Lightweight Real-time Network Communication Protocol for Commodity Cluster Systems*

Hai Jin, Minghu Zhang, Pengliu Tan, Hanhua Chen, Li Xu

Cluster and Grid Computing Lab.
Huazhong University of Science and Technology, Wuhan, 430074, China
Email: hjin@hust.edu.cn

Abstract. Lightweight real-time network communication is crucial for commodity cluster systems and embedded control systems. This paper introduces the design, implementation and evaluation of SS-RTUDP, a novel zero-copy data path based real-time communication protocol with efficient communication resources management. To avoid unpredictable overheads during SS-RTUDP packets transmission, all communication resources are pre-allocated. A feasible fragmentation mechanism is also proposed for transmitting SS-RTUDP packets larger than the network *MTU*. On the other hand, the additional real-time traffic smoother provides high priorities to SS-RTUDP packets and also smoothes peak packets arrival curve. The prototype of SS-RTUDP is implemented under Linux systems and performance evaluations over Fast/Gigabit Ethernet are provided. The measurement results prove that SS-RTUDP can provide not only much lower latency and higher communication bandwidth than traditional UDP protocol, but also good real-time network communication performance for commodity cluster systems.

1 Introduction

Network environment within cluster system has changed to Fast/Gigabit Ethernet with higher bandwidth and lower error-rate. Thus, the user expects mostly in cluster systems is the network end-to-end communication performance. The performance of the communication sub-system in a cluster system is determined by the transmission rate of the network hardware, the processing ability of I/O buses, and also the software overheads of network protocol. In recent years, while the network hardware and I/O buses provide gigabit bandwidth, the traditional software overheads of network communication protocol (such as TCP/IP stack) have become the bottleneck, especially in cluster system with large amount of message passing [1].

To improve the communication performance in cluster system, several approaches have been considered, such as using multiple parallel networks, implementing lightweight protocols, avoiding data buffering and copying, avoiding system call invoking, overlapping communication and computation, using fast interrupt paths or

* This paper is supported by National 863 Hi-Tech R&D Project under grant No.2002AA1Z2102.

avoiding interrupt processing, and introducing Jumbo Frames [2].

Two main approaches adopted to reduce the software overheads of a communication protocol are the improvement of the TCP/IP layers and the substitution of the TCP/IP layers by alternative ones [3]. The former focuses mainly on *zero-copy* architectures [4][5], which are capable of moving data between application domains and network interfaces without any CPU and memory bus intensive copy operations. Two alternatives can be considered to the latter approach: communication protocol with efficient OS support [6][7], and the user-level network communication [8]~[13]. But most of these facilities require special NIC hardware.

In this paper, we propose SS-RTUDP (***S**imple and **S**tatic-resources-allocation **R**eal-**T**ime **U**ser **D**atagram **P**rotocol*), a lightweight real-time network communication protocol over commodity Fast/Gigabit Ethernets. High data throughout, low latency and real-time communication are the major requirements for SS-RTUDP and the zero-copy data path and a fragmentation mechanism are adopted in SS-RTUDP. To satisfy real-time communication requirements, an adaptive real-time traffic smoother is employed and all communication resources are pre-allocated.

The organization of this paper is as follows. Section 2 describes the processing of original UDP protocol. Section 3 introduces the design and implementation of SS-RTUDP in detail. The performance measurements of SS-RTUDP prototype system are shown in Section 4. Section 5 states the conclusions and directions of future work.

2 Overhead Analysis of UDP Protocol

The overhead of UDP protocol consists of per-packet and per-byte costs [14]. The per-packet costs include the OS overheads and the protocol processing overheads. The per-byte costs include data-copying overheads and checksum calculation overheads. Fig. 1 shows the data and control flow of UDP/IP processing on Linux system. These costs are analyzed using the Intel PRO/1000 Gigabit Ethernet card with Intel Celeron 2.0GHz PC (RedHat9.0, Linux 2.4.20 kernel). Table 1 shows the total overhead during sending and receiving a UDP packet over Gigabit Ethernet. The most dominant overheads are for UDP/IP layer processing, including memory data copies, checksum calculations, resources allocating, etc.

Table 1. Total cost evaluation of UDP processing

Overheads list	Gigabit Ethernet	
	Costs(μs)	***(%)***
System call and socket layer	***3.2***	***4.7%***
UDP/IP layer(sender)	***14.8***	***21.7%***
UDP/IP layer(receiver)	***23.5***	***34.5%***
Device driver layer	***6.8***	***10.0%***
NIC interrupt processing	***6.9***	***10.1%***
DMA and media transmit	***5.5***	***8.1%***
Others	***7.4***	***10.9%***
Total	***68.1***	***100%***

3 SS-RTUDP Approach

The SS-RTUDP design issues can be summarized as follows:

- Use commodity hardware and without modifying hardware firmware.
- Minimal modification to the Linux kernel and good portability to other systems, such as real-time systems and embedded control systems.
- Coexisting with standard protocol such as TCP/IP stack.
- Lightweight communication guarantee (zero-copy data path).
- Provide soft real-time communication.

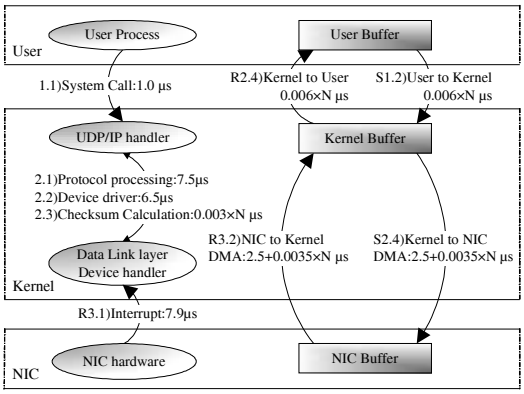

Fig. 1. Data and control flow and their costs using UDP protocol

3.1 Network Communication Architecture with SS-RTUDP

The network communication architecture with SS-RTUDP is shown in Fig. 2. Three main parts are involved in SS-RTUDP protocol: user-level application library, SS-RTUDP/IP layer and real-time traffic smoother.

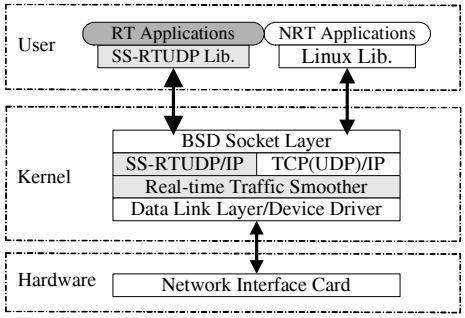

Fig. 2. Network communication architecture with SS-RTUDP protocol

3.1.1 Application Programming Interfaces (APIs)

In current SS-RTUDP communication library, five APIs similar to traditional

communication APIs are provided: *rt_socket(), rt_mmap(), rt_sendto(), rt_recvfrom() and rt_socket_close()*. Table 2 shows the main functions of them.

Table 2. User-level APIs provided by SS-RTUDP

API	Function description
rt_socket()	*1) Apply for a new real-time socket* *2) Allocate appropriate kernel buffer for the socket* *3) Initialize a socket buffer pool for the socket* *4) Return the physical address of the kernel buffer allocated*
rt_mmap()	*Remap the user send-buffer space to real-time socket's kernel buffer*
rt_sendto()	*Transmit a SS-RTUDP packet*
rt_recvfrom()	*Receive a SS-RTUDP packet*
rt_socket_close()	*Close a real-time socket*

3.1.2 Zero-copy Sending Data Path in SS-RTUDP

In SS-RTUDP protocol, the network *MTU* is 1500 bytes, the header length of IP protocol (H_{ip}) 20 bytes, the header length of SS-RTUDP protocol ($H_{SS\text{-}RTUDP}$) 8 bytes, and the Ethernet hardware header (H_{hh}) 16 bytes. A SS-RTUDP packet larger than *MTU* must be split into several IP fragments before transmission. The headers and the data of an IP fragment must be assembled within continuous physical address space due to *DMA (Direct Memory Access)* requirements. Fig. 3 describes the fragmentation mechanism, where L_{data} stands for data size of a SS-RTUDP packet to be sent while L_{left} the data size left to be sent, and pointer p points to head of the socket buffer.

1. Lock user send-buffer and set $L_{left}=L_{data}$.
2. Allocate a socket buffer from the socket's *rtskb_pool*.
3. Assemble the IP, SS-RTUDP and Ethernet headers from p.
4. If $L_{left} \leq P_{max}$, transmit the single packet from p with $H_{ip}+H_{SS\text{-}RTUDP}+H_{hh}+L_{left}$ size using *DMA* to the network and jump to step9; Else, transmit the first fragment with $H_{ip}+H_{SS\text{-}RTUDP}+H_{hh}+P_{max}$ size from p to network using *DMA* and set $L_{left}=L_{left}-P_{max}$.
5. Set $p=p+1480$, and allocate a socket buffer from the socket's *rtskb_pool*.
6. Assemble the IP and Ethernet headers from p.
7. If $L_{left} \leq 1480$, transmit the last fragment with $H_{ip}+H_{hh}+L_{left}$ size from p to the network using *DMA*; Otherwise, transmit the fragment with $H_{ip}+H_{SS\text{-}RTUDP}+H_{hh}+P_{max}$ size from p to the network using *DMA*.
8. Set $L_{left}=L_{left}-1480$ and jump to step 5.
9. Unlock user send-buffer.

Fig. 3. A fragmentation mechanism for SS-RTUDP protocol

Based on the fragmentation mechanism in Fig. 3, we design a zero-copy data sending path in SS-RTUDP shown in Fig. 4. The user send-buffer is remapped to the kernel buffer first using the *rt_mmap()* system call and all the IP fragments are assembled in continuous physical space without additional kernel buffer for the headers of each IP fragment.

3.2 Real-time Communication Considerations for SS-RTUDP Protocol

Two aspects are considered to provide real-time performance for SS-RTUDP. One is to avoid dynamic communication resources allocation. The other is to add real-time traffic smoother to provide higher priority to SS-RTUDP packets than other packets.

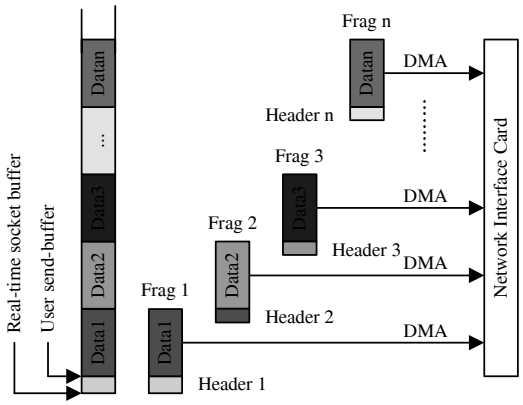

Fig. 4. Zero-copy data sending path in SS-RTUDP

3.2.1 Pre-allocation of Network Resources

To satisfy real-time communication performance of SS-RTUDP, the unpredictable overheads in the communication path must be avoided. The main unpredictable operation is dynamic kernel buffers allocation to buffer the data being copied from the user space. During the initialization of SS-RTUDP protocol, a global buffer pool is initialized, where four kinds of kernel buffer blocks (1, 4, 8 and 16 continuous physical memory pages) are pre-allocated and pinned down. The main functions of *rt_socket()* system call are to apply one appropriate kernel buffer block and initiate a socket buffer pool (*rtskb_pool*). Through remapping its send-buffer to the socket kernel buffer using the *rt_mmap()* system call, the user application can directly use the socket kernel buffer. The socket buffer needed during packet sending is not dynamically allocated but directly achieved from the *rtskb_pool*, and the buffer space of each socket buffer is set to appropriate position of the socket kernel buffer block.

3.2.2 Real-Time Traffic Smoother

The main functions of the real-time traffic smoother are to control the *non-real-time* (*NRT*) packet arrival rate to appropriate input limit, without affecting the *real-time* (*RT*) packet arrival rate. The real-time traffic smoother provides statistical real-time communication performance that the probability of packet lost is less than a certain loss tolerance, *Z* [15]:

$$Pr_{(packet\ loss\ rate)} \leq Z \qquad (1)$$

The traffic smoother is leaky bucket-based [16], where *credit bucket depth* (*CBD*), the capacity of the credit bucket, and a *refresh period* (*RP*) are defined. Every *RP*, up to *CBD* credits, are replenished to the bucket. In our implementation, the unit of credit is packet. We set the input limit of *packet arrival rate* (*PAR*) as the value of *CBD/RP*,

which determines the average throughput available:

$$PAR=CBD/RP \quad (2)$$

By fixing the values of *CBD* and varying the values of *RP* ($RP_{min} \le RP \le P_{max}$), it is possible to control the burst nature of packets flows generated. We set RP_0 ($RP_{min}<RP_0<RP_{max}$) as the initialization value (also the average value) of *RP*.

$$PAR_0=CBD/RP_0 \quad (3)$$

Fig. 5 shows the smoothing and refreshing procedures provided by the real-time traffic smoother. When a *NRT* packet arrives from the IP layer, the traffic smoother sends it to the NIC and removes one credit if there is at least one credit in the bucket. Otherwise, the *NRT* packet is buffered. A *RT* packet is not affected by the traffic smoothing, but it consumes a credit also.

	Smoothing		**Refreshing**
1	if(SS-RTUDP){	1	RP = RP_0;
2	send-to-NIC();	2	if (packet lost in the latest RP region)
3	CNS = CNS-1;}	3	RP = min(RP_{max}, 2×RP);
4	else {	4	else
5	if(CNS ≥ 0){	5	RP = max(RP_{min}, RP- Δ RP);
6	send-to-NIC();	6	if (CurrentTime == NextRefreshTime)
7	CNS = CNS -1;}	7	{
8	else	8	CNS = min(CBD, CNS+CBD);
9	send-back-to-queue();	9	NextRefreshTime = CurrentTime+RP;
10	}	10	}

RP_0 : The initialization value of RP
RP_{max} : the maximal value of RP
RP_{min} : the minimal value of RP
CNS : Current Number of Credits

Fig. 5. Smoothing and refreshing procedures

Fig. 6 shows the effect of traffic smoothing. Fig. 6 (a) shows the symmetric burst input arrival rate during $[0, 6RP_0]$ from the IP layer to the NIC and Fig. 6 (b) shows the smoothed output arrival rate from the NIC to the network.

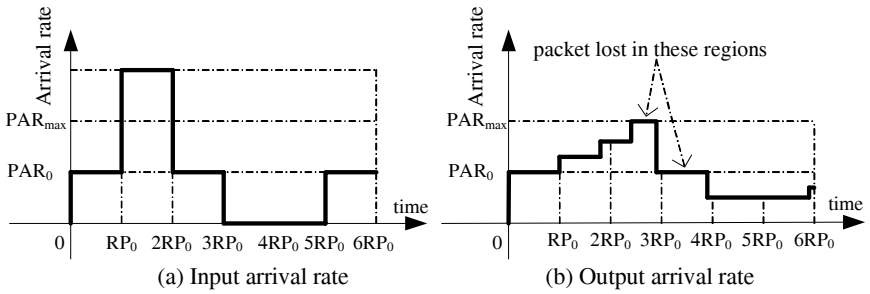

Fig. 6. Traffic smoothing results

4 Performance Analysis

All the evaluations of SS-RTUDP are made using a cluster with 24 nodes. Table 3 shows the configuration of the evaluation environment.

Table 3. Evaluation environment configuration

Hardware		***Intel Celeron 2.0GHz CPU; 256MB DDR Memory; 33MHz/32bit PCI bus***
NIC	***Fast Ethernet***	***3COM 3C905B 10/100M***
	Gigabit Ethernet	***Intel PRO/1000MT Gigabit Ethernet card***
		3COM 3c905B Fast Ethernet card
Ethernet switches		***D-link DES-1024$^+$Fast Ethernet switch***
Operating systems		***RedHat 9.0, Linux kernel 2.4.20***
Traffic smoother parameters		***CBD=40, RP_0=10ms, RP_{max}=100ms, ΔRP=2ms, RP_{min}=5ms***

The basic application level communication bandwidth and latency over Fast/Gigabit Ethernet and the real-time communication performance over Fast Ethernet, are measured and compared with traditional UDP/IP in this section. We use the modified NetPIPE [17] benchmark to evaluate network latency and bandwidth. The real-time performance is measured by packet loss rate.

4.1 Latency and Bandwidth

In this experiment, two nodes are connected directly by the cross-over cable. The latency and bandwidth results over Fast Ethernet are shown in Fig. 7.

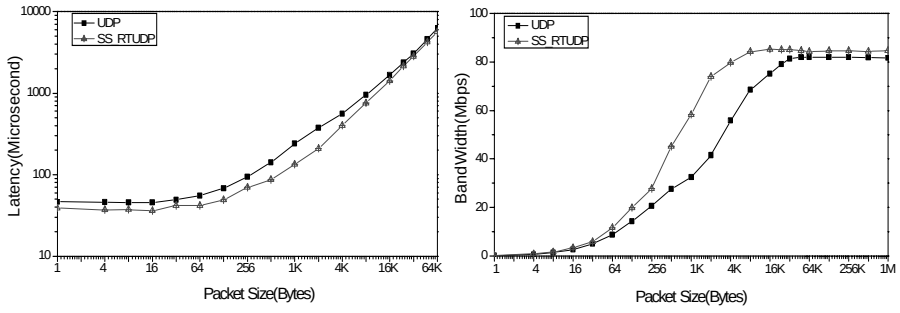

Fig. 7. Latency and bandwidth over Fast Ethernet

In Fig. 7, SS-RTUDP over Fast Ethernet can achieve 37 μs latency in a 4 bytes message, which is a little smaller than 45.8 μs of UDP. Especially, the latency decreases over 36% over SS-RTUDP ranging between 256 to 4096 bytes. Fig. 7 also shows that SS-RTUDP can achieve a maximal bandwidth of 89.6 Mbps which is a little higher than 81.9 Mbps of UDP. These results also show that SS-RTUDP can achieve half of the maximal bandwidth below 512 bytes packet size, while this value is over 2000 bytes in UDP. Obviously, the performance improved by SS-RTUDP protocol is limited over Fast Ethernet due to low media transmitting rate.

Fig. 8 shows the latency and available bandwidth achieved over Gigabit Ethernet. The results show that SS-RTUDP can exploit much more Gigabit Ethernet capacity than that of traditional UDP protocol. Over Gigabit Ethernet, SS-RTUDP can achieve 39 μs latency in a 4 bytes message, which is much lower than 65.8 μs of UDP

(40.1% improvement), and a maximal bandwidth of 545.2 Mbps compared with 396.6 Mbps of UDP (37.5% improvement).

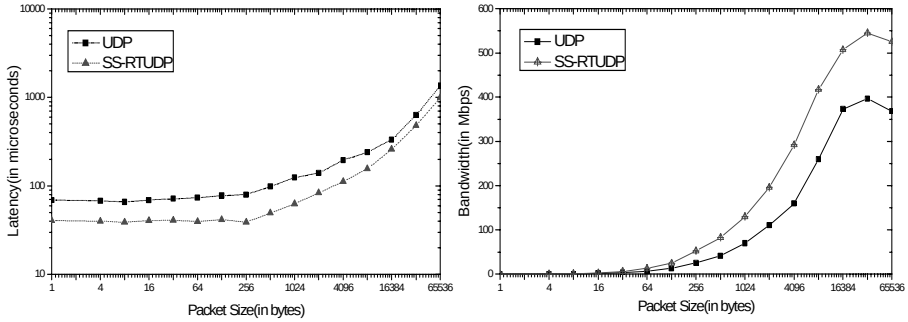

Fig. 8. Latency and bandwidth over Gigabit Ethernet

4.2 Real-time Performance Evaluation

For two nodes connected directly, an additional long-time and high arriving rate TCP-stream (occupies over 80% of physical available bandwidth) is added. The experimental results in Fig. 9 show that even in very congested environment, SS-RTUDP can maintain normal and steady performance.

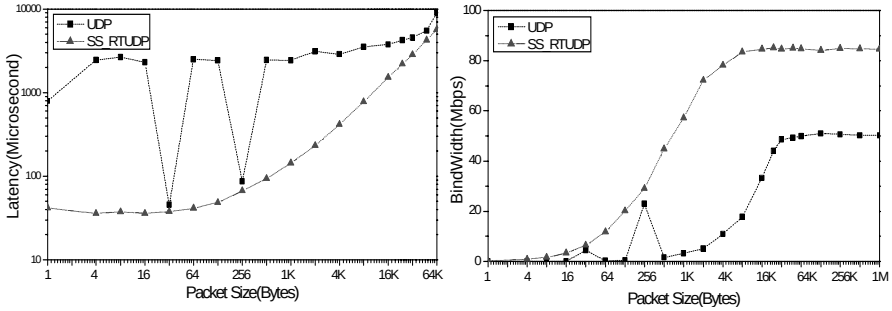

Fig. 9. Latency and bandwidth measurements under congested environments

When 4 cluster nodes are connected through the Ethernet switch shown in Fig.10, the total packet arrival rate from node 1~3 to node 4 through the switch is 105.51 Mbps. Based on the results of Fig. 7, we set 300 μs as the transferring deadline to a RT packet and 1500 μs to a NRT packet. Table 4 shows the experimental results.

Table 4. Packet loss rate measurement

Condition	Type	Sent	Lost	Loss rate
Without Traffic Smoother	RT	360000	56881	15.8%
	NRT	2250000	380250	16.9%
With Traffic Smoother	RT	360000	265	0.074%
	NRT	2250000	507329	22.6%

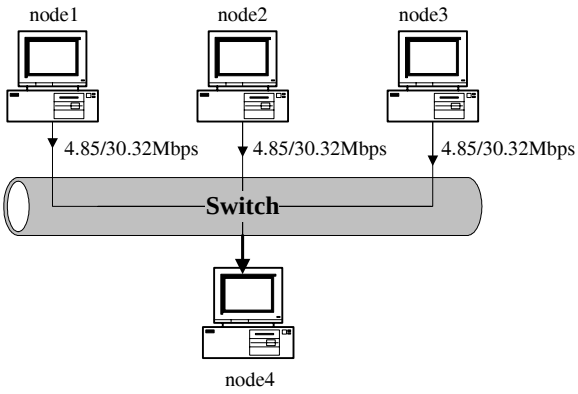

Fig. 10. Multiple nodes environment for real-time performance tests

From Table 4 we find that the *RT* packet loss rate reduces from 15.8% (without real-time traffic smoother) to 0.074% (with real-time traffic smoother), while the *NRT* packet loss rate increases from 16.9% to 22.6%. The results prove that the real-time traffic smoother provides good performance guarantee for real-time communications.

5 Conclusions and Future Works

In this paper, we propose a lightweight real-time network communication protocol for commodity cluster systems, called SS-RTUDP. Through eliminating data copies in communication data path, simplifying the data checksum calculations, adopting fragmentation mechanism for large real-time packets, SS-RTUDP provides both lower latency and higher bandwidth than original UDP protocol over Fast/Gigabit Ethernet, especially over Gigabit Ethernet. Pre-allocated network resources and additional real-time traffic smoother also provides good real-time network communication performance for commodity cluster systems.

Many works need to be improved to current implementation of SS-RTUDP, such as lightweight interrupt processing and true zero-copy data receiving path. In the next step, we will implement the SS-RTUDP protocol under real-time micro kernel to enhance the real-time performance over Gigabit Ethernets. Another goal is to plant SS-RTUDP protocol to embedded control systems.

References

[1] S. Di and W. Zheng, "Reduced Communication Protocol for Clusters", *Proceedings of Advances in Parallel and Distributed Computing.* Shanghai, China, 1997, pp.314-319.

[2] R. Buyya, *High Performance Cluster Computing Architectures and Systems*, Volume 1, 1st Edition, USA, Prentice-Hall, Inc., 1999, pp.182-184.

[3] A. F. Diaz, J. Ortega, A. Canas, F. J. Fernandez, M. Anguita and A. Prieto, "The lightweight protocol CLIC on Gigabit Ethernet", *Proceedings of International Parallel*

and Distributed Processing Symposium(IPDPS'2003), April 2003.

[4] K. Christian, M. Michel, and R. Felix, "Speculative Defragmentation – A Technique to Improve the Communication Software Efficiency for Gigabit Ethernet", *Proceedings of the 9th International Symposium on High-Performance Distributed Computing*, Pittsburgh, PA, USA, 2000, pp.131-138.

[5] K. A. Skevik, P. Thomas, G. Plagemann, and V. Goebel, and P. Halvorsen, "Evaluation of a Zero-Copy Protocol Implementation", *Proceedings of the 27th EUROMICRO Conference,* Warsaw, Poland, 2001, pp.324-330.

[6] G. Chiola and G. Ciaccio, "Efficient parallel processing on low-cost clusters with GAMMA active ports", *Parallel Computing*, Vol.26, 2000, pp.333-354.

[7] A. F. Díaz, J. Ortega, A. Cañas, F. J. Fernández, and A. Prieto, "The Lightweight Protocol CLIC: Performance of an MPI implementation on CLIC", *IEEE International Conference on Cluster Computing (CLUSTER'2001)*, October, 2001, pp.391-398.

[8] K. Ghouas, K. Omang and H. Bugge, "VIA over SCI - Consequences of a Zero Copy Implementation, and Comparison with VIA over Myrinet", *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'2001)*, San Francisco, California, USA, 2001, pp.1632-1639.

[9] Y. Chen, Z. Q. Jiao, J. Xie, Z. H. Du, and S. L. Li, "Design and Implementation of a High Performance VIA Based on Myrinet", *Journal of Software*, Vol.14, No.2, 2003, pp.285-292.

[10] S. Pakin, V. Karacheti, and A. Chien, "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors", *IEEE Parallel and Distributed Technlogy*, Vol.5, No.2, April/June, 1997.

[11] L. Prylli and B. Tourancheau, "BIP: a new protocol designed for high performance networking on Myrinet", *Proceedings of Workshop PC-NOW at IPPS/SPDP98,* April, 1998, pp.472-485.

[12] T. Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: a user-level network interface for parallel and distributed computing", *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, December, 1995.

[13] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal, "User-level Network Interface Protocols", *IEEE Computer*, November, 1998, pp.53-60.

[14] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead". *IEEE Communications Magazine*, Vol.27, No.6, June 1989, pp.23-29.

[15] C. C. Chou and K. G. Shin, "Statistical real_time channels on multiaccess networks", *IEEE Transaction on Parallel and Distributed Systems*, Vol.8, Aug. 1997, pp.769-780.

[16] R. L. Cruz, "A Calculus for network delay, Part I: Network Elements in Isolation", *IEEE Transaction on Information Theory*, Vol.37, No.1, Jan. 1991, pp.114-131.

[17] NetPIPE: A Network Protocol Independent Performance Evaluator, http://www.scl.ameslab.gov/netpipe/.