오프셋을 활용한 효율적인 TCP SACK 메커니즘

최 린[†] · 홍 충 선[††]

요 약

TCP SACK은 sink의 순차적인 필드 상태를 나타내는 유일한 메커니즘이며, 여러 가지 변형된 TCP들은 최적의 성능을 위해서 SACK 메커니즘을 적용할 수 있다. RFC 2018에서 SACK 옵션은 수신자 측에 쌓여진 데이터 큐 각각의 연속된 블록으로 2개의 32비트로 정의되어 있다. TCP 옵션 필드는 최대 40바이트 길이를 가지기 때문에 에러가 발생하였을 때, TCP 수신자 큐에 있는 모든 데이터 블록들을 알려줄 수 있는 사용 가능한 옵션 공간이 충분하지 않으며, TCP 송신자가 TCP sink에 의해서 수신된 패킷들을 불필요하게 재전송하게 된다. 이러한 문제들을 해결하기 위해서 본 논문에서는 TCP SACK의 성능을 향상시키고 불필요한 재전송을 제거하기 위해서 "one-byte offset based SACK mechanism" 이라는 새로운 방식을 제시한다. 제안된 방식의 분석과 시뮬레이션 결과 제안된 방식은 최소한의 바이트를 사용하기 때문에 다른 메커니즘들보다 오버헤드를 줄였고, 유무선 통합 환경에서 에러율이 적은 효율적인 메커니즘임을 입증하였다.

키워드 : TCP, SACK, HACK

An Effective Solution to Overcome the Restriction of SACK Blocks' Number in TCP SACK

Cui, Lin Hong[†] · Choong Seon[††]

ABSTRACT

TCP SACK is the unique mechanism to reflect the situation of sink's sequence space, some TCP variants and proposals can perform in conjunction with SACK mechanism for achieving optimal performance. By definition of RFC 2018, however, each contiguous block of data queued at the data receiver is defined in the SACK option by two 32-bit unsigned integers in network byte order. Since TCP Options field has a 40-byte maximum length, when error bursts occur, we note that the available option space may not be sufficient to report all blocks present in the receiver's queue and lead to unnecessarily force the TCP sender to retransmit packets that have actually been received by TCP sink. For overcoming this restriction, in this thesis, a new solution named "one-byte offset based SACK mechanism" is designed to further improve the performance of TCP SACK and prevent those unwanted retransmissions. The results of both theory analysis and simulation also show that his proposed scheme operates simply and more effectively than the other existing schemes by means of the least bytes and most robust mechanism to the high packet error rates often seen in networks environment.

Key Words : Transmission Control Protocol, Selective Acknowledgment, Header Checksum

1. Introduction

Transmission Control Protocol(TCP) is a transport protocol. TCP provides reliable connection-oriented data delivery by employing acknowledgments(ACKs), sequence numbers, and timers. TCP has been widely applied in many applications : WWW, E-mail, file transfer, remote login, database access, X-windows, satellite communication, etc. Currently, most network traffics are carried by TCP.

TCP uses a cumulative acknowledgment scheme in which out-of-sequenced data packets cannot be covered by acknowledgement number field in TCP header. It results that TCP sender only can retransmit just one dropped packet per round-trip time. Hence, since multiple packets are lost from a window, this forces the sender to either wait a more round trip time to find out each lost packet, or unnecessarily retransmit segments which have been successfully received. Therefore, multiple packet losses from a data window of TCP can have a catastrophic effect on TCP throughput. In order to correct this unwanted behavior, the TCP Selective Acknowledgment(TCP SACK) scheme has been proposed in the

※ 본 연구는 MIC 및 ITRC Project, 그리고, 정통부 IT학과 장비지원사업의 경희대학교 대응연구비로 이루어졌음.
† 준 회 원 : 경북대학교 컴퓨터과학과 박사과정
†† 종신회원 : 경희대학교 컴퓨터공학과 교수
논문접수 : 2005년 7월 21일, 심사완료 : 2005년 11월 14일

face of multiply dropped segments. With this SACK mechanism, the data receiver can inform the sender the information about all segments that have arrived successfully, thus TCP sender can retransmit those dropped segments at the same time as defined in RFC 2018(TCP Selective Acknowledgement Options)[1].

Although with the emergence of new TCP variants, a plenty of studies show that TCP SACK is not the most perfect in comparison with other TCP variants' throughput, SACK has an outstanding advantage which cannot be possessed by others, that is, SACK mechanism is the unique mechanism to reflect the situation of sink's sequence space, other TCP variants and proposals can perform in conjunction with SACK mechanism for achieving optimal performance. For example, TCP Header Checksum (HACK) mechanism can achieve the most optimal performance as running together with TCP SACK [2].

However, by definition of RFC 2018[1], each contiguous block of data queued at the data receiver is defined in the SACK option by two 32-bit unsigned integers as two edges' sequence number, hence a SACK option that specifies n blocks will have a length of 8*n+2 bytes. Because TCP Options field has a length limit of 40 bytes, and Timestamp option is expected for TCP extensions for high performance[3], which takes an additional 10bytes (plus two bytes of padding), thus SACK option will have the room for only three SACK blocks at most in usual case, which is also the default value in TCP SACK in the network simulator(ns-2). Furthermore, if the SACK option is to be used with both the Timestamp option and T/TCP(TCP Extensions for Transactions)[4], the TCP option space would have room for only two SACK blocks [7].

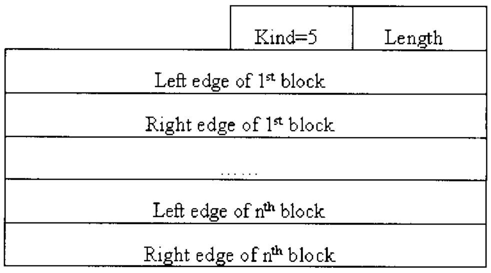
This restriction can, under error bursts' environment, cause unnecessary retransmission because of no enough space to convey all information about received segments. Thereby it will bring unnecessary shrinkage of TCP congestion window and degrade the performance of TCP connections.

In this paper, we present a novel solution named "One-byte Offset Based SACK Mechanism"(hereinafter referred to as "OOBSM") to overcome this restriction. By implementation, it is shown that the TCP receiver can convey entire information of receiver's data queue using a few bytes(<10bytes), no matter how poor the network environment is. In other words, we optimize the structure of TCP SACK option format by minimizing its size by an order of magnitude down to several bytes. The proposed OOBSM scheme thus ensures that the SACK mechanism can be unrestrictedly implemented in conjunction with any other mechanisms related to the TCP option field for achieving a better performance.

2. Related Works

2.1 Standard SACK Option Format

The standard SACK option format shown in (Fig. 1) is defined in RFC 2018[1]. Its limitation has been described in section 1.



(Fig. 1) Standard SACK option format

2.2 Option Format of HACK + SACK

The HACK scheme was proposed to extend TCP by adding a separate checksum for the header portion of each segment while traditional TCP carried only one checksum, which was for the whole segment. By means of this strategy in lossy environments, even if corruptions occur in the segments' data portion due to the random nature of the bit errors, as long as the header is integrated, TCP receiver will send a prompt special ACK back to the sender to indicate the corrupted segment at once. Hence TCP sender can distinguish corruption from congestion and retransmit this corrupted segment immediately without the necessaries to wait for three duplicate ACKs and shrink congestion window.

Nevertheless, HACK will only ask for retransmissions of the corrupted segments whose headers can be recovered and have no help for those irretrievable ones. As a reinforcement leverage upon the out of order packets, if SACK is also activated, then all damaged segments will be detected and handled by different schemes accordingly.

For example, if say 5 packets are corrupted, HACK may only be able to recover the headers of packets 2, 4 and 5 with packets 1 and 3 being irretrievable. Under this situation, HACK will only ask for retransmissions of the packets 2, 4 and 5 in spite of packets 1 and 3. This will leave gaps in the receiving window and lead to degradation of TCP performance finally unless SACK is also activated at the same time for detecting out-of-order segments. Therefore, TCP can achieve a better performance when HACK is implemented in conjunction with SACK.

However, The HACK scheme also needs to occupy option space in TCP header, and its option format is shown in (Fig. 2).

	Kind=X	Length
Left edge of corrupted segment		
Right edge of corrupted segment		
	Kind=5	Length
Left edge of 1st SACK block		
Right edge of 1st SACK block		
Left edge of 2nd SACK block		
Right edge of 2nd SACK block		

(Fig. 2) Option Format of HACK + SACK

2.3 Related Work

In [13], a concept of offset is presented in modification of the SACK option format for sending more information about SACK blocks. The modification of SACK option format is proposed as (Fig. 3). By presentation of [13], instead of sending all absolute 32-bit sequence number for all edges of each segment block, only send one 32-bit absolute sequence number for the right edge of the 1st block(denote it by A). For the rest, represents other edges as offsets from edge A, and denote them by O12, O21, O22, ···On1, On2, where O12 is the offset of the left edge of first block from A, O21 and O22 are the offsets from A to right and left edges of the second block respectively, and so on.

	Kind=5	Length	X
Right edge of 1st block (A) (32-bit absolute sequence number)			
Offset for left edge of 1st block from A	......	......	
	Offset for right edge of nth block from A	Offset for left edge of nth block from A	mis-match part

(Fig. 3) SACK option format proposed in [13]

We noted that, this field which specified the size in bits of each offset is an extra byte comparing with standard SACK option format, labeled 'X'. Its value(X) is variable for every ACK, and result depends on the biggest number among all offsets(denote it by Omax), equaling to:

This means that all offsets belong to one ACK can be represented by X bits. Therefore, as long as sending the number 'X' to TCP sender in SACK option field, TCP sender can correctly read all offsets and be aware of all edges' absolute sequence numbers by subtracting the offset from the value of A.

By definition of SACK option format proposed in [13], n blocks will have a length:

$$X = \lceil \log_2 O_{\max} \rceil \tag{1}$$

$$Length_n = \left\lceil \left(\frac{(2n-1)*X}{8} + 7\right) \right\rceil \tag{2}$$

and the maximum number of n will be(with timestamp option):

$$N_{\max} = \left\lfloor \frac{(30-7)\times 4}{X} + \frac{1}{2} \right\rfloor \tag{3}$$

3. Our Proposal

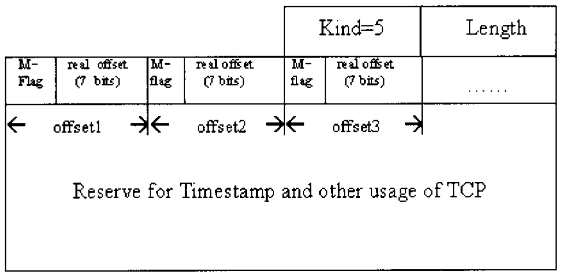
3.1 The Proposed OOBSM Mechanism

Since TCP does not change segment size once the TCP connection is established, and normally all segments preceding the last one has the maximum segment size(MSS) as their size. Hence, given a baseline in absolute sequence number, as long as we know a offset in segment unit from baseline to certain point, then we can accurately work out that points' absolute sequence number by calculating the formula of "baseline in absolute sequence number+offset in segment unit*size of segment in bytes". We well use this characteristic in this paper and present a very effective SACK mechanism. In this OOBSM mechanism, we regard the absolute sequence number indicated by acknowledgement number field in TCP header as baseline, regard every edge of SACK blocks as respective point. Then, we can calculate the first point's absolute sequence number by above formula as long as we know the offset from the baseline to that point, and then regard this point's absolute sequence number as new baseline to compute next point's absolute sequence number. By implementing this cycle till last point, TCP source can be aware of entire situation of sink's sequence space correctly. Therefore, TCP source certainly can effectively retransmit those segments which have actually been lost together with new segments in a burst of data sending and avoid any unnecessary retransmission.

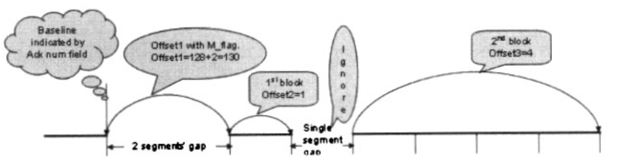
In order to minimize the size of SACK option, referring to (Fig. 4) and (Fig. 5), in addition to using Acknowledgement number field as the baseline, we represent every SACK block by one or two one-byte offsets based on segment unit instead of two 32-bit absolute sequence numbers or sequence number's offsets. As for using one or two offsets to describe one SACK block, it depends on the size of the gap previous that SACK block. If the gap is single segment gap, then one offset is enough; if not, two offsets are necessary. It is because in sink's sequence space, gap and SACK block are one-by-one, therefore, while the gap previous SACK block is single segment gap, the left edge's absolute sequence number of that SACK block can be known as the newest baseline plus the size of segment in bytes by default even if we ignore the corresponding offset. That is to say, between every two one-byte offsets which are smaller than 128, we ignore one offset which related to single segment gap. This method do not influence running efficiency and final correct result at all, but do greatly reduce the size of SACK option and reserve a plenty of option space for other usage of TCP.

In particular, those one-byte offsets named offset*1*, offset*2*, …, offset*n* respectively represent the interval from the first SACK block's left edge with respect to the baseline, the interval from the first SACK block's right edge with respect to the first SACK block's left edge, the interval from the second SACK block's left edge with respect to the first SACK block's right edge, and so on. Wherein, considering the one-by-one relationship between gap and SACK block and the fixed size in bytes of each segment, we ignore those offset corresponding to single segment gap. Obviously, by 32-bit absolute sequence number as baseline(by acknowledge number field), size of segment in bytes and each one-byte offsets based on segment unit, the sender can easily be aware of entire accurate situation of sink's sequence space.

As shown in above (Fig. 4), in our proposed SACK option format, each offset field consists of 1byte. Wherein,



(Fig. 4) Proposed SACK option format



(Fig. 5) Illustration of Proposed SACK

first bit is multiple segments' gap flag(denoted it by M-flag), and other 7 bits represent real offset, 7 bits can represent up to 27-1=127. We know that if the interval between 2 adjacent gaps in sink's sequence space is larger than 127 segments, that means TCP sink have sent up to 127 duplicate ACKs, it will directly lead TCP sender to cause either retransmission timeout or fast retransmission after receiving 3 duplicate ACKs, so we think 1 byte is enough. As for the interpretation of multiple segments' gap flag(M_flag), "1" indicates that this offset corresponds to multiple lost segments' gap; otherwise, "0".

All in all, by OOBSM mechanism, 40 bytes are available for(40-2)/2=19 SACK blocks at least, considering the fact we ignore those offsets indicate single segment's gaps which mostly occur in sequence space, the available maximum number of SACK blocks is far bigger than 19. Actually. It is clear that we do not need to represent so many SACK blocks' information by one ACK packet. Therefore, we can minimize the payload of ACK packet and save enough option space to other usage of TCP so as to further improve TCP performance.

3.2 Evaluation

Through analysis and comparing with RFC 2018[1] and proposal in [13] respectively under the same scenario, we can easily understand the difference from each other. In this scenario, we assume the starting sequence number is 5000 and TCP sender sends a burst of 11 segments, each containing 1000 data bytes. In this sampled scenario, every other segment is lost. (Fig. 6) illustrates the case of current implementation under this scenario; (Fig. 7) shows the case of implementing proposal in [13]; and (Fig. 8) represents the case of implementing OOBSM mechanism.

3.2.1 Current Implementation

From (Fig. 6), it is noted that, in the current implementation, by the 9th segment(sequence number 13000~13999), 34 bytes(2+4*8) have been used up for sending information of 4 SACK blocks. When the 11th segment is received, we are only able to send information about the latest 4 blocks, losing information about the 3rd segment (sequence number 7000-7999). Especially, if time stamp option is used, then just only 3 blocks can be sent, that is, the 5th segment's information will also be discarded(sequence number 9000~9999). While error burst occurs in return path of ACK, it means that TCP sender will unnecessarily retransmit 3rd and 5th segments which have successfully arrived at receiver in fact.

Data packet	ACK Num field	First Block		Second Block		Third Block		Forth Block	
		L edge	R	L	R	L	R	L	R
5000	Normal ACK 6000	NA	NA	NA	NA	NA	NA	NA	NA
6000 (Lost)									
7000	Dup ACK: 6000	7000	8000	NA	NA	NA	NA	NA	NA
8000 (Lost)									
9000	Dup ACK: 6000	9000	10000	7000	8000	NA	NA	NA	NA
10000 (Lost)									
11000	Dup ACK: 6000	11000	12000	9000	10000	7000	8000	NA	NA
12000 (Lost)									
13000	Dup ACK: 6000	13000	14000	11000	12000	9000	10000	7000	8000
14000 (Lost)									
15000	Dup ACK: 6000	15000	16000	13000	14000	11000	12000	9000	10000

(Fig. 6) Case with the current implementation

3.2.2 Implementation of Proposal in [13]

By the implementation of the proposed scheme in [13] as shown in (Fig. 7), because the maximum offset=16000~7000=9000, according to the algorithm defined in [13], we have to allocate every offset field X bits:

$$X=\lceil \log_2 9000 \rceil = 14\ (bits) \tag{4}$$

Hence, the size of SACK option in bits is equal to:

=8(kind)+8(length)+8(define X)+32(absolute sequence number for A)+9(the number of offsets)*14(length of each off-set field)

=182 bits=23 bytes.

If considering timestamp option, the size of used SACK option space will be up to 33 bytes.

That is to say, if not considering the factor of performing in conjunction with other mechanism(e.g. HACK+SACK, etc.), it seems that this approach is also robust enough to cope with errors burst's scenarios. However, it must be pointed out that, for each ACK packet, TCP sink has to compute the maximum offset(Omax), and then take a long runtime to accurately separate every offset field in bits, has to perform a plenty of read or write operation base on bits instead of bytes, it will really take much run-time. Especially, although this proposal can also improve the performance when it performs individually, it will also feel no enough SACK option space when it tries to work in conjunction with other TCP variants or proposals for achieving optimal performance. In conclusion, their usage of SACK option field is inefficient comparing with ours.

Data packet	ACK Num field	X	First Block		Second Block		Third Block		Forth Block		Fifth Block	
			R	L	R	L	R	L	R	L	R	L
5000	Normal: 6000	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
6000 (Lost)												
7000	Dup ack: 6000	10	8000	1000	NA	NA	NA	NA	NA	NA	NA	NA
8000 (Lost)												
9000	Dup ack: 6000	12	10000	1000	2000	3000	NA	NA	NA	NA	NA	NA
10000 (Lost)												
11000	Dup ack: 6000	13	12000	1000	2000	3000	4000	5000	NA	NA	NA	NA
12000 (Lost)												
13000	Dup ack: 6000	13	14000	1000	2000	3000	4000	5000	6000	7000	NA	NA
14000 (Lost)												
15000	Dup ack: 6000	14	16000	1000	2000	3000	4000	5000	6000	7000	8000	9000

(Fig. 7) the implementation of proposal in [13]

3.2.3 Implementation of OOBSM mechanism

Now let us take a look at the implementation of proposed OOBSM mechanism as shown in (Fig. 8). We only need to allocate SACK option space with 1(kind)+1(length)+5*1(offsets)=7bytes, we can still left 33 bytes. Thus, the implementation of proposed OOBSM mechanism is more effective. Especially as performing in conjunction with other mechanisms for improving TCP performance, this mechanism can cope well by using least bytes under any scenarios. Therefore, the proposed OOBSM mechanism not only can convey more information to TCP sender, but also can greatly shrink the SACK option space furthest. It proved that this mechanism is more effective than others.

4. Simulation

We performed the simulation using the ns-2 network simulator, version 2.27[14] and compared our proposed OOBSM's behavior with the implementations of both standard SACK and modified SACK in [13].

Our simulations consist of a single TCP flow traversing a 2M bandwidth 0ms delay link and the flow lasts 200 seconds. In all simulations, each segment contains 1000 bytes user data and each packet size is 1040 bytes. The standard SACK option is assumed to have room for three SACK blocks. The maxburst parameter, which limits the number of packets that can be sent in response to a single incoming ACK packet, is experimental, and is not necessarily recommended for current SACK implementations.

In order to represent SACK blocks' edges by one-byte offset instead of 32-bit absolute sequence number, we did the followings:

-modified the structure of the function of hdr_tcp{};
-replaced the sentence of "int sack_area[NSA+1][2]" by the sentence of "u_char sack2_area[NSA2+1]";
-replaced these two sentences of

"int& sa_left(int n) { return (sack_area_[n][0]); }" and "int& sa_right(int n) {return (sack_area_[n][1]);}" by "u_char& sa2_offset(int n) {return (sack2_area_[n]);}";
-changed the upper limit of maximum number of SACK blocks from 3 to 15.

For sending correct information of entire sink's sequence space by a set of one-byte offsets, we modified the following function:

void Sacker::append_ack (hdr_cmn ch, hdr_tcp* h, int old_seqno)*

On the contrary, in order to let TCP sender be aware of entire situation of sink's sequence space correctly by single ACK, we modified the following function:

int ScoreBoardRQ::UpdateScoreBoard (int last_ack_, hdr_tcp tcph)*

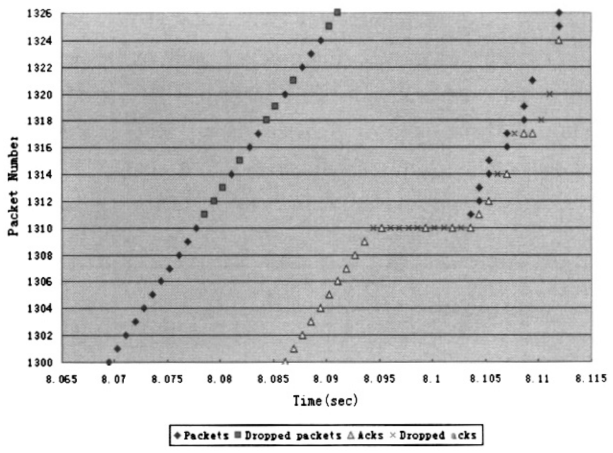
Data packet	ACK Num field	offset1	offset2	offset3	offset4	Offset5
5000	Normal ACK: 6000	NA	NA	NA	NA	NA
6000 (Lost)						
7000	Dup ACK: 6000	1	NA	NA	NA	NA
8000 (Lost)						
9000	Dup ACK: 6000	1	1	NA	NA	NA
10000 (Lost)						
11000	Dup ACK: 6000	1	1	1	NA	NA
12000 (Lost)						
13000	Dup ACK: 6000	1	1	1	1	NA
14000 (Lost)						
15000	Dup ACK: 6000	1	1	1	1	1

(Fig. 8) Case with our proposed implementation

In the implementations of both OOBSM and modified SACK scheme in [13] in simulator, the parameter "maxburst" is limited to four packets even if the sender's congestion window would allow more packets to be sent.

In addition, we used two states error model in order to product error burst effect and in this two states error model, every good state lasts 0.5 seconds and every bad state lasts 0.015 seconds, when this link is in good state, it will continue to last the same state at the next instant with probability 0.95, and with a probability 0.05 the transition to the bad state takes place. Similarly, when this link is in bad state, it will continue to last the same state at the next instant with probability 0.55, and with a probability 0.45 the transition to the good state takes place. Also, this link is error free in good state and takes place packet dropping with probability 0.25 in bad state.

By analysing the behavior of original TCP SACK in (Fig. 9), it is noted that TCP connection suffers two

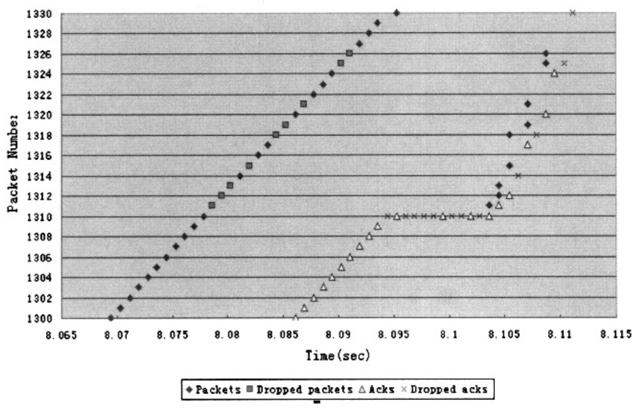


(Fig. 9) The behavior of original TCP SACK

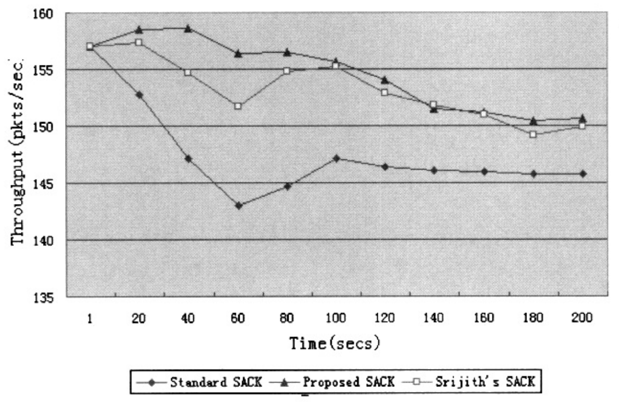
consecutive bad state from around 8.078 second to 8.108. Especially, the situation from around 8.078 second to 8.092 second will directly lead to five SACK blocks in receiver's date queue. In detail, the first block consists of a single segment with number 1314, the second block consists of two segments with number 1316 and 1317, the third block is also a single segment block with number 1320, the fourth block consists of three segments with number from 1322 to 1324 and the fifth block consists of two segments with number 1327 and 1328. Because original TCP SACK only has option space for up to three SACK blocks, receiver has to report last three SACK block information to sender and lose first and second SACK block information. It results sender unnecessarily to retransmit those segments with number 1314, 1316 and 1317 at 8.105 second and 8.107 second respectively which actually have been received by receiver.

In contrary to the behavior of original TCP SACK in (Fig. 9), referring to (Fig. 10), although five SACK blocks are also created in receiver's data queue under the same lossy environment, our proposed OOBSM mechanism can avoid those unwanted retransmission perfectly.

(Fig. 11) shows that when TCP connections are established over a long period under error bursts' environment, a number of unnecessary retransmission of packets can decrease the throughput of TCP connections. Thus the problem with TCP SACK that we see graphically in (Fig. 9) can lead to lower throughout. However, We see that the implementation of proposed OOBSM can avoid most unnecessary degradation of TCP performance and always gives a better throughput than others. Although the modified SACK mechanism proposed in [13] also performs better than the implementation of standard TCP SACK, because of fussy bit-based operation and insufficient capability, its behavior is not good as ours yet when the environment further deteriorates(see the case at 60sec).



(Fig. 10) The behavior of proposed TCP SACK



(Fig. 11) Comparison of various SACK's throughput

5. Conclusions

In this paper, we talked about the advantages of SACK mechanism, that is, this mechanism can perform in conjunction with other mechanisms in order to achieve the optimal performances.

However, the restriction of SACK blocks' number in the implementation of standard SACK mechanism indicates that it unfit to handle scenarios under error bursts' environment. Especially, the space-lacking shortcoming will be much notable if there are other mechanisms using selective acknowledgments for TCP congestion control.

Our proposed OOBSM mechanism can primely overcome above problems. Although the proposed scheme in [13] can also improve the performance of SACK mechanism when it performs individually, however, through evaluating its algorithm and performance, it is shown that there is no enough SACK option space yet when it tries to work in conjunction with other mechanisms for optimal performance. In addition, its algorithm decides that it will take much more option space and much more runtime than ours. The result of simulations also show that the implementation of our proposed OOBSM mechanism is better than others by means of the least bytes and most robust mechanism to the high packet error rates often seen in wireless networks.

References

[1] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP selective acknowledgment and op-tions", RFC 2018, IETF, October, 1996.

[2] Balan, R.K.; Lee, B.P.; Kumar, K.R.R.; Jacob, L.; Seah, W.K.G.; Ananda, A.L. "TCP HACK : TCP Header Checksum Option to Improve Performance over Lossy Links", INFOCOM 2001. IEEE , Volume : 1, 22~26 April, 2001 Pages : 309~318 Vol.1.

[3] V. Jacobson, R. Braden, D. Borman, "TCP Extensions for High Performance", RFC 1323, IETF, May, 1992.

[4] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, "An Extension to the Selective Acknowledgement(SACK) Option for TCP", RFC 2883, IETF, July, 2000.

[5] M. Allman, NASA Glenn, V. Paxson, W. Stevens "TCP Congestion Control", RFC 2581, IETF, April 1999.

[6] Jacobson, V., "Congestion Avoidance and Control", Computer Communication Review, Vol.18, No.4, pp.314~329, Aug., 1988.

[7] K. Fall, S. Floyd, "Simulation based Comparisons of Tahoe, Reno, and SACK TCP", Computer Communications Review, Vol.26, pp.5~21, 1996.

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

[9] Braden, R., "Requirements for Internet Hosts-Communication Layers", STD 3, RFC 1122, October, 1989.

[10] Luigi A. Grieco and Saverio Mascolo, "Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control", ACM, 2004.

[11] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang, "TCP Westwood : End-to-end bandwidth estimation for efficient transport over wired and wireless networks," in ACM Mobicom 2001, Rome, Italy, July, 2001.

[12] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. "TCP Selective Acknowledgment Options,". (Internet draft, work in progress), 1996.

[13] K.N. Srijith, Lillykutty Jacob, A.L. Ananda, "Worst-Case Performance Limitation of TCP SACK and a Feasible Solution", IEEE Journal on Selected Areas in Commu- nications, 2002.

[14] Network Simulator(ns-2), available from http://www.isi.edu/nsnam/ns/.

최 린

e-mail : lincui@cs.knu.ac.kr

1987년 Tianjin University 전자공학과(공학사)

2005년 경희대학교 컴퓨터공학과(공학석사)

2005년~현재 경북대학교 컴퓨터과학과 박사과정

관심분야 : 무선네트워크, 네트워크 QoS

홍 충 선

e-mail : cshong@khu.ac.kr

1983년 경희대학교 전자공학과(학사)

1985년 경희대학교 대학원 전자공학과(공학석사)

1997년 Keio University 정보통신공학전공(공학박사)

1988년~1999년 KT 통신망연구소 수석연구원/연구실장

1999년~현재 경희대학교 컴퓨터공학과 부교수

관심분야 : 초고속통신망, 이동네트워크, 네트워크보안, 센서네트워크라우팅