A Time-Triggered Ethernet Protocol for Real-Time CORBA

Stefan Lankes, Andreas Jabs
Lehrstuhl für Betriebssysteme,
RWTH Aachen, Kopernikusstr. 16,
52056 Aachen, Germany
E-mail: {stefan, jabs}@lfbs.rwth-aachen.de

Michael Reke
VEMAC GmbH & Co. KG
Krantzstr. 7, Halle 33A,
52070 Aachen, Germany
E-mail: reke@vemac.de

Abstract

The Real-Time CORBA and minimumCORBA specifications are important steps towards defining standard-based middleware which can satisfy real-time requirements in an embedded system. These real-time middlewares must be based on a real-time operating system (RTOS) and a real-time network. This article presents a new time-triggered ethernet protocol that has been implemented under RTLinux. Furthermore it describes a Real-Time CORBA implementation called ROFES, which is based on this new real-time network.

1. Introduction

First generation real-time applications were running on single processor environments since the problems to be solved were relatively simple. Nowadays, applications like avionics, telecommunication, process control and distributed interactive simulation need real-time properties in a distributed environment [2]. Middlewares like *CORBA* [10] and *DCOM*[1] help to improve the flexibility, extensibility, maintainability and reuseability of distributed applications. Because these middleware architectures were not designed for real-time applications, the *Object Management Group* has specified a real-time extension for CORBA [11] and called it *Real-Time CORBA*. A first implementation of a Real-Time CORBA is *TAO*[2] from the *Center for Distributed Object Computing* at Washington University. We are developing another version of Real-Time CORBA especially for embedded systems and call our implementation *Real-Time CORBA for embedded Systems*[3] (ROFES) [8]. In order to achieve this aim, we are developing a completely new Real-Time CORBA implemantation.

[1] http://www.microsoft.com/com/tech/dcom.asp
[2] http://www.cs.wustl.edu/~schmidt/TAO.html
[3] http://www.lfbs.rwth-aachen.de/~stefan/rofes

To serve deterministic requirements, real-time middlewares must at least be based on a *real-time operating system (RTOS)*. However, the cost aspect and the acceptance of operating systems like Windows NT, Sun Solaris and Linux have generated a need for real-time functionality in these operating systems. Many companies and universities are developing real-time extensions for Windows NT and Linux. A very interesting real-time extension for Linux is the *RTLinux* project [1] from the New Mexico Institute of Mining and Technology. They implemented (see figure 1) a simple real-time kernel underneath the operating system, with Linux itself running as just one task within that real-time kernel. Linux runs at the lowest priority and can be preempted at any time by real-time tasks with a higher-priority. The design philosophy of RTLinux was to minimize the changes made to Linux itself, providing only the necessary essentials for implementing real-time application.

Beside an real-time operating system, real-time middlewares like ROFES also need a real-time network. [5] and [6] show that a time-triggered protocol is the better choice for a hard real-time system than an event-triggered protocol. For this reason, we decided (see [13]) to use a time-triggered protocol for our real-time network. RTLinux exhibits very good real-time characteristics and gives us a good platform to implement a software-based time-triggered protocol for existing network adapters. The cost aspects for an embedded system are very important. Therefore our real-time network uses ethernet network cards because ethernet is a very popular, sophisticated and low-cost technology.

This article is organized as follows: Section 2 summarizes the basic principles of real-time networks. Before the actual implementation of ROFES is described in the succeeding section 4, the article presents the new ethernet-based real-time network in section 3, which is used by ROFES. Section 5 presents and discusses our first results. Finally, some general assessments of the lessons learned are provided and some conclusions are drawn in section 6.

2. Networking under Real-Time Conditions

Real-time networks have to meet special requirements, to get a temporal deterministic behavior. While in standard LANs bandwidth is most important, under real-time conditions a small jitter in protocol latency is much more desired. Applications trust in this well repeatable transmission time, and a bad jitter basically destroys the real-time performance of the whole system.

The jitter is defined as the maximum deviation in protocol latency within a supervised interval of time and N values for the latency $t_{Li}, i = 0..(N-1)$

$$jitter = max(t_{Li}) - min(t_{Li}) \quad (1)$$

In a real-time network, the jitter in protocol latency should remain within given bounds, only depending on the temporal synchronization of the network members and be independent of the network load.

In [5] Kopetz shows that a time-triggered network with its *time division media access* strategy (TDMA) is used best to keep the jitter small.

2.1. Known Architectures

Today various network architectures for real-time-systems are in use. For many applications the *Control Area Network* (CAN) is deployed, because of its priority-based access. But under hard real-time conditions, communication is restricted to periodic data by using *Rate Monotonic Scheduling* [17], which was first introduced by Liu and Layland in [9]. The deadline for each send operation is implicitly given by its period. The network utilization factor has to be below its maximum, which can be calculated by

$$\rho_{max} = n(\sqrt[n]{2} - 1) \quad (2)$$

$$\lim_{n\to\infty} \rho_{max} = ln(2) \approx 0.69 \quad (3)$$

where n is the number of connections. If there is a need for deadlines earlier than the next period, this system cannot be used.

An example of a time-triggered network access is the *Time-Triggered Protocol for SAE Class C Applications* (TTP/C), which was devoleped at the TU Vienna and is now provided by TTTech. It controls the exchange of messages between different electronic modules to a TTP/C network. In a time-triggered architecture, the communication system decides autonomously (according to a static schedule) when to transmit a message. Every controller contains its own control data, which is stored in a personalized message descriptor list (MEDL), that specifies the instant a message has to be transmitted by the controller. A TTP/C network consists of electronic modules that are connected by two replicated channels. One or more electronic modules can be combined to a fault-tolerant unit (FTU). Access to the bus is controlled according to a cyclic *time-division multiple access* (TDMA) scheme derived from a notion of global time. The sequence of slots in which each electronic module sends at most one message forms a TDMA round. After a TDMA round has completed, the next TDMA round, with the same temporal access pattern but possibly different messages, is started[16].

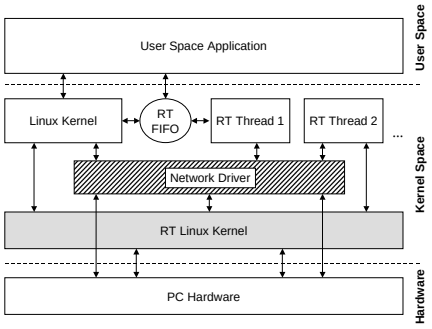

Figure 1. Parallel access to the time-triggered Ethernet driver

3. Design and Implementation of the Ethernet-based Real-Time Network

We built up a network, which based on the *time division media access* (TDMA), in a PC environment using usual Ethernet-adapters in combination with a driver for a RTOS. This pure software realization has to be seen in contrast to other time-triggered networks, which are based on costly special hardware solutions [16][5]. For accomplishing the temporal requirements RTLinux is used as RTOS. The whole network control is abstracted in one kernel module, which can be loaded dynamically into the kernel. This driver (see figure 1) provides parallel access to the network for real-time applications, which are also implemented in kernel-modules under RTLinux, and for non real-time Linux applications. This implementation continues the design principle of Real-Time-Linux with its parallel real-time threads and non real-time Linux applications.The non real-time Linux applications use the usual network API and are handeled by the driver like a RTLinux application with the lowest priority. Therefore every time-slot, that is not used by the real-time system, is available. The network communication is done in priority-based order.

The Ethernet protocol (*Carrier Sense Multiple Access / Collision Detection* (CSMA/CD)) is implemented in the hardware of each *Network Interface Card* (NIC). In order to get real-time performance on Ethernet, collisions have to

be avoided. Because of the random retransmission time periods, CSMA/CD provides a non determined temporal behavior. In our network each node has exclusive access to the network within its scheduled time slot. This guarantees that no collision occurs and the hardware-based Ethernet protocol is de facto transparent.

3.1. Temporal Synchronization

In a time-triggered network, each node has to be synchronized to a global time to determine the absolute beginning and end of the time slots. The jitter of the network communication depends directly on the accuracy and precision of this global time. Because of this, it is very important to figure out an useful algorithm for the temporal synchronization. In our case, the local time within a node is based on the operating system's clock, which itself is based on the processor's clock. In comparison to a perfect clock, typical maximum drift rates of these physical PC oscillators are between 10^{-2} and $10^{-7}\frac{sec}{sec}$ [5]. For this reason, clock synchronization has to be done during the whole runtime, even if there is no misfunction of the oscillator.

With this work a kind of continuous, precise and simple software-based synchronization is introduced. In our network, one periodic thread does the send and receive operations on the NIC. The synchronization algorithm controls the period of that thread to compensate offset and drift-rate of the node in comparison to the global time. The global time is detected by evaluating the arrival times of the incoming data packets. Ramanathan classified in [12] the known synchronization algorithms into three different categories: software, hardware and hybrid synchronization. Our algorithm realizes a software synchronization but uses the principles of the hardware synchronization (PLL). It does a continuous internal synchronization of each nodes clock to the global time, which is provided by a master. An external synchronization as described in [7] should be done in future work, i.e. by deploying the NTP-Protocol. In the following a detailed explanation on the implementation of our synchronization strategy is given.

In terms of electrical engineering, the corresponding representations for offset and drift rate are frequency and phase. The synchronization task can be interpreted as phase-locked coupling of an oscillator (in our case this is the periodic send and receive thread) to a master-oscillator (in our case this is the global time). The standard solution to this problem is the well known *phase-locked-loop* (PLL). Such a system is based on a controllable oscillator with variable frequency, which is controlled by a feedback controller, i.e. with PID-characteristic. The basic structure is shown in figure 2. The frequency of the variable oscillator is defined as

$$f_{rt} = f_0 + k\Delta f_{rt} \tag{4}$$

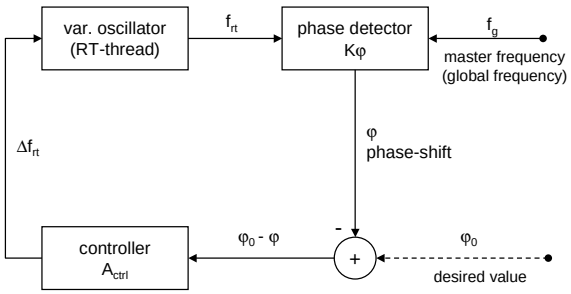

Figure 2. Structure of a PLL system

where the period of the real-time thread for sending and receiving is given by $T_{rt} = \frac{1}{f_{rt}}$, while $T_0 = \frac{1}{f_0}$ is the base-period at the start of network operation. So the controller does a differential variation of the thread-frequency Δf_{rt}. The phase detector does a delay measurement between the start of the receive-timeslot and the arrival of the data-packet. This can be interpreted as the phase-shift between the local (thread-)frequency f_{rt} and the global (communication-) frequency f_g on the network. The phase-shift φ is compared with the desired value φ_0 and serves as $\Delta\varphi = \varphi_0 - \varphi$ the input value for the controller with the transfer function A_{ctrl}. The output value of the controller represents the differential variation of the threads period. Thus, a closed loop is formed.

For realization of the transfer function, a digital recursive PID-controller is used. The algorithm is derived from the time-domain function[4].

$$u_{PID}(t) = K_R\left[e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau + T_D\frac{d}{dt}e(t)\right] \tag{5}$$

The approximation of the integral and the derivative yields the discrete function as

$$\begin{aligned} u_{PID}(k) &= K_R e(k) + K_R\frac{T}{T_I}\sum_{\nu=0}^{k-1} e(\tau) \\ &+ K_R\frac{T_D}{T}\left[e(\tau) - e(\tau-1)\right] \end{aligned} \tag{6}$$

A recursive version of equation 6 can be found by computation of $u(k) - u(k-1)$:

$$u_{PID}(k) = u_{PID}(k-1) + q_0 e(k)$$

[4]

- $e(t)$ input value (in our case this is $\Delta\varphi$)
- $u(t)$ output value (in our case this is Δf)
- K_R proportional constant
- T_I integration constant
- T_D derivative constant
- T sampling rate

$$+ \quad q_1 e(k-1) + q_2 e(k-2) \qquad (7)$$

with $q_0 = K_R(1 + \frac{T_D}{T})$, $q_1 = -K_R(1 + 2\frac{T_D}{T} - \frac{T}{T_1})$ and $q_2 = K_R\frac{T_D}{T}$. This algorithm represents a very simple way of software synchronization with high effiency and allows a good characterization of its behaviour in terms of controller technology.

We show that the phase-detector provides the only measured value for the control-loop and — because of that — is most important for the accuracy of the temporal behavior. In our implementation, the phase-detector is replaced by a time-delay-detector, that supplies the time-delay from the beginning of the receive-time-slot to the arrival of the data packet. This time delay is used as input of the controller, instead of the phase. For measuring the Time Stamp Counter of Pentium processor [15] is used, which is the most granular timer within a PC. The error of this counter caused by the quartz oscillators drift rate can be neglected for measuring these small time periods, but the main problems of every software based synchronization are the errors caused by reading the internal clock [7]. In our case the reading of the Time Stamp Counter lasts much longer than one tick of this counter, consequently that its usable precision is reduced. To meet in this problem and for fault-tolerance the measured time-delays are averaged over time. The maximum and minimum value within the consideration-window are not used for average [13]. This is similar to the FTA algorithm presented in [7].

The main task for the controller is to synchronize a new node to an existing network. Furthermore, the clocks drift rate has to be compensated. Every node has to know the propagation delay of a packet on the network. This time can be different for each node depending on the internal processing time and has to be measured before a new node participates in the network (see figure 3). For this reason, a new node first listens to the network traffic and its period of the send/receive-thread is adjusted by the controller to a desired time-delay t_0 of $20\mu s$. This means that the expected arrival of a data-packet is $20\mu s$ after the start of a timeslot, which is an averaged value for the propagation delay on the investigated network. Now, the new node sends a resync-packet as an answer to a sync-packet. The master detects the arrival of this packet and measures the propagation delay for the new node. This operation is shown in figure 3. With the next sync-packet, this delay is sent to the new node using this time as the new desired value. After the controller has adjusted this new desired value, the client takes part in the network.

Figure 4 illustrates the network with the new participating client. The propagation delay is $20 + x$, where x can have a positive or negative value. After this first synchronization, the controller continues work with the measured propagation delay until the network is stopped.

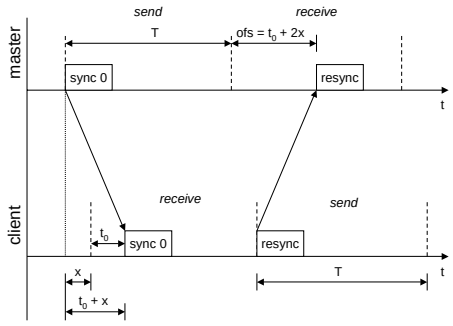

Figure 3. Measurement of the propagation delay by a sync/resync-operation

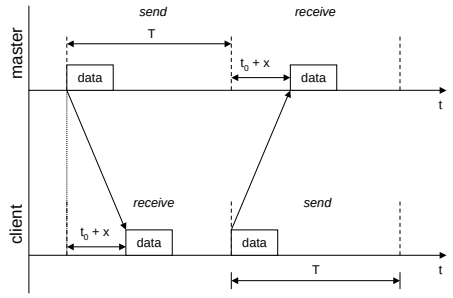

Figure 4. Network timing after synchronization

3.2. Media Access Protocol

The protocol provides time-triggered media access and dynamic network configuration in master/client-architecture. The whole time is shared equally between all network members. So the same bandwidth is assigned to each node. In each cycle, a pair of sync and resync packets is used for the dynamic network configuration. New members are anounced by this mechanism or nodes will be ignored, that are not available any longer.

If there are no bytes to be sent within a time slot, a dummy packet is sent. This is necessary for fault-tolerance and for accurate temporal synchronization. The implementation of the different phases of communication is done by a state machine. In figure 5, the subdivision of the time slots is given in example for three nodes.

The protocol provides a dynamic network configuration in contrast to other protocols like TTP/C (see 2.1), which are based on a static schedule. It offers a closed concept for handling non real-time (Linux TCP/IP) packets. It provides a guaranteed latency and jitter, but has to be seen only as a base protocol for future work. A lot of work has to be done

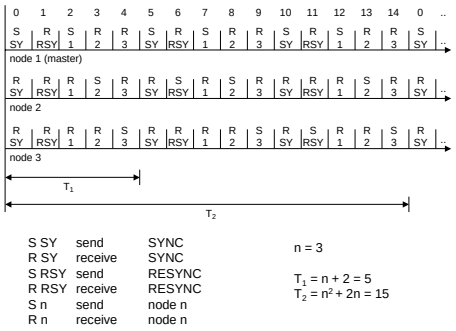

Figure 5. Subdivision of time slots in example for three nodes

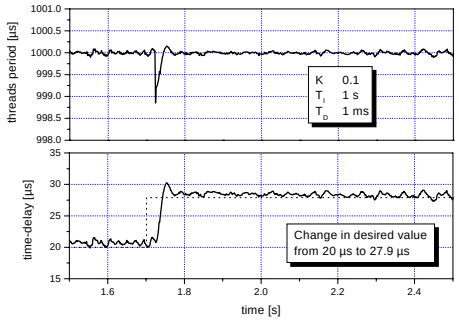

Figure 6. Reference reaction of the control loop

for fault-tolerance, i.e. a strategy for a master-fault has to be implemented. The protocol will be elaborated concerning its future use in middlewares.

3.3. Performance Evaluation of the Ethernet-based Real-Time Network

For performance evaluation, latency and jitter are the basic criterions to focus on. Two different times have to be distinguished:

1. The time between the call of the network-driver by the application on the one side and the arrival of the data packet on the other side. We denote this time *protocol latency*.

2. The time the drivers need to transmit a message to another node. We denote this time *network latency*.

Protocol latency and jitter are mainly determined by the base-period T_0 of the send/receive-thread and the used protocol[5]. In our implementation $T_0 = 1ms$ was used. The protocol latency depends on the cycle time and is calculated by

$$t_L = T_0(n+2) + T_0 \tag{8}$$

where n is the number of the network members (see also figure 5, "T_1"). In our implementation, a waiting application is signaled to put the data to the driver one time-slot before the next send-operation of the driver. This is taken in consideration by adding one base-period T_0 to t_L (see (8)). The receive-operation is synchronized and causes no additional latency. The jitter of the protocol latency is mainly determined by the way the application is calling, i.e. low priority of the calling thread causes a higher jitter. A detailed analysis of this jitter is presented in section 4.2.

[5]Concerning to the protocol latency the propagation delay can be neglected, because it amounts only to microseconds.

The network latency determines the performance of the network without taking the protocol in consideration. It represents the minimum in latency and jitter that can be attained with this system. The network latency is equal to the propagation delay of a data packet on the network. It is measured at start time of each node (see figure 3). In the investigated set of three PCs (PII, 400Mhz, 128MB), this time has been evaluated to be about $27\mu s$ by our time-delay detector as described above. The jitter is determined by the controlling behavior of the used algorithm for synchronizing. It has been evaluated to be about $3\mu s$. In the following an overview to the temporal behavior of the controller is given.

Two different reactions of control loops are of interest:

1. *Reference reaction.* This describes the controllers' behavior concerning a change of the desired value.

2. *Disturbance reaction.* This describes the controllers' behavior to disturbing external impacts.

The reference reaction is important for a fast synchronization after a new node has got his individual propagation delay by the master at start of its network participating. In opposite to that a good disturbing reaction is important for accurate compensation of the clocks' drift rate during the network operation. Each adjustment of the controllers' parameters is a compromise between these two reactions. We found the best adjustment for our controller at $K = 0.1$, $T_I = 1s$ and $T_D = 1ms$.

The controller should react within a small period to a change in the desired value. In our case, changes are completed in less than $0.5s$. Figure 6 shows the reaction of the controller after it has received the new desired value of $27.9\mu s$ from the master, which represents the propagation delay for this example.

The jitter of the network latency is mainly determined by the disturbance reaction of the controller. The jitter of the

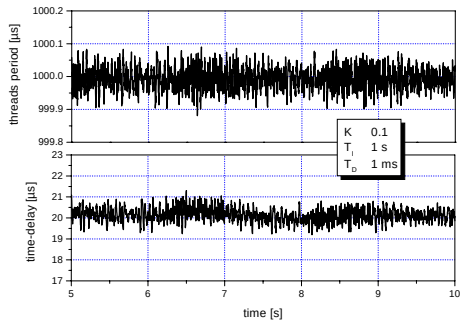

Figure 7. Disturbance reaction of the control loop

	Linux	*RTLinux*
with RTTI and exception handling	428 KByte	—
without RTTI and exception handling	170 KByte	143 KByte

Table 1. Code size of various ROFES implementations

investigated set of PCs had a value of $3\mu s$. Figure 7 shows the disturbance reaction of the controller for $5s$. In an experiment with a desired value of $20\mu s$ the measured time-delay and the adjusted period of the send/receive-thread have been recorded over a period of $5s$. By this, the average of time-delay has been measured to be $20.0717\mu s$ with a standard deviation of $0.3808\mu s$. In another experiment the drift-rate of the PC-oscillator has been measured to be $\rho = 7.31\mu s$. Because of that the controller had to adjust the send/receive-threads period to $T = 999.99269\mu s$. The average during the recorded period has been measured to $999.99270\mu s$, which means a very small standard deviation of $0.0360\mu s$. So the controller provides a very precise synchronization to the global time.

4. Design of ROFES based on a time-triggered ethernet protocol

4.1. Requirements of RTLinux

Our time-triggered ethernet protocol is based on RTLinux, which runs in the kernel space of Linux as well as any ROFES application. Only a small subset of POSIX is implemented in Real-Time Linux kernel. For instance the system calls *brk*, *malloc* and *free* are not implemented in RTLinux. But these system calls manage the memory allocation and deallocation of a process. Instead of these system calls, RTLinux uses the Linux kernel functions *kmalloc* and *kfree*. ROFES is implemented in C++ and the standard C++ operators *new* and *delete* base on the system calls *malloc* and *free*. Therefore, these C++ operators had to be overloaded and replaced by operators which are only based on the kernel functions *kmalloc* and *kfree*.

The most C++-Compilers support the exception handling and the run-time type information (RTTI) of the C++ standard. Unfortunately these features are also based on system calls, which are not implemented in RTLinux. Therefore ROFES cannot use these features.

On standard operating system, ROFES supports the CORBA exception handling, which is based on the standard exception handling of C++. Unfortunately, on RTLinux this is not suitable and ROFES had to renounce the support of the CORBA exception handling. In near future, we want to implement a exception handling for ROFES, which is not based on C++ exception handling. Consequently this new exception handling will be implemented even on RTLinux.

But the renouncement of exception handling also exhibits some advantages. The code size of a CORBA implementation, which supports exceptions handling, is much larger than the code size of a CORBA implementation, which does not support exception handling. This was evaluated on Linux. The code size of ROFES, which supports exception handling and the run-time type information, is about 200 KByte larger than the same implementation, with these features disabled. Table 1 presents the exact code size of various ROFES implementations.

4.2. Requirements of the time-triggered ethernet protocol

Section 3.3 presented an analysis of the protocol latency. The average latency and delay jitter of Real-Time CORBA requests depend on the time of sending a message. In best case, the sender can directly send its message and does not have to wait for its time-slot, while in worst case the sender has to wait a whole period to transmit its message. Consequently, the delay jitter of sending a message is equal to the period of the protocol and is calculated by

$$T_{jitter} = T_0(n+2) \tag{9}$$

where n is the number of the network members and T_0 the base-period of the send/receive-thread. This dependency is not ideal for real-time applications.

But most distributed real-time applications base on *spontaneous methods*. These methods send their requests to a server when the real-time clock reaches specific values determined at design time. On the assumption that the period

of the spontaneous methods is a multiple of the network period, the average latency and delay jitter can be optimized. The driver of the time-triggered network provides a function, which synchronizes the period of spontaneous methods with the period of the time-triggered network. This synchronization guarantees that the system spawns a spontaneous method when it reaches the exclusive time-slot to transmit a message. Thus the application must not wait for its time-slot and can directly send its message.

5. Performance Evaluation of ROFES

The benchmark in this section evaluates the performance of ROFES. On the client side, a single high-priority thread and a variable number of low-priority threads run concurrently. The high-priority threads spawn spontaneous methods and the low-priority threads invoke permanent requests. The requests of the low- and high-priority threads have no arguments. On the server-side, a servant is created and configured to process the client requests at the same priority at which the operation were originally invoked on the client side. This means that the benchmark uses the client propagated priority model. The low-priority threads use the CORBA priorities 8192, 8446, 8704, 8960, 9216 and 9472, while the high-priority thread uses the CORBA priority 16384. In order to receive the optimal results, the benchmark uses a priority banded connection. A priority banded connection is a new facility of Real-Time CORBA and allows a client to communicate with the server via multiple transport connections — each dedicated for carrying invocations with different CORBA priority or range of priorities. A client establishes a priority banded connection by sending the *_bind_priority_band* request to the server which specifies the range of priorities the connection will be used for. This allows the server to allocate the necessary resources. The selection of the appropriate connection for each invocation is transparent to the application and is done by the ORB based on the value of the priority model (server declared or client propagated).

The test platform consists of two 400 MHz Pentium II systems with 128 MBytes RAM. As operating systems *Linux*, *RTLinux* and *LynxOS 3.0.1* are used for these computers. The systems are connected with 100 Mbps Ethernet devices and SCI adapter cards. The *Scalable Coherent Interface* (SCI) [3] is a high performance network and offers transparent read and write access to remote memory. Memory segments of each compute node can be mapped into the virtual address space of all cluster compute nodes and be used to assemble globally shared data structures. Accesses to segments of memory that are physically located on remote compute nodes are transparently mapped to the address range of the SCI adapter and served via a respective network transaction. Figure 8 illustrates the SCI-network

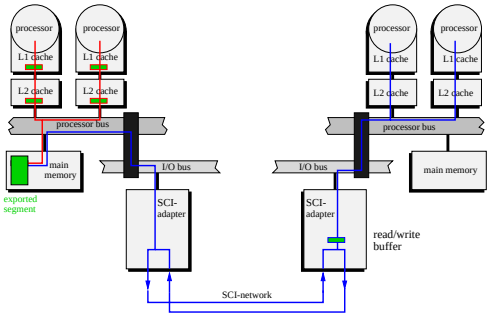

Figure 8. *The system architecture of a SCI-Cluster*

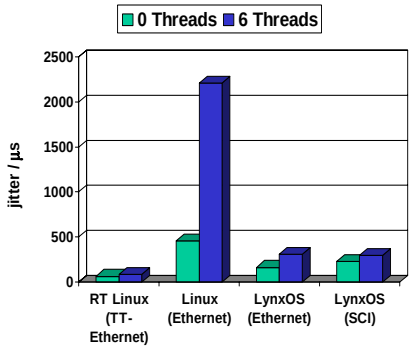

Figure 9. Jitter of the high-priority thread on various architectures

and [8] declares in detail the design and the performance of our SCI-based Real-Time CORBA.

The test platform is not an embedded system. However the memory footprints of the benchmarks are very low, the results should also be meaningful to the results on an embedded system.

Figure 9 shows the delay jitter of the high-priority thread on various architectures. The solution using Linux as operating system and TCP/IP as network protocol shows that Linux is not suitable for distributed real-time applications. By using six low-priority threads, the delay jitter of the high-priority thread increases dramatically. On a real-time operating system like LynxOS the delay jitter is clearly better.

The difference between the delay jitters of the SCI- and Ethernet-based solution on LynxOS is minimal. SCI has a higher bandwidth and a smaller latency than Ethernet. Figure 9 shows that these advantages has no positive effect for CORBA requests with a small number of arguments. But [8] shows that for CORBA requests with larger arguments, the SCI-based solution has a smaller delay jitter than the Ethernet-based solution.

On RTLinux, the benchmark uses the time-triggered ethernet protocol. To get the optimal performance of this protocol, the high-priority thread synchronizes its period with

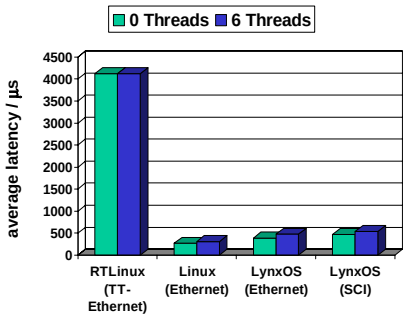

Figure 10. Average latency of the high-priority thread on various architectures

the period of the network. This optimization strategy is discussed in section 4.2. This solution has the smallest delay jitter and it ranges between $60\mu s$ and $80\mu s$.

Figure 10 shows the average latency of the high-priority thread on various archtitectures. Expectedly, RTLinux has the largest average latency but is independent to the number of threads. The test platform consists only of two nodes. From figure 5 follows that the network period of the time-triggered protocol is $4ms$ and this matches to the average latency of the protocol. The high-priority thread runs on the client node and sends a request at the begin of the time-slot before the node gets its sending time-slot. Therefore, one millisecond passes before the message is transmitted. The server on the master node receives the message and builds a reply to the client. But before the server can send its reply, it must wait three milliseconds for its next sending time-slot. The summation of these waiting times explains the average latency of $4ms$. A more exact synchronization of the spontaneous methods with the network period would reduce the average latency.

6. Conclusions and Future Work

This paper describes a new ethernet-based time-triggered protocol. The protocol represents an economical alternative to existing solutions for certain time-triggered real-time applications. The second part of this paper presents one of the first Real-Time CORBA implementations for RTLinux. This implementation called ROFES uses the new ethernet-based time-triggered protocol and shows its potential.

The paper shows that the average latency and delay jitter of the time-triggered network would be optimized, when the period of spontaneous methods is synchronized with the period of the network. This synchronization guarantees that the system spawns a spontaneous method when it gets the exclusive time-slot to transmit a message. Thus the application must not wait for its time-slot and can directly send its message. To get an intuitive programming scheme for this synchronization, we want to implement a CORBA service, which automatically synchronizes the period of spontaneous methods with the period of the underlaying time-triggered network. Kim presents in [4] a important fundament for such a programming scheme.

References

[1] M. Barabarnov. A Linux-based Real-Time Operating Systems. Master's thesis, New Mexico Institution of Mining and Technology, 1997.

[2] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *Proceedings of OOPSLA '97*. ACM, 1997.

[3] IEEE. *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI)*, 1992.

[4] K. H. Kim and E. H. Shokri. Two CORBA Services Enabling TMO Network Programming. In *IEEE CS 4th Int'l Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 1999)*, Santa Barbara, USA, January 1999.

[5] H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[6] H. Kopetz. A Comparison of CAN and TTP. In *Proc. of the IFAC Distributed Computer Systems Workshop*, Como, Italy, September 1998.

[7] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Trans. on Computer*, Vol. 36(No. 8):933–940, August 1987.

[8] S. Lankes, M. Pfeiffer, and T. Bemmerl. Design and Implementation of a SCI-based Real-Time CORBA. In *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May 2001.

[9] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20:pp. 46–61, 1997.

[10] OMG Technical Document formal/98-07-01. *The Common Object Request Broker – Architecture and Specification*, 2.2 edition, 1998.

[11] OMG Technical Document orbos/98-10-05. *Realtime CORBA – Joint Submission*, 1998.

[12] P. Ramanathan, K. Shin, and R. Butler. Fault-Tolerant Clock Synchrnization in Distributed Systems. *IEEE Computer*, Vol. 23(No. 10):33–42, October 1990.

[13] M. Reke. Entwicklung und Implementierung eines Netzwerkkarten-Treibers sowohl unter Linux als auch RTLinux zur Realisierung eines echtzeitfähigen Netzwerkprotokolls. Master's thesis, Lehrstuhl für Betriebssysteme, RWTH Aachen, February 2001.

[14] D. C. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *IEEE Computer*, 33(6):56–63, 2000.

[15] T. Shanley. *Pentium Pro and Pentium II Architecture*. Addison-Wesley, 1998.

[16] TTTech Computertechnik. *Specification of the TTP/C Protocol*, 1999.

[17] U. Wenkebach. CAN in der Medizintechnik. *Elektronik*, 16Q97 Issue, 1997.