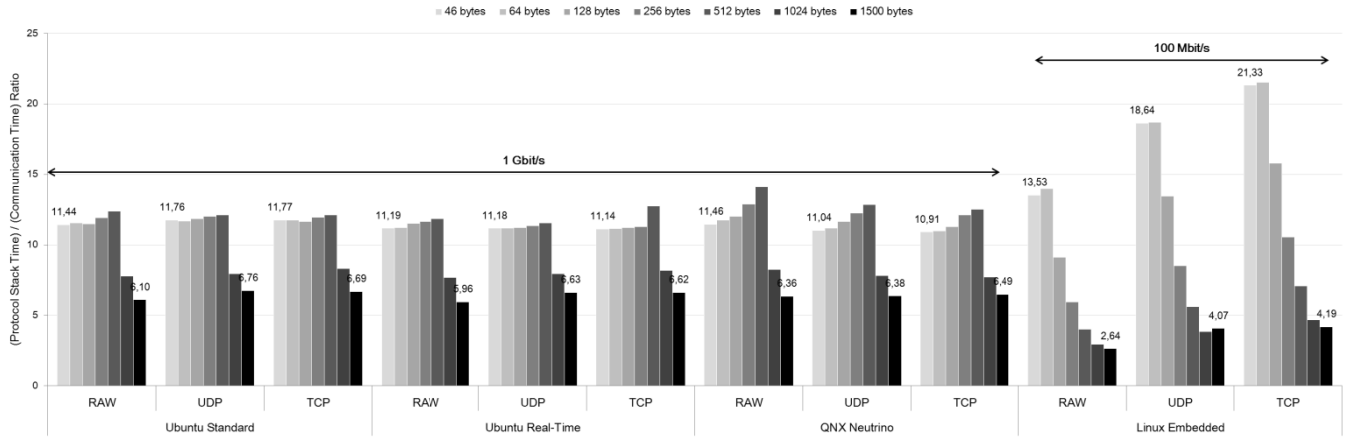

Fig. 9 – (Protocol Stack Time) / (Communication Time) ratio for each OS, type of socket used and payload length

F. Global comments on the results

Looking globally at the experimental results we can say that for fast computing platforms, such as current COTS PCs, the extra delay imposed by using the IP protocol stack is minimal and thus, UDP sockets might be preferred to RAW sockets given the higher abstraction level that they provide. This, however, is not true for slower computing platforms, e.g., an AVR32 at 140MHz for which there is still a significant additional delay incurred by the IP stack.

On the other hand, full featured general purpose OSs without any real-time patch exhibit very strong worst-case deviations in all kinds of sockets. This situation can be improved either by using real-time kernel patches, real-time operating systems, or by stripping down the OS to include the minimal modules and services that are effectively required by the application, particularly when data storage devices and graphical displays are not used.

The TCP stack also includes an inevitable potential for large delay variations due to retransmissions, mutual impact of different concurrent connections or simply due to the complex control flow of its own code.

Finally, the results shown in this paper present lower bounds for the delays incurred by the protocol stack and the impact of the OS in the end nodes, thus representing the best performance that can be achieved. In common application scenarios, there will be further interference in the execution of the protocol stack and the application code, leading to further delays and higher variations.

G. Relevance fordesign and education

The work presented in this paper serves the purpose of raising the awareness of the impact that the operating system and protocol stack have on the time taken to communicate with Ethernet. As it was shown in the previous section, the protocol stack may take one order of magnitude more time to execute than the time needed to actually transfer the respective packets.

Certainly, in real systems there will be further delays in the end nodes, caused by preemption and interruptions, and in the wire, due to interference from other traffic. But in any case, except when the network delays grow substantially, e.g. due to high load, the protocol stack will take a rather significant time to execute, incurring in a significant overhead that cannot be overlooked when assessing the network-related delays.

This awareness is particularly relevant for distributed embedded systems designers who can use the results in this paper to better chose a suitable combination of protocol stack / OS, with the right balance between level of abstraction, reliability, timeliness and overhead. However, we just provide guidelines. In fact, we cannot provide a model to estimate the overhead with any kind of platform and specific measurements must be done for each case. We believe, nevertheless, that our results are representative for a relatively wide range of platforms.

Therefore, in order to maximize the awareness to this effect, we believe these experiments can positively integrate courses on embedded systems design. In fact, the experiments carried out in this work can be easily replicated in a laboratory on computer networks, on real-time networks, or even on distributed systems, allowing to observe the protocol stack / OS overhead. Along two to three 2h lab classes, the students can learn about sockets of different types, namely RAW, UDP and TCP, and write communicating code that can run on different OSs using the sockets API. Then, a basic introduction on time measurements will allow assessing the Round Trip Delays. These measurements must then be subject to an adequate statistical treatment in order to obtain the desired temporal parameters. We believe that such a set of lab assignments can raise the awareness to the impact of protocol stack / OS on the network delay and lead later on to the design of more efficient Ethernet-based embedded systems.

V. CONCLUSIONS

Ethernet continues being a widely used computer communications protocol that offers interesting features from relatively low cost to high bandwidth. There is also a growing interest in using this protocol in embedded systems, from airplanes to cars, which are frequently subject to timing constraints. In such case, the end-to-end delays associated with the data transfers are of major importance. However, these are influenced not only by network parameters, but also by features of the end nodes, namely related to the operating system and protocol stack. In this paper we have analyzed the impact of these two features and carried out an experimental characterization of the best achievable performance with three different protocol stacks, accessed with RAW, UDP and TCP sockets, and four different standard and widely available operating systems, namely an Ubuntu distribution with standard Linux and another with the kernel real-time patches both running on COTS PCs, an embedded Linux running on a resource-constrained platform and QNX Neutrino also running on COTS PCs.

The results obtained allow discriminating the situations in which the extra overhead of UDP sockets is negligible with respect to RAW ones, what is the benefit of using a real-time OS, and the kind of variation of the end-to-end delay that we can expect in each of the tested cases. We believe that this information can be useful for distributed embedded systems designers as well as for embedded systems education, to raise awareness to the sources of delay in communications that might be hidden to the non-expert user.

VI. REFERENCES

[1] Decotignie, J.D., Ethernet-Based Real-Time and Industrial Communications. Proceedings of the IEEE, 2005. 93(6): p. 1102-1117.

[2] Meyer, E.E. ARPA Network Protocol Notes. [Online] 1970 [cited 2010 April]; Available from: http://www.faqs.org/rfcs/rfc46.html.

[3] Zimmermann, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. in IEEE Transactions on Communications. 1980.

[4] Kozierok, C.M. The TCP/IP Guide. [Online] 2005 [cited 2010 April]; Available from: http://www.tcpipguide.com/free/index.htm.

[5] Wikipedia. User Datagram Protocol. [Online] [cited 2010 April]; Available from: http://en.wikipedia.org/wiki/User_Datagram_Protocol.

[6] Dunkels, A., Full TCP/IP for 8-bit architectures, in Proceedings of the 1st international conference on Mobile systems, applications and services. 2003, ACM: San Francisco, California. p. 85-98.

[7] Zhigang, H., Y. Yansong, and S. Ninghui. High performance Sockets over kernel level virtual interface architecture. in High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on. 2005.

[8] Mohamed, N., J. Al-jaroodi, H. Jiang, and D. Swanson. A user-level socket layer over multiple physical network interfaces. in 14th International Conference on Parallel and Distributed Computing and Systems. 2002. Cambridge, USA.