논문 01-26-2A-9

PWD 적용 여부에 따른 TCP NewReno의 초기 동적 전환에 관한 분석

주 창 회*, 정회원 박 세 웅**

Analysis of Start-up Transition Dynamics of TCP NewREno

Changhee Joo*, Saewoong Bahk** *Regular Members*

요 약

기존의 TCP Reno는 하나의 정체 윈도우 내에 여러 개의 패킷 손실이 발생하면 timeout이 발생하여 성능을 저하시켰으므로 이를 개선시키고자 NewReno가 제안되었다. NewReno는 송신자 측의 알고리즘을 변화시켜서 여러 개의 패킷 손실을 timeout없이 복구할 수 있으며 여러 다양한 선택적 기능을 가지고 있다. 그러나 1 RTT 동안에 한 개의 손실된 패킷만을 복구할 수 있으므로 많은 수의 패킷 손실이 발생했을 경우에 복구하는 데에 많은 시간이 요구된다. 일반적으로 TCP는 초기에 많은 패킷의 손실을 겪게 되는 경우가 많으므로 NewReno의 초기 기간 복구 과정은 TCP의 성능에 큰 영향을 끼칠 수 있다. 또한 네트워크의 대역폭이 커지고 WWW와 같이 짧은 기간에 끝나는 TCP 연결의 사용이 증가함에 따라 TCP 연결에서 초기 기간 동안의 성능은 점점 중요해진다.

본 논문에서는 TCP NewReno의 초기 동적 전환 과정의 중요함을 보이고 부분적 윈도우 감소 알고리즘을 사용한, 또는 사용하지 않은 TCP NewReno의 자세한 분석을 기반으로 모의 실험을 통해 분석 결과의 타당성을 검증하고 부분적 윈도우 감소 알고리즘을 사용한 NewReno의 장점과 단점들을 지적한다.

ABSTRACT

NewReno has been proposed to improve the performance of TCP by preventing unnecessary timeout, which is due to multiple packet losses from a single window. It can recover multiple packet losses without any changes at receivers, and has a few variants differing in procedures during recovery. But it has a limitation that it can recover only one packet during one RTT. It takes a long time to recover multiple packet losses. Therefore NewReno can still suffer from performance degradation if it experiences multiple packet losses in its start-up period. As the network bandwidth grows and the applications that use short-lived TCP connections increases, the start-up period forms a major part in a connection, and the overall performance depends significantly on its start-up dynamics.

In this paper, we show the importance of start-up transition dynamics of TCP NewReno with and without partial window deflation (PWD) and show their behaviors through extensive simulations. Particularly, we focus on the dynamics during Fast Recovery

I. Introduction

TCP has two important recovery procedures to provide reliable communication between the sender and the receiver. One is the retransmission by timeout and the other is Fast Retransmit followed by Fast Recovery [1] [2]. After a packet is sent, if the corresponding ACK is not received within a calculated retransmission timeout (RTO), the timer expires, and the sender retransmits the

* 서울대학교 전기공학부 데이터통신망 연구실(jchee@netlab.snu.ac.kr)
논문번호 : 00290-0726, 접수일자 : 2000년 7월 26일

packet and restarts its Slow-Start. Fast Retransmit makes the sender retransmit the packet immediately after receiving three duplicate acknowledgements (ACKs) rather than wait for the timer to expire. In TCP Reno, Fast Recovery prevents the sender from entering Slow-Start after a single packet drop from a window.

But in TCP Reno, when multiple packets are dropped from a single window, Fast Retransmit and Fast Recovery can recover only one of the lost packets. The rest are often recovered by Slow-Start followed by timeout [3] [4]. Especially when the sender starts its data transfer without any information about the network capacity, it usually ends up outputting too many packets and thus experiencing multiple packet losses from a window.

When there are multiple packet drops, the first dropped packet is retransmitted by Fast Retransmit. The ACK for the retransmitted packet will acknowledge some but not all of the packets transmitted before Fast Retransmit. It is called a partial ACK. In TCP NewReno the sender responds to a partial ACK by inferring that the indicated packet has been lost, and retransmits the packet [5]. After all the losses are recovered, the sender continues its transfer in Congestion Avoidance. Since this modification avoids timeout and ensuing Slow-Start, the improvement of throughput and the reduction of unnecessary retransmissions are expected.

In this paper we first briefly introduce variants of TCP NewReno in Section II. In Section III, we show the importance of start-up transition dynamics of TCP NewReno. Then, we compare NewReno variants through simulations in Section IV, followed by the conclusions in Section V.

Ⅱ. TCP NewReno variants

TCP NewReno includes an important change from Reno algorithm at the sender. It eliminates unnecessary timeout when multiple packets are lost from a window. NewReno treats a partial ACK as an indication that the packet immediately following an acknowledged packet in the sequence has been lost and should be retransmitted. The retransmitted packet, if not dropped, causes another partial ACK or a positive ACK[1]. Thus TCP NewReno can recover from multiple packet drops without unnecessary timeout.

[5] proposes several possible variants of NewReno according to the response to partial ACKs: when to reset a retransmit timer after a partial ACK, how many packets to be retransmitted after each partial ACK, and whether the sender avoids multiple Fast Retransmits caused by the retransmission of packets already received at the receiver. Since we investigate the situation where multiple packets in a window are dropped in transition, we use NewReno variants that have the following properties.

- Resets the retransmit timer after receiving each partial ACK (Slow-but-Steady variant).
- Retransmits a single packet after receiving each partial ACK.
- Avoids multiple Fast Retransmits (Less careful variant).

Another classifier of TCP NewReno algorithms is whether the partial window deflation (PWD) algorithm is used or not [5]. Using the PWD algorithm, the sender considers another packet is dropped. On receiving a partial ACK, the sender retransmits the unacknowledged packet indicated by the partial ACK. It also deflates *cwnd* by the amount of data newly acknowledged instead of resetting *cwnd* to the Slow-Start threshold (*ssthresh*), and adds back one packet to *cwnd*. Then it sends a new packet if new *cwnd* permits. This algorithm attempts to prevent packet burstiness when Fast Recovery eventually ends, by making approximately *ssthresh* amount of data outstanding in the network. In contrast, NewReno without PWD reduces *cwnd* to *ssthresh* when it receives a partial ACK. If multiple packets are dropped from a window, the sender can send

1) The positive ACK informs the sender that all packets sent before Fast Retransmit are successfully received at the receiver, that is, all lost packets are recovered.

only a few new packets during a lengthy recovery period and can not avoid packet burstiness after Fast Recovery.

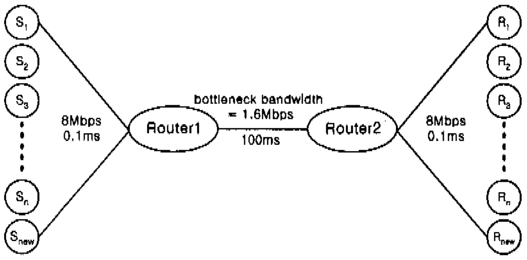


Fig. 1 Simulation topology

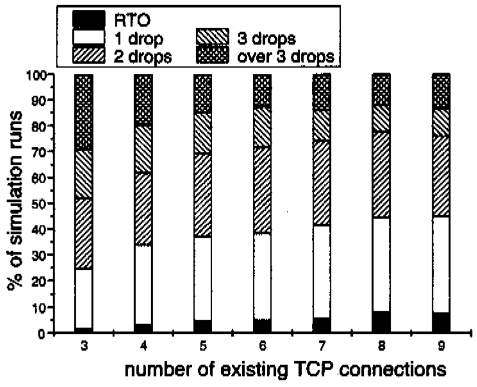
Ⅲ. Simulations: Start-up Transition Dynamics

Since the sender has no information about the network capacity in its start-up period, it can not choose a proper initial *ssthresh* and often sends too many packets, leading to packet losses. Therefore TCP usually makes its first transition from Slow-Start to Congestion Avoidance by loss detection when *cwnd* is below *ssthresh*. We investigate how often and how many a TCP sender experiences multiple packet losses during its start-up period.

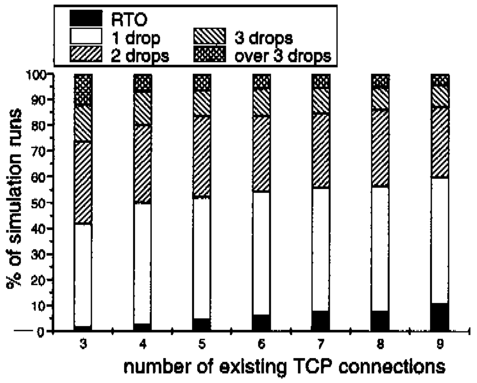
All our simulations are done with the NS simulator (version 2.1b5) [6]. Fig. 1 shows a simple topology for our simulations. The TCP connections are driven by long-lived FTP sessions. The bottleneck bandwidth is 1.6Mbps unless stated otherwise. We use RED [7] as a queue management policy at Router1. The physical queue length limit is 60 packets, the minimum threshold (min_{th}) 5 packets, the maximum threshold (max_{th}) 15 packets, and the linear term (max_p) 0.1. We assume that all packets have the same size of 1000 bytes. The advertised receiver window size is set to infinite, to not affect the sender window size. Other parameters are set to default values.

As shown in Fig. 1 we establish n TCP connections at time 0 and wait 50 sec for them to be stable in the steady state. Then we make a new TCP connection with a random delay and trace it to see how many packets are dropped before the sender detects the first drop. We change n from 3 to 9 so as to accordingly reduce the fair share per connection in the simulation. Fig. 2 shows how the newly starting TCP connection detected its packet loss. In the figure, 'RTO' represents the fraction of losses detected by timeout whereas the others do them by Fast Retransmit. In case Fast Retransmit is triggered, we count how many packets sent before Fast Retransmit are dropped, and classify them into 1 drop, 2 drops, 3 drops, and over 3 drops. Among them, 2 drops, 3 drops, and over 3 drops are multiple drops.

Both Figs. 2(a) and 2(b) show that multiple drops frequently occur. Multiple drops is dominant even when the number of existing TCP



(a) Without delayed ACK



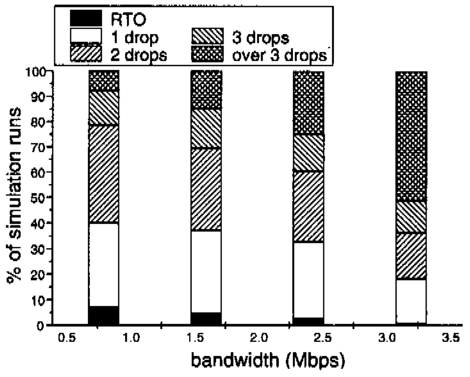
(b) With delayed ACK

Fig. 2 Statistics of detected drop patterns with 1.6Mbps bottleneck bandwidth.

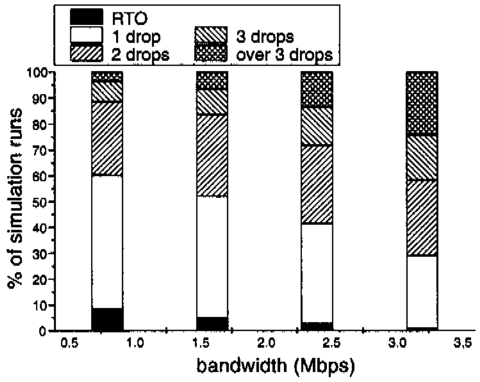
connections is small in Fig. 2(a). Although the percentage of multiple drops decreases as the number of existing connections increases, it still occupies a significant portion: even in case of 9 competing connections, the percentage of more than 3 drops is 13.5%, 3 drops 10.6%, and 2 drops 31.1%. Hence, the total percentage of multiple drops is 56.2% while 1 drop 37.4% and RTO 7.5%. We find that the number of multiple drops reduces by smaller amount as the number of existing TCP connections increase. We can also observe similar results in Fig. 2(b). The percentage of multiple drops is smaller but still significant. In case of 9 competing connections, the percentage of over 3 drops is 4.6%, 3 drops 8.3%, and 2 drops 27.1%. The total percentage of multiple drops is 40% while 1 drop 49.4% and RTO 10.6%. Since the delayed ACK policy makes TCP more conservative, the sender has smaller number of multiple drops. The percentage of multiple drops is over 55% without delayed ACK and over 40% with delayed ACK in entire simulations.

Multiple packet drops makes NewReno suffer from performance degradation in its start-up period because NewReno can recover only one packet during one round trip time (RTT). It takes *d* RTTs to recover all losses when *d* packets are lost. If the sender can retransmit only lost packets but can not send new packets during a lengthy recovery period, it obviously leads to the performance degradation. Also, if the sender does not have as many packets in flight as *cwnd* when the recovery process ends, it will send packets in a burst. This packet burstiness may cause various problems in the network.

We now simulate with other bottleneck bandwidths ranging from 0.8Mbps to 3.2Mbps. In this simulation experiments, we fix the number of existing TCP connections to 5 while retaining the other settings. The results are shown in Fig. 3. The number of packet drops within a window greatly increases as the bottleneck bandwidth gets larger. The sender which has a larger fair share of the bandwidth is more likely to go over its



(a) Without delayed ACK



(b) With delayed ACK

Fig. 3 Statistics of detected drop patterns with 5 TCP connections.

fair share due to the exponential increase of *cwnd* in Slow-Start. So the percentage of multiple drops increases and that of RTO decreases with larger bandwidth irrespective of whether delayed ACK is used. But this property does not stem from small RED thresholds settings. In case of the queue length of 100 packets, min_{th} of 10 packets, max_{th} of 30 packets and max_p of 0.1, the percentage of RTO, 1 drop, 2 drops, 3 drops and over 3 drops is 0.8%, 16.7%, 19.6%, 14% and 48.9%, respectively when the bottleneck bandwidth is 3.2Mbps and delayed the ACK is not used. These results are similar to the values observed in the previous simulation, which uses RED settings of the queue length of 60 packets, min_{th} of 5 packets, max_{th} of 15 packets and max_p of 0.1.

When the network has more bandwidth and a

Table I. Start-up transition performance of NewReno with and without PWD

	NewReno without PWD	NewReno with PWD
New packets sent during the k-th RTT-period	$N_1 = \max(0, W-d), N_k < W$	$N_k = \max(0, W-d+k-1)$
Total new packets sent during Fast Recovery	$<2W$	$\frac{1}{2}W(W-1) - \max(0, \frac{1}{2}(W-d)(W-d-1))$
Packet burst size right after Fast Recovery	$\leq W$	1

larger queue size, a TCP sender is likely to have a larger window size. This will cause more packet drops within a window and results in a longer recovery period. Furthermore, many applications use TCP for relatively small amount of data transfer and create several TCP connections at a time. This suggests that good TCP start-up performance will become increasingly important.

The anaysis of transition dynamics is shown in [8]. They analyze the number of new packets sent during recovery and the packet burst size after recovery. Table I. summarizes their analysis results. W denote the sender window size at the time when the sender sends a packet that will be dropped first and d the number of packet drops from a single window. Then A recovery period can be partitioned into d RTT-periods to help us understand how TCP NewReno behaves during recovery. N_k is the number of new packets sent during the k-th RTT-period of recovery. The measurement is done at the end of the k-th RTT-period during Fast Recovery, that is, just after the k-th partial ACK is received.

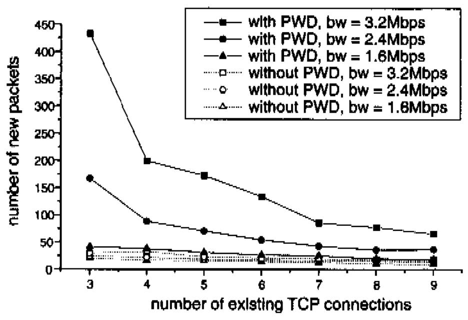
Ⅳ. Simulations: Comparison of TCP NewReno with/witout PWD

We run NewReno with and without PWD at the sender and with and without delayed ACK at the receiver to verify the analysis results. Throughout simulations, we measure the total number of new packets sent during recovery and the size of burstiness after the recovery ends. We use the same topology in Fig. 1.

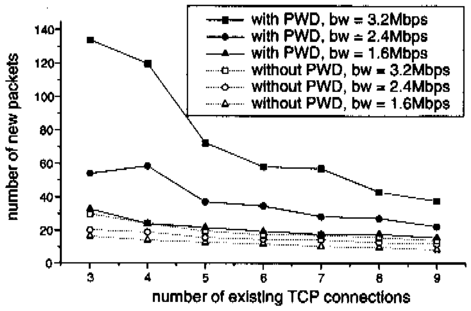
Fig. 4 shows the average total number of new packets sent during recovery with different bottleneck bandwidths. Since timeout makes TCP exit the recovery and start new behaviors, we exclude the cases where timeout occurs before or during recovery. As the number of existing TCP connections increases, both NewReno with and without PWD send fewer new packets. Since the sender has smaller fair share of the bottleneck bandwidth with increasing number of TCP connections, it detects a drop at a smaller window size. Therefore, it sends a few new packets during recovery.

NewReno with PWD sends more new packets than NewReno without PWD for all cases and the differences between them become larger with the increase of the bottleneck bandwidth. The fair share of a connection increases along with the bottleneck bandwidth and makes the sender have a relatively larger window size when detecting a drop. But since the total number of new packets of NewReno without PWD is limited by $2W$ from Table I., it only slightly increases with the bottleneck bandwidth. But in case of NewReno with PWD, since the total number of new packets is related to the square of W, it increases faster. Hence the differences between NewReno with and without PWD get larger with the increase of the bottleneck bandwidth.

Figs. 5 and 6 show the average size of packet burstiness and the size of packet burstiness against the window size. We exclude the



(a) Without delayed ACK



(b) With delayed ACK

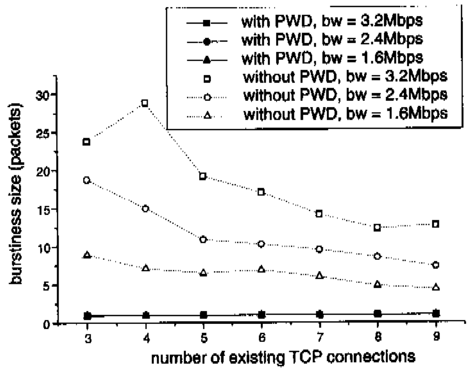
Fig. 4 Average total number of new packets sent during recovery.

simulation runs in which any packet is dropped during recovery. Packet drops during recovery reduce the packet burst size, often down to 0, and cause another Fast Recovery just after the recovery ends. In Fig. 5, we observe NewReno without PWD sends larger packet bursts with larger fair share of the bandwidth. The increase of the fair share of the bandwidth can be caused by decreasing the number of existing TCP connections or increasing the bottleneck bandwidth. When the delayed ACK policy is used, NewReno without PWD has a smaller window size and, thus has a smaller packet burst. But NewReno with PWD sends one packet when the recovery ends regardless of the fair share of the bandwidth.

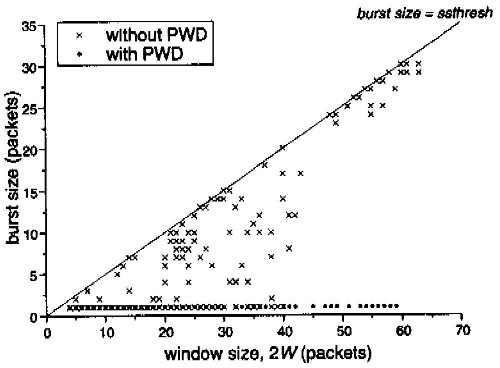
In Fig. 6, we show that the packet burst size of NewReno without PWD is smaller than *W*, and that of NewReno with PWD is exactly one. Each measurement is done when the recovery ends. In Fig 6(a), all crosses are below the oblique line representing *burst size = ssthresh* and all dots indicate the burst size of one. So NewReno without PWD has the packet burst which is limited by *W*, and NewReno with PWD has the packet burst size of one. When the delayed ACK policy is used, the burst size of NewReno with PWD is one or two and that of NewReno without PWD is still limited by *W* in Fig. 6(b). A two-packet burst of NewReno with PWD occurs when the delayed ACK policy omits one duplicate ACK before Fast Retransmit due to its every other acknowledgement mechanism. The omission of one duplicate ACK makes the sender estimate one more packet drop and exit recovery one RTT faster. Then since the sender has *W* - 2 packets in flight at that time, it sends the two-packet burst.

The omission of one duplicate ACK also reduces the total number of new packets by about the number of drops, *d*. Since the sender overestimates the drop by one, its new packets sent in each RTT during recovery are reduced by one. Hence the total number of new packets gets smaller than that of our analysis.

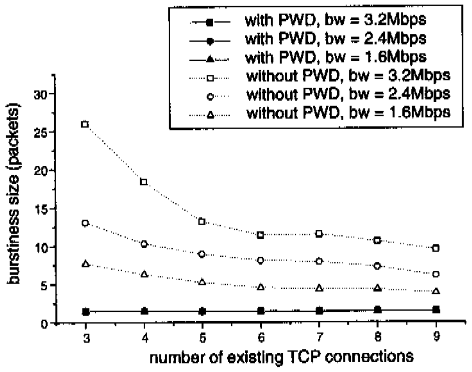
The difference between the algorithms with and without PWD is evident. NewReno with PWD enhances the transition dynamics by sending more new packets, resulting in the smaller burst size. It increases the utilization of the recovery period by sending more new packets and avoiding the packet burst after recovery. But since it has more aggressive properties, it may have a higher probability of packet drops during recovery. Throughout simulations, we observe that NewReno without PWD has slightly lower drop probabilities than NewReno with PWD, and that the delayed ACK policy and the increase in the number of existing TCP connections reduce the probability of packet drops during recovery. Although NewReno without PWD has lower probabilities of packet drops than NewReno with PWD, the differences are not significant. Rather, the difference was observed where the delayed ACK policy is used. The delayed ACK policy decreases the drop probability by about 20%.



(a) Without delayed ACK

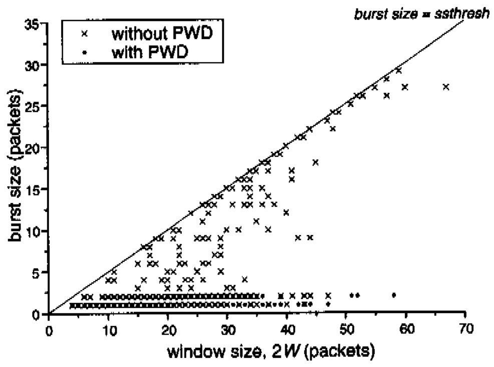


(a) Without delayed ACK



(b) With delayed ACK

Fig. 5 Average size of packet burstiness after recovery.



(b) With delayed ACK

Fig. 6 Packet burst size versus window size.

V. Conclusions

Since TCP NewReno algorithm with larger bottleneck bandwidth can result in a lengthy recovery period in its start-up transition, its transition dynamics becomes more important. When the sender detects a drop after the connection establishment, it may have too large window size due to the exponential window increase in Slow-Start. So it can experience multiple packet drops within a window and enter a lengthy recovery period because recovering all drops takes as many RTTs as the number of packet drops. NewReno's with and without PWD behave differentially in terms of the transition dynamics and show different properties. In this paper, we analyzed NewReno's with and without PWD focusing on new packets sent during Fast Recovery, and packet burstiness after Fast Recovery ends.

The simulation results bear out the analysis in the number of new packets sent and the packet burst size of NewReno with and without PWD. The average numbers of them showed the statistical feature of NewReno dynamics, and the details showed their correlations with the window size. Throughout simulations, we observed that NewReno with PWD generally performs better than NewReno without PWD. The PWD algorithm allows the sender to send an appropriate number of packets in an RTT and avoid packet burstiness. This property enhances the performance of NewReno during transition and allows the sender to send more packets.

Enhanced TCP recovery algorithm based on the simulation results is a future work. It should improve TCP performance without being excessively aggressive for fairness.

Reference

[1] V. Jacobson, "Congestion avoidance and control", *In Proceedings of the ACM SIGCOMM'88*, August 1988.

[2] M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", *RFC 2581*, April 1999.

[3] J. C. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", *In Proceedings of the ACM SIGCOMM'96*, 1996.

[4] K. Fall, S. Floyd, "Simulation based companions of Tahoe, Reno and SACK TCP", *Computer Communication Review*, July 1996.

[5] S. Floyd, T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", *RFC 2582*, April 1999.

[6] The UCB/LBNL/VINT Network Simulator (NS), *URL "http://www-mash.cs.berkeley.edu/ns"*.

[7] S. Floyd, V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, August 1993.

[8] C. Joo, S. Bahk, "Start-up Transition Behaviour of TCP NewReno", *IEE Electronics Letters*, Vol. 35, No. 21, 1999.

주 창 희(Changhee Joo)
1998년 2월 : 서울대학교 전기공학부 졸업
2000년 2월 : 서울대학교 전기공학부 석사
2000년 3월~현재 : 서울대학교 전기공학부 박사과정
<주관심 분야> 컴퓨터 네트웍

박 세 웅(Saewoong Bahk) 정회원
1984년 2월 : 서울대학교 전기공학과 졸업
1988년 2월 : 서울대학교 전기공학과 석사
1991년 : University of Pennsylvania 박사
<주관심 분야> 컴퓨터 네트웍