# Real-Time Communication Support for Embedded Linux over Ethernet

**Sang-Hun Lee and Hyun-Wook Jin**
Department of Computer Science and Engineering, Konkuk University, Seoul 143-701, Korea
{mir1004, jinh}@konkuk.ac.kr

**Abstract -** *Many embedded systems are exploiting embedded Linux due to its rich features and device drivers. However, since embedded Linux is inherited from traditional monolithic kernel design, it has limitations to support real-time communication. One such example is that the protocol stacks of embedded Linux are following the bottom half based architecture. The user-level operations of a higher priority process can get delayed while the bottom half handles the packets for even lower priority processes. In addition, the send/recv semantics of traditional protocol stacks are not sufficient to support various real-time embedded applications. In this paper, we aim to design and implement a lightweight protocol stack that can support real-time communication over Ethernet without Linux kernel modification. The proposed protocol, called RTDiP, provides priority awareness, communication semantic for synchronization, and low communication overhead. The experimental results show that RTDiP can improve the communication latency up to 79% compared with TCP/IP. We also show that RTDiP can guarantee the execution time of real-time applications with less disturbance by communication of lower priority process and can provide predictable communication latency to real-time applications.*

**Keywords:** Embedded Linux, Real-Time Communication, Ethernet, Process Priority, Synchronization

## 1 Introduction

Many emerging embedded systems are connected through network and collaborate together. In factory automation systems, for example, the manufacturing machines, monitoring nodes, and control machines are connected via local network. Today many manufacturers depend on serial-based industrial networks. However, the better alternative for such network can be Ethernet [1][2] due to its high performance and widely accepted open standard.

Embedded Linux is employed to many embedded systems since it can provide rich features and device drivers. Embedded Linux also provides real-time supports such as POSIX.1b [3], which includes real-time process scheduling policies, real-time signals, etc. However, since embedded Linux is inherited from traditional monolithic kernel design, it has limitations to support real-time communication. For instance, the network protocol stacks (i.e., TCP/IP suite) implemented in the kernel process the packets in a first-in-first-service manner without the knowledge of priority of corresponding process. Consequently, if massive packets for low priority processes are received, the packets for a high priority process get delayed significantly. Similarly, user-level operations of a higher priority process can also miss the deadline because of the packets for low priority process. This is mainly because the protocol stacks of embedded Linux are implemented as the bottom half of traditional monolithic kernel.

In addition, the semantics of data transmission provided by TCP/IP is not sufficient to support various real-time embedded applications. In TCP/IP, the data transmitted to the receiver are passed to the application in sequence without intentional packet drop. Many helper applications of real-time systems, such as synchronization and shared state managers, keep interesting information updated through communication between nodes. To attain the packet containing the latest information, the applications have to first drain all the packets that hold outdated information though the information in the packets is not valid anymore. This is because the traditional communication semantic is to deliver all received packets to the application in sequence, which makes guaranteeing real-time requirements even harder in many critical real-time application domains.

In this paper, we aim to design and implement a lightweight protocol stack that can support real-time communication over Ethernet. We call the protocol stack as RTDiP (Real-Time Direct Protocol). RTDiP can process the packets belong to higher priority process first. Moreover, RTDiP provides a new communication semantic for synchronization. The performance measurement shows that RTDiP can improve the communication latency up to 79% compared with TCP/IP. We also show that RTDiP can guarantee the execution time of real-time applications with less disturbance by communication of lower priority process

This research is supported in part by grant #CR070019 from Seoul R&BD Program and grant from Institute for Information Technology Advancement.

and can provide predictable communication latency to real-time applications.

The rest of the paper is organized as follows: Section 2 describes existing design of protocol stacks implemented in the embedded Linux kernel and its limitation in the real-time computing point of view. In Section 3, we suggest RTDiP, which can overcome the limitations of existing protocol stacks. Section 4 presents the performance measurement results. We discuss related work in Section 5 and finally conclude the paper in Section 6.

## 2 Background

In this section, we describe the design of existing protocol stacks implemented in embedded Linux focusing on the receiver side.

When an application calls a receive system call, if there is any packet that already has been received in the kernel buffer, the data in the packet is copied to the user buffer. Otherwise, the application is forced to block.

When a packet arrives from the network to the Ethernet controller on the embedded board, the controller generates an interrupt. Then the interrupt handler implemented in the device driver moves the packet to the kernel buffer and passes it to the bottom half. The bottom half implements deferrable functions, which correspond to the IP and TCP layers. From Linux kernel version 2.4, the bottom half for networking is implemented as softirq but, in this paper, we do not distinguish bottom half and softirq and use *bottom half* because it is more general term. The bottom half is scheduled by the interrupt handler to be executed when there is no interrupts to be processed. It is to be noted that if the bottom half has work to do, no applications can run until the bottom half finishes its task. That is, the priority of the bottom half is lower than that of the interrupt handler but higher than any applications. Thus, the bottom half can occupy the CPU resource to handle the packets for low priority processes even if a higher priority process is ready to run for urgent deadline.

The packets arrived in the kernel buffers are queued until the corresponding application calls the receive system call. The system call delivers the packets queued in the kernel in FIFO manner to the application. A helper application of a real-time system, for example, can send packets with latest state information periodically to client nodes. The packets are queued on the client node unless the receiver read the packet as soon as it arrives. Therefore, several packets can be accumulated on the client and the most updated information can be visible to the receiver only after all the packets are dequeued. This results in nondeterministic latency and makes guaranteeing the real-time requirements very hard. Though TCP provides the urgent data flag to let the application know the urgent data has been arrived, TCP still gives the packets to the application in sequence.

## 3 RTDiP : Real-Time Direct Protocol

As described in the previous section, existing design of protocol stacks implemented in embedded Linux has limitations to support real-time communication. In this section, we suggest a new design of protocol stack called RTDiP over Ethernet, which can provide priority awareness, communication semantic for synchronization, and low communication overhead. We carefully design RTDiP so that we can simply implement it by modifying the device driver but without any kernel modification. Moreover, RTDiP can harmonize with all process scheduling policies implemented in embedded Linux.

### 3.1 Process Priority Aware Protocol Stack

The traditional protocol stacks on the receiver side is implemented based on the bottom half architecture. As we have discussed, since the bottom half has higher priority than any of applications, a high priority process can get delayed while the bottom half is busy with the packets for a lower priority process.

To tackle this issue, we get rid of the bottom half from the RTDiP data path and efficiently distribute the work, which has been done by the bottom half, into user library, system call, and the interrupt handler. When the packet is received, the interrupt handler of the Ethernet controller quickly decides which connection is corresponding to the packet and queues it to the backlog queue. We also carefully modify the interrupt handler so that the embedded Linux can still support TCP/IP packets without interference with RTDiP.

The user library and the system call are responsible for the data movement between the kernel and user buffers. Since the backlog queue is shared between the system call and the interrupt handler, we need synchronization to access the data structure. To do this, we disable the interrupt during the system call deletes a packet from the backlog queue. It is to be noted that the deletion overhead is very small and constant. On the other hand, the interrupt handler does not raise a lock to access the queue because it has higher priority and assumes single processor system.

That is, most of actual RTDiP packet processing is done by the user library and system call. Since the process scheduler assigns the CPU resource to the processes based on their priority, the higher priority process can have more chance to call the receive system call than lower priority processes. Therefore, without any modification of the process scheduler, RTDiP can process the packets for the higher priority process preferentially and prevent the delay of higher priority process.

### 3.2 Communication Semantic for Synchronization

The existing application interfaces such as Sockets provide the send/recv communication semantics that deliver the packets in sequence to the application. Thus the protocol stacks keep the packets to the queue and give them to the application when it is available. However, as we have discussed, synchronization managers can be interested only in the last packet that contains the latest information. In such case, the application has to pull out all the packets from queue to obtain the last packet but the previous packets are not useful to the application.

In order to efficiently support the synchronization applications of real-time, we suggest a new communication semantic that deliver the last packet ignoring the previous packets. In the synchronization communication semantic, RTDiP does not queue the packets but keep only the last packet received. There are two critical requirements when we design and implement the communication semantic for synchronization: i) time stamping and ii) predictable latency.

Many synchronization applications require the information of when the packet has been arrived on that node because the application does not know how long the packet has been stayed in the kernel after receiving. Though the protocol stacks in embedded Linux also keep such information, it is not exposed to the applications. On the other hand, RTDiP keeps the timestamp information generated by the interrupt handler and gives it to the applications so that they can perform accurate synchronization operations.

In addition, each connection holds the latest packet of synchronization semantic dropping the previous packets for synchronization. Thus, it can save more memory than send/recv semantic. More importantly, RTDiP can guarantee that the latency of receiving operation is predictable due to the fact that the first packet passed by the receive system call is always the latest packet received.

## 3.3 Low Communication Overhead

Many of embedded nodes are connected in LAN environment [4][5] though they are managed through IP network via a gateway. The protocol stacks in embedded Linux, however, targets general purpose WAN environment, which can result in heavy communication overhead and low performance in LAN environment.

To achieve better performance in LAN environment, we suggest a thin transport layer over Ethernet. We define the packet format of RTDiP as shown in Figure 1. As we can see in the figure, the RTDiP header includes destination port, length of user data, packet priority, and flag for communication semantics. The interrupt handler does demultiplexing using the destination port and queues the packet into the backlog queue of which overhead is not significant. The RTDiP user library and system call move the data between the user and kernel buffers. Figure 2 shows the data path of RTDiP and compares it with that of existing protocol stacks (e.g., TCP/IP). As we can see, since RTDiP skips TCP/IP layers and directly accesses the Ethernet layer it can provide lower communication overhead. In addition, on the sender side, we reduce the number of data copy into one. Existing protocol stacks first copy the user data into the kernel buffer. Then, the data in the kernel buffer is again moved to the Ethernet controller. In RTDiP, we directly move

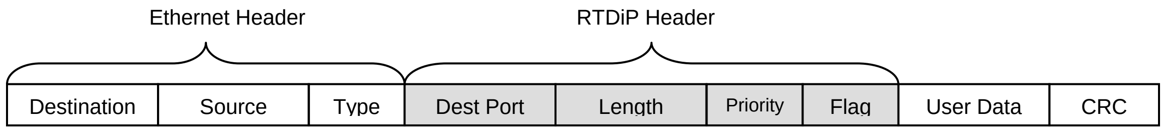

**Figure 1. RTDiP Packet Format**

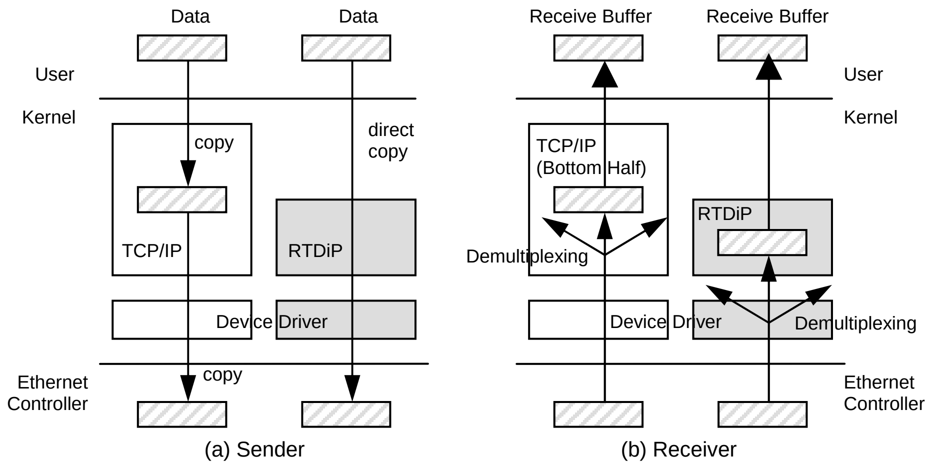

**Figure 2. Comparison of Data Path between RTDiP and TCP/IP**

the user data into the Ethernet controller.

The priority in the packet header defines the priority of packets in the connection, which is applicable to the packets of send/recv semantics. The system call returns the highest priority packet first. For the same priority packets, the system call returns the packets in FIFO manner. The flag field specifies whether the packet is for send/recv semantics or synchronization semantic. If a packet is specified as a synchronization packet, the interrupt handler manipulates it as described in Section 3.2.

## 4 Performance Measurement

We have implemented RTDiP in the smc91x device driver. For the experiments, we have used a pair of embedded boards [6] equipped with Intel XScale PXA270 (520MHz), 128MB SDRAM, 32MB Flash ROM, and LAN91C111 Ethernet controller. We have installed embedded Linux (kernel version 2.6.12) on the embedded boards and connected them directly without either switch or hub. In this section, we compare the performance of RTDiP with TCP/IP.

### 4.1 Process Priority Awareness

In this experiment, we run an application of high priority, which performs mathematical computation, while running a networking process with lower priority as background. The computing process wakes up for every 20ms and does the computation. The networking process receives 1KB packets continuously from a remote node with a flow control. We have chosen 1KB message size because RTDiP and TCP/IP show similar communication bandwidth with the message size as shown in Section 4.3. We use the FIFO real-time scheduler of embedded Linux for the computing process and the OTHER non-real-time scheduler for the networking process. The intention of the experiment is to see how much the real-time application (i.e., computing process) is affected by the communication of non-real-time application (i.e., networking process).

Figure 3 shows the snapshot of computing process's execution time for 1sec. It is to be noted that RTDiP-send/recv does not affect the execution time of real-time application much, which shows indeed almost constant value. This means that RTDiP can provide process priority awareness and performance predictability. On the other hand, in case of TCP/IP, the execution time of computing process is fluctuates significantly. The execution time increases up to 51% compared with the minimum execution time. Moreover, we can observe that the execution period of computing process is also varied because the execution time is not guaranteed. In a real system, such period variation may lead to a deadline miss.

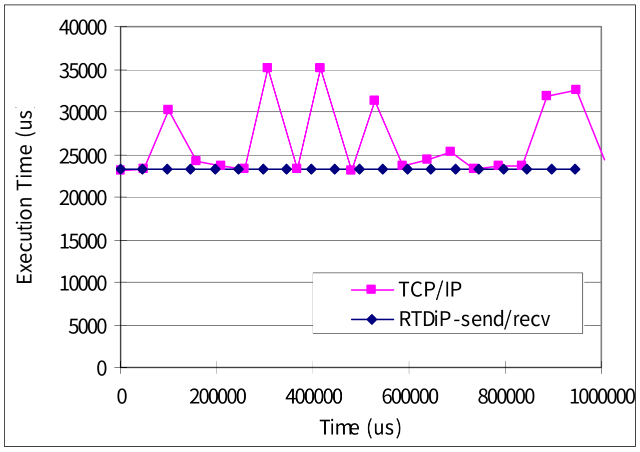

Figure 3. Real-Time Process Execution Time

### 4.2 Communication Performance for Synchronization

In this section, we consider the synchronization scenario in which a server process sends the latest state information to the remote process periodically. The receiver on the remote node wakes up periodically and updates local information to the data in the last packet received. We vary the periods of sending operation on the sender and updating operation on the receiver, respectively. We measure the time difference between when the receiver wakes up and when finishes updating the state information successfully.

Figures 4, 5, and 6 show the measurement results of information updating overhead. We have varied the both periods of sending and updating from 100ms to 1sec. the size of each information packet is 1KB. Figures 4 and 5 show that TCP/IP and RTDiP-send/recv take significant time to update the information especially when the sending period is larger than the updating period. This is because the packets of send/recv semantics are queued on the receiver side. Thus the application has to pull out the packets to get the last one. On the other hand, the information updating overhead of RTDiP-sync in Figure 6 is bounded below 90us regardless of the period values. This demonstrates that RTDiP-sync can provide predictable latency for synchronization managers.

For a large sending period together with a small updating period, it happens often that there is no packet received when the updating process wakes up. Thus the updating process quickly goes back to sleep without actual updating, which reports very low updating overhead. This is the reason why all three graphs show very low overhead for such period range.

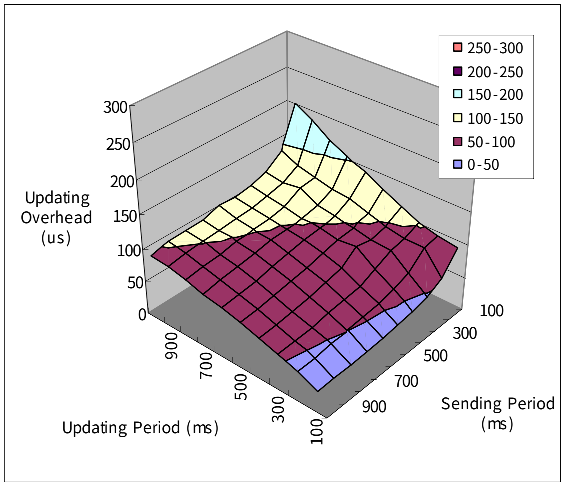

Figure 4. Information Updating Overhead of TCP/IP

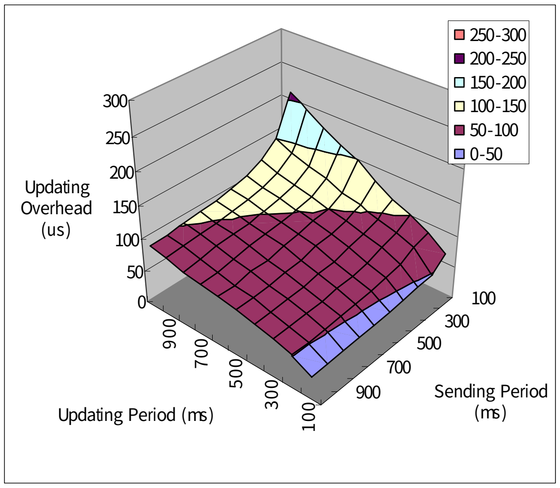

Figure 5. Information Updating Overhead of RTDiP-send/recv

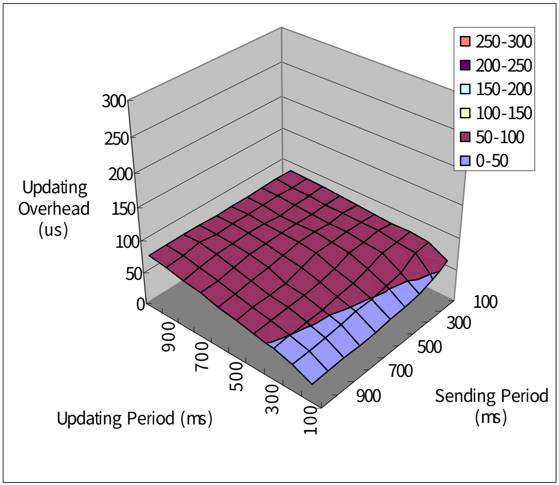

Figure 6. Information Updating Overhead of RTDiP-sync

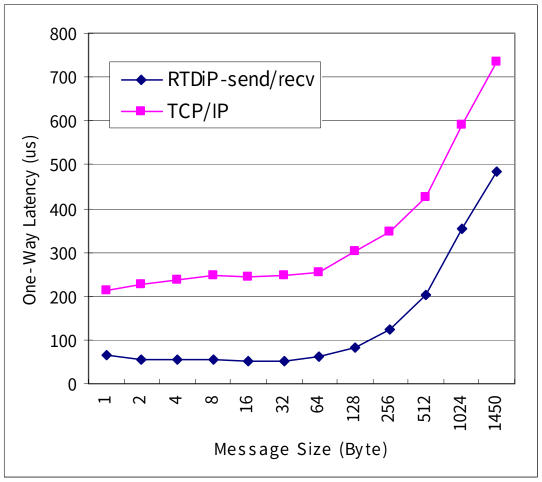

Figure 7. One-Way Communication Latency

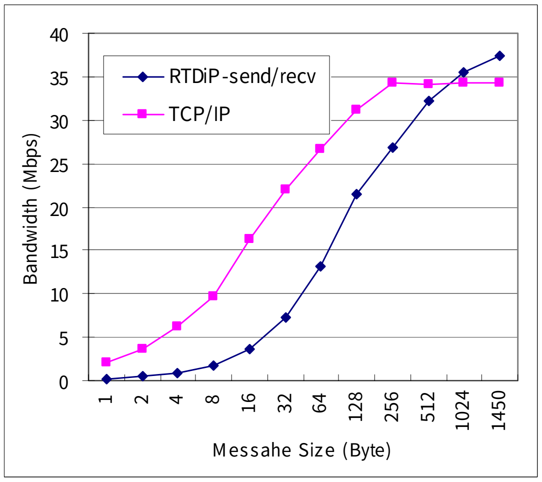

Figure 8. Communication Bandwidth

## 4.3 Communication Latency and Bandwidth

To measure the basic performance benefit of RTDiP, we have implemented the micro-benchmarks that measure one-way communication latency and bandwidth. The communication latency test is carried out in a standard ping-pong fashion. The bandwidth test measures the utilized bandwidth while transmitting packets with window-based flow control.

Figure 7 shows the one-way communication latency varying the user message size from 1B to 1450B. As we can observe, RTDiP-send/recv reduces the latency up to 79% compared with TCP/IP. This is because RTDiP directly accesses Ethernet skipping the heavy TCP/IP layers and does less data copy as described in Section 3.3. The dominant overhead of RTDiP is the interrupt handler overhead in which the data movement cost between the kernel buffer and Ethernet controller is included. In general, this overhead is evitable because an efficient way to move the data, such as DMA, is not supported in embedded systems.

The bandwidth results are shown in Figure 8. We can see that for small and middle size messages, TCP/IP achieves higher bandwidth than RTDiP-send/recv. This is because TCP is a stream protocol and merges small messages into large one when several packets are buffered on the sender side, which results in better network utilization. On the other hand, RTDiP is a datagram protocol and deals with different messages as separate packets. For large messages, however, as shown in figure, RTDiP-send/recv presents higher bandwidth than TCP/IP.

## 5 Related Work

There have been many researches to provide the real-time communication for user-level processes. The standalone user-level daemons that perform central management of communication requests from the processes has been proposed in [7][8][9]. The daemon obtains the priority of each process and schedule the communication based on this. This scheme, however, should keep track of changing the process priority. Since the real-time schedulers can change the process priority at any time, the daemon has to be notified whenever the priority is changed, which is not easy. In addition, the data movement between user processes and daemon is expensive. Compared with this scheme, RTDiP can adapt to dynamic changing of priority flexibly and provide low communication overhead.

Intelligent network controllers can perform demultiplexing instead of a daemon process for user-level protocols as suggested in [10][11][12]. In this case, we can implement the full TCP/IP stacks as user library. However, it is not general to employ such intelligent network controller in the embedded systems because of board space, price, etc.

An upcall-based real-time communication support has been proposed in [13]. In this mechanism, each process registers the upcall handler that handles the packet arrived. Though the upcall has its own priority, it is not tied with the process priority and cannot be changed dynamically when the process priority is altered by scheduler.

The user-level middleware scheduling proposed in [14] schedules the processes based on their real-time requirements. However, since the kernel-level scheduler is not aware about the middleware-level scheduler and the real-time requirements of processes, it is hard to guarantee the real-time requirements in accurate.

There are also efforts to adopt Ethernet as interconnection of manufacturing systems providing real-time communication [2][15][16] but these are focusing on the media access protocol for real-time communication. On the other hand, our work is to provide higher protocol layers that can handle the packet based on the priority of corresponding process.

## 6 Conclustions and Future Work

In this paper, we have designed and implemented a lightweight protocol stack that can support real-time communication over Ethernet without Linux kernel modification. The proposed protocol called, RTDiP, provides priority awareness, communication semantic for synchronization, and low communication overhead. The experimental results have showed that RTDiP can improve the communication latency 79% compared with TCP/IP. We also have showed that RTDiP can guarantee the execution time of real-time applications with less disturbance by communication of lower priority process and can provide predictable communication latency to real-time applications.

As future work, we intend to implement one-copy data receive for the communication semantic of synchronization. The current implementation of synchronization semantic incurs still two data copies on the receiver side like send/recv. We try to reduce the number of data copy using the memory mapping mechanism. In addition, we plan to implement a thin IP layer so that RTDiP can be utilized even in WAN environment.

## References

[1] IEEE Std. 802.3: LAN/MAN CSMA/CD Access Method, December 2005.

[2] EtherCAT Technology Group, "EtherCAT - Ethernet for Control Automation Technology," http://www.ethercat.org/.

[3] IEEE Std. 1003.1b-1993, IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX®) - Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension [C language], 1993.

[4] F. Kanehiro, Y. Ishiwata, H. Saito, K. Akachi, G. Miyamori, T. Isozumi, K. Kaneko, and H. Hirukawa, "Distributed Control System of Humanoid Robots based on Real-time Ethernet," In Proc. of 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2471-2477, October 2006.

[5] K. H. Kim, E. Henrich, C. Im, M. -C. Kim, S. -J. Kim, Y. Li, S. Liu, S. -M. Yoo, L. -C. Zheng, and Q. Zhou, "Distributed Computing Based Streaming and Play of Music Ensemble Realized Through TMO Programming," In Proc. of IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems (WORDS2005), pp. 129-138, February 2005.

[6] Huins, Inc., http://www.huins.com/.

[7] T. Nakajima and H. Tokuda, "User-level Real-Time Network System on Microkernel-basedOperating Systems," Real-Time Systems, 14(1), pp. 45-60, 1998.

[8] K. M. Zuberi and K. G. Shin, "An Efficient End-Host Protocol Processing Architecture for Real-Time Audio and Video Traffic," In Proc. of Network and Operating System Support for Digital Audio and Video (NOSSDAV), July 1998.

[9] P. A. Dinda, "The Minet TCP/IP Stack," Technical Report NWU-CS-02-08, Department of Computer Science, Northwestern University, 2002.

[10] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing Network Protocols at User Level," IEEE/ACM Transactions on Networking, 1(5), pp.554-565, 1993.

[11] D. Riddoch, K. Mansley, and S. Pope, "Distributed Computing with the CLAN Network ," In Proc. of the 27th Annual IEEE Conference on Local Computer Networks (LCN02), 2002

[12] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. J. Regnier, "CSP: A Novel System Architecture for Scalable Internet and Communication Services," In Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems, pp. 61-72, March 2001.

[13] R. Gopalakrishnan and G. M. Parulkar, "A real-time upcall facility for protocol processing with QoS guarantees," In Proc. of ACM Symposium on Operating Systems Principles (SOSP), 1995.

[14] C. Shen, O. González, K. Ramamritham, and I. Mizunuma, "User Level Scheduling of Communicating Real-Time Tasks," In Proc. of IEEE Real Time Technology and Applications Symposium, 1999.

[15] A. Moldovansky, "Utilization of Modern Switching Technology in EtherNet/IP Networks," In Proc. of International Workshop on Real-Time LANs in the Internet Age (RTLIA'2002) in conjunction with the 14th Euromicro Conference on Real-Time Systems, June 2002.

[16] IEEE Std. 1588 -2002, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems."