PAPER

Momentary Recovery Algorithm: A New Look at the Traditional Problem of TCP

Jae-Hyun HWANG[†a)], **See-Hwan YOO**[†], ***Nonmembers***, ***and*** **Chuck YOO**[†b)], ***Member***

SUMMARY Traditional TCP has a good congestion control strategy that adapts its sending rate in accordance with network congestion. In addition, a fast recovery algorithm can help TCP achieve better throughput by responding to temporary network congestion well. However, if multiple packet losses occur, the time to enter congestion avoidance phase would be delayed due to the long recovery time. Moreover, during the recovery phase, TCP freezes congestion window size until all lost packets are recovered, and this can make recovery time much longer resulting in performance degradation. To mitigate such recovery overhead, we propose *Momentary recovery* algorithm that recovers packet loss without an extra recovery phase. As other TCP and variants, our algorithm also halves the congestion window size when packet drop is detected, but it performs congestion avoidance phase immediately as if loss recovery is completed. For lost packets, TCP sender transmits them along with normal packets as long as congestion window permits rather than performs fast retransmission. In this manner, we can eliminate recovery overhead efficiently and reach steady state *momentarily* after network congestion. Finally, we provide a simulation based study on TCP recovery behaviors and confirm that our *Momentary recovery* algorithm always shows better performance compared with NewReno, SACK, and FACK.

key words: *transmission control protocol, loss recovery algorithm, congestion control, recovery overhead*

1. Introduction

TCP (Transmission Control Protocol) [1] is a popular end-to-end transport protocol for reliable communication. To provide reliability, TCP has some detection and recovery algorithms for lost packets. Besides reliability, it performs congestion control that adapts the sending rate against network congestion. These congestion control and recovery algorithms are associated closely with each other. Assuming that packet loss implies network congestion, TCP has two end-to-end loss detection methods. One is to set a retransmission timer, and the other is to receive three duplicate ACKs, which triggers fast retransmit and fast recovery [2]. By the latter method TCP Reno can recover from temporary network congestion fast without waiting for a RTO (Retransmission TimeOut). This is however vulnerable to multiple packet losses while efficient to single loss since Reno intends to recover only the first lost packet. To overcome such inefficiency, TCP NewReno lasts the fast recovery phase until all lost packets are acknowledged by partial ACK [3]. It can retransmit a lost packet per one RTT (Round-Trip Time), but still spends a long time to recover burst packet losses. The Selective-ACK (SACK) option contains acknowledged sequence blocks in a TCP header so that the sender can retransmit multiple lost packets at a time [4], [5].

These recovery mechanisms have been widely used with most TCP variants, and they are strictly coupled with the congestion control algorithm. That is, when packet loss is detected by three duplicate acknowledgements (ACKs), TCP reduces the current window size for congestion control and the window is not re-opened until completion of the fast recovery phase*. Up to our knowledge, all existing recovery mechanisms adopt this rule, "*congestion window can be opened only after the recovery is finished.*" However, we have observed that the current TCP suffers from multiple losses since the window size is frozen unnecessarily during the fast recovery phase. Why doesn't TCP perform additive increase upon reducing its window size? We could not find any strong reason that the congestion avoidance phase should be delayed during the recovery phase, and even the well-known performance model of TCP Reno does not consider the fast recovery overhead [6] (i.e. the TCP throughput model excludes the recovery phase from its window evolution.) Our insight is that early TCP designers did not assume multiple packet losses, and so far, it has been natural that the congestion avoidance restarts after all lost packets recover. From this insight, we propose *Momentary recovery* algorithm that recovers packet loss without the fast recovery phase. This can be possible ideally by decoupling the loss recovery from the congestion control. When packet loss is detected, our algorithm shrinks the congestion window size by $1/2$ as standard TCP does, but it performs additive increase immediately instead of fast recovery assuming that all lost packets are already acknowledged up to the highest SACKed sequence. And then, the lost packets and new data packets can be transmitted as long as the congestion window permits. In this manner, we can eliminate recovery overhead efficiently and reach steady state *momentarily* after network congestion. Moreover, our algorithm is easy to be deployed since it requires a slight modification only at the sender side as long as SACK option is used.

The rest of this paper is organized as follows. Section 2 briefly reviews the traditional recovery algorithms.

Manuscript received April 22, 2009.
Manuscript revised August 5, 2009.

[†]The authors are with the Department of Computer Science and Engineering, Korea University, Seoul, Korea.
a) E-mail: jhhwang@os.korea.ac.kr
b) E-mail: hxy@os.korea.ac.kr

DOI: 10.1587/transcom.E92.B.3765

*In this paper, we separate temporary increments by duplicate ACKs from opening the congestion window size(*cwnd*) during fast recovery since it does not aim to perform additive increase.

In Sect. 3, we explain an assumption and recovery process of our algorithm in details. Section 4 performs simulation experiments and evaluates our algorithm comparing with existing ones. We also examine the friendliness toward NewReno+SACK flows. In Sect. 5, we discuss the performance of recovery algorithms on a high bandwidth and delay product environment. Finally, this paper concludes in Sect. 6.

2. Traditional Recovery Algorithms

In this section, we review existing loss recovery algorithms to compare with our *Momentary recovery* algorithm. The most widely used recovery algorithm is NewReno, which adopts a fast recovery algorithm from TCP Reno. Whenever receiving three duplicate ACKs, TCP Reno decreases the congestion window size and enters the recovery phase. It does not wait until the retransmission timer expires; thus it is called fast retransmit. In addition, it performs congestion avoidance from half the previous congestion window instead of beginning slow-start; thus, it is called fast recovery. However, TCP Reno is fragile when multiple packets are lost in the same congestion window. Since Reno escapes from recovery phase on receiving the first partial ACK, the sender may wait until the retransmission timer expires in the case of multiple losses, which requires very much time (at least one second.)

TCP NewReno [3] overcomes the suffering from multiple packet losses. Unlike Reno, it remains in the fast recovery phase until all the lost packets are successfully acknowledged. The sender keeps the highest sent sequence at the beginning of recovery and performs retransmission for each partial ACK instead of incurring the retransmission timeout. NewReno also enters congestion avoidance after the recovery phase to adapt the sending rate as AIMD(Additive Increase, Multiplicative Decrease) algorithm. This recovery mechanism however requires still a long time since it can recover only one lost packet per RTT.

SACK [5] uses a more efficient recovery algorithm by changing the acknowledgement mechanism. It utilizes TCP option fields effectively. The receiver gives a feedback to the sender in order to inform its acknowledged state. It sends out a duplicate ACK with optional SACK information, which contains in-order acknowledged sequence blocks. When three duplicate ACKs are received, the sender calculates the *pipe*, the estimate of outstanding packets in the network, and retransmits multiple lost packets within the pipe size.

FACK (Forward-ACK) [7] is a variant of SACK. The holes, namely the unacknowledged packets between SACK blocks, generally can be treated in two ways. The first approach is to consider all the holes to be outstanding in the network according to SACK standard promoted by IETF. On the other hand, FACK considers the holes as lost packets and can retransmit them more aggressively. The FACK approach hence achieves better performance than the first approach in most cases. However, it is hard to decide the unacknowledged holes to be lost completely since packets can be reordered in the network [8], [9]. For this reason, Linux TCP enables FACK by default but disables it only when the packet reordering is detected [10].

There are another recovery algorithms that are not based on the traditional approach. Yang et al. proposed a new retransmission strategy, called *Bulk Repeat* [11], for TCP in high error situations. Bulk Repeat has three modifications: 1) retransmit all unacknowledged packets when packet loss is detected, 2) use a fixed timeout value instead of exponential backoff, and 3) keep *cwnd* being fixed in spite of losses. In addition, Bulk Repeat adopts a loss discrimination algorithm (LDA) in order to distinguish between congestion and wireless losses since the modifications are only for wireless links. However, it highly depends on the accuracy in LDA. That is, if the loss differentiation is failed, the network safety can be broken seriously since Bulk Repeat has very aggressive recovery properties.

Ladas et al. proposed a retransmission prioritization algorithm; retransmitting packets are prioritized over originally transmitting ones. However, it requires a signalling channel for status information exchange and is applied only to slow and lossy networks [12]. Wang and Shin proposed *Robust Recovery* [13] to make a TCP flow more robust to bursty packet losses. Robust Recovery keeps track of the number of duplicate ACKs to detect any further packet loss without SACK, but it has a weak point that its self-clocking can be lost by loss of ACKs since it relies on returning ACKs to inject new packets into the network. Finally, Kothari et al. proposed an adaptive flow control algorithm [14], which is an extension to *Delayed Fast Recovery* [15]. Both of two algorithms assume that the first loss is a random loss and delays the fast recovery until the second loss occurs. However, their assumption is not true — recall that the early fast retransmission and recovery algorithms are designed for single packet loss. Thus, such delayed recovery can infringe Reno's bandwidth.

3. Momentary Recovery Algorithm

As mentioned in Sect. 1, this paper claims that the responsibility of TCP for network congestion is to reduce the congestion window size, not wait for recovery completion. Fast recovery algorithm in TCP Reno was originally designed to recover single packet loss, whose recovery overhead might be trivial. However, in the case of multiple packet losses, it can cost several RTTs that result in performance degradation. Our algorithm starts from an intuition that if TCP sender does not need to retransmit lost packets, it will be able to enter the avoidance phase immediately after congestion control. We achieve this goal by separating recovery process from congestion control, and this section describes detailed *Momentary recovery* algorithm and its assumption.

3.1 Assumption

To eliminate the fast recovery overhead as described in

Sect. 1, our algorithm is based on a premise; after receiving three duplicate ACKs, holes between SACK blocks are regarded as lost packets. Most TCP variants derived from Reno typically triggers the fast recovery when three duplicate ACKs are received whereas FACK performs its recovery process when the difference between the last ACK and the forward ACK[†] sequence is more than three. The loss recovery in FACK algorithm therefore can be triggered by only one SACK. This means that FACK has higher probability for false fast retransmit than Reno if packet reordering has a place in the network, and this can lead to degradation in performance due to unnecessary window reduction. Nevertheless, without packet reordering, TCP with FACK algorithm can recover efficiently from multiple packet losses. In our algorithm, to mitigate premature decision in FACK algorithm, the recovery phase is triggered by three duplicate ACKs like other TCP variants, but upon triggering the loss recovery the holes can be retransmitted as FACK algorithm does. We also believe that this assumption does not cause additional congestion since its aggressiveness is comparable to the case of FACK.

3.2 Recovery from Packet Losses Momentarily

The main idea of *Momentary recovery* algorithm is to conceptually remove traditional fast recovery phase. In other words, it halves the congestion window after detecting packet loss and enters the congestion avoidance phase immediately without extra recovery phase. This can be realized by the assumption described above and the forward ACK. The reason why we use the forward ACK is that the unacknowledged holes are the major cause to preclude the window from sliding to the right. Our insight is that restarting the congestion avoidance from forward ACK sequence will make the sliding window run on wheel.

In addition, our assumption is required to support such operation of the sliding window. For instance, let m be the number of lost packets when the window size is n. If the holes between SACK blocks are losses after receiving $n-m$ duplicate ACKs ($n-m \geq 3$), we will be able to send packets as much as the window size at once since actually there is no outstanding packet in the network at this time.

Algorithm 1 presents the packet receiving function of *Momentary recovery* by a simple pseudo code. In the function, *ack_no* variable indicates the current ACK sequence, and normal/recovery mode can be switched by *recovery* variable. The *moment_ack* variable points the left side of sliding window; it is equal to the last ACK sequence (*ack_no*) in normal mode and the last SACKed sequence (forward ACK) in recovery mode. Figure 1 shows the operation of sliding window in recovery mode. This is similar with FACK algorithm, but different in keeping additive increase consistently. Furthermore, retransmitted packets are also applied into congestion avoidance algorithm treating them as normal packets. We count the number of retransmitted packets by *retransmit* variable so that the total of all outstanding packets cannot exceed the congestion window size.

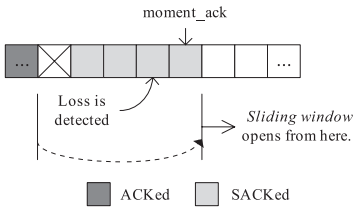


Fig. 1 Operation of sliding window in the recovery mode.

Algorithm 1 Packet receiving function

```
if recovery == false then {normal mode}
   if ack_no > moment_ack then
      moment_ack = ack_no
      update congestion window
   else if packet loss is detected then
      recovery = true
      halve congestion window
   end if
else {recovery mode}
   update congestion window
   if holes between SACK blocks are covered? then
      retransmit = retransmit−number of holes covered
   end if
   if ack_no covers all loss packets? then
      recovery = false
      moment_ack = ack_no
   else
      moment_ack = forward ACK number
   end if
end if
```

Algorithm 2 Packet sending function

```
while (next_seq ≤ moment_ack + cwnd − retransmit) do
   if no retransmission is needed then
      seqno = next_seq
      next_seq = next_seq + 1
   else
      seqno = next retransmitting packet
      retransmit = retransmit + 1
   end if
   send a seqno packet
end while
```

Algorithm 2 presents the packet sending function. Since we separate loss recovery from congestion control, the receiving function does not perform fast retransmit, but the sending function checks whether or not retransmission is needed before sending a normal packet. If it is, *retransmit* variable is added by one to limit the number of outstanding normal packets to *cwnd - retransmit.*

Figure 2 shows an example for recovery process of *Momentary recovery* algorithm. In the first round, the sender transmits 1-6 packets and sequence 1 is dropped. By three duplicate ACKs, packet loss is detected and the congestion window is halved in round 2. At this time, our recovery algorithm opens the sliding window from 7 to 9, but the sender

[†]We use the term *FACK* as a recovery algorithm and *forward ACK* as the highest SACKed sequence as defined in [7].

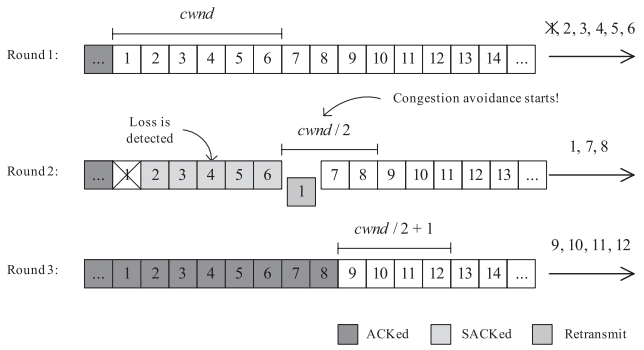


Fig. 2 Example of congestion control and loss recovery with *Momentary recovery* algorithm.

transmits 1, 7, and 8 packets to retransmit the lost packet. Finally, when the ACK for sequence 8 that recovers the lost packet is returned, the sender can transmit $cwnd/2+1$ packets from 9. This process shows that our algorithm performs the congestion avoidance immediately after packet loss regardless of the loss recovery.

4. Simulation Results

In this section, we perform simulation experiments using NS-2 (version 2.31) simulator [16] to show the effectiveness of our algorithm. We implement *Momentary recovery* into NS-2 and compare to NewReno, NewReno+SACK, and FACK. For the existing variants, implementation of NS-2 and its default settings are used, but we use Slow-but-Steady algorithm rather than Impatient algorithm for the loss recovery of NewReno since Slow-but-Steady shows better performance in our simulations[†]. This resets the retransmission timer for every partial ACK so that RTO would not be expired until the fast recovery is completed [17]. The packet size is fixed by 1000 bytes, and delayed ACK is not used. Lastly, the congestion window size is not limited by flow control in all simulations.

4.1 Recovery Efficiency

The best way to validate a recovery algorithm is to compare its behavior with the existing algorithms under same packet loss situations. Since such experiment requires no complex environment, all experiments in this subsection are performed over a simple dumpbell network topology that has single bottleneck link. The round-trip propagation delay is about 200 ms, and the bottleneck capacity is 1 Mbps. We set router buffers to large enough to overflow no packets, but some specified packets by intention for fair comparison between recovery algorithms in this subsection.

Figure 3 shows sending behaviors of recovery algorithms by sequence number for multiple losses. The black boxes in Fig. 3 indicate sequences of transmitting packets, and receiving ACKs are marked by the small red spots. The initial slow-start threshold (*ssthresh*) value is 10 in order to enter the steady state quickly, and the sender enters congestion avoidance phase before packet loss in this simulation. We enforce to drop the sequence from 24 to 30 for all cases consistently, and the congestion window size is about 11.4 just before packet loss.

In Fig. 3(a), NewReno transmits additional packets for two duplicate ACKs at the epoch after the losses occur, and performs fast retransmit for the third duplicate ACK reducing its congestion window to 5.7. After then, it retransmits a lost packet for each partial ACK until all lost packets are recovered. In the fast recovery phase, it can start to send new data packets from the fourth retransmitted packet as a new window opens. NewReno basically is able to retransmit lost packets steadily, but takes a long time to enter the congestion avoidance phase when multiple losses occur since its average recovery time is $(the\ number\ of\ lost\ packets) \times RTT$. In other words, slow-closing of the left side in the sliding window makes the sending rate depressed.

The recovery time can be faster with SACK option since SACK blocks inform the sender which packet is unacknowledged at the receiver side. However, SACK does not retransmit the unacknowledged packets directly since the halved window size limits total outstanding packets by the pipe variable. In Fig. 3(b), the sender retransmits two lost packets for each partial ACK according to [8], so it takes about four RTTs to finish the recovery phase.

FACK retransmits lost packets more aggressively leading to be faster than SACK in recovery time. However, as shown in Fig. 3(c), FACK does not send an additional packet for a duplicate ACK before detecting packet loss. This is because FACK triggers recovery mode by the number of holes between SACK blocks, hence it is difficult in detecting packet loss when the congestion window is small. For instance, if the window size is 3 and the first packet is dropped, the existing method can trigger fast recovery by sending additional packets, but FACK would result in retransmission timeout in this case.

Lastly, *Momentary recovery* sends two additional packets by duplicate ACKs and retransmits lost packets as much as the halved congestion window size (2 + 3) after packet loss as shown in Fig. 3(d). In the next RTT, we can find that six (5 + 1) packets are sent by congestion avoidance algorithm. Our recovery algorithm also shows that the time for receiving the full ACK is comparably fast with FACK since it treats the holes of SACK as lost packets after receiving three duplicate ACKs.

Figure 4 shows the variation of the congestion window for the case of Fig. 3, and we could know straightforwardly that our algorithm performs additive increase immediately after window reduction. In the other hand, other existing algorithms enter the congestion avoidance phase only when the ACK covers all lost packets is received compared with Fig. 3. The difference of restarting time for the congestion avoidance between *Momentary recovery* and existing algorithms is about 6, 3, and 1 RTT with NewReno,

[†]NewReno uses Impatient algorithm by default in NS-2, which resets the retransmission timer only for the first partial ACK.

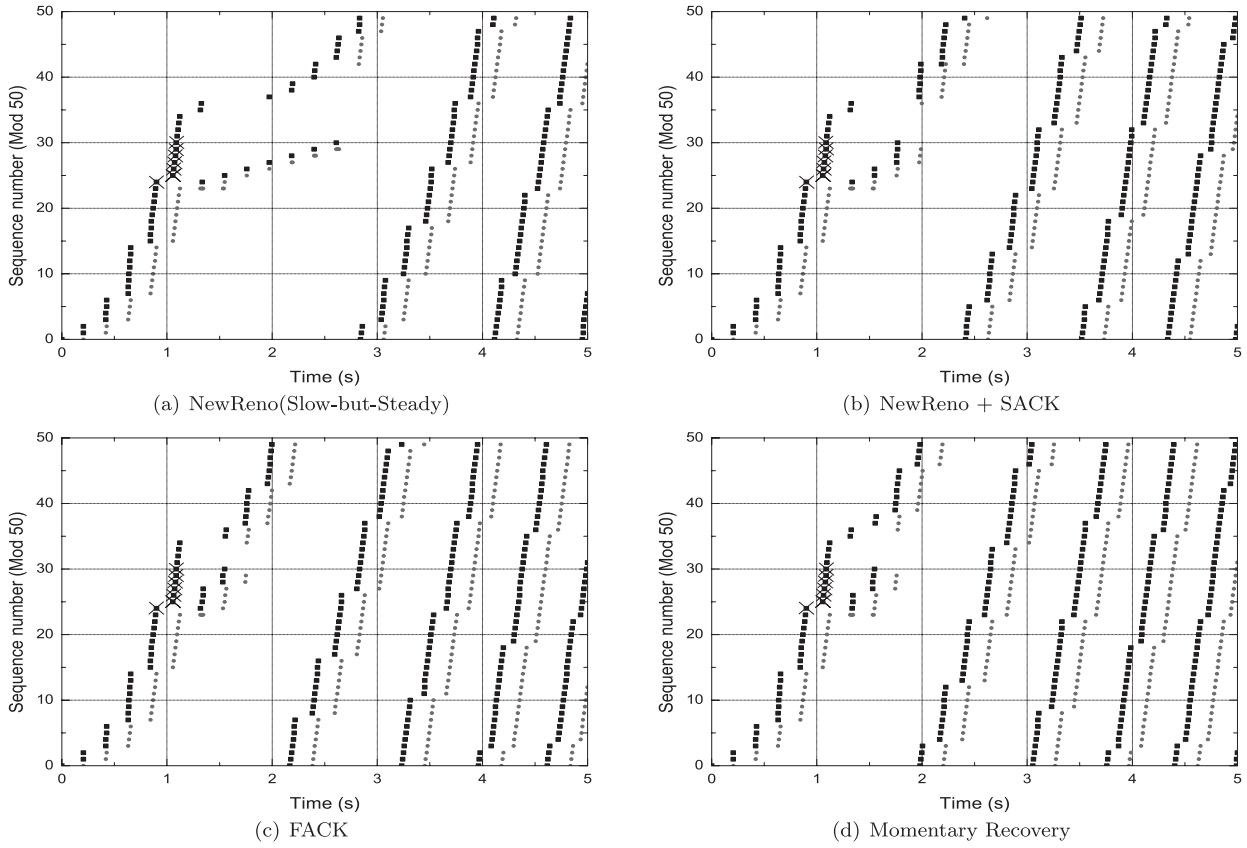


Fig. 3 Comparison between loss recovery behaviors.

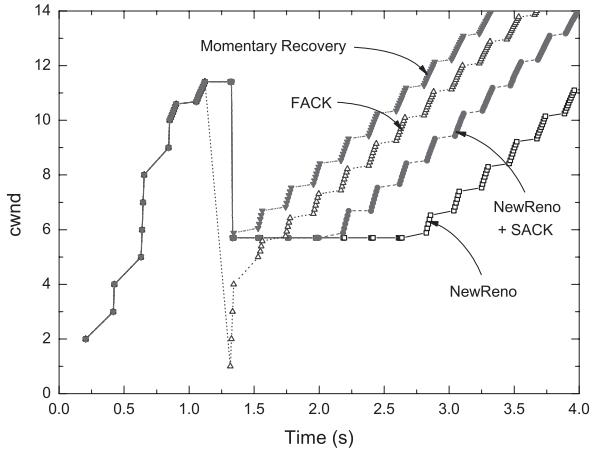


Fig. 4 Variation in congestion window size.

NewReno+SACK, and FACK, respectively in this simulation.

Figure 5 shows sequence numbers varying with the number of lost packets from 1 to 8 — consistent pattern in all graphs. Figures 5(a)–(c) generate 1 to 3 packet losses, which are less than half of the window size while Figs. 5(d)–(f) are the case of burst multiple losses above half of the window size. As shown in Fig. 5, *Momentary recovery* algorithm always shows the fastest increment, and this is more notable as the number of lost packets is increased. Since it is delayed to enter the congestion avoidance phase as the recovery time becomes longer, the effectiveness of the proposed algorithm can be enlarged relatively. In addition, the recovery time generally is in proportion to RTT, so the throughput difference ratio between our algorithm and existing ones would be much larger in long RTT environments. As a result, we can confirm that *Momentary recovery* leads TCP to achieve better performance by improving the inefficiency of existing recovery algorithms.

4.2 Network Safety

We validate network safety in terms of the friendliness toward NewReno with SACK (standard) flows in this subsection. The definition of TCP-friendliness is: a flow is considered TCP-friendly when it does not reduce the long-term throughput of any co-existent TCP flow more than another TCP flow on the same path under the same network conditions [18]. This is a very important factor when a new TCP variant is designed. In fact, *Momentary recovery* algorithm is not a congestion control algorithm, but we also have to investigate whether or not a *Momentary recovery* flow infringes other TCP flows since it is true that TCP can gain additional throughput improvement by our algorithm. We also perform some simulations for the fairness between Momentary recovery embedded TCP flows, but do not present

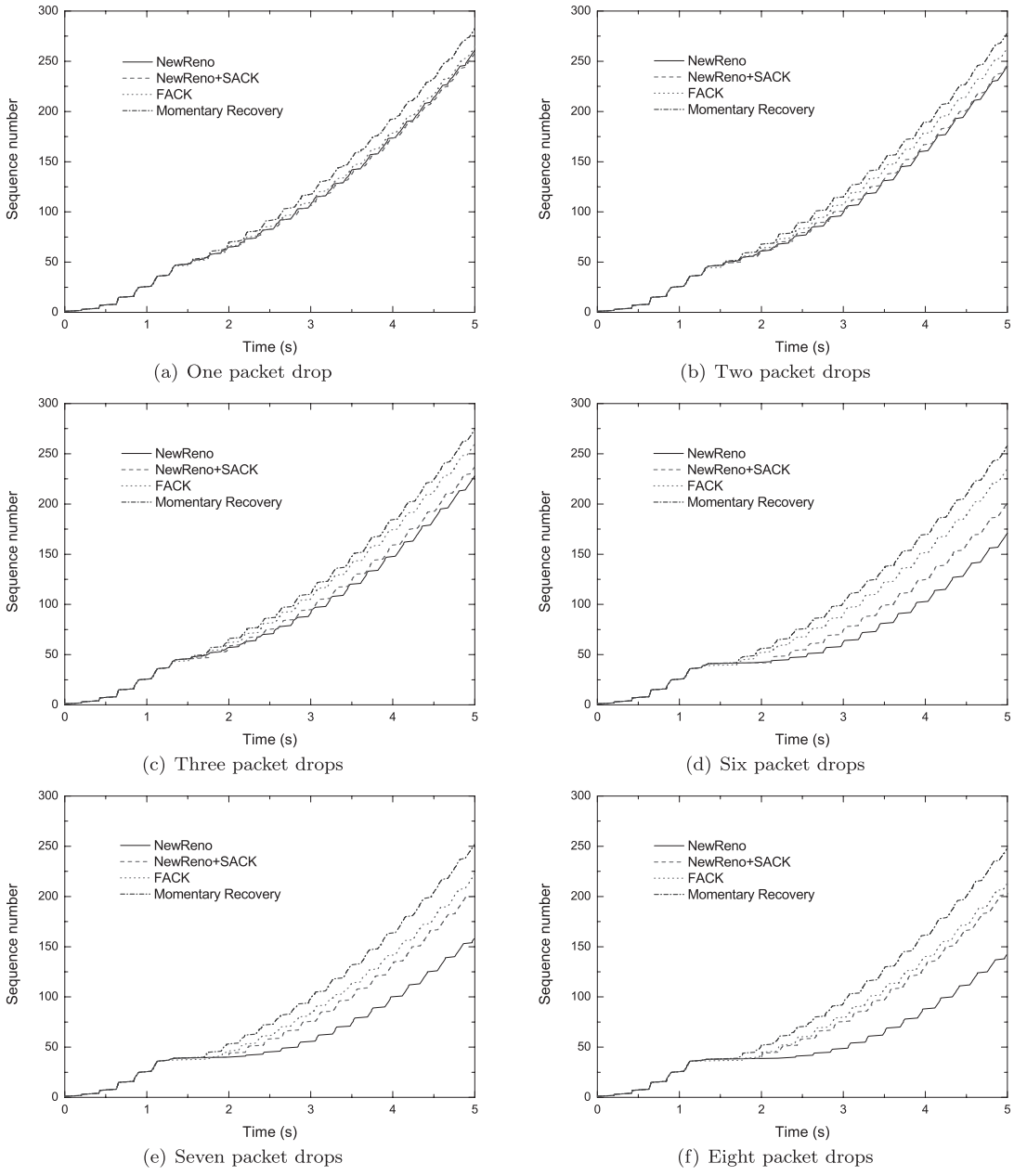


Fig. 5 Sequence graphs for multiple packet losses.

the results since they showed perfect fairness for bandwidth sharing according to Jain's fairness index function [19].

Figure 6 depicts our simulation topology for the friendliness comparison. The buffer size of bottleneck router is ten thousand packets, and there is no random or artificial loss. We run a thousand TCP connections for 500 seconds varying the number of flows for each TCP scheme. Table 1 presents average throughputs (kbps) between same TCP schemes. In Table 1, we can observe that the Momentary flows occupies more bandwidth portion by fast restoring to the steady state. However, we note that the friendliness is not injured seriously since the standard flows keep their own portions. The declines of standard flows in throughput are only within 3% compared with the best case (i.e. 1000 standard connections.) These results shows that our algorithm does not harm in keeping the friendliness to the NewReno+SACK flow.

4.3 Overall Performance

Finally, to evaluate overall performance, we use a more complex topology shown in Fig. 7, which contains reverse and cross background TCP-SACK traffics. The bottleneck links that connect all intermediate routers between sender (S) and receiver (R) have 5 Mbps bandwidth and 20 ms propagation delay, and the others are 10 Mbps and 10 ms, respectively. The router buffers are set to 40 packets and random loss is given at R2-R3 link (forward direction), and R5-

R4 link (reverse direction). We measure the goodput and the number of retransmitted packets for NewReno, SACK, FACK, and Momentary recovery TCP.

Table 2 lists the results, which are averaged by 100 iterations. From this, it can be seen that the Momentary recovery embedded TCP shows the highest goodput for all cases. When there is no random loss, our algorithm shows better goodput than SACK and FACK as much as 10% and 20% respectively. We can also observe that the number of retransmitted packets of Momentary recovery TCP is comparable with other TCP schemes. This result supports that it does not make the congestion condition worse by fast entering the congestion avoidance phase since packet retransmission implies the network congestion in the absence of random loss. In the cases of presence of random loss, its performance gain gradually increases as the loss rate increases. Especially, it achieves better goodput up to about 27%, 28%, and 38% at loss rate of 3% compared with NewReno, SACK, and FACK respectively. We note that the number of retransmitted packets of our TCP scheme is larger than that of NewReno and FACK as the loss rate increases. This is because Momentary recovery TCP has higher goodput, and this means more packets to be sent, which in turn causes more packet losses. In fact, in the comparison with FACK, the increase ratio (33%) of retransmitted packets is less than the increase rate of goodput (38%) at loss rate of 3%.

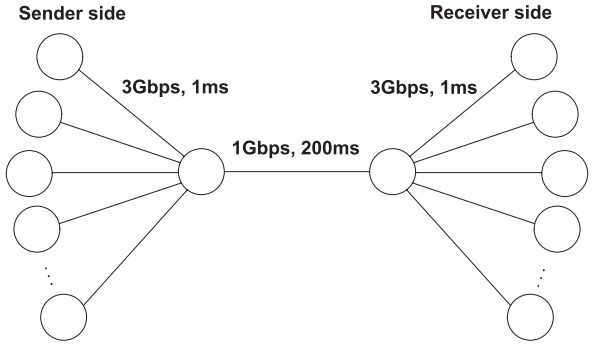


Fig. 6 Simulation topology for friendliness comparison: number of source and destination pairs for each scheme are given in the result table.

Table 1 Friendliness comparison (kbps).

Standard flows	Moment. flows	Standard Avg. throughput	Moment. Avg. throughput
0	1000	N/A	838.59
300	700	812.07	848.08
500	500	799.11	863.35
700	300	820.45	856.72
1000	0	822.33	N/A

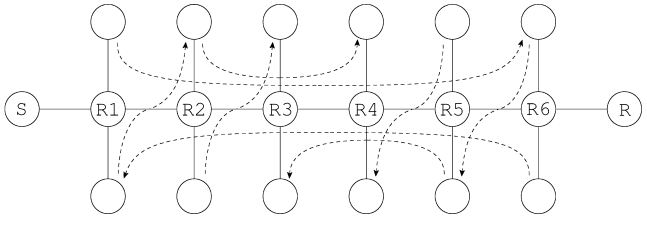


Fig. 7 Simulation topology for performance evaluation.

5. Discussion

As shown in Sect. 4, our recovery algorithm is very efficient regardless of existence of random loss. In this section, we would like to consider a high bandwidth and delay product (BDP) environment. In such networks, TCP congestion window size can reach thousands to ten-thousands, and this may result in hundreds of consecutive packet losses. To investigate the situation, we perform some simulations that run each recovery algorithm for 100 seconds over the topology in Fig. 6. We set *ssthresh* to 1000 since TCP requires a very long time to reach the maximum sending rate by its AIMD algorithm.

Through the simulations, we observe that the congestion window size reaches about 1000 packets and 190 consecutive packets in a window are dropped when congestion occurs. Table 3 shows the goodputs and the number of packet losses. As we expected, NewReno shows poor goodput, and its recovery time was about 76 seconds, which

Table 3 Performance comparison over a high bandwidth and delay product environment.

Recovery scheme	Avg. throughput (kbps)	Number of packet losses
NewReno	1852.03	224
SACK	4571.09	368
FACK	11575.16	190
Moment.	11593.67	190

Table 2 Overall performance: goodput (kbps) and number of retransmitted packets.

Loss rate	NewReno		SACK		FACK		Moment.	
(%)	goodput	rexmits	goodput	rexmits	goodput	rexmits	goodput	rexmits
0.0	304.76	172	311.65	262	339.98	170	375.64	166
0.1	343.79	179	355.71	255	367.35	157	397.08	174
0.5	286.67	198	287.84	264	299.09	186	327.76	203
1.0	209.06	248	209.73	316	212.83	229	246.86	261
3.0	105.49	353	104.76	396	96.75	286	133.80	380

indicate 190 RTTs. SACK recovers from multiple losses in a shorter time, but after the recovery phase, it causes an extra congestion due to a sudden increase in the sliding window. FACK and our algorithm show high goodput without additional packet loss, and we could know that FACK is also efficient when the loss ratio in a window is so low (i.e. 190/1000.) Finally, we believe that if a high-speed TCP variant such as CUBIC [20] that adopts a more aggressive congestion control algorithm uses our algorithm, it can achieve better performance improvement since the congestion control of our algorithm can be performed separately from the loss recovery.

6. Conclusion

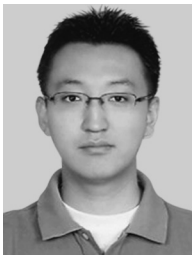
In this paper, we propose a new recovery algorithm, named *Momentary recovery* that improves the existing fast recovery algorithm efficiently. Traditionally, TCP triggers fast retransmit and fast recovery by duplicate ACKs, and enters congestion avoidance phase after loss recovery is finished. During the recovery phase, TCP freezes its *cwnd* value, which results in performance degradation as the recovery takes longer. To alleviate such suffering from freezing the window size, our *Momentary recovery* algorithm removes the recovery phase conceptually by decoupling loss recovery from congestion control. It performs additive increase during the momentary recovery mode and slides the window to the right by SACK information. Through simulation experiments, we confirm that our algorithm shows optimal recovery behavior compared with NewReno, SACK, and FACK, and also preserves the fairness and friendliness characteristics of TCP. Finally, our algorithm can be embedded in any TCP variant and deployed easily since it requires a slight modification only at the sender side when the receiver side uses SACK option.

Acknowledgment

This work was supported by the Second Brain Korea 21 Project, and partially supported by the Seoul Research and Business Development Program, Smart(Ubiquitous) City Consortium and Seoul Grid Center.

References

[1] W.R. Stevens, TCP/IP illustrated (vol.1): The protocols, Addison-Wesley Longman Publishing, 1993.
[2] W. Stevens, "TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," RFC 2001, IETF, 1997.
[3] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno modification to TCP's fast recovery algorithm," RFC 3782, IETF, 2004.
[4] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno and SACK TCP," SIGCOMM Computer Communication Review, vol.26, no.3, pp.5–21, 1996.
[5] E. Blanton, M. Allman, K. Fall, and L. Wang, "A conservative selective acknowledgment (SACK)-based loss recovery algorithm for TCP," RFC 3517, IETF, 2003.
[6] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Reno performance: A simple model and its empirical validation," IEEE/ACM Trans. Netw., vol.8, no.2, pp.133–145, 2000.
[7] M. Mathis and J. Mahdavi, "Forward acknowledgement: Refining TCP congestion control," Proc. ACM SIGCOMM, 1996.
[8] A. Gurtov and R. Ludwig, "Responding to spurious timeouts in TCP," Proc. IEEE INFOCOM, 2003.
[9] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A reordering-robust TCP with DSACK," Proc. 11th IEEE International Conference on Network Protocols, 2003.
[10] P. Sarolahti and A. Kuznetsov, "Congestion control in Linux TCP," Proc. FREENIX Track: 2002 USENIX Annual Technical Conference, 2002.
[11] G. Yang, R. Wang, M. Sanadidi, and M. Gerla, "TCPW with bulk repeat in next generation wireless networks," Proc. IEEE International Conference on Communications (ICC), 2003.
[12] C. Ladas, R. Edwards, M. Mahdavi, and G. Manson, "TCP retransmission prioritisation for rapid recovery in slow and lossy networks," Proc. 5th European Personal Mobile Communications Conference, 2003.
[13] H. Wang and K. Shin, "Robust TCP congestion recovery," J. High Speed Networks, vol.13, no.2, pp.103–121, 2004.
[14] N. Kothari, B. Gambhava, and K. Dasgupta, "Adaptive flow control: An extension to delayed fast recovery," Proc. 15th International Conference on Advanced Computing and Communications, 2007.
[15] N. Kothari and K. Dasgupta, "Performance enhancement of SACK TCP protocol for wireless network by delaying fast recovery," Proc. IFIP International Conference on Wireless and Optical Communications Networks, 2006.
[16] The Network Simulator NS-2, http://www.isi.edu/nsnam/ns/
[17] N. Parvez, A. Mahanti, and C. Williamson, "TCP NewReno: Slow-but-steady or impatient?," Proc. IEEE International Conference on Communications (ICC), 2006.
[18] J. Widmer, R. Denda, and M. Mauve, "A survey on TCP-friendly congestion control," IEEE Netw., vol.15, no.3, pp.28–37, 2001.
[19] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," DEC, Res. Rep.TR-301, 1984.
[20] I. Rhee and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," Proc. 3rd International Workshop on Protocols for FAST Long-Distance Networks (PFLDnet), 2005.



Jae-Hyun Hwang received the B.S. degree in computer science from Catholic University, Seoul, Korea, in 2003 and the M.S. degree in computer science from Korea University, Seoul, Korea, in 2005. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interests are in network protocol design and kernel networking.

See-Hwan Yoo received the B.S. and M.S. degrees in computer science from Korea University, Seoul, Korea, in 2002 and 2004, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interests are in the system virtualization.

Chuck Yoo received the B.S. degree in electronic engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. He worked as a researcher in Sun Microsystems Laboratory, from 1990 to 1995. He is now a Professor in Department of Computer Science and Engineering, Korea University, Seoul, Korea. His research interests include high performance networks, multimedia streaming, and operating systems. He served as a member of the organizing committee for NOSSDAV 2001.