Implementing Real-Time Communication on a Token-Ring Network

Frank Rößler and Andreas Kühlmeyer

University of Kaiserslautern, Germany
Computer Networks Group

E-mail: {roessler, akuehlm}@informatik.uni-kl.de

Abstract. We have developed a real-time communication service running on PCs interconnected with a conventional token-ring local area network. The implementation was guided by main concepts of quality of service architectures proposed in the literature. It is shown how these concepts are to be developed into a real implementation and how restrictions regarding the run time platform and the underlying network service influence performance results and predictability.

1 Introduction

Due to evolving high-speed transmission and switching technologies it is basically possible to meet ever-increasing user requirements on throughput and end-to-end delay. As a consequence traditional network services such as remote login or file transfer nowadays have enough network resources available to perform properly without any kind of resource management. However, on their way from the sending application to the receiver side different end-to-end connections (flows) contend for the same network resources and thus mutually influence delivered quality of service. Therefore modern applications with great demands on network resources will dramatically degrade in performance when resource sharing is not suitably restricted.

We have developed a real-time communication service, which regulates resource sharing and runs on PCs (under the real-time operating system QNX) interconnected with a token-ring local area network. The implementation was guided by main concepts of quality of service (QoS) architectures currently proposed in the literature. Thus a system architecture has evolved that comprises typical mechanisms for QoS provision as well as QoS control. QoS provision regulates network load by the means of a resource reservation protocol, which signals QoS requirements across the network and initiates admission control as well as resource reservation. These activities are performed during the establishment of real-time connections. During data transfer proper timing behaviour is enforced by applying traffic control and scheduling mechanisms when reserved resources are assigned to individual packets.

The remainder of the paper is organized as follows: Section 2 describes the main aspects regarding structure and behaviour of our QoS architecture while implementation issues are discussed in Section 3. To evaluate our approach end-to-end performance measurements were conducted and timing behaviour inside the network driver was studied. We summarize the results in Section 4. Section 5 gives some concluding remarks.

2 Quality of service architecture

On principle, a QoS architecture aims to separate mutual interaction of different flows w.r.t. quality of service. This section introduces the QoS architecture we implemented on top of a conventional token-ring local area network. We assume that the reader is already familiar with basic token-ring operation.

2.1 Service model

Currently our service model supports unidirectional, unicast connections with bounded delay and guaranteed throughput where the traffic load is constrained by maximum packet size and minimum

6th Open Workshop on High Speed Networks, Stuttgart, Germany, 1997

packet interarrival time. Connections are established and closed dynamically while only the calling user is allowed to close a connection. Interaction at the service interface is quite simple: for establishing a connection the calling user has to specify the required QoS (maximum packet delay) including the expected amount of traffic load (maximum packet size, minimum packet interarrival time), the receiver's node address, and the receiver's port number. On success the service provider returns a connection identifier, which is to be used for later send, receive and close requests. Before a connection can be established the called user must have opened a local port in order to register with the communication service. A connection request may be refused due to lack of resources or an unknown receiver. Note that for each connection the required maximum packet delay must not exceed the specified minimum interarrival time. In order to reduce complexity we assumed that the host processor, i.e., the CPU on which the application processes and the network driver run, suffers no capacity shortage[1]. As a consequence guarantees have to be withdrawn under heavy processor load as long as processor scheduling is not integrated into our QoS architecture.

2.2 Resource reservation protocol

A token-ring builds the trivial case of a packet-switched network where no intermediate systems exist. Thus the resource reservation protocol does not need to fix a route. Nevertheless, traffic loads are to be announced to the communication system, and the availability of sufficient network resources must be checked prior to data transfer. For this purpose we introduced a central resource

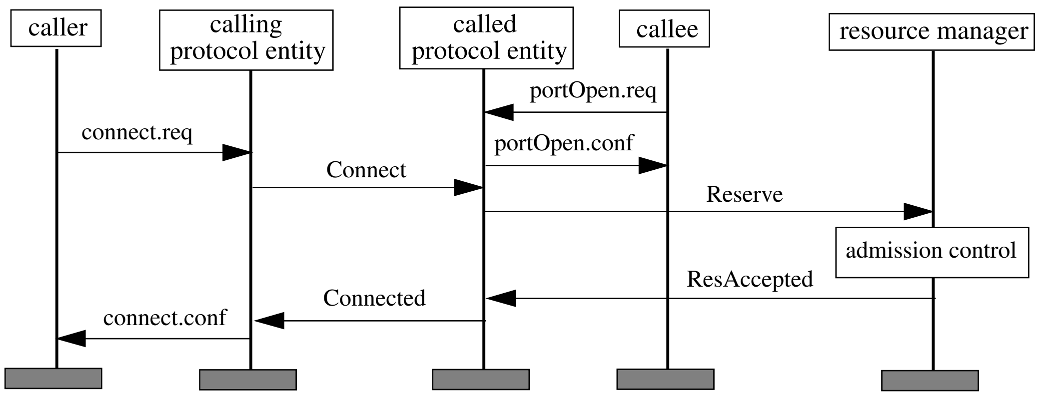

Figure 1: connection establishment (QoS provision)

manager residing on a dedicated network node. The protocol operation for the establishment of a new real-time connection is illustrated in Figure 1. Essentially, the calling protocol entity (when invoked by the user application) requests a connection from the called protocol entity, which itself requests necessary resources from the resource manager. As a prerequisite the receiving application must have registered with the called protocol entity by opening a port (the connection would be refused otherwise). Additional protocol functionalities (not shown in Figure 1) are error control to ensure reliable transfer of control messages and connection tear down.

2.3 Flow shaping

In general flow shaping enforces that flows behave as contracted, because predictable timing behaviour is only achievable, if arrival patterns at each resource comply an expected (from the scheduler's viewpoint) traffic load. Packets of a particular flow that arrive early are delayed, dismissed, or otherwise punished (under certain circumstances some scheduling disciplines allow processing of early packets without violating other performance guarantees). As a consequence flow shaping ensures throughput guarantees and protects the scheduler from overload situations (even transient). Sometimes flow shaping compensates for delay-jitter by completely reconstructing the original traffic pattern as observed at the network entry points. Note that violation of user

[1]. Section 4 will justify this assumption to some degree.

specified traffic load may either happen because of varying network load, which inevitably distorts traffic patterns, or misbehaving users.

According to Section 2.1 application processes are not allowed to issue send requests with a higher frequency than agreed on. If a particular traffic contract is violated, the send request is delayed until the specified interarrival time has been completely elapsed. As the token-ring is assumed to be the only bottleneck it is sufficient to control traffic load at the network driver interface (Figure 2).

2.4 Scheduling discipline

The scheduling discipline resolves contention of the different flows for a particular resource (e.g., CPUs or network interface). As the scheduler orders the processing of the packets from all traversing connections it imposes individual waiting times on the different flows. For this purpose various scheduling disciplines have been conceived, for instance, rate-monotonic-scheduling, earliest-deadline-first, virtual-clock, weighted fair queueing, stop-and-go or hierarchical round robin.

In our case scheduling is performed jointly by token-ring medium access control (MAC) and the driver software. Though media access is priority-driven[1] our token-ring hardware serves locally buffered frames in FIFO order, i.e., global priority arbitration is applied only among the frames at the head of the adapter queues. Because of this insufficiency our driver software additionally manages a four-level priority queue and hands packets in a one-by-one mode to the hardware (Figure 2). In cooperation with the priority-driven media access this realizes - within certain limitations - a global fixed-priority scheduling of token-ring frames, where preemption of ongoing transmissions is not allowed. Fixed-priority scheduling is well studied in the literature [2] so that we can apply existing results to assess predictability and timeliness of the approach. To complete the scheduling

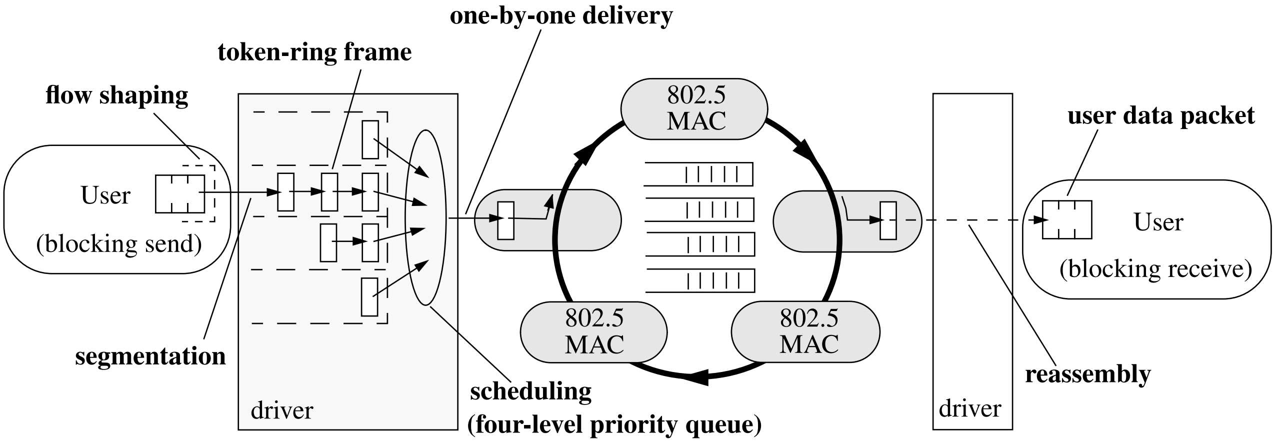

Figure 2: data transfer (QoS control)

discipline we have to determine how priorities are assigned to token-ring frames. First of all, frames that belong to the same connection are assigned the same priority. This priority level is determined during connection establishment. Furthermore, we employ deadline monotonic priority assignment, i.e., connections with shorter maximum packet delay get higher priorities than connections with longer delays. For deadline monotonic priority assignment Leung and Whitehead [4] have shown the optimality of this strategy under fixed priority scheduling with deadlines at or before the end of the task arrival period. This priority assignment strategy is employed for both the driver scheduling and the token-ring MAC, i.e., frames keep their priority on the way through driver and network adapter.

The chosen token-ring adapter provides only 4 user priority levels. Thus more than one natural connection priority (according to strict deadline monotonic scheduling) must be grouped into the

1. Selection of the next frame to be transmitted is based on a special, circulating frame called token, which gives permission to use the medium and realizes global priority arbitration by the means of priority and reservation fields.

same system priority. As priority reconfiguration causes some overhead, when ongoing data transfers must not be disturbed, we a priori determined four intervals of delay values that were monotonically assigned to the available priority levels.

2.5 Admission control

Consider that at some point in time the overall network traffic load consists of a set of n connections with interarrival times T_i and maximum packet delays D_i . Under deadline monotonic scheduling there is a necessary and sufficient condition, which ensures that each data packet will be transmitted before its deadline has elapsed [3]. This schedulability test is shaded below, where C_j denotes the processing time of data packets.

$$\forall\, i \;\; 1 \le i \le n. \quad \min_{0 < t \le D_i} \left(\sum_{j=1}^{i} \frac{(C_j + \mathrm{Overhead}_j)}{t} \left\lceil \frac{t}{T_j} \right\rceil + \frac{\mathrm{Blocking}_i}{t} \right) \le 1$$

Schedulability results for ideal deadline monotonic scheduling, however, assume perfect preemption, immediate recognition of transmit requests, a separate priority level for each connection, and that priority inversion does not occur. As these assumptions do not hold for a token-ring this schedulability test has to be adapted, which results in an elaboration of the *Overhead* and *Blocking* terms [5]. Overhead, e.g., includes the transmission of frame headers in addition to user data. On the other hand blocking, e.g., takes into account that frames may be delayed due to an ongoing transmission of lower priority or due to the limited amount of priority levels [1]. Note that a trade-off between preemtability and protocol overhead resulted in the definition of a maximum frame length, which requires segmentation and reassembly inside the driver (Figure 2).

3 Implementation issues

The software structure resembles the developed QoS architecture as we mapped QoS provision (resource reservation protocol, admission control test) and QoS control (scheduling inside the token-ring driver) on separate operating system processes (called *protocol machine* and *driver* in the following). As a third component the driver software comprises a set of library routines for initialization, opening and closing connections as well as user ports, sending and receiving messages, and reporting network status, which are linked into the user's program. Note that flow shaping is performed during the send procedure call in the user's program (Figure 2).

For establishing and closing connections user processes forward service requests to the protocol machine via blocking IPC. Subsequently, the protocol machine creates a proper protocol data unit and hands it (also via blocking IPC) to the driver. Note that the user process leaves blocking state not before the service request is completely processed, whereas the protocol machine only waits until the driver has transmitted the PDU on the network. Incoming PDUs are transferred asynchronously from the driver to the protocol machine. For data requests a user process interacts directly with the driver process (again via blocking IPC).

Figure 3 illustrates the interaction between the driver process and the token-ring adapter. It is shown how one-by-one transmission of token-ring frames is realized: when a frame has been transmitted the adapter issues a *frame sent interrupt*. Subsequently, the driver writes the highest priority frame into a DMA frame buffer in system memory and interrupts the adapter (*transmit valid interrupt*). Finally, the adapter reads the frame via direct memory access and transmits it after a free token has been captured. Note that receive operation is similarly triggered by the means of *receive valid interrupts* and *frame received interrupts*.

It turns out that the driver process generally performs a main loop where essentially frame received and frame sent interrupts, send and receive requests from user processes, as well as send requests from the protocol machine are processed.

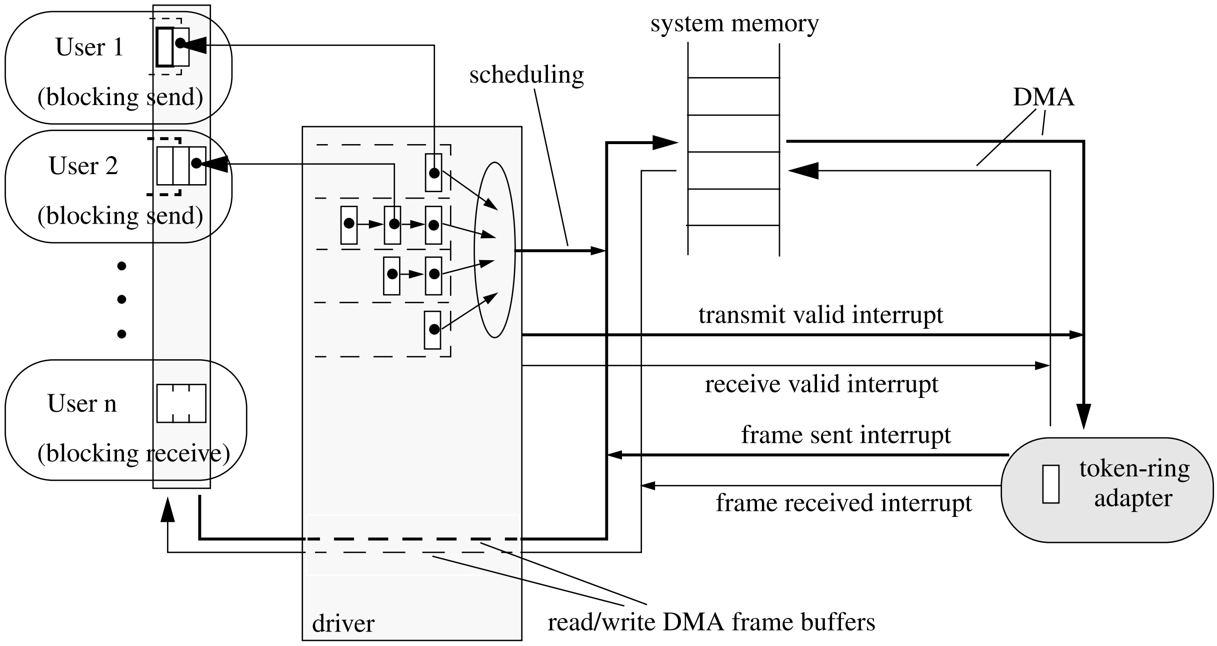

Figure 3: transmit and receive operation (one-by-one mode)

4 Performance measurements

We applied two different approaches for performance measurement. First we conducted empirical tests in the sense of test applications that used the token-ring driver. Thereby we were able to determine general limitations of the driver and the hardware. Then we took a deeper look inside the driver in order to find out how the empirical results are to be explained and to assess the efficiency of our real-time-capable driver compared to an ordinary driver, which drops predictability and uses all capabilities of the adapter to achieve the highest possible transfer rate.

4.1 Empirical tests

By empirical tests we examined how many real-time connections actually worked regardless of the limitations of admission control. Therefore we defined three different types of connections (Table 1). Note that the gross transfer rate includes 27 bytes of protocol overhead. Fast connections repre-

name	minimum interarrival time	maximum packet delay	packet size	gross transfer rate
fast	10ms	10ms	10 bytes	29.6 kbps
medium	28ms	28ms	230 bytes	73.4 kbps
slow	100ms	100ms	230 bytes	20.6 kbps

Table 1: connection types

sent controlling processes, e.g., critical industrial processes and medium connections correspond to audio connections with telephone quality (64kbps). The driver was configured in a way that the different types of connections were linked to different token-ring priorities. We checked different scenarios - some with only one type of connection, some with several types - and tried to find the limits by increasing the amount of connections until QoS guarantees were violated.

To compensate for blocking send requests we distributed the connections to several processes. We used up to five PCs (three as senders and two as receivers) to drive the token-ring to the theoretical limit of 4 Mbps and modified the driver to ignore admission control, which was necessary to explore the limits. Together with the real-time connections we started a workload generator on every sending node, which used a non-real-time connection to flood the net as much as possible. Because of the large amount of tests we conducted, we only present some results here.

First we used only one sending node, where we could successfully establish up to 24 connections of the medium type distributed to 3 processes, giving a gross rate of 1762 kbps. The workload generator could only send about 20 kbps, so the system was really at its limits. Other tests confirmed that about 1.8 Mbps was the maximum gross rate the driver could transmit on a single node (see Section 4.2). With fast connections one process only handled up to 2 connections. We successfully established a total of 12 connections, giving a gross rate of 355 kbps, where the workload generator was still able to transmit 450 kbps.

Next, we employed all three sending nodes. Table 2 shows the results, where for each node the

connection type	node 1	node 2	node 3	total gross rate	total workload
fast	12 (6)	12 (6)	6 (3)	888 kbps	1519 kbps
medium	15 (3)	15 (3)	15 (3)	3304 kbps	489 kbps
slow	60 (3)	60 (3)	60 (3)	3701 kbps	58 kbps

Table 2: empirical results using several nodes

number of possible connections is given together with the number of processes (in brackets). The three sending nodes are capable of consuming all the capacity of the token-ring (4 Mbps) as every node can send as much as 1.8 Mbps. The tests have shown that large block sizes (slow and medium connections) yield a utilization by real-time data of up to 92 % where the rest is consumed by the workload generator and due to the fact that a free token must be forwarded after each frame transmission (one-by-one transmission mode). For smaller block sizes considerably less is achieved. As the next section will show this is because the maximum frame rate is reached earlier than the maximum bandwidth of the token-ring. Note that the missing fast connections of node 3 (one would expect 12 instead of 6 as in the case of node 1 and 2) are due to the limited number of receive nodes.

To complete the empirical tests we also established a mix of slow and medium connections to see how the different types of connections interact. We could, e.g., establish 7 medium and 30 slow connections on each node, giving a total gross rate of 3392 kbps, while the workload generator reached 358 kbps.

4.2 Precise time measurements

Two types of time measurements were performed, namely from outside and inside the driver. First we took the viewpoint of an application and measured the time taken by a send request. Then we monitored exactly inside the driver where time is consumed when a packet is sent. We used a special performance monitoring register of the used Intel Pentium CPU called *time stamp counter*, which allows very precise time measurements with the granularity of the CPU clock (in our case 1/90μs). Every measurement took about 1μs and therefore did not influence the driver noticeably. We ran the tests several times and used the minimum time required, so that we could eliminate the disturbing effects of other running (e.g., system) processes. Data packets were sent in a way that the flow control mechanism never delayed the sending process. Furthermore, the packets were transmitted immediately as no other sending process was active.

block size (*byte*)	4	10	30	60	120	180	232	233	300	464	465
time outside (μs)	649	658	682	760	911	1065	1197	1800	1920	2343	2945

Table 3: results of time measurements from the outside

Table 3 summarizes results of the time measurements from the outside. The time needed for sending a block was at least 649μs even for very small blocks of 4 bytes. The time increased linear with the size of the data blocks up to a size of 232 bytes. This is the preconfigured maximum size of a single token-ring frame. Blocks of 233 bytes and more are split up into two frames. This explains the large gap between 232 and 233 bytes. The same effect is visible between 464 bytes (2 frames) and 465 bytes (3 frames). Applying linear regression it turned out that the time spent for a send request is as follows:

$$T_{outside} = 23\ \mu s + 2.44\ \mu s/byte * blocksize + 600\ \mu s * \#frames$$

As every frame needs at least 623µs we have a maximum rate of about 1600 frames per second. The 23µs match the time QNX needs for IPC between the application and the driver. The highest possible transfer rates are achieved with a block size of a multiple of 232 bytes, giving a rate of at most 1.59 Mbps and a gross rate of 1.78 Mbps. To get a better understanding of the formula the driver was further examined from the inside. Time measuring was started when the driver received a packet from the application and stopped when the driver returned the answer. Table 4 shows the results. It turns out that the flow control

block size (*byte*)	4	232	233	464	465
time inside the driver (µs)	549	1096	1696	2239	2841
time outside (µs)	649	1197	1800	2343	2945

Table 4: time spent inside the driver

mechanism in the send routine and the communication between the application and the driver need about 100µs. Next we measured the time the token-ring adapter needs to transmit the frames. Therefore the time between a transmit valid interrupt and the corresponding frame sent interrupt (Figure 3) was observed. Table 5 shows the results. It turns out

block size (*byte*)	4	232
time for adapter (µs)	531	1072
time inside the driver (µs)	549	1096

Table 5: time taken by the adapter

that most of the time actually is required by the token-ring adapter. Optimizing the software would be useless as the limiting factor is the adapter itself.

By comparison with the maximum transfer rate the adapter can reach without QoS control we evaluated how the unfavourable one-by-one transmission mode affects the performance of the adapter. Therefore the driver was modified to use normal transmission mode. *M* frames of 230 bytes were provided to the adapter and subsequently, the transmission time consumed by the adapter was measured. Table 6 shows the results. Note that the

M	1	2	3	5	7	9	
time (µs)	1065	1969	2876	4683	6491	8300	161+*M**904

Table 6: normal transmission mode

given time corresponds to the time the adapter consumes (as in Table 5). The adapter initially needs 161µs and additional 904µs per 230-bytes frame[1]. Furthermore, smaller frames were examined (Table 7). It turned out that in normal transmission mode the driver needs an initial time of 168µs and additional 315µs for each frame plus 2.55µs per byte. Thus the maximum transfer rate for normal transmission mode using frames of 232 bytes would be 232 bytes / (315µs + 232 * 2.55µs) = 2.05 Mbps (gross rate: 2.29 Mbps). Com-

[1]. Together with the 100µs for the IPC this yields 1165µs, which corresponds quite well with the predicted $T_{outside}$=1184µs.

pared to the rate for one-by-one transmission mode (net 1.59 Mbps, gross 1.78 Mbps) normal transmission mode leads to better transfer rates per node.

block size *(byte)*	**time** (µs) **for** M=1	**time** (µs) **for** *M*=5	**time** (µs) **for** *M*=9	**linear regression**
50	611	2400	4190	163+447**M*
100	737	2988	5243	173+563**M*
150	867	3649	6432	171+695**M*
200	993	4304	7615	165+828**M*
			overall:	168+*M**(315+2.55**B*)

Table 7: normal transmission mode, various frame lengths

5 Conclusion

Our approach classifies as an open-loop approach to resource management, i.e., QoS demands are to be guaranteed a priori and under worst case assumptions. On the other hand we applied a network adapter, which turned out to be not well suited for resource management, although a token-ring adapter conforming to the IEEE 802.5 standard could be. For example, it is not possible to configure the token-ring holding time and the adapter serves buffered frames in FIFO order, thereby forcing us to one-by-one transmission mode. As a consequence token priority may be reduced between successive high priority transmissions and a node with only non-real-time connections established is able to receive the token regularly and capture large portions of the network bandwidth by low priority traffic. A countermeasure would be to additionally restrict non-real-time traffic in situations where no real-time connections are established locally. The various interferences resulted in a rather low utilization of the available network bandwidth. Thus resource management for hard real-time requirements needs support even from the lowest hardware level. However, mechanisms for QoS monitoring and maintenance during data transfer as well as reduced obligation of guarantees can improve the situation.

Section 2 assumes that the host processor suffers no capacity shortage, i.e., operating system scheduling was not integrated into the QoS architecture. During performance measurements we carefully avoided processor overload, which is not always possible in a real application setting. For example, simulation runs of a control application on top of the real-time token-ring quickly revealed this insufficiency. It is well-known that end-to-end guarantees need an integrated approach including communications subsystem, operating system, as well as distributed application level.

References

[1] D. I. Katcher, S. S. Sathaye, and J. K. Strosnider, “Fixed Priority Scheduling with Limited Priority Levels”, IEEE Transactions on Computers

[2] M. H. Klein, T. Raliya, B. Pollak, and R. Obenza, “A Practitioner’s Handbook for Real-Time Analysis”, Kluwer Academic Publishers, 1993

[3] J. Lehoczky, L. Sha, and Y. Ding, “The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior”, Real-Time Systems Symposium, 1989

[4] J. L. T. Leung and J. Whitehead, “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks”, Performance Evaluation (Netherlands), 2(4), 1982

[5] S. S. Sathaye and J. K. Strosnider, “Conventional and Early Token Release Scheduling Models for the IEEE 802.5 Token Ring”, Real-Time Systems Journal, 7, 1994