Kang et al. EURASIP Journal on Wireless Communications and Networking 2012, **2012**:149
http://jwcn.eurasipjournals.com/content/2012/1/149

EURASIP Journal on Wireless Communications and Networking
a SpringerOpen Journal

RESEARCH **Open Access**

Implementation and evaluation of a new TCP loss recovery architecture

Moonsoo Kang[1], Hosung Park[2] and Jeonghoon Mo[2*]

Abstract

In this article, we propose a novel loss recovery algorithm of transmission control protocol (TCP) using packet transmission order, which shows a steady loss recovery ability even though packet loss rate increases. This leads to a significant throughput increase of TCP with heavy packet loss. We have verified the performance increase of the new TCP under various environments such as a wireless network, and multimedia transmission through simulation. Moreover, we implemented the proposed idea in Linux and conducted some experiments in a real environment. Even though the experiment results did not perfectly agree with the simulation results, we obtained a similar throughput increase to that of the simulation.

Keywords: TCP, enhanced loss recovery, wireless TCP

1 Introduction

The transmission control protocol (TCP) has been very successfully used within the Internet, and it is being implemented and used in various operating systems such as Windows, Linux, and Mac OS. The congestion control and loss recovery algorithm of this protocol have been improved over the last two decades. In the late 1980s, Tahoe and Reno developed the first generation of TCPs with new congestion control features to address losses more efficiently [1,2].

However, the limitations with bursty losses led to the development of SACK and FACK protocol, which adopt a selective ACK rather than a cumulative ACK scheme [3,4].

As wireless networks became more popular, the use of a wireless lossy channel presents new issues to TCP engineers. Considerable efforts have been made to improve TCP efficiency over wireless channels [5-7]. Some researchers attempted to modify congestion control in order to retain the congestion window during wireless loss [8-10]. Others have differentiated congestion losses and wireless channel losses [11-13]. However, the results of these studies were not very promising because the developed methods require help from other network elements such as base stations or routers, and the proposed differentiation algorithms did not perform well. Other TCP variants such as TCP-RR [14], TCP-PR [15], and TCP-DCR [16] address the issue of packet reordering or persistent congestion.

Even with numerous proposals, limited attention has been paid to the loss recovery algorithm or architecture of TCP. Most work has focused on congestion control rather than loss recovery. Even SACK or FACK implementations, though they enhance loss recovery, maintain the single linked list architecture. We previously proposed a new TCP architecture based on two lists in order to improve the TCP loss performance [17,18]. We implemented the new idea in a linux setting and performed an evaluation of the method.

We verified the following statements through this article.

- The loss recovery ability of the current loss recovery algorithms varies with the number of packet losses in a window, leading to frequent RTO expiration and lower throughput. We show the weaknesses of the loss recovery algorithms in detail with the simulation results.

- The proposed new loss recovery algorithm consistently recovers lost packets at a higher packet loss rate. We also explained the reason to use packet transmission order to address the aforementioned problem of the current loss recovery algorithms.

* Correspondence: j.mo@yonsei.ac.kr
[2]Department of Information and Industrial Engineering, Yonsei University, Seoul, Republic of Korea
Full list of author information is available at the end of the article

- We implemented the proposed idea in a Linux and tested the implemented code in a real environment. Even though the experiment results did not perfectly agree with the simulation results, we obtained a similar throughput to that of the simulation.

- From the real experiment, we found that the congestion controls of wireless TCP did not function as proposed. However, even though wireless TCP shows poor throughput, a strong loss recovery algorithm can increase the performance of wireless TCP.

The rest of of article is organized as follows. Section 2 proposes new loss recovery architecture and algorithm for wireless TCP. It analyzes the limitations of current loss recovery algorithms and shows why packet transmission order should be used to improve the loss recovery ability. Section 3 validates the proposed idea with various simulation results. Section 4 describes the Linux implementation of the proposed idea and Section 5 shows experiment results. Section 6 finalizes this paper with conclusions and future research direction.

2 New architecture and loss recovery algorithm

Though the architecture and the algorithm were presented in our conference papers [17,18], we summarize them here for completeness. We named our proposal LE, an abbreviation of *L*oss r*E*silience. The main design goal of LE is to achieve loss recovery ability resilient to ACK starvation. To realize the goal, TCP needs to maintain the correct number of pending packets in networks even *during loss recovery*.

2.1 Two lists structure

We propose a new loss recovery algorithm based on a new data structure–two packet transmission order lists. The important difference between our proposal and the existing algorithms is the basic data structure. The data structure of the previous algorithms is sequential number based packet list which has difficulty remembering packet transmission order. However, our proposal is able to reflect the packet transmission order.

As shown in Figure 1b, LE manages two lists, WAITLIST and RTXLIST. When a packet is newly sent or resent, it is inserted at the end of WAITLIST to record the transmission order. Therefore, the list naturally represents all of the currently outstanding packets. When a packet is determined to be lost, the packet is transferred to the end of RTXLIST. Each entry in the lists contains three variables *dupCnt*, *timeStamp*, and *seqNum*.

2.2 New loss recovery algorithms

2.2.1 Per packet acking process

Whenever an ACK, *including time stamp and SACK options*, arrives, the *timeStamp* in the packet is compared

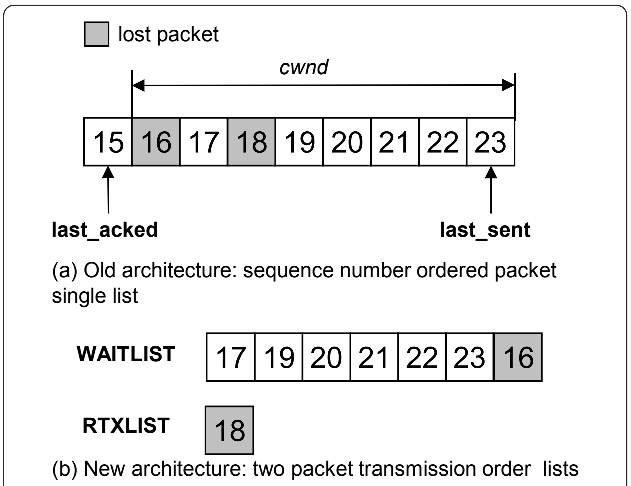

lost packet
cwnd
15
16
17
18
19
20
21
22
23
last_acked
last_sent
(a) Old architecture: sequence number ordered packet single list
WAITLIST
17
19
20
21
22
23
16
RTXLIST
18
(b) New architecture: two packet transmission order lists

Figure 1 Comparison between old and new architecture.

to the time stamp of the ACK. If the *timeStamp* is less than the time stamp of the ACK and the *seqNum* is less than or equal to the cumulative number of the ACK, the packet is acked. If the *timeStamp* is equal to the time stamp of the ACK, SACK blocks are used for acking the packet. If the packet is not acked, the *dupCnt* increases by one. If *dupCnt* reaches the *dupThresh* (usually 3), the packet is considered as a loss and is moved to the end of RTXLIST for retransmission. If the *timeStamp* is larger than the time stamp of the ACK, scanning WAITLIST stops. Figure 2 depicts an example of this situation.

2.2.2 Retransmission-first packet transmission

LE can transmit a new packet or lost packet if the number of on-flying packets is less than *cwnd*. As the number of on-flying packets is the same as the number of packets in WAITLIST, if the number is less than *cwnd*, LE transmits packets. If RTXLIST is not empty, LE gives priority to RTXLIST and sends the packets first. New packets can be sent only when RTXLIST is empty and when the transmission condition is still valid. After the *timeStamp* is updated and the *dupCnt* is set to zero, the packet is removed from RTXLIST and inserted at the end of WAITLIST.

2.2.3 Per RTT congestion window reduction

LE has no discrete states between the loss recovery period and the normal period. To prevent LE from too frequently reducing *cwnd*, LE introduces the variable *las_loss_time*. When the first lost packet is detected, the lost time is recorded in this variable. Whenever a packet loss occurs, LE checks if the difference between the current time and the *last_loss_time* is greater than an RTT. If so, LE updates the *last_loss_time* with the current time and uses the congestion control routine. Otherwise, LE ignores the packet loss since LE treats multiple packet losses in an RTT as a single loss event, as does NewReno.

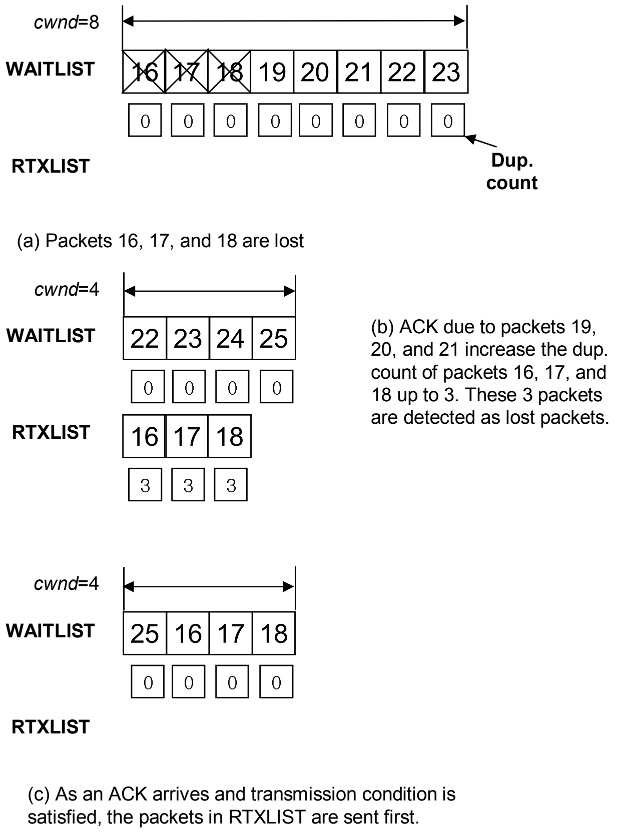

cwnd=8
WAITLIST
16 17 18 19 20 21 22 23
0 0 0 0 0 0 0 0
RTXLIST
Dup. count
(a) Packets 16, 17, and 18 are lost
cwnd=4
WAITLIST
22 23 24 25
0 0 0 0
RTXLIST
16 17 18
3 3 3
(b) ACK due to packets 19, 20, and 21 increase the dup. count of packets 16, 17, and 18 up to 3. These 3 packets are detected as lost packets.
cwnd=4
WAITLIST
25 16 17 18
0 0 0 0
RTXLIST
(c) As an ACK arrives and transmission condition is satisfied, the packets in RTXLIST are sent first.

Figure 2 An example of LE's acking process when combined with AIMD.

2.2.4 Fine grain RTT and RTO timer setting

LE calculates RTT when the time stamp of an ACK that acknowledges a packet which matches with the *timeStamp* of the packet because the ack was generated due to the packet. The RTO timer is set when there are no on-flying packets and when a new packet is sent. When the *last_acked* variable, which stores the last cumulative sequence number, is advanced, the RTO timer is reset. When the *last_acked* cannot be advanced at the RTO timeout value, RTO expiry will occur.

Observation: Assume that n packets are lost. As long as LE receives at least three duplicate ACKs, all lost packets can be recovered without RTO expiry.

Proof. Let us assume that n packets are lost in the last RTT period. Then, LE receives duplicate ACKs due to the loss. When LE receives the first two duplicate ACKs, since the number of on-flying packets is reduced by two, LE can transmit at least two new packets. Receiving the third duplicate ACK, LE sets the *dupCnt* of the first lost packet to 3 and moves the packet into RTXLIST. According to the retransmission first strategy, LE retransmits the head of RTXLIST and simultaneously sets the value of *cwnd* to $\frac{w_0}{2}$, where w_0 is the size of *cwnd* before detecting the first packet loss. In the next RTT, when LE receives ACKs due to two new sent packets and one retransmitted packet, the *dupCnt* of all lost packets will be three and they will all be moved to RTXLIST, which results in WAITLIST to be empty. Therefore, LE can send as many packets as *cwnd*. Therefore, in the next RTT period, LE receives at least three ACKs and does not invoke RTO expiry.

2.3 Contributions of LE

2.3.1 Extending the self clocking property by resolving ACK starvation during loss recovery

By retaining the acket transmission order, LE measures the exact number of lost packets upon a single duplicate ACK, making it possible to strictly extend the self clocking property. In other words, LE transmits the number of packets equal to that exiting networks even during loss recovery. Therefore, even when there are not enough duplicate ACKs to transmit lost packets during loss recovery, LE does not incur RTO expiry and successfully recovers lost packets.

2.3.2 Simple code by decoupling loss recovery and congestion control

An advantage of the separate RTXLIST is the decoupling transmission decision from the loss recovery. Note that in our architecture, loss recovery is not affected by transmission of lost packets, and its role ends after moving packets to RTXLIST. The transmission is handled by congestion control. Nor does the transmitter depend on whether the packet is new or a retransmitted one except that it gives priority to those in RTXLIST. This decoupling simplifies the source code.

2.3.3 Simple and efficient loss recovery

SACK and FACK cannot recover the loss of retransmitted packets until timer-expiration. The proposed method can handle heavy losses including loss of retransmitted packets by keeping the transmission order information in WAITLIST, a natural extension of [19]. Existing loss recovery algorithms limit the scope of packets for recovery to those that have a sequence number between lastAcked and lastSent. After recovering the packets up to lastSent, SACK returns to normal phase. Therefore, when packets after lastSent are lost, SACK reenters the loss recovery phase. Considering the cost of each transition, freezing the transmission for the third duplicate ACKs, and re-initializing the scoreboard data structure, scope limitation can cause inefficiency. By not limiting the scope, our method can be considered as a constant loss recovery mechanism.

2.3.4 Resilience to packet reordering

Transmission control protocol considers acket reordering as a sign of congestion and controls its window size. Ideas such as variable dup count or non-use of duplicate ACK have been proposed [14]. Even though we did not propose a solution to the reordering problem, due to our enhanced architecture we believe it is more resilient to this problem. Moreover, techniques such as variable duplicate ACKs can be combined with our proposal.

3 Performance evaluation

In this section, we discuss performance of our loss recovery proposal in the view of loss recovery ability, loss recovery effect on wireless TCP, and multimedia transmission. To analyze the loss recovery ability and effects on various TCPs, and multimedia transmission, we conducted extensive NS-2 [20] simulations to inspect the effects of loss recovery in various scenarios. For the simulation, the dumbbell topology shown in Figure 3 was used and the size of the TCP packet was set to 1 kb, and the default simulation time was 100s if not mentioned otherwise.

3.1 Loss recovery ability

To evaluate the loss recovery ability of LE, we compare it with various TCP implementations. The loss recovery duration, the time between the first loss and full packet recovery, was used as the performance metric. Figure 4 shows the results of different TCP implementations. The x-axis corresponds to the number of lost packets in an RTT, while the y-axis corresponds to the recovery duration. We randomly choose a certain number of packets among all outstanding packets, dropped them into the bottleneck links and measured the duration. Each point in the plots is the average value of ten different runs. We assumed that a retransmitted packet is not lost again. If we allow loss of retransmitted packet, the conventional TCPs except LE will eventually fall into RTO expiry irrespective of the number of lost packet [19]. Through the assumption, we need to exclude the RTO expiry due to repeated lost of retransmitted packet because the purpose of the simulation is to measure how fast each TCP will successfully recover all the lost packets as the number of lost packet increases. We limited the maximum *cwnd* to 32 for ease of simulation in order to differentiate each packet train in the bottleneck link.

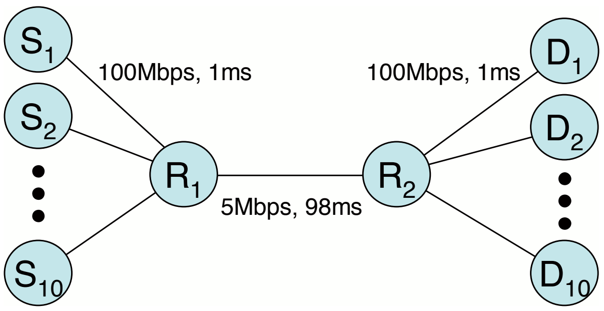


Figure 3 Basic simulation topology: dumbbell shape.

3.1.1 Overall loss recovery performance is in the order of LE, FACK, SACK, NewReno and Reno

As shown in Figure 4, the duration of NewReno is lower than that of Reno, which is partly due to the fact that NewReno has a superior counting mechanism. SACK outperforms NewReno for up to 18 lost packets. SACK

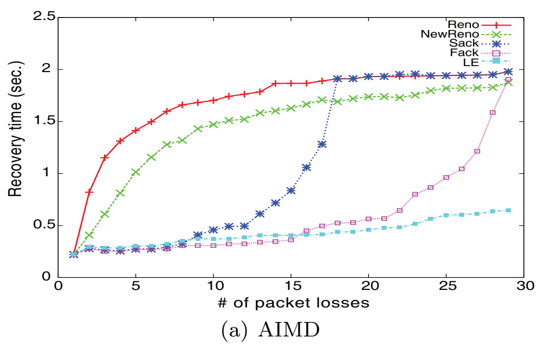


Figure 4 Average loss recovery period vs. packet loss rate.

recovers the lost packets up to eight lost packets $\left(\frac{cwnd}{4}\right)$ in a single RTT. If the number of losses is greater than 18 $\left(\frac{cwnd}{2}+2\right)$, SACK suffers from RTO expiry, as shown in Figure 4. Observe that the performance of SACK becomes worse than that of NewReno if the number of losses is greater than 18. Even though NewReno experiences RTO expiry at the same time as does SACK, the *retransmission ambiguity problem* [21] allows NewReno to possess a shorter recovery time than that of SACK.[a] FACK exhibited single RTT recovery up to 15 lost packets. From that point on, the loss recovery duration increased exponentially. The recovery duration of LE was almost linear up to 29 of 32 losses, which confirming a superior loss recovery ability than those of other variations.

3.2 Fairness of LE

The goal of this simulation was to measure LE's fairness with SACK. We choose FACK's fairness as a baseline. First, we ran a total of ten flows combining SACK and FACK. The number i of SACK flows was variably set to 1, 3, 5, 7, 9. The number of FACK flows was set to be 10 - i. LE was also tested under the same condition. We assume that packet loss rate is 0% in this simulation.

3.2.1 LE shares fair bandwidth with SACK

Figure 5b shows LE's fairness with regard to that of SACK. Comparing Figure 5b with Figure 5a, representing FACK's fairness with SACK, shows that the fairness of LE is comparable to that of SACK. This was verified from the fact that LE showed the same throughput curve as those of the other methods at the same packet loss rate, as shown in Figure 6a. Because the effect of removal of RTO expiry was amortized during the session, LE is not considered to be an aggressive method.

3.3 LE with different congestion control algorithms

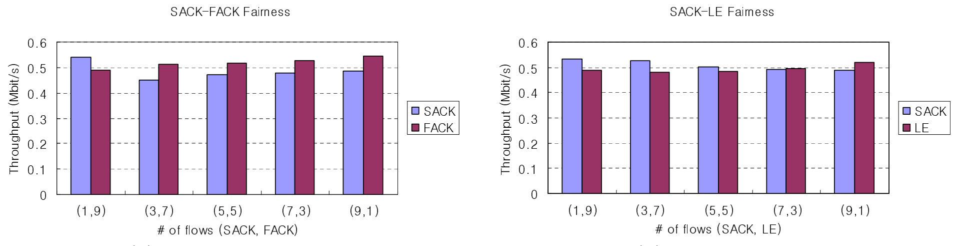
We compared the impacts of the loss recovery algorithms with the congestion control algorithms, additive increase multiplicative decrease (AIMD), fixed window, and that of Westwood. Congestion control referred to the way *cwnd* was set by TCP. In the case of AIMD, traditional Reno type congestion window control was used. Fixed window control does not change its window size; this algorithm was introduced to determine the impact of loss recovery in the absence of the congestion control algorithm. The Westwood congestion control utilizes available bandwidth information for setting *cwnd* values. We compared five different loss recovery algorithms: Reno, newReno, SACK, FACK and LE. The loss recovery of Reno and that of newReno are similar, but newReno improves retransmission during the fast recovery phase. SACK is an improvement over Reno and newReno due to utilizing of the selective ACK option. FACK aims to trigger faster retransmission by utilizing additional variables compared to those of SACK.

Figure 6 shows the throughputs of 15 congestion control and loss recovery combinations. The x-axes in all graphs correspond to packet loss rate, the duration of each simulation run was 100s, and ten runs were averaged for each point in the graph. We generated single ftp flow between a sender and a receiver in (a)-(c) while there are ten simultaneous ftp flows in (d).

3.3.1 LE with AIMD congestion control

LE under AIMD exhibits the same throughput curve as do the other loss recovery algorithms. We replaced the loss recovery part of TCP-Reno with our algorithm. To our surprise, when typical AIMD control of TCP was used, the throughputs of different TCP implementations were very similar. LE performed slightly better, but the impact was marginal, as shown in [[22], Figure 6a].

The number of losses in a single window was typically less than one or two packets in most cases, the reason why the advanced loss recovery of LE was not very helpful in this case. When the loss rate was low, the number of lost packet was typically one, when the loss rate was high, the average *cwnd* was very low, for example, 4. Even with a 10% loss rate, as the average window size was small, the number of lost packets in a single RTT was small.



(a) SACK and FACK (b) SACK and LE

Figure 5 Fairness of LE.

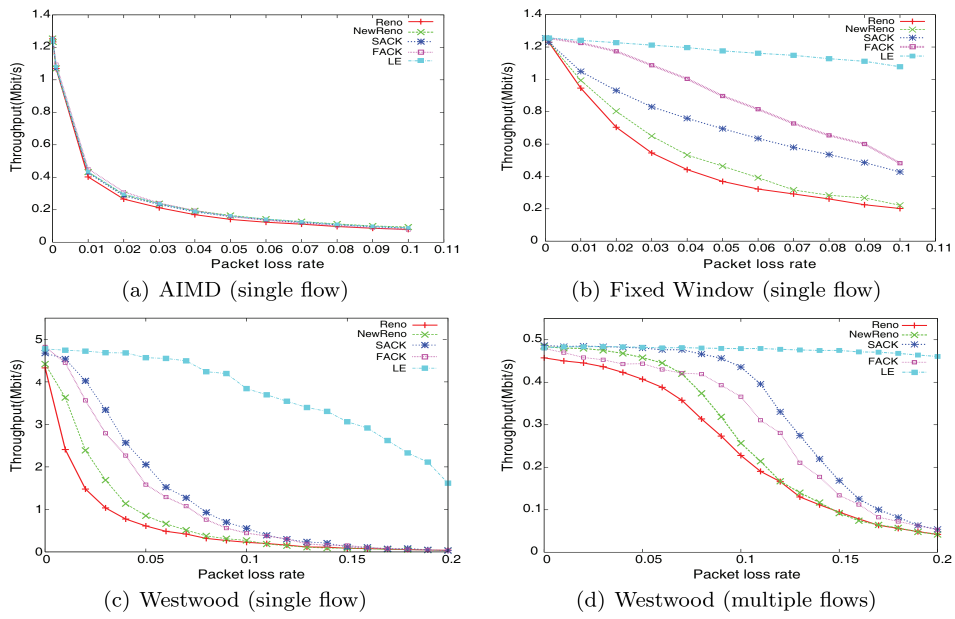


Figure 6 Performance of LE with different congestion control algorithms: AIMD, fixed, and Westwood.

3.3.2 LE with fixed window

The gain of LE increases as cwnd becomes larger. To remove the impact of congestion control on loss recovery, the congestion window was set to 32 packets and the test was repeated. The results of this simulation are shown in Figure 6b. The throughput of LE was superior to those of other loss recovery algorithms, demonstrating the effectiveness of LE's loss recovery algorithm. The throughputs of Reno, newReno, SACK, and FACK were inferior to that of LE because they frequently experience RTO expiry, during which they cannot send packets. As LE can quickly detect the loss of retransmitted packets or tolerate insufficient duplicate ACKs, LE does not experience RTO expiry and therefore saves time, maintaining the transmission rate.

3.3.3 LE with Westwood congestion control

Transmission control protocol-Westwood adopts a better congestion control algorithm by estimating the available network bandwidth and is proposed for wireless networks [23]. We coupled our loss recovery algorithms with the congestion control of TCP-Westwood and generated a single ftp flow, the throughput of which is shown in Figure 6c. We also generated ten simultaneous ftp flows, and the average throughput is shown in Figure 6d.

- *LE extends the performance of TCP Westwood*. Using the congestion control algorithm of Westwood, the performance of LE is significantly better than those of the other loss recovery algorithms, as shown in the Figure 6. Even though TCP Westwood intelligently maintains a larger *cwnd* compared to the AIMD under random packet loss, the existing loss recovery cannot maintain the throughput because of RTO expiry as packet loss increases. This occurrence is illustrated in Figure 6c. In the case of multiple flows, similar results are shown in in Figure 6d. LE flows achieve almost full bandwidth utilization because LE avoids RTO expiry and associated wasted time.

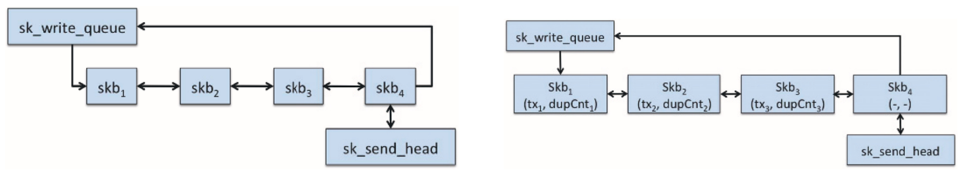
4 Implementation of LE

We implemented the proposed architecture using the Linux kernel version of 2.6.17 to assess the proposed idea.

4.1 Logical implementations of the two lists

Instead of implementing the two lists explicitly, we mimicked their behaviors by slightly modifying the existing architecture.

The linux version of TCP uses a data structure called *scoreboard*[b] to maintain a linked list of packets that have been transmitted or will be transmitted as shown in Figure 7a. Each entry in the list is called a socket buffer (SKB), and contains a pointer to a data packet and related control information such as the status. The scoreboard maintains the SKBs of a transmitted packet until it is properly ACKed by a receiver. Upon reception of an ACK, TCP releases the corresponding SKB from the list. Two pointers are used in the scoreboard; the first one, *sk_write_queue*, points the first unacknowledged packet and the second one, *sk_send_head*, points the beginning of the packets that have not yet been transmitted. The packets between the two pointers are



(a) Original Scoreboard (b) Modified Scoreboard

Figure 7 Implementation of two lists.

those transmitted but not acked yet. The remaining packets are to be transmitted next.

To implement the two-list architecture, we added an additional control field, *transmission order*, to the SKB data structure, to show the packet transmission order. Whenever a packet was initially sent or resent, its transmission order was recorded. Therefore, each packet from the *sk_write_queue* to the *sk_send_head* had its own transmission order. The RTXLIST was implemented using the status variable in the SKB data structure. When the status variable was set as *LOST*, the packet was considered to be in the RTXLIST. Sequential traversing of the list allows for determination of those packets in the RTXLIST. The *dupCnt* variable was also declared in the SKB in order to implement the per packet acking process. Whenever a duplicated ACK arrived, the *dupCnt*s of the corresponding SKB increased by one up to the threshold of 3. Every SKB having the *dupCnt*s value was marked as LOST.

Our implementation is advantageous as follows. First, by minimizing the restructuring of the SKB data structure, most of existing code can be reused without major code development. Most of existing TCP codes were reused except the parts related to packet recovery and retransmission. Second, byte sequential ordering of packets is needed when a packet is retransmitted due to RTO timeout. By maintaining the sequential ordering, the RTO timeout operation can be easily conducted.

4.2 Fine-grain RTT and RTO timer setting

In order to implement *per packet RTT measurement*, which is used for *per RTT congestion control* and for *fine-grained RTO timer setting*, we need to identify the packet generating the current incoming ACK.

Initially, we only considered the Linux TCP timestamp for the above purpose. However, we found this is not sufficient. Some packets belonging to a single window may sometimes have the same timestamp value due to the granularity of the Linux TCP timestamp, on the order of tens of milliseconds.

Additional information should be used to differentiate the packets with the same timestamp. The variations in cumulative ACK number and SACK blocks may be a solution. For example, the increase in the cumulative ACK number correctly determines which packet generates this ACK. In the case of a duplicated ACK, the change in SACK blocks also provides information on the increase in the cumulative ACK. Therefore, we need to remember a few of the consecutive incoming ACKs to trace the change in acking. Combining the differences with the timestamp, the packet corresponding to an incoming ACK can be determined.

For rapid implementation, however, we simply included *transmission order* in the TCP header instead of maintaining a few of ACKs. If the packet arrives at a TCP receiver, the receiver echoes the transmission order value in its replying ACK. We used the transmission order in an ACK to find the associated packet in order to measure per packet RTT.

4.3 Detection of lost packets

Upon arrival of a normal or duplicated ACK, TCP sequentially compared the cumulative ACK number with the bytes of the already sent SKBs in WAITLIST. The SKBs having bytes less than or equivalent to the cumulative ACK number were considered to be successfully arrived. If the bytes of an SKB did not belong to the cumulative ACK number, SACK blocks were used to ack the SKB. If the bytes of the SKB were less than the largest byte in the SACK blocks and were not acked, the *dupCnt* in the SKB increased by one. If the *dupCnt* reached three, the SKB was marked as LOST. WAITLIST scanning stopped until the next SKB reached the value pointed by *sk_send_head*. The recognition of a lost packet triggered the congestion control as described in Section 2.2.

5 Experimental results

5.1 Testbed setup

To evaluate the performance of LE, we set up a testbed as shown in Figure 8, consisting of three linux-based PCs with kernel version 2.6.17. The two outer PCs were used to mimic a sender and a receiver, while the middle one emulated a network. We used the Netem [24] network emulator, which enabled us to control packet drop rates and round trip delays. To measure performance, we used the *iperf* software tool [25]. In the emulation, we fixed RTT to 20 ms and varied the packet drop rate

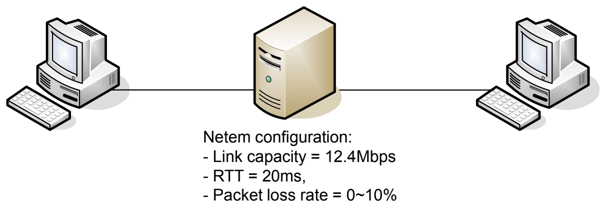


Figure 8 Testbed configuration.

from 0 to 10%. We limit the packet drop rate up to 10% because TCP-RR is likely to malfunction as the packet loss rate becomes more than 10% while TCP-Westwood affords to achieve a transmission rate more than 10% and even up to 20%. Thus, considering a realistic packet drop rate for the fair comparison between TCP-Westwood and TCP-RR, we have used 10% as the maximum packet drop rate.

5.2 LE with various congestion control algorithms

We evaluated the performances of the new loss recovery algorithm with two different congestion control algorithms: TCP-westwood and TCP-RR. The results are shown in Figure 9, and our main observations are as follows.

- Though the new loss recovery algorithm achieved better performance with TCP-Westwood, the performance improvement was not as significant as that in the simulation of Figure 6c.
- With TCP-RR congestion control, a performance improvement of 50% was observed when the loss rate is high.

5.2.1 TCP-Westwood

First, the performance of LE was better than that of FACK. At the loss rate of 10%, LE achieved more than 4 Mbps while SACK and FACK achieves approximately 3.5 Mbps. The line with triangle tick shows the throughput with a fixed congestion window of 10, which can be considered as an upper bound. Observe that the throughput slowly decreased with increasing loss rate. Even though LE did not approach the ideal line, it achieved almost 80% of the ideal value.

Second, though LE achieved a relatively higher throughput, the testbed results were very different from the simulation results of Figure 6c. We expected a much better performance of LE with Westwood congestion control. We investigated the issue to determine the possible causes of the large discrepancies between the simulation and emulation. It turned out that the Westwood available bandwidth algorithm did not function as well in the testbed experiment as in the simulation. When packet loss rate was less than 0.1%, the measured bandwidth approached the correct one. However, when the loss rate was greater than 1%, the measured bandwidth is smaller by 1 Mbps, which is why the testbed results are poorer. Because the estimated bandwidth was so low in the testbed, a worse performance was concluded for Westwood. For this reason, the throughputs of all loss recovery algorithms exponentially decreased at packet loss rates greater than 1%. This experiment led us to conclude that the bandwidth estimation algorithm of TCP-Westwood does not perform well in a real environment.

5.2.2 Simplified TCP-RR

Because the congestion control algorithm of TCP-Westwood was not efficient, we implemented another TCP variation called TCP-RR [14] to validate the effectivness of the new proposal. Briefly, TCP-RR assumes that only RTO expiry is equivalent to packet loss due to network congestion. Multiple duplicate ACKs are due to wireless transmission error. However, to maintain the fairness with another TCP, the time of RTO expiry is less than the traditional RTO expiry. We simply implemented TCP-RR-like congestion control by adjusting the time of RTO expiry to a smaller value.

The right plot of Figure 9 shows the performance difference between LE and FACK. The performance degradation with packet error rate was smooth, and TCP-RR

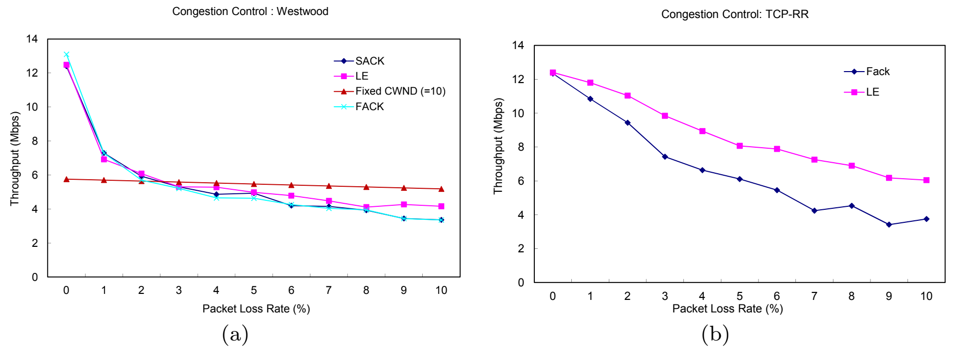


Figure 9 Loss recovery performance of LE versus FACK when TCP-Westwood (left) and TCP-RR (right) congestion algorithms are used.

maintained good throughput with a packet loss rate of 10%. LE with TCP-RR achieved 6 Mbps throughput at the loss rate of 10%.

6 Conclusions

We proposed a new loss recovery architecture for TCP and validated its effectiveness through extensive simulation and emulation experiments. The new architecture maintains packet transmission order so that loss recovery is more effective. This feature is the greatest difference from the conventional loss recovery in TCP whose loss recovery ability is seriously affected by the number of lost packets and frequently causes unnecessary RTO expiry. Our method sustains the loss recovery ability in TCP irrespective of the number of lost packets. We combined the new loss recovery algorithm with various TCP congestion control algorithms such as AIMD, TCP-Westwood, and TCP-RR and demonstrated higher throughput for higher loss rates. We also demonstrated that the proposed algorithm can be used for video transmission over a lossy channel. We will extend our TCP for multimedia transmission based on overlay multicast in which TCP sessions are used either to construct a logical multicast tree or to deliver multimedia streams.

Endnotes

[a]In this simulation, NewReno eventually falls into RTO expiry after retransmitting the fifth lost packet because RTO expiry is much faster than sequential loss recovery when NewReno should recover a numerous losses. However, after RTO expiry retransmits the fifth packet, the ACK of the fifth packet immediately arrives due to the initial packet transmission. [b]The term scoreboard was first introduced for TCP SACK in order to maintain the packet acking status. In Linux, it is used as a base data structure for all kinds of TCP to keep track of the packet acking status.

Acknowledgements
This study for the first author was supported by research funds from the Chosun University, 2007.

Author details
[1]School of Computer Engineering, Chosun University, 375 Seosuk-dong, Dong-gu, Gwangju 501-759, Republic of Korea [2]Department of Information and Industrial Engineering, Yonsei University, Seoul, Republic of Korea

Competing interests
The authors declare that they have no competing interests.

Received: 31 December 2011 Accepted: 19 April 2012
Published: 19 April 2012

References

1. V Jacobson, MJ Karels, Congestion avoidance and control. ACM SIGCOMM. **18**(4), 314–239 (1988)
2. K Fall, S Floyd, Simulation based comparisons of Tahoe, Reno and Sack TCP. Comput Commun Rev. **26**(3), 5–21 (1996)
3. M Mathis, J Mahdavi, S Floyd, A Romanow, TCP Selective Acknowledgement Options, RFC 2018. (Apr 1996)
4. M Mathis, J Mahdavi, Forward acknowledgment: refining TCP congestion control, in *Proceedings of SIGCOMM'96* (August 1996)
5. H Balakrishnam, S Seshan, E Amir, R Katz, Improving TCP/IP performance over wireless networks, in *First ACM International Conference on Mobile Computing and Networking (MOBI-COM)* (Nov 1995)
6. A Bakre, BR Badrinath, Indirect TCP for mobile host, in *15th International Conference on Distributed Computing Systems* (1995)
7. B Hari, NP Venkata, S Srinivasan, HK Randy, A comparison of mechanisms for improving TCP performance over wireless links. IEEE ACM Trans Netw. **5**(6), 756–769 (1997)
8. S Mascolo, MY Sanadidi, C Casetti, M Gerla, R Wang, TCP Westwood: end-to-end congestion control for wired/wireless networks. Wirel Netw J. **8**, 467–479 (2002)
9. A Capone, L Fratta, F Martignon, Bandwidth estimation scheme for TCP over wireless networks. IEEE Trans Mobile Comput. **3**(2), 129–143 (2004)
10. K Xu, Y Tian, N Ansari, TCP-Jersey for wireless IP communications. IEEE JSAC. **22**(4), 747–756 (2004)
11. B Saad, HV Nitin, Discriminating congestion losses from wireless losses using inter-arrival times at the reciever, in *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology* (1999)
12. S Cen, PC Cosman, GM Voelker, End-to-end differentiation of congestion and wireless losses. IEEE/ACM Trans Netw. **11**(5), 703–717 (2003)
13. EH-K Wu, MZ Chen, JTCP: jitter-based tcp for heterogeneous wireless networks. IEEE J Sel Areas Commun. **22**(4), 757–766 (2004)
14. M Zhang, B Karp, S Floyd, L Peterson, RR-TCP: a reordering-robust TCP with DSACK. in *Proc of ICNP* (Nov 2003)
15. S Bohacek, J Hespanha, J Lee, C Lim, K Obraczka, TCP-PR: TCP for persistent packet reordering, in *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems* (May 2003)
16. S Bhandarka, NE Sadry, ALN Reddy, NH Vaidya, TCP-DCR: a novel protocol for tolerating wireless channel errors. IEEE Trans Mobile Comput. **4**(5), 517–529 (2005)
17. M Kang, J Mo, A new loss recovery architecture for wireless TCP. IEEE Commun Lett. **9**(11), 1018–1020 (2005)
18. M Kang, J Mo, Packet transmission order based TCP loss recovery algorithm: extending self clocking property to resolve ACK Starvation. WOWMOM. **2007**, 1–8 (2007)
19. B Kim, D Kim, J Lee, Lost retransmission detection for TCP SACK. IEEE Commun Lett. **8**(9), 600–602 (2004)
20. UCB/LBNL/VINT Network Simulator http://http//www.isi.edu/nsnam/ns
21. P Karn, C Partridge, Estimating round-trip times in reliable transport protocols, in *Proceedings of the ACM SIGCOMM '87* (Aug 1987)
22. Q Ni, T Turletti, W Fu, Simulation-based analysis of TCP behavior over hybrid wireless & wired networks, in *The 1st International Workshop on Wired/Wireless Internet Communications (WWIC 2002), in conjunction with Internet Computing 02*, Las Vegas, Nevada, USA, (24-27 June 2002)
23. S Mascolo, C Casetti, M Gerla, MY Sanadidi, R Wang, TCP westwood: Bandwidth estimation for enhanced transport over wireless links, in *Proc of ACM MobiCom* 287–297 (2001). 2001
24. Network Emulation http://linux-net.osdl.org/index.php/Netem
25. A Tirumala, End-to-end bandwidth measurement using iperf. SC'2001 Conference CD, ACM SIGARCH/IEEE (2001)

doi:10.1186/1687-1499-2012-149
Cite this article as: Kang *et al.*: **Implementation and evaluation of a new TCP loss recovery architecture.** *EURASIP Journal on Wireless Communications and Networking* 2012 **2012**:149.