Transmission Timer Approach for Rate Based Pacing TCP with Hardware Support

Katsushi Kobayashi
ikob@koganei.wide.ad.jp

Abstract— **Transmission timer framework is proposed for rate based pacing (RBP) TCP, in order to mitigate a burstiness in TCP slow-start. In this approach, host software specifies the time for each packet that should be sent out, and gives a precise interframe gap for the data stream. An RBP TCP implementation including a transmission timer aware network interface card is demonstrated. Also, a quantitative analysis of the burstiness in TCP slow-start is done with a burstiness index that we are proposed.**

I. INTRODUCTION

TCP slow-start brings an exponential congestion window growth until detecting congestion. After TCP slow-start, the congestion window is expected to be an appropriate for the current end-to-end network capacity, i.e. the bandwidth delay product. Current TCP slow-start implementation exhibit a burstiness packet traffic behavior, since window size jumps up each round-trip-time (RTT) period. This burstiness leads a packet losses on the queue of bottleneck node, even though the congestion window is very small compared with the bottleneck bandwidth capacity. This loss makes slow-start to stop in early phases, and suppress throughput.

Rate-based pacing (RBP) for avoiding this burstiness has been studied to improve TCP performance with both simulation and implementation approaches [1], [2]. On fast network, precise RBP management is required, e.g., packets should be sent 60 μsec. interval in the case of 200 Mbps bandwidth and 1500 bytes packet size. This interval is very small compared with OS scheduler's order (1 - 10 msec.). It is difficult to accomplish RBP in this order without hardware support.

Hiraki *et al.* proposed a precise packet pacing framework with network interface card (NIC) hardware support [3]. TCP stack in operating system computes interframe gap with congestion window size and RTT for every TCP packets. TCP stack passes the packet with the interframe gap to the NIC device via IP and ethernet layers. The NIC maintains all TCP connections in a table, and schedules packet for each flow. When NIC takes a packet to transmit with interframe gap, NIC identifies the connection using the header data, and schedules the transmission time computed with the last packet transmitted time recorded in the table. NIC should interpret transport protocol, and should have connection management, i.e. initiate, lookup, removal, functions. Takano *et al.* proposed an precise pacing without any special hardware inserting dummy packet [4]. Ethernet layer module computes the packet gap with the window size and the RTT obtained with calling back upper layer protocol control block, i.e. TCP control block (TCPCB) and inserts appropriate interframe gap using dummy packets. In RBP approaches above mentioned, lower layer protocol module, ethernet driver or NIC hardware, depends on upper transport protocols. This layer violation arises implementation issues for the modules. Transport protocol beyond TCP, e.g. TFRC, DCCP, SCTP are being standardized and developed, and other newer protocols will be coming . Precise RBP may be also useful for them, and more important for the protocol lacking of acknowledge pacing mechanism shown in TCP. For minimizing the effort to support new protocols, more simplified RBP approach is expected.

National Institute of Communications Technology, JAPAN

In this paper, we show a precise packet pacing framework using transmission timer for each packets with hardware support in section II. We present a transmission timer NIC implementation using network processor, and a precise RBP TCP stack implementation on FreeBSD, in III and IV, respectively. In section V, we demonstrate the performance of our RBP TCP implementation in TCP slow-start and present an index for burstiness.

II. TRANSMISSION TIMER

Interframe gap, g, in RBP TCP can be obtained as the following based on RTT, $cwnd$, and MSS as round-trip-time, congestion window size, and maximum segment size, respectively:

$$g = \frac{RTT}{cwnd} \cdot MSS \tag{1}$$

When higher data rate communication, interframe gap in RBP is smaller than the scheduler tick of operating system as mentioned in the previous section. During TCP slow-start, congestion window $cwnd$ size is increased by

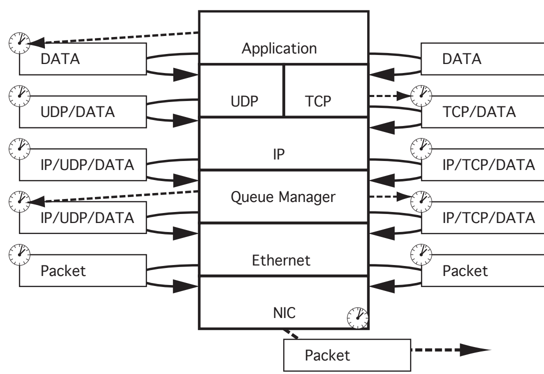


Fig. 1. Transmission timer management in Host

receiving each acknowledge packets, and $cwnd$ grows in exponentially. Therefore, the interframe gap for RBP is frequent changing in short period. Usual queuing implementation as ALTQ and QDISC provide various queuing disciplines, and enables rate control for each and/or group of flow as CBQ, WFQ, etc. However, these framework focus fairness and prioritization among multiple flows in longer term more than RTT, and cannot follow dynamic interframe gap changing requirement from transport layer. Some packets should be sent in sooner than other packets even in the same flow, e.g. retransmit segment on TCP. For minimizing implementation effort, the framework should not call back upper layer protocols. We summarize the requirements as the followings:

- A precise packet pacing more than OS scheduler granularity.
- Follow frequent interframe gap (rate) changing.
- Re-order packets at sender host even on the same connection.
- Free from upper layer (transport) protocols.

For accomplishing the above, we adopt per packet transmission timer (Fig. 1). The packet transmission timer field is added on the data structure handling network packet in operating system, e.g. `mbuf` in FreeBSD, `skbuf` in Linux. The transmission timer represents when the packet should be sent out, and also requires enough wrap-around period. The timer can be specified in system time form as `timeval` structure in UNIX. A network device schedules the packet transmission based on the timer value. If the timer has an appropriate granularity for the interface bandwidth, precise RBP could be accomplished. Transmission timer enables any type of interframe gap control not only RBP in TCP slow-start but also packet re-order before sent out. Note that some transports are sensitive for packet re-order, and the transmission timer value must be decided keeping with the packet order in this case. A packet is simply passed to a lower layer with transmission timer. A lower layer module is not required to know transport protocol and to call back the protocol control block of the transport.

The timer value can be manipulated by every layer. In case of TCP slow-start, the timer value is specified using $cwnd$ and RTT by TCP stack. In other transport framework, the timer value can be obtained with either window based control or rate based. On usual queuing approaches, the transmission time value can be decided by queuing manager, possibly taking account an existing timer value specified by upper layer. Also, application program can specify timer value for each UDP packet, if an appropriate API is provided.

III. Network interface support

In our approach, a packet data with transmission timer attribute is passed from CPU to NIC. Transmission timer expressed in system time style is useful on operating system, since it is difficult to predict a processing delay in operating system, and wall clock enables to share common timer among kernel sub-modules. However, usual BUS system does not provide real time clock sharing function, and a NIC attached on host BUS cannot share it between the CPU.

A couple of approaches are possible to notify transmission timer information to NIC lacking common real time clock system. The first is that NIC has another real time clock, and the clock is synchronized with the host one. The second is that the transmission timer is converted from absolute time expression to relative time, when the device driver initiates packet transmission. The first approach requires not only packet transmission timer function but also a real time clock system on NIC. The synchronize action should be done in carefully, since clock adjustment action may cause unexpected behavior of packet transmission timing. Using the second does not require any real time clock related mechanism on NIC side. A fluctuation would be happen due to data transmission delay from host to NIC. However, this fluctuation effect might be a limited in the current host performance compared with network bandwidth.

We developed a NIC having per-packet transmission timer function for precise RBP. The NIC is based on Intel IXP2400 network processor development platform hardware, ENP2611 from Radisys Corp. ENP2611 has 64-bit/66MHz PCI BUS, and has three gigabit ethernet interfaces. The NIC function is implemented as a software of ENP2611 board. The ENP2611 behaves as a NIC device attached on host PCI BUS with mapped I/O, DMA, and Tx, Rx interrupt. Although ENP2611 has real time clock system itself, we adopt relative transmission timer expres-

sion when passing a packet to NIC. The NIC supports three transmission classes, transmission timer, high priority, and best effort class. High priority and best effort classes provide the same feature as usual NIC.

The NIC can manage packet transmission time in 26.67 nsec. granularity. This granularity corresponds to the packet timing control limit of IXP2400 as 16 cycles of 600MHz system clock. The transmission timer class manages a packet using two level schedulers, rough and precise one. The rough scheduler has 4,096 ring time slots. Each slot is given 27.3 μsec. (1,024 $\times$ 26.67 nsec.) time window. Therefore, the NIC can make up to 111 msec. (4,096 $\times$ 27.3 μsec.) delays. This delay limit is larger than usual kernel scheduler tick, and can support any delay combined with the NIC and operating system scheduler. The ring time slots proceed each time window unit. The packet is assigned to an appropriate slot using own transmission timer value. Each slot can accept up to 15 packets, the packets in a slot are sorted into transmission timer order. If a packet is found in the first slot, the precise scheduler takes all packets, and schedules it to be sent out.

When host requires hardware packet pacing, the host specifies the timer value in the DMA descriptor as $1/2^{32}$ second unit. NIC schedules the packet transmission time using the timer value.

IV. Device driver and TCP implementation

We implemented transmission timer framework on FreeBSD 4.10 including the device driver for the NIC above mentioned. We add a transmission timer field in `mbuf` structure as `struct timeval` format. The device driver compares the timer field and the system time. If the timer field represents later time than the system time, the device driver computes the difference of them, and specifies it into the DMA descriptor to handle corresponding packet data. Otherwise, the driver does not specify any delay, and NIC sends the packet as soon as possible.

We also implemented RBP into the TCP New Reno stack. Note that the original TCP New Reno stack performance of FreeBSD shown poor performance under fast long network. The original TCP stack does not cache the `mbuf` pointer last transmitted, and has to traverses unacked packet data already sent out from the oldest to the recent one at each packet transmission. This traverse process suppresses TCP performance in larger congestion window. We replaced this buffer traverse part in `tcp_output()` borrowed from NetBSD 2.0. When sending out a TCP segment in slow-start, the interframe gap time is computed using the measured RTT and the $cwnd$ packet by packet. The transmission timer value is obtained as the sum of the interframe gap and the timer value of the last packet cached in the TCPCB. If the transmission timer value shows kernel scheduler tick or more later, the sending out action is postponed to the next kernel scheduler interrupt. This RBP is similar to other usual software based approaches, and can enable a rough controlled RBP without NIC support [3]. For precise interframe gap computation, an accurate RTT measurement is required. Original TCP already has RTT measurement function. However, this measurement unit is based on kernel scheduler tick, and is insufficient accuracy for precise RBP. Another RTT measurement function is implemented into TCP stack. The accuracy of measurement is in μsec. In our RBP TCP implementation on FreeBSD, less than 1,000 lines hacking was required except the device the driver of the NIC. Almost all modification is mainly done on TCP stack. The modification would be a light compared with the original TCP related code about 10,000 lines on FreeBSD. Also, we didn't make any modification in IP stack, since the timer information is handed over with `mbuf`.

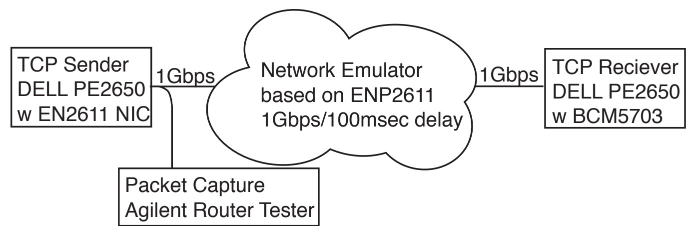


Fig. 2. Experimental setup for TCP performance evaluation

V. RBP TCP performance

The performance evaluation configuration is shown in Fig. 2. We use DELL Poweredge 2650 server having 3.06GHz Xeon CPU with 1GB RAM for both sender and receiver. The gigabit ethernet NIC of the sender side is the NIC we developed, and the sender side is Broadcom 5703 equipped on Poweredge 2650 server. The OS of all hosts is FreeBSD 4.10. TCP delayed acknowledge function on the receiver side is disabled for accurate RTT measurement. The network emulator box is also based on ENP2611 [5]. The emulator provides 100 msec. delay between sender and receiver with wire-rate speed at full-duplex 1000BaseTX. All experiments have been conducted with iperf 1.7.0, and packets are captured by Agilent Router Tester system with 20 nsec. accuracy time stamp.

Our RBP implementation accesses real time clock at every packet. The real time clock access is expensive for the host, since the real time clock is usually attached outside of CPU. We made preliminary evaluation for the effect of real time clock access to the RBP TCP performance. The configuration is the same shown in Fig. 2,

	elapsed time	accuracy of result
`microtime()`	~ 5 μ sec.	μ sec.
`getmicrotime()`	< 30 nsec.	depend on scheduler clock
New `getmicrotime()`	120 nsec.	μ sec.

Note that the elapsed time results of the above including the overhead of reading TSC. The overhead of reading TSC is a larger than original `getmicrotime()`. Therefore, the result of original `getmicrotime()` includes unnegligible error.

TABLE I
PERFORMANCE DIFFERENCE BETWEEN GETTING CURRENT TIME.

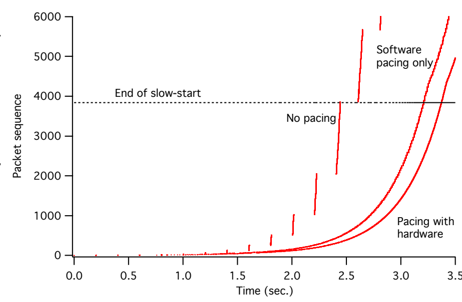


Fig. 3. Sequence of packet plot of original TCP, software-only RBP TCP, and RBP with hardware support. (HZ=100)

and Broadcom 5703 is used for TCP sender interface instead of ENP2611 NIC with no delay on the network emulator. When using original real time clock access, `microtime()`, TCP data transfer throughput is suppressed 20% due to CPU overload. Faster call is prepared for FreeBSD as `getmicrotime()`. This call refers the cached value updated at each scheduler timer period. Therefore, the accuracy of `getmicrotime()` depends on the schedular timer, and the accuracy is not an enough for precise RBP. We implemented a new call that returns a more accurate clock value and is faster than usual calls. The new call is implemented with CPU clock time stamp counter (TSC) in Intel x86 architecture. TSC counts CPU clock cycles, and is used for performance estimation for a program code in an accurate granularity [6]. When updating the clock value cache for `getmicrotime()`, the TSC value at the schedular period is also cached. The new call evaluates the gap between the current TSC value and the cached one, and corrects the cached clock value for `getmicrotime()` with the gap of TSC. We compared the the performance getting a real time clock value using TSC. The result is shown in Table I. If `microtime()` is called at each packet sending, the transmit throughput is limited up to 200K packet/sec. only with real time clock access, i.e. up to 2.4 Gbps in the case of 1500 Bytes packet size. When replacing with the new call, the throughput is 300M packet/sec. that corresponds more than 3 Tbps transmit bandwidth. The `microtime()` replaced version RBP TCP reaches the same data transfer performance with the original TCP as 940 Mbps.

The TCP packet sequence plots are shown in Fig. 3. Although we evaluated on the same configuration, the window growth trends are clearly different. This is because the RTT measurement results are somewhat larger on RBP probably due to a delay added on RBP process.

We analyzed RBP TCP behaviors in slow-start using the interframe gaps with captured packets. Fig. 4 shows the interframe gap distribution on various configurations. Since the limitation of capturing system packet buffer (60MB), the maximum window size in experiments are limited to 2.5MB, 1770 segments $\times$ 1480 MSS. In all RBP TCP, the rate of congestion window increase in suppressed compared with the original TCP. The interframe gap fluctuation is keeping under 50 μsec. on RBP with hardware pacing during slow-start Fig. 4(a) and (d). In software only RBP, the fluctuation of interframe gap depends on the kernel scheduler tick, 10 msec. and 1 msec. in Fig. 4(b) and (e), respectively.

Fig. 4 plots may present the RBP TCP mitigates a burstiness of original TCP. However, it is difficult to understand how much burstiness is mitigated. We define a quantitative index for comparing burstiness in TCP slow-start. We define a virtual bottleneck bandwidth $BW(t)$ depending on connection time t using exponentially growth $cwnd$ and RTT. If we assume no delayed acknowledge mechanism, the bandwidth is as the following:

$$cwnd = 2^{\frac{t}{RTT}} \cdot MSS \qquad (2)$$

$$BW(t) = cwnd\ /RTT = 2^{\frac{t}{RTT}} \cdot MSS\ /RTT \qquad (3)$$

Interframe gap $\sigma_i(t)$ for i-th packet assuming the virtual bottleneck bandwidth at the packet transmission time t_i is estimated as the following:

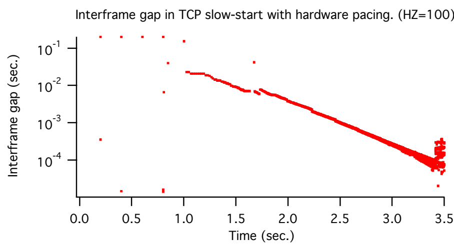
$$\sigma_i(t) = \frac{MSS}{BW(t)} = \frac{RTT}{2^{\frac{t_i}{RTT}}} \qquad (4)$$

A delay d_i until i-th packet starting transmission can be obtained taking into account accumulation of the buffered packets previous sent as:

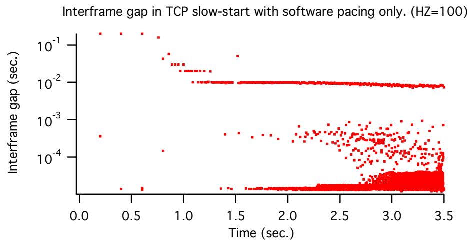
$$d_i = max\left(d_{i-2} + \sigma_{i-1} - (t_i - t_{i-1}), 0\right) \qquad (5)$$

Burstiness index b_i for each packet can be obtained normalizing by RTT, and burstiness index of the connection B is obtained as the followings:

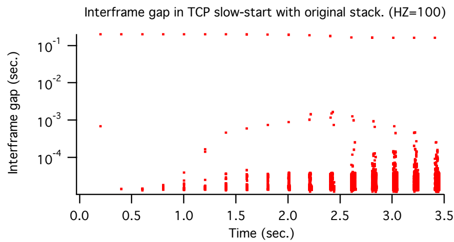
$$b_i = d_i\ /RTT \qquad (6)$$



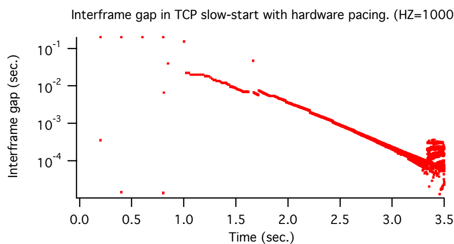
(a)



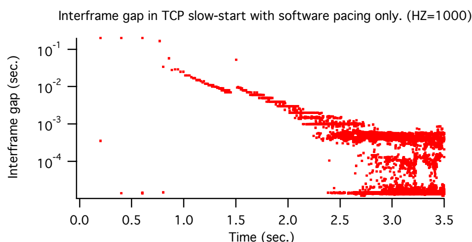
(b)



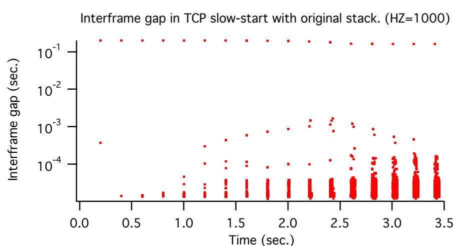
(c)



(d)



(e)



(f)

Fig. 4. interframe gap plot in TCP slow-start

Scheduler tick	10 msec.	1 msec.
RBP with HW	1.5×10^{-7}	1.5×10^{-7}
Software only RBP	4.1×10^{-4}	5.9×10^{-6}
no RBP	0.31	0.31

TABLE II

BURSTINESS INDEX IN SLOW-START

$$B = \sum_{i}^{N} b_i \;/N \tag{7}$$

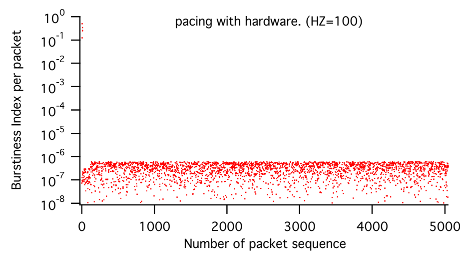
Larger index value represents more burstiness traffic behavior in slow-start. The index of original TCP slow-start without any limit might be reach 0.5, since the virtual delay plot should be saw-tooth pattern. Fig. 5 and Table II shows the burstiness index per packet plot and connection, respectively. In order to avoid the error just after the connection startup, we ignore first 20 packets for burstiness index of slow-start. 5,000 captured data packets have been taken into account in order to evaluate burstiness index of slow-start.

The burstiness indices of connection without RBP TCP are quite large compared with other RBP TCP results. Software only pacing contributes to mitigate the burstiness of TCP not only slow-start also overall time. The result of burstiness index on software only TCP RBP on 1 msec. scheduler tick, 5.9×10^{-6}, shows comparable with hardware support results, 1.5×10^{-7}. In burstiness plot for software only RBP, Fig.5(b) and (e), a small burstiness is still found at just after connection start. The order of per packet index is 10^{-2} and 10^{-3} on 10 and 1 msec. scheduler clock tick, respectively. On hardware RBP support, burstiness indices don't depend on kernel scheduler clock tick resolution, and burstiness of slow-start is clearly suppressed compared with other approaches (Fig.5(a) and (d)).

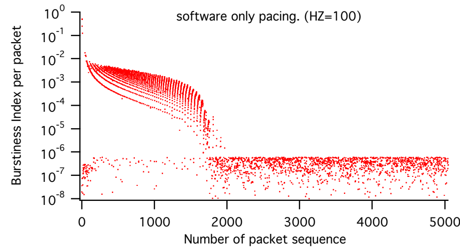
VI. CONCLUSION AND FUTURE WORK

We present that the packet pacing using transmission timer for each packet provides simple collaboration framework through network layers, not only operating system but also network interface hardware. We also propose a burstiness index for TCP slow-start using virtual bottleneck. The burstiness index might help to improve TCP slow-start performance with giving quantitative scale.

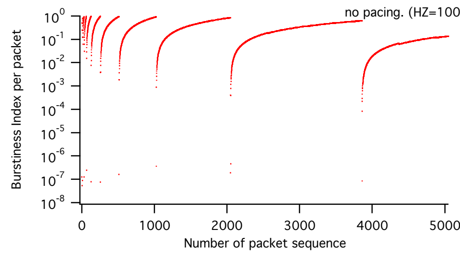
We also implemented transmission timer feature on UDP. FreeBSD provides `SO_TIMESTAMP` socket option, which returns the datagram received time as ancillary data in `timeval` format [7]. Linux also provides the the similar option. The options can be used only when receiving a datagram. We extended `SO_TIMESTAMP` option to both



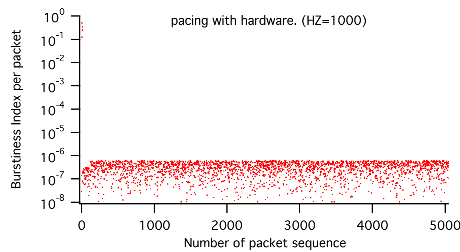
(a)



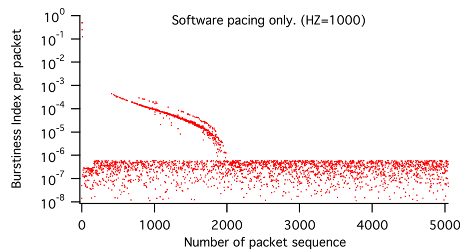
(b)



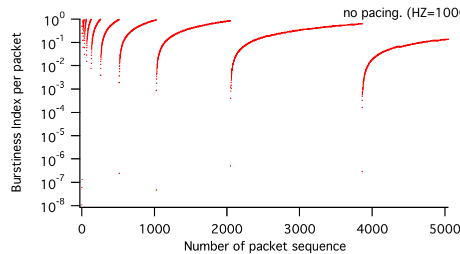
(c)



(d)



(e)



(f)

Fig. 5. Per Packet burstiness index plot in TCP slow-start

sending and receiving. When an application enables the option, the application can specify the appropriate sending time for each packet using `sendmsg()`. We believe the option is a useful for a precise timing control compared with the usual way as busy-loop for making an interval.

REFERENCES

[1] V. Visweswarauah and J. Heidemann,"Rate Based Pacing for TCP", `http://www.isi.edu/lsam/publications/rate_based_pacing/`, 1997

[2] A. Aggarwal *et al.*,"Understanding the Performance of TCP Pacing", *in Proceedings of IEEE Conference on Computer Communications(INFOCOM)*, Mar. 2000.

[3] K. Hiraki *et al.*,"End-node transmission rate control kind to intermediate routers - towards 10Gbps era", *in Proceedings of the 2nd International Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2004.

[4] R. Takano *et al.*,"Design and Evaluation of Precise Software Pacing Mechanism for Fast Long-Distance Networks", *in Proceedings of the 3rd International Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2005.

[5] K. Nakauchi and K. Kobayashi,"Studying Congestion Control with Explicit Router Feedback Using Hardware-based Network Emulator", *in Proceedings of the 3rd International Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2005.

[6] Intel Corp., "Intel Architecture Software Developer's Manual, Vol. 2, Instruction Set Reference", 1999

[7] W.R. Stevens, "UNIX Network Programming Vol.1, Second Edition:Networking API's Sockets and XTI ", Prentice Hall, 1998